Time left 2:50:14

Question **3**

Not yet answered

Marked out of 1.00

Match each suggested semaphore implementation (discussed in class)

with the problems that it faces

```
struct semaphore {
        int val;
        spinlock lk;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                spinunlock(&(s->sl));
                spinlock(&(s->sl));
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

Choose...

```
struct semaphore {
        int val;
        spinlock lk;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0)
        ;
        (s->val)--;
        spinunlock(&(s->sl));
}
```

Choose...

```
struct semaphore {
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
block(semaphore *s) {
        listappend(s->l, current);
        spinunlock(&(s->sl));
        schedule();
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                block(s);
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
signal(seamphore *s) {
        spinlock(*(s->sl));
        (s->val)++;
        x = dequeue(s->sl) and enqueue(readyq, x);
        spinunlock(*(s->sl));
}
```

Choose...

```
struct semaphore {
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
block(semaphore *s) {
        listappend(s->l, current);
        schedule();
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                block(s);
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

Choose...

◄ Random Quiz - 6 (xv6 file system)

Jump to...

Homework questions: Basics of MM, xv6 booting ►

Question **10**

Not yet answered

Marked out of 1.00

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid.
"..." means some code.

Time left 2:12:36

```
void
yield(void)
{
...
release(&ptable.lock);
}
```

Release the lock held by some another process

```
void
panic(char *s)
{
...
  panicked = 1;
```

If you don't do this, a process may be running on two processors parallely

```
void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs);
```

Disable interrupts to avoid another process's pointer being returned

◄ Random Quiz - 6 (xv6 file system)

Jump to...

Homework questions: Basics of MM, xv6 booting ►

Question **12**

Not yet answered

Marked out of 2.00

Select all the correct statements about synchronization primitives.

Time left 2:04:57

Select one or more:

- [ ] a. Thread that is going to block should not be holding any spinlock
- [x] b. Semaphores can be used for synchronization scenarioes like ordered execution
- [x] c. Mutexes can be implemented using spinlock
- [ ] d. Blocking means one process passing over control to another process
- [ ] e. Semaphores are always a good substitute for spinlocks
- [ ] f. Blocking means moving the process to a wait queue and spinning
- [ ] g. Blocking means moving the process to a wait queue and calling scheduler
- [ ] h. All synchronization primitives are implemented essentially with some hardware assistance.
- [ ] i. Spinlocks are good for multiprocessor scenarios, for small critical sections
- [ ] j. Mutexes can be implemented without any hardware assistance
- [x] k. Spinlocks consume CPU time
- [x] l. Mutexes can be implemented using blocking and wakeup

◄ Random Quiz - 6 (xv6 file system)

Jump to...

Homework questions: Basics of MM, xv6 booting ►

Time left 1:48:32

Question **13**

Not yet answered

Marked out of 2.00

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid.

"..." means some code.

```
void
yield(void)
{
...
release(&ptable.lock);
}
```

Ensure that no printing happens on other processors

```
struct proc*
myproc(void) {
...
  pushcli();
  c = mycpu();
  p = c->proc;
  popcli();
...
}
```

Disable interrupts to avoid another process's pointer being returned

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
```

Atomic compare and swap instruction (to be expanded inline into code)

```
void
acquire(struct spinlock *lk)
{
...
__sync_synchronize();
```

Avoid a self-deadlock

```
void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs);
```

Disable interrupts to avoid deadlocks

```
void
acquire(struct spinlock *lk)
{
  pushcli();
```

Traverse ebp chain to get sequence of instructions followed in functions calls

```
void
sleep(void *chan, struct spinlock *lk)
{
  ...
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }
```

Release the lock held by some another process

```
void
panic(char *s)
{
...
  panicked = 1;
```

If you don't do this, a process may be running on two processors parallely

◄ Random Quiz - 6 (xv6 file system)

Jump to...

Homework questions: Basics of MM, xv6 booting ►

Question **14**

Not yet answered

Marked out of 3.00

Time left 1:53:37

List down all changes required to xv6 code, in order to add the system call chown().

Every change should be mentioned in terms of either of the following:

(a) pseudo-code of new function to be added

(b) prototype of any new function or new system call to be added

(c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added

(d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure

(e) Name and a one-line description of new userland functionality to be added

(f) Changes to Makefile

(g) Any other change in a maximum of 20 words per change.

Paragraph

(a) int chown (char * path , char *ownername);

(d)there is no need of new data structures or changes to the existing data structure

(e)by using this system call ,user to change the owner if the file

(f)_trychown\ at the end of UPROGS

Path: p

◄ Random Quiz - 6 (xv6 file system)

Jump to...

Homework questions: Basics of MM, xv6 booting ►

Question **17**

Not yet answered

Marked out of 1.00

Match the code with it's functionality

S1 = 0; S2 = 0;
P2:
Statement1;
Signal(S2);

P1:
Wait(S2);
Statemetn2;
Signal(S1);

P3:
Wait(S1);
Statement S3;

Execution order P1, P2, P3

Time left 1:35:43

S = 0
P1:
Statement1;
Signal(S)

P2:
Wait(S)
Statment2;

Execution order P3, P2, P1

S = 5
Wait(S)
Critical Section
Signal(S)

Execution order P2, then P1

S = 1
Wait(S)
Critical Section
Signal(S);

Counting semaphore

◄ Random Quiz - 6 (xv6 file system)

Jump to...

Homework questions: Basics of MM, xv6 booting ►

Question **18**

Not yet answered

Marked out of 2.00

Time left 1:34:55

Write all changes required to xv6 to add a buddy allocator.

Every change should be mentioned in terms of either of the following:

(a) pseudo-code of new function to be added

(b) prototype of any new function or new system call to be added

(c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added

(d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure

(e) Name and a one-line description of new userland functionality to be added

(f) Changes to Makefile

(g) Any other change in a maximum of 20 words per change.

Paragraph

Path: p

◄ Random Quiz - 6 (xv6 file system)

Jump to...

Homework questions: Basics of MM, xv6 booting ►

Started on	Wednesday, 19 April 2023, 8:46 PM
State	Finished
Completed on	Wednesday, 19 April 2023, 9:26 PM
Time taken	39 mins 31 secs
Grade	**15.60** out of 30.00 (**51.99**%)

Question **1**

Incorrect

Mark 0.00 out of 2.00

Select all the correct statements about synchronization primitives.

Select one or more:

- [ ] a. Blocking means moving the process to a wait queue and calling scheduler
- [x] b. Blocking means moving the process to a wait queue and spinning ✖
- [x] c. Semaphores are always a good substitute for spinlocks ✖
- [ ] d. Spinlocks are good for multiprocessor scenarios, for small critical sections
- [ ] e. All synchronization primitives are implemented essentially with some hardware assistance.
- [x] f. Blocking means one process passing over control to another process ✖
- [x] g. Semaphores can be used for synchronization scenarioes like ordered execution ✔
- [ ] h. Mutexes can be implemented without any hardware assistance
- [ ] i. Spinlocks consume CPU time
- [ ] j. Mutexes can be implemented using spinlock
- [ ] k. Thread that is going to block should not be holding any spinlock
- [ ] l. Mutexes can be implemented using blocking and wakeup

Your answer is incorrect.

The correct answers are: Spinlocks are good for multiprocessor scenarios, for small critical sections, Spinlocks consume CPU time, Semaphores can be used for synchronization scenarioes like ordered execution, Mutexes can be implemented using spinlock, Mutexes can be implemented using blocking and wakeup, Thread that is going to block should not be holding any spinlock, Blocking means moving the process to a wait queue and calling scheduler, All synchronization primitives are implemented essentially with some hardware assistance.

Question **2**

Partially correct

Mark 0.50 out of 1.00

For each of the resource allocation diagram shown,

infer whether the graph contains at least one deadlock or not.

Yes	No		
			✔
			✖

: Yes

Started on	Wednesday, 19 April 2023, 6:16 PM
State	Finished
Completed on	Wednesday, 19 April 2023, 8:31 PM
Time taken	2 hours 14 mins
Overdue	14 mins 47 secs
Grade	**14.08** out of 30.00 (**46.92**%)

Question **1**

Incorrect

Mark 0.00 out of 1.00

Select all correct statements about file system recovery (without journaling) programs e.g. fsck

Select one or more:

- [x] a. Recovery programs are needed only if the file system has a delayed-write policy. ✔
- [ ] b. Recovery is possible due to redundancy in file system data structures
- [x] c. It is possible to lose data as part of recovery ✔
- [x] d. A recovery program, most typically, builds the file system data structure and checks for inconsistencies ✔
- [x] e. Recovery programs recalculate most of the metadata summaries (e.g. free inode count) ✔
- [x] f. They are used to recover deleted files ✖
- [x] g. They may take very long time to execute ✔
- [ ] h. Even with a write-through policy, it is possible to need a recovery program.
- [ ] i. They can make changes to the on-disk file system

Your answer is incorrect.

The correct answers are: Recovery is possible due to redundancy in file system data structures, A recovery program, most typically, builds the file system data structure and checks for inconsistencies, It is possible to lose data as part of recovery, They may take very long time to execute, They can make changes to the on-disk file system, Recovery programs recalculate most of the metadata summaries (e.g. free inode count), Recovery programs are needed only if the file system has a delayed-write policy., Even with a write-through policy, it is possible to need a recovery program.

Question **2**

Partially correct

Mark 0.70 out of 1.00

Mark the statements as True or False, w.r.t. thrashing

True	False		
◉☑	○✗	Thrashing is particular to demand paging systems, and does not apply to pure paging systems.	✔
○☑	◉✗	During thrashing the CPU is under-utilised as most time is spent in I/O	✖
◉☑	○✗	Thrashing occurs when the total size of all processe's locality exceeds total memory size.	✔
◉☑	○✗	Thrashing can be limited if local replacement is used.	✔
○✗	◉☑	Thrashing occurs because some process is doing lot of disk I/O.	✔
◉☑	○✗	The working set model is an attempt at approximating the locality of a process.	✔
◉☑	○✗	Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.	✔
◉✗	○☑	Thrashing can occur even if entire memory is not in use.	✖
○✗	◉☑	mmap() solves the problem of thrashing.	✔
◉✗	○☑	Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.	✖

Thrashing is particular to demand paging systems, and does not apply to pure paging systems.: True
During thrashing the CPU is under-utilised as most time is spent in I/O: True
Thrashing occurs when the total size of all processe's locality exceeds total memory size.: True
Thrashing can be limited if local replacement is used.: True
Thrashing occurs because some process is doing lot of disk I/O.: False
The working set model is an attempt at approximating the locality of a process.: True
Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.: True
Thrashing can occur even if entire memory is not in use.: False
mmap() solves the problem of thrashing.: False
Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.: False

Question **3**

Partially correct

Mark 0.25 out of 1.00

Match each suggested semaphore implementation (discussed in class)

with the problems that it faces

```
struct semaphore {
        int val;
        spinlock lk;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                spinunlock(&(s->sl));
                spinlock(&(s->sl));
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

too much spinning, bounded wait not guaranteed ✔

```
struct semaphore {
        int val;
        spinlock lk;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0)
        ;
        (s->val)--;
        spinunlock(&(s->sl));
}
```

blocks holding a spinlock ✖

```
struct semaphore {
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
block(semaphore *s) {
        listappend(s->l, current);
        spinunlock(&(s->sl));
        schedule();
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                block(s);
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
signal(seamphore *s) {
        spinlock(*(s->sl));
        (s->val)++;
        x = dequeue(s->sl) and enqueue(readyq, x);
        spinunlock(*(s->sl));
}
```

too much spinning, bounded wait not guaranteed ✖

```
struct semaphore {
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
block(semaphore *s) {
        listappend(s->l, current);
        schedule();
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                block(s);
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

deadlock ✖

Your answer is partially correct.

You have correctly selected 1.

The correct answer is:

```
struct semaphore {
        int val;
        spinlock lk;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                spinunlock(&(s->sl));
                spinlock(&(s->sl));
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

→ too much spinning, bounded wait not guaranteed,

```
struct semaphore {
        int val;
        spinlock lk;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0)
        ;
        (s->val)--;
        spinunlock(&(s->sl));
}
```

→ deadlock,

```
struct semaphore {
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
block(semaphore *s) {
        listappend(s->l, current);
        spinunlock(&(s->sl));
        schedule();
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                block(s);
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
signal(seamphore *s) {
        spinlock(*(s->sl));
        (s->val)++;
        x = dequeue(s->sl) and enqueue(readyq, x);
        spinunlock(*(s->sl));
}
```

→ not holding lock after unblock,

```
struct semaphore {
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
block(semaphore *s) {
        listappend(s->l, current);
        schedule();
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                block(s);
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

→ blocks holding a spinlock

Question **4**

Partially correct

Mark 0.88 out of 1.00

Mark the statements as True or False, w.r.t. passing of arguments to system calls in xv6 code.

True	False		
◉☑	○✖	The arguments to system call originally reside on process stack.	✔
◉☑	○✖	String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer	✔
◉☑	○✖	The arguments are accessed in the kernel code using esp on the trapframe.	✔
○✖	◉☑	Integer arguments are stored in eax, ebx, ecx, etc. registers	✔
◉☑	○✖	Integer arguments are copied from user memory to kernel memory using argint()	✔
○✖	◉☑	String arguments are first copied to trapframe and then from trapframe to kernel's other variables.	✔
◉☑	○✖	The functions like argint(), argstr() make the system call arguments available in the kernel.	✔
◉✖	○☑	The arguments to system call are copied to kernel stack in trapasm.S	✖

The arguments to system call originally reside on process stack.: True
String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer: True
The arguments are accessed in the kernel code using esp on the trapframe.: True
Integer arguments are stored in eax, ebx, ecx, etc. registers: False
Integer arguments are copied from user memory to kernel memory using argint(): True
String arguments are first copied to trapframe and then from trapframe to kernel's other variables.: False
The functions like argint(), argstr() make the system call arguments available in the kernel.: True
The arguments to system call are copied to kernel stack in trapasm.S: False

Question **5**

Correct

Mark 1.00 out of 1.00

Map the technique with it's feature/problem

static loading	wastage of physical memory	✔
static linking	large executable file	✔
dynamic linking	small executable file	✔
dynamic loading	allocate memory only if needed	✔

The correct answer is: static loading → wastage of physical memory, static linking → large executable file, dynamic linking → small executable file, dynamic loading → allocate memory only if needed

Question **6**

Partially correct

Mark 0.10 out of 2.00

Compare paging with demand paging and select the correct statements.

Select one or more:

- [ ] a. Demand paging requires additional hardware support, compared to paging.
- [x] b. Demand paging always increases effective memory access time. ✔
- [x] c. Both demand paging and paging support shared memory pages. ✔
- [x] d. TLB hit ration has zero impact in effective memory access time in demand paging. ✖
- [ ] e. Paging requires NO hardware support in CPU
- [x] f. Paging requires some hardware support in CPU ✔
- [x] g. The meaning of valid-invalid bit in page table is different in paging and demand-paging. ✔
- [x] h. Calculations of number of bits for page number and offset are same in paging and demand paging. ✔
- [x] i. With paging, it's possible to have user programs bigger than physical memory. ✖
- [ ] j. With demand paging, it's possible to have user programs bigger than physical memory.

Your answer is partially correct.

You have correctly selected 5.

The correct answers are: Demand paging requires additional hardware support, compared to paging., Both demand paging and paging support shared memory pages., With demand paging, it's possible to have user programs bigger than physical memory., Demand paging always increases effective memory access time., Paging requires some hardware support in CPU, Calculations of number of bits for page number and offset are same in paging and demand paging., The meaning of valid-invalid bit in page table is different in paging and demand-paging.

Question **7**

Correct

Mark 1.00 out of 1.00

Given that the memory access time is 150 ns, probability of a page fault is 0.9 and page fault handling time is 6 ms,
The effective memory access time in nanoseconds is:

Answer: 5400015 ✔

The correct answer is: 5400015.00

Question **8**

Correct

Mark 1.00 out of 1.00

Assuming a 8- KB page size, what is the page numbers for the address 803160 reference in decimal :

(give answer also in decimal)

Answer: 98 ✔

The correct answer is: 98

Question **9**

Partially correct

Mark 1.50 out of 2.00

For Virtual File System to work, which of the following changes are required to be done to an existing OS code (e.g. xv6)?

- [x] a. Each file-system writer needs to provide the set of function pointers for VFS, and these function pointers need to be setup in generic inode of "/" of that file system during mount() ✔
- [x] b. A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/" ✔
- [ ] c. The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems.
- [x] d. The generic inode needs to have a field representing if this inode is a mount point and also to refer/point to the root of the mounted file system's inode. ✔
- [x] e. The filesystem related system calls (e.g. read, write) need to invoke the file system specific functions (e.g. ext2_read, ext2_write, ntfs_read, ntfs_write) using function pointers. ✔
- [ ] f. The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup.
- [x] g. Each open() needs to copy the function pointers from the inode of the parent directory into the inode of the child (if not already done), unless it's traversing a mount point. (This may be done as part of lookup() which is called by open()) ✔
- [x] h. The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories ✔

The correct answers are: A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/", The filesystem related system calls (e.g. read, write) need to invoke the file system specific functions (e.g. ext2_read, ext2_write, ntfs_read, ntfs_write) using function pointers., The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup., The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems., The generic inode needs to have a field representing if this inode is a mount point and also to refer/point to the root of the mounted file system's inode., The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories, Each file-system writer needs to provide the set of function pointers for VFS, and these function pointers need to be setup in generic inode of "/" of that file system during mount(), Each open() needs to copy the function pointers from the inode of the parent directory into the inode of the child (if not already done), unless it's traversing a mount point. (This may be done as part of lookup() which is called by open())

Question **10**

Partially correct

Mark 0.33 out of 1.00

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid.
"..." means some code.

```
void
yield(void)
{
...
release(&ptable.lock);
}
```

Release the lock held by some another process ✔

```
void
panic(char *s)
{
...
  panicked = 1;
```

If you don't do this, a process may be running on two processors parallely ✖

```
void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs);
```

Disable interrupts to avoid another process's pointer being returned ✖

Your answer is partially correct.

You have correctly selected 1.

The correct answer is: void
yield(void)
{
...
release(&ptable.lock);
} → Release the lock held by some another process, void
panic(char *s)
{
...
  panicked = 1; → Ensure that no printing happens on other processors, void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs); → Traverse ebp chain to get sequence of instructions followed in functions calls

Question **11**

Partially correct

Mark 0.75 out of 1.00

Select all correct statements about journalling (logging) in file systems like ext3

Select one or more:

- [ ] a. A different device driver is always needed to access the journal
- [x] b. The purpose of journal is to speed up file system recovery ✔
- [x] c. Most typically a transaction in journal is recorded atomically (full or none) ✔
- [ ] d. Journals are often stored circularly
- [ ] e. Journals must be maintained on the same device that hosts the file system
- [x] f. the journal contains a summary of all changes made as part of a single transaction ✔
- [ ] g. Journal is hosted in the same device that hosts the swap space

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: The purpose of journal is to speed up file system recovery, the journal contains a summary of all changes made as part of a single transaction, Most typically a transaction in journal is recorded atomically (full or none), Journals are often stored circularly

Question **12**

Partially correct

Mark 1.00 out of 2.00

Select all the correct statements about synchronization primitives.

Select one or more:

- [ ] a. Thread that is going to block should not be holding any spinlock
- [x] b. Semaphores can be used for synchronization scenarioes like ordered execution ✔
- [x] c. Mutexes can be implemented using spinlock ✔
- [ ] d. Blocking means one process passing over control to another process
- [ ] e. Semaphores are always a good substitute for spinlocks
- [ ] f. Blocking means moving the process to a wait queue and spinning
- [ ] g. Blocking means moving the process to a wait queue and calling scheduler
- [ ] h. All synchronization primitives are implemented essentially with some hardware assistance.
- [ ] i. Spinlocks are good for multiprocessor scenarios, for small critical sections
- [ ] j. Mutexes can be implemented without any hardware assistance
- [x] k. Spinlocks consume CPU time ✔
- [x] l. Mutexes can be implemented using blocking and wakeup ✔

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: Spinlocks are good for multiprocessor scenarios, for small critical sections, Spinlocks consume CPU time, Semaphores can be used for synchronization scenarioes like ordered execution, Mutexes can be implemented using spinlock, Mutexes can be implemented using blocking and wakeup, Thread that is going to block should not be holding any spinlock, Blocking means moving the process to a wait queue and calling scheduler, All synchronization primitives are implemented essentially with some hardware assistance.

Question **13**

Partially correct

Mark 0.50 out of 2.00

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid.

"..." means some code.

```
void
yield(void)
{
...
release(&ptable.lock);
}
```

Ensure that no printing happens on other processors

✖

```
struct proc*
myproc(void) {
...
  pushcli();
  c = mycpu();
  p = c->proc;
  popcli();
...
}
```

Disable interrupts to avoid another process's pointer being returned

✔

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write
operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
```

Atomic compare and swap instruction (to be expanded inline into code)

✔

```
void
acquire(struct spinlock *lk)
{
...
__sync_synchronize();
```

Avoid a self-deadlock

✖

```
void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs);
```

Disable interrupts to avoid deadlocks

✖

```
void
acquire(struct spinlock *lk)
{
  pushcli();
```

Traverse ebp chain to get sequence of instructions followed in functions calls

✖

```
void
sleep(void *chan, struct spinlock *lk)
{
  ...
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }
```

Release the lock held by some another process

✖

```
void
panic(char *s)
{
...
  panicked = 1;
```

If you don't do this, a process may be running on two processors parallely

✖

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: void

```
yield(void)
{
...
release(&ptable.lock);
}
```

→ Release the lock held by some another process, struct proc*

```
myproc(void) {
...
  pushcli();
  c = mycpu();
  p = c->proc;
  popcli();
...
}
```

→ Disable interrupts to avoid another process's pointer being returned, static inline uint

```
xchg(volatile uint *addr, uint newval)
{
  uint result;
```

```
  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
} → Atomic compare and swap instruction (to be expanded inline into code), void
acquire(struct spinlock *lk)
{
...
__sync_synchronize();



  → Tell compiler not to reorder memory access beyond this line, void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs);



  → Traverse ebp chain to get sequence of instructions followed in functions calls, void
acquire(struct spinlock *lk)
{
  pushcli();
  → Disable interrupts to avoid deadlocks, void
sleep(void *chan, struct spinlock *lk)
{
  ...
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  } → Avoid a self-deadlock, void
panic(char *s)
{
...
  panicked = 1; → Ensure that no printing happens on other processors
```

Question **14**

Partially correct

Mark 0.86 out of 1.00

Select T/F for statements about Volume Managers.
Do pay attention to the use of the words physical partition and physical volume.

True	False			
◉☑	○✖	A logical volume may span across multiple physical partitions	✔	since a physical volume is made up of physical partitions, and a volume can span across multiple PVs, it can also span across multiple PP
○☑	○✖	A logical volume may span across multiple physical volumes	✖	
◉☑	○✖	A physical partition should be initialized as a physial volume, before it can be used by volume manager.	✔	
◉☑	○✖	A volume group consists of multiple physical volumes	✔	
◉☑	○✖	The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.	✔	
◉☑	○✖	A logical volume can be extended in size but upto the size of volume group	✔	
◉☑	○✖	The volume manager stores additional metadata on the physical disk partitions	✔	

A logical volume may span across multiple physical partitions: True
A logical volume may span across multiple physical volumes: True
A physical partition should be initialized as a physial volume, before it can be used by volume manager.: True
A volume group consists of multiple physical volumes: True
The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.: True
A logical volume can be extended in size but upto the size of volume group: True
The volume manager stores additional metadata on the physical disk partitions: True

Question **15**

Partially correct

Mark 0.50 out of 1.00

For each of the resource allocation diagram shown,

infer whether the graph contains at least one deadlock or not.

Yes	No		
◉☑	○✖		✔
○☑	◉✖		✖

: Yes

: Yes

Question **16**

Partially correct

Mark 0.80 out of 1.00

Select all the correct statements w.r.t user and kernel threads

Select one or more:

- [x] a. all three models, that is many-one, one-one, many-many , require a user level thread library ✔
- [ ] b. A process may not block in many-one model, if a thread makes a blocking system call
- [ ] c. one-one model can be implemented even if there are no kernel threads
- [x] d. one-one model increases kernel's scheduling load ✔
- [ ] e. A process blocks in many-one model even if a single thread makes a blocking system call
- [x] f. many-one model can be implemented even if there are no kernel threads ✔
- [x] g. many-one model gives no speedup on multicore processors ✔

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: many-one model can be implemented even if there are no kernel threads, all three models, that is many-one, one-one, many-many , require a user level thread library, one-one model increases kernel's scheduling load, many-one model gives no speedup on multicore processors, A process blocks in many-one model even if a single thread makes a blocking system call

Question **17**

Incorrect

Mark 0.00 out of 1.00

Match the code with it's functionality

S1 = 0; S2 = 0;
P2:
Statement1;
Signal(S2);

P1:
Wait(S2);
Statemetn2;
Signal(S1);

P3:
Wait(S1);
Statement S3;

Execution order P1, P2, P3 ✖

S = 0
P1:
Statement1;
Signal(S)

P2:
Wait(S)
Statment2;

Execution order P3, P2, P1 ✖

S = 5
Wait(S)
Critical Section
Signal(S)

Execution order P2, then P1 ✖

S = 1
Wait(S)
Critical Section
Signal(S);

Counting semaphore ✖

Your answer is incorrect.

The correct answer is: S1 = 0; S2 = 0;
P2:
Statement1;
Signal(S2);

P1:
Wait(S2);
Statemetn2;
Signal(S1);

P3:
Wait(S1);
Statement S3; → Execution order P2, P1, P3, S = 0
P1:
Statement1;

Signal(S)

P2:
Wait(S)
Statment2; → Execution order P1, then P2, S = 5
Wait(S)
Critical Section
Signal(S) → Counting semaphore, S = 1
Wait(S)
Critical Section
Signal(S); → Binary Semaphore for mutual exclusion

Question **18**

Not answered

Marked out of 2.00

Write all changes required to xv6 to add a buddy allocator.

Every change should be mentioned in terms of either of the following:

(a) pseudo-code of new function to be added

(b) prototype of any new function or new system call to be added

(c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added

(d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure

(e) Name and a one-line description of new userland functionality to be added

(f) Changes to Makefile

(g) Any other change in a maximum of 20 words per change.

Question **19**

Correct

Mark 1.00 out of 1.00

Note: for this question you get full marks if you select all and only correct options, you get ZERO if at least one option is wrong or not selected.

Select all the correct statements about log structured file systems.

- [ ] a. a transaction is said to be committed when all operations are written to file system
- [x] b. even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery ✔
- [x] c. file system recovery may end up losing data ✔
- [ ] d. file system recovery recovers all the lost data
- [x] e. log may be kept on same block device or another block device ✔

Your answer is correct.

The correct answers are: file system recovery may end up losing data, log may be kept on same block device or another block device, even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery

Question **20**

Incorrect

Mark 0.00 out of 1.00

Suppose a kernel uses a buddy allocator. The smallest chunk that can be allocated is of size 32 bytes. One bit is used to track each such chunk, where 1 means allocated and 0 means free. The chunk looks like this as of now:

11010010

Now, there is a request for a chunk of 45 bytes.

After this allocation, the bitmap, indicating the status of the buddy allocator will be

Answer: 11001110 ✖

The correct answer is: 11011110

Question **21**

Complete

Mark 0.25 out of 3.00

List down all changes required to xv6 code, in order to add the system call chown().

Every change should be mentioned in terms of either of the following:

(a) pseudo-code of new function to be added

(b) prototype of any new function or new system call to be added

(c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added

(d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure

(e) Name and a one-line description of new userland functionality to be added

(f) Changes to Makefile

(g) Any other change in a maximum of 20 words per change.

(a)int chown(char* path, char *owner_name);

d) there is no need to change the data structure and no need to add new data structure

(e)allow user to change the owner of the file

(f)_chown\

Comment:

Question **22**

Correct

Mark 1.00 out of 1.00

Calculate the average waiting time using
FCFS scheduling
for the following workload

assuming that they arrive in this order during the first time unit:

Process	Burst Time
P1	2
P2	6
P3	2
P4	3

Write only a number in the answer upto two decimal points.

Answer: 5 ✔

P2 waits for 2 units

P3 waits for 2+6 units

P4 waits for 2 + 6 +2 units of time

Total waiting = 2 + 2 + 6 + 2 + 6 + 2 = 20 units

Average waiting time = 20/4 = 5

The correct answer is: 5

Question **23**

Partially correct

Mark 0.67 out of 1.00

Given that a kernel has 1000 KB of total memory, and holes of sizes (in that order) 300 KB, 200 KB, 100 KB, 250 KB. For each of the requests on the left side, match it with the chunk chosen using the specified algorithm.

Consider each request as first request.

150 KB, first fit	200 KB	✖
150 KB, best fit	300 KB	✖
220 KB, best fit	250 KB	✔
200 KB, first fit	300 KB	✔
100 KB, worst fit	300 KB	✔
50 KB, worst fit	300 KB	✔

The correct answer is: 150 KB, first fit → 300 KB, 150 KB, best fit → 200 KB, 220 KB, best fit → 250 KB, 200 KB, first fit → 300 KB, 100 KB, worst fit → 300 KB, 50 KB, worst fit → 300 KB

◄ Random Quiz - 6 (xv6 file system)

Jump to...

Homework questions: Basics of MM, xv6 booting ►

Started on	Wednesday, 19 April 2023, 8:46 PM
State	Finished
Completed on	Wednesday, 19 April 2023, 9:26 PM
Time taken	39 mins 31 secs
Grade	**15.60** out of 30.00 (**51.99**%)

Question **1**

Incorrect

Mark 0.00 out of 2.00

Select all the correct statements about synchronization primitives.

Select one or more:

- [ ] a. Blocking means moving the process to a wait queue and calling scheduler
- [x] b. Blocking means moving the process to a wait queue and spinning ✖
- [x] c. Semaphores are always a good substitute for spinlocks ✖
- [ ] d. Spinlocks are good for multiprocessor scenarios, for small critical sections
- [ ] e. All synchronization primitives are implemented essentially with some hardware assistance.
- [x] f. Blocking means one process passing over control to another process ✖
- [x] g. Semaphores can be used for synchronization scenarioes like ordered execution ✔
- [ ] h. Mutexes can be implemented without any hardware assistance
- [ ] i. Spinlocks consume CPU time
- [ ] j. Mutexes can be implemented using spinlock
- [ ] k. Thread that is going to block should not be holding any spinlock
- [ ] l. Mutexes can be implemented using blocking and wakeup

Your answer is incorrect.

The correct answers are: Spinlocks are good for multiprocessor scenarios, for small critical sections, Spinlocks consume CPU time, Semaphores can be used for synchronization scenarioes like ordered execution, Mutexes can be implemented using spinlock, Mutexes can be implemented using blocking and wakeup, Thread that is going to block should not be holding any spinlock, Blocking means moving the process to a wait queue and calling scheduler, All synchronization primitives are implemented essentially with some hardware assistance.

Question **2**

Partially correct

Mark 0.50 out of 1.00

For each of the resource allocation diagram shown,

infer whether the graph contains at least one deadlock or not.

Yes	No		
			✔
			✖

: Yes

: Yes

Question **3**

Partially correct

Mark 0.25 out of 1.00

Match each suggested semaphore implementation (discussed in class)

with the problems that it faces

```
struct semaphore {
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
block(semaphore *s) {
        listappend(s->l, current);
        spinunlock(&(s->sl));
        schedule();
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                block(s);
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
signal(seamphore *s) {
        spinlock(*(s->sl));
        (s->val)++;
        x = dequeue(s->sl) and enqueue(readyq, x);
        spinunlock(*(s->sl));
}
```

deadlock ✖

```
struct semaphore {
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
block(semaphore *s) {
        listappend(s->l, current);
        schedule();
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                block(s);
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

not holding lock after unblock ✖

```
struct semaphore {
        int val;
        spinlock lk;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                spinunlock(&(s->sl));
                spinlock(&(s->sl));
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

too much spinning, bounded wait not guaranteed ✔

```
struct semaphore {
        int val;
        spinlock lk;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0)
        ;
        (s->val)--;
        spinunlock(&(s->sl));
}
```

blocks holding a spinlock ✖

Your answer is partially correct.

You have correctly selected 1.

The correct answer is:

```
struct semaphore {
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
block(semaphore *s) {
        listappend(s->l, current);
        spinunlock(&(s->sl));
        schedule();
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                block(s);
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
signal(seamphore *s) {
        spinlock(*(s->sl));
        (s->val)++;
        x = dequeue(s->sl) and enqueue(readyq, x);
        spinunlock(*(s->sl));
}
```

→ not holding lock after unblock,

```
struct semaphore {
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
block(semaphore *s) {
        listappend(s->l, current);
        schedule();
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                block(s);
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

→ blocks holding a spinlock,

```
struct semaphore {
        int val;
        spinlock lk;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                spinunlock(&(s->sl));
                spinlock(&(s->sl));
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

→ too much spinning, bounded wait not guaranteed,

```
struct semaphore {
        int val;
        spinlock lk;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0)
        ;
        (s->val)--;
        spinunlock(&(s->sl));
}
```

→ deadlock

Question **4**

Partially correct

Mark 0.75 out of 1.00

Match the code with it's functionality

S = 1
Wait(S)
Critical Section
Signal(S);

Binary Semaphore for mutual exclusion ✔

S = 0
P1:
Statement1;
Signal(S)

P2:
Wait(S)
Statment2;

Execution order P1, then P2 ✔

S = 5
Wait(S)
Critical Section
Signal(S)

Execution order P1, P2, P3 ✖

S1 = 0; S2 = 0;
P2:
Statement1;
Signal(S2);

P1:
Wait(S2);
Statemetn2;
Signal(S1);

P3:
Wait(S1);
Statement S3;

Execution order P2, P1, P3 ✔

Your answer is partially correct.

You have correctly selected 3.
The correct answer is: S = 1
Wait(S)
Critical Section
Signal(S); → Binary Semaphore for mutual exclusion, S = 0
P1:
Statement1;
Signal(S)

P2:
Wait(S)
Statment2; → Execution order P1, then P2, S = 5
Wait(S)
Critical Section
Signal(S) → Counting semaphore, S1 = 0; S2 = 0;

P2:
Statement1;
Signal(S2);

P1:
Wait(S2);
Statemetn2;
Signal(S1);

P3:
Wait(S1);
Statement S3; → Execution order P2, P1, P3

Question **5**

Partially correct

Mark 1.71 out of 2.00

Compare paging with demand paging and select the correct statements.

Select one or more:

- [ ] a. With paging, it's possible to have user programs bigger than physical memory.
- [x] b. With demand paging, it's possible to have user programs bigger than physical memory.✔
- [x] c. Paging requires some hardware support in CPU✔
- [ ] d. Paging requires NO hardware support in CPU
- [x] e. Calculations of number of bits for page number and offset are same in paging and demand paging.✔
- [ ] f. TLB hit ration has zero impact in effective memory access time in demand paging.
- [x] g. Demand paging always increases effective memory access time.✔
- [ ] h. The meaning of valid-invalid bit in page table is different in paging and demand-paging.
- [x] i. Demand paging requires additional hardware support, compared to paging.✔
- [x] j. Both demand paging and paging support shared memory pages.✔

Your answer is partially correct.

You have correctly selected 6.
The correct answers are: Demand paging requires additional hardware support, compared to paging., Both demand paging and paging support shared memory pages., With demand paging, it's possible to have user programs bigger than physical memory., Demand paging always increases effective memory access time., Paging requires some hardware support in CPU, Calculations of number of bits for page number and offset are same in paging and demand paging., The meaning of valid-invalid bit in page table is different in paging and demand-paging.

Question **6**

Partially correct

Mark 0.33 out of 1.00

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid.
"..." means some code.

```
void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs);
```

Disable interrupts to avoid another process's pointer being returned ✖

```
void
panic(char *s)
{
...
  panicked = 1;
```

If you don't do this, a process may be running on two processors parallely ✖

```
void
yield(void)
{
...
release(&ptable.lock);
}
```

Release the lock held by some another process ✔

Your answer is partially correct.

You have correctly selected 1.
The correct answer is: void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs); → Traverse ebp chain to get sequence of instructions followed in functions calls, void
panic(char *s)
{
...
  panicked = 1; → Ensure that no printing happens on other processors, void
yield(void)
{
...
release(&ptable.lock);
} → Release the lock held by some another process

Question **7**

Correct

Mark 1.00 out of 1.00

Given that a kernel has 1000 KB of total memory, and holes of sizes (in that order) 300 KB, 200 KB, 100 KB, 250 KB. For each of the requests on the left side, match it with the chunk chosen using the specified algorithm.

Consider each request as first request.

150 KB, first fit	300 KB	✔
220 KB, best fit	250 KB	✔
100 KB, worst fit	300 KB	✔
200 KB, first fit	300 KB	✔
50 KB, worst fit	300 KB	✔
150 KB, best fit	200 KB	✔

The correct answer is: 150 KB, first fit → 300 KB, 220 KB, best fit → 250 KB, 100 KB, worst fit → 300 KB, 200 KB, first fit → 300 KB, 50 KB, worst fit → 300 KB, 150 KB, best fit → 200 KB

Question **8**

Correct

Mark 1.00 out of 1.00

Select T/F for statements about Volume Managers.
Do pay attention to the use of the words physical partition and physical volume.

True	False			
◉ ✔	○ ✖	A logical volume may span across multiple physical partitions	✔	since a physical volume is made up of physical partitions, and a volume can span across multiple PVs, it can also span across multiple PP
◉ ✔	○ ✖	A volume group consists of multiple physical volumes	✔	
◉ ✔	○ ✖	A logical volume may span across multiple physical volumes	✔	
◉ ✔	○ ✖	A logical volume can be extended in size but upto the size of volume group	✔	
◉ ✔	○ ✖	The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.	✔	
◉ ✔	○ ✖	A physical partition should be initialized as a physial volume, before it can be used by volume manager.	✔	
◉ ✔	○ ✖	The volume manager stores additional metadata on the physical disk partitions	✔	

A logical volume may span across multiple physical partitions: True
A volume group consists of multiple physical volumes: True
A logical volume may span across multiple physical volumes: True
A logical volume can be extended in size but upto the size of volume group: True
The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.: True
A physical partition should be initialized as a physial volume, before it can be used by volume manager.: True
The volume manager stores additional metadata on the physical disk partitions: True

Question **9**

Incorrect

Mark 0.00 out of 1.00

Suppose a kernel uses a buddy allocator. The smallest chunk that can be allocated is of size 32 bytes. One bit is used to track each such chunk, where 1 means allocated and 0 means free. The chunk looks like this as of now:

11010010

Now, there is a request for a chunk of 45 bytes.

After this allocation, the bitmap, indicating the status of the buddy allocator will be

Answer: 11001101 ✖

The correct answer is: 11011110

Question **10**

Not answered

Marked out of 2.00

Write all changes required to xv6 to add a buddy allocator.

Every change should be mentioned in terms of either of the following:

(a) pseudo-code of new function to be added

(b) prototype of any new function or new system call to be added

(c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added

(d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure

(e) Name and a one-line description of new userland functionality to be added

(f) Changes to Makefile

(g) Any other change in a maximum of 20 words per change.

Question **11**

Partially correct

Mark 0.80 out of 1.00

Select all the correct statements w.r.t user and kernel threads

Select one or more:

- [x] a. many-one model can be implemented even if there are no kernel threads ✔
- [ ] b. one-one model can be implemented even if there are no kernel threads
- [x] c. all three models, that is many-one, one-one, many-many , require a user level thread library ✔
- [x] d. many-one model gives no speedup on multicore processors ✔
- [x] e. one-one model increases kernel's scheduling load ✔
- [ ] f. A process blocks in many-one model even if a single thread makes a blocking system call
- [ ] g. A process may not block in many-one model, if a thread makes a blocking system call

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: many-one model can be implemented even if there are no kernel threads, all three models, that is many-one, one-one, many-many , require a user level thread library, one-one model increases kernel's scheduling load, many-one model gives no speedup on multicore processors, A process blocks in many-one model even if a single thread makes a blocking system call

Question **12**

Partially correct

Mark 0.50 out of 1.00

Select all correct statements about journalling (logging) in file systems like ext3

Select one or more:

- [x] a. the journal contains a summary of all changes made as part of a single transaction ✔
- [ ] b. Journals are often stored circularly
- [ ] c. Journals must be maintained on the same device that hosts the file system
- [x] d. The purpose of journal is to speed up file system recovery ✔
- [ ] e. Most typically a transaction in journal is recorded atomically (full or none)
- [ ] f. Journal is hosted in the same device that hosts the swap space
- [ ] g. A different device driver is always needed to access the journal

Your answer is partially correct.

You have correctly selected 2.

The correct answers are: The purpose of journal is to speed up file system recovery, the journal contains a summary of all changes made as part of a single transaction, Most typically a transaction in journal is recorded atomically (full or none), Journals are often stored circularly

Question **13**

Correct

Mark 1.00 out of 1.00

Note: for this question you get full marks if you select all and only correct options, you get ZERO if at least one option is wrong or not selected.

Select all the correct statements about log structured file systems.

- [ ] a. a transaction is said to be committed when all operations are written to file system
- [x] b. file system recovery may end up losing data ✔
- [x] c. log may be kept on same block device or another block device ✔
- [x] d. even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery ✔
- [ ] e. file system recovery recovers all the lost data

Your answer is correct.

The correct answers are: file system recovery may end up losing data, log may be kept on same block device or another block device, even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery

Question **14**

Complete

Mark 0.25 out of 3.00

List down all changes required to xv6 code, in order to add the system call chown().

Every change should be mentioned in terms of either of the following:

(a) pseudo-code of new function to be added

(b) prototype of any new function or new system call to be added

(c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added

(d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure

(e) Name and a one-line description of new userland functionality to be added

(f) Changes to Makefile

(g) Any other change in a maximum of 20 words per change.

(a) int chown (char * path , char *ownername);

(d)there is no need of new data structures or changes to the existing data structure

(e)by using this system call ,user to change the owner if the file

(f)_trychown\ at the end of UPROGS

Comment:

Question **15**

Partially correct

Mark 1.00 out of 2.00

For Virtual File System to work, which of the following changes are required to be done to an existing OS code (e.g. xv6)?

- [ ] a. The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems.
- [x] b. The generic inode needs to have a field representing if this inode is a mount point and also to refer/point to the root of the mounted file system's inode. ✔
- [x] c. Each file-system writer needs to provide the set of function pointers for VFS, and these function pointers need to be setup in generic inode of "/" of that file system during mount() ✔
- [x] d. A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/" ✔
- [x] e. Each open() needs to copy the function pointers from the inode of the parent directory into the inode of the child (if not already done), unless it's traversing a mount point. (This may be done as part of lookup() which is called by open()) ✔
- [ ] f. The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup.
- [ ] g. The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories
- [ ] h. The filesystem related system calls (e.g. read, write) need to invoke the file system specific functions (e.g. ext2_read, ext2_write, ntfs_read, ntfs_write) using function pointers.

The correct answers are: A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/", The filesystem related system calls (e.g. read, write) need to invoke the file system specific functions (e.g. ext2_read, ext2_write, ntfs_read, ntfs_write) using function pointers., The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup., The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems., The generic inode needs to have a field representing if this inode is a mount point and also to refer/point to the root of the mounted file system's inode., The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories, Each file-system writer needs to provide the set of function pointers for VFS, and these function pointers need to be setup in generic inode of "/" of that file system during mount(), Each open() needs to copy the function pointers from the inode of the parent directory into the inode of the child (if not already done), unless it's traversing a mount point. (This may be done as part of lookup() which is called by open())

Question **16**

Partially correct

Mark 0.75 out of 1.00

Map the technique with it's feature/problem

dynamic linking	small executable file	✔
static loading	wastage of physical memory	✔
static linking	large executable file	✔
dynamic loading	small executable file	✖

The correct answer is: dynamic linking → small executable file, static loading → wastage of physical memory, static linking → large executable file, dynamic loading → allocate memory only if needed

Question **17**

Correct

Mark 1.00 out of 1.00

Mark the statements as True or False, w.r.t. thrashing

True	False		
◉☑	○✖	Thrashing can be limited if local replacement is used.	✔
○✖	◉☑	Thrashing occurs because some process is doing lot of disk I/O.	✔
◉☑	○✖	Thrashing occurs when the total size of all processe's locality exceeds total memory size.	✔
◉☑	○✖	During thrashing the CPU is under-utilised as most time is spent in I/O	✔
◉☑	○✖	Thrashing is particular to demand paging systems, and does not apply to pure paging systems.	✔
○✖	◉☑	Thrashing can occur even if entire memory is not in use.	✔
◉☑	○✖	Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.	✔
○✖	◉☑	Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.	✔
◉☑	○✖	The working set model is an attempt at approximating the locality of a process.	✔
○✖	◉☑	mmap() solves the problem of thrashing.	✔

Thrashing can be limited if local replacement is used.: True
Thrashing occurs because some process is doing lot of disk I/O.: False
Thrashing occurs when the total size of all processe's locality exceeds total memory size.: True
During thrashing the CPU is under-utilised as most time is spent in I/O: True
Thrashing is particular to demand paging systems, and does not apply to pure paging systems.: True
Thrashing can occur even if entire memory is not in use.: False
Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.: True
Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.: False
The working set model is an attempt at approximating the locality of a process.: True
mmap() solves the problem of thrashing.: False

Question **18**

Partially correct

Mark 0.75 out of 2.00

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid.

"..." means some code.

```
struct proc*
myproc(void) {
...
  pushcli();
  c = mycpu();
  p = c->proc;
  popcli();
...
}
```

Traverse ebp chain to get sequence of instructions followed in functions calls

✖

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
```

Atomic compare and swap instruction (to be expanded inline into code)

✔

```
void
sleep(void *chan, struct spinlock *lk)
{
  ...
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }
```

Release the lock held by some another process

✖

```
void
panic(char *s)
{
...
  panicked = 1;
```

If you don't do this, a process may be running on two processors parallely

✖

```
void
acquire(struct spinlock *lk)
{
  pushcli();
```

Release the lock held by some another process

✖

```
void
acquire(struct spinlock *lk)
{
...
__sync_synchronize();
```

Tell compiler not to reorder memory access beyond this line

✔

```
void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs);
```

If you don't do this, a process may be running on two processors parallely

✖

```
void
yield(void)
{
...
release(&ptable.lock);
}
```

Release the lock held by some another process

✔

Your answer is partially correct.

You have correctly selected 3.
The correct answer is: struct proc*

```
myproc(void) {
...
  pushcli();
  c = mycpu();
  p = c->proc;
  popcli();
...
}
```

→ Disable interrupts to avoid another process's pointer being returned, static inline uint

```
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
```

→ Atomic compare and swap instruction (to be expanded inline into code), void

```
sleep(void *chan, struct spinlock *lk)
{
```

```
  ...
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  } → Avoid a self-deadlock, void
panic(char *s)
{
...
  panicked = 1; → Ensure that no printing happens on other processors, void
acquire(struct spinlock *lk)
{
  pushcli();
  → Disable interrupts to avoid deadlocks, void
acquire(struct spinlock *lk)
{
...
__sync_synchronize();



  → Tell compiler not to reorder memory access beyond this line, void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs);



  → Traverse ebp chain to get sequence of instructions followed in functions calls, void
yield(void)
{
...
release(&ptable.lock);
}



  → Release the lock held by some another process
```

Question **19**

Correct

Mark 1.00 out of 1.00

Calculate the average waiting time using
FCFS scheduling
for the following workload

assuming that they arrive in this order during the first time unit:

Process	Burst Time
P1	2
P2	6
P3	2
P4	3

Write only a number in the answer upto two decimal points.

Answer: 5 ✔

P2 waits for 2 units

P3 waits for 2+6 units

P4 waits for 2 + 6 +2 units of time

Total waiting = 2 + 2 + 6 + 2 + 6 + 2 = 20 units

Average waiting time = 20/4 = 5

The correct answer is: 5

Question **20**

Incorrect

Mark 0.00 out of 1.00

Select all correct statements about file system recovery (without journaling) programs e.g. fsck

Select one or more:

- [x] a. Recovery programs recalculate most of the metadata summaries (e.g. free inode count) ✔
- [x] b. It is possible to lose data as part of recovery ✔
- [x] c. Even with a write-through policy, it is possible to need a recovery program. ✔
- [x] d. They can make changes to the on-disk file system ✔
- [x] e. A recovery program, most typically, builds the file system data structure and checks for inconsistencies ✔
- [x] f. They may take very long time to execute ✔
- [x] g. They are used to recover deleted files ✖
- [ ] h. Recovery programs are needed only if the file system has a delayed-write policy.
- [ ] i. Recovery is possible due to redundancy in file system data structures

Your answer is incorrect.

The correct answers are: Recovery is possible due to redundancy in file system data structures, A recovery program, most typically, builds the file system data structure and checks for inconsistencies, It is possible to lose data as part of recovery, They may take very long time to execute, They can make changes to the on-disk file system, Recovery programs recalculate most of the metadata summaries (e.g. free inode count), Recovery programs are needed only if the file system has a delayed-write policy., Even with a write-through policy, it is possible to need a recovery program.

Question **21**

Correct

Mark 1.00 out of 1.00

Given that the memory access time is 150 ns, probability of a page fault is 0.8 and page fault handling time is 6 ms,
The effective memory access time in nanoseconds is:

Answer: 4800030 ✔

The correct answer is: 4800030.00

Question **22**

Correct

Mark 1.00 out of 1.00

Assuming a 8- KB page size, what is the page numbers for the address 1005699 reference in decimal :

(give answer also in decimal)

Answer: 123 ✔

The correct answer is: 123

Question **23**

Correct

Mark 1.00 out of 1.00

Mark the statements as True or False, w.r.t. passing of arguments to system calls in xv6 code.

True	False		
✗	✓	String arguments are first copied to trapframe and then from trapframe to kernel's other variables.	✔
✗	✓	The arguments to system call are copied to kernel stack in trapasm.S	✔
✓	✗	The arguments are accessed in the kernel code using esp on the trapframe.	✔
✓	✗	String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer	✔
✓	✗	Integer arguments are copied from user memory to kernel memory using argint()	✔
✓	✗	The arguments to system call originally reside on process stack.	✔
✗	✓	Integer arguments are stored in eax, ebx, ecx, etc. registers	✔
✓	✗	The functions like argint(), argstr() make the system call arguments available in the kernel.	✔

String arguments are first copied to trapframe and then from trapframe to kernel's other variables.: False
The arguments to system call are copied to kernel stack in trapasm.S: False
The arguments are accessed in the kernel code using esp on the trapframe.: True
String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer: True
Integer arguments are copied from user memory to kernel memory using argint(): True
The arguments to system call originally reside on process stack.: True
Integer arguments are stored in eax, ebx, ecx, etc. registers: False
The functions like argint(), argstr() make the system call arguments available in the kernel.: True

◄ Random Quiz - 6 (xv6 file system)

Jump to...

Homework questions: Basics of MM, xv6 booting ►

Started on	Saturday, 12 February 2022, 10:00:21 AM
State	Finished
Completed on	Saturday, 12 February 2022, 11:25:53 AM
Time taken	1 hour 25 mins
Grade	**4.94** out of 10.00 (**49**%)

Question **1**

Complete

Mark 0.00 out of 0.50

Select all the correct statements about code of bootmain() in xv6

```
void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* pa;

  elf = (struct elfhdr*)0x10000;  // scratch space

  // Read 1st page off disk
  readseg((uchar*)elf, 4096, 0);

  // Is this an ELF executable?
  if(elf->magic != ELF_MAGIC)
    return;  // let bootasm.S handle error

  // Load each program segment (ignores ph flags).
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
  for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
  }

  // Call the entry point from the ELF header.
  // Does not return!
  entry = (void(*)(void))(elf->entry);
  entry();
}
```

Also, inspect the relevant parts of the xv6 code. binary files, etc and run commands as you deem fit to answer this question.

- [x] a. The elf->entry is set by the linker in the kernel file and it's 0x80000000
- [ ] b. The condition if(ph->memsz > ph->filesz) is never true.
- [x] c. The readseg finally invokes the disk I/O code using assembly instructions
- [x] d. The elf->entry is set by the linker in the kernel file and it's 0x80000000
- [ ] e. The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded
- [x] f. The kernel file gets loaded at the Physical address 0x10000 +0x80000000 in memory.
- [x] g. The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it.
- [ ] h. The elf->entry is set by the linker in the kernel file and it's 8010000c
- [ ] i. The kernel file gets loaded at the Physical address 0x10000 in memory.
- [ ] j. The stosb() is used here, to fill in some space in memory with zeroes
- [ ] k. The kernel file has only two program headers

The correct answers are: The kernel file gets loaded at the Physical address 0x10000 in memory., The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it., The elf->entry is set by the linker in the kernel file and it's 8010000c, The readseg finally invokes the disk I/O code using assembly instructions, The stosb() is used here, to fill in some space in memory with zeroes, The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded, The kernel file has only two program headers

Question **2**

Complete

Mark 0.50 out of 0.50

What's the trapframe in xv6?

- a. A frame of memory that contains all the trap handler's addresses
- b. A frame of memory that contains all the trap handler code's function pointers
- c. The IDT table
- d. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S
- e. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by code in trapasm.S only
- f. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware only
- g. A frame of memory that contains all the trap handler code

The correct answer is: The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S

Question **3**

Complete

Mark 0.21 out of 0.50

Order the events that occur on a timer interrupt:

Jump to a code pointed by IDT	3
Jump to scheduler code	2
Select another process for execution	5
Change to kernel stack of currently running process	4
Set the context of the new process	6
Save the context of the currently running process	1
Execute the code of the new process	7

The correct answer is: Jump to a code pointed by IDT → 2, Jump to scheduler code → 4, Select another process for execution → 5, Change to kernel stack of currently running process → 1, Set the context of the new process → 6, Save the context of the currently running process → 3, Execute the code of the new process → 7

Question **4**

Complete

Mark 0.50 out of 0.50

Suppose a program does a scanf() call.

Essentially the scanf does a read() system call.

This call will obviously "block" waiting for the user input.

In terms of OS data structures and execution of code, what does it mean?

Select one:

- a. OS code for read() will move PCB of current process to a wait queue and call scheduler
- b. OS code for read() will call scheduler
- c. OS code for read() will move the PCB of this process to a wait queue and return from the system call
- d. read() will return and process will be taken to a wait queue
- e. read() returns and process calls scheduler()

The correct answer is: OS code for read() will move PCB of current process to a wait queue and call scheduler

Question **5**

Complete

Mark 0.50 out of 0.50

In bootasm.S, on the line

```
ljmp    $(SEG_KCODE<<3), $start32
```

The SEG_KCODE << 3, that is shifting of 1 by 3 bits is done because

- a. The value 8 is stored in code segment
- b. The code segment is 16 bit and only upper 13 bits are used for segment number
- c. While indexing the GDT using CS, the value in CS is always divided by 8
- d. The ljmp instruction does a divide by 8 on the first argument
- e. The code segment is 16 bit and only lower 13 bits are used for segment number

The correct answer is: The code segment is 16 bit and only upper 13 bits are used for segment number

Question **6**

Complete

Mark 0.40 out of 0.50

Select Yes if the mentioned element should be a part of PCB

Select No otherwise.

Yes	No	
◉	○	Memory management information about that process
◉	○	Process context
○	◉	Function pointers to all system calls
◉	○	PID
◉	○	EIP at the time of context switch
◉	○	List of opened files
◉	○	PID of Init
◉	○	Process state
○	◉	Pointer to the parent process
○	◉	Pointer to IDT

Memory management information about that process: Yes
Process context: Yes
Function pointers to all system calls: No
PID: Yes
EIP at the time of context switch: Yes
List of opened files: Yes
PID of Init: No
Process state: Yes
Pointer to the parent process: Yes
Pointer to IDT: No

Question **7**

Complete

Mark 0.40 out of 1.00

Mark the statements, w.r.t. the scheduler of xv6 as True or False

True	False	
○	●	The work of selecting and scheduling a process is done only in `scheduler()` and not in `sched()`
●	○	`swtch` is a function that saves old context, loads new context, and returns to last EIP in the new context
○	●	The variable c->scheduler on first processor uses the stack allocated entry.S
●	○	`sched()` and `scheduler()` are co-routines
○	●	The function `scheduler()` executes using the kernel-only stack
○	●	When a process is scheduled for execution, it resumes execution in `sched()` after the call to `swtch()`
○	●	`swtch` is a function that does not return to the caller
○	●	the control returns to `mycpu()->intena = intena; ();` after `swtch(&p->context, mycpu()->scheduler);` in `sched()`
○	●	the control returns to `switchkvm();` after `swtch(&(c->scheduler), p->context);` in `scheduler()`
●	○	`sched()` *calls* `scheduler()` and `scheduler()` *calls* `sched()`

The work of selecting and scheduling a process is done only in `scheduler()` and not in `sched()`: True
`swtch` is a function that saves old context, loads new context, and returns to last EIP in the new context: True
The variable c->scheduler on first processor uses the stack allocated entry.S: True
`sched()` and `scheduler()` are co-routines: True
The function `scheduler()` executes using the kernel-only stack: True
When a process is scheduled for execution, it resumes execution in `sched()` after the call to `swtch()`
: True
`swtch` is a function that does not return to the caller: True
the control returns to `mycpu()->intena = intena; ();` after `swtch(&p->context, mycpu()->scheduler);` in `sched()`: False
the control returns to `switchkvm();` after `swtch(&(c->scheduler), p->context);` in `scheduler()`: False
`sched()` *calls* `scheduler()` and `scheduler()` *calls* `sched()`: False

Question **8**

Complete

Mark 0.00 out of 0.50

Consider the following programs

exec1.c

```
#include <unistd.h>
#include <stdio.h>
int main() {
   execl("./exec2", "./exec2", NULL);
}
```

exec2.c

```
#include <unistd.h>
#include <stdio.h>
int main() {
   execl("/bin/ls", "/bin/ls", NULL);
  printf("hello\n");
}
```

Compiled as

```
cc    exec1.c   -o exec1
cc    exec2.c   -o exec2
```

And run as

```
$ ./exec1
```

Explain the output of the above command (./exec1)

Assume that /bin/ls , i.e. the 'ls' program exists.

Select one:

- a. "ls" runs on current directory
- b. Execution fails as the call to execl() in exec1 fails
- c. Execution fails as one exec can't invoke another exec
- d. Program prints hello
- e. Execution fails as the call to execl() in exec2 fails

The correct answer is: "ls" runs on current directory

Question **9**

Complete

Mark 0.00 out of 0.50

For each line of code mentioned on the left side, select the location of sp/esp that is in use

cli in bootasm.S	0x7c00 to 0
readseg((uchar*)elf, 4096, 0); in bootmain.c	Immaterial as the stack is not used here
ljmp $(SEG_KCODE<<3), $start32 in bootasm.S	0x10000 to 0x7c00
call bootmain in bootasm.S	The 4KB area in kernel image, loaded in memory, named as 'stack'
jmp *%eax in entry.S	0x7c00 to 0x10000

The correct answer is: cli
in bootasm.S → Immaterial as the stack is not used here, readseg((uchar*)elf, 4096, 0);
in bootmain.c → 0x7c00 to 0, ljmp $(SEG_KCODE<<3), $start32
in bootasm.S → Immaterial as the stack is not used here, call bootmain
in bootasm.S → 0x7c00 to 0, jmp *%eax
in entry.S → The 4KB area in kernel image, loaded in memory, named as 'stack'

Question **10**

Complete

Mark 0.63 out of 1.00

Select the correct statements about interrupt handling in xv6 code

- [x] a. xv6 uses the 64th entry in IDT for system calls
- [ ] b. xv6 uses the 0x64th entry in IDT for system calls
- [x] c. On any interrupt/syscall/exception the control first jumps in vectors.S
- [ ] d. The function trap() is the called irrespective of hardware interrupt/system-call/exception
- [x] e. Before going to alltraps, the kernel stack contains upto 5 entries.
- [ ] f. Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt
- [x] g. The trapframe pointer in struct proc, points to a location on kernel stack
- [ ] h. The function trap() is the called only in case of hardware interrupt
- [ ] i. The trapframe pointer in struct proc, points to a location on user stack
- [ ] j. On any interrupt/syscall/exception the control first jumps in trapasm.S
- [x] k. The CS and EIP are changed only after pushing user code's SS,ESP on stack
- [ ] l. The CS and EIP are changed only immediately on a hardware interrupt
- [ ] m. All the 256 entries in the IDT are filled

The correct answers are: All the 256 entries in the IDT are filled, Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt, xv6 uses the 64th entry in IDT for system calls, On any interrupt/syscall/exception the control first jumps in vectors.S, Before going to alltraps, the kernel stack contains upto 5 entries., The trapframe pointer in struct proc, points to a location on kernel stack, The function trap() is the called irrespective of hardware interrupt/system-call/exception, The CS and EIP are changed only after pushing user code's SS,ESP on stack

Question **11**

Complete

Mark 0.50 out of 0.50

Order the sequence of events, in scheduling process P1 after process P0

timer interrupt occurs	2
Process P0 is running	1
context of P1 is loaded from P1's PCB	4
Process P1 is running	6
Control is passed to P1	5
context of P0 is saved in P0's PCB	3

The correct answer is: timer interrupt occurs → 2, Process P0 is running → 1, context of P1 is loaded from P1's PCB → 4, Process P1 is running → 6, Control is passed to P1 → 5, context of P0 is saved in P0's PCB → 3

Question **12**

Complete

Mark 0.00 out of 1.00

Select the sequence of events that are NOT possible, assuming a non-interruptible kernel code

(Note: non-interruptible kernel code means, if the kernel code is executing, then interrupts will be disabled).

Note: A possible sequence may have some missing steps in between. An impossible sequence will will have n and n+1th steps such that n+1th step can not follow n'th step.

Select one or more:

- [ ] a. P1 running
  P1 makes system call
  timer interrupt
  Scheduler
  P2 running
  timer interrupt
  Scheuler
  P1 running
  P1's system call return

- [ ] b. P1 running
  P1 makes system call
  system call returns
  P1 running
  timer interrupt
  Scheduler running
  P2 running

- [ ] c. P1 running
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P3 running
  Hardware interrupt
  Interrupt unblocks P1
  Interrupt returns
  P3 running
  Timer interrupt
  Scheduler
  P1 running

- [x] d. P1 running
  keyboard hardware interrupt
  keyboard interrupt handler running
  interrupt handler returns
  P1 running
  P1 makes sytem call
  system call returns
  P1 running
  timer interrupt
  scheduler
  P2 running

- [ ] e.
  P1 running
  P1 makes sytem call
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again

- [ ] f. P1 running
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again

The correct answers are: P1 running
P1 makes sytem call and blocks
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again, P1 running
P1 makes system call
timer interrupt
Scheduler
P2 running
timer interrupt
Scheuler
P1 running
P1's system call return,
P1 running
P1 makes sytem call
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

Question **13**

Complete

Mark 0.50 out of 0.50

Some part of the bootloader of xv6 is written in assembly while some part is written in C. Why is that so?
Select all the appropriate choices

- [x] a. The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C
- [x] b. The setting up of the most essential memory management infrastructure needs assembly code
- [ ] c. The code in assembly is required for transition to protected mode, from real mode; but calling convention was applicable all the time
- [ ] d. The code for reading ELF file can not be written in assembly

The correct answers are: The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C, The setting up of the most essential memory management infrastructure needs assembly code

Question **14**

Complete

Mark 0.17 out of 0.50

The bootmain() function has this code

```
elf = (struct elfhdr*)0x10000;  // scratch space
readseg((uchar*)elf, 4096, 0);
```

Mark the statements as True or False with respect to this code.

In these statements 0x1000 is referred to as ADDRESS

True	False	
◉	○	The value ADDRESS is changed to a 0 the program could still work
◉	○	It the value of ADDRESS is changed to a higher number (upto a limit), the program could still work
◉	○	This line loads the kernel code at ADDRESS
◉	○	It the value of ADDRESS is changed to a lower number (upto a limit), the program could still work
◉	○	This line effectively loads the ELF header and the program headers at ADDRESS
◉	○	If the value of ADDRESS is changed, then the program will not work

The value ADDRESS is changed to a 0 the program could still work: False
It the value of ADDRESS is changed to a higher number (upto a limit), the program could still work: True
This line loads the kernel code at ADDRESS: False
It the value of ADDRESS is changed to a lower number (upto a limit), the program could still work: True
This line effectively loads the ELF header and the program headers at ADDRESS: False
If the value of ADDRESS is changed, then the program will not work: False

Question **15**

Complete

Mark 0.50 out of 1.00

Which parts of the xv6 code in bootasm.S bootmain.c , entry.S and in the codepath related to scheduler() and trap handling() can also be written in some other way, and still ensure that xv6 works properly?

Writing code is not necessary. You only need to comment on which part of the code could be changed to something else or written in another fashion.

Maximum two points to be written.

We can use a scheduling algorithm. We can use the kernel stack in scheduler function in entrty.S and bootmain.c .

Question **16**

Complete

Mark 0.13 out of 0.50

Select all the correct statements about zombie processes

Select one or more:

- [ ] a. If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent
- [ ] b. Zombie processes are harmless even if OS is up for long time
- [x] c. A zombie process occupies space in OS data structures
- [ ] d. A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it
- [x] e. A process can become zombie if it finishes, but the parent has finished before it
- [x] f. init() typically keeps calling wait() for zombie processes to get cleaned up
- [ ] g. A process becomes zombie when it's parent finishes
- [x] h. A zombie process remains zombie forever, as there is no way to clean it up

The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it, A process can become zombie if it finishes, but the parent has finished before it, A zombie process occupies space in OS data structures, If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

◄ Extra Reading on Linkers: A writeup by Ian Taylor (keep changing url string from 38 to 39, and so on)

Jump to...

(Code) IPC - Shm, Messages ►

Started on	Saturday, 20 February 2021, 2:51 PM
State	Finished
Completed on	Saturday, 20 February 2021, 3:55 PM
Time taken	1 hour 3 mins
Grade	**7.30** out of 20.00 (**37**%)

Question **1**

Partially correct

Mark 0.80 out of 1.00

Select all the correct statements about the state of a process.

- [x] a. A process can self-terminate only when it's running ✔
- [x] b. Typically, it's represented as a number in the PCB ✔
- [x] c. A process that is running is not on the ready queue ✔
- [x] d. Processes in the ready queue are in the ready state ✔
- [ ] e. It is not maintained in the data structures by kernel, it is only for conceptual understanding of programmers
- [x] f. Changing from running state to waiting state results in "giving up the CPU" ✔
- [ ] g. A process in ready state is ready to receive interrupts
- [x] h. A waiting process starts running after the wait is over ✖
- [x] i. A process changes from running to ready state on a timer interrupt ✔
- [x] j. A process in ready state is ready to be scheduled ✔
- [x] k. A running process may terminate, or go to wait or become ready again ✔
- [x] l. A process waiting for I/O completion is typically woken up by the particular interrupt handler code ✔
- [ ] m. A process waiting for any condition is woken up by another process only
- [ ] n. A process changes from running to ready state on a timer interrupt or any I/O wait

Your answer is partially correct.

You have selected too many options.
The correct answers are: Typically, it's represented as a number in the PCB, A process in ready state is ready to be scheduled, Processes in the ready queue are in the ready state, A process that is running is not on the ready queue, A running process may terminate, or go to wait or become ready again, A process changes from running to ready state on a timer interrupt, Changing from running state to waiting state results in "giving up the CPU", A process can self-terminate only when it's running, A process waiting for I/O completion is typically woken up by the particular interrupt handler code

Question **2**

Incorrect

Mark 0.00 out of 1.00

For each line of code mentioned on the left side, select the location of sp/esp that is in use

jmp *%eax in entry.S	0x7c00 to 0x10000	✖
ljmp $(SEG_KCODE<<3), $start32 in bootasm.S	0x10000 to 0x7c00	✖
call bootmain in bootasm.S	0x7c00 to 0x10000	✖
cli in bootasm.S	0x7c00 to 0	✖
readseg((uchar*)elf, 4096, 0); in bootmain.c	The 4KB area in kernel image, loaded in memory, named as 'stack'	✖

Your answer is incorrect.

The correct answer is: jmp *%eax in entry.S → The 4KB area in kernel image, loaded in memory, named as 'stack', ljmp $(SEG_KCODE<<3), $start32 in bootasm.S → Immaterial as the stack is not used here, call bootmain in bootasm.S → 0x7c00 to 0, cli in bootasm.S → Immaterial as the stack is not used here, readseg((uchar*)elf, 4096, 0); in bootmain.c → 0x7c00 to 0

Question **3**

Correct

Mark 0.25 out of 0.25

Order the following events in boot process (from 1 onwards)

Boot loader	2	✔
Shell	6	✔
BIOS	1	✔
OS	3	✔
Init	4	✔
Login interface	5	✔

Your answer is correct.

The correct answer is: Boot loader → 2, Shell → 6, BIOS → 1, OS → 3, Init → 4, Login interface → 5

Question **4**

Partially correct

Mark 0.30 out of 0.50

Consider the following command and it's output:

```
$ ls -lht xv6.img kernel
-rw-rw-r-- 1 abhijit abhijit 4.9M Feb 15 11:09 xv6.img
-rwxrwxr-x 1 abhijit abhijit 209K Feb 15 11:09 kernel*
```

Following code in bootmain()

```
  readseg((uchar*)elf, 4096, 0);
```

and following selected lines from Makefile

```
xv6.img: bootblock kernel
    dd if=/dev/zero of=xv6.img count=10000
    dd if=bootblock of=xv6.img conv=notrunc
    dd if=kernel of=xv6.img seek=1 conv=notrunc

kernel: $(OBJS) entry.o entryother initcode kernel.ld
    $(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode entryother
    $(OBJDUMP) -S kernel > kernel.asm
    $(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > kernel.sym
```

Also read the code of bootmain() in xv6 kernel.
Select the options that describe the meaning of these lines and their correlation.

- [ ] a. Althought the size of the kernel file is 209 Kb, only 4Kb out of it is the actual kernel code and remaining part is all zeroes.
- [x] b. The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files ✔
- [x] c. The kernel.ld file contains instructions to the linker to link the kernel properly ✔
- [ ] d. The bootmain() code does not read the kernel completely in memory
- [ ] e. readseg() reads first 4k bytes of kernel in memory
- [ ] f. Althought the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.
- [ ] g. The kernel.asm file is the final kernel file
- [ ] h. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is not read as it is user programs.
- [x] i. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain(). ✔

Your answer is partially correct.

You have correctly selected 3.
The correct answers are: The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain()., readseg() reads first 4k bytes of kernel in memory, The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files, The kernel.ld file contains instructions to the linker to link the kernel properly, Althought the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.

Question **5**

Partially correct

Mark 0.50 out of 1.00

```
int f() {
    int count;
    for (count = 0; count< 2; count ++) {
        if (fork() ==0)
            printf("Operating-System\n");
     }
     printf("TYCOMP\n");
 }
```

The number of times "Operating-System" is printed, is:

Answer: 3

The correct answer is: 7.00

Question **6**

Partially correct

Mark 0.40 out of 0.50

Select Yes/True if the mentioned element must be a part of PCB

Select No/False otherwise.

Yes	No		
◉☑	○✗	PID	✔
◉☑	○✗	Process context	✔
◉☑	○✗	List of opened files	✔
◉☑	○✗	Process state	✔
◉✗	○☑	Parent's PID	✖
○✗	◉☑	Pointer to IDT	✔
○✗	◉☑	Function pointers to all system calls	✔
◉☑	○✗	Memory management information about that process	✔
○☑	◉✗	Pointer to the parent process	✖
◉☑	○✗	EIP at the time of context switch	✔

PID: Yes
Process context: Yes
List of opened files: Yes
Process state: Yes
Parent's PID: No
Pointer to IDT: No
Function pointers to all system calls: No
Memory management information about that process: Yes
Pointer to the parent process: Yes
EIP at the time of context switch: Yes

Question **7**

Incorrect

Mark 0.00 out of 1.00

Select all the correct statements about code of bootmain() in xv6

```
void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* pa;

  elf = (struct elfhdr*)0x10000;  // scratch space

  // Read 1st page off disk
  readseg((uchar*)elf, 4096, 0);

  // Is this an ELF executable?
  if(elf->magic != ELF_MAGIC)
    return;  // let bootasm.S handle error

  // Load each program segment (ignores ph flags).
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
  for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
  }

  // Call the entry point from the ELF header.
  // Does not return!
  entry = (void(*)(void))(elf->entry);
  entry();
}
```

Also, inspect the relevant parts of the xv6 code. binary files, etc and run commands as you deem fit to answer this question.

- a. The kernel file gets loaded at the Physical address 0x10000 +0x80000000 in memory. ✗
- b. The elf->entry is set by the linker in the kernel file and it's 0x80000000 ✗
- c. The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded ✓
- d. The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it. ✓
- e. The kernel file has only two program headers ✓
- f. The elf->entry is set by the linker in the kernel file and it's 0x80000000 ✗
- g. The readseg finally invokes the disk I/O code using assembly instructions ✓
- h. The elf->entry is set by the linker in the kernel file and it's 8010000c ✓
- i. The kernel file gets loaded at the Physical address 0x10000 in memory. ✓
- j. The condition if(ph->memsz > ph->filesz) is never true. ✗
- k. The stosb() is used here, to fill in some space in memory with zeroes ✓

Your answer is incorrect.

The correct answers are: The kernel file gets loaded at the Physical address 0x10000 in memory., The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it., The elf->entry is set by the linker in the kernel file and it's 8010000c, The readseg finally invokes the disk I/O code using assembly instructions, The stosb() is used here, to fill in some space in memory with zeroes, The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded, The kernel file has only two program headers

Question **8**

Partially correct

Mark 0.13 out of 0.25

Which of the following are NOT a part of job of a typical compiler?

- [x] a. Check the program for logical errors ✔
- [ ] b. Convert high level langauge code to machine code
- [ ] c. Process the # directives in a C program
- [ ] d. Invoke the linker to link the function calls with their code, extern globals with their declaration
- [ ] e. Check the program for syntactical errors
- [ ] f. Suggest alternative pieces of code that can be written

Your answer is partially correct.

You have correctly selected 1.
The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

Question **9**

Correct

Mark 0.25 out of 0.25

Rank the following storage systems from slowest (first) to fastest(last)

Cache	6	✔
Hard Disk	3	✔
RAM	5	✔
Optical Disks	2	✔
Non volatile memory	4	✔
Registers	7	✔
Magnetic Tapes	1	✔

Your answer is correct.

The correct answer is: Cache → 6, Hard Disk → 3, RAM → 5, Optical Disks → 2, Non volatile memory → 4, Registers → 7, Magnetic Tapes → 1

Question **10**

Partially correct

Mark 0.21 out of 0.50

Which of the following parts of a C program do not have any corresponding machine code ?

- [ ] a. local variable declaration
- [ ] b. global variables
- [x] c. function calls ✗
- [x] d. #directives ✓
- [ ] e. expressions
- [ ] f. pointer dereference
- [x] g. typedefs ✓

Your answer is partially correct.

You have correctly selected 2.
The correct answers are: #directives, typedefs, global variables

Question **11**

Correct

Mark 0.25 out of 0.25

Match a system call with it's description

pipe	create an unnamed FIFO storage with 2 ends - one for reading and another for writing	✓
dup	create a copy of the specified file descriptor into smallest available file descriptor	✓
dup2	create a copy of the specified file descriptor into another specified file descriptor	✓
exec	execute a binary file overlaying the image of current process	✓
fork	create an identical child process	✓

Your answer is correct.

The correct answer is: pipe → create an unnamed FIFO storage with 2 ends - one for reading and another for writing, dup → create a copy of the specified file descriptor into smallest available file descriptor, dup2 → create a copy of the specified file descriptor into another specified file descriptor, exec → execute a binary file overlaying the image of current process, fork → create an identical child process

Question **12**

Correct

Mark 0.25 out of 0.25

Match the register with the segment used with it.

eip	cs	✔
edi	es	✔
esi	ds	✔
ebp	ss	✔
esp	ss	✔

Your answer is correct.

The correct answer is: eip → cs, edi → es, esi → ds, ebp → ss, esp → ss

Question **13**

Correct

Mark 0.25 out of 0.25

What's the trapframe in xv6?

- a. A frame of memory that contains all the trap handler code
- b. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware only
- c. The IDT table
- d. A frame of memory that contains all the trap handler code's function pointers
- e. A frame of memory that contains all the trap handler's addresses
- f. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S ✔
- g. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by code in trapasm.S only

Your answer is correct.

The correct answer is: The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S

Question **14**

Incorrect

Mark 0.00 out of 0.50

Select all the correct statements about linking and loading.

Select one or more:

- a. Continuous memory management schemes can support dynamic linking and dynamic loading. ✗
- b. Loader is last stage of the linker program ✗
- c. Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently) ✓
- d. Dynamic linking and loading is not possible without demand paging or demand segmentation. ✓
- e. Dynamic linking essentially results in relocatable code. ✓
- f. Continuous memory management schemes can support static linking and static loading. (may be inefficiently) ✓
- g. Loader is part of the operating system ✓
- h. Static linking leads to non-relocatable code ✗
- i. Dynamic linking is possible with continous memory management, but variable sized partitions only. ✗

Your answer is incorrect.

The correct answers are: Continuous memory management schemes can support static linking and static loading. (may be inefficiently), Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently), Dynamic linking essentially results in relocatable code., Loader is part of the operating system, Dynamic linking and loading is not possible without demand paging or demand segmentation.

Question **15**

Incorrect

Mark 0.00 out of 0.25

In bootasm.S, on the line

```
ljmp    $(SEG_KCODE<<3), $start32
```

The SEG_KCODE << 3, that is shifting of 1 by 3 bits is done because

- a. The value 8 is stored in code segment
- b. The code segment is 16 bit and only upper 13 bits are used for segment number
- c. The code segment is 16 bit and only lower 13 bits are used for segment number ✗
- d. While indexing the GDT using CS, the value in CS is always divided by 8
- e. The ljmp instruction does a divide by 8 on the first argument

Your answer is incorrect.

The correct answer is: The code segment is 16 bit and only upper 13 bits are used for segment number

Question **16**

Partially correct

Mark 0.07 out of 0.50

Order the events that occur on a timer interrupt:

Change to kernel stack	1	✗
Jump to a code pointed by IDT	2	✗
Jump to scheduler code	5	✗
Set the context of the new process	4	✗
Save the context of the currently running process	3	✓
Execute the code of the new process	6	✗
Select another process for execution	7	✗

Your answer is partially correct.

You have correctly selected 1.

The correct answer is: Change to kernel stack → 2, Jump to a code pointed by IDT → 1, Jump to scheduler code → 4, Set the context of the new process → 6, Save the context of the currently running process → 3, Execute the code of the new process → 7, Select another process for execution → 5

Question **17**

Incorrect

Mark 0.00 out of 1.00

Consider the two programs given below to implement the command (ignore the fact that error checks are not done on return values of functions)

$ ls . /tmp/asdfksdf >/tmp/ddd 2>&1

Program 1

```
int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(1);
    dup(fd);
    close(2);
    dup(fd);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Program 2

```
int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    close(1);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(2);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Select all the correct statements about the programs

Select one or more:

- a. Both programs are correct ✗
- b. Program 2 makes sure that there is one file offset used for '2' and '1' ✗
- c. Only Program 2 is correct ✗
- d. Program 2 does 1>&2 ✗
- e. Program 2 ensures 2>&1 and does not ensure > /tmp/ddd ✗
- f. Program 1 makes sure that there is one file offset used for '2' and '1' ✓
- g. Program 1 is correct for > /tmp/ddd but not for 2>&1 ✗
- h. Program 1 does 1>&2 ✗
- i. Both program 1 and 2 are incorrect ✗
- j. Program 2 is correct for > /tmp/ddd but not for 2>&1 ✗
- k. Only Program 1 is correct ✓
- l. Program 1 ensures 2>&1 and does not ensure > /tmp/ddd ✗

Your answer is incorrect.

The correct answers are: Only Program 1 is correct, Program 1 makes sure that there is one file offset used for '2' and '1'

Question **18**

Correct

Mark 0.25 out of 0.25

Select the option which best describes what the CPU does during it's powered ON lifetime

- a. Ask the user what is to be done, and execute that task
- b. Ask the OS what is to be done, and execute that task
- c. Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, Ask the User or the OS what is to be done next, repeat
- d. Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, repeat ✔
- e. Fetch instruction specified by OS, Decode and execute it, repeat
- f. Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, Ask OS what is to be done next, repeat

The correct answer is: Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, repeat

Question **19**

Partially correct

Mark 0.86 out of 1.00

Consider the following code and MAP the file to which each fd points at the end of the code.

```
int main(int argc, char *argv[]) {
   int fd1, fd2 = 1, fd3 = 1, fd4 = 1;

   fd1 = open("/tmp/1", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
   fd2 = open("/tmp/2", O_RDDONLY);
   fd3 = open("/tmp/3", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
   close(0);
   close(1);
   dup(fd2);
   dup(fd3);
   close(fd3);
   dup2(fd2, fd4);
   printf("%d %d %d %d\n", fd1, fd2, fd3, fd4);
   return 0;
}
```

1	closed	✖
fd4	/tmp/2	✔
fd2	/tmp/2	✔
fd1	/tmp/1	✔
2	stderr	✔
0	/tmp/2	✔
fd3	closed	✔

Your answer is partially correct.

You have correctly selected 6.

The correct answer is: 1 → /tmp/3, fd4 → /tmp/2, fd2 → /tmp/2, fd1 → /tmp/1, 2 → stderr, 0 → /tmp/2, fd3 → closed

Question **20**

Incorrect

Mark 0.00 out of 2.00

Following code claims to implement the command

/bin/ls -l | /usr/bin/head -3 | /usr/bin/tail -1

Fill in the blanks to make the code work.

Note: Do not include space in writing any option. x[1][2] should be written without any space, and so is the case with [1] or [2]. Pay attention to exact syntax and do not write any extra character like ';' or = etc.

```
int main(int argc, char *argv[]) {
   int pid1, pid2;
   int pfd[
```

1

✖][2];

```
   pipe(
```

2

✖);

```
   pid1 =
```

3

✖ ;

```
   if(pid1 != 0) {
      close(pfd[0]
```

0

✖);

```
      close(
```

pid1

✖);

```
      dup(
```

pid2

✖);

```
      execl("/bin/ls", "/bin/ls", "
```

1

✖ ", NULL);

```
   }
   pipe(
```

✖);

✖ = fork();

```
   if(pid2 == 0) {
      close(
```

✖ ;

```
      close(0);
      dup(
```

✖);

```
      close(pfd[1]
```

```
✖ );
    close(
✖ );
    dup(
✖ );
    execl("/usr/bin/head", "/usr/bin/head", "
✖ ", NULL);
  } else {
    close(pfd
✖ );
    close(
✖ );
    dup(
✖ );
    close(pfd
✖ );
    execl("/usr/bin/tail", "/usr/bin/tail", "
✖ ", NULL);
 }
}
```

Question **21**

Partially correct

Mark 0.11 out of 1.00

Select all the correct statements about calling convention on x86 32-bit.

- [x] a. Return address is one location above the ebp ✔
- [x] b. Parameters may be passed in registers or on stack ✔
- [x] c. Space for local variables is allocated by substracting the stack pointer inside the code of the called function ✔
- [x] d. The ebp pointers saved on the stack constitute a chain of activation records ✔
- [x] e. The two lines in the beginning of each function, "push %ebp; mov %esp, %ebp", create space for local variables ✖
- [x] f. Parameters may be passed in registers or on stack ✔
- [x] g. The return value is either stored on the stack or returned in the eax register ✖
- [ ] h. Paramters are pushed on the stack in left-right order
- [ ] i. during execution of a function, ebp is pointing to the old ebp
- [x] j. Space for local variables is allocated by substracting the stack pointer inside the code of the caller function ✖
- [x] k. Compiler may allocate more memory on stack than needed ✔

Your answer is partially correct.

You have selected too many options.
The correct answers are: Compiler may allocate more memory on stack than needed, Parameters may be passed in registers or on stack, Parameters may be passed in registers or on stack, Return address is one location above the ebp, during execution of a function, ebp is pointing to the old ebp, Space for local variables is allocated by substracting the stack pointer inside the code of the called function, The ebp pointers saved on the stack constitute a chain of activation records

Question **22**

Correct

Mark 1.00 out of 1.00

Match the program with it's output (ignore newlines in the output. Just focus on the count of the number of 'hi')

main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }	hi	✔
main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }	hi hi	✔
main() { int i = NULL; fork(); printf("hi\n"); }	hi hi	✔
main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }	hi	✔

Your answer is correct.

The correct answer is: main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi, main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi hi, main() { int i = NULL; fork(); printf("hi\n"); } → hi hi, main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi

Question **23**

Incorrect

Mark 0.00 out of 0.50

Some part of the bootloader of xv6 is written in assembly while some part is written in C. Why is that so?
Select all the appropriate choices

- a. The code in assembly is required for transition to protected mode, from real mode; but calling convention was applicable all the time ✗
- b. The setting up of the most essential memory management infrastructure needs assembly code ✓
- c. The code for reading ELF file can not be written in assembly ✗
- d. The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C ✓

Your answer is incorrect.

The correct answers are: The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C, The setting up of the most essential memory management infrastructure needs assembly code

Question **24**

Incorrect

Mark 0.00 out of 0.50

xv6.img: bootblock kernel

```
    dd if=/dev/zero of=xv6.img count=10000
    dd if=bootblock of=xv6.img conv=notrunc
    dd if=kernel of=xv6.img seek=1 conv=notrunc
```

Consider above lines from the Makefile. Which of the following is incorrect?

- [x] a. The size of the kernel file is nearly 5 MB ✔
- [x] b. The kernel is located at block-1 of the xv6.img ✖
- [x] c. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies 10,000 blocks on the disk. ✖
- [ ] d. The size of xv6.img is exactly = (size of bootblock) + (size of kernel)
- [x] e. The bootblock is located on block-0 of the xv6.img ✖
- [x] f. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk. ✔
- [x] g. The bootblock may be 512 bytes or less (looking at the Makefile instruction) ✖
- [x] h. The xv6.img is the virtual disk that is created by combining the bootblock and the kernel file. ✖
- [x] i. The size of the xv6.img is nearly 5 MB ✖
- [x] j. xv6.img is the virtual processor used by the qemu emulator ✔
- [x] k. Blocks in xv6.img after kernel may be all zeroes. ✖

Your answer is incorrect.

The correct answers are: xv6.img is the virtual processor used by the qemu emulator, The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk., The size of the kernel file is nearly 5 MB, The size of xv6.img is exactly = (size of bootblock) + (size of kernel)

Question **25**

Incorrect

Mark 0.00 out of 1.00

Select the sequence of events that are NOT possible, assuming a non-interruptible kernel code

Select one or more:

- [ ] a. P1 running
  P1 makes system call
  timer interrupt
  Scheduler
  P2 running
  timer interrupt
  Scheuler
  P1 running
  P1's system call return

- [x] b. P1 running ✔
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again

- [ ] c. P1 running
  P1 makes system call
  system call returns
  P1 running
  timer interrupt
  Scheduler running
  P2 running

- [x] d. P1 running ✖
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P3 running
  Hardware interrupt
  Interrupt unblocks P1
  Interrupt returns
  P3 running
  Timer interrupt
  Scheduler
  P1 running

- [ ] e.
  P1 running
  P1 makes sytem call
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again

- [x] f. P1 running ✖
  keyboard hardware interrupt
  keyboard interrupt handler running
  interrupt handler returns
  P1 running
  P1 makes sytem call
  system call returns

P1 running
timer interrupt
scheduler
P2 running

Your answer is incorrect.

The correct answers are: P1 running
P1 makes sytem call and blocks
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again, P1 running
P1 makes system call
timer interrupt
Scheduler
P2 running
timer interrupt
Scheuler
P1 running
P1's system call return,
P1 running
P1 makes sytem call
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

Question **26**

Correct

Mark 0.25 out of 0.25

Which of the following are the files related to bootloader in xv6?

- a. bootasm.s and entry.S
- b. bootasm.S and bootmain.c ✔
- c. bootasm.S, bootmain.c and bootblock.c
- d. bootmain.c and bootblock.S

Your answer is correct.

The correct answer is: bootasm.S and bootmain.c

Question **27**

Correct

Mark 0.25 out of 0.25

Match the following parts of a C program to the layout of the process in memory

Instructions	Text section	✔
Local Variables	Stack Section	✔
Dynamically allocated memory	Heap Section	✔
Global and static data	Data section	✔

Your answer is correct.

The correct answer is:
Instructions → Text section, Local Variables → Stack Section,
Dynamically allocated memory → Heap Section,
Global and static data → Data section

Question **28**

Incorrect

Mark 0.00 out of 0.50

What will this program do?

```
int main() {
fork();
execl("/bin/ls", "/bin/ls", NULL);
printf("hello");
}
```

- a. one process will run ls, another will print hello
- b. run ls once ✖
- c. run ls twice
- d. run ls twice and print hello twice
- e. run ls twice and print hello twice, but output will appear in some random order

Your answer is incorrect.

The correct answer is: run ls twice

Question **29**

Correct

Mark 0.25 out of 0.25

What is the OS Kernel?

- a. The code that controls hardware, abstracts access to hardware resources using system calls, creates an environment for processes to be created and run ✔ correct
- b. The set of tools like compiler, linker, loader, terminal, shell, etc.
- c. Only the system programs like compiler, linker, loader, etc.
- d. Everything that I see on my screen

The correct answer is: The code that controls hardware, abstracts access to hardware resources using system calls, creates an environment for processes to be created and run

Question **30**

Correct

Mark 0.50 out of 0.50

Which of the following is/are not saved during context switch?

- [ ] a. Program Counter
- [ ] b. General Purpose Registers
- [x] c. Bus ✔
- [ ] d. Stack Pointer
- [ ] e. MMU related registers/information
- [x] f. Cache ✔
- [x] g. TLB ✔

Your answer is correct.

The correct answers are: TLB, Cache, Bus

Question **31**

Partially correct

Mark 0.10 out of 0.25

Select the order in which the various stages of a compiler execute.

Linking	3	✗
Syntatical Analysis	2	✓
Pre-processing	1	✓
Intermediate code generation	does not exist	✗
Loading	4	✗

Your answer is partially correct.

You have correctly selected 2.
The correct answer is: Linking → 4, Syntatical Analysis → 2, Pre-processing → 1, Intermediate code generation → 3, Loading → does not exist

Question **32**

Partially correct

Mark 0.08 out of 0.50

Order the sequence of events, in scheduling process P1 after process P0

context of P0 is saved in P0's PCB	2	✗
context of P1 is loaded from P1's PCB	3	✗
Process P1 is running	5	✗
timer interrupt occurs	6	✗
Process P0 is running	1	✓
Control is passed to P1	4	✗

Your answer is partially correct.

You have correctly selected 1.
The correct answer is: context of P0 is saved in P0's PCB → 3, context of P1 is loaded from P1's PCB → 4, Process P1 is running → 6, timer interrupt occurs → 2, Process P0 is running → 1, Control is passed to P1 → 5

Question **33**

Not answered

Marked out of 1.00

Select the correct statements about interrupt handling in xv6 code

- a. On any interrupt/syscall/exception the control first jumps in vectors.S
- b. The trapframe pointer in struct proc, points to a location on user stack
- c. Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt
- d. xv6 uses the 64th entry in IDT for system calls
- e. The CS and EIP are changed only after pushing user code's SS,ESP on stack
- f. The trapframe pointer in struct proc, points to a location on kernel stack
- g. The function trap() is the called only in case of hardware interrupt
- h. The CS and EIP are changed only immediately on a hardware interrupt
- i. All the 256 entries in the IDT are filled
- j. On any interrupt/syscall/exception the control first jumps in trapasm.S
- k. The function trap() is the called irrespective of hardware interrupt/system-call/exception
- l. xv6 uses the 0x64th entry in IDT for system calls
- m. Before going to alltraps, the kernel stack contains upto 5 entries.

Your answer is incorrect.

The correct answers are: All the 256 entries in the IDT are filled, Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt, xv6 uses the 64th entry in IDT for system calls, On any interrupt/syscall/exception the control first jumps in vectors.S, Before going to alltraps, the kernel stack contains upto 5 entries., The trapframe pointer in struct proc, points to a location on kernel stack, The function trap() is the called irrespective of hardware interrupt/system-call/exception, The CS and EIP are changed only after pushing user code's SS,ESP on stack

◄ (Assignment) Change free list management in xv6

Jump to...

Started on	Tuesday, 22 March 2022, 1:59:34 PM
State	Finished
Completed on	Tuesday, 22 March 2022, 4:14:53 PM
Time taken	2 hours 15 mins
Grade	**24.57** out of 40.00 (**61**%)

Question **1**

Partially correct

Mark 0.10 out of 1.00

Select all the correct statements w.r.t user and kernel threads

Select one or more:

- [x] a. many-one model can be implemented even if there are no kernel threads ✔
- [ ] b. many-one model gives no speedup on multicore processors
- [x] c. all three models, that is many-one, one-one, many-many , require a user level thread library ✔
- [ ] d. A process blocks in many-one model even if a single thread makes a blocking system call
- [x] e. A process may not block in many-one model, if a thread makes a blocking system call ✖
- [x] f. one-one model increases kernel's scheduling load ✔
- [ ] g. one-one model can be implemented even if there are no kernel threads

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: many-one model can be implemented even if there are no kernel threads, all three models, that is many-one, one-one, many-many , require a user level thread library, one-one model increases kernel's scheduling load, many-one model gives no speedup on multicore processors, A process blocks in many-one model even if a single thread makes a blocking system call

Question **2**

Partially correct

Mark 0.50 out of 1.00

Select all correct statements w.r.t. Major and Minor page faults on Linux

- [ ] a. Thrashing is possible only due to major page faults
- [ ] b. Minor page fault may occur because the page was freed, but still tagged and available in the free page list
- [x] c. Minor page fault may occur because of a page fault during fork(), on code of an already running process ✔
- [x] d. Major page faults are likely to occur in more numbers at the beginning of the process ✔
- [x] e. Minor page fault may occur because the page was a shared memory page ✔
- [ ] f. Minor page faults are an improvement of the page buffering techniques

The correct answers are: Minor page fault may occur because the page was a shared memory page, Minor page fault may occur because of a page fault during fork(), on code of an already running process, Minor page fault may occur because the page was freed, but still tagged and available in the free page list, Major page faults are likely to occur in more numbers at the beginning of the process, Thrashing is possible only due to major page faults, Minor page faults are an improvement of the page buffering techniques

Question **3**

Correct

Mark 2.00 out of 2.00

W.r.t. Memory management in xv6,

xv6 uses physical memory upto 224 MB onlyMark statements True or False

True	False		
◉	○	The stack allocated in entry.S is used as stack for scheduler's context for first processor	✔
○	◉	The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context	✔
◉	○	The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir	✔
◉	○	The kernel code and data take up less than 2 MB space	✔
◉	○	The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context	✔
◉	○	xv6 uses physical memory upto 224 MB only	✔
◉	○	The free page-frame are created out of nearly 222 MB	✔
○	◉	The switchkvm() call in scheduler() changes CR3 to use page directory of new process	✔
◉	○	PHYSTOP can be increased to some extent, simply by editing memlayout.h	✔
◉	○	The process's address space gets mapped on frames, obtained from ~2MB:224MB range	✔
○	◉	The kernel's page table given by kpgdir variable is used as stack for scheduler's context	✔

The stack allocated in entry.S is used as stack for scheduler's context for first processor: True
The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context: False
The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir: True
The kernel code and data take up less than 2 MB space: True
The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context: True
xv6 uses physical memory upto 224 MB only: True
The free page-frame are created out of nearly 222 MB: True
The switchkvm() call in scheduler() changes CR3 to use page directory of new process: False
PHYSTOP can be increased to some extent, simply by editing memlayout.h: True
The process's address space gets mapped on frames, obtained from ~2MB:224MB range: True
The kernel's page table given by kpgdir variable is used as stack for scheduler's context: False

Question **4**

Correct

Mark 1.00 out of 1.00

Given that a kernel has 1000 KB of total memory, and holes of sizes (in that order) 300 KB, 200 KB, 100 KB, 250 KB. For each of the requests on the left side, match it with the chunk chosen using the specified algorithm.

Consider each request as first request.

50 KB, worst fit	300 KB	✔
100 KB, worst fit	300 KB	✔
200 KB, first fit	300 KB	✔
150 KB, best fit	200 KB	✔
220 KB, best fit	250 KB	✔
150 KB, first fit	300 KB	✔

The correct answer is: 50 KB, worst fit → 300 KB, 100 KB, worst fit → 300 KB, 200 KB, first fit → 300 KB, 150 KB, best fit → 200 KB, 220 KB, best fit → 250 KB, 150 KB, first fit → 300 KB

Question **5**

Partially correct

Mark 0.60 out of 1.00

Choice of the global or local replacement strategy is a subjective choice for kernel programmers. There are advantages and disadvantages on either side. Out of the following statements, that advocate either global or local replacement strategy, select those statements that have a logically CONSISTENT argument. (That is any statement that is logically correct about either global or local replacement)

Consistent	Inconsistent		
◉☑	○✖	Local replacement can lead to under-utilisation of memory, because a process may not use all the pages allocated to it all the time.	✔
○☑	◉✖	Global replacement may give highly variable per process completion time because number of page faults become un-predictable.	✖
◉☑	○✖	Global replacement can be preferred when greater throughput (number of processes completing per unit time) is a concern, because each process tries to complete at the expense of others, thus leading to overall more processes completing (unless thrashing occurs).	✔
◉☑	○✖	Local replacement results in more predictable per-process completion time because number of page faults can be better predicted.	✔
○☑	◉✖	Local replacement can be preferred when avoiding thrashing is a major concern because with local replacement and minimum number of frames allocated, a process is always able to progress and cascading inter-process page faults are avoided.	✖

Local replacement can lead to under-utilisation of memory, because a process may not use all the pages allocated to it all the time.: Consistent
Global replacement may give highly variable per process completion time because number of page faults become un-predictable.: Consistent
Global replacement can be preferred when greater throughput (number of processes completing per unit time) is a concern, because each process tries to complete at the expense of others, thus leading to overall more processes completing (unless thrashing occurs).: Consistent
Local replacement results in more predictable per-process completion time because number of page faults can be better predicted.: Consistent
Local replacement can be preferred when avoiding thrashing is a major concern because with local replacement and minimum number of frames allocated, a process is always able to progress and cascading inter-process page faults are avoided.: Consistent

Question **6**

Correct

Mark 1.00 out of 1.00

Map the functionality/use with function/variable in xv6 code.

Setup kernel part of a page table, and switch to that page table	kvmalloc() ✔
Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices	setupkvm() ✔
return a free page, if available; 0, otherwise	kalloc() ✔
Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed	mappages() ✔
Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary	walkpgdir() ✔
Array listing the kernel memory mappings, to be used by setupkvm()	kmap[] ✔

Your answer is correct.

The correct answer is: Setup kernel part of a page table, and switch to that page table → kvmalloc(), Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices → setupkvm(), return a free page, if available; 0, otherwise → kalloc(), Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed → mappages(), Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary → walkpgdir(), Array listing the kernel memory mappings, to be used by setupkvm() → kmap[]

Question **7**

Partially correct

Mark 0.20 out of 1.00

Mark statements True/False w.r.t. change of states of a process. Note that a statement is true only if the claim and argument both are true.

Reference: The process state diagram (and your understanding of how kernel code works). Note - the diagram does not show zombie state!

Figure 3.2 Diagram of process state.

True	False		
◉☑	○✖	Every forked process has to go through ZOMBIE state, at least for a small duration.	✔
◉✖	○☑	A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first	✖
○☑	◉✖	A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet	✖
○☑	◉✖	Only a process in READY state is considered by scheduler	✖
◉✖	○☑	A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.	✖

Every forked process has to go through ZOMBIE state, at least for a small duration.: True
A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first: False
A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet: True
Only a process in READY state is considered by scheduler: True
A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.: False

Question **8**

Correct

Mark 2.00 out of 2.00

Consider the reference string
6 4 2 0 1 2 6 9 2 0 5

If the number of page frames is 3, then total number of page faults (including initial), using FIFO replacement is:

Answer: 10 ✔

#6# 6,4# 6,4,2 #0,4,2# 0,1,2 #0,1,6 #9,1,6# 9,2,6# 9,2,0 #5,2,0

The correct answer is: 10

Question **9**

Incorrect

Mark 0.00 out of 1.00

Select all the correct statements about linking and loading.

Select one or more:

- [ ] a. Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently)
- [ ] b. Dynamic linking essentially results in relocatable code.
- [ ] c. Continuous memory management schemes can support static linking and static loading. (may be inefficiently)
- [ ] d. Loader is last stage of the linker program
- [ ] e. Dynamic linking and loading is not possible without demand paging or demand segmentation.
- [ ] f. Static linking leads to non-relocatable code
- [ ] g. Continuous memory management schemes can support dynamic linking and dynamic loading.
- [x] h. Dynamic linking is possible with continous memory management, but variable sized partitions only. ✖
- [x] i. Loader is part of the operating system ✔

Your answer is incorrect.

The correct answers are: Continuous memory management schemes can support static linking and static loading. (may be inefficiently), Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently), Dynamic linking essentially results in relocatable code., Loader is part of the operating system, Dynamic linking and loading is not possible without demand paging or demand segmentation.

Question **10**

Correct

Mark 1.00 out of 1.00

Consider a computer system with a 32-bit logical address and 4- KB page size. The system supports up to 512 MB of physical memory. How many entries are there in each of the following?

Write answer as a decimal number.

A conventional, single-level page table:

1048576

✔

An inverted page table:

131072

✔

Question **11**

Incorrect

Mark 0.00 out of 1.00

W.r.t. xv6 code, match the state of a process with a code that sets the state

RUNNING	Choose...
ZOMBIE	Choose...
EMBRYO	Choose...
SLEEPING	Choose...
UNUSED	Choose...
RUNNABLE	scheduler() ✖

The correct answer is: RUNNING → scheduler(), ZOMBIE → exit(), called by process itself, EMBRYO → fork()->allocproc() before setting up the UVM, SLEEPING → sleep(), called by any process blocking itself, UNUSED → wait(), called by parent process, RUNNABLE → wakeup(), called by an interrupt handler

Question **12**

Not answered

Marked out of 1.00

Select the correct statements about interrupt handling in xv6 code

- a. The trapframe pointer in struct proc, points to a location on user stack
- b. The CS and EIP are changed only immediately on a hardware interrupt
- c. The CS and EIP are changed only after pushing user code's SS,ESP on stack
- d. The trapframe pointer in struct proc, points to a location on kernel stack
- e. On any interrupt/syscall/exception the control first jumps in vectors.S
- f. The function trap() is the called irrespective of hardware interrupt/system-call/exception
- g. All the 256 entries in the IDT are filled
- h. The function trap() is the called only in case of hardware interrupt
- i. xv6 uses the 64th entry in IDT for system calls
- j. xv6 uses the 0x64th entry in IDT for system calls
- k. Before going to alltraps, the kernel stack contains upto 5 entries.
- l. Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt
- m. On any interrupt/syscall/exception the control first jumps in trapasm.S

Your answer is incorrect.

The correct answers are: All the 256 entries in the IDT are filled, Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt, xv6 uses the 64th entry in IDT for system calls, On any interrupt/syscall/exception the control first jumps in vectors.S, Before going to alltraps, the kernel stack contains upto 5 entries., The trapframe pointer in struct proc, points to a location on kernel stack, The function trap() is the called irrespective of hardware interrupt/system-call/exception, The CS and EIP are changed only after pushing user code's SS,ESP on stack

Question **13**

Correct

Mark 1.00 out of 1.00

The complete range of virtual addresses (after main() in main.c is over), from which the free pages used by kalloc() and kfree() is derived,are:

- a. end, (4MB + PHYSTOP)
- b. P2V(end), PHYSTOP
- c. end, P2V(4MB + PHYSTOP)
- d. end, PHYSTOP
- e. end, 4MB
- f. end, P2V(PHYSTOP) ✔
- g. P2V(end), P2V(PHYSTOP)

Your answer is correct.

The correct answer is: end, P2V(PHYSTOP)

Question **14**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about MMU and it's functionality (on a non-demand paged system)

Select one or more:

- [x] a. Illegal memory access is detected in hardware by MMU and a trap is raised ✔
- [ ] b. Illegal memory access is detected by operating system
- [ ] c. MMU is a separate chip outside the processor
- [ ] d. The operating system interacts with MMU for every single address translation
- [x] e. Logical to physical address translations in MMU are done in hardware, automatically ✔
- [ ] f. Logical to physical address translations in MMU are done with specific machine instructions
- [x] g. MMU is inside the processor ✔
- [x] h. The Operating system sets up relevant CPU registers to enable proper MMU translations ✔

Your answer is correct.

The correct answers are: MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

Question **15**

Incorrect

Mark 0.00 out of 2.00

Order the following events, in the creation of init() process in xv6:

1. ✖ initcode is selected by scheduler for execution
2. ✖ kernel stack is allocated for initcode process
3. ✖ values are set in the trapframe of initcode
4. ✖ sys_exec runs
5. ✖ initcode process is set to be runnable
6. ✖ code is set to start in forkret() when process gets scheduled
7. ✖ Arguments on setup on process stack for /init
8. ✖ trapframe and context pointers are set to proper location
9. ✖ trap() runs
10. ✖ userinit() is called
11. ✖ the header of "/init" ELF file is ready by kernel
12. ✖ empty struct proc is obtained for initcode
13. ✖ Stack is allocated for "/init" process
14. ✖ function pointer from syscalls[] array is invoked
15. ✖ memory mappings are created for "/init" process
16. ✖ page table mappings of 'initcode' are replaced by makpings of 'init'
17. ✖ initcode calls exec system call
18. ✖ initcode process runs
19. ✖ name of process "/init" is copied in struct proc

Your answer is incorrect.

Grading type: Relative to the next item (including last)

Grade details: 0 / 19 = 0%

Here are the scores for each item in this response:

1. 0 / 1 = 0%
2. 0 / 1 = 0%
3. 0 / 1 = 0%
4. 0 / 1 = 0%
5. 0 / 1 = 0%
6. 0 / 1 = 0%
7. 0 / 1 = 0%
8. 0 / 1 = 0%
9. 0 / 1 = 0%
10. 0 / 1 = 0%
11. 0 / 1 = 0%
12. 0 / 1 = 0%
13. 0 / 1 = 0%
14. 0 / 1 = 0%
15. 0 / 1 = 0%

16. 0 / 1 = 0%
17. 0 / 1 = 0%
18. 0 / 1 = 0%
19. 0 / 1 = 0%

The correct order for these items is as follows:

1. userinit() is called
2. empty struct proc is obtained for initcode
3. kernel stack is allocated for initcode process
4. trapframe and context pointers are set to proper location
5. code is set to start in forkret() when process gets scheduled
6. kernel memory mappings are created for initcode
7. values are set in the trapframe of initcode
8. initcode process is set to be runnable
9. initcode is selected by scheduler for execution
10. initcode process runs
11. initcode calls exec system call
12. trap() runs
13. function pointer from syscalls[] array is invoked
14. sys_exec runs
15. the header of "/init" ELF file is ready by kernel
16. memory mappings are created for "/init" process
17. Stack is allocated for "/init" process
18. Arguments on setup on process stack for /init
19. name of process "/init" is copied in struct proc
20. page table mappings of 'initcode' are replaced by makpings of 'init'

Question **16**

Partially correct

Mark 0.56 out of 1.00

Mark the statements as True or False, w.r.t. mmap()

True	False		
		mmap() can be implemented on both demand paged and non-demand paged systems.	✖
		MAP_FIXED guarantees that the mapping is always done at the specified address	✖
		MAP_PRIVATE leads to a mapping that is copy-on-write	✔
		on failure mmap() returns (void *)-1	✔
		MAP_SHARED leads to a mapping that is copy-on-write	✔
		mmap() results in changes to buffer-cache of the kernel.	✖
		on failure mmap() returns NULL	✔
		mmap() results in changes to page table of a process.	✖
		mmap() is a system call	✔

mmap() can be implemented on both demand paged and non-demand paged systems.: True
MAP_FIXED guarantees that the mapping is always done at the specified address: False
MAP_PRIVATE leads to a mapping that is copy-on-write: True
on failure mmap() returns (void *)-1: True
MAP_SHARED leads to a mapping that is copy-on-write: False
mmap() results in changes to buffer-cache of the kernel.: False
on failure mmap() returns NULL: False
mmap() results in changes to page table of a process.: True
mmap() is a system call: True

Question **17**

Incorrect

Mark 0.00 out of 1.00

If one thread opens a file with read privileges then

Select one:

- a. other threads in the same process can also read from that file
- b. none of these
- c. any other thread cannot read from that file
- d. other threads in the another process can also read from that file ✖

Your answer is incorrect.

The correct answer is: other threads in the same process can also read from that file

Question **18**

Partially correct

Mark 0.60 out of 1.00

Mark the statements about named and un-named pipes as True or False

True	False		
◉✓	○✗	Named pipe exists as a file	✔
◉✓	○✗	Un-named pipes are inherited by a child process from parent.	✔
◉✗	○✓	The buffers for named-pipe are in process-memory while the buffers for the un-named pipe are in kernel memory.	✖
◉✓	○✗	Both types of pipes are an extension of the idea of "message passing".	✔
◉✗	○✓	A named pipe has a name decided by the kernel.	✖
○✓	◉✗	Un-named pipes can be used for communication between only "related" processes, if the common ancestor created it.	✖
◉✓	○✗	Both types of pipes provide FIFO communication.	✔
○✗	◉✓	Named pipes can be used for communication between only "related" processes.	✔
◉✓	○✗	Named pipes can exist beyond the life-time of processes using them.	✔
◉✗	○✓	The pipe() system call can be used to create either a named or un-named pipe.	✖

Named pipe exists as a file: True
Un-named pipes are inherited by a child process from parent.: True
The buffers for named-pipe are in process-memory while the buffers for the un-named pipe are in kernel memory.: False
Both types of pipes are an extension of the idea of "message passing".: True
A named pipe has a name decided by the kernel.: False
Un-named pipes can be used for communication between only "related" processes, if the common ancestor created it.: True
Both types of pipes provide FIFO communication.: True
Named pipes can be used for communication between only "related" processes.: False
Named pipes can exist beyond the life-time of processes using them.: True
The pipe() system call can be used to create either a named or un-named pipe.: False

Question **19**

Partially correct

Mark 0.67 out of 1.00

Select the most common causes of use of IPC by processes

- [ ] a. More modular code
- [x] b. Breaking up a large task into small tasks and speeding up computation, on multiple core machines ✔
- [ ] c. More security checks
- [x] d. Sharing of information of common interest ✔
- [ ] e. Get the kernel performance statistics

The correct answers are: Sharing of information of common interest, Breaking up a large task into small tasks and speeding up computation, on multiple core machines, More modular code

Question **20**

Correct

Mark 1.00 out of 1.00

For each function/code-point, select the status of segmentation setup in xv6

after seginit() in main()	gdt setup with 5 entries (0 to 4) on one processor	✔
bootmain()	gdt setup with 3 entries, at start32 symbol of bootasm.S	✔
after startothers() in main()	gdt setup with 5 entries (0 to 4) on all processors	✔
entry.S	gdt setup with 3 entries, at start32 symbol of bootasm.S	✔
kvmalloc() in main()	gdt setup with 3 entries, at start32 symbol of bootasm.S	✔
bootasm.S	gdt setup with 3 entries, at start32 symbol of bootasm.S	✔

Your answer is correct.

The correct answer is: after seginit() in main() → gdt setup with 5 entries (0 to 4) on one processor, bootmain() → gdt setup with 3 entries, at start32 symbol of bootasm.S, after startothers() in main() → gdt setup with 5 entries (0 to 4) on all processors, entry.S → gdt setup with 3 entries, at start32 symbol of bootasm.S, kvmalloc() in main() → gdt setup with 3 entries, at start32 symbol of bootasm.S, bootasm.S → gdt setup with 3 entries, at start32 symbol of bootasm.S

Question **21**

Partially correct

Mark 0.50 out of 1.00

Mark whether the given sequence of events is possible or not-possible. Also, select the reason for your answer.

For each sequence it's a not-possible sequence if some important event is not mentioned in the sequence.

Assume that the kernel code is non-interruptible and uniprocessor system.

```
Process P1, user code executing
Timer interrupt
Context changes to kernel context
Generic interrupt handler runs
Generic interrupt handler calls Scheduler
Scheduler selects P2 for execution
After scheduler, Process P2 user code executing
```

This sequence of events is: not-possible ✔

Because

Generic interrupt handler can not call scheduler ✖

Question **22**

Partially correct

Mark 0.63 out of 1.00

Mark the statements as True or False, w.r.t. passing of arguments to system calls in xv6 code.

True	False		
✖	☑	Integer arguments are stored in eax, ebx, ecx, etc. registers	✖
☑	✖	String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer	✔
☑	✖	The functions like argint(), argstr() make the system call arguments available in the kernel.	✔
✖	☑	String arguments are first copied to trapframe and then from trapframe to kernel's other variables.	✖
☑	✖	The arguments to system call originally reside on process stack.	✔
✖	☑	The arguments to system call are copied to kernel stack in trapasm.S	✖
☑	✖	Integer arguments are copied from user memory to kernel memory using argint()	✔
☑	✖	The arguments are accessed in the kernel code using esp on the trapframe.	✔

Integer arguments are stored in eax, ebx, ecx, etc. registers: False
String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer: True
The functions like argint(), argstr() make the system call arguments available in the kernel.: True
String arguments are first copied to trapframe and then from trapframe to kernel's other variables.: False
The arguments to system call originally reside on process stack.: True
The arguments to system call are copied to kernel stack in trapasm.S: False
Integer arguments are copied from user memory to kernel memory using argint(): True
The arguments are accessed in the kernel code using esp on the trapframe.: True

Question **23**

Not answered

Marked out of 1.00

Given below is a sequence of reference bits on pages before the second chance algorithm runs. Before the algorithm runs, the counter is at the page marked (x). Write the sequence of reference bits after the second chance algorithm has executed once. In the answer write PRECISELY one space BETWEEN each number and do not mention (x).

0 0 1(x) 1 0 1 1

Answer:

The correct answer is: 0 0 0 0 0 1 1

Question **24**

Correct

Mark 2.00 out of 2.00

For the reference string
3 4 3 5 2

using FIFO replacement policy for pages,

consider the number of page faults for 2, 3 and 4 page frames.
Select the correct statement.

Select one:

- a. Exhibit Balady's anomaly between 3 and 4 frames
- b. Do not exhibit Balady's anomaly
- c. Exhibit Balady's anomaly between 2 and 3 frames

Your answer is correct.

The correct answer is: Do not exhibit Balady's anomaly

Question **25**

Correct

Mark 1.00 out of 1.00

For the reference string
3 4 3 5 2

using LRU replacement policy for pages,

consider the number of page faults for 2, 3 and 4 page frames.
Select the most correct statement.

Select one:

- a. LRU will never exhibit Balady's anomaly ✔
- b. Exhibit Balady's anomaly between 2 and 3 frames
- c. This example does not exhibit Balady's anomaly
- d. Exhibit Balady's anomaly between 3 and 4 frames

Your answer is correct.

The correct answer is: LRU will never exhibit Balady's anomaly

Question **26**

Partially correct

Mark 0.55 out of 1.00

Select all the correct statements about process states.

Note that in this question you lose marks for every incorrect choice that you make, proportional to actual number of incorrect choices.

- [ ] a. Process state is implemented as a string
- [x] b. Process state is stored in the PCB ✔
- [ ] c. A process becomes ZOMBIE when another process bites into it's memory
- [x] d. Process state is stored in the processor ✖
- [ ] e. The scheduler can change state of a process from RUNNALBE to RUNNING and vice-versa
- [x] f. The scheduler can change state of a process from RUNNALBE to RUNNING ✔
- [x] g. A process becomes ZOMBIE when it calls exit() ✔
- [ ] h. Process state is changed only by interrupt handlers
- [ ] i. Process state can be implemented as just a number

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: Process state is stored in the PCB, Process state can be implemented as just a number, The scheduler can change state of a process from RUNNALBE to RUNNING, A process becomes ZOMBIE when it calls exit()

Question **27**

Partially correct

Mark 0.38 out of 1.00

Consider a demand-paging system with the following time-measured utilizations:
CPU utilization : 20%
Paging disk: 97.7%
Other I/O devices: 5%

For each of the following, indicate whether it will (or is likely to) improve CPU utilization (even if by a small amount). Explain your answers.

a. Install a faster CPU : Yes ✖

b. Install a bigger paging disk. : Yes ✖

c. Increase the degree of multiprogramming. : Yes ✖

d. Decrease the degree of multiprogramming. : Yes ✔

e. Install more main memory.: Yes ✔

f. Install a faster hard disk or multiple controllers with multiple hard disks. : Yes ✔

g. Add prepaging to the page-fetch algorithms. :
May be ✖

h. Increase the page size. : May be ✖

Question **28**

Incorrect

Mark 0.00 out of 1.00

Suppose a kernel uses a buddy allocator. The smallest chunk that can be allocated is of size 32 bytes. One bit is used to track each such chunk, where 1 means allocated and 0 means free. The chunk looks like this as of now:

10011010

Now, there is a request for a chunk of 50 bytes.

After this allocation, the bitmap, indicating the status of the buddy allocator will be

Answer: 10110010 ✖

The correct answer is: 11111010

Question **29**

Partially correct

Mark 0.75 out of 1.00

Select the correct points of comparison between POSIX and System V shared memory.

- [x] a. POSIX shared memory is newer than System V shared memory ✔
- [x] b. POSIX shared memory is "thread safe", System V is not ✔
- [x] c. System V is more prevalent than POSIX even today ✔
- [ ] d. POSIX allows giving name to shared memory, System V does not

The correct answers are: POSIX shared memory is newer than System V shared memory, POSIX shared memory is "thread safe", System V is not, POSIX allows giving name to shared memory, System V does not, System V is more prevalent than POSIX even today

Question **30**

Partially correct

Mark 0.67 out of 1.00

Figure 9.8 Paging hardware.

Mark the statements as True or False, w.r.t. the above diagram (note that the diagram does not cover all details of what actually happens!)

True	False		
◉☑	○✖	The combining of f and d is done by MMU	✔
◉✖	○☑	There are total 3 memory references in this diagram	✖
◉☑	○✖	The split of logical address into p and d is done by MMU	✔
◉☑	○✖	The page table is in physical memory and must be continuous	✔
◉✖	○☑	Using the offset d in the physical page-frame is done by MMU	✖
◉☑	○✖	The logical address issued by CPU is the same one generated by compiler	✔

The combining of f and d is done by MMU: True
There are total 3 memory references in this diagram: False
The split of logical address into p and d is done by MMU: True
The page table is in physical memory and must be continuous: True
Using the offset d in the physical page-frame is done by MMU: False
The logical address issued by CPU is the same one generated by compiler: True

Question **31**

Partially correct

Mark 0.50 out of 1.00

Select all the correct statements about signals

Select one or more:

- [ ] a. SIGKILL definitely kills a process because it's code runs in kernel mode of CPU
- [x] b. Signals are delivered to a process by another process ✗
- [ ] c. The signal handler code runs in kernel mode of CPU
- [x] d. SIGKILL definitely kills a process because it can't be caught or ignored, and it's default action terminates the process ✓
- [x] e. The signal handler code runs in user mode of CPU ✓
- [x] f. A signal handler can be invoked asynchronously or synchronously depending on signal type ✓
- [ ] g. Signal handlers once replaced can't be restored
- [ ] h. Signals are delivered to a process by kernel

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: Signals are delivered to a process by kernel, A signal handler can be invoked asynchronously or synchronously depending on signal type, The signal handler code runs in user mode of CPU, SIGKILL definitely kills a process because it can't be caught or ignored, and it's default action terminates the process

Question **32**

Correct

Mark 1.00 out of 1.00

The data structure used in kalloc() and kfree() in xv6 is

- ( ) a. Singly linked circular list
- (•) b. Singly linked NULL terminated list ✓
- ( ) c. Double linked NULL terminated list
- ( ) d. Doubly linked circular list

Your answer is correct.

The correct answer is: Singly linked NULL terminated list

Question **33**

Partially correct

Mark 1.78 out of 2.00

Match the description of a memory management function with the name of the function that provides it, in xv6

Load contents from ELF into existing pages	loaduvm()	✔
Mark the page as in-accessible	clearpteu()	✔
setup the kernel part in the page table	setupkvm()	✔
Switch to kernel page table	switchkvm()	✔
Create a copy of the page table of a process	copyuvm()	✔
Copy the code pages of a process	No such function	✔
Setup and load the user page table for initcode process	inituvm()	✔
Switch to user page table	switchuvm()	✔
Load contents from ELF into pages after allocating the pages first	inituvm()	✖

The correct answer is: Load contents from ELF into existing pages → loaduvm(), Mark the page as in-accessible → clearpteu(), setup the kernel part in the page table → setupkvm(), Switch to kernel page table → switchkvm(), Create a copy of the page table of a process → copyuvm(), Copy the code pages of a process → No such function, Setup and load the user page table for initcode process → inituvm(), Switch to user page table → switchuvm(), Load contents from ELF into pages after allocating the pages first → No such function

Question **34**

Partially correct

Mark 0.60 out of 1.00

Mark the statements as True or False, w.r.t. thrashing

True	False		
✖	✔	Thrashing occurs because some process is doing lot of disk I/O.	✖
✔	✖	Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.	✔
✖	✔	mmap() solves the problem of thrashing.	✔
✔	✖	The working set model is an attempt at approximating the locality of a process.	✔
✔	✖	Thrashing is particular to demand paging systems, and does not apply to pure paging systems.	✔
✖	✔	Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.	✖
✖	✔	Thrashing can occur even if entire memory is not in use.	✖
✔	✖	During thrashing the CPU is under-utilised as most time is spent in I/O	✖
✔	✖	Thrashing can be limited if local replacement is used.	✔
✔	✖	Thrashing occurs when the total size of all processe's locality exceeds total memory size.	✔

Thrashing occurs because some process is doing lot of disk I/O.: False
Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.: True
mmap() solves the problem of thrashing.: False
The working set model is an attempt at approximating the locality of a process.: True
Thrashing is particular to demand paging systems, and does not apply to pure paging systems.: True
Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.: False
Thrashing can occur even if entire memory is not in use.: False
During thrashing the CPU is under-utilised as most time is spent in I/O: True
Thrashing can be limited if local replacement is used.: True
Thrashing occurs when the total size of all processe's locality exceeds total memory size.: True

Question **35**

Correct

Mark 1.00 out of 1.00

After virtual memory is implemented

(select T/F for each of the following)One Program's size can be larger than physical memory size

True	False		
◉	○	Cumulative size of all programs can be larger than physical memory size	✔
◉	○	Code need not be completely in memory	✔
◉	○	One Program's size can be larger than physical memory size	✔
○	◉	Virtual addresses become available to executing process	✔
○	◉	Virtual access to memory is granted to all processes	✔
◉	○	Relatively less I/O may be possible during process execution	✔
◉	○	Logical address space could be larger than physical address space	✔

Cumulative size of all programs can be larger than physical memory size: True
Code need not be completely in memory: True
One Program's size can be larger than physical memory size: True
Virtual addresses become available to executing process: False
Virtual access to memory is granted to all processes: False
Relatively less I/O may be possible during process execution: True
Logical address space could be larger than physical address space: True

◄ (Optional Assignment) lseek system call in xv6

Jump to...

Feedback on Quiz-2 ►

Started on	Thursday, 18 March 2021, 2:46 PM
State	Finished
Completed on	Thursday, 18 March 2021, 3:50 PM
Time taken	1 hour 4 mins
Grade	**10.36** out of 20.00 (**52**%)

Question **1**

Partially correct

Mark 0.57 out of 1.00

Mark True, the actions done as part of code of swtch() in swtch.S, in xv6

True	False		
		Restore new callee saved registers from kernel stack of new context	✔
		Save old callee saved registers on kernel stack of old context	✔
		Save old callee saved registers on user stack of old context	✔
		Switch from old process context to new process context	✖
		Switch from one stack (old) to another(new)	✖
		Restore new callee saved registers from user stack of new context	✔
		Jump to code in new context	✖

Restore new callee saved registers from kernel stack of new context: True
Save old callee saved registers on kernel stack of old context: True
Save old callee saved registers on user stack of old context: False
Switch from old process context to new process context: False
Switch from one stack (old) to another(new): True
Restore new callee saved registers from user stack of new context: False
Jump to code in new context: False

Question **2**

Partially correct

Mark 0.17 out of 0.50

For each function/code-point, select the status of segmentation setup in xv6

bootmain()	gdt setup with 3 entries, right from first line of code of bootloader	✖
kvmalloc() in main()	gdt setup with 5 entries (0 to 4) on one processor	✖
after startothers() in main()	gdt setup with 5 entries (0 to 4) on all processors	✔
after seginit() in main()	gdt setup with 5 entries (0 to 4) on all processors	✖
bootasm.S	gdt setup with 3 entries, right from first line of code of bootloader	✖
entry.S	gdt setup with 3 entries, at start32 symbol of bootasm.S	✔

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: bootmain() → gdt setup with 3 entries, at start32 symbol of bootasm.S, kvmalloc() in main() → gdt setup with 3 entries, at start32 symbol of bootasm.S, after startothers() in main() → gdt setup with 5 entries (0 to 4) on all processors, after seginit() in main() → gdt setup with 5 entries (0 to 4) on one processor, bootasm.S → gdt setup with 3 entries, at start32 symbol of bootasm.S, entry.S → gdt setup with 3 entries, at start32 symbol of bootasm.S

Question **3**

Partially correct

Mark 0.38 out of 1.00

Compare paging with demand paging and select the correct statements.

Select one or more:

- [x] a. The meaning of valid-invalid bit in page table is different in paging and demand-paging. ✔
- [x] b. Demand paging requires additional hardware support, compared to paging. ✔
- [ ] c. Paging requires some hardware support in CPU
- [x] d. With paging, it's possible to have user programs bigger than physical memory. ✖
- [x] e. Both demand paging and paging support shared memory pages. ✔
- [ ] f. Demand paging always increases effective memory access time.
- [x] g. With demand paging, it's possible to have user programs bigger than physical memory. ✔
- [x] h. Calculations of number of bits for page number and offset are same in paging and demand paging. ✔
- [ ] i. TLB hit ration has zero impact in effective memory access time in demand paging.
- [ ] j. Paging requires NO hardware support in CPU

Your answer is partially correct.

You have correctly selected 5.

The correct answers are: Demand paging requires additional hardware support, compared to paging., Both demand paging and paging support shared memory pages., With demand paging, it's possible to have user programs bigger than physical memory., Demand paging always increases effective memory access time., Paging requires some hardware support in CPU, Calculations of number of bits for page number and offset are same in paging and demand paging., The meaning of valid-invalid bit in page table is different in paging and demand-paging.

Question **4**

Partially correct

Mark 0.44 out of 0.50

Suppose a processor supports base(relocation register) + limit scheme of MMU.

Assuming this, mark the statements as True/False

True	False		
✔	✖	The OS may terminate the process while handling the interrupt of memory violation	✔
✔	✖	The hardware detects any memory access beyond the limit value and raises an interrupt	✔
✖	✔	The hardware may terminate the process while handling the interrupt of memory violation	✖
✔	✖	The OS sets up the relocation and limit registers when the process is scheduled	✔
✔	✖	The compiler generates machine code assuming continuous memory address space for process, and calculating appropriate sizes for code, and data;	✔
✖	✔	The process sets up it's own relocation and limit registers when the process is scheduled	✔
✖	✔	The OS detects any memory access beyond the limit value and raises an interrupt	✔
✖	✔	The compiler generates machine code assuming appropriately sized semgments for code, data and stack.	✔

The OS may terminate the process while handling the interrupt of memory violation: True
The hardware detects any memory access beyond the limit value and raises an interrupt: True
The hardware may terminate the process while handling the interrupt of memory violation: False
The OS sets up the relocation and limit registers when the process is scheduled: True
The compiler generates machine code assuming continuous memory address space for process, and calculating appropriate sizes for code, and data;: True
The process sets up it's own relocation and limit registers when the process is scheduled: False
The OS detects any memory access beyond the limit value and raises an interrupt: False
The compiler generates machine code assuming appropriately sized semgments for code, data and stack.: False

Question **5**

Correct

Mark 0.50 out of 0.50

Consider the following list of free chunks, in continuous memory management:

10k, 25k, 12k, 7k, 9k, 13k

Suppose there is a request for chunk of size 9k, then the free chunk selected under each of the following schemes will be

Best fit:

9k

✔

First fit:

10k

✔

Worst fit:

25k

✔

Question **6**

Partially correct

Mark 0.50 out of 1.00

Select all the correct statements about MMU and it's functionality

Select one or more:

- [ ] a. MMU is a separate chip outside the processor
- [x] b. MMU is inside the processor ✔
- [ ] c. Logical to physical address translations in MMU are done with specific machine instructions
- [x] d. The operating system interacts with MMU for every single address translation ✖
- [x] e. Illegal memory access is detected in hardware by MMU and a trap is raised ✔
- [ ] f. The Operating system sets up relevant CPU registers to enable proper MMU translations
- [x] g. Logical to physical address translations in MMU are done in hardware, automatically ✔
- [ ] h. Illegal memory access is detected by operating system

Your answer is partially correct.

You have correctly selected 3.
The correct answers are: MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

Question **7**

Incorrect

Mark 0.00 out of 0.50

Assuming a 8- KB page size, what is the page numbers for the address 874815 reference in decimal :

(give answer also in decimal)

Answer: 2186 ✖

The correct answer is: 107

Question **8**

Incorrect

Mark 0.00 out of 0.25

Select the compiler's view of the process's address space, for each of the following MMU schemes:
(Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

Segmentation, then paging	Many continuous chunks each of page size	✖
Relocation + Limit	Many continuous chunks of same size	✖
Segmentation	one continuous chunk	✖
Paging	many continuous chunks of variable size	✖

Your answer is incorrect.

The correct answer is: Segmentation, then paging → many continuous chunks of variable size, Relocation + Limit → one continuous chunk, Segmentation → many continuous chunks of variable size, Paging → one continuous chunk

Question **9**

Incorrect

Mark 0.00 out of 0.50

Suppose the memory access time is 180ns and TLB hit ratio is 0.3, then effective memory access time is (in nanoseconds);

Answer: 192 ✖

The correct answer is: 306.00

Question **10**

Correct

Mark 0.50 out of 0.50

In xv6, The struct context is given as

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

Select all the reasons that explain why only these 5 registers are included in the struct context.

- [x] a. The segment registers are same across all contexts, hence they need not be saved ✔
- [x] b. esp is not saved in context, because context{} is on stack and it's address is always argument to swtch() ✔
- [ ] c. xv6 tries to minimize the size of context to save memory space
- [ ] d. esp is not saved in context, because it's not part of the context
- [x] e. eax, ecx, edx are caller save, hence no need to save ✔

Your answer is correct.

The correct answers are: The segment registers are same across all contexts, hence they need not be saved, eax, ecx, edx are caller save, hence no need to save, esp is not saved in context, because context{} is on stack and it's address is always argument to swtch()

Question **11**

Partially correct

Mark 0.83 out of 1.50

Arrange the following events in order, in page fault handling:

Disk interrupt wakes up the process	7	✔
The reference bit is found to be invalid by MMU	1	✔
OS makes available an empty frame	6	✖
Restart the instruction that caused the page fault	9	✔
A hardware interrupt is issued	3	✖
OS schedules a disk read for the page (from backing store)	5	✔
Process is kept in wait state	4	✖
Page tables are updated for the process	8	✔
Operating system decides that the page was not in memory	2	✖

Your answer is partially correct.

You have correctly selected 5.

The correct answer is: Disk interrupt wakes up the process → 7, The reference bit is found to be invalid by MMU → 1, OS makes available an empty frame → 4, Restart the instruction that caused the page fault → 9, A hardware interrupt is issued → 2, OS schedules a disk read for the page (from backing store) → 5, Process is kept in wait state → 6, Page tables are updated for the process → 8, Operating system decides that the page was not in memory → 3

Question **12**

Incorrect

Mark 0.00 out of 0.50

Suppose a kernel uses a buddy allocator. The smallest chunk that can be allocated is of size 32 bytes. One bit is used to track each such chunk, where 1 means allocated and 0 means free. The chunk looks like this as of now:

00001010

Now, there is a request for a chunk of 70 bytes.

After this allocation, the bitmap, indicating the status of the buddy allocator will be

Answer: 11101010 ✖

The correct answer is: 11111010

Question **13**

Incorrect

Mark 0.00 out of 0.25

The complete range of virtual addresses (after main() in main.c is over), from which the free pages used by kalloc() and kfree() is derived,are:

- a. end, 4MB
- b. P2V(end), P2V(PHYSTOP)
- c. end, P2V(4MB + PHYSTOP)
- d. P2V(end), PHYSTOP ✖
- e. end, (4MB + PHYSTOP)
- f. end, PHYSTOP
- g. end, P2V(PHYSTOP)

Your answer is incorrect.

The correct answer is: end, P2V(PHYSTOP)

Question **14**

Partially correct

Mark 0.33 out of 0.50

Match the pair

Hashed page table	Linear search on collsion done by OS (e.g. SPARC Solaris) typically	✔
Inverted Page table	Linear/Parallel search using frame number in page table	✖
Hierarchical Paging	More memory access time per hierarchy	✔

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: Hashed page table → Linear search on collsion done by OS (e.g. SPARC Solaris) typically, Inverted Page table → Linear/Parallel search using page number in page table, Hierarchical Paging → More memory access time per hierarchy

Question **15**

Partially correct

Mark 0.29 out of 0.50

After virtual memory is implemented

(select T/F for each of the following)One Program's size can be larger than physical memory size

True	False		
◉☑	○✖	Code need not be completely in memory	✔
◉☑	○✖	Cumulative size of all programs can be larger than physical memory size	✔
◉✖	○☑	Virtual access to memory is granted	✖
◉☑	○✖	Logical address space could be larger than physical address space	✔
◉✖	○☑	Virtual addresses are available	✖
○☑	◉✖	Relatively less I/O may be possible during process execution	✖
◉☑	○✖	One Program's size can be larger than physical memory size	✔

Code need not be completely in memory: True
Cumulative size of all programs can be larger than physical memory size: True
Virtual access to memory is granted: False
Logical address space could be larger than physical address space: True
Virtual addresses are available: False
Relatively less I/O may be possible during process execution: True
One Program's size can be larger than physical memory size: True

Question **16**

Partially correct

Mark 0.64 out of 1.00

W.r.t. Memory management in xv6,

xv6 uses physical memory upto 224 MB onlyMark statements True or False

True	False		
		The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context	✔
		The stack allocated in entry.S is used as stack for scheduler's context for first processor	✔
		The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir	✔
		The free page-frame are created out of nearly 222 MB	✖
		The kernel code and data take up less than 2 MB space	✔
		The switchkvm() call in scheduler() changes CR3 to use page directory of new process	✖
		The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context	✔
		PHYSTOP can be increased to some extent, simply by editing memlayout.h	✔
		xv6 uses physical memory upto 224 MB only	✖
		The process's address space gets mapped on frames, obtained from ~2MB:224MB range	✔
		The kernel's page table given by kpgdir variable is used as stack for scheduler's context	✖

The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context: True
The stack allocated in entry.S is used as stack for scheduler's context for first processor: True
The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir: True
The free page-frame are created out of nearly 222 MB: True
The kernel code and data take up less than 2 MB space: True
The switchkvm() call in scheduler() changes CR3 to use page directory of new process: False
The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context: False
PHYSTOP can be increased to some extent, simply by editing memlayout.h: True
xv6 uses physical memory upto 224 MB only: True
The process's address space gets mapped on frames, obtained from ~2MB:224MB range: True
The kernel's page table given by kpgdir variable is used as stack for scheduler's context: False

Question **17**

Incorrect

Mark 0.00 out of 1.50

Consider the reference string
6 4 2 0 1 2 6 9 2 0 5

If the number of page frames is 3, then total number of page faults (including initial), using LRU replacement is:

Answer: 8 ✖

#6# 6,4# 6,4,2 # 0,4,2#0,1,2#6,1,2#6,9,2#0,9,2#0,5,2

The correct answer is: 9

Question **18**

Partially correct

Mark 0.31 out of 0.50

Consider the image given below, which explains how paging works.

Figure 9.8 Paging hardware.

Mention whether each statement is True or False, with respect to this image.

True	False		
◉☑	○✖	The PTBR is present in the CPU as a register	✔
○✖	◉☑	The page table is indexed using frame number	✔
○☑	◉✖	The page table is indexed using page number	✖
◉✖	○☑	The locating of the page table using PTBR also involves paging translation	✖
○✖	◉☑	Size of page table is always determined by the size of RAM	✔
◉☑	○✖	The page table is itself present in Physical memory	✔
○☑	○✖	Maximum Size of page table is determined by number of bits used for page number	✖
◉☑	○✖	The physical address may not be of the same size (in bits) as the logical address	✔

The PTBR is present in the CPU as a register: True
The page table is indexed using frame number: False
The page table is indexed using page number: True
The locating of the page table using PTBR also involves paging translation: False
Size of page table is always determined by the size of RAM: False
The page table is itself present in Physical memory: True
Maximum Size of page table is determined by number of bits used for page number: True
The physical address may not be of the same size (in bits) as the logical address: True

Question **19**

Correct

Mark 2.00 out of 2.00

Given below is shared memory code with two processes sharing a memory segment.
The first process sends a user input string to second process. The second capitalizes the string. Then the first process prints the capitalized version.
Fill in the blanks to complete the code.

```
// First process
#define SHMSZ    27

int main()
{
   char c;
   int shmid;
   key_t key;
   char *shm, *s, string[128];
   key = 5679;
   if ((shmid =
```

shmget

✔ (key, SHMSZ, IPC_CREAT | 0666)) < 0) {

```
      perror("shmget");
      exit(1);
   }
   if ((shm =
```

shmat

✔ (shmid, NULL, 0)) == (char *) -1) {

```
      perror("shmat");
      exit(1);
   }
   s = shm;
   *s = '$';
   scanf("%s", string);
   strcpy(s + 1, string);
   *s = '
```

@

✔ '; //note the quotes

```
   while(*s != '
```

$

✔ ')

```
      sleep(1);
   printf("%s\n", s + 1);
   exit(0);
}
```

```
//Second process
#define SHMSZ    27
int main()
{
   int shmid;
   key_t key;
   char *shm, *s;
   int i;
   char string[128];
   key =
```

5679

```
    ;
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }
    s =
```

shm

```
    ;
    while(*s != '@')
        sleep(1);
    for(i = 0; i < strlen(s + 1); i++)
        s[i + 1] =  toupper(s[i + 1]);
    *s = '$';
    exit(0);
}
```

Question **20**

Partially correct

Mark 0.25 out of 0.50

Map the functionality/use with function/variable in xv6 code.

return a free page, if available; 0, otherwise	kinit1() ✗
Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed	mappages() ✓
Array listing the kernel memory mappings, to be used by setupkvm()	kmap[] ✓
Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices	kvmalloc() ✗
Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary	walkpgdir() ✓
Setup kernel part of a page table, and switch to that page table	setupkvm() ✗

Your answer is partially correct.

You have correctly selected 3.

The correct answer is: return a free page, if available; 0, otherwise → kalloc(), Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed → mappages(), Array listing the kernel memory mappings, to be used by setupkvm() → kmap[], Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices → setupkvm(), Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary → walkpgdir(), Setup kernel part of a page table, and switch to that page table → kvmalloc()

Question **21**

Partially correct

Mark 1.53 out of 2.50

Order events in xv6 timer interrupt code

(Transition from process P1 to P2's code.)

P2 is selected and marked RUNNING	12	✔
Change of stack from user stack to kernel stack of P1	3	✔
Timer interrupt occurs	2	✔
alltraps() will call iret	17	✖
change to context of P2, P2's kernel stack in use now	13	✔
P2's trap() will return to alltraps	16	✖
jump in vector.S	4	✔
P2 will return from sched() in yield()	14	✖
yield() is called	8	✔
trap() is called	7	✔
Process P2 is executing	18	✖
P1 is marked as RUNNABLE	9	✔
P2's yield() will return in trap()	15	✖
Process P1 is executing	1	✔
sched() is called,	11	✖
change to context of the scheduler, scheduler's stack in use now	10	✖
jump to alltraps	5	✔
Trapframe is built on kernel stack of P1	6	✔

Your answer is partially correct.

You have correctly selected 11.

The correct answer is: P2 is selected and marked RUNNING → 12, Change of stack from user stack to kernel stack of P1 → 3, Timer interrupt occurs → 2, alltraps() will call iret → 18, change to context of P2, P2's kernel stack in use now → 13, P2's trap() will return to alltraps → 17, jump in vector.S → 4, P2 will return from sched() in yield() → 15, yield() is called → 8, trap() is called → 7, Process P2 is executing → 14, P1 is marked as RUNNABLE → 9, P2's yield() will return in trap() → 16, Process P1 is executing → 1, sched() is called, → 10, change to context of the scheduler, scheduler's stack in use now → 11, jump to alltraps → 5, Trapframe is built on kernel stack of P1 → 6

Question **22**

Incorrect

Mark 0.00 out of 1.00

Given that the memory access time is 200 ns, probability of a page fault is 0.7 and page fault handling time is 8 ms,
The effective memory access time in nanoseconds is:

Answer: 192 ✖

The correct answer is: 5600060.00

Question **23**

Correct

Mark 0.25 out of 0.25

Select the state that is not possible after the given state, for a process:

New: Running ✔

Ready : Waiting ✔

Running: : None of these ✔

Waiting: Running ✔

Question **24**

Partially correct

Mark 0.63 out of 1.00

Select the correct statements about sched() and scheduler() in xv6 code

- [x] a. scheduler() switches to the selected process's context ✔
- [x] b. When either sched() or scheduler() is called, it does not return immediately to caller ✔
- [ ] c. After call to swtch() in sched(), the control moves to code in scheduler()
- [x] d. Each call to sched() or scheduler() involves change of one stack inside swtch() ✔
- [ ] e. After call to swtch() in scheduler(), the control moves to code in sched()
- [x] f. When either sched() or scheduler() is called, it results in a context switch ✔
- [x] g. sched() switches to the scheduler's context ✔
- [ ] h. sched() and scheduler() are co-routines

Your answer is partially correct.

You have correctly selected 5.
The correct answers are: sched() and scheduler() are co-routines, When either sched() or scheduler() is called, it does not return immediately to caller, When either sched() or scheduler() is called, it results in a context switch, sched() switches to the scheduler's context, scheduler() switches to the selected process's context, After call to swtch() in scheduler(), the control moves to code in sched(), After call to swtch() in sched(), the control moves to code in scheduler(), Each call to sched() or scheduler() involves change of one stack inside swtch()

Question **25**

Correct

Mark 0.25 out of 0.25

The data structure used in kalloc() and kfree() in xv6 is

- a. Doubly linked circular list
- b. Singly linked circular list
- c. Double linked NULL terminated list
- d. Singly linked NULL terminated list ✔

Your answer is correct.

The correct answer is: Singly linked NULL terminated list

◄ (Assignment) lseek system call in xv6

Jump to...

Started on	Saturday, 22 May 2021, 8:00 AM
State	Finished
Completed on	Saturday, 22 May 2021, 9:30 AM
Time taken	1 hour 30 mins
Grade	**26.12** out of 40.00 (**65**%)

Question **1**
Incorrect
Mark 0.00 out of 1.00

A 4 GB disk with 1 KB of block size would require these many number of **blocks** for it's free block bitmap:

Answer: 4096 ✖

The correct answer is: 512

Question **2**
Correct
Mark 1.00 out of 1.00

Given that the memory access time is 110 ns, probability of a page fault is 0.5 and page fault handling time is 12 ms,
The effective memory access time in nanoseconds is:

Answer: 6000165 ✔

The correct answer is: 6000055.00

Question **3**
Incorrect
Mark 0.00 out of 1.00

The maximum size of a file in number of blocks of BSIZE in xv6 code is
(write a number only)

Answer: 268 ✖

The correct answer is: 138

Question **4**
Incorrect
Mark 0.00 out of 1.00

Calculate the average waiting time using
Round Robin scheduling with time quantum of 5 time units
for the following workload
assuming that they arrive in the order written below.

Process	Burst Time
P1	5
P2	7
P3	6
P4	2

Write only a number in the answer upto two decimal points.

Answer: 40.75 ✖

The correct answer is: 10.25

Question **5**

Correct

Mark 1.00 out of 1.00

For the reference string

4 2 5 1 0 1 2 5 4 1 2

the number of page faults, including initial ones,

with FIFO replacement and 2 frames are :

Answer: 10 ✔

4 -

4 2

5 2

5 1

0 1

-

2 1

2 5

4 5

4 1

2 1

The correct answer is: 10

Question **6**

Correct

Mark 1.00 out of 1.00

Assuming a 16- KB page size, what is the page number for the address 428517 reference in decimal :

(give answer also in decimal)

Answer: 27 ✔

The correct answer is: 26

Question **7**

Correct

Mark 1.00 out of 1.00

In the code below assume that each function can be executed concurrently by many threads/processes.
Ignore syntactical issues, and focus on the semantics.

This program is an example of

```
spinlock a, b;  // assume initialized
thread1() {
    spinlock(b);
    //some code;
    spinlock(a);
    //some code;
    spinunlock(b);
    spinunlock(a);
}
thread2() {
    spinlock(a);
    //some code;
    spinlock(b);
    //some code;
    spinunlock(b);
    spinunlock(a);
}
```

- (selected) a. Deadlock ✔
- b. Self Deadlock
- c. None of these
- d. Deadlock or livelock depending on actual race
- e. Livelock

Your answer is correct.

The correct answer is: Deadlock

Question **8**

Partially correct

Mark 1.33 out of 2.00

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid.
"..." means some code.

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
```

Atomic compare and swap instruction (to be expanded inline into code) ✔

```
void
sleep(void *chan, struct spinlock *lk)
{
  ...
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }
```

If you don't do this, a process may be running on two processors parallely ✖

```
void
acquire(struct spinlock *lk)
{
...
__sync_synchronize();
```

Tell compiler not to reorder memory access beyond this line ✔

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
} → Atomic compare and swap instruction (to be expanded inline into code), void
sleep(void *chan, struct spinlock *lk)
{
  ...
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  } → Avoid a self-deadlock, void
acquire(struct spinlock *lk)
{
...
__sync_synchronize(); → Tell compiler not to reorder memory access beyond this line

Question **9**

Correct

Mark 1.00 out of 1.00

Predict the output of the program given here.

Assume that all the path names for the programs are correct. For example "/usr/bin/echo" will actually run echo command.

Assume that there is no mixing of printf output on screen if two of them run concurrently.

In the answer replace a new line by a single space.

For example::

`good`

`output`

should be written as `good output`

```
--
main() {
    int i;
    i = fork();
    if(i == 0)
        execl("/usr/bin/echo", "/usr/bin/echo", "hi", 0);
    else
        wait(0);
    fork();
    execl("/usr/bin/echo", "/usr/bin/echo", "one", 0);
}
```

Answer: hi one one ✔

The correct answer is: hi one one

Question **10**

Partially correct

Mark 1.67 out of 2.00

Select all the blocks that may need to be written back to disk (if updated, of-course), as "Yes", when an operation of deleting a file is carried out on ext2 file system.

An option has to be correct entirely to be marked "Yes"

Superblock	Yes	✔
One or multiple data blocks of the parent directory	No	✔
One or more data bitmap blocks for the parent directory	No	✔
Block bitmap(s) for all the blocks of the file	No	✖
Possibly one block bitmap corresponding to the parent directory	Yes	✔
Data blocks of the file	No	✔

Your answer is partially correct.

only one data block of parent directory. multiple blocks not possible. an entry is always contained within one single block

You have correctly selected 5.

The correct answer is: Superblock → Yes, One or multiple data blocks of the parent directory → No, One or more data bitmap blocks for the parent directory → No, Block bitmap(s) for all the blocks of the file → Yes, Possibly one block bitmap corresponding to the parent directory → Yes, Data blocks of the file → No

Question **11**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about bootloader.

Every wrong selection will deduct marks proportional to 1/n where n is total wrong choices in the question.

You will get minimum a zero.

- [x] a. Modern Bootloaders often allow configuring the way an OS boots ✔
- [x] b. Bootloaders allow selection of OS to boot from ✔
- [ ] c. Bootloader must be one sector in length
- [ ] d. The bootloader loads the BIOS
- [x] e. LILO is a bootloader ✔

Your answer is correct.

The correct answers are: LILO is a bootloader, Modern Bootloaders often allow configuring the way an OS boots, Bootloaders allow selection of OS to boot from

Question **12**

Incorrect

Mark 0.00 out of 1.00

For each of the resource allocation diagram shown,

infer whether the graph contains at least one deadlock or not.

Yes	No		
○	●		✖
●	○		✖

: Yes

: No

Question **13**

Partially correct

Mark 0.71 out of 1.00

Mark the statements about device drivers by marking as True or False.

True	False		
		It's possible that a particular hardware has multiple device drivers available for it.	✖
		xv6 has device drivers for IDE disk and console.	✔
		A disk driver converts OS's logical view of disk into physical locations on disk.	✔
		A device driver code is specific to a hardware device	✔
		All devices of the same type (e.g. 2 hard disks) can typically use the same device driver	✔
		Writing a device driver mandatorily demands reading the technical documentation about the hardware.	✖
		Device driver is an intermediary between the end-user and OS	✔

It's possible that a particular hardware has multiple device drivers available for it.: True
xv6 has device drivers for IDE disk and console.: True
A disk driver converts OS's logical view of disk into physical locations on disk.: True
A device driver code is specific to a hardware device: True
All devices of the same type (e.g. 2 hard disks) can typically use the same device driver: True
Writing a device driver mandatorily demands reading the technical documentation about the hardware.: True
Device driver is an intermediary between the end-user and OS: False

Question **14**

Partially correct

Mark 0.33 out of 1.00

Consider this program.
Some statements are identified using the // comment at the end.

Assume that = is an atomic operation.

```
#include <stdio.h>
#include <pthread.h>
long c = 0, c1 = 0, c2 = 0, run = 1;
void *thread1(void *arg) {
    while(run == 1) {//E
        c = 10; //A
        c1 = c2 + 5; //B
    }
}
void *thread2(void *arg) {
    while(run == 1) {//F
        c = 20;//C
        c2 = c1 + 3;//D
    }
}
int main() {
    pthread_t th1, th2;
    pthread_create(&th1, NULL, thread1, NULL);
    pthread_create(&th2, NULL, thread2, NULL);
    sleep(2);
    run = 0;
    fprintf(stdout, "c = %ld c1+c2 = %ld c1 = %ld c2 = %ld \n", c, c1+c2, c1, c2);
    fflush(stdout);
}
```

Which statements are part of the critical Section?

Yes	No		
✗	✓	F	✗
✓	✗	D	✓
✗	✓	C	✗
✗	✓	A	✗
✓	✗	B	✓
✗	✓	E	✗

F: No
D: Yes
C: No
A: No
B: Yes
E: No

Question **15**

Partially correct

Mark 1.43 out of 2.00

Mark statements as T/F

All statements are in the context of preventing deadlocks.

True	False		
◉	○	A process holding one resources and waiting for just one more resource can also be involved in a deadlock.	✔
◉	○	If a resource allocation graph contains a cycle then there is a guarantee of a deadlock	✖
◉	○	The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order	✖
◉	○	Circular wait is avoided by enforcing a lock ordering	✔
◉	○	Hold and wait means a thread/process holding some locks and waiting for acquiring some.	✔
◉	○	Deadlock is possible if all the conditions are met at the same time: Mutual exclusion, hold and wait, no pre-emption, circular wait.	✔
◉	○	Mutual exclusion is a necessary condition for deadlock because it brings in locks on which deadlock happens	✔

A process holding one resources and waiting for just one more resource can also be involved in a deadlock.: True
If a resource allocation graph contains a cycle then there is a guarantee of a deadlock: False
The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order: False
Circular wait is avoided by enforcing a lock ordering: True
Hold and wait means a thread/process holding some locks and waiting for acquiring some.: True
Deadlock is possible if all the conditions are met at the same time: Mutual exclusion, hold and wait, no pre-emption, circular wait.: True
Mutual exclusion is a necessary condition for deadlock because it brings in locks on which deadlock happens: True

Question **16**

Correct

Mark 1.00 out of 1.00

Match the left side use(or non-use) of a synchronization primitive with the best option on the right side.

This is the smallest primitive made available in software, using the hardware provided atomic instructions	spinlock	✔
This tool is useful for event-wait scenarios	semaphore	✔
This tool is more useful on multiprocessor systems	spinlock	✔
This tool is quite attractive in solving the main bounded buffer problem	semaphore	✔
This tool is very useful for waiting for 'something'	condition variables	✔

Your answer is correct.

The correct answer is: This is the smallest primitive made available in software, using the hardware provided atomic instructions → spinlock, This tool is useful for event-wait scenarios → semaphore, This tool is more useful on multiprocessor systems → spinlock, This tool is quite attractive in solving the main bounded buffer problem → semaphore, This tool is very useful for waiting for 'something' → condition variables

Question **17**

Correct

Mark 1.00 out of 1.00

The permissions -rwx--x--x on a file mean

- [x] a. The file can be read only by the owner ✔
- [ ] b. 'cat' on the file by owner will not work
- [ ] c. 'cat' on the file by any user will work
- [x] d. 'rm' on the file by any user will work ✔
- [x] e. The file can be executed by anyone ✔
- [x] f. The file can be written only by the owner ✔

Your answer is correct.

The correct answers are: The file can be executed by anyone, The file can be read only by the owner, The file can be written only by the owner, 'rm' on the file by any user will work

Question **18**

Incorrect

Mark 0.00 out of 1.00

Note: for this question you get full marks if you select all and only correct options, you get ZERO if at least one option is wrong or not selected.

Select all the correct statements about log structured file systems.

- [x] a. a transaction is said to be committed when all operations are written to file system ✖
- [x] b. log may be kept on same block device or another block device ✔
- [ ] c. file system recovery may end up losing data
- [x] d. even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery ✔
- [x] e. file system recovery recovers all the lost data ✖

Your answer is incorrect.

The correct answers are: file system recovery may end up losing data, log may be kept on same block device or another block device, even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery

Question **19**

Incorrect

Mark 0.00 out of 1.00

Consider the structure of directory entry in ext2, as shown in this diagram.

Select the correct statements about the directory entry in ext2 file system.

The correct formula for rec_len is (when entries are continuously stored)

- a. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + (-1) * (strlen(name)%4)
- b. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + (strlen(name)- 4) %4
- c. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + 4 - (strlen(name)%4) ✖
- d. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + (-1) * (strlen(name)- 4)
- e. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) %4
- f. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + strlen(name)

Your answer is incorrect.

The correct answer is: rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + (-1) * (strlen(name)- 4)

Question **20**

Partially correct

Mark 0.50 out of 1.00

Mark whether the given sequence of events is possible or not-possible. Also, select the reason for your answer.

For each sequence it's a not-possible sequence if some important event is not mentioned in the sequence.

Assume that the kernel code is non-interruptible and uniprocessor system.

```
Process P1 executing a system call
Timer interrupt
Generic interrupt handler runs
Scheduler runs
Scheduler selects P2 for execution
P2 returns from timer interrupt handler
Process p2, user code executing
```

This sequence of events is: not-possible ✔

Because

Interrupts are not possible when kernel code is running ✖

Question **21**

Incorrect

Mark 0.00 out of 1.00

The given semaphore implementation faces which problem?

Assume any suitable code for signal()

Note: blocks means waits in a wait queue.

```
struct semaphore {
        int val;
        spinlock lk;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0)
        ;
        (s->val)--;
        spinunlock(&(s->sl));
}
```

- a. blocks holding a spinlock
- b. deadlock
- c. too much spinning, bounded wait not guaranteed ✖
- d. not holding lock after unblock

Your answer is incorrect.

The correct answer is: deadlock

Question **22**

Partially correct

Mark 0.80 out of 1.00

Mark statements True/False w.r.t. change of states of a process.

Reference: The process state diagram (and your understanding of how kernel code works)

Figure 3.2 Diagram of process state.

True	False		
○	●	A process in RUNNING state only can become TERMINATED because scheduler moves it to ZOMBIE state	✔
●	○	A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.	✖
●	○	A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred	✔
●	○	Every process has to go through ZOMBIE state, at least for a small duration.	✔
●	○	Only a process in READY state is considered by scheduler	✔

A process in RUNNING state only can become TERMINATED because scheduler moves it to ZOMBIE state: False
A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.: False
A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred: True
Every process has to go through ZOMBIE state, at least for a small duration.: True
Only a process in READY state is considered by scheduler: True

Question **23**

Correct

Mark 1.00 out of 1.00

Select T/F for statements about Volume Managers.
Do pay attention to the use of the words physical partition and physical volume.

True	False			
◉	○	The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.	✔	
◉	○	A logical volume can be extended in size but upto the size of volume group	✔	
◉	○	A logical volume may span across multiple physical volumes	✔	
◉	○	The volume manager stores additional metadata on the physical disk partitions	✔	
◉	○	A physical partition should be initialized as a physial volume, before it can be used by volume manager.	✔	
◉	○	A volume group consists of multiple physical volumes	✔	
◉	○	A logical volume may span across multiple physical partitions	✔	since a physical volume is made up of physical partitions, and a volume can span across multiple PVs, it can also span across multiple PP

The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.: True
A logical volume can be extended in size but upto the size of volume group: True
A logical volume may span across multiple physical volumes: True
The volume manager stores additional metadata on the physical disk partitions: True
A physical partition should be initialized as a physial volume, before it can be used by volume manager.: True
A volume group consists of multiple physical volumes: True
A logical volume may span across multiple physical partitions: True

Question **24**

Correct

Mark 1.00 out of 1.00

Map the block allocation scheme with the problem it suffers from

(Match pairs 1-1, match a scheme with the problem that it suffers from relatively the most, compared to others)

Continuous allocation	need for compaction	✔
Linked allocation	Too many seeks	✔
Indexed Allocation	Overhead of reading metadata blocks	✔

Your answer is correct.

The correct answer is: Continuous allocation → need for compaction, Linked allocation → Too many seeks, Indexed Allocation → Overhead of reading metadata blocks

Question **25**

Correct

Mark 1.00 out of 1.00

This one is not a system call:

- ○ a. open
- ○ b. read
- ○ c. write
- ◉ d. scheduler ✔

Your answer is correct.

The correct answer is: scheduler

Question **26**

Correct

Mark 1.00 out of 1.00

Match the pairs.

This question is based on your general knowledge about operating systems/related concepts and their features.

Java thereads	monitors,re-entrant locks, semaphores ✔
Linux threads	atomic-instructions, spinlocks, etc. ✔
POSIX threads	semaphore, mutex, condition variables ✔

Your answer is correct.

The correct answer is: Java thereads → monitors,re-entrant locks, semaphores, Linux threads → atomic-instructions, spinlocks, etc., POSIX threads → semaphore, mutex, condition variables

Question **27**

Correct

Mark 1.00 out of 1.00

Consider the following list of free chunks, in continuous memory management:

7k, 15k, 21k, 14k, 19k, 6k

Suppose there is a request for chunk of size 5k, then the free chunk selected under each of the following schemes will be

Best fit: 6k ✔

First fit: 7k ✔

Worst fit: 21k ✔

Question **28**

Correct

Mark 1.00 out of 1.00

This one is not a scheduling algorithm

- a. Round Robin
- b. SJF
- c. Mergesort ✔
- d. FCFS

Your answer is correct.

The correct answer is: Mergesort

Question **29**

Correct

Mark 1.00 out of 1.00

Mark whether the concept is related to scheduling or not.

Yes	No		
✓		timer interrupt	✓
✓		context-switch	✓
✓		ready-queue	✓
	✓	file-table	✓
✓		runnable process	✓

timer interrupt: Yes
context-switch: Yes
ready-queue: Yes
file-table: No
runnable process: Yes

Question **30**

Partially correct

Mark 1.00 out of 2.00

Map ext2 data structure features with their purpose

Many copies of Superblock	Choose...
Free blocks count in superblock and group descriptor ✖	Redundancy to ensure the most crucial data structure is not lost
Used directories count in group descriptor ✖	is redundant and helps do calculations of directory entries faster
Combining file type and access rights in one variable ✔	saves 1 byte of space
rec_len field in directory entry ✖	Try to keep all the data of a directory and it's file close together in a group
File Name is padded ✔	aligns all memory accesses on word boundary, improving performance
Inode bitmap is one block ✔	limits total number of files that can belong to a group
Block bitmap is one block ✔	Limits the size of a block group, thus improvising on purpose of a group
Mount count in superblock ✔	to enforce file check after certain amount of mounts at boot time
Inode table location in Group Descriptor ✖	is redundant and helps do calculations of directory entries faster
Inode table ✔	All inodes are kept together so that one disk read leads to reading many inodes together, effectively doing a buffering of subsequent inode reads, and to save space on disk
A group ✖	Redundancy to ensure the most crucial data structure is not lost

Your answer is partially correct.

You have correctly selected 6.

The correct answer is: Many copies of Superblock → Redundancy to ensure the most crucial data structure is not lost, Free blocks count in superblock and group descriptor → Redundancy to help fsck restore consistency, Used directories count in group descriptor → attempt is made to evenly spread the first-level directories, this count is used there, Combining file type and access rights in one variable → saves 1 byte of space, rec_len field in directory entry → allows holes and linking of entries in directory, File Name is padded → aligns all memory accesses on word boundary, improving performance, Inode bitmap is one block → limits total number of files that can belong to a group, Block bitmap is one block → Limits the size of a block group, thus improvising on purpose of a group, Mount count in superblock → to enforce file check after certain amount of mounts at boot time, Inode table location in Group Descriptor → Obvious, as it's per group and not per file-system, Inode table → All inodes are kept together so that one disk read leads to reading many inodes together, effectively doing a buffering of subsequent inode reads, and to save space on disk, A group → Try to keep all the data of a directory and it's file close together in a group

Question **31**

Partially correct

Mark 1.85 out of 2.00

Mark True/False

Statements about scheduling and scheduling algorithms

True	False		
●	○	The nice() system call is used to set priorities for processes	✔
●	○	Aging is used to ensure that low-priority processes do not starve in priority scheduling.	✔
○	●	In non-pre-emptive priority scheduling, the highest priority process is scheduled and runs until it gives up CPU.	✖
●	○	xv6 code does not care about Processor Affinity	✔
●	○	In pre-emptive priority scheduling, priority is implemented by assigning more time quantum to higher priority process.	✔
●	○	A scheduling algorithm is non-premptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.	✔
●	○	Processor Affinity refers to memory accesses of a process being stored on cache of that processor	✔
●	○	Response time will be quite poor on non-interruptible kernels	✔
●	○	Shortest Remaining Time First algorithm is nothing but pre-emptive Shortest Job First algorithm	✔
●	○	On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread	✔
●	○	Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.	✔
●	○	Pre-emptive scheduling leads to many race conditions in kernel code.	✔
●	○	Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.	✔

The nice() system call is used to set priorities for processes: True
Aging is used to ensure that low-priority processes do not starve in priority scheduling.: True
In non-pre-emptive priority scheduling, the highest priority process is scheduled and runs until it gives up CPU.: True
xv6 code does not care about Processor Affinity: True
In pre-emptive priority scheduling, priority is implemented by assigning more time quantum to higher priority process.: True
A scheduling algorithm is non-premptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.: True
Processor Affinity refers to memory accesses of a process being stored on cache of that processor: True
Response time will be quite poor on non-interruptible kernels: True
Shortest Remaining Time First algorithm is nothing but pre-emptive Shortest Job First algorithm: True
On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread: True
Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.: True
Pre-emptive scheduling leads to many race conditions in kernel code.: True
Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.: True

Question **32**

Partially correct

Mark 1.17 out of 2.00

The unix file semantics demand that changes to any open file are visible immediately to any other processes accessing that file at that point in time.

Select the data-structure/programmatic features that ensure the implementation of unix semantics. (Assume there is no mmap())

Yes	No		
○	●	All processes accessing the same file share the file descriptor among themselves	✔
○	●	The pointer entry in the file descriptor array entry points to the data of the file directly	✔
●	○	There is only one global file structure per on-disk file.	✖
○	●	All file accesses are made using only global variables	✔
●	○	The 'file offset' is shared among all the processes that access the file.	✖
○	●	No synchronization is implemented so that changes are made available immediately.	✔
●	○	A single spinlock is to be used to protect the unique global 'file structure' representing the file, thus synchronizing access, and making other processes wait for earlier process to finish writing so that writes get visible immediately.	✖
●	○	There is only one in-memory copy of the on disk file's contents in kernel memory/buffers	✔
●	○	The file descriptors in every PCB are pointers to the same global file structure.	✖
○	●	The file descriptor array is external to PCB and all processes that share a file, have pointers to same file-descriptors' array	✔
●	○	All file structures representing any open file, give access to the same in-memory copy of the file's contents	✔
●	○	The 'file offset' index is stored outside the file-structure to which file-descriptor array points	✖

All processes accessing the same file share the file descriptor among themselves: No
The pointer entry in the file descriptor array entry points to the data of the file directly: No
There is only one global file structure per on-disk file.: No
All file accesses are made using only global variables: No
The 'file offset' is shared among all the processes that access the file.: No
No synchronization is implemented so that changes are made available immediately.: No
A single spinlock is to be used to protect the unique global 'file structure' representing the file, thus synchronizing access, and making other processes wait for earlier process to finish writing so that writes get visible immediately.: No
There is only one in-memory copy of the on disk file's contents in kernel memory/buffers: Yes
The file descriptors in every PCB are pointers to the same global file structure.: No
The file descriptor array is external to PCB and all processes that share a file, have pointers to same file-descriptors' array: No
All file structures representing any open file, give access to the same in-memory copy of the file's contents: Yes
The 'file offset' index is stored outside the file-structure to which file-descriptor array points: No

Question **33**

Partially correct

Mark 0.33 out of 2.00

Map the function in xv6's file system code, to it's perceived logical layer.

namei	inode	✗
filestat()	Choose...	
dirlookup	directory	✓
ialloc	file descriptor	✗
stati	Choose...	
ideintr	buffer cache	✗
bread	Choose...	
balloc	file descriptor	✗
sys_chdir()	system call	✓
skipelem	system call	✗
commit	system call	✗
bmap	system call	✗

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: namei → pathname lookup, filestat() → file descriptor, dirlookup → directory, ialloc → inode, stati → inode, ideintr → disk driver, bread → buffer cache, balloc → block allocation on disk, sys_chdir() → system call, skipelem → pathname lookup, commit → logging, bmap → inode

◄ Course Exit Feedback

Jump to...

xv6-public-master ►

Started on	Monday, 24 January 2022, 7:07:42 PM
State	Finished
Completed on	Monday, 24 January 2022, 8:08:11 PM
Time taken	1 hour
Grade	**8.90** out of 20.00 (**45**%)

Question **1**

Complete

Mark 0.80 out of 1.00

Match the register with the segment used with it.

ebp	ss
eip	cs
edi	ds
esp	ss
esi	ds

The correct answer is: ebp → ss, eip → cs, edi → es, esp → ss, esi → ds

Question **2**

Complete

Mark 1.00 out of 1.00

```
int value = 5;
int main()
{
   pid_t pid;
   pid = fork();
   if (pid == 0) { /* child process */
      value += 15;
      return 0;
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("%d", value); /* LINE A */
   }
   return 0;
}
```

What's the value printed here at LINE A?

Answer: 5

The correct answer is: 5

Question **3**

Complete

Mark 0.50 out of 0.50

Is the command "cat README > done &" possible on xv6? (Note the & in the end)

- a. no
- b. yes

The correct answer is: yes

Question **4**

Complete

Mark 0.00 out of 2.00

xv6.img: bootblock kernel

```
    dd if=/dev/zero of=xv6.img count=10000
    dd if=bootblock of=xv6.img conv=notrunc
    dd if=kernel of=xv6.img seek=1 conv=notrunc
```

Consider above lines from the Makefile. Which of the following is incorrect?

- [x] a. The xv6.img is the virtual disk that is created by combining the bootblock and the kernel file.
- [x] b. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk.
- [ ] c. The size of xv6.img is exactly = (size of bootblock) + (size of kernel)
- [ ] d. xv6.img is the virtual processor used by the qemu emulator
- [ ] e. The size of the kernel file is nearly 5 MB
- [ ] f. Blocks in xv6.img after kernel may be all zeroes.
- [ ] g. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies 10,000 blocks on the disk.
- [ ] h. The kernel is located at block-1 of the xv6.img
- [ ] i. The bootblock is located on block-0 of the xv6.img
- [ ] j. The bootblock may be 512 bytes or less (looking at the Makefile instruction)
- [x] k. The size of the xv6.img is nearly 5 MB

The correct answers are: xv6.img is the virtual processor used by the qemu emulator, The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk., The size of the kernel file is nearly 5 MB, The size of xv6.img is exactly = (size of bootblock) + (size of kernel)

Question **5**

Complete

Mark 0.43 out of 1.00

Rank the following storage systems from slowest (first) to fastest(last)

You can drag and drop the items below/above each other.

- Registers
- Cache
- Main memory
- Nonvolatile memory
- Magnetic tapes
- Optical disk
- Hard-disk drives

The correct order for these items is as follows:

1. Magnetic tapes
2. Optical disk
3. Hard-disk drives
4. Nonvolatile memory
5. Main memory
6. Cache
7. Registers

Question **6**

Complete

Mark 1.00 out of 1.00

How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security) ?

Select one:

- a. It disallows hardware interrupts when a process is running
- b. It prohibits one process from accessing other process's memory
- c. It prohibits a user mode process from running privileged instructions
- d. It prohibits invocation of kernel code completely, if a user program is running

The correct answer is: It prohibits a user mode process from running privileged instructions

Question **7**

Complete

Mark 0.00 out of 2.00

Which of the following are NOT a part of job of a typical compiler?

- [ ] a. Suggest alternative pieces of code that can be written
- [x] b. Check the program for syntactical errors
- [ ] c. Check the program for logical errors
- [x] d. Convert high level langauge code to machine code
- [x] e. Process the # directives in a C program
- [ ] f. Invoke the linker to link the function calls with their code, extern globals with their declaration

The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

Question **8**

Complete

Mark 0.00 out of 2.00

Match the program with it's output (ignore newlines in the output. Just focus on the count of the number of 'hi')

main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }	hi hi
main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }	hi hi hi hi
main() { int i = NULL; fork(); printf("hi\n"); }	No output
main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }	hi

The correct answer is: main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi, main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi, main() { int i = NULL; fork(); printf("hi\n"); } → hi hi, main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi hi

Question **9**

Complete

Mark 0.83 out of 2.00

Select all statements that correctly explain the use/purpose of system calls.

Select one or more:

- [x] a. Provide an environment for process creation
- [ ] b. Switch from user mode to kernel mode
- [ ] c. Handle ALL types of interrupts
- [ ] d. Handle exceptions like division by zero
- [x] e. Run each instruction of an application program
- [x] f. Allow I/O device access to user processes
- [x] g. Provide services for accessing files

The correct answers are: Switch from user mode to kernel mode, Provide services for accessing files, Allow I/O device access to user processes, Provide an environment for process creation

Question **10**

Complete

Mark 0.50 out of 0.50

Compare multiprogramming with multitasking

- (•) a. A multiprogramming system is not necessarily multitasking
- ( ) b. A multitasking system is not necessarily multiprogramming

The correct answer is: A multiprogramming system is not necessarily multitasking

Question **11**

Complete

Mark 0.60 out of 1.00

Select all the correct statements about two modes of CPU operation

Select one or more:

- [x] a. The two modes are essential for a multitasking system
- [ ] b. Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode
- [x] c. The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously
- [ ] d. The two modes are essential for a multiprogramming system
- [x] e. There is an instruction like 'iret' to return from kernel mode to user mode

The correct answers are: The two modes are essential for a multiprogramming system, The two modes are essential for a multitasking system, There is an instruction like 'iret' to return from kernel mode to user mode, The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously, Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode

Question **12**

Complete

Mark 0.50 out of 0.50

Is the terminal a part of the kernel on GNU/Linux systems?

- (●) a. no
- ( ) b. yes

The correct answer is: no

Question **13**

Complete

Mark 1.00 out of 1.00

Why should a program exist in memory before it starts executing ?

- ( ) a. Because the hard disk is a slow medium
- (●) b. Becase the processor can run instructions and access data only from memory
- ( ) c. Because the variables of the program are stored in memory
- ( ) d. Because the memory is volatile

The correct answer is: Becase the processor can run instructions and access data only from memory

Question **14**

Complete

Mark 1.33 out of 2.00

Which of the following instructions should be privileged?

Select one or more:

- [ ] a. Read the clock.
- [x] b. Access memory management unit of the processor
- [ ] c. Access I/O device.
- [x] d. Turn off interrupts.
- [ ] e. Set value of a memory location
- [x] f. Set value of timer.
- [ ] g. Access a general purpose register
- [x] h. Modify entries in device-status table
- [ ] i. Switch from user to kernel mode.

The correct answers are: Set value of timer., Access memory management unit of the processor, Turn off interrupts., Modify entries in device-status table, Access I/O device., Switch from user to kernel mode.

Question **15**

Complete

Mark 0.07 out of 2.00

Select all the correct statements about calling convention on x86 32-bit.

- [ ] a. The ebp pointers saved on the stack constitute a chain of activation records
- [x] b. The return value is either stored on the stack or returned in the eax register
- [ ] c. The two lines in the beginning of each function, "push %ebp; mov %esp, %ebp", create space for local variables
- [x] d. Parameters may be passed in registers or on stack
- [ ] e. Return address is one location above the ebp
- [ ] f. Parameters may be passed in registers or on stack
- [x] g. Compiler may allocate more memory on stack than needed
- [ ] h. Space for local variables is allocated by substracting the stack pointer inside the code of the called function
- [ ] i. Paramters are pushed on the stack in left-right order
- [ ] j. during execution of a function, ebp is pointing to the old ebp
- [ ] k. Space for local variables is allocated by substracting the stack pointer inside the code of the caller function

The correct answers are: Compiler may allocate more memory on stack than needed, Parameters may be passed in registers or on stack, Parameters may be passed in registers or on stack, Return address is one location above the ebp, during execution of a function, ebp is pointing to the old ebp, Space for local variables is allocated by substracting the stack pointer inside the code of the called function, The ebp pointers saved on the stack constitute a chain of activation records

Question **16**

Complete

Mark 0.33 out of 0.50

Order the following events in boot process (from 1 onwards)

Shell	3
BIOS	1
Init	4
OS	6
Login interface	5
Boot loader	2

The correct answer is: Shell → 6, BIOS → 1, Init → 4, OS → 3, Login interface → 5, Boot loader → 2

◄ (Task) Compulsory xv6 task

Jump to...

(Optional Assignment) Shell Programming(Conformance tests) ►

Started on	Wednesday, 9 February 2022, 7:00:12 PM
State	Finished
Completed on	Wednesday, 9 February 2022, 7:46:38 PM
Time taken	46 mins 26 secs
Grade	**3.00** out of 11.00 (**27**%)

Question **1**

Complete

Mark 0.00 out of 1.00

The number of GDT entries setup during boot process of xv6 is

- a. 2
- b. 3
- c. 0
- d. 256
- e. 4
- f. 255

The correct answer is: 3

Question **2**

Complete

Mark 0.00 out of 1.00

x86 provides which of the following type of memory management options?

- a. segmentation and one level paging
- b. segmentation or one or two level paging
- c. segmentation and two level paging
- d. segmentation or paging
- e. segmentation and one or two level paging
- f. segmentation only

The correct answer is: segmentation and one or two level paging

Question **3**

Complete

Mark 0.00 out of 1.00

which of the following is not a difference between real mode and protected mode

- a. in real mode general purpose registers are 16 bit, in protected mode they are 32 bit
- b. in real mode the addressable memory is more than in protected mode
- c. in real mode the addressable memory is less than in protected mode
- d. in real mode the segment is multiplied by 16, in protected mode segment is used as index in GDT
- e. processor starts in real mode

The correct answer is: in real mode the addressable memory is more than in protected mode

Question **4**

Complete

Mark 0.00 out of 1.00

The kernel ELF file contains how many Program headers?

- a. 4
- b. 2
- c. 3
- d. 9
- e. 10

The correct answer is: 3

Question **5**

Not answered

Marked out of 0.50

code line, MMU setting: Match the line of xv6 code with the MMU setup employed

Answer:

The correct answer is: inb $0x64,%al

Question **6**

Complete

Mark 1.00 out of 1.00

The kernel is loaded at Physical Address

- a. 0x00100000
- b. 0x0010000
- c. 0x80100000
- d. 0x80000000

The correct answer is: 0x00100000

Question **7**

Complete

Mark 0.00 out of 1.00

Why is the code of entry() in Assembly and not in C?

- a. Because the symbol entry() is inside the ELF file
- b. Because the kernel code must begin in assembly
- c. Because it needs to setup paging
- d. There is no particular reason, it could also be in C

The correct answer is: Because it needs to setup paging

Question **8**

Complete

Mark 1.00 out of 1.00

The ljmp instruction in general does

- a. change the CS and EIP to 32 bit mode, and jumps to next line of code
- b. change the CS and EIP to 32 bit mode, and jumps to new value of EIP
- c. change the CS and EIP to 32 bit mode
- d. change the CS and EIP to 32 bit mode, and jumps to kernel code

The correct answer is: change the CS and EIP to 32 bit mode, and jumps to new value of EIP

Question **9**

Complete

Mark 0.00 out of 1.00

The variable $stack in entry.S is

- a. located at the value given by %esp as setup by bootmain()
- b. located at 0x7c00
- c. a memory region allocated as a part of entry.S
- d. located at less than 0x7c00
- e. located at 0

The correct answer is: a memory region allocated as a part of entry.S

Question **10**

Not answered

Marked out of 0.50

Match the pairs of which action is taken by whom

Answer:

The correct answer is: kernel

Question **11**

Complete

Mark 0.00 out of 1.00

ELF Magic number is

- a. 0xFFFFFFFFF
- b. 0
- c. 0xELFELFELF
- d. 0xELF
- e. 0x0x464CELF
- f. 0x464C457FL
- g. 0x464C457FU

The correct answer is: 0x464C457FU

Question **12**

Complete

Mark 1.00 out of 1.00

The right side of line of code "entry = (void(*)(void))(elf->entry)" means

- a. Convert the "entry" in ELF structure into void
- b. Get the "entry" in ELF structure and convert it into a function pointer accepting no arguments and returning nothing
- c. Get the "entry" in ELF structure and convert it into a function void pointer
- d. Get the "entry" in ELF structure and convert it into a void pointer

The correct answer is: Get the "entry" in ELF structure and convert it into a function pointer accepting no arguments and returning nothing

◄ Homework questions: Basics of MM, xv6 booting

Jump to...

(Code) Files, redirection, dup, (IPC)pipe ►

Started on	Monday, 21 February 2022, 7:00:28 PM
State	Finished
Completed on	Monday, 21 February 2022, 7:49:24 PM
Time taken	48 mins 56 secs
Grade	**8.30** out of 10.00 (**83**%)

Question **1**

Complete

Mark 0.80 out of 1.00

Match the elements of C program to their place in memory

Local Static variables	Data
Global variables	Data
Code of main()	Code
Function code	Code
Arguments	Stack
Malloced Memory	Heap
#include files	No Memory needed
#define MACROS	No memory needed
Local Variables	Stack
Global Static variables	Data

The correct answer is: Local Static variables → Data, Global variables → Data, Code of main() → Code, Function code → Code, Arguments → Stack, Malloced Memory → Heap, #include files → No memory needed, #define MACROS → No Memory needed, Local Variables → Stack, Global Static variables → Data

Question **2**

Complete

Mark 1.00 out of 1.00

What will be the output of this program

```
int main() {
  int fd;
  printf("%d ",   open("/etc/passwd", O_RDONLY));
  close(1);
  fd = printf("%d ", open("/etc/passwd", O_RDONLY));
  close(fd);
   fd = printf("%d ", open("/etc/passwd", O_RDONLY));
}
```

- a. 3 1 2
- b. 3 3 3
- c. 3 1 1
- d. 1 1 1
- e. 3 4 5
- f. 2 2 2

The correct answer is: 3 1 1

Question **3**

Complete

Mark 1.00 out of 1.00

Arrange in correct order, the files involved in execution of system call

vectors.S	2
trap.c	4
usys.S	1
trapasm.S	3

The correct answer is: vectors.S → 2, trap.c → 4, usys.S → 1, trapasm.S → 3

Question **4**

Complete

Mark 1.00 out of 1.00

The "push 0" in vectors.S is

- a. A placeholder to match the size of struct trapframe
- b. Place for the error number value
- c. To indicate that it's a system call and not a hardware interrupt
- d. To be filled in as the return value of the system call

The correct answer is: Place for the error number value

Question **5**

Complete

Mark 0.00 out of 1.00

A process blocks itself means

- a. The kernel code of an interrupt handler, moves the process to a waiting queue and calls scheduler
- b. The application code calls the scheduler
- c. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler
- d. The kernel code of system call calls scheduler

The correct answer is: The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

Question **6**

Complete

Mark 1.00 out of 1.00

Select the odd one out

- a. Kernel stack of new process to Process stack of new process
- b. Process stack of running process to kernel stack of running process
- c. Kernel stack of running process to kernel stack of scheduler
- d. Kernel stack of scheduler to kernel stack of new process
- e. Kernel stack of new process to kernel stack of scheduler

The correct answer is: Kernel stack of new process to kernel stack of scheduler

Question **7**

Complete

Mark 0.50 out of 0.50

Match the File descriptors to their meaning

0	Standard Input
2	Standard error
1	Standard output

The correct answer is: 0 → Standard Input, 2 → Standard error, 1 → Standard output

Question **8**

Complete

Mark 1.00 out of 1.00

Which of the following is not a task of the code of swtch() function

- [ ] a. Switch stacks
- [x] b. Change the kernel stack location
- [ ] c. Load the new context
- [x] d. Jump to next context EIP
- [x] e. Save the return value of the old context code
- [ ] f. Save the old context

The correct answers are: Save the return value of the old context code, Change the kernel stack location

Question **9**

Complete

Mark 0.50 out of 0.50

Match the names of PCB structures with kernel

linux	struct task_struct
xv6	struct proc

The correct answer is: linux → struct task_struct, xv6 → struct proc

Question **10**

Complete

Mark 0.50 out of 0.50

Match the MACRO with it's meaning

KERNBASE	2 GB
KERNLINK	2.224 GB
PHYSTOP	224 MB

The correct answer is: KERNBASE → 2 GB, KERNLINK → 2.224 GB, PHYSTOP → 224 MB

Question **11**

Complete

Mark 1.00 out of 1.00

The trapframe, in xv6, is built by the

- a. hardware, trapasm.S
- b. hardware, vectors.S
- c. hardware, vectors.S, trapasm.S, trap()
- d. hardware, vectors.S, trapasm.S
- e. vectors.S, trapasm.S

The correct answer is: hardware, vectors.S, trapasm.S

Question **12**

Complete

Mark 0.00 out of 0.50

Which of the following state transitions are not possible?

- [ ] a. Running -> Waiting
- [x] b. Ready -> Waiting
- [x] c. Waiting -> Terminated
- [x] d. Ready -> Terminated

The correct answers are: Ready -> Terminated, Waiting -> Terminated, Ready -> Waiting

◄ Description of some possible course mini projects

Jump to...

(Code) mmap related programs ►

Started on	Saturday, 26 February 2022, 5:18:30 PM
State	Finished
Completed on	Saturday, 26 February 2022, 6:30:44 PM
Time taken	1 hour 12 mins
Grade	**8.55** out of 15.00 (**57**%)

Question **1**

Complete

Mark 0.50 out of 0.50

Map the technique with it's feature/problem

static linking	large executable file
dynamic loading	allocate memory only if needed
dynamic linking	small executable file
static loading	wastage of physical memory

The correct answer is: static linking → large executable file, dynamic loading → allocate memory only if needed, dynamic linking → small executable file, static loading → wastage of physical memory

Question **2**

Complete

Mark 0.00 out of 1.00

Calculate the EAT in NANO-seconds (upto 2 decimal points) w.r.t. a page fault, given

Memory access time = 299 ns

Average page fault service time = 6 ms

Page fault rate = 0.8

Answer: 4.80

The correct answer is: 4800059.80

Question **3**

Complete

Mark 1.00 out of 1.00

Given six memory partitions of 300 KB , 600 KB , 350 KB , 200 KB , 750 KB , and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB and 500 KB (in order)?

best fit 115 KB	125 KB
best fit 500 KB	600 KB
worst fit 500 KB	635 KB
worst fit 115 KB	750 KB
first fit 500 KB	600 KB
first fit 115 KB	300 KB

The correct answer is: best fit 115 KB → 125 KB, best fit 500 KB → 600 KB, worst fit 500 KB → 635 KB, worst fit 115 KB → 750 KB, first fit 500 KB → 600 KB, first fit 115 KB → 300 KB

Question **4**

Complete

Mark 0.29 out of 1.00

Compare paging with demand paging and select the correct statements.

Select one or more:

- [ ] a. TLB hit ration has zero impact in effective memory access time in demand paging.
- [x] b. Both demand paging and paging support shared memory pages.
- [ ] c. Calculations of number of bits for page number and offset are same in paging and demand paging.
- [ ] d. Demand paging requires additional hardware support, compared to paging.
- [ ] e. The meaning of valid-invalid bit in page table is different in paging and demand-paging.
- [ ] f. Paging requires NO hardware support in CPU
- [ ] g. With paging, it's possible to have user programs bigger than physical memory.
- [ ] h. With demand paging, it's possible to have user programs bigger than physical memory.
- [ ] i. Paging requires some hardware support in CPU
- [x] j. Demand paging always increases effective memory access time.

The correct answers are: Demand paging requires additional hardware support, compared to paging., Both demand paging and paging support shared memory pages., With demand paging, it's possible to have user programs bigger than physical memory., Demand paging always increases effective memory access time., Paging requires some hardware support in CPU, Calculations of number of bits for page number and offset are same in paging and demand paging., The meaning of valid-invalid bit in page table is different in paging and demand-paging.

Question **5**

Complete

Mark 0.36 out of 0.50

Map the parts of a C code to the memory regions they are related to

local variables	stack
global un-initialized variables	bss
static variables	data
global initialized variables	data
function arguments	stack
malloced memory	stack
functions	stack

The correct answer is: local variables → stack, global un-initialized variables → bss, static variables → data, global initialized variables → data, function arguments → stack, malloced memory → heap, functions → code

Question **6**

Complete

Mark 0.75 out of 1.00

which of the following, do you think, are valid concerns for making the kernel pageable?

- [x] a. The kernel must have some dedicated frames for it's own work
- [ ] b. No data structure of kernel should be pageable
- [x] c. The kernel's own page tables should not be pageable
- [ ] d. The disk driver and disk interrupt handler should not be pageable
- [ ] e. No part of kernel code should be pageable.
- [x] f. The page fault handler should not be pageable

The correct answers are: The kernel's own page tables should not be pageable, The page fault handler should not be pageable, The kernel must have some dedicated frames for it's own work, The disk driver and disk interrupt handler should not be pageable

Question **7**

Complete

Mark 0.75 out of 1.00

W.r.t the figure given below, mark the given statements as True or False.

Figure 10.9 Need for page replacement.

True	False	
◉	○	Kernel occupies two page frames
◉	○	Handling this scenario demands two disk I/Os
◉	○	Local replacement means chose any of the frames 2, 3, 6
○	◉	Local replacement means chose any of the frame from 2 to 7
◉	○	Global replacement means chose any of the frame from 0 to 7
○	◉	Global replacement means chose any of the frame from 2 to 7
◉	○	The kernel's pages can not used for replacement if kernel is not pageable.
◉	○	Page 1 of process 1 needs a replacement

Kernel occupies two page frames: True
Handling this scenario demands two disk I/Os: True
Local replacement means chose any of the frames 2, 3, 6: True
Local replacement means chose any of the frame from 2 to 7: False
Global replacement means chose any of the frame from 0 to 7: False
Global replacement means chose any of the frame from 2 to 7: True
The kernel's pages can not used for replacement if kernel is not pageable.: True
Page 1 of process 1 needs a replacement: True

Question **8**

Complete

Mark 0.67 out of 1.00

Shared memory is possible with which of the following memory management schemes ?

Select one or more:

- [x] a. paging
- [x] b. segmentation
- [ ] c. continuous memory management
- [ ] d. demand paging

The correct answers are: paging, segmentation, demand paging

Question **9**

Complete

Mark 0.60 out of 1.00

Given below is the "maps" file for a particular instance of "vim.basic" process.

Mark the given statements as True or False, w.r.t. the contents of the map file.

```
55a43501b000-55a435049000 r--p 00000000 103:05 917529                    /usr/bin/vim.basic
55a435049000-55a435248000 r-xp 0002e000 103:05 917529                    /usr/bin/vim.basic
55a435248000-55a4352b6000 r--p 0022d000 103:05 917529                    /usr/bin/vim.basic
55a4352b7000-55a4352c5000 r--p 0029b000 103:05 917529                    /usr/bin/vim.basic
55a4352c5000-55a4352e2000 rw-p 002a9000 103:05 917529                    /usr/bin/vim.basic
55a4352e2000-55a4352f0000 rw-p 00000000 00:00 0
55a436bc9000-55a436e5b000 rw-p 00000000 00:00 0                          [heap]
7f275b0a3000-7f275b0a6000 r--p 00000000 103:05 917901                    /usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so
7f275b0a6000-7f275b0ad000 r-xp 00003000 103:05 917901                    /usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so
7f275b0ad000-7f275b0af000 r--p 0000a000 103:05 917901                    /usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so
7f275b0af000-7f275b0b0000 r--p 0000b000 103:05 917901                    /usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so
7f275b0b0000-7f275b0b1000 rw-p 0000c000 103:05 917901                    /usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so
7f275b0b1000-7f275b0b7000 rw-p 00000000 00:00 0
7f275b0b7000-7f275b8f5000 r--p 00000000 103:05 925247                    /usr/lib/locale/locale-archive
7f275b8f5000-7f275b8fa000 rw-p 00000000 00:00 0
7f275b8fa000-7f275b8fc000 r--p 00000000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b8fc000-7f275b901000 r-xp 00002000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b901000-7f275b904000 r--p 00007000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b904000-7f275b905000 ---p 0000a000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b905000-7f275b906000 r--p 0000a000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b906000-7f275b907000 rw-p 0000b000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b907000-7f275b90a000 r--p 00000000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b90a000-7f275b921000 r-xp 00003000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b921000-7f275b932000 r--p 0001a000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b932000-7f275b933000 ---p 0002b000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b933000-7f275b934000 r--p 0002b000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b934000-7f275b935000 rw-p 0002c000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b935000-7f275b937000 rw-p 00000000 00:00 0
7f275b937000-7f275b938000 r--p 00000000 103:05 917914                    /usr/lib/x86_64-linux-
gnu/libutil-2.31.so
7f275b938000-7f275b939000 r-xp 00001000 103:05 917914                    /usr/lib/x86_64-linux-
gnu/libutil-2.31.so
7f275b939000-7f275b93a000 r--p 00002000 103:05 917914                    /usr/lib/x86_64-linux-
gnu/libutil-2.31.so
7f275b93a000-7f275b93b000 r--p 00002000 103:05 917914                    /usr/lib/x86_64-linux-
gnu/libutil-2.31.so
7f275b93b000-7f275b93c000 rw-p 00003000 103:05 917914                    /usr/lib/x86_64-linux-
```

```
gnu/libutil-2.31.so
7f275b93c000-7f275b93e000 r--p 00000000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b93e000-7f275b94f000 r-xp 00002000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b94f000-7f275b955000 r--p 00013000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b955000-7f275b956000 ---p 00019000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b956000-7f275b957000 r--p 00019000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b957000-7f275b958000 rw-p 0001a000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b958000-7f275b95c000 r--p 00000000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b95c000-7f275b978000 r-xp 00004000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b978000-7f275b982000 r--p 00020000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b982000-7f275b983000 ---p 0002a000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b983000-7f275b985000 r--p 0002a000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b985000-7f275b986000 rw-p 0002c000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b986000-7f275b988000 r--p 00000000 103:05 924057                    /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b988000-7f275b98d000 r-xp 00002000 103:05 924057                    /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b98d000-7f275b98f000 r--p 00007000 103:05 924057                    /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b98f000-7f275b990000 r--p 00008000 103:05 924057                    /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b990000-7f275b991000 rw-p 00009000 103:05 924057                    /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b991000-7f275b995000 r--p 00000000 103:05 921934                    /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b995000-7f275b9a3000 r-xp 00004000 103:05 921934                    /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b9a3000-7f275b9a9000 r--p 00012000 103:05 921934                    /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b9a9000-7f275b9aa000 r--p 00017000 103:05 921934                    /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b9aa000-7f275b9ab000 rw-p 00018000 103:05 921934                    /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b9ab000-7f275b9ad000 rw-p 00000000 00:00 0
7f275b9ad000-7f275b9af000 r--p 00000000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9af000-7f275b9b4000 r-xp 00002000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b4000-7f275b9b5000 r--p 00007000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b5000-7f275b9b6000 ---p 00008000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b6000-7f275b9b7000 r--p 00008000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b7000-7f275b9b8000 rw-p 00009000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b8000-7f275b9ba000 r--p 00000000 103:05 924277                    /usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0
7f275b9ba000-7f275ba1e000 r-xp 00002000 103:05 924277                    /usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0
```

```
7f275ba1e000-7f275ba46000 r--p 00066000 103:05 924277                    /usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0
7f275ba46000-7f275ba47000 r--p 0008d000 103:05 924277                    /usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0
7f275ba47000-7f275ba48000 rw-p 0008e000 103:05 924277                    /usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0
7f275ba48000-7f275ba6d000 r--p 00000000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275ba6d000-7f275bbe5000 r-xp 00025000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275bbe5000-7f275bc2f000 r--p 0019d000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275bc2f000-7f275bc30000 ---p 001e7000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275bc30000-7f275bc33000 r--p 001e7000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275bc33000-7f275bc36000 rw-p 001ea000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275bc36000-7f275bc3a000 rw-p 00000000 00:00 0
7f275bc3a000-7f275bc41000 r--p 00000000 103:05 917906                    /usr/lib/x86_64-linux-
gnu/libpthread-2.31.so
7f275bc41000-7f275bc52000 r-xp 00007000 103:05 917906                    /usr/lib/x86_64-linux-
gnu/libpthread-2.31.so
7f275bc52000-7f275bc57000 r--p 00018000 103:05 917906                    /usr/lib/x86_64-linux-
gnu/libpthread-2.31.so
7f275bc57000-7f275bc58000 r--p 0001c000 103:05 917906                    /usr/lib/x86_64-linux-
gnu/libpthread-2.31.so
7f275bc58000-7f275bc59000 rw-p 0001d000 103:05 917906                    /usr/lib/x86_64-linux-
gnu/libpthread-2.31.so
7f275bc59000-7f275bc5d000 rw-p 00000000 00:00 0
7f275bc5d000-7f275bcce000 r--p 00000000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275bcce000-7f275bf29000 r-xp 00071000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275bf29000-7f275c142000 r--p 002cc000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275c142000-7f275c143000 ---p 004e5000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275c143000-7f275c149000 r--p 004e5000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275c149000-7f275c190000 rw-p 004eb000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275c190000-7f275c1b3000 rw-p 00000000 00:00 0
7f275c1b3000-7f275c1b4000 r--p 00000000 103:05 917894                    /usr/lib/x86_64-linux-
gnu/libdl-2.31.so
7f275c1b4000-7f275c1b6000 r-xp 00001000 103:05 917894                    /usr/lib/x86_64-linux-
gnu/libdl-2.31.so
7f275c1b6000-7f275c1b7000 r--p 00003000 103:05 917894                    /usr/lib/x86_64-linux-
gnu/libdl-2.31.so
7f275c1b7000-7f275c1b8000 r--p 00003000 103:05 917894                    /usr/lib/x86_64-linux-
gnu/libdl-2.31.so
7f275c1b8000-7f275c1b9000 rw-p 00004000 103:05 917894                    /usr/lib/x86_64-linux-
gnu/libdl-2.31.so
7f275c1b9000-7f275c1bb000 rw-p 00000000 00:00 0
7f275c1bb000-7f275c1c0000 r-xp 00000000 103:05 923815                    /usr/lib/x86_64-linux-
gnu/libgpm.so.2
7f275c1c0000-7f275c3bf000 ---p 00005000 103:05 923815                    /usr/lib/x86_64-linux-
gnu/libgpm.so.2
7f275c3bf000-7f275c3c0000 r--p 00004000 103:05 923815                    /usr/lib/x86_64-linux-
gnu/libgpm.so.2
7f275c3c0000-7f275c3c1000 rw-p 00005000 103:05 923815                    /usr/lib/x86_64-linux-
gnu/libgpm.so.2
```

```
7f275c3c1000-7f275c3c3000 r--p 00000000 103:05 923315                    /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3c3000-7f275c3c8000 r-xp 00002000 103:05 923315                    /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3c8000-7f275c3ca000 r--p 00007000 103:05 923315                    /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3ca000-7f275c3cb000 r--p 00008000 103:05 923315                    /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3cb000-7f275c3cc000 rw-p 00009000 103:05 923315                    /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3cc000-7f275c3cf000 r--p 00000000 103:05 923446                    /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3cf000-7f275c3d9000 r-xp 00003000 103:05 923446                    /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3d9000-7f275c3dd000 r--p 0000d000 103:05 923446                    /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3dd000-7f275c3de000 r--p 00010000 103:05 923446                    /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3de000-7f275c3df000 rw-p 00011000 103:05 923446                    /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3df000-7f275c3e5000 r--p 00000000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c3e5000-7f275c3fe000 r-xp 00006000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c3fe000-7f275c405000 r--p 0001f000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c405000-7f275c406000 ---p 00026000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c406000-7f275c407000 r--p 00026000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c407000-7f275c408000 rw-p 00027000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c408000-7f275c40a000 rw-p 00000000 00:00 0
7f275c40a000-7f275c418000 r--p 00000000 103:05 924540                    /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c418000-7f275c427000 r-xp 0000e000 103:05 924540                    /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c427000-7f275c435000 r--p 0001d000 103:05 924540                    /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c435000-7f275c439000 r--p 0002a000 103:05 924540                    /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c439000-7f275c43a000 rw-p 0002e000 103:05 924540                    /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c43a000-7f275c449000 r--p 00000000 103:05 917895                    /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c449000-7f275c4f0000 r-xp 0000f000 103:05 917895                    /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c4f0000-7f275c587000 r--p 000b6000 103:05 917895                    /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c587000-7f275c588000 r--p 0014c000 103:05 917895                    /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c588000-7f275c589000 rw-p 0014d000 103:05 917895                    /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c589000-7f275c58b000 rw-p 00000000 00:00 0
7f275c5ae000-7f275c5af000 r--p 00000000 103:05 917889                    /usr/lib/x86_64-linux-gnu/ld-
2.31.so
7f275c5af000-7f275c5d2000 r-xp 00001000 103:05 917889                    /usr/lib/x86_64-linux-gnu/ld-
2.31.so
7f275c5d2000-7f275c5da000 r--p 00024000 103:05 917889                    /usr/lib/x86_64-linux-gnu/ld-
2.31.so
7f275c5db000-7f275c5dc000 r--p 0002c000 103:05 917889                    /usr/lib/x86_64-linux-gnu/ld-
2.31.so
```

```
7f275c5dc000-7f275c5dd000 rw-p 0002d000 103:05 917889                    /usr/lib/x86_64-linux-gnu/ld-
2.31.so
7f275c5dd000-7f275c5de000 rw-p 00000000 00:00 0
7ffd22d2f000-7ffd22d50000 rw-p 00000000 00:00 0                          [stack]
7ffd22db0000-7ffd22db4000 r--p 00000000 00:00 0                          [vvar]
7ffd22db4000-7ffd22db6000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

True	False	
○	●	The size of the heap is one page
●	○	This is a virtual memory map (not physical memory map)
○	●	vim.basic uses the math library
●	○	The 5th entry 55a4352c5000-55a4352e2000 **may** correspond to "data" of the vim.basic
●	○	The size of the stack is one page

The size of the heap is one page: False
This is a virtual memory map (not physical memory map): True
vim.basic uses the math library: True
The 5th entry 55a4352c5000-55a4352e2000 **may** correspond to "data" of the vim.basic: True
The size of the stack is one page: False

Question **10**

Complete

Mark 0.00 out of 1.00

Select all the correct statements, w.r.t. Copy on Write

- [ ] a. Fork() used COW technique to improve performance of new process creation.
- [x] b. Vfork() assumes that there will be no write, but rather exec()
- [x] c. COW helps us save memory
- [ ] d. If either parent or child modifies a COW-page, then a copy of the page is made and page table entry is updated
- [ ] e. use of COW during fork() is useless if child called exit()
- [x] f. use of COW during fork() is useless if exec() is called by the child

The correct answers are: Fork() used COW technique to improve performance of new process creation., If either parent or child modifies a COW-page, then a copy of the page is made and page table entry is updated, COW helps us save memory, Vfork() assumes that there will be no write, but rather exec()

Question **11**

Complete

Mark 1.00 out of 1.00

Assuming a 8- KB page size, what is the page numbers for the address 1093943 reference in decimal :

(give answer also in decimal)

Answer: 134

The correct answer is: 134

Question **12**

Complete

Mark 0.00 out of 1.00

Order the following events, related to page fault handling, in correct order

1. Page fault handler detects that it's a page fault and not illegal memory access
2. Disk interrupt handler runs
3. MMU detects that a page table entry is marked "invalid"
4. Page faulted process is moved to ready-queue
5. Empty frame is found
6. Page fault interrupt is generated
7. Other processes scheduled by scheduler
8. Page table of page faulted process is updated
9. Disk Interrupt occurs
10. Disk read is issued
11. Page fault handler in kernel starts executing
12. Page faulting process is made to wait in a queue

The correct order for these items is as follows:

1. MMU detects that a page table entry is marked "invalid"
2. Page fault interrupt is generated
3. Page fault handler in kernel starts executing
4. Page fault handler detects that it's a page fault and not illegal memory access
5. Empty frame is found
6. Disk read is issued
7. Page faulting process is made to wait in a queue
8. Other processes scheduled by scheduler
9. Disk Interrupt occurs
10. Disk interrupt handler runs
11. Page table of page faulted process is updated
12. Page faulted process is moved to ready-queue

Question **13**

Complete

Mark 1.00 out of 1.00

Page sizes are a power of 2 because

Select one:

- a. Certain bits are reserved for offset in logical address. Hence page size = 2^(no.of offset bits)
- b. Power of 2 calculations are highly efficient
- c. Certain bits are reserved for offset in logical address. Hence page size = 2^(32 - no.of offset bits)
- d. operating system calculations happen using power of 2
- e. MMU only understands numbers that are power of 2

The correct answer is: Certain bits are reserved for offset in logical address. Hence page size = 2^(no.of offset bits)

Question **14**

Complete

Mark 1.00 out of 1.00

Given below is the output of the command "ps -eo min_flt,maj_flt,cmd" on a Linux Desktop system. Select the statements that are consistent with the output

```
626729 482768 /usr/lib/firefox/firefox -contentproc -parentBuildID 20220202182137 -prefsLen 9256 -prefMapSize 264738 -appDir /usr/lib/firefox/browser 6094 true rdd
2167 687 /usr/sbin/apache2 -k start
1265185 222 /usr/bin/gnome-shell
102648 111 /usr/sbin/mysqld
9813 0 bash
15497 370 /usr/bin/gedit --gapplication-service
```

- [x] a. All of the processes here exihibit some good locality of reference
- [x] b. Firefox has likely been running for a large amount of time
- [x] c. The bash shell is mostly busy doing work within a particular locality
- [x] d. Apache web-server has not been doing much work

The correct answers are: Firefox has likely been running for a large amount of time, Apache web-server has not been doing much work, The bash shell is mostly busy doing work within a particular locality, All of the processes here exihibit some good locality of reference

Question **15**

Complete

Mark 0.14 out of 1.00

Suppose two processes share a library between them. The library consists of 5 pages, and these 5 pages are mapped to frames 9, 15, 23, 4, 7 respectively. Process P1 has got 6 pages, first 3 of which consist of process's own code/data and 3 correspond to library's pages 0, 2, 4. Process P2 has got 7 pages, first 3 of which consist of processe's own code/data and remaining 4 correspond to library's pages 0, 1, 3, 4. Fill in the blanks for page table entries of P1 and P2.

Page table of P1, Page 5	23
Page table of P1, Page 3	9
Page table of P2, Page 0	6
Page table of P2, Page 3	7
Page table of P2, Page 4	9
Page table of P2, Page 1	5
Page table of P1, Page 4	9

The correct answer is: Page table of P1, Page 5 → 7, Page table of P1, Page 3 → 9, Page table of P2, Page 0 → 9, Page table of P2, Page 3 → 4, Page table of P2, Page 4 → 7, Page table of P2, Page 1 → 15, Page table of P1, Page 4 → 23

Question **16**

Complete

Mark 0.50 out of 1.00

For the reference string
3 4 3 5 2

the number of page faults (including initial ones) using

FIFO replacement and 2 page frames is :

4

FIFO replacement and 3 page frames is :

3

◄ (Code) mmap related programs

Jump to...

Points from Mid-term feedback ►

Started on	Monday, 7 March 2022, 7:00:12 PM
State	Finished
Completed on	Monday, 7 March 2022, 8:00:04 PM
Time taken	59 mins 52 secs
Grade	**9.78** out of 15.00 (**65**%)

Question **1**

Complete

Mark 1.00 out of 1.00

Why is there a call to kinit2? Why is it not merged with knit1?

- a. call to seginit() makes it possible to actually use PHYSTOP in argument to kinit2()
- b. When kinit1() is called there is a need for few page frames, but later knit2() is called to serve need of more page frames
- c. Because there is a limit on the values that the argumets to knit1() can take.
- d. knit2 refers to virtual addresses beyond 4MB, which are not mapped before kvalloc() is called

The correct answer is: knit2 refers to virtual addresses beyond 4MB, which are not mapped before kvalloc() is called

Question **2**

Complete

Mark 1.50 out of 1.50

Arrange the following in the correct order of execution (w.r.t. 'init')

initcode() returns in forkret()	6
'initcode' process is marked RUNNABLE	3
'initcode' struct proc is created	2
userinit() is called	1
mpmain() calls scheduler()	4
initcode() calls exec("/init", ...)	8
initcode() returns from trapret()	7
scheduler() schedules initcode() process	5

The correct answer is: initcode() returns in forkret() → 6, 'initcode' process is marked RUNNABLE → 3, 'initcode' struct proc is created → 2, userinit() is called → 1, mpmain() calls scheduler() → 4, initcode() calls exec("/init", ...) → 8, initcode() returns from trapret() → 7, scheduler() schedules initcode() process → 5

Question **3**

Complete

Mark 0.00 out of 2.00

exec() does this: curproc->tf->eip = elf.entry, but userinit() does this: p->tf->eip = 0; Select all the statements from below, that collectively explain this

- [x] a. the 'entry' in initcode is anyways 0
- [x] b. exec() loads from ELF file and the address of first instruction to be executed is given by 'entry'
- [ ] c. the initcode is created using objcopy, which discards all relocation information and symbols (like entry)
- [x] d. elf.entry is anyways 0, so both statements mean the same
- [x] e. In userinit() the function inituvm() has mapped the code of 'initcode' to be starting at virtual address 0
- [ ] f. the code of 'initcode' is loaded at physical address 0

The correct answers are: exec() loads from ELF file and the address of first instruction to be executed is given by 'entry', In userinit() the function inituvm() has mapped the code of 'initcode' to be starting at virtual address 0, the initcode is created using objcopy, which discards all relocation information and symbols (like entry)

Question **4**

Complete

Mark 1.00 out of 1.00

What does userinit() do ?

- a. initializes the users
- b. sets up the 'initcode' process to start execution in forkret ()
- c. sets up the 'initcode' process to start execution in forkret()
- d. sets up the 'initcode' process to start execution in trapret()
- e. sets up the 'init' process to start execution in forkret()
- f. initializes the process 'init' and starts executing it

The correct answer is: sets up the 'initcode' process to start execution in forkret()

Question **5**

Complete

Mark 0.67 out of 1.00

Which of the following is done by mappages()?

- [ ] a. allocate page directory if required
- [ ] b. allocate page frame if required
- [x] c. allocate page table if required
- [x] d. create page table mappings for the range given by "va" and "va + size"
- [ ] e. create page table mappings to the range given by "pa" and "pa + size"

The correct answers are: create page table mappings for the range given by "va" and "va + size", allocate page table if required, create page table mappings to the range given by "pa" and "pa + size"

Question **6**

Complete

Mark 0.00 out of 1.00

Select the statement that most correctly describes what setupkvm() does

- a. creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global arrray
- b. creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global arrray and makes kpgdir point to it
- c. creates a 1-level page table for the use by the kernel, as specified in kmap[] global array
- d. creates a 2-level page table for the use of the kernel, as specified in gdtdesc

The correct answer is: creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global arrray

Question **7**

Complete

Mark 0.00 out of 1.00

The approximate number of page frames created by kinit1 is

- a. 4
- b. 16
- c. 4000
- d. 3000
- e. 2000
- f. 1000
- g. 10

The correct answer is: 3000

Question **8**

Complete

Mark 1.20 out of 1.50

Which of the following is DONE by allocproc() ?

- [ ] a. setup kernel memory mappings for the process
- [x] b. Select an UNUSED struct proc for use
- [ ] c. ensure that the process starts in trapret()
- [x] d. allocate kernel stack for the process
- [ ] e. allocate PID to the process
- [x] f. ensure that the process starts in forkret()
- [x] g. setup the trapframe and context pointers appropriately
- [ ] h. setup the contents of the trapframe of the process properly

The correct answers are: Select an UNUSED struct proc for use, allocate PID to the process, allocate kernel stack for the process, setup the trapframe and context pointers appropriately, ensure that the process starts in forkret()

Question **9**

Complete

Mark 1.00 out of 1.00

Map the virtual address to physical address in xv6

KERNLINK	0x100000
0xFE000000	0xFE000000
80108000	0x108000
KERNBASE	0

The correct answer is: KERNLINK → 0x100000, 0xFE000000 → 0xFE000000, 80108000 → 0x108000, KERNBASE → 0

Question **10**

Complete

Mark 0.42 out of 1.00

Select all the correct statements about initcode

- [ ] a. code of initcode is loaded at virtual address 0
- [x] b. The data and stack of initcode is mapped to one single page in userinit()
- [ ] c. code of 'initcode' is loaded along with the kernel during booting
- [x] d. the size of 'initcode' is 2c
- [ ] e. code of initcode is loaded in memory by the kernel during userinit()
- [x] f. initcode is the 'init' process
- [x] g. initcode essentially calls exec("/init",...)

The correct answers are: code of 'initcode' is loaded along with the kernel during booting, the size of 'initcode' is 2c, The data and stack of initcode is mapped to one single page in userinit(), initcode essentially calls exec("/init",...)

Question **11**

Complete

Mark 1.00 out of 1.00

The variable 'end' used as argument to kinit1 has the value

- (●) a. 801154a8
- ( ) b. 81000000
- ( ) c. 80110000
- ( ) d. 80102da0
- ( ) e. 8010a48c
- ( ) f. 80000000

The correct answer is: 801154a8

Question **12**

Complete

Mark 1.00 out of 1.00

What does seginit() do?

- ( ) a. Nothing significant, just repetition of earlier GDT setup but with 2-level paging setup done
- ( ) b. Nothing significant, just repetition of earlier GDT setup but with free frames list created now
- ( ) c. Adds two additional entries to GDT corresponding to Code and Data segments, but to be used in privilege level 0
- (●) d. Adds two additional entries to GDT corresponding to Code and Data segments, but to be used in privilege level 3
- ( ) e. Nothing significant, just repetition of earlier GDT setup but with kernel page table allocated now

The correct answer is: Adds two additional entries to GDT corresponding to Code and Data segments, but to be used in privilege level 3

Question **13**

Complete

Mark 1.00 out of 1.00

Does exec() code around clearptau() lead to wastage of one page frame?

- a. no
- b. yes

The correct answer is: yes

◄ Questions for test on kalloc/kfree/kvmalloc, etc.

Jump to...

(Optional Assignment) Slab allocator in xv6 ►

Memory Management Basics

Summary

- Understanding how the processor architecture drives the memory management features of OS and system programs (compilers, linkers)
- Understanding how different hardware designs lead to different memory management schemes by operating systems

Addresses issued by CPU

- During the entire ‘on’ time of the CPU
  - Addresses are “issued” by the CPU on address bus
  - One address to fetch instruction from location specified by PC
  - Zero or more addresses depending on instruction
    - e.g. mov $0x300, r1 # move contents of address 0x300 to r1 --> one extra address issued on address bus

Memory layout of a C program

$ size /bin/ls

text	data	bss	dec	hex	filename
128069	4688	4824	137581	2196d	/bin/ls

Desired from a multi-tasking system

- Multiple processes in RAM at the same time (multi-programming)
- Processes should not be able to see/touch each other’s code, data (globals), stack, heap, etc.
- Further advanced requirements
  - Process could reside anywhere in RAM
  - Process need not be continuous in RAM
  - Parts of process could be moved anywhere in RAM

Different ‘times’

- Different actions related to memory management for a program are taken at different times. So let’s know the different ‘times’
- Compile time
  - When compiler is compiling your C code
- Load time
  - When you execute “./myprogram” and it’s getting loaded in RAM by loader i.e. exec()
- Run time
  - When the process is alive, and getting scheduled by the OS

Different types of Address binding

Which binding is actually used, is mandated by processor features + OS

- Compile time address binding
  - Address of code/variables is fixed by compiler
  - Very rigid scheme
  - Location of process in RAM can not be changed ! Non-relocatable code.
- Load time address binding
  - Address of code/variables is fixed by loader
  - Location of process in RAM is decided at load time, but can’t be changed later
  - Flexible scheme, relocatable code
- Run time address binding
  - Address of code/variables is fixed at the time of executing the code
  - Very flexible scheme , highly relocatable code
  - Location of process in RAM is decided at load time, but CAN be changed later also

Simplest case

- Suppose the address issued by CPU reaches the RAM controller directly

Simplest case

- How does this impact the compiler and OS ?
- When a process is running the addresses issued by it, will reach the RAM directly
- So exact addresses of globals, addresses in “jmp” and “call” must be part the machine instructions generated by compiler
  - How will the compiler know the addresses, at “compile time” ?

Example
address: instruction

1000: mov $0x300, r1
1004: add r1, -3
1008: jnz 1000

Sequence of addressed sent by CPU: 1000, 0x300, 1004, 1008, 1000, 0x300, ...

Simplest case

Example
1000: mov $0x300, r1
1004: add r1, -3
1008: jnz 1000

- Solution: compiler assumes some fixed addresses for globals, code, etc.
- OS loads the program exactly at the same addresses specified in the executable file. **Non-relocatable code.**
- Now program can execute properly.

Simplest case

Example
1000: mov $0x300, r1
1004: add r1, -3
1008: jnz 1000

- Problem with this solution
  - Programs once loaded in RAM must stay there, can't be moved
  - What about 2 programs?
    - Compilers being "programs", will make same assumptions and are likely to generate same/overlapping addresses for two different programs
    - Hence only one program can be in memory at a time !
    - No need to check for any memory boundary violations – all memory belongs to one process
- Example: DOS

Base/Relocation + Limit scheme

Figure 9.6 Hardware support for relocation and limit registers.

- Base and Limit are two registers inside CPU's Memory Management Unit
- 'base' is added to the address generated by CPU
- The result is compared with base+limit and if less passed to memory, else hardware interrupt is raised

Memory Management Unit (MMU)

Figure 9.4 Memory management unit (MMU).

- Is part of the CPU chip, acts on every memory address issue by execution unit of the CPU
- In the scheme just discussed, the base, limit calculation parts are part of MMU

Base/Relocation + Limit scheme

- Compiler's work
  - Assume that the process is one continous chunk in memory, with a size limit
  - Assume that the process starts at address zero (!) and calculate addresses for globals, code, etc. And accordingly generate machine code

Base/Relocation + Limit scheme

Figure 9.6 Hardware support for relocation and limit registers.

- OS's work
  - While loading the process in memory – must load as one continous segment
  - Fill in the 'base' register with the actual address of the process in RAM.
  - Setup the limit to be the size of the process as set by compiler in the executable file. *Remember the base+limit in OS's own data structures.*

Base/Relocation + Limit scheme

Figure 9.6 Hardware support for relocation and limit registers.

- Combined effect
  - **"Relocatable code"** – the process can go anywhere in RAM at the time of loading
  - Some memory violations can be detected – a memory access beyond base+limit will raise interrupt, thus running OS in turn, which may take action against the process

Example scenario of memory in base+limit scheme

It should be possible to have relocatable code even with “simplest case”

By doing extra work during “loading”.

How?

Next scheme: Segmentation Multiple base +limit pairs

- Multiple sets of base + limit registers
- Whenever an address is issued by execution unit of CPU, it will also include reference to some base register
  - And hence limit register paired to that base register will be used for error checking
- Compiler: can assume a separate chunk of memory for code, data, stack, heap, etc. And accordingly calculate addresses . Each “segment” starting at address 0.
- OS: will load the different ‘sections’ in different memory regions and accordingly set different ‘base’ registers
-

Next scheme: Multiple base +limit pairs

Next scheme: Multiple base +limit pairs, with further indirection

- Base + limit pairs can also be stored in some memory location (not in registers). Question: how will the cpu know where it’s in memory?
  - One CPU register to point to the location of table in memory
- Segment registers still in use, they give an index in this table
- This is x86 segmentation
  - Flexibility to have lot more “base+limits” in the array/table in memory

Next scheme:
Multiple base +limit pairs, with further indirection

Figure 9.22 IA-32 segmentation.

Problems with segmentation schemes

- OS needs to find a continuous free chunk of memory that fits the size of the “segment”
  - If not available, your exec() can fail due to lack of memory
- Suppose 50k is needed
  - Possible that mong 3 free chunks total 100K may be available, but no single chunk of 50k!
  - **External fragmentation**
- Solution to external fragmentation: **compaction** – move the chunks around and make a continous big chunk available. Time consuming, tricky.

Solving external fragmentation problem

- Process should not be continous in memory!
- Divide the continuous process image in smaller chunks (let’s say 4k each) and locate the chunks anywhere in the physical memory
  - Need a way to map the *logical* memory addresses into *actual physical memory addresses*

Figure 9.9 Paging model of logical and physical memory.

Figure 9.10 Paging example for a 32-byte memory with 4-byte pages.

Figure 9.8 Paging hardware.

Paging

- Process is assumed to be composed of equally sized “pages” (e.g. 4k page)
- Actual memory is considered to be divided into page “frames”.
- CPU generated logical address is split into a page number and offset
- A page table base register inside CPU will give location of an in memory table called page table
- Page number used as offset in a table called page table, which gives the physical page frame number
- Frame number + offset are combined to get physical memory address

Paging

- Compiler: assume the process to be one continous chunk of memory (!) . Generate addresses accordingly
- OS: at exec() time – allocate different frames to process, allocate a page table(!), setup the page table to map page numbers with frame numbers, setup the page table base register, start the process
- Now hardware will take care of all translations of logical addresses to physical addresses

X86 memory management

Figure 9.21 Logical to physical address translation in IA-32.

Segmentation in x86

Figure 9.22 IA-32 segmentation.

- The selector is automatically chosen using Code Segment (CS) register, or Data Segment (DS) register depending on which type of memory address is being fetched
- Descriptor table is in memory
- The location of Descriptor table (Global DT- GDT or Local DT – LDT) is given by a GDT-register i.e. GDTR or LDT-register i.e. LDTR
-

Paging in x86

Figure 9.23 Paging in the IA-32 architecture.

- Depending on a flag setup in CR3 register, either 4 MB or 4 KB pages can be enabled
- Page directory, page table are both in memory

XV6 bootloader

Abhijit A. M.
abhijit.comp@coep.ac.in

Credits:
xv6 book by Cox, Kaashoek, Morris
Notes by Prof. Sorav Bansal

A word of caution

- We begin reading xv6 code
- But it’s not possible to read this code in a “linear fashion”
  - The dependency between knowing OS concepts and reading/writing a kernel that is written using all concepts

What we have seen ....

- Compilation process, calling conventions
- Basics of Memory Management by OS
- Basics of x86 architecture
  - Registers, segments, memory management unit, addressing, some basic machine instructions,
- ELF files
  - Objdump, program headers
  - Symbol tables

Boot-process

- **Bootloader itself**
  - **Is loaded by the BIOS at a fixed location in memory and BIOS makes it run**
  - **Our job, as OS programmers, is to write the bootloader code**
- **Bootloader does**
  - **Pick up code of OS from a ‘known’ location and loads it in memory**
  - **Makes the OS run**
- **Xv6 bootloader: bootasm.S bootmain.c (see Makefile)**

bootloader

- BIOS Runs (automatically)
- Loads boot sector into RAM at 0x7c00
- Starts executing that code
  - Make sure that your bootloader is loaded at 0x7c00
  - Makefile has

```
bootblock:  bootblock.S bootmain.c
    $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S .....
    ...
    $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o
    bootasm.o bootmain.o
```

Resuls in:

```
00007c00 <start>:  in bootblock.asm
```

Processor starts in real mode

- **Processor starts in real mode – works like 16 bit 8088**
- **eight 16-bit general-purpose registers,**
- **Segment registers %cs, %ds, %es, and %ss --> additional bits necessary to generate 20-bit memory addresses from 16-bit registers.**

  addr = seg << 4 + addr

Effective memory translation in the beginning
At _start in bootasm.S:

%cs=0 %ip=7c00.

So effective address = 0*16+ip = ip

bootloader

- First instruction is ‘cli’
  - disable interrupts
- So that until your code loads all hardware interrupt handlers, no interrupt will occur

Zeroing registers

```
# Zero data segment registers DS, ES, and SS.
xorw    %ax,%ax          # Set %ax to zero
movw    %ax,%ds          # -> Data Segment
movw    %ax,%es          # -> Extra Segment
movw    %ax,%ss          # -> Stack Segment
```

- zero ax and ds, es, ss
- BIOS did not put in anything perhaps

A not so necessary detail Enable 21 bit address

```
seta20.1:
  inb     $0x64,%al            # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1
  movb    $0xd1,%al            # 0xd1 -> port
0x64
  outb    %al,$0x64
seta20.2:
  inb     $0x64,%al            # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2
  movb    $0xdf,%al            # 0xdf -> port 0x60
  outb    %al,$0x60
```

- **Seg:off with 16 bit segments can actually address more than 20 bits of memory. After 0x100000 (=2^20), 8086 wrapped addresses to 0.**
- **80286 introduced 21st bit of address. But older software required 20 bits only. BIOS disabled 21st bit. Some OS needed 21st Bit. So enable it.**
- **Write to Port 0x64 and 0x60 -> keyboard controller**
  - **to enable 21st bit out of address translation**
  - **Why? Before the A20, i.e. 21st bit was introduced, it belonged to keyboard controller**
  - **For more details see https://en.wikipedia.org/wiki/A20_line**

    https://en.wikipedia.org/wiki/A20_line

After this

Some instructions are run to enter protected mode

And further code runs in protected mode

Real mode Vs protected mode

- Real mode 16 bit registers
- Protected mode
  - Enables segmentation + Paging both
    - No longer seg*16+offset calculations
    - Segment registers is index into segment descriptor table. But segment:offset pairs continue
      - mov %esp, $32 # SS will be used with esp
    - More in next few slides
    - Other segment registers need to be explicitely mentioned in instructions
      - Mov FS:$200, 30
  - 32 bit registers
    - can address upto 2^{32} memory
    - Can do arithmetic in 32 bits

X86 address : protected mode address translation

Both Segmentation and Paging are used in x86
X86 allows optionally one-level or two-level paging
Segmentation is a must to setup, paging is optional (needs to be enabled)
Hence different OS can use segmentation+paging in different ways

X86 segmentation

Figure 8.22 IA-32 segmentation.

Paging concept, hierarchical paging

Figure 9.8 Paging hardware.

Figure 9.16 Address translation for a two-level 32-bit paging architecture.

X86 paging

Figure 8.23 Paging in the IA-32 architecture.

Page Directory Entry (PDE)
Page Table Entry (PTE)

PDE

PTE

P	Present
W	Writable
U	User
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use

CR3

PWT Page-level writes transparent

PCT Page-level cache disable

CR4

VME	Virtual-8086 mode extensions	MCE	Machine check enable
PVI	Protected-mode virtual interrupts	PGE	Page-global enable
TSD	Time stamp disable	PCE	Performance counter enable
DE	Debugging extensions	OSFXSR	OS FXSAVE/FXRSTOR support
PSE	Page size extensions	OSXMM-EXCPT	OS unmasked exception support
PAE	Physical-address extension		

Segmentation + Paging

Selector value is implicit based on address being accessed.
Instruction: CS
Stack Variable: SS
Data: DS
etc.

Segmentation + Paging setup of xv6

- xv6 configures the segmentation hardware by setting Base = 0, Limit = 4 GB
  - translate logical to linear addresses without change, so that they are always equal.
  - Segmentation is pratically off
- Once paging is enabled, the only interesting address mapping in the system will be linear to physical.
  - In xv6 paging is NOT enabled while loading kernel
  - After kernel is loaded 4 MB pages are used for a while
  - Later the kernel switches to 4 kB pages!

GDT Entry

31 … 16			15 … 0	
Base 0:15			Limit 0:15	
63 … 56	55 … 52	51 … 48	47 … 40	39 … 32
Base 24:31	Flags	Limit 16:19	Access Byte	Base 16:23

asm.h

```
#define SEG_ASM(type,base,lim)                                  \
        .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);      \
        .byte (((base) >> 16) & 0xff), (0x90 | (type)),         \
                (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
```

Segment selector

TI Table index (0=GDT, 1=LDT)
RPL Requester privilege level

Note in 16 bit mode, segment selector is 16 bit, here it's 13 bit + 3 bits

EFLAGS register

Status flags
- CF Carry flag
- PF Parity flag
- AF Auxiliary carry flag
- ZF Zero flag
- SF Sign flag
- OF Overflow flag

Control flags
- DF Direction flag

System flags
- TF Trap flag
- IF Interrupt enable flag
- IOPL I/O privilege level
- NT Nested task
- RF Resume flag
- VM Virtual-8086 mode
- AC Alignment check
- VIF Virtual intr. flag
- VIP Virtual intr. pending
- ID ID flag

lgdt

```
lgdt    gdtdesc
...
# Bootstrap GDT
.p2align 2   # force 4 byte alignment
gdt:
  SEG_NULLASM     # null seg
  SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
  SEG_ASM(STA_W, 0x0, 0xffffffff)

   # data seg

gdtdesc:
  .word   (gdtdesc - gdt - 1)
# sizeof(gdt) - 1
  .long   gdt
```

- load the processor's (GDT) register with the value gdtdesc which points to the table gdt.
- table gdt : The table has a null entry, one entry for executable code, and one entry to data.
- all segments have a base address of zero and the maximum possible limit
- The code segment descriptor has a flag set that indicates that the code should run in 32-bit mode
- With this setup, when the boot loader enters protected mode, logical addresses map one-to-one to physical addresses.
- At gdtdesc we have this data

  2, <4 byte addr of gdt>

  Total 6 bytes

  - GDTR is : <address 4 byte>, <table limit 2 byte>
  - So lgdt gdtdesc loads these two values in GDTR

bootasm.S after “lgdt gdtdesc” till jump to “entry”

Base	Limit	Permissions
0	4GB	Write
0	4GB	Read, Execute
0	0	0

GDT

Still
Logical Address = Physical address!

But with GDT in picture and
Protected Mode operation

During this time,

Loading kernel from ELF into physical memory

Addresses in “kernel” file translate to same physical address!

Prepare to enable protected mode

- Prepare to enable protected mode by setting the 1 bit (CR0_PE) in register %cr0

```
movl    %cr0, %eax
orl     $CR0_PE, %eax
movl    %eax, %cr0
```

CR0

PE	Protection enabled	ET	Extension type	NW	Not write-through
MP	Monitor coprocessor	NE	Numeric error	CD	Cache disable
EM	Emulation	WP	Write protect	PG	Paging
TS	Task switched	AM	Alignment mask		

PG: Paging enabled or not **WP: Write protecion on/off**

PE: Protection Enabled --> protected mode.

Complete transition to 32 bit mode

```
ljmp   $(SEG_KCODE<<3), $start32
```

Complete the transition to 32-bit protected mode by using a long jmp

to reload %cs (=1) and %eip (=start32).

Note that ‘start32’ is the address of next instruction after ljmp.

Note: The segment descriptors are set up with no translation (that is 0-4GB setting), so that the mapping is still the identity mapping.

Jumping to “C” code

```
 movw    $(SEG_KDATA<<3), %ax    # Our data
segment selector
  movw    %ax, %ds                # -> DS: Data
Segment
  movw    %ax, %es                # -> ES: Extra
Segment
  movw    %ax, %ss                # -> SS: Stack
Segment
  movw    $0, %ax                 # Zero segments
not ready for use
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS

  # Set up the stack pointer and call into C.
  movl    $start, %esp
  call    bootmain
```

- Setup Data, extra, stack segment with SEG_KDATA (=2), FS & GS (=0)
- Copy “$start” i.e. 7c00 to stack-ptr
  - It will grow from 7c00 to 0000
- Call bootmain() a C function
  - In bootmain.c

Setup now

Logical Address = offset
Physical Address
CS, SS, etc.
Selector
3
2
1
0
DS
SS
CS
Base
Limit
Permissions
0
4GB
Write
0
4GB
Read, Execute
0
0
0
GDT

bootmain(): already in memory, as part of ‘bootblock’

- bootmain.c , expects to find a copy of the kernel executable on the disk starting at the second sector (sector = 1).
  - Why?
- The kernel is an ELF format binary
- Bootmain loads the first 4096 bytes of the ELF binary. It places the in-memory copy at address 0x10000
- readseg() is a function that runs OUT instructions in particular IO ports, to issue commands to read from Disk

```
void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* pa;

  elf = (struct elfhdr*)0x10000;  // scratch space

  // Read 1st page off disk
  readseg((uchar*)elf, 4096, 0);
```

bootmain()

- Check if it's really ELF or not
- Next load kernel code from ELF file "kernel" into memory

```
// Is this an ELF executable?
if(elf->magic != ELF_MAGIC)
    return;  // let bootasm.S handle error
```

ELF

```
struct elfhdr {
  uint magic;  // must equal ELF_MAGIC
  uchar elf[12];
  ushort type;
  ushort machine;
  uint version;
  uint entry;
  uint phoff; // where is program header table
  uint shoff;
  uint flags;
  ushort ehsize;
  ushort phentsize;
  ushort phnum;  // no. Of program header entries
  ushort shentsize;
  ushort shnum;
  ushort shstrndx;
};
```

ELF

```
// Program  header
struct proghdr {
  uint type; // Loadable segment , Dynamic linking information , Interpreter information , Thread-Local Storage template , etc.
  uint off; //Offset of the segment in the file image.
  uint vaddr; //Virtual address of the segment in memory.
  uint paddr; // physical address to load this program, if PA is relevant
  uint filesz; //Size in bytes of the segment in the file image.
  uint memsz; //Size in bytes of the segment in memory. May be 0.
  uint flags;
  uint align;
};
```

Run ‘objdump -x -a kernel | head -15’ & see this

kernel: file format elf32-i386
kernel
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c

Code to be loaded at KERNBASE + KERNLINK

Diff between memsz & filesz, will be filled with zeroes in memory

Program Header:
LOAD **off** 0x00001000 **vaddr** 0x80100000 **paddr** 0x00100000 align 2**12
filesz 0x0000a516 **memsz** 0x000154a8 **flags** rwx
STACK **off** 0x00000000 **vaddr** 0x00000000 **paddr** 0x00000000 align 2**4
filesz 0x00000000 **memsz** 0x00000000 **flags** rwx

Stack : everything zeroes

Load code from ELF to memory

```
  // Load each program segment (ignores ph flags).
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
  // Abhijit: number of program headers
  for(; ph < eph; ph++){
    // Abhijit: iterate over each program header
    pa = (uchar*)ph->paddr;
    // Abhijit: the physical address to load program
    /* Abhijit: read ph->filesz bytes, into 'pa',
       from ph->off in kernel/disk */
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
      stosb(pa + ph->filesz, 0, ph->memsz - ph-
>filesz); // Zero the reminder section*/
}
```

Jump to Entry

```
// Call the entry point from the ELF header.
// Does not return!
/* Abhijit:
 * elf->entry was set by Linker using kernel.ld
 * This is address 0x80100000 specified in
kernel.ld
 * See kernel.asm for kernel assembly code).
 */
entry = (void(*)(void))(elf->entry);
entry();
```

Figure 8.23 Paging in the IA-32 architecture.

To understand further code

Remember: 4 MB pages are possible

From entry:
Till: inside main(), before kvmalloc()

From entry:
Till: inside main(), before kvmalloc()

Even now, every Logical address = Physical address, but through Page dir

entrypgdir in main.c, is used by entry()

```
#define PTE_P        0x001   // Present
#define PTE_W        0x002   // Writeable
#define PTE_U        0x004   // User
#define PTE_PS       0x080   // Page Size
#define PDXSHIFT     22      // offset of
PDX in a linear address
```

```
__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {

  // Map VA's [0, 4MB) to PA's [0, 4MB)
  [0] = (0) | PTE_P | PTE_W | PTE_PS,

  // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB). This is entry 512
  [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

This is entry page directory during entry(), beginning of kernel
Mapping 0:0x400000 (i.e. 0: 4MB) to physical addresses 0:0x400000. is required as long as entry is executing at low addresses, but will eventually be removed.
This mapping restricts the kernel instructions and data to 4 Mbytes.

entry() in entry.S

```
entry:
  movl    %cr4, %eax
  orl     $(CR4_PSE), %eax
  movl    %eax, %cr4
  movl    $(V2P_WO(entrypgdir)), %eax
  movl    %eax, %cr3
  movl    %cr0, %eax
  orl     $(CR0_PG|CR0_WP), %eax
  movl    %eax, %cr0
  movl $(stack + KSTACKSIZE), %esp
  mov $main, %eax
  jmp *%eax
```

- # Turn on page size extension for 4Mbyte pages
- # Set page directory. 4 MB pages (temporarily only. More later)
- # Turn on paging.
- # Set up the stack pointer.
- # Jump to main(), and switch to executing at high addresses. The indirect call is needed because the assembler produces a PC-relative instruction for a direct jump.
-

More about entry()

```
movl    $(V2P_WO(entrypgdir)), %eax
movl    %eax, %cr3
```

-> Here we use physical address using V2P_WO because paging is not turned on yet

- **V2P is simple: substract 0x80000000 i.e. KERNBASE from address**

More about entry()

```
movl    %cr0, %eax
orl     $(CR0_PG|CR0_WP), %eax
movl    %eax, %cr0
```

This turns on paging

After this also, entry() is running and processor is executing code at lower addresses

- But we have already set 0'th entry in pgdir to address 0
- So it still works!

entry()

```
  movl $(stack + KSTACKSIZE), %esp

  mov $main, %eax

  jmp *%eax

.comm stack, KSTACKSIZE

# Abhijit: allocate here 'stack' of size = KSTACKSIZE
```

- # Set up the stack pointer.
- # Abhijit: +KSTACKSIZE is done as stack grows downwards
- # Jump to main(), and switch to executing at high addresses. The indirect call is needed because the assembler produces a PC-relative instruction for a direct jump.

bootmasm.S bootmain.c: Steps

1) Starts in “real” mode, 16 bit mode. Does some misc legacy work.
2) Runs instructions to do MMU set-up for protected-mode & only segmentation (0-4GB, identity mapping), changes to protected mode.
3) Reads kernel ELF file and loads it in RAM, as per instructions in ELF file
4) Sets up paging (4 MB pages)
5) Runs main() of kernel

Code from bootasm.S bootmain.c is over!
Kernel is loaded.
Now kernel is going to prepare itself

Processes

Abhijit A M
abhijit.comp@coep.ac.in

Process related data structures in kernel code

- Kernel needs to maintain following types of data structures for managing processes
  - List of all processes
  - Memory management details for each, files opened by each etc.
  - Scheduling information about the process
  - Status of the process
  - List of processes “waiting” for different events to occur,
  - Etc.

Figure 3.3 Process control block (PCB).

Process Control Block

- A record representing a process in operating system's data structures
- OS maintains a "list"of PCBs, one for each process
- Called "`struct task_struct`" in Linux kernel code and "`struct proc`" in xv6 code

Figure 3.3 Process control block (PCB).

Fields in PCB

- Process ID (PID)
- Process State
- Program counter
- Registers
- Memory limits of the process
- Accounting information
- I/O status
- Scheduling information
- array of file descriptors (list of open files)
- ...etc

List of open files

Program did

```
rfd = open("/etc/passwd", O_RDONLY)
```

List of open files

- The PCB contains an array of pointers, called file descriptor array (fd[ ]), pointers to structures representing files
- When open() system call is made
  - A new file structure is created and relevant information is stored in it
  - Smallest available of fd [ ] pointers is made to point to this new struct file
  - The index of this fd [ ] pointer is returned by open
- When subsequent calls are made to read(fd, ...) or write(fd, ...) , etc.
  - The kernel gets the “fd” as an index in the fd[ ] array and is able to locate the file structure for that file

```
// XV6 Code : Per-process state
enum procstate { UNUSED, EMBRYO, SLEEPING,
RUNNABLE, RUNNING, ZOMBIE };

struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this
process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};

struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

```
struct file {
  enum { FD_NONE,
FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

Process Queues/Lists inside OS

- Different types of queues/lists can be maintained by OS for the processes
  - A queue of processes which need to be scheduled
  - A queue of processes which have requested input/output to a device and hence need to be put on hold/wait
  - List of processes currently running on multiple CPUs
  - Etc.

```
// Linux data structure
struct task_struct {
    long state;/*state of the process */
    struct sched_entity se; /* scheduling information */
    struct task_struct *parent; /*this process's parent */
    struct list_head children; /*this process's children */
    struct files_struct *files; /* list of open files */
    struct mm_struct *mm;/*address space */
```

```
struct list_head {
    struct list_head
*next, *prev;
};
```

Figure 3.4 The ready queue and wait queues.

Figure 3.5 Queueing-diagram representation of process scheduling.

Figure 3.2 Diagram of process state.

Conceptual diagram

“Giving up” CPU by a process or blocking

```
int main() {
i = j + k;
scanf("%d", &k);
}
int scanf(char *x, ...) {
...
read(0, ..., ...);
}
int read(int fd, char *buf, int len) {
...
__asm__ { "int 0x80..."}
...
}
```

OS Syscall

```
sys_read(int fd, char *buf, int len) {
  file f  = current->fdarray[fd];
  int offset = f->position;
  ...
  disk_read(... , offset, ...);
  // Do what now?
  //asynchronous read
  //Interrupt will occur when the disk read is complete
  // Move the process from ready queue to a wait queue and call scheduler!
  // This is called "blocking"
  Return the data read ;
}
disk_read(...., offset, .... ) {
__asm__("outb PORT ..");
return;
}
```

“Giving up” CPU by a process or blocking

The relevant code in xv6 is in

Sleep()

The wakeup code is in wakeup() and wakeup1()

To be seen later

Context Switch

- Context
  - Execution context of a process
  - CPU registers, process state, memory management information, all configurations of the CPU that are specific to execution of a process/kernel
- Context Switch
  - Change the context from one process/OS to OS/another process
  - Need to save the old context and load new context
  - Where to save? --> PCB of the process

Context Switch

- Is an overhead
- No useful work happening while doing a context switch
- Time can vary from hardware to hardware
- Special instructions may be available to save a set of registers in one go

Figure 3.6 Diagram showing context switch from process to process.

Pecularity of context switch

- When a process is running, the function calls work in LIFO fashion
  - Made possible due to calling convention
- When an interrupt occurs
  - It can occur anytime
  - Context switch can happen in the middle of execution of any function
- After context switch
  - One process takes place of another
  - This “switch” is obviously not going to happen using calling convention, as no “call” is happening
  - Code for context switch must be in assembly!

NEXT: XV6 code overview

1. Understanding how traps are handled
2. How timer interrupt goes to scheduler
3. How scheduling takes place
4. How a “blocking” system call (e.g. read()) “blocks”

Inter Process Communication

Revision of process related concepts

- PCB, struct proc
- Process lifecycle – different states
- Memory layout
- Memory management
- Interrupts handling, system call handling, code from xv6
- Scheduler, code of scheduler in xv6

IPC: Inter Process Communication

- Processes within a system may be independent or cooperating
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing, e.g. copy paste
  - Computation speedup, e.g. matrix multiplication
  - Modularity, e.g. chrome – separate process for display, separate for fetching data
  - Convenience,
- Cooperating processes need interprocess communication (IPC)
- Two models of IPC
  - Shared memory
  - Message passing

Shared Memory Vs Message Passing

Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

Each requires OS to provide system calls for

- **Creating the IPC mechanism**
- **To read/write using the IPC mechanism**
- **Delete the IPC mechanism**

Note: processes communicating with each other with the help of OS!

Example of co-operating processes: Producer Consumer Problem

- **Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process**
  - **unbounded-buffer places no practical limit on the size of the buffer**
  - **bounded-buffer assumes that there is a fixed buffer size**

Example of co-operating processes: Producer Consumer Problem

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Can only use BUFFER_SIZE-1 elements

Example of co-operating processes: Producer Consumer Problem

- **Code of Producer**

```
while (true) {
      /* Produce an item */
    while (((in = (in + 1) % BUFFER SIZE count)  == out)
     ;   /* do nothing -- no free buffers */
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
}
```

Example of co-operating processes: Producer Consumer Problem

- **Code of Consumer**

```
while (true) {
        while (in == out)
                ; // do nothing -- nothing to consume

      // remove an item from the buffer
      item = buffer[out];
      out = (out + 1) % BUFFER SIZE;
      return item;
 }
```

Pipes

Pipes for IPC

- **Two types**
  - **Unnamed Pipes or ordinary pipes**
  - **Named Pipe**

Ordinary pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the write-end of the pipe)
- Consumer reads from the other end (the read-end of the pipe)
- Ordinary pipes are therefore unidirectional
- Requires a parent-child (or sibling, etc) kind of relationship between communicating processes

parent
fd(0)
fd(1)
child
fd(0)
fd(1)
pipe

Named pipes

- Also called FIFO
- Processes can create a “file” that acts as pipe. Multiple processes can share the file to read/write as a FIFO
- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems
- Is not deleted automatically by OS

Named pipes

- int mkfifo(const char *pathname, mode_t mode);
- Example

Shared Memory

System V shared memory

- Process first creates shared memory segment

```
segment id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

- Process wanting access to that shared memory must attach to it

```
shared_memory = (char *) shmat(id, NULL, 0);
```

- Now the process could write to the shared memory

```
sprintf(shared_memory, "Writing to shared memory");
```

- When done, a process can detach the shared memory from its address space

```
shmdt(shared_memory);
```

Example of Shared memory
POSIX Shared Memory

- **What is POSIX?**
  - **Portable Operating System Interface (POSIX)**
  - **family of standards**
  - **specified by the IEEE Computer Society**
  - **for maintaining compatibility between operating systems.**
  - **API (system calls), shells, utility commands for compatibility among UNIXes and variants**

POSIX Shared Memory

- shm_open
- ftruncate
- Mmap
- See the example in Textbook

Message passing

Message Passing

- Message system – processes communicate with each other using send(), receive() like syscalls given by OS
- IPC facility provides two operations:
  - send(message) – message size fixed or variable
  - Receive(message)
- If P and Q wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via send/receive
- Communication link can be implemented in a variety of ways

Message Passing using "Naming"

- Pass a message by "naming" the receiver
  - A) Direct communication with receiver
    - Receiver is identified by sender directly using it's name
  - B) Indirect communication with receiver
    - Receiver is identified by sender in-directly using it's 'location of receipt'

Message passing using direct communication

- **Processes must name each other explicitly:**
  - **send (P, message) – send a message to process P**
  - **receive(Q, message) – receive a message from process Q**
- **Properties of communication link**
  - **Links are established automatically**
  - **A link is associated with exactly one pair of communicating processes**
  - **Between each pair there exists exactly one link**
  - **The link may be unidirectional, but is usually bi-directional**

Message passing using IN-direct communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Message passing using IN-direct communication

- **Operations**
  - **create a new mailbox**
  - **send and receive messages through mailbox**
  - **destroy a mailbox**
- **Primitives are defined as:**
  - **send(A, message) – send a message to mailbox A**
  - **receive(A, message) – receive a message from mailbox A**

Message passing using IN-direct communication

- **Mailbox sharing**
  - **P1, P2, and P3 share mailbox A**
  - **P1, sends; P2 and P3 receive**
  - **Who gets the message?**
- **Solutions**
  - **Allow a link to be associated with at most two processes**
  - **Allow only one process at a time to execute a receive operation**
  - **Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.**

Message Passing implementation: Synchronization issues

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
  - Blocking send has the sender block until the message is received
  - Blocking receive has the receiver block until a message is available
- Non-blocking is considered asynchronous
  - Non-blocking send has the sender send the message and continue
  - Non-blocking receive has the receiver receive a valid message or null

Producer consumer using blocking send and receive

Producer

```
message next produced;
while (true) {
/* produce an item in next_produced */
send(next_produced);
}
```

Consumer

```
message next_consumed;
while (true) {
receive(next_consumed);
}
```

Message Passing implementation: choice of Buffering

- **Queue of messages attached to the link; implemented in one of three ways**
  1. **Zero capacity – 0 messages**
     - **Sender must wait for receiver (rendezvous)**
  2. **Bounded capacity – finite length of n messages**
     - **Sender must wait if link full**
  3. **Unbounded capacity – infinite length**
     - **Sender never waits**

Processes in xv6 code

Process Table

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

- One single global array of processes
- Protected by

  `ptable.lock`

Layout of process's VA space

xv6 schema!

different from Linux

Logical layout of memory for a process

- Address 0: code
- Then globals
- Then stack
- Then heap
- Each processe's address space maps kernel's text, data also --> so that system calls run with these mappings
- Kernel code can directly access user memory now

Kernel mappings in user address space actual location of kernel

- **Kernel is loaded at 0x100000 physical address**
- **PA 0 to 0x100000 is BIOS and devices**
- **Process's page table will map**

  VA 0x80000000 to PA 0x00000 and

  VA 0x8010000 to 0x100000

Kernel mappings in user address space actual location of kernel

- **Kernel is not loaded at the PA 0x80000000 because some systems may not have that much memory**
- **0x80000000 is called KERNBASE in xv6**

Imp Concepts

- A process has two stacks
  - user stack: used when user code is running
  - kernel stack: used when kernel is running on behalf of a process
- Note: there is a third stack also!
  - The kernel stack used by the scheduler itself
  - Not a per process stack

Struct proc

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state. allocated, ready to run, running, wait-
ing for I/O, or exiting.
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process. Process's context
  void *chan;                  // If non-zero, sleeping on chan. More when we discuss
sleep, wakeup
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files, used by open(), read(),...
  struct inode *cwd;           // Current directory, changed with "chdir()"
  char name[16];               // Process name (for debugging)
};
```

struct proc diagram: Very imp!

sz = ELF-code->memsz (includes data, check “ld -N”
+ 2*4096 (for stack)

In use only when you are in kernel on a “trap” = interrupt/syscall. “tf” always used. trapret,forkret used during fork()

Memory Layout of a user process

Memory Layout of a user process

After exec()

Note the argc, argv on stack

The "guard page" is just a mapping in page table. No frame allocated. It's marked as invalid. So if stack grows (due to many function calls), then OS will detect it with an exception

Handling Traps

Handling traps

- Transition from user mode to kernel mode
  - On a system call
  - On a hardware interrupt
  - User program doing illegal work (exception)
- Actions needed, particularly w.r.t. to hardware interrupts
  - Change to kernel mode & switch to kernel stack
  - Kernel to work with devices, if needed
  - Kernel to understand interface of device

Handling traps

- **Actions needed on a trap**
  - **Save the processor's registers (context) for future use**
  - **Set up the system to run kernel code (kernel context) on kernel stack**
  - **Start kernel in appropriate place (sys call, intr handler, etc)**
  - **Kernel to get all info related to event (which block I/O done?, which sys call called, which process did exception and what type, get arguments to system call, etc)**

Privilege level

- The x86 has 4 protection levels, numbered 0 (most privilege) to 3 (least privilege).
- In practice, most operating systems use only 2 levels: 0 and 3, which are then called kernel mode and user mode, respectively.
- The current privilege level with which the x86 executes instructions is stored in %cs register, in the field CPL.

Privilege level

- **Changes automatically on**

  “int” instruction

  hardware interrupt

  exeception

- **Changes back on**

  iret

- **“int” 10 --> makes 10th hardware interrupt. S/w interrupt can be used to create hardware interrupt’**
- **Xv6 uses “int 64” for actual system calls**

Interrupt Descriptor Table (IDT)

- IDT defines intertupt handlers
- Has 256 entries
  - each giving the %cs and %eip to be used when handling the corresponding interrupt.
- Interrupts 0-31 are defined for software exceptions, like divide errors or attempts to access invalid memory addresses.
- Xv6 maps the 32 hardware interrupts to the range 32-63
- and uses interrupt 64 as the system call interrupt

Interrupt Descriptor Table (IDT) entries

```
// Gate descriptors for interrupts and traps
struct gatedesc {
  uint off_15_0 : 16;   // low 16 bits of offset in segment
  uint cs : 16;         // code segment selector
  uint args : 5;        // # args, 0 for interrupt/trap
gates
  uint rsv1 : 3;        // reserved(should be zero I guess)
  uint type : 4;        // type(STS_{IG32,TG32})
  uint s : 1;           // must be 0 (system)
  uint dpl : 2;         // descriptor(meaning new) privilege
level
  uint p : 1;           // Present
  uint off_31_16 : 16;  // high bits of offset in segment
};
```

Setting IDT entries

```
void
tvinit(void)
{
  int i;
  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
            vectors[T_SYSCALL], DPL_USER);
  /* value 1 in second argument --> don't disable
interrupts
         * DPL_USER means that processes can raise this
interrupt. */
    initlock(&tickslock, "time");
}
```

Setting IDT entries

```
#define SETGATE(gate, istrap, sel, off, d)                    \
{                                                             \
  (gate).off_15_0 = (uint)(off) & 0xffff;                    \
  (gate).cs = (sel);                                          \
  (gate).args = 0;                                            \
  (gate).rsv1 = 0;                                            \
  (gate).type = (istrap) ? STS_TG32 : STS_IG32;               \
  (gate).s = 0;                                               \
  (gate).dpl = (d);                                           \
  (gate).p = 1;                                               \
  (gate).off_31_16 = (uint)(off) >> 16;                      \
}
```

Setting IDT entries

Vectors.S

```
# generated by vectors.pl - do
not edit
# handlers
.globl alltraps
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
.globl vector1
vector1:
  pushl $0
  pushl $1
 jmp alltraps
```

trapasm.S

```
#include "mmu.h"
# vectors.S sends all traps
here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  Pushal
  ....
```

How will interrupts be handled?

On int instruction/interrupt the CPU does this:

- Fetch the n'th descriptor from the IDT, where n is the argument of int.
- Check that CPL in %cs is <= DPL, where DPL is the privilege level in the descriptor.
- Save %esp and %ss in CPU-internal registers, but only if the target segment selector's PL < CPL.
  - Switching from user mode to kernel mode. Hence save user code's SS and ESP
- Load %ss and %esp from a task segment descriptor.
  - Stack changes to kernel stack now. TS descriptor is on GDT, index given by TR register. See switchuvm()
- Push %ss. // optional
- Push %esp. // optional (also changes ss,esp using TSS)
- Push %eflags.
- Push %cs.
- Push %eip.
- Clear the IF bit in %eflags, but only on an interrupt.
- Set %cs and %eip to the values in the descriptor.

After “int” ‘s job is done

- IDT was already set
  - Remember vectors.S
- So jump to 64th entry in vector’s

```
vector64:
  pushl $0
  pushl $64
  jmp alltraps
```

  - So now stack has ss, esp,eflags, cs, eip, 0 (for error code), 64
  - Next run alltraps from trapasm.S

alltraps:

```
# Build trap frame.
pushl %ds
pushl %es
pushl %fs
pushl %gs
pushal // push all gen purpose regs
# Set up data segments.
movw $(SEG_KDATA<<3), %ax
movw %ax, %ds
movw %ax, %es
# Call trap(tf), where tf=%esp
pushl %esp # first arg to trap()
call trap
addl $4, %esp
```

- Now stack contains
- ss, esp,eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi
  - This is the struct trapframe !
  - So the kernel stack now contains the trapframe
  - Trapframe is a part of kernel stcak

trap()

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
  switch(tf->trapno){
  .....
```

- Argument is trapframe
- In alltraps
  - Before “call trap”, there was “push %esp” and stack had the trapframe
  - Remember calling convention --> when a function is called, the stack contains the arguments in reverse order (here only 1 arg)

trap()

- Has a switch
  - switch(tf->trapno)
  - Q: who set this trapno?
- Depending on the type of trap
  - Call interrupt handler
    - Timer
      - wakeup(&ticks)
    - IDE: disk interrupt
      - Ideintr()
    - KBD
      - Kbdintr()
    - COM1
      - Uatrintr()
    - If Timer
      - Call yield() -- calls sched()
    - If process was killed (how is that done?
      - Call exit()!

when trap() returns

- #Back in alltraps

```
  call trap
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

- Stack had (trapframe)
  - ss, esp,eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi, esp
- add $4 %esp
  - esp
- popal
  - eax, ecx, edx, ebx, oesp, ebp, esi, edi
- Then gs, fs, es, ds
- add $0x8, %esp
  - 0 (for error code), 64
- iret
  - ss, esp,eflags, cs, eip,

Scheduler

Scheduler – in most simple terms

- **Selects a process to execute and passes control to it !**
  - **The process is chosen out of “READY” state processes**
  - **Saving of context of “earlier” process and loading of context of “next” process needs to happen**
- **Questions**
  - **What are the different scenarios in which a scheduler called ?**
  - **What are the intricacies of “passing control”**
  - **What is “context” ?**

Steps in scheduling scheduling

- Suppose you want to switch from P1 to P2 on a timer interrupt
- P1 was doing

```
F() {  i++; j++;}
```

- P2 was doing

```
G() { x--; y++; }
```

- P1 will experience a timer interrupt, switch to kernel (scheduler) and scheduler will schedule P2

4 stacks need to change!

- **User stack of process -> kernel stack of process**
  - **Switch to kernel stack**
  - **The normal sequence on any interrupt !**
- **Kernel stack of process -> kernel stack of scheduler**
  - **Why?**
- **Kernel stack of scheduler -> kernel stack of new process . Why?**
- **Kernel stack of new process -> user stack of new process**

scheduler()

- Disable interrupts
- Find a RUNNABLE process. Simple round-robin!
- c->proc = p
- switchuvm(p) : Save TSS of scheduler’s stack and make CR3 to point to new process pagedir
- p->state = RUNNING
- swtch(&(c->scheduler), p->context)

swtch

```
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

scheduler()

- swtch(&(c->scheduler), p->context)
- Note that when scheduler() was called, when P1 was running
- After call to swtch() shown above
  - The call does NOT return!
  - The new process P2 given by ‘p’ starts running !
  - Let’s review swtch() again

swtch(old, new)

- The magic function in swtch.S
- Saves callee-save registers of old context
- Switches esp to new-context's stack
- Pop callee-save registers from new context

  ret

  - where? in the case of first process – returns to forkret() because stack was setup like that !
  - in case of other processes, return where?
    - Return address given on kernel stack. But what's that?
    - The EIP in p->context
    - When was EIP set in p->context ?

scheduler()

- Called from?
  - mpmain()
  - No where else!
- sched() is another scheduler function !
  - Who calls sched() ?
    - exit() - a process exiting calls sched ()
    - yield() - a process interrupted by timer calls yield()
    - sleep() - a process going to wait calls sleep()

sched()

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();
  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
/*A*/  mycpu()->intena = intena;
}
```

- get current process
- Error checking code (ignore as of now)
- get interrupt enabled status on current CPU (ignore as of now)
- call to swtch
  - Note the arguments' order
  - p->context first, mycpu()->scheduler second
- swtch() is a function call
  - pushes address of */*A*/* on stack of current process p
  - switches stack to mycpu()->scheduler. Then pops EIP from that stack and jumps there.
  - when was mycpu()->scheduler set? Ans: during scheduler()!

sched() and schduler()

```
sched() {
...
  swtch(&p->context, mycpu()->scheduler); /* X */

}
```

```
scheduler(void) {
...
      swtch(&(c->scheduler), p->context);  / * Y */
}
```

- scheduler() saves context in c->scheduler, sched() saves context in p->context
- after swtch() call in sched(), the control jumps to Y in scheduler
  - Switch from process stack to scheduler's stack
- after swtch() call in scheduler(), the control jumps to X in sched()
  - Switch from scheduler's stack to new process's stack
- Set of co-operating functions

sched() and scheduler() as co-routines

- In sched()

  swtch(&p->context, mycpu()->scheduler);

- In scheduler()

  swtch(&(c->scheduler), p->context);

- These two keep switching between processes
- These two functions work together to achieve scheduling
- Using asynchronous jumps
- Hence they are co-routines

To summarize

- On a timer interrupt during P1
  - trap() is called. Stack has changed from P1's user stack to P1's kernel stack
  - trap()->yield()
  - yield()->sched()
  - sched() -> swtch(&p->context, c->scheduler()
  - Stack changes to scheduler's kernel stack.
  - Switches to location "Y" in scheduler().
-
- Now the loop in scheduler()
  - calls switchkvm()
  - Then continues to find next process (P2) to run
  - Then calls switchuvm(p): changing the page table to the P2's page tables
  - then calls swtch(&c->scheduler, p2's->context)
  - Stack changes to P2's kernel stack.
  - P2 runs the last instruction it was was in ! Where was it?
    - mycpu()->intena = intena; in sched()
    - Then returns to the one who called sched() i.e. exit/sleep, etc
    - Finally returns from it's own "TRAP" handler and returns to P2's user stack and user code

Memory Management – Continued

More on Linking, Loading, Paging

Review of last class

- **MMU : Hardware features for MM**
- **OS: Sets up MMU for a process, then schedules process**
- **Compiler : Generates object code for a particular OS + MMU architecture**
- **MMU: Detects memory violations and raises interrupt --> Effectively passing control to OS**

More on Linking and Loading

- **Static Linking: All object code combined at link time and a big object code file is created**
- **Static Loading: All the code is loaded in memory at the time of exec()**
- **Problem: Big executable files, need to load functions even if they do not execute**
- **Solution: Dynamic Linking and Dynamic Loading**

Dynamic Linking

- **Linker is normally invoked as a part of compilation process**
  - **Links**
    - **function code to function calls**
    - **references to global variables with “extern” declarations**
- **Dynamic Linker**
  - **Does not combine function code with the object code file**
  - **Instead introduces a “stub” code that is indirect reference to actual code**
  - **At the time of “loading” (or executing!) the program in memory, the “link-loader” (part of OS!) will pick up the relevant code from the library machine code file (e.g. libc.so.6)**

Dynamic Linking on Linux

```
#include <stdio.h>
int main() {
int a, b;
scanf("%d%d", &a, &b);
printf("%d %d\n", a, b);
return 0;
}
```

PLT: Procedure Linkage Table
used to call external procedures/functions whose address is to be resolved by the dynamic linker at run time.

Output of objdump -x -D

Disassembly of section .text:

```
0000000000001189 <main>:
    11d4:        callq  1080 <printf@plt>
```

Disassembly of section .plt.got:

```
0000000000001080 <printf@plt>:
    1080:      endbr64
    1084:      bnd jmpq *0x2f3d(%rip)        # 3fc8
<printf@GLIBC_2.2.5>
    108b:       nopl   0x0(%rax,%rax,1)
```

Dynamic Loading

- **Loader**
  - Loads the program in memory
  - Part of exec() code
  - Needs to understand the format of the executable file (e.g. the ELF format)
- **Dynamic Loading**
  - Load a part from the ELF file only if needed during execution
  - Delayed loading
  - Needs a more sophisticated memory management by operating system – to be seen during this series of lectures

Dynamic Linking, Loading

- **Dynamic linking necessarily demands an advanced type of loader that understands dynamic linking**
  - **Hence called ‘link-loader’**
  - **Static or dynamic loading is still a choice**
- **Question: which of the MMU options will alllow for which type of linking, loading ?**

Continous memory management

What is Continous memory management?

- **Entire process is hosted as one continous chunk in RAM**
- **Memory is typically divided into two partitions**
  - **One for OS and other for processes**
  - **OS most typically located in “high memory” addresses, because interrupt vectors map to that location (Linux, Windows) !**

Hardware support needed: base + limit (or relocation + limit)

Figure 9.6 Hardware support for relocation and limit registers.

Problems faced by OS

- Find a continuous chunk for the process being forked
- Different processes are of different sizes
  - Allocate a size parameter in the PCB
- After a process is over – free the memory occupied by it
- Maintain a list of free areas, and occupied areas
  - Can be done using an array, or linked list

Variable partition scheme

Figure 9.7 Variable partition.

Problem: how to find a “hole” to fit in new process

- Suppose there are 3 free memory regions of sizes 30k, 40k, 20k
- The newly created process (during fork() + exec()) needs 15k
- Which region to allocate to it ?

Strategies for finding a free chunk

- **6k, 17k, 16k, 40k holes . Need 15k.**
- **Best fit: Find the smallest hole, larger than process. Ans: 16k**
- **Worst fit: Find the largest hole. Ans: 40k**
- **First fit: Find the “first” hole larger than the process. Ans: 17k**

Problem : External fragmentation

- **Free chunks: 30k, 40k, 20k**
- **The newly created process (during fork() + exec()) needs 50k**
- **Total free memory: 30+40+20 = 90k**
  - **But can’t allocate 50k !**

Solution to external fragmentation

- **Compaction !**
- **OS moves the process chunks in memory to make available continous memory region**
  - **Then it must update the memory management information in PCB (e.g. base of the process) of each process**
- **Time consuming**
- **Possible only if the relocation+limit scheme of MMU is available**

Another solution to external fragmentation: Fixed size partitions

- **Fixed partition scheme**
- **Memory is divided by OS into chunks of equal size: e.g., say, 50k**
  - **If total 1M memory, then 20 such chunks**
- **Allocate one or more chunks to a process, such that the total size is >= the size of the process**
  - **E.g.if request is 50k, allocate 1 chunk**
  - **If request is 40k, still allocate 1 chunk**
  - **If request is 60k, then allocate 2 chunks**
- **Leads to internal fragmentation**
  - **space wasted in the case of 40k or 60k requests above**

Fixed partition scheme

- **OS needs to keep track of**
  - **Which partition is free and which is used by which process**
  - **Free partitions can simply be tracked using a bitmap or a list of numbers**
  - **Each process's PCB will contain list of partitions allocated to it**

Solution to internal fragmentation

- **Reduce the size of the fixed sized partition**
- **How small then ?**
  - **Smaller partitions mean more overhead for the operating system in allocating deallocating**

Paging

An extended version of fixed size partitions

- **Partition = page**
  - Process = logically continuous sequence of bytes, divided in ‘page’ sizes
  - Memory divided into equally sized page ‘frames’
- **Important distinction**
  - Process need not be continous in RAM
  - Different page sized chunks of process can go in any page frame
  - Page table to map pages into frames

Logical address seen as

Paging hardware

Figure 9.8 Paging hardware.

MMU’s job

To translate a logical address generated by the CPU to a physical address:

1. Extract the page number p and use it as an index into the page table.

(Page table location is stored in a hardware register

Also stored in PCB of the process, so that it can be used to load the hardware register on a context switch)

2. Extract the corresponding frame number f from the page table.

3. Replace the page number p in the logical address with the frame number f .

Job of OS

- **Allocate a page table for the process, at time of fork()/exec()**
  - **Allocate frames to process**
  - **Fill in page table entries**
- **In PCB of each process, maintain**
  - **Page table location (address)**
  - **List of pages frames allocated to this process**
- **During context switch of the process, load the PTBR using the PCB**

Job of OS

- **Maintain a list of all page frames**
  - **Allocated frames**
  - **Free Frames (called frame table)**
  - **Can be done using simple linked list**
  - **Innovative data structures can also be used to maintain free and allocated frames list (e.g. xv6 code)**
  -

Figure 9.11 Free frames (a) before allocation and (b) after allocation.

Disadvantage of Paging

- **Each memory access results in two memory accesses!**
  - **One for page table, and one for the actual memory location !**
  - **Done as part of execution of instruction in hardware (not by OS!)**
  - **Slow down by 50%**

Speeding up paging

- Translation Lookaside Buffer (TLB)
- Part of CPU hardware
- A cache of Page table entries
- Searched in parallel for a page number

Speedup due to TLB

- **Hit ratio**
- **Effective memory access time**

  = Hit ratio * 1 memory access time + miss ratio * 2 memory access time

- **Example: memory access time 10ns, hit ratio = 0.8, then**

  effective access time = 0.80 × 10 + 0.20 × 20

  = 12 nanoseconds

Figure 9.13 Valid (v) or invalid (i) bit in a page table.

Memory protection with paging

X86 PDE and PTE

PDE

PTE

P	Present
W	Writable
U	User
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use

Figure 9.14 Sharing of standard C library in a paging environment.

Shared pages (e.g. library) with paging

Paging: problem of large PT

- **64 bit address**
- **Suppose 20 bit offset**
  - **That means 2^20 = 1 MB pages**
  - **44 bit page number: 2^44 that is trillion sized page table!**
  - **Can’t have that big continuous page table!**

Paging: problem of large PT

- 32 bit address
- Suppose 12 bit offset
  - That means 2^12 = 4 KB pages
  - 20 bit page number: 2^20 that is a million entries
  - Can’t always have that big continuous page table as well, for each process!

Hierarchical paging

Figure 9.15 A two-level page-table scheme.

outer page	inner page	offset
p_1	p_2	d
42	10	12

More hierarchy

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

Problems with hierarchical paging

- **More number of memory accesses with each level !**
  - **Too slow !**
- **OS data structures also needed in that proportion**

Hashed page table

Figure 9.17 Hashed page table.

Inverted page table

Figure 9.18 Inverted page table.

Normal page table – one per process --> Too much memory consumed

Inverted page table : global table – only one
Needs to store PID in the table entry

Examples of systems using inverted page
tables include the 64-bit Ultra SPARC and Power PC

virtual address
consists of a triple:
<process-id, page-number, offset>

Case Study: Oracle SPARC Solaris

- 64 bit SPARC processor , 64 bit Solaris OS
- Uses Hashed page tables
  - one for the kernel and one for all user processes.
  - Each hash-table entry : base + span (#pages)
    - Reduces number of entries required

Case Study: Oracle SPARC Solaris

- **Cashing levels: TLB (on CPU), TSB(in Memory), Page Tables (in Memory)**
  - **CPU implements a TLB that holds translation table entries (TTE s) for fast hardware lookups.**
  - **A cache of these TTEs resides in a in-memory translation storage buffer (TSB), which includes an entry per recently accessed page**
  - **When a virtual address reference occurs, the hardware searches the TLB for a translation.**
  - **If none is found, the hardware walks through the in memory TSB looking for the TTE that corresponds to the virtual address that caused the lookup**

Case Study: Oracle SPARC Solaris

- If a match is found in the TSB , the CPU copies the TSB entry into the TLB , and the memory translation completes.
- If no match is found in the TSB , the kernel is interrupted to search the hash table.
- The kernel then creates a TTE from the appropriate hash table and stores it in the TSB for automatic loading into the TLB by the CPU memory-management unit.
- Finally, the interrupt handler returns control to the MMU , which completes the address translation and retrieves the requested byte or word from main memory.

Swapping

Figure 9.19 Standard swapping of two processes using a disk as a backing store.

Swapping

- **Standard swapping**
  - Entire process swapped in or swapped out
  - With continous memory management
- **Swapping with paging**
  - Some pages are “paged out” and some “paged in”
  - Term “paging” refers to paging with swapping now

Figure 9.20 Swapping with paging.

Words of caution about ‘paging’

- **Not as simple as it sounds when it comes to implementation**
  - **Writing OS code for this is challenging**

Mark True, the actions done as part of code of swtch() in swtch.S, in xv6

Restore new callee saved registers from kernel stack of new context: True

Save old callee saved registers on kernel stack of old context: True

Save old callee saved registers on user stack of old context: False

Switch from old process context to new process context: False

Switch from one stack (old) to another(new): True

Restore new callee saved registers from user stack of new context: False

Jump to code in new context: False

For each function/code-point, select the status of segmentation setup in xv6

: bootmain() → gdt setup with 3 entries, at start32 symbol of bootasm.S,

kvmalloc() in main() → gdt setup with 3 entries, at start32 symbol of bootasm.S,

after startothers() in main() → gdt setup with 5 entries (0 to 4) on all processors,

after seginit() in main() → gdt setup with 5 entries (0 to 4) on one processor,

bootasm.S → gdt setup with 3 entries, at start32 symbol of bootasm.S,

entry.S → gdt setup with 3 entries, at start32 symbol of bootasm.S

Compare paging with demand paging and select the correct statements.

The correct answers are: Demand paging requires additional hardware support, compared to paging., Both demand paging and paging support shared memory pages., With demand paging, it's possible to have user programs bigger than physical memory., Demand paging always increases effective memory access time., Paging requires some hardware support in CPU, Calculations of number of bits for page number and offset are same in paging and demand paging., The meaning of valid-invalid bit in page table is different in paging and demandpaging.

The OS may terminate the process while handling the interrupt of memory violation: True

The hardware detects any memory access beyond the limit value and raises an interrupt: True

The hardware may terminate the process while handling the interrupt of memory violation: False

The OS sets up the relocation and limit registers when the process is scheduled: True

The compiler generates machine code assuming continuous memory address space for process, and calculating appropriate sizes for code, and data;: True

The process sets up it's own relocation and limit registers when the process is scheduled: False

The OS detects any memory access beyond the limit value and raises an interrupt: False

The compiler generates machine code assuming appropriately sized semgments for code, data and stack.: False

---

Consider the following list of free chunks, in continuous memory management: 10k, 25k, 12k, 7k, 9k, 13k Suppose there is a request for chunk of size 9k, then the free chunk selected under each of the following schemes will be Best fit: 9k First fit: 10k Worst fit: 25k

---

Select all the correct statements about MMU and it's functionality Select one or more:

The correct answers are: MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

---

The correct answers are: MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

The correct answer is: 107

Select the compiler's view of the process's address space, for each of the following MMU schemes: (Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

The correct answer is: Segmentation, then paging → many continuous chunks of variable size, Relocation + Limit → one continuous chunk, Segmentation → many continuous chunks of variable size, Paging → one continuous chunk

---

Suppose the memory access time is 180ns and TLB hit ratio is 0.3, then effective memory access time is (in nanoseconds); The correct answer is: 306.00

In xv6, The struct context is given as struct context { uint edi; uint esi; uint ebx; uint ebp; uint eip; }; Select all the reasons that explain why only these 5 registers are included in the struct context.

The correct answers are: The segment registers are same across all contexts, hence they need not be saved, eax, ecx, edx are caller save, hence no need to save, esp is not saved in context, because context{} is on stack and it's address is always argument to swtch()

Arrange the following events in order, in page fault handling:

The correct answer is: Disk interrupt wakes up the process → 7, The reference bit is found to be invalid by MMU → 1, OS makes available an empty frame → 4, Restart the instruction that caused the page fault → 9, A hardware interrupt is issued → 2, OS schedules a disk read for the page (from backing store) → 5, Process is kept in wait state → 6, Page tables are updated for the process → 8, Operating system decides that the page was not in memory → 3

Suppose a kernel uses a buddy allocator. The smallest chunk that can be allocated is of size 32 bytes. One bit is used to track each such chunk, where 1 means allocated and 0 means free. The chunk looks like this as of now: 00001010

The correct answer is: 11111010

The complete range of virtual addresses (after main() in main.c is over), from which the free pages used by kalloc() and kfree() is derived,are: The correct answer is: end, P2V(PHYSTOP)

Match the pair

the correct answer is: Hashed page table → Linear search on collsion done by OS (e.g. SPARC Solaris) typically, Inverted Page table → Linear/Parallel search using page number in page table, Hierarchical Paging → More memory access time per hierarchy

---

After virtual memory is implemented (select T/F for each of the following)One Program's size can be larger than physical memory size

Code need not be completely in memory: True Cumulative size of all programs can be larger than physical memory size: True Virtual access to memory is granted: False Logical address space could be larger than physical address space: True Virtual addresses are available: False Relatively less I/O may be possible during process execution: True One Program's size can be larger than physical memory size: True

---

W.r.t. Memory management in xv6, xv6 uses physical memory upto 224 MB onlyMark statements True or False

The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context: True

The stack allocated in entry.S is used as stack for scheduler's context for first processor: True

The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir: True

The free page-frame are created out of nearly 222 MB: True

The kernel code and data take up less than 2 MB space: True

The switchkvm() call in scheduler() changes CR3 to use page directory of new process: False

The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context: False

PHYSTOP can be increased to some extent, simply by editing memlayout.h: True

xv6 uses physical memory upto 224 MB only: True

The process's address space gets mapped on frames, obtained from ~2MB:224MB range: True

The kernel's page table given by kpgdir variable is used as stack for scheduler's context: False

Consider the reference string 6 4 2 0 1 2 6 9 2 0 5 If the number of page frames is 3, then total number of page faults (including initial), using LRU replacement is: Answer: #6# 6,4# 6,4,2 # 0,4,2#0,1,2#6,1,2#6,9,2#0,9,2#0,5,2 The correct answer is: 9

Consider the image given below, which explains how paging works.

The PTBR is present in the CPU as a register: True The page table is indexed using frame number: False The page table is indexed using page number: True The locating of the page table using PTBR also involves paging translation: False Size of page table is always determined by the size of RAM: False The page table is itself present in Physical memory: True Maximum Size of page table is determined by number of bits used for page number: True The physical address may not be of the same size (in bits) as the logical address: True

---

Given below is shared memory code with two processes sharing a memory segment. The first process sends a user input string to second process. The second capitalizes the string. Then the first process prints the capitalized version. Fill in the blanks to complete the code. // First process #define SHMSZ 27 int main() { char c; int shmid; key_t key; char *shm, *s, string[128]; key = 5679; if ((shmid = shmget (key, SHMSZ, IPC_CREAT | 0666)) < 0) { perror("shmget"); exit(1); } if ((shm = shmat (shmid, NULL, 0)) == (char *) -1) { perror("shmat"); exit(1); } s = shm; *s = '$'; scanf("%s", string); strcpy(s + 1, string); *s = ' @ '; //note the quotes while(*s != ' $ ') sleep(1); printf("%s\n", s + 1); exit(0); } //Second process

#define SHMSZ 27 int main() { int shmid; key_t key; char *shm, *s; int i; char string[128]; key = 5679

if ((shmid = shmget(key, SHMSZ, 0666)) < 0) { perror("shmget"); exit(1); } if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) { perror("shmat"); exit(1); } s = shm ; while(*s != '@') sleep(1); for(i = 0; i < strlen(s + 1); i++) s[i + 1] = toupper(s[i + 1]); *s = '$'; exit(0); }

Map the functionality/use with function/variable in xv6 code.

You have correctly selected 3. The correct answer is: return a free page, if available; 0, otherwise → kalloc(), Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed → mappages(), Array listing the kernel memory mappings, to be used by setupkvm() → kmap[], Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices → setupkvm(), Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary → walkpgdir(), Setup kernel part of a page table, and switch to that page table → kvmalloc()

Order events in xv6 timer interrupt code (Transition from process P1 to P2's code.)

The correct answer is: P2 is selected and marked RUNNING → 12, Change of stack from user stack to kernel stack of P1 → 3, Timer interrupt occurs → 2, alltraps() will call iret → 18, change to context of P2, P2's kernel stack in use now → 13, P2's trap() will return to alltraps → 17, jump in vector.S → 4, P2 will return from sched() in yield() → 15, yield() is called → 8, trap() is called → 7, Process P2 is executing →

14, P1 is marked as RUNNABLE → 9, P2's yield() will return in trap() → 16, Process P1 is executing → 1, sched() is called, → 10, change to context of the scheduler, scheduler's stack in use now → 11, jump to alltraps → 5, Trapframe is built on kernel stack of P1 →6

Given that the memory access time is 200 ns, probability of a page fault is 0.7 and page fault handling time is 8 ms, The effective memory access time in nanoseconds is: The correct answer is: 5600060.00

Select the state that is not possible after the given state, for a process: New: Ready : Running: : Waiting: Running Waiting None of these Running

---

Select the correct statements about sched() and scheduler() in xv6 code

You have correctly selected 5. The correct answers are: sched() and scheduler() are co-routines, When either sched() or scheduler() is called, it does not return immediately to caller, When either sched() or scheduler() is called, it results in a context switch, sched() switches to the scheduler's context, scheduler() switches to the selected process's context, After call to swtch() in scheduler(), the control moves to code in sched(), After call to swtch() in sched(), the control moves to code in scheduler(), Each call to sched() or scheduler() involves change of one stack inside swtch()

The data structure used in kalloc() and kfree() in xv6 is

Your answer is correct. The correct answer is: Singly linked NULL terminated list

Virtual Memory

Introduction

- Virtual memory != Virtual address

  Virtual address is address issued by CPU’s execution unit, later converted by MMU to physical address

  Virtual memory is a memory management technique employed by OS (with hardware support, of course)

Unused parts of program

```
int a[4096][4096]
int f(int m[][4096]) {
    int i, j;
    for(i = 0; i < 1024; i++)
        m[0][i] = 200;
}
int main() {
    int i, j;
    for(i = 0; i < 1024; i++)
        a[1][i] = 200;
    if(random() == 10)
          f(a);
}
```

All parts of array a[] not accessed

Function f() may not be called

Some problems with schemes discussed so far

- Code needs to be in memory to execute, But entire program rarely used
  - Error code, unusual routines, large data structures are rarely used
- So, entire program code, data not needed at same time
- So, consider ability to execute partially-loaded program
  - One Program no longer constrained by limits of physical memory
  - One Program and collection of programs could be larger than physical memory

What is virtual memory?

- Virtual memory – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory Larger Than Physical Memory

Virtual Address space

Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc

Shared pages Using Virtual Memory

System libraries shared via mapping into virtual address space

Shared memory by mapping same page-frames into page tables of involved processes

Pages can be shared during `fork()`, speeding process creation (more later)

Demand Paging

Demand Paging

- Load a “page” to memory when it’s neded (on demand)

  Less I/O needed, no unnecessary I/O

  Less memory needed

  Faster response

  More users

-

Demand Paging

- Options:

  Load entire process at load time : achieves little

  Load some pages at load time: good

  Load no pages at load time: pure demand paging

New meaning for valid/invalid bits in page table

Frame #	valid-invalid bit
	v
	v
	v
	v
	i
….	
	i
	i

- With each page table entry a valid–invalid bit is associated

  v: in-memory – memory resident

  i : not-in-memory or illegal

- During address translation, if valid–invalid bit in page table entry is I : **raises trap called page fault**

Page

Table

With

Some pages

Not

In memory

Page fault

Page fault

- Page fault is a hardware interrupt
- It occurs when the page table entry corresponding to current memory access is “i”
- All actions that a kernel takes on a hardware interrupt are taken!

  Change of stack to kernel stack

  Saving the context of process

  Switching to kernel code

On a Page fault

1) Operating system looks at another data structure (table), most likely in PCB itself, to decide:

   If it's Invalid reference -> abort the process (segfault)

   Just not in memory -> Need to get the page in memory

2) Get empty frame (this may be complicated!)

3) Swap page into frame via scheduled disk/IO operation

4) Reset tables to indicate page now in memory.

5) Set validation bit = v

6) Restart the instruction that caused the page fault

Additional problems

- Extreme case – start process with *no* pages in memory

  OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault

  And for every other process pages on first access

  Pure demand paging

- Actually, a given instruction could access multiple pages -> multiple page faults

  Pain decreased because of **locality of reference**

- Hardware support needed for demand paging

  Page table with valid / invalid bit

  Secondary memory (swap device with **swap space**)

  Instruction restart

Instruction restart

- A critical Problem
- Consider an instruction that could access several different locations

```
movarray 0x100, 0x200, 20
```

copy 20 bytes from address 0x100 to address 0x200

```
movarray 0x100, 0x110, 20
```

what to do in this case?

Handling A Page Fault

Page fault handling

1) Trap to the operating system
2) Default trap handling():

    Save the process registers and process state

    Determine that the interrupt was a page fault. Run page fault handler.

3) Page fault handler(): Check that the page reference was legal and determine the location of the page on the disk. If illegal, terminate process.
4) Find a free frame. Issue a read from the disk to a free frame:

    Process waits in a queue for disk read. Meanwhile many processes may get scheduled.

    Disk DMA hardware transfers data to the free frame and raises interrupt in end

Page fault handling

6) (as said on last slide) While waiting, allocate the CPU to some other process
7) (as said on last slide) Receive an interrupt from the disk I/O subsystem (I/O completed)
8) Default interrupt handling():

    Save the registers and process state for the other user

    Determine that the interrupt was from the disk

9) Disk interrupt handler():

    Figure out that the interrupt was for our waiting process

    Make the process runnable

10) Wait for the CPU to be allocated to this process again

    Kernel restores the page table of the process, marks entry as “v”

    Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of demand paging

Page Fault Rate 0 <= p <= 1

if p = 0 no page faults

if p = 1, every reference is a fault

Effective (memory) Access Time (EAT)

EAT = (1 – p) * memory access time +

p * (page fault overhead // Kernel code execution time

+ swap page out // time to write an occupied frame to disk

+ swap page in // time to read data from disk into free frame

+ restart overhead) // time to reset process context, restart it

Performance of demand paging

Memory access time = 200 nanoseconds

Average page-fault service time = 8 milliseconds
EAT = (1 – p) x 200 + p (8 milliseconds)

= (1 – p x 200 + p x 8,000,000

= 200 + p x 7,999,800

If one access out of 1,000 causes a page fault, then

EAT = 8.2 microseconds.

This is a slowdown by a factor of 40!!

If want performance degradation < 10 percent

220 > 200 + 7,999,800 x p
20 > 7,999,800 x p

p < .0000025

< one page fault in every 400,000 memory accesses

An optimization: Copy on write

The problem with fork() and exec(). Consider the case of a shell

```
scanf("%s", cmd);
if(strcmp(cmd, "exit") == 0)
    return 0;
pid = fork(); // A->B
if(pid == 0) {
    ret = execl(cmd, cmd, NULL);
    if(ret == -1) {
        perror("execution failed");
        exit(errno);
    }
} else {
    wait(0);
}
```

- During fork()
  - Pages of parent were duplicated
  - Equal amount of page frames were allocated
  - Page table for child differed from parent (as it has another set of frames)
- In exec()
  - The page frames of child were taken away and new frames were allocated
  - Child’s page table was rebuilt!
- Waste of time during fork() if the exec() was to be called immediately

An optimization: Copy on write

- Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call exec()
  - Very efficient

Before Process 1 Modifies Page C

After Process 1 Modifies Page C

Challenges and improvements in implementation

- Choice of backing store

  For stack, heap pages: on swap partition

  For code, shared library? : swap partition or the actual executable file on the file-system?

  If the choice is file-system for code, then the page-fault handler needs to call functions related to file-system

- Is the page table itself pagable?

  If no, good

  If Yes, then there can be page faults in accessing the page tables themselves! More complicated!

- Is the kernel code pagable?

  If no, good

  If yes, life is very complicated ! Page fault in running kernel code, interrupt handlers, system calls, etc.

Page replacement

Review

- Concept of virtual memory, demand paging.
- Page fault
- Performance degradation due to page fault: Need to reduce #page faults to a minimum
- Page fault handling process, broad steps: (1) Trap (2) Locate on disk (3) find free frame (4) schedule disk I/O (5) update page table (6) resume
- More on (3) today

List of free frames

- Kernel needs to maintain a list of free frames
- At the time of loading the kernel, the list is created
- Frames are used for allocating memory to a process

  But may also be used for managing kernel’s own data structures also
- More processes --> more demand for frames

What if no free frame found on page fault?

- Page frames in use depends on “Degree of multiprogramming”

  More multiprogramming -> overallocation of frames

  Also in demand from the kernel, I/O buffers, etc

  How much to allocate to each process? How many processes to allow?

- Page replacement – find some page(frame) in memory, but not really in use, page it out

  Questions : terminate process? Page out process? replace the page?

  For performance, need an algorithm which will result in minimum number of page faults

- Bad choices may result in same page being brought into memory several times

Need for Page Replacement

Figure 10.9 Need for page replacement.

Page replacement

- Strategies for performance

  Prevent over-allocation of memory by modifying page-fault service routine to include page replacement

  Use modify (dirty) bit in page table. To reduce overhead of page transfers – only modified pages are written to disk. If page is not modified, just reuse it (a copy is already there in backing store)

Basic Page replacement

1) Find the location of the desired page on disk
2) Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a victim frame & write victim frame to disk if dirty
3) Bring the desired page into the free frame; update the page table of process and global frame table/list
4) Continue the process by restarting the instruction that caused the trap

- Note now potentially 2 page transfers for page fault – increasing EAT

Page Replacement

Two problems to solve

- Frame-allocation algorithm determines
  - How many frames to give each process
  - Which frames to replace
- Page-replacement algorithm
  - Want lowest page-fault rate on both first access and re-access

Evaluating algorithm: Reference string

- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
- In all our examples, the reference string is 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

An Expectation

More page Frames Means less faults

FIFO Algorithm

reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
	0	0	0		3	3	3	2	2	2			1	1			1	0	0
		1	1		1	0	0	0	3	3			3	2			2	2	1

page frames

FIFO Algorithm

1	7	2	4	0	7
2	0	3	2	1	0
3	1	0	3	2	1

15 page faults

- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (3 pages can be in memory at a time per process)
- **Belady's Anomaly**
- Adding more frames can cause more page faults!
- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5

FIFO Algorithm: Balady's anamoly

Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal #replacements for the example on the next slide
- How do you know this?
  - Can’t read the future
- Used for measuring how well your algorithm performs

Optimal page replacement

reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1

7	7	7	2		2		2			2			2				7		
	0	0	0		0		4			0			0				0		
		1	1		3		3			3			1				1		

page frames

Least Recently Used: an approximation of optimal

Use past knowledge rather than future
Replace page that has not been used in the most amount of time
Associate time of last use with each page
12 faults – better than FIFO but worse than OPT
Generally good algorithm and frequently used
But how to implement?

LRU: Counter implementation

- **Counter implementation**
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed

LRU: Stack implementation

- Keep a stack of page numbers in a double link form:
- Page referenced: move it to the top
  - requires 6 pointers to be changed and
  - each update more expensive
  - But no need of a search for replacement

Use Of A Stack to Record The Most Recent Page References

Stack algorithms

- An algorithm for which it can be shown that the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with n + 1 frames
- Do not suffer from Balady’s anamoly
- For example: Optimal, LRU

LRU: Approximation algorithms

- LRU needs special hardware and still slow
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
  - We do not know the order, however

LRU: Approximation algorithms

- **Second-chance algorithm**
  - FIFO + hardware-provided reference bit. If bit is 0 select, if bit is 1, then set it to 0 and move to next one.
- **An implementation of second-chance: Clock replacement**
  - If page to be replaced has
  - Reference bit = 0 -> replace it
  - reference bit = 1 then:
    - set reference bit 0, leave page in memory
    - replace next page, subject to same rules

Second-Chance (clock) Page-Replacement Algorithm

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common
- **LFU Algorithm**: replaces page with smallest count
- **MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Page buffering algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

Major and Minor page faults

- Most modern OS refer to these two types
- Major fault
  - Fault + page not in memory
- Minor fault
  - Fault, but page is in memory
  - For example shared memory pages; second instance of fork(), page already on free-frame list,
- On Linux run
  - `$ ps -eo min_flt,maj_flt,cmd`

Special rules for special applications

- All of earlier algorithms have OS guessing about future page access
- But some applications have better knowledge – e.g. databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can given direct access to the disk, getting out of the way of the applications
  - Raw disk mode
- Bypasses buffering, locking, etc

Allocation of frames

- Each process needs *minimum* number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- Maximum of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

Fixed allocation of frames

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames

  Keep some as free frame buffer pool

- Proportional allocation – Allocate according to the size of process

  Dynamic as degree of multiprogramming, process sizes change

s_i = size of process p_i

$S = \sum s_i$

m = total number of frames

a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 64$

$s_1 = 10$

$s_2 = 127$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority Allocation of frames

- Use a proportional allocation scheme using priorities rather than size
- If process Pi generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

Global Vs Local allocation

- **Global replacement –** process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common
- **Local replacement –** each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

Virtual Memory – Remaining topics

Agenda

- Problem of Thrashing and possible solutions
- Mmap(), Memory mapped files
- Kernel Memory Management
- Other Considerations

Thrashing

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
- This leads to:
  - Low CPU utilization
  - Operating system thinking that it needs to increase the degree of multiprogramming
  - Another process added to the system
- Thrashing : a process is busy swapping pages in and out

Thrashing

Demand paging and thrashing

- Why does demand paging work?
  - Locality model
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?
  - size of locality > total memory size
  - Limit effects by using local or priority page replacement

Locality In A Memory-Reference Pattern

Working set model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  - Example: 10,000 instructions
- Working Set Size, WSS_i (working set of Process P_i) =
  - total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program

Working set model

- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - Approximation of locality
- if $D > m$ (total available frames) $\Rightarrow$ Thrashing
- Policy if $D > m$, then suspend or swap out one of the processes

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000

  Timer interrupts after every 5000 time units

  Keep in memory 2 bits for each page

  Whenever a timer interrupts copy (to memory) and sets the values of all reference bits to 0

  If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

Page fault frequency

- More direct approach than WSS
- Establish “acceptable” page-fault frequency rate and use local replacement policy

  If actual rate too low, process loses frame

  If actual rate too high, process gains frame

Working Sets and Page Fault Rates

Memory Mapped Files

Memory-Mapped Files

- First, let’s see a demo of using mmap()

Memory-Mapped Files

Use MAP_SHARED flag

Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory
- A file is initially read using demand paging
  - A page-sized portion of the file is read from the file system into a physical page
  - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies and speeds file access by driving file I/O through memory rather than read() and write() system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- But when does written data make it to disk?
  - Periodically and / or at file close() time
  - For example, when the pager scans for dirty pages

Memory-Mapped Files

- Some OSes uses memory mapped files for standard I/O
- Process can explicitly request memory mapping a file via mmap() system call
    - Now file mapped into process address space
- For standard I/O (open(), read(), write(), close()), mmap anyway
    - But map file into kernel address space
    - Process still does read() and write()
        - Copies data to and from kernel space and user space
    - Uses efficient memory management subsystem
        - Avoids needing separate subsystem
- COW can be used for read/write non-shared pages
- Memory mapped files can be used for shared memory (although again via separate system calls)

Allocating Kernel Memory

Allocating kernel memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
  - I.e. for device I/O
-

Buddy Allocator

Buddy Allocator

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using power-of-2 allocator
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
  - Continue until appropriate sized chunk available

Buddy Allocator

- Continue until appropriate sized chunk available
- For example, assume 256KB chunk available, kernel requests 21KB

  Split into AL and Ar of 128KB each

    One further divided into BL and BR of 64KB

      One further into CL and CR of 32KB each – one used to satisfy request
- Advantage – quickly coalesce unused chunks into larger chunk
- Disadvantage - fragmentation

Slab Allocator

Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

Other considerations

Other Considerations -- Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume s pages are prepaged and α of the pages is used

  Is cost of $s * \alpha$ save pages faults > or < than the cost of prepaging

  $s * (1- \alpha)$ unnecessary pages?

  α near zero –––> prepaging loses

Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - **Resolution**
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time

TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB
  Otherwise there is a high degree of page faults
- Increase the Page Size
  This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
  This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

Program Structure

- Program structure
- `Int[128,128] data;`
- Each row is stored in one page
- Program 1

```
for (j = 0; j <128; j++)
     for (i = 0; i < 128; i++)
          data[i,j] = 0;
```

128 x 128 = 16,384 page faults

- Program 2

```
for (i = 0; i < 128; i++)
     for (j = 0; j < 128; j++)
          data[i,j] = 0;
```

128 page faults

I/O Interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm

Threads, Signals

Abhijit A M
abhijit.comp@coep.ac.in

Threads

- thread — a fundamental unit of CPU utilization
  - A separate control flow within a program
  - set of instructions that execute “concurrently” with other parts of the code
  - Note the difference: Concurrency: progress at the same time, Parallel: execution at the same time
- Threads run within application
  - An application can be divided into multiple parts
  - Each part may be written to execute as a threads
- Let’s see an example

Threads

- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight, due to the very nature of threads
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Single vs Mulththreaded process

A mulththreaded server

Benefits of threads

- Responsiveness
- Resource Sharing
- Economy
- Scalability

Single vs Multicore systems

Single core : Concurrency possible

Multicore : parallel execution possible

Multicore programming

- Multicore systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
  -

User vs Kernel Threads

- User Threads: Thread management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads
- Kernel Threads: Supported by the Kernel
- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

User threads vs Kernel Threads

- User threads
  - User level library provides a "typedef" called threads
  - The scheduling of threads needs to be implemented in the user level library
  - Need some type of timer handling functionality at user level execution of CPU
    - OS needs to provide system calls for this
  - Kernel does not know that there are threads!

- Kernel Threads
  - Kernel implements concept of threads
  - Still, there may be a user level library, that maps kernel concept of threads to "user concept" since applications link with user level libraries
  - Kernel does scheduling!

Mulththreading models

- How to map user threads to kernel threads?
  - Many-to-One
  - One-to-One
  - Many-to-Many
- What if there are no kernel threads?
  - Then only “one” process. Hence many-one mapping possible, to be done by user level thread library
  - Is One-One possible?

Many-One Model

- **Many user-level threads mapped to single kernel thread**
- **Examples:**
  - **Solaris Green Threads**
  - **GNU Portable Threads**

One-One Model

- **Each user-level thread maps to kernel thread**
- **Examples**
  - **Windows NT/XP/2000**
  - **Linux**
  - **Solaris 9 and later**

Many-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the ThreadFiber package

Two Level Model

- Similar to M:M, except that it allows a user thread to be bound to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

Thread Libraries

Thread libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
-

Demo of pthreads code

Demonstration on Linux – see the code, compile and execute it.

Other libraries

- Windows threading API
  - CreateThread(...)
  - WaitForSingleObject(...)
  - CloseHandle(...)
- Java Threads
  - The Threads class
  - The Runnable Interface

Issues with threads

- **Semantics of fork() and exec() system calls**
  - **Does fork() duplicate only the calling thread or all threads?**
- **Thread cancellation of target thread**
  - **Terminating a thread before it has finished**
  - **Two general approaches:**
    - **Asynchronous cancellation terminates the target thread immediately.**
    - **Deferred cancellation allows the target thread to periodically check if it should be cancelled.**

More on threads

Thread pools

- **Some kernels/libraries can provide system calls to : Create a number of threads in a pool where they await work, assign work/function to a waiting thread**
- **Advantages:**
  - **Usually slightly faster to service a request with an existing thread than create a new thread**
  - **Allows the number of threads in the application(s) to be bound to the size of the pool**

Thread Local Storage (TLS)

- **Thread-specific data, Thread Local Storage (TLS)**
  - Not local, but global kind of data for all functions of a thread, more like "static" data
  - Create Facility needed for data private to thread
  - Allows each thread to have its own copy of data
  - Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
  - gcc compiler provides the storage class keyword thread for declaring TLS data

```
static __thread int threadID;
```

```
int arr[16];
int f() {
    a(); b(); c();
}
int g() {
    x(); y();
}
int main() {
    th_create(...,f,...);
    th_create(...,g,...);
}
//arr is visible to all of them!
//need data for only f,a,b,c
//need data for only g,x,y
```

Scheduler activations for threads

Library

```
--
th_setup(int n) {
    max_threads = n;
    curr_threads = 0;
}
th_create(...., fn,....) {
    if(curr_threads < max_threads)
        create kernel thread;
    schedule fn on one of the kernel
threads;
}
```

application

```
---
f() {
    scanf();
}
g() {
    recv();
}
h() {...}; i() {...}
main()
    th_setup(2);
    th_create(...,f,...);
    th_create(...,g,...);
    th_create(...,h,...);
    th_create(...,i,...);
}
```

Scheduler activations for threads

- **Scheduler Activations**
  - **Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application**
  - **Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library**
  - **This communication allows an application to maintain the correct number kernel threads**

Issues with threads

- **Scheduler Activations: LWP approach**
  - **An intermediate data structure LWP**
  - **appears to be a virtual processor on which the application can schedule a user thread to run.**
  - **Each LWP attached to a kernel thread**
  - **Typically one LWP per blocking call, e.g. 5 file I/Os in one process, then 5 LWPs needed**

Issues with threads

- **Scheduler Activations: LWP approach**
  - **Kernel needs to inform application about events like: a thread is about to block, or wait is over**
  - **This will help application relinquish the LWP or request a new LWP**

Issues with threads

- **The actual upcalls**

Linux threads

- Only threads (called task), no processes!
- Process is a thread that shares many particular resources with the parent thread
- Clone() system call to create a thread

Linux threads

- clone() takes options to determine sharing on process create
- struct task_struct points to process data structures (shared or unique depending on clone options)
- fork() is a wrapper on top of clone()

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Issues in implementing threads project

- How to implement a user land library for threads?
- How to handle 1-1, many-one, many-many implementations?
- Identifying the support required from OS and hardware
- Identifying the libraries that will help in implementation

Issues in implementing threads project

- Understand the clone() system call completely
  - Try out various possible ways of calling it
  - Passing different options
  - Passing a user-land buffer as stack
- How to save and restore context?
  - C: setjmp, longjmp
  - Setcontext, getcontext(), makecontext(), swapcontext() functions
- Sigaction is more powerful than signal
  - Learn SIGALRM handling for timer and scheduler, timer_create() & timer_stop() system calls
- Customized data structure to store threads, and manage thread-lists for scheduling

Signals

Signals

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- Signal handling
  - Synchronous and asynchronous
- A signal handler (a function) is used to process signals
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Then, signal is “handled” by the handler

Signals

- **More about signals**
  - **Different signals are typically identified as different numbers**
  - **Operating systems provide system calls like kill() and signal() to enable processes to deliver and receive signals**
  - **Signal() - is used by a process to specify a “signal handler” – a code that should run on receiving a signal**
  - **Kill() is used by a process to send another process a signal**
  - **There are restrictions on which process can send which signal to other processes**

Demo

- Let’s see a demo of signals with respect to processes
- Let’s see signal.h
  - /usr/include/signal.h
  - /usr/include/asm-generic/signal.h
  - /usr/include/linux/signal.h
  - /usr/include/sys/signal.h
  - /usr/include/x86_64-linux-gnu/asm/signal.h
  - /usr/include/x86_64-linux-gnu/sys/signal.h
- man 7 signal
- Important signals: SIGKILL, SIGUSR1, SIGSEGV, SIGALRM, SIGCLD, SIGINT, SIGPIPE, ...

Signal handling by OS

Process 12323 {

signal(19, abcd);

}

OS: sys_signal {

Note down that process 12323 wants to handle signal number 19 with function abcd

}

Process P1 {

kill (12323, 19) ;

}

OS: sys_kill {

Note down in PCB of process 12323 that signal number 19 is pending for you.

}

When process 12323 is scheduled, at that time the OS will check for pending signals, and invoke the appropriate signal handler for a pending signal.

Threads and Signals

- **Signal handling Options:**
  - **Deliver the signal to the thread to which the signal applies**
  - **Deliver the signal to every thread in the process**
  - **Deliver the signal to certain threads in the process**
  - **Assign a specific thread to receive all signals for the process**

Some numbers and their ‘meaning’

- These numbers occur very frequently in discussion
- 0x 80000000 = 2 GB = KERNBASE
- 0x 100000 = 1 MB = EXTMEM
- 0x 80100000 = 2GB + 1MB = KERNLINK
- 0x E000000 = 224 MB = PHYSTOP
- 0x FE000000 = 3.96 GB = 4064 MB = DEVSPACE
  - 4096 – 4064 = 32 MB left on top

X86 Memory Management

X86 address : protected mode address translation

X86 paging

Figure 8.23 Paging in the IA-32 architecture.

Segmentation + Paging

Selector value is implicit based on address being accessed.
Instruction: CS
Stack Variable: SS
Data: DS
etc.

GDT Entry

31 16			15 0	
Base 0:15			Limit 0:15	
63 56	55 52	51 48	47 40	39 32
Base 24:31	Flags	Limit 16:19	Access Byte	Base 16:23

Page Directory Entry (PDE)
Page Table Entry (PTE)

PDE

PTE

P	Present
W	Writable
U	User
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use

Segment selector

TI Table index (0=GDT, 1=LDT)
RPL Requester privilege level

EFLAGS register

Status flags
- CF Carry flag
- PF Parity flag
- AF Auxiliary carry flag
- ZF Zero flag
- SF Sign flag
- OF Overflow flag

Control flags
- DF Direction flag

System flags
- TF Trap flag
- IF Interrupt enable flag
- IOPL I/O privilege level
- NT Nested task
- RF Resume flag
- VM Virtual-8086 mode
- AC Alignment check
- VIF Virtual intr. flag
- VIP Virtual intr. pending
- ID ID flag

CR0

PE	Protection enabled	ET	Extension type	NW	Not write-through
MP	Monitor coprocessor	NE	Numeric error	CD	Cache disable
EM	Emulation	WP	Write protect	PG	Paging
TS	Task switched	AM	Alignment mask		

PG: Paging enabled or not **WP: Write protecion on/off**
PE: Protection Enabled --> protected mode.

CR2

CR3

PWT Page-level writes transparent

PCT Page-level cache disable

CR4

VME	Virtual-8086 mode extensions	MCE	Machine check enable
PVI	Protected-mode virtual interrupts	PGE	Page-global enable
TSD	Time stamp disable	PCE	Performance counter enable
DE	Debugging extensions	OSFXSR	OS FXSAVE/FXRSTOR support
PSE	Page size extensions	OSXMM-EXCPT	OS unmasked exception support
PAE	Physical-address extension		

mmu.h : paging related macros

```
#define PTXSHIFT        12      // offset of PTX in a linear address
#define PDXSHIFT        22      // offset of PDX in a linear address
#define PDX(va)   (((uint)(va) >> PDXSHIFT) & 0x3FF)// page directory index
#define PTX(va)   (((uint)(va) >> PTXSHIFT) & 0x3FF)// page table index
// construct virtual address from indexes and offset
#define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT |
(o)))
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table   | Offset within Page  |
// |      Index     |      Index     |                     |
// +----------------+----------------+---------------------+
//  \--- PDX(va) --/ \--- PTX(va) --/
```

mmu.h : paging related macros

```
// Page directory and page table constants.
#define NPDENTRIES      1024    // # directory
entries per page directory
#define NPTENTRIES      1024    // # PTEs per
page table
#define PGSIZE          4096    // bytes mapped
by a page
#define PGROUNDUP(sz)  (((sz)+PGSIZE-1) &
~(PGSIZE-1))
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
```

mmu.h : paging related macros

```
// Page table/directory entry flags.
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_PS          0x080   // Page Size

// Address in page table or page directory entry
#define PTE_ADDR(pte)   ((uint)(pte) & ~0xFFF) // get
all but last 12 bits
#define PTE_FLAGS(pte)  ((uint)(pte) &  0xFFF) // get
last 12 bits
```

mmu.h : Segmentation related macros

```
// various segment selectors.
#define SEG_KCODE 1  // kernel code
#define SEG_KDATA 2  // kernel data+stack
#define SEG_UCODE 3  // user code
#define SEG_UDATA 4  // user data+stack
#define SEG_TSS   5  // this process's task state
```

mmu.h : Segmentation related macros

```
// various segment selectors.
#define SEG_KCODE 1  // kernel code
#define SEG_KDATA 2  // kernel data+stack
#define SEG_UCODE 3  // user code
#define SEG_UDATA 4  // user data+stack
#define SEG_TSS   5  // this process's task state
#define NSEGS     6
```

mmu.h : Segmentation related macros

```
struct segdesc { // 64 bit in size
  uint lim_15_0 : 16;  // Low bits of segment limit
  uint base_15_0 : 16; // Low bits of segment base address
  uint base_23_16 : 8; // Middle bits of segment base address
  uint type : 4;       // Segment type (see STS_ constants)
  uint s : 1;          // 0 = system, 1 = application
  uint dpl : 2;        // Descriptor Privilege Level
  uint p : 1;          // Present
  uint lim_19_16 : 4;  // High bits of segment limit
  uint avl : 1;        // Unused (available for software use)
  uint rsv1 : 1;       // Reserved
  uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
  uint g : 1;          // Granularity: limit scaled by 4K when set
  uint base_31_24 : 8; // High bits of segment base address
};
```

mmu.h : Segmentation related code

```
// Application segment type bits
#define STA_X       0x8     // Executable segment
#define STA_W       0x2     // Writeable (non-executable segments)
#define STA_R       0x2     // Readable (executable segments)

// System segment type bits
#define STS_T32A    0x9     // Available 32-bit TSS
#define STS_IG32    0xE     // 32-bit Interrupt Gate
#define STS_TG32    0xF     // 32-bit Trap Gate
```

Code from bootasm.S bootmain.c is over!
Kernel is loaded.
Now kernel is going to prepare itself

main() in main.c

- Initializes “free list” of page frames
  - In 2 steps. Why?
- Sets up page table for kernel
- Detects configuration of all processors
- Starts all processors
  - Just like the first processor
- Creates the first process!

- Initializes
  - LAPIC on each processor, IOAPIC
  - Disables PIC
  - “Console” hardware (the standard I/O)
  - Serial Port
  - Interrupt Descriptor Table
  - Buffer Cache
  - Files Table
  - Hard Disk (IDE)

main() in main.c

```
int

main(void) {

  kinit1(end, P2V(4*1024*1024)); // phys page allocator

  kvmalloc();      // kernel page table
```

```
void

kinit1(void *vstart, void *vend) {

  initlock(&kmem.lock, "kmem");

  kmem.use_lock = 0;

  freerange(vstart, vend);

}
```

main() in main.c

```
void

freerange(void *vstart, void
*vend)

{

  char *p;

  p =
(char*)PGROUNDUP((uint)vst
art);

  for(; p + PGSIZE <=
(char*)vend; p += PGSIZE)

    kfree(p);

}
```

```
kfree(char *v) {
  struct run *r;
  if((uint)v % PGSIZE || v <
end || V2P(v) >= PHYSTOP)
    panic("kfree");
  // Fill with junk to catch
dangling refs.
  memset(v, 1, PGSIZE);
  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
  if(kmem.use_lock)
    release(&kmem.lock); }
```

Free List in XV6 Obtained after main() -> kinit1()

Pages obtained Between
end = 801154a8 = 2049 MB to P2V(4MB) = 2052 MB
Remember
Right now Logical = Physical address.

Actually like this in memory

kmem

Seen independently

Back to main()

```
int
main(void) {
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
```

```
// Allocate one page table for the machine for the kernel address
// space for scheduler processes.
void
kvmalloc(void)
{
  kpgdir = setupkvm();
  switchkvm();
}
```

Back to main()

```
int
main(void) {
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
```

```
// Allocate one page table for the machine for the kernel address
// space for scheduler processes.
void
kvmalloc(void)
{
  kpgdir = setupkvm(); // global var kpgdir
  switchkvm();
}
```

```
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;
  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (P2V(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0) {
      freevm(pgdir);
      return 0;
    }
  return pgdir;
}
```

```
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {
 { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
 { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
};
```

kmap[] mappings done in kvmalloc(). This shows segmentwise, entries are done in page directory and page table for corresponding VA-> PA mappings

0x80108000 =data = 2049.3125 MB Is obtained from kernel.sym

Remidner: PDE and PTE entries

PDE

PTE

P	Present
W	Writable
U	User
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use

Before mappages()

What mappages() Does

After kvmalloc() in main()
RAM
Physical Addr
Linear Address
Logical Address = offset
Dir
pg
Offset
4MB
0
CS, SS, etc.
Selector
Base
Limit
Permissions
3
2
1
0
0
4GB
Write
0
4GB
Read, Execute
0
0
0
GDT
DS
SS
CS
CR3
kpgdir
Page Table
Now Linear Address = Logical Address != Physical Address

main()->seginit()

- Re-initialize GDT
- Once and forever now
- Just set 4 entries
  - All spanning 4 GB
  - Differing only in permissions and privilege level

After seginit() in main().
On the processor where we started booting
RAM
Physical Addr
Linear Address
Logical Address = offset
Dir
pg
Offset
4MB
0
CS, SS, etc.
Selector
Base
Limit
Permissions
DPL
4
0
4GB
Write
3
3
0
4GB
Read, Execute
3
2
0
4GB
Write
0
1
0
4GB
Read, Execute
0
0
0
0
0
DS
SS
CS
GDT
Page Table
CR3
kpgdir
Now Linear Address = Logical Address != Physical Address

After seginit()

- While running kernel code, necessary to switch CS, DS, SS to index 1,2,2 in GDT
- While running user code, necessary to switch CS, DS, SS to index 3,4,4 in GDT
- This happens automatically as part of “trap” handling (covered separately)

Memory Management

X86 page table hardware

Layout of process's VA space

Memory Layout of a user process

After exec()

Note the argc, argv on stack

Memory Layout of a user process

The "guard page" is just a mapping in page table. No frame allocated. It's marked as invalid. So if stack grows (due to many function calls), then OS will detect it with an exception

Memory Layout of a user process

On sbrk()
The system call to grow process's address space.
Calls growproc()

growproc()
Allocate a frame, Add an entry in page table at the top (above proc->sz)
//This entry can't go beyond KERNBASE
Calls switchuvm()

Switchuvm()
Ultimately loads CR3, invalidating cache

Free List in XV6

Actually like this in memory

kmem

Seen independently

exec()

- **sys_exec()**

  exec(path, argv)

- **exec(parth, argv)**

```
ip = namei(path))
readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf)
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
}
```

exec()

- **exec(parth, argv)**

```
// Allocate two pages at the next page boundary.
 // Make the first inaccessible.  Use the second as the user stack.
 sz = PGROUNDUP(sz);
 if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
// Push argument strings, prepare rest of stack in ustack.
  for(argc = 0; argv[argc]; argc++) {
    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
```

Creation of first process by kernel

Why first process needs ‘special’ treatment?

- Normally process is created using fork()
  - and typically followed by a call to exec()
- Fork will use the PCB of existing process to create a new process
  - as a clone
- The first process has nothing to copy from!
- So it’s PCB needs to “built” by kernel code

Why first process needs ‘special’ treatment?

- XV6 approach
  - Create the process as if it was created by “fork”
  - Ensure that the process starts in a call to “exec”
  - Let “Exec” do the rest of the JOB as expected
  - In this case exec() will call
    - exec(“/init”, NULL);
- See the code of init.c
  - opens console() device for I/O; dups 0 on 1 and 2!
    - Same device file for I/O
  - forks a process and execs (“sh”) on it.
  - Itself keeps waiting for zombie processes

Why first process needs ‘special’ treatment?

- What needs to be done ?
  - Build struct proc by hand
  - How data structures (proc, stack, etc) are hand-crafted so that when kernel returns, the process starts in code of init

Imp Concepts

- A process has two stacks
  - user stack: used when user code is running
  - kernel stack: used when kernel is running on behalf of a process
- Note: there is a third stack also!
  - The kernel stack used by the scheduler itself
  - Not a per process stack

Imp Concepts

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

struct proc diagram: Very imp!

sz = ELF-code->memsz (includes data, check “ld -N”
+ 2*4096 (for stack)

setupkvm()
does this mapping

These mappings need to be created per process

Memory Layout of a user process After exec()

Note the argc, argv on stack

stack is just one page.

size of text and data is derived from ELF file

main()->userinit()
Creating first process by hand

- Code of the first process
  - initcode.S and init.c
  - init.c is compiled into “/init” file
    - During make !
  - Trick:
    - Use initcode.S to “exec(“/init”)”
    - And let exec() do rest of the job
  - But before you do exec()
    - Process must exist as if it was forked() and running

main()->userinit()
Creating first process by hand

```
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  // Abhijit: obtain proc 'p', with stack initialized
  // and trapframe created and eip set to 'forkret'
  p = allocproc();
// let’s see what allocproc() does
```

First process creation
Let's revisit struct proc

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state. allocated, ready to run, running, wait-
ing for I/O, or exiting.
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process. Process's context
  void *chan;                  // If non-zero, sleeping on chan. More when we discuss sleep, wakeup
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files, used by open(), read(),...
  struct inode *cwd;           // Current directory, changed with "chdir()"
  char name[16];               // Process name (for debugging)
};
```

struct proc diagram

sz = ELF-code->memsz (includes data, check “ld -N”
+ 2*4096 (for stack)

allocproc()

```
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;
  acquire(&ptable.lock);
  for(p = ptable.proc; p <
&ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;
  release(&ptable.lock);
  return 0;
```

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;

release(&ptable.lock);
```

allocproc() setting up stack

```
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;
// Abhijit KSTCKSIZE = PGSIZE
  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;
  // Set up new context to start executing at
forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;
  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
```

allocproc() setting up stack

```
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;
// Abhijit KSTCKSIZE = PGSIZE
  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;
  // Set up new context to start executing at
forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;
  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
```

allocproc() setting up stack

```
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;
// Abhijit KSTCKSIZE = PGSIZE
  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;
  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;
  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
```

allocproc() setting up stack

```
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;
// Abhijit KSTCKSIZE = PGSIZE
  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;
  // Set up new context to start executing at
forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;
  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
```

allocproc() setting up stack

```
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;
// Abhijit KSTCKSIZE = PGSIZE
  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;
  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;
  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
```

Next in userinit()

```
  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S
  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");
  acquire(&ptable.lock);
  p->state = RUNNABLE;
  release(&ptable.lock);
```

After userinit()

Q: why there is no user stack yet?

main()->mpmain()

```
static void
mpmain(void)
{
  cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
  idtinit();       // load idt register
  xchg(&(mycpu()->started), 1); // tell startothers() we're up
  scheduler();     // start running processes
}
```

- **Load IDT register**
  - Copy from idt[] array into IDTR
- **Call scheduler()**
  - One process has already been made runnable
  - Let’s enter scheduler now

Before reading scheduler(): Note

- The esp is still pointing to the stack which was allocated in entry.S !
  - this is the kernel only stack
  - Not the per process kernel stack.
- CR3 points to kpgdir
- Struct cpu[ ] has been setup up already
  - apicid – in mpinit()
  - segdesc gdt – in seginit()
  - started – in mpmain()
- Fields in cpu[] not yet set
  - context * scheduler --> will be setup in sched()
  - taskstate ts --> large structure, only parts used in switchuvm()
  - ncli, intena --> used while locking
  - proc *proc -> set during scheduler()

scheduler()

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
     sti();
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
     // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
```

scheduler() called first time
Code/stack of "initcode"
proc
cpu *c
p
proc
context
kstack
tf
sz = 4096
name = "initcode"
pgdir
cwd
state = RUNNALBE
Trapframe
CS =3
DS = ES = SS =4
EFLAGS = FL_IF
ESP = 4096
EIP = 0
trapret()
eip=forkret()
ebp
ebx
esi
edi
Page table
Page table
Page table
Toward s Kernel pages
pgdir
kpgdir
CR3
stack
entry.S
esp
Inode for "/"

scheduler()

```
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE)
    continue;

  // Switch to chosen process.  It is the process's job
  // to release ptable.lock and then reacquire it
  // before jumping back to us.
  c->proc = p;
  switchuvm(p);
  p->state = RUNNING;
```

after
switchuvm()
in schedul
Code/stack
of
“initcode”
proc
cpu
*c
p
proc
context
kstack
tf
sz = 4096
name = “initcode”
pgdir
cwd
state = RUNNING
Trapframe
CS =3
DS = ES = SS =4
EFLAGS = FL_IF
ESP = 4096
EIP = 0
trapret()
eip=forkret()
ebp
ebx
esi
edi
Page table
Page table
Kernel
pages
Page table
pgdir
CR3
kpgdir
stack
entry.S
esp
Inode
for “/”

scheduler()

```
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE)
    continue;

  // Switch to chosen process.  It is the process's job
  // to release ptable.lock and then reacquire it
  // before jumping back to us.
  c->proc = p;
  switchuvm(p);
  p->state = RUNNING
  swtch(&(c->scheduler), p->context);
;
```

at call to swtch()
Code/stack of "initcode"
proc
cpu *c
p
proc
context
kstack
tf
sz = 4096
name = "initcode"
pgdir
cwd
state = RUNNING
Trapframe
CS =3
DS = ES = SS =4
EFLAGS = FL_IF
ESP = 4096
EIP = 0
trapret()
eip=forkret()
ebp
ebx
esi
edi
Page table
Page table
Page table
Toward s Kernel pages
pgdir
CR3
kpgdir
addr of p->context
--
addr of c->scheduler
--
ret value in scheduler()
esp
Inode for "/"

swtch

```
swtch:
  #Abhijit: swtch was called through a function call.
  #So %eip was saved on stack already
  movl 4(%esp), %eax    # Abhijit: eax = old
  movl 8(%esp), %edx    # Abhijit: edx = new
```

during swtch()
Code/stack of "initcode"
proc
cpu *c
p
proc
context
kstack
tf
sz = 4096
name = "initcode"
pgdir
cwd
state = RUNNING
Trapframe
CS =3
DS = ES = SS =4
EFLAGS = FL_IF
ESP = 4096
EIP = 0
trapret()
eip=forkret()
ebp
ebx
esi
edi
Page table
Page table
Page table
Toward s Kernel pages
pgdir
CR3
kpgdir
addr of p->context
--
addr of c->scheduler
--
ret value in scheduler()
esp
Inode for "/"
eax = &c->scheduler
edx = &p->context

swtch

```
swtch:
  #Abhijit: swtch was called through a function call.
  #So %eip was saved on stack already
  movl 4(%esp), %eax    # Abhijit: eax = old
  movl 8(%esp), %edx    # Abhijit: edx = new
  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi            # Abhijit: esp = esp + 16
```

during swtch()
Code/stack of "initcode"
proc
cpu *c
p
proc
context
kstack
tf
sz = 4096
name = "initcode"
pgdir
cwd
state = RUNNING
Page table
Page table
Page table
Toward s Kernel pages
pgdir
CR3
kpgdir
addr of p->context
addr of c->scheduler
ret value in scheduler()
ebp,ebx, esi,edi
esp
Inode for "/"
Trapframe
CS =3
DS = ES = SS =4
EFLAGS = FL_IF
ESP = 4096
EIP = 0
trapret()
eip=forkret()
ebp
ebx
esi
edi
eax = &c->scheduler
edx = &p->context

swtch

```
swtch:
  #Abhijit: swtch was called through a function call.
  #So %eip was saved on stack already
  movl 4(%esp), %eax    # Abhijit: eax = old
  movl 8(%esp), %edx    # Abhijit: edx = new
  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi            # Abhijit: esp = esp + 16
  # Switch stacks
  movl %esp, (%eax)     # Abhijit: *old = updated old stack
  movl %edx, %esp       # Abhijit: esp = new
```

during swtch()
Code/stack of "initcode"
p
proc
context
kstack
tf
sz = 4096
name = "initcode"
pgdir
cwd
state = RUNNING
proc
cpu *c
Trapframe
CS =3
DS = ES = SS =4
EFLAGS = FL_IF
ESP = 4096
EIP = 0
trapret()
eip=forkret()
ebp
ebx
esi
edi
Page table
Page table
Page table
Toward s Kernel pages
pgdir
CR3
kpgdir
addr of p->context
addr of c->scheduler
ret value in scheduler()
ebp,ebx, esi,edi
esp
Inode for "/"
c->scheduler
edx = &p->context

swtch

```
swtch:
  #Abhijit: swtch was called through a function call.
  #So %eip was saved on stack already
  movl 4(%esp), %eax    # Abhijit: eax = old
  movl 8(%esp), %edx    # Abhijit: edx = new
  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi            # Abhijit: esp = esp + 16
  # Switch stacks
  movl %esp, (%eax)     # Abhijit: *old = updated old stack
  movl %edx, %esp       # Abhijit: esp = new
  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp     # Abhijit: newesp = newesp - 16, context restored
```

during swtch()
Code/stack of "initcode"
p
proc
context
kstack
tf
sz = 4096
name = "initcode"
pgdir
cwd
state = RUNNING
proc
cpu *c
Trapframe
CS =3
DS = ES = SS =4
EFLAGS = FL_IF
ESP = 4096
EIP = 0
trapret()
eip=forkret()
ebp
ebx
esi
edi
Page table
Page table
Page table
Toward s Kernel pages
pgdir
CR3
kpgdir
addr of p->context
addr of c->scheduler
ret value in scheduler()
ebp,ebx, esi,edi
esp
Inode for "/"
c->scheduler
edx = &p->context

swtch

```
swtch:
  #Abhijit: swtch was called through a function call.
  #So %eip was saved on stack already
  movl 4(%esp), %eax    # Abhijit: eax = old
  movl 8(%esp), %edx    # Abhijit: edx = new
  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi            # Abhijit: esp = esp + 16
  # Switch stacks
  movl %esp, (%eax)     # Abhijit: *old = updated old stack
  movl %edx, %esp       # Abhijit: esp = new
  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp      # Abhijit: newesp = newesp - 16, context restored
  ret            # Abhijit: will pop from esp now -> function where to
return.
```

after "ret" from swtch() just before forkret()
Code/stack of "initcode"
p
proc
context
kstack
tf
sz = 4096
name = "initcode"
pgdir
cwd
state = RUNNING
proc
cpu *c
Trapframe
CS =3
DS = ES = SS =4
EFLAGS = FL_IF
ESP = 4096
EIP = 0
trapret()
Page table
Page table
Page table
Toward s Kernel pages
pgdir
CR3
kpgdir
addr of p->context
addr of c->scheduler
ret value in scheduler()
ebp,ebx, esi,edi
esp
Inode for "/"
c->scheduler
edx = &p->context

After swtch()

- Process is running in forkret()
- c->csheduler has saved the old kernel stack
  - with the context of p, return value in scheduler, ebp, ebx, esi, edi on stack
  - remember {edi, esi, ebx, ebp, ret-value } = context
  - The c->scheduler is pointing to old context
- CR3 is pointing to process pgdir

after forkret()
just before
trapret()
begins
Code/stack
of
"initcode"
proc
cpu
*c
p
proc
context
kstack
tf
sz = 4096
name = "initcode"
pgdir
cwd
state = RUNNING
Trapframe
CS =3
DS = ES = SS =4
EFLAGS = FL_IF
ESP = 4096
EIP = 0
trapret()
Page table
Page table
Page table
Toward
s
Kernel
pages
pgdir
CR3
kpgdir
addr of
p->context
addr of
c->scheduler
ret value in
scheduler()
ebp,ebx,
esi,edi
esp
Inode
for "/"
c->scheduler
edx = &p->context

After iret in trapret

- **The CS, EIP, ESP will be changed**
  - **to values already stored on trapframe**
  - **this is done by iret**
- **Hence after this user code will run**
  - **On user stack!**
- **Hence code of *initcode* will run now**

at the end
of trapret()
eip
proc
cpu
*c
Code/stack
of
“initcode”
p
proc
context
kstack
tf
sz = 4096
name = “initcode”
pgdir
cwd
state = RUNNING
Page table
Page table
Toward
s
Kernel
pages
Page table
pgdir
kpgdir
CR3
esp
addr of
p->context
addr of
c->scheduler
ret value in
scheduler()
ebp,ebx,
esi,edi
Inode
for “/”
c->scheduler
edx = &p->context

initcode

```
# char init[] = "/init\0";
init:
  .string "/init\0"

# char *argv[] = { init, 0 };
.p2align 2
argv:
  .long init
  .long 0
```

```
start:
  pushl $argv
  pushl $init
  pushl $0  // where caller pc would be
  movl $SYS_exec, %eax
  int $T_SYSCALL

# for(;;) exit();
exit:
  movl $SYS_exit, %eax
  int $T_SYSCALL
  jmp exit
```

```
00000000 <start>:
   0:     68 24 00 00 00          push   $0x24
   5:     68 1c 00 00 00          push   $0x1c
   a:     6a 00                    push   $0x0
   c:     b8 07 00 00 00          mov    $0x7,%eax
  11:     cd 40                    int    $0x40

00000013 <exit>:
  13:     b8 02 00 00 00          mov    $0x2,%eax
  18:     cd 40                 int    $0x40
  1a:     eb f7                 jmp    13 <exit>
0000001c <init>:
          "/init\0"
00000024 <argv>:
      1c 00
      00 00
```

on sys_exec()
+ all traps()
eip
proc
cpu
*c
alltraps():
0x24
0x1c
0
code
p
proc
context
kstack
tf
sz = 4096
name = “initcode”
pgdir
cwd
state = RUNNING
ss =4,esp, eflags
cs = 3, eip
0,64,ds,es,fs,gs,
gen registers,
add of this esp,
ret add in alltraps()
Page table
Page table
Toward
s
Kernel
pages
Page table
pgdir
CR3
kpgdir
addr of
p->context
addr of
c->scheduler
ret value in
scheduler()
ebp,ebx,
esi,edi
esp
Inode
for “/”
c->scheduler
edx = &p->context

Understanding fork() and exec()

First, revising some concepts already learnt then code of fork(), exec()

First process creation
Let's revisit struct proc

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state. allocated, ready to run, running, wait-
ing for I/O, or exiting.
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process. Process's context
  void *chan;                  // If non-zero, sleeping on chan. More when we discuss sleep, wakeup
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files, used by open(), read(),...
  struct inode *cwd;           // Current directory, changed with "chdir()"
  char name[16];               // Process name (for debugging)
};
```

struct proc diagram

sz = ELF-code->memsz (includes data, check “ld -N”
+ 2*4096 (for stack)

fork()/exec() are syscalls. On every syscall this happens

- Fetch the n'th descriptor from the IDT, where n is the argument of int.
- Check that CPL in %cs is <= DPL, where DPL is the privilege level in the descriptor.
- Save %esp and %ss in CPU-internal registers, but only if the target segment selector's PL < CPL.
  - Switching from user mode to kernel mode. Hence save user code's SS and ESP
- Load %ss and %esp from a task segment descriptor.
  - Stack changes to kernel stack now. TS descriptor is on GDT, index given by TR register. See switchuvm()
- Push %ss. // optional
- Push %esp. // optional (also changes ss,esp using TSS)
- Push %eflags.
- Push %cs.
- Push %eip.
- Clear the IF bit in %eflags, but only on an interrupt.
- Set %cs and %eip to the values in the descriptor.

After “int” ‘s job is done

- IDT was already set, during idtinit()
  - Remember vectors.S – gives jump locations for each interrupt
- “int 64” ->jump to 64th entry in vector table

```
vector64:
  pushl $0
  pushl $64
  jmp alltraps
```

  - So now stack has ss, esp,eflags, cs, eip, 0 (for error code), 64
  - Next run alltraps from trapasm.S

alltraps:

```
# Build trap frame.
pushl %ds
pushl %es
pushl %fs
pushl %gs
pushal // push all gen purpose regs
# Set up data segments.
movw $(SEG_KDATA<<3), %ax
movw %ax, %ds
movw %ax, %es
# Call trap(tf), where tf=%esp
pushl %esp # first arg to trap()
call trap
addl $4, %esp
```

- Now stack contains

  ss, esp,eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi

  - This is the struct trapframe !
  - So the kernel stack now contains the trapframe
  - Trapframe is a part of kernel stcak

trap()

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
  switch(tf->trapno){
  .....
```

- Argument is trapframe
- In alltraps
  - Before “call trap”, there was “push %esp” and stack had the trapframe
  - Remember calling convention --> when a function is called, the stack contains the arguments in reverse order (here only 1 arg)

trap()

- Has a switch
  - switch(tf->trapno)
  - Q: who set this trapno?
- Depending on the type of trap
  - Call interrupt handler
    - Timer
      - wakeup(&ticks)
    - IDE: disk interrupt
      - Ideintr()
    - KBD
      - Kbdintr()
    - COM1
      - Uatrintr()
    - If Timer
      - Call yield() -- calls sched()
    - If process was killed (how is that done?
      - Call exit()!

when trap() returns

- #Back in alltraps

```
call trap
addl $4, %esp

# Return falls through to trapret...
.globl trapret
trapret:
popal
popl %gs
popl %fs
popl %es
popl %ds
addl $0x8, %esp  # trapno and errcode
iret
```

-

- Stack had (trapframe)
  - ss, esp,eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi, esp
- add $4 %esp
  - esp
- popal
  - eax, ecx, edx, ebx, oesp, ebp, esi, edi
- Then gs, fs, es, ds
- add $0x8, %esp
  - 0 (for error code), 64
- iret
  - ss, esp,eflags, cs, eip,

understanding fork()

- **What should fork do?**
  - **Create a copy of the existing process**
  - **child is same as parent, except pid, parent-child relation, return value (pid or 0)**
  - **Please go through every member of struct proc, understand it's meaning to appreciate what fork() should do**
  - **create a struct proc, and**
    - **duplicate pages, page directory, sz, state,trapframe,context, ofile (and files!), cwd, name**
    - **modify: pid, parent, trapframe, state**

understanding fork()

```
int
sys_fork(void)
{
  return fork();
}
```

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }
```

after allocproc()
-- we studied this -- same as creation of first process

understanding fork()

```
// Copy process state from proc.
if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
  kfree(np->kstack);
  np->kstack = 0;
  np->state = UNUSED;
  return -1;
}
np->sz = curproc->sz;
np->parent = curproc;
*np->tf = *curproc->tf;
```

- copy the pages, page tables, page directory
  - no copy on write here!
  - Rewind if operation of copyuvm() fails
- copy size
- set parent of child
- copy trapframe (structure is copied)

```
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;  pte_t *pte;  uint pa, i, flags;
  char *mem;
  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
      panic("copyuvm: page not present");
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto bad;
    memmove(mem, (char*)P2V(pa), PGSIZE);
    if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
      kfree(mem);
      goto bad;
    }
  }
  return d;
bad:
  freevm(d);
  return 0;
}
```

understanding fork()->copyuvm()

- Map kernel pages
- for every page in parent's VM address space
  - allocate a PTE for child
  - set flags
  - copy data
  - map pages in child's page directory/tables

understanding fork()

```
np->tf->eax = 0;
for(i = 0; i < NOFILE; i++)
  if(curproc->ofile[i])
    np->ofile[i] = filedup(curproc->ofile[i]);
np->cwd = idup(curproc->cwd);
safestrcpy(np->name, curproc->name, sizeof(curproc->name));
pid = np->pid;
acquire(&ptable.lock);
np->state = RUNNABLE;
release(&ptable.lock);
```

- set return value of child to 0
  - eax contains return value, it's on TF
- copy each struct file
- copy current working dir inode
- copy name
- set pid of child
- set child "RUNNABLE"

exec() - different prototype

- **int exec(char*, char**);**
  - **usage: to print README and test.txt using “cat”**

```
int main(int argc, char *argv[])
{
   char *cmd = "/cat";
   char *argstr[4] = { "/cat", "README",
"test.txt", 0};
   exec(cmd, argstr);
}
```

note: to really run this code in xv6, you need to make changes to Makefile. First, add this program to UPROGS, then write a file test.txt using Linux, and add ‘test.txt’ to list of files in ‘mkfs’ target in Makefile

sys_exec()

```
int
sys_exec(void)
{
  char *path, *argv[MAXARG];
  int i;
  uint uargv, uarg;
  if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
    return -1;
  }
  memset(argv, 0, sizeof(argv));
  for(i=0;; i++){
    if(i >= NELEM(argv))
      return -1;
    if(fetchint(uargv+4*i, (int*)&uarg) < 0)
      return -1;
    if(uarg == 0){
      argv[i] = 0;
      break;
    }
    if(fetchstr(uarg, &argv[i]) < 0)
      return -1;
  }
  return exec(path, argv);
}
```

- **argstr(n,), argint(n,)**
  - **Fetch the n'th argument from *process stack* using p->tf->esp + offset**
  - **Again: revise calling conventions**
  - **0'th argument: name of executable file**
  - **1st Argument: address of the array of arguments**
    - **store in *uargv***

sys_exec()

```
int sys_exec(void)
{
  char *path, *argv[MAXARG];
  int i;   uint uargv, uarg;
  if(argstr(0, &path) < 0 || argint(1,
(int*)&uargv) < 0){
    return -1;
  }
  memset(argv, 0, sizeof(argv));
  for(i=0;; i++){
    if(i >= NELEM(argv))     return -1;
    if(fetchint(uargv+4*i, (int*)&uarg) < 0)
      return -1;
    if(uarg == 0){
      argv[i] = 0;       break;
    }
    if(fetchstr(uarg, &argv[i]) < 0)
      return -1;
  }
  return exec(path, argv);
}
```

- the local array argv[] (allocated on kernel stack, obviously) set to 0
- fetch every next argument from array of arguments
  - Sets the address of argument in argv[1]
- call exec
  - beware: mistake to assume that this exec() is the exec() called from user code! NO!

What should exec() do?

- **Remember, it came from fork()**
  - **so proc & within it tf, context, kstack, pgdir-tables-pages, all exist.**
  - **Code, stack pages exist, and mappings exist through proc->pgdir**
- **Hence**
  - **read the ELF executable file (argv[0])**
  - **create a new page dir – create mappings for kernel and user code+data; copy data from ELF to these pages (later discard old pagedir)**
  - **Copy the argv onto the user stack – so that when new process starts it has it's main(argc, argv[]) built**
  - **set values of other fields in proc to start program correctly**

User stack after call to exec() is over

normally data on stack on fn call: ret value, first arg, second arg, ...

main(int argc, char *argv[])

argv[] is address of array of string; string itself is an adress. Hence 2 levels of indirection on stack

exec()

```
int
exec(char *path, char **argv)
{
...
  uint argc, sz, sp,
ustack[3+MAXARG+1];
...
  if((ip = namei(path)) == 0){
    end_op();
    cprintf("exec: fail\n");
    return -1;
  }
```

- **ustack**
  - **used to build the arguments to be pushed on user-stack**
- **namei**
  - **get the inode of the executable file**

exec()

```
// Check ELF header
if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
  goto bad;
if(elf.magic != ELF_MAGIC)
  goto bad;

if((pgdir = setupkvm()) == 0)
  goto bad;
```

- **readi**
  - **read ELF header**
- **setupkvm()**
  - **creating a *new* page directory and mapping kernel pages**

exec()

```
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
  if(ph.type != ELF_PROG_LOAD)
    continue;
  if(ph.memsz < ph.filesz)
    goto bad;
  if(ph.vaddr + ph.memsz < ph.vaddr)
    goto bad;
  if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
  if(ph.vaddr % PGSIZE != 0)
    goto bad;
  if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) <
0)
    goto bad;
}
```

- Read ELF program headers from ELF file
- Map the code/data into pagedir-pagetable-pages
- Copy data from ELF file into the pages allocated

exec()

```
sz = PGROUNDUP(sz);
if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
  goto bad;
clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
sp = sz;
```

- Allocate 2 pages on top of proc->sz
- One page for stack
- one page for guard page
- Clear the valid flag on guard page

exec()

```
// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
  if(argc >= MAXARG)
    goto bad;
  sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
  if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
    goto bad;
  ustack[3+argc] = sp;
}
ustack[3+argc] = 0;
ustack[0] = 0xffffffff;  // fake return PC
ustack[1] = argc;
ustack[2] = sp - (argc+1)*4;  // argv pointer
sp -= (3+argc+1) * 4;
if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
  goto bad;
```

- For each entry in argv[]
  - copy it on user-stack
  - remember it's location on user stack in ustack
- add extra entries (to be copied to user stack) to ustack
- copy argc, argv pointer
- take sp to bottom
- copy ustack to user stack

This is what the code on earlier slide did

exec()

```
// Save program name for debugging.
for(last=s=path; *s; s++)
  if(*s == '/')
    last = s+1;
safestrcpy(curproc->name, last, sizeof(curproc->name));

// Commit to the user image.
oldpgdir = curproc->pgdir;
curproc->pgdir = pgdir;
curproc->sz = sz;
curproc->tf->eip = elf.entry;  // main
curproc->tf->esp = sp;
switchuvm(curproc);
freevm(oldpgdir);
return 0;
```

- copy name of new process in proc->name
- change to new page directory
- change new size
- tf->eip will be used when we return from exec() to jump to user code. Set to to first instruction of code, given by elf.entry
- Set user stack pointer to "sp" (bottom of stack of arguments)
- Update TSS, change CR3 to newpagedir
- free old page dir

return 0 from exec()?

- We know exec() does not return !
- This was exec() function !
  - Returns to sys_exec()
- sys_exec() also returns , where?
  - Remember we are still in kernel code, running on kernel stack. p->kstack has the trapframe setup
  - There is context struct on stack. Why?
  - sys_exec() returns to trapret(), the trap frame will be popped !
    - with “iret” jump into new program !
  - New program is not old program , which could have accessed return value of sys_exec()

File Systems

Abhijit A M
abhijit.comp@coep.ac.in

Introduction

- **Human end user's view of file sytem on a modern desktop operating system**
  - **Files, directories(folders), hierarchy – acyclic graph like structure**
  - **Windows Vs Linux logical organization: multiple partitions (C:, D:,etc.), vs single logical namespace starting at "/"**

Introduction

- **Secondary and Tertiary memory**
  - **Hard disks, Pen drives, CD-ROMs, DVDs, Magnetic Tapes, Portable disks, etc. Used for storing files**
  - **Each comes with a hardware “controller” that acts as an intermediary in the hardware/software boundary**
  - **IDE, SATA, SCSI, SAS, etc. Protocols : Different types of cables, speeds, signaling mechanisms**
  - **Controllers provide a block/sector based read/write access**
    - **Block size is most typically 512 bytes**
    - **Can’t read byte no. 33 directly. Must read sector 0 (byte 0 to 511) in memory and then access the byte no 33 in memory.**

Introduction

- **OS and File system**
  - **OS bridges the gap between end user and stoage hardware controller**
  - **Provides data structure to map the logical view of end users onto disk storage**
  - **Essentially an implementation of the acyclic graph on the sequential sector-based disk storage**
    - **Both in memory and on-disk**
  - **Provides system calls (open, read, write, ..., etc.) to enable access to files, folders**

What we are going to learn

- **The operating system interface (system calls, commands/utilities) for accessing files in a file-sysetm**
- **Design aspects of OS to implement the file system**

What is a file?

- A sequence of bytes , with
  - A name
  - Permissions
  - Owner
  - Timestamps,
  - Etc.
- Types: Text files, binary files
  - Text: All bytes are human readable
  - Binary: Non-text
- Types: ODT, MP4, TXT, DOCX, etc.
  - Most typically describing the organization of the data inside the file
  - Each type serving the needs of a particular application

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

What is a file?

- The sequence of bytes can be interpreted to be
  - Just a sequence of bytes
    - E.g. a text file
  - Sequence of records/structures
    - E.g. a file of student records
  - A complexly organized, collection of records and bytes
    - E.g. a “ODT” or “DOCX” file
- What’s the role of OS in above mentioned file type, and organization?
  - Mostly NO role on Unixes, Linuxes!
  - They are handled by applications !
  - Types handled by OS: normal file, directory, block device file, character device file, FIFO file (named pipe), etc.
  - Also types handled by OS: executable file, non-executable file

File attributes

- Run

  $ ls -l

  on Linux

  To see file listing with different attributes

- Different OSes and file-systems provide different sets of file attributes
  - Some attributes are common to most, while some are different
  - E.g. name, size, owner can be found on most systems
  - “Executable” permission may not be found on all systems

File protection attributes

- File owner/creator should be able to control:
  - what can be done
  - by whom
- Types of access
  - Read
  - Write
  - Execute
  - Append
  - Delete
  - List
- Linux: See commands: “chown”, “chgrp” and “chmod”

- Linux file permissions
  - For owner, group and others
  - Read, write and execute
  - Total 9 permissions

A Sample UNIX Directory Listing

-rw-rw-r--	1 pbg	staff	31200	Sep 3 08:30	intro.ps
drwx------	5 pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2 pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2 pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1 pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1 pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4 pbg	faculty	512	Jul 31 10:31	lib/
drwx------	3 pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3 pbg	staff	512	Jul 8 09:35	test/

Windows XP Access List

Access methods

- OS system calls may provide two types of access to files
  - Sequential Access
    - read next
    - write next
    - reset
    - no read after last write (rewrite)
  - Linux provides sequential access using open(), read(), write(), ...
- Direct Access
  - read n
  - write n
  - position to n
    read next
    write next
  - rewrite n

  n = relative block number
- pread(), pwrite() on Linux

Device Drivers

- Hardware manufacturers provide “hardware controllers” for their devices
- Hardware controllers can operate the hardware, as instructed
- Hardware controllers are instructed by writing to particular I/O ports using CPU’s machine instructions
  - This bridges the hardware-software gap
- OS programmers, typically, write one “device driver” code that interacts with one hardware controller
  - This is pure C code
  - Which calls I/O instructions for hardware controller
  - This code is independent of most of the OS code
  - Often this code is like an add-on Module , eg. Linux kernel

Disk device driver

- OS views the disk as a logical sequence of blocks
  - OS's assumed block size may be > sector size
- OS Talks to disk controller
- Helps the OS Convert it's view of "logical block" of the disk, into physical sector numbers
- Acts as a translator between rest of OS and hardware controller
- xv6: ide.c

OS’s job now

- To implement the logical view of file system as seen by end user
- Using the logical block-based view offerred by the device driver

Volume Managers

- Special type of kernel device drives, which reside on top of disk device drivers
- Provide a more abstract view of the underlying hardware
- E.g. Can combine two physical hard disks, and present them as one
- Allow end users to
  - "combine one more more physical disks to create physical volumes"
  - "create volume groups out of a set of physical volumes",
  - "create logical volums within volume groups"

Logical Volume Management Components

Formatting

- Physical hard disk divided into partitions
  - Partitions also known as minidisks, slices
- A raw disk partition is accessible using device driver – but no block contains any data !
  - Like an un-initialized array, or sectors/blocks
- Formatting
  - Creating an initialized data structure on the partition, so that it can start storing the acyclic graph tree structure on it
  - Different formats depending on different implementations of the directory tree structure: ext4, NTFS, vfat, VxFS, ReiserFS, WafleFS, etc.
- Formatting happens on “a physical partition” or “a logical volume made available by volume manager”

Different types of “layouts”
Single level directory

Naming problem

Grouping problem

Different types of “layouts”
Two level directory

Path name

Can have the same file name for different user

Efficient searching

No grouping capability

Tree Structured directories

Acyclic Graph Directories

General Graph directory

Mounting of a file system: before

Mounting of a file system: after

$sudo mount */dev/sda5 /users*

Remote mounting: NFS

- **Network file system**
- **$ sudo mount 10.2.1.2:/x/y /a/b**
  - **The /x/y partition on 10.2.1.2 will be made available under the folde /a/b on this computer**
-

File sharing semantics

- Consistency semantics specify how multiple users are to access a shared file simultaneously
- Unix file system (UFS) implements:
  - Writes to an open file visible immediately to other users of the same open file
  - One mode of sharing file pointer to allow multiple users to read and write concurrently
- AFS has session semantics
  - Writes only visible to sessions starting after the file is closed

Implementing file systems

File system on disk

- Disk I/O in terms of sectors (512 bytes)
- File system: implementation of acyclic graph using the linear sequence of sectors
- Device driver: available to rest of the OS code to access disk using a block number

File system implementation: layering

File system: Layering

- Device drivers manage I/O devices at the I/O control layer
  - Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” outputs low-level hardware specific commands to hardware controller
- Basic file system given command like “retrieve block 123” translates to device driver
  - Also manages memory buffers and caches (allocation, freeing, replacement)
  - Buffers hold data in transit
  - Caches hold frequently used data
- File organization module understands files, logical address, and physical blocks
  - Translates logical block # to physical block #
  - Manages free space, disk allocation
- Logical file system manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks (inodes in Unix)
  - Directory management
  - Protection

Application programs

```
int main() {
        char buf[128]; int count;
        fd = open(...):
        read(fd, buf, count);
}
```

OS

Logical file system:

```
sys_read(int fd, char *buf, int count) {
        file *fp = currproc->fdarray[fd];
        file_read(fp, ...);
}
```

File organization module:

```
file_read(file *fp, char *buf, int count) {
        offset = fp->current-offset;
        translate offset into blockno;
        basic_read(blockno, buf, count);

}
```

Basic File system:

```
basic_read(int blockno, char *buf, ...) {
        os_buffer *bp;
        sectorno = calculation on blockno;
        disk_driver_read(sectono, bp );
        move-process-to-wait-queue;
        copy-data-to-user-buffer(bp, buf);

}
```

IO Control, Device driver:

```
disk_driver_read(sectorno) {
        issue instructions to disk controller
(often assembly code)
        to read sectorno  into specific
location;
}
```

XV6 does it slightely differently, but following the layering principle!

Layering advantages

- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
  - Logical layers can be implemented by any coding method according to OS designer
- Many file systems, sometimes many within an operating system
  - Each with its own format (CD-ROM is ISO 9660; Unix has UFS, FFS; Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray, Linux has more than 40 types, with extended file system ext2 and ext3 leading; plus distributed file systems, etc)
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE
  - The Virtual File System (to be discussed later) helps us combine multiple physically different file systems into one logical layer at the top

File system implementation: Different problems to be solved

- What to do at boot time ?
- How to store directories and files on the partition ?
  - Complex problem. Hiearchy + storage allocation + efficiency + limits on file/directory sizes + links (hard, soft)
- How to manage list of free sectors/blocks?
- How to store the summary information about the complete file system
- How to mount a file system , how to unmount

File system implementation

- We have system calls at the API level, but how do we implement their functions?
  - On-disk and in-memory structures. Let's see some of the important ones.
  - Boot control block contains info needed by system to boot OS from that volume
    - Not always needed. Needed if volume contains OS, usually first block of volume
  - Volume control block (superblock, master file table) contains volume details
    - Total # of blocks, # of free blocks, block size, free block pointers or array
  - Per-file File Control Block (FCB) contains many details about the file
    - Inode(FCB) number, permissions, size, dates
  - Directory structure organizes the files
    - Names and inode numbers, master file table

A typical file control block (inode)

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Why does it NOT contain the

Name of the file ?

In memory data structures

- **Mount table**
  - **storing file system mounts, mount points, file system types**
- **See next slide for “file” realated data structures**
- **Buffers**
  - **hold data blocks from secondary storage**

In memory data structures: for open,read,write, ...

Open returns a file handle for subsequent use

Data from read eventually copied to specified user process memory address

At boot time

- **Root partition**
  - **Contains the file system hosting OS**
  - **“mounted” at boot time – contains “/”**
    - **Normally can’t be unmounted!**
- **Check all other partitions**
  - **Specified in /*etc*/fstab on Linux**
  - **Check if the data structure on them is consistent**
    - **Consistent != perfect/accurate/complete**

Directory Implementation

- **Problem**
  - **Directory contains files and/or subdirectories**
  - **Operations required – create files/directories, access files/directories, search for a file (during lookup), etc.**
  - **Directory needs to give location of each file on disk**

Directory Implementation

- Linear list of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - Linear search time
    - Could keep ordered alphabetically via linked list or use B+ tree
  - Ext2 improves upon this approach.
- Hash Table – linear list with hash data structure
  - Decreases directory search time
  - Collisions – situations where two file names hash to the same location
  - Only good if entries are fixed size, or use chained-overflow method

Disk space allocation for files

- File contain data and need disk blocks/sectors for storing it
- File system layer does the allocation of blocks on disk to files
- Files need to
  - Be created, expanded, deleted, shrunk, etc.
  - How to accommodate these requirements?

Contiguous Allocation of Disk Space

directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous allocation

- Each file occupies set of contiguous blocks
- Best performance in most cases
- Simple – only starting location (block #) and length (number of blocks) are required
- Problems include finding space for file, knowing file size, external fragmentation, need for compaction off-line (downtime) or on-line

Linked allocation of blocks to a file

Linked allocation of blocks to a file

- Linked allocation
  - Each file a linked list of blocks
  - File ends at nil pointer
  - No external fragmentation
  - Each block contains pointer to next block (i.e. data + pointer to next block)
  - No compaction, external fragmentation
- Free space management system called when new block needed
- Improve efficiency by clustering blocks into groups but increases internal fragmentation
- Reliability can be a problem
- Locating a block can take many I/Os and disk seeks

FAT: File Allocation Table

- FAT (File Allocation Table), a variation
  - Beginning of volume has table, indexed by block number
  - Much like a linked list, but faster on disk and cacheable
  - New block allocation simple

FAT: File Allocation Table

Variants: FAT8, FAT12, FAT16, FAT32, VFAT, ...

Indexed allocation

Indexed allocation

- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K bytes and block size of 512 bytes. We need only 1 block for index table

Multi level indexing

Unix UFS: combined scheme for block allocation

Free Space Management

- File system maintains free-space list to track available blocks/clusters
  - Bit vector or bit map (n blocks)
  - Or Linked list

Free Space Management: bit vector

- Each block is represented by 1 bit.
- If the block is free, the bit is 1; if the block is allocated, the bit is 0.
  - For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17
  - 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bitmap would be 00111100111111000110000011100000 ...
- A 1- TB disk with 4- KB blocks would require 32 MB (2^{40} / 2^{12} = 2^{28} bits = 2^{25} bytes = 2^{5} MB) to store its bitmap

Free Space Management: Linked list (not in memory, on disk!)

Further improvements on link list method of free-blocks

- Grouping
- Counting
- Space Maps (ZFS)
- Read as homework

Ext2 FS layout

Ext2 FS Layout

```
struct ext2_super_block {
    __le32  s_inodes_count;     /* Inodes count */
    __le32  s_blocks_count;     /* Blocks count */
    __le32  s_r_blocks_count;   /* Reserved blocks count */
    __le32  s_free_blocks_count;    /* Free blocks count */
    __le32  s_free_inodes_count;    /* Free inodes count */
    __le32  s_first_data_block; /* First Data Block */
    __le32  s_log_block_size;   /* Block size */
    __le32  s_log_frag_size;    /* Fragment size */
    __le32  s_blocks_per_group; /* # Blocks per group */
    __le32  s_frags_per_group;  /* # Fragments per group */
    __le32  s_inodes_per_group; /* # Inodes per group */
    __le32  s_mtime;        /* Mount time */
    __le32  s_wtime;        /* Write time */
    __le16  s_mnt_count;        /* Mount count */
    __le16  s_max_mnt_count;    /* Maximal mount count */
    __le16  s_magic;        /* Magic signature */
    __le16  s_state;        /* File system state */
    __le16  s_errors;       /* Behaviour when detecting errors */
```

```
struct ext2_super_block {
...
    __le16  s_minor_rev_level;  /* minor revision level */
    __le32  s_lastcheck;        /* time of last check */
    __le32  s_checkinterval;    /* max. time between checks */
    __le32  s_creator_os;       /* OS */
    __le32  s_rev_level;        /* Revision level */
    __le16  s_def_resuid;       /* Default uid for reserved blocks */
    __le16  s_def_resgid;       /* Default gid for reserved blocks */
    __le32  s_first_ino;        /* First non-reserved inode */
    __le16   s_inode_size;      /* size of inode structure */
    __le16  s_block_group_nr;   /* block group # of this superblock */
    __le32  s_feature_compat;   /* compatible feature set */
    __le32  s_feature_incompat;     /* incompatible feature set */
    __le32  s_feature_ro_compat;    /* readonly-compatible feature set */
    __u8    s_uuid[16];     /* 128-bit uuid for volume */
    char    s_volume_name[16];  /* volume name */
    char    s_last_mounted[64];     /* directory where last mounted */
    __le32  s_algorithm_usage_bitmap; /* For compression */
```

```
struct ext2_super_block {
...
    __u8    s_prealloc_blocks;  /* Nr of blocks to try to preallocate*/
    __u8    s_prealloc_dir_blocks;  /* Nr to preallocate for dirs */
    __u16   s_padding1;
    /*
     * Journaling support valid if EXT3_FEATURE_COMPAT_HAS_JOURNAL set.
     */
    __u8    s_journal_uuid[16]; /* uuid of journal superblock */
    __u32   s_journal_inum;     /* inode number of journal file */
    __u32   s_journal_dev;      /* device number of journal file */
    __u32   s_last_orphan;      /* start of list of inodes to delete */
    __u32   s_hash_seed[4];     /* HTREE hash seed */
    __u8    s_def_hash_version; /* Default hash version to use */
    __u8    s_reserved_char_pad;
    __u16   s_reserved_word_pad;
    __le32  s_default_mount_opts;
    __le32  s_first_meta_bg;    /* First metablock block group */
    __u32   s_reserved[190];    /* Padding to the end of the block */
```

```
struct ext2_group_desc
{
    __le32  bg_block_bitmap;        /* Blocks bitmap block */
    __le32  bg_inode_bitmap;        /* Inodes bitmap block */
    __le32  bg_inode_table;     /* Inodes table block */
    __le16  bg_free_blocks_count;   /* Free blocks count */
    __le16  bg_free_inodes_count;   /* Free inodes count */
    __le16  bg_used_dirs_count; /* Directories count */
    __le16  bg_pad;
    __le32  bg_reserved[3];
};
```

```
struct ext2_inode {
    __le16  i_mode;     /* File mode */
    __le16  i_uid;      /* Low 16 bits of Owner Uid */
    __le32  i_size;     /* Size in bytes */
    __le32  i_atime;    /* Access time */
    __le32  i_ctime;    /* Creation time */
    __le32  i_mtime;    /* Modification time */
    __le32  i_dtime;    /* Deletion Time */
    __le16  i_gid;      /* Low 16 bits of Group Id */
    __le16  i_links_count;  /* Links count */
    __le32  i_blocks;   /* Blocks count */
    __le32  i_flags;    /* File flags */
```

Inode in ext2

```
struct ext2_inode {
   ...
    union {
        struct {
            __le32  l_i_reserved1;
        } linux1;
        struct {
            __le32  h_i_translator;
        } hurd1;
        struct {
            __le32  m_i_reserved1;
        } masix1;
    } osd1;             /* OS dependent 1 */
    __le32  i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
    __le32  i_generation;   /* File version (for NFS) */
    __le32  i_file_acl; /* File ACL */
    __le32  i_dir_acl;  /* Directory ACL */
    __le32  i_faddr;    /* Fragment address */
```

```
struct ext2_inode {
   ...
    union {
        struct {
            __u8    l_i_frag;   /* Fragment number */           __u8    l_i_fsize;  /* Fragment size */
            __u16   i_pad1;             __le16  l_i_uid_high;   /* these 2 fields    */
            __le16  l_i_gid_high;   /* were reserved2[0] */
            __u32   l_i_reserved2;
        } linux2;
        struct {
            __u8    h_i_frag;   /* Fragment number */           __u8    h_i_fsize;  /* Fragment size */
            __le16  h_i_mode_high;          __le16  h_i_uid_high;
            __le16  h_i_gid_high;
            __le32  h_i_author;
        } hurd2;
        struct {
            __u8    m_i_frag;   /* Fragment number */           __u8    m_i_fsize;  /* Fragment size */
            __u16   m_pad1;             __u32   m_i_reserved2[2];
        } masix2;
    } osd2;             /* OS dependent 2 */
```

Ext2 FS Layout: Directory entry

Let’s see a program to read superblock of an ext2 file system.

Efficiency and Performance (and the risks created while trying to achieve it!)

Efficiency

- **Efficiency dependent on:**
  - **Disk allocation and directory algorithms**
  - **Types of data kept in file's directory entry**
  - **Pre-allocation or as-needed allocation of metadata structures**
  - **Fixed-size or varying-size data structures**
  -

Performance

- Keeping data and metadata close together
- Buffer cache – separate section of main memory for frequently used blocks
- Synchronous writes sometimes requested by apps or needed by OS
- No buffering / caching – writes must hit disk before acknowledgement
- Asynchronous writes more common, buffer-able, faster
- Free-behind and read-ahead – techniques to optimize sequential access
- Reads frequently slower than writes

Page cache

- A page cache caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

I/O Without a Unified Buffer Cache

Unified buffer cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching
- But which caches get priority, and what replacement algorithms to use?

I/O Using a Unified Buffer Cache

Recovery

- **Problem. Consider creating a file on ext2 file system.**
  - **Following on disk data structures will/may get modified**
  - **Directory data block, new directory data block, block bitmap, inode table, inode table bitmap, group descriptor, super block, data blocks for new file, more data block bitmaps, ...**
  - **All cached in memory by OS**
- **Delayed write – OS writes changes in its in-memory data structures, and schedules writes to disk when convenient**
  - **Possible that some of the above changes are written, but some are not**
  - **Inconsistent data structure! --> Example: inode table written, inode bitmap written, but directory data block not written**

Recovery

- Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - Can be slow and sometimes fails
- Use system programs to back up data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by restoring data from backup

Log structured file systems

- Log structured (or journaling) file systems record each metadata update to the file system as a transaction
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log (sequentially)
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system structures
  - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata

Journaling file systems

- Veritas FS
- Ext3, Ext4
- Xv6 file system!

File Systems

Abhijit A M
abhijit.comp@coep.ac.in

System calls related to files/file-system

- Open(2), chmod(2), chown(2), close(2), dup(2), fcntl(2), link(2), lseek(2), mknod(2), mmap(2), mount(2), read(2), stat(2), umask(2), unlink(2), write(2), fstat(2), access(2), readlink(2), ...

Implementing file systems

File system on disk

- Disk I/O in terms of sectors (512 bytes)
- File system: implementation of acyclic graph using the linear sequence of sectors
- Device driver: available to rest of the OS code to access disk using a block number

File system implementation: layering

Application programs

```
int main() {
        char buf[128]; int count;
        fd = open(...):
        read(fd, buf, count);
}
```

OS

Logical file system:

```
sys_read(int fd, char *buf, int count) {
        file *fp = currproc->fdarray[fd];
        file_read(fp, ...);
}
```

File organization module:

```
file_read(file *fp, char *buf, int count) {
        offset = fp->current-offset;
        translate offset into blockno;
        basic_read(blockno, buf, count);

}
```

Basic File system:

```
basic_read(int blockno, char *buf, ...) {
        os_buffer *bp;
        sectorno = calculation on blockno;
        disk_driver_read(sectono, bp );
        move-process-to-wait-queue;
        copy-data-to-user-buffer(bp, buf);

}
```

IO Control, Device driver:

```
disk_driver_read(sectorno) {
        issue instructions to disk controller
(often assembly code)
        to read sectorno  into specific
location;
}
```

XV6 does it slightely differently, but following the layering principle!

A typical file control block (inode)

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Why does it NOT contain the

Name of the file ?

Disk space allocation for files

- File contain data and need disk blocks/sectors for storing it
- File system layer does the allocation of blocks on disk to files
- Files need to
  - Be created, expanded, deleted, shrunk, etc.
  - How to accommodate these requirements?

Contiguous Allocation of Disk Space

directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous allocation

- Each file occupies set of contiguous blocks
- Best performance in most cases
- Simple – only starting location (block #) and length (number of blocks) are required
- Problems include finding space for file, knowing file size, external fragmentation, need for compaction off-line (downtime) or on-line

Linked allocation of blocks to a file

Linked allocation of blocks to a file

- Linked allocation
  - Each file a linked list of blocks
  - File ends at nil pointer
  - No external fragmentation
  - Each block contains pointer to next block (i.e. data + pointer to next block)
  - No compaction, external fragmentation
- Free space management system called when new block needed
- Improve efficiency by clustering blocks into groups but increases internal fragmentation
- Reliability can be a problem
- Locating a block can take many I/Os and disk seeks

FAT: File Allocation Table

- FAT (File Allocation Table), a variation
  - Beginning of volume has table, indexed by block number
  - Much like a linked list, but faster on disk and cacheable
  - New block allocation simple

FAT: File Allocation Table

Variants: FAT8, FAT12, FAT16, FAT32, VFAT, ...

Indexed allocation

Indexed allocation

- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K bytes and block size of 512 bytes. We need only 1 block for index table

Multi level indexing

Unix UFS: combined scheme for block allocation

Free Space Management

- File system maintains free-space list to track available blocks/clusters
  - Bit vector or bit map (n blocks)
  - Or Linked list

Free Space Management: bit vector

- Each block is represented by 1 bit.
- If the block is free, the bit is 1; if the block is allocated, the bit is 0.
  - For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17
  - 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bitmap would be 00111100111111000110000011100000 ...
- A 1- TB disk with 4- KB blocks would require 32 MB (2^{40} / 2^{12} = 2^{28} bits = 2^{25} bytes = 2^{5} MB) to store its bitmap

Free Space Management: Linked list (not in memory, on disk!)

Further improvements on link list method of free-blocks

- Grouping
- Counting
- Space Maps (ZFS)
- Read as homework

Directory Implementation

- **Problem**
  - **Directory contains files and/or subdirectories**
  - **Operations required – create files/directories, access files/directories, search for a file (during lookup), etc.**
  - **Directory needs to give location of each file on disk**

Directory Implementation

- Linear list of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - Linear search time
    - Could keep ordered alphabetically via linked list or use B+ tree
  - Ext2 improves upon this approach.
- Hash Table – linear list with hash data structure
  - Decreases directory search time
  - Collisions – situations where two file names hash to the same location
  - Only good if entries are fixed size, or use chained-overflow method

Ext2 FS layout

Ext2 FS Layout

```
struct ext2_super_block {
    __le32  s_inodes_count;     /* Inodes count */
    __le32  s_blocks_count;     /* Blocks count */
    __le32  s_r_blocks_count;   /* Reserved blocks count */
    __le32  s_free_blocks_count;    /* Free blocks count */
    __le32  s_free_inodes_count;    /* Free inodes count */
    __le32  s_first_data_block; /* First Data Block */
    __le32  s_log_block_size;   /* Block size */
    __le32  s_log_frag_size;    /* Fragment size */
    __le32  s_blocks_per_group; /* # Blocks per group */
    __le32  s_frags_per_group;  /* # Fragments per group */
    __le32  s_inodes_per_group; /* # Inodes per group */
    __le32  s_mtime;        /* Mount time */
    __le32  s_wtime;        /* Write time */
    __le16  s_mnt_count;        /* Mount count */
    __le16  s_max_mnt_count;    /* Maximal mount count */
    __le16  s_magic;        /* Magic signature */
    __le16  s_state;        /* File system state */
    __le16  s_errors;       /* Behaviour when detecting errors */
```

```
struct ext2_super_block {
...
    __le16  s_minor_rev_level;  /* minor revision level */
    __le32  s_lastcheck;        /* time of last check */
    __le32  s_checkinterval;    /* max. time between checks */
    __le32  s_creator_os;       /* OS */
    __le32  s_rev_level;        /* Revision level */
    __le16  s_def_resuid;       /* Default uid for reserved blocks */
    __le16  s_def_resgid;       /* Default gid for reserved blocks */
    __le32  s_first_ino;        /* First non-reserved inode */
    __le16   s_inode_size;      /* size of inode structure */
    __le16  s_block_group_nr;   /* block group # of this superblock */
    __le32  s_feature_compat;   /* compatible feature set */
    __le32  s_feature_incompat;     /* incompatible feature set */
    __le32  s_feature_ro_compat;    /* readonly-compatible feature set */
    __u8    s_uuid[16];     /* 128-bit uuid for volume */
    char    s_volume_name[16];  /* volume name */
    char    s_last_mounted[64];     /* directory where last mounted */
    __le32  s_algorithm_usage_bitmap; /* For compression */
```

```
struct ext2_super_block {
...
    __u8    s_prealloc_blocks;  /* Nr of blocks to try to preallocate*/
    __u8    s_prealloc_dir_blocks;  /* Nr to preallocate for dirs */
    __u16   s_padding1;
    /*
     * Journaling support valid if EXT3_FEATURE_COMPAT_HAS_JOURNAL set.
     */
    __u8    s_journal_uuid[16]; /* uuid of journal superblock */
    __u32   s_journal_inum;     /* inode number of journal file */
    __u32   s_journal_dev;      /* device number of journal file */
    __u32   s_last_orphan;      /* start of list of inodes to delete */
    __u32   s_hash_seed[4];     /* HTREE hash seed */
    __u8    s_def_hash_version; /* Default hash version to use */
    __u8    s_reserved_char_pad;
    __u16   s_reserved_word_pad;
    __le32  s_default_mount_opts;
    __le32  s_first_meta_bg;    /* First metablock block group */
    __u32   s_reserved[190];    /* Padding to the end of the block */
```

```
struct ext2_group_desc
{
    __le32  bg_block_bitmap;        /* Blocks bitmap block */
    __le32  bg_inode_bitmap;        /* Inodes bitmap block */
    __le32  bg_inode_table;     /* Inodes table block */
    __le16  bg_free_blocks_count;   /* Free blocks count */
    __le16  bg_free_inodes_count;   /* Free inodes count */
    __le16  bg_used_dirs_count; /* Directories count */
    __le16  bg_pad;
    __le32  bg_reserved[3];
};
```

```
struct ext2_inode {
    __le16  i_mode;     /* File mode */
    __le16  i_uid;      /* Low 16 bits of Owner Uid */
    __le32  i_size;     /* Size in bytes */
    __le32  i_atime;    /* Access time */
    __le32  i_ctime;    /* Creation time */
    __le32  i_mtime;    /* Modification time */
    __le32  i_dtime;    /* Deletion Time */
    __le16  i_gid;      /* Low 16 bits of Group Id */
    __le16  i_links_count;  /* Links count */
    __le32  i_blocks;   /* Blocks count */
    __le32  i_flags;    /* File flags */
```

Inode in ext2

```
struct ext2_inode {
   ...
    union {
        struct {
            __le32  l_i_reserved1;
        } linux1;
        struct {
            __le32  h_i_translator;
        } hurd1;
        struct {
            __le32  m_i_reserved1;
        } masix1;
    } osd1;             /* OS dependent 1 */
    __le32  i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
    __le32  i_generation;   /* File version (for NFS) */
    __le32  i_file_acl; /* File ACL */
    __le32  i_dir_acl;  /* Directory ACL */
    __le32  i_faddr;    /* Fragment address */
```

```
struct ext2_inode {
   ...
    union {
        struct {
            __u8    l_i_frag;   /* Fragment number */           __u8    l_i_fsize;  /* Fragment size */
            __u16   i_pad1;             __le16  l_i_uid_high;   /* these 2 fields    */
            __le16  l_i_gid_high;   /* were reserved2[0] */
            __u32   l_i_reserved2;
        } linux2;
        struct {
            __u8    h_i_frag;   /* Fragment number */           __u8    h_i_fsize;  /* Fragment size */
            __le16  h_i_mode_high;          __le16  h_i_uid_high;
            __le16  h_i_gid_high;
            __le32  h_i_author;
        } hurd2;
        struct {
            __u8    m_i_frag;   /* Fragment number */           __u8    m_i_fsize;  /* Fragment size */
            __u16   m_pad1;             __u32   m_i_reserved2[2];
        } masix2;
    } osd2;             /* OS dependent 2 */
```

Ext2 FS Layout: Directory entry

Let’s see a program to read superblock of an ext2 file system.

Efficiency and Performance (and the risks created while trying to achieve it!)

In memory data structures

- **Mount table**
  - **storing file system mounts, mount points, file system types**
- **See next slide for “file” realated data structures**
- **Buffers**
  - **hold data blocks from secondary storage**

In memory data structures: for open,read,write, ...

Open returns a file handle for subsequent use

Data from read eventually copied to specified user process memory address

Efficiency

- **Efficiency dependent on:**
  - **Disk allocation and directory algorithms**
  - **Types of data kept in file's directory entry**
  - **Pre-allocation or as-needed allocation of metadata structures**
  - **Fixed-size or varying-size data structures**
  -

Performance

- Keeping data and metadata close together
- Buffer cache – separate section of main memory for frequently used blocks
- Synchronous writes sometimes requested by apps or needed by OS
- No buffering / caching – writes must hit disk before acknowledgement
- Asynchronous writes more common, buffer-able, faster
- Free-behind and read-ahead – techniques to optimize sequential access
- Reads frequently slower than writes

Page cache

- A page cache caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

I/O Without a Unified Buffer Cache

Unified buffer cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching
- But which caches get priority, and what replacement algorithms to use?

I/O Using a Unified Buffer Cache

Recovery

- **Problem. Consider creating a file on ext2 file system.**
  - **Following on disk data structures will/may get modified**
  - **Directory data block, new directory data block, block bitmap, inode table, inode table bitmap, group descriptor, super block, data blocks for new file, more data block bitmaps, ...**
  - **All cached in memory by OS**
- **Delayed write – OS writes changes in its in-memory data structures, and schedules writes to disk when convenient**
  - **Possible that some of the above changes are written, but some are not**
  - **Inconsistent data structure! --> Example: inode table written, inode bitmap written, but directory data block not written**

Recovery

- Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - Can be slow and sometimes fails
- Use system programs to back up data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by restoring data from backup

Log structured file systems

- Log structured (or journaling) file systems record each metadata update to the file system as a transaction
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log (sequentially)
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system structures
  - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata

Journaling file systems

- Veritas FS
- Ext3, Ext4
- Xv6 file system!

File System Code

open,read, write, close, pipe, fstat, chdir, dup, mknod, link, unlink, mkdir,

Files, Inodes, Buffers

What we already know

- File system related system calls
  - deal with ‘fd’ arrays (ofile in xv6). open() returns first empty index. open should ideallly locate the inode on disk and initialize some data structures
  - maintain ‘offsets’ within a ‘file’ to support sequential read/write
  - dup() like system calls duplicate pointers in fd-array
  - read/write like system calls, going through ‘ofile’ array, should locate data of file on disk
  - We need functions to read/write from disk – that is disk driver
  - cache data of files in OS data structures for performance : buffering
  - Need to handle on disk data structures as well
- Faster recovery (like journaling in ext3) is desired

xv6 file handling code

- Is a very good example in ‘design’ of a layered and modular architecture
- Splits the entire work into different modules, and modules into functions properly
- The task of each function is neatly defined and compartamentalized

Layers of xv6 file system code

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

Normally, any upper layer can call any lower layer below

Abhijit: Block allocator should be considered as another Layer!

Layout of xv6 file system

May see the code of mkfs.c to get insight into the layout

```
struct superblock {
  uint size;         // Size of file system image (blocks)
  uint nblocks;      // Number of data blocks
  uint ninodes;      // Number of inodes.
  uint nlog;         // Number of log blocks
  uint logstart;     // Block number of first log block
  uint inodestart;   // Block number of first inode block
  uint bmapstart;    // Block number of first free map block
};
#define ROOTINO 1  // root i-number
#define BSIZE 512  // block size
```

Layout of xv6 file system

```
#define NDIRECT 12
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT)
// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEV only)
  short minor;          // Minor device number (T_DEV only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
};

#define DIRSIZ 14

struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

File on disk

dinode
type
major
minor
nlink
size
address 1
.....
address 12
indirect
data
...
data
indirect block
address 1
.....
address 128
data
...
data

Let’s discuss lowest layer first

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

Normally, any upper layer can call any lower layer below

ide.c: idewait, ideinit, idestart, ideintr, iderw

```
static struct spinlock idelock;
static struct buf *idequeue;
static int havedisk1;
```

- ideinit
  - was called from main.c: main()
  - Initialized IDE controller by writing to certain ports
  - havedisk=1 setup
  - Initialize idelock
- idewait
  - BUSY loop waiting for IDE to be ready

ide.c: idewait, ideinit, idestart, ideintr, iderw

- void idestart(buf *b)
  - static void idestart(struct buf *b)
  - Calculate sector number on disk using b->blockno
  - Issue a read/write command to IDE controller.
  - (This is the first buf on idequeue)
- ideintr
  - Take idelock. Called on IDE interrupt (through alltraps()->trap())
  - Wakeup the process waiting on first buffer in buffer *idequeue;
  - call idestart(). Release idelock.
- iderw(buf *b)
  - Move buf b to end of idequeue
  - Call idestart() if not running, sleep on idelock

Let’s see buffer cache layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

Normally, any upper layer can call any lower layer below

Reminder: After main()->binit()

Conceptually Linked liks this

Buffers keep moving on list, as LRU

struct bcache

struct buf

```
struct buf {
  int flags;  // 0 or B_VALID or B_DIRTY
  uint dev;   // device number
  uint blockno; // seq block number on device
  struct sleeplock lock; // Lock to be held by process using it
  uint refcnt; // Number of live accesses to the buf
  struct buf *prev; //  cache list
  struct buf *next; // cache list
  struct buf *qnext; // disk queue
  uchar data[BSIZE];  // data 512 bytes
};
#define B_VALID 0x2  // buffer has been read from disk
#define B_DIRTY 0x4  // buffer needs to be written to disk
```

buffer cache:
static struct buf* bget(uint dev, uint blockno)

- The bcache.head list is maintained on Most Recently Used (MRU) basis
  - head.next is the Most Recently Used (MRU) buffer
  - hence head.prev is the Least Recently Used (LRU)
- Look for a buffer with b->blockno = blockno and b->dev = dev
  - Search the head.next list for existing buffer (MRU order)
  - Else search the head.prev list for empty buffer
  - panic() if found in-use or empty buffer
- Increment b->refcnt ; Returns buffer locked
- Does not change the list structure, just returns a buf in use

buffer cache:
struct buf* bread(uint dev, uint blockno)

```
struct buf*
bread(uint dev, uint blockno)
{
  struct buf *b;
  b = bget(dev, blockno);
  if((b->flags & B_VALID) == 0) {
    iderw(b);
  }
  return b; // locked buffer
}
```

```
void
bwrite(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("bwrite");
  b->flags |= B_DIRTY;
  iderw(b);
}
```

Recollect: iderw moves buf to tail of idequeue, calls idestart() and sleep()

buffer cache:
void brelse(struct buf *b)

- release lock on buffer
- b->refcnt = 0
- If b->refcnt = 0
  - Means buffer will no longer be used
  - Move it to front of the front of bcache.head

Overall in this diagram

Buffers keep moving to the front of the list and around
The list always contains NBUF=30 buffers
head.next is always the MRU and head.prev is always LRU buffer

Let's see logging layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

Normally, any upper layer can call any lower layer below

log in xv6

- a mechanism of recovery from disk
- Concept: multiple write operations needed for system calls (e.g. ‘open’ system call to create a file in a directory)
  - some writes succed and some don’t
  - leading to inconsistencies on disk
- In the log, all changes for a ‘transaction’ (an operation) are either written completely or not at all
- During recovery, completed operations can be “rerun” and incomplete operations neglected

log in xv6

- xv6 system call does not directly write the on-disk file system data structures.
- A system call calls begin_op() at begining and end_op() at end
  - begin_op() increments log.outstanding
  - end_op() decrements log.outstanding, and if it's 0, then calls commit()
- During the code of system call, whenever a buffer is modified, (and done with)
  - log_write() is called
  - This copies the block in an array of blocks inside log, the block is not written in it's actual place in FS as of now
- when finally commit() is called, all modified blocks are copied to disk in the file system

log

```
struct logheader { // ON DISK
  int n; // number of entries in use in block[] below
  int block[LOGSIZE]; // List of block numbers stored
};
struct log { // only in memory
  struct spinlock lock;
  int start;  // first log block on disk (starts with logheader)
  int size; // total number of log blocks (in use out of 30)
  int outstanding; // how many FS sys calls are executing.
  int committing;  // in commit(), please wait.
  int dev; // FS device
  struct logheader lh;  // copy of the on disk logheader
};
struct log log;
```

log on disk

log.start
log.start + 1
log.start+30
boot block
super block
log
data of 52nd block
data of 68nd block
...
log
indoes | bitmap | data ....
block[30]
Start, size, etc
n =2
52
68
0
1
2
3
.....
29
logheader

Typical use case of logging

```
/* In a system call code * /
begin_op();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```

prepare for logging. Wait if logging system is not ready or 'committing'. ++outstanding

read and get access to a data block – as a buffer

modify buffer

note down this buffer for writing, in log. proxy for bwrite(). Mark B_DIRTY. Absorb multiple writes into one.

Syscall done. write log and all blocks. --outstanding.

If outstanding = 0, commit().

match colors in code and comments on right-side

Example of calls to logging

```
//file_write() code
begin_op();
ilock(f->ip);
   /*loop */ r = writei(f->ip, ...);
iunlock(f->ip);
end_op();
```

- each writei() in turn calls bread(), log_write() and brelse()
  - also calles iupdate(ip) which also calls bread, log_write and brelse
- Multiple writes are combined between begin_op() and end_op()

Logging functions

- Initlog()
  - Set fields in global log.xyz variables, using FS superblock
  - Recovery if needed
  - Called from first forkret()
- Following three called by FS code
- begin_op(void)
  - Increment log.outstanding
- end_op(void)
  - Decrement log.oustanding and call commit() if it's zero
- log_write(buf *)
  - Remember the specified block number in log.lh.block[] array
  - Set the block to be dirty
- write_log(void)
  - Called only from commit()
  - Use block numbers specified in log.lh.block and copy those blocks from memory to log-blocks
- commit(void)
  - Called only from end_op()
  - write_log()
  - Write header to disk log-header
  - Copy from log blocks to actual FS blocks
  - Reset and write log header again

Let’s see block allocation layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

Normally, any upper layer can call any lower layer below

Abhijit: Block allocator should be considered as another Layer!

allocating & deallocating blocks on DISK

- balloc(devno)
  - looks for a block whose bitmap bit is zero, indicating that it is free.
  - On finding updates the bitmap and returns the block.
  - balloc() calls bread()->bget to get a block from disk in a buffer.
    - Race prevented by the fact that the buffer cache only lets one process use any one bitmap block at a time.
  - Calls log_write(bp);
    - Thus writes to bitmap blocks are also logged

- bfree(devno, blockno)
  - finds the right bitmap block and clears the right bit.
  - Also calls log_write()

Let’s see Inode Layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

On disk & in memory inodes

```
struct {
  struct spinlock lock;
  struct inode inode[NINODE];
} icache;

// On-disk inode structure
struct dinode {
  short type;     // File type
  short major;   // T_DEV Major device number
  short minor;   // Minor device number
  short nlink;    // Number of links
  uint size;       // Size of file (bytes)
  uint addrs[NDIRECT+1];   /
};
```

```
// in-memory copy of an inode
struct inode {
  uint dev;          // Device number
  uint inum;         // Inode number
  int ref;           // Reference count
  struct sleeplock lock; // protects everything below here
  int valid;         //  been read from disk?

  short type;        // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

In memory inodes

- Kernel keeps a subset of on disk inodes, those in use, in memory
  - as long as ‘ref’ is >0
- The iget and iput functions acquire and release pointers to an inode, modifying the ref count.
- See the caller graph of iget()
  - all those who call iget()
- Sleep lock in ‘inode’ protects
  - fields in inode
  - data blocks of inode

iget and iupdate

- iget
  - searches for an existing/free inode in icache and returns pointer to one
  - if found, increments ref and returns pointer to inode
  - else gets empty inode , initializes, ref=1 and return
  - No lock held after iget()
  - Code must call ilock() after iget() to get lock
  - During lookup (later), many processes can iget() an inode, but only one holds the lock

- iupdate(inode *ip)
  - read on disk block of inode
  - get on disk inode
  - modify it as specified in ‘ip’
  - modify disk block of inode
  - log_write(disk block of inode)

itrunc , iput

- iput(ip)
  - if ref is 1
    - itrunc(ip)
    - type = 0
    - iupdate(ip)
    - i->valid = 0 // free in memory
  - else
    - ref--

- itrunc(ip)
  - write all data blocks of inode to disk
    - using bfree()
  - ip->size = 0
    - Inode is freed from use
  - iupdate(ip)
  - called from iput() only when ‘ref’ becomes zero

race in iput ?

- A concurrent thread might be waiting in ilock to use this inode
  - and won't be prepared to find the in ode is not longer allocated
- This is not possible. Why?
  - no way for a syscall to get a ref to a inode with ip->ref = 1

```
void
iput(struct inode *ip)
{
  acquiresleep(&ip->lock);
  if(ip->valid && ip->nlink == 0){
    acquire(&icache.lock);
    int r = ip->ref;
    release(&icache.lock);
    if(r == 1){
      // inode has no links and no other references: truncate and free.
      itrunc(ip);
```

buffer and inode cache

- to read an inode, it's block must be read in a buffer
- So the buffer always contains a copy of the on-disk dinode
  - duplicate copy in in-memory inode
- The inode cache is write-through,
  - code that modifies a cached inode must immediately write it to disk with iupdate
- Inode may still exist in the buffer cache

allocating inode

- ialloc(dev, type)
  - Loop over all disk inodes
  - read inode (from it's block)
  - if it's free (note inum)
  - zero on disk inode
  - write on disk inode (as zeroes)
  - return iget(dev, inum)
- panic if no free inodes

- ilock
  - code must acquire ilock before using inode's data/fields
  - llock reads inode if it's already not in memory

Trouble with iput() and crashes

- iput() doesn't truncate a file immediately when the link count for the file drops to zero, because
  - some process might still hold a reference to the inode in memory: a process might still be reading and writing to the file, because it successfully opened it.
- if a crash happens before the last process closes the file descriptor for the file,
  - then the file will be marked allocated on disk but no directory entry points to it
- Unsolved problem.
- How to solve it?

Get Inode data: bmap(ip, bn)

- **Allocate ‘bn’th block for the file given by inode ‘ip’**
- **Allocate block on disk and store it in either direct entries or block of indirect entries**
  - **allocate block of indirect entries if needed using balloc()**

writing/reading data at a given offset in file

readi(struct inode *ip, char *dst, uint off, uint n)

writei(struct inode *ip, char *src, uint off, uint n)

- **Calculate the block number in file where ‘off’ belongs**
- **Read sufficient blocks to read ‘n’ bytes**
- **using bread(), brelse()**
- **Call devsw.read if inode is a device Inode.**
- **Writei() also updates size if required**

Reading Directory Layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

directory entry

```
#define DIRSIZ 14

struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

Data of a directory file is a sequence of such entries. To find a name, just get all the data blocks and search the name

How to get the data for a directory? We already know the ans!

struct inode*
dirlookup(struct inode *dp, char *name, uint *poff)

- **Given a pointer to directory inode (dp), name of file to be searched**
  - **return the pointer to inode of that file (NULL if not found)**
  - **set the 'offset' of the entry found, inside directories data blocks, in poff**
- **How was 'dp' obtained? Who should be calling dirlookup? Why is poff returned?**
  - **During resolution of pathnames?**
- **Code: call readi() to get data of dp, search name in it, name comes with inode-num, iget() that inode-num**

int
dirlink(struct inode *dp, char *name, uint inum)

- Create a new entry for ‘name’_’inum’ in directory given by ‘dp’
  - inode number must have been obtained before calling this. How to do that?
- Use dirlookup() to verify entry does not exist!
- Get empty slot in directory’s data block
- Make directory entry
- Update directory inode! writei()

namex

- Called by namei(), or nameiparent()
- Just iteratively split a path using “/” separator and get inode for last component
- iget() root inode, then
- Repeatedly calls
  - split on “/”, dirlookup() for next component
-

races in namex()

- Crucial. Called so many times!
- one kernel thread is looking up a pathname another kernel thread may be changing the directory by calling unlink
  - when executing dirlookup in namex, the lookup thread holds the lock on the directory and dirlookup() returns an inode that was obtained using iget.
- Deadlock? next points to the same inode as ip when looking up ".". Locking next before releasing the lock on ip would result in a deadlock.
  - namex unlocks the directory before obtaining a lock on next.

File descriptor layer code

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

data structures related to “file” layer

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe; // used only if it works as a pipe
  struct inode *ip;
  uint off;
};
// interesting no lock in struct file !
```

```
struct proc {
...
  struct file *ofile[NOFILE];  // Open files per process
...
}
struct {
  struct spinlock lock;
  struct file file[NFILE];
} ftable; //global table from which ‘file’ is allocated to every process
```

Lock is used to protect updates to every entry in the array

Multiple processes accessing same file.

- Each will get a different ‘struct file’
  - but share the inode !
  - different offset in struct file, for each process
  - Also true, if same process opens file many times
- File can be a PIPE (more later)
  - what about STDIN, STDOUT, STDERR files ?
  - Figure out!
- ref
  - used if the file was ‘duped’ or process forked . in that case the ‘struct file’ is shared

file layer functions

- **filealloc**
  - find an empty struct file in ‘ftable’ and return it
  - set ref = 1
- **filedup(file *)**
  - simply ref++
- **fileclose**
  - --ref
  - if ref = 0
    - free struct file
    - iput() / pipeclose()
    - note – transaction if iput() called
- **filestat**
  - simply return fields from inode, after holding lock. on inodes for files only.

file layer functions

- **fileread**
  - **call readi() or piperead()**
  - **readi() later calls device-read or inode read (using bread())**
- **filewrite**
  - **call pipewrite() or writei()**
  - **writei() is called in a loop, within a transaction**

- **Why does readi() call read on the device , why not fileread() itself call device read ?**

pipes

```
struct pipe {
  struct spinlock lock;
  char data[PIPESIZE];
  uint nread;
 // number of bytes read
  uint nwrite;
 // number of bytes written
  int readopen;
  // read fd is still open
  int writeopen;
 // write fd is still open
};
```

- functions
  - pipealloc
  - pipeclose
  - piprad
  - pipewrite
-

pipes

- pipealloc
  - allocate two struct file
  - allocate pipe itself using kalloc (it's a big structure with array)
  - init lock
  - initialize both struct file as 2 ends (r/w)
- pipewrite
  - wait if pipe full
  - write to pipe
  - wakeup processes waiting to read
- piperead
  - wait if no data
  - read from pipe
  - wakeup processes waiting to write
- Good producer consumer code !

Further to reading system call code now

- Now we are ready to read the code of system calls on file system
  - sys_open, sys_write, sys_read , etc.
- Advise: Before you read code of these, contemplate on what these functions should do using the functions we have studied so far.
- Also think of locks that need to be held.

Pointers in C

Abhijit A.M.
abhijit13@gmail.com

(C) Abhijit A.M.
Shared under Creative Commons Attribution Sharealike International License V3.0

Courtsey: Programmer Jokes on telegram

Pointer

- Pointer is a variable which can store addresses
- Pointer has a “type” (except void pointer)
  - e.g. int *p; char *cp; double *dp;
  - Here p, cp, dp are respectively pointers to integer, character, and double
- Size of pointer is decided by compiler

Operations on (and related to) Pointers

&

*

=

+int -int

-

[ ]

Operations related to pointers: &

```
int *p, i;
p = &i;
```

- & fetches the address of variable
  - Called Referencing operator
  - Here, &i is address of i
  - RHS is address of integer, LHS is variable which can store address of integer
- Diagram of this operation is shown on left side
- No need of assuming some value for address (e.g. address 1028). Just the diagram is sufficient to understand the concept

Operations on pointers: *

```
int *p, i, j;
i = 10;
p = &i;
j  = *p;
```

- *** fetches the value stored at a given address**
  - **Called dereferencing operator**
- ***p : here "value of p" is itself an address *(of i)*, so *p is value stored at "value of p" that is i**
- **Diagram's make it easy to understand, *p is simply the contents of box p points to**

Operations on pointers: *

```
int *p, i, j;
i = 10;
p = &i;
j  = *p;
```

```
char *cp, c, d;
c = 'x';
cp = &c;
d  = *cp;
```

- What is the difference between * in the two codes?
- The *p fetches the value at given address in sizeof(int) bytes, while *cp fetches the value at given address in sizeof(char)=1 bytes
- * works based on the size of the type!

Operations on pointers: =

```
int *p, i, *q;
i = 10;
p = &i;
q  = p;
```

- **Pointers can be copied**
  - **Can arrays be copied?**
- **Thumb rule:**
  - **After pointer copy, both pointers point to same location**
- **Common Mistake**
  - **One pointer pointing to another**

Operations on pointers: +- int

```
int *p, a[4], *q;
p =  &a[1];
q = p + 2;
*q = 40;
```

- C allows adding or substracting an int from a pointer!
  - Weird, but true!
  - e.g. int *p, n; p + n;
- The result is a pointer of the same type
- The resultant pointer points *n type locations ahead(for +) or before (for -)*
- *A type location is equal to sizeof(type).*

Problems: Draw diagrams for the code

```
int main() {
    int *p, *q, a[3], b;
    a[0] = 10; b = 2;
    p = &a[1];
    q = p + 1;
    p = q – b;
    *p = 30;
}
```

```
int main() {
    int *p, *q, a[3], b;
    a[0] = 10; b = 1;
    p = &a[3];
    q = p - 3;
    p = q + 1;
    *(q + 1) = 30;
    *(p – 1) = 20;
}
```

Operations on pointers: substracting two pointers

```
int *p, a[4], *q, x;
p =  &a[1];
q = p + 2;
x = q – p;
```

- **Two pointers of the *same type* can be substracted**
  - **Result type is int**
  - **Result value = no. Of elements of sizeof(type) between two pointers**
- **Logically derives from adding/substracting int to pointers**
  - **p = q + 2 ==> p – q = 2**

Operations on pointers: [ ]

```
int *p, a[4], *q;
p =  &a[1];
p[2] = 20;
p[-1] = 10;
```

- **Interestingly, C allows [ ] notation to be applied to all pointers!**
  - **You must be knowing that [ ] is normally used for arrays**
- **p [i] means * (p + i)**
  - **P is a pointer and i is an integer (or i is a pointer and p is a pointer is also allowed)**

A peculiar thing about arrays

- **Name of an array is equivalent to the address of (the zeroeth element) the array**

  int a[3];

  Now

  a* means *&a[0]

- **Because it's an address, it can be stored in a pointer**

  int a[3], *p;

  p = a;

- **What do the following mean?**

  Int a[3] = {1, 2, 3}, *p;

  a + 1;

  ***(a + 1);**

  p = (a + 2)

- **Exception: when passed to sizeof() operator, the name is not the address**

Pointer *as if it was an array*

- **Combine the concepts of**
  - **Pointer arithmetic (+- int)**
  - **[ ] notation for pointers**
  - **Array name as address of array**

```
int a[3], *p;
p = a;
p[2] = 10;
```

- **Here we are using p as if it was an array name**
  - **Possible only if p was pointing to array base**

Pointers != Arrays

- Array is a continuous collection of elements of one type.
- Array has name, the name also stands for the address of the array
- [ ] is allowed operation on arrays
- Array name can't be reassigned to another address

- Pointer is a variable that can store an address
- Pointers can be of various types
- [ ], = , +- int, substraction are operations on pointers
- Pointers can be reassigned to point to different addresses

Arrays as arguments to functions

```
char f(char *c) {
    return c[0];
}
int main() {
    char arr[16], x;
    x = f(arr);
}
```

- Here actual argument is ‘arr’
- Name of array is address of array, that is &arr[0]
  - Address of char
- So formal argument is char *

2-d array as argument to function

```
char f(char **c) {
return c[0][2];
}
int main() {
char arr[16][6], x;
x = f(arr);
}
```

```
char f(char c[][6]) {
 return c[0][2];
}
int main() {
 char arr[16][6], x;
 x = f(arr);
}
```

Which one is correct?
You can answer by just applying the rules we learnt

2-d array as argument to function

```
char f(char (*c)[6]) {
return c[0][2];
}
int main() {
char arr[16][6], x;
x = f(arr);
}
```

```
char f(char c[][6]) {
 return c[0][2];
}
int main() {
 char arr[16][6], x;
 x = f(arr);
}
```

Which one is correct?
(The second one is same as on the earlier slide)
You can answer by just applying the rules we learnt

Dynamic Memory Allocation

Concept of (Binding) “Time”

- **Compile Time**
  - **When you are running commands like**

    cc program.c -o program
- **Load time**
  - **After you type commands like**

    ./program

    Before the main() starts running
- **Program Run Time**
  - **After you type commands like**

    ./program

    When the main() of the program starts running, till it exits
- **Function Call Time**
  - **Interval between the call of a function and before the called function starts running. Part of “Run Time”.**

Lifetime of variables and Memory Allocation

- **Global Variables, Static Variables (g, t, and s here)**
  - **Allocated Memory at *load time***
  - **They are alive (available) till the program exits**
- **Local Variables, Formal Arguments (i, j, k in main; a, b, x, y in f)**
  - **Allocated Memory on *function call***
  - **They are alive (available) as long as function is running**
- **Dynamically allocated memory (Run time allocation)**
  - **Allocated on explicit call to functions like *malloc()***
  - **Alive as long as functions like *free()* are not called on the memory**

```
int g;
static int t = 20;
int f(int a, int b) {
    int x, y;
    static int s = 10;
    x = a + b + 5 + g + s;
    return x;
}
int main() {
    int i, j, k, *p;
    g = 10;
    i = 20; j = 30;
    p = malloc(8);
    k = f(i, j);
}
```

Man pages

Understanding library functions:

Red the manual pages, using ‘man’ command

> man sqrt

> man 3 printf

> man 3 open

 # see section 3 of man pages for C library functions

malloc()

- malloc() is a standard C library function for allocating memory dynamically (at run time)
- #include <stdlib.h> for malloc()
- Function prototype
  - Run “man malloc” to see it

    ```
    void *malloc(size_t size);
    ```

  - size_t is a typedef in stdlib.h
    - size_t is unsigned long
  - Reads a number, allocates those many bytes and returns the address of allocated memory (zero'th byte)
- Additional info: malloc() gets the memory from the OS and then gives to your program

void *

- **A void pointer is a typeless pointer;**
  - **Pure address**
  - **No type --> No "size" information about the type**
- **You can declare a void pointer**

  void *p, *q
- **Void pointer can store *any* address**

  void *p; int a; p = &a; char c; p = &c;
- **Void pointers can be copied**

  void *p, *q; int a; p = &a; q = p;
  - **q also stores address of a now**
- **The Dereferencing operator has no meaning when applied to a void pointer**

  void *p; int a; p = &a;
  - **What does *p mean now? * Needs "size" of the type for its work. Void pointers have no type and so no size information associated with it.**
  - **Note: [ ] is also dereferencing**

malloc()

- malloc() returns a “void *”
  - Returns a pure address
  - This address can be stored in a “void *” variable
  - This address can also be stored in any pointer variable
- Suppose we do
  - void *p; p = malloc(8);
  - Now what meaningful operations can we do with the malloced memory? --> only copy!
  - So normally return value of malloc is stored using some typed pointer

malloc()

```
int *p;
p = malloc(8)
```

- This code allocates 8 bytes and then pointer p will point to the malloced memory
- This code can result in a “warning” because we are converting “void *” to a “int *” with '='

malloc()

```
int *p;
p  = (int *) malloc(8);
```

- This code does away with the warning as we are converting the “void *” into “int *” using explicit type casting
- Suppose size of integer was 4 bytes, then what does this code mean?

malloc()

```
int *p;
p  = (int *) malloc(8);
```

- p points to 8 byte location
- However now, *p means deferencing in “4 bytes”
  - as size of int is 4 bytes
- p[0] means *(p + 0) that is *p
- p[1] means *(p + 1) where (p + 1) is a pointer 4 bytes ahead of p
  - Using this trick we are treating the 8 bytes as if it was a 2 integer array !

malloc()

```
int *p;
p  = (int *) malloc(8);
```

- p points to 8 byte location
- However now, *p means deferencing in “4 bytes”
  - as size of int is 4 bytes

- p[0] means *(p + 0) that is *p
- p[1] means *(p + 1) where (p + 1) is a pointer 4 bytes ahead of p
  - Using this trick we are treating the 8 bytes as if it was a 2 integer array !

Malloc(): Allocating arrays dynamically

```
int *p;
p  = (int *) malloc(
        sizeof(int) * 4);
```

- We can allocate arrays of any type dynamically using malloc()
- Use of sizeof(int) here makes sure that the code is *portable,* appropriately sized array will be allocated irrespective of size of integer
  - Code on earlier slide assumes 4 byte integer
- This code allocates array of 4 integers
  - Can be acessed as p[0], p[1], p[2] and p[3]

malloc(): Allocating array of structures

```
typedef struct test {
        int a, b;
        double g;
}test;
test *p;
p  = (test *) malloc(
        sizeof(test) * 4);
```

- This code allocates an array of 4 structures
- p points to the array of structures
- p[0] is the 0th structure, p[1] is the 1st structure ...
- p[0].a, p[1].g is the way to access the inner elements of structures

Homework

- **Write code using malloc() to**
  - **Allocate an array of 10 doubles**
  - **Read n from user and allocate an array of n shorts**
- **What does the following code mean?**

  int *p; p = malloc(9); // suppose sizeof(int) is 4

  Note: No need to write this sort of code in this course!
- **What does the following code mean?**

  int *ip; Char *p; p = malloc(8); ip = p; p[0] = 10;

  Note: again, no need to write code like this in this course!

free()

- free() will give the malloced memory back
- malloc() and free() work together to manage what is called as “heap memory” which the memory management library has obtained from the OS
- Usage

```
void free(void *p);
```

- free() must be given an address which was returned by malloc()
- Rule: Every malloc() must have a corresponding free()

Pointer to Pointer

```
int **pp, *p, p;
pp = (int **) malloc(sizeof(int *) * 4);
pp[1] = (int *)malloc(sizeof(int) * 3);
```

- A pointer to pointer, is essentially a pointer!
  - Can store an address, has a type
  - The type is “pointer to pointer” so all * or [ ] operations work with sizeof(pointer) which is typically 4 bytes
- The code above allocates an array of pointers, and then allocates an array of integers.
  -

2-d array with pointer to pointer

```
#include <stdlib.h>
#define N 4
int main() {
    int **pp, i, j;
    pp = (int **) malloc
                (sizeof(int *) * N) ;
    for(i = 0; i < N; i++)
        pp[i] = (int *)
                malloc(sizeof(int) * N);
    for(i = 0; i < N; i++)
        for(j = 0; j < N; j++)
            pp[i][j] = i + j;
    return 0;
}
```

2-d array with pointer to pointer

```
#include <stdlib.h>
#define N 4
int main() {
    int **pp, i, j;
    pp = (int **) malloc
                (sizeof(int *) * N) ;
    for(i = 0; i < N; i++)
        pp[i] = (int *)
                malloc(sizeof(int) * N);
    for(i = 0; i < N; i++)
        for(j = 0; j < N; j++)
            pp[i][j] = i + j;
    return 0;
}
```

- How does pp[i][j] work here?
- pp[i] is *(pp + i) which is the pointer at the i'th location in the array of pointers allocated
  - Here * works with *size* = *sizeof a pointer*
- pp[i][j] is *(pp[i] + j)
  - Since pp[i] is a integer pointer, so here * works with sizeof(int)
  - pp[i] is the address of array (indicated in red), so pp[i] + j is pointer to the j'th element of that array

Self Referential Pointers in Structures

Self Referential Pointers

- "Self Referential Pointer" is a kind of a misnomer
- C allows structures like this

```
struct test {
  int a;
  struct test *p;
};
```

- The pointer p, can point to any variable of the type "struct test" (or be NULL)

Self Referential Pointers

- Consider following code

```
typedef struct test {
    int a;
    struct test *p;
}test;
test  x, y, *p, *q;
p = &x;
x.p = &y;
q = &y;
y.p = &y;
```

Self referential structures allow us to create a variety of "linked" *structures of data*

-> notation

```
typedef struct test {
    int a;
    struct test *p
}test;
test  m, n, *x;
m.a = 20;
x = &n;
(*x).a = 40;
x->a = 50;
x->p = &m;
x->p->p = x;
```

- (*x) is the entire structure to which x points
  - (*x).a is the variable 'a' in that structure
- x->a is another notation for (*x).a

2 Common Problems involving Pointers:

(Dereferencing) Dangling Pointers
And
Garbage Memory

NULL

- NULL is a symbolic constant defined in stdio.h
  - #define NULL 0
- This is a special pointer value, defined by C language
  - The number 0 is not the same as the address 0 ! They are different types !
- It is guaranteed that a pointer is NOT NULL, unless programmer sets it to NULL

A side note on NULL

- **NULL is not the number 0**
- **NULL is not necessarily the address 0**
- **NULL is just a special value for pointers told to us by C language.**
  - **Very often we need special values for a certain type**
  - **E.g. the value '\0' for a character is universally taken to be a special value indicating end of a character sequence in an array**
  - **INT_MAX , INT_MIN are values #defined in limits.h for integers**
  - **These type of values are used in programs to indicate either an *unused variable* or *empty variable* or *error value* on that variable**
- **Int i = NULL; char c = NULL; works**
  - **Why do you want to do it? Instead of saying int i = 0; char c = 0;**

Dangling Pointers

- Total 3 Possibilities for a pointer value
  - Points to memory owned by program
    - Local variables, Global Variables, Malloced Memory
  - = NULL
  - Dangling
    - Some texts differentiate between ‘wild’ (uninitialized) and ‘dangling’ pointers
- Dereferencing (that is *) a dangling (or NULL) pointer is NOT to be done
  - Even if you find that dereferencing a dangling pointer “*does not create problem”* in your code, it is still not to be done!
  - Result: The OS punishes your code by terminating it, and saying “Segmentation Fault”

Dangling Pointers

- A pointer is dangling when declared & not initialized
- A pointer becomes non dangling, when assigned to a “good” memory location like local variables, global variables, malloc-ed memory
- A pointer can become dangling again due to
  - Mistakes in pointer arithmetic
  - Mistakes in manipulation of dynamically allocated memory, e.g. Linked list pointers
  - On deallocation of malloced memory using free()
  - etc

```
int main() {
        int *p, i, j, *q;// p,q are dangling

}
```

Dangling Pointers Example: 1

```
int main() {
        int *p, i, j, *q;// p,q are dangling
        p = &i; // p not dangling

}
```

Dangling Pointers Example: 1

```
int main() {
        int *p, i, j, *q;// p,q are dangling
        p = &i; // p not dangling
        i = 10; // *p = 10

}
```

Dangling Pointers Example: 1

```
int main() {
        int *p, i, j, *q;// p,q are dangling
        p = &i; // p not dangling
        i = 10; // *p = 10
        q = &j; // q not dangling

}
```

Dangling Pointers Example: 1

```
int main() {
        int *p, i, j, *q;// p,q are dangling
        p = &i; // p not dangling
        i = 10; // *p = 10
        q = &j; // q not dangling
        q = (int *)malloc(sizeof(int) * 4);
                        // q not dangling

}
```

Dangling Pointers Example: 1

```
int main() {
        int *p, i, j, *q;// p,q are dangling
        p = &i; // p not dangling
        i = 10; // *p = 10
        q = &j; // q not dangling
        q = (int *)malloc(sizeof(int) * 4);
                                // q not dangling
        q = q + 3; // q not dangling
       }
```

Dangling Pointers Example: 1

```
int main() {
        int *p, i, j, *q;// p,q are dangling
        p = &i; // p not dangling
        i = 10; // *p = 10
        q = &j; // q not dangling
        q = (int *)malloc(sizeof(int) * 4);
                                // q not dangling
        q = q + 3; // q not dangling
        q = q + 1; // q IS dangling now

}
```

Dangling Pointers Example: 1

```
int main() {
        int *p, i, j, *q;// p,q are dangling
        p = &i; // p not dangling
        i = 10; // *p = 10
        q = &j; // q not dangling
        q = (int *)malloc(sizeof(int) * 4);
                                // q not dangling
        q = q + 3; // q not dangling
        q = q + 1; // q IS dangling now
        p = q - 6; // p IS also dangling
}
```

Dangling Pointers Example: 1

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
}
```

Dangling Pointers Example: 2

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
        a.val = 20;
}
```

Dangling Pointers Example: 2

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
        a.val = 20;
        p = &a;

}
```

Dangling Pointers Example: 2

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
        a.val = 20;
        p = &a;
        a.next = (node *)malloc(sizeof(node));

}
```

Dangling Pointers Example: 2

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
        a.val = 20;
        p = &a;
        a.next = (node *)malloc(sizeof(node));
        a.next->val = 50;

}
```

Dangling Pointers Example: 2

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
        a.val = 20;
        p = &a;
        a.next = (node *)malloc(sizeof(node));
        a.next->val = 50;
        q = a.next;

}
```

Dangling Pointers Example: 2

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
        a.val = 20;
        p = &a;
        a.next = (node *)malloc(sizeof(node));
        a.next->val = 50;
        q = a.next;
        free(a.next);

}
```

Dangling Pointers Example: 2

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
        a.val = 20;
        p = &a;
        a.next = (node *)malloc(sizeof(node));
        a.next->val = 50;
        q = a.next;
        free(a.next);
        q->val = 100; // segfault

}
```

Dangling Pointers Example: 2

Remember

- Dereferencing a dangling pointer is a cause of segmentation fault
- Dereferencing can occur using * or [ ]
- Having a dangling pointer does not cause segmentation fault
  - Dereferencing it causes
  - Having dangling pointers is not wrong, but why have them?

Garbage Memory

```
int *p, i;
p = malloc
        (sizeof(int) * 4);
p = &i;
```

- p was pointing to malloced memory
- In the end p points to i
- How to access the malloced memory now?
  - No way! It's Lost!
  - It became “garbage memory”
- Problems like this result in memory waste
  - Also called as Memory leak
- Solution
  - Keep another point pointing to malloced memory

    p = malloc(sizeof(int) * 4);

    q = p;

    p = &i;
  - Malloced memory is available through pointer q

Segmentation Fault

What is segmentation fault?

- A program accessing an “illegal” memory location
  - Is punished by the operating system for doing so
  - The program gets “terminated” (killed)
  - “segmentation” comes from “memory management” concepts in operating systems
- It is ALWAYS a fault of the programmer!
- Beware
  - Bad compilers may generate machine code which hide some memory violations
  - If a programmer does illegal memory access, segmentation fault may not occur sometimes!
    - OS may “forgive” your program :-;

Some Typical Reasons for Seg-fault

Deferencing Dangling Pointers

Array Index Violation

Incorrect Function Arguments

Dangling Pointer Dereference: Some examples

```
int *p, i;
*p = 20;
```

```
int *p, i;
p = malloc(
        sizeof(int)*2);
free(p);
p[2] = 20;
```

```
int *p, i;
p = &i;
p[1] = 20;
```

```
int *f(int *p) {
    int x = *p + 2;
    return &x;
}
int main() {
    int i, *q;
    scanf("%d", &i);
    q = f(&i);
    *q = 20;
}
```

Don't return address of a local variable! The variable disappears after function call, and the returned address is a dangling pointer.

Array Index Violation

Valid indices for an array of size n: 0 .. n-1

Accessing any index <=-1 or >=n *may* result in seg-fault (it may not result in some cases, but it is STILL WRONG to do it!)

Try this code:

```
#include <stdio.h>
int main() {
    int a[16], i = 0;
    while(1) {
        a[i] = i;
        printf("%d\n", a[i]);
        i++;
    }
}
```

At what value of i does the code seg-fault? Try 2-3 times.

Reason: OS may allocate more than 16-integer size memory for the program. So the seg fault *may not* occur at index 16 or slightly higher indices also. Rule: wrong to access index >=n

Functions: Pointer Arguments

- Rule: When you want a function to change ACTUAL arguments
  - Function takes pointer arguments
  - Call passes address of “actual argument”
- Example: Swap function (correct code)

```
void swap(int *a, int *b ) {
        int temp;
        temp = *a;
        *a = * b;
        *b = temp;
}
int main() {
    int m = 20, n = 30;
    swap(&m, &n);
```

Incorrect Pointer Aruments - Segfault

```
int i;

scanf("%d", i);
```

Scanf tries to do something like

*i = value // segfault here

Segfault occurs in scanf, although the reason is call to scanf

- Note that this is basically a dangling pointer dereference which took place inside scanf()

Guidelines to avoid segfaults

- Always initialize pointers to NULL
  - This will not avoid segfaults, but may lead to early detection of a problem
- While accessing arrays, make sure that array index is valid
- Never return address of local variable of a function
- DO NOT IGNORE compiler's warnings
  - Compilers often warn about potential dangling references, type conversions which have a potential for segfaults
  - Make sure that you rewrite code to remove all warnings
  - Use “-Wall” option with gcc. > cc -Wall program.c -o program

Let’s draw diagrams for some programs using pointers, malloc, free, ...

Introduction to Linux Desktop
and
Command Line
11 Jan 2022

Abhijit A. M.
abhijit.comp@coep.ac.in

Why GNU/Linux ?

Why GNU/Linux ?

1. Programmer's Paradise : most versatile, vast, all pervasive programming environment

2. Free Software (or Open Source?) : Free as in freedom. *Freely* Use, copy, modify, redistribute.

3. Highly Productive : Do more in less time

4. Better quality, more secure, very few crashes

Why Command Line ?

1. Not for everyone ! Absolutely !

2. Those who do it are way more productive than others

3. Makes you think !

4. Portable. Learn once, use everywhere on all Unixes, Linuxes, etc.

Few Key Concepts

- *Files* don't open themselves
  - Always some application/program open()s a file.
- Files don't display themselves
  - A file is displayed by the program which opens it. Each program has it's own way of handling ifles

Few Key Concepts

- **Programs don't run themselves**
  - **You click on a program, or run a command --> equivalent to request to Operating System to run it. The OS runs your program**
- **Users (humans) request OS to run programs, using Graphical or Command line interface**
  - **and programs open files**

Path names

- Tree like directory structure
- Root directory called /
- A complete path name for a file
  - /home/student/a.c
- Relative path names

  concept: every running program has a *current* working directory

  . current directory

  .. parent directory

  ./Desktop/xyz/../p.c

A command

- Name of an executable file
  - For example: 'ls' is actually “/bin/ls”
- Command takes arguments
  - E.g. ls /tmp/
- Command takes options
  - E.g. ls -a

A command

- Command can take both arguments and options
  - E.g. ls -a /tmp/
- Options and arguments are basically argv[] of the main() of that program

Basic Navigation Commands

- pwd
- ls
  - ls -l
  - ls -l /tmp/
  - ls -l /home/student/Desktop
  - ls -l ./Desktop
  - ls -a

    \ls -F
- cd
  - cd /tmp/
  - cd
  - cd /home/student/Desktop
- notation: ~
  - cd ~
  - cd ~/Desktop
  - ls ~/Desktop

Map these commands to navigation using a graphical file browser

The Shell

- **Shell = Cover**
- **Covers some of the Operating System's "System Calls" (mainly fork+exec) for the *Applications***
- **Talks with Users and Applications and does some talk with OS**

Not a very accurate diagram !

http://code.snipcademy.com/tutorials/linux-command-line/environment/

The Shell

Shell waits for user's input

Requests the OS to run a program which the user has asked to run

Again waits for user's input

GUI is a Shell !

Let's Understand fork() and exec()

```
#include <unistd.h>
int main() {
        fork();
        printf("hi\n");
        return 0;
}
```

```
#include <unistd.h>
int main() {
        printf("hi\n");
        execl("/bin/ls", "ls",
NULL);
        printf("bye\n");
        return 0;
}
```

A simple shell

```
#include <stdio.h>
#include <unistd.h>
int main() {
        char string[128];
        int pid;
        while(1) {
                printf("prompt>");
                scanf("%s", string);
                pid = fork();
                if(pid == 0) {
                        execl(string, string, NULL);
                } else {
                        wait(0);
                }
        }
```

File Permissions on Linux

- Two types of users
  - root and non-root
  - Users can grouped into 'groups'
- 3 sets of 3 permission
  - Octal notation
  - Read = 4, Write = 2, Execute = 1
  - 644 means
    - Read-Write for owner, Read for Group, Read for others
- chmod command uses these notations
-

File Permissions on Linux

```
-rw-r--r-- 1 abhijit abhijit 1183744 May 16 12:48 01_linux_basics.ppt
-rw-r--r-- 1 abhijit abhijit  341736 May 17 10:39 Debian Family Tree.svg
drwxr-xr-x 2 abhijit abhijit    4096 May 17 11:16 fork-exec
-rw-r--r-- 1 abhijit abhijit 7831341 May 11 12:13 foss.odp
```

Owner size name

3 sets of 3 permissions

3 sets = user (owner), group, others

last-modification

hard link count

3 permissions = read, write, execute

File Permissions on Linux

- r on a file : can read the file
  - open(.... O_RDONLY) works
- w on a file: can modify the file
- x on a file: can ask the os to run the file as an executable program
- r on a directory: can do 'ls'
- w on a directory: can add/remove files from that directory (even without 'r'!)
- x on a directory: can 'cd' to that directory

Access rights examples

- **-rw-r--r--**
  Readable and writable for file owner, only readable for others

- **-rw-r-----**
  Readable and writable for file owner, only readable for users belonging to the file group.

- **drwx------**
  Directory only accessible by its owner

- **-------r-x**
  File executable by others but neither by your friends nor by yourself. Nice protections for a trap...

Man Pages

- Manpage
  - $ man ls
  - $ man 2 mkdir
  - $ man man
  - $ man -k mkdir
- Manpage sections
  - 1 User-level cmds and apps
    - /bin/mkdir
  - 2 System calls
    - int mkdir(const char *, …);
  - 3 Library calls
    - int printf(const char *, …);
  - 4 Device drivers and network protocols
    - /dev/tty
  - 5 Standard file formats
    - /etc/hosts
  - 6 Games and demos
    - /usr/games/fortune
  - 7 Misc. files and docs
    - man 7 locale
  - 8 System admin. Cmds
    - /sbin/reboot

GNU / Linux filesystem structure

Not imposed by the system. Can vary from one system to the other, even between two GNU/Linux installations!

/	Root directory
/bin/	Basic, essential system commands
/boot/	Kernel images, initrd and configuration files
/dev/	Files representing devices
	/dev/hda: first IDE hard disk
/etc/	System and application configuration files
/home/	User directories
/lib/	Basic system shared libraries

GNU / Linux filesystem structure

/lost+found	Corrupt files the system tried to recover
/media	Mount points for removable media:
	/media/usbdisk, /media/cdrom
/mnt/	Mount points for temporarily mounted filesystems
/opt/	Specific tools installed by the sysadmin
	/usr/local/ often used instead
/proc/	Access to system information
	/proc/cpuinfo, /**proc/version** …
/root/	root user home directory
/sbin/	Administrator-only commands
/sys/	System and device controls
	(cpu frequency, device power, etc.)

GNU / Linux filesystem structure

/tmp/	Temporary files
/usr/	Regular user tools (not essential to the system) **/usr/bin/**, **/usr/lib/**, **/usr/sbin**…
/usr/local/	Specific software installed by the sysadmin (often preferred to **/opt/**)
/var/	Data used by the system or system servers **/var/log/**, **/var/spool/mail** (incoming mail), **/var/spool/lpd** (print jobs)…

Files: cut, copy, paste, remove,

- cat <filenames>
  - cat /etc/passwd
  - cat fork.c
  - cat <filename1> <filename2>
- cp <source> <target>
  - cp a.c b.c
  - cp a.c /tmp/
  - cp a.c /tmp/b.c
  - cp -r ./folder1 /tmp/
  - cp -r ./folder1 /tmp/folder2
- mv <source> <target>
  - mv a.c b.c
  - mv a.c /tmp/
  - mv a.c /tmp/b.c
- rm <filename>
  - rm a.c
  - rm a.c b.c c.c
  - rm -r /tmp/a
- mkdir
  - mkdir /tmp/a /tmp/b
- rmdir
  - rmdir /tmp/a /tmp/b

Useful Commands

- echo
  - echo hi
  - echo hi there
  - echo “hi there”
  - j=5; echo $j
- sort
  - sort
  - sort < /etc/passwd
- firefox
- libreoffice
- grep
  - grep bash /etc/passwd
  - grep -i display /etc/passwd
  - egrep -i 'a|b' /etc/passwd
- less <filename>
- head <filename>
  - head -5 <filename>
  - tail -10 <filename>

Useful Commands

- **alias**

  alias ll='ls -l'

- **tar**

  tar cvf folder.tar folder

- **gzip**

  gzip a.c

- **touch**

  touch xy.txt

  touch a.c

- **strings**

  strings a.out

- **adduser**

  sudo adduser test

- **su**

  su administrator

Useful Commands

- **df**

  df -h

- **du**

  du -hs .

- **bc**
- **time**
- **date**
- **diff**
- **wc**

Network Related Commands

- ifconfig
- ssh
- scp
- telnet
- ping
- w
- last
- whoami

Unix job control

- **Start a background process:**
  - **gedit a.c &**
  - **gedit**
    ***hit* ctrl-z**
    bg
- **Where did it go?**
  - **jobs**
  - **ps**
- **Terminate the job: kill it**
  - **kill %*jobid***
  - **kill *pid***
- **Bring it back into the foreground**
  - **fg %1**

Shell Wildcards

- ? (question mark)
  - Any ene character
  - ls a?c
  - ls ??c
- *
  - any number of characters
  - ls *
  - ls d*
  - echo *
- [ ]
  - Matches a range
  - ls a[1-3].c
- {}
  - ls pic[1-3].{txt,jpg}

Configuration Files

- Most applications have configuration files in TEXT format
- Most of them are in */etc*
- */etc/*passwd and /*etc*/shadow
  - Text files containing user accounts
- */etc/*resolv.conf
  - DNS configuration
- */etc/network/interfaces*
  - *Network configuration*
- */etc/*hosts
  - Local database of Hostname-IP mappings
- */etc/*apache2/apache2.conf
  - Apache webserver configuration

~/.bashrc file

- **~/.bashrc**

  Shell script read each time a **bash** shell is started
- You can use this file to define
  - Your default environment variables (**PATH**, **EDITOR**...).
  - Your aliases.
  - Your prompt (see the **bash** manual for details).
  - A greeting message.
- Also ~/.bash_history

Special devices (1)

Device files with a special behavior or contents

- **/dev/null**
  The data sink! Discards all data written to this file.
  Useful to get rid of unwanted output, typically log information:
  mplayer black_adder_4th.avi &> /dev/null

- **/dev/zero**
  Reads from this file always return **\0** characters
  Useful to create a file filled with zeros:
  dd if=/dev/zero of=disk.img bs=1k count=2048

See **man null** or **man zero** for details

Special devices (2)

- **/dev/random**
  Returns random bytes when read. Mainly used by cryptographic programs. Uses interrupts from some device drivers as sources of true randomness ("entropy").
  Reads can be blocked until enough entropy is gathered.

- **/dev/urandom**
  For programs for which pseudo random numbers are fine.
  Always generates random bytes, even if not enough entropy is available (in which case it is possible, though still difficult, to predict future byte sequences from past ones).

See **man random** for details.

Special devices (3)

- **/dev/full**
  Mimics a full device.
  Useful to check that your application properly handles this kind of situation.

See **man full** for details.

Files names and inodes

Hard Links Vs Soft Links

Shell Programming

Shell Programming

- is “programming”
- Any programming: Use existing tool to create new utilities
- Shell Programming: Use shell facilities to create better utilities
  - Know “commands” --> More tools
  - Know how to combine them

Shell Variables

- Shell supports variables
  - Try:
    - j=5; echo $j
    - No space in j=5
  - Try
    - set
    - Shows all set variables
  - Try
    - a=10; b=20; echo ab
  - What did you learn?
  - All variables are strings

Shell's predefined variables

- **USER**
  - Name of current user
- **HOME**
  - Home directory of current $USER
- **PS1**
  - The prompt
- **LINES**
  - No. of lines of the screen
-
- **HOSTNAME**
  - Name of the computer
- **OLDPWD**
  - Previous working directory
- **PATH**
  - List of locations to search for a command's executable file
- **$?**
  - Return value of previou command

Redirection

- cmd > filename
  - Redirects the output to a file
  - Try:
    - ls > /tmp/xyz
      cat /tmp/xyz
    - echo hi > /tmp/abc
      cat /tmp/abc
- cmd < filename
  - Reads the input from a file instead of keyboard
  - Think of a command now!

Pipes

- Try
  - last | less
  - grep bash /etc/passwd | head – 1
  - grep bash /etc/passwd | head – 2 | tail -1
- Connect the output of LHS command as input of RHS command
- Concept of *filters* – programs which read input only from stdin (keyboard, e.g. scanf, getchar), and write output to *stdout* (e.g. printf, putchar)
- Programs can be connected using pipes if they are filters
- Most Unix/Linux commands are filters !

The *test* command

- test
  - test 10 -eq 10
  - test “10” == “10”
  - test 10 -eq 9
  - test 10 -gt 9
  - test “10” >= “9”
  - test -f /etc/passwd
  - test -d ~/desktop
  - ...

Shortcut notation for calling test

[ ]

[10 -eq 10]

Note the space after '[' and before ']'

The *expr* command and backticks

- expr
  - expr 1 + 2
  - a=2; expr $a + 2
  - a=2; b=3; expr $a + $b
  - a=2;b=3; expr $a * $b
  - a=2;b=3; expr $a | $b
- Used for mathematical calculations
- backticks ``
  - j=`ls`; echo $j
  - j=`expr 1 + 2`; echo $j
-

if then else

```
if [ $a -lt $b ]
then
    echo $a
else
    echo $b
fi
```

```
if [ $a -lt $b ];then
echo $a; else  echo $b;
fi
```

0 TRUE

Nonzero FALSE

while do done

```
while [ $a -lt $b ]
do
    echo $a
    a=`expr $a + 1`
done
```

```
while [ $a -lt $b ]
do
    echo $a
    a=$(($a + 1))
done
```

```
while [ $a -lt $b ]; do
echo $a;   a=`expr $a +
1`; done
```

for x in ... do done

```
for i in {1..10}
do
    echo $i
done

for i in *
do
    echo $i
done
```

```
for i in *; do   echo $i;
done
```

case ... esac

```
read space
case $space in
      [1-6]*)
        Message="one to 6"
        ;;
      [7-8]*)
        Message="7 or 8"
        ;;
      9[1-8])
       Message="9 with a number"
        ;;
      *)
       Message="Default"
        ;;
esac
echo $Message
```

Syntax

```
;;
)  after option
* for default
esac
```

Try these things

- Print 3rd line from /etc/passwd, which contains the word bash
- Print numbers from 1 to 1000
- Create files named like this: file1, file2, file3, ... filen where n is read from user
  - Read i%5th file from /etc/passwd and store it in file*i*
- Find all files ending in .c or .h and create a .tar.gz file of these files
-

The Golden Mantra

Everything can be done from command line !

Command line is far more powerful than graphical interface

Command line makes you a better programmer

Mounting

Partitions

Disk Management

File Action View Help

Volume	Layout	Type	File System	Status	Capacity	Free Space	% Fr
(E:)	Partition	Basic	FAT32	Healthy	9.76 GB	8.37 GB	85 %
(F:)	Partition	Basic	FAT32	Healthy	9.76 GB	7.24 GB	74 %
OLDDRIVE (H:)	Partition	Basic	FAT32	Healthy (A...	4.99 GB	586 MB	11 %
WINDOWS XP (C:)	Partition	Basic	FAT32	Healthy (S...	9.76 GB	2.61 GB	26 %
Windows Vista (G:)	Partition	Basic	NTFS	Healthy	8.00 GB	1.61 GB	20 %
XPBACKUP (I:)	Partition	Basic	FAT32	Healthy	4.99 GB	4.33 GB	86 %
XXCOPY (J:)	Partition	Basic	FAT32	Healthy	9.00 GB	4.32 GB	47 %

Disk 0
Basic
37.30 GB
Online

WINDOWS XP (C:)
9.77 GB FAT32
Healthy (System)

(E:)
9.77 GB FAT32
Healthy

(F:)
9.77 GB FAT32
Healthy

Windows Vista (G:)
8.00 GB NTFS
Healthy

Disk 1
Basic
19.01 GB
Online

OLDDRIVE (H:)
5.00 GB FAT32
Healthy (Active)

XPBACKUP (I:)
5.00 GB FAT32
Healthy

XXCOPY (J:)
9.01 GB FAT32
Healthy

Primary partition Extended partition Logical drive

Windows Namespace

c:\temp\songs\xyz.mp3

- Root is C:\ or D:\ etc
- Separator is also “\”

Windows Namespace

- C:\ D:\ Are partitions of the disk drive
- Typical convention: C: contains programs, D: contains data

- One “tree” per partition
  - Together they make a “forest”

Linux Namespace: On a partition

/usr/songs/xyz.mp3

- On every partition:
  - Root is “/”
  - Separator is also “/”

Linux namespace: Mount

- Linux namespace is a single “tree” and not a “forest” like Windows
- Combining of multiple trees is done through “mount”

Linux namespace
Mounting a partition

Mounting another partition on */home/guest/mydir/* directory

Mounted disk partition

/home/guest/mydir/x/b.cpp → way to access the file on the other disk partition

Let's go for a live installation Demo !

Some Shell Gimmics

Terminal Tricks

Ctrl + n : same as Down arrow.

Ctrl + p : same as Up arrow.

Ctrl + r : begins a backward search through command history.(keep pressing Ctrl + r to move backward)

Ctrl + s : to stop output to terminal.

Ctrl + q : to resume output to terminal after Ctrl + s.

Terminal Tricks

Ctrl + a : move to the beginning of line.

Ctrl + e : move to the end of line.

Ctrl + d : if you've type something, Ctrl + d deletes the character under the cursor, else, it escapes the current shell.

Ctrl + k : delete all text from the cursor to the end of line.

Ctrl + t : transpose the character before the cursor with the one under the cursor

Terminal Tricks

Ctrl + w : cut the word before the cursor; then Ctrl + y paste it

Ctrl + u : cut the line before the cursor; then Ctrl + y paste it

Ctrl + _ : undo typing.

Ctrl + l : equivalent to clear.

Ctrl + x + Ctrl + e : launch editor defined by $EDITOR to input your command.

Run from history

First: What's history?

Ans: Run 'history

$ history

$!53

$!!

$!cat

$!c

Math

```
$ echo $(( 10 + 5 ))  #15
$ x=1
$ echo $(( x++ )) #1 , notice that it is still 1, since it's post-incremen
$ echo $(( x++ )) #2
```

More Math

```
$ seq 10|paste -sd+|bc
# How does that work ?
```

Using expr

```
$ expr 10+20 #30
$ expr 10\*20 #600
$ expr 30 \> 20 #1
```

More Math

Using bc

$ bc

```
obase=16
ibase=16
AA+1
AB
```

More Math

Using bc

$ bc

```
ibase=16
obase=16
AA+1
07 17
# what went wrong?
```

Fun with grep

```
$ grep -Eo '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}'
# above will only search for IP addresses!
$ grep -v bbo filename
$ grep -w ‘abhijit’ /etc/passwd
$ grep -v '^#' file.txt
$ grep -v ^$  file.txt
```

xargs: convert stdin to args

$ find . | grep something | xargs rm

xargs is highly powerful

rsync

The magic tool to sync your folders!

$ rsync -rvupt ~/myfiles /media/abhijit/PD

$ rsync -rvupt --delete ~/myfiles /media/abhijit/PD

find

```
$ find .
$ find . -type f
$ find . -type d
$ find . -name '*.php'
$ find / -type f -size +4G
$ find . -type f -empty -delete
$ find . -type f | wc -l
```

Download : wget, curl

$ wget foss.coep.org.in

$ wget –r foss.coep.org.in

$ wget -r –convert-links foss.coep.org.in

$ wget -r –convert-links --no-parent foss.coep.org.in/fossmeet/

$ curl https://raw.githubusercontent.com/onceupon/Bash-Oneliner/master/README.md | pandoc -f markdown -t man | man -l -

 # curl is more powerful than wget. Curl can upload. Curl supports many protocols, wget only HTTP/FTP.

Random data

```
# shuffle numbers from 0-100, then pick 15 of them randomly
$ shuf -i 0-100 -n 15

# Random pick 100 lines from a file
$ shuf -n 100 filename

#generate 5 password each of length 13
$ pwgen 13 5

echo $((RANDOM % 10))
```

Run commands remotely

$ ssh administrator@foss.coep.org.in

$ ssh -X administrator@foss.coep.org.in

$ ssh -X administrator@foss.coep.org.in firefox

System Information

Show memory usage,. **# print 10 times, at 1 second interval**

$ free -c 10 -mhs 1

 # Display CPU and IO statistics for devices and partitions.
 # refresh every second

$ iostat -x -t 1

 # Display bandwidth usage on an network interface (e.g. enp175s0f0)

$ sudo iftop -i wlo1

 # Tell how long the system has been running and number of users

$ uptime

Surf the web

$ w3m

$ links

Add a user without commands

Know how to edit the /*etc*/passwd and /*etc/shadow files*

More tricks

Show 10 Largest Open Files

```
$ lsof / | awk '{ if($7 > 1048576) print $7/1048576 "MB" " " $9 " " $1 }' | sort -n -u | tail
```

Generate a sequence of numbers

```
$ echo {01..10}
```

More tricks

 # Rename all items in a directory to lower case

```
$ for i in *; do mv "$i" "${i,,}"; done
```

 # List IP addresses connected to your server on port 80

```
$ netstat -tn 2>/dev/null | grep :80 | awk '{print $5}' | cut -d: -f1 | sort | uniq -c | sort -nr | head
```

Credits:
https://onceupon.github.io/Bash-Oneliner/

http://www.bashoneliners.com/

User Administration

Users and Groups

- There is a privileged user called “root”
  - Can do anything, like “administrator” on Windows
  - Can't login in graphical mode !
- Other users are normal users
- Some users are given “sudo” privileges: called *sudoers*
  - Sudo means “do as a superuser”
  - Password is asked, when the otherwise normal user tries to do administrative task
  - The first user account created on Ubuntu, is by default with sudo privileges

Adding/Deleting/Changing users

- System → Administration → Users and Groups
- Click on “Add” to add a user
  - Asks for password !
  - Provide the details asked for
  - Verify the user was created, by doing 'switch user'
- Try 'deleting' the user created
- Groups: Various groups of users meant for different purposes
  - Every user by default belongs to her own group
  - Add the user explicitly to other groups

Software installation

Some terms

- **.deb**
  - **The "setup" file. The installer package. Similar to Setup.exe on windows.**
  - **Contains all *binary* files and some *shell scripts***
- **repository**
  - **A collection of .deb files, categorized according to type (security, main, etc.)**
- **Software source/ ubuntu mirror**
  - **A computer on internet having all .deb files for ubuntu**

Software Installation Concept

- **Online installation**
  - **Use the "Ubuntu Software Center" to select the software, click and install !**
  - **Software is fetched automatically and installed !**
  - **Much easier than Windows !**
- **Offline installation**
  - **Collect ALL .deb files for your application**
  - **Select all, and install using package manager**

Software Installation

- **When we install using Software Center**
  - **.deb files are stored in /var/cache/apt/archives folder**
- **One needs to be a *sudoer* to install software**
- **Try installing some software on your own and try them out !**

Network configuration

Setting up network for a desktop

- **DHCP**
  - **Nothing needs to be done !**
  - **Default during installing Linux**
- **Static I/P**
  - **System → Preferences → Network connections**
  - **System → Administration → Network**
  - **System → Administration → Network tools**

Network setup for wireless

- **Just plug and Play !**
- **Network icon shows available wireless network, just click and connect.**

Reliance/Tata/Idea USB devices

- Each has a different procedure
- One needs to search the web for setting it up
  - Most of the devices work plug and play on Ubuntu 12.04
  - Some do not work, as fault of the providers, they have not given an installer CD for Linux!
    - Still Linux community have found ways to work around it !
    - My Reliance netconnect worked faster and more steadily on Linux than Windows !

Disk Management

Partition

- What is C:\ , D:\, E:\ etc on your computer ?
  - "Drive" is the popular term
  - Typically one of them represents a CD/DVD RW
- What do the others represent ?
  - They are "partitions" of your "hard disk"

Partition

- Your hard disk is one contiguous chunk of storage
  - Lot of times we need to "logically separate" our storage
  - Partition is a "logical division" of the storage
  - Every "drive" is a partition
- A logical chunk of storage is partition
  - Hard disk partitions (C:, D:), CD-ROM, Pen drive, ...

Partitions

Disk Management
File Action View Help
Volume | Layout | Type | File System | Status | Capacity | Free Space | % Fr
(E:) | Partition | Basic | FAT32 | Healthy | 9.76 GB | 8.37 GB | 85 %
(F:) | Partition | Basic | FAT32 | Healthy | 9.76 GB | 7.24 GB | 74 %
OLDDRIVE (H:) | Partition | Basic | FAT32 | Healthy (A... | 4.99 GB | 586 MB | 11 %
WINDOWS XP (C:) | Partition | Basic | FAT32 | Healthy (S... | 9.76 GB | 2.61 GB | 26 %
Windows Vista (G:) | Partition | Basic | NTFS | Healthy | 8.00 GB | 1.61 GB | 20 %
XPBACKUP (I:) | Partition | Basic | FAT32 | Healthy | 4.99 GB | 4.33 GB | 86 %
XXCOPY (J:) | Partition | Basic | FAT32 | Healthy | 9.00 GB | 4.32 GB | 47 %
Disk 0
Basic
37.30 GB
Online
WINDOWS XP (C:)
9.77 GB FAT32
Healthy (System)
(E:)
9.77 GB FAT32
Healthy
(F:)
9.77 GB FAT32
Healthy
Windows Vista (G:)
8.00 GB NTFS
Healthy
Disk 1
Basic
19.01 GB
Online
OLDDRIVE (H:)
5.00 GB FAT32
Healthy (Active)
XPBACKUP (I:)
5.00 GB FAT32
Healthy
XXCOPY (J:)
9.01 GB FAT32
Healthy
Primary partition
Extended partition
Logical drive

Managing partitions and hard drives

- System → Administration → Disk Utility
- Had drive partition names on Linux
  - /dev/sda → Entire hard drive
  - /dev/sda1, /dev/sda2, /dev/sda3, ... Different partitions of the hard drive
  - Each partition has a *type* – ext4, ext3, ntfs, fat32, etc.
- Pen drives can also be managed from here
- Formatting can also be done from here

Boot from a live CD.
Go to terminal.

Find out the partition for your UBUNTU installation. Let's say it is /dev/sda5

```
$ sudo mount /dev/sda5 /tmp/a
$ cd /tmp/a
$ mount -t proc /proc ./proc
$ sudo mount --rbind /dev ./dev
$ sudo mount --rbind /run ./run
$ sudo mount --rbind /sys ./sys
$ sudo chroot .
# update-initramfs  -c -k 5.11.0-44-generic
# update-grub
# mount /dev/sda1 ./boot/efi/ # optional
# grub-install /dev/sda
```

Memory Management Basics

Summary

- Understanding how the processor architecture drives the memory management features of OS and system programs (compilers, linkers)
- Understanding how different hardware designs lead to different memory management schemes by operating systems

Addresses issued by CPU

- During the entire ‘on’ time of the CPU
  - Addresses are “issued” by the CPU on address bus
  - One address to fetch instruction from location specified by PC
  - Zero or more addresses depending on instruction
    - e.g. mov $0x300, r1 # move contents of address 0x300 to r1 --> one extra address issued on address bus

Memory layout of a C program

$ size /bin/ls

text	data	bss	dec	hex	filename
128069	4688	4824	137581	2196d	/bin/ls

Desired from a multi-tasking system

- Multiple processes in RAM at the same time (multi-programming)
- Processes should not be able to see/touch each other’s code, data (globals), stack, heap, etc.
- Further advanced requirements
  - Process could reside anywhere in RAM
  - Process need not be continuous in RAM
  - Parts of process could be moved anywhere in RAM

Different ‘times’

- Different actions related to memory management for a program are taken at different times. So let’s know the different ‘times’
- Compile time
  - When compiler is compiling your C code
- Load time
  - When you execute “./myprogram” and it’s getting loaded in RAM by loader i.e. exec()
- Run time
  - When the process is alive, and getting scheduled by the OS

Different types of Address binding

Which binding is actually used, is mandated by processor features + OS

- Compile time address binding
  - Address of code/variables is fixed by compiler
  - Very rigid scheme
  - Location of process in RAM can not be changed ! Non-relocatable code.
- Load time address binding
  - Address of code/variables is fixed by loader
  - Location of process in RAM is decided at load time, but can’t be changed later
  - Flexible scheme, relocatable code
- Run time address binding
  - Address of code/variables is fixed at the time of executing the code
  - Very flexible scheme , highly relocatable code
  - Location of process in RAM is decided at load time, but CAN be changed later also

Simplest case

- Suppose the address issued by CPU reaches the RAM controller directly

Simplest case

- How does this impact the compiler and OS ?
- When a process is running the addresses issued by it, will reach the RAM directly
- So exact addresses of globals, addresses in “jmp” and “call” must be part the machine instructions generated by compiler
  - How will the compiler know the addresses, at “compile time” ?

Example
address: instruction

1000: mov $0x300, r1
1004: add r1, -3
1008: jnz 1000

Sequence of addressed sent by CPU: 1000, 0x300, 1004, 1008, 1000, 0x300, ...

Simplest case

Example
1000: mov $0x300, r1
1004: add r1, -3
1008: jnz 1000

- Solution: compiler assumes some fixed addresses for globals, code, etc.
- OS loads the program exactly at the same addresses specified in the executable file. **Non-relocatable code.**
- Now program can execute properly.

Simplest case

Example
1000: mov $0x300, r1
1004: add r1, -3
1008: jnz 1000

- Problem with this solution
  - Programs once loaded in RAM must stay there, can't be moved
  - What about 2 programs?
    - Compilers being "programs", will make same assumptions and are likely to generate same/overlapping addresses for two different programs
    - Hence only one program can be in memory at a time !
    - No need to check for any memory boundary violations – all memory belongs to one process
- Example: DOS

Base/Relocation + Limit scheme

Figure 9.6 Hardware support for relocation and limit registers.

- Base and Limit are two registers inside CPU's Memory Management Unit
- 'base' is added to the address generated by CPU
- The result is compared with base+limit and if less passed to memory, else hardware interrupt is raised

Memory Management Unit (MMU)

Figure 9.4 Memory management unit (MMU).

- Is part of the CPU chip, acts on every memory address issue by execution unit of the CPU
- In the scheme just discussed, the base, limit calculation parts are part of MMU

Base/Relocation + Limit scheme

- Compiler's work
  - Assume that the process is one continous chunk in memory, with a size limit
  - Assume that the process starts at address zero (!) and calculate addresses for globals, code, etc. And accordingly generate machine code

Base/Relocation + Limit scheme

Figure 9.6 Hardware support for relocation and limit registers.

- OS's work
  - While loading the process in memory – must load as one continous segment
  - Fill in the 'base' register with the actual address of the process in RAM.
  - Setup the limit to be the size of the process as set by compiler in the executable file. *Remember the base+limit in OS's own data structures.*

Base/Relocation + Limit scheme

Figure 9.6 Hardware support for relocation and limit registers.

- Combined effect
  - **"Relocatable code"** – the process can go anywhere in RAM at the time of loading
  - Some memory violations can be detected – a memory access beyond base+limit will raise interrupt, thus running OS in turn, which may take action against the process

Example scenario of memory in base+limit scheme

It should be possible to have relocatable code even with “simplest case”

By doing extra work during “loading”.

How?

Next scheme: Segmentation Multiple base +limit pairs

- Multiple sets of base + limit registers
- Whenever an address is issued by execution unit of CPU, it will also include reference to some base register
  - And hence limit register paired to that base register will be used for error checking
- Compiler: can assume a separate chunk of memory for code, data, stack, heap, etc. And accordingly calculate addresses . Each “segment” starting at address 0.
- OS: will load the different ‘sections’ in different memory regions and accordingly set different ‘base’ registers
-

Next scheme: Multiple base +limit pairs

Next scheme: Multiple base +limit pairs, with further indirection

- Base + limit pairs can also be stored in some memory location (not in registers). Question: how will the cpu know where it’s in memory?
  - One CPU register to point to the location of table in memory
- Segment registers still in use, they give an index in this table
- This is x86 segmentation
  - Flexibility to have lot more “base+limits” in the array/table in memory

Next scheme:
Multiple base +limit pairs, with further indirection

Figure 9.22 IA-32 segmentation.

Problems with segmentation schemes

- OS needs to find a continuous free chunk of memory that fits the size of the “segment”
  - If not available, your exec() can fail due to lack of memory
- Suppose 50k is needed
  - Possible that mong 3 free chunks total 100K may be available, but no single chunk of 50k!
  - **External fragmentation**
- Solution to external fragmentation: **compaction** – move the chunks around and make a continous big chunk available. Time consuming, tricky.

Solving external fragmentation problem

- Process should not be continous in memory!
- Divide the continuous process image in smaller chunks (let’s say 4k each) and locate the chunks anywhere in the physical memory
  - Need a way to map the *logical* memory addresses into *actual physical memory addresses*

Figure 9.9 Paging model of logical and physical memory.

Figure 9.10 Paging example for a 32-byte memory with 4-byte pages.

Figure 9.8 Paging hardware.

Paging

- Process is assumed to be composed of equally sized “pages” (e.g. 4k page)
- Actual memory is considered to be divided into page “frames”.
- CPU generated logical address is split into a page number and offset
- A page table base register inside CPU will give location of an in memory table called page table
- Page number used as offset in a table called page table, which gives the physical page frame number
- Frame number + offset are combined to get physical memory address

Paging

- Compiler: assume the process to be one continous chunk of memory (!) . Generate addresses accordingly
- OS: at exec() time – allocate different frames to process, allocate a page table(!), setup the page table to map page numbers with frame numbers, setup the page table base register, start the process
- Now hardware will take care of all translations of logical addresses to physical addresses

X86 memory management

Figure 9.21 Logical to physical address translation in IA-32.

Segmentation in x86

Figure 9.22 IA-32 segmentation.

- The selector is automatically chosen using Code Segment (CS) register, or Data Segment (DS) register depending on which type of memory address is being fetched
- Descriptor table is in memory
- The location of Descriptor table (Global DT- GDT or Local DT – LDT) is given by a GDT-register i.e. GDTR or LDT-register i.e. LDTR
-

Figure 9.23 Paging in the IA-32 architecture.

Paging in x86

- Depending on a flag setup in CR3 register, either 4 MB or 4 KB pages can be enabled
- Page directory, page table are both in memory

XV6 bootloader

Abhijit A. M.
abhijit.comp@coep.ac.in

Credits:
xv6 book by Cox, Kaashoek, Morris
Notes by Prof. Sorav Bansal

A word of caution

- We begin reading xv6 code
- But it’s not possible to read this code in a “linear fashion”
  - The dependency between knowing OS concepts and reading/writing a kernel that is written using all concepts

What we have seen ....

- Compilation process, calling conventions
- Basics of Memory Management by OS
- Basics of x86 architecture
  - Registers, segments, memory management unit, addressing, some basic machine instructions,
- ELF files
  - Objdump, program headers
  - Symbol tables

Boot-process

- **Bootloader itself**
  - **Is loaded by the BIOS at a fixed location in memory and BIOS makes it run**
  - **Our job, as OS programmers, is to write the bootloader code**
- **Bootloader does**
  - **Pick up code of OS from a ‘known’ location and loads it in memory**
  - **Makes the OS run**
- **Xv6 bootloader: bootasm.S bootmain.c (see Makefile)**

bootloader

- BIOS Runs (automatically)
- Loads boot sector into RAM at 0x7c00
- Starts executing that code
  - Make sure that your bootloader is loaded at 0x7c00
  - Makefile has

```
bootblock:  bootblock.S bootmain.c
    $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S .....
    ...
    $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o
    bootasm.o bootmain.o
```

Resuls in:

```
00007c00 <start>:  in bootblock.asm
```

Processor starts in real mode

- **Processor starts in real mode – works like 16 bit 8088**
- **eight 16-bit general-purpose registers,**
- **Segment registers %cs, %ds, %es, and %ss --> additional bits necessary to generate 20-bit memory addresses from 16-bit registers.**

  addr = seg << 4 + addr

Effective memory translation in the beginning
At _start in bootasm.S:

%cs=0 %ip=7c00.

So effective address = 0*16+ip = ip

bootloader

- First instruction is ‘cli’
  - disable interrupts
- So that until your code loads all hardware interrupt handlers, no interrupt will occur

Zeroing registers

```
# Zero data segment registers DS, ES, and SS.
xorw    %ax,%ax          # Set %ax to zero
movw    %ax,%ds          # -> Data Segment
movw    %ax,%es          # -> Extra Segment
movw    %ax,%ss          # -> Stack Segment
```

- zero ax and ds, es, ss
- BIOS did not put in anything perhaps

A not so necessary detail Enable 21 bit address

```
seta20.1:
  inb     $0x64,%al            # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1
  movb    $0xd1,%al            # 0xd1 -> port
0x64
  outb    %al,$0x64
seta20.2:
  inb     $0x64,%al            # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2
  movb    $0xdf,%al            # 0xdf -> port 0x60
  outb    %al,$0x60
```

- **Seg:off with 16 bit segments can actually address more than 20 bits of memory. After 0x100000 (=2^20), 8086 wrapped addresses to 0.**
- **80286 introduced 21st bit of address. But older software required 20 bits only. BIOS disabled 21st bit. Some OS needed 21st Bit. So enable it.**
- **Write to Port 0x64 and 0x60 -> keyboard controller**
  - **to enable 21st bit out of address translation**
  - **Why? Before the A20, i.e. 21st bit was introduced, it belonged to keyboard controller**
  - **For more details see https://en.wikipedia.org/wiki/A20_line**

    https://en.wikipedia.org/wiki/A20_line

After this

Some instructions are run to enter protected mode

And further code runs in protected mode

Real mode Vs protected mode

- Real mode 16 bit registers
- Protected mode
  - Enables segmentation + Paging both
    - No longer seg*16+offset calculations
    - Segment registers is index into segment descriptor table. But segment:offset pairs continue
      - mov %esp, $32 # SS will be used with esp
    - More in next few slides
    - Other segment registers need to be explicitely mentioned in instructions
      - Mov FS:$200, 30
  - 32 bit registers
    - can address upto 2^{32} memory
    - Can do arithmetic in 32 bits

X86 address : protected mode address translation

Both Segmentation and Paging are used in x86
X86 allows optionally one-level or two-level paging
Segmentation is a must to setup, paging is optional (needs to be enabled)
Hence different OS can use segmentation+paging in different ways

X86 segmentation

Figure 8.22 IA-32 segmentation.

Paging concept, hierarchical paging

Figure 9.8 Paging hardware.

Figure 9.16 Address translation for a two-level 32-bit paging architecture.

X86 paging

Figure 8.23 Paging in the IA-32 architecture.

Page Directory Entry (PDE)
Page Table Entry (PTE)

PDE

PTE

P	Present
W	Writable
U	User
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use

CR3

31
12 11
5 4 3 2 0
CR3
Page-Directory-Table Base Address
PCD
PWT
PWT Page-level writes transparent
PCT Page-level cache disable

CR4

31
11 10 9 8 7 6 5 4 3 2 1 0
CR4
OSXM
OSFX
PCE
PGE
MCE
PAE
PSE
DE
TSD
PVI
VME
VME Virtual-8086 mode extensions
PVI Protected-mode virtual interrupts
TSD Time stamp disable
DE Debugging extensions
PSE Page size extensions
PAE Physical-address extension
MCE Machine check enable
PGE Page-global enable
PCE Performance counter enable
OSFXSR OS FXSAVE/FXRSTOR support
OSXMM-EXCPT OS unmasked exception support

Segmentation + Paging

Selector value is implicit based on address being accessed.
Instruction: CS
Stack Variable: SS
Data: DS
etc.

Segmentation + Paging setup of xv6

- xv6 configures the segmentation hardware by setting Base = 0, Limit = 4 GB
  - translate logical to linear addresses without change, so that they are always equal.
  - Segmentation is pratically off
- Once paging is enabled, the only interesting address mapping in the system will be linear to physical.
  - In xv6 paging is NOT enabled while loading kernel
  - After kernel is loaded 4 MB pages are used for a while
  - Later the kernel switches to 4 kB pages!

GDT Entry

31 … 16			15 … 0	
Base 0:15			Limit 0:15	
63 … 56	55 … 52	51 … 48	47 … 40	39 … 32
Base 24:31	Flags	Limit 16:19	Access Byte	Base 16:23

asm.h

```
#define SEG_ASM(type,base,lim)                                  \
        .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);      \
        .byte (((base) >> 16) & 0xff), (0x90 | (type)),         \
                (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
```

Segment selector

TI Table index (0=GDT, 1=LDT)
RPL Requester privilege level

Note in 16 bit mode, segment selector is 16 bit, here it's 13 bit + 3 bits

EFLAGS register

Status flags
- CF Carry flag
- PF Parity flag
- AF Auxiliary carry flag
- ZF Zero flag
- SF Sign flag
- OF Overflow flag

Control flags
- DF Direction flag

System flags
- TF Trap flag
- IF Interrupt enable flag
- IOPL I/O privilege level
- NT Nested task
- RF Resume flag
- VM Virtual-8086 mode
- AC Alignment check
- VIF Virtual intr. flag
- VIP Virtual intr. pending
- ID ID flag

lgdt

```
lgdt    gdtdesc
...
# Bootstrap GDT
.p2align 2   # force 4 byte alignment
gdt:
  SEG_NULLASM     # null seg
  SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
  SEG_ASM(STA_W, 0x0, 0xffffffff)

   # data seg

gdtdesc:
  .word   (gdtdesc - gdt - 1)
# sizeof(gdt) - 1
  .long   gdt
```

- load the processor's (GDT) register with the value gdtdesc which points to the table gdt.
- table gdt : The table has a null entry, one entry for executable code, and one entry to data.
- all segments have a base address of zero and the maximum possible limit
- The code segment descriptor has a flag set that indicates that the code should run in 32-bit mode
- With this setup, when the boot loader enters protected mode, logical addresses map one-to-one to physical addresses.
- At gdtdesc we have this data

  2, <4 byte addr of gdt>

  Total 6 bytes

  - GDTR is : <address 4 byte>, <table limit 2 byte>
  - So lgdt gdtdesc loads these two values in GDTR

bootasm.S after “lgdt gdtdesc” till jump to “entry”

Base	Limit	Permissions
0	4GB	Write
0	4GB	Read, Execute
0	0	0

GDT

Still
Logical Address = Physical address!

But with GDT in picture and
Protected Mode operation

During this time,

Loading kernel from ELF into physical memory

Addresses in “kernel” file translate to same physical address!

Prepare to enable protected mode

- Prepare to enable protected mode by setting the 1 bit (CR0_PE) in register %cr0

```
movl    %cr0, %eax
orl     $CR0_PE, %eax
movl    %eax, %cr0
```

CR0

PE	Protection enabled	ET	Extension type	NW	Not write-through
MP	Monitor coprocessor	NE	Numeric error	CD	Cache disable
EM	Emulation	WP	Write protect	PG	Paging
TS	Task switched	AM	Alignment mask		

PG: Paging enabled or not **WP: Write protecion on/off**

PE: Protection Enabled --> protected mode.

Complete transition to 32 bit mode

```
ljmp   $(SEG_KCODE<<3), $start32
```

Complete the transition to 32-bit protected mode by using a long jmp

to reload %cs (=1) and %eip (=start32).

Note that ‘start32’ is the address of next instruction after ljmp.

Note: The segment descriptors are set up with no translation (that is 0-4GB setting), so that the mapping is still the identity mapping.

Jumping to “C” code

```
 movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %ss                # -> SS: Stack Segment
  movw    $0, %ax                 # Zero segments not ready for use
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS

  # Set up the stack pointer and call into C.
  movl    $start, %esp
  call    bootmain
```

- **Setup Data, extra, stack segment with SEG_KDATA (=2), FS & GS (=0)**
- **Copy “$start” i.e. 7c00 to stack-ptr**
  - **It will grow from 7c00 to 0000**
- **Call bootmain() a C function**
  - **In bootmain.c**

Setup now

Logical Address = offset
Physical Address
CS, SS, etc.
Selector
3
2
1
0
DS
SS
CS
Base
Limit
Permissions
0
4GB
Write
0
4GB
Read, Execute
0
0
0
GDT

bootmain(): already in memory, as part of ‘bootblock’

- bootmain.c , expects to find a copy of the kernel executable on the disk starting at the second sector (sector = 1).
  - Why?
- The kernel is an ELF format binary
- Bootmain loads the first 4096 bytes of the ELF binary. It places the in-memory copy at address 0x10000
- readseg() is a function that runs OUT instructions in particular IO ports, to issue commands to read from Disk

```
void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* pa;

  elf = (struct elfhdr*)0x10000;  // scratch space

  // Read 1st page off disk
  readseg((uchar*)elf, 4096, 0);
```

bootmain()

- Check if it's really ELF or not
- Next load kernel code from ELF file "kernel" into memory

```
// Is this an ELF executable?
if(elf->magic != ELF_MAGIC)
    return;  // let bootasm.S handle error
```

ELF

```
struct elfhdr {
  uint magic;  // must equal ELF_MAGIC
  uchar elf[12];
  ushort type;
  ushort machine;
  uint version;
  uint entry;
  uint phoff; // where is program header table
  uint shoff;
  uint flags;
  ushort ehsize;
  ushort phentsize;
  ushort phnum;  // no. Of program header entries
  ushort shentsize;
  ushort shnum;
  ushort shstrndx;
};
```

ELF

```
// Program  header
struct proghdr {
  uint type; // Loadable segment , Dynamic linking information , Interpreter information , Thread-Local Storage template , etc.
  uint off; //Offset of the segment in the file image.
  uint vaddr; //Virtual address of the segment in memory.
  uint paddr; // physical address to load this program, if PA is relevant
  uint filesz; //Size in bytes of the segment in the file image.
  uint memsz; //Size in bytes of the segment in memory. May be 0.
  uint flags;
  uint align;
};
```

Run ‘objdump -x -a kernel | head -15’ & see this

kernel: file format elf32-i386
kernel
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c

Code to be loaded at KERNBASE + KERNLINK

Diff between memsz & filesz, will be filled with zeroes in memory

Program Header:
LOAD **off** 0x00001000 **vaddr** 0x80100000 **paddr** 0x00100000 align 2**12
filesz 0x0000a516 **memsz** 0x000154a8 **flags** rwx
STACK **off** 0x00000000 **vaddr** 0x00000000 **paddr** 0x00000000 align 2**4
filesz 0x00000000 **memsz** 0x00000000 **flags** rwx

Stack : everything zeroes

Load code from ELF to memory

```
// Load each program segment (ignores ph flags).
ph = (struct proghdr*)((uchar*)elf + elf->phoff);
eph = ph + elf->phnum;
// Abhijit: number of program headers
for(; ph < eph; ph++){
  // Abhijit: iterate over each program header
  pa = (uchar*)ph->paddr;
  // Abhijit: the physical address to load program
  /* Abhijit: read ph->filesz bytes, into 'pa',
     from ph->off in kernel/disk */
  readseg(pa, ph->filesz, ph->off);
  if(ph->memsz > ph->filesz)
    stosb(pa + ph->filesz, 0, ph->memsz - ph-
>filesz); // Zero the reminder section*/
}
```

Jump to Entry

```
// Call the entry point from the ELF header.
// Does not return!
/* Abhijit:
 * elf->entry was set by Linker using kernel.ld
 * This is address 0x80100000 specified in
kernel.ld
 * See kernel.asm for kernel assembly code).
 */
entry = (void(*)(void))(elf->entry);
entry();
```

Figure 8.23 Paging in the IA-32 architecture.

To understand further code

Remember: 4 MB pages are possible

From entry:
Till: inside main(), before kvmalloc()

From entry:
Till: inside main(), before kvmalloc()

Even now, every Logical address = Physical address, but through Page dir

entrypgdir in main.c, is used by entry()

```
#define PTE_P        0x001   // Present
#define PTE_W        0x002   // Writeable
#define PTE_U        0x004   // User
#define PTE_PS       0x080   // Page Size
#define PDXSHIFT     22      // offset of PDX in a linear address
```

```
__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {

  // Map VA's [0, 4MB) to PA's [0, 4MB)
  [0] = (0) | PTE_P | PTE_W | PTE_PS,

  // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB). This is entry 512
  [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

This is entry page directory during entry(), beginning of kernel
Mapping 0:0x400000 (i.e. 0: 4MB) to physical addresses 0:0x400000. is required as long as entry is executing at low addresses, but will eventually be removed.
This mapping restricts the kernel instructions and data to 4 Mbytes.

entry() in entry.S

```
entry:
  movl    %cr4, %eax
  orl     $(CR4_PSE), %eax
  movl    %eax, %cr4
  movl    $(V2P_WO(entrypgdir)), %eax
  movl    %eax, %cr3
  movl    %cr0, %eax
  orl     $(CR0_PG|CR0_WP), %eax
  movl    %eax, %cr0
  movl $(stack + KSTACKSIZE), %esp
  mov $main, %eax
  jmp *%eax
```

- # Turn on page size extension for 4Mbyte pages
- # Set page directory. 4 MB pages (temporarily only. More later)
- # Turn on paging.
- # Set up the stack pointer.
- # Jump to main(), and switch to executing at high addresses. The indirect call is needed because the assembler produces a PC-relative instruction for a direct jump.
-

More about entry()

```
movl    $(V2P_WO(entrypgdir)), %eax
movl    %eax, %cr3
```

-> Here we use physical address using V2P_WO because paging is not turned on yet

- **V2P is simple: substract 0x80000000 i.e. KERNBASE from address**

More about entry()

```
movl    %cr0, %eax
orl     $(CR0_PG|CR0_WP), %eax
movl    %eax, %cr0
```

This turns on paging

After this also, entry() is running and processor is executing code at lower addresses

- But we have already set 0'th entry in pgdir to address 0
- So it still works!

entry()

```
  movl $(stack + KSTACKSIZE), %esp

  mov $main, %eax

  jmp *%eax

.comm stack, KSTACKSIZE

# Abhijit: allocate here 'stack' of size = KSTACKSIZE
```

- # Set up the stack pointer.
- # Abhijit: +KSTACKSIZE is done as stack grows downwards
- # Jump to main(), and switch to executing at high addresses. The indirect call is needed because the assembler produces a PC-relative instruction for a direct jump.

bootmasm.S bootmain.c: Steps

1) Starts in “real” mode, 16 bit mode. Does some misc legacy work.
2) Runs instructions to do MMU set-up for protected-mode & only segmentation (0-4GB, identity mapping), changes to protected mode.
3) Reads kernel ELF file and loads it in RAM, as per instructions in ELF file
4) Sets up paging (4 MB pages)
5) Runs main() of kernel

Code from bootasm.S bootmain.c is over!
Kernel is loaded.
Now kernel is going to prepare itself

Processes

Abhijit A M
abhijit.comp@coep.ac.in

Process related data structures in kernel code

- Kernel needs to maintain following types of data structures for managing processes
  - List of all processes
  - Memory management details for each, files opened by each etc.
  - Scheduling information about the process
  - Status of the process
  - List of processes “waiting” for different events to occur,
  - Etc.

Figure 3.3 Process control block (PCB).

Process Control Block

- A record representing a process in operating system's data structures
- OS maintains a "list"of PCBs, one for each process
- Called "`struct task_struct`" in Linux kernel code and "`struct proc`" in xv6 code

Figure 3.3 Process control block (PCB).

Fields in PCB

- Process ID (PID)
- Process State
- Program counter
- Registers
- Memory limits of the process
- Accounting information
- I/O status
- Scheduling information
- array of file descriptors (list of open files)
- ...etc

List of open files

Program did

```
rfd = open("/etc/passwd", O_RDONLY)
```

List of open files

- The PCB contains an array of pointers, called file descriptor array (fd[ ]), pointers to structures representing files
- When open() system call is made
  - A new file structure is created and relevant information is stored in it
  - Smallest available of fd [ ] pointers is made to point to this new struct file
  - The index of this fd [ ] pointer is returned by open
- When subsequent calls are made to read(fd, ...) or write(fd, ...) , etc.
  - The kernel gets the “fd” as an index in the fd[ ] array and is able to locate the file structure for that file

```
// XV6 Code : Per-process state
enum procstate { UNUSED, EMBRYO, SLEEPING,
RUNNABLE, RUNNING, ZOMBIE };

struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this
process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};

struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

```
struct file {
  enum { FD_NONE,
FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

Process Queues/Lists inside OS

- Different types of queues/lists can be maintained by OS for the processes
  - A queue of processes which need to be scheduled
  - A queue of processes which have requested input/output to a device and hence need to be put on hold/wait
  - List of processes currently running on multiple CPUs
  - Etc.

```
// Linux data structure
struct task_struct {
    long state;/*state of the process */
    struct sched_entity se; /* scheduling information */
    struct task_struct *parent; /*this process's parent */
    struct list_head children; /*this process's children */
    struct files_struct *files; /* list of open files */
    struct mm_struct *mm;/*address space */
```

```
struct list_head {
    struct list_head
*next, *prev;
};
```

Figure 3.4 The ready queue and wait queues.

Figure 3.5 Queueing-diagram representation of process scheduling.

Figure 3.2 Diagram of process state.

Conceptual diagram

“Giving up” CPU by a process or blocking

```
int main() {
i = j + k;
scanf("%d", &k);
}
int scanf(char *x, ...) {
...
read(0, ..., ...);
}
int read(int fd, char *buf, int len) {
...
__asm__ { "int 0x80..."}
...
}
```

OS Syscall

```
sys_read(int fd, char *buf, int len) {
  file f  = current->fdarray[fd];
  int offset = f->position;
  ...
  disk_read(... , offset, ...);
  // Do what now?
  //asynchronous read
  //Interrupt will occur when the disk read is complete
  // Move the process from ready queue to a wait queue and call scheduler!
  // This is called "blocking"
  Return the data read ;
}
disk_read(...., offset, .... ) {
__asm__("outb PORT ..");
return;
}
```

“Giving up” CPU by a process or blocking

The relevant code in xv6 is in

Sleep()

The wakeup code is in wakeup() and wakeup1()

To be seen later

Context Switch

- Context
  - Execution context of a process
  - CPU registers, process state, memory management information, all configurations of the CPU that are specific to execution of a process/kernel
- Context Switch
  - Change the context from one process/OS to OS/another process
  - Need to save the old context and load new context
  - Where to save? --> PCB of the process

Context Switch

- Is an overhead
- No useful work happening while doing a context switch
- Time can vary from hardware to hardware
- Special instructions may be available to save a set of registers in one go

Figure 3.6 Diagram showing context switch from process to process.

Pecularity of context switch

- When a process is running, the function calls work in LIFO fashion
  - Made possible due to calling convention
- When an interrupt occurs
  - It can occur anytime
  - Context switch can happen in the middle of execution of any function
- After context switch
  - One process takes place of another
  - This “switch” is obviously not going to happen using calling convention, as no “call” is happening
  - Code for context switch must be in assembly!

NEXT: XV6 code overview

1. Understanding how traps are handled
2. How timer interrupt goes to scheduler
3. How scheduling takes place
4. How a “blocking” system call (e.g. read()) “blocks”

Inter Process Communication

Revision of process related concepts

- PCB, struct proc
- Process lifecycle – different states
- Memory layout
- Memory management
- Interrupts handling, system call handling, code from xv6
- Scheduler, code of scheduler in xv6

IPC: Inter Process Communication

- Processes within a system may be independent or cooperating
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing, e.g. copy paste
  - Computation speedup, e.g. matrix multiplication
  - Modularity, e.g. chrome – separate process for display, separate for fetching data
  - Convenience,
- Cooperating processes need interprocess communication (IPC)
- Two models of IPC
  - Shared memory
  - Message passing

Shared Memory Vs Message Passing

Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

Each requires OS to provide system calls for

- **Creating the IPC mechanism**
- **To read/write using the IPC mechanism**
- **Delete the IPC mechanism**

Note: processes communicating with each other with the help of OS!

Example of co-operating processes: Producer Consumer Problem

- **Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process**
  - **unbounded-buffer places no practical limit on the size of the buffer**
  - **bounded-buffer assumes that there is a fixed buffer size**

Example of co-operating processes: Producer Consumer Problem

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Can only use BUFFER_SIZE-1 elements

Example of co-operating processes: Producer Consumer Problem

- **Code of Producer**

```
while (true) {
      /* Produce an item */
    while (((in = (in + 1) % BUFFER SIZE count)  == out)
     ;   /* do nothing -- no free buffers */
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
}
```

Example of co-operating processes: Producer Consumer Problem

- **Code of Consumer**

```
while (true) {
        while (in == out)
                ; // do nothing -- nothing to consume

      // remove an item from the buffer
      item = buffer[out];
      out = (out + 1) % BUFFER SIZE;
      return item;
 }
```

Pipes

Pipes for IPC

- **Two types**
  - **Unnamed Pipes or ordinary pipes**
  - **Named Pipe**

Ordinary pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the write-end of the pipe)
- Consumer reads from the other end (the read-end of the pipe)
- Ordinary pipes are therefore unidirectional
- Requires a parent-child (or sibling, etc) kind of relationship between communicating processes

parent
fd(0)
fd(1)
child
fd(0)
fd(1)
pipe

Named pipes

- Also called FIFO
- Processes can create a “file” that acts as pipe. Multiple processes can share the file to read/write as a FIFO
- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems
- Is not deleted automatically by OS

Named pipes

- int mkfifo(const char *pathname, mode_t mode);
- Example

Shared Memory

System V shared memory

- Process first creates shared memory segment

```
segment id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

- Process wanting access to that shared memory must attach to it

```
shared_memory = (char *) shmat(id, NULL, 0);
```

- Now the process could write to the shared memory

```
sprintf(shared_memory, "Writing to shared memory");
```

- When done, a process can detach the shared memory from its address space

```
shmdt(shared_memory);
```

Example of Shared memory
POSIX Shared Memory

- **What is POSIX?**
  - **Portable Operating System Interface (POSIX)**
  - **family of standards**
  - **specified by the IEEE Computer Society**
  - **for maintaining compatibility between operating systems.**
  - **API (system calls), shells, utility commands for compatibility among UNIXes and variants**

POSIX Shared Memory

- shm_open
- ftruncate
- Mmap
- See the example in Textbook

Message passing

Message Passing

- Message system – processes communicate with each other using send(), receive() like syscalls given by OS
- IPC facility provides two operations:
  - send(message) – message size fixed or variable
  - Receive(message)
- If P and Q wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via send/receive
- Communication link can be implemented in a variety of ways

Message Passing using "Naming"

- Pass a message by "naming" the receiver
  - A) Direct communication with receiver
    - Receiver is identified by sender directly using it's name
  - B) Indirect communication with receiver
    - Receiver is identified by sender in-directly using it's 'location of receipt'

Message passing using direct communication

- **Processes must name each other explicitly:**
  - **send (P, message) – send a message to process P**
  - **receive(Q, message) – receive a message from process Q**
- **Properties of communication link**
  - **Links are established automatically**
  - **A link is associated with exactly one pair of communicating processes**
  - **Between each pair there exists exactly one link**
  - **The link may be unidirectional, but is usually bi-directional**

Message passing using IN-direct communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Message passing using IN-direct communication

- **Operations**
  - **create a new mailbox**
  - **send and receive messages through mailbox**
  - **destroy a mailbox**
- **Primitives are defined as:**
  - **send(A, message) – send a message to mailbox A**
  - **receive(A, message) – receive a message from mailbox A**

Message passing using IN-direct communication

- **Mailbox sharing**
  - **P1, P2, and P3 share mailbox A**
  - **P1, sends; P2 and P3 receive**
  - **Who gets the message?**
- **Solutions**
  - **Allow a link to be associated with at most two processes**
  - **Allow only one process at a time to execute a receive operation**
  - **Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.**

Message Passing implementation: Synchronization issues

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
  - Blocking send has the sender block until the message is received
  - Blocking receive has the receiver block until a message is available
- Non-blocking is considered asynchronous
  - Non-blocking send has the sender send the message and continue
  - Non-blocking receive has the receiver receive a valid message or null

Producer consumer using blocking send and receive

Producer

```
message next produced;
while (true) {
/* produce an item in next_produced */
send(next_produced);
}
```

Consumer

```
message next_consumed;
while (true) {
receive(next_consumed);
}
```

Message Passing implementation: choice of Buffering

- **Queue of messages attached to the link; implemented in one of three ways**
  1. **Zero capacity – 0 messages**
     - **Sender must wait for receiver (rendezvous)**
  2. **Bounded capacity – finite length of n messages**
     - **Sender must wait if link full**
  3. **Unbounded capacity – infinite length**
     - **Sender never waits**

Processes in xv6 code

Process Table

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

- One single global array of processes
- Protected by

  `ptable.lock`

Layout of process's VA space

xv6 schema!

different from Linux

Logical layout of memory for a process

- Address 0: code
- Then globals
- Then stack
- Then heap
- Each processe's address space maps kernel's text, data also --> so that system calls run with these mappings
- Kernel code can directly access user memory now

Kernel mappings in user address space actual location of kernel

- **Kernel is loaded at 0x100000 physical address**
- **PA 0 to 0x100000 is BIOS and devices**
- **Process's page table will map**

  VA 0x80000000 to PA 0x00000 and

  VA 0x8010000 to 0x100000

Kernel mappings in user address space actual location of kernel

- **Kernel is not loaded at the PA 0x80000000 because some systems may not have that much memory**
- **0x80000000 is called KERNBASE in xv6**

Imp Concepts

- A process has two stacks
  - user stack: used when user code is running
  - kernel stack: used when kernel is running on behalf of a process
- Note: there is a third stack also!
  - The kernel stack used by the scheduler itself
  - Not a per process stack

Struct proc

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state. allocated, ready to run, running, wait-
ing for I/O, or exiting.
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process. Process's context
  void *chan;                  // If non-zero, sleeping on chan. More when we discuss
sleep, wakeup
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files, used by open(), read(),...
  struct inode *cwd;           // Current directory, changed with "chdir()"
  char name[16];               // Process name (for debugging)
};
```

struct proc diagram: Very imp!

sz = ELF-code->memsz (includes data, check “ld -N”
+ 2*4096 (for stack)

In use only when you are in kernel on a “trap” = interrupt/syscall. “tf” always used. trapret,forkret used during fork()

Memory Layout of a user process

Memory Layout of a user process

After exec()

Note the argc, argv on stack

The "guard page" is just a mapping in page table. No frame allocated. It's marked as invalid. So if stack grows (due to many function calls), then OS will detect it with an exception

Handling Traps

Handling traps

- Transition from user mode to kernel mode
  - On a system call
  - On a hardware interrupt
  - User program doing illegal work (exception)
- Actions needed, particularly w.r.t. to hardware interrupts
  - Change to kernel mode & switch to kernel stack
  - Kernel to work with devices, if needed
  - Kernel to understand interface of device

Handling traps

- **Actions needed on a trap**
  - **Save the processor's registers (context) for future use**
  - **Set up the system to run kernel code (kernel context) on kernel stack**
  - **Start kernel in appropriate place (sys call, intr handler, etc)**
  - **Kernel to get all info related to event (which block I/O done?, which sys call called, which process did exception and what type, get arguments to system call, etc)**

Privilege level

- The x86 has 4 protection levels, numbered 0 (most privilege) to 3 (least privilege).
- In practice, most operating systems use only 2 levels: 0 and 3, which are then called kernel mode and user mode, respectively.
- The current privilege level with which the x86 executes instructions is stored in %cs register, in the field CPL.

Privilege level

- **Changes automatically on**

  “int” instruction

  hardware interrupt

  exeception

- **Changes back on**

  iret

- **“int” 10 --> makes 10th hardware interrupt. S/w interrupt can be used to create hardware interrupt’**
- **Xv6 uses “int 64” for actual system calls**

Interrupt Descriptor Table (IDT)

- IDT defines intertupt handlers
- Has 256 entries
  - each giving the %cs and %eip to be used when handling the corresponding interrupt.
- Interrupts 0-31 are defined for software exceptions, like divide errors or attempts to access invalid memory addresses.
- Xv6 maps the 32 hardware interrupts to the range 32-63
- and uses interrupt 64 as the system call interrupt

Interrupt Descriptor Table (IDT) entries

```
// Gate descriptors for interrupts and traps
struct gatedesc {
  uint off_15_0 : 16;   // low 16 bits of offset in segment
  uint cs : 16;         // code segment selector
  uint args : 5;        // # args, 0 for interrupt/trap
gates
  uint rsv1 : 3;        // reserved(should be zero I guess)
  uint type : 4;        // type(STS_{IG32,TG32})
  uint s : 1;           // must be 0 (system)
  uint dpl : 2;         // descriptor(meaning new) privilege
level
  uint p : 1;           // Present
  uint off_31_16 : 16;  // high bits of offset in segment
};
```

Setting IDT entries

```
void
tvinit(void)
{
  int i;
  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
            vectors[T_SYSCALL], DPL_USER);
  /* value 1 in second argument --> don't disable
interrupts
         * DPL_USER means that processes can raise this
interrupt. */
    initlock(&tickslock, "time");
}
```

Setting IDT entries

```
#define SETGATE(gate, istrap, sel, off, d)                    \
{                                                             \
  (gate).off_15_0 = (uint)(off) & 0xffff;                    \
  (gate).cs = (sel);                                          \
  (gate).args = 0;                                            \
  (gate).rsv1 = 0;                                            \
  (gate).type = (istrap) ? STS_TG32 : STS_IG32;               \
  (gate).s = 0;                                               \
  (gate).dpl = (d);                                           \
  (gate).p = 1;                                               \
  (gate).off_31_16 = (uint)(off) >> 16;                      \
}
```

Setting IDT entries

Vectors.S

```
# generated by vectors.pl - do not edit
# handlers
.globl alltraps
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
.globl vector1
vector1:
  pushl $0
  pushl $1
 jmp alltraps
```

trapasm.S

```
#include "mmu.h"
# vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  Pushal
  ....
```

How will interrupts be handled?

On int instruction/interrupt the CPU does this:

- Fetch the n'th descriptor from the IDT, where n is the argument of int.
- Check that CPL in %cs is <= DPL, where DPL is the privilege level in the descriptor.
- Save %esp and %ss in CPU-internal registers, but only if the target segment selector's PL < CPL.
  - Switching from user mode to kernel mode. Hence save user code's SS and ESP
- Load %ss and %esp from a task segment descriptor.
  - Stack changes to kernel stack now. TS descriptor is on GDT, index given by TR register. See switchuvm()
- Push %ss. // optional
- Push %esp. // optional (also changes ss,esp using TSS)
- Push %eflags.
- Push %cs.
- Push %eip.
- Clear the IF bit in %eflags, but only on an interrupt.
- Set %cs and %eip to the values in the descriptor.

After “int” ‘s job is done

- IDT was already set
  - Remember vectors.S
- So jump to 64th entry in vector’s

```
vector64:
  pushl $0
  pushl $64
  jmp alltraps
```

  - So now stack has ss, esp,eflags, cs, eip, 0 (for error code), 64
  - Next run alltraps from trapasm.S

alltraps:

```
# Build trap frame.
pushl %ds
pushl %es
pushl %fs
pushl %gs
pushal // push all gen purpose regs
# Set up data segments.
movw $(SEG_KDATA<<3), %ax
movw %ax, %ds
movw %ax, %es
# Call trap(tf), where tf=%esp
pushl %esp # first arg to trap()
call trap
addl $4, %esp
```

- Now stack contains
- ss, esp,eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi
  - This is the struct trapframe !
  - So the kernel stack now contains the trapframe
  - Trapframe is a part of kernel stcak

trap()

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
  switch(tf->trapno){
  .....
```

- Argument is trapframe
- In alltraps
  - Before “call trap”, there was “push %esp” and stack had the trapframe
  - Remember calling convention --> when a function is called, the stack contains the arguments in reverse order (here only 1 arg)

trap()

- Has a switch
  - switch(tf->trapno)
  - Q: who set this trapno?
- Depending on the type of trap
  - Call interrupt handler
    - Timer
      - wakeup(&ticks)
    - IDE: disk interrupt
      - Ideintr()
    - KBD
      - Kbdintr()
    - COM1
      - Uatrintr()
    - If Timer
      - Call yield() -- calls sched()
    - If process was killed (how is that done?
      - Call exit()!

when trap() returns

- #Back in alltraps

```
call trap
addl $4, %esp

# Return falls through to trapret...
.globl trapret
trapret:
popal
popl %gs
popl %fs
popl %es
popl %ds
addl $0x8, %esp  # trapno and errcode
iret
```

-

- Stack had (trapframe)
  - ss, esp,eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi, esp
- add $4 %esp
  - esp
- popal
  - eax, ecx, edx, ebx, oesp, ebp, esi, edi
- Then gs, fs, es, ds
- add $0x8, %esp
  - 0 (for error code), 64
- iret
  - ss, esp,eflags, cs, eip,

Scheduler

Scheduler – in most simple terms

- **Selects a process to execute and passes control to it !**
  - **The process is chosen out of “READY” state processes**
  - **Saving of context of “earlier” process and loading of context of “next” process needs to happen**
- **Questions**
  - **What are the different scenarios in which a scheduler called ?**
  - **What are the intricacies of “passing control”**
  - **What is “context” ?**

Steps in scheduling scheduling

- Suppose you want to switch from P1 to P2 on a timer interrupt
- P1 was doing

```
F() {  i++; j++;}
```

- P2 was doing

```
G() { x--; y++; }
```

- P1 will experience a timer interrupt, switch to kernel (scheduler) and scheduler will schedule P2

4 stacks need to change!

- **User stack of process -> kernel stack of process**
  - **Switch to kernel stack**
  - **The normal sequence on any interrupt !**
- **Kernel stack of process -> kernel stack of scheduler**
  - **Why?**
- **Kernel stack of scheduler -> kernel stack of new process . Why?**
- **Kernel stack of new process -> user stack of new process**

scheduler()

- Disable interrupts
- Find a RUNNABLE process. Simple round-robin!
- c->proc = p
- switchuvm(p) : Save TSS of scheduler’s stack and make CR3 to point to new process pagedir
- p->state = RUNNING
- swtch(&(c->scheduler), p->context)

swtch

```
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

scheduler()

- swtch(&(c->scheduler), p->context)
- Note that when scheduler() was called, when P1 was running
- After call to swtch() shown above
  - The call does NOT return!
  - The new process P2 given by ‘p’ starts running !
  - Let’s review swtch() again

swtch(old, new)

- The magic function in swtch.S
- Saves callee-save registers of old context
- Switches esp to new-context's stack
- Pop callee-save registers from new context

  ret

  - where? in the case of first process – returns to forkret() because stack was setup like that !
  - in case of other processes, return where?
    - Return address given on kernel stack. But what's that?
    - The EIP in p->context
    - When was EIP set in p->context ?

scheduler()

- Called from?
  - mpmain()
  - No where else!
- sched() is another scheduler function !
  - Who calls sched() ?
    - exit() - a process exiting calls sched ()
    - yield() - a process interrupted by timer calls yield()
    - sleep() - a process going to wait calls sleep()

sched()

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();
  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
/*A*/  mycpu()->intena = intena;
}
```

- get current process
- Error checking code (ignore as of now)
- get interrupt enabled status on current CPU (ignore as of now)
- call to swtch
  - Note the arguments' order
  - p->context first, mycpu()->scheduler second
- swtch() is a function call
  - pushes address of */*A*/* on stack of current process p
  - switches stack to mycpu()->scheduler. Then pops EIP from that stack and jumps there.
  - when was mycpu()->scheduler set? Ans: during scheduler()!

sched() and schduler()

```
sched() {
...
  swtch(&p->context, mycpu()->scheduler); /* X */

}
```

```
scheduler(void) {
...
      swtch(&(c->scheduler), p->context);  / * Y */
}
```

- scheduler() saves context in c->scheduler, sched() saves context in p->context
- after swtch() call in sched(), the control jumps to Y in scheduler
  - Switch from process stack to scheduler's stack
- after swtch() call in scheduler(), the control jumps to X in sched()
  - Switch from scheduler's stack to new process's stack
- Set of co-operating functions

sched() and scheduler() as co-routines

- In sched()

  swtch(&p->context, mycpu()->scheduler);

- In scheduler()

  swtch(&(c->scheduler), p->context);

- These two keep switching between processes
- These two functions work together to achieve scheduling
- Using asynchronous jumps
- Hence they are co-routines

To summarize

- On a timer interrupt during P1
  - trap() is called. Stack has changed from P1's user stack to P1's kernel stack
  - trap()->yield()
  - yield()->sched()
  - sched() -> swtch(&p->context, c->scheduler()
  - Stack changes to scheduler's kernel stack.
  - Switches to location "Y" in scheduler().
-
- Now the loop in scheduler()
  - calls switchkvm()
  - Then continues to find next process (P2) to run
  - Then calls switchuvm(p): changing the page table to the P2's page tables
  - then calls swtch(&c->scheduler, p2's->context)
  - Stack changes to P2's kernel stack.
  - P2 runs the last instruction it was was in ! Where was it?
    - mycpu()->intena = intena; in sched()
    - Then returns to the one who called sched() i.e. exit/sleep, etc
    - Finally returns from it's own "TRAP" handler and returns to P2's user stack and user code

Memory Management – Continued

More on Linking, Loading, Paging

Review of last class

- **MMU : Hardware features for MM**
- **OS: Sets up MMU for a process, then schedules process**
- **Compiler : Generates object code for a particular OS + MMU architecture**
- **MMU: Detects memory violations and raises interrupt --> Effectively passing control to OS**

More on Linking and Loading

- **Static Linking: All object code combined at link time and a big object code file is created**
- **Static Loading: All the code is loaded in memory at the time of exec()**
- **Problem: Big executable files, need to load functions even if they do not execute**
- **Solution: Dynamic Linking and Dynamic Loading**

Dynamic Linking

- **Linker is normally invoked as a part of compilation process**
  - **Links**
    - **function code to function calls**
    - **references to global variables with “extern” declarations**
- **Dynamic Linker**
  - **Does not combine function code with the object code file**
  - **Instead introduces a “stub” code that is indirect reference to actual code**
  - **At the time of “loading” (or executing!) the program in memory, the “link-loader” (part of OS!) will pick up the relevant code from the library machine code file (e.g. libc.so.6)**

Dynamic Linking on Linux

```
#include <stdio.h>
int main() {
int a, b;
scanf("%d%d", &a, &b);
printf("%d %d\n", a, b);
return 0;
}
```

PLT: Procedure Linkage Table
used to call external procedures/functions whose address is to be resolved by the dynamic linker at run time.

Output of objdump -x -D

Disassembly of section .text:

```
0000000000001189 <main>:
    11d4:        callq  1080 <printf@plt>
```

Disassembly of section .plt.got:

```
0000000000001080 <printf@plt>:
    1080:      endbr64
    1084:      bnd jmpq *0x2f3d(%rip)        # 3fc8
<printf@GLIBC_2.2.5>
    108b:       nopl   0x0(%rax,%rax,1)
```

Dynamic Loading

- **Loader**
  - Loads the program in memory
  - Part of exec() code
  - Needs to understand the format of the executable file (e.g. the ELF format)
- **Dynamic Loading**
  - Load a part from the ELF file only if needed during execution
  - Delayed loading
  - Needs a more sophisticated memory management by operating system – to be seen during this series of lectures

Dynamic Linking, Loading

- **Dynamic linking necessarily demands an advanced type of loader that understands dynamic linking**
  - **Hence called ‘link-loader’**
  - **Static or dynamic loading is still a choice**
- **Question: which of the MMU options will alllow for which type of linking, loading ?**

Continous memory management

What is Continous memory management?

- **Entire process is hosted as one continous chunk in RAM**
- **Memory is typically divided into two partitions**
  - **One for OS and other for processes**
  - **OS most typically located in “high memory” addresses, because interrupt vectors map to that location (Linux, Windows) !**

Hardware support needed: base + limit (or relocation + limit)

Figure 9.6 Hardware support for relocation and limit registers.

Problems faced by OS

- Find a continuous chunk for the process being forked
- Different processes are of different sizes
  - Allocate a size parameter in the PCB
- After a process is over – free the memory occupied by it
- Maintain a list of free areas, and occupied areas
  - Can be done using an array, or linked list

Variable partition scheme

Figure 9.7 Variable partition.

Problem: how to find a “hole” to fit in new process

- Suppose there are 3 free memory regions of sizes 30k, 40k, 20k
- The newly created process (during fork() + exec()) needs 15k
- Which region to allocate to it ?

Strategies for finding a free chunk

- **6k, 17k, 16k, 40k holes . Need 15k.**
- **Best fit: Find the smallest hole, larger than process. Ans: 16k**
- **Worst fit: Find the largest hole. Ans: 40k**
- **First fit: Find the “first” hole larger than the process. Ans: 17k**

Problem : External fragmentation

- **Free chunks: 30k, 40k, 20k**
- **The newly created process (during fork() + exec()) needs 50k**
- **Total free memory: 30+40+20 = 90k**
  - **But can’t allocate 50k !**

Solution to external fragmentation

- **Compaction !**
- **OS moves the process chunks in memory to make available continous memory region**
  - **Then it must update the memory management information in PCB (e.g. base of the process) of each process**
- **Time consuming**
- **Possible only if the relocation+limit scheme of MMU is available**

Another solution to external fragmentation: Fixed size partitions

- **Fixed partition scheme**
- **Memory is divided by OS into chunks of equal size: e.g., say, 50k**
  - **If total 1M memory, then 20 such chunks**
- **Allocate one or more chunks to a process, such that the total size is >= the size of the process**
  - **E.g.if request is 50k, allocate 1 chunk**
  - **If request is 40k, still allocate 1 chunk**
  - **If request is 60k, then allocate 2 chunks**
- **Leads to internal fragmentation**
  - **space wasted in the case of 40k or 60k requests above**

Fixed partition scheme

- **OS needs to keep track of**
  - **Which partition is free and which is used by which process**
  - **Free partitions can simply be tracked using a bitmap or a list of numbers**
  - **Each process's PCB will contain list of partitions allocated to it**

Solution to internal fragmentation

- **Reduce the size of the fixed sized partition**
- **How small then ?**
  - **Smaller partitions mean more overhead for the operating system in allocating deallocating**

Paging

An extended version of fixed size partitions

- **Partition = page**
  - Process = logically continuous sequence of bytes, divided in ‘page’ sizes
  - Memory divided into equally sized page ‘frames’
- **Important distinction**
  - Process need not be continous in RAM
  - Different page sized chunks of process can go in any page frame
  - Page table to map pages into frames

Logical address seen as

Paging hardware

Figure 9.8 Paging hardware.

MMU’s job

To translate a logical address generated by the CPU to a physical address:

1. Extract the page number p and use it as an index into the page table.

(Page table location is stored in a hardware register

Also stored in PCB of the process, so that it can be used to load the hardware register on a context switch)

2. Extract the corresponding frame number f from the page table.

3. Replace the page number p in the logical address with the frame number f .

Job of OS

- **Allocate a page table for the process, at time of fork()/exec()**
  - **Allocate frames to process**
  - **Fill in page table entries**
- **In PCB of each process, maintain**
  - **Page table location (address)**
  - **List of pages frames allocated to this process**
- **During context switch of the process, load the PTBR using the PCB**

Job of OS

- **Maintain a list of all page frames**
  - **Allocated frames**
  - **Free Frames (called frame table)**
  - **Can be done using simple linked list**
  - **Innovative data structures can also be used to maintain free and allocated frames list (e.g. xv6 code)**
  -

Figure 9.11 Free frames (a) before allocation and (b) after allocation.

Disadvantage of Paging

- **Each memory access results in two memory accesses!**
  - **One for page table, and one for the actual memory location !**
  - **Done as part of execution of instruction in hardware (not by OS!)**
  - **Slow down by 50%**

Speeding up paging

- Translation Lookaside Buffer (TLB)
- Part of CPU hardware
- A cache of Page table entries
- Searched in parallel for a page number

Speedup due to TLB

- **Hit ratio**
- **Effective memory access time**

  = Hit ratio * 1 memory access time + miss ratio * 2 memory access time

- **Example: memory access time 10ns, hit ratio = 0.8, then**

  effective access time = 0.80 × 10 + 0.20 × 20

  = 12 nanoseconds

Figure 9.13 Valid (v) or invalid (i) bit in a page table.

Memory protection with paging

X86 PDE and PTE

PDE

PTE

P	Present
W	Writable
U	User
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use

Figure 9.14 Sharing of standard C library in a paging environment.

Shared pages (e.g. library) with paging

Paging: problem of large PT

- **64 bit address**
- **Suppose 20 bit offset**
  - **That means 2^20 = 1 MB pages**
  - **44 bit page number: 2^44 that is trillion sized page table!**
  - **Can’t have that big continuous page table!**

Paging: problem of large PT

- 32 bit address
- Suppose 12 bit offset
  - That means 2^12 = 4 KB pages
  - 20 bit page number: 2^20 that is a million entries
  - Can’t always have that big continuous page table as well, for each process!

Hierarchical paging

Figure 9.15 A two-level page-table scheme.

outer page	inner page	offset
p_1	p_2	d
42	10	12

More hierarchy

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

Problems with hierarchical paging

- **More number of memory accesses with each level !**
  - **Too slow !**
- **OS data structures also needed in that proportion**

Hashed page table

Figure 9.17 Hashed page table.

Inverted page table

Figure 9.18 Inverted page table.

Normal page table – one per process --> Too much memory consumed

Inverted page table : global table – only one
Needs to store PID in the table entry

Examples of systems using inverted page
tables include the 64-bit Ultra SPARC and Power PC

virtual address
consists of a triple:
<process-id, page-number, offset>

Case Study: Oracle SPARC Solaris

- 64 bit SPARC processor , 64 bit Solaris OS
- Uses Hashed page tables
  - one for the kernel and one for all user processes.
  - Each hash-table entry : base + span (#pages)
    - Reduces number of entries required

Case Study: Oracle SPARC Solaris

- **Cashing levels: TLB (on CPU), TSB(in Memory), Page Tables (in Memory)**
  - **CPU implements a TLB that holds translation table entries (TTE s) for fast hardware lookups.**
  - **A cache of these TTEs resides in a in-memory translation storage buffer (TSB), which includes an entry per recently accessed page**
  - **When a virtual address reference occurs, the hardware searches the TLB for a translation.**
  - **If none is found, the hardware walks through the in memory TSB looking for the TTE that corresponds to the virtual address that caused the lookup**

Case Study: Oracle SPARC Solaris

- If a match is found in the TSB , the CPU copies the TSB entry into the TLB , and the memory translation completes.
- If no match is found in the TSB , the kernel is interrupted to search the hash table.
- The kernel then creates a TTE from the appropriate hash table and stores it in the TSB for automatic loading into the TLB by the CPU memory-management unit.
- Finally, the interrupt handler returns control to the MMU , which completes the address translation and retrieves the requested byte or word from main memory.

Swapping

Figure 9.19 Standard swapping of two processes using a disk as a backing store.

Swapping

- **Standard swapping**
  - Entire process swapped in or swapped out
  - With continous memory management
- **Swapping with paging**
  - Some pages are “paged out” and some “paged in”
  - Term “paging” refers to paging with swapping now

Figure 9.20 Swapping with paging.

Words of caution about ‘paging’

- **Not as simple as it sounds when it comes to implementation**
  - **Writing OS code for this is challenging**

Mark True, the actions done as part of code of swtch() in swtch.S, in xv6

Restore new callee saved registers from kernel stack of new context: True

Save old callee saved registers on kernel stack of old context: True

Save old callee saved registers on user stack of old context: False

Switch from old process context to new process context: False

Switch from one stack (old) to another(new): True

Restore new callee saved registers from user stack of new context: False

Jump to code in new context: False

For each function/code-point, select the status of segmentation setup in xv6

: bootmain() → gdt setup with 3 entries, at start32 symbol of bootasm.S,

kvmalloc() in main() → gdt setup with 3 entries, at start32 symbol of bootasm.S,

after startothers() in main() → gdt setup with 5 entries (0 to 4) on all processors,

after seginit() in main() → gdt setup with 5 entries (0 to 4) on one processor,

bootasm.S → gdt setup with 3 entries, at start32 symbol of bootasm.S,

entry.S → gdt setup with 3 entries, at start32 symbol of bootasm.S

Compare paging with demand paging and select the correct statements.

The correct answers are: Demand paging requires additional hardware support, compared to paging., Both demand paging and paging support shared memory pages., With demand paging, it's possible to have user programs bigger than physical memory., Demand paging always increases effective memory access time., Paging requires some hardware support in CPU, Calculations of number of bits for page number and offset are same in paging and demand paging., The meaning of valid-invalid bit in page table is different in paging and demandpaging.

The OS may terminate the process while handling the interrupt of memory violation: True

The hardware detects any memory access beyond the limit value and raises an interrupt: True

The hardware may terminate the process while handling the interrupt of memory violation: False

The OS sets up the relocation and limit registers when the process is scheduled: True

The compiler generates machine code assuming continuous memory address space for process, and calculating appropriate sizes for code, and data;: True

The process sets up it's own relocation and limit registers when the process is scheduled: False

The OS detects any memory access beyond the limit value and raises an interrupt: False

The compiler generates machine code assuming appropriately sized semgments for code, data and stack.: False

---

Consider the following list of free chunks, in continuous memory management: 10k, 25k, 12k, 7k, 9k, 13k Suppose there is a request for chunk of size 9k, then the free chunk selected under each of the following schemes will be Best fit: 9k First fit: 10k Worst fit: 25k

---

Select all the correct statements about MMU and it's functionality Select one or more:

The correct answers are: MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

---

The correct answers are: MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

The correct answer is: 107

Select the compiler's view of the process's address space, for each of the following MMU schemes: (Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

The correct answer is: Segmentation, then paging → many continuous chunks of variable size, Relocation + Limit → one continuous chunk, Segmentation → many continuous chunks of variable size, Paging → one continuous chunk

---

Suppose the memory access time is 180ns and TLB hit ratio is 0.3, then effective memory access time is (in nanoseconds); The correct answer is: 306.00

In xv6, The struct context is given as struct context { uint edi; uint esi; uint ebx; uint ebp; uint eip; }; Select all the reasons that explain why only these 5 registers are included in the struct context.

The correct answers are: The segment registers are same across all contexts, hence they need not be saved, eax, ecx, edx are caller save, hence no need to save, esp is not saved in context, because context{} is on stack and it's address is always argument to swtch()

Arrange the following events in order, in page fault handling:

The correct answer is: Disk interrupt wakes up the process → 7, The reference bit is found to be invalid by MMU → 1, OS makes available an empty frame → 4, Restart the instruction that caused the page fault → 9, A hardware interrupt is issued → 2, OS schedules a disk read for the page (from backing store) → 5, Process is kept in wait state → 6, Page tables are updated for the process → 8, Operating system decides that the page was not in memory → 3

Suppose a kernel uses a buddy allocator. The smallest chunk that can be allocated is of size 32 bytes. One bit is used to track each such chunk, where 1 means allocated and 0 means free. The chunk looks like this as of now: 00001010

The correct answer is: 11111010

The complete range of virtual addresses (after main() in main.c is over), from which the free pages used by kalloc() and kfree() is derived,are: The correct answer is: end, P2V(PHYSTOP)

Match the pair

the correct answer is: Hashed page table → Linear search on collsion done by OS (e.g. SPARC Solaris) typically, Inverted Page table → Linear/Parallel search using page number in page table, Hierarchical Paging → More memory access time per hierarchy

---

After virtual memory is implemented (select T/F for each of the following)One Program's size can be larger than physical memory size

Code need not be completely in memory: True Cumulative size of all programs can be larger than physical memory size: True Virtual access to memory is granted: False Logical address space could be larger than physical address space: True Virtual addresses are available: False Relatively less I/O may be possible during process execution: True One Program's size can be larger than physical memory size: True

---

W.r.t. Memory management in xv6, xv6 uses physical memory upto 224 MB onlyMark statements True or False

The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context: True

The stack allocated in entry.S is used as stack for scheduler's context for first processor: True

The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir: True

The free page-frame are created out of nearly 222 MB: True

The kernel code and data take up less than 2 MB space: True

The switchkvm() call in scheduler() changes CR3 to use page directory of new process: False

The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context: False

PHYSTOP can be increased to some extent, simply by editing memlayout.h: True

xv6 uses physical memory upto 224 MB only: True

The process's address space gets mapped on frames, obtained from ~2MB:224MB range: True

The kernel's page table given by kpgdir variable is used as stack for scheduler's context: False

Consider the reference string 6 4 2 0 1 2 6 9 2 0 5 If the number of page frames is 3, then total number of page faults (including initial), using LRU replacement is: Answer: #6# 6,4# 6,4,2 # 0,4,2#0,1,2#6,1,2#6,9,2#0,9,2#0,5,2 The correct answer is: 9

Consider the image given below, which explains how paging works.

The PTBR is present in the CPU as a register: True The page table is indexed using frame number: False The page table is indexed using page number: True The locating of the page table using PTBR also involves paging translation: False Size of page table is always determined by the size of RAM: False The page table is itself present in Physical memory: True Maximum Size of page table is determined by number of bits used for page number: True The physical address may not be of the same size (in bits) as the logical address: True

---

Given below is shared memory code with two processes sharing a memory segment. The first process sends a user input string to second process. The second capitalizes the string. Then the first process prints the capitalized version. Fill in the blanks to complete the code. // First process #define SHMSZ 27 int main() { char c; int shmid; key_t key; char *shm, *s, string[128]; key = 5679; if ((shmid = shmget (key, SHMSZ, IPC_CREAT | 0666)) < 0) { perror("shmget"); exit(1); } if ((shm = shmat (shmid, NULL, 0)) == (char *) -1) { perror("shmat"); exit(1); } s = shm; *s = '$'; scanf("%s", string); strcpy(s + 1, string); *s = ' @ '; //note the quotes while(*s != ' $ ') sleep(1); printf("%s\n", s + 1); exit(0); } //Second process

#define SHMSZ 27 int main() { int shmid; key_t key; char *shm, *s; int i; char string[128]; key = 5679

if ((shmid = shmget(key, SHMSZ, 0666)) < 0) { perror("shmget"); exit(1); } if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) { perror("shmat"); exit(1); } s = shm ; while(*s != '@') sleep(1); for(i = 0; i < strlen(s + 1); i++) s[i + 1] = toupper(s[i + 1]); *s = '$'; exit(0); }

Map the functionality/use with function/variable in xv6 code.

You have correctly selected 3. The correct answer is: return a free page, if available; 0, otherwise → kalloc(), Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed → mappages(), Array listing the kernel memory mappings, to be used by setupkvm() → kmap[], Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices → setupkvm(), Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary → walkpgdir(), Setup kernel part of a page table, and switch to that page table → kvmalloc()

Order events in xv6 timer interrupt code (Transition from process P1 to P2's code.)

The correct answer is: P2 is selected and marked RUNNING → 12, Change of stack from user stack to kernel stack of P1 → 3, Timer interrupt occurs → 2, alltraps() will call iret → 18, change to context of P2, P2's kernel stack in use now → 13, P2's trap() will return to alltraps → 17, jump in vector.S → 4, P2 will return from sched() in yield() → 15, yield() is called → 8, trap() is called → 7, Process P2 is executing →

14, P1 is marked as RUNNABLE → 9, P2's yield() will return in trap() → 16, Process P1 is executing → 1, sched() is called, → 10, change to context of the scheduler, scheduler's stack in use now → 11, jump to alltraps → 5, Trapframe is built on kernel stack of P1 →6

Given that the memory access time is 200 ns, probability of a page fault is 0.7 and page fault handling time is 8 ms, The effective memory access time in nanoseconds is: The correct answer is: 5600060.00

Select the state that is not possible after the given state, for a process: New: Ready : Running: : Waiting: Running Waiting None of these Running

---

Select the correct statements about sched() and scheduler() in xv6 code

You have correctly selected 5. The correct answers are: sched() and scheduler() are co-routines, When either sched() or scheduler() is called, it does not return immediately to caller, When either sched() or scheduler() is called, it results in a context switch, sched() switches to the scheduler's context, scheduler() switches to the selected process's context, After call to swtch() in scheduler(), the control moves to code in sched(), After call to swtch() in sched(), the control moves to code in scheduler(), Each call to sched() or scheduler() involves change of one stack inside swtch()

The data structure used in kalloc() and kfree() in xv6 is

Your answer is correct. The correct answer is: Singly linked NULL terminated list

Virtual Memory

Introduction

- Virtual memory != Virtual address

  Virtual address is address issued by CPU’s execution unit, later converted by MMU to physical address

  Virtual memory is a memory management technique employed by OS (with hardware support, of course)

Unused parts of program

```
int a[4096][4096]
int f(int m[][4096]) {
    int i, j;
    for(i = 0; i < 1024; i++)
        m[0][i] = 200;
}
int main() {
    int i, j;
    for(i = 0; i < 1024; i++)
        a[1][i] = 200;
    if(random() == 10)
          f(a);
}
```

All parts of array a[] not accessed

Function f() may not be called

Some problems with schemes discussed so far

- Code needs to be in memory to execute, But entire program rarely used
  - Error code, unusual routines, large data structures are rarely used
- So, entire program code, data not needed at same time
- So, consider ability to execute partially-loaded program
  - One Program no longer constrained by limits of physical memory
  - One Program and collection of programs could be larger than physical memory

What is virtual memory?

- Virtual memory – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory Larger Than Physical Memory

Virtual Address space

Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc

Shared pages Using Virtual Memory

System libraries shared via mapping into virtual address space

Shared memory by mapping same page-frames into page tables of involved processes

Pages can be shared during `fork()`, speeding process creation (more later)

Demand Paging

Demand Paging

- Load a “page” to memory when it’s neded (on demand)

  Less I/O needed, no unnecessary I/O

  Less memory needed

  Faster response

  More users

-

Demand Paging

- Options:

  Load entire process at load time : achieves little

  Load some pages at load time: good

  Load no pages at load time: pure demand paging

New meaning for valid/invalid bits in page table

Frame #	valid-invalid bit
	v
	v
	v
	v
	i
….	
	i
	i

- With each page table entry a valid–invalid bit is associated

  v: in-memory – memory resident

  i : not-in-memory or illegal

- During address translation, if valid–invalid bit in page table entry is I : **raises trap called page fault**

Page

Table

With

Some pages

Not

In memory

Page fault

Page fault

- Page fault is a hardware interrupt
- It occurs when the page table entry corresponding to current memory access is “i”
- All actions that a kernel takes on a hardware interrupt are taken!

  Change of stack to kernel stack

  Saving the context of process

  Switching to kernel code

On a Page fault

1) Operating system looks at another data structure (table), most likely in PCB itself, to decide:

   If it's Invalid reference -> abort the process (segfault)

   Just not in memory -> Need to get the page in memory

2) Get empty frame (this may be complicated!)

3) Swap page into frame via scheduled disk/IO operation

4) Reset tables to indicate page now in memory.

5) Set validation bit = v

6) Restart the instruction that caused the page fault

Additional problems

- Extreme case – start process with *no* pages in memory

  OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault

  And for every other process pages on first access

  Pure demand paging

- Actually, a given instruction could access multiple pages -> multiple page faults

  Pain decreased because of **locality of reference**

- Hardware support needed for demand paging

  Page table with valid / invalid bit

  Secondary memory (swap device with **swap space**)

  Instruction restart

Instruction restart

- A critical Problem
- Consider an instruction that could access several different locations

```
movarray 0x100, 0x200, 20
```

copy 20 bytes from address 0x100 to address 0x200

```
movarray 0x100, 0x110, 20
```

what to do in this case?

Handling A Page Fault

Page fault handling

1) Trap to the operating system
2) Default trap handling():

    Save the process registers and process state

    Determine that the interrupt was a page fault. Run page fault handler.

3) Page fault handler(): Check that the page reference was legal and determine the location of the page on the disk. If illegal, terminate process.
4) Find a free frame. Issue a read from the disk to a free frame:

    Process waits in a queue for disk read. Meanwhile many processes may get scheduled.

    Disk DMA hardware transfers data to the free frame and raises interrupt in end

Page fault handling

6) (as said on last slide) While waiting, allocate the CPU to some other process
7) (as said on last slide) Receive an interrupt from the disk I/O subsystem (I/O completed)
8) Default interrupt handling():

    Save the registers and process state for the other user

    Determine that the interrupt was from the disk

9) Disk interrupt handler():

    Figure out that the interrupt was for our waiting process

    Make the process runnable

10) Wait for the CPU to be allocated to this process again

    Kernel restores the page table of the process, marks entry as “v”

    Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of demand paging

Page Fault Rate 0 <= p <= 1

if p = 0 no page faults

if p = 1, every reference is a fault

Effective (memory) Access Time (EAT)

EAT = (1 – p) * memory access time +

p * (page fault overhead // Kernel code execution time

+ swap page out // time to write an occupied frame to disk

+ swap page in // time to read data from disk into free frame

+ restart overhead) // time to reset process context, restart it

Performance of demand paging

Memory access time = 200 nanoseconds

Average page-fault service time = 8 milliseconds
EAT = (1 – p) x 200 + p (8 milliseconds)

= (1 – p x 200 + p x 8,000,000

= 200 + p x 7,999,800

If one access out of 1,000 causes a page fault, then

EAT = 8.2 microseconds.

This is a slowdown by a factor of 40!!

If want performance degradation < 10 percent

220 > 200 + 7,999,800 x p
20 > 7,999,800 x p

p < .0000025

< one page fault in every 400,000 memory accesses

An optimization: Copy on write

The problem with fork() and exec(). Consider the case of a shell

```
scanf("%s", cmd);
if(strcmp(cmd, "exit") == 0)
    return 0;
pid = fork(); // A->B
if(pid == 0) {
    ret = execl(cmd, cmd, NULL);
    if(ret == -1) {
        perror("execution failed");
        exit(errno);
    }
} else {
    wait(0);
}
```

- During fork()
  - Pages of parent were duplicated
  - Equal amount of page frames were allocated
  - Page table for child differed from parent (as it has another set of frames)
- In exec()
  - The page frames of child were taken away and new frames were allocated
  - Child’s page table was rebuilt!
- Waste of time during fork() if the exec() was to be called immediately

An optimization: Copy on write

- Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call exec()
  - Very efficient

Before Process 1 Modifies Page C

After Process 1 Modifies Page C

Challenges and improvements in implementation

- Choice of backing store

  For stack, heap pages: on swap partition

  For code, shared library? : swap partition or the actual executable file on the file-system?

  If the choice is file-system for code, then the page-fault handler needs to call functions related to file-system

- Is the page table itself pagable?

  If no, good

  If Yes, then there can be page faults in accessing the page tables themselves! More complicated!

- Is the kernel code pagable?

  If no, good

  If yes, life is very complicated ! Page fault in running kernel code, interrupt handlers, system calls, etc.

Page replacement

Review

- Concept of virtual memory, demand paging.
- Page fault
- Performance degradation due to page fault: Need to reduce #page faults to a minimum
- Page fault handling process, broad steps: (1) Trap (2) Locate on disk (3) find free frame (4) schedule disk I/O (5) update page table (6) resume
- More on (3) today

List of free frames

- Kernel needs to maintain a list of free frames
- At the time of loading the kernel, the list is created
- Frames are used for allocating memory to a process

  But may also be used for managing kernel’s own data structures also
- More processes --> more demand for frames

What if no free frame found on page fault?

- Page frames in use depends on “Degree of multiprogramming”

  More multiprogramming -> overallocation of frames

  Also in demand from the kernel, I/O buffers, etc

  How much to allocate to each process? How many processes to allow?

- Page replacement – find some page(frame) in memory, but not really in use, page it out

  Questions : terminate process? Page out process? replace the page?

  For performance, need an algorithm which will result in minimum number of page faults

- Bad choices may result in same page being brought into memory several times

Need for Page Replacement

Figure 10.9 Need for page replacement.

Page replacement

- Strategies for performance

  Prevent over-allocation of memory by modifying page-fault service routine to include page replacement

  Use modify (dirty) bit in page table. To reduce overhead of page transfers – only modified pages are written to disk. If page is not modified, just reuse it (a copy is already there in backing store)

Basic Page replacement

1) Find the location of the desired page on disk
2) Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a victim frame & write victim frame to disk if dirty
3) Bring the desired page into the free frame; update the page table of process and global frame table/list
4) Continue the process by restarting the instruction that caused the trap

- Note now potentially 2 page transfers for page fault – increasing EAT

Page Replacement

Two problems to solve

- Frame-allocation algorithm determines
  - How many frames to give each process
  - Which frames to replace
- Page-replacement algorithm
  - Want lowest page-fault rate on both first access and re-access

Evaluating algorithm: Reference string

- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
- In all our examples, the reference string is 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

An Expectation

More page Frames Means less faults

FIFO Algorithm

reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
	0	0	0		3	3	3	2	2	2			1	1			1	0	0
		1	1		1	0	0	0	3	3			3	2			2	2	1

page frames

FIFO Algorithm

1	7	2	4	0	7
2	0	3	2	1	0
3	1	0	3	2	1

15 page faults

- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (3 pages can be in memory at a time per process)
- **Belady's Anomaly**
- Adding more frames can cause more page faults!
- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5

FIFO Algorithm: Balady's anamoly

Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal #replacements for the example on the next slide
- How do you know this?
  - Can’t read the future
- Used for measuring how well your algorithm performs

Optimal page replacement

reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1

7	7	7	2		2		2			2			2				7		
	0	0	0		0		4			0			0				0		
		1	1		3		3			3			1				1		

page frames

Least Recently Used: an approximation of optimal

Use past knowledge rather than future
Replace page that has not been used in the most amount of time
Associate time of last use with each page
12 faults – better than FIFO but worse than OPT
Generally good algorithm and frequently used
But how to implement?

LRU: Counter implementation

- **Counter implementation**
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed

LRU: Stack implementation

- Keep a stack of page numbers in a double link form:
- Page referenced: move it to the top
  - requires 6 pointers to be changed and
  - each update more expensive
  - But no need of a search for replacement

Use Of A Stack to Record The Most Recent Page References

Stack algorithms

- An algorithm for which it can be shown that the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with n + 1 frames
- Do not suffer from Balady’s anamoly
- For example: Optimal, LRU

LRU: Approximation algorithms

- LRU needs special hardware and still slow
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
  - We do not know the order, however

LRU: Approximation algorithms

- **Second-chance algorithm**
  - FIFO + hardware-provided reference bit. If bit is 0 select, if bit is 1, then set it to 0 and move to next one.
- **An implementation of second-chance: Clock replacement**
  - If page to be replaced has
  - Reference bit = 0 -> replace it
  - reference bit = 1 then:
    - set reference bit 0, leave page in memory
    - replace next page, subject to same rules

Second-Chance (clock) Page-Replacement Algorithm

(a)

(b)

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common
- **LFU Algorithm**: replaces page with smallest count
- **MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Page buffering algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

Major and Minor page faults

- Most modern OS refer to these two types
- Major fault
  - Fault + page not in memory
- Minor fault
  - Fault, but page is in memory
  - For example shared memory pages; second instance of fork(), page already on free-frame list,
- On Linux run
  - `$ ps -eo min_flt,maj_flt,cmd`

Special rules for special applications

- All of earlier algorithms have OS guessing about future page access
- But some applications have better knowledge – e.g. databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can given direct access to the disk, getting out of the way of the applications
  - Raw disk mode
- Bypasses buffering, locking, etc

Allocation of frames

- Each process needs *minimum* number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- Maximum of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

Fixed allocation of frames

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames

  Keep some as free frame buffer pool

- Proportional allocation – Allocate according to the size of process

  Dynamic as degree of multiprogramming, process sizes change

s_i = size of process p_i

$S = \sum s_i$

m = total number of frames

a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 64$

$s_1 = 10$

$s_2 = 127$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority Allocation of frames

- Use a proportional allocation scheme using priorities rather than size
- If process Pi generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

Global Vs Local allocation

- **Global replacement –** process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common
- **Local replacement –** each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

Virtual Memory – Remaining topics

Agenda

- Problem of Thrashing and possible solutions
- Mmap(), Memory mapped files
- Kernel Memory Management
- Other Considerations

Thrashing

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
- This leads to:
  - Low CPU utilization
  - Operating system thinking that it needs to increase the degree of multiprogramming
  - Another process added to the system
- Thrashing : a process is busy swapping pages in and out

Thrashing

Demand paging and thrashing

- Why does demand paging work?
  - Locality model
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?
  - size of locality > total memory size
  - Limit effects by using local or priority page replacement

Locality In A Memory-Reference Pattern

Working set model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  - Example: 10,000 instructions
- Working Set Size, WSS_i (working set of Process P_i) =
  - total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program

Working set model

- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - Approximation of locality
- if $D > m$ (total available frames) $\Rightarrow$ Thrashing
- Policy if $D > m$, then suspend or swap out one of the processes

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000

  Timer interrupts after every 5000 time units

  Keep in memory 2 bits for each page

  Whenever a timer interrupts copy (to memory) and sets the values of all reference bits to 0

  If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

Page fault frequency

- More direct approach than WSS
- Establish “acceptable” page-fault frequency rate and use local replacement policy

  If actual rate too low, process loses frame

  If actual rate too high, process gains frame

Working Sets and Page Fault Rates

Memory Mapped Files

Memory-Mapped Files

- First, let’s see a demo of using mmap()

Memory-Mapped Files

Use MAP_SHARED flag

Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory
- A file is initially read using demand paging
  - A page-sized portion of the file is read from the file system into a physical page
  - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies and speeds file access by driving file I/O through memory rather than read() and write() system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- But when does written data make it to disk?
  - Periodically and / or at file close() time
  - For example, when the pager scans for dirty pages

Memory-Mapped Files

- Some OSes uses memory mapped files for standard I/O
- Process can explicitly request memory mapping a file via mmap() system call
    - Now file mapped into process address space
- For standard I/O (open(), read(), write(), close()), mmap anyway
    - But map file into kernel address space
    - Process still does read() and write()
        - Copies data to and from kernel space and user space
    - Uses efficient memory management subsystem
        - Avoids needing separate subsystem
- COW can be used for read/write non-shared pages
- Memory mapped files can be used for shared memory (although again via separate system calls)

Allocating Kernel Memory

Allocating kernel memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
  - I.e. for device I/O
-

Buddy Allocator

Buddy Allocator

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using power-of-2 allocator
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
  - Continue until appropriate sized chunk available

Buddy Allocator

- Continue until appropriate sized chunk available
- For example, assume 256KB chunk available, kernel requests 21KB

  Split into AL and Ar of 128KB each

  One further divided into BL and BR of 64KB

  One further into CL and CR of 32KB each – one used to satisfy request
- Advantage – quickly coalesce unused chunks into larger chunk
- Disadvantage - fragmentation

Slab Allocator

Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

Other considerations

Other Considerations -- Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume s pages are prepaged and α of the pages is used

  Is cost of $s * \alpha$ save pages faults > or < than the cost of prepaging

  $s * (1- \alpha)$ unnecessary pages?

  α near zero –––> prepaging loses

Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - **Resolution**
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time

TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB
  Otherwise there is a high degree of page faults
- Increase the Page Size
  This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
  This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

Program Structure

- Program structure
- `Int[128,128] data;`
- Each row is stored in one page
- Program 1

```
for (j = 0; j <128; j++)
     for (i = 0; i < 128; i++)
          data[i,j] = 0;
```

128 x 128 = 16,384 page faults

- Program 2

```
for (i = 0; i < 128; i++)
     for (j = 0; j < 128; j++)
          data[i,j] = 0;
```

128 page faults

I/O Interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm

Threads, Signals

Abhijit A M
abhijit.comp@coep.ac.in

Threads

- thread — a fundamental unit of CPU utilization
  - A separate control flow within a program
  - set of instructions that execute “concurrently” with other parts of the code
  - Note the difference: Concurrency: progress at the same time, Parallel: execution at the same time
- Threads run within application
  - An application can be divided into multiple parts
  - Each part may be written to execute as a threads
- Let’s see an example

Threads

- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight, due to the very nature of threads
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Single vs Mulththreaded process

A mulththreaded server

Benefits of threads

- Responsiveness
- Resource Sharing
- Economy
- Scalability

Single vs Multicore systems

Single core : Concurrency possible

Multicore : parallel execution possible

Multicore programming

- Multicore systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
  -

User vs Kernel Threads

- User Threads: Thread management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads
- Kernel Threads: Supported by the Kernel
- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

User threads vs Kernel Threads

- **User threads**
  - **User level library provides a "typedef" called threads**
  - **The scheduling of threads needs to be implemented in the user level library**
  - **Need some type of timer handling functionality at user level execution of CPU**
    - **OS needs to provide system calls for this**
  - **Kernel does not know that there are threads!**

- **Kernel Threads**
  - **Kernel implements concept of threads**
  - **Still, there may be a user level library, that maps kernel concept of threads to "user concept" since applications link with user level libraries**
  - **Kernel does scheduling!**

Mulththreading models

- How to map user threads to kernel threads?
  - Many-to-One
  - One-to-One
  - Many-to-Many
- What if there are no kernel threads?
  - Then only “one” process. Hence many-one mapping possible, to be done by user level thread library
  - Is One-One possible?

Many-One Model

- **Many user-level threads mapped to single kernel thread**
- **Examples:**
  - **Solaris Green Threads**
  - **GNU Portable Threads**

One-One Model

- **Each user-level thread maps to kernel thread**
- **Examples**
  - **Windows NT/XP/2000**
  - **Linux**
  - **Solaris 9 and later**

Many-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the ThreadFiber package

Two Level Model

- Similar to M:M, except that it allows a user thread to be bound to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

Thread Libraries

Thread libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
-

Demo of pthreads code

Demonstration on Linux – see the code, compile and execute it.

Other libraries

- Windows threading API
  - CreateThread(...)
  - WaitForSingleObject(...)
  - CloseHandle(...)
- Java Threads
  - The Threads class
  - The Runnable Interface

Issues with threads

- Semantics of fork() and exec() system calls
  - Does fork() duplicate only the calling thread or all threads?
- Thread cancellation of target thread
  - Terminating a thread before it has finished
  - Two general approaches:
    - Asynchronous cancellation terminates the target thread immediately.
    - Deferred cancellation allows the target thread to periodically check if it should be cancelled.

More on threads

Thread pools

- **Some kernels/libraries can provide system calls to : Create a number of threads in a pool where they await work, assign work/function to a waiting thread**
- **Advantages:**
  - **Usually slightly faster to service a request with an existing thread than create a new thread**
  - **Allows the number of threads in the application(s) to be bound to the size of the pool**

Thread Local Storage (TLS)

- **Thread-specific data, Thread Local Storage (TLS)**
  - Not local, but global kind of data for all functions of a thread, more like "static" data
  - Create Facility needed for data private to thread
  - Allows each thread to have its own copy of data
  - Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
  - gcc compiler provides the storage class keyword thread for declaring TLS data

```
static __thread int threadID;
```

```
int arr[16];
int f() {
    a(); b(); c();
}
int g() {
    x(); y();
}
int main() {
    th_create(...,f,...);
    th_create(...,g,...);
}
//arr is visible to all of them!
//need data for only f,a,b,c
//need data for only g,x,y
```

Scheduler activations for threads

Library

```
--
th_setup(int n) {
    max_threads = n;
    curr_threads = 0;
}
th_create(...., fn,....) {
    if(curr_threads < max_threads)
        create kernel thread;
    schedule fn on one of the kernel
threads;
}
```

application

```
---
f() {
    scanf();
}
g() {
    recv();
}
h() {...}; i() {...}
main()
    th_setup(2);
    th_create(...,f,...);
    th_create(...,g,...);
    th_create(...,h,...);
    th_create(...,i,...);
}
```

Scheduler activations for threads

- **Scheduler Activations**
  - **Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application**
  - **Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library**
  - **This communication allows an application to maintain the correct number kernel threads**

Issues with threads

- **Scheduler Activations: LWP approach**
  - **An intermediate data structure LWP**
  - **appears to be a virtual processor on which the application can schedule a user thread to run.**
  - **Each LWP attached to a kernel thread**
  - **Typically one LWP per blocking call, e.g. 5 file I/Os in one process, then 5 LWPs needed**

Issues with threads

- **Scheduler Activations: LWP approach**
  - **Kernel needs to inform application about events like: a thread is about to block, or wait is over**
  - **This will help application relinquish the LWP or request a new LWP**

Issues with threads

- **The actual upcalls**

Linux threads

- Only threads (called task), no processes!
- Process is a thread that shares many particular resources with the parent thread
- Clone() system call to create a thread

Linux threads

- clone() takes options to determine sharing on process create
- struct task_struct points to process data structures (shared or unique depending on clone options)
- fork() is a wrapper on top of clone()

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Issues in implementing threads project

- How to implement a user land library for threads?
- How to handle 1-1, many-one, many-many implementations?
- Identifying the support required from OS and hardware
- Identifying the libraries that will help in implementation

Issues in implementing threads project

- Understand the clone() system call completely
  - Try out various possible ways of calling it
  - Passing different options
  - Passing a user-land buffer as stack
- How to save and restore context?
  - C: setjmp, longjmp
  - Setcontext, getcontext(), makecontext(), swapcontext() functions
- Sigaction is more powerful than signal
  - Learn SIGALRM handling for timer and scheduler, timer_create() & timer_stop() system calls
- Customized data structure to store threads, and manage thread-lists for scheduling

Signals

Signals

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- Signal handling
  - Synchronous and asynchronous
- A signal handler (a function) is used to process signals
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Then, signal is “handled” by the handler

Signals

- **More about signals**
  - **Different signals are typically identified as different numbers**
  - **Operating systems provide system calls like kill() and signal() to enable processes to deliver and receive signals**
  - **Signal() - is used by a process to specify a “signal handler” – a code that should run on receiving a signal**
  - **Kill() is used by a process to send another process a signal**
  - **There are restrictions on which process can send which signal to other processes**

Demo

- Let’s see a demo of signals with respect to processes
- Let’s see signal.h
  - /usr/include/signal.h
  - /usr/include/asm-generic/signal.h
  - /usr/include/linux/signal.h
  - /usr/include/sys/signal.h
  - /usr/include/x86_64-linux-gnu/asm/signal.h
  - /usr/include/x86_64-linux-gnu/sys/signal.h
- man 7 signal
- Important signals: SIGKILL, SIGUSR1, SIGSEGV, SIGALRM, SIGCLD, SIGINT, SIGPIPE, ...

Signal handling by OS

Process 12323 {

signal(19, abcd);

}

OS: sys_signal {

Note down that process 12323 wants to handle signal number 19 with function abcd

}

Process P1 {

kill (12323, 19) ;

}

OS: sys_kill {

Note down in PCB of process 12323 that signal number 19 is pending for you.

}

When process 12323 is scheduled, at that time the OS will check for pending signals, and invoke the appropriate signal handler for a pending signal.

Threads and Signals

- **Signal handling Options:**
  - **Deliver the signal to the thread to which the signal applies**
  - **Deliver the signal to every thread in the process**
  - **Deliver the signal to certain threads in the process**
  - **Assign a specific thread to receive all signals for the process**

Some numbers and their ‘meaning’

- These numbers occur very frequently in discussion
- 0x 80000000 = 2 GB = KERNBASE
- 0x 100000 = 1 MB = EXTMEM
- 0x 80100000 = 2GB + 1MB = KERNLINK
- 0x E000000 = 224 MB = PHYSTOP
- 0x FE000000 = 3.96 GB = 4064 MB = DEVSPACE
  - 4096 – 4064 = 32 MB left on top

X86 Memory Management

X86 address : protected mode address translation

X86 paging

Figure 8.23 Paging in the IA-32 architecture.

Segmentation + Paging

Selector value is implicit based on address being accessed.
Instruction: CS
Stack Variable: SS
Data: DS
etc.

GDT Entry

31 16			15 0	
Base 0:15			Limit 0:15	
63 56	55 52	51 48	47 40	39 32
Base 24:31	Flags	Limit 16:19	Access Byte	Base 16:23

Page Directory Entry (PDE)
Page Table Entry (PTE)

PDE

PTE

P	Present
W	Writable
U	User
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use

Segment selector

TI Table index (0=GDT, 1=LDT)
RPL Requester privilege level

EFLAGS register

Status flags
- CF Carry flag
- PF Parity flag
- AF Auxiliary carry flag
- ZF Zero flag
- SF Sign flag
- OF Overflow flag

Control flags
- DF Direction flag

System flags
- TF Trap flag
- IF Interrupt enable flag
- IOPL I/O privilege level
- NT Nested task
- RF Resume flag
- VM Virtual-8086 mode
- AC Alignment check
- VIF Virtual intr. flag
- VIP Virtual intr. pending
- ID ID flag

CR0

PE	Protection enabled	ET	Extension type	NW	Not write-through
MP	Monitor coprocessor	NE	Numeric error	CD	Cache disable
EM	Emulation	WP	Write protect	PG	Paging
TS	Task switched	AM	Alignment mask		

PG: Paging enabled or not **WP: Write protecion on/off**
PE: Protection Enabled --> protected mode.

CR2

CR3

31
12 11
5 4 3 2 0
CR3
Page-Directory-Table Base Address
PCD
PWT
PWT Page-level writes transparent
PCT Page-level cache disable

CR4

31
11 10 9 8 7 6 5 4 3 2 1 0
CR4
OSXM
OSFX
PCE
PGE
MCE
PAE
PSE
DE
TSD
PVI
VME
VME Virtual-8086 mode extensions
PVI Protected-mode virtual interrupts
TSD Time stamp disable
DE Debugging extensions
PSE Page size extensions
PAE Physical-address extension
MCE Machine check enable
PGE Page-global enable
PCE Performance counter enable
OSFXSR OS FXSAVE/FXRSTOR support
OSXMM-EXCPT OS unmasked exception support

mmu.h : paging related macros

```
#define PTXSHIFT        12      // offset of PTX in a linear address
#define PDXSHIFT        22      // offset of PDX in a linear address
#define PDX(va)   (((uint)(va) >> PDXSHIFT) & 0x3FF)// page directory index
#define PTX(va)   (((uint)(va) >> PTXSHIFT) & 0x3FF)// page table index
// construct virtual address from indexes and offset
#define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT |
(o)))
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table   | Offset within Page  |
// |      Index     |      Index     |                     |
// +----------------+----------------+---------------------+
//  \--- PDX(va) --/ \--- PTX(va) --/
```

mmu.h : paging related macros

```
// Page directory and page table constants.
#define NPDENTRIES      1024    // # directory
entries per page directory
#define NPTENTRIES      1024    // # PTEs per
page table
#define PGSIZE          4096    // bytes mapped
by a page
#define PGROUNDUP(sz)  (((sz)+PGSIZE-1) &
~(PGSIZE-1))
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
```

mmu.h : paging related macros

```
// Page table/directory entry flags.
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_PS          0x080   // Page Size

// Address in page table or page directory entry
#define PTE_ADDR(pte)   ((uint)(pte) & ~0xFFF) // get
all but last 12 bits
#define PTE_FLAGS(pte)  ((uint)(pte) &  0xFFF) // get
last 12 bits
```

mmu.h : Segmentation related macros

```
// various segment selectors.
#define SEG_KCODE 1  // kernel code
#define SEG_KDATA 2  // kernel data+stack
#define SEG_UCODE 3  // user code
#define SEG_UDATA 4  // user data+stack
#define SEG_TSS   5  // this process's task state
```

mmu.h : Segmentation related macros

```
// various segment selectors.
#define SEG_KCODE 1  // kernel code
#define SEG_KDATA 2  // kernel data+stack
#define SEG_UCODE 3  // user code
#define SEG_UDATA 4  // user data+stack
#define SEG_TSS   5  // this process's task state
#define NSEGS     6
```

mmu.h : Segmentation related macros

```
struct segdesc { // 64 bit in size
  uint lim_15_0 : 16;  // Low bits of segment limit
  uint base_15_0 : 16; // Low bits of segment base address
  uint base_23_16 : 8; // Middle bits of segment base address
  uint type : 4;       // Segment type (see STS_ constants)
  uint s : 1;          // 0 = system, 1 = application
  uint dpl : 2;        // Descriptor Privilege Level
  uint p : 1;          // Present
  uint lim_19_16 : 4;  // High bits of segment limit
  uint avl : 1;        // Unused (available for software use)
  uint rsv1 : 1;       // Reserved
  uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
  uint g : 1;          // Granularity: limit scaled by 4K when set
  uint base_31_24 : 8; // High bits of segment base address
};
```

mmu.h : Segmentation related code

```
// Application segment type bits
#define STA_X       0x8     // Executable segment
#define STA_W       0x2     // Writeable (non-executable segments)
#define STA_R       0x2     // Readable (executable segments)

// System segment type bits
#define STS_T32A    0x9     // Available 32-bit TSS
#define STS_IG32    0xE     // 32-bit Interrupt Gate
#define STS_TG32    0xF     // 32-bit Trap Gate
```

Code from bootasm.S bootmain.c is over!
Kernel is loaded.
Now kernel is going to prepare itself

main() in main.c

- Initializes “free list” of page frames
  - In 2 steps. Why?
- Sets up page table for kernel
- Detects configuration of all processors
- Starts all processors
  - Just like the first processor
- Creates the first process!

- Initializes
  - LAPIC on each processor, IOAPIC
  - Disables PIC
  - “Console” hardware (the standard I/O)
  - Serial Port
  - Interrupt Descriptor Table
  - Buffer Cache
  - Files Table
  - Hard Disk (IDE)

main() in main.c

```
int

main(void) {

  kinit1(end, P2V(4*1024*1024)); // phys page allocator

  kvmalloc();      // kernel page table
```

```
void

kinit1(void *vstart, void *vend) {

  initlock(&kmem.lock, "kmem");

  kmem.use_lock = 0;

  freerange(vstart, vend);

}
```

main() in main.c

```
void

freerange(void *vstart, void
*vend)

{

  char *p;

  p =
(char*)PGROUNDUP((uint)vst
art);

  for(; p + PGSIZE <=
(char*)vend; p += PGSIZE)

    kfree(p);

}
```

```
kfree(char *v) {
  struct run *r;
  if((uint)v % PGSIZE || v <
end || V2P(v) >= PHYSTOP)
    panic("kfree");
  // Fill with junk to catch
dangling refs.
  memset(v, 1, PGSIZE);
  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
  if(kmem.use_lock)
    release(&kmem.lock); }
```

Free List in XV6 Obtained after main() -> kinit1()

Pages obtained Between
end = 801154a8 = 2049 MB to P2V(4MB) = 2052 MB
Remember
Right now Logical = Physical address.

Actually like this in memory

kmem

Seen independently

Back to main()

```
int
main(void) {
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
```

```
// Allocate one page table for the machine for the kernel address
// space for scheduler processes.
void
kvmalloc(void)
{
  kpgdir = setupkvm();
  switchkvm();
}
```

Back to main()

```
int
main(void) {
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
```

```
// Allocate one page table for the machine for the kernel address
// space for scheduler processes.
void
kvmalloc(void)
{
  kpgdir = setupkvm(); // global var kpgdir
  switchkvm();
}
```

```
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;
  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (P2V(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0) {
      freevm(pgdir);
      return 0;
    }
  return pgdir;
}
```

```
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {
 { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
 { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
};
```

kmap[] mappings done in kvmalloc(). This shows segmentwise, entries are done in page directory and page table for corresponding VA-> PA mappings

0x80108000 =data = 2049.3125 MB Is obtained from kernel.sym

Remidner: PDE and PTE entries

PDE

PTE

P	Present
W	Writable
U	User
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use

Before mappages()

What mappages() Does

After kvmalloc() in main()
RAM
Physical Addr
Linear Address
Logical Address = offset
Dir
pg
Offset
4MB
0
CS, SS, etc.
Selector
Base
Limit
Permissions
3
2
1
0
0
4GB
Write
0
4GB
Read, Execute
0
0
0
DS
SS
CS
GDT
Page Table
CR3
kpgdir
Now Linear Address = Logical Address != Physical Address

main()->seginit()

- Re-initialize GDT
- Once and forever now
- Just set 4 entries
  - All spanning 4 GB
  - Differing only in permissions and privilege level

After seginit() in main().
On the processor where we started booting
RAM
Physical Addr
Linear Address
Logical Address = offset
Dir
pg
Offset
4MB
0
CS, SS, etc.
Selector
Base
Limit
Permissions
DPL
4
0
4GB
Write
3
3
0
4GB
Read, Execute
3
2
0
4GB
Write
0
1
0
4GB
Read, Execute
0
0
0
0
0
DS
SS
CS
GDT
Page Table
CR3
kpgdir
Now Linear Address = Logical Address != Physical Address

After seginit()

- While running kernel code, necessary to switch CS, DS, SS to index 1,2,2 in GDT
- While running user code, necessary to switch CS, DS, SS to index 3,4,4 in GDT
- This happens automatically as part of “trap” handling (covered separately)

Memory Management

X86 page table hardware

Layout of process's VA space

Memory Layout of a user process

After exec()

Note the argc, argv on stack

Memory Layout of a user process

The "guard page" is just a mapping in page table. No frame allocated. It's marked as invalid. So if stack grows (due to many function calls), then OS will detect it with an exception

Memory Layout of a user process

On sbrk()
The system call to grow process's address space.
Calls growproc()

growproc()
Allocate a frame, Add an entry in page table at the top (above proc->sz)
//This entry can't go beyond KERNBASE
Calls switchuvm()

Switchuvm()
Ultimately loads CR3, invalidating cache

Free List in XV6

Actually like this in memory

Seen independently

exec()

- **sys_exec()**

  exec(path, argv)

- **exec(parth, argv)**

```
ip = namei(path))
readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf)
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
}
```

exec()

- **exec(parth, argv)**

```
// Allocate two pages at the next page boundary.
 // Make the first inaccessible.  Use the second as the user stack.
 sz = PGROUNDUP(sz);
 if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
// Push argument strings, prepare rest of stack in ustack.
  for(argc = 0; argv[argc]; argc++) {
    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
```

Creation of first process by kernel

Why first process needs ‘special’ treatment?

- Normally process is created using fork()
  - and typically followed by a call to exec()
- Fork will use the PCB of existing process to create a new process
  - as a clone
- The first process has nothing to copy from!
- So it’s PCB needs to “built” by kernel code

Why first process needs ‘special’ treatment?

- XV6 approach
  - Create the process as if it was created by “fork”
  - Ensure that the process starts in a call to “exec”
  - Let “Exec” do the rest of the JOB as expected
  - In this case exec() will call
    - exec(“/init”, NULL);
- See the code of init.c
  - opens console() device for I/O; dups 0 on 1 and 2!
    - Same device file for I/O
  - forks a process and execs (“sh”) on it.
  - Itself keeps waiting for zombie processes

Why first process needs ‘special’ treatment?

- What needs to be done ?
  - Build struct proc by hand
  - How data structures (proc, stack, etc) are hand-crafted so that when kernel returns, the process starts in code of init

Imp Concepts

- A process has two stacks
  - user stack: used when user code is running
  - kernel stack: used when kernel is running on behalf of a process
- Note: there is a third stack also!
  - The kernel stack used by the scheduler itself
  - Not a per process stack

Imp Concepts

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

struct proc diagram: Very imp!

In use only when you are in kernel on a “trap” = interrupt/syscall. “tf” always used. trapret,forkret used during fork()

sz = ELF-code->memsz (includes data, check “ld -N”
+ 2*4096 (for stack)

setupkvm()
does this mapping

These mappings need to be created per process

Memory Layout of a user process After exec()

Note the argc, argv on stack

stack is just one page.

size of text and data is derived from ELF file

main()->userinit()
Creating first process by hand

- Code of the first process
  - initcode.S and init.c
  - init.c is compiled into “/init” file
    - During make !
  - Trick:
    - Use initcode.S to “exec(“/init”)”
    - And let exec() do rest of the job
  - But before you do exec()
    - Process must exist as if it was forked() and running

main()->userinit()
Creating first process by hand

```
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  // Abhijit: obtain proc 'p', with stack initialized
  // and trapframe created and eip set to 'forkret'
  p = allocproc();
// let’s see what allocproc() does
```

First process creation
Let's revisit struct proc

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state. allocated, ready to run, running, wait-
ing for I/O, or exiting.
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process. Process's context
  void *chan;                  // If non-zero, sleeping on chan. More when we discuss sleep, wakeup
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files, used by open(), read(),...
  struct inode *cwd;           // Current directory, changed with "chdir()"
  char name[16];               // Process name (for debugging)
};
```

struct proc diagram

sz = ELF-code->memsz (includes data, check “ld -N”
+ 2*4096 (for stack)

In use only when you are in kernel on a “trap” = interrupt/syscall. “tf” always used. trapret,forkret used during fork()

allocproc()

```
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;
  acquire(&ptable.lock);
  for(p = ptable.proc; p <
&ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;
  release(&ptable.lock);
  return 0;
```

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;

release(&ptable.lock);
```

allocproc() setting up stack

```
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;
// Abhijit KSTCKSIZE = PGSIZE
  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;
  // Set up new context to start executing at
forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;
  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
```

allocproc() setting up stack

```
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;
// Abhijit KSTCKSIZE = PGSIZE
  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;
  // Set up new context to start executing at
forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;
  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
```

allocproc() setting up stack

```
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;
// Abhijit KSTCKSIZE = PGSIZE
  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;
  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;
  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
```

allocproc() setting up stack

```
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;
// Abhijit KSTCKSIZE = PGSIZE
  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;
  // Set up new context to start executing at
forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;
  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
```

allocproc() setting up stack

```
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;
// Abhijit KSTCKSIZE = PGSIZE
  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;
  // Set up new context to start executing at
forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;
  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
```

Next in userinit()

```
initproc = p;
if((p->pgdir = setupkvm()) == 0)
  panic("userinit: out of memory?");
inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
p->sz = PGSIZE;
memset(p->tf, 0, sizeof(*p->tf));
p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
p->tf->es = p->tf->ds;
p->tf->ss = p->tf->ds;
p->tf->eflags = FL_IF;
p->tf->esp = PGSIZE;
p->tf->eip = 0;  // beginning of initcode.S
safestrcpy(p->name, "initcode", sizeof(p->name));
p->cwd = namei("/");
acquire(&ptable.lock);
p->state = RUNNABLE;
release(&ptable.lock);
```

After userinit()

Q: why there is no user stack yet?

main()->mpmain()

```
static void
mpmain(void)
{
  cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
  idtinit();       // load idt register
  xchg(&(mycpu()->started), 1); // tell startothers() we're up
  scheduler();     // start running processes
}
```

- **Load IDT register**
  - Copy from idt[] array into IDTR
- **Call scheduler()**
  - One process has already been made runnable
  - Let’s enter scheduler now

Before reading scheduler(): Note

- The esp is still pointing to the stack which was allocated in entry.S !
  - this is the kernel only stack
  - Not the per process kernel stack.
- CR3 points to kpgdir
- Struct cpu[ ] has been setup up already
  - apicid – in mpinit()
  - segdesc gdt – in seginit()
  - started – in mpmain()
- Fields in cpu[] not yet set
  - context * scheduler --> will be setup in sched()
  - taskstate ts --> large structure, only parts used in switchuvm()
  - ncli, intena --> used while locking
  - proc *proc -> set during scheduler()

scheduler()

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
     sti();
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
     // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
```

scheduler() called first time
Code/stack of "initcode"
proc
cpu *c
p
proc
context
kstack
tf
sz = 4096
name = "initcode"
pgdir
cwd
state = RUNNALBE
Trapframe
CS =3
DS = ES = SS =4
EFLAGS = FL_IF
ESP = 4096
EIP = 0
trapret()
eip=forkret()
ebp
ebx
esi
edi
Page table
Page table
Page table
Toward s Kernel pages
pgdir
kpgdir
CR3
stack
entry.S
esp
Inode for "/"

scheduler()

```
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE)
    continue;

  // Switch to chosen process.  It is the process's job
  // to release ptable.lock and then reacquire it
  // before jumping back to us.
  c->proc = p;
  switchuvm(p);
  p->state = RUNNING;
```

after
switchuvm()
in schedul
Code/stack
of
“initcode”
proc
cpu
*c
p
proc
context
kstack
tf
sz = 4096
name = “initcode”
pgdir
cwd
state = RUNNING
Trapframe
CS =3
DS = ES = SS =4
EFLAGS = FL_IF
ESP = 4096
EIP = 0
trapret()
eip=forkret()
ebp
ebx
esi
edi
Page table
Page table
Kernel
pages
Page table
pgdir
CR3
kpgdir
stack
entry.S
esp
Inode
for “/”

scheduler()

```
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE)
    continue;

  // Switch to chosen process.  It is the process's job
  // to release ptable.lock and then reacquire it
  // before jumping back to us.
  c->proc = p;
  switchuvm(p);
  p->state = RUNNING
  swtch(&(c->scheduler), p->context);
;
```

at call to swtch()
Code/stack of "initcode"
proc
cpu *c
p
proc
context
kstack
tf
sz = 4096
name = "initcode"
pgdir
cwd
state = RUNNING
Trapframe
CS =3
DS = ES = SS =4
EFLAGS = FL_IF
ESP = 4096
EIP = 0
trapret()
eip=forkret()
ebp
ebx
esi
edi
Page table
Page table
Page table
Toward s Kernel pages
pgdir
CR3
kpgdir
addr of p->context
--
addr of c->scheduler
--
ret value in scheduler()
esp
Inode for "/"

swtch

```
swtch:
  #Abhijit: swtch was called through a function call.
  #So %eip was saved on stack already
  movl 4(%esp), %eax    # Abhijit: eax = old
  movl 8(%esp), %edx    # Abhijit: edx = new
```

during swtch()
Code/stack of "initcode"
proc
cpu *c
p
proc
context
kstack
tf
sz = 4096
name = "initcode"
pgdir
cwd
state = RUNNING
Trapframe
CS =3
DS = ES = SS =4
EFLAGS = FL_IF
ESP = 4096
EIP = 0
trapret()
eip=forkret()
ebp
ebx
esi
edi
Page table
Page table
Page table
Toward s Kernel pages
pgdir
CR3
kpgdir
addr of p->context
--
addr of c->scheduler
--
ret value in scheduler()
esp
Inode for "/"
eax = &c->scheduler
edx = &p->context

swtch

```
swtch:
  #Abhijit: swtch was called through a function call.
  #So %eip was saved on stack already
  movl 4(%esp), %eax    # Abhijit: eax = old
  movl 8(%esp), %edx    # Abhijit: edx = new
  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi            # Abhijit: esp = esp + 16
```

during swtch()
Code/stack of "initcode"
proc
cpu *c
p
proc
context
kstack
tf
sz = 4096
name = "initcode"
pgdir
cwd
state = RUNNING
Page table
Page table
Page table
Toward s Kernel pages
pgdir
CR3
kpgdir
addr of p->context
addr of c->scheduler
ret value in scheduler()
ebp,ebx, esi,edi
esp
Inode for "/"
Trapframe
CS =3
DS = ES = SS =4
EFLAGS = FL_IF
ESP = 4096
EIP = 0
trapret()
eip=forkret()
ebp
ebx
esi
edi
eax = &c->scheduler
edx = &p->context

swtch

```
swtch:
  #Abhijit: swtch was called through a function call.
  #So %eip was saved on stack already
  movl 4(%esp), %eax    # Abhijit: eax = old
  movl 8(%esp), %edx    # Abhijit: edx = new
  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi            # Abhijit: esp = esp + 16
  # Switch stacks
  movl %esp, (%eax)     # Abhijit: *old = updated old stack
  movl %edx, %esp       # Abhijit: esp = new
```

during swtch()
Code/stack of "initcode"
p
proc
context
kstack
tf
sz = 4096
name = "initcode"
pgdir
cwd
state = RUNNING
proc
cpu *c
Trapframe
CS =3
DS = ES = SS =4
EFLAGS = FL_IF
ESP = 4096
EIP = 0
trapret()
eip=forkret()
ebp
ebx
esi
edi
Page table
Page table
Page table
Toward s Kernel pages
pgdir
CR3
kpgdir
addr of p->context
addr of c->scheduler
ret value in scheduler()
ebp,ebx, esi,edi
esp
Inode for "/"
c->scheduler
edx = &p->context

swtch

```
swtch:
  #Abhijit: swtch was called through a function call.
  #So %eip was saved on stack already
  movl 4(%esp), %eax    # Abhijit: eax = old
  movl 8(%esp), %edx    # Abhijit: edx = new
  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi            # Abhijit: esp = esp + 16
  # Switch stacks
  movl %esp, (%eax)     # Abhijit: *old = updated old stack
  movl %edx, %esp       # Abhijit: esp = new
  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp     # Abhijit: newesp = newesp - 16, context restored
```

during swtch()
Code/stack of "initcode"
p
proc
context
kstack
tf
sz = 4096
name = "initcode"
pgdir
cwd
state = RUNNING
proc
cpu *c
Trapframe
CS =3
DS = ES = SS =4
EFLAGS = FL_IF
ESP = 4096
EIP = 0
trapret()
eip=forkret()
ebp
ebx
esi
edi
Page table
Page table
Page table
Toward s Kernel pages
pgdir
CR3
kpgdir
addr of p->context
addr of c->scheduler
ret value in scheduler()
ebp,ebx, esi,edi
esp
Inode for "/"
c->scheduler
edx = &p->context

swtch

```
swtch:
  #Abhijit: swtch was called through a function call.
  #So %eip was saved on stack already
  movl 4(%esp), %eax    # Abhijit: eax = old
  movl 8(%esp), %edx    # Abhijit: edx = new
  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi            # Abhijit: esp = esp + 16
  # Switch stacks
  movl %esp, (%eax)     # Abhijit: *old = updated old stack
  movl %edx, %esp       # Abhijit: esp = new
  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp      # Abhijit: newesp = newesp - 16, context restored
  ret            # Abhijit: will pop from esp now -> function where to
return.
```

after "ret" from swtch() just before forkret()
Code/stack of "initcode"
proc
cpu *c
p
proc
context
kstack
tf
sz = 4096
name = "initcode"
pgdir
cwd
state = RUNNING
Trapframe
CS =3
DS = ES = SS =4
EFLAGS = FL_IF
ESP = 4096
EIP = 0
trapret()
Page table
Page table
Page table
Toward s Kernel pages
pgdir
CR3
kpgdir
addr of p->context
addr of c->scheduler
ret value in scheduler()
ebp,ebx, esi,edi
esp
Inode for "/"
c->scheduler
edx = &p->context

After swtch()

- Process is running in forkret()
- c->csheduler has saved the old kernel stack
  - with the context of p, return value in scheduler, ebp, ebx, esi, edi on stack
  - remember {edi, esi, ebx, ebp, ret-value } = context
  - The c->scheduler is pointing to old context
- CR3 is pointing to process pgdir

after forkret()
just before
trapret()
begins
Code/stack
of
"initcode"
proc
cpu
*c
p
proc
context
kstack
tf
sz = 4096
name = "initcode"
pgdir
cwd
state = RUNNING
Trapframe
CS =3
DS = ES = SS =4
EFLAGS = FL_IF
ESP = 4096
EIP = 0
trapret()
Page table
Page table
Page table
Toward
s
Kernel
pages
pgdir
CR3
kpgdir
addr of
p->context
addr of
c->scheduler
ret value in
scheduler()
ebp,ebx,
esi,edi
esp
Inode
for "/"
c->scheduler
edx = &p->context

After iret in trapret

- **The CS, EIP, ESP will be changed**
  - **to values already stored on trapframe**
  - **this is done by iret**
- **Hence after this user code will run**
  - **On user stack!**
- **Hence code of *initcode* will run now**

at the end
of trapret()
eip
proc
cpu
*c
Code/stack
of
“initcode”
p
proc
context
kstack
tf
sz = 4096
name = “initcode”
pgdir
cwd
state = RUNNING
Page table
Page table
Toward
s
Kernel
pages
Page table
pgdir
kpgdir
CR3
esp
addr of
p->context
addr of
c->scheduler
ret value in
scheduler()
ebp,ebx,
esi,edi
Inode
for “/”
c->scheduler
edx = &p->context

initcode

```
# char init[] = "/init\0";
init:
  .string "/init\0"

# char *argv[] = { init, 0 };
.p2align 2
argv:
  .long init
  .long 0
```

```
start:
  pushl $argv
  pushl $init
  pushl $0  // where caller pc would be
  movl $SYS_exec, %eax
  int $T_SYSCALL

# for(;;) exit();
exit:
  movl $SYS_exit, %eax
  int $T_SYSCALL
  jmp exit
```

```
00000000 <start>:
   0:     68 24 00 00 00          push   $0x24
   5:     68 1c 00 00 00          push   $0x1c
   a:     6a 00                    push   $0x0
   c:     b8 07 00 00 00          mov    $0x7,%eax
  11:     cd 40                    int    $0x40

00000013 <exit>:
  13:     b8 02 00 00 00          mov    $0x2,%eax
  18:     cd 40                 int    $0x40
  1a:     eb f7                 jmp    13 <exit>
0000001c <init>:
          ”/init\0”
00000024 <argv>:
      1c 00
      00 00
```

on sys_exec()
+ all traps()
eip
proc
cpu
*c
alltraps():
0x24
0x1c
0
code
p
proc
context
kstack
tf
sz = 4096
name = “initcode”
pgdir
cwd
state = RUNNING
ss =4,esp, eflags
cs = 3, eip
0,64,ds,es,fs,gs,
gen registers,
add of this esp,
ret add in alltraps()
Page table
Page table
Toward
s
Kernel
pages
Page table
pgdir
CR3
kpgdir
addr of
p->context
addr of
c->scheduler
ret value in
scheduler()
ebp,ebx,
esi,edi
esp
Inode
for “/”
c->scheduler
edx = &p->context

Understanding fork() and exec()

First, revising some concepts already learnt then code of fork(), exec()

First process creation
Let's revisit struct proc

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state. allocated, ready to run, running, wait-
ing for I/O, or exiting.
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process. Process's context
  void *chan;                  // If non-zero, sleeping on chan. More when we discuss sleep, wakeup
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files, used by open(), read(),...
  struct inode *cwd;           // Current directory, changed with "chdir()"
  char name[16];               // Process name (for debugging)
};
```

struct proc diagram

sz = ELF-code->memsz (includes data, check “ld -N”
+ 2*4096 (for stack)

In use only when you are in kernel on a “trap” = interrupt/syscall. “tf” always used. trapret,forkret used during fork()

fork()/exec() are syscalls. On every syscall this happens

- Fetch the n'th descriptor from the IDT, where n is the argument of int.
- Check that CPL in %cs is <= DPL, where DPL is the privilege level in the descriptor.
- Save %esp and %ss in CPU-internal registers, but only if the target segment selector's PL < CPL.
  - Switching from user mode to kernel mode. Hence save user code's SS and ESP
- Load %ss and %esp from a task segment descriptor.
  - Stack changes to kernel stack now. TS descriptor is on GDT, index given by TR register. See switchuvm()
- Push %ss. // optional
- Push %esp. // optional (also changes ss,esp using TSS)
- Push %eflags.
- Push %cs.
- Push %eip.
- Clear the IF bit in %eflags, but only on an interrupt.
- Set %cs and %eip to the values in the descriptor.

After “int” ‘s job is done

- IDT was already set, during idtinit()
  - Remember vectors.S – gives jump locations for each interrupt
- “int 64” ->jump to 64th entry in vector table

```
vector64:
  pushl $0
  pushl $64
  jmp alltraps
```

  - So now stack has ss, esp,eflags, cs, eip, 0 (for error code), 64
  - Next run alltraps from trapasm.S

alltraps:

```
# Build trap frame.
pushl %ds
pushl %es
pushl %fs
pushl %gs
pushal // push all gen purpose regs
# Set up data segments.
movw $(SEG_KDATA<<3), %ax
movw %ax, %ds
movw %ax, %es
# Call trap(tf), where tf=%esp
pushl %esp # first arg to trap()
call trap
addl $4, %esp
```

- Now stack contains

  ss, esp,eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi

  - This is the struct trapframe !
  - So the kernel stack now contains the trapframe
  - Trapframe is a part of kernel stcak

trap()

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
  switch(tf->trapno){
  .....
```

- Argument is trapframe
- In alltraps
  - Before “call trap”, there was “push %esp” and stack had the trapframe
  - Remember calling convention --> when a function is called, the stack contains the arguments in reverse order (here only 1 arg)

trap()

- Has a switch
  - switch(tf->trapno)
  - Q: who set this trapno?
- Depending on the type of trap
  - Call interrupt handler
    - Timer
      - wakeup(&ticks)
    - IDE: disk interrupt
      - Ideintr()
    - KBD
      - Kbdintr()
    - COM1
      - Uatrintr()
    - If Timer
      - Call yield() -- calls sched()
    - If process was killed (how is that done?
      - Call exit()!

when trap() returns

- #Back in alltraps

```
  call trap
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

-

- Stack had (trapframe)
  - ss, esp,eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi, esp
- add $4 %esp
  - esp
- popal
  - eax, ecx, edx, ebx, oesp, ebp, esi, edi
- Then gs, fs, es, ds
- add $0x8, %esp
  - 0 (for error code), 64
- iret
  - ss, esp,eflags, cs, eip,

understanding fork()

- **What should fork do?**
  - **Create a copy of the existing process**
  - **child is same as parent, except pid, parent-child relation, return value (pid or 0)**
  - **Please go through every member of struct proc, understand it's meaning to appreciate what fork() should do**
  - **create a struct proc, and**
    - **duplicate pages, page directory, sz, state,trapframe,context, ofile (and files!), cwd, name**
    - **modify: pid, parent, trapframe, state**

understanding fork()

```
int
sys_fork(void)
{
  return fork();
}
```

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }
```

after allocproc()
-- we studied this -- same as creation of first process

understanding fork()

```
// Copy process state from proc.
if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
  kfree(np->kstack);
  np->kstack = 0;
  np->state = UNUSED;
  return -1;
}
np->sz = curproc->sz;
np->parent = curproc;
*np->tf = *curproc->tf;
```

- copy the pages, page tables, page directory
  - no copy on write here!
  - Rewind if operation of copyuvm() fails
- copy size
- set parent of child
- copy trapframe (structure is copied)

```
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;  pte_t *pte;  uint pa, i, flags;
  char *mem;
  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
      panic("copyuvm: page not present");
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto bad;
    memmove(mem, (char*)P2V(pa), PGSIZE);
    if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
      kfree(mem);
      goto bad;
    }
  }
  return d;
bad:
  freevm(d);
  return 0;
}
```

understanding fork()->copyuvm()

- Map kernel pages
- for every page in parent's VM address space
  - allocate a PTE for child
  - set flags
  - copy data
  - map pages in child's page directory/tables

understanding fork()

```
np->tf->eax = 0;
for(i = 0; i < NOFILE; i++)
  if(curproc->ofile[i])
    np->ofile[i] = filedup(curproc->ofile[i]);
np->cwd = idup(curproc->cwd);
safestrcpy(np->name, curproc->name, sizeof(curproc->name));
pid = np->pid;
acquire(&ptable.lock);
np->state = RUNNABLE;
release(&ptable.lock);
```

- set return value of child to 0
  - eax contains return value, it's on TF
- copy each struct file
- copy current working dir inode
- copy name
- set pid of child
- set child "RUNNABLE"

exec() - different prototype

- **int exec(char*, char**);**
  - **usage: to print README and test.txt using “cat”**

```
int main(int argc, char *argv[])
{
   char *cmd = "/cat";
   char *argstr[4] = { "/cat", "README",
"test.txt", 0};
   exec(cmd, argstr);
}
```

note: to really run this code in xv6, you need to make changes to Makefile. First, add this program to UPROGS, then write a file test.txt using Linux, and add ‘test.txt’ to list of files in ‘mkfs’ target in Makefile

sys_exec()

```
int
sys_exec(void)
{
  char *path, *argv[MAXARG];
  int i;
  uint uargv, uarg;
  if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
    return -1;
  }
  memset(argv, 0, sizeof(argv));
  for(i=0;; i++){
    if(i >= NELEM(argv))
      return -1;
    if(fetchint(uargv+4*i, (int*)&uarg) < 0)
      return -1;
    if(uarg == 0){
      argv[i] = 0;
      break;
    }
    if(fetchstr(uarg, &argv[i]) < 0)
      return -1;
  }
  return exec(path, argv);
}
```

- **argstr(n,), argint(n,)**
  - **Fetch the n'th argument from *process stack* using p->tf->esp + offset**
  - **Again: revise calling conventions**
  - **0'th argument: name of executable file**
  - **1st Argument: address of the array of arguments**
    - **store in *uargv***

sys_exec()

```
int sys_exec(void)
{
  char *path, *argv[MAXARG];
  int i;   uint uargv, uarg;
  if(argstr(0, &path) < 0 || argint(1,
(int*)&uargv) < 0){
    return -1;
  }
  memset(argv, 0, sizeof(argv));
  for(i=0;; i++){
    if(i >= NELEM(argv))     return -1;
    if(fetchint(uargv+4*i, (int*)&uarg) < 0)
      return -1;
    if(uarg == 0){
      argv[i] = 0;       break;
    }
    if(fetchstr(uarg, &argv[i]) < 0)
      return -1;
  }
  return exec(path, argv);
}
```

- the local array argv[] (allocated on kernel stack, obviously) set to 0
- fetch every next argument from array of arguments
  - Sets the address of argument in argv[1]
- call exec
  - beware: mistake to assume that this exec() is the exec() called from user code! NO!

What should exec() do?

- Remember, it came from fork()
  - so proc & within it tf, context, kstack, pgdir-tables-pages, all exist.
  - Code, stack pages exist, and mappings exist through proc->pgdir
- Hence
  - read the ELF executable file (argv[0])
  - create a new page dir – create mappings for kernel and user code+data; copy data from ELF to these pages (later discard old pagedir)
  - Copy the argv onto the user stack – so that when new process starts it has it's main(argc, argv[]) built
  - set values of other fields in proc to start program correctly

User stack after call to exec() is over

normally data on stack on fn call: ret value, first arg, second arg, ...

main(int argc, char *argv[])

argv[] is address of array of string; string itself is an adress. Hence 2 levels of indirection on stack

exec()

```
int
exec(char *path, char **argv)
{
...
  uint argc, sz, sp,
ustack[3+MAXARG+1];
...
  if((ip = namei(path)) == 0){
    end_op();
    cprintf("exec: fail\n");
    return -1;
  }
```

- **ustack**
  - **used to build the arguments to be pushed on user-stack**
- **namei**
  - **get the inode of the executable file**

exec()

```
// Check ELF header
if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
  goto bad;
if(elf.magic != ELF_MAGIC)
  goto bad;

if((pgdir = setupkvm()) == 0)
  goto bad;
```

- **readi**
  - **read ELF header**
- **setupkvm()**
  - **creating a *new* page directory and mapping kernel pages**

exec()

```
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
  if(ph.type != ELF_PROG_LOAD)
    continue;
  if(ph.memsz < ph.filesz)
    goto bad;
  if(ph.vaddr + ph.memsz < ph.vaddr)
    goto bad;
  if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
  if(ph.vaddr % PGSIZE != 0)
    goto bad;
  if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) <
0)
    goto bad;
}
```

- Read ELF program headers from ELF file
- Map the code/data into pagedir-pagetable-pages
- Copy data from ELF file into the pages allocated

exec()

```
sz = PGROUNDUP(sz);
if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
  goto bad;
clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
sp = sz;
```

- Allocate 2 pages on top of proc->sz
- One page for stack
- one page for guard page
- Clear the valid flag on guard page

exec()

```
// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
  if(argc >= MAXARG)
    goto bad;
  sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
  if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
    goto bad;
  ustack[3+argc] = sp;
}
ustack[3+argc] = 0;
ustack[0] = 0xffffffff;  // fake return PC
ustack[1] = argc;
ustack[2] = sp - (argc+1)*4;  // argv pointer
sp -= (3+argc+1) * 4;
if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
  goto bad;
```

- For each entry in argv[]
  - copy it on user-stack
  - remember it's location on user stack in ustack
- add extra entries (to be copied to user stack) to ustack
- copy argc, argv pointer
- take sp to bottom
- copy ustack to user stack

This is what the code on earlier slide did

exec()

```
// Save program name for debugging.
for(last=s=path; *s; s++)
  if(*s == '/')
    last = s+1;
safestrcpy(curproc->name, last, sizeof(curproc->name));

// Commit to the user image.
oldpgdir = curproc->pgdir;
curproc->pgdir = pgdir;
curproc->sz = sz;
curproc->tf->eip = elf.entry;  // main
curproc->tf->esp = sp;
switchuvm(curproc);
freevm(oldpgdir);
return 0;
```

- copy name of new process in proc->name
- change to new page directory
- change new size
- tf->eip will be used when we return from exec() to jump to user code. Set to to first instruction of code, given by elf.entry
- Set user stack pointer to "sp" (bottom of stack of arguments)
- Update TSS, change CR3 to newpagedir
- free old page dir

return 0 from exec()?

- We know exec() does not return !
- This was exec() function !
  - Returns to sys_exec()
- sys_exec() also returns , where?
  - Remember we are still in kernel code, running on kernel stack. p->kstack has the trapframe setup
  - There is context struct on stack. Why?
  - sys_exec() returns to trapret(), the trap frame will be popped !
    - with “iret” jump into new program !
  - New program is not old program , which could have accessed return value of sys_exec()

File Systems

Abhijit A M
abhijit.comp@coep.ac.in

Introduction

- **Human end user's view of file sytem on a modern desktop operating system**
  - **Files, directories(folders), hierarchy – acyclic graph like structure**
  - **Windows Vs Linux logical organization: multiple partitions (C:, D:,etc.), vs single logical namespace starting at "/"**

Introduction

- **Secondary and Tertiary memory**
  - **Hard disks, Pen drives, CD-ROMs, DVDs, Magnetic Tapes, Portable disks, etc. Used for storing files**
  - **Each comes with a hardware “controller” that acts as an intermediary in the hardware/software boundary**
  - **IDE, SATA, SCSI, SAS, etc. Protocols : Different types of cables, speeds, signaling mechanisms**
  - **Controllers provide a block/sector based read/write access**
    - **Block size is most typically 512 bytes**
    - **Can’t read byte no. 33 directly. Must read sector 0 (byte 0 to 511) in memory and then access the byte no 33 in memory.**

Introduction

- **OS and File system**
  - **OS bridges the gap between end user and stoage hardware controller**
  - **Provides data structure to map the logical view of end users onto disk storage**
  - **Essentially an implementation of the acyclic graph on the sequential sector-based disk storage**
    - **Both in memory and on-disk**
  - **Provides system calls (open, read, write, ..., etc.) to enable access to files, folders**

What we are going to learn

- **The operating system interface (system calls, commands/utilities) for accessing files in a file-sysetm**
- **Design aspects of OS to implement the file system**

What is a file?

- A sequence of bytes , with
  - A name
  - Permissions
  - Owner
  - Timestamps,
  - Etc.
- Types: Text files, binary files
  - Text: All bytes are human readable
  - Binary: Non-text
- Types: ODT, MP4, TXT, DOCX, etc.
  - Most typically describing the organization of the data inside the file
  - Each type serving the needs of a particular application

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

What is a file?

- The sequence of bytes can be interpreted to be
  - Just a sequence of bytes
    - E.g. a text file
  - Sequence of records/structures
    - E.g. a file of student records
  - A complexly organized, collection of records and bytes
    - E.g. a “ODT” or “DOCX” file
- What’s the role of OS in above mentioned file type, and organization?
  - Mostly NO role on Unixes, Linuxes!
  - They are handled by applications !
  - Types handled by OS: normal file, directory, block device file, character device file, FIFO file (named pipe), etc.
  - Also types handled by OS: executable file, non-executable file

File attributes

- Run

  $ ls -l

  on Linux

  To see file listing with different attributes

- Different OSes and file-systems provide different sets of file attributes
  - Some attributes are common to most, while some are different
  - E.g. name, size, owner can be found on most systems
  - “Executable” permission may not be found on all systems

File protection attributes

- File owner/creator should be able to control:
  - what can be done
  - by whom
- Types of access
  - Read
  - Write
  - Execute
  - Append
  - Delete
  - List
- Linux: See commands: “chown”, “chgrp” and “chmod”

- Linux file permissions
  - For owner, group and others
  - Read, write and execute
  - Total 9 permissions

A Sample UNIX Directory Listing

-rw-rw-r--	1 pbg	staff	31200	Sep 3 08:30	intro.ps
drwx------	5 pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2 pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2 pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1 pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1 pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4 pbg	faculty	512	Jul 31 10:31	lib/
drwx------	3 pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3 pbg	staff	512	Jul 8 09:35	test/

Windows XP Access List

Access methods

- OS system calls may provide two types of access to files
  - Sequential Access
    - read next
    - write next
    - reset
    - no read after last write (rewrite)
  - Linux provides sequential access using open(), read(), write(), ...
- Direct Access
  - read n
  - write n
  - position to n
    read next
    write next
  - rewrite n

  n = relative block number
- pread(), pwrite() on Linux

Device Drivers

- Hardware manufacturers provide “hardware controllers” for their devices
- Hardware controllers can operate the hardware, as instructed
- Hardware controllers are instructed by writing to particular I/O ports using CPU’s machine instructions
  - This bridges the hardware-software gap
- OS programmers, typically, write one “device driver” code that interacts with one hardware controller
  - This is pure C code
  - Which calls I/O instructions for hardware controller
  - This code is independent of most of the OS code
  - Often this code is like an add-on Module , eg. Linux kernel

Disk device driver

- OS views the disk as a logical sequence of blocks
  - OS's assumed block size may be > sector size
- OS Talks to disk controller
- Helps the OS Convert it's view of "logical block" of the disk, into physical sector numbers
- Acts as a translator between rest of OS and hardware controller
- xv6: ide.c

OS’s job now

- To implement the logical view of file system as seen by end user
- Using the logical block-based view offerred by the device driver

Volume Managers

- Special type of kernel device drives, which reside on top of disk device drivers
- Provide a more abstract view of the underlying hardware
- E.g. Can combine two physical hard disks, and present them as one
- Allow end users to
  - "combine one more more physical disks to create physical volumes"
  - "create volume groups out of a set of physical volumes",
  - "create logical volums within volume groups"

Logical Volume Management Components

Formatting

- Physical hard disk divided into partitions
  - Partitions also known as minidisks, slices
- A raw disk partition is accessible using device driver – but no block contains any data !
  - Like an un-initialized array, or sectors/blocks
- Formatting
  - Creating an initialized data structure on the partition, so that it can start storing the acyclic graph tree structure on it
  - Different formats depending on different implementations of the directory tree structure: ext4, NTFS, vfat, VxFS, ReiserFS, WafleFS, etc.
- Formatting happens on “a physical partition” or “a logical volume made available by volume manager”

Different types of “layouts”
Single level directory

Naming problem

Grouping problem

Different types of “layouts”
Two level directory

Path name

Can have the same file name for different user

Efficient searching

No grouping capability

Tree Structured directories

Acyclic Graph Directories

General Graph directory

Mounting of a file system: before

Mounting of a file system: after

$sudo mount */dev/sda5 /users*

Remote mounting: NFS

- **Network file system**
- **$ sudo mount 10.2.1.2:/x/y /a/b**
  - **The /x/y partition on 10.2.1.2 will be made available under the folde /a/b on this computer**
-

File sharing semantics

- Consistency semantics specify how multiple users are to access a shared file simultaneously
- Unix file system (UFS) implements:
  - Writes to an open file visible immediately to other users of the same open file
  - One mode of sharing file pointer to allow multiple users to read and write concurrently
- AFS has session semantics
  - Writes only visible to sessions starting after the file is closed

Implementing file systems

File system on disk

- Disk I/O in terms of sectors (512 bytes)
- File system: implementation of acyclic graph using the linear sequence of sectors
- Device driver: available to rest of the OS code to access disk using a block number

File system implementation: layering

File system: Layering

- Device drivers manage I/O devices at the I/O control layer
  - Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” outputs low-level hardware specific commands to hardware controller
- Basic file system given command like “retrieve block 123” translates to device driver
  - Also manages memory buffers and caches (allocation, freeing, replacement)
  - Buffers hold data in transit
  - Caches hold frequently used data
- File organization module understands files, logical address, and physical blocks
  - Translates logical block # to physical block #
  - Manages free space, disk allocation
- Logical file system manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks (inodes in Unix)
  - Directory management
  - Protection

Application programs

```
int main() {
        char buf[128]; int count;
        fd = open(...):
        read(fd, buf, count);
}
```

OS

Logical file system:

```
sys_read(int fd, char *buf, int count) {
        file *fp = currproc->fdarray[fd];
        file_read(fp, ...);
}
```

File organization module:

```
file_read(file *fp, char *buf, int count) {
        offset = fp->current-offset;
        translate offset into blockno;
        basic_read(blockno, buf, count);

}
```

Basic File system:

```
basic_read(int blockno, char *buf, ...) {
        os_buffer *bp;
        sectorno = calculation on blockno;
        disk_driver_read(sectono, bp );
        move-process-to-wait-queue;
        copy-data-to-user-buffer(bp, buf);

}
```

IO Control, Device driver:

```
disk_driver_read(sectorno) {
        issue instructions to disk controller
(often assembly code)
        to read sectorno  into specific
location;
}
```

XV6 does it slightely differently, but following the layering principle!

Layering advantages

- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
  - Logical layers can be implemented by any coding method according to OS designer
- Many file systems, sometimes many within an operating system
  - Each with its own format (CD-ROM is ISO 9660; Unix has UFS, FFS; Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray, Linux has more than 40 types, with extended file system ext2 and ext3 leading; plus distributed file systems, etc)
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE
  - The Virtual File System (to be discussed later) helps us combine multiple physically different file systems into one logical layer at the top

File system implementation: Different problems to be solved

- What to do at boot time ?
- How to store directories and files on the partition ?
  - Complex problem. Hiearchy + storage allocation + efficiency + limits on file/directory sizes + links (hard, soft)
- How to manage list of free sectors/blocks?
- How to store the summary information about the complete file system
- How to mount a file system , how to unmount

File system implementation

- We have system calls at the API level, but how do we implement their functions?
  - On-disk and in-memory structures. Let's see some of the important ones.
  - Boot control block contains info needed by system to boot OS from that volume
    - Not always needed. Needed if volume contains OS, usually first block of volume
  - Volume control block (superblock, master file table) contains volume details
    - Total # of blocks, # of free blocks, block size, free block pointers or array
  - Per-file File Control Block (FCB) contains many details about the file
    - Inode(FCB) number, permissions, size, dates
  - Directory structure organizes the files
    - Names and inode numbers, master file table

A typical file control block (inode)

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Why does it NOT contain the

Name of the file ?

In memory data structures

- **Mount table**
  - **storing file system mounts, mount points, file system types**
- **See next slide for “file” realated data structures**
- **Buffers**
  - **hold data blocks from secondary storage**

In memory data structures: for open,read,write, ...

Open returns a file handle for subsequent use

Data from read eventually copied to specified user process memory address

At boot time

- **Root partition**
  - **Contains the file system hosting OS**
  - **“mounted” at boot time – contains “/”**
    - **Normally can’t be unmounted!**
- **Check all other partitions**
  - **Specified in /*etc*/fstab on Linux**
  - **Check if the data structure on them is consistent**
    - **Consistent != perfect/accurate/complete**

Directory Implementation

- **Problem**
  - **Directory contains files and/or subdirectories**
  - **Operations required – create files/directories, access files/directories, search for a file (during lookup), etc.**
  - **Directory needs to give location of each file on disk**

Directory Implementation

- Linear list of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - Linear search time
    - Could keep ordered alphabetically via linked list or use B+ tree
  - Ext2 improves upon this approach.
- Hash Table – linear list with hash data structure
  - Decreases directory search time
  - Collisions – situations where two file names hash to the same location
  - Only good if entries are fixed size, or use chained-overflow method

Disk space allocation for files

- File contain data and need disk blocks/sectors for storing it
- File system layer does the allocation of blocks on disk to files
- Files need to
  - Be created, expanded, deleted, shrunk, etc.
  - How to accommodate these requirements?

Contiguous Allocation of Disk Space

directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous allocation

- Each file occupies set of contiguous blocks
- Best performance in most cases
- Simple – only starting location (block #) and length (number of blocks) are required
- Problems include finding space for file, knowing file size, external fragmentation, need for compaction off-line (downtime) or on-line

Linked allocation of blocks to a file

Linked allocation of blocks to a file

- **Linked allocation**
  - **Each file a linked list of blocks**
  - **File ends at nil pointer**
  - **No external fragmentation**
  - **Each block contains pointer to next block (i.e. data + pointer to next block)**
  - **No compaction, external fragmentation**
- **Free space management system called when new block needed**
- **Improve efficiency by clustering blocks into groups but increases internal fragmentation**
- **Reliability can be a problem**
- **Locating a block can take many I/Os and disk seeks**

FAT: File Allocation Table

- FAT (File Allocation Table), a variation
  - Beginning of volume has table, indexed by block number
  - Much like a linked list, but faster on disk and cacheable
  - New block allocation simple

FAT: File Allocation Table

Variants: FAT8, FAT12, FAT16, FAT32, VFAT, ...

Indexed allocation

Indexed allocation

- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K bytes and block size of 512 bytes. We need only 1 block for index table

Multi level indexing

Unix UFS: combined scheme for block allocation

Free Space Management

- File system maintains free-space list to track available blocks/clusters
  - Bit vector or bit map (n blocks)
  - Or Linked list

Free Space Management: bit vector

- Each block is represented by 1 bit.
- If the block is free, the bit is 1; if the block is allocated, the bit is 0.
  - For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17
  - 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bitmap would be 00111100111111000110000011100000 ...
- A 1- TB disk with 4- KB blocks would require 32 MB (2^{40} / 2^{12} = 2^{28} bits = 2^{25} bytes = 2^{5} MB) to store its bitmap

Free Space Management: Linked list (not in memory, on disk!)

Further improvements on link list method of free-blocks

- Grouping
- Counting
- Space Maps (ZFS)
- Read as homework

Ext2 FS layout

Ext2 FS Layout

```
struct ext2_super_block {
    __le32  s_inodes_count;     /* Inodes count */
    __le32  s_blocks_count;     /* Blocks count */
    __le32  s_r_blocks_count;   /* Reserved blocks count */
    __le32  s_free_blocks_count;    /* Free blocks count */
    __le32  s_free_inodes_count;    /* Free inodes count */
    __le32  s_first_data_block; /* First Data Block */
    __le32  s_log_block_size;   /* Block size */
    __le32  s_log_frag_size;    /* Fragment size */
    __le32  s_blocks_per_group; /* # Blocks per group */
    __le32  s_frags_per_group;  /* # Fragments per group */
    __le32  s_inodes_per_group; /* # Inodes per group */
    __le32  s_mtime;        /* Mount time */
    __le32  s_wtime;        /* Write time */
    __le16  s_mnt_count;        /* Mount count */
    __le16  s_max_mnt_count;    /* Maximal mount count */
    __le16  s_magic;        /* Magic signature */
    __le16  s_state;        /* File system state */
    __le16  s_errors;       /* Behaviour when detecting errors */
```

```
struct ext2_super_block {
...
    __le16  s_minor_rev_level;  /* minor revision level */
    __le32  s_lastcheck;        /* time of last check */
    __le32  s_checkinterval;    /* max. time between checks */
    __le32  s_creator_os;       /* OS */
    __le32  s_rev_level;        /* Revision level */
    __le16  s_def_resuid;       /* Default uid for reserved blocks */
    __le16  s_def_resgid;       /* Default gid for reserved blocks */
    __le32  s_first_ino;        /* First non-reserved inode */
    __le16   s_inode_size;      /* size of inode structure */
    __le16  s_block_group_nr;   /* block group # of this superblock */
    __le32  s_feature_compat;   /* compatible feature set */
    __le32  s_feature_incompat;     /* incompatible feature set */
    __le32  s_feature_ro_compat;    /* readonly-compatible feature set */
    __u8    s_uuid[16];     /* 128-bit uuid for volume */
    char    s_volume_name[16];  /* volume name */
    char    s_last_mounted[64];     /* directory where last mounted */
    __le32  s_algorithm_usage_bitmap; /* For compression */
```

```
struct ext2_super_block {
...
    __u8    s_prealloc_blocks;  /* Nr of blocks to try to preallocate*/
    __u8    s_prealloc_dir_blocks;  /* Nr to preallocate for dirs */
    __u16   s_padding1;
    /*
     * Journaling support valid if EXT3_FEATURE_COMPAT_HAS_JOURNAL set.
     */
    __u8    s_journal_uuid[16]; /* uuid of journal superblock */
    __u32   s_journal_inum;     /* inode number of journal file */
    __u32   s_journal_dev;      /* device number of journal file */
    __u32   s_last_orphan;      /* start of list of inodes to delete */
    __u32   s_hash_seed[4];     /* HTREE hash seed */
    __u8    s_def_hash_version; /* Default hash version to use */
    __u8    s_reserved_char_pad;
    __u16   s_reserved_word_pad;
    __le32  s_default_mount_opts;
    __le32  s_first_meta_bg;    /* First metablock block group */
    __u32   s_reserved[190];    /* Padding to the end of the block */
```

```
struct ext2_group_desc
{
    __le32  bg_block_bitmap;        /* Blocks bitmap block */
    __le32  bg_inode_bitmap;        /* Inodes bitmap block */
    __le32  bg_inode_table;     /* Inodes table block */
    __le16  bg_free_blocks_count;   /* Free blocks count */
    __le16  bg_free_inodes_count;   /* Free inodes count */
    __le16  bg_used_dirs_count; /* Directories count */
    __le16  bg_pad;
    __le32  bg_reserved[3];
};
```

```
struct ext2_inode {
    __le16  i_mode;     /* File mode */
    __le16  i_uid;      /* Low 16 bits of Owner Uid */
    __le32  i_size;     /* Size in bytes */
    __le32  i_atime;    /* Access time */
    __le32  i_ctime;    /* Creation time */
    __le32  i_mtime;    /* Modification time */
    __le32  i_dtime;    /* Deletion Time */
    __le16  i_gid;      /* Low 16 bits of Group Id */
    __le16  i_links_count;  /* Links count */
    __le32  i_blocks;   /* Blocks count */
    __le32  i_flags;    /* File flags */
```

Inode in ext2

```
struct ext2_inode {
   ...
    union {
        struct {
            __le32  l_i_reserved1;
        } linux1;
        struct {
            __le32  h_i_translator;
        } hurd1;
        struct {
            __le32  m_i_reserved1;
        } masix1;
    } osd1;             /* OS dependent 1 */
    __le32  i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
    __le32  i_generation;   /* File version (for NFS) */
    __le32  i_file_acl; /* File ACL */
    __le32  i_dir_acl;  /* Directory ACL */
    __le32  i_faddr;    /* Fragment address */
```

```
struct ext2_inode {
   ...
    union {
        struct {
            __u8    l_i_frag;   /* Fragment number */            __u8    l_i_fsize;  /* Fragment size */
            __u16   i_pad1;             __le16  l_i_uid_high;   /* these 2 fields    */
            __le16  l_i_gid_high;   /* were reserved2[0] */
            __u32   l_i_reserved2;
        } linux2;
        struct {
            __u8    h_i_frag;   /* Fragment number */            __u8    h_i_fsize;  /* Fragment size */
            __le16  h_i_mode_high;            __le16  h_i_uid_high;
            __le16  h_i_gid_high;
            __le32  h_i_author;
        } hurd2;
        struct {
            __u8    m_i_frag;   /* Fragment number */            __u8    m_i_fsize;  /* Fragment size */
            __u16   m_pad1;             __u32   m_i_reserved2[2];
        } masix2;
    } osd2;             /* OS dependent 2 */
```

Ext2 FS Layout: Directory entry

Let’s see a program to read superblock of an ext2 file system.

Efficiency and Performance (and the risks created while trying to achieve it!)

Efficiency

- **Efficiency dependent on:**
  - **Disk allocation and directory algorithms**
  - **Types of data kept in file's directory entry**
  - **Pre-allocation or as-needed allocation of metadata structures**
  - **Fixed-size or varying-size data structures**
  -

Performance

- Keeping data and metadata close together
- Buffer cache – separate section of main memory for frequently used blocks
- Synchronous writes sometimes requested by apps or needed by OS
- No buffering / caching – writes must hit disk before acknowledgement
- Asynchronous writes more common, buffer-able, faster
- Free-behind and read-ahead – techniques to optimize sequential access
- Reads frequently slower than writes

Page cache

- A page cache caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

I/O Without a Unified Buffer Cache

Unified buffer cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching
- But which caches get priority, and what replacement algorithms to use?

I/O Using a Unified Buffer Cache

Recovery

- **Problem. Consider creating a file on ext2 file system.**
  - **Following on disk data structures will/may get modified**
  - **Directory data block, new directory data block, block bitmap, inode table, inode table bitmap, group descriptor, super block, data blocks for new file, more data block bitmaps, ...**
  - **All cached in memory by OS**
- **Delayed write – OS writes changes in its in-memory data structures, and schedules writes to disk when convenient**
  - **Possible that some of the above changes are written, but some are not**
  - **Inconsistent data structure! --> Example: inode table written, inode bitmap written, but directory data block not written**

Recovery

- Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - Can be slow and sometimes fails
- Use system programs to back up data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by restoring data from backup

Log structured file systems

- Log structured (or journaling) file systems record each metadata update to the file system as a transaction
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log (sequentially)
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system structures
  - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata

Journaling file systems

- Veritas FS
- Ext3, Ext4
- Xv6 file system!

File Systems

Abhijit A M
abhijit.comp@coep.ac.in

System calls related to files/file-system

- Open(2), chmod(2), chown(2), close(2), dup(2), fcntl(2), link(2), lseek(2), mknod(2), mmap(2), mount(2), read(2), stat(2), umask(2), unlink(2), write(2), fstat(2), access(2), readlink(2), ...

Implementing file systems

File system on disk

- Disk I/O in terms of sectors (512 bytes)
- File system: implementation of acyclic graph using the linear sequence of sectors
- Device driver: available to rest of the OS code to access disk using a block number

File system implementation: layering

Application programs

```
int main() {
        char buf[128]; int count;
        fd = open(...):
        read(fd, buf, count);
}
```

OS

Logical file system:

```
sys_read(int fd, char *buf, int count) {
        file *fp = currproc->fdarray[fd];
        file_read(fp, ...);
}
```

File organization module:

```
file_read(file *fp, char *buf, int count) {
        offset = fp->current-offset;
        translate offset into blockno;
        basic_read(blockno, buf, count);

}
```

Basic File system:

```
basic_read(int blockno, char *buf, ...) {
        os_buffer *bp;
        sectorno = calculation on blockno;
        disk_driver_read(sectono, bp );
        move-process-to-wait-queue;
        copy-data-to-user-buffer(bp, buf);

}
```

IO Control, Device driver:

```
disk_driver_read(sectorno) {
        issue instructions to disk controller
(often assembly code)
        to read sectorno  into specific
location;
}
```

XV6 does it slightely differently, but following the layering principle!

A typical file control block (inode)

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Why does it NOT contain the

Name of the file ?

Disk space allocation for files

- File contain data and need disk blocks/sectors for storing it
- File system layer does the allocation of blocks on disk to files
- Files need to
  - Be created, expanded, deleted, shrunk, etc.
  - How to accommodate these requirements?

Contiguous Allocation of Disk Space

directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous allocation

- Each file occupies set of contiguous blocks
- Best performance in most cases
- Simple – only starting location (block #) and length (number of blocks) are required
- Problems include finding space for file, knowing file size, external fragmentation, need for compaction off-line (downtime) or on-line

Linked allocation of blocks to a file

Linked allocation of blocks to a file

- **Linked allocation**
  - **Each file a linked list of blocks**
  - **File ends at nil pointer**
  - **No external fragmentation**
  - **Each block contains pointer to next block (i.e. data + pointer to next block)**
  - **No compaction, external fragmentation**
- **Free space management system called when new block needed**
- **Improve efficiency by clustering blocks into groups but increases internal fragmentation**
- **Reliability can be a problem**
- **Locating a block can take many I/Os and disk seeks**

FAT: File Allocation Table

- FAT (File Allocation Table), a variation
  - Beginning of volume has table, indexed by block number
  - Much like a linked list, but faster on disk and cacheable
  - New block allocation simple

FAT: File Allocation Table

Variants: FAT8, FAT12, FAT16, FAT32, VFAT, ...

Indexed allocation

Indexed allocation

- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K bytes and block size of 512 bytes. We need only 1 block for index table

Multi level indexing

Unix UFS: combined scheme for block allocation

Free Space Management

- File system maintains free-space list to track available blocks/clusters
  - Bit vector or bit map (n blocks)
  - Or Linked list

Free Space Management: bit vector

- Each block is represented by 1 bit.
- If the block is free, the bit is 1; if the block is allocated, the bit is 0.
  - For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17
  - 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bitmap would be 00111100111111000110000011100000 ...
- A 1- TB disk with 4- KB blocks would require 32 MB (2^{40} / 2^{12} = 2^{28} bits = 2^{25} bytes = 2^{5} MB) to store its bitmap

Free Space Management: Linked list (not in memory, on disk!)

Further improvements on link list method of free-blocks

- Grouping
- Counting
- Space Maps (ZFS)
- Read as homework

Directory Implementation

- **Problem**
  - **Directory contains files and/or subdirectories**
  - **Operations required – create files/directories, access files/directories, search for a file (during lookup), etc.**
  - **Directory needs to give location of each file on disk**

Directory Implementation

- Linear list of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - Linear search time
    - Could keep ordered alphabetically via linked list or use B+ tree
  - Ext2 improves upon this approach.
- Hash Table – linear list with hash data structure
  - Decreases directory search time
  - Collisions – situations where two file names hash to the same location
  - Only good if entries are fixed size, or use chained-overflow method

Ext2 FS layout

Ext2 FS Layout

```
struct ext2_super_block {
    __le32  s_inodes_count;     /* Inodes count */
    __le32  s_blocks_count;     /* Blocks count */
    __le32  s_r_blocks_count;   /* Reserved blocks count */
    __le32  s_free_blocks_count;    /* Free blocks count */
    __le32  s_free_inodes_count;    /* Free inodes count */
    __le32  s_first_data_block; /* First Data Block */
    __le32  s_log_block_size;   /* Block size */
    __le32  s_log_frag_size;    /* Fragment size */
    __le32  s_blocks_per_group; /* # Blocks per group */
    __le32  s_frags_per_group;  /* # Fragments per group */
    __le32  s_inodes_per_group; /* # Inodes per group */
    __le32  s_mtime;        /* Mount time */
    __le32  s_wtime;        /* Write time */
    __le16  s_mnt_count;        /* Mount count */
    __le16  s_max_mnt_count;    /* Maximal mount count */
    __le16  s_magic;        /* Magic signature */
    __le16  s_state;        /* File system state */
    __le16  s_errors;       /* Behaviour when detecting errors */
```

```
struct ext2_super_block {
...
    __le16  s_minor_rev_level;  /* minor revision level */
    __le32  s_lastcheck;        /* time of last check */
    __le32  s_checkinterval;    /* max. time between checks */
    __le32  s_creator_os;       /* OS */
    __le32  s_rev_level;        /* Revision level */
    __le16  s_def_resuid;       /* Default uid for reserved blocks */
    __le16  s_def_resgid;       /* Default gid for reserved blocks */
    __le32  s_first_ino;        /* First non-reserved inode */
    __le16   s_inode_size;      /* size of inode structure */
    __le16  s_block_group_nr;   /* block group # of this superblock */
    __le32  s_feature_compat;   /* compatible feature set */
    __le32  s_feature_incompat;     /* incompatible feature set */
    __le32  s_feature_ro_compat;    /* readonly-compatible feature set */
    __u8    s_uuid[16];     /* 128-bit uuid for volume */
    char    s_volume_name[16];  /* volume name */
    char    s_last_mounted[64];     /* directory where last mounted */
    __le32  s_algorithm_usage_bitmap; /* For compression */
```

```
struct ext2_super_block {
...
    __u8    s_prealloc_blocks;  /* Nr of blocks to try to preallocate*/
    __u8    s_prealloc_dir_blocks;  /* Nr to preallocate for dirs */
    __u16   s_padding1;
    /*
     * Journaling support valid if EXT3_FEATURE_COMPAT_HAS_JOURNAL set.
     */
    __u8    s_journal_uuid[16]; /* uuid of journal superblock */
    __u32   s_journal_inum;     /* inode number of journal file */
    __u32   s_journal_dev;      /* device number of journal file */
    __u32   s_last_orphan;      /* start of list of inodes to delete */
    __u32   s_hash_seed[4];     /* HTREE hash seed */
    __u8    s_def_hash_version; /* Default hash version to use */
    __u8    s_reserved_char_pad;
    __u16   s_reserved_word_pad;
    __le32  s_default_mount_opts;
    __le32  s_first_meta_bg;    /* First metablock block group */
    __u32   s_reserved[190];    /* Padding to the end of the block */
```

```
struct ext2_group_desc
{
    __le32  bg_block_bitmap;        /* Blocks bitmap block */
    __le32  bg_inode_bitmap;        /* Inodes bitmap block */
    __le32  bg_inode_table;     /* Inodes table block */
    __le16  bg_free_blocks_count;   /* Free blocks count */
    __le16  bg_free_inodes_count;   /* Free inodes count */
    __le16  bg_used_dirs_count; /* Directories count */
    __le16  bg_pad;
    __le32  bg_reserved[3];
};
```

```
struct ext2_inode {
    __le16  i_mode;     /* File mode */
    __le16  i_uid;      /* Low 16 bits of Owner Uid */
    __le32  i_size;     /* Size in bytes */
    __le32  i_atime;    /* Access time */
    __le32  i_ctime;    /* Creation time */
    __le32  i_mtime;    /* Modification time */
    __le32  i_dtime;    /* Deletion Time */
    __le16  i_gid;      /* Low 16 bits of Group Id */
    __le16  i_links_count;  /* Links count */
    __le32  i_blocks;   /* Blocks count */
    __le32  i_flags;    /* File flags */
```

Inode in ext2

```
struct ext2_inode {
   ...
    union {
        struct {
            __le32  l_i_reserved1;
        } linux1;
        struct {
            __le32  h_i_translator;
        } hurd1;
        struct {
            __le32  m_i_reserved1;
        } masix1;
    } osd1;             /* OS dependent 1 */
    __le32  i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
    __le32  i_generation;   /* File version (for NFS) */
    __le32  i_file_acl; /* File ACL */
    __le32  i_dir_acl;  /* Directory ACL */
    __le32  i_faddr;    /* Fragment address */
```

```
struct ext2_inode {
   ...
    union {
        struct {
            __u8    l_i_frag;   /* Fragment number */            __u8    l_i_fsize;  /* Fragment size */
            __u16   i_pad1;             __le16  l_i_uid_high;   /* these 2 fields    */
            __le16  l_i_gid_high;   /* were reserved2[0] */
            __u32   l_i_reserved2;
        } linux2;
        struct {
            __u8    h_i_frag;   /* Fragment number */            __u8    h_i_fsize;  /* Fragment size */
            __le16  h_i_mode_high;            __le16  h_i_uid_high;
            __le16  h_i_gid_high;
            __le32  h_i_author;
        } hurd2;
        struct {
            __u8    m_i_frag;   /* Fragment number */            __u8    m_i_fsize;  /* Fragment size */
            __u16   m_pad1;             __u32   m_i_reserved2[2];
        } masix2;
    } osd2;             /* OS dependent 2 */
```

Ext2 FS Layout: Directory entry

Let’s see a program to read superblock of an ext2 file system.

Efficiency and Performance (and the risks created while trying to achieve it!)

In memory data structures

- **Mount table**
  - **storing file system mounts, mount points, file system types**
- **See next slide for “file” realated data structures**
- **Buffers**
  - **hold data blocks from secondary storage**

In memory data structures: for open,read,write, ...

Open returns a file handle for subsequent use

Data from read eventually copied to specified user process memory address

Efficiency

- **Efficiency dependent on:**
  - **Disk allocation and directory algorithms**
  - **Types of data kept in file's directory entry**
  - **Pre-allocation or as-needed allocation of metadata structures**
  - **Fixed-size or varying-size data structures**
  -

Performance

- Keeping data and metadata close together
- Buffer cache – separate section of main memory for frequently used blocks
- Synchronous writes sometimes requested by apps or needed by OS
- No buffering / caching – writes must hit disk before acknowledgement
- Asynchronous writes more common, buffer-able, faster
- Free-behind and read-ahead – techniques to optimize sequential access
- Reads frequently slower than writes

Page cache

- A page cache caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

I/O Without a Unified Buffer Cache

Unified buffer cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching
- But which caches get priority, and what replacement algorithms to use?

I/O Using a Unified Buffer Cache

Recovery

- **Problem. Consider creating a file on ext2 file system.**
  - **Following on disk data structures will/may get modified**
  - **Directory data block, new directory data block, block bitmap, inode table, inode table bitmap, group descriptor, super block, data blocks for new file, more data block bitmaps, ...**
  - **All cached in memory by OS**
- **Delayed write – OS writes changes in its in-memory data structures, and schedules writes to disk when convenient**
  - **Possible that some of the above changes are written, but some are not**
  - **Inconsistent data structure! --> Example: inode table written, inode bitmap written, but directory data block not written**

Recovery

- Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - Can be slow and sometimes fails
- Use system programs to back up data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by restoring data from backup

Log structured file systems

- Log structured (or journaling) file systems record each metadata update to the file system as a transaction
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log (sequentially)
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system structures
  - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata

Journaling file systems

- Veritas FS
- Ext3, Ext4
- Xv6 file system!

File System Code

open,read, write, close, pipe, fstat, chdir, dup, mknod, link, unlink, mkdir,

Files, Inodes, Buffers

What we already know

- File system related system calls
  - deal with ‘fd’ arrays (ofile in xv6). open() returns first empty index. open should ideallly locate the inode on disk and initialize some data structures
  - maintain ‘offsets’ within a ‘file’ to support sequential read/write
  - dup() like system calls duplicate pointers in fd-array
  - read/write like system calls, going through ‘ofile’ array, should locate data of file on disk
  - We need functions to read/write from disk – that is disk driver
  - cache data of files in OS data structures for performance : buffering
  - Need to handle on disk data structures as well
- Faster recovery (like journaling in ext3) is desired

xv6 file handling code

- Is a very good example in ‘design’ of a layered and modular architecture
- Splits the entire work into different modules, and modules into functions properly
- The task of each function is neatly defined and compartamentalized

Layers of xv6 file system code

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

Normally, any upper layer can call any lower layer below

Abhijit: Block allocator should be considered as another Layer!

Layout of xv6 file system

May see the code of mkfs.c to get insight into the layout

```
struct superblock {
  uint size;         // Size of file system image (blocks)
  uint nblocks;      // Number of data blocks
  uint ninodes;      // Number of inodes.
  uint nlog;         // Number of log blocks
  uint logstart;     // Block number of first log block
  uint inodestart;   // Block number of first inode block
  uint bmapstart;    // Block number of first free map block
};
#define ROOTINO 1  // root i-number
#define BSIZE 512  // block size
```

Layout of xv6 file system

```
#define NDIRECT 12
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT)
// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEV only)
  short minor;          // Minor device number (T_DEV only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
};

#define DIRSIZ 14

struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

File on disk

dinode
type
major
minor
nlink
size
address 1
.....
address 12
indirect
data
...
data
indirect block
address 1
.....
address 128
data
...
data

Let’s discuss lowest layer first

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

Normally, any upper layer can call any lower layer below

ide.c: idewait, ideinit, idestart, ideintr, iderw

```
static struct spinlock idelock;
static struct buf *idequeue;
static int havedisk1;
```

- ideinit
  - was called from main.c: main()
  - Initialized IDE controller by writing to certain ports
  - havedisk=1 setup
  - Initialize idelock
- idewait
  - BUSY loop waiting for IDE to be ready

ide.c: idewait, ideinit, idestart, ideintr, iderw

- void idestart(buf *b)
  - static void idestart(struct buf *b)
  - Calculate sector number on disk using b->blockno
  - Issue a read/write command to IDE controller.
  - (This is the first buf on idequeue)
- ideintr
  - Take idelock. Called on IDE interrupt (through alltraps()->trap())
  - Wakeup the process waiting on first buffer in buffer *idequeue;
  - call idestart(). Release idelock.
- iderw(buf *b)
  - Move buf b to end of idequeue
  - Call idestart() if not running, sleep on idelock

Let’s see buffer cache layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

Normally, any upper layer can call any lower layer below

Reminder: After main()->binit()

Conceptually Linked liks this

Buffers keep moving on list, as LRU

struct bcache

struct buf

```
struct buf {
  int flags;  // 0 or B_VALID or B_DIRTY
  uint dev;   // device number
  uint blockno; // seq block number on device
  struct sleeplock lock; // Lock to be held by process using it
  uint refcnt; // Number of live accesses to the buf
  struct buf *prev; //  cache list
  struct buf *next; // cache list
  struct buf *qnext; // disk queue
  uchar data[BSIZE];  // data 512 bytes
};
#define B_VALID 0x2  // buffer has been read from disk
#define B_DIRTY 0x4  // buffer needs to be written to disk
```

buffer cache:
static struct buf* bget(uint dev, uint blockno)

- The bcache.head list is maintained on Most Recently Used (MRU) basis
  - head.next is the Most Recently Used (MRU) buffer
  - hence head.prev is the Least Recently Used (LRU)
- Look for a buffer with b->blockno = blockno and b->dev = dev
  - Search the head.next list for existing buffer (MRU order)
  - Else search the head.prev list for empty buffer
  - panic() if found in-use or empty buffer
- Increment b->refcnt ; Returns buffer locked
- Does not change the list structure, just returns a buf in use

buffer cache:
struct buf* bread(uint dev, uint blockno)

```
struct buf*
bread(uint dev, uint blockno)
{
  struct buf *b;
  b = bget(dev, blockno);
  if((b->flags & B_VALID) == 0) {
    iderw(b);
  }
  return b; // locked buffer
}
```

```
void
bwrite(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("bwrite");
  b->flags |= B_DIRTY;
  iderw(b);
}
```

Recollect: iderw moves buf to tail of idequeue, calls idestart() and sleep()

buffer cache:
void brelse(struct buf *b)

- release lock on buffer
- b->refcnt = 0
- If b->refcnt = 0
  - Means buffer will no longer be used
  - Move it to front of the front of bcache.head

Overall in this diagram

Buffers keep moving to the front of the list and around
The list always contains NBUF=30 buffers
head.next is always the MRU and head.prev is always LRU buffer

Let's see logging layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

Normally, any upper layer can call any lower layer below

log in xv6

- a mechanism of recovery from disk
- Concept: multiple write operations needed for system calls (e.g. ‘open’ system call to create a file in a directory)
  - some writes succed and some don’t
  - leading to inconsistencies on disk
- In the log, all changes for a ‘transaction’ (an operation) are either written completely or not at all
- During recovery, completed operations can be “rerun” and incomplete operations neglected

log in xv6

- xv6 system call does not directly write the on-disk file system data structures.
- A system call calls begin_op() at begining and end_op() at end
  - begin_op() increments log.outstanding
  - end_op() decrements log.outstanding, and if it's 0, then calls commit()
- During the code of system call, whenever a buffer is modified, (and done with)
  - log_write() is called
  - This copies the block in an array of blocks inside log, the block is not written in it's actual place in FS as of now
- when finally commit() is called, all modified blocks are copied to disk in the file system

log

```
struct logheader { // ON DISK
  int n; // number of entries in use in block[] below
  int block[LOGSIZE]; // List of block numbers stored
};
struct log { // only in memory
  struct spinlock lock;
  int start;  // first log block on disk (starts with logheader)
  int size; // total number of log blocks (in use out of 30)
  int outstanding; // how many FS sys calls are executing.
  int committing;  // in commit(), please wait.
  int dev; // FS device
  struct logheader lh;  // copy of the on disk logheader
};
struct log log;
```

log on disk

log.start
log.start + 1
log.start+30
boot block
super block
log
data of 52nd block
data of 68nd block
...
log
indoes | bitmap | data ....
block[30]
Start, size, etc
n =2
52
68
0
1
2
3
.....
29
logheader

Typical use case of logging

```
/* In a system call code * /
begin_op();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```

prepare for logging. Wait if logging system is not ready or 'committing'. ++outstanding

read and get access to a data block – as a buffer

modify buffer

note down this buffer for writing, in log. proxy for bwrite(). Mark B_DIRTY. Absorb multiple writes into one.

Syscall done. write log and all blocks. --outstanding.

If outstanding = 0, commit().

match colors in code and comments on right-side

Example of calls to logging

```
//file_write() code
begin_op();
ilock(f->ip);
   /*loop */ r = writei(f->ip, ...);
iunlock(f->ip);
end_op();
```

- each writei() in turn calls bread(), log_write() and brelse()
  - also calles iupdate(ip) which also calls bread, log_write and brelse
- Multiple writes are combined between begin_op() and end_op()

Logging functions

- Initlog()
  - Set fields in global log.xyz variables, using FS superblock
  - Recovery if needed
  - Called from first forkret()
- Following three called by FS code
- begin_op(void)
  - Increment log.outstanding
- end_op(void)
  - Decrement log.oustanding and call commit() if it's zero
- log_write(buf *)
  - Remember the specified block number in log.lh.block[] array
  - Set the block to be dirty
- write_log(void)
  - Called only from commit()
  - Use block numbers specified in log.lh.block and copy those blocks from memory to log-blocks
- commit(void)
  - Called only from end_op()
  - write_log()
  - Write header to disk log-header
  - Copy from log blocks to actual FS blocks
  - Reset and write log header again

Let’s see block allocation layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

Normally, any upper layer can call any lower layer below

Abhijit: Block allocator should be considered as another Layer!

allocating & deallocating blocks on DISK

- balloc(devno)
  - looks for a block whose bitmap bit is zero, indicating that it is free.
  - On finding updates the bitmap and returns the block.
  - balloc() calls bread()->bget to get a block from disk in a buffer.
    - Race prevented by the fact that the buffer cache only lets one process use any one bitmap block at a time.
  - Calls log_write(bp);
    - Thus writes to bitmap blocks are also logged

- bfree(devno, blockno)
  - finds the right bitmap block and clears the right bit.
  - Also calls log_write()

Let’s see Inode Layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

On disk & in memory inodes

```
struct {
  struct spinlock lock;
  struct inode inode[NINODE];
} icache;

// On-disk inode structure
struct dinode {
  short type;     // File type
  short major;   // T_DEV Major device number
  short minor;   // Minor device number
  short nlink;    // Number of links
  uint size;       // Size of file (bytes)
  uint addrs[NDIRECT+1];   /
};
```

```
// in-memory copy of an inode
struct inode {
  uint dev;          // Device number
  uint inum;         // Inode number
  int ref;           // Reference count
  struct sleeplock lock; // protects everything below here
  int valid;         //  been read from disk?

  short type;        // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

In memory inodes

- Kernel keeps a subset of on disk inodes, those in use, in memory
  - as long as ‘ref’ is >0
- The iget and iput functions acquire and release pointers to an inode, modifying the ref count.
- See the caller graph of iget()
  - all those who call iget()
- Sleep lock in ‘inode’ protects
  - fields in inode
  - data blocks of inode

iget and iupdate

- iget
  - searches for an existing/free inode in icache and returns pointer to one
  - if found, increments ref and returns pointer to inode
  - else gets empty inode , initializes, ref=1 and return
  - No lock held after iget()
  - Code must call ilock() after iget() to get lock
  - During lookup (later), many processes can iget() an inode, but only one holds the lock

- iupdate(inode *ip)
  - read on disk block of inode
  - get on disk inode
  - modify it as specified in ‘ip’
  - modify disk block of inode
  - log_write(disk block of inode)

itrunc , iput

- iput(ip)
  - if ref is 1
    - itrunc(ip)
    - type = 0
    - iupdate(ip)
    - i->valid = 0 // free in memory
  - else
    - ref--

- itrunc(ip)
  - write all data blocks of inode to disk
    - using bfree()
  - ip->size = 0
    - Inode is freed from use
  - iupdate(ip)
  - called from iput() only when ‘ref’ becomes zero

race in iput ?

- A concurrent thread might be waiting in ilock to use this inode
  - and won't be prepared to find the in ode is not longer allocated
- This is not possible. Why?
  - no way for a syscall to get a ref to a inode with ip->ref = 1

```
void
iput(struct inode *ip)
{
  acquiresleep(&ip->lock);
  if(ip->valid && ip->nlink == 0){
    acquire(&icache.lock);
    int r = ip->ref;
    release(&icache.lock);
    if(r == 1){
      // inode has no links and no other references: truncate and free.
      itrunc(ip);
```

buffer and inode cache

- to read an inode, it's block must be read in a buffer
- So the buffer always contains a copy of the on-disk dinode
  - duplicate copy in in-memory inode
- The inode cache is write-through,
  - code that modifies a cached inode must immediately write it to disk with iupdate
- Inode may still exist in the buffer cache

allocating inode

- ialloc(dev, type)
  - Loop over all disk inodes
  - read inode (from it's block)
  - if it's free (note inum)
  - zero on disk inode
  - write on disk inode (as zeroes)
  - return iget(dev, inum)
- panic if no free inodes

- ilock
  - code must acquire ilock before using inode's data/fields
  - llock reads inode if it's already not in memory

Trouble with iput() and crashes

- iput() doesn't truncate a file immediately when the link count for the file drops to zero, because
  - some process might still hold a reference to the inode in memory: a process might still be reading and writing to the file, because it successfully opened it.
- if a crash happens before the last process closes the file descriptor for the file,
  - then the file will be marked allocated on disk but no directory entry points to it
- Unsolved problem.
- How to solve it?

Get Inode data: bmap(ip, bn)

- **Allocate ‘bn’th block for the file given by inode ‘ip’**
- **Allocate block on disk and store it in either direct entries or block of indirect entries**
  - **allocate block of indirect entries if needed using balloc()**

writing/reading data at a given offset in file

readi(struct inode *ip, char *dst, uint off, uint n)

writei(struct inode *ip, char *src, uint off, uint n)

- **Calculate the block number in file where ‘off’ belongs**
- **Read sufficient blocks to read ‘n’ bytes**
- **using bread(), brelse()**
- **Call devsw.read if inode is a device Inode.**
- **Writei() also updates size if required**

Reading Directory Layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

directory entry

```
#define DIRSIZ 14

struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

Data of a directory file is a sequence of such entries. To find a name, just get all the data blocks and search the name

How to get the data for a directory? We already know the ans!

struct inode*
dirlookup(struct inode *dp, char *name, uint *poff)

- **Given a pointer to directory inode (dp), name of file to be searched**
  - **return the pointer to inode of that file (NULL if not found)**
  - **set the 'offset' of the entry found, inside directories data blocks, in poff**
- **How was 'dp' obtained? Who should be calling dirlookup? Why is poff returned?**
  - **During resolution of pathnames?**
- **Code: call readi() to get data of dp, search name in it, name comes with inode-num, iget() that inode-num**

int
dirlink(struct inode *dp, char *name, uint inum)

- Create a new entry for ‘name’_’inum’ in directory given by ‘dp’
  - inode number must have been obtained before calling this. How to do that?
- Use dirlookup() to verify entry does not exist!
- Get empty slot in directory’s data block
- Make directory entry
- Update directory inode! writei()

namex

- Called by namei(), or nameiparent()
- Just iteratively split a path using “/” separator and get inode for last component
- iget() root inode, then
- Repeatedly calls
  - split on “/”, dirlookup() for next component
-

races in namex()

- Crucial. Called so many times!
- one kernel thread is looking up a pathname another kernel thread may be changing the directory by calling unlink
  - when executing dirlookup in namex, the lookup thread holds the lock on the directory and dirlookup() returns an inode that was obtained using iget.
- Deadlock? next points to the same inode as ip when looking up ".". Locking next before releasing the lock on ip would result in a deadlock.
  - namex unlocks the directory before obtaining a lock on next.

File descriptor layer code

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

data structures related to “file” layer

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe; // used only if it works as a pipe
  struct inode *ip;
  uint off;
};
// interesting no lock in struct file !
```

```
struct proc {
...
  struct file *ofile[NOFILE];  // Open files per process
...
}
struct {
  struct spinlock lock;
  struct file file[NFILE];
} ftable; //global table from which ‘file’ is allocated to every process
```

Lock is used to protect updates to every entry in the array

Multiple processes accessing same file.

- Each will get a different ‘struct file’
  - but share the inode !
  - different offset in struct file, for each process
  - Also true, if same process opens file many times
- File can be a PIPE (more later)
  - what about STDIN, STDOUT, STDERR files ?
  - Figure out!
- ref
  - used if the file was ‘duped’ or process forked . in that case the ‘struct file’ is shared

file layer functions

- **filealloc**
  - find an empty struct file in ‘ftable’ and return it
  - set ref = 1
- **filedup(file *)**
  - simply ref++
- **fileclose**
  - --ref
  - if ref = 0
    - free struct file
    - iput() / pipeclose()
    - note – transaction if iput() called
- **filestat**
  - simply return fields from inode, after holding lock. on inodes for files only.

file layer functions

- **fileread**
  - **call readi() or piperead()**
  - **readi() later calls device-read or inode read (using bread())**
- **filewrite**
  - **call pipewrite() or writei()**
  - **writei() is called in a loop, within a transaction**

- **Why does readi() call read on the device , why not fileread() itself call device read ?**

pipes

```
struct pipe {
  struct spinlock lock;
  char data[PIPESIZE];
  uint nread;
 // number of bytes read
  uint nwrite;
 // number of bytes written
  int readopen;
  // read fd is still open
  int writeopen;
 // write fd is still open
};
```

- functions
  - pipealloc
  - pipeclose
  - piprea d
  - pipewrite
-

pipes

- pipealloc
  - allocate two struct file
  - allocate pipe itself using kalloc (it's a big structure with array)
  - init lock
  - initialize both struct file as 2 ends (r/w)
- pipewrite
  - wait if pipe full
  - write to pipe
  - wakeup processes waiting to read
- piperead
  - wait if no data
  - read from pipe
  - wakeup processes waiting to write
- Good producer consumer code !

Further to reading system call code now

- Now we are ready to read the code of system calls on file system
  - sys_open, sys_write, sys_read , etc.
- Advise: Before you read code of these, contemplate on what these functions should do using the functions we have studied so far.
- Also think of locks that need to be held.

Pointers in C

Abhijit A.M.
abhijit13@gmail.com

(C) Abhijit A.M.
Shared under Creative Commons Attribution Sharealike International License V3.0

Courtsey: Programmer Jokes on telegram

Pointer

- Pointer is a variable which can store addresses
- Pointer has a “type” (except void pointer)
  - e.g. int *p; char *cp; double *dp;
  - Here p, cp, dp are respectively pointers to integer, character, and double
- Size of pointer is decided by compiler

Operations on (and related to) Pointers

&

*

=

+int -int

-

[ ]

Operations related to pointers: &

```
int *p, i;
p = &i;
```

- & fetches the address of variable
  - Called Referencing operator
  - Here, &i is address of i
  - RHS is address of integer, LHS is variable which can store address of integer
- Diagram of this operation is shown on left side
- No need of assuming some value for address (e.g. address 1028). Just the diagram is sufficient to understand the concept

Operations on pointers: *

```
int *p, i, j;
i = 10;
p = &i;
j  = *p;
```

- *** fetches the value stored at a given address**
  - **Called dereferencing operator**
- ***p : here "value of p" is itself an address *(of i)*, so *p is value stored at "value of p" that is i**
- **Diagram's make it easy to understand, *p is simply the contents of box p points to**

Operations on pointers: *

```
int *p, i, j;
i = 10;
p = &i;
j  = *p;
```

```
char *cp, c, d;
c = 'x';
cp = &c;
d  = *cp;
```

- What is the difference between * in the two codes?
- The *p fetches the value at given address in sizeof(int) bytes, while *cp fetches the value at given address in sizeof(char)=1 bytes
- * works based on the size of the type!

Operations on pointers: =

```
int *p, i, *q;
i = 10;
p = &i;
q  = p;
```

- **Pointers can be copied**
  - **Can arrays be copied?**
- **Thumb rule:**
  - **After pointer copy, both pointers point to same location**
- **Common Mistake**
  - **One pointer pointing to another**

Operations on pointers: +- int

```
int *p, a[4], *q;
p =  &a[1];
q = p + 2;
*q = 40;
```

- C allows adding or substracting an int from a pointer!
  - Weird, but true!
  - e.g. int *p, n; p + n;
- The result is a pointer of the same type
- The resultant pointer points *n type locations ahead(for +) or before (for -)*
- *A type location is equal to sizeof(type).*

Problems: Draw diagrams for the code

```
int main() {
    int *p, *q, a[3], b;
    a[0] = 10; b = 2;
    p = &a[1];
    q = p + 1;
    p = q – b;
    *p = 30;
}
```

```
int main() {
    int *p, *q, a[3], b;
    a[0] = 10; b = 1;
    p = &a[3];
    q = p - 3;
    p = q + 1;
    *(q + 1) = 30;
    *(p – 1) = 20;
}
```

Operations on pointers: substracting two pointers

```
int *p, a[4], *q, x;
p =  &a[1];
q = p + 2;
x = q – p;
```

- **Two pointers of the *same type* can be substracted**
  - **Result type is int**
  - **Result value = no. Of elements of sizeof(type) between two pointers**
- **Logically derives from adding/substracting int to pointers**
  - **p = q + 2 ==> p – q = 2**

Operations on pointers: [ ]

```
int *p, a[4], *q;
p =  &a[1];
p[2] = 20;
p[-1] = 10;
```

- **Interestingly, C allows [ ] notation to be applied to all pointers!**
  - **You must be knowing that [ ] is normally used for arrays**
- **p [i] means * (p + i)**
  - **P is a pointer and i is an integer (or i is a pointer and p is a pointer is also allowed)**

A peculiar thing about arrays

- **Name of an array is equivalent to the address of (the zeroeth element) the array**

  int a[3];

  Now

  a* means *&a[0]

- **Because it's an address, it can be stored in a pointer**

  int a[3], *p;

  p = a;

- **What do the following mean?**

  Int a[3] = {1, 2, 3}, *p;

  a + 1;

  ***(a + 1);**

  p = (a + 2)

- **Exception: when passed to sizeof() operator, the name is not the address**

Pointer *as if it was an array*

- **Combine the concepts of**
  - **Pointer arithmetic (+- int)**
  - **[ ] notation for pointers**
  - **Array name as address of array**

```
int a[3], *p;
p = a;
p[2] = 10;
```

- **Here we are using p as if it was an array name**
  - **Possible only if p was pointing to array base**

Pointers != Arrays

- Array is a continuous collection of elements of one type.
- Array has name, the name also stands for the address of the array
- [ ] is allowed operation on arrays
- Array name can't be reassigned to another address

- Pointer is a variable that can store an address
- Pointers can be of various types
- [ ], = , +- int, substraction are operations on pointers
- Pointers can be reassigned to point to different addresses

Arrays as arguments to functions

```
char f(char *c) {
    return c[0];
}
int main() {
    char arr[16], x;
    x = f(arr);
}
```

- Here actual argument is ‘arr’
- Name of array is address of array, that is &arr[0]
  - Address of char
- So formal argument is char *

2-d array as argument to function

```
char f(char **c) {
return c[0][2];
}
int main() {
char arr[16][6], x;
x = f(arr);
}
```

```
char f(char c[][6]) {
 return c[0][2];
}
int main() {
 char arr[16][6], x;
 x = f(arr);
}
```

Which one is correct?
You can answer by just applying the rules we learnt

2-d array as argument to function

```
char f(char (*c)[6]) {
return c[0][2];
}
int main() {
char arr[16][6], x;
x = f(arr);
}
```

```
char f(char c[][6]) {
 return c[0][2];
}
int main() {
 char arr[16][6], x;
 x = f(arr);
}
```

Which one is correct?
(The second one is same as on the earlier slide)
You can answer by just applying the rules we learnt

Dynamic Memory Allocation

Concept of (Binding) “Time”

- **Compile Time**
  - **When you are running commands like**

    cc program.c -o program
- **Load time**
  - **After you type commands like**

    ./program

    Before the main() starts running
- **Program Run Time**
  - **After you type commands like**

    ./program

    When the main() of the program starts running, till it exits
- **Function Call Time**
  - **Interval between the call of a function and before the called function starts running. Part of “Run Time”.**

Lifetime of variables and Memory Allocation

- **Global Variables, Static Variables (g, t, and s here)**
  - **Allocated Memory at *load time***
  - **They are alive (available) till the program exits**
- **Local Variables, Formal Arguments (i, j, k in main; a, b, x, y in f)**
  - **Allocated Memory on *function call***
  - **They are alive (available) as long as function is running**
- **Dynamically allocated memory (Run time allocation)**
  - **Allocated on explicit call to functions like *malloc()***
  - **Alive as long as functions like *free()* are not called on the memory**

```
int g;
static int t = 20;
int f(int a, int b) {
    int x, y;
    static int s = 10;
    x = a + b + 5 + g + s;
    return x;
}
int main() {
    int i, j, k, *p;
    g = 10;
    i = 20; j = 30;
    p = malloc(8);
    k = f(i, j);
}
```

Man pages

Understanding library functions:

Red the manual pages, using ‘man’ command

> man sqrt

> man 3 printf

> man 3 open

 # see section 3 of man pages for C library functions

malloc()

- malloc() is a standard C library function for allocating memory dynamically (at run time)
- #include <stdlib.h> for malloc()
- Function prototype
  - Run “man malloc” to see it

    ```
    void *malloc(size_t size);
    ```

  - size_t is a typedef in stdlib.h
    - size_t is unsigned long
  - Reads a number, allocates those many bytes and returns the address of allocated memory (zero'th byte)
- Additional info: malloc() gets the memory from the OS and then gives to your program

void *

- **A void pointer is a typeless pointer;**
  - **Pure address**
  - **No type --> No "size" information about the type**
- **You can declare a void pointer**

  void *p, *q
- **Void pointer can store *any* address**

  void *p; int a; p = &a; char c; p = &c;
- **Void pointers can be copied**

  void *p, *q; int a; p = &a; q = p;
  - **q also stores address of a now**
- **The Dereferencing operator has no meaning when applied to a void pointer**

  void *p; int a; p = &a;
  - **What does *p mean now? * Needs "size" of the type for its work. Void pointers have no type and so no size information associated with it.**
  - **Note: [ ] is also dereferencing**

malloc()

- malloc() returns a “void *”
  - Returns a pure address
  - This address can be stored in a “void *” variable
  - This address can also be stored in any pointer variable
- Suppose we do
  - void *p; p = malloc(8);
  - Now what meaningful operations can we do with the malloced memory? --> only copy!
  - So normally return value of malloc is stored using some typed pointer

malloc()

```
int *p;
p = malloc(8)
```

- This code allocates 8 bytes and then pointer p will point to the malloced memory
- This code can result in a “warning” because we are converting “void *” to a “int *” with '='

malloc()

```
int *p;
p  = (int *) malloc(8);
```

- This code does away with the warning as we are converting the “void *” into “int *” using explicit type casting
- Suppose size of integer was 4 bytes, then what does this code mean?

malloc()

```
int *p;
p  = (int *) malloc(8);
```

- p points to 8 byte location
- However now, *p means deferencing in “4 bytes”
  - as size of int is 4 bytes

- p[0] means *(p + 0) that is *p
- p[1] means *(p + 1) where (p + 1) is a pointer 4 bytes ahead of p
  - Using this trick we are treating the 8 bytes as if it was a 2 integer array !

malloc()

```
int *p;
p  = (int *) malloc(8);
```

- p points to 8 byte location
- However now, *p means deferencing in “4 bytes”
  - as size of int is 4 bytes

- p[0] means *(p + 0) that is *p
- p[1] means *(p + 1) where (p + 1) is a pointer 4 bytes ahead of p
  - Using this trick we are treating the 8 bytes as if it was a 2 integer array !

Malloc(): Allocating arrays dynamically

```
int *p;
p  = (int *) malloc(
        sizeof(int) * 4);
```

- We can allocate arrays of any type dynamically using malloc()
- Use of sizeof(int) here makes sure that the code is *portable,* appropriately sized array will be allocated irrespective of size of integer
  - Code on earlier slide assumes 4 byte integer
- This code allocates array of 4 integers
  - Can be acessed as p[0], p[1], p[2] and p[3]

malloc(): Allocating array of structures

```
typedef struct test {
        int a, b;
        double g;
}test;
test *p;
p  = (test *) malloc(
        sizeof(test) * 4);
```

- This code allocates an array of 4 structures
- p points to the array of structures
- p[0] is the 0th structure, p[1] is the 1st structure ...
- p[0].a, p[1].g is the way to access the inner elements of structures

Homework

- **Write code using malloc() to**
  - **Allocate an array of 10 doubles**
  - **Read n from user and allocate an array of n shorts**
- **What does the following code mean?**

  int *p; p = malloc(9); // suppose sizeof(int) is 4

  Note: No need to write this sort of code in this course!
- **What does the following code mean?**

  int *ip; Char *p; p = malloc(8); ip = p; p[0] = 10;

  Note: again, no need to write code like this in this course!

free()

- free() will give the malloced memory back
- malloc() and free() work together to manage what is called as “heap memory” which the memory management library has obtained from the OS
- Usage

```
void free(void *p);
```

- free() must be given an address which was returned by malloc()
- Rule: Every malloc() must have a corresponding free()

Pointer to Pointer

```
int **pp, *p, p;
pp = (int **) malloc(sizeof(int *) * 4);
pp[1] = (int *)malloc(sizeof(int) * 3);
```

- A pointer to pointer, is essentially a pointer!
  - Can store an address, has a type
  - The type is “pointer to pointer” so all * or [ ] operations work with sizeof(pointer) which is typically 4 bytes
- The code above allocates an array of pointers, and then allocates an array of integers.
  -

2-d array with pointer to pointer

```
#include <stdlib.h>
#define N 4
int main() {
    int **pp, i, j;
    pp = (int **) malloc
                (sizeof(int *) * N) ;
    for(i = 0; i < N; i++)
        pp[i] = (int *)
                malloc(sizeof(int) * N);
    for(i = 0; i < N; i++)
        for(j = 0; j < N; j++)
            pp[i][j] = i + j;
    return 0;
}
```

2-d array with pointer to pointer

```
#include <stdlib.h>
#define N 4
int main() {
    int **pp, i, j;
    pp = (int **) malloc
                (sizeof(int *) * N) ;
    for(i = 0; i < N; i++)
        pp[i] = (int *)
                malloc(sizeof(int) * N);
    for(i = 0; i < N; i++)
        for(j = 0; j < N; j++)
            pp[i][j] = i + j;
    return 0;
}
```

- How does pp[i][j] work here?
- pp[i] is *(pp + i) which is the pointer at the i'th location in the array of pointers allocated
  - Here * works with *size* = *sizeof a pointer*
- pp[i][j] is *(pp[i] + j)
  - Since pp[i] is a integer pointer, so here * works with sizeof(int)
  - pp[i] is the address of array (indicated in red), so pp[i] + j is pointer to the j'th element of that array

Self Referential Pointers in Structures

Self Referential Pointers

- "Self Referential Pointer" is a kind of a misnomer
- C allows structures like this

```
struct test {
  int a;
  struct test *p;
};
```

- The pointer p, can point to any variable of the type "struct test" (or be NULL)

Self Referential Pointers

- Consider following code

```
typedef struct test {
    int a;
    struct test *p;
}test;
test  x, y, *p, *q;
p = &x;
x.p = &y;
q = &y;
y.p = &y;
```

Self referential structures allow us to create a variety of "linked" *structures of data*

-> notation

```
typedef struct test {
    int a;
    struct test *p
}test;
test  m, n, *x;
m.a = 20;
x = &n;
(*x).a = 40;
x->a = 50;
x->p = &m;
x->p->p = x;
```

- (*x) is the entire structure to which x points
  - (*x).a is the variable 'a' in that structure
- x->a is another notation for (*x).a

2 Common Problems involving Pointers:

(Dereferencing) Dangling Pointers
And
Garbage Memory

NULL

- NULL is a symbolic constant defined in stdio.h
  - #define NULL 0
- This is a special pointer value, defined by C language
  - The number 0 is not the same as the address 0 ! They are different types !
- It is guaranteed that a pointer is NOT NULL, unless programmer sets it to NULL

A side note on NULL

- **NULL is not the number 0**
- **NULL is not necessarily the address 0**
- **NULL is just a special value for pointers told to us by C language.**
  - **Very often we need special values for a certain type**
  - **E.g. the value '\0' for a character is universally taken to be a special value indicating end of a character sequence in an array**
  - **INT_MAX , INT_MIN are values #defined in limits.h for integers**
  - **These type of values are used in programs to indicate either an *unused variable* or *empty variable* or *error value* on that variable**
- **Int i = NULL; char c = NULL; works**
  - **Why do you want to do it? Instead of saying int i = 0; char c = 0;**

Dangling Pointers

- Total 3 Possibilities for a pointer value
  - Points to memory owned by program
    - Local variables, Global Variables, Malloced Memory
  - = NULL
  - Dangling
    - Some texts differentiate between ‘wild’ (uninitialized) and ‘dangling’ pointers
- Dereferencing (that is *) a dangling (or NULL) pointer is NOT to be done
  - Even if you find that dereferencing a dangling pointer “*does not create problem”* in your code, it is still not to be done!
  - Result: The OS punishes your code by terminating it, and saying “Segmentation Fault”

Dangling Pointers

- A pointer is dangling when declared & not initialized
- A pointer becomes non dangling, when assigned to a “good” memory location like local variables, global variables, malloc-ed memory
- A pointer can become dangling again due to
  - Mistakes in pointer arithmetic
  - Mistakes in manipulation of dynamically allocated memory, e.g. Linked list pointers
  - On deallocation of malloced memory using free()
  - etc

```
int main() {
        int *p, i, j, *q;// p,q are dangling

}
```

Dangling Pointers Example: 1

```
int main() {
        int *p, i, j, *q;// p,q are dangling
        p = &i; // p not dangling

}
```

Dangling Pointers Example: 1

```
int main() {
        int *p, i, j, *q;// p,q are dangling
        p = &i; // p not dangling
        i = 10; // *p = 10

}
```

Dangling Pointers Example: 1

```
int main() {
        int *p, i, j, *q;// p,q are dangling
        p = &i; // p not dangling
        i = 10; // *p = 10
        q = &j; // q not dangling

}
```

Dangling Pointers Example: 1

```
int main() {
        int *p, i, j, *q;// p,q are dangling
        p = &i; // p not dangling
        i = 10; // *p = 10
        q = &j; // q not dangling
        q = (int *)malloc(sizeof(int) * 4);
                        // q not dangling

}
```

Dangling Pointers Example: 1

```
int main() {
        int *p, i, j, *q;// p,q are dangling
        p = &i; // p not dangling
        i = 10; // *p = 10
        q = &j; // q not dangling
        q = (int *)malloc(sizeof(int) * 4);
                                // q not dangling
        q = q + 3; // q not dangling
       }
```

Dangling Pointers Example: 1

```
int main() {
        int *p, i, j, *q;// p,q are dangling
        p = &i; // p not dangling
        i = 10; // *p = 10
        q = &j; // q not dangling
        q = (int *)malloc(sizeof(int) * 4);
                                // q not dangling
        q = q + 3; // q not dangling
        q = q + 1; // q IS dangling now

}
```

Dangling Pointers Example: 1

```
int main() {
        int *p, i, j, *q;// p,q are dangling
        p = &i; // p not dangling
        i = 10; // *p = 10
        q = &j; // q not dangling
        q = (int *)malloc(sizeof(int) * 4);
                                // q not dangling
        q = q + 3; // q not dangling
        q = q + 1; // q IS dangling now
        p = q - 6; // p IS also dangling
}
```

Dangling Pointers Example: 1

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
}
```

Dangling Pointers Example: 2

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
        a.val = 20;
}
```

Dangling Pointers Example: 2

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
        a.val = 20;
        p = &a;

}
```

Dangling Pointers Example: 2

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
        a.val = 20;
        p = &a;
        a.next = (node *)malloc(sizeof(node));

}
```

Dangling Pointers Example: 2

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
        a.val = 20;
        p = &a;
        a.next = (node *)malloc(sizeof(node));
        a.next->val = 50;

}
```

Dangling Pointers Example: 2

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
        a.val = 20;
        p = &a;
        a.next = (node *)malloc(sizeof(node));
        a.next->val = 50;
        q = a.next;

}
```

Dangling Pointers Example: 2

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
        a.val = 20;
        p = &a;
        a.next = (node *)malloc(sizeof(node));
        a.next->val = 50;
        q = a.next;
        free(a.next);

}
```

Dangling Pointers Example: 2

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
        a.val = 20;
        p = &a;
        a.next = (node *)malloc(sizeof(node));
        a.next->val = 50;
        q = a.next;
        free(a.next);
        q->val = 100; // segfault

}
```

Dangling Pointers Example: 2

Remember

- Dereferencing a dangling pointer is a cause of segmentation fault
- Dereferencing can occur using * or [ ]
- Having a dangling pointer does not cause segmentation fault
  - Dereferencing it causes
  - Having dangling pointers is not wrong, but why have them?

Garbage Memory

```
int *p, i;
p = malloc
        (sizeof(int) * 4);
p = &i;
```

- p was pointing to malloced memory
- In the end p points to i
- How to access the malloced memory now?
  - No way! It's Lost!
  - It became “garbage memory”
- Problems like this result in memory waste
  - Also called as Memory leak
- Solution
  - Keep another point pointing to malloced memory

    ```
    p = malloc(sizeof(int) * 4);
    q  = p;
    p = &i;
    ```

  - Malloced memory is available through pointer q

Segmentation Fault

What is segmentation fault?

- A program accessing an “illegal” memory location
  - Is punished by the operating system for doing so
  - The program gets “terminated” (killed)
  - “segmentation” comes from “memory management” concepts in operating systems
- It is ALWAYS a fault of the programmer!
- Beware
  - Bad compilers may generate machine code which hide some memory violations
  - If a programmer does illegal memory access, segmentation fault may not occur sometimes!
    - OS may “forgive” your program :-;

Some Typical Reasons for Seg-fault

Deferencing Dangling Pointers

Array Index Violation

Incorrect Function Arguments

Dangling Pointer Dereference: Some examples

```
int *p, i;
*p = 20;
```

```
int *p, i;
p = malloc(
        sizeof(int)*2);
free(p);
p[2] = 20;
```

```
int *p, i;
p = &i;
p[1] = 20;
```

```
int *f(int *p) {
    int x = *p + 2;
    return &x;
}
int main() {
    int i, *q;
    scanf("%d", &i);
    q = f(&i);
    *q = 20;
}
```

Don't return address of a local variable! The variable disappears after function call, and the returned address is a dangling pointer.

Array Index Violation

Valid indices for an array of size n: 0 .. n-1

Accessing any index <=-1 or >=n *may* result in seg-fault (it may not result in some cases, but it is STILL WRONG to do it!)

Try this code:

```
#include <stdio.h>
int main() {
    int a[16], i = 0;
    while(1) {
        a[i] = i;
        printf("%d\n", a[i]);
        i++;
    }
}
```

At what value of i does the code seg-fault? Try 2-3 times.

Reason: OS may allocate more than 16-integer size memory for the program. So the seg fault *may not* occur at index 16 or slightly higher indices also. Rule: wrong to access index >=n

Functions: Pointer Arguments

- Rule: When you want a function to change ACTUAL arguments
  - Function takes pointer arguments
  - Call passes address of “actual argument”
- Example: Swap function (correct code)

```
void swap(int *a, int *b ) {
        int temp;
        temp = *a;
        *a = * b;
        *b = temp;
}
int main() {
    int m = 20, n = 30;
    swap(&m, &n);
```

Incorrect Pointer Aruments - Segfault

```
int i;

scanf("%d", i);
```

Scanf tries to do something like

*i = value // segfault here

Segfault occurs in scanf, although the reason is call to scanf

- Note that this is basically a dangling pointer dereference which took place inside scanf()

Guidelines to avoid segfaults

- Always initialize pointers to NULL
  - This will not avoid segfaults, but may lead to early detection of a problem
- While accessing arrays, make sure that array index is valid
- Never return address of local variable of a function
- DO NOT IGNORE compiler's warnings
  - Compilers often warn about potential dangling references, type conversions which have a potential for segfaults
  - Make sure that you rewrite code to remove all warnings
  - Use “-Wall” option with gcc. > cc -Wall program.c -o program

Let’s draw diagrams for some programs using pointers, malloc, free, ...

Introduction to Linux Desktop
and
Command Line
11 Jan 2022

Abhijit A. M.
abhijit.comp@coep.ac.in

Why GNU/Linux ?

Why GNU/Linux ?

1. Programmer's Paradise : most versatile, vast, all pervasive programming environment

2. Free Software (or Open Source?) : Free as in freedom. *Freely* Use, copy, modify, redistribute.

3. Highly Productive : Do more in less time

4. Better quality, more secure, very few crashes

Why Command Line ?

1. Not for everyone ! Absolutely !

2. Those who do it are way more productive than others

3. Makes you think !

4. Portable. Learn once, use everywhere on all Unixes, Linuxes, etc.

Few Key Concepts

- *Files* don't open themselves
  - Always some application/program open()s a file.
- Files don't display themselves
  - A file is displayed by the program which opens it. Each program has it's own way of handling ifles

Few Key Concepts

- **Programs don't run themselves**
  - **You click on a program, or run a command --> equivalent to request to Operating System to run it. The OS runs your program**
- **Users (humans) request OS to run programs, using Graphical or Command line interface**
  - **and programs open files**

Path names

- Tree like directory structure
- Root directory called /
- A complete path name for a file
  - /home/student/a.c
- Relative path names

  concept: every running program has a *current* working directory

  . current directory

  .. parent directory

  ./Desktop/xyz/../p.c

A command

- Name of an executable file
  - For example: 'ls' is actually “/bin/ls”
- Command takes arguments
  - E.g. ls /tmp/
- Command takes options
  - E.g. ls -a

A command

- Command can take both arguments and options
  - E.g. ls -a /tmp/
- Options and arguments are basically argv[] of the main() of that program

Basic Navigation Commands

- pwd
- ls
  - ls -l
  - ls -l /tmp/
  - ls -l /home/student/Desktop
  - ls -l ./Desktop
  - ls -a

    \ls -F
- cd
  - cd /tmp/
  - cd
  - cd /home/student/Desktop
- notation: ~
  - cd ~
  - cd ~/Desktop
  - ls ~/Desktop

Map these commands to navigation using a graphical file browser

The Shell

- **Shell = Cover**
- **Covers some of the Operating System's "System Calls" (mainly fork+exec) for the *Applications***
- **Talks with Users and Applications and does some talk with OS**

Not a very accurate diagram !

http://code.snipcademy.com/tutorials/linux-command-line/environment/

The Shell

Shell waits for user's input

Requests the OS to run a program which the user has asked to run

Again waits for user's input

GUI is a Shell !

Let's Understand fork() and exec()

```
#include <unistd.h>

int main() {

        fork();

        printf("hi\n");

        return 0;

}
```

```
#include <unistd.h>

int main() {

        printf("hi\n");

        execl("/bin/ls", "ls",
NULL);

        printf("bye\n");

        return 0;

}
```

A simple shell

```
#include <stdio.h>
#include <unistd.h>
int main() {
        char string[128];
        int pid;
        while(1) {
                printf("prompt>");
                scanf("%s", string);
                pid = fork();
                if(pid == 0) {
                        execl(string, string, NULL);
                } else {
                        wait(0);
                }
        }
```

File Permissions on Linux

- Two types of users
  - root and non-root
  - Users can grouped into 'groups'
- 3 sets of 3 permission
  - Octal notation
  - Read = 4, Write = 2, Execute = 1
  - 644 means
    - Read-Write for owner, Read for Group, Read for others
- chmod command uses these notations
-

File Permissions on Linux

```
-rw-r--r-- 1 abhijit abhijit 1183744 May 16 12:48 01_linux_basics.ppt
-rw-r--r-- 1 abhijit abhijit  341736 May 17 10:39 Debian Family Tree.svg
drwxr-xr-x 2 abhijit abhijit    4096 May 17 11:16 fork-exec
-rw-r--r-- 1 abhijit abhijit 7831341 May 11 12:13 foss.odp
```

3 sets of 3 permissions

3 sets = user (owner), group, others

3 permissions = read, write, execute

hard link count

Owner

size

name

last-modification

File Permissions on Linux

- r on a file : can read the file
  - open(.... O_RDONLY) works
- w on a file: can modify the file
- x on a file: can ask the os to run the file as an executable program
- r on a directory: can do 'ls'
- w on a directory: can add/remove files from that directory (even without 'r'!)
- x on a directory: can 'cd' to that directory

Access rights examples

- **-rw-r--r--**
  Readable and writable for file owner, only readable for others

- **-rw-r-----**
  Readable and writable for file owner, only readable for users belonging to the file group.

- **drwx------**
  Directory only accessible by its owner

- **-------r-x**
  File executable by others but neither by your friends nor by yourself. Nice protections for a trap...

Man Pages

- Manpage
  - $ man ls
  - $ man 2 mkdir
  - $ man man
  - $ man -k mkdir
- Manpage sections
  - 1 User-level cmds and apps
    - /bin/mkdir
  - 2 System calls
    - int mkdir(const char *, …);
  - 3 Library calls
    - int printf(const char *, …);
  - 4 Device drivers and network protocols
    - /dev/tty
  - 5 Standard file formats
    - /etc/hosts
  - 6 Games and demos
    - /usr/games/fortune
  - 7 Misc. files and docs
    - man 7 locale
  - 8 System admin. Cmds
    - /sbin/reboot

GNU / Linux filesystem structure

Not imposed by the system. Can vary from one system to the other, even between two GNU/Linux installations!

/	Root directory
/bin/	Basic, essential system commands
/boot/	Kernel images, initrd and configuration files
/dev/	Files representing devices
	/dev/hda: first IDE hard disk
/etc/	System and application configuration files
/home/	User directories
/lib/	Basic system shared libraries

GNU / Linux filesystem structure

/lost+found	Corrupt files the system tried to recover
/media	Mount points for removable media:
	/media/usbdisk, /media/cdrom
/mnt/	Mount points for temporarily mounted filesystems
/opt/	Specific tools installed by the sysadmin
	/usr/local/ often used instead
/proc/	Access to system information
	/proc/cpuinfo, /**proc/version** …
/root/	root user home directory
/sbin/	Administrator-only commands
/sys/	System and device controls
	(cpu frequency, device power, etc.)

GNU / Linux filesystem structure

/tmp/	Temporary files
/usr/	Regular user tools (not essential to the system) **/usr/bin/**, **/usr/lib/**, **/usr/sbin**…
/usr/local/	Specific software installed by the sysadmin (often preferred to **/opt/**)
/var/	Data used by the system or system servers **/var/log/**, **/var/spool/mail** (incoming mail), **/var/spool/lpd** (print jobs)…

Files: cut, copy, paste, remove,

- cat <filenames>
  - cat /etc/passwd
  - cat fork.c
  - cat <filename1> <filename2>
- cp <source> <target>
  - cp a.c b.c
  - cp a.c /tmp/
  - cp a.c /tmp/b.c
  - cp -r ./folder1 /tmp/
  - cp -r ./folder1 /tmp/folder2
- mv <source> <target>
  - mv a.c b.c
  - mv a.c /tmp/
  - mv a.c /tmp/b.c
- rm <filename>
  - rm a.c
  - rm a.c b.c c.c
  - rm -r /tmp/a
- mkdir
  - mkdir /tmp/a /tmp/b
- rmdir
  - rmdir /tmp/a /tmp/b

Useful Commands

- echo
  - echo hi
  - echo hi there
  - echo “hi there”
  - j=5; echo $j
- sort
  - sort
  - sort < /etc/passwd
- firefox
- libreoffice
- grep
  - grep bash /etc/passwd
  - grep -i display /etc/passwd
  - egrep -i 'a|b' /etc/passwd
- less <filename>
- head <filename>
  - head -5 <filename>
  - tail -10 <filename>

Useful Commands

- **alias**

  alias ll='ls -l'

- **tar**

  tar cvf folder.tar folder

- **gzip**

  gzip a.c

- **touch**

  touch xy.txt

  touch a.c

- **strings**

  strings a.out

- **adduser**

  sudo adduser test

- **su**

  su administrator

Useful Commands

- **df**

  df -h

- **du**

  du -hs .

- **bc**
- **time**
- **date**
- **diff**
- **wc**

Network Related Commands

- ifconfig
- ssh
- scp
- telnet
- ping
- w
- last
- whoami

Unix job control

- **Start a background process:**
  - **gedit a.c &**
  - **gedit**
    ***hit* ctrl-z**
    bg
- **Where did it go?**
  - **jobs**
  - **ps**
- **Terminate the job: kill it**
  - **kill %*jobid***
  - **kill *pid***
- **Bring it back into the foreground**
  - **fg %1**

Shell Wildcards

- ? (question mark)
  - Any ene character
  - ls a?c
  - ls ??c
- *
  - any number of characters
  - ls *
  - ls d*
  - echo *
- [ ]
  - Matches a range
  - ls a[1-3].c
- {}
  - ls pic[1-3].{txt,jpg}

Configuration Files

- Most applications have configuration files in TEXT format
- Most of them are in */etc*
- */etc/*passwd and /*etc*/shadow
  - Text files containing user accounts
- */etc/*resolv.conf
  - DNS configuration
- */etc/network/interfaces*
  - *Network configuration*
- */etc/*hosts
  - Local database of Hostname-IP mappings
- */etc/*apache2/apache2.conf
  - Apache webserver configuration

~/.bashrc file

- **~/.bashrc**

  Shell script read each time a **bash** shell is started
- You can use this file to define
  - Your default environment variables (**PATH**, **EDITOR**...).
  - Your aliases.
  - Your prompt (see the **bash** manual for details).
  - A greeting message.
- Also ~/.bash_history

Special devices (1)

Device files with a special behavior or contents

- **/dev/null**
  The data sink! Discards all data written to this file.
  Useful to get rid of unwanted output, typically log information:
  mplayer black_adder_4th.avi &> /dev/null

- **/dev/zero**
  Reads from this file always return **\0** characters
  Useful to create a file filled with zeros:
  dd if=/dev/zero of=disk.img bs=1k count=2048

See **man null** or **man zero** for details

Special devices (2)

- **/dev/random**
  Returns random bytes when read. Mainly used by cryptographic programs. Uses interrupts from some device drivers as sources of true randomness ("entropy").
  Reads can be blocked until enough entropy is gathered.

- **/dev/urandom**
  For programs for which pseudo random numbers are fine.
  Always generates random bytes, even if not enough entropy is available (in which case it is possible, though still difficult, to predict future byte sequences from past ones).

See **man random** for details.

Special devices (3)

- **/dev/full**
  Mimics a full device.
  Useful to check that your application properly handles this kind of situation.

See **man full** for details.

Files names and inodes

Hard Links Vs Soft Links

Shell Programming

Shell Programming

- is “programming”
- Any programming: Use existing tool to create new utilities
- Shell Programming: Use shell facilities to create better utilities
  - Know “commands” --> More tools
  - Know how to combine them

Shell Variables

- Shell supports variables
  - Try:
    - j=5; echo $j
    - No space in j=5
  - Try
    - set
    - Shows all set variables
  - Try
    - a=10; b=20; echo ab
  - What did you learn?
  - All variables are strings

Shell's predefined variables

- **USER**
  - Name of current user
- **HOME**
  - Home directory of current $USER
- **PS1**
  - The prompt
- **LINES**
  - No. of lines of the screen
-
- **HOSTNAME**
  - Name of the computer
- **OLDPWD**
  - Previous working directory
- **PATH**
  - List of locations to search for a command's executable file
- **$?**
  - Return value of previou command

Redirection

- cmd > filename
  - Redirects the output to a file
  - Try:
    - ls > /tmp/xyz
      cat /tmp/xyz
    - echo hi > /tmp/abc
      cat /tmp/abc
- cmd < filename
  - Reads the input from a file instead of keyboard
  - Think of a command now!

Pipes

- Try
  - last | less
  - grep bash /etc/passwd | head – 1
  - grep bash /etc/passwd | head – 2 | tail -1
- Connect the output of LHS command as input of RHS command
- Concept of *filters* – programs which read input only from stdin (keyboard, e.g. scanf, getchar), and write output to *stdout* (e.g. printf, putchar)
- Programs can be connected using pipes if they are filters
- Most Unix/Linux commands are filters !

The *test* command

- test
  - test 10 -eq 10
  - test “10” == “10”
  - test 10 -eq 9
  - test 10 -gt 9
  - test “10” >= “9”
  - test -f /etc/passwd
  - test -d ~/desktop
  - ...

Shortcut notation for calling test

[ ]

[10 -eq 10]

Note the space after '[' and before ']'

The *expr* command and backticks

- expr
  - expr 1 + 2
  - a=2; expr $a + 2
  - a=2; b=3; expr $a + $b
  - a=2;b=3; expr $a * $b
  - a=2;b=3; expr $a | $b
- Used for mathematical calculations

- backticks ``
  - j=`ls`; echo $j
  - j=`expr 1 + 2`; echo $j
-

if then else

```
if [ $a -lt $b ]
then
    echo $a
else
    echo $b
fi
```

```
if [ $a -lt $b ];then
echo $a; else  echo $b;
fi
```

0 TRUE

Nonzero FALSE

while do done

```
while [ $a -lt $b ]
do
    echo $a
    a=`expr $a + 1`
done
```

```
while [ $a -lt $b ]; do
echo $a;   a=`expr $a +
1`; done
```

```
while [ $a -lt $b ]
do
    echo $a
    a=$(($a + 1))
done
```

for x in ... do done

```
for i in {1..10}
do
    echo $i
done

for i in *
do
    echo $i
done
```

```
for i in *; do   echo $i;
done
```

case ... esac

```
read space
case $space in
      [1-6]*)
        Message="one to 6"
        ;;
      [7-8]*)
        Message="7 or 8"
        ;;
      9[1-8])
       Message="9 with a number"
        ;;
      *)
       Message="Default"
        ;;
esac
echo $Message
```

Syntax

```
;;
)  after option
* for default
esac
```

Try these things

- Print 3rd line from /etc/passwd, which contains the word bash
- Print numbers from 1 to 1000
- Create files named like this: file1, file2, file3, ... filen where n is read from user
  - Read i%5th file from /etc/passwd and store it in file*i*
- Find all files ending in .c or .h and create a .tar.gz file of these files
-

The Golden Mantra

Everything can be done from command line !

Command line is far more powerful than graphical interface

Command line makes you a better programmer

Mounting

Partitions

Disk Management

File Action View Help

Volume	Layout	Type	File System	Status	Capacity	Free Space	% Fr
(E:)	Partition	Basic	FAT32	Healthy	9.76 GB	8.37 GB	85 %
(F:)	Partition	Basic	FAT32	Healthy	9.76 GB	7.24 GB	74 %
OLDDRIVE (H:)	Partition	Basic	FAT32	Healthy (A...	4.99 GB	586 MB	11 %
WINDOWS XP (C:)	Partition	Basic	FAT32	Healthy (S...	9.76 GB	2.61 GB	26 %
Windows Vista (G:)	Partition	Basic	NTFS	Healthy	8.00 GB	1.61 GB	20 %
XPBACKUP (I:)	Partition	Basic	FAT32	Healthy	4.99 GB	4.33 GB	86 %
XXCOPY (J:)	Partition	Basic	FAT32	Healthy	9.00 GB	4.32 GB	47 %

Disk 0
Basic
37.30 GB
Online

- **WINDOWS XP (C:)** 9.77 GB FAT32 Healthy (System)
- **(E:)** 9.77 GB FAT32 Healthy
- **(F:)** 9.77 GB FAT32 Healthy
- **Windows Vista (G:)** 8.00 GB NTFS Healthy

Disk 1
Basic
19.01 GB
Online

- **OLDDRIVE (H:)** 5.00 GB FAT32 Healthy (Active)
- **XPBACKUP (I:)** 5.00 GB FAT32 Healthy
- **XXCOPY (J:)** 9.01 GB FAT32 Healthy

Primary partition | Extended partition | Logical drive

Windows Namespace

c:\temp\songs\xyz.mp3

- Root is C:\ or D:\ etc
- Separator is also "\"

Windows Namespace

- C:\ D:\ Are partitions of the disk drive
- Typical convention: C: contains programs, D: contains data

- One “tree” per partition
  - Together they make a “forest”

Linux Namespace: On a partition

/usr/songs/xyz.mp3

- On every partition:
  - Root is “/”
  - Separator is also “/”

Linux namespace: Mount

- Linux namespace is a single “tree” and not a “forest” like Windows
- Combining of multiple trees is done through “mount”

Linux namespace
Mounting a partition

Mounting another partition on */home/guest/mydir/* directory

Mounted disk partition

/home/guest/mydir/x/b.cpp → way to access the file on the other disk partition

Let's go for a live installation Demo !

Some Shell Gimmics

Terminal Tricks

Ctrl + n : same as Down arrow.

Ctrl + p : same as Up arrow.

Ctrl + r : begins a backward search through command history.(keep pressing Ctrl + r to move backward)

Ctrl + s : to stop output to terminal.

Ctrl + q : to resume output to terminal after Ctrl + s.

Terminal Tricks

Ctrl + a : move to the beginning of line.

Ctrl + e : move to the end of line.

Ctrl + d : if you've type something, Ctrl + d deletes the character under the cursor, else, it escapes the current shell.

Ctrl + k : delete all text from the cursor to the end of line.

Ctrl + t : transpose the character before the cursor with the one under the cursor

Terminal Tricks

Ctrl + w : cut the word before the cursor; then Ctrl + y paste it

Ctrl + u : cut the line before the cursor; then Ctrl + y paste it

Ctrl + _ : undo typing.

Ctrl + l : equivalent to clear.

Ctrl + x + Ctrl + e : launch editor defined by $EDITOR to input your command.

Run from history

First: What's history?

Ans: Run 'history

$ history

$!53

$!!

$!cat

$!c

Math

```
$ echo $(( 10 + 5 ))  #15

$ x=1

$ echo $(( x++ )) #1 , notice that it is still 1, since it's post-incremen

$ echo $(( x++ )) #2
```

More Math

```
$ seq 10|paste -sd+|bc
# How does that work ?
```

Using expr

```
$ expr 10+20 #30
$ expr 10\*20 #600
$ expr 30 \> 20 #1
```

More Math

Using bc

$ bc

```
obase=16
ibase=16
AA+1
AB
```

More Math

Using bc

$ bc

```
ibase=16
obase=16
AA+1
07 17
# what went wrong?
```

Fun with grep

```
$ grep -Eo '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}'
# above will only search for IP addresses!
$ grep -v bbo filename
$ grep -w ‘abhijit’ /etc/passwd
$ grep -v '^#' file.txt
$ grep -v ^$  file.txt
```

xargs: convert stdin to args

$ find . | grep something | xargs rm

xargs is highly powerful

rsync

The magic tool to sync your folders!

$ rsync -rvupt ~/myfiles /media/abhijit/PD

$ rsync -rvupt --delete ~/myfiles /media/abhijit/PD

find

```
$ find .
$ find . -type f
$ find . -type d
$ find . -name '*.php'
$ find / -type f -size +4G
$ find . -type f -empty -delete
$ find . -type f | wc -l
```

Download : wget, curl

$ wget foss.coep.org.in

$ wget –r foss.coep.org.in

$ wget -r –convert-links foss.coep.org.in

$ wget -r –convert-links --no-parent foss.coep.org.in/fossmeet/

$ curl https://raw.githubusercontent.com/onceupon/Bash-Oneliner/master/README.md | pandoc -f markdown -t man | man -l -

 # curl is more powerful than wget. Curl can upload. Curl supports many protocols, wget only HTTP/FTP.

Random data

```
# shuffle numbers from 0-100, then pick 15 of them randomly
$ shuf -i 0-100 -n 15

# Random pick 100 lines from a file
$ shuf -n 100 filename

#generate 5 password each of length 13
$ pwgen 13 5

echo $((RANDOM % 10))
```

Run commands remotely

$ ssh administrator@foss.coep.org.in

$ ssh -X administrator@foss.coep.org.in

$ ssh -X administrator@foss.coep.org.in firefox

System Information

Show memory usage,. **# print 10 times, at 1 second interval**

$ free -c 10 -mhs 1

 # Display CPU and IO statistics for devices and partitions.
 # refresh every second

$ iostat -x -t 1

 # Display bandwidth usage on an network interface (e.g. enp175s0f0)

$ sudo iftop -i wlo1

 # Tell how long the system has been running and number of users

$ uptime

Surf the web

$ w3m

$ links

Add a user without commands

Know how to edit the /*etc*/passwd and /*etc/shadow files*

More tricks

Show 10 Largest Open Files

```
$ lsof / | awk '{ if($7 > 1048576) print $7/1048576 "MB" " " $9 " " $1 }' | sort -n -u | tail
```

Generate a sequence of numbers

```
$ echo {01..10}
```

More tricks

 # Rename all items in a directory to lower case

```
$ for i in *; do mv "$i" "${i,,}"; done
```

 # List IP addresses connected to your server on port 80

```
$ netstat -tn 2>/dev/null | grep :80 | awk '{print $5}' | cut -d: -f1 | sort | uniq -c | sort -nr | head
```

Credits:
https://onceupon.github.io/Bash-Oneliner/

http://www.bashoneliners.com/

User Administration

Users and Groups

- There is a privileged user called “root”
  - Can do anything, like “administrator” on Windows
  - Can't login in graphical mode !
- Other users are normal users
- Some users are given “sudo” privileges: called *sudoers*
  - Sudo means “do as a superuser”
  - Password is asked, when the otherwise normal user tries to do administrative task
  - The first user account created on Ubuntu, is by default with sudo privileges

Adding/Deleting/Changing users

- System → Administration → Users and Groups
- Click on “Add” to add a user
  - Asks for password !
  - Provide the details asked for
  - Verify the user was created, by doing 'switch user'
- Try 'deleting' the user created
- Groups: Various groups of users meant for different purposes
  - Every user by default belongs to her own group
  - Add the user explicitly to other groups

Software installation

Some terms

- **.deb**
  - **The "setup" file. The installer package. Similar to Setup.exe on windows.**
  - **Contains all *binary* files and some *shell scripts***
- **repository**
  - **A collection of .deb files, categorized according to type (security, main, etc.)**
- **Software source/ ubuntu mirror**
  - **A computer on internet having all .deb files for ubuntu**

Software Installation Concept

- **Online installation**
  - **Use the "Ubuntu Software Center" to select the software, click and install !**
  - **Software is fetched automatically and installed !**
  - **Much easier than Windows !**
- **Offline installation**
  - **Collect ALL .deb files for your application**
  - **Select all, and install using package manager**

Software Installation

- **When we install using Software Center**
  - **.deb files are stored in /var/cache/apt/archives folder**
- **One needs to be a *sudoer* to install software**
- **Try installing some software on your own and try them out !**

Network configuration

Setting up network for a desktop

- **DHCP**
  - **Nothing needs to be done !**
  - **Default during installing Linux**
- **Static I/P**
  - **System → Preferences → Network connections**
  - **System → Administration → Network**
  - **System → Administration → Network tools**

Network setup for wireless

- **Just plug and Play !**
- **Network icon shows available wireless network, just click and connect.**

Reliance/Tata/Idea USB devices

- Each has a different procedure
- One needs to search the web for setting it up
  - Most of the devices work plug and play on Ubuntu 12.04
  - Some do not work, as fault of the providers, they have not given an installer CD for Linux!
    - Still Linux community have found ways to work around it !
    - My Reliance netconnect worked faster and more steadily on Linux than Windows !

Disk Management

Partition

- What is C:\ , D:\, E:\ etc on your computer ?
  - "Drive" is the popular term
  - Typically one of them represents a CD/DVD RW
- What do the others represent ?
  - They are "partitions" of your "hard disk"

Partition

- Your hard disk is one contiguous chunk of storage
  - Lot of times we need to “logically separate” our storage
  - Partition is a “logical division” of the storage
  - Every “drive” is a partition
- A logical chunk of storage is partition
  - Hard disk partitions (C:, D:), CD-ROM, Pen drive, ...

Partitions

Disk Management
File Action View Help
Volume Layout Type File System Status Capacity Free Space % Fr
(E:) Partition Basic FAT32 Healthy 9.76 GB 8.37 GB 85 %
(F:) Partition Basic FAT32 Healthy 9.76 GB 7.24 GB 74 %
OLDDRIVE (H:) Partition Basic FAT32 Healthy (A... 4.99 GB 586 MB 11 %
WINDOWS XP (C:) Partition Basic FAT32 Healthy (S... 9.76 GB 2.61 GB 26 %
Windows Vista (G:) Partition Basic NTFS Healthy 8.00 GB 1.61 GB 20 %
XPBACKUP (I:) Partition Basic FAT32 Healthy 4.99 GB 4.33 GB 86 %
XXCOPY (J:) Partition Basic FAT32 Healthy 9.00 GB 4.32 GB 47 %
Disk 0
Basic
37.30 GB
Online
WINDOWS XP (C:)
9.77 GB FAT32
Healthy (System)
(E:)
9.77 GB FAT32
Healthy
(F:)
9.77 GB FAT32
Healthy
Windows Vista (G:)
8.00 GB NTFS
Healthy
Disk 1
Basic
19.01 GB
Online
OLDDRIVE (H:)
5.00 GB FAT32
Healthy (Active)
XPBACKUP (I:)
5.00 GB FAT32
Healthy
XXCOPY (J:)
9.01 GB FAT32
Healthy
Primary partition
Extended partition
Logical drive

Managing partitions and hard drives

- System → Administration → Disk Utility
- Had drive partition names on Linux
  - /dev/sda → Entire hard drive
  - /dev/sda1, /dev/sda2, /dev/sda3, ... Different partitions of the hard drive
  - Each partition has a *type* – ext4, ext3, ntfs, fat32, etc.
- Pen drives can also be managed from here
- Formatting can also be done from here

Boot from a live CD.
Go to terminal.

Find out the partition for your UBUNTU installation. Let's say it is /dev/sda5

```
$ sudo mount /dev/sda5 /tmp/a
$ cd /tmp/a
$ mount -t proc /proc ./proc
$ sudo mount --rbind /dev ./dev
$ sudo mount --rbind /run ./run
$ sudo mount --rbind /sys ./sys
$ sudo chroot .
# update-initramfs  -c -k 5.11.0-44-generic
# update-grub
# mount /dev/sda1 ./boot/efi/ # optional
# grub-install /dev/sda
```

Started on	Saturday, 22 May 2021, 8:00 AM
State	Finished
Completed on	Saturday, 22 May 2021, 9:30 AM
Time taken	1 hour 30 mins
Grade	**26.12** out of 40.00 (**65**%)

Question **1**
Incorrect
Mark 0.00 out of 1.00

A 4 GB disk with 1 KB of block size would require these many number of **blocks** for it's free block bitmap:

Answer: 4096 ✖

The correct answer is: 512

Question **2**
Correct
Mark 1.00 out of 1.00

Given that the memory access time is 110 ns, probability of a page fault is 0.5 and page fault handling time is 12 ms,
The effective memory access time in nanoseconds is:

Answer: 6000165 ✔

The correct answer is: 6000055.00

Question **3**
Incorrect
Mark 0.00 out of 1.00

The maximum size of a file in number of blocks of BSIZE in xv6 code is
(write a number only)

Answer: 268 ✖

The correct answer is: 138

Question **4**
Incorrect
Mark 0.00 out of 1.00

Calculate the average waiting time using
Round Robin scheduling with time quantum of 5 time units
for the following workload
assuming that they arrive in the order written below.

Process	Burst Time
P1	5
P2	7
P3	6
P4	2

Write only a number in the answer upto two decimal points.

Answer: 40.75 ✖

The correct answer is: 10.25

Question **5**

Correct

Mark 1.00 out of 1.00

For the reference string

4 2 5 1 0 1 2 5 4 1 2

the number of page faults, including initial ones,

with FIFO replacement and 2 frames are :

Answer: 10 ✔

4 -

4 2

5 2

5 1

0 1

-

2 1

2 5

4 5

4 1

2 1

The correct answer is: 10

Question **6**

Correct

Mark 1.00 out of 1.00

Assuming a 16- KB page size, what is the page number for the address 428517 reference in decimal :

(give answer also in decimal)

Answer: 27 ✔

The correct answer is: 26

Question **7**

Correct

Mark 1.00 out of 1.00

In the code below assume that each function can be executed concurrently by many threads/processes.
Ignore syntactical issues, and focus on the semantics.

This program is an example of

```
spinlock a, b;  // assume initialized
thread1() {
    spinlock(b);
    //some code;
    spinlock(a);
    //some code;
    spinunlock(b);
    spinunlock(a);
}
thread2() {
    spinlock(a);
    //some code;
    spinlock(b);
    //some code;
    spinunlock(b);
    spinunlock(a);
}
```

- (selected) a. Deadlock ✔
- b. Self Deadlock
- c. None of these
- d. Deadlock or livelock depending on actual race
- e. Livelock

Your answer is correct.

The correct answer is: Deadlock

Question **8**

Partially correct

Mark 1.33 out of 2.00

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid.
"..." means some code.

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
```

Atomic compare and swap instruction (to be expanded inline into code) ✔

```
void
sleep(void *chan, struct spinlock *lk)
{
  ...
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }
```

If you don't do this, a process may be running on two processors parallely ✖

```
void
acquire(struct spinlock *lk)
{
...
__sync_synchronize();
```

Tell compiler not to reorder memory access beyond this line ✔

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
} → Atomic compare and swap instruction (to be expanded inline into code), void
sleep(void *chan, struct spinlock *lk)
{
  ...
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  } → Avoid a self-deadlock, void
acquire(struct spinlock *lk)
{
...
__sync_synchronize(); → Tell compiler not to reorder memory access beyond this line

Question **9**

Correct

Mark 1.00 out of 1.00

Predict the output of the program given here.

Assume that all the path names for the programs are correct. For example "/usr/bin/echo" will actually run echo command.

Assume that there is no mixing of printf output on screen if two of them run concurrently.

In the answer replace a new line by a single space.

For example::

`good`

`output`

should be written as `good output`

```
--
main() {
    int i;
    i = fork();
    if(i == 0)
        execl("/usr/bin/echo", "/usr/bin/echo", "hi", 0);
    else
        wait(0);
    fork();
    execl("/usr/bin/echo", "/usr/bin/echo", "one", 0);
}
```

Answer: hi one one ✔

The correct answer is: hi one one

Question **10**

Partially correct

Mark 1.67 out of 2.00

Select all the blocks that may need to be written back to disk (if updated, of-course), as "Yes", when an operation of deleting a file is carried out on ext2 file system.

An option has to be correct entirely to be marked "Yes"

Superblock	Yes	✔
One or multiple data blocks of the parent directory	No	✔
One or more data bitmap blocks for the parent directory	No	✔
Block bitmap(s) for all the blocks of the file	No	✖
Possibly one block bitmap corresponding to the parent directory	Yes	✔
Data blocks of the file	No	✔

Your answer is partially correct.

only one data block of parent directory. multiple blocks not possible. an entry is always contained within one single block

You have correctly selected 5.

The correct answer is: Superblock → Yes, One or multiple data blocks of the parent directory → No, One or more data bitmap blocks for the parent directory → No, Block bitmap(s) for all the blocks of the file → Yes, Possibly one block bitmap corresponding to the parent directory → Yes, Data blocks of the file → No

Question **11**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about bootloader.

Every wrong selection will deduct marks proportional to 1/n where n is total wrong choices in the question.

You will get minimum a zero.

- [x] a. Modern Bootloaders often allow configuring the way an OS boots ✔
- [x] b. Bootloaders allow selection of OS to boot from ✔
- [ ] c. Bootloader must be one sector in length
- [ ] d. The bootloader loads the BIOS
- [x] e. LILO is a bootloader ✔

Your answer is correct.

The correct answers are: LILO is a bootloader, Modern Bootloaders often allow configuring the way an OS boots, Bootloaders allow selection of OS to boot from

Question **12**

Incorrect

Mark 0.00 out of 1.00

For each of the resource allocation diagram shown,

infer whether the graph contains at least one deadlock or not.

Yes	No		
○	●		✖
●	○		✖

: Yes

: No

Question **13**

Partially correct

Mark 0.71 out of 1.00

Mark the statements about device drivers by marking as True or False.

True	False		
		It's possible that a particular hardware has multiple device drivers available for it.	✖
		xv6 has device drivers for IDE disk and console.	✔
		A disk driver converts OS's logical view of disk into physical locations on disk.	✔
		A device driver code is specific to a hardware device	✔
		All devices of the same type (e.g. 2 hard disks) can typically use the same device driver	✔
		Writing a device driver mandatorily demands reading the technical documentation about the hardware.	✖
		Device driver is an intermediary between the end-user and OS	✔

It's possible that a particular hardware has multiple device drivers available for it.: True
xv6 has device drivers for IDE disk and console.: True
A disk driver converts OS's logical view of disk into physical locations on disk.: True
A device driver code is specific to a hardware device: True
All devices of the same type (e.g. 2 hard disks) can typically use the same device driver: True
Writing a device driver mandatorily demands reading the technical documentation about the hardware.: True
Device driver is an intermediary between the end-user and OS: False

Question **14**

Partially correct

Mark 0.33 out of 1.00

Consider this program.
Some statements are identified using the // comment at the end.

Assume that = is an atomic operation.

```
#include <stdio.h>
#include <pthread.h>
long c = 0, c1 = 0, c2 = 0, run = 1;
void *thread1(void *arg) {
    while(run == 1) {//E
        c = 10; //A
        c1 = c2 + 5; //B
    }
}
void *thread2(void *arg) {
    while(run == 1) {//F
        c = 20;//C
        c2 = c1 + 3;//D
    }
}
int main() {
    pthread_t th1, th2;
    pthread_create(&th1, NULL, thread1, NULL);
    pthread_create(&th2, NULL, thread2, NULL);
    sleep(2);
    run = 0;
    fprintf(stdout, "c = %ld c1+c2 = %ld c1 = %ld c2 = %ld \n", c, c1+c2, c1, c2);
    fflush(stdout);
}
```

Which statements are part of the critical Section?

Yes	No		
✗	✓	F	✗
✓	✗	D	✓
✗	✓	C	✗
✗	✓	A	✗
✓	✗	B	✓
✗	✓	E	✗

F: No
D: Yes
C: No
A: No
B: Yes
E: No

Question **15**

Partially correct

Mark 1.43 out of 2.00

Mark statements as T/F

All statements are in the context of preventing deadlocks.

True	False		
◉	○	A process holding one resources and waiting for just one more resource can also be involved in a deadlock.	✔
◉	○	If a resource allocation graph contains a cycle then there is a guarantee of a deadlock	✖
◉	○	The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order	✖
◉	○	Circular wait is avoided by enforcing a lock ordering	✔
◉	○	Hold and wait means a thread/process holding some locks and waiting for acquiring some.	✔
◉	○	Deadlock is possible if all the conditions are met at the same time: Mutual exclusion, hold and wait, no pre-emption, circular wait.	✔
◉	○	Mutual exclusion is a necessary condition for deadlock because it brings in locks on which deadlock happens	✔

A process holding one resources and waiting for just one more resource can also be involved in a deadlock.: True
If a resource allocation graph contains a cycle then there is a guarantee of a deadlock: False
The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order: False
Circular wait is avoided by enforcing a lock ordering: True
Hold and wait means a thread/process holding some locks and waiting for acquiring some.: True
Deadlock is possible if all the conditions are met at the same time: Mutual exclusion, hold and wait, no pre-emption, circular wait.: True
Mutual exclusion is a necessary condition for deadlock because it brings in locks on which deadlock happens: True

Question **16**

Correct

Mark 1.00 out of 1.00

Match the left side use(or non-use) of a synchronization primitive with the best option on the right side.

This is the smallest primitive made available in software, using the hardware provided atomic instructions	spinlock	✔
This tool is useful for event-wait scenarios	semaphore	✔
This tool is more useful on multiprocessor systems	spinlock	✔
This tool is quite attractive in solving the main bounded buffer problem	semaphore	✔
This tool is very useful for waiting for 'something'	condition variables	✔

Your answer is correct.

The correct answer is: This is the smallest primitive made available in software, using the hardware provided atomic instructions → spinlock, This tool is useful for event-wait scenarios → semaphore, This tool is more useful on multiprocessor systems → spinlock, This tool is quite attractive in solving the main bounded buffer problem → semaphore, This tool is very useful for waiting for 'something' → condition variables

Question **17**

Correct

Mark 1.00 out of 1.00

The permissions -rwx--x--x on a file mean

- [x] a. The file can be read only by the owner ✔
- [ ] b. 'cat' on the file by owner will not work
- [ ] c. 'cat' on the file by any user will work
- [x] d. 'rm' on the file by any user will work ✔
- [x] e. The file can be executed by anyone ✔
- [x] f. The file can be written only by the owner ✔

Your answer is correct.

The correct answers are: The file can be executed by anyone, The file can be read only by the owner, The file can be written only by the owner, 'rm' on the file by any user will work

Question **18**

Incorrect

Mark 0.00 out of 1.00

Note: for this question you get full marks if you select all and only correct options, you get ZERO if at least one option is wrong or not selected.

Select all the correct statements about log structured file systems.

- [x] a. a transaction is said to be committed when all operations are written to file system ✖
- [x] b. log may be kept on same block device or another block device ✔
- [ ] c. file system recovery may end up losing data
- [x] d. even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery ✔
- [x] e. file system recovery recovers all the lost data ✖

Your answer is incorrect.

The correct answers are: file system recovery may end up losing data, log may be kept on same block device or another block device, even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery

Question **19**

Incorrect

Mark 0.00 out of 1.00

Consider the structure of directory entry in ext2, as shown in this diagram.

Select the correct statements about the directory entry in ext2 file system.

The correct formula for rec_len is (when entries are continuously stored)

- a. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + (-1) * (strlen(name)%4)
- b. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + (strlen(name)- 4) %4
- c. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + 4 - (strlen(name)%4) ✖
- d. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + (-1) * (strlen(name)- 4)
- e. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) %4
- f. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + strlen(name)

Your answer is incorrect.

The correct answer is: rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + (-1) * (strlen(name)- 4)

Question **20**

Partially correct

Mark 0.50 out of 1.00

Mark whether the given sequence of events is possible or not-possible. Also, select the reason for your answer.

For each sequence it's a not-possible sequence if some important event is not mentioned in the sequence.

Assume that the kernel code is non-interruptible and uniprocessor system.

```
Process P1 executing a system call
Timer interrupt
Generic interrupt handler runs
Scheduler runs
Scheduler selects P2 for execution
P2 returns from timer interrupt handler
Process p2, user code executing
```

This sequence of events is: not-possible ✔

Because

Interrupts are not possible when kernel code is running ✖

Question **21**

Incorrect

Mark 0.00 out of 1.00

The given semaphore implementation faces which problem?

Assume any suitable code for signal()

Note: blocks means waits in a wait queue.

```
struct semaphore {
        int val;
        spinlock lk;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0)
        ;
        (s->val)--;
        spinunlock(&(s->sl));
}
```

- a. blocks holding a spinlock
- b. deadlock
- c. too much spinning, bounded wait not guaranteed ✖
- d. not holding lock after unblock

Your answer is incorrect.

The correct answer is: deadlock

Question **22**

Partially correct

Mark 0.80 out of 1.00

Mark statements True/False w.r.t. change of states of a process.

Reference: The process state diagram (and your understanding of how kernel code works)

Figure 3.2 Diagram of process state.

True	False		
○	●	A process in RUNNING state only can become TERMINATED because scheduler moves it to ZOMBIE state	✔
●	○	A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.	✖
●	○	A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred	✔
●	○	Every process has to go through ZOMBIE state, at least for a small duration.	✔
●	○	Only a process in READY state is considered by scheduler	✔

A process in RUNNING state only can become TERMINATED because scheduler moves it to ZOMBIE state: False
A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.: False
A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred: True
Every process has to go through ZOMBIE state, at least for a small duration.: True
Only a process in READY state is considered by scheduler: True

Question **23**

Correct

Mark 1.00 out of 1.00

Select T/F for statements about Volume Managers.
Do pay attention to the use of the words physical partition and physical volume.

True	False			
◉	○	The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.	✔	
◉	○	A logical volume can be extended in size but upto the size of volume group	✔	
◉	○	A logical volume may span across multiple physical volumes	✔	
◉	○	The volume manager stores additional metadata on the physical disk partitions	✔	
◉	○	A physical partition should be initialized as a physial volume, before it can be used by volume manager.	✔	
◉	○	A volume group consists of multiple physical volumes	✔	
◉	○	A logical volume may span across multiple physical partitions	✔	since a physical volume is made up of physical partitions, and a volume can span across multiple PVs, it can also span across multiple PP

The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.: True
A logical volume can be extended in size but upto the size of volume group: True
A logical volume may span across multiple physical volumes: True
The volume manager stores additional metadata on the physical disk partitions: True
A physical partition should be initialized as a physial volume, before it can be used by volume manager.: True
A volume group consists of multiple physical volumes: True
A logical volume may span across multiple physical partitions: True

Question **24**

Correct

Mark 1.00 out of 1.00

Map the block allocation scheme with the problem it suffers from

(Match pairs 1-1, match a scheme with the problem that it suffers from relatively the most, compared to others)

Continuous allocation	need for compaction	✔
Linked allocation	Too many seeks	✔
Indexed Allocation	Overhead of reading metadata blocks	✔

Your answer is correct.

The correct answer is: Continuous allocation → need for compaction, Linked allocation → Too many seeks, Indexed Allocation → Overhead of reading metadata blocks

Question **25**

Correct

Mark 1.00 out of 1.00

This one is not a system call:

- ○ a. open
- ○ b. read
- ○ c. write
- ◉ d. scheduler ✔

Your answer is correct.

The correct answer is: scheduler

Question **26**

Correct

Mark 1.00 out of 1.00

Match the pairs.

This question is based on your general knowledge about operating systems/related concepts and their features.

Java thereads	monitors,re-entrant locks, semaphores ✔
Linux threads	atomic-instructions, spinlocks, etc. ✔
POSIX threads	semaphore, mutex, condition variables ✔

Your answer is correct.

The correct answer is: Java thereads → monitors,re-entrant locks, semaphores, Linux threads → atomic-instructions, spinlocks, etc., POSIX threads → semaphore, mutex, condition variables

Question **27**

Correct

Mark 1.00 out of 1.00

Consider the following list of free chunks, in continuous memory management:

7k, 15k, 21k, 14k, 19k, 6k

Suppose there is a request for chunk of size 5k, then the free chunk selected under each of the following schemes will be

Best fit: 6k ✔

First fit: 7k ✔

Worst fit: 21k ✔

Question **28**

Correct

Mark 1.00 out of 1.00

This one is not a scheduling algorithm

- a. Round Robin
- b. SJF
- c. Mergesort ✔
- d. FCFS

Your answer is correct.

The correct answer is: Mergesort

Question **29**

Correct

Mark 1.00 out of 1.00

Mark whether the concept is related to scheduling or not.

Yes	No		
✓		timer interrupt	✓
✓		context-switch	✓
✓		ready-queue	✓
	✓	file-table	✓
✓		runnable process	✓

timer interrupt: Yes
context-switch: Yes
ready-queue: Yes
file-table: No
runnable process: Yes

Question **30**

Partially correct

Mark 1.00 out of 2.00

Map ext2 data structure features with their purpose

Many copies of Superblock	Choose...
Free blocks count in superblock and group descriptor ✖	Redundancy to ensure the most crucial data structure is not lost
Used directories count in group descriptor ✖	is redundant and helps do calculations of directory entries faster
Combining file type and access rights in one variable ✔	saves 1 byte of space
rec_len field in directory entry ✖	Try to keep all the data of a directory and it's file close together in a group
File Name is padded ✔	aligns all memory accesses on word boundary, improving performance
Inode bitmap is one block ✔	limits total number of files that can belong to a group
Block bitmap is one block ✔	Limits the size of a block group, thus improvising on purpose of a group
Mount count in superblock ✔	to enforce file check after certain amount of mounts at boot time
Inode table location in Group Descriptor ✖	is redundant and helps do calculations of directory entries faster
Inode table ✔	All inodes are kept together so that one disk read leads to reading many inodes together, effectively doing a buffering of subsequent inode reads, and to save space on disk
A group ✖	Redundancy to ensure the most crucial data structure is not lost

Your answer is partially correct.

You have correctly selected 6.

The correct answer is: Many copies of Superblock → Redundancy to ensure the most crucial data structure is not lost, Free blocks count in superblock and group descriptor → Redundancy to help fsck restore consistency, Used directories count in group descriptor → attempt is made to evenly spread the first-level directories, this count is used there, Combining file type and access rights in one variable → saves 1 byte of space, rec_len field in directory entry → allows holes and linking of entries in directory, File Name is padded → aligns all memory accesses on word boundary, improving performance, Inode bitmap is one block → limits total number of files that can belong to a group, Block bitmap is one block → Limits the size of a block group, thus improvising on purpose of a group, Mount count in superblock → to enforce file check after certain amount of mounts at boot time, Inode table location in Group Descriptor → Obvious, as it's per group and not per file-system, Inode table → All inodes are kept together so that one disk read leads to reading many inodes together, effectively doing a buffering of subsequent inode reads, and to save space on disk, A group → Try to keep all the data of a directory and it's file close together in a group

Question **31**

Partially correct

Mark 1.85 out of 2.00

Mark True/False

Statements about scheduling and scheduling algorithms

True	False		
●	○	The nice() system call is used to set priorities for processes	✔
●	○	Aging is used to ensure that low-priority processes do not starve in priority scheduling.	✔
○	●	In non-pre-emptive priority scheduling, the highest priority process is scheduled and runs until it gives up CPU.	✖
●	○	xv6 code does not care about Processor Affinity	✔
●	○	In pre-emptive priority scheduling, priority is implemented by assigning more time quantum to higher priority process.	✔
●	○	A scheduling algorithm is non-premptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.	✔
●	○	Processor Affinity refers to memory accesses of a process being stored on cache of that processor	✔
●	○	Response time will be quite poor on non-interruptible kernels	✔
●	○	Shortest Remaining Time First algorithm is nothing but pre-emptive Shortest Job First algorithm	✔
●	○	On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread	✔
●	○	Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.	✔
●	○	Pre-emptive scheduling leads to many race conditions in kernel code.	✔
●	○	Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.	✔

The nice() system call is used to set priorities for processes: True
Aging is used to ensure that low-priority processes do not starve in priority scheduling.: True
In non-pre-emptive priority scheduling, the highest priority process is scheduled and runs until it gives up CPU.: True
xv6 code does not care about Processor Affinity: True
In pre-emptive priority scheduling, priority is implemented by assigning more time quantum to higher priority process.: True
A scheduling algorithm is non-premptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.: True
Processor Affinity refers to memory accesses of a process being stored on cache of that processor: True
Response time will be quite poor on non-interruptible kernels: True
Shortest Remaining Time First algorithm is nothing but pre-emptive Shortest Job First algorithm: True
On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread: True
Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.: True
Pre-emptive scheduling leads to many race conditions in kernel code.: True
Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.: True

Question **32**

Partially correct

Mark 1.17 out of 2.00

The unix file semantics demand that changes to any open file are visible immediately to any other processes accessing that file at that point in time.

Select the data-structure/programmatic features that ensure the implementation of unix semantics. (Assume there is no mmap())

Yes	No		
○	●	All processes accessing the same file share the file descriptor among themselves	✔
○	●	The pointer entry in the file descriptor array entry points to the data of the file directly	✔
●	○	There is only one global file structure per on-disk file.	✖
○	●	All file accesses are made using only global variables	✔
●	○	The 'file offset' is shared among all the processes that access the file.	✖
○	●	No synchronization is implemented so that changes are made available immediately.	✔
●	○	A single spinlock is to be used to protect the unique global 'file structure' representing the file, thus synchronizing access, and making other processes wait for earlier process to finish writing so that writes get visible immediately.	✖
●	○	There is only one in-memory copy of the on disk file's contents in kernel memory/buffers	✔
●	○	The file descriptors in every PCB are pointers to the same global file structure.	✖
○	●	The file descriptor array is external to PCB and all processes that share a file, have pointers to same file-descriptors' array	✔
●	○	All file structures representing any open file, give access to the same in-memory copy of the file's contents	✔
●	○	The 'file offset' index is stored outside the file-structure to which file-descriptor array points	✖

All processes accessing the same file share the file descriptor among themselves: No
The pointer entry in the file descriptor array entry points to the data of the file directly: No
There is only one global file structure per on-disk file.: No
All file accesses are made using only global variables: No
The 'file offset' is shared among all the processes that access the file.: No
No synchronization is implemented so that changes are made available immediately.: No
A single spinlock is to be used to protect the unique global 'file structure' representing the file, thus synchronizing access, and making other processes wait for earlier process to finish writing so that writes get visible immediately.: No
There is only one in-memory copy of the on disk file's contents in kernel memory/buffers: Yes
The file descriptors in every PCB are pointers to the same global file structure.: No
The file descriptor array is external to PCB and all processes that share a file, have pointers to same file-descriptors' array: No
All file structures representing any open file, give access to the same in-memory copy of the file's contents: Yes
The 'file offset' index is stored outside the file-structure to which file-descriptor array points: No

Question **33**

Partially correct

Mark 0.33 out of 2.00

Map the function in xv6's file system code, to it's perceived logical layer.

namei	inode	✗
filestat()	Choose...	
dirlookup	directory	✓
ialloc	file descriptor	✗
stati	Choose...	
ideintr	buffer cache	✗
bread	Choose...	
balloc	file descriptor	✗
sys_chdir()	system call	✓
skipelem	system call	✗
commit	system call	✗
bmap	system call	✗

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: namei → pathname lookup, filestat() → file descriptor, dirlookup → directory, ialloc → inode, stati → inode, ideintr → disk driver, bread → buffer cache, balloc → block allocation on disk, sys_chdir() → system call, skipelem → pathname lookup, commit → logging, bmap → inode

◄ Course Exit Feedback

Jump to...

xv6-public-master ►

Started on	Saturday, 30 April 2022, 2:05:41 PM
State	Finished
Completed on	Saturday, 30 April 2022, 6:47:56 PM
Time taken	4 hours 42 mins
Grade	**34.99** out of 40.00 (**87**%)

Question **1**

Correct

Mark 1.00 out of 1.00

which of the following scheduling algorithms discriminate either in favor of or against short processes, from the perspective of minimizing waiting time.

For	Against		
✗	✓	FCFS	✓
✓	✗	Round Robin	✓
✓	✗	Multilevel feedback queues	✓

FCFS: Against
Round Robin: For
Multilevel feedback queues: For

Question **2**

Correct

Mark 1.00 out of 1.00

Consider a computer system with a 32-bit logical address and 8- KB page size. The system supports up to 1 GB of physical memory. How many entries are there in each of the following?

a. A conventional, single-level page table (write a decimal number):

524288

✓

b. An inverted page table (number of entries only, write a decimal number):

131072

✓

Question **3**

Correct

Mark 1.00 out of 1.00

Select T/F for the disk block allocation scheme related statements

True	False		
◉	○	Continuous allocation leads to faster file access	✔
◉	○	FAT uses linked allocation	✔
◉	○	The unix inode is based on indexed allocation	✔
◉	○	Continuous allocation allows to fetch any block with just one seek	✔
◉	○	NVM storage devices are pushing for search for new block allocation schemes	✔
◉	○	Continuous allocation may involve a costly search for free space	✔
◉	○	Maximum file size limit is determined by disk block allocation scheme, as one of the factors.	✔
◉	○	Linked allocation does away with file size limitation to a large extent	✔

Continuous allocation leads to faster file access: True
FAT uses linked allocation: True
The unix inode is based on indexed allocation: True
Continuous allocation allows to fetch any block with just one seek: True
NVM storage devices are pushing for search for new block allocation schemes: True
Continuous allocation may involve a costly search for free space: True
Maximum file size limit is determined by disk block allocation scheme, as one of the factors.: True
Linked allocation does away with file size limitation to a large extent: True

Question **4**

Correct

Mark 1.00 out of 1.00

Which one of the following is not a system call

- ◉ a. lock ✔
- ○ b. open
- ○ c. lseek
- ○ d. mount

The correct answer is: lock

Question **5**

Partially correct

Mark 1.13 out of 1.50

The following processes are being scheduled using a pre-emptive, priority-based, round-robin scheduling algorithm.

Process	Priority	Burst	Arrival
P_1	8	15	0
P_2	3	20	0
P_3	4	20	20
P_4	4	20	25
P_5	5	5	45
P_6	5	15	55

Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. The scheduler will execute the currently highest priority process for it's full duration, unless it gets pre-empted by newly arriving higher priority process. For processes with the same priority, a round-robin scheduler will be used with a time quantum of 10 units. If a process is pre-empted by a higher-priority process, the pre-empted process is placed at the front of the queue of same priority processes (so that it's turn continues when higher priority process is over).

The order in which the processes get scheduled is (write your answer without a space, e.g. P1,P2,P3,P4,P5) :

P1,P2,P3,P4,P3,P5,P3,P6,P4,P2

✖

The turn-around time for the process P2 is:

95

✔

The turn-around time for the process P6 is:

15

✔

The process P5 finishes at time unit:

50

✔

0-15 P1

15-20 P2

20-40 P3

40-45 P4

45-50 P5

50-55 P4

55-70 P6

70-80 P4

80-95 P2

--

P2 turnaround time = 95 - 15 = 80

Question **6**

Partially correct

Mark 1.62 out of 3.00

Suppose it is required to add the chown() (without notion of a group, just the notion of owner and others) system call to xv6. Select the changes, from the options given below, which are an absolute must to effect the addition of this system call.

Changes w.r.t. other system calls should be selected if those system calls are necessary for the implementation of the chown() system call.

Must	Not-Must		
✖	✔	Add a EUID field in the struct proc representing Effective user ID	✖
✔	✖	Add a mode field representing file permissions, inside dinode	✔
✖	✔	Add a UID field in the struct proc represenging the USER-ID of the user who started this process	✖
✔	✖	Add a uid field representing the owner, inside dinode	✔
✖	✔	Add few extra files belonging to different users, using mkfs.c	✔
✖	✔	Modify exec() to check SUID bit on the executable file and set EUID of the process to UID of the executable	✖
✔	✖	Add the chown() system call which checks if the user with id "0" is calling this system call and then change the ownership of the on-disk and in-memory inode.	✔
✖	✔	Add a setuid(), seteuid() system call, callable only by the user with ID or EUID equal to "0" to set the new user id of the process	✖
✖	✔	Create an application program su.c which itself is a SUID program, and calls setuid() and setuid() with specified user-ID and then does exec() of a shell.	✖
✔	✖	Modify mkfs.c to add owner and permissions to each file created	✔
✖	✔	Inherit the UID and EUID of the parent in fork()	✖
✖	✔	Mandatorily create a file like "passwd" which maps user-names to user-IDs and modify other utilities (like "ls") to show user-name instead of user-ID for the files.	✔
✖	✔	Add the username field to the on disk inode, that is dinode.	✔

Add a EUID field in the struct proc representing Effective user ID: Not-Must
Add a mode field representing file permissions, inside dinode: Must
Add a UID field in the struct proc represenging the USER-ID of the user who started this process: Not-Must
Add a uid field representing the owner, inside dinode: Must
Add few extra files belonging to different users, using mkfs.c: Not-Must
Modify exec() to check SUID bit on the executable file and set EUID of the process to UID of the executable: Not-Must
Add the chown() system call which checks if the user with id "0" is calling this system call and then change the ownership of the on-disk and in-memory inode.: Must
Add a setuid(), seteuid() system call, callable only by the user with ID or EUID equal to "0" to set the new user id of the process: Not-Must
Create an application program su.c which itself is a SUID program, and calls setuid() and setuid() with specified user-ID and then does exec() of a shell.: Not-Must

Modify mkfs.c to add owner and permissions to each file created: Must
Inherit the UID and EUID of the parent in fork(): Not-Must
Mandatorily create a file like "passwd" which maps user-names to user-IDs and modify other utilities (like "ls") to show user-name instead of user-ID for the files.: Not-Must
Add the username field to the on disk inode, that is dinode.: Not-Must

Question **7**

Correct

Mark 1.00 out of 1.00

The following diagram that explains the concept of multi-threading

Figure 4.1 Single-threaded and multithreaded processes.

Leads to the following conclusions about implementation of threads

True	False		
○ ✗	◉ ☑	A multi-threaded program can be split in multiple ELF files.	✔
◉ ☑	○ ✗	The memory management for the process in the kernel should not (most typically) change due to creation of a thread.	✔
◉ ☑	○ ✗	Switching between threads requires a "context switch" with change of registers involved in it	✔
◉ ☑	○ ✗	Each thread should be represented by a function that starts execution on a separate stack.	✔

A multi-threaded program can be split in multiple ELF files.: False
The memory management for the process in the kernel should not (most typically) change due to creation of a thread.: True
Switching between threads requires a "context switch" with change of registers involved in it: True
Each thread should be represented by a function that starts execution on a separate stack.: True

Question **8**

Correct

Mark 1.00 out of 1.00

Consider this program.
Some statements are identified using the // comment at the end.
Assume that = is an atomic operation.

```
#include <stdio.h>
#include <pthread.h>
long c = 0, c1 = 0, c2 = 0, run = 1;
void *thread1(void *arg) {
    while(run == 1) {//E
        c = c1; //A
        c1++; //B
    }
}
void *thread2(void *arg) {
    while(run == 1) {//F
        c = c2;//C
        c2++;//D
    }
}
int main() {
    pthread_t th1, th2;
    pthread_create(&th1, NULL, thread1, NULL);
    pthread_create(&th2, NULL, thread2, NULL);
    sleep(2);
    run = 0;
    fprintf(stdout, "c = %ld c1+c2 = %ld c1 = %ld c2 = %ld \n", c, c1+c2, c1, c2);
    fflush(stdout);
}
```

Which statements are part of the critical Section?

Yes	No		
✗	✓	E	✓
✗	✓	F	✓
✗	✓	B	✓
✗	✓	D	✓
✓	✗	A	✓
✓	✗	C	✓

As per Remzi's book: "A critical section is a piece of code that accesses a shared variable (or more generally, a shared resource and must not be concurrently executed by more than one thread."

As per Galvin's Book: a critical section, in which the process may be accessing — and updating — data that is shared with at least one other process. The important feature of the system is that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Since A and C refer to a variable that is shared, it's critical section. Here the hardware guarantees (as = is atomic) that the critical section is accessed by only one thread at a time.

E: No
F: No
B: No
D: No
A: Yes
C: Yes

Question **9**

Partially correct

Mark 0.38 out of 1.00

It is proposed that when a process does an illegal memory access, xv6 terminate the process by printing the error message "Illegal Memory Access". Select all the changes that need to be done to xv6 for this as True (Note that the changes proposed here may not cover the exhaustive list of all changes required) and the un-necessary/wrong changes as False.

Required	Un-necessary/Wrong		
		Change in the Makefile and instruct cc/ld to start the code of each program at some address other than 0	✗
		Ensure that the address 0 is mapped to invalid	✗
		Mark each page as readonly in the page table mappings	✓
		Change allocuvm() to call mappages() with proper permissions on each page table entry	✗
		Add code that checks if the illegal memory access trap was due to an actual illegal memory access.	✗
		Handle the Illegal memory acceess trap in trap() function, and terminate the currently running process.	✓
		Change exec to treat text/data sections separately and call allocuvm() with proper flags for page table entries	✓
		Change mappages() to set specified permissions on each page table entry	✗

Change in the Makefile and instruct cc/ld to start the code of each program at some address other than 0: Required
Ensure that the address 0 is mapped to invalid: Required
Mark each page as readonly in the page table mappings: Un-necessary/Wrong
Change allocuvm() to call mappages() with proper permissions on each page table entry: Required
Add code that checks if the illegal memory access trap was due to an actual illegal memory access.: Un-necessary/Wrong
Handle the Illegal memory acceess trap in trap() function, and terminate the currently running process.: Required
Change exec to treat text/data sections separately and call allocuvm() with proper flags for page table entries: Required
Change mappages() to set specified permissions on each page table entry: Un-necessary/Wrong

Question **10**

Partially correct

Mark 0.63 out of 1.00

Select which of the following data structures may need an update, in ext2 file system, when 5 bytes get removed from the end of an existing file

Yes	No		
✗	✓	The inode of the parent directory	✓
✓	✗	The superblock	✗
✓	✗	Inode of the file	✓
✗	✓	The inode bitmap	✓
✓	✗	The group descriptor	✗
✗	✓	A free block on the file-system	✗
✓	✗	One of the block bitmaps	✓
✗	✓	The boot sector	✓

The inode of the parent directory: No
The superblock: Yes
Inode of the file: Yes
The inode bitmap: No
The group descriptor: Yes
A free block on the file-system: No
One of the block bitmaps: Yes
The boot sector: No

Question **11**

Partially correct

Mark 1.75 out of 2.00

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid.

"..." means some code.

```
void
sleep(void *chan, struct spinlock *lk)
{
  ...
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }
```

If you don't do this, a process may be running on two processors parallely ✖

```
void
panic(char *s)
{
...
  panicked = 1;
```

Ensure that no printing happens on other processors ✔

```
struct proc*
myproc(void) {
...
  pushcli();
  c = mycpu();
  p = c->proc;
  popcli();
...
}
```

Disable interrupts to avoid another process's pointer being returned ✔

```
void
acquire(struct spinlock *lk)
{
...
__sync_synchronize();
```

Tell compiler not to reorder memory access beyond this line ✔

```
void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs);
```

Traverse ebp chain to get sequence of instructions followed in functions calls ✔

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
```

Atomic compare and swap instruction (to be expanded inline into code) ✔

```
void
yield(void)
{
...
release(&ptable.lock);
}
```

Release the lock held by some another process ✔

```
void
acquire(struct spinlock *lk)
{
  pushcli();
```

Disable interrupts to avoid deadlocks ✔

Your answer is partially correct.

You have correctly selected 7.

The correct answer is: void

```
sleep(void *chan, struct spinlock *lk)
{
  ...
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  } → Avoid a self-deadlock, void
panic(char *s)
{
...
  panicked = 1; → Ensure that no printing happens on other processors, struct proc*
myproc(void) {
...
  pushcli();
  c = mycpu();
  p = c->proc;
  popcli();
...
}


  → Disable interrupts to avoid another process's pointer being returned, void
acquire(struct spinlock *lk)
{
...
__sync_synchronize();



  → Tell compiler not to reorder memory access beyond this line, void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs);



  → Traverse ebp chain to get sequence of instructions followed in functions calls, static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
} → Atomic compare and swap instruction (to be expanded inline into code), void
```

```
yield(void)
{
...
release(&ptable.lock);
}
```

→ Release the lock held by some another process, void

```
acquire(struct spinlock *lk)
{
  pushcli();
```

→ Disable interrupts to avoid deadlocks

Question **12**

Correct

Mark 1.00 out of 1.00

Mark statements as T/F

All statements are in the context of preventing deadlocks.

True	False		
◉	○	Mutual exclusion is a necessary condition for deadlock because it brings in locks on which deadlock happens	✔
○	◉	The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order	✔
◉	○	The lock ordering to be followed to avoid circular wait is a protocol to be followed by programmers.	✔
◉	○	Circular wait is avoided by enforcing a lock ordering	✔
◉	○	If a resource allocation graph contains a cycle then there is a possibility of a deadlock	✔
○	◉	If a resource allocation graph contains a cycle then there is a guarantee of a deadlock	✔
◉	○	Deadlock is not possible if any of these conditions is not met: Mutual exclusion, hold and wait, no pre-emption, circular wait.	✔

Mutual exclusion is a necessary condition for deadlock because it brings in locks on which deadlock happens: True
The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order: False
The lock ordering to be followed to avoid circular wait is a protocol to be followed by programmers.: True
Circular wait is avoided by enforcing a lock ordering: True
If a resource allocation graph contains a cycle then there is a possibility of a deadlock: True
If a resource allocation graph contains a cycle then there is a guarantee of a deadlock: False
Deadlock is not possible if any of these conditions is not met: Mutual exclusion, hold and wait, no pre-emption, circular wait.: True

Question **13**

Correct

Mark 1.00 out of 1.00

Predict the output of the program given here.

Assume that there is no mixing of printf output on screen if two of them run concurrently.

In the answer replace a new line by a single space.

For example::

```
good
output
```

should be written as `good output`

```
--
```

```
int main() {
  int pid;
  printf("hi\n");
  pid = fork();
  if(pid == 0) {
    exit(0);
  }
  printf("bye\n");
  fork();
  printf("ok\n");
}
```

Answer: hi bye ok ok ✔

The correct answer is: hi bye ok ok

Question **14**

Partially correct

Mark 0.63 out of 1.00

For Virtual File System to work, which of the following changes are required to be done to an existing OS code (e.g. xv6)?

- [ ] a. A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/"
- [x] b. The filesystem related system calls (e.g. read, write) need to invoke the file system specific functions (e.g. ext2_read, ext2_write, ntfs_read, ntfs_write) using function pointers. ✔
- [x] c. The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup. ✔
- [ ] d. The generic inode needs to have a field representing if this inode is a mount point and also to refer/point to the root of the mounted file system's inode.
- [x] e. Each open() needs to copy the function pointers from the inode of the parent directory into the inode of the child (if not already done), unless it's traversing a mount point. (This may be done as part of lookup() which is called by open()) ✔
- [x] f. Each file-system writer needs to provide the set of function pointers for VFS, and these function pointers need to be setup in generic inode of "/" of that file system during mount() ✔
- [ ] g. The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems.
- [x] h. The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories ✔

The correct answers are: A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/", The filesystem related system calls (e.g. read, write) need to invoke the file system specific functions (e.g. ext2_read, ext2_write, ntfs_read, ntfs_write) using function pointers., The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup., The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems., The generic inode needs to have a field representing if this inode is a mount point and also to refer/point to the root of the mounted file system's inode., The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories, Each file-system writer needs to provide the set of function pointers for VFS, and these function pointers need to be setup in generic inode of "/" of that file system during mount(), Each open() needs to copy the function pointers from the inode of the parent directory into the inode of the child (if not already done), unless it's traversing a mount point. (This may be done as part of lookup() which is called by open())

Question **15**

Partially correct

Mark 0.40 out of 0.50

Which of the following statements are correct about xv6 and multiprocessor support ?

- [ ] a. Each processor on xv6 starts code execution in mpenter() when first processor sets "started" to 1.
- [x] b. xv6 supports only SMP ✔
- [x] c. In xv6 on x86, the first processor configures other processors in mpinit() ✔
- [x] d. xv6 supports a variable number of processors (upto 8) and adjusts it's data structures according to number of processors ✔
- [x] e. At any point in time, after main(), the kernel may be parallely executing on any of the processors. ✔

The correct answers are: xv6 supports only SMP, xv6 supports a variable number of processors (upto 8) and adjusts it's data structures according to number of processors, Each processor on xv6 starts code execution in mpenter() when first processor sets "started" to 1., In xv6 on x86, the first processor configures other processors in mpinit(), At any point in time, after main(), the kernel may be parallely executing on any of the processors.

Question **16**

Correct

Mark 1.00 out of 1.00

Match the diagram with the threading model

Two-Level ✔

Many-Many ✔

One-One ✔

Many-One ✔

Your answer is correct.

The correct answer is:

→ Two-Level,

→ Many-Many,

→ One-

One,

→ Many-One

Question **17**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about Shell

- [x] a. The essential job of the shell is to fork-exec the specified application ✔
- [ ] b. Examples of shell are: bash, ksh, csh, msh, ooosh, nosh, etc.
- [ ] c. Shell is a layer on top of hardware
- [ ] d. Shell converts the application code into system calls.
- [x] e. The default shell for a user is specified in /etc/passwd on typical GNU/Linux systems. ✔
- [x] f. Examples of shell are: bash, ksh, csh, zsh, sh, etc. ✔

The correct answers are: The essential job of the shell is to fork-exec the specified application, Examples of shell are: bash, ksh, csh, zsh, sh, etc., The default shell for a user is specified in /etc/passwd on typical GNU/Linux systems.

Question **18**

Correct

Mark 1.00 out of 1.00

Which one of the following is not a scheduling algorithm

- a. Multilevel Feedback Queue
- b. Priority
- c. FCFS
- d. paging ✔

The correct answer is: paging

Question **19**

Correct

Mark 1.00 out of 1.00

Mark the statements as True/False w.r.t. the basic concepts of memory management.

True	False		
✗	✓	The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.	✓
✗	✓	The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.	✓
✓	✗	When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.	✓
✗	✓	When a process is executing, each virtual address is converted into physical address by the kernel directly.	✓
✓	✗	The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.	✓
✗	✓	The kernel refers to the page table for converting each virtual address to physical address.	✓
✓	✗	The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.	✓

The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.: False
The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.: False
When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.: True
When a process is executing, each virtual address is converted into physical address by the kernel directly.: False
The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.: True
The kernel refers to the page table for converting each virtual address to physical address.: False
The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.: True

Question **20**

Partially correct

Mark 0.83 out of 1.00

Which of the following instructions should be privileged?

- [x] a. Modify entries in device-status table. ✔
- [ ] b. Read the clock.
- [x] c. Switch from user to kernel mode. ✔
- [x] d. Set value of timer. ✔
- [x] e. Turn off interrupts. ✔
- [x] f. Access I/O device. ✔
- [ ] g. Issue a trap instruction.

The correct answers are: Set value of timer., Access I/O device., Issue a trap instruction., Switch from user to kernel mode., Modify entries in device-status table., Turn off interrupts.

Question **21**

Correct

Mark 1.00 out of 1.00

Calculate the average waiting time using
FCFS scheduling
for the following workload

assuming that they arrive in this order during the first time unit:

Process	Burst Time
P1	2
P2	6
P3	2
P4	3

Write only a number in the answer upto two decimal points.

Answer: 5.00 ✔

P2 waits for 2 units

P3 waits for 2+6 units

P4 waits for 2 + 6 +2 units of time

Total waiting = 2 + 2 + 6 + 2 + 6 + 2 = 20 units

Average waiting time = 20/4 = 5

The correct answer is: 5

Question **22**

Partially correct

Mark 0.25 out of 0.50

Select all the correct statements related to implementation of a Hypervisor like VirtualBox

- [ ] a. All Traps in hardware are handled by HOST OS, but Host OS may hand it over to Guest OS if trap was from within guest OS
- [ ] b. Typically they run using CPU privilege level 1 or 2 (lower than kernel's but higher than applications)
- [x] c. When an application runs a privileged instruction it traps into the actual hardware ✔
- [x] d. The HOST OS determines whether a process (in Guest) or the guest OS itself was doing privileged instruction depending on the privilege level. ✔

The correct answers are: Typically they run using CPU privilege level 1 or 2 (lower than kernel's but higher than applications), When an application runs a privileged instruction it traps into the actual hardware, All Traps in hardware are handled by HOST OS, but Host OS may hand it over to Guest OS if trap was from within guest OS, The HOST OS determines whether a process (in Guest) or the guest OS itself was doing privileged instruction depending on the privilege level.

Question **23**

Partially correct

Mark 0.75 out of 1.00

Match each suggested semaphore implementation (discussed in class)

with the problems that it faces

```
struct semaphore {
        int val;
        spinlock lk;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                spinunlock(&(s->sl));
                spinlock(&(s->sl));
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

too much spinning, bounded wait not guaranteed ✔

```
struct semaphore {
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
block(semaphore *s) {
        listappend(s->l, current);
        schedule();
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                block(s);
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

blocks holding a spinlock ✔

```
struct semaphore {
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
block(semaphore *s) {
        listappend(s->l, current);
        spinunlock(&(s->sl));
        schedule();
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                block(s);
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
signal(seamphore *s) {
        spinlock(*(s->sl));
        (s->val)++;
        x = dequeue(s->sl) and enqueue(readyq, x);
        spinunlock(*(s->sl));
}
```

not holding lock after unblock ✔

```
struct semaphore {
        int val;
        spinlock lk;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0)
        ;
        (s->val)--;
        spinunlock(&(s->sl));
}
```

livelock ✖

Your answer is partially correct.

You have correctly selected 3.

The correct answer is:

```
struct semaphore {
        int val;
        spinlock lk;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                spinunlock(&(s->sl));
                spinlock(&(s->sl));
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

→ too much spinning, bounded wait not guaranteed,

```
struct semaphore {
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
block(semaphore *s) {
        listappend(s->l, current);
        schedule();
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                block(s);
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

→ blocks holding a spinlock,

```
struct semaphore {
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
block(semaphore *s) {
        listappend(s->l, current);
        spinunlock(&(s->sl));
        schedule();
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                block(s);
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
signal(seamphore *s) {
        spinlock(*(s->sl));
        (s->val)++;
        x = dequeue(s->sl) and enqueue(readyq, x);
        spinunlock(*(s->sl));
}
```

→ not holding lock after unblock,

```
struct semaphore {
        int val;
        spinlock lk;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0)
        ;
        (s->val)--;
        spinunlock(&(s->sl));
}
```

→ deadlock

Question **24**

Correct

Mark 1.00 out of 1.00

In an ext2 file system, if the block size is 4KB and partition size is 2 GB, then the number of block groups will be:

Answer: 16 ✔

size * 1024 * 1024 / 4 --> no of blocks

each group = 8 * 4 * 1024 blocks = 32768 blocks

so size * 1024 * 1024 / (4 * 32768) number of groups

The correct answer is: 16.00

Question **25**

Correct

Mark 1.00 out of 1.00

Match the pair

Inverted Page table	Linear/Parallel search using page number in page table	✔
Hierarchical Paging	More memory access time per hierarchy	✔
Hashed page table	Linear search on collsion done by OS (e.g. SPARC Solaris) typically	✔

Your answer is correct.

The correct answer is: Inverted Page table → Linear/Parallel search using page number in page table, Hierarchical Paging → More memory access time per hierarchy, Hashed page table → Linear search on collsion done by OS (e.g. SPARC Solaris) typically

Question **26**

Correct

Mark 1.00 out of 1.00

Write the possible contents of the file /tmp/xyz after this program.

In the answer if you want to mention any non-text character, then write \0 For example abc\0\0 means abc followed by any two non-text characters

```
int main(int argc, char *argv[]) {
   int fd1, fd2, n, i;
   char buf[128];

   fd1 = open("/tmp/xyz", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
   write(fd1, "hello", 5);
   fd2 = open("/tmp/xyz", O_WRONLY, S_IRUSR|S_IWUSR);
   write(fd2, "bye", 3);
   close(fd1);
   close(fd2);
   return 0;
}
```

Answer: byelo ✔

The correct answer is: byelo

Question **27**

Correct

Mark 1.00 out of 1.00

Select the correct statements about hard and soft links

Select one or more:

- [ ] a. Soft links increase the link count of the actual file inode
- [x] b. Hard links enforce separation of filename from it's metadata in on-disk data structures. ✔
- [ ] c. Hard links can span across partitions while soft links can't
- [ ] d. Deleting a soft link deletes only the actual file
- [x] e. Soft links can span across partitions while hard links can't ✔
- [x] f. Deleting a soft link deletes the link, not the actual file ✔
- [ ] g. Soft link shares the inode of actual file
- [x] h. Deleting a hard link deletes the file, only if link count was 1 ✔
- [ ] i. Deleting a soft link deletes both the link and the actual file
- [ ] j. Deleting a hard link always deletes the file
- [x] k. Hard links share the inode ✔
- [x] l. Hard links increase the link count of the actual file inode ✔

Your answer is correct.

The correct answers are: Soft links can span across partitions while hard links can't, Hard links increase the link count of the actual file inode, Deleting a soft link deletes the link, not the actual file, Deleting a hard link deletes the file, only if link count was 1, Hard links share the inode, Hard links enforce separation of filename from it's metadata in on-disk data structures.

Question **28**

Partially correct

Mark 2.54 out of 3.00

Suppose you are required to implement the priority scheduling algorithm in xv6. Select all the options that correctly reflect the changes that are required to be done in code:

Needed/Optional	Not Needed		
◉☑	○✖	Set the priority of the new process to the default(using inheritance) during fork()	✔
◉☑	○✖	Add a system call (like nice()) that allows to set priority of a process	✔
○✖	◉☑	Change sched() to set the priority of the process.	✔
◉☑	○✖	Add a priority field to the struct proc	✔
○✖	◉☑	Change scheduler() to calculate the priority of the process using user specified value	✔
◉☑	○✖	The nice() system call needs to acquire the ptable.lock for setting the priority.	✔
○✖	◉☑	Change swtch() to set the timer value for the process	✔
◉✖	○☑	Change yield() to re-set the timer value to zero for the process	✖
◉☑	○✖	Optionally remove the default timer value of 10000000 in lapicinit()	✔
◉✖	○☑	Change exec() to add a priority to the process.	✖
○✖	◉☑	Set the priority of the process as per information in ELF file	✔
◉☑	○✖	Insert code in scheduler() after if(p->state != RUNNABLE); continue; by a code that selects the highest priority process.	✔
◉☑	○✖	Set a new value for the timer before you call swtch() in scheduler()	✔

Set the priority of the new process to the default(using inheritance) during fork(): Needed/Optional
Add a system call (like nice()) that allows to set priority of a process: Needed/Optional
Change sched() to set the priority of the process.: Not Needed
Add a priority field to the struct proc: Needed/Optional
Change scheduler() to calculate the priority of the process using user specified value: Not Needed
The nice() system call needs to acquire the ptable.lock for setting the priority.: Needed/Optional
Change swtch() to set the timer value for the process: Not Needed
Change yield() to re-set the timer value to zero for the process: Not Needed
Optionally remove the default timer value of 10000000 in lapicinit(): Needed/Optional
Change exec() to add a priority to the process.: Not Needed
Set the priority of the process as per information in ELF file: Not Needed
Insert code in scheduler() after if(p->state != RUNNABLE); continue; by a code that selects the highest priority process.: Needed/Optional
Set a new value for the timer before you call swtch() in scheduler(): Needed/Optional

Question **29**

Correct

Mark 0.50 out of 0.50

Doing a lookup on the pathname /a/b/b/c/d for opening the file "d" requires reading 6 ✔ no. of inodes. Assume that there are no hard/soft links on the path.

Write the answer as a number.

The correct answer is: 6

Question **30**

Partially correct

Mark 1.86 out of 2.00

Given below is an incomplete code for reader-writer lock with preference for readers. Fill in the blanks to complete the code.

Note1:

In your answer, if an expression is to be written, then please separate all tokens by exactly one space in between. For example

`i = i + 1` is correct while

`i=i+1` is not correct or

`i= i +1` is also not correct.

Note2:

Correct: a->b++ (no spaces here!)

Any other notation is incorrect.

Note3: *The code of downgrade() and upgrade() has proportional to 4/7 marks.*

Code of rwlock:

```
typedef struct rwlock {
   int nActive;
   int nPendingReads;
   int nPendingWrites;
   spinlock_t
```

sl

✔ ;

```
   condition canRead;
   condition canWrite;
}rwlock;

void lockShared(rwlock *r) {
   spin_lock(&r->sl);
```

r->nPendingReads++

✔ ;

```
   while(r->nActive < 0)
      wait(
```

&r->canRead

✔ , &r->sl);

r->nActive++

✔ ;

```
   r->nPendingReads--;
   spin_unlock(&r->sl);
}
void unlockShared(rwlock *r) {
   spin_lock(&r->sl);
```

r->nActive--

✔ ;

```
   if(r->nActive == 0) {
      spin_unlock(&r->sl);
      do_signal(
```

&r->canWrite

✔);

```
   } else
      spin_unlock(&r->sl);
}
void lockExclusive(rwlock *r) {
   spin_lock(&r->sl);
   r->nPendingWrites++;
   while(r->nActive || r->nPendingReads)
      wait(
```

&r->canWrite

✔ ,

&r->sl

✔);

```
    r->nPendingWrites--;
```

r->nActive = -1

✔ ;

```
    spin_unlock(&r->sl);
}
void unlockExclusive(rwlock *r) {
    boolean_t wakeReaders;
    int i;
    spin_lock(&r->sl);
```

r->nActive = 0

✔ ;

```
    if(r->nPendingReads != 0) {
        for(i = 0; i <
```

r->nPendingReads

✔ ; i++)

```
            do_signal(&r->canRead);
    }
    else
        do_signal(
```

&r->canWrite

✔);

```
    spin_unlock(&r->sl);
}
void downgrade(rwlock *r) {
    boolean_t wakeReaders;
    int i;
    spin_lock(&r->sl);
    r->nActive =
```

1

✔ ;

```
    if(r->nPendingReads != 0)
        for(i = 0; i <
```

nPendingReads

✖ ; i++)

```
            do_signal(
```

&r->canRead

✔);

```
    spin_unlock(&r->sl);
}
void upgrade(rwlock *r) {
    spin_lock(&r->sl);
    if(r->nActive == 1) {
        r->nActive =
```

-1

✔ ;

```
    } else {
        r->nPendingWrites++;
        r->nActive--;
        while(r->nActive !=
```

0

✔)

```
            wait(
```

&r->canWrite

✔ , &r->sl);

r->nPendingWrites--

✔ ;
r->nActive =

-1

✔ ;
}
spin_unlock(&r->sl);
}

Question **31**

Correct

Mark 1.00 out of 1.00

For each of the resource allocation diagram shown,

infer whether the graph contains at least one deadlock or not.

Yes	No		
◉	○		✔
◉	○		✔
◉	○		✔

: Yes

: Yes

: Yes

Question **32**

Correct

Mark 1.00 out of 1.00

Map each signal with it's meaning

SIGPIPE	Broken Pipe	✔
SIGUSR1	User Defined Signal	✔
SIGSEGV	Invalid Memory Reference	✔
SIGALRM	Timer Signal from alarm()	✔
SIGCHLD	Child Stopped or Terminated	✔

The correct answer is: SIGPIPE → Broken Pipe, SIGUSR1 → User Defined Signal, SIGSEGV → Invalid Memory Reference, SIGALRM → Timer Signal from alarm(), SIGCHLD → Child Stopped or Terminated

Question **33**

Correct

Mark 1.00 out of 1.00

Not a FOSS: Which of the following is/are not a FOSS operating system ?

- [ ] a. Minix
- [ ] b. Open Solaris
- [ ] c. xv6
- [ ] d. GNU/Linux
- [ ] e. Darwin (core kernel of Mac-OS)
- [ ] f. FreeBSD
- [x] g. Windows ✔
- [ ] h. BSD Unix

The correct answer is: Windows

Question **34**

Correct

Mark 1.00 out of 1.00

Will this code work for a spinlock() operation? The intention here is to call compare-and-swap() only if the lock is not held (the if condition checks for the same).

```
void spinlock(int *lock) {
{
   while (true) {
      if (*lock == 0) {
         /* lock appears to be available */
         if (!compare_and_swap(lock, 0, 1))
         break
      }
   }
}
```

- ( ) a. No, because this breaks the atomicity requirement of compare-and-test.
- ( ) b. Yes, because there is no race to update the lock variable
- ( ) c. No, because in the case of both processes succeeding in the "if" condition, both may end up acquiring the lock.
- (•) d. Yes, because no matter in which order the if-check and compare-and-swap run in multiple processes, only one process will succeed in compare-and-swap() and others will keep looping in while-loop. ✔

Your answer is correct.

The correct answer is: Yes, because no matter in which order the if-check and compare-and-swap run in multiple processes, only one process will succeed in compare-and-swap() and others will keep looping in while-loop.

Question **35**

Partially correct

Mark 0.75 out of 1.00

Mark the statements as True/False with respect to Mobile systems and swapping.

True	False		
◉	○	Mobile systems generally do not support swapping	✔
○	◉	Size of flash memory is one reason for restriction on use of swap on mobile devices.	✖
◉	○	Limited number of write operations that flash memory can tolerate, is one reason for limitations of swapping on mobile devices.	✔
◉	○	Poor throughput between main memory and flash memory is one reason for restriction on use of swap on mobile devices.	✔

Mobile systems generally do not support swapping: True
Size of flash memory is one reason for restriction on use of swap on mobile devices.: True
Limited number of write operations that flash memory can tolerate, is one reason for limitations of swapping on mobile devices.: True
Poor throughput between main memory and flash memory is one reason for restriction on use of swap on mobile devices.: True

◄ Course Exit Feedback

Jump to...

Operating Systems Even-sem-21-22

Topic-wise Quiz-3 (processes, trap handling, scheduler)

Topic-wise Quiz-3 (processes, trap handling, scheduler)

Attempts allowed: 1

This quiz closed on Monday, 21 February 2022, 8:00:00 PM

To continue with this quiz attempt you must open your webcam, and it will be used to monitor you during the quiz.

Time limit: 55 mins

Summary of your previous attempts

State	Grade / 10.00	Review
Finished Submitted Monday, 21 February 2022, 7:59:27 PM	8.90	Review

Your final grade for this quiz is 8.90/10.00.

No more attempts are allowed

Back to the course

◀ Description of some possible course mini projects

Jump to...

(Code) mmap related programs ▶

You are logged in as 111903018 Ankit Nehul_Div1 (Log out)
OS-even-sem-21-22

Search/Create/Approve Courses
Search Courses
New Course Request(for Faculty only)
Approve Requests(For Admins only)
Learn Moodle
COEP Moodle videos for teachers
Moodle Video Tutorials
Student Tutorials
Teacher Tutorials
Change Password
Preferences
English (en)
English (en)
मराठी (mr)
हिंदी (hi)

Data retention summary
Get the mobile app
Give feedback about this software

Started on	Saturday, 12 February 2022, 10:00:21 AM
State	Finished
Completed on	Saturday, 12 February 2022, 11:25:53 AM
Time taken	1 hour 25 mins
Grade	**4.94** out of 10.00 (**49**%)

Question **1**

Complete

Mark 0.00 out of 0.50

Select all the correct statements about code of bootmain() in xv6

```
void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* pa;

  elf = (struct elfhdr*)0x10000;  // scratch space

  // Read 1st page off disk
  readseg((uchar*)elf, 4096, 0);

  // Is this an ELF executable?
  if(elf->magic != ELF_MAGIC)
    return;  // let bootasm.S handle error

  // Load each program segment (ignores ph flags).
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
  for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
  }

  // Call the entry point from the ELF header.
  // Does not return!
  entry = (void(*)(void))(elf->entry);
  entry();
}
```

Also, inspect the relevant parts of the xv6 code. binary files, etc and run commands as you deem fit to answer this question.

- [x] a. The elf->entry is set by the linker in the kernel file and it's 0x80000000
- [ ] b. The condition if(ph->memsz > ph->filesz) is never true.
- [x] c. The readseg finally invokes the disk I/O code using assembly instructions
- [x] d. The elf->entry is set by the linker in the kernel file and it's 0x80000000
- [ ] e. The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded
- [x] f. The kernel file gets loaded at the Physical address 0x10000 +0x80000000 in memory.
- [x] g. The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it.
- [ ] h. The elf->entry is set by the linker in the kernel file and it's 8010000c
- [ ] i. The kernel file gets loaded at the Physical address 0x10000 in memory.
- [ ] j. The stosb() is used here, to fill in some space in memory with zeroes
- [ ] k. The kernel file has only two program headers

The correct answers are: The kernel file gets loaded at the Physical address 0x10000 in memory., The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it., The elf->entry is set by the linker in the kernel file and it's 8010000c, The readseg finally invokes the disk I/O code using assembly instructions, The stosb() is used here, to fill in some space in memory with zeroes, The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded, The kernel file has only two program headers

Question **2**

Complete

Mark 0.50 out of 0.50

What's the trapframe in xv6?

- a. A frame of memory that contains all the trap handler's addresses
- b. A frame of memory that contains all the trap handler code's function pointers
- c. The IDT table
- d. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S
- e. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by code in trapasm.S only
- f. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware only
- g. A frame of memory that contains all the trap handler code

The correct answer is: The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S

Question **3**

Complete

Mark 0.21 out of 0.50

Order the events that occur on a timer interrupt:

Jump to a code pointed by IDT	3
Jump to scheduler code	2
Select another process for execution	5
Change to kernel stack of currently running process	4
Set the context of the new process	6
Save the context of the currently running process	1
Execute the code of the new process	7

The correct answer is: Jump to a code pointed by IDT → 2, Jump to scheduler code → 4, Select another process for execution → 5, Change to kernel stack of currently running process → 1, Set the context of the new process → 6, Save the context of the currently running process → 3, Execute the code of the new process → 7

Question **4**

Complete

Mark 0.50 out of 0.50

Suppose a program does a scanf() call.

Essentially the scanf does a read() system call.

This call will obviously "block" waiting for the user input.

In terms of OS data structures and execution of code, what does it mean?

Select one:

- a. OS code for read() will move PCB of current process to a wait queue and call scheduler
- b. OS code for read() will call scheduler
- c. OS code for read() will move the PCB of this process to a wait queue and return from the system call
- d. read() will return and process will be taken to a wait queue
- e. read() returns and process calls scheduler()

The correct answer is: OS code for read() will move PCB of current process to a wait queue and call scheduler

Question **5**

Complete

Mark 0.50 out of 0.50

In bootasm.S, on the line

```
ljmp    $(SEG_KCODE<<3), $start32
```

The SEG_KCODE << 3, that is shifting of 1 by 3 bits is done because

- a. The value 8 is stored in code segment
- b. The code segment is 16 bit and only upper 13 bits are used for segment number
- c. While indexing the GDT using CS, the value in CS is always divided by 8
- d. The ljmp instruction does a divide by 8 on the first argument
- e. The code segment is 16 bit and only lower 13 bits are used for segment number

The correct answer is: The code segment is 16 bit and only upper 13 bits are used for segment number

Question **6**

Complete

Mark 0.40 out of 0.50

Select Yes if the mentioned element should be a part of PCB

Select No otherwise.

Yes	No	
◉	○	Memory management information about that process
◉	○	Process context
○	◉	Function pointers to all system calls
◉	○	PID
◉	○	EIP at the time of context switch
◉	○	List of opened files
◉	○	PID of Init
◉	○	Process state
○	◉	Pointer to the parent process
○	◉	Pointer to IDT

Memory management information about that process: Yes
Process context: Yes
Function pointers to all system calls: No
PID: Yes
EIP at the time of context switch: Yes
List of opened files: Yes
PID of Init: No
Process state: Yes
Pointer to the parent process: Yes
Pointer to IDT: No

Question **7**

Complete

Mark 0.40 out of 1.00

Mark the statements, w.r.t. the scheduler of xv6 as True or False

True	False	
○	●	The work of selecting and scheduling a process is done only in `scheduler()` and not in `sched()`
●	○	`swtch` is a function that saves old context, loads new context, and returns to last EIP in the new context
○	●	The variable c->scheduler on first processor uses the stack allocated entry.S
●	○	`sched()` and `scheduler()` are co-routines
○	●	The function `scheduler()` executes using the kernel-only stack
○	●	When a process is scheduled for execution, it resumes execution in `sched()` after the call to `swtch()`
○	●	`swtch` is a function that does not return to the caller
○	●	the control returns to `mycpu()->intena = intena; ();` after `swtch(&p->context, mycpu()->scheduler);` in `sched()`
○	●	the control returns to `switchkvm();` after `swtch(&(c->scheduler), p->context);` in `scheduler()`
●	○	`sched()` *calls* `scheduler()` and `scheduler()` *calls* `sched()`

The work of selecting and scheduling a process is done only in `scheduler()` and not in `sched()`: True
`swtch` is a function that saves old context, loads new context, and returns to last EIP in the new context: True
The variable c->scheduler on first processor uses the stack allocated entry.S: True
`sched()` and `scheduler()` are co-routines: True
The function `scheduler()` executes using the kernel-only stack: True
When a process is scheduled for execution, it resumes execution in `sched()` after the call to `swtch()`
: True
`swtch` is a function that does not return to the caller: True
the control returns to `mycpu()->intena = intena; ();` after `swtch(&p->context, mycpu()->scheduler);` in `sched()`: False
the control returns to `switchkvm();` after `swtch(&(c->scheduler), p->context);` in `scheduler()`: False
`sched()` *calls* `scheduler()` and `scheduler()` *calls* `sched()`: False

Question **8**

Complete

Mark 0.00 out of 0.50

Consider the following programs

exec1.c

```
#include <unistd.h>
#include <stdio.h>
int main() {
   execl("./exec2", "./exec2", NULL);
}
```

exec2.c

```
#include <unistd.h>
#include <stdio.h>
int main() {
   execl("/bin/ls", "/bin/ls", NULL);
  printf("hello\n");
}
```

Compiled as

```
cc    exec1.c   -o exec1
cc    exec2.c   -o exec2
```

And run as

```
$ ./exec1
```

Explain the output of the above command (./exec1)

Assume that /bin/ls , i.e. the 'ls' program exists.

Select one:

- a. "ls" runs on current directory
- b. Execution fails as the call to execl() in exec1 fails
- c. Execution fails as one exec can't invoke another exec
- d. Program prints hello
- e. Execution fails as the call to execl() in exec2 fails

The correct answer is: "ls" runs on current directory

Question **9**

Complete

Mark 0.00 out of 0.50

For each line of code mentioned on the left side, select the location of sp/esp that is in use

cli in bootasm.S	0x7c00 to 0
readseg((uchar*)elf, 4096, 0); in bootmain.c	Immaterial as the stack is not used here
ljmp $(SEG_KCODE<<3), $start32 in bootasm.S	0x10000 to 0x7c00
call bootmain in bootasm.S	The 4KB area in kernel image, loaded in memory, named as 'stack'
jmp *%eax in entry.S	0x7c00 to 0x10000

The correct answer is: cli
in bootasm.S → Immaterial as the stack is not used here, readseg((uchar*)elf, 4096, 0);
in bootmain.c → 0x7c00 to 0, ljmp $(SEG_KCODE<<3), $start32
in bootasm.S → Immaterial as the stack is not used here, call bootmain
in bootasm.S → 0x7c00 to 0, jmp *%eax
in entry.S → The 4KB area in kernel image, loaded in memory, named as 'stack'

Question **10**

Complete

Mark 0.63 out of 1.00

Select the correct statements about interrupt handling in xv6 code

- [x] a. xv6 uses the 64th entry in IDT for system calls
- [ ] b. xv6 uses the 0x64th entry in IDT for system calls
- [x] c. On any interrupt/syscall/exception the control first jumps in vectors.S
- [ ] d. The function trap() is the called irrespective of hardware interrupt/system-call/exception
- [x] e. Before going to alltraps, the kernel stack contains upto 5 entries.
- [ ] f. Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt
- [x] g. The trapframe pointer in struct proc, points to a location on kernel stack
- [ ] h. The function trap() is the called only in case of hardware interrupt
- [ ] i. The trapframe pointer in struct proc, points to a location on user stack
- [ ] j. On any interrupt/syscall/exception the control first jumps in trapasm.S
- [x] k. The CS and EIP are changed only after pushing user code's SS,ESP on stack
- [ ] l. The CS and EIP are changed only immediately on a hardware interrupt
- [ ] m. All the 256 entries in the IDT are filled

The correct answers are: All the 256 entries in the IDT are filled, Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt, xv6 uses the 64th entry in IDT for system calls, On any interrupt/syscall/exception the control first jumps in vectors.S, Before going to alltraps, the kernel stack contains upto 5 entries., The trapframe pointer in struct proc, points to a location on kernel stack, The function trap() is the called irrespective of hardware interrupt/system-call/exception, The CS and EIP are changed only after pushing user code's SS,ESP on stack

Question **11**

Complete

Mark 0.50 out of 0.50

Order the sequence of events, in scheduling process P1 after process P0

timer interrupt occurs	2
Process P0 is running	1
context of P1 is loaded from P1's PCB	4
Process P1 is running	6
Control is passed to P1	5
context of P0 is saved in P0's PCB	3

The correct answer is: timer interrupt occurs → 2, Process P0 is running → 1, context of P1 is loaded from P1's PCB → 4, Process P1 is running → 6, Control is passed to P1 → 5, context of P0 is saved in P0's PCB → 3

Question **12**

Complete

Mark 0.00 out of 1.00

Select the sequence of events that are NOT possible, assuming a non-interruptible kernel code

(Note: non-interruptible kernel code means, if the kernel code is executing, then interrupts will be disabled).

Note: A possible sequence may have some missing steps in between. An impossible sequence will will have n and n+1th steps such that n+1th step can not follow n'th step.

Select one or more:

- [ ] a. P1 running
  P1 makes system call
  timer interrupt
  Scheduler
  P2 running
  timer interrupt
  Scheuler
  P1 running
  P1's system call return

- [ ] b. P1 running
  P1 makes system call
  system call returns
  P1 running
  timer interrupt
  Scheduler running
  P2 running

- [ ] c. P1 running
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P3 running
  Hardware interrupt
  Interrupt unblocks P1
  Interrupt returns
  P3 running
  Timer interrupt
  Scheduler
  P1 running

- [x] d. P1 running
  keyboard hardware interrupt
  keyboard interrupt handler running
  interrupt handler returns
  P1 running
  P1 makes sytem call
  system call returns
  P1 running
  timer interrupt
  scheduler
  P2 running

- [ ] e.
  P1 running
  P1 makes sytem call
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again

- [ ] f. P1 running
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again

The correct answers are: P1 running
P1 makes sytem call and blocks
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again, P1 running
P1 makes system call
timer interrupt
Scheduler
P2 running
timer interrupt
Scheuler
P1 running
P1's system call return,
P1 running
P1 makes sytem call
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

Question **13**

Complete

Mark 0.50 out of 0.50

Some part of the bootloader of xv6 is written in assembly while some part is written in C. Why is that so?
Select all the appropriate choices

- [x] a. The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C
- [x] b. The setting up of the most essential memory management infrastructure needs assembly code
- [ ] c. The code in assembly is required for transition to protected mode, from real mode; but calling convention was applicable all the time
- [ ] d. The code for reading ELF file can not be written in assembly

The correct answers are: The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C, The setting up of the most essential memory management infrastructure needs assembly code

Question **14**

Complete

Mark 0.17 out of 0.50

The bootmain() function has this code

```
elf = (struct elfhdr*)0x10000;  // scratch space
readseg((uchar*)elf, 4096, 0);
```

Mark the statements as True or False with respect to this code.

In these statements 0x1000 is referred to as ADDRESS

True	False	
◉	○	The value ADDRESS is changed to a 0 the program could still work
◉	○	It the value of ADDRESS is changed to a higher number (upto a limit), the program could still work
◉	○	This line loads the kernel code at ADDRESS
◉	○	It the value of ADDRESS is changed to a lower number (upto a limit), the program could still work
◉	○	This line effectively loads the ELF header and the program headers at ADDRESS
◉	○	If the value of ADDRESS is changed, then the program will not work

The value ADDRESS is changed to a 0 the program could still work: False
It the value of ADDRESS is changed to a higher number (upto a limit), the program could still work: True
This line loads the kernel code at ADDRESS: False
It the value of ADDRESS is changed to a lower number (upto a limit), the program could still work: True
This line effectively loads the ELF header and the program headers at ADDRESS: False
If the value of ADDRESS is changed, then the program will not work: False

Question **15**

Complete

Mark 0.50 out of 1.00

Which parts of the xv6 code in bootasm.S bootmain.c , entry.S and in the codepath related to scheduler() and trap handling() can also be written in some other way, and still ensure that xv6 works properly?

Writing code is not necessary. You only need to comment on which part of the code could be changed to something else or written in another fashion.

Maximum two points to be written.

We can use a scheduling algorithm. We can use the kernel stack in scheduler function in entrty.S and bootmain.c .

Question **16**

Complete

Mark 0.13 out of 0.50

Select all the correct statements about zombie processes

Select one or more:

- [ ] a. If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent
- [ ] b. Zombie processes are harmless even if OS is up for long time
- [x] c. A zombie process occupies space in OS data structures
- [ ] d. A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it
- [x] e. A process can become zombie if it finishes, but the parent has finished before it
- [x] f. init() typically keeps calling wait() for zombie processes to get cleaned up
- [ ] g. A process becomes zombie when it's parent finishes
- [x] h. A zombie process remains zombie forever, as there is no way to clean it up

The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it, A process can become zombie if it finishes, but the parent has finished before it, A zombie process occupies space in OS data structures, If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

◄ Extra Reading on Linkers: A writeup by Ian Taylor (keep changing url string from 38 to 39, and so on)

Jump to...

(Code) IPC - Shm, Messages ►

Started on	Saturday, 20 February 2021, 2:51 PM
State	Finished
Completed on	Saturday, 20 February 2021, 3:55 PM
Time taken	1 hour 3 mins
Grade	**7.30** out of 20.00 (**37**%)

Question **1**

Partially correct

Mark 0.80 out of 1.00

Select all the correct statements about the state of a process.

- [x] a. A process can self-terminate only when it's running ✔
- [x] b. Typically, it's represented as a number in the PCB ✔
- [x] c. A process that is running is not on the ready queue ✔
- [x] d. Processes in the ready queue are in the ready state ✔
- [ ] e. It is not maintained in the data structures by kernel, it is only for conceptual understanding of programmers
- [x] f. Changing from running state to waiting state results in "giving up the CPU" ✔
- [ ] g. A process in ready state is ready to receive interrupts
- [x] h. A waiting process starts running after the wait is over ✖
- [x] i. A process changes from running to ready state on a timer interrupt ✔
- [x] j. A process in ready state is ready to be scheduled ✔
- [x] k. A running process may terminate, or go to wait or become ready again ✔
- [x] l. A process waiting for I/O completion is typically woken up by the particular interrupt handler code ✔
- [ ] m. A process waiting for any condition is woken up by another process only
- [ ] n. A process changes from running to ready state on a timer interrupt or any I/O wait

Your answer is partially correct.

You have selected too many options.
The correct answers are: Typically, it's represented as a number in the PCB, A process in ready state is ready to be scheduled, Processes in the ready queue are in the ready state, A process that is running is not on the ready queue, A running process may terminate, or go to wait or become ready again, A process changes from running to ready state on a timer interrupt, Changing from running state to waiting state results in "giving up the CPU", A process can self-terminate only when it's running, A process waiting for I/O completion is typically woken up by the particular interrupt handler code

Question **2**

Incorrect

Mark 0.00 out of 1.00

For each line of code mentioned on the left side, select the location of sp/esp that is in use

jmp *%eax in entry.S	0x7c00 to 0x10000	✖
ljmp $(SEG_KCODE<<3), $start32 in bootasm.S	0x10000 to 0x7c00	✖
call bootmain in bootasm.S	0x7c00 to 0x10000	✖
cli in bootasm.S	0x7c00 to 0	✖
readseg((uchar*)elf, 4096, 0); in bootmain.c	The 4KB area in kernel image, loaded in memory, named as 'stack'	✖

Your answer is incorrect.

The correct answer is: jmp *%eax in entry.S → The 4KB area in kernel image, loaded in memory, named as 'stack', ljmp $(SEG_KCODE<<3), $start32 in bootasm.S → Immaterial as the stack is not used here, call bootmain in bootasm.S → 0x7c00 to 0, cli in bootasm.S → Immaterial as the stack is not used here, readseg((uchar*)elf, 4096, 0); in bootmain.c → 0x7c00 to 0

Question **3**

Correct

Mark 0.25 out of 0.25

Order the following events in boot process (from 1 onwards)

Boot loader	2	✔
Shell	6	✔
BIOS	1	✔
OS	3	✔
Init	4	✔
Login interface	5	✔

Your answer is correct.

The correct answer is: Boot loader → 2, Shell → 6, BIOS → 1, OS → 3, Init → 4, Login interface → 5

Question **4**

Partially correct

Mark 0.30 out of 0.50

Consider the following command and it's output:

```
$ ls -lht xv6.img kernel
-rw-rw-r-- 1 abhijit abhijit 4.9M Feb 15 11:09 xv6.img
-rwxrwxr-x 1 abhijit abhijit 209K Feb 15 11:09 kernel*
```

Following code in bootmain()

```
  readseg((uchar*)elf, 4096, 0);
```

and following selected lines from Makefile

```
xv6.img: bootblock kernel
    dd if=/dev/zero of=xv6.img count=10000
    dd if=bootblock of=xv6.img conv=notrunc
    dd if=kernel of=xv6.img seek=1 conv=notrunc

kernel: $(OBJS) entry.o entryother initcode kernel.ld
    $(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode entryother
    $(OBJDUMP) -S kernel > kernel.asm
    $(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > kernel.sym
```

Also read the code of bootmain() in xv6 kernel.
Select the options that describe the meaning of these lines and their correlation.

- [ ] a. Althought the size of the kernel file is 209 Kb, only 4Kb out of it is the actual kernel code and remaining part is all zeroes.
- [x] b. The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files ✔
- [x] c. The kernel.ld file contains instructions to the linker to link the kernel properly ✔
- [ ] d. The bootmain() code does not read the kernel completely in memory
- [ ] e. readseg() reads first 4k bytes of kernel in memory
- [ ] f. Althought the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.
- [ ] g. The kernel.asm file is the final kernel file
- [ ] h. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is not read as it is user programs.
- [x] i. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain(). ✔

Your answer is partially correct.

You have correctly selected 3.
The correct answers are: The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain()., readseg() reads first 4k bytes of kernel in memory, The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files, The kernel.ld file contains instructions to the linker to link the kernel properly, Althought the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.

Question **5**

Partially correct

Mark 0.50 out of 1.00

```
int f() {
    int count;
    for (count = 0; count< 2; count ++) {
        if (fork() ==0)
            printf("Operating-System\n");
     }
     printf("TYCOMP\n");
 }
```

The number of times "Operating-System" is printed, is:

Answer: 3

The correct answer is: 7.00

Question **6**

Partially correct

Mark 0.40 out of 0.50

Select Yes/True if the mentioned element must be a part of PCB

Select No/False otherwise.

Yes	No		
◉☑	○✗	PID	✔
◉☑	○✗	Process context	✔
◉☑	○✗	List of opened files	✔
◉☑	○✗	Process state	✔
◉✗	○☑	Parent's PID	✖
○✗	◉☑	Pointer to IDT	✔
○✗	◉☑	Function pointers to all system calls	✔
◉☑	○✗	Memory management information about that process	✔
○☑	◉✗	Pointer to the parent process	✖
◉☑	○✗	EIP at the time of context switch	✔

PID: Yes
Process context: Yes
List of opened files: Yes
Process state: Yes
Parent's PID: No
Pointer to IDT: No
Function pointers to all system calls: No
Memory management information about that process: Yes
Pointer to the parent process: Yes
EIP at the time of context switch: Yes

Question **7**

Incorrect

Mark 0.00 out of 1.00

Select all the correct statements about code of bootmain() in xv6

```
void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* pa;

  elf = (struct elfhdr*)0x10000;  // scratch space

  // Read 1st page off disk
  readseg((uchar*)elf, 4096, 0);

  // Is this an ELF executable?
  if(elf->magic != ELF_MAGIC)
    return;  // let bootasm.S handle error

  // Load each program segment (ignores ph flags).
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
  for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
  }

  // Call the entry point from the ELF header.
  // Does not return!
  entry = (void(*)(void))(elf->entry);
  entry();
}
```

Also, inspect the relevant parts of the xv6 code. binary files, etc and run commands as you deem fit to answer this question.

- a. The kernel file gets loaded at the Physical address 0x10000 +0x80000000 in memory. ✗
- b. The elf->entry is set by the linker in the kernel file and it's 0x80000000 ✗
- c. The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded ✓
- d. The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it. ✓
- e. The kernel file has only two program headers ✓
- f. The elf->entry is set by the linker in the kernel file and it's 0x80000000 ✗
- g. The readseg finally invokes the disk I/O code using assembly instructions ✓
- h. The elf->entry is set by the linker in the kernel file and it's 8010000c ✓
- i. The kernel file gets loaded at the Physical address 0x10000 in memory. ✓
- j. The condition if(ph->memsz > ph->filesz) is never true. ✗
- k. The stosb() is used here, to fill in some space in memory with zeroes ✓

Your answer is incorrect.

The correct answers are: The kernel file gets loaded at the Physical address 0x10000 in memory., The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it., The elf->entry is set by the linker in the kernel file and it's 8010000c, The readseg finally invokes the disk I/O code using assembly instructions, The stosb() is used here, to fill in some space in memory with zeroes, The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded, The kernel file has only two program headers

Question **8**

Partially correct

Mark 0.13 out of 0.25

Which of the following are NOT a part of job of a typical compiler?

- [x] a. Check the program for logical errors ✔
- [ ] b. Convert high level langauge code to machine code
- [ ] c. Process the # directives in a C program
- [ ] d. Invoke the linker to link the function calls with their code, extern globals with their declaration
- [ ] e. Check the program for syntactical errors
- [ ] f. Suggest alternative pieces of code that can be written

Your answer is partially correct.

You have correctly selected 1.
The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

Question **9**

Correct

Mark 0.25 out of 0.25

Rank the following storage systems from slowest (first) to fastest(last)

Cache	6	✔
Hard Disk	3	✔
RAM	5	✔
Optical Disks	2	✔
Non volatile memory	4	✔
Registers	7	✔
Magnetic Tapes	1	✔

Your answer is correct.

The correct answer is: Cache → 6, Hard Disk → 3, RAM → 5, Optical Disks → 2, Non volatile memory → 4, Registers → 7, Magnetic Tapes → 1

Question **10**

Partially correct

Mark 0.21 out of 0.50

Which of the following parts of a C program do not have any corresponding machine code ?

- [ ] a. local variable declaration
- [ ] b. global variables
- [x] c. function calls ✗
- [x] d. #directives ✓
- [ ] e. expressions
- [ ] f. pointer dereference
- [x] g. typedefs ✓

Your answer is partially correct.

You have correctly selected 2.
The correct answers are: #directives, typedefs, global variables

Question **11**

Correct

Mark 0.25 out of 0.25

Match a system call with it's description

pipe	create an unnamed FIFO storage with 2 ends - one for reading and another for writing	✓
dup	create a copy of the specified file descriptor into smallest available file descriptor	✓
dup2	create a copy of the specified file descriptor into another specified file descriptor	✓
exec	execute a binary file overlaying the image of current process	✓
fork	create an identical child process	✓

Your answer is correct.

The correct answer is: pipe → create an unnamed FIFO storage with 2 ends - one for reading and another for writing, dup → create a copy of the specified file descriptor into smallest available file descriptor, dup2 → create a copy of the specified file descriptor into another specified file descriptor, exec → execute a binary file overlaying the image of current process, fork → create an identical child process

Question **12**

Correct

Mark 0.25 out of 0.25

Match the register with the segment used with it.

eip	cs	✔
edi	es	✔
esi	ds	✔
ebp	ss	✔
esp	ss	✔

Your answer is correct.

The correct answer is: eip → cs, edi → es, esi → ds, ebp → ss, esp → ss

Question **13**

Correct

Mark 0.25 out of 0.25

What's the trapframe in xv6?

- a. A frame of memory that contains all the trap handler code
- b. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware only
- c. The IDT table
- d. A frame of memory that contains all the trap handler code's function pointers
- e. A frame of memory that contains all the trap handler's addresses
- f. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S ✔
- g. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by code in trapasm.S only

Your answer is correct.

The correct answer is: The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S

Question **14**

Incorrect

Mark 0.00 out of 0.50

Select all the correct statements about linking and loading.

Select one or more:

- a. Continuous memory management schemes can support dynamic linking and dynamic loading. ✗
- b. Loader is last stage of the linker program ✗
- c. Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently) ✓
- d. Dynamic linking and loading is not possible without demand paging or demand segmentation. ✓
- e. Dynamic linking essentially results in relocatable code. ✓
- f. Continuous memory management schemes can support static linking and static loading. (may be inefficiently) ✓
- g. Loader is part of the operating system ✓
- h. Static linking leads to non-relocatable code ✗
- i. Dynamic linking is possible with continous memory management, but variable sized partitions only. ✗

Your answer is incorrect.

The correct answers are: Continuous memory management schemes can support static linking and static loading. (may be inefficiently), Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently), Dynamic linking essentially results in relocatable code., Loader is part of the operating system, Dynamic linking and loading is not possible without demand paging or demand segmentation.

Question **15**

Incorrect

Mark 0.00 out of 0.25

In bootasm.S, on the line

```
ljmp    $(SEG_KCODE<<3), $start32
```

The SEG_KCODE << 3, that is shifting of 1 by 3 bits is done because

- a. The value 8 is stored in code segment
- b. The code segment is 16 bit and only upper 13 bits are used for segment number
- c. The code segment is 16 bit and only lower 13 bits are used for segment number ✗
- d. While indexing the GDT using CS, the value in CS is always divided by 8
- e. The ljmp instruction does a divide by 8 on the first argument

Your answer is incorrect.

The correct answer is: The code segment is 16 bit and only upper 13 bits are used for segment number

Question **16**

Partially correct

Mark 0.07 out of 0.50

Order the events that occur on a timer interrupt:

Change to kernel stack	1	✗
Jump to a code pointed by IDT	2	✗
Jump to scheduler code	5	✗
Set the context of the new process	4	✗
Save the context of the currently running process	3	✓
Execute the code of the new process	6	✗
Select another process for execution	7	✗

Your answer is partially correct.

You have correctly selected 1.

The correct answer is: Change to kernel stack → 2, Jump to a code pointed by IDT → 1, Jump to scheduler code → 4, Set the context of the new process → 6, Save the context of the currently running process → 3, Execute the code of the new process → 7, Select another process for execution → 5

Question **17**

Incorrect

Mark 0.00 out of 1.00

Consider the two programs given below to implement the command (ignore the fact that error checks are not done on return values of functions)

$ ls . /tmp/asdfksdf >/tmp/ddd 2>&1

Program 1

```
int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(1);
    dup(fd);
    close(2);
    dup(fd);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Program 2

```
int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    close(1);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(2);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Select all the correct statements about the programs

Select one or more:

- a. Both programs are correct ✗
- b. Program 2 makes sure that there is one file offset used for '2' and '1' ✗
- c. Only Program 2 is correct ✗
- d. Program 2 does 1>&2 ✗
- e. Program 2 ensures 2>&1 and does not ensure > /tmp/ddd ✗
- f. Program 1 makes sure that there is one file offset used for '2' and '1' ✓
- g. Program 1 is correct for > /tmp/ddd but not for 2>&1 ✗
- h. Program 1 does 1>&2 ✗
- i. Both program 1 and 2 are incorrect ✗
- j. Program 2 is correct for > /tmp/ddd but not for 2>&1 ✗
- k. Only Program 1 is correct ✓
- l. Program 1 ensures 2>&1 and does not ensure > /tmp/ddd ✗

Your answer is incorrect.

The correct answers are: Only Program 1 is correct, Program 1 makes sure that there is one file offset used for '2' and '1'

Question **18**

Correct

Mark 0.25 out of 0.25

Select the option which best describes what the CPU does during it's powered ON lifetime

- a. Ask the user what is to be done, and execute that task
- b. Ask the OS what is to be done, and execute that task
- c. Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, Ask the User or the OS what is to be done next, repeat
- d. Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, repeat ✔
- e. Fetch instruction specified by OS, Decode and execute it, repeat
- f. Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, Ask OS what is to be done next, repeat

The correct answer is: Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, repeat

Question **19**

Partially correct

Mark 0.86 out of 1.00

Consider the following code and MAP the file to which each fd points at the end of the code.

```
int main(int argc, char *argv[]) {
   int fd1, fd2 = 1, fd3 = 1, fd4 = 1;

   fd1 = open("/tmp/1", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
   fd2 = open("/tmp/2", O_RDDONLY);
   fd3 = open("/tmp/3", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
   close(0);
   close(1);
   dup(fd2);
   dup(fd3);
   close(fd3);
   dup2(fd2, fd4);
   printf("%d %d %d %d\n", fd1, fd2, fd3, fd4);
   return 0;
}
```

1	closed	✖
fd4	/tmp/2	✔
fd2	/tmp/2	✔
fd1	/tmp/1	✔
2	stderr	✔
0	/tmp/2	✔
fd3	closed	✔

Your answer is partially correct.

You have correctly selected 6.

The correct answer is: 1 → /tmp/3, fd4 → /tmp/2, fd2 → /tmp/2, fd1 → /tmp/1, 2 → stderr, 0 → /tmp/2, fd3 → closed

Question **20**

Incorrect

Mark 0.00 out of 2.00

Following code claims to implement the command

/bin/ls -l | /usr/bin/head -3 | /usr/bin/tail -1

Fill in the blanks to make the code work.

Note: Do not include space in writing any option. x[1][2] should be written without any space, and so is the case with [1] or [2]. Pay attention to exact syntax and do not write any extra character like ';' or = etc.

```
int main(int argc, char *argv[]) {
   int pid1, pid2;
   int pfd[
```

1

✖][2];

```
   pipe(
```

2

✖);

```
   pid1 =
```

3

✖ ;

```
   if(pid1 != 0) {
      close(pfd[0]
```

0

✖);

```
      close(
```

pid1

✖);

```
      dup(
```

pid2

✖);

```
      execl("/bin/ls", "/bin/ls", "
```

1

✖ ", NULL);

```
   }
   pipe(
```

✖);

✖ = fork();

```
   if(pid2 == 0) {
      close(
```

✖ ;

```
      close(0);
      dup(
```

✖);

```
      close(pfd[1]
```

```
✖ );
    close(
✖ );
    dup(
✖ );
    execl("/usr/bin/head", "/usr/bin/head", "
✖ ", NULL);
  } else {
    close(pfd
✖ );
    close(
✖ );
    dup(
✖ );
    close(pfd
✖ );
    execl("/usr/bin/tail", "/usr/bin/tail", "
✖ ", NULL);
 }
}
```

Question **21**

Partially correct

Mark 0.11 out of 1.00

Select all the correct statements about calling convention on x86 32-bit.

- [x] a. Return address is one location above the ebp ✔
- [x] b. Parameters may be passed in registers or on stack ✔
- [x] c. Space for local variables is allocated by substracting the stack pointer inside the code of the called function ✔
- [x] d. The ebp pointers saved on the stack constitute a chain of activation records ✔
- [x] e. The two lines in the beginning of each function, "push %ebp; mov %esp, %ebp", create space for local variables ✖
- [x] f. Parameters may be passed in registers or on stack ✔
- [x] g. The return value is either stored on the stack or returned in the eax register ✖
- [ ] h. Paramters are pushed on the stack in left-right order
- [ ] i. during execution of a function, ebp is pointing to the old ebp
- [x] j. Space for local variables is allocated by substracting the stack pointer inside the code of the caller function ✖
- [x] k. Compiler may allocate more memory on stack than needed ✔

Your answer is partially correct.

You have selected too many options.
The correct answers are: Compiler may allocate more memory on stack than needed, Parameters may be passed in registers or on stack, Parameters may be passed in registers or on stack, Return address is one location above the ebp, during execution of a function, ebp is pointing to the old ebp, Space for local variables is allocated by substracting the stack pointer inside the code of the called function, The ebp pointers saved on the stack constitute a chain of activation records

Question **22**

Correct

Mark 1.00 out of 1.00

Match the program with it's output (ignore newlines in the output. Just focus on the count of the number of 'hi')

main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }	hi	✔
main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }	hi hi	✔
main() { int i = NULL; fork(); printf("hi\n"); }	hi hi	✔
main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }	hi	✔

Your answer is correct.

The correct answer is: main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi, main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi hi, main() { int i = NULL; fork(); printf("hi\n"); } → hi hi, main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi

Question **23**

Incorrect

Mark 0.00 out of 0.50

Some part of the bootloader of xv6 is written in assembly while some part is written in C. Why is that so?
Select all the appropriate choices

- a. The code in assembly is required for transition to protected mode, from real mode; but calling convention was applicable all the time ✗
- b. The setting up of the most essential memory management infrastructure needs assembly code ✓
- c. The code for reading ELF file can not be written in assembly ✗
- d. The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C ✓

Your answer is incorrect.

The correct answers are: The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C, The setting up of the most essential memory management infrastructure needs assembly code

Question **24**

Incorrect

Mark 0.00 out of 0.50

xv6.img: bootblock kernel

```
    dd if=/dev/zero of=xv6.img count=10000
    dd if=bootblock of=xv6.img conv=notrunc
    dd if=kernel of=xv6.img seek=1 conv=notrunc
```

Consider above lines from the Makefile. Which of the following is incorrect?

- [x] a. The size of the kernel file is nearly 5 MB ✔
- [x] b. The kernel is located at block-1 of the xv6.img ✖
- [x] c. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies 10,000 blocks on the disk. ✖
- [ ] d. The size of xv6.img is exactly = (size of bootblock) + (size of kernel)
- [x] e. The bootblock is located on block-0 of the xv6.img ✖
- [x] f. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk. ✔
- [x] g. The bootblock may be 512 bytes or less (looking at the Makefile instruction) ✖
- [x] h. The xv6.img is the virtual disk that is created by combining the bootblock and the kernel file. ✖
- [x] i. The size of the xv6.img is nearly 5 MB ✖
- [x] j. xv6.img is the virtual processor used by the qemu emulator ✔
- [x] k. Blocks in xv6.img after kernel may be all zeroes. ✖

Your answer is incorrect.

The correct answers are: xv6.img is the virtual processor used by the qemu emulator, The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk., The size of the kernel file is nearly 5 MB, The size of xv6.img is exactly = (size of bootblock) + (size of kernel)

Question **25**

Incorrect

Mark 0.00 out of 1.00

Select the sequence of events that are NOT possible, assuming a non-interruptible kernel code

Select one or more:

- [ ] a. P1 running
  P1 makes system call
  timer interrupt
  Scheduler
  P2 running
  timer interrupt
  Scheuler
  P1 running
  P1's system call return

- [x] b. P1 running ✔
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again

- [ ] c. P1 running
  P1 makes system call
  system call returns
  P1 running
  timer interrupt
  Scheduler running
  P2 running

- [x] d. P1 running ✖
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P3 running
  Hardware interrupt
  Interrupt unblocks P1
  Interrupt returns
  P3 running
  Timer interrupt
  Scheduler
  P1 running

- [ ] e.
  P1 running
  P1 makes sytem call
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again

- [x] f. P1 running ✖
  keyboard hardware interrupt
  keyboard interrupt handler running
  interrupt handler returns
  P1 running
  P1 makes sytem call
  system call returns

P1 running
timer interrupt
scheduler
P2 running

Your answer is incorrect.

The correct answers are: P1 running
P1 makes sytem call and blocks
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again, P1 running
P1 makes system call
timer interrupt
Scheduler
P2 running
timer interrupt
Scheuler
P1 running
P1's system call return,
P1 running
P1 makes sytem call
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

Question **26**

Correct

Mark 0.25 out of 0.25

Which of the following are the files related to bootloader in xv6?

- a. bootasm.s and entry.S
- b. bootasm.S and bootmain.c ✔
- c. bootasm.S, bootmain.c and bootblock.c
- d. bootmain.c and bootblock.S

Your answer is correct.

The correct answer is: bootasm.S and bootmain.c

Question **27**

Correct

Mark 0.25 out of 0.25

Match the following parts of a C program to the layout of the process in memory

Instructions	Text section	✔
Local Variables	Stack Section	✔
Dynamically allocated memory	Heap Section	✔
Global and static data	Data section	✔

Your answer is correct.

The correct answer is:
Instructions → Text section, Local Variables → Stack Section,
Dynamically allocated memory → Heap Section,
Global and static data → Data section

Question **28**

Incorrect

Mark 0.00 out of 0.50

What will this program do?

```
int main() {
fork();
execl("/bin/ls", "/bin/ls", NULL);
printf("hello");
}
```

- a. one process will run ls, another will print hello
- b. run ls once ✖
- c. run ls twice
- d. run ls twice and print hello twice
- e. run ls twice and print hello twice, but output will appear in some random order

Your answer is incorrect.

The correct answer is: run ls twice

Question **29**

Correct

Mark 0.25 out of 0.25

What is the OS Kernel?

- a. The code that controls hardware, abstracts access to hardware resources using system calls, creates an environment for processes to be created and run ✔ correct
- b. The set of tools like compiler, linker, loader, terminal, shell, etc.
- c. Only the system programs like compiler, linker, loader, etc.
- d. Everything that I see on my screen

The correct answer is: The code that controls hardware, abstracts access to hardware resources using system calls, creates an environment for processes to be created and run

Question **30**

Correct

Mark 0.50 out of 0.50

Which of the following is/are not saved during context switch?

- [ ] a. Program Counter
- [ ] b. General Purpose Registers
- [x] c. Bus ✔
- [ ] d. Stack Pointer
- [ ] e. MMU related registers/information
- [x] f. Cache ✔
- [x] g. TLB ✔

Your answer is correct.

The correct answers are: TLB, Cache, Bus

Question **31**

Partially correct

Mark 0.10 out of 0.25

Select the order in which the various stages of a compiler execute.

Linking	3	✗
Syntatical Analysis	2	✓
Pre-processing	1	✓
Intermediate code generation	does not exist	✗
Loading	4	✗

Your answer is partially correct.

You have correctly selected 2.
The correct answer is: Linking → 4, Syntatical Analysis → 2, Pre-processing → 1, Intermediate code generation → 3, Loading → does not exist

Question **32**

Partially correct

Mark 0.08 out of 0.50

Order the sequence of events, in scheduling process P1 after process P0

context of P0 is saved in P0's PCB	2	✗
context of P1 is loaded from P1's PCB	3	✗
Process P1 is running	5	✗
timer interrupt occurs	6	✗
Process P0 is running	1	✓
Control is passed to P1	4	✗

Your answer is partially correct.

You have correctly selected 1.
The correct answer is: context of P0 is saved in P0's PCB → 3, context of P1 is loaded from P1's PCB → 4, Process P1 is running → 6, timer interrupt occurs → 2, Process P0 is running → 1, Control is passed to P1 → 5

Question **33**

Not answered

Marked out of 1.00

Select the correct statements about interrupt handling in xv6 code

- a. On any interrupt/syscall/exception the control first jumps in vectors.S
- b. The trapframe pointer in struct proc, points to a location on user stack
- c. Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt
- d. xv6 uses the 64th entry in IDT for system calls
- e. The CS and EIP are changed only after pushing user code's SS,ESP on stack
- f. The trapframe pointer in struct proc, points to a location on kernel stack
- g. The function trap() is the called only in case of hardware interrupt
- h. The CS and EIP are changed only immediately on a hardware interrupt
- i. All the 256 entries in the IDT are filled
- j. On any interrupt/syscall/exception the control first jumps in trapasm.S
- k. The function trap() is the called irrespective of hardware interrupt/system-call/exception
- l. xv6 uses the 0x64th entry in IDT for system calls
- m. Before going to alltraps, the kernel stack contains upto 5 entries.

Your answer is incorrect.

The correct answers are: All the 256 entries in the IDT are filled, Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt, xv6 uses the 64th entry in IDT for system calls, On any interrupt/syscall/exception the control first jumps in vectors.S, Before going to alltraps, the kernel stack contains upto 5 entries., The trapframe pointer in struct proc, points to a location on kernel stack, The function trap() is the called irrespective of hardware interrupt/system-call/exception, The CS and EIP are changed only after pushing user code's SS,ESP on stack

◄ (Assignment) Change free list management in xv6

Jump to...

Started on	Tuesday, 22 March 2022, 1:59:34 PM
State	Finished
Completed on	Tuesday, 22 March 2022, 4:14:53 PM
Time taken	2 hours 15 mins
Grade	**24.57** out of 40.00 (**61**%)

Question **1**

Partially correct

Mark 0.10 out of 1.00

Select all the correct statements w.r.t user and kernel threads

Select one or more:

- [x] a. many-one model can be implemented even if there are no kernel threads ✔
- [ ] b. many-one model gives no speedup on multicore processors
- [x] c. all three models, that is many-one, one-one, many-many , require a user level thread library ✔
- [ ] d. A process blocks in many-one model even if a single thread makes a blocking system call
- [x] e. A process may not block in many-one model, if a thread makes a blocking system call ✖
- [x] f. one-one model increases kernel's scheduling load ✔
- [ ] g. one-one model can be implemented even if there are no kernel threads

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: many-one model can be implemented even if there are no kernel threads, all three models, that is many-one, one-one, many-many , require a user level thread library, one-one model increases kernel's scheduling load, many-one model gives no speedup on multicore processors, A process blocks in many-one model even if a single thread makes a blocking system call

Question **2**

Partially correct

Mark 0.50 out of 1.00

Select all correct statements w.r.t. Major and Minor page faults on Linux

- [ ] a. Thrashing is possible only due to major page faults
- [ ] b. Minor page fault may occur because the page was freed, but still tagged and available in the free page list
- [x] c. Minor page fault may occur because of a page fault during fork(), on code of an already running process ✔
- [x] d. Major page faults are likely to occur in more numbers at the beginning of the process ✔
- [x] e. Minor page fault may occur because the page was a shared memory page ✔
- [ ] f. Minor page faults are an improvement of the page buffering techniques

The correct answers are: Minor page fault may occur because the page was a shared memory page, Minor page fault may occur because of a page fault during fork(), on code of an already running process, Minor page fault may occur because the page was freed, but still tagged and available in the free page list, Major page faults are likely to occur in more numbers at the beginning of the process, Thrashing is possible only due to major page faults, Minor page faults are an improvement of the page buffering techniques

Question **3**

Correct

Mark 2.00 out of 2.00

W.r.t. Memory management in xv6,

xv6 uses physical memory upto 224 MB onlyMark statements True or False

True	False		
◉	○	The stack allocated in entry.S is used as stack for scheduler's context for first processor	✔
○	◉	The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context	✔
◉	○	The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir	✔
◉	○	The kernel code and data take up less than 2 MB space	✔
◉	○	The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context	✔
◉	○	xv6 uses physical memory upto 224 MB only	✔
◉	○	The free page-frame are created out of nearly 222 MB	✔
○	◉	The switchkvm() call in scheduler() changes CR3 to use page directory of new process	✔
◉	○	PHYSTOP can be increased to some extent, simply by editing memlayout.h	✔
◉	○	The process's address space gets mapped on frames, obtained from ~2MB:224MB range	✔
○	◉	The kernel's page table given by kpgdir variable is used as stack for scheduler's context	✔

The stack allocated in entry.S is used as stack for scheduler's context for first processor: True
The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context: False
The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir: True
The kernel code and data take up less than 2 MB space: True
The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context: True
xv6 uses physical memory upto 224 MB only: True
The free page-frame are created out of nearly 222 MB: True
The switchkvm() call in scheduler() changes CR3 to use page directory of new process: False
PHYSTOP can be increased to some extent, simply by editing memlayout.h: True
The process's address space gets mapped on frames, obtained from ~2MB:224MB range: True
The kernel's page table given by kpgdir variable is used as stack for scheduler's context: False

Question **4**

Correct

Mark 1.00 out of 1.00

Given that a kernel has 1000 KB of total memory, and holes of sizes (in that order) 300 KB, 200 KB, 100 KB, 250 KB. For each of the requests on the left side, match it with the chunk chosen using the specified algorithm.

Consider each request as first request.

50 KB, worst fit	300 KB	✔
100 KB, worst fit	300 KB	✔
200 KB, first fit	300 KB	✔
150 KB, best fit	200 KB	✔
220 KB, best fit	250 KB	✔
150 KB, first fit	300 KB	✔

The correct answer is: 50 KB, worst fit → 300 KB, 100 KB, worst fit → 300 KB, 200 KB, first fit → 300 KB, 150 KB, best fit → 200 KB, 220 KB, best fit → 250 KB, 150 KB, first fit → 300 KB

Question **5**

Partially correct

Mark 0.60 out of 1.00

Choice of the global or local replacement strategy is a subjective choice for kernel programmers. There are advantages and disadvantages on either side. Out of the following statements, that advocate either global or local replacement strategy, select those statements that have a logically CONSISTENT argument. (That is any statement that is logically correct about either global or local replacement)

Consistent	Inconsistent		
◉☑	○✖	Local replacement can lead to under-utilisation of memory, because a process may not use all the pages allocated to it all the time.	✔
○☑	◉✖	Global replacement may give highly variable per process completion time because number of page faults become un-predictable.	✖
◉☑	○✖	Global replacement can be preferred when greater throughput (number of processes completing per unit time) is a concern, because each process tries to complete at the expense of others, thus leading to overall more processes completing (unless thrashing occurs).	✔
◉☑	○✖	Local replacement results in more predictable per-process completion time because number of page faults can be better predicted.	✔
○☑	◉✖	Local replacement can be preferred when avoiding thrashing is a major concern because with local replacement and minimum number of frames allocated, a process is always able to progress and cascading inter-process page faults are avoided.	✖

Local replacement can lead to under-utilisation of memory, because a process may not use all the pages allocated to it all the time.: Consistent
Global replacement may give highly variable per process completion time because number of page faults become un-predictable.: Consistent
Global replacement can be preferred when greater throughput (number of processes completing per unit time) is a concern, because each process tries to complete at the expense of others, thus leading to overall more processes completing (unless thrashing occurs).: Consistent
Local replacement results in more predictable per-process completion time because number of page faults can be better predicted.: Consistent
Local replacement can be preferred when avoiding thrashing is a major concern because with local replacement and minimum number of frames allocated, a process is always able to progress and cascading inter-process page faults are avoided.: Consistent

Question **6**

Correct

Mark 1.00 out of 1.00

Map the functionality/use with function/variable in xv6 code.

Setup kernel part of a page table, and switch to that page table	kvmalloc() ✔
Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices	setupkvm() ✔
return a free page, if available; 0, otherwise	kalloc() ✔
Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed	mappages() ✔
Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary	walkpgdir() ✔
Array listing the kernel memory mappings, to be used by setupkvm()	kmap[] ✔

Your answer is correct.

The correct answer is: Setup kernel part of a page table, and switch to that page table → kvmalloc(), Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices → setupkvm(), return a free page, if available; 0, otherwise → kalloc(), Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed → mappages(), Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary → walkpgdir(), Array listing the kernel memory mappings, to be used by setupkvm() → kmap[]

Question **7**

Partially correct

Mark 0.20 out of 1.00

Mark statements True/False w.r.t. change of states of a process. Note that a statement is true only if the claim and argument both are true.

Reference: The process state diagram (and your understanding of how kernel code works). Note - the diagram does not show zombie state!

Figure 3.2 Diagram of process state.

True	False		
◉☑	○✖	Every forked process has to go through ZOMBIE state, at least for a small duration.	✔
◉✖	○☑	A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first	✖
○☑	◉✖	A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet	✖
○☑	◉✖	Only a process in READY state is considered by scheduler	✖
◉✖	○☑	A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.	✖

Every forked process has to go through ZOMBIE state, at least for a small duration.: True
A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first: False
A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet: True
Only a process in READY state is considered by scheduler: True
A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.: False

Question **8**

Correct

Mark 2.00 out of 2.00

Consider the reference string
6 4 2 0 1 2 6 9 2 0 5

If the number of page frames is 3, then total number of page faults (including initial), using FIFO replacement is:

Answer: 10 ✔

#6# 6,4# 6,4,2 #0,4,2# 0,1,2 #0,1,6 #9,1,6# 9,2,6# 9,2,0 #5,2,0

The correct answer is: 10

Question **9**

Incorrect

Mark 0.00 out of 1.00

Select all the correct statements about linking and loading.

Select one or more:

- [ ] a. Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently)
- [ ] b. Dynamic linking essentially results in relocatable code.
- [ ] c. Continuous memory management schemes can support static linking and static loading. (may be inefficiently)
- [ ] d. Loader is last stage of the linker program
- [ ] e. Dynamic linking and loading is not possible without demand paging or demand segmentation.
- [ ] f. Static linking leads to non-relocatable code
- [ ] g. Continuous memory management schemes can support dynamic linking and dynamic loading.
- [x] h. Dynamic linking is possible with continous memory management, but variable sized partitions only. ✖
- [x] i. Loader is part of the operating system ✔

Your answer is incorrect.

The correct answers are: Continuous memory management schemes can support static linking and static loading. (may be inefficiently), Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently), Dynamic linking essentially results in relocatable code., Loader is part of the operating system, Dynamic linking and loading is not possible without demand paging or demand segmentation.

Question **10**

Correct

Mark 1.00 out of 1.00

Consider a computer system with a 32-bit logical address and 4- KB page size. The system supports up to 512 MB of physical memory. How many entries are there in each of the following?

Write answer as a decimal number.

A conventional, single-level page table:

1048576

✔

An inverted page table:

131072

✔

Question **11**

Incorrect

Mark 0.00 out of 1.00

W.r.t. xv6 code, match the state of a process with a code that sets the state

RUNNING	Choose...
ZOMBIE	Choose...
EMBRYO	Choose...
SLEEPING	Choose...
UNUSED	Choose...
RUNNABLE	scheduler() ✖

The correct answer is: RUNNING → scheduler(), ZOMBIE → exit(), called by process itself, EMBRYO → fork()->allocproc() before setting up the UVM, SLEEPING → sleep(), called by any process blocking itself, UNUSED → wait(), called by parent process, RUNNABLE → wakeup(), called by an interrupt handler

Question **12**

Not answered

Marked out of 1.00

Select the correct statements about interrupt handling in xv6 code

- a. The trapframe pointer in struct proc, points to a location on user stack
- b. The CS and EIP are changed only immediately on a hardware interrupt
- c. The CS and EIP are changed only after pushing user code's SS,ESP on stack
- d. The trapframe pointer in struct proc, points to a location on kernel stack
- e. On any interrupt/syscall/exception the control first jumps in vectors.S
- f. The function trap() is the called irrespective of hardware interrupt/system-call/exception
- g. All the 256 entries in the IDT are filled
- h. The function trap() is the called only in case of hardware interrupt
- i. xv6 uses the 64th entry in IDT for system calls
- j. xv6 uses the 0x64th entry in IDT for system calls
- k. Before going to alltraps, the kernel stack contains upto 5 entries.
- l. Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt
- m. On any interrupt/syscall/exception the control first jumps in trapasm.S

Your answer is incorrect.

The correct answers are: All the 256 entries in the IDT are filled, Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt, xv6 uses the 64th entry in IDT for system calls, On any interrupt/syscall/exception the control first jumps in vectors.S, Before going to alltraps, the kernel stack contains upto 5 entries., The trapframe pointer in struct proc, points to a location on kernel stack, The function trap() is the called irrespective of hardware interrupt/system-call/exception, The CS and EIP are changed only after pushing user code's SS,ESP on stack

Question **13**

Correct

Mark 1.00 out of 1.00

The complete range of virtual addresses (after main() in main.c is over), from which the free pages used by kalloc() and kfree() is derived,are:

- a. end, (4MB + PHYSTOP)
- b. P2V(end), PHYSTOP
- c. end, P2V(4MB + PHYSTOP)
- d. end, PHYSTOP
- e. end, 4MB
- f. end, P2V(PHYSTOP) ✔
- g. P2V(end), P2V(PHYSTOP)

Your answer is correct.

The correct answer is: end, P2V(PHYSTOP)

Question **14**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about MMU and it's functionality (on a non-demand paged system)

Select one or more:

- [x] a. Illegal memory access is detected in hardware by MMU and a trap is raised ✔
- [ ] b. Illegal memory access is detected by operating system
- [ ] c. MMU is a separate chip outside the processor
- [ ] d. The operating system interacts with MMU for every single address translation
- [x] e. Logical to physical address translations in MMU are done in hardware, automatically ✔
- [ ] f. Logical to physical address translations in MMU are done with specific machine instructions
- [x] g. MMU is inside the processor ✔
- [x] h. The Operating system sets up relevant CPU registers to enable proper MMU translations ✔

Your answer is correct.

The correct answers are: MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

Question **15**

Incorrect

Mark 0.00 out of 2.00

Order the following events, in the creation of init() process in xv6:

1. ✖ initcode is selected by scheduler for execution
2. ✖ kernel stack is allocated for initcode process
3. ✖ values are set in the trapframe of initcode
4. ✖ sys_exec runs
5. ✖ initcode process is set to be runnable
6. ✖ code is set to start in forkret() when process gets scheduled
7. ✖ Arguments on setup on process stack for /init
8. ✖ trapframe and context pointers are set to proper location
9. ✖ trap() runs
10. ✖ userinit() is called
11. ✖ the header of "/init" ELF file is ready by kernel
12. ✖ empty struct proc is obtained for initcode
13. ✖ Stack is allocated for "/init" process
14. ✖ function pointer from syscalls[] array is invoked
15. ✖ memory mappings are created for "/init" process
16. ✖ page table mappings of 'initcode' are replaced by makpings of 'init'
17. ✖ initcode calls exec system call
18. ✖ initcode process runs
19. ✖ name of process "/init" is copied in struct proc

Your answer is incorrect.

Grading type: Relative to the next item (including last)

Grade details: 0 / 19 = 0%

Here are the scores for each item in this response:

1. 0 / 1 = 0%
2. 0 / 1 = 0%
3. 0 / 1 = 0%
4. 0 / 1 = 0%
5. 0 / 1 = 0%
6. 0 / 1 = 0%
7. 0 / 1 = 0%
8. 0 / 1 = 0%
9. 0 / 1 = 0%
10. 0 / 1 = 0%
11. 0 / 1 = 0%
12. 0 / 1 = 0%
13. 0 / 1 = 0%
14. 0 / 1 = 0%
15. 0 / 1 = 0%

16. 0 / 1 = 0%
17. 0 / 1 = 0%
18. 0 / 1 = 0%
19. 0 / 1 = 0%

The correct order for these items is as follows:

1. userinit() is called
2. empty struct proc is obtained for initcode
3. kernel stack is allocated for initcode process
4. trapframe and context pointers are set to proper location
5. code is set to start in forkret() when process gets scheduled
6. kernel memory mappings are created for initcode
7. values are set in the trapframe of initcode
8. initcode process is set to be runnable
9. initcode is selected by scheduler for execution
10. initcode process runs
11. initcode calls exec system call
12. trap() runs
13. function pointer from syscalls[] array is invoked
14. sys_exec runs
15. the header of "/init" ELF file is ready by kernel
16. memory mappings are created for "/init" process
17. Stack is allocated for "/init" process
18. Arguments on setup on process stack for /init
19. name of process "/init" is copied in struct proc
20. page table mappings of 'initcode' are replaced by makpings of 'init'

Question **16**

Partially correct

Mark 0.56 out of 1.00

Mark the statements as True or False, w.r.t. mmap()

True	False		
		mmap() can be implemented on both demand paged and non-demand paged systems.	✖
		MAP_FIXED guarantees that the mapping is always done at the specified address	✖
		MAP_PRIVATE leads to a mapping that is copy-on-write	✔
		on failure mmap() returns (void *)-1	✔
		MAP_SHARED leads to a mapping that is copy-on-write	✔
		mmap() results in changes to buffer-cache of the kernel.	✖
		on failure mmap() returns NULL	✔
		mmap() results in changes to page table of a process.	✖
		mmap() is a system call	✔

mmap() can be implemented on both demand paged and non-demand paged systems.: True
MAP_FIXED guarantees that the mapping is always done at the specified address: False
MAP_PRIVATE leads to a mapping that is copy-on-write: True
on failure mmap() returns (void *)-1: True
MAP_SHARED leads to a mapping that is copy-on-write: False
mmap() results in changes to buffer-cache of the kernel.: False
on failure mmap() returns NULL: False
mmap() results in changes to page table of a process.: True
mmap() is a system call: True

Question **17**

Incorrect

Mark 0.00 out of 1.00

If one thread opens a file with read privileges then

Select one:

- a. other threads in the same process can also read from that file
- b. none of these
- c. any other thread cannot read from that file
- d. other threads in the another process can also read from that file ✖

Your answer is incorrect.

The correct answer is: other threads in the same process can also read from that file

Question **18**

Partially correct

Mark 0.60 out of 1.00

Mark the statements about named and un-named pipes as True or False

True	False		
◉✓	○✗	Named pipe exists as a file	✔
◉✓	○✗	Un-named pipes are inherited by a child process from parent.	✔
◉✗	○✓	The buffers for named-pipe are in process-memory while the buffers for the un-named pipe are in kernel memory.	✖
◉✓	○✗	Both types of pipes are an extension of the idea of "message passing".	✔
◉✗	○✓	A named pipe has a name decided by the kernel.	✖
○✓	◉✗	Un-named pipes can be used for communication between only "related" processes, if the common ancestor created it.	✖
◉✓	○✗	Both types of pipes provide FIFO communication.	✔
○✗	◉✓	Named pipes can be used for communication between only "related" processes.	✔
◉✓	○✗	Named pipes can exist beyond the life-time of processes using them.	✔
◉✗	○✓	The pipe() system call can be used to create either a named or un-named pipe.	✖

Named pipe exists as a file: True
Un-named pipes are inherited by a child process from parent.: True
The buffers for named-pipe are in process-memory while the buffers for the un-named pipe are in kernel memory.: False
Both types of pipes are an extension of the idea of "message passing".: True
A named pipe has a name decided by the kernel.: False
Un-named pipes can be used for communication between only "related" processes, if the common ancestor created it.: True
Both types of pipes provide FIFO communication.: True
Named pipes can be used for communication between only "related" processes.: False
Named pipes can exist beyond the life-time of processes using them.: True
The pipe() system call can be used to create either a named or un-named pipe.: False

Question **19**

Partially correct

Mark 0.67 out of 1.00

Select the most common causes of use of IPC by processes

- [ ] a. More modular code
- [x] b. Breaking up a large task into small tasks and speeding up computation, on multiple core machines ✔
- [ ] c. More security checks
- [x] d. Sharing of information of common interest ✔
- [ ] e. Get the kernel performance statistics

The correct answers are: Sharing of information of common interest, Breaking up a large task into small tasks and speeding up computation, on multiple core machines, More modular code

Question **20**

Correct

Mark 1.00 out of 1.00

For each function/code-point, select the status of segmentation setup in xv6

after seginit() in main()	gdt setup with 5 entries (0 to 4) on one processor	✔
bootmain()	gdt setup with 3 entries, at start32 symbol of bootasm.S	✔
after startothers() in main()	gdt setup with 5 entries (0 to 4) on all processors	✔
entry.S	gdt setup with 3 entries, at start32 symbol of bootasm.S	✔
kvmalloc() in main()	gdt setup with 3 entries, at start32 symbol of bootasm.S	✔
bootasm.S	gdt setup with 3 entries, at start32 symbol of bootasm.S	✔

Your answer is correct.

The correct answer is: after seginit() in main() → gdt setup with 5 entries (0 to 4) on one processor, bootmain() → gdt setup with 3 entries, at start32 symbol of bootasm.S, after startothers() in main() → gdt setup with 5 entries (0 to 4) on all processors, entry.S → gdt setup with 3 entries, at start32 symbol of bootasm.S, kvmalloc() in main() → gdt setup with 3 entries, at start32 symbol of bootasm.S, bootasm.S → gdt setup with 3 entries, at start32 symbol of bootasm.S

Question **21**

Partially correct

Mark 0.50 out of 1.00

Mark whether the given sequence of events is possible or not-possible. Also, select the reason for your answer.

For each sequence it's a not-possible sequence if some important event is not mentioned in the sequence.

Assume that the kernel code is non-interruptible and uniprocessor system.

```
Process P1, user code executing
Timer interrupt
Context changes to kernel context
Generic interrupt handler runs
Generic interrupt handler calls Scheduler
Scheduler selects P2 for execution
After scheduler, Process P2 user code executing
```

This sequence of events is: not-possible ✔

Because

Generic interrupt handler can not call scheduler ✖

Question **22**

Partially correct

Mark 0.63 out of 1.00

Mark the statements as True or False, w.r.t. passing of arguments to system calls in xv6 code.

True	False		
✖	☑	Integer arguments are stored in eax, ebx, ecx, etc. registers	✖
☑	✖	String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer	✔
☑	✖	The functions like argint(), argstr() make the system call arguments available in the kernel.	✔
✖	☑	String arguments are first copied to trapframe and then from trapframe to kernel's other variables.	✖
☑	✖	The arguments to system call originally reside on process stack.	✔
✖	☑	The arguments to system call are copied to kernel stack in trapasm.S	✖
☑	✖	Integer arguments are copied from user memory to kernel memory using argint()	✔
☑	✖	The arguments are accessed in the kernel code using esp on the trapframe.	✔

Integer arguments are stored in eax, ebx, ecx, etc. registers: False
String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer: True
The functions like argint(), argstr() make the system call arguments available in the kernel.: True
String arguments are first copied to trapframe and then from trapframe to kernel's other variables.: False
The arguments to system call originally reside on process stack.: True
The arguments to system call are copied to kernel stack in trapasm.S: False
Integer arguments are copied from user memory to kernel memory using argint(): True
The arguments are accessed in the kernel code using esp on the trapframe.: True

Question **23**

Not answered

Marked out of 1.00

Given below is a sequence of reference bits on pages before the second chance algorithm runs. Before the algorithm runs, the counter is at the page marked (x). Write the sequence of reference bits after the second chance algorithm has executed once. In the answer write PRECISELY one space BETWEEN each number and do not mention (x).

0 0 1(x) 1 0 1 1

Answer:

The correct answer is: 0 0 0 0 0 1 1

Question **24**

Correct

Mark 2.00 out of 2.00

For the reference string
3 4 3 5 2

using FIFO replacement policy for pages,

consider the number of page faults for 2, 3 and 4 page frames.
Select the correct statement.

Select one:

- a. Exhibit Balady's anomaly between 3 and 4 frames
- b. Do not exhibit Balady's anomaly
- c. Exhibit Balady's anomaly between 2 and 3 frames

Your answer is correct.

The correct answer is: Do not exhibit Balady's anomaly

Question **25**

Correct

Mark 1.00 out of 1.00

For the reference string
3 4 3 5 2

using LRU replacement policy for pages,

consider the number of page faults for 2, 3 and 4 page frames.
Select the most correct statement.

Select one:

- a. LRU will never exhibit Balady's anomaly ✔
- b. Exhibit Balady's anomaly between 2 and 3 frames
- c. This example does not exhibit Balady's anomaly
- d. Exhibit Balady's anomaly between 3 and 4 frames

Your answer is correct.

The correct answer is: LRU will never exhibit Balady's anomaly

Question **26**

Partially correct

Mark 0.55 out of 1.00

Select all the correct statements about process states.

Note that in this question you lose marks for every incorrect choice that you make, proportional to actual number of incorrect choices.

- [ ] a. Process state is implemented as a string
- [x] b. Process state is stored in the PCB ✔
- [ ] c. A process becomes ZOMBIE when another process bites into it's memory
- [x] d. Process state is stored in the processor ✖
- [ ] e. The scheduler can change state of a process from RUNNALBE to RUNNING and vice-versa
- [x] f. The scheduler can change state of a process from RUNNALBE to RUNNING ✔
- [x] g. A process becomes ZOMBIE when it calls exit() ✔
- [ ] h. Process state is changed only by interrupt handlers
- [ ] i. Process state can be implemented as just a number

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: Process state is stored in the PCB, Process state can be implemented as just a number, The scheduler can change state of a process from RUNNALBE to RUNNING, A process becomes ZOMBIE when it calls exit()

Question **27**

Partially correct

Mark 0.38 out of 1.00

Consider a demand-paging system with the following time-measured utilizations:
CPU utilization : 20%
Paging disk: 97.7%
Other I/O devices: 5%

For each of the following, indicate whether it will (or is likely to) improve CPU utilization (even if by a small amount). Explain your answers.

a. Install a faster CPU : Yes ✖

b. Install a bigger paging disk. : Yes ✖

c. Increase the degree of multiprogramming. : Yes ✖

d. Decrease the degree of multiprogramming. : Yes ✔

e. Install more main memory.: Yes ✔

f. Install a faster hard disk or multiple controllers with multiple hard disks. : Yes ✔

g. Add prepaging to the page-fetch algorithms. :
May be ✖

h. Increase the page size. : May be ✖

Question **28**

Incorrect

Mark 0.00 out of 1.00

Suppose a kernel uses a buddy allocator. The smallest chunk that can be allocated is of size 32 bytes. One bit is used to track each such chunk, where 1 means allocated and 0 means free. The chunk looks like this as of now:

10011010

Now, there is a request for a chunk of 50 bytes.

After this allocation, the bitmap, indicating the status of the buddy allocator will be

Answer: 10110010 ✖

The correct answer is: 11111010

Question **29**

Partially correct

Mark 0.75 out of 1.00

Select the correct points of comparison between POSIX and System V shared memory.

- [x] a. POSIX shared memory is newer than System V shared memory ✔
- [x] b. POSIX shared memory is "thread safe", System V is not ✔
- [x] c. System V is more prevalent than POSIX even today ✔
- [ ] d. POSIX allows giving name to shared memory, System V does not

The correct answers are: POSIX shared memory is newer than System V shared memory, POSIX shared memory is "thread safe", System V is not, POSIX allows giving name to shared memory, System V does not, System V is more prevalent than POSIX even today

Question **30**

Partially correct

Mark 0.67 out of 1.00

Figure 9.8 Paging hardware.

Mark the statements as True or False, w.r.t. the above diagram (note that the diagram does not cover all details of what actually happens!)

True	False		
◉☑	○✖	The combining of f and d is done by MMU	✔
◉✖	○☑	There are total 3 memory references in this diagram	✖
◉☑	○✖	The split of logical address into p and d is done by MMU	✔
◉☑	○✖	The page table is in physical memory and must be continuous	✔
◉✖	○☑	Using the offset d in the physical page-frame is done by MMU	✖
◉☑	○✖	The logical address issued by CPU is the same one generated by compiler	✔

The combining of f and d is done by MMU: True
There are total 3 memory references in this diagram: False
The split of logical address into p and d is done by MMU: True
The page table is in physical memory and must be continuous: True
Using the offset d in the physical page-frame is done by MMU: False
The logical address issued by CPU is the same one generated by compiler: True

Question **31**

Partially correct

Mark 0.50 out of 1.00

Select all the correct statements about signals

Select one or more:

- [ ] a. SIGKILL definitely kills a process because it's code runs in kernel mode of CPU
- [x] b. Signals are delivered to a process by another process ✗
- [ ] c. The signal handler code runs in kernel mode of CPU
- [x] d. SIGKILL definitely kills a process because it can't be caught or ignored, and it's default action terminates the process ✓
- [x] e. The signal handler code runs in user mode of CPU ✓
- [x] f. A signal handler can be invoked asynchronously or synchronously depending on signal type ✓
- [ ] g. Signal handlers once replaced can't be restored
- [ ] h. Signals are delivered to a process by kernel

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: Signals are delivered to a process by kernel, A signal handler can be invoked asynchronously or synchronously depending on signal type, The signal handler code runs in user mode of CPU, SIGKILL definitely kills a process because it can't be caught or ignored, and it's default action terminates the process

Question **32**

Correct

Mark 1.00 out of 1.00

The data structure used in kalloc() and kfree() in xv6 is

- ( ) a. Singly linked circular list
- (•) b. Singly linked NULL terminated list ✓
- ( ) c. Double linked NULL terminated list
- ( ) d. Doubly linked circular list

Your answer is correct.

The correct answer is: Singly linked NULL terminated list

Question **33**

Partially correct

Mark 1.78 out of 2.00

Match the description of a memory management function with the name of the function that provides it, in xv6

Load contents from ELF into existing pages	loaduvm()	✔
Mark the page as in-accessible	clearpteu()	✔
setup the kernel part in the page table	setupkvm()	✔
Switch to kernel page table	switchkvm()	✔
Create a copy of the page table of a process	copyuvm()	✔
Copy the code pages of a process	No such function	✔
Setup and load the user page table for initcode process	inituvm()	✔
Switch to user page table	switchuvm()	✔
Load contents from ELF into pages after allocating the pages first	inituvm()	✖

The correct answer is: Load contents from ELF into existing pages → loaduvm(), Mark the page as in-accessible → clearpteu(), setup the kernel part in the page table → setupkvm(), Switch to kernel page table → switchkvm(), Create a copy of the page table of a process → copyuvm(), Copy the code pages of a process → No such function, Setup and load the user page table for initcode process → inituvm(), Switch to user page table → switchuvm(), Load contents from ELF into pages after allocating the pages first → No such function

Question **34**

Partially correct

Mark 0.60 out of 1.00

Mark the statements as True or False, w.r.t. thrashing

True	False		
✖	✔	Thrashing occurs because some process is doing lot of disk I/O.	✖
✔	✖	Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.	✔
✖	✔	mmap() solves the problem of thrashing.	✔
✔	✖	The working set model is an attempt at approximating the locality of a process.	✔
✔	✖	Thrashing is particular to demand paging systems, and does not apply to pure paging systems.	✔
✖	✔	Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.	✖
✖	✔	Thrashing can occur even if entire memory is not in use.	✖
✔	✖	During thrashing the CPU is under-utilised as most time is spent in I/O	✖
✔	✖	Thrashing can be limited if local replacement is used.	✔
✔	✖	Thrashing occurs when the total size of all processe's locality exceeds total memory size.	✔

Thrashing occurs because some process is doing lot of disk I/O.: False
Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.: True
mmap() solves the problem of thrashing.: False
The working set model is an attempt at approximating the locality of a process.: True
Thrashing is particular to demand paging systems, and does not apply to pure paging systems.: True
Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.: False
Thrashing can occur even if entire memory is not in use.: False
During thrashing the CPU is under-utilised as most time is spent in I/O: True
Thrashing can be limited if local replacement is used.: True
Thrashing occurs when the total size of all processe's locality exceeds total memory size.: True

Question **35**

Correct

Mark 1.00 out of 1.00

After virtual memory is implemented

(select T/F for each of the following)One Program's size can be larger than physical memory size

True	False		
◉	○	Cumulative size of all programs can be larger than physical memory size	✔
◉	○	Code need not be completely in memory	✔
◉	○	One Program's size can be larger than physical memory size	✔
○	◉	Virtual addresses become available to executing process	✔
○	◉	Virtual access to memory is granted to all processes	✔
◉	○	Relatively less I/O may be possible during process execution	✔
◉	○	Logical address space could be larger than physical address space	✔

Cumulative size of all programs can be larger than physical memory size: True
Code need not be completely in memory: True
One Program's size can be larger than physical memory size: True
Virtual addresses become available to executing process: False
Virtual access to memory is granted to all processes: False
Relatively less I/O may be possible during process execution: True
Logical address space could be larger than physical address space: True

◄ (Optional Assignment) lseek system call in xv6

Jump to...

Feedback on Quiz-2 ►

Started on	Thursday, 18 March 2021, 2:46 PM
State	Finished
Completed on	Thursday, 18 March 2021, 3:50 PM
Time taken	1 hour 4 mins
Grade	**10.36** out of 20.00 (**52**%)

Question **1**

Partially correct

Mark 0.57 out of 1.00

Mark True, the actions done as part of code of swtch() in swtch.S, in xv6

True	False		
		Restore new callee saved registers from kernel stack of new context	✔
		Save old callee saved registers on kernel stack of old context	✔
		Save old callee saved registers on user stack of old context	✔
		Switch from old process context to new process context	✖
		Switch from one stack (old) to another(new)	✖
		Restore new callee saved registers from user stack of new context	✔
		Jump to code in new context	✖

Restore new callee saved registers from kernel stack of new context: True
Save old callee saved registers on kernel stack of old context: True
Save old callee saved registers on user stack of old context: False
Switch from old process context to new process context: False
Switch from one stack (old) to another(new): True
Restore new callee saved registers from user stack of new context: False
Jump to code in new context: False

Question **2**

Partially correct

Mark 0.17 out of 0.50

For each function/code-point, select the status of segmentation setup in xv6

bootmain()	gdt setup with 3 entries, right from first line of code of bootloader	✖
kvmalloc() in main()	gdt setup with 5 entries (0 to 4) on one processor	✖
after startothers() in main()	gdt setup with 5 entries (0 to 4) on all processors	✔
after seginit() in main()	gdt setup with 5 entries (0 to 4) on all processors	✖
bootasm.S	gdt setup with 3 entries, right from first line of code of bootloader	✖
entry.S	gdt setup with 3 entries, at start32 symbol of bootasm.S	✔

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: bootmain() → gdt setup with 3 entries, at start32 symbol of bootasm.S, kvmalloc() in main() → gdt setup with 3 entries, at start32 symbol of bootasm.S, after startothers() in main() → gdt setup with 5 entries (0 to 4) on all processors, after seginit() in main() → gdt setup with 5 entries (0 to 4) on one processor, bootasm.S → gdt setup with 3 entries, at start32 symbol of bootasm.S, entry.S → gdt setup with 3 entries, at start32 symbol of bootasm.S

Question **3**

Partially correct

Mark 0.38 out of 1.00

Compare paging with demand paging and select the correct statements.

Select one or more:

- [x] a. The meaning of valid-invalid bit in page table is different in paging and demand-paging. ✔
- [x] b. Demand paging requires additional hardware support, compared to paging. ✔
- [ ] c. Paging requires some hardware support in CPU
- [x] d. With paging, it's possible to have user programs bigger than physical memory. ✖
- [x] e. Both demand paging and paging support shared memory pages. ✔
- [ ] f. Demand paging always increases effective memory access time.
- [x] g. With demand paging, it's possible to have user programs bigger than physical memory. ✔
- [x] h. Calculations of number of bits for page number and offset are same in paging and demand paging. ✔
- [ ] i. TLB hit ration has zero impact in effective memory access time in demand paging.
- [ ] j. Paging requires NO hardware support in CPU

Your answer is partially correct.

You have correctly selected 5.

The correct answers are: Demand paging requires additional hardware support, compared to paging., Both demand paging and paging support shared memory pages., With demand paging, it's possible to have user programs bigger than physical memory., Demand paging always increases effective memory access time., Paging requires some hardware support in CPU, Calculations of number of bits for page number and offset are same in paging and demand paging., The meaning of valid-invalid bit in page table is different in paging and demand-paging.

Question **4**

Partially correct

Mark 0.44 out of 0.50

Suppose a processor supports base(relocation register) + limit scheme of MMU.

Assuming this, mark the statements as True/False

True	False		
✔	✖	The OS may terminate the process while handling the interrupt of memory violation	✔
✔	✖	The hardware detects any memory access beyond the limit value and raises an interrupt	✔
✖	✔	The hardware may terminate the process while handling the interrupt of memory violation	✖
✔	✖	The OS sets up the relocation and limit registers when the process is scheduled	✔
✔	✖	The compiler generates machine code assuming continuous memory address space for process, and calculating appropriate sizes for code, and data;	✔
✖	✔	The process sets up it's own relocation and limit registers when the process is scheduled	✔
✖	✔	The OS detects any memory access beyond the limit value and raises an interrupt	✔
✖	✔	The compiler generates machine code assuming appropriately sized semgments for code, data and stack.	✔

The OS may terminate the process while handling the interrupt of memory violation: True
The hardware detects any memory access beyond the limit value and raises an interrupt: True
The hardware may terminate the process while handling the interrupt of memory violation: False
The OS sets up the relocation and limit registers when the process is scheduled: True
The compiler generates machine code assuming continuous memory address space for process, and calculating appropriate sizes for code, and data;: True
The process sets up it's own relocation and limit registers when the process is scheduled: False
The OS detects any memory access beyond the limit value and raises an interrupt: False
The compiler generates machine code assuming appropriately sized semgments for code, data and stack.: False

Question **5**

Correct

Mark 0.50 out of 0.50

Consider the following list of free chunks, in continuous memory management:

10k, 25k, 12k, 7k, 9k, 13k

Suppose there is a request for chunk of size 9k, then the free chunk selected under each of the following schemes will be

Best fit:

9k

✔

First fit:

10k

✔

Worst fit:

25k

✔

Question **6**

Partially correct

Mark 0.50 out of 1.00

Select all the correct statements about MMU and it's functionality

Select one or more:

- [ ] a. MMU is a separate chip outside the processor
- [x] b. MMU is inside the processor ✔
- [ ] c. Logical to physical address translations in MMU are done with specific machine instructions
- [x] d. The operating system interacts with MMU for every single address translation ✖
- [x] e. Illegal memory access is detected in hardware by MMU and a trap is raised ✔
- [ ] f. The Operating system sets up relevant CPU registers to enable proper MMU translations
- [x] g. Logical to physical address translations in MMU are done in hardware, automatically ✔
- [ ] h. Illegal memory access is detected by operating system

Your answer is partially correct.

You have correctly selected 3.
The correct answers are: MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

Question **7**

Incorrect

Mark 0.00 out of 0.50

Assuming a 8- KB page size, what is the page numbers for the address 874815 reference in decimal :

(give answer also in decimal)

Answer: 2186 ✖

The correct answer is: 107

Question **8**

Incorrect

Mark 0.00 out of 0.25

Select the compiler's view of the process's address space, for each of the following MMU schemes:
(Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

Segmentation, then paging	Many continuous chunks each of page size	✖
Relocation + Limit	Many continuous chunks of same size	✖
Segmentation	one continuous chunk	✖
Paging	many continuous chunks of variable size	✖

Your answer is incorrect.

The correct answer is: Segmentation, then paging → many continuous chunks of variable size, Relocation + Limit → one continuous chunk, Segmentation → many continuous chunks of variable size, Paging → one continuous chunk

Question **9**

Incorrect

Mark 0.00 out of 0.50

Suppose the memory access time is 180ns and TLB hit ratio is 0.3, then effective memory access time is (in nanoseconds);

Answer: 192 ✖

The correct answer is: 306.00

Question **10**

Correct

Mark 0.50 out of 0.50

In xv6, The struct context is given as

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

Select all the reasons that explain why only these 5 registers are included in the struct context.

- [x] a. The segment registers are same across all contexts, hence they need not be saved ✔
- [x] b. esp is not saved in context, because context{} is on stack and it's address is always argument to swtch() ✔
- [ ] c. xv6 tries to minimize the size of context to save memory space
- [ ] d. esp is not saved in context, because it's not part of the context
- [x] e. eax, ecx, edx are caller save, hence no need to save ✔

Your answer is correct.

The correct answers are: The segment registers are same across all contexts, hence they need not be saved, eax, ecx, edx are caller save, hence no need to save, esp is not saved in context, because context{} is on stack and it's address is always argument to swtch()

Question **11**

Partially correct

Mark 0.83 out of 1.50

Arrange the following events in order, in page fault handling:

Disk interrupt wakes up the process	7	✔
The reference bit is found to be invalid by MMU	1	✔
OS makes available an empty frame	6	✖
Restart the instruction that caused the page fault	9	✔
A hardware interrupt is issued	3	✖
OS schedules a disk read for the page (from backing store)	5	✔
Process is kept in wait state	4	✖
Page tables are updated for the process	8	✔
Operating system decides that the page was not in memory	2	✖

Your answer is partially correct.

You have correctly selected 5.

The correct answer is: Disk interrupt wakes up the process → 7, The reference bit is found to be invalid by MMU → 1, OS makes available an empty frame → 4, Restart the instruction that caused the page fault → 9, A hardware interrupt is issued → 2, OS schedules a disk read for the page (from backing store) → 5, Process is kept in wait state → 6, Page tables are updated for the process → 8, Operating system decides that the page was not in memory → 3

Question **12**

Incorrect

Mark 0.00 out of 0.50

Suppose a kernel uses a buddy allocator. The smallest chunk that can be allocated is of size 32 bytes. One bit is used to track each such chunk, where 1 means allocated and 0 means free. The chunk looks like this as of now:

00001010

Now, there is a request for a chunk of 70 bytes.

After this allocation, the bitmap, indicating the status of the buddy allocator will be

Answer: 11101010 ✖

The correct answer is: 11111010

Question **13**

Incorrect

Mark 0.00 out of 0.25

The complete range of virtual addresses (after main() in main.c is over), from which the free pages used by kalloc() and kfree() is derived,are:

- a. end, 4MB
- b. P2V(end), P2V(PHYSTOP)
- c. end, P2V(4MB + PHYSTOP)
- d. P2V(end), PHYSTOP ✖
- e. end, (4MB + PHYSTOP)
- f. end, PHYSTOP
- g. end, P2V(PHYSTOP)

Your answer is incorrect.

The correct answer is: end, P2V(PHYSTOP)

Question **14**

Partially correct

Mark 0.33 out of 0.50

Match the pair

Hashed page table	Linear search on collsion done by OS (e.g. SPARC Solaris) typically	✔
Inverted Page table	Linear/Parallel search using frame number in page table	✖
Hierarchical Paging	More memory access time per hierarchy	✔

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: Hashed page table → Linear search on collsion done by OS (e.g. SPARC Solaris) typically, Inverted Page table → Linear/Parallel search using page number in page table, Hierarchical Paging → More memory access time per hierarchy

Question **15**

Partially correct

Mark 0.29 out of 0.50

After virtual memory is implemented

(select T/F for each of the following)One Program's size can be larger than physical memory size

True	False		
◉☑	○✖	Code need not be completely in memory	✔
◉☑	○✖	Cumulative size of all programs can be larger than physical memory size	✔
◉✖	○☑	Virtual access to memory is granted	✖
◉☑	○✖	Logical address space could be larger than physical address space	✔
◉✖	○☑	Virtual addresses are available	✖
○☑	◉✖	Relatively less I/O may be possible during process execution	✖
◉☑	○✖	One Program's size can be larger than physical memory size	✔

Code need not be completely in memory: True
Cumulative size of all programs can be larger than physical memory size: True
Virtual access to memory is granted: False
Logical address space could be larger than physical address space: True
Virtual addresses are available: False
Relatively less I/O may be possible during process execution: True
One Program's size can be larger than physical memory size: True

Question **16**

Partially correct

Mark 0.64 out of 1.00

W.r.t. Memory management in xv6,

xv6 uses physical memory upto 224 MB onlyMark statements True or False

True	False		
		The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context	✔
		The stack allocated in entry.S is used as stack for scheduler's context for first processor	✔
		The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir	✔
		The free page-frame are created out of nearly 222 MB	✖
		The kernel code and data take up less than 2 MB space	✔
		The switchkvm() call in scheduler() changes CR3 to use page directory of new process	✖
		The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context	✔
		PHYSTOP can be increased to some extent, simply by editing memlayout.h	✔
		xv6 uses physical memory upto 224 MB only	✖
		The process's address space gets mapped on frames, obtained from ~2MB:224MB range	✔
		The kernel's page table given by kpgdir variable is used as stack for scheduler's context	✖

The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context: True
The stack allocated in entry.S is used as stack for scheduler's context for first processor: True
The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir: True
The free page-frame are created out of nearly 222 MB: True
The kernel code and data take up less than 2 MB space: True
The switchkvm() call in scheduler() changes CR3 to use page directory of new process: False
The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context: False
PHYSTOP can be increased to some extent, simply by editing memlayout.h: True
xv6 uses physical memory upto 224 MB only: True
The process's address space gets mapped on frames, obtained from ~2MB:224MB range: True
The kernel's page table given by kpgdir variable is used as stack for scheduler's context: False

Question **17**

Incorrect

Mark 0.00 out of 1.50

Consider the reference string
6 4 2 0 1 2 6 9 2 0 5

If the number of page frames is 3, then total number of page faults (including initial), using LRU replacement is:

Answer: 8 ✖

#6# 6,4# 6,4,2 # 0,4,2#0,1,2#6,1,2#6,9,2#0,9,2#0,5,2

The correct answer is: 9

Question **18**

Partially correct

Mark 0.31 out of 0.50

Consider the image given below, which explains how paging works.

Figure 9.8 Paging hardware.

Mention whether each statement is True or False, with respect to this image.

True	False		
◉☑	○✖	The PTBR is present in the CPU as a register	✔
○✖	◉☑	The page table is indexed using frame number	✔
○☑	◉✖	The page table is indexed using page number	✖
◉✖	○☑	The locating of the page table using PTBR also involves paging translation	✖
○✖	◉☑	Size of page table is always determined by the size of RAM	✔
◉☑	○✖	The page table is itself present in Physical memory	✔
○☑	○✖	Maximum Size of page table is determined by number of bits used for page number	✖
◉☑	○✖	The physical address may not be of the same size (in bits) as the logical address	✔

The PTBR is present in the CPU as a register: True
The page table is indexed using frame number: False
The page table is indexed using page number: True
The locating of the page table using PTBR also involves paging translation: False
Size of page table is always determined by the size of RAM: False
The page table is itself present in Physical memory: True
Maximum Size of page table is determined by number of bits used for page number: True
The physical address may not be of the same size (in bits) as the logical address: True

Question **19**

Correct

Mark 2.00 out of 2.00

Given below is shared memory code with two processes sharing a memory segment.
The first process sends a user input string to second process. The second capitalizes the string. Then the first process prints the capitalized version.
Fill in the blanks to complete the code.

```
// First process
#define SHMSZ    27

int main()
{
   char c;
   int shmid;
   key_t key;
   char *shm, *s, string[128];
   key = 5679;
   if ((shmid =
```

shmget

✔ (key, SHMSZ, IPC_CREAT | 0666)) < 0) {

```
      perror("shmget");
      exit(1);
   }
   if ((shm =
```

shmat

✔ (shmid, NULL, 0)) == (char *) -1) {

```
      perror("shmat");
      exit(1);
   }
   s = shm;
   *s = '$';
   scanf("%s", string);
   strcpy(s + 1, string);
   *s = '
```

@

✔ '; //note the quotes

```
   while(*s != '
```

$

✔ ')

```
      sleep(1);
   printf("%s\n", s + 1);
   exit(0);
}
```

```
//Second process
#define SHMSZ    27
int main()
{
   int shmid;
   key_t key;
   char *shm, *s;
   int i;
   char string[128];
   key =
```

5679

```
    ;
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }
    s =
```

shm

```
    ;
    while(*s != '@')
        sleep(1);
    for(i = 0; i < strlen(s + 1); i++)
        s[i + 1] =  toupper(s[i + 1]);
    *s = '$';
    exit(0);
}
```

Question **20**

Partially correct

Mark 0.25 out of 0.50

Map the functionality/use with function/variable in xv6 code.

return a free page, if available; 0, otherwise	kinit1() ✗
Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed	mappages() ✓
Array listing the kernel memory mappings, to be used by setupkvm()	kmap[] ✓
Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices	kvmalloc() ✗
Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary	walkpgdir() ✓
Setup kernel part of a page table, and switch to that page table	setupkvm() ✗

Your answer is partially correct.

You have correctly selected 3.

The correct answer is: return a free page, if available; 0, otherwise → kalloc(), Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed → mappages(), Array listing the kernel memory mappings, to be used by setupkvm() → kmap[], Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices → setupkvm(), Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary → walkpgdir(), Setup kernel part of a page table, and switch to that page table → kvmalloc()

Question **21**

Partially correct

Mark 1.53 out of 2.50

Order events in xv6 timer interrupt code

(Transition from process P1 to P2's code.)

P2 is selected and marked RUNNING	12	✔
Change of stack from user stack to kernel stack of P1	3	✔
Timer interrupt occurs	2	✔
alltraps() will call iret	17	✖
change to context of P2, P2's kernel stack in use now	13	✔
P2's trap() will return to alltraps	16	✖
jump in vector.S	4	✔
P2 will return from sched() in yield()	14	✖
yield() is called	8	✔
trap() is called	7	✔
Process P2 is executing	18	✖
P1 is marked as RUNNABLE	9	✔
P2's yield() will return in trap()	15	✖
Process P1 is executing	1	✔
sched() is called,	11	✖
change to context of the scheduler, scheduler's stack in use now	10	✖
jump to alltraps	5	✔
Trapframe is built on kernel stack of P1	6	✔

Your answer is partially correct.

You have correctly selected 11.

The correct answer is: P2 is selected and marked RUNNING → 12, Change of stack from user stack to kernel stack of P1 → 3, Timer interrupt occurs → 2, alltraps() will call iret → 18, change to context of P2, P2's kernel stack in use now → 13, P2's trap() will return to alltraps → 17, jump in vector.S → 4, P2 will return from sched() in yield() → 15, yield() is called → 8, trap() is called → 7, Process P2 is executing → 14, P1 is marked as RUNNABLE → 9, P2's yield() will return in trap() → 16, Process P1 is executing → 1, sched() is called, → 10, change to context of the scheduler, scheduler's stack in use now → 11, jump to alltraps → 5, Trapframe is built on kernel stack of P1 → 6

Question **22**

Incorrect

Mark 0.00 out of 1.00

Given that the memory access time is 200 ns, probability of a page fault is 0.7 and page fault handling time is 8 ms,
The effective memory access time in nanoseconds is:

Answer: 192 ✖

The correct answer is: 5600060.00

Question **23**

Correct

Mark 0.25 out of 0.25

Select the state that is not possible after the given state, for a process:

New: Running ✔

Ready : Waiting ✔

Running: : None of these ✔

Waiting: Running ✔

Question **24**

Partially correct

Mark 0.63 out of 1.00

Select the correct statements about sched() and scheduler() in xv6 code

- [x] a. scheduler() switches to the selected process's context ✔
- [x] b. When either sched() or scheduler() is called, it does not return immediately to caller ✔
- [ ] c. After call to swtch() in sched(), the control moves to code in scheduler()
- [x] d. Each call to sched() or scheduler() involves change of one stack inside swtch() ✔
- [ ] e. After call to swtch() in scheduler(), the control moves to code in sched()
- [x] f. When either sched() or scheduler() is called, it results in a context switch ✔
- [x] g. sched() switches to the scheduler's context ✔
- [ ] h. sched() and scheduler() are co-routines

Your answer is partially correct.

You have correctly selected 5.
The correct answers are: sched() and scheduler() are co-routines, When either sched() or scheduler() is called, it does not return immediately to caller, When either sched() or scheduler() is called, it results in a context switch, sched() switches to the scheduler's context, scheduler() switches to the selected process's context, After call to swtch() in scheduler(), the control moves to code in sched(), After call to swtch() in sched(), the control moves to code in scheduler(), Each call to sched() or scheduler() involves change of one stack inside swtch()

Question **25**

Correct

Mark 0.25 out of 0.25

The data structure used in kalloc() and kfree() in xv6 is

- a. Doubly linked circular list
- b. Singly linked circular list
- c. Double linked NULL terminated list
- d. Singly linked NULL terminated list ✔

Your answer is correct.

The correct answer is: Singly linked NULL terminated list

◄ (Assignment) lseek system call in xv6

Jump to...

Started on	Saturday, 22 May 2021, 8:00 AM
State	Finished
Completed on	Saturday, 22 May 2021, 9:30 AM
Time taken	1 hour 30 mins
Grade	**26.12** out of 40.00 (**65**%)

Question **1**
Incorrect
Mark 0.00 out of 1.00

A 4 GB disk with 1 KB of block size would require these many number of **blocks** for it's free block bitmap:

Answer: 4096 ✖

The correct answer is: 512

Question **2**
Correct
Mark 1.00 out of 1.00

Given that the memory access time is 110 ns, probability of a page fault is 0.5 and page fault handling time is 12 ms,
The effective memory access time in nanoseconds is:

Answer: 6000165 ✔

The correct answer is: 6000055.00

Question **3**
Incorrect
Mark 0.00 out of 1.00

The maximum size of a file in number of blocks of BSIZE in xv6 code is
(write a number only)

Answer: 268 ✖

The correct answer is: 138

Question **4**
Incorrect
Mark 0.00 out of 1.00

Calculate the average waiting time using
Round Robin scheduling with time quantum of 5 time units
for the following workload
assuming that they arrive in the order written below.

Process	Burst Time
P1	5
P2	7
P3	6
P4	2

Write only a number in the answer upto two decimal points.

Answer: 40.75 ✖

The correct answer is: 10.25

Question **5**

Correct

Mark 1.00 out of 1.00

For the reference string

4 2 5 1 0 1 2 5 4 1 2

the number of page faults, including initial ones,

with FIFO replacement and 2 frames are :

Answer: 10 ✔

4 -

4 2

5 2

5 1

0 1

-

2 1

2 5

4 5

4 1

2 1

The correct answer is: 10

Question **6**

Correct

Mark 1.00 out of 1.00

Assuming a 16- KB page size, what is the page number for the address 428517 reference in decimal :

(give answer also in decimal)

Answer: 27 ✔

The correct answer is: 26

Question **7**

Correct

Mark 1.00 out of 1.00

In the code below assume that each function can be executed concurrently by many threads/processes.
Ignore syntactical issues, and focus on the semantics.

This program is an example of

```
spinlock a, b;  // assume initialized
thread1() {
    spinlock(b);
    //some code;
    spinlock(a);
    //some code;
    spinunlock(b);
    spinunlock(a);
}
thread2() {
    spinlock(a);
    //some code;
    spinlock(b);
    //some code;
    spinunlock(b);
    spinunlock(a);
}
```

- (selected) a. Deadlock ✔
- b. Self Deadlock
- c. None of these
- d. Deadlock or livelock depending on actual race
- e. Livelock

Your answer is correct.

The correct answer is: Deadlock

Question **8**

Partially correct

Mark 1.33 out of 2.00

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid.
"..." means some code.

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
```

Atomic compare and swap instruction (to be expanded inline into code) ✔

```
void
sleep(void *chan, struct spinlock *lk)
{
  ...
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }
```

If you don't do this, a process may be running on two processors parallely ✖

```
void
acquire(struct spinlock *lk)
{
...
__sync_synchronize();
```

Tell compiler not to reorder memory access beyond this line ✔

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
} → Atomic compare and swap instruction (to be expanded inline into code), void
sleep(void *chan, struct spinlock *lk)
{
  ...
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  } → Avoid a self-deadlock, void
acquire(struct spinlock *lk)
{
...
__sync_synchronize(); → Tell compiler not to reorder memory access beyond this line

Question **9**

Correct

Mark 1.00 out of 1.00

Predict the output of the program given here.

Assume that all the path names for the programs are correct. For example "/usr/bin/echo" will actually run echo command.

Assume that there is no mixing of printf output on screen if two of them run concurrently.

In the answer replace a new line by a single space.

For example::

`good`

`output`

should be written as `good output`

```
--
main() {
    int i;
    i = fork();
    if(i == 0)
        execl("/usr/bin/echo", "/usr/bin/echo", "hi", 0);
    else
        wait(0);
    fork();
    execl("/usr/bin/echo", "/usr/bin/echo", "one", 0);
}
```

Answer: hi one one ✔

The correct answer is: hi one one

Question **10**

Partially correct

Mark 1.67 out of 2.00

Select all the blocks that may need to be written back to disk (if updated, of-course), as "Yes", when an operation of deleting a file is carried out on ext2 file system.

An option has to be correct entirely to be marked "Yes"

Superblock	Yes	✔
One or multiple data blocks of the parent directory	No	✔
One or more data bitmap blocks for the parent directory	No	✔
Block bitmap(s) for all the blocks of the file	No	✖
Possibly one block bitmap corresponding to the parent directory	Yes	✔
Data blocks of the file	No	✔

Your answer is partially correct.

only one data block of parent directory. multiple blocks not possible. an entry is always contained within one single block

You have correctly selected 5.

The correct answer is: Superblock → Yes, One or multiple data blocks of the parent directory → No, One or more data bitmap blocks for the parent directory → No, Block bitmap(s) for all the blocks of the file → Yes, Possibly one block bitmap corresponding to the parent directory → Yes, Data blocks of the file → No

Question **11**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about bootloader.

Every wrong selection will deduct marks proportional to 1/n where n is total wrong choices in the question.

You will get minimum a zero.

- [x] a. Modern Bootloaders often allow configuring the way an OS boots ✔
- [x] b. Bootloaders allow selection of OS to boot from ✔
- [ ] c. Bootloader must be one sector in length
- [ ] d. The bootloader loads the BIOS
- [x] e. LILO is a bootloader ✔

Your answer is correct.

The correct answers are: LILO is a bootloader, Modern Bootloaders often allow configuring the way an OS boots, Bootloaders allow selection of OS to boot from

Question **12**

Incorrect

Mark 0.00 out of 1.00

For each of the resource allocation diagram shown,

infer whether the graph contains at least one deadlock or not.

Yes	No		
○	●		✖
●	○		✖

: Yes

: No

Question **13**

Partially correct

Mark 0.71 out of 1.00

Mark the statements about device drivers by marking as True or False.

True	False		
		It's possible that a particular hardware has multiple device drivers available for it.	✖
		xv6 has device drivers for IDE disk and console.	✔
		A disk driver converts OS's logical view of disk into physical locations on disk.	✔
		A device driver code is specific to a hardware device	✔
		All devices of the same type (e.g. 2 hard disks) can typically use the same device driver	✔
		Writing a device driver mandatorily demands reading the technical documentation about the hardware.	✖
		Device driver is an intermediary between the end-user and OS	✔

It's possible that a particular hardware has multiple device drivers available for it.: True
xv6 has device drivers for IDE disk and console.: True
A disk driver converts OS's logical view of disk into physical locations on disk.: True
A device driver code is specific to a hardware device: True
All devices of the same type (e.g. 2 hard disks) can typically use the same device driver: True
Writing a device driver mandatorily demands reading the technical documentation about the hardware.: True
Device driver is an intermediary between the end-user and OS: False

Question **14**

Partially correct

Mark 0.33 out of 1.00

Consider this program.
Some statements are identified using the // comment at the end.

Assume that = is an atomic operation.

```
#include <stdio.h>
#include <pthread.h>
long c = 0, c1 = 0, c2 = 0, run = 1;
void *thread1(void *arg) {
    while(run == 1) {//E
        c = 10; //A
        c1 = c2 + 5; //B
    }
}
void *thread2(void *arg) {
    while(run == 1) {//F
        c = 20;//C
        c2 = c1 + 3;//D
    }
}
int main() {
    pthread_t th1, th2;
    pthread_create(&th1, NULL, thread1, NULL);
    pthread_create(&th2, NULL, thread2, NULL);
    sleep(2);
    run = 0;
    fprintf(stdout, "c = %ld c1+c2 = %ld c1 = %ld c2 = %ld \n", c, c1+c2, c1, c2);
    fflush(stdout);
}
```

Which statements are part of the critical Section?

Yes	No		
✗	✓	F	✗
✓	✗	D	✓
✗	✓	C	✗
✗	✓	A	✗
✓	✗	B	✓
✗	✓	E	✗

F: No
D: Yes
C: No
A: No
B: Yes
E: No

Question **15**

Partially correct

Mark 1.43 out of 2.00

Mark statements as T/F

All statements are in the context of preventing deadlocks.

True	False		
◉	○	A process holding one resources and waiting for just one more resource can also be involved in a deadlock.	✔
◉	○	If a resource allocation graph contains a cycle then there is a guarantee of a deadlock	✖
◉	○	The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order	✖
◉	○	Circular wait is avoided by enforcing a lock ordering	✔
◉	○	Hold and wait means a thread/process holding some locks and waiting for acquiring some.	✔
◉	○	Deadlock is possible if all the conditions are met at the same time: Mutual exclusion, hold and wait, no pre-emption, circular wait.	✔
◉	○	Mutual exclusion is a necessary condition for deadlock because it brings in locks on which deadlock happens	✔

A process holding one resources and waiting for just one more resource can also be involved in a deadlock.: True
If a resource allocation graph contains a cycle then there is a guarantee of a deadlock: False
The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order: False
Circular wait is avoided by enforcing a lock ordering: True
Hold and wait means a thread/process holding some locks and waiting for acquiring some.: True
Deadlock is possible if all the conditions are met at the same time: Mutual exclusion, hold and wait, no pre-emption, circular wait.: True
Mutual exclusion is a necessary condition for deadlock because it brings in locks on which deadlock happens: True

Question **16**

Correct

Mark 1.00 out of 1.00

Match the left side use(or non-use) of a synchronization primitive with the best option on the right side.

This is the smallest primitive made available in software, using the hardware provided atomic instructions	spinlock	✔
This tool is useful for event-wait scenarios	semaphore	✔
This tool is more useful on multiprocessor systems	spinlock	✔
This tool is quite attractive in solving the main bounded buffer problem	semaphore	✔
This tool is very useful for waiting for 'something'	condition variables	✔

Your answer is correct.

The correct answer is: This is the smallest primitive made available in software, using the hardware provided atomic instructions → spinlock, This tool is useful for event-wait scenarios → semaphore, This tool is more useful on multiprocessor systems → spinlock, This tool is quite attractive in solving the main bounded buffer problem → semaphore, This tool is very useful for waiting for 'something' → condition variables

Question **17**

Correct

Mark 1.00 out of 1.00

The permissions -rwx--x--x on a file mean

- [x] a. The file can be read only by the owner ✔
- [ ] b. 'cat' on the file by owner will not work
- [ ] c. 'cat' on the file by any user will work
- [x] d. 'rm' on the file by any user will work ✔
- [x] e. The file can be executed by anyone ✔
- [x] f. The file can be written only by the owner ✔

Your answer is correct.

The correct answers are: The file can be executed by anyone, The file can be read only by the owner, The file can be written only by the owner, 'rm' on the file by any user will work

Question **18**

Incorrect

Mark 0.00 out of 1.00

Note: for this question you get full marks if you select all and only correct options, you get ZERO if at least one option is wrong or not selected.

Select all the correct statements about log structured file systems.

- [x] a. a transaction is said to be committed when all operations are written to file system ✖
- [x] b. log may be kept on same block device or another block device ✔
- [ ] c. file system recovery may end up losing data
- [x] d. even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery ✔
- [x] e. file system recovery recovers all the lost data ✖

Your answer is incorrect.

The correct answers are: file system recovery may end up losing data, log may be kept on same block device or another block device, even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery

Question **19**

Incorrect

Mark 0.00 out of 1.00

Consider the structure of directory entry in ext2, as shown in this diagram.

Select the correct statements about the directory entry in ext2 file system.

The correct formula for rec_len is (when entries are continuously stored)

- a. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + (-1) * (strlen(name)%4)
- b. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + (strlen(name)- 4) %4
- c. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + 4 - (strlen(name)%4) ✖
- d. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + (-1) * (strlen(name)- 4)
- e. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) %4
- f. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + strlen(name)

Your answer is incorrect.

The correct answer is: rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + (-1) * (strlen(name)- 4)

Question **20**

Partially correct

Mark 0.50 out of 1.00

Mark whether the given sequence of events is possible or not-possible. Also, select the reason for your answer.

For each sequence it's a not-possible sequence if some important event is not mentioned in the sequence.

Assume that the kernel code is non-interruptible and uniprocessor system.

```
Process P1 executing a system call
Timer interrupt
Generic interrupt handler runs
Scheduler runs
Scheduler selects P2 for execution
P2 returns from timer interrupt handler
Process p2, user code executing
```

This sequence of events is: not-possible ✔

Because

Interrupts are not possible when kernel code is running ✖

Question **21**

Incorrect

Mark 0.00 out of 1.00

The given semaphore implementation faces which problem?

Assume any suitable code for signal()

Note: blocks means waits in a wait queue.

```
struct semaphore {
        int val;
        spinlock lk;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0)
        ;
        (s->val)--;
        spinunlock(&(s->sl));
}
```

- a. blocks holding a spinlock
- b. deadlock
- c. too much spinning, bounded wait not guaranteed ✖
- d. not holding lock after unblock

Your answer is incorrect.

The correct answer is: deadlock

Question **22**

Partially correct

Mark 0.80 out of 1.00

Mark statements True/False w.r.t. change of states of a process.

Reference: The process state diagram (and your understanding of how kernel code works)

Figure 3.2 Diagram of process state.

True	False		
○	●	A process in RUNNING state only can become TERMINATED because scheduler moves it to ZOMBIE state	✔
●	○	A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.	✖
●	○	A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred	✔
●	○	Every process has to go through ZOMBIE state, at least for a small duration.	✔
●	○	Only a process in READY state is considered by scheduler	✔

A process in RUNNING state only can become TERMINATED because scheduler moves it to ZOMBIE state: False
A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.: False
A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred: True
Every process has to go through ZOMBIE state, at least for a small duration.: True
Only a process in READY state is considered by scheduler: True

Question **23**

Correct

Mark 1.00 out of 1.00

Select T/F for statements about Volume Managers.
Do pay attention to the use of the words physical partition and physical volume.

True	False			
◉	○	The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.	✔	
◉	○	A logical volume can be extended in size but upto the size of volume group	✔	
◉	○	A logical volume may span across multiple physical volumes	✔	
◉	○	The volume manager stores additional metadata on the physical disk partitions	✔	
◉	○	A physical partition should be initialized as a physial volume, before it can be used by volume manager.	✔	
◉	○	A volume group consists of multiple physical volumes	✔	
◉	○	A logical volume may span across multiple physical partitions	✔	since a physical volume is made up of physical partitions, and a volume can span across multiple PVs, it can also span across multiple PP

The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.: True
A logical volume can be extended in size but upto the size of volume group: True
A logical volume may span across multiple physical volumes: True
The volume manager stores additional metadata on the physical disk partitions: True
A physical partition should be initialized as a physial volume, before it can be used by volume manager.: True
A volume group consists of multiple physical volumes: True
A logical volume may span across multiple physical partitions: True

Question **24**

Correct

Mark 1.00 out of 1.00

Map the block allocation scheme with the problem it suffers from

(Match pairs 1-1, match a scheme with the problem that it suffers from relatively the most, compared to others)

Continuous allocation	need for compaction	✔
Linked allocation	Too many seeks	✔
Indexed Allocation	Overhead of reading metadata blocks	✔

Your answer is correct.

The correct answer is: Continuous allocation → need for compaction, Linked allocation → Too many seeks, Indexed Allocation → Overhead of reading metadata blocks

Question **25**

Correct

Mark 1.00 out of 1.00

This one is not a system call:

- ○ a. open
- ○ b. read
- ○ c. write
- ◉ d. scheduler ✔

Your answer is correct.

The correct answer is: scheduler

Question **26**

Correct

Mark 1.00 out of 1.00

Match the pairs.

This question is based on your general knowledge about operating systems/related concepts and their features.

Java thereads	monitors,re-entrant locks, semaphores ✔
Linux threads	atomic-instructions, spinlocks, etc. ✔
POSIX threads	semaphore, mutex, condition variables ✔

Your answer is correct.

The correct answer is: Java thereads → monitors,re-entrant locks, semaphores, Linux threads → atomic-instructions, spinlocks, etc., POSIX threads → semaphore, mutex, condition variables

Question **27**

Correct

Mark 1.00 out of 1.00

Consider the following list of free chunks, in continuous memory management:

7k, 15k, 21k, 14k, 19k, 6k

Suppose there is a request for chunk of size 5k, then the free chunk selected under each of the following schemes will be

Best fit: 6k ✔

First fit: 7k ✔

Worst fit: 21k ✔

Question **28**

Correct

Mark 1.00 out of 1.00

This one is not a scheduling algorithm

- a. Round Robin
- b. SJF
- c. Mergesort ✔
- d. FCFS

Your answer is correct.

The correct answer is: Mergesort

Question **29**

Correct

Mark 1.00 out of 1.00

Mark whether the concept is related to scheduling or not.

Yes	No		
✓		timer interrupt	✓
✓		context-switch	✓
✓		ready-queue	✓
	✓	file-table	✓
✓		runnable process	✓

timer interrupt: Yes
context-switch: Yes
ready-queue: Yes
file-table: No
runnable process: Yes

Question **30**

Partially correct

Mark 1.00 out of 2.00

Map ext2 data structure features with their purpose

Many copies of Superblock	Choose...
Free blocks count in superblock and group descriptor ✖	Redundancy to ensure the most crucial data structure is not lost
Used directories count in group descriptor ✖	is redundant and helps do calculations of directory entries faster
Combining file type and access rights in one variable ✔	saves 1 byte of space
rec_len field in directory entry ✖	Try to keep all the data of a directory and it's file close together in a group
File Name is padded ✔	aligns all memory accesses on word boundary, improving performance
Inode bitmap is one block ✔	limits total number of files that can belong to a group
Block bitmap is one block ✔	Limits the size of a block group, thus improvising on purpose of a group
Mount count in superblock ✔	to enforce file check after certain amount of mounts at boot time
Inode table location in Group Descriptor ✖	is redundant and helps do calculations of directory entries faster
Inode table ✔	All inodes are kept together so that one disk read leads to reading many inodes together, effectively doing a buffering of subsequent inode reads, and to save space on disk
A group ✖	Redundancy to ensure the most crucial data structure is not lost

Your answer is partially correct.

You have correctly selected 6.

The correct answer is: Many copies of Superblock → Redundancy to ensure the most crucial data structure is not lost, Free blocks count in superblock and group descriptor → Redundancy to help fsck restore consistency, Used directories count in group descriptor → attempt is made to evenly spread the first-level directories, this count is used there, Combining file type and access rights in one variable → saves 1 byte of space, rec_len field in directory entry → allows holes and linking of entries in directory, File Name is padded → aligns all memory accesses on word boundary, improving performance, Inode bitmap is one block → limits total number of files that can belong to a group, Block bitmap is one block → Limits the size of a block group, thus improvising on purpose of a group, Mount count in superblock → to enforce file check after certain amount of mounts at boot time, Inode table location in Group Descriptor → Obvious, as it's per group and not per file-system, Inode table → All inodes are kept together so that one disk read leads to reading many inodes together, effectively doing a buffering of subsequent inode reads, and to save space on disk, A group → Try to keep all the data of a directory and it's file close together in a group

Question **31**

Partially correct

Mark 1.85 out of 2.00

Mark True/False

Statements about scheduling and scheduling algorithms

True	False		
●	○	The nice() system call is used to set priorities for processes	✔
●	○	Aging is used to ensure that low-priority processes do not starve in priority scheduling.	✔
○	●	In non-pre-emptive priority scheduling, the highest priority process is scheduled and runs until it gives up CPU.	✖
●	○	xv6 code does not care about Processor Affinity	✔
●	○	In pre-emptive priority scheduling, priority is implemented by assigning more time quantum to higher priority process.	✔
●	○	A scheduling algorithm is non-premptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.	✔
●	○	Processor Affinity refers to memory accesses of a process being stored on cache of that processor	✔
●	○	Response time will be quite poor on non-interruptible kernels	✔
●	○	Shortest Remaining Time First algorithm is nothing but pre-emptive Shortest Job First algorithm	✔
●	○	On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread	✔
●	○	Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.	✔
●	○	Pre-emptive scheduling leads to many race conditions in kernel code.	✔
●	○	Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.	✔

The nice() system call is used to set priorities for processes: True
Aging is used to ensure that low-priority processes do not starve in priority scheduling.: True
In non-pre-emptive priority scheduling, the highest priority process is scheduled and runs until it gives up CPU.: True
xv6 code does not care about Processor Affinity: True
In pre-emptive priority scheduling, priority is implemented by assigning more time quantum to higher priority process.: True
A scheduling algorithm is non-premptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.: True
Processor Affinity refers to memory accesses of a process being stored on cache of that processor: True
Response time will be quite poor on non-interruptible kernels: True
Shortest Remaining Time First algorithm is nothing but pre-emptive Shortest Job First algorithm: True
On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread: True
Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.: True
Pre-emptive scheduling leads to many race conditions in kernel code.: True
Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.: True

Question **32**

Partially correct

Mark 1.17 out of 2.00

The unix file semantics demand that changes to any open file are visible immediately to any other processes accessing that file at that point in time.

Select the data-structure/programmatic features that ensure the implementation of unix semantics. (Assume there is no mmap())

Yes	No		
○	●	All processes accessing the same file share the file descriptor among themselves	✔
○	●	The pointer entry in the file descriptor array entry points to the data of the file directly	✔
●	○	There is only one global file structure per on-disk file.	✖
○	●	All file accesses are made using only global variables	✔
●	○	The 'file offset' is shared among all the processes that access the file.	✖
○	●	No synchronization is implemented so that changes are made available immediately.	✔
●	○	A single spinlock is to be used to protect the unique global 'file structure' representing the file, thus synchronizing access, and making other processes wait for earlier process to finish writing so that writes get visible immediately.	✖
●	○	There is only one in-memory copy of the on disk file's contents in kernel memory/buffers	✔
●	○	The file descriptors in every PCB are pointers to the same global file structure.	✖
○	●	The file descriptor array is external to PCB and all processes that share a file, have pointers to same file-descriptors' array	✔
●	○	All file structures representing any open file, give access to the same in-memory copy of the file's contents	✔
●	○	The 'file offset' index is stored outside the file-structure to which file-descriptor array points	✖

All processes accessing the same file share the file descriptor among themselves: No
The pointer entry in the file descriptor array entry points to the data of the file directly: No
There is only one global file structure per on-disk file.: No
All file accesses are made using only global variables: No
The 'file offset' is shared among all the processes that access the file.: No
No synchronization is implemented so that changes are made available immediately.: No
A single spinlock is to be used to protect the unique global 'file structure' representing the file, thus synchronizing access, and making other processes wait for earlier process to finish writing so that writes get visible immediately.: No
There is only one in-memory copy of the on disk file's contents in kernel memory/buffers: Yes
The file descriptors in every PCB are pointers to the same global file structure.: No
The file descriptor array is external to PCB and all processes that share a file, have pointers to same file-descriptors' array: No
All file structures representing any open file, give access to the same in-memory copy of the file's contents: Yes
The 'file offset' index is stored outside the file-structure to which file-descriptor array points: No

Question **33**

Partially correct

Mark 0.33 out of 2.00

Map the function in xv6's file system code, to it's perceived logical layer.

namei	inode	✗
filestat()	Choose...	
dirlookup	directory	✓
ialloc	file descriptor	✗
stati	Choose...	
ideintr	buffer cache	✗
bread	Choose...	
balloc	file descriptor	✗
sys_chdir()	system call	✓
skipelem	system call	✗
commit	system call	✗
bmap	system call	✗

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: namei → pathname lookup, filestat() → file descriptor, dirlookup → directory, ialloc → inode, stati → inode, ideintr → disk driver, bread → buffer cache, balloc → block allocation on disk, sys_chdir() → system call, skipelem → pathname lookup, commit → logging, bmap → inode

◄ Course Exit Feedback

Jump to...

xv6-public-master ►

Started on	Monday, 24 January 2022, 7:07:42 PM
State	Finished
Completed on	Monday, 24 January 2022, 8:08:11 PM
Time taken	1 hour
Grade	**8.90** out of 20.00 (**45**%)

Question **1**

Complete

Mark 0.80 out of 1.00

Match the register with the segment used with it.

ebp	ss
eip	cs
edi	ds
esp	ss
esi	ds

The correct answer is: ebp → ss, eip → cs, edi → es, esp → ss, esi → ds

Question **2**

Complete

Mark 1.00 out of 1.00

```
int value = 5;
int main()
{
   pid_t pid;
   pid = fork();
   if (pid == 0) { /* child process */
      value += 15;
      return 0;
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("%d", value); /* LINE A */
   }
   return 0;
}
```

What's the value printed here at LINE A?

Answer: 5

The correct answer is: 5

Question **3**

Complete

Mark 0.50 out of 0.50

Is the command "cat README > done &" possible on xv6? (Note the & in the end)

- a. no
- b. yes

The correct answer is: yes

Question **4**

Complete

Mark 0.00 out of 2.00

xv6.img: bootblock kernel

```
    dd if=/dev/zero of=xv6.img count=10000
    dd if=bootblock of=xv6.img conv=notrunc
    dd if=kernel of=xv6.img seek=1 conv=notrunc
```

Consider above lines from the Makefile. Which of the following is incorrect?

- [x] a. The xv6.img is the virtual disk that is created by combining the bootblock and the kernel file.
- [x] b. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk.
- [ ] c. The size of xv6.img is exactly = (size of bootblock) + (size of kernel)
- [ ] d. xv6.img is the virtual processor used by the qemu emulator
- [ ] e. The size of the kernel file is nearly 5 MB
- [ ] f. Blocks in xv6.img after kernel may be all zeroes.
- [ ] g. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies 10,000 blocks on the disk.
- [ ] h. The kernel is located at block-1 of the xv6.img
- [ ] i. The bootblock is located on block-0 of the xv6.img
- [ ] j. The bootblock may be 512 bytes or less (looking at the Makefile instruction)
- [x] k. The size of the xv6.img is nearly 5 MB

The correct answers are: xv6.img is the virtual processor used by the qemu emulator, The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk., The size of the kernel file is nearly 5 MB, The size of xv6.img is exactly = (size of bootblock) + (size of kernel)

Question **5**

Complete

Mark 0.43 out of 1.00

Rank the following storage systems from slowest (first) to fastest(last)

You can drag and drop the items below/above each other.

- Registers
- Cache
- Main memory
- Nonvolatile memory
- Magnetic tapes
- Optical disk
- Hard-disk drives

The correct order for these items is as follows:

1. Magnetic tapes
2. Optical disk
3. Hard-disk drives
4. Nonvolatile memory
5. Main memory
6. Cache
7. Registers

Question **6**

Complete

Mark 1.00 out of 1.00

How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security) ?

Select one:

- a. It disallows hardware interrupts when a process is running
- b. It prohibits one process from accessing other process's memory
- c. It prohibits a user mode process from running privileged instructions
- d. It prohibits invocation of kernel code completely, if a user program is running

The correct answer is: It prohibits a user mode process from running privileged instructions

Question **7**

Complete

Mark 0.00 out of 2.00

Which of the following are NOT a part of job of a typical compiler?

- [ ] a. Suggest alternative pieces of code that can be written
- [x] b. Check the program for syntactical errors
- [ ] c. Check the program for logical errors
- [x] d. Convert high level langauge code to machine code
- [x] e. Process the # directives in a C program
- [ ] f. Invoke the linker to link the function calls with their code, extern globals with their declaration

The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

Question **8**

Complete

Mark 0.00 out of 2.00

Match the program with it's output (ignore newlines in the output. Just focus on the count of the number of 'hi')

main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }	hi hi
main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }	hi hi hi hi
main() { int i = NULL; fork(); printf("hi\n"); }	No output
main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }	hi

The correct answer is: main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi, main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi, main() { int i = NULL; fork(); printf("hi\n"); } → hi hi, main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi hi

Question **9**

Complete

Mark 0.83 out of 2.00

Select all statements that correctly explain the use/purpose of system calls.

Select one or more:

- [x] a. Provide an environment for process creation
- [ ] b. Switch from user mode to kernel mode
- [ ] c. Handle ALL types of interrupts
- [ ] d. Handle exceptions like division by zero
- [x] e. Run each instruction of an application program
- [x] f. Allow I/O device access to user processes
- [x] g. Provide services for accessing files

The correct answers are: Switch from user mode to kernel mode, Provide services for accessing files, Allow I/O device access to user processes, Provide an environment for process creation

Question **10**

Complete

Mark 0.50 out of 0.50

Compare multiprogramming with multitasking

- (•) a. A multiprogramming system is not necessarily multitasking
- ( ) b. A multitasking system is not necessarily multiprogramming

The correct answer is: A multiprogramming system is not necessarily multitasking

Question **11**

Complete

Mark 0.60 out of 1.00

Select all the correct statements about two modes of CPU operation

Select one or more:

- [x] a. The two modes are essential for a multitasking system
- [ ] b. Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode
- [x] c. The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously
- [ ] d. The two modes are essential for a multiprogramming system
- [x] e. There is an instruction like 'iret' to return from kernel mode to user mode

The correct answers are: The two modes are essential for a multiprogramming system, The two modes are essential for a multitasking system, There is an instruction like 'iret' to return from kernel mode to user mode, The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously, Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode

Question **12**

Complete

Mark 0.50 out of 0.50

Is the terminal a part of the kernel on GNU/Linux systems?

- (●) a. no
- ( ) b. yes

The correct answer is: no

Question **13**

Complete

Mark 1.00 out of 1.00

Why should a program exist in memory before it starts executing ?

- ( ) a. Because the hard disk is a slow medium
- (●) b. Becase the processor can run instructions and access data only from memory
- ( ) c. Because the variables of the program are stored in memory
- ( ) d. Because the memory is volatile

The correct answer is: Becase the processor can run instructions and access data only from memory

Question **14**

Complete

Mark 1.33 out of 2.00

Which of the following instructions should be privileged?

Select one or more:

- [ ] a. Read the clock.
- [x] b. Access memory management unit of the processor
- [ ] c. Access I/O device.
- [x] d. Turn off interrupts.
- [ ] e. Set value of a memory location
- [x] f. Set value of timer.
- [ ] g. Access a general purpose register
- [x] h. Modify entries in device-status table
- [ ] i. Switch from user to kernel mode.

The correct answers are: Set value of timer., Access memory management unit of the processor, Turn off interrupts., Modify entries in device-status table, Access I/O device., Switch from user to kernel mode.

Question **15**

Complete

Mark 0.07 out of 2.00

Select all the correct statements about calling convention on x86 32-bit.

- [ ] a. The ebp pointers saved on the stack constitute a chain of activation records
- [x] b. The return value is either stored on the stack or returned in the eax register
- [ ] c. The two lines in the beginning of each function, "push %ebp; mov %esp, %ebp", create space for local variables
- [x] d. Parameters may be passed in registers or on stack
- [ ] e. Return address is one location above the ebp
- [ ] f. Parameters may be passed in registers or on stack
- [x] g. Compiler may allocate more memory on stack than needed
- [ ] h. Space for local variables is allocated by substracting the stack pointer inside the code of the called function
- [ ] i. Paramters are pushed on the stack in left-right order
- [ ] j. during execution of a function, ebp is pointing to the old ebp
- [ ] k. Space for local variables is allocated by substracting the stack pointer inside the code of the caller function

The correct answers are: Compiler may allocate more memory on stack than needed, Parameters may be passed in registers or on stack, Parameters may be passed in registers or on stack, Return address is one location above the ebp, during execution of a function, ebp is pointing to the old ebp, Space for local variables is allocated by substracting the stack pointer inside the code of the called function, The ebp pointers saved on the stack constitute a chain of activation records

Question **16**

Complete

Mark 0.33 out of 0.50

Order the following events in boot process (from 1 onwards)

Shell	3
BIOS	1
Init	4
OS	6
Login interface	5
Boot loader	2

The correct answer is: Shell → 6, BIOS → 1, Init → 4, OS → 3, Login interface → 5, Boot loader → 2

◄ (Task) Compulsory xv6 task

Jump to...

(Optional Assignment) Shell Programming(Conformance tests) ►

Started on	Wednesday, 9 February 2022, 7:00:12 PM
State	Finished
Completed on	Wednesday, 9 February 2022, 7:46:38 PM
Time taken	46 mins 26 secs
Grade	**3.00** out of 11.00 (**27**%)

Question **1**

Complete

Mark 0.00 out of 1.00

The number of GDT entries setup during boot process of xv6 is

- a. 2
- b. 3
- c. 0
- d. 256
- e. 4
- f. 255

The correct answer is: 3

Question **2**

Complete

Mark 0.00 out of 1.00

x86 provides which of the following type of memory management options?

- a. segmentation and one level paging
- b. segmentation or one or two level paging
- c. segmentation and two level paging
- d. segmentation or paging
- e. segmentation and one or two level paging
- f. segmentation only

The correct answer is: segmentation and one or two level paging

Question **3**

Complete

Mark 0.00 out of 1.00

which of the following is not a difference between real mode and protected mode

- a. in real mode general purpose registers are 16 bit, in protected mode they are 32 bit
- b. in real mode the addressable memory is more than in protected mode
- c. in real mode the addressable memory is less than in protected mode
- d. in real mode the segment is multiplied by 16, in protected mode segment is used as index in GDT
- e. processor starts in real mode

The correct answer is: in real mode the addressable memory is more than in protected mode

Question **4**

Complete

Mark 0.00 out of 1.00

The kernel ELF file contains how many Program headers?

- a. 4
- b. 2
- c. 3
- d. 9
- e. 10

The correct answer is: 3

Question **5**

Not answered

Marked out of 0.50

code line, MMU setting: Match the line of xv6 code with the MMU setup employed

Answer:

The correct answer is: inb $0x64,%al

Question **6**

Complete

Mark 1.00 out of 1.00

The kernel is loaded at Physical Address

- a. 0x00100000
- b. 0x0010000
- c. 0x80100000
- d. 0x80000000

The correct answer is: 0x00100000

Question **7**

Complete

Mark 0.00 out of 1.00

Why is the code of entry() in Assembly and not in C?

- a. Because the symbol entry() is inside the ELF file
- b. Because the kernel code must begin in assembly
- c. Because it needs to setup paging
- d. There is no particular reason, it could also be in C

The correct answer is: Because it needs to setup paging

Question **8**

Complete

Mark 1.00 out of 1.00

The ljmp instruction in general does

- a. change the CS and EIP to 32 bit mode, and jumps to next line of code
- b. change the CS and EIP to 32 bit mode, and jumps to new value of EIP
- c. change the CS and EIP to 32 bit mode
- d. change the CS and EIP to 32 bit mode, and jumps to kernel code

The correct answer is: change the CS and EIP to 32 bit mode, and jumps to new value of EIP

Question **9**

Complete

Mark 0.00 out of 1.00

The variable $stack in entry.S is

- a. located at the value given by %esp as setup by bootmain()
- b. located at 0x7c00
- c. a memory region allocated as a part of entry.S
- d. located at less than 0x7c00
- e. located at 0

The correct answer is: a memory region allocated as a part of entry.S

Question **10**

Not answered

Marked out of 0.50

Match the pairs of which action is taken by whom

Answer:

The correct answer is: kernel

Question **11**

Complete

Mark 0.00 out of 1.00

ELF Magic number is

- a. 0xFFFFFFFFF
- b. 0
- c. 0xELFELFELF
- d. 0xELF
- e. 0x0x464CELF
- f. 0x464C457FL
- g. 0x464C457FU

The correct answer is: 0x464C457FU

Question **12**

Complete

Mark 1.00 out of 1.00

The right side of line of code "entry = (void(*)(void))(elf->entry)" means

- a. Convert the "entry" in ELF structure into void
- b. Get the "entry" in ELF structure and convert it into a function pointer accepting no arguments and returning nothing
- c. Get the "entry" in ELF structure and convert it into a function void pointer
- d. Get the "entry" in ELF structure and convert it into a void pointer

The correct answer is: Get the "entry" in ELF structure and convert it into a function pointer accepting no arguments and returning nothing

◄ Homework questions: Basics of MM, xv6 booting

Jump to...

(Code) Files, redirection, dup, (IPC)pipe ►

Started on	Monday, 21 February 2022, 7:00:28 PM
State	Finished
Completed on	Monday, 21 February 2022, 7:49:24 PM
Time taken	48 mins 56 secs
Grade	**8.30** out of 10.00 (**83**%)

Question **1**

Complete

Mark 0.80 out of 1.00

Match the elements of C program to their place in memory

Local Static variables	Data
Global variables	Data
Code of main()	Code
Function code	Code
Arguments	Stack
Malloced Memory	Heap
#include files	No Memory needed
#define MACROS	No memory needed
Local Variables	Stack
Global Static variables	Data

The correct answer is: Local Static variables → Data, Global variables → Data, Code of main() → Code, Function code → Code, Arguments → Stack, Malloced Memory → Heap, #include files → No memory needed, #define MACROS → No Memory needed, Local Variables → Stack, Global Static variables → Data

Question **2**

Complete

Mark 1.00 out of 1.00

What will be the output of this program

```
int main() {
  int fd;
  printf("%d ",   open("/etc/passwd", O_RDONLY));
  close(1);
  fd = printf("%d ", open("/etc/passwd", O_RDONLY));
  close(fd);
   fd = printf("%d ", open("/etc/passwd", O_RDONLY));
}
```

- a. 3 1 2
- b. 3 3 3
- c. 3 1 1
- d. 1 1 1
- e. 3 4 5
- f. 2 2 2

The correct answer is: 3 1 1

Question **3**

Complete

Mark 1.00 out of 1.00

Arrange in correct order, the files involved in execution of system call

vectors.S	2
trap.c	4
usys.S	1
trapasm.S	3

The correct answer is: vectors.S → 2, trap.c → 4, usys.S → 1, trapasm.S → 3

Question **4**

Complete

Mark 1.00 out of 1.00

The "push 0" in vectors.S is

- a. A placeholder to match the size of struct trapframe
- b. Place for the error number value
- c. To indicate that it's a system call and not a hardware interrupt
- d. To be filled in as the return value of the system call

The correct answer is: Place for the error number value

Question **5**

Complete

Mark 0.00 out of 1.00

A process blocks itself means

- a. The kernel code of an interrupt handler, moves the process to a waiting queue and calls scheduler
- b. The application code calls the scheduler
- c. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler
- d. The kernel code of system call calls scheduler

The correct answer is: The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

Question **6**

Complete

Mark 1.00 out of 1.00

Select the odd one out

- a. Kernel stack of new process to Process stack of new process
- b. Process stack of running process to kernel stack of running process
- c. Kernel stack of running process to kernel stack of scheduler
- d. Kernel stack of scheduler to kernel stack of new process
- e. Kernel stack of new process to kernel stack of scheduler

The correct answer is: Kernel stack of new process to kernel stack of scheduler

Question **7**

Complete

Mark 0.50 out of 0.50

Match the File descriptors to their meaning

0	Standard Input
2	Standard error
1	Standard output

The correct answer is: 0 → Standard Input, 2 → Standard error, 1 → Standard output

Question **8**

Complete

Mark 1.00 out of 1.00

Which of the following is not a task of the code of swtch() function

- [ ] a. Switch stacks
- [x] b. Change the kernel stack location
- [ ] c. Load the new context
- [x] d. Jump to next context EIP
- [x] e. Save the return value of the old context code
- [ ] f. Save the old context

The correct answers are: Save the return value of the old context code, Change the kernel stack location

Question **9**

Complete

Mark 0.50 out of 0.50

Match the names of PCB structures with kernel

linux	struct task_struct
xv6	struct proc

The correct answer is: linux → struct task_struct, xv6 → struct proc

Question **10**

Complete

Mark 0.50 out of 0.50

Match the MACRO with it's meaning

KERNBASE	2 GB
KERNLINK	2.224 GB
PHYSTOP	224 MB

The correct answer is: KERNBASE → 2 GB, KERNLINK → 2.224 GB, PHYSTOP → 224 MB

Question **11**

Complete

Mark 1.00 out of 1.00

The trapframe, in xv6, is built by the

- a. hardware, trapasm.S
- b. hardware, vectors.S
- c. hardware, vectors.S, trapasm.S, trap()
- d. hardware, vectors.S, trapasm.S
- e. vectors.S, trapasm.S

The correct answer is: hardware, vectors.S, trapasm.S

Question **12**

Complete

Mark 0.00 out of 0.50

Which of the following state transitions are not possible?

- [ ] a. Running -> Waiting
- [x] b. Ready -> Waiting
- [x] c. Waiting -> Terminated
- [x] d. Ready -> Terminated

The correct answers are: Ready -> Terminated, Waiting -> Terminated, Ready -> Waiting

◄ Description of some possible course mini projects

Jump to...

(Code) mmap related programs ►

Started on	Saturday, 26 February 2022, 5:18:30 PM
State	Finished
Completed on	Saturday, 26 February 2022, 6:30:44 PM
Time taken	1 hour 12 mins
Grade	**8.55** out of 15.00 (**57**%)

Question **1**

Complete

Mark 0.50 out of 0.50

Map the technique with it's feature/problem

static linking	large executable file
dynamic loading	allocate memory only if needed
dynamic linking	small executable file
static loading	wastage of physical memory

The correct answer is: static linking → large executable file, dynamic loading → allocate memory only if needed, dynamic linking → small executable file, static loading → wastage of physical memory

Question **2**

Complete

Mark 0.00 out of 1.00

Calculate the EAT in NANO-seconds (upto 2 decimal points) w.r.t. a page fault, given

Memory access time = 299 ns

Average page fault service time = 6 ms

Page fault rate = 0.8

Answer: 4.80

The correct answer is: 4800059.80

Question **3**

Complete

Mark 1.00 out of 1.00

Given six memory partitions of 300 KB , 600 KB , 350 KB , 200 KB , 750 KB , and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB and 500 KB (in order)?

best fit 115 KB	125 KB
best fit 500 KB	600 KB
worst fit 500 KB	635 KB
worst fit 115 KB	750 KB
first fit 500 KB	600 KB
first fit 115 KB	300 KB

The correct answer is: best fit 115 KB → 125 KB, best fit 500 KB → 600 KB, worst fit 500 KB → 635 KB, worst fit 115 KB → 750 KB, first fit 500 KB → 600 KB, first fit 115 KB → 300 KB

Question **4**

Complete

Mark 0.29 out of 1.00

Compare paging with demand paging and select the correct statements.

Select one or more:

- [ ] a. TLB hit ration has zero impact in effective memory access time in demand paging.
- [x] b. Both demand paging and paging support shared memory pages.
- [ ] c. Calculations of number of bits for page number and offset are same in paging and demand paging.
- [ ] d. Demand paging requires additional hardware support, compared to paging.
- [ ] e. The meaning of valid-invalid bit in page table is different in paging and demand-paging.
- [ ] f. Paging requires NO hardware support in CPU
- [ ] g. With paging, it's possible to have user programs bigger than physical memory.
- [ ] h. With demand paging, it's possible to have user programs bigger than physical memory.
- [ ] i. Paging requires some hardware support in CPU
- [x] j. Demand paging always increases effective memory access time.

The correct answers are: Demand paging requires additional hardware support, compared to paging., Both demand paging and paging support shared memory pages., With demand paging, it's possible to have user programs bigger than physical memory., Demand paging always increases effective memory access time., Paging requires some hardware support in CPU, Calculations of number of bits for page number and offset are same in paging and demand paging., The meaning of valid-invalid bit in page table is different in paging and demand-paging.

Question **5**

Complete

Mark 0.36 out of 0.50

Map the parts of a C code to the memory regions they are related to

local variables	stack
global un-initialized variables	bss
static variables	data
global initialized variables	data
function arguments	stack
malloced memory	stack
functions	stack

The correct answer is: local variables → stack, global un-initialized variables → bss, static variables → data, global initialized variables → data, function arguments → stack, malloced memory → heap, functions → code

Question **6**

Complete

Mark 0.75 out of 1.00

which of the following, do you think, are valid concerns for making the kernel pageable?

- [x] a. The kernel must have some dedicated frames for it's own work
- [ ] b. No data structure of kernel should be pageable
- [x] c. The kernel's own page tables should not be pageable
- [ ] d. The disk driver and disk interrupt handler should not be pageable
- [ ] e. No part of kernel code should be pageable.
- [x] f. The page fault handler should not be pageable

The correct answers are: The kernel's own page tables should not be pageable, The page fault handler should not be pageable, The kernel must have some dedicated frames for it's own work, The disk driver and disk interrupt handler should not be pageable

Question **7**

Complete

Mark 0.75 out of 1.00

W.r.t the figure given below, mark the given statements as True or False.

Figure 10.9 Need for page replacement.

True	False	
◉	○	Kernel occupies two page frames
◉	○	Handling this scenario demands two disk I/Os
◉	○	Local replacement means chose any of the frames 2, 3, 6
○	◉	Local replacement means chose any of the frame from 2 to 7
◉	○	Global replacement means chose any of the frame from 0 to 7
○	◉	Global replacement means chose any of the frame from 2 to 7
◉	○	The kernel's pages can not used for replacement if kernel is not pageable.
◉	○	Page 1 of process 1 needs a replacement

Kernel occupies two page frames: True
Handling this scenario demands two disk I/Os: True
Local replacement means chose any of the frames 2, 3, 6: True
Local replacement means chose any of the frame from 2 to 7: False
Global replacement means chose any of the frame from 0 to 7: False
Global replacement means chose any of the frame from 2 to 7: True
The kernel's pages can not used for replacement if kernel is not pageable.: True
Page 1 of process 1 needs a replacement: True

Question **8**

Complete

Mark 0.67 out of 1.00

Shared memory is possible with which of the following memory management schemes ?

Select one or more:

- [x] a. paging
- [x] b. segmentation
- [ ] c. continuous memory management
- [ ] d. demand paging

The correct answers are: paging, segmentation, demand paging

Question **9**

Complete

Mark 0.60 out of 1.00

Given below is the "maps" file for a particular instance of "vim.basic" process.

Mark the given statements as True or False, w.r.t. the contents of the map file.

```
55a43501b000-55a435049000 r--p 00000000 103:05 917529                    /usr/bin/vim.basic
55a435049000-55a435248000 r-xp 0002e000 103:05 917529                    /usr/bin/vim.basic
55a435248000-55a4352b6000 r--p 0022d000 103:05 917529                    /usr/bin/vim.basic
55a4352b7000-55a4352c5000 r--p 0029b000 103:05 917529                    /usr/bin/vim.basic
55a4352c5000-55a4352e2000 rw-p 002a9000 103:05 917529                    /usr/bin/vim.basic
55a4352e2000-55a4352f0000 rw-p 00000000 00:00 0
55a436bc9000-55a436e5b000 rw-p 00000000 00:00 0                          [heap]
7f275b0a3000-7f275b0a6000 r--p 00000000 103:05 917901                    /usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so
7f275b0a6000-7f275b0ad000 r-xp 00003000 103:05 917901                    /usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so
7f275b0ad000-7f275b0af000 r--p 0000a000 103:05 917901                    /usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so
7f275b0af000-7f275b0b0000 r--p 0000b000 103:05 917901                    /usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so
7f275b0b0000-7f275b0b1000 rw-p 0000c000 103:05 917901                    /usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so
7f275b0b1000-7f275b0b7000 rw-p 00000000 00:00 0
7f275b0b7000-7f275b8f5000 r--p 00000000 103:05 925247                    /usr/lib/locale/locale-archive
7f275b8f5000-7f275b8fa000 rw-p 00000000 00:00 0
7f275b8fa000-7f275b8fc000 r--p 00000000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b8fc000-7f275b901000 r-xp 00002000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b901000-7f275b904000 r--p 00007000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b904000-7f275b905000 ---p 0000a000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b905000-7f275b906000 r--p 0000a000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b906000-7f275b907000 rw-p 0000b000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b907000-7f275b90a000 r--p 00000000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b90a000-7f275b921000 r-xp 00003000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b921000-7f275b932000 r--p 0001a000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b932000-7f275b933000 ---p 0002b000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b933000-7f275b934000 r--p 0002b000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b934000-7f275b935000 rw-p 0002c000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b935000-7f275b937000 rw-p 00000000 00:00 0
7f275b937000-7f275b938000 r--p 00000000 103:05 917914                    /usr/lib/x86_64-linux-
gnu/libutil-2.31.so
7f275b938000-7f275b939000 r-xp 00001000 103:05 917914                    /usr/lib/x86_64-linux-
gnu/libutil-2.31.so
7f275b939000-7f275b93a000 r--p 00002000 103:05 917914                    /usr/lib/x86_64-linux-
gnu/libutil-2.31.so
7f275b93a000-7f275b93b000 r--p 00002000 103:05 917914                    /usr/lib/x86_64-linux-
gnu/libutil-2.31.so
7f275b93b000-7f275b93c000 rw-p 00003000 103:05 917914                    /usr/lib/x86_64-linux-
```

```
gnu/libutil-2.31.so
7f275b93c000-7f275b93e000 r--p 00000000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b93e000-7f275b94f000 r-xp 00002000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b94f000-7f275b955000 r--p 00013000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b955000-7f275b956000 ---p 00019000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b956000-7f275b957000 r--p 00019000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b957000-7f275b958000 rw-p 0001a000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b958000-7f275b95c000 r--p 00000000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b95c000-7f275b978000 r-xp 00004000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b978000-7f275b982000 r--p 00020000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b982000-7f275b983000 ---p 0002a000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b983000-7f275b985000 r--p 0002a000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b985000-7f275b986000 rw-p 0002c000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b986000-7f275b988000 r--p 00000000 103:05 924057                    /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b988000-7f275b98d000 r-xp 00002000 103:05 924057                    /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b98d000-7f275b98f000 r--p 00007000 103:05 924057                    /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b98f000-7f275b990000 r--p 00008000 103:05 924057                    /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b990000-7f275b991000 rw-p 00009000 103:05 924057                    /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b991000-7f275b995000 r--p 00000000 103:05 921934                    /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b995000-7f275b9a3000 r-xp 00004000 103:05 921934                    /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b9a3000-7f275b9a9000 r--p 00012000 103:05 921934                    /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b9a9000-7f275b9aa000 r--p 00017000 103:05 921934                    /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b9aa000-7f275b9ab000 rw-p 00018000 103:05 921934                    /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b9ab000-7f275b9ad000 rw-p 00000000 00:00 0
7f275b9ad000-7f275b9af000 r--p 00000000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9af000-7f275b9b4000 r-xp 00002000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b4000-7f275b9b5000 r--p 00007000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b5000-7f275b9b6000 ---p 00008000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b6000-7f275b9b7000 r--p 00008000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b7000-7f275b9b8000 rw-p 00009000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b8000-7f275b9ba000 r--p 00000000 103:05 924277                    /usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0
7f275b9ba000-7f275ba1e000 r-xp 00002000 103:05 924277                    /usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0
```

```
7f275ba1e000-7f275ba46000 r--p 00066000 103:05 924277                    /usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0
7f275ba46000-7f275ba47000 r--p 0008d000 103:05 924277                    /usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0
7f275ba47000-7f275ba48000 rw-p 0008e000 103:05 924277                    /usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0
7f275ba48000-7f275ba6d000 r--p 00000000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275ba6d000-7f275bbe5000 r-xp 00025000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275bbe5000-7f275bc2f000 r--p 0019d000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275bc2f000-7f275bc30000 ---p 001e7000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275bc30000-7f275bc33000 r--p 001e7000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275bc33000-7f275bc36000 rw-p 001ea000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275bc36000-7f275bc3a000 rw-p 00000000 00:00 0
7f275bc3a000-7f275bc41000 r--p 00000000 103:05 917906                    /usr/lib/x86_64-linux-
gnu/libpthread-2.31.so
7f275bc41000-7f275bc52000 r-xp 00007000 103:05 917906                    /usr/lib/x86_64-linux-
gnu/libpthread-2.31.so
7f275bc52000-7f275bc57000 r--p 00018000 103:05 917906                    /usr/lib/x86_64-linux-
gnu/libpthread-2.31.so
7f275bc57000-7f275bc58000 r--p 0001c000 103:05 917906                    /usr/lib/x86_64-linux-
gnu/libpthread-2.31.so
7f275bc58000-7f275bc59000 rw-p 0001d000 103:05 917906                    /usr/lib/x86_64-linux-
gnu/libpthread-2.31.so
7f275bc59000-7f275bc5d000 rw-p 00000000 00:00 0
7f275bc5d000-7f275bcce000 r--p 00000000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275bcce000-7f275bf29000 r-xp 00071000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275bf29000-7f275c142000 r--p 002cc000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275c142000-7f275c143000 ---p 004e5000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275c143000-7f275c149000 r--p 004e5000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275c149000-7f275c190000 rw-p 004eb000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275c190000-7f275c1b3000 rw-p 00000000 00:00 0
7f275c1b3000-7f275c1b4000 r--p 00000000 103:05 917894                    /usr/lib/x86_64-linux-
gnu/libdl-2.31.so
7f275c1b4000-7f275c1b6000 r-xp 00001000 103:05 917894                    /usr/lib/x86_64-linux-
gnu/libdl-2.31.so
7f275c1b6000-7f275c1b7000 r--p 00003000 103:05 917894                    /usr/lib/x86_64-linux-
gnu/libdl-2.31.so
7f275c1b7000-7f275c1b8000 r--p 00003000 103:05 917894                    /usr/lib/x86_64-linux-
gnu/libdl-2.31.so
7f275c1b8000-7f275c1b9000 rw-p 00004000 103:05 917894                    /usr/lib/x86_64-linux-
gnu/libdl-2.31.so
7f275c1b9000-7f275c1bb000 rw-p 00000000 00:00 0
7f275c1bb000-7f275c1c0000 r-xp 00000000 103:05 923815                    /usr/lib/x86_64-linux-
gnu/libgpm.so.2
7f275c1c0000-7f275c3bf000 ---p 00005000 103:05 923815                    /usr/lib/x86_64-linux-
gnu/libgpm.so.2
7f275c3bf000-7f275c3c0000 r--p 00004000 103:05 923815                    /usr/lib/x86_64-linux-
gnu/libgpm.so.2
7f275c3c0000-7f275c3c1000 rw-p 00005000 103:05 923815                    /usr/lib/x86_64-linux-
gnu/libgpm.so.2
```

```
7f275c3c1000-7f275c3c3000 r--p 00000000 103:05 923315                    /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3c3000-7f275c3c8000 r-xp 00002000 103:05 923315                    /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3c8000-7f275c3ca000 r--p 00007000 103:05 923315                    /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3ca000-7f275c3cb000 r--p 00008000 103:05 923315                    /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3cb000-7f275c3cc000 rw-p 00009000 103:05 923315                    /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3cc000-7f275c3cf000 r--p 00000000 103:05 923446                    /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3cf000-7f275c3d9000 r-xp 00003000 103:05 923446                    /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3d9000-7f275c3dd000 r--p 0000d000 103:05 923446                    /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3dd000-7f275c3de000 r--p 00010000 103:05 923446                    /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3de000-7f275c3df000 rw-p 00011000 103:05 923446                    /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3df000-7f275c3e5000 r--p 00000000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c3e5000-7f275c3fe000 r-xp 00006000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c3fe000-7f275c405000 r--p 0001f000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c405000-7f275c406000 ---p 00026000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c406000-7f275c407000 r--p 00026000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c407000-7f275c408000 rw-p 00027000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c408000-7f275c40a000 rw-p 00000000 00:00 0
7f275c40a000-7f275c418000 r--p 00000000 103:05 924540                    /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c418000-7f275c427000 r-xp 0000e000 103:05 924540                    /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c427000-7f275c435000 r--p 0001d000 103:05 924540                    /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c435000-7f275c439000 r--p 0002a000 103:05 924540                    /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c439000-7f275c43a000 rw-p 0002e000 103:05 924540                    /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c43a000-7f275c449000 r--p 00000000 103:05 917895                    /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c449000-7f275c4f0000 r-xp 0000f000 103:05 917895                    /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c4f0000-7f275c587000 r--p 000b6000 103:05 917895                    /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c587000-7f275c588000 r--p 0014c000 103:05 917895                    /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c588000-7f275c589000 rw-p 0014d000 103:05 917895                    /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c589000-7f275c58b000 rw-p 00000000 00:00 0
7f275c5ae000-7f275c5af000 r--p 00000000 103:05 917889                    /usr/lib/x86_64-linux-gnu/ld-
2.31.so
7f275c5af000-7f275c5d2000 r-xp 00001000 103:05 917889                    /usr/lib/x86_64-linux-gnu/ld-
2.31.so
7f275c5d2000-7f275c5da000 r--p 00024000 103:05 917889                    /usr/lib/x86_64-linux-gnu/ld-
2.31.so
7f275c5db000-7f275c5dc000 r--p 0002c000 103:05 917889                    /usr/lib/x86_64-linux-gnu/ld-
2.31.so
```

```
7f275c5dc000-7f275c5dd000 rw-p 0002d000 103:05 917889                    /usr/lib/x86_64-linux-gnu/ld-
2.31.so
7f275c5dd000-7f275c5de000 rw-p 00000000 00:00 0
7ffd22d2f000-7ffd22d50000 rw-p 00000000 00:00 0                          [stack]
7ffd22db0000-7ffd22db4000 r--p 00000000 00:00 0                          [vvar]
7ffd22db4000-7ffd22db6000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

True	False	
○	●	The size of the heap is one page
●	○	This is a virtual memory map (not physical memory map)
○	●	vim.basic uses the math library
●	○	The 5th entry 55a4352c5000-55a4352e2000 **may** correspond to "data" of the vim.basic
●	○	The size of the stack is one page

The size of the heap is one page: False
This is a virtual memory map (not physical memory map): True
vim.basic uses the math library: True
The 5th entry 55a4352c5000-55a4352e2000 **may** correspond to "data" of the vim.basic: True
The size of the stack is one page: False

Question **10**

Complete

Mark 0.00 out of 1.00

Select all the correct statements, w.r.t. Copy on Write

- [ ] a. Fork() used COW technique to improve performance of new process creation.
- [x] b. Vfork() assumes that there will be no write, but rather exec()
- [x] c. COW helps us save memory
- [ ] d. If either parent or child modifies a COW-page, then a copy of the page is made and page table entry is updated
- [ ] e. use of COW during fork() is useless if child called exit()
- [x] f. use of COW during fork() is useless if exec() is called by the child

The correct answers are: Fork() used COW technique to improve performance of new process creation., If either parent or child modifies a COW-page, then a copy of the page is made and page table entry is updated, COW helps us save memory, Vfork() assumes that there will be no write, but rather exec()

Question **11**

Complete

Mark 1.00 out of 1.00

Assuming a 8- KB page size, what is the page numbers for the address 1093943 reference in decimal :

(give answer also in decimal)

Answer: 134

The correct answer is: 134

Question **12**

Complete

Mark 0.00 out of 1.00

Order the following events, related to page fault handling, in correct order

1. Page fault handler detects that it's a page fault and not illegal memory access
2. Disk interrupt handler runs
3. MMU detects that a page table entry is marked "invalid"
4. Page faulted process is moved to ready-queue
5. Empty frame is found
6. Page fault interrupt is generated
7. Other processes scheduled by scheduler
8. Page table of page faulted process is updated
9. Disk Interrupt occurs
10. Disk read is issued
11. Page fault handler in kernel starts executing
12. Page faulting process is made to wait in a queue

The correct order for these items is as follows:

1. MMU detects that a page table entry is marked "invalid"
2. Page fault interrupt is generated
3. Page fault handler in kernel starts executing
4. Page fault handler detects that it's a page fault and not illegal memory access
5. Empty frame is found
6. Disk read is issued
7. Page faulting process is made to wait in a queue
8. Other processes scheduled by scheduler
9. Disk Interrupt occurs
10. Disk interrupt handler runs
11. Page table of page faulted process is updated
12. Page faulted process is moved to ready-queue

Question **13**

Complete

Mark 1.00 out of 1.00

Page sizes are a power of 2 because

Select one:

- a. Certain bits are reserved for offset in logical address. Hence page size = 2^(no.of offset bits)
- b. Power of 2 calculations are highly efficient
- c. Certain bits are reserved for offset in logical address. Hence page size = 2^(32 - no.of offset bits)
- d. operating system calculations happen using power of 2
- e. MMU only understands numbers that are power of 2

The correct answer is: Certain bits are reserved for offset in logical address. Hence page size = 2^(no.of offset bits)

Question **14**

Complete

Mark 1.00 out of 1.00

Given below is the output of the command "ps -eo min_flt,maj_flt,cmd" on a Linux Desktop system. Select the statements that are consistent with the output

```
626729 482768 /usr/lib/firefox/firefox -contentproc -parentBuildID 20220202182137 -prefsLen 9256 -prefMapSize 264738 -appDir /usr/lib/firefox/browser 6094 true rdd
2167 687 /usr/sbin/apache2 -k start
1265185 222 /usr/bin/gnome-shell
102648 111 /usr/sbin/mysqld
9813 0 bash
15497 370 /usr/bin/gedit --gapplication-service
```

- [x] a. All of the processes here exihibit some good locality of reference
- [x] b. Firefox has likely been running for a large amount of time
- [x] c. The bash shell is mostly busy doing work within a particular locality
- [x] d. Apache web-server has not been doing much work

The correct answers are: Firefox has likely been running for a large amount of time, Apache web-server has not been doing much work, The bash shell is mostly busy doing work within a particular locality, All of the processes here exihibit some good locality of reference

Question **15**

Complete

Mark 0.14 out of 1.00

Suppose two processes share a library between them. The library consists of 5 pages, and these 5 pages are mapped to frames 9, 15, 23, 4, 7 respectively. Process P1 has got 6 pages, first 3 of which consist of process's own code/data and 3 correspond to library's pages 0, 2, 4. Process P2 has got 7 pages, first 3 of which consist of processe's own code/data and remaining 4 correspond to library's pages 0, 1, 3, 4. Fill in the blanks for page table entries of P1 and P2.

Page table of P1, Page 5	23
Page table of P1, Page 3	9
Page table of P2, Page 0	6
Page table of P2, Page 3	7
Page table of P2, Page 4	9
Page table of P2, Page 1	5
Page table of P1, Page 4	9

The correct answer is: Page table of P1, Page 5 → 7, Page table of P1, Page 3 → 9, Page table of P2, Page 0 → 9, Page table of P2, Page 3 → 4, Page table of P2, Page 4 → 7, Page table of P2, Page 1 → 15, Page table of P1, Page 4 → 23

Question **16**

Complete

Mark 0.50 out of 1.00

For the reference string
3 4 3 5 2

the number of page faults (including initial ones) using

FIFO replacement and 2 page frames is :

4

FIFO replacement and 3 page frames is :

3

◄ (Code) mmap related programs

Jump to...

Points from Mid-term feedback ►

Started on	Monday, 7 March 2022, 7:00:12 PM
State	Finished
Completed on	Monday, 7 March 2022, 8:00:04 PM
Time taken	59 mins 52 secs
Grade	**9.78** out of 15.00 (**65**%)

Question **1**

Complete

Mark 1.00 out of 1.00

Why is there a call to kinit2? Why is it not merged with knit1?

- a. call to seginit() makes it possible to actually use PHYSTOP in argument to kinit2()
- b. When kinit1() is called there is a need for few page frames, but later knit2() is called to serve need of more page frames
- c. Because there is a limit on the values that the argumets to knit1() can take.
- d. knit2 refers to virtual addresses beyond 4MB, which are not mapped before kvalloc() is called

The correct answer is: knit2 refers to virtual addresses beyond 4MB, which are not mapped before kvalloc() is called

Question **2**

Complete

Mark 1.50 out of 1.50

Arrange the following in the correct order of execution (w.r.t. 'init')

initcode() returns in forkret()	6
'initcode' process is marked RUNNABLE	3
'initcode' struct proc is created	2
userinit() is called	1
mpmain() calls scheduler()	4
initcode() calls exec("/init", ...)	8
initcode() returns from trapret()	7
scheduler() schedules initcode() process	5

The correct answer is: initcode() returns in forkret() → 6, 'initcode' process is marked RUNNABLE → 3, 'initcode' struct proc is created → 2, userinit() is called → 1, mpmain() calls scheduler() → 4, initcode() calls exec("/init", ...) → 8, initcode() returns from trapret() → 7, scheduler() schedules initcode() process → 5

Question **3**

Complete

Mark 0.00 out of 2.00

exec() does this: curproc->tf->eip = elf.entry, but userinit() does this: p->tf->eip = 0; Select all the statements from below, that collectively explain this

- [x] a. the 'entry' in initcode is anyways 0
- [x] b. exec() loads from ELF file and the address of first instruction to be executed is given by 'entry'
- [ ] c. the initcode is created using objcopy, which discards all relocation information and symbols (like entry)
- [x] d. elf.entry is anyways 0, so both statements mean the same
- [x] e. In userinit() the function inituvm() has mapped the code of 'initcode' to be starting at virtual address 0
- [ ] f. the code of 'initcode' is loaded at physical address 0

The correct answers are: exec() loads from ELF file and the address of first instruction to be executed is given by 'entry', In userinit() the function inituvm() has mapped the code of 'initcode' to be starting at virtual address 0, the initcode is created using objcopy, which discards all relocation information and symbols (like entry)

Question **4**

Complete

Mark 1.00 out of 1.00

What does userinit() do ?

- a. initializes the users
- b. sets up the 'initcode' process to start execution in forkret ()
- c. sets up the 'initcode' process to start execution in forkret()
- d. sets up the 'initcode' process to start execution in trapret()
- e. sets up the 'init' process to start execution in forkret()
- f. initializes the process 'init' and starts executing it

The correct answer is: sets up the 'initcode' process to start execution in forkret()

Question **5**

Complete

Mark 0.67 out of 1.00

Which of the following is done by mappages()?

- [ ] a. allocate page directory if required
- [ ] b. allocate page frame if required
- [x] c. allocate page table if required
- [x] d. create page table mappings for the range given by "va" and "va + size"
- [ ] e. create page table mappings to the range given by "pa" and "pa + size"

The correct answers are: create page table mappings for the range given by "va" and "va + size", allocate page table if required, create page table mappings to the range given by "pa" and "pa + size"

Question **6**

Complete

Mark 0.00 out of 1.00

Select the statement that most correctly describes what setupkvm() does

- a. creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global arrray
- b. creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global arrray and makes kpgdir point to it
- c. creates a 1-level page table for the use by the kernel, as specified in kmap[] global array
- d. creates a 2-level page table for the use of the kernel, as specified in gdtdesc

The correct answer is: creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global arrray

Question **7**

Complete

Mark 0.00 out of 1.00

The approximate number of page frames created by kinit1 is

- a. 4
- b. 16
- c. 4000
- d. 3000
- e. 2000
- f. 1000
- g. 10

The correct answer is: 3000

Question **8**

Complete

Mark 1.20 out of 1.50

Which of the following is DONE by allocproc() ?

- [ ] a. setup kernel memory mappings for the process
- [x] b. Select an UNUSED struct proc for use
- [ ] c. ensure that the process starts in trapret()
- [x] d. allocate kernel stack for the process
- [ ] e. allocate PID to the process
- [x] f. ensure that the process starts in forkret()
- [x] g. setup the trapframe and context pointers appropriately
- [ ] h. setup the contents of the trapframe of the process properly

The correct answers are: Select an UNUSED struct proc for use, allocate PID to the process, allocate kernel stack for the process, setup the trapframe and context pointers appropriately, ensure that the process starts in forkret()

Question **9**

Complete

Mark 1.00 out of 1.00

Map the virtual address to physical address in xv6

KERNLINK	0x100000
0xFE000000	0xFE000000
80108000	0x108000
KERNBASE	0

The correct answer is: KERNLINK → 0x100000, 0xFE000000 → 0xFE000000, 80108000 → 0x108000, KERNBASE → 0

Question **10**

Complete

Mark 0.42 out of 1.00

Select all the correct statements about initcode

- [ ] a. code of initcode is loaded at virtual address 0
- [x] b. The data and stack of initcode is mapped to one single page in userinit()
- [ ] c. code of 'initcode' is loaded along with the kernel during booting
- [x] d. the size of 'initcode' is 2c
- [ ] e. code of initcode is loaded in memory by the kernel during userinit()
- [x] f. initcode is the 'init' process
- [x] g. initcode essentially calls exec("/init",...)

The correct answers are: code of 'initcode' is loaded along with the kernel during booting, the size of 'initcode' is 2c, The data and stack of initcode is mapped to one single page in userinit(), initcode essentially calls exec("/init",...)

Question **11**

Complete

Mark 1.00 out of 1.00

The variable 'end' used as argument to kinit1 has the value

- (●) a. 801154a8
- ( ) b. 81000000
- ( ) c. 80110000
- ( ) d. 80102da0
- ( ) e. 8010a48c
- ( ) f. 80000000

The correct answer is: 801154a8

Question **12**

Complete

Mark 1.00 out of 1.00

What does seginit() do?

- ( ) a. Nothing significant, just repetition of earlier GDT setup but with 2-level paging setup done
- ( ) b. Nothing significant, just repetition of earlier GDT setup but with free frames list created now
- ( ) c. Adds two additional entries to GDT corresponding to Code and Data segments, but to be used in privilege level 0
- (●) d. Adds two additional entries to GDT corresponding to Code and Data segments, but to be used in privilege level 3
- ( ) e. Nothing significant, just repetition of earlier GDT setup but with kernel page table allocated now

The correct answer is: Adds two additional entries to GDT corresponding to Code and Data segments, but to be used in privilege level 3

Question **13**

Complete

Mark 1.00 out of 1.00

Does exec() code around clearptau() lead to wastage of one page frame?

- a. no
- b. yes

The correct answer is: yes

◄ Questions for test on kalloc/kfree/kvmalloc, etc.

Jump to...

(Optional Assignment) Slab allocator in xv6 ►

Started on	Monday, 16 January 2023, 9:00 PM
State	Finished
Completed on	Monday, 16 January 2023, 10:07 PM
Time taken	1 hour 6 mins
Grade	**13.25** out of 15.00 (**88.33**%)

Question **1**

Correct

Mark 1.00 out of 1.00

When you turn your computer ON, you are often shown an option like "Press F9 for boot options". What does this mean?

- a. The choice of booting slowly or fast
- b. The choice of which OS to boot from
- c. The BIOS allows us to choose the boot device, the device from which the boot loader will be loaded ✔
- d. The choice of the boot loader (e.g. GRUB or Windows-Loader)

The correct answer is: The BIOS allows us to choose the boot device, the device from which the boot loader will be loaded

Question **2**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about bootloader.

Every wrong selection will deduct marks proportional to 1/n where n is total wrong choices in the question.

You will get minimum a zero.

- [x] a. Modern Bootloaders often allow configuring the way an OS boots ✔
- [ ] b. Bootloader must be one sector in length
- [ ] c. The bootloader loads the BIOS
- [x] d. Bootloaders allow selection of OS to boot from ✔
- [x] e. LILO is a bootloader ✔

Your answer is correct.

The correct answers are: LILO is a bootloader, Modern Bootloaders often allow configuring the way an OS boots, Bootloaders allow selection of OS to boot from

Question **3**

Correct

Mark 2.00 out of 2.00

What will this program do?

```
int main() {
fork();
execl("/bin/ls", "/bin/ls", NULL);
printf("hello");
}
```

- a. one process will run ls, another will print hello
- b. run ls once
- c. run ls twice and print hello twice
- d. run ls twice and print hello twice, but output will appear in some random order
- e. run ls twice ✔

Your answer is correct.

The correct answer is: run ls twice

Question **4**

Correct

Mark 1.00 out of 1.00

Order the following events in boot process (from 1 onwards)

Login interface	5	✔
Init	4	✔
OS	3	✔
BIOS	1	✔
Shell	6	✔
Boot loader	2	✔

Your answer is correct.

The correct answer is: Login interface → 5, Init → 4, OS → 3, BIOS → 1, Shell → 6, Boot loader → 2

Question **5**

Partially correct

Mark 0.50 out of 1.00

Select the correct statements about hard and soft links

Select one or more:

- [ ] a. Soft link shares the inode of actual file
- [x] b. Deleting a hard link deletes the file, only if link count was 1 ✔
- [ ] c. Soft links increase the link count of the actual file inode
- [x] d. Soft links can span across partitions while hard links can't ✔
- [ ] e. Deleting a soft link deletes only the actual file
- [x] f. Deleting a hard link always deletes the file ✖
- [ ] g. Hard links can span across partitions while soft links can't
- [x] h. Deleting a soft link deletes the link, not the actual file ✔
- [ ] i. Hard links increase the link count of the actual file inode
- [ ] j. Deleting a soft link deletes both the link and the actual file
- [x] k. Hard links share the inode ✔
- [ ] l. Hard links enforce separation of filename from it's metadata in on-disk data structures.

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: Soft links can span across partitions while hard links can't, Hard links increase the link count of the actual file inode, Deleting a soft link deletes the link, not the actual file, Deleting a hard link deletes the file, only if link count was 1, Hard links share the inode, Hard links enforce separation of filename from it's metadata in on-disk data structures.

Question **6**

Partially correct

Mark 1.75 out of 3.00

Select correct statements about mounting

Select one or more:

- [ ] a. Mounting makes all disk partitions available as one name space
- [ ] b. Mounting deletes all data at the mount-point
- [x] c. Even in operating systems with a pluggable kernel module for file systems, the code for mounting any particular file system must be already present in the operating system system kernel ✖
- [x] d. It's possible to mount a partition on one computer, into namespace of another computer. ✔
- [x] e. Mounting is attaching a disk-partition with a filesystem on it, into another file system name-space ✔
- [ ] f. On Linuxes mounting can be done only while booting the OS
- [x] g. The existing name-space at the mount-point is no longer visible after mounting ✔
- [ ] h. The mount point can be a file as well
- [x] i. The mount point must be a directory ✔
- [x] j. In operating systems with a pluggable kernel module for file systems, the code for mounting a particular file system is provided by the module of that file system. ✔

Your answer is partially correct.

You have correctly selected 5.

The correct answers are: Mounting is attaching a disk-partition with a filesystem on it, into another file system name-space, The mount point must be a directory, The existing name-space at the mount-point is no longer visible after mounting, Mounting makes all disk partitions available as one name space, In operating systems with a pluggable kernel module for file systems, the code for mounting a particular file system is provided by the module of that file system., It's possible to mount a partition on one computer, into namespace of another computer.

Question **7**

Correct

Mark 1.00 out of 1.00

Compare multiprogramming with multitasking

- ( ) a. A multitasking system is not necessarily multiprogramming
- (•) b. A multiprogramming system is not necessarily multitasking ✔

The correct answer is: A multiprogramming system is not necessarily multitasking

Question **8**

Correct

Mark 1.00 out of 1.00

Is the terminal a part of the kernel on GNU/Linux systems?

- a. yes
- b. no ✔ wrong

The correct answer is: no

Question **9**

Correct

Mark 1.00 out of 1.00

Consider the following programs

exec1.c

```
#include <unistd.h>
#include <stdio.h>
int main() {
   execl("./exec2", "./exec2", NULL);
}
```

exec2.c

```
#include <unistd.h>
#include <stdio.h>
int main() {
   execl("/bin/ls", "/bin/ls", NULL);
  printf("hello\n");
}
```

Compiled as

```
cc    exec1.c   -o exec1
cc    exec2.c   -o exec2
```

And run as

```
$ ./exec1
```

Explain the output of the above command (./exec1)

Assume that /bin/ls , i.e. the 'ls' program exists.

Select one:

- a. Execution fails as the call to execl() in exec2 fails
- b. Execution fails as the call to execl() in exec1 fails
- c. "ls" runs on current directory ✔
- d. Execution fails as one exec can't invoke another exec
- e. Program prints hello

Your answer is correct.

The correct answer is: "ls" runs on current directory

Question **10**

Correct

Mark 1.00 out of 1.00

A process blocks itself means

- a. The kernel code of an interrupt handler, moves the process to a waiting queue and calls scheduler
- b. The kernel code of system call calls scheduler
- c. The application code calls the scheduler
- d. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler ✔

The correct answer is: The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

Question **11**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about two modes of CPU operation

Select one or more:

- [x] a. Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode ✔
- [x] b. The two modes are essential for a multitasking system ✔
- [x] c. There is an instruction like 'iret' to return from kernel mode to user mode ✔
- [x] d. The two modes are essential for a multiprogramming system ✔
- [x] e. The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously ✔

Your answer is correct.

The correct answers are: The two modes are essential for a multiprogramming system, The two modes are essential for a multitasking system, There is an instruction like 'iret' to return from kernel mode to user mode, The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously, Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode

Question **12**

Correct

Mark 1.00 out of 1.00

which of the following is not a difference between real mode and protected mode

- a. in real mode the addressable memory is less than in protected mode
- b. in real mode the addressable memory is more than in protected mode ✔
- c. processor starts in real mode
- d. in real mode general purpose registers are 16 bit, in protected mode they are 32 bit
- e. in real mode the segment is multiplied by 16, in protected mode segment is used as index in GDT

The correct answer is: in real mode the addressable memory is more than in protected mode

◄ Random Quiz - 1 (Pre-Requisite Quiz)

Jump to...

Random Quiz - 3 (processes, memory management, event driven kernel), compilation-linking-loading, ipc-pipes ►

Started on	Thursday, 2 February 2023, 9:01 PM
State	Finished
Completed on	Thursday, 2 February 2023, 10:55 PM
Time taken	1 hour 53 mins
Grade	**16.46** out of 20.00 (**82.28**%)

Question **1**

Complete

Mark 1.00 out of 1.00

Which of the following are NOT a part of job of a typical compiler?

- [x] a. Check the program for logical errors
- [ ] b. Convert high level langauge code to machine code
- [ ] c. Invoke the linker to link the function calls with their code, extern globals with their declaration
- [x] d. Suggest alternative pieces of code that can be written
- [ ] e. Check the program for syntactical errors
- [ ] f. Process the # directives in a C program

The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

Question **2**

Complete

Mark 1.00 out of 1.00

Consider the following code and MAP the file to which each fd points at the end of the code.

```
int main(int argc, char *argv[]) {
   int fd1, fd2 = 1, fd3 = 1, fd4 = 1;

   fd1 = open("/tmp/1", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
   fd2 = open("/tmp/2", O_RDDONLY);
   fd3 = open("/tmp/3", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
   close(0);
   close(1);
   dup(fd2);
   dup(fd3);
   close(fd3);
   dup2(fd2, fd4);
   printf("%d %d %d %d\n", fd1, fd2, fd3, fd4);
   return 0;
}
```

fd4	/tmp/2
fd2	/tmp/2
1	/tmp/3
fd1	/tmp/1
fd3	closed
0	/tmp/2
2	stderr

The correct answer is: fd4 → /tmp/2, fd2 → /tmp/2, 1 → /tmp/3, fd1 → /tmp/1, fd3 → closed, 0 → /tmp/2, 2 → stderr

Question **3**

Complete

Mark 0.75 out of 1.00

Select the sequence of events that are NOT possible, assuming an interruptible kernel code

Select one or more:

- [ ] a. P1 running
  keyboard hardware interrupt
  keyboard interrupt handler running
  interrupt handler returns
  P1 running
  P1 makes sytem call
  system call returns
  P1 running
  timer interrupt
  scheduler
  P2 running
- [ ] b. P1 running
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P3 running
  Hardware interrupt
  Interrupt unblocks P1
  Interrupt returns
  P3 running
  Timer interrupt
  Scheduler
  P1 running
- [ ] c. P1 running
  P1 makes system call
  system call returns
  P1 running
  timer interrupt
  Scheduler running
  P2 running
- [x] d. P1 running
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again
- [x] e.
  P1 running
  P1 makes sytem call
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again

f. P1 running
P1 makes system call
timer interrupt
Scheduler
P2 running
timer interrupt
Scheuler
P1 running
P1's system call return

The correct answers are: P1 running
P1 makes sytem call and blocks
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again,
P1 running
P1 makes sytem call
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

Question **4**

Complete

Mark 0.00 out of 1.00

Select all the correct statements about named pipes and ordinary(unnamed) pipe

Select one or more:

- [x] a. named pipes can be used between multiple processes but ordinary pipes can not be used
- [x] b. named pipe can be used between any processes
- [ ] c. both named and unnamed pipes require some kind of agreed protocol to be effectively used among multiple processes
- [x] d. named pipes are more efficient than ordinary pipes
- [x] e. ordinary pipe can only be used between related processes
- [x] f. a named pipe exists as a file on the file system
- [ ] g. named pipe exists even if the processes using it do exit()

The correct answers are: ordinary pipe can only be used between related processes, named pipe can be used between any processes, a named pipe exists as a file on the file system, named pipe exists even if the processes using it do exit(), both named and unnamed pipes require some kind of agreed protocol to be effectively used among multiple processes

Question **5**

Complete

Mark 4.50 out of 5.00

Following code claims to implement the command

/bin/ls -l | /usr/bin/head -3 | /usr/bin/tail -1

Fill in the blanks to make the code work.

Note: Do not include space in writing any option. x[1][2] should be written without any space, and so is the case with [1] or [2]. Pay attention to exact syntax and do not write any extra character like ';' or = etc.

```
int main(int argc, char *argv[]) {
   int pid1, pid2;
   int pfd[
2
][2];

   pipe(
pfd[0]
);
   pid1 =
fork()
;
   if(pid1 != 0) {
      close(pfd[0]
[0]
);
      close(
1
);
      dup(
pfd[0][1]
);
      execl("/bin/ls", "/bin/ls", "
-l
", NULL);
   }
   pipe(
pfd[1]
);

pid2
= fork();
   if(pid2 == 0) {
      close(
pfd[0][1]
;
```

```
        close(0);
        dup(
[ pfd[1][0] ]
);
        close(pfd[1]
[ [0] ]
);
        close(
[ 1 ]
);
        dup(
[ pfd[1][1] ]
);
        execl("/usr/bin/head", "/usr/bin/head", "
[ -3 ]
", NULL);
    } else {
        close(pfd
[ [1][1] ]
);
        close(
[ 0 ]
);
        dup(
[ pfd[1][0] ]
);
        close(pfd
[ [0][0] ]
);
        execl("/usr/bin/tail", "/usr/bin/tail", "
[ -1 ]
", NULL);
  }
}
```

Question **6**

Complete

Mark 1.00 out of 1.00

Select the compiler's view of the process's address space, for each of the following MMU schemes:
(Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

Paging	one continuous chunk
Segmentation, then paging	many continuous chunks of variable size
Segmentation	many continuous chunks of variable size
Relocation + Limit	one continuous chunk

The correct answer is: Paging → one continuous chunk, Segmentation, then paging → many continuous chunks of variable size, Segmentation → many continuous chunks of variable size, Relocation + Limit → one continuous chunk

Question **7**

Complete

Mark 1.40 out of 2.00

Match the elements of C program to their place in memory

Local Variables	Stack
Malloced Memory	Heap
Global Static variables	Data
Global variables	Stack
Code of main()	Code
Local Static variables	Data
Function code	Data
Arguments	Stack
#define MACROS	No memory needed
#include files	No memory needed

The correct answer is: Local Variables → Stack, Malloced Memory → Heap, Global Static variables → Data, Global variables → Data, Code of main() → Code, Local Static variables → Data, Function code → Code, Arguments → Stack, #define MACROS → No Memory needed, #include files → No memory needed

Question **8**

Complete

Mark 0.63 out of 1.00

Consider the image given below, which explains how paging works.

Figure 9.8 Paging hardware.

Mention whether each statement is True or False, with respect to this image.

True	False	
◉	○	The physical address may not be of the same size (in bits) as the logical address
◉	○	The locating of the page table using PTBR also involves paging translation
◉	○	Size of page table is always determined by the size of RAM
○	◉	The page table is indexed using frame number
◉	○	The PTBR is present in the CPU as a register
◉	○	The page table is indexed using page number
○	◉	Maximum Size of page table is determined by number of bits used for page number
◉	○	The page table is itself present in Physical memory

The physical address may not be of the same size (in bits) as the logical address: True
The locating of the page table using PTBR also involves paging translation: False
Size of page table is always determined by the size of RAM: False
The page table is indexed using frame number: False
The PTBR is present in the CPU as a register: True
The page table is indexed using page number: True
Maximum Size of page table is determined by number of bits used for page number: True
The page table is itself present in Physical memory: True

Question **9**

Complete

Mark 0.67 out of 1.00

Select the sequence of events that are NOT possible, assuming a non-interruptible kernel code

(Note: non-interruptible kernel code means, if the kernel code is executing, then interrupts will be disabled).

Note: A possible sequence may have some missing steps in between. An impossible sequence will will have n and n+1th steps such that n+1th step can not follow n'th step.

Select one or more:

- [ ] a. P1 running
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P3 running
  Hardware interrupt
  Interrupt unblocks P1
  Interrupt returns
  P3 running
  Timer interrupt
  Scheduler
  P1 running
- [ ] b. P1 running
  P1 makes system call
  system call returns
  P1 running
  timer interrupt
  Scheduler running
  P2 running
- [x] c. P1 running
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again
- [ ] d. P1 running
  keyboard hardware interrupt
  keyboard interrupt handler running
  interrupt handler returns
  P1 running
  P1 makes sytem call
  system call returns
  P1 running
  timer interrupt
  scheduler
  P2 running
- [x] e. P1 running
  P1 makes system call
  timer interrupt
  Scheduler
  P2 running

timer interrupt
Scheuler
P1 running
P1's system call return

f.

P1 running
P1 makes sytem call
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

The correct answers are: P1 running
P1 makes sytem call and blocks
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again, P1 running
P1 makes system call
timer interrupt
Scheduler
P2 running
timer interrupt
Scheuler
P1 running
P1's system call return,
P1 running
P1 makes sytem call
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

Question **10**

Complete

Mark 1.00 out of 1.00

Consider the two programs given below to implement the command (ignore the fact that error checks are not done on return values of functions)

$ ls . /tmp/asdfksdf >/tmp/ddd 2>&1

Program 1

```
int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(1);
    dup(fd);
    close(2);
    dup(fd);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Program 2

```
int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    close(1);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(2);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Select all the correct statements about the programs

Select one or more:

- [ ] a. Both programs are correct
- [ ] b. Program 2 is correct for > /tmp/ddd but not for 2>&1
- [ ] c. Program 1 is correct for > /tmp/ddd but not for 2>&1
- [ ] d. Program 2 ensures 2>&1 and does not ensure > /tmp/ddd
- [ ] e. Program 2 makes sure that there is one file offset used for '2' and '1'
- [ ] f. Program 1 ensures 2>&1 and does not ensure > /tmp/ddd
- [ ] g. Only Program 2 is correct
- [ ] h. Program 1 does 1>&2
- [x] i. Program 1 makes sure that there is one file offset used for '2' and '1'
- [ ] j. Program 2 does 1>&2
- [ ] k. Both program 1 and 2 are incorrect
- [x] l. Only Program 1 is correct

The correct answers are: Only Program 1 is correct, Program 1 makes sure that there is one file offset used for '2' and '1'

Question **11**

Complete

Mark 0.71 out of 1.00

Order the events that occur on a timer interrupt:

Jump to a code pointed by IDT	2
Save the context of the currently running process	3
Execute the code of the new process	6
Jump to scheduler code	4
Change to kernel stack of currently running process	1
Select another process for execution	5
Set the context of the new process	7

The correct answer is: Jump to a code pointed by IDT → 2, Save the context of the currently running process → 3, Execute the code of the new process → 7, Jump to scheduler code → 4, Change to kernel stack of currently running process → 1, Select another process for execution → 5, Set the context of the new process → 6

Question **12**

Complete

Mark 0.80 out of 1.00

Select all the correct statements about zombie processes

Select one or more:

- [ ] a. A process becomes zombie when it's parent finishes
- [x] b. A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it
- [x] c. If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent
- [ ] d. Zombie processes are harmless even if OS is up for long time
- [x] e. A process can become zombie if it finishes, but the parent has finished before it
- [ ] f. A zombie process occupies space in OS data structures
- [x] g. init() typically keeps calling wait() for zombie processes to get cleaned up
- [ ] h. A zombie process remains zombie forever, as there is no way to clean it up

The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it, A process can become zombie if it finishes, but the parent has finished before it, A zombie process occupies space in OS data structures, If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

Question **13**

Complete

Mark 1.00 out of 1.00

Select the state that is not possible after the given state, for a process:

New: Running

Ready : Waiting

Running: : None of these

Waiting: Running

Question **14**

Complete

Mark 1.00 out of 1.00

Select the order in which the various stages of a compiler execute.

Pre-processing	1
Intermediate code generation	3
Linking	4
Syntatical Analysis	2
Loading	does not exist

The correct answer is: Pre-processing → 1, Intermediate code generation → 3, Linking → 4, Syntatical Analysis → 2, Loading → does not exist

Question **15**

Complete

Mark 1.00 out of 1.00

A process blocks itself means

a. The application code calls the scheduler
b. The kernel code of system call calls scheduler
c. The kernel code of an interrupt handler, moves the process to a waiting queue and calls scheduler
d. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

The correct answer is: The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

◄ Random Quiz - 2: bootloader, system calls, fork-exec, open-read-write, linux-basics, processes

Jump to...

Random Quiz 4 : Scheduling, signals, segmentation, paging, compilation, process-state ►

Started on	Thursday, 16 February 2023, 9:00 PM
State	Finished
Completed on	Thursday, 16 February 2023, 10:43 PM
Time taken	1 hour 43 mins
Grade	**13.00** out of 15.00 (**86.68**%)

Question **1**

Correct

Mark 1.00 out of 1.00

Which of the following statements is false ?

Select one:

- a. A process scheduling algorithm is preemptive if the CPU can be forcibly removed from a process.
- b. Time sharing systems generally use preemptive CPU scheduling.
- c. Real time systems generally use non preemptive CPU scheduling. ✔
- d. Response time is more predictable in preemptive systems than in non preemptive systems.

Your answer is correct.

The correct answer is: Real time systems generally use non preemptive CPU scheduling.

Question **2**

Correct

Mark 1.00 out of 1.00

Order the sequence of events, in scheduling process P1 after process P0

timer interrupt occurs	2 ✔
Process P1 is running	6 ✔
context of P0 is saved in P0's PCB	3 ✔
Control is passed to P1	5 ✔
context of P1 is loaded from P1's PCB	4 ✔
Process P0 is running	1 ✔

Your answer is correct.

The correct answer is: timer interrupt occurs → 2, Process P1 is running → 6, context of P0 is saved in P0's PCB → 3, Control is passed to P1 → 5, context of P1 is loaded from P1's PCB → 4, Process P0 is running → 1

Question **3**

Correct

Mark 1.00 out of 1.00

Select the compiler's view of the process's address space, for each of the following MMU schemes:
(Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

Paging	one continuous chunk ✔
Relocation + Limit	one continuous chunk ✔
Segmentation	many continuous chunks of variable size ✔
Segmentation, then paging	many continuous chunks of variable size ✔

Your answer is correct.

The correct answer is: Paging → one continuous chunk, Relocation + Limit → one continuous chunk, Segmentation → many continuous chunks of variable size, Segmentation, then paging → many continuous chunks of variable size

Question **4**

Correct

Mark 1.00 out of 1.00

Match the names of PCB structures with kernel

linux struct task_struct ✔

xv6 struct proc ✔

The correct answer is: linux → struct task_struct, xv6 → struct proc

Question **5**

Correct

Mark 1.00 out of 1.00

Select the state that is not possible after the given state, for a process:

New: Running ✔

Ready : Waiting ✔

Running: : None of these ✔

Waiting: Running ✔

Question **6**

Partially correct

Mark 0.47 out of 1.00

Select all the correct statements about zombie processes

Select one or more:

- [ ] a. A zombie process occupies space in OS data structures
- [x] b. init() typically keeps calling wait() for zombie processes to get cleaned up ✔
- [x] c. If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent ✔
- [x] d. A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it ✔
- [x] e. Zombie processes are harmless even if OS is up for long time ✖
- [x] f. A process can become zombie if it finishes, but the parent has finished before it ✔
- [ ] g. A process becomes zombie when it's parent finishes
- [ ] h. A zombie process remains zombie forever, as there is no way to clean it up

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it, A process can become zombie if it finishes, but the parent has finished before it, A zombie process occupies space in OS data structures, If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

Question **7**

Partially correct

Mark 0.86 out of 1.00

Mark True/False

Statements about scheduling and scheduling algorithms

True	False			
◉☑	○✖	xv6 code does not care about Processor Affinity	✔	
○☑	◉✖	Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.	✖	
◉☑	○✖	Response time will be quite poor on non-interruptible kernels	✔	
◉☑	○✖	Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.	✔	
◉☑	○✖	Processor Affinity refers to memory accesses of a process being stored on cache of that processor	✔	
◉☑	○✖	A scheduling algorithm is non-premptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.	✔	
○✖	◉☑	On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread	✔	It's the negation of this. Time NOT spent in idle thread.

xv6 code does not care about Processor Affinity: True
Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.: True
Response time will be quite poor on non-interruptible kernels: True
Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.: True
Processor Affinity refers to memory accesses of a process being stored on cache of that processor: True
A scheduling algorithm is non-premptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.: True
On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread: False

Question **8**

Correct

Mark 1.00 out of 1.00

Mark whether the concept is related to scheduling or not.

Yes	No		
✗	✓	file-table	✓
✓	✗	context-switch	✓
✓	✗	timer interrupt	✓
✓	✗	ready-queue	✓
✓	✗	runnable process	✓

file-table: No
context-switch: Yes
timer interrupt: Yes
ready-queue: Yes
runnable process: Yes

Question **9**

Partially correct

Mark 0.50 out of 1.00

Select all the correct statements about signals

Select one or more:

- [ ] a. Signals are delivered to a process by another process
- [ ] b. The signal handler code runs in user mode of CPU
- [x] c. A signal handler can be invoked asynchronously or synchronously depending on signal type ✔
- [x] d. SIGKILL definitely kills a process because it can't be caught or ignored, and it's default action terminates the process ✔
- [ ] e. Signal handlers once replaced can't be restored
- [x] f. The signal handler code runs in kernel mode of CPU ✖
- [x] g. Signals are delivered to a process by kernel ✔
- [ ] h. SIGKILL definitely kills a process because it's code runs in kernel mode of CPU

Your answer is partially correct.

You have correctly selected 3.
The correct answers are: Signals are delivered to a process by kernel, A signal handler can be invoked asynchronously or synchronously depending on signal type, The signal handler code runs in user mode of CPU, SIGKILL definitely kills a process because it can't be caught or ignored, and it's default action terminates the process

Question **10**

Correct

Mark 1.00 out of 1.00

Map each signal with it's meaning

SIGPIPE	Broken Pipe	✔
SIGCHLD	Child Stopped or Terminated	✔
SIGUSR1	User Defined Signal	✔
SIGALRM	Timer Signal from alarm()	✔
SIGSEGV	Invalid Memory Reference	✔

The correct answer is: SIGPIPE → Broken Pipe, SIGCHLD → Child Stopped or Terminated, SIGUSR1 → User Defined Signal, SIGALRM → Timer Signal from alarm(), SIGSEGV → Invalid Memory Reference

Question **11**

Partially correct

Mark 0.80 out of 1.00

Mark statements True/False w.r.t. change of states of a process. Note that a statement is true only if the claim and argument both are true.

Reference: The process state diagram (and your understanding of how kernel code works). Note - the diagram does not show zombie state!

Figure 3.2 Diagram of process state.

True	False		
◉☑	○✖	Every forked process has to go through ZOMBIE state, at least for a small duration.	✔
◉☑	○✖	Only a process in READY state is considered by scheduler	✔
◉☑	○✖	A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet	✔
◉✖	○☑	A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first	✖
○✖	◉☑	A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.	✔

Every forked process has to go through ZOMBIE state, at least for a small duration.: True
Only a process in READY state is considered by scheduler: True
A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet: True
A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first: False
A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.: False

Question **12**

Correct

Mark 1.00 out of 1.00

Which of the following parts of a C program do not have any corresponding machine code ?

- [ ] a. pointer dereference
- [ ] b. function calls
- [x] c. global variables ✔
- [ ] d. local variable declaration
- [x] e. #directives ✔
- [x] f. typedefs ✔
- [ ] g. expressions

Your answer is correct.

The correct answers are: #directives, typedefs, global variables

Question **13**

Partially correct

Mark 1.38 out of 2.00

Select all the correct statements about the state of a process.

- [x] a. A process in ready state is ready to be scheduled ✔
- [ ] b. A waiting process starts running after the wait is over
- [ ] c. A process changes from running to ready state on a timer interrupt or any I/O wait
- [x] d. A process waiting for I/O completion is typically woken up by the particular interrupt handler code ✔
- [x] e. Processes in the ready queue are in the ready state ✔
- [ ] f. A process changes from running to ready state on a timer interrupt
- [ ] g. It is not maintained in the data structures by kernel, it is only for conceptual understanding of programmers
- [x] h. A process waiting for any condition is woken up by another process only ✖
- [x] i. A running process may terminate, or go to wait or become ready again ✔
- [x] j. A process can self-terminate only when it's running ✔
- [x] k. Typically, it's represented as a number in the PCB ✔
- [x] l. A process that is running is not on the ready queue ✔
- [x] m. Changing from running state to waiting state results in "giving up the CPU" ✔
- [ ] n. A process in ready state is ready to receive interrupts

Your answer is partially correct.

You have correctly selected 8.

The correct answers are: Typically, it's represented as a number in the PCB, A process in ready state is ready to be scheduled, Processes in the ready queue are in the ready state, A process that is running is not on the ready queue, A running process may terminate, or go to wait or become ready again, A process changes from running to ready state on a timer interrupt, Changing from running state to waiting state results in "giving up the CPU", A process can self-terminate only when it's running, A process waiting for I/O completion is typically woken up by the particular interrupt handler code

Question **14**

Correct

Mark 1.00 out of 1.00

Which of the following are NOT a part of job of a typical compiler?

- [ ] a. Invoke the linker to link the function calls with their code, extern globals with their declaration
- [x] b. Check the program for logical errors ✔
- [ ] c. Convert high level langauge code to machine code
- [ ] d. Check the program for syntactical errors
- [x] e. Suggest alternative pieces of code that can be written ✔
- [ ] f. Process the # directives in a C program

Your answer is correct.

The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

◄ Random Quiz - 3 (processes, memory management, event driven kernel), compilation-linking-loading, ipc-pipes

Jump to...

Homework questions: Basics of MM, xv6 booting ►

Started on	Thursday, 18 March 2021, 2:46 PM
State	Finished
Completed on	Thursday, 18 March 2021, 3:50 PM
Time taken	1 hour 4 mins
Grade	**10.36** out of 20.00 (**52**%)

Question **1**

Partially correct

Mark 0.57 out of 1.00

Mark True, the actions done as part of code of swtch() in swtch.S, in xv6

True	False		
		Restore new callee saved registers from kernel stack of new context	✔
		Save old callee saved registers on kernel stack of old context	✔
		Save old callee saved registers on user stack of old context	✔
		Switch from old process context to new process context	✖
		Switch from one stack (old) to another(new)	✖
		Restore new callee saved registers from user stack of new context	✔
		Jump to code in new context	✖

Restore new callee saved registers from kernel stack of new context: True
Save old callee saved registers on kernel stack of old context: True
Save old callee saved registers on user stack of old context: False
Switch from old process context to new process context: False
Switch from one stack (old) to another(new): True
Restore new callee saved registers from user stack of new context: False
Jump to code in new context: False

Question **2**

Partially correct

Mark 0.17 out of 0.50

For each function/code-point, select the status of segmentation setup in xv6

bootmain()	gdt setup with 3 entries, right from first line of code of bootloader	✖
kvmalloc() in main()	gdt setup with 5 entries (0 to 4) on one processor	✖
after startothers() in main()	gdt setup with 5 entries (0 to 4) on all processors	✔
after seginit() in main()	gdt setup with 5 entries (0 to 4) on all processors	✖
bootasm.S	gdt setup with 3 entries, right from first line of code of bootloader	✖
entry.S	gdt setup with 3 entries, at start32 symbol of bootasm.S	✔

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: bootmain() → gdt setup with 3 entries, at start32 symbol of bootasm.S, kvmalloc() in main() → gdt setup with 3 entries, at start32 symbol of bootasm.S, after startothers() in main() → gdt setup with 5 entries (0 to 4) on all processors, after seginit() in main() → gdt setup with 5 entries (0 to 4) on one processor, bootasm.S → gdt setup with 3 entries, at start32 symbol of bootasm.S, entry.S → gdt setup with 3 entries, at start32 symbol of bootasm.S

Question **3**

Partially correct

Mark 0.38 out of 1.00

Compare paging with demand paging and select the correct statements.

Select one or more:

- [x] a. The meaning of valid-invalid bit in page table is different in paging and demand-paging. ✔
- [x] b. Demand paging requires additional hardware support, compared to paging. ✔
- [ ] c. Paging requires some hardware support in CPU
- [x] d. With paging, it's possible to have user programs bigger than physical memory. ✖
- [x] e. Both demand paging and paging support shared memory pages. ✔
- [ ] f. Demand paging always increases effective memory access time.
- [x] g. With demand paging, it's possible to have user programs bigger than physical memory. ✔
- [x] h. Calculations of number of bits for page number and offset are same in paging and demand paging. ✔
- [ ] i. TLB hit ration has zero impact in effective memory access time in demand paging.
- [ ] j. Paging requires NO hardware support in CPU

Your answer is partially correct.

You have correctly selected 5.

The correct answers are: Demand paging requires additional hardware support, compared to paging., Both demand paging and paging support shared memory pages., With demand paging, it's possible to have user programs bigger than physical memory., Demand paging always increases effective memory access time., Paging requires some hardware support in CPU, Calculations of number of bits for page number and offset are same in paging and demand paging., The meaning of valid-invalid bit in page table is different in paging and demand-paging.

Question **4**

Partially correct

Mark 0.44 out of 0.50

Suppose a processor supports base(relocation register) + limit scheme of MMU.

Assuming this, mark the statements as True/False

True	False		
✔	✖	The OS may terminate the process while handling the interrupt of memory violation	✔
✔	✖	The hardware detects any memory access beyond the limit value and raises an interrupt	✔
✖	✔	The hardware may terminate the process while handling the interrupt of memory violation	✖
✔	✖	The OS sets up the relocation and limit registers when the process is scheduled	✔
✔	✖	The compiler generates machine code assuming continuous memory address space for process, and calculating appropriate sizes for code, and data;	✔
✖	✔	The process sets up it's own relocation and limit registers when the process is scheduled	✔
✖	✔	The OS detects any memory access beyond the limit value and raises an interrupt	✔
✖	✔	The compiler generates machine code assuming appropriately sized semgments for code, data and stack.	✔

The OS may terminate the process while handling the interrupt of memory violation: True
The hardware detects any memory access beyond the limit value and raises an interrupt: True
The hardware may terminate the process while handling the interrupt of memory violation: False
The OS sets up the relocation and limit registers when the process is scheduled: True
The compiler generates machine code assuming continuous memory address space for process, and calculating appropriate sizes for code, and data;: True
The process sets up it's own relocation and limit registers when the process is scheduled: False
The OS detects any memory access beyond the limit value and raises an interrupt: False
The compiler generates machine code assuming appropriately sized semgments for code, data and stack.: False

Question **5**

Correct

Mark 0.50 out of 0.50

Consider the following list of free chunks, in continuous memory management:

10k, 25k, 12k, 7k, 9k, 13k

Suppose there is a request for chunk of size 9k, then the free chunk selected under each of the following schemes will be

Best fit:

9k

✔

First fit:

10k

✔

Worst fit:

25k

✔

Question **6**

Partially correct

Mark 0.50 out of 1.00

Select all the correct statements about MMU and it's functionality

Select one or more:

- [ ] a. MMU is a separate chip outside the processor
- [x] b. MMU is inside the processor ✔
- [ ] c. Logical to physical address translations in MMU are done with specific machine instructions
- [x] d. The operating system interacts with MMU for every single address translation ✖
- [x] e. Illegal memory access is detected in hardware by MMU and a trap is raised ✔
- [ ] f. The Operating system sets up relevant CPU registers to enable proper MMU translations
- [x] g. Logical to physical address translations in MMU are done in hardware, automatically ✔
- [ ] h. Illegal memory access is detected by operating system

Your answer is partially correct.

You have correctly selected 3.
The correct answers are: MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

Question **7**

Incorrect

Mark 0.00 out of 0.50

Assuming a 8- KB page size, what is the page numbers for the address 874815 reference in decimal :

(give answer also in decimal)

Answer: 2186 ✖

The correct answer is: 107

Question **8**

Incorrect

Mark 0.00 out of 0.25

Select the compiler's view of the process's address space, for each of the following MMU schemes:
(Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

Segmentation, then paging	Many continuous chunks each of page size	✖
Relocation + Limit	Many continuous chunks of same size	✖
Segmentation	one continuous chunk	✖
Paging	many continuous chunks of variable size	✖

Your answer is incorrect.

The correct answer is: Segmentation, then paging → many continuous chunks of variable size, Relocation + Limit → one continuous chunk, Segmentation → many continuous chunks of variable size, Paging → one continuous chunk

Question **9**

Incorrect

Mark 0.00 out of 0.50

Suppose the memory access time is 180ns and TLB hit ratio is 0.3, then effective memory access time is (in nanoseconds);

Answer: 192 ✖

The correct answer is: 306.00

Question **10**

Correct

Mark 0.50 out of 0.50

In xv6, The struct context is given as

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

Select all the reasons that explain why only these 5 registers are included in the struct context.

- [x] a. The segment registers are same across all contexts, hence they need not be saved ✔
- [x] b. esp is not saved in context, because context{} is on stack and it's address is always argument to swtch() ✔
- [ ] c. xv6 tries to minimize the size of context to save memory space
- [ ] d. esp is not saved in context, because it's not part of the context
- [x] e. eax, ecx, edx are caller save, hence no need to save ✔

Your answer is correct.

The correct answers are: The segment registers are same across all contexts, hence they need not be saved, eax, ecx, edx are caller save, hence no need to save, esp is not saved in context, because context{} is on stack and it's address is always argument to swtch()

Question **11**

Partially correct

Mark 0.83 out of 1.50

Arrange the following events in order, in page fault handling:

Disk interrupt wakes up the process	7	✔
The reference bit is found to be invalid by MMU	1	✔
OS makes available an empty frame	6	✖
Restart the instruction that caused the page fault	9	✔
A hardware interrupt is issued	3	✖
OS schedules a disk read for the page (from backing store)	5	✔
Process is kept in wait state	4	✖
Page tables are updated for the process	8	✔
Operating system decides that the page was not in memory	2	✖

Your answer is partially correct.

You have correctly selected 5.

The correct answer is: Disk interrupt wakes up the process → 7, The reference bit is found to be invalid by MMU → 1, OS makes available an empty frame → 4, Restart the instruction that caused the page fault → 9, A hardware interrupt is issued → 2, OS schedules a disk read for the page (from backing store) → 5, Process is kept in wait state → 6, Page tables are updated for the process → 8, Operating system decides that the page was not in memory → 3

Question **12**

Incorrect

Mark 0.00 out of 0.50

Suppose a kernel uses a buddy allocator. The smallest chunk that can be allocated is of size 32 bytes. One bit is used to track each such chunk, where 1 means allocated and 0 means free. The chunk looks like this as of now:

00001010

Now, there is a request for a chunk of 70 bytes.

After this allocation, the bitmap, indicating the status of the buddy allocator will be

Answer: 11101010 ✖

The correct answer is: 11111010

Question **13**

Incorrect

Mark 0.00 out of 0.25

The complete range of virtual addresses (after main() in main.c is over), from which the free pages used by kalloc() and kfree() is derived,are:

- a. end, 4MB
- b. P2V(end), P2V(PHYSTOP)
- c. end, P2V(4MB + PHYSTOP)
- d. P2V(end), PHYSTOP ✖
- e. end, (4MB + PHYSTOP)
- f. end, PHYSTOP
- g. end, P2V(PHYSTOP)

Your answer is incorrect.

The correct answer is: end, P2V(PHYSTOP)

Question **14**

Partially correct

Mark 0.33 out of 0.50

Match the pair

Hashed page table	Linear search on collsion done by OS (e.g. SPARC Solaris) typically	✔
Inverted Page table	Linear/Parallel search using frame number in page table	✖
Hierarchical Paging	More memory access time per hierarchy	✔

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: Hashed page table → Linear search on collsion done by OS (e.g. SPARC Solaris) typically, Inverted Page table → Linear/Parallel search using page number in page table, Hierarchical Paging → More memory access time per hierarchy

Question **15**

Partially correct

Mark 0.29 out of 0.50

After virtual memory is implemented

(select T/F for each of the following)One Program's size can be larger than physical memory size

True	False		
◉☑	○✖	Code need not be completely in memory	✔
◉☑	○✖	Cumulative size of all programs can be larger than physical memory size	✔
◉✖	○☑	Virtual access to memory is granted	✖
◉☑	○✖	Logical address space could be larger than physical address space	✔
◉✖	○☑	Virtual addresses are available	✖
○☑	◉✖	Relatively less I/O may be possible during process execution	✖
◉☑	○✖	One Program's size can be larger than physical memory size	✔

Code need not be completely in memory: True
Cumulative size of all programs can be larger than physical memory size: True
Virtual access to memory is granted: False
Logical address space could be larger than physical address space: True
Virtual addresses are available: False
Relatively less I/O may be possible during process execution: True
One Program's size can be larger than physical memory size: True

Question **16**

Partially correct

Mark 0.64 out of 1.00

W.r.t. Memory management in xv6,

xv6 uses physical memory upto 224 MB onlyMark statements True or False

True	False		
		The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context	✔
		The stack allocated in entry.S is used as stack for scheduler's context for first processor	✔
		The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir	✔
		The free page-frame are created out of nearly 222 MB	✖
		The kernel code and data take up less than 2 MB space	✔
		The switchkvm() call in scheduler() changes CR3 to use page directory of new process	✖
		The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context	✔
		PHYSTOP can be increased to some extent, simply by editing memlayout.h	✔
		xv6 uses physical memory upto 224 MB only	✖
		The process's address space gets mapped on frames, obtained from ~2MB:224MB range	✔
		The kernel's page table given by kpgdir variable is used as stack for scheduler's context	✖

The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context: True
The stack allocated in entry.S is used as stack for scheduler's context for first processor: True
The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir: True
The free page-frame are created out of nearly 222 MB: True
The kernel code and data take up less than 2 MB space: True
The switchkvm() call in scheduler() changes CR3 to use page directory of new process: False
The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context: False
PHYSTOP can be increased to some extent, simply by editing memlayout.h: True
xv6 uses physical memory upto 224 MB only: True
The process's address space gets mapped on frames, obtained from ~2MB:224MB range: True
The kernel's page table given by kpgdir variable is used as stack for scheduler's context: False

Question **17**

Incorrect

Mark 0.00 out of 1.50

Consider the reference string
6 4 2 0 1 2 6 9 2 0 5

If the number of page frames is 3, then total number of page faults (including initial), using LRU replacement is:

Answer: 8 ✖

#6# 6,4# 6,4,2 # 0,4,2#0,1,2#6,1,2#6,9,2#0,9,2#0,5,2

The correct answer is: 9

Question **18**

Partially correct

Mark 0.31 out of 0.50

Consider the image given below, which explains how paging works.

Figure 9.8 Paging hardware.

Mention whether each statement is True or False, with respect to this image.

True	False		
◉☑	○✖	The PTBR is present in the CPU as a register	✔
○✖	◉☑	The page table is indexed using frame number	✔
○☑	◉✖	The page table is indexed using page number	✖
◉✖	○☑	The locating of the page table using PTBR also involves paging translation	✖
○✖	◉☑	Size of page table is always determined by the size of RAM	✔
◉☑	○✖	The page table is itself present in Physical memory	✔
○☑	○✖	Maximum Size of page table is determined by number of bits used for page number	✖
◉☑	○✖	The physical address may not be of the same size (in bits) as the logical address	✔

The PTBR is present in the CPU as a register: True
The page table is indexed using frame number: False
The page table is indexed using page number: True
The locating of the page table using PTBR also involves paging translation: False
Size of page table is always determined by the size of RAM: False
The page table is itself present in Physical memory: True
Maximum Size of page table is determined by number of bits used for page number: True
The physical address may not be of the same size (in bits) as the logical address: True

Question **19**

Correct

Mark 2.00 out of 2.00

Given below is shared memory code with two processes sharing a memory segment.
The first process sends a user input string to second process. The second capitalizes the string. Then the first process prints the capitalized version.
Fill in the blanks to complete the code.

```
// First process
#define SHMSZ    27

int main()
{
   char c;
   int shmid;
   key_t key;
   char *shm, *s, string[128];
   key = 5679;
   if ((shmid =
```

shmget

✔ (key, SHMSZ, IPC_CREAT | 0666)) < 0) {

```
      perror("shmget");
      exit(1);
   }
   if ((shm =
```

shmat

✔ (shmid, NULL, 0)) == (char *) -1) {

```
      perror("shmat");
      exit(1);
   }
   s = shm;
   *s = '$';
   scanf("%s", string);
   strcpy(s + 1, string);
   *s = '
```

@

✔ '; //note the quotes

```
   while(*s != '
```

$

✔ ')

```
      sleep(1);
   printf("%s\n", s + 1);
   exit(0);
}
```

```
//Second process
#define SHMSZ    27
int main()
{
   int shmid;
   key_t key;
   char *shm, *s;
   int i;
   char string[128];
   key =
```

5679

```
    ;
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }
    s =
```

shm

```
    ;
    while(*s != '@')
        sleep(1);
    for(i = 0; i < strlen(s + 1); i++)
        s[i + 1] =  toupper(s[i + 1]);
    *s = '$';
    exit(0);
}
```

Question **20**

Partially correct

Mark 0.25 out of 0.50

Map the functionality/use with function/variable in xv6 code.

return a free page, if available; 0, otherwise	kinit1() ✗
Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed	mappages() ✓
Array listing the kernel memory mappings, to be used by setupkvm()	kmap[] ✓
Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices	kvmalloc() ✗
Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary	walkpgdir() ✓
Setup kernel part of a page table, and switch to that page table	setupkvm() ✗

Your answer is partially correct.

You have correctly selected 3.

The correct answer is: return a free page, if available; 0, otherwise → kalloc(), Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed → mappages(), Array listing the kernel memory mappings, to be used by setupkvm() → kmap[], Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices → setupkvm(), Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary → walkpgdir(), Setup kernel part of a page table, and switch to that page table → kvmalloc()

Question **21**

Partially correct

Mark 1.53 out of 2.50

Order events in xv6 timer interrupt code

(Transition from process P1 to P2's code.)

P2 is selected and marked RUNNING	12	✔
Change of stack from user stack to kernel stack of P1	3	✔
Timer interrupt occurs	2	✔
alltraps() will call iret	17	✖
change to context of P2, P2's kernel stack in use now	13	✔
P2's trap() will return to alltraps	16	✖
jump in vector.S	4	✔
P2 will return from sched() in yield()	14	✖
yield() is called	8	✔
trap() is called	7	✔
Process P2 is executing	18	✖
P1 is marked as RUNNABLE	9	✔
P2's yield() will return in trap()	15	✖
Process P1 is executing	1	✔
sched() is called,	11	✖
change to context of the scheduler, scheduler's stack in use now	10	✖
jump to alltraps	5	✔
Trapframe is built on kernel stack of P1	6	✔

Your answer is partially correct.

You have correctly selected 11.

The correct answer is: P2 is selected and marked RUNNING → 12, Change of stack from user stack to kernel stack of P1 → 3, Timer interrupt occurs → 2, alltraps() will call iret → 18, change to context of P2, P2's kernel stack in use now → 13, P2's trap() will return to alltraps → 17, jump in vector.S → 4, P2 will return from sched() in yield() → 15, yield() is called → 8, trap() is called → 7, Process P2 is executing → 14, P1 is marked as RUNNABLE → 9, P2's yield() will return in trap() → 16, Process P1 is executing → 1, sched() is called, → 10, change to context of the scheduler, scheduler's stack in use now → 11, jump to alltraps → 5, Trapframe is built on kernel stack of P1 → 6

Question **22**

Incorrect

Mark 0.00 out of 1.00

Given that the memory access time is 200 ns, probability of a page fault is 0.7 and page fault handling time is 8 ms,
The effective memory access time in nanoseconds is:

Answer: 192 ✖

The correct answer is: 5600060.00

Question **23**

Correct

Mark 0.25 out of 0.25

Select the state that is not possible after the given state, for a process:

New: Running ✔

Ready : Waiting ✔

Running: : None of these ✔

Waiting: Running ✔

Question **24**

Partially correct

Mark 0.63 out of 1.00

Select the correct statements about sched() and scheduler() in xv6 code

- [x] a. scheduler() switches to the selected process's context ✔
- [x] b. When either sched() or scheduler() is called, it does not return immediately to caller ✔
- [ ] c. After call to swtch() in sched(), the control moves to code in scheduler()
- [x] d. Each call to sched() or scheduler() involves change of one stack inside swtch() ✔
- [ ] e. After call to swtch() in scheduler(), the control moves to code in sched()
- [x] f. When either sched() or scheduler() is called, it results in a context switch ✔
- [x] g. sched() switches to the scheduler's context ✔
- [ ] h. sched() and scheduler() are co-routines

Your answer is partially correct.

You have correctly selected 5.
The correct answers are: sched() and scheduler() are co-routines, When either sched() or scheduler() is called, it does not return immediately to caller, When either sched() or scheduler() is called, it results in a context switch, sched() switches to the scheduler's context, scheduler() switches to the selected process's context, After call to swtch() in scheduler(), the control moves to code in sched(), After call to swtch() in sched(), the control moves to code in scheduler(), Each call to sched() or scheduler() involves change of one stack inside swtch()

Question **25**

Correct

Mark 0.25 out of 0.25

The data structure used in kalloc() and kfree() in xv6 is

- a. Doubly linked circular list
- b. Singly linked circular list
- c. Double linked NULL terminated list
- d. Singly linked NULL terminated list ✔

Your answer is correct.

The correct answer is: Singly linked NULL terminated list

◄ (Assignment) lseek system call in xv6

Jump to...

Started on	Saturday, 20 February 2021, 2:51 PM
State	Finished
Completed on	Saturday, 20 February 2021, 3:55 PM
Time taken	1 hour 3 mins
Grade	**7.30** out of 20.00 (**37**%)

Question **1**

Partially correct

Mark 0.80 out of 1.00

Select all the correct statements about the state of a process.

- [x] a. A process can self-terminate only when it's running ✔
- [x] b. Typically, it's represented as a number in the PCB ✔
- [x] c. A process that is running is not on the ready queue ✔
- [x] d. Processes in the ready queue are in the ready state ✔
- [ ] e. It is not maintained in the data structures by kernel, it is only for conceptual understanding of programmers
- [x] f. Changing from running state to waiting state results in "giving up the CPU" ✔
- [ ] g. A process in ready state is ready to receive interrupts
- [x] h. A waiting process starts running after the wait is over ✖
- [x] i. A process changes from running to ready state on a timer interrupt ✔
- [x] j. A process in ready state is ready to be scheduled ✔
- [x] k. A running process may terminate, or go to wait or become ready again ✔
- [x] l. A process waiting for I/O completion is typically woken up by the particular interrupt handler code ✔
- [ ] m. A process waiting for any condition is woken up by another process only
- [ ] n. A process changes from running to ready state on a timer interrupt or any I/O wait

Your answer is partially correct.

You have selected too many options.
The correct answers are: Typically, it's represented as a number in the PCB, A process in ready state is ready to be scheduled, Processes in the ready queue are in the ready state, A process that is running is not on the ready queue, A running process may terminate, or go to wait or become ready again, A process changes from running to ready state on a timer interrupt, Changing from running state to waiting state results in "giving up the CPU", A process can self-terminate only when it's running, A process waiting for I/O completion is typically woken up by the particular interrupt handler code

Question **2**

Incorrect

Mark 0.00 out of 1.00

For each line of code mentioned on the left side, select the location of sp/esp that is in use

jmp *%eax in entry.S	0x7c00 to 0x10000	✖
ljmp $(SEG_KCODE<<3), $start32 in bootasm.S	0x10000 to 0x7c00	✖
call bootmain in bootasm.S	0x7c00 to 0x10000	✖
cli in bootasm.S	0x7c00 to 0	✖
readseg((uchar*)elf, 4096, 0); in bootmain.c	The 4KB area in kernel image, loaded in memory, named as 'stack'	✖

Your answer is incorrect.

The correct answer is: jmp *%eax in entry.S → The 4KB area in kernel image, loaded in memory, named as 'stack', ljmp $(SEG_KCODE<<3), $start32 in bootasm.S → Immaterial as the stack is not used here, call bootmain in bootasm.S → 0x7c00 to 0, cli in bootasm.S → Immaterial as the stack is not used here, readseg((uchar*)elf, 4096, 0); in bootmain.c → 0x7c00 to 0

Question **3**

Correct

Mark 0.25 out of 0.25

Order the following events in boot process (from 1 onwards)

Boot loader	2	✔
Shell	6	✔
BIOS	1	✔
OS	3	✔
Init	4	✔
Login interface	5	✔

Your answer is correct.

The correct answer is: Boot loader → 2, Shell → 6, BIOS → 1, OS → 3, Init → 4, Login interface → 5

Question **4**

Partially correct

Mark 0.30 out of 0.50

Consider the following command and it's output:

```
$ ls -lht xv6.img kernel
-rw-rw-r-- 1 abhijit abhijit 4.9M Feb 15 11:09 xv6.img
-rwxrwxr-x 1 abhijit abhijit 209K Feb 15 11:09 kernel*
```

Following code in bootmain()

```
  readseg((uchar*)elf, 4096, 0);
```

and following selected lines from Makefile

```
xv6.img: bootblock kernel
    dd if=/dev/zero of=xv6.img count=10000
    dd if=bootblock of=xv6.img conv=notrunc
    dd if=kernel of=xv6.img seek=1 conv=notrunc

kernel: $(OBJS) entry.o entryother initcode kernel.ld
    $(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode entryother
    $(OBJDUMP) -S kernel > kernel.asm
    $(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > kernel.sym
```

Also read the code of bootmain() in xv6 kernel.
Select the options that describe the meaning of these lines and their correlation.

- [ ] a. Althought the size of the kernel file is 209 Kb, only 4Kb out of it is the actual kernel code and remaining part is all zeroes.
- [x] b. The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files ✔
- [x] c. The kernel.ld file contains instructions to the linker to link the kernel properly ✔
- [ ] d. The bootmain() code does not read the kernel completely in memory
- [ ] e. readseg() reads first 4k bytes of kernel in memory
- [ ] f. Althought the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.
- [ ] g. The kernel.asm file is the final kernel file
- [ ] h. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is not read as it is user programs.
- [x] i. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain(). ✔

Your answer is partially correct.

You have correctly selected 3.
The correct answers are: The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain()., readseg() reads first 4k bytes of kernel in memory, The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files, The kernel.ld file contains instructions to the linker to link the kernel properly, Althought the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.

Question **5**

Partially correct

Mark 0.50 out of 1.00

```
int f() {
    int count;
    for (count = 0; count< 2; count ++) {
        if (fork() ==0)
            printf("Operating-System\n");
     }
     printf("TYCOMP\n");
 }
```

The number of times "Operating-System" is printed, is:

Answer: 3

The correct answer is: 7.00

Question **6**

Partially correct

Mark 0.40 out of 0.50

Select Yes/True if the mentioned element must be a part of PCB

Select No/False otherwise.

Yes	No		
◉☑	○✗	PID	✔
◉☑	○✗	Process context	✔
◉☑	○✗	List of opened files	✔
◉☑	○✗	Process state	✔
◉✗	○☑	Parent's PID	✖
○✗	◉☑	Pointer to IDT	✔
○✗	◉☑	Function pointers to all system calls	✔
◉☑	○✗	Memory management information about that process	✔
○☑	◉✗	Pointer to the parent process	✖
◉☑	○✗	EIP at the time of context switch	✔

PID: Yes
Process context: Yes
List of opened files: Yes
Process state: Yes
Parent's PID: No
Pointer to IDT: No
Function pointers to all system calls: No
Memory management information about that process: Yes
Pointer to the parent process: Yes
EIP at the time of context switch: Yes

Question **7**

Incorrect

Mark 0.00 out of 1.00

Select all the correct statements about code of bootmain() in xv6

```
void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* pa;

  elf = (struct elfhdr*)0x10000;  // scratch space

  // Read 1st page off disk
  readseg((uchar*)elf, 4096, 0);

  // Is this an ELF executable?
  if(elf->magic != ELF_MAGIC)
    return;  // let bootasm.S handle error

  // Load each program segment (ignores ph flags).
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
  for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
  }

  // Call the entry point from the ELF header.
  // Does not return!
  entry = (void(*)(void))(elf->entry);
  entry();
}
```

Also, inspect the relevant parts of the xv6 code. binary files, etc and run commands as you deem fit to answer this question.

- a. The kernel file gets loaded at the Physical address 0x10000 +0x80000000 in memory. ✗
- b. The elf->entry is set by the linker in the kernel file and it's 0x80000000 ✗
- c. The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded ✓
- d. The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it. ✓
- e. The kernel file has only two program headers ✓
- f. The elf->entry is set by the linker in the kernel file and it's 0x80000000 ✗
- g. The readseg finally invokes the disk I/O code using assembly instructions ✓
- h. The elf->entry is set by the linker in the kernel file and it's 8010000c ✓
- i. The kernel file gets loaded at the Physical address 0x10000 in memory. ✓
- j. The condition if(ph->memsz > ph->filesz) is never true. ✗
- k. The stosb() is used here, to fill in some space in memory with zeroes ✓

Your answer is incorrect.

The correct answers are: The kernel file gets loaded at the Physical address 0x10000 in memory., The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it., The elf->entry is set by the linker in the kernel file and it's 8010000c, The readseg finally invokes the disk I/O code using assembly instructions, The stosb() is used here, to fill in some space in memory with zeroes, The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded, The kernel file has only two program headers

Question **8**

Partially correct

Mark 0.13 out of 0.25

Which of the following are NOT a part of job of a typical compiler?

- [x] a. Check the program for logical errors ✔
- [ ] b. Convert high level langauge code to machine code
- [ ] c. Process the # directives in a C program
- [ ] d. Invoke the linker to link the function calls with their code, extern globals with their declaration
- [ ] e. Check the program for syntactical errors
- [ ] f. Suggest alternative pieces of code that can be written

Your answer is partially correct.

You have correctly selected 1.
The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

Question **9**

Correct

Mark 0.25 out of 0.25

Rank the following storage systems from slowest (first) to fastest(last)

Cache	6	✔
Hard Disk	3	✔
RAM	5	✔
Optical Disks	2	✔
Non volatile memory	4	✔
Registers	7	✔
Magnetic Tapes	1	✔

Your answer is correct.

The correct answer is: Cache → 6, Hard Disk → 3, RAM → 5, Optical Disks → 2, Non volatile memory → 4, Registers → 7, Magnetic Tapes → 1

Question **10**

Partially correct

Mark 0.21 out of 0.50

Which of the following parts of a C program do not have any corresponding machine code ?

- [ ] a. local variable declaration
- [ ] b. global variables
- [x] c. function calls ✗
- [x] d. #directives ✓
- [ ] e. expressions
- [ ] f. pointer dereference
- [x] g. typedefs ✓

Your answer is partially correct.

You have correctly selected 2.
The correct answers are: #directives, typedefs, global variables

Question **11**

Correct

Mark 0.25 out of 0.25

Match a system call with it's description

pipe	create an unnamed FIFO storage with 2 ends - one for reading and another for writing	✓
dup	create a copy of the specified file descriptor into smallest available file descriptor	✓
dup2	create a copy of the specified file descriptor into another specified file descriptor	✓
exec	execute a binary file overlaying the image of current process	✓
fork	create an identical child process	✓

Your answer is correct.

The correct answer is: pipe → create an unnamed FIFO storage with 2 ends - one for reading and another for writing, dup → create a copy of the specified file descriptor into smallest available file descriptor, dup2 → create a copy of the specified file descriptor into another specified file descriptor, exec → execute a binary file overlaying the image of current process, fork → create an identical child process

Question **12**

Correct

Mark 0.25 out of 0.25

Match the register with the segment used with it.

eip	cs	✔
edi	es	✔
esi	ds	✔
ebp	ss	✔
esp	ss	✔

Your answer is correct.

The correct answer is: eip → cs, edi → es, esi → ds, ebp → ss, esp → ss

Question **13**

Correct

Mark 0.25 out of 0.25

What's the trapframe in xv6?

- a. A frame of memory that contains all the trap handler code
- b. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware only
- c. The IDT table
- d. A frame of memory that contains all the trap handler code's function pointers
- e. A frame of memory that contains all the trap handler's addresses
- f. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S ✔
- g. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by code in trapasm.S only

Your answer is correct.

The correct answer is: The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S

Question **14**

Incorrect

Mark 0.00 out of 0.50

Select all the correct statements about linking and loading.

Select one or more:

- a. Continuous memory management schemes can support dynamic linking and dynamic loading. ✗
- b. Loader is last stage of the linker program ✗
- c. Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently) ✓
- d. Dynamic linking and loading is not possible without demand paging or demand segmentation. ✓
- e. Dynamic linking essentially results in relocatable code. ✓
- f. Continuous memory management schemes can support static linking and static loading. (may be inefficiently) ✓
- g. Loader is part of the operating system ✓
- h. Static linking leads to non-relocatable code ✗
- i. Dynamic linking is possible with continous memory management, but variable sized partitions only. ✗

Your answer is incorrect.

The correct answers are: Continuous memory management schemes can support static linking and static loading. (may be inefficiently), Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently), Dynamic linking essentially results in relocatable code., Loader is part of the operating system, Dynamic linking and loading is not possible without demand paging or demand segmentation.

Question **15**

Incorrect

Mark 0.00 out of 0.25

In bootasm.S, on the line

```
ljmp    $(SEG_KCODE<<3), $start32
```

The SEG_KCODE << 3, that is shifting of 1 by 3 bits is done because

- a. The value 8 is stored in code segment
- b. The code segment is 16 bit and only upper 13 bits are used for segment number
- c. The code segment is 16 bit and only lower 13 bits are used for segment number ✗
- d. While indexing the GDT using CS, the value in CS is always divided by 8
- e. The ljmp instruction does a divide by 8 on the first argument

Your answer is incorrect.

The correct answer is: The code segment is 16 bit and only upper 13 bits are used for segment number

Question **16**

Partially correct

Mark 0.07 out of 0.50

Order the events that occur on a timer interrupt:

Change to kernel stack	1	✗
Jump to a code pointed by IDT	2	✗
Jump to scheduler code	5	✗
Set the context of the new process	4	✗
Save the context of the currently running process	3	✓
Execute the code of the new process	6	✗
Select another process for execution	7	✗

Your answer is partially correct.

You have correctly selected 1.

The correct answer is: Change to kernel stack → 2, Jump to a code pointed by IDT → 1, Jump to scheduler code → 4, Set the context of the new process → 6, Save the context of the currently running process → 3, Execute the code of the new process → 7, Select another process for execution → 5

Question **17**

Incorrect

Mark 0.00 out of 1.00

Consider the two programs given below to implement the command (ignore the fact that error checks are not done on return values of functions)

$ ls . /tmp/asdfksdf >/tmp/ddd 2>&1

Program 1

```
int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(1);
    dup(fd);
    close(2);
    dup(fd);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Program 2

```
int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    close(1);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(2);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Select all the correct statements about the programs

Select one or more:

- a. Both programs are correct ✗
- b. Program 2 makes sure that there is one file offset used for '2' and '1' ✗
- c. Only Program 2 is correct ✗
- d. Program 2 does 1>&2 ✗
- e. Program 2 ensures 2>&1 and does not ensure > /tmp/ddd ✗
- f. Program 1 makes sure that there is one file offset used for '2' and '1' ✓
- g. Program 1 is correct for > /tmp/ddd but not for 2>&1 ✗
- h. Program 1 does 1>&2 ✗
- i. Both program 1 and 2 are incorrect ✗
- j. Program 2 is correct for > /tmp/ddd but not for 2>&1 ✗
- k. Only Program 1 is correct ✓
- l. Program 1 ensures 2>&1 and does not ensure > /tmp/ddd ✗

Your answer is incorrect.

The correct answers are: Only Program 1 is correct, Program 1 makes sure that there is one file offset used for '2' and '1'

Question **18**

Correct

Mark 0.25 out of 0.25

Select the option which best describes what the CPU does during it's powered ON lifetime

- a. Ask the user what is to be done, and execute that task
- b. Ask the OS what is to be done, and execute that task
- c. Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, Ask the User or the OS what is to be done next, repeat
- d. Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, repeat ✔
- e. Fetch instruction specified by OS, Decode and execute it, repeat
- f. Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, Ask OS what is to be done next, repeat

The correct answer is: Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, repeat

Question **19**

Partially correct

Mark 0.86 out of 1.00

Consider the following code and MAP the file to which each fd points at the end of the code.

```
int main(int argc, char *argv[]) {
   int fd1, fd2 = 1, fd3 = 1, fd4 = 1;

   fd1 = open("/tmp/1", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
   fd2 = open("/tmp/2", O_RDDONLY);
   fd3 = open("/tmp/3", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
   close(0);
   close(1);
   dup(fd2);
   dup(fd3);
   close(fd3);
   dup2(fd2, fd4);
   printf("%d %d %d %d\n", fd1, fd2, fd3, fd4);
   return 0;
}
```

1	closed	✖
fd4	/tmp/2	✔
fd2	/tmp/2	✔
fd1	/tmp/1	✔
2	stderr	✔
0	/tmp/2	✔
fd3	closed	✔

Your answer is partially correct.

You have correctly selected 6.

The correct answer is: 1 → /tmp/3, fd4 → /tmp/2, fd2 → /tmp/2, fd1 → /tmp/1, 2 → stderr, 0 → /tmp/2, fd3 → closed

Question **20**

Incorrect

Mark 0.00 out of 2.00

Following code claims to implement the command

/bin/ls -l | /usr/bin/head -3 | /usr/bin/tail -1

Fill in the blanks to make the code work.

Note: Do not include space in writing any option. x[1][2] should be written without any space, and so is the case with [1] or [2]. Pay attention to exact syntax and do not write any extra character like ';' or = etc.

```
int main(int argc, char *argv[]) {
   int pid1, pid2;
   int pfd[
```

1

✖][2];

```
   pipe(
```

2

✖);

```
   pid1 =
```

3

✖ ;

```
   if(pid1 != 0) {
      close(pfd[0]
```

0

✖);

```
      close(
```

pid1

✖);

```
      dup(
```

pid2

✖);

```
      execl("/bin/ls", "/bin/ls", "
```

1

✖ ", NULL);

```
   }
   pipe(
```

✖);

✖ = fork();

```
   if(pid2 == 0) {
      close(
```

✖ ;

```
      close(0);
      dup(
```

✖);

```
      close(pfd[1]
```

```
✖ );
    close(
✖ );
    dup(
✖ );
    execl("/usr/bin/head", "/usr/bin/head", "
✖ ", NULL);
  } else {
    close(pfd
✖ );
    close(
✖ );
    dup(
✖ );
    close(pfd
✖ );
    execl("/usr/bin/tail", "/usr/bin/tail", "
✖ ", NULL);
 }
}
```

Question **21**

Partially correct

Mark 0.11 out of 1.00

Select all the correct statements about calling convention on x86 32-bit.

- [x] a. Return address is one location above the ebp ✔
- [x] b. Parameters may be passed in registers or on stack ✔
- [x] c. Space for local variables is allocated by substracting the stack pointer inside the code of the called function ✔
- [x] d. The ebp pointers saved on the stack constitute a chain of activation records ✔
- [x] e. The two lines in the beginning of each function, "push %ebp; mov %esp, %ebp", create space for local variables ✖
- [x] f. Parameters may be passed in registers or on stack ✔
- [x] g. The return value is either stored on the stack or returned in the eax register ✖
- [ ] h. Paramters are pushed on the stack in left-right order
- [ ] i. during execution of a function, ebp is pointing to the old ebp
- [x] j. Space for local variables is allocated by substracting the stack pointer inside the code of the caller function ✖
- [x] k. Compiler may allocate more memory on stack than needed ✔

Your answer is partially correct.

You have selected too many options.
The correct answers are: Compiler may allocate more memory on stack than needed, Parameters may be passed in registers or on stack, Parameters may be passed in registers or on stack, Return address is one location above the ebp, during execution of a function, ebp is pointing to the old ebp, Space for local variables is allocated by substracting the stack pointer inside the code of the called function, The ebp pointers saved on the stack constitute a chain of activation records

Question **22**

Correct

Mark 1.00 out of 1.00

Match the program with it's output (ignore newlines in the output. Just focus on the count of the number of 'hi')

main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }	hi	✔
main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }	hi hi	✔
main() { int i = NULL; fork(); printf("hi\n"); }	hi hi	✔
main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }	hi	✔

Your answer is correct.

The correct answer is: main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi, main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi hi, main() { int i = NULL; fork(); printf("hi\n"); } → hi hi, main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi

Question **23**

Incorrect

Mark 0.00 out of 0.50

Some part of the bootloader of xv6 is written in assembly while some part is written in C. Why is that so?
Select all the appropriate choices

- a. The code in assembly is required for transition to protected mode, from real mode; but calling convention was applicable all the time ✗
- b. The setting up of the most essential memory management infrastructure needs assembly code ✓
- c. The code for reading ELF file can not be written in assembly ✗
- d. The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C ✓

Your answer is incorrect.

The correct answers are: The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C, The setting up of the most essential memory management infrastructure needs assembly code

Question **24**

Incorrect

Mark 0.00 out of 0.50

xv6.img: bootblock kernel

```
    dd if=/dev/zero of=xv6.img count=10000
    dd if=bootblock of=xv6.img conv=notrunc
    dd if=kernel of=xv6.img seek=1 conv=notrunc
```

Consider above lines from the Makefile. Which of the following is incorrect?

- [x] a. The size of the kernel file is nearly 5 MB ✔
- [x] b. The kernel is located at block-1 of the xv6.img ✖
- [x] c. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies 10,000 blocks on the disk. ✖
- [ ] d. The size of xv6.img is exactly = (size of bootblock) + (size of kernel)
- [x] e. The bootblock is located on block-0 of the xv6.img ✖
- [x] f. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk. ✔
- [x] g. The bootblock may be 512 bytes or less (looking at the Makefile instruction) ✖
- [x] h. The xv6.img is the virtual disk that is created by combining the bootblock and the kernel file. ✖
- [x] i. The size of the xv6.img is nearly 5 MB ✖
- [x] j. xv6.img is the virtual processor used by the qemu emulator ✔
- [x] k. Blocks in xv6.img after kernel may be all zeroes. ✖

Your answer is incorrect.

The correct answers are: xv6.img is the virtual processor used by the qemu emulator, The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk., The size of the kernel file is nearly 5 MB, The size of xv6.img is exactly = (size of bootblock) + (size of kernel)

Question **25**

Incorrect

Mark 0.00 out of 1.00

Select the sequence of events that are NOT possible, assuming a non-interruptible kernel code

Select one or more:

- [ ] a. P1 running
  P1 makes system call
  timer interrupt
  Scheduler
  P2 running
  timer interrupt
  Scheuler
  P1 running
  P1's system call return

- [x] b. P1 running ✔
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again

- [ ] c. P1 running
  P1 makes system call
  system call returns
  P1 running
  timer interrupt
  Scheduler running
  P2 running

- [x] d. P1 running ✖
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P3 running
  Hardware interrupt
  Interrupt unblocks P1
  Interrupt returns
  P3 running
  Timer interrupt
  Scheduler
  P1 running

- [ ] e.
  P1 running
  P1 makes sytem call
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again

- [x] f. P1 running ✖
  keyboard hardware interrupt
  keyboard interrupt handler running
  interrupt handler returns
  P1 running
  P1 makes sytem call
  system call returns

P1 running
timer interrupt
scheduler
P2 running

Your answer is incorrect.

The correct answers are: P1 running
P1 makes sytem call and blocks
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again, P1 running
P1 makes system call
timer interrupt
Scheduler
P2 running
timer interrupt
Scheuler
P1 running
P1's system call return,
P1 running
P1 makes sytem call
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

Question **26**

Correct

Mark 0.25 out of 0.25

Which of the following are the files related to bootloader in xv6?

- a. bootasm.s and entry.S
- b. bootasm.S and bootmain.c ✔
- c. bootasm.S, bootmain.c and bootblock.c
- d. bootmain.c and bootblock.S

Your answer is correct.

The correct answer is: bootasm.S and bootmain.c

Question **27**

Correct

Mark 0.25 out of 0.25

Match the following parts of a C program to the layout of the process in memory

Instructions	Text section	✔
Local Variables	Stack Section	✔
Dynamically allocated memory	Heap Section	✔
Global and static data	Data section	✔

Your answer is correct.

The correct answer is:
Instructions → Text section, Local Variables → Stack Section,
Dynamically allocated memory → Heap Section,
Global and static data → Data section

Question **28**

Incorrect

Mark 0.00 out of 0.50

What will this program do?

```
int main() {
fork();
execl("/bin/ls", "/bin/ls", NULL);
printf("hello");
}
```

- a. one process will run ls, another will print hello
- b. run ls once ✖
- c. run ls twice
- d. run ls twice and print hello twice
- e. run ls twice and print hello twice, but output will appear in some random order

Your answer is incorrect.

The correct answer is: run ls twice

Question **29**

Correct

Mark 0.25 out of 0.25

What is the OS Kernel?

- a. The code that controls hardware, abstracts access to hardware resources using system calls, creates an environment for processes to be created and run ✔ correct
- b. The set of tools like compiler, linker, loader, terminal, shell, etc.
- c. Only the system programs like compiler, linker, loader, etc.
- d. Everything that I see on my screen

The correct answer is: The code that controls hardware, abstracts access to hardware resources using system calls, creates an environment for processes to be created and run

Question **30**

Correct

Mark 0.50 out of 0.50

Which of the following is/are not saved during context switch?

- [ ] a. Program Counter
- [ ] b. General Purpose Registers
- [x] c. Bus ✔
- [ ] d. Stack Pointer
- [ ] e. MMU related registers/information
- [x] f. Cache ✔
- [x] g. TLB ✔

Your answer is correct.

The correct answers are: TLB, Cache, Bus

Question **31**

Partially correct

Mark 0.10 out of 0.25

Select the order in which the various stages of a compiler execute.

Linking	3	✗
Syntatical Analysis	2	✓
Pre-processing	1	✓
Intermediate code generation	does not exist	✗
Loading	4	✗

Your answer is partially correct.

You have correctly selected 2.
The correct answer is: Linking → 4, Syntatical Analysis → 2, Pre-processing → 1, Intermediate code generation → 3, Loading → does not exist

Question **32**

Partially correct

Mark 0.08 out of 0.50

Order the sequence of events, in scheduling process P1 after process P0

context of P0 is saved in P0's PCB	2	✗
context of P1 is loaded from P1's PCB	3	✗
Process P1 is running	5	✗
timer interrupt occurs	6	✗
Process P0 is running	1	✓
Control is passed to P1	4	✗

Your answer is partially correct.

You have correctly selected 1.
The correct answer is: context of P0 is saved in P0's PCB → 3, context of P1 is loaded from P1's PCB → 4, Process P1 is running → 6, timer interrupt occurs → 2, Process P0 is running → 1, Control is passed to P1 → 5

Question **33**

Not answered

Marked out of 1.00

Select the correct statements about interrupt handling in xv6 code

- a. On any interrupt/syscall/exception the control first jumps in vectors.S
- b. The trapframe pointer in struct proc, points to a location on user stack
- c. Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt
- d. xv6 uses the 64th entry in IDT for system calls
- e. The CS and EIP are changed only after pushing user code's SS,ESP on stack
- f. The trapframe pointer in struct proc, points to a location on kernel stack
- g. The function trap() is the called only in case of hardware interrupt
- h. The CS and EIP are changed only immediately on a hardware interrupt
- i. All the 256 entries in the IDT are filled
- j. On any interrupt/syscall/exception the control first jumps in trapasm.S
- k. The function trap() is the called irrespective of hardware interrupt/system-call/exception
- l. xv6 uses the 0x64th entry in IDT for system calls
- m. Before going to alltraps, the kernel stack contains upto 5 entries.

Your answer is incorrect.

The correct answers are: All the 256 entries in the IDT are filled, Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt, xv6 uses the 64th entry in IDT for system calls, On any interrupt/syscall/exception the control first jumps in vectors.S, Before going to alltraps, the kernel stack contains upto 5 entries., The trapframe pointer in struct proc, points to a location on kernel stack, The function trap() is the called irrespective of hardware interrupt/system-call/exception, The CS and EIP are changed only after pushing user code's SS,ESP on stack

◄ (Assignment) Change free list management in xv6

Jump to...

Started on	Friday, 17 March 2023, 2:33 PM
State	Finished
Completed on	Friday, 17 March 2023, 4:54 PM
Time taken	2 hours 21 mins
Grade	**6.37** out of 10.00 (**63.73**%)

Question **1**

Incorrect

Mark 0.00 out of 0.50

The first instruction that runs when you do "make qemu" is

`cli`

from bootasm.S

Why?

- a. "cli" stands for clear screen and the screen should be cleared before OS boots.
- b. "cli" that is Command Line Interface needs to be enabled first
- c. "cli" clears all registers and makes them zero, so that processor is as good as "new"
- d. "cli" clears the pipeline of the CPU so that it is as good as "fresh" CPU
- e. "cli" enables interrupts, it is required because the kernel supports interrupts.
- f. "cli" disables interrupts. It is required because as of now there are no interrupt handlers available ✖
- g. It disables interrupts. It is required because the interrupt handlers of kernel are not yet installed.
- h. "cli" enables interrupts, it is required because the kernel must handle interrupts.

Your answer is incorrect.

The correct answer is: It disables interrupts. It is required because the interrupt handlers of kernel are not yet installed.

Question **2**

Incorrect

Mark 0.00 out of 0.50

The struct buf has a sleeplock, and not a spinlock, because

- a. struct buf is used as a general purpose cache by kernel and cache operations take lot of time, so better to use sleeplock rather than spinlock
- b. sleeplock is preferable because it is used in interrupt context and spinlock can not be used in interrupt context
- c. It could be a spinlock, but xv6 has chosen sleeplock for purpose of demonstrating how to use a sleeplock.
- d. struct buf is used for disk I/o which takes lot of time, so sleeping/blocking is the only option available. ✖
- e. struct buf is used for disk I/o which takes lot of time, so sleeping/blocking is preferred to spinning/busy-wait for the desired buf.

Your answer is incorrect.

The correct answer is: struct buf is used for disk I/o which takes lot of time, so sleeping/blocking is preferred to spinning/busy-wait for the desired buf.

Question **3**

Partially correct

Mark 0.25 out of 0.50

when is each of the following stacks allocated?

kernel stack of process	during fork() in allocproc()	✔
kernel stack for scheduler, on first processor	on the arrival of a hardware interrupt	✖
user stack of process	during exec()	✖
kernel stack for the scheduler, on other processors	in main()->startothers()	✔

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: kernel stack of process → during fork() in allocproc(), kernel stack for scheduler, on first processor → in entry.S, user stack of process → during fork() in copyuvm(), kernel stack for the scheduler, on other processors → in main()->startothers()

Question **4**

Partially correct

Mark 0.56 out of 0.75

Mark statements as True/False w.r.t. ptable.lock

True	False			
✖	✔	A process can sleep on ptable.lock if it can't aquire it.	✖	It's a spinlock!
✔	✖	It is taken by one process but released by another process, running on same processor	✖	
✔	✖	ptable.lock protects the proc[] array and all struct proc in the array	✔	
✖	✔	The swtch() in scheduler() is called without holding the ptable.lock when control jumps to it from sched()	✔	No. it's always held. sched() will hold the lock.
✔	✖	the rule of "never block holding a spinlock" does not apply to ptable.lock in xv6	✔	sched() is called only if you hold ptable.lock
✔	✖	One sequence of function calls which takes and releases the ptable.lock is this: iderw->sleep, acquire(ptable.lock)->sched->swtch()->scheduler()->swtch()->yield(), release(ptable.lock)	✔	One process slept, another was scheduled and it came out of timer interrupt.
✖	✔	ptable.lock can be held by different processes on different processors at the same time	✔	No lock can be held like this!
✖	✔	ptable.lock is acquired but never released	✔	how is that possible?

A process can sleep on ptable.lock if it can't aquire it.: False
It is taken by one process but released by another process, running on same processor: True
ptable.lock protects the proc[] array and all struct proc in the array: True
The swtch() in scheduler() is called without holding the ptable.lock when control jumps to it from sched(): False
the rule of "never block holding a spinlock" does not apply to ptable.lock in xv6: True
One sequence of function calls which takes and releases the ptable.lock is this:
iderw->sleep, acquire(ptable.lock)->sched->swtch()->scheduler()->swtch()->yield(), release(ptable.lock): True
ptable.lock can be held by different processes on different processors at the same time: False
ptable.lock is acquired but never released: False

Question **5**

Correct

Mark 0.50 out of 0.50

Consider the following command and it's output:

```
$ ls -lht xv6.img kernel
-rw-rw-r-- 1 abhijit abhijit 4.9M Feb 15 11:09 xv6.img
-rwxrwxr-x 1 abhijit abhijit 209K Feb 15 11:09 kernel*
```

Following code in bootmain()

```
  readseg((uchar*)elf, 4096, 0);
```

and following selected lines from Makefile

```
xv6.img: bootblock kernel
    dd if=/dev/zero of=xv6.img count=10000
    dd if=bootblock of=xv6.img conv=notrunc
    dd if=kernel of=xv6.img seek=1 conv=notrunc

kernel: $(OBJS) entry.o entryother initcode kernel.ld
    $(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode entryother
    $(OBJDUMP) -S kernel > kernel.asm
    $(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > kernel.sym
```

Also read the code of bootmain() in xv6 kernel.
Select the options that describe the meaning of these lines and their correlation.

- [x] a. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain(). ✔
- [ ] b. Althought the size of the kernel file is 209 Kb, only 4Kb out of it is the actual kernel code and remaining part is all zeroes.
- [ ] c. The bootmain() code does not read the kernel completely in memory
- [x] d. Althought the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes. ✔
- [x] e. readseg() reads first 4k bytes of kernel in memory ✔
- [ ] f. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is not read as it is user programs.
- [x] g. The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files ✔
- [x] h. The kernel.ld file contains instructions to the linker to link the kernel properly ✔
- [ ] i. The kernel.asm file is the final kernel file

Your answer is correct.

The correct answers are: The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain()., readseg() reads first 4k bytes of kernel in memory, The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files, The kernel.ld file contains instructions to the linker to link the kernel properly, Althought the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.

Question **6**

Partially correct

Mark 0.43 out of 0.75

code line, MMU setting: Match the line of xv6 code with the MMU setup employed

movw %ax, %gs	real mode	✗
jmp *%eax	protected mode with segmentation and 4 MB pages	✓
inb $0x64,%al	real mode	✓
ljmp $(SEG_KCODE<<3), $start32	protected mode with only segmentation	✗
readseg((uchar*)elf, 4096, 0);	protected mode with only segmentation	✓
orl $CR0_PE, %eax	real mode	✓
movl $(V2P_WO(entrypgdir)), %eax	protected mode with segmentation and 4 MB pages	✗

The correct answer is: movw %ax, %gs → protected mode with only segmentation, jmp *%eax → protected mode with segmentation and 4 MB pages, inb $0x64,%al → real mode, ljmp $(SEG_KCODE<<3), $start32 → real mode, readseg((uchar*)elf, 4096, 0); → protected mode with only segmentation, orl $CR0_PE, %eax → real mode, movl $(V2P_WO(entrypgdir)), %eax → protected mode with only segmentation

Question **7**

Correct

Mark 0.50 out of 0.50

Which of the following is DONE by allocproc() ?

- [ ] a. setup kernel memory mappings for the process
- [x] b. Select an UNUSED struct proc for use ✓
- [x] c. allocate kernel stack for the process ✓
- [ ] d. ensure that the process starts in trapret()
- [ ] e. setup the contents of the trapframe of the process properly
- [x] f. allocate PID to the process ✓
- [x] g. ensure that the process starts in forkret() ✓
- [x] h. setup the trapframe and context pointers appropriately ✓

The correct answers are: Select an UNUSED struct proc for use, allocate PID to the process, allocate kernel stack for the process, setup the trapframe and context pointers appropriately, ensure that the process starts in forkret()

Question **8**

Partially correct

Mark 0.25 out of 0.50

Mark statements as True/False w.r.t. the creation of free page list in xv6.

True	False			
☐	◉	kmem.use_lock is set to 1 after free page list is created, so that kmem.lock is taken before accessing kmem.freelist.	✗	
◉	☐	The pointers that link the pages together are in the first 4 bytes of the pages themselves	✓	

◉	☐	if(kmem.use_lock)
		acquire(&kmem.lock);
		is not done when called from kinit1() because there is no need to take the lock when kinit1() is running because interrupts are disabled and only one processor is running
			✓	

☐	◉	free page list is a singly circular linked list.	✓	it's singly linked NULL terminated list.

◉	☐	if(kmem.use_lock)
		acquire(&kmem.lock);
		this "if" condition is true, when kinit2() runs because multi-processor support has been enabled by now.
			✗	No. kinit2() calls kfree() and then initializes use_lock.

☐	◉	the kmem.lock is used by kfree() and kalloc() only.	✗	

kmem.use_lock is set to 1 after free page list is created, so that kmem.lock is taken before accessing kmem.freelist.: True
The pointers that link the pages together are in the first 4 bytes of the pages themselves: True
if(kmem.use_lock)
acquire(&kmem.lock);
is not done when called from kinit1() because there is no need to take the lock when kinit1() is running because interrupts are disabled and only one processor is running: True
free page list is a singly circular linked list.: False
if(kmem.use_lock)
acquire(&kmem.lock);
this "if" condition is true, when kinit2() runs because multi-processor support has been enabled by now.: False
the kmem.lock is used by kfree() and kalloc() only.: True

Question **9**

Partially correct

Mark 0.55 out of 1.00

Given below is code of sleeplock in xv6.

```
// Long-term locks for processes
struct sleeplock {
  uint locked;       // Is the lock held?
  struct spinlock lk; // spinlock protecting this sleep lock

  // For debugging:
  char *name;        // Name of lock.
  int pid;           // Process holding lock
};



void
acquiresleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  while (lk->locked) {
    sleep(lk, &lk->lk);
  }
  lk->locked = 1;
  lk->pid = myproc()->pid;
  release(&lk->lk);
}

void
releasesleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  lk->locked = 0;
  lk->pid = 0;
  wakeup(lk);
  release(&lk->lk);
}
```

Mark the statements as True/False w.r.t. this code.

True	False			
✖	☑	Wakeup() will wakeup the first process waiting for the lock	✖	Wakeup() will wakeup all processes waiting for the lock

✖	☑	acquire(&lk->lk);
		while (lk->locked) {
		sleep(lk, &lk->lk);
		}
		could also be written as
		acquire(&lk->lk);
		if (lk->locked) {
		sleep(lk, &lk->lk);
		}
			✖	loop is required because other process might have obtained the lock before this process returns from sleep().

True	False			
✖	☑	A process has acquired the sleeplock when it comes out of sleep()	✖	
✖	☑	sleep() is called holding a spinlock. This could be avoided by releasing the lock before calling sleep() and acquiring it again after call to sleep()	✖	
☑	✖	sleep() is the function which blocks a process.	✔	
☑	✖	The process which called acquiresleep() and then got blocked, is woken up by the timer interrupt	✔	it's woken up by another process which called releasesleep() and then wakeup()
✖	☑	the 'spinlock lk' protects 'locked' variable, but not the 'name' nor the 'pid'	✖	
☑	✖	the 'spinlock lk' is needed in a sleeplock, because access to the sleeplock for locking/unlocking itself creates a critical section	✔	
☑	✖	The spinlock lk->lk is held when the process comes out of sleep()	✔	
☑	✖	Sleeplock() will ensure that either the process gets the lock or the process gets blocked.	✔	
☑	✖	All processes waiting for the sleeplock will have a race for aquiring lk->lk spinlock, because all are woken up	✔	wakeup() wakes up all processes, and they "thunder" to take the spinlock.

Wakeup() will wakeup the first process waiting for the lock: False

```
acquire(&lk->lk);
  while (lk->locked) {
    sleep(lk, &lk->lk);
  }
```

could also be written as

```
acquire(&lk->lk);
  if (lk->locked) {
    sleep(lk, &lk->lk);
  }
```

: False

A process has acquired the sleeplock when it comes out of sleep(): False

sleep() is called holding a spinlock. This could be avoided by releasing the lock before calling sleep() and acquiring it again after call to sleep(): False

sleep() is the function which blocks a process.: True

The process which called acquiresleep() and then got blocked, is woken up by the timer interrupt: True

the 'spinlock lk' protects 'locked' variable, but not the 'name' nor the 'pid': False

the 'spinlock lk' is needed in a sleeplock, because access to the sleeplock for locking/unlocking itself creates a critical section: True

The spinlock lk->lk is held when the process comes out of sleep(): True

Sleeplock() will ensure that either the process gets the lock or the process gets blocked.: True

All processes waiting for the sleeplock will have a race for aquiring lk->lk spinlock, because all are woken up: True

Question **10**

Partially correct

Mark 0.50 out of 1.00

Mark the statements as True/False w.r.t. swtch()

True	False			
◉☑	○✖	p->context used in scheduler()->swtch() was **Generally** set when the process was interrupted earlier, and came via sched()->swtch()	✔	That's the only place when p->context is changed.
◉☑	○✖	swtch() is written in assembly language, because it violates calling convention, by changing the stack itself.	✔	
◉✖	○☑	swtch() is written in assembly language because it violates the calling convention by pushing parameters on the stack on its own.	✖	any function can push anything on stack, but remove it properly, that will not affect calling convention.
◉✖	○☑	swtch() called from scheduler() changes the stack from the process's kernel stack to the scheduler's kernel stack.	✖	it does reverse!
◉☑	○✖	swtch() is called only from sched() or scheduler()	✔	
◉✖	○☑	switch stores the old context on new stack, and restores new context from old stack.	✖	old goes on old, new comes from new stack
◉✖	○☑	sched() is the only place when p->context is set	✖	no. allocproc() also sets it.
◉✖	○☑	movl %esp, (%eax) means, *(c->scheduler) = contents of esp When swtch() is called from scheduler()	✖	No. it means c->scheduler = contents of esp.
◉☑	○✖	push in swtch() happens on old stack, while pop happens from new stack	✔	
◉☑	○✖	swtch() changes the context from "old" to "new"	✔	yeah, that's the definition

p->context used in scheduler()->swtch() was **Generally** set when the process was interrupted earlier, and came via sched()->swtch(): True
swtch() is written in assembly language, because it violates calling convention, by changing the stack itself.: True
swtch() is written in assembly language because it violates the calling convention by pushing parameters on the stack on its own.: False
swtch() called from scheduler() changes the stack from the process's kernel stack to the scheduler's kernel stack.: False
swtch() is called only from sched() or scheduler(): True
switch stores the old context on new stack, and restores new context from old stack.: False
sched() is the only place when p->context is set: False
movl %esp, (%eax)
means, *(c->scheduler) = contents of esp
When swtch() is called from scheduler(): False
push in swtch() happens on old stack, while pop happens from new stack: True
swtch() changes the context from "old" to "new": True

Question **11**

Correct

Mark 0.25 out of 0.25

Select the odd one out

- a. Kernel stack of new process to kernel stack of scheduler ✔
- b. Kernel stack of new process to Process stack of new process
- c. Kernel stack of running process to kernel stack of scheduler
- d. Process stack of running process to kernel stack of running process
- e. Kernel stack of scheduler to kernel stack of new process

The correct answer is: Kernel stack of new process to kernel stack of scheduler

Question **12**

Correct

Mark 0.25 out of 0.25

The variable 'end' used as argument to kinit1 has the value

- a. 80102da0
- b. 81000000
- c. 80000000
- d. 80110000
- e. 801154a8 ✔
- f. 8010a48c

The correct answer is: 801154a8

Question **13**

Partially correct

Mark 0.36 out of 0.50

With respect to this diagram, mark statements as True/False.

True	False			
◉☑	○✖	This diagram only shows the absolutely defined virtual->physical mappings, not the mappings defined at run time by kernel.	✔	
○☑	◉✖	"Kernel data + memory" on right side, here refers to the region from which pages are allocated to the kernel and process both.	✖	"Kernel data + memory" on LEFT side, here refers to the virtual addresses of kernel used at run time.
◉☑	○✖	The kernel virtual addresses start from KERNLINK = KERNBASE + EXTMEM	✔	
◉☑	○✖	The process's pages are mapped into physical memory from 1.03125 MB to PHYSTOP.	✔	
◉☑	○✖	The kernel file, after compilation, has maximum virtual address up to "data" as shown in the diagram, which is equal to "end" variable	✔	

True	False		
◉☑	○✖	When bootloader loads the kernel, then physical memory from EXTMEM upto EXTMEM + data is occupied.	✔
○☑	◉✖	PHYSTOP can be changed , but that needs kernel recompilation and re-execution.	✖

This diagram only shows the absolutely defined virtual->physical mappings, not the mappings defined at run time by kernel.: True
"Kernel data + memory" on right side, here refers to the region from which pages are allocated to the kernel and process both.: True
The kernel virtual addresses start from KERNLINK = KERNBASE + EXTMEM: True
The process's pages are mapped into physical memory from 1.03125 MB to PHYSTOP.: True
The kernel file, after compilation, has maximum virtual address up to "data" as shown in the diagram, which is equal to "end" variable: True
When bootloader loads the kernel, then physical memory from EXTMEM upto EXTMEM + data is occupied.: True
PHYSTOP can be changed , but that needs kernel recompilation and re-execution.: True

Question **14**

Correct

Mark 0.25 out of 0.25

Which of the following is not a task of the code of swtch() function

- [x] a. Save the return value of the old context code ✔
- [ ] b. Load the new context
- [x] c. Change the kernel stack location ✔
- [ ] d. Save the old context
- [ ] e. Switch stacks
- [ ] f. Jump to next context EIP

The correct answers are: Save the return value of the old context code, Change the kernel stack location

Question **15**

Correct

Mark 0.50 out of 0.50

We often use terms like "swtch() changes stack from process's kernel stack to scheduler's stack", or "the values are pushed on stack", or "the stack is initialized to the new page", etc. while discussing xv6 on x86.

Which of the following most accurately describes the meaning of "stack" in such sentences?

- a. The stack variable used in the program being discussed
- b. The region of memory where the kernel remembers all the function calls made
- c. The region of memory allocated by kernel for storing the parameters of functions
- d. The ss:esp pair ✔
- e. The "stack" variable declared in "stack.S" in xv6
- f. the region of memory which is currently used as stack by processor
- g. The stack segment

Your answer is correct.

The correct answer is: The ss:esp pair

Question **16**

Partially correct

Mark 0.31 out of 0.50

Mark the statements as True/False, with respect to the use of the variable "chan" in struct proc.

True	False		
◉✔	○✖	The value of 'chan' is changed only in sleep()	✔
◉✖	○✔	when chan is NULL, the 'state' in proc must be RUNNABLE.	✖
◉✔	○✖	chan stores the address of the variable, representing a condition, for which the process is waiting.	✔
◉✔	○✖	'chan' is used only by the sleep() and wakeup1() functions.	✔
◉✖	○✔	chan is the head pointer to a linked list of processes, waiting for a particular event to occur	✖
◉✔	○✖	When chan is not NULL, the 'state' in struct proc must be SLEPING	✔
◉✔	○✖	in xv6, the address of an appropriate variable is used as a "condition" for a waiting process.	✔
◉✖	○✔	Changing the state of a process automatically changes the value of 'chan'	✖

The value of 'chan' is changed only in sleep(): True
when chan is NULL, the 'state' in proc must be RUNNABLE.: False
chan stores the address of the variable, representing a condition, for which the process is waiting.: True
'chan' is used only by the sleep() and wakeup1() functions.: True
chan is the head pointer to a linked list of processes, waiting for a particular event to occur: False
When chan is not NULL, the 'state' in struct proc must be SLEPING: True
in xv6, the address of an appropriate variable is used as a "condition" for a waiting process.: True
Changing the state of a process automatically changes the value of 'chan': False

Question **17**

Correct

Mark 0.25 out of 0.25

Match function with it's meaning

iderw	Issue a disk read/write for a buffer, block the issuing process	✔
idewait	Wait for disc controller to be ready	✔
idestart	tell disc controller to start I/O for the first buffer on idequeue	✔
ideinit	Initialize the disc controller	✔
ideintr	disk interrupt handler, transfer data from conroller to buffer, wake up processes waiting for this buffer, start I/O for next buffer	✔

Your answer is correct.

The correct answer is: iderw → Issue a disk read/write for a buffer, block the issuing process, idewait → Wait for disc controller to be ready, idestart → tell disc controller to start I/O for the first buffer on idequeue, ideinit → Initialize the disc controller, ideintr → disk interrupt handler, transfer data from conroller to buffer, wake up processes waiting for this buffer, start I/O for next buffer

Question **18**

Correct

Mark 0.25 out of 0.25

Why is there a call to kinit2? Why is it not merged with knit1?

- a. Because there is a limit on the values that the argumets to knit1() can take.
- b. call to seginit() makes it possible to actually use PHYSTOP in argument to kinit2()
- c. When kinit1() is called there is a need for few page frames, but later knit2() is called to serve need of more page frames
- d. knit2 refers to virtual addresses beyond 4MB, which are not mapped before kvalloc() is called ✔

The correct answer is: knit2 refers to virtual addresses beyond 4MB, which are not mapped before kvalloc() is called

Question **19**

Correct

Mark 0.25 out of 0.25

Which of the following call sequence is impossible in xv6?

- a. Process 1: write() -> sys_write()-> file_write() -- timer interrupt -> trap() -> yield() -> sched() -> switch() (jumps to)-> scheduler() ->swtch() (jumps to)-> Process 2 (return call sequence) sched() -> yield() -> trap-> user-code
- b. Process 1: write() -> sys_write()-> file_write() -> writei() -> bread() -> bget() -> iderw() -> sleep() -> sched() -> switch() (jumps to)-> scheduler() ->swtch()(jumps to)-> Process 2 (return call sequence) sched() -> yield() -> trap-> user-code
- c. Process 1: timer interrupt -> trap() -> yield() -> sched() -> switch() -> scheduler()-> Process 2 runs -> write -> sys_write() -> trap()-> ... ✔

Your answer is correct.

The correct answer is: Process 1: timer interrupt -> trap() -> yield() -> sched() -> switch() -> scheduler()-> Process 2 runs -> write -> sys_write() -> trap()-> ...

Question **20**

Partially correct

Mark 0.17 out of 0.50

Mark statements as True/False, w.r.t. the given diagram

Started on	Sunday, 18 February 2024, 1:59 PM
State	Finished
Completed on	Sunday, 18 February 2024, 3:52 PM
Time taken	1 hour 53 mins
Grade	**13.72** out of 15.00 (**91.46**%)

Question **1**

Correct

Mark 0.50 out of 0.50

What's the trapframe in xv6?

- a. A frame of memory that contains all the trap handler's addresses
- b. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware only
- c. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by code in trapasm.S only
- d. A frame of memory that contains all the trap handler code
- e. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S ✔
- f. The IDT table
- g. A frame of memory that contains all the trap handler code's function pointers

Your answer is correct.

The correct answer is: The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S

Question **2**

Correct

Mark 0.50 out of 0.50

The trapframe, in xv6, is built by the

- a. vectors.S, trapasm.S
- b. hardware, vectors.S, trapasm.S, trap()
- c. hardware, vectors.S
- d. hardware, trapasm.S
- e. hardware, vectors.S, trapasm.S ✔

The correct answer is: hardware, vectors.S, trapasm.S

Question **3**

Correct

Mark 0.50 out of 0.50

The ljmp instruction in general does

- a. change the CS and EIP to 32 bit mode
- b. change the CS and EIP to 32 bit mode, and jumps to kernel code
- c. change the CS and EIP to 32 bit mode, and jumps to next line of code
- d. change the CS and EIP to 32 bit mode, and jumps to new value of EIP ✔

The correct answer is: change the CS and EIP to 32 bit mode, and jumps to new value of EIP

Question **4**

Correct

Mark 0.50 out of 0.50

The kernel is loaded at Physical Address

- a. 0x80000000
- b. 0x80100000
- c. 0x0010000
- d. 0x00100000 ✔

The correct answer is: 0x00100000

Question **5**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about MMU and it's functionality (on a non-demand paged system)

Select one or more:

- [ ] a. The operating system interacts with MMU for every single address translation
- [ ] b. Illegal memory access is detected by operating system
- [x] c. The Operating system sets up relevant CPU registers to enable proper MMU translations ✔
- [x] d. MMU is inside the processor ✔
- [x] e. Logical to physical address translations in MMU are done in hardware, automatically ✔
- [x] f. Illegal memory access is detected in hardware by MMU and a trap is raised ✔
- [ ] g. MMU is a separate chip outside the processor
- [ ] h. Logical to physical address translations in MMU are done with specific machine instructions

Your answer is correct.

The correct answers are: MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

Question **6**

Correct

Mark 0.50 out of 0.50

```
int value = 5;
int main()
{
    pid_t pid;
    pid = fork();
    if (pid == 0) { /* child process */
        value += 15;
        return 0;
    }
    else if (pid > 0) { /* parent process */
        wait(NULL);
        printf("%d", value); /* LINE A */
    }
    return 0;
}
```

What's the value printed here at LINE A?

Answer: 5 ✔

The correct answer is: 5

Question **7**

Correct

Mark 0.50 out of 0.50

The variable 'end' used as argument to kinit1 has the value

- a. 81000000
- b. 80110000
- c. 80000000
- d. 8010a48c
- e. 801154a8 ✔
- f. 80102da0

The correct answer is: 801154a8

Question **8**

Correct

Mark 0.50 out of 0.50

A process blocks itself means

- a. The kernel code of system call calls scheduler
- b. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler ✔
- c. The kernel code of an interrupt handler, moves the process to a waiting queue and calls scheduler
- d. The application code calls the scheduler

The correct answer is: The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

Question **9**

Partially correct

Mark 0.20 out of 0.50

Select all the correct statements about zombie processes

Select one or more:

- [ ] a. Zombie processes are harmless even if OS is up for long time
- [ ] b. A process becomes zombie when it's parent finishes
- [x] c. init() typically keeps calling wait() for zombie processes to get cleaned up ✔
- [x] d. A process can become zombie if it finishes, but the parent has finished before it ✔
- [ ] e. A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it
- [ ] f. A zombie process remains zombie forever, as there is no way to clean it up
- [ ] g. A zombie process occupies space in OS data structures
- [ ] h. If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent

Your answer is partially correct.

You have correctly selected 2.

The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it, A process can become zombie if it finishes, but the parent has finished before it, A zombie process occupies space in OS data structures, If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

Question **10**

Partially correct

Mark 0.88 out of 1.00

Suppose a processor supports base(relocation register) + limit scheme of MMU.

Assuming this, mark the statements as True/False

True	False		
☑	✗	The OS may terminate the process while handling the interrupt of memory violation	✔
✗	☑	The hardware may terminate the process while handling the interrupt of memory violation	✔
✗	☑	The compiler generates machine code assuming appropriately sized semgments for code, data and stack.	✔
☑	✗	The compiler generates machine code assuming continuous memory address space for process, and calculating appropriate sizes for code, and data;	✔
☑	✗	The hardware detects any memory access beyond the limit value and raises an interrupt	✔
✗	☑	The OS detects any memory access beyond the limit value and raises an interrupt	✖
✗	☑	The process sets up it's own relocation and limit registers when the process is scheduled	✔
☑	✗	The OS sets up the relocation and limit registers when the process is scheduled	✔

The OS may terminate the process while handling the interrupt of memory violation: True
The hardware may terminate the process while handling the interrupt of memory violation: False
The compiler generates machine code assuming appropriately sized semgments for code, data and stack.: False
The compiler generates machine code assuming continuous memory address space for process, and calculating appropriate sizes for code, and data;: True
The hardware detects any memory access beyond the limit value and raises an interrupt: True
The OS detects any memory access beyond the limit value and raises an interrupt: False
The process sets up it's own relocation and limit registers when the process is scheduled: False
The OS sets up the relocation and limit registers when the process is scheduled: True

Question **11**

Partially correct

Mark 0.30 out of 0.50

xv6.img: bootblock kernel

```
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
```

Consider above lines from the Makefile. Which of the following is INCORRECT?

- [x] a. xv6.img is the virtual processor used by the qemu emulator ✔
- [ ] b. The bootblock may be 512 bytes or less (looking at the Makefile instruction)
- [ ] c. The bootblock is located on block-0 of the xv6.img
- [x] d. The size of the kernel file is nearly 5 MB ✔
- [x] e. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies 10,000 blocks on the disk. ✖
- [x] f. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk. ✔
- [ ] g. The kernel is located at block-1 of the xv6.img
- [ ] h. The size of the xv6.img is nearly 5 MB
- [ ] i. Blocks in xv6.img after kernel may be all zeroes.
- [ ] j. The xv6.img is the virtual disk that is created by combining the bootblock and the kernel file.
- [ ] k. The size of xv6.img is exactly = (size of bootblock) + (size of kernel)

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: xv6.img is the virtual processor used by the qemu emulator, The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk., The size of the kernel file is nearly 5 MB, The size of xv6.img is exactly = (size of bootblock) + (size of kernel)

Question **12**

Partially correct

Mark 0.33 out of 0.50

Which of the following state transitions are not possible?

- [ ] a. Waiting -> Terminated
- [ ] b. Running -> Waiting
- [x] c. Ready -> Waiting ✔
- [x] d. Ready -> Terminated ✔

The correct answers are: Ready -> Terminated, Waiting -> Terminated, Ready -> Waiting

Question **13**

Partially correct

Mark 0.40 out of 0.50

Match the elements of C program to their place in memory

Arguments	Stack	✔
Code of main()	Code	✔
Global Static variables	Data	✔
#include files	No Memory needed	✖
Malloced Memory	Heap	✔
Function code	Code	✔
Global variables	Data	✔
#define MACROS	No memory needed	✖
Local Static variables	Data	✔
Local Variables	Stack	✔

The correct answer is: Arguments → Stack, Code of main() → Code, Global Static variables → Data, #include files → No memory needed, Malloced Memory → Heap, Function code → Code, Global variables → Data, #define MACROS → No Memory needed, Local Static variables → Data, Local Variables → Stack

Question **14**

Correct

Mark 1.00 out of 1.00

Match the program with it's output (ignore newlines in the output. Just focus on the count of the number of 'hi')

main() { int i = NULL; fork(); printf("hi\n"); }	hi hi	✔
main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }	hi	✔
main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }	hi hi	✔
main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }	hi	✔

Your answer is correct.

The correct answer is: main() { int i = NULL; fork(); printf("hi\n"); } → hi hi, main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi, main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi hi, main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi

Question **15**

Correct

Mark 1.00 out of 1.00

Mark the statements as True/False w.r.t. the basic concepts of memory management.

True	False		
✗	✓	The kernel refers to the page table for converting each virtual address to physical address.	✓
✗	✓	When a process is executing, each virtual address is converted into physical address by the kernel directly.	✓
✗	✓	The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.	✓
✓	✗	The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.	✓
✓	✗	When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.	✓
✓	✗	The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.	✓
✗	✓	The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.	✓

The kernel refers to the page table for converting each virtual address to physical address.: False
When a process is executing, each virtual address is converted into physical address by the kernel directly.: False
The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.: False
The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.: True
When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.: True
The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.: True
The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.: False

Question **16**

Correct

Mark 0.50 out of 0.50

Match the File descriptors to their meaning

1 Standard output ✔

0 Standard Input ✔

2 Standard error ✔

The correct answer is: 1 → Standard output, 0 → Standard Input, 2 → Standard error

Question **17**

Partially correct

Mark 0.25 out of 0.50

Some part of the bootloader of xv6 is written in assembly while some part is written in C. Why is that so?
Select all the appropriate choices

- [ ] a. The setting up of the most essential memory management infrastructure needs assembly code
- [ ] b. The code for reading ELF file can not be written in assembly
- [ ] c. The code in assembly is required for transition to protected mode, from real mode; but calling convention was applicable all the time
- [x] d. The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C ✔

Your answer is partially correct.

You have correctly selected 1.

The correct answers are: The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C, The setting up of the most essential memory management infrastructure needs assembly code

Question **18**

Correct

Mark 0.50 out of 0.50

The number of GDT entries setup during boot process of xv6 is

- a. 0
- b. 3 ✔
- c. 255
- d. 4
- e. 2
- f. 256

The correct answer is: 3

Question **19**

Partially correct

Mark 0.86 out of 1.00

Select all the correct statements about calling convention on x86 32-bit.

- [x] a. Compiler may allocate more memory on stack than needed ✔
- [ ] b. The two lines in the beginning of each function, "push %ebp; mov %esp, %ebp", create space for local variables
- [x] c. Parameters may be passed in registers or on stack ✔
- [ ] d. Space for local variables is allocated by substracting the stack pointer inside the code of the caller function
- [ ] e. Paramters are pushed on the stack in left-right order
- [x] f. Return address is one location above the ebp ✔
- [ ] g. during execution of a function, ebp is pointing to the old ebp
- [x] h. The ebp pointers saved on the stack constitute a chain of activation records ✔
- [x] i. Space for local variables is allocated by substracting the stack pointer inside the code of the called function ✔
- [ ] j. The return value is either stored on the stack or returned in the eax register
- [x] k. Parameters may be passed in registers or on stack ✔

Your answer is partially correct.

You have correctly selected 6.

The correct answers are: Compiler may allocate more memory on stack than needed, Parameters may be passed in registers or on stack, Parameters may be passed in registers or on stack, Return address is one location above the ebp, during execution of a function, ebp is pointing to the old ebp, Space for local variables is allocated by substracting the stack pointer inside the code of the called function, The ebp pointers saved on the stack constitute a chain of activation records

Question **20**

Correct

Mark 0.50 out of 0.50

How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security) ?

Select one:

- a. It prohibits a user mode process from running privileged instructions ✔
- b. It prohibits invocation of kernel code completely, if a user program is running
- c. It disallows hardware interrupts when a process is running
- d. It prohibits one process from accessing other process's memory

Your answer is correct.

The correct answer is: It prohibits a user mode process from running privileged instructions

Question **21**

Correct

Mark 0.50 out of 0.50

Match the register pairs

IP	CS	✔
SP	SS	✔
BP	SS	✔
SI	DS	✔
DI	DS	✔

The correct answer is: IP → CS, SP → SS, BP → SS, SI → DS, DI → DS

Question **22**

Correct

Mark 1.00 out of 1.00

Consider the image given below, which explains how paging works.

Figure 9.8 Paging hardware.

Mention whether each statement is True or False, with respect to this image.

True	False		
◉☑	○✖	The physical address may not be of the same size (in bits) as the logical address	✔
○✖	◉☑	The page table is indexed using frame number	✔
◉☑	○✖	The PTBR is present in the CPU as a register	✔
◉☑	○✖	Maximum Size of page table is determined by number of bits used for page number	✔
◉☑	○✖	The page table is indexed using page number	✔
○✖	◉☑	Size of page table is always determined by the size of RAM	✔
○✖	◉☑	The locating of the page table using PTBR also involves paging translation	✔
◉☑	○✖	The page table is itself present in Physical memory	✔

The physical address may not be of the same size (in bits) as the logical address: True
The page table is indexed using frame number: False
The PTBR is present in the CPU as a register: True
Maximum Size of page table is determined by number of bits used for page number: True
The page table is indexed using page number: True
Size of page table is always determined by the size of RAM: False
The locating of the page table using PTBR also involves paging translation: False
The page table is itself present in Physical memory: True

Question **23**

Correct

Mark 0.50 out of 0.50

The variable $stack in entry.S is

- a. located at 0x7c00
- b. located at 0
- c. located at less than 0x7c00
- d. located at the value given by %esp as setup by bootmain()
- e. a memory region allocated as a part of entry.S ✔

The correct answer is: a memory region allocated as a part of entry.S

Question **24**

Correct

Mark 0.50 out of 0.50

The right side of line of code "entry = (void(*)(void))(elf->entry)" means

- a. Get the "entry" in ELF structure and convert it into a void pointer
- b. Get the "entry" in ELF structure and convert it into a function void pointer
- c. Get the "entry" in ELF structure and convert it into a function pointer accepting no arguments and returning nothing ✔
- d. Convert the "entry" in ELF structure into void

The correct answer is: Get the "entry" in ELF structure and convert it into a function pointer accepting no arguments and returning nothing

◄ Homework questions: Basics of MM, xv6 booting

Jump to...

Quiz-2 (15 Marks) ►

Started on	Friday, 24 February 2023, 2:44 PM
State	Finished
Completed on	Friday, 24 February 2023, 4:39 PM
Time taken	1 hour 55 mins
Grade	**8.78** out of 10.00 (**87.77**%)

Question **1**

Correct

Mark 0.50 out of 0.50

What is meant by formatting a disk/partition?

- a. erasing all data on the disk/partition
- b. storing all the necessary programs on the disk/partition
- c. creating layout of empty directory tree/graph data structure ✔
- d. writing zeroes on all sectors

The correct answer is: creating layout of empty directory tree/graph data structure

Question **2**

Partially correct

Mark 0.44 out of 0.50

How does the compiler calculate addresses for the different parts of a C program, when paging is used?

Text	starting with 0	✔
Static variables	Immediately after the text, along with globals	✔
Local variables	An offset with respect to stack pointer (esp)	✔
#include files	No memory allocated, they are handled by linker	✖
#define	No memory allocated, they are handled by pre-processor	✔
typedef	No memory allocated, as they are not variables, but only conceptual definition of a type	✔
Global variables	Immediately after the text	✔
malloced memory	Heap (handled by the malloc-free library, using OS's system calls)	✔

Your answer is partially correct.

You have correctly selected 7.

The correct answer is: Text → starting with 0, Static variables → Immediately after the text, along with globals, Local variables → An offset with respect to stack pointer (esp), #include files → No memory allocated for the file, but if it contains variables, then variables may be allocated memory, #define → No memory allocated, they are handled by pre-processor, typedef → No memory allocated, as they are not variables, but only conceptual definition of a type, Global variables → Immediately after the text, malloced memory → Heap (handled by the malloc-free library, using OS's system calls)

Question **3**

Correct

Mark 0.50 out of 0.50

Consider the two programs given below to implement the command (ignore the fact that error checks are not done on return values of functions)

$ ls . /tmp/asdfksdf >/tmp/ddd 2>&1

Program 1

```
int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(1);
    dup(fd);
    close(2);
    dup(fd);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Program 2

```
int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    close(1);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(2);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Select all the correct statements about the programs

Select one or more:

- [x] a. Program 1 makes sure that there is one file offset used for '2' and '1' ✔
- [ ] b. Program 1 does 1>&2
- [ ] c. Both programs are correct
- [ ] d. Both program 1 and 2 are incorrect
- [ ] e. Program 2 does 1>&2
- [ ] f. Program 2 makes sure that there is one file offset used for '2' and '1'
- [ ] g. Program 2 ensures 2>&1 and does not ensure > /tmp/ddd
- [ ] h. Program 1 is correct for > /tmp/ddd but not for 2>&1
- [ ] i. Program 2 is correct for > /tmp/ddd but not for 2>&1
- [ ] j. Program 1 ensures 2>&1 and does not ensure > /tmp/ddd
- [x] k. Only Program 1 is correct ✔
- [ ] l. Only Program 2 is correct

Your answer is correct.

The correct answers are: Only Program 1 is correct, Program 1 makes sure that there is one file offset used for '2' and '1'

Question **4**

Partially correct

Mark 0.43 out of 0.50

You must have seen the error message "Segmentation fault, core dumped" very often.

With respect to this error message, mark the statements as True/False.

True	False			
✗	✓	The term "core" refers to the core code of the kernel.	✓	core means memory, all memory for the process.
✗	✓	On Linux, the message is printed only because the memory management scheme is segmentation	✓	No, it's just a term used, even if paging is used for memory management.
✓	✗	The process has definitely performed illegal memory access.	✓	
✓	✗	On Linux, the process was sent a SIGSEGV signal and the default handler for the signal is "Term", so the process is terminated.	✓	
✗	✓	The illegal memory access was detected by the kernel and the process was punished by kernel.	✗	"detection" is done by CPU, not kernel.
✓	✗	The core file can be analysed later using a debugger, to determine what went wrong.	✓	use gdb ./core ./executable-filename
✓	✗	The image of the process is stored in a file called "core", if the ulimit allows so.	✓	see ulimit -a

The term "core" refers to the core code of the kernel.: False
On Linux, the message is printed only because the memory management scheme is segmentation: False
The process has definitely performed illegal memory access.: True
On Linux, the process was sent a SIGSEGV signal and the default handler for the signal is "Term", so the process is terminated.: True
The illegal memory access was detected by the kernel and the process was punished by kernel.: False
The core file can be analysed later using a debugger, to determine what went wrong.: True
The image of the process is stored in a file called "core", if the ulimit allows so.: True

Question **5**

Correct

Mark 0.50 out of 0.50

Which of the following instructions should be privileged?

Select one or more:

- [x] a. Access memory management unit of the processor ✔
- [x] b. Turn off interrupts. ✔
- [ ] c. Read the clock.
- [x] d. Set value of timer. ✔
- [x] e. Switch from user to kernel mode. ✔ This instruction (like INT) is itself privileged - and that is why it not only changes the mode, but also ensures a jump to an ISR (kernel code)
- [ ] f. Access a general purpose register
- [ ] g. Set value of a memory location
- [x] h. Modify entries in device-status table ✔
- [x] i. Access I/O device. ✔

Your answer is correct.

The correct answers are: Set value of timer., Access memory management unit of the processor, Turn off interrupts., Modify entries in device-status table, Access I/O device., Switch from user to kernel mode.

Question **6**

Correct

Mark 0.50 out of 0.50

Map the block allocation scheme with the problem it suffers from

(Match pairs 1-1, match a scheme with the problem that it suffers from relatively the most, compared to others)

Indexed Allocation	Overhead of reading metadata blocks	✔
Continuous allocation	need for compaction	✔
Linked allocation	Too many seeks	✔

Your answer is correct.

The correct answer is: Indexed Allocation → Overhead of reading metadata blocks, Continuous allocation → need for compaction, Linked allocation → Too many seeks

Question **7**

Partially correct

Mark 0.43 out of 0.50

Following code claims to implement the command

/bin/ls -l | /usr/bin/head -3 | /usr/bin/tail -1

Fill in the blanks to make the code work.

Note: Do not include space in writing any option. x[1][2] should be written without any space, and so is the case with [1] or [2]. Pay attention to exact syntax and do not write any extra character like ';' or = etc.

```
int main(int argc, char *argv[]) {
    int pid1, pid2;
    int pfd[ 2 ✔ ][2];

    pipe( pfd[0] ✔ );
    pid1 = fork() ✔ ;
    if(pid1 != 0) {
        close(pfd[0] [0] ✔ );
        close( 1 ✔ );
        dup( pfd[0][1] ✔ );
        execl("/bin/ls", "/bin/ls", " -1 ✖ ", NULL);
    }
    pipe( pfd[1] ✔ );
    pid2 ✔ = fork();
    if(pid2 == 0) {
        close( pfd[0][1] ✖ ;
        close(0);
        dup( pfd[1][0] ✖ );
        close(pfd[1] [0] ✔ );
        close( 1 ✔ );
        dup( pfd[1][1] ✔ );
        execl("/usr/bin/head", "/usr/bin/head", " -3 ✔ ", NULL);
    } else {
        close(pfd [1][1] ✔ );
        close( 0 ✔ );
        dup( pfd[1][0] ✔ );
        close(pfd [0][0] ✔ );
```

```
        execl("/usr/bin/tail", "/usr/bin/tail", "  -1  ", NULL);
    }
}
```

Question **8**

Partially correct

Mark 0.45 out of 0.50

Match the elements of C program to their place in memory

Arguments	Stack	✔
Function code	Code	✔
#define MACROS	No memory needed	✖
#include files	No memory needed	✔
Global Static variables	Data	✔
Malloced Memory	Heap	✔
Global variables	Data	✔
Local Static variables	Data	✔
Local Variables	Stack	✔
Code of main()	Code	✔

The correct answer is: Arguments → Stack, Function code → Code, #define MACROS → No Memory needed, #include files → No memory needed, Global Static variables → Data, Malloced Memory → Heap, Global variables → Data, Local Static variables → Data, Local Variables → Stack, Code of main() → Code

Question **9**

Correct

Mark 0.50 out of 0.50

Doing a lookup on the pathname /a/b/b/c/d for opening the file "d" requires reading 6 ✔ no. of inodes. Assume that there are no hard/soft links on the path.

Write the answer as a number.

The correct answer is: 6

Question **10**

Partially correct

Mark 0.42 out of 0.50

Mark the statements about device drivers by marking as True or False.

True	False		
✖	✔	Different devices of the same type (e.g. 2 IDE hard disks) must need different device drivers.	✖
✔	✖	It's possible that a particular hardware has multiple device drivers available for it.	✔
✔	✖	Device driver is part of OS code	✔
✔	✖	Device driver is an intermediary between the hardware controller and OS	✔
✖	✔	Device driver is part of hardware	✔
✔	✖	Writing a device driver mandatorily demands reading the technical documentation about the hardware.	✔

Different devices of the same type (e.g. 2 IDE hard disks) must need different device drivers.: False
It's possible that a particular hardware has multiple device drivers available for it.: True
Device driver is part of OS code: True
Device driver is an intermediary between the hardware controller and OS: True
Device driver is part of hardware: False
Writing a device driver mandatorily demands reading the technical documentation about the hardware.: True

Question **11**

Correct

Mark 0.50 out of 0.50

Map each signal with it's meaning

SIGSEGV	Invalid Memory Reference	✔
SIGALRM	Timer Signal from alarm()	✔
SIGPIPE	Broken Pipe	✔
SIGUSR1	User Defined Signal	✔
SIGCHLD	Child Stopped or Terminated	✔

The correct answer is: SIGSEGV → Invalid Memory Reference, SIGALRM → Timer Signal from alarm(), SIGPIPE → Broken Pipe, SIGUSR1 → User Defined Signal, SIGCHLD → Child Stopped or Terminated

Question **12**

Partially correct

Mark 0.25 out of 0.50

Select all the blocks that may need to be written back to disk (if updated, of-course), as "Yes", when an operation of deleting a file is carried out on ext2 file system.

An option has to be correct entirely to be marked "Yes"

Block bitmap(s) for all the blocks of the file	Yes	✔
Data blocks of the file	Yes	✖
One or multiple data blocks of the parent directory	Yes	✖
Superblock	Yes	✔
Possibly one block bitmap corresponding to the parent directory	Yes	✔
One or more data bitmap blocks for the parent directory	Yes	✖

Your answer is partially correct.

only one data block of parent directory. multiple blocks not possible. an entry is always contained within one single block

You have correctly selected 3.

The correct answer is: Block bitmap(s) for all the blocks of the file → Yes, Data blocks of the file → No, One or multiple data blocks of the parent directory → No, Superblock → Yes, Possibly one block bitmap corresponding to the parent directory → Yes, One or more data bitmap blocks for the parent directory → No

Question **13**

Correct

Mark 0.50 out of 0.50

Select the compiler's view of the process's address space, for each of the following MMU schemes:
(Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

Segmentation	many continuous chunks of variable size	✔
Segmentation, then paging	many continuous chunks of variable size	✔
Paging	one continuous chunk	✔
Relocation + Limit	one continuous chunk	✔

Your answer is correct.

The correct answer is: Segmentation → many continuous chunks of variable size, Segmentation, then paging → many continuous chunks of variable size, Paging → one continuous chunk, Relocation + Limit → one continuous chunk

Question **14**

Partially correct

Mark 0.30 out of 0.50

Mark the statements about named and un-named pipes as True or False

True	False		
◉ ✔	○ ✖	Named pipe exists as a file	✔
○ ✔	◉ ✖	Un-named pipes can be used for communication between only "related" processes, if the common ancestor created it.	✖
◉ ✔	○ ✖	Both types of pipes are an extension of the idea of "message passing".	✔
○ ✖	○ ✔	A named pipe has a name decided by the kernel.	✖
◉ ✔	○ ✖	Un-named pipes are inherited by a child process from parent.	✔
◉ ✔	○ ✖	Named pipes can exist beyond the life-time of processes using them.	✔
◉ ✔	○ ✖	Both types of pipes provide FIFO communication.	✔
◉ ✖	○ ✔	The buffers for named-pipe are in process-memory while the buffers for the un-named pipe are in kernel memory.	✖
○ ✖	◉ ✔	Named pipes can be used for communication between only "related" processes.	✔
◉ ✖	○ ✔	The pipe() system call can be used to create either a named or un-named pipe.	✖

Named pipe exists as a file: True
Un-named pipes can be used for communication between only "related" processes, if the common ancestor created it.: True
Both types of pipes are an extension of the idea of "message passing".: True
A named pipe has a name decided by the kernel.: False
Un-named pipes are inherited by a child process from parent.: True
Named pipes can exist beyond the life-time of processes using them.: True
Both types of pipes provide FIFO communication.: True
The buffers for named-pipe are in process-memory while the buffers for the un-named pipe are in kernel memory.: False
Named pipes can be used for communication between only "related" processes.: False
The pipe() system call can be used to create either a named or un-named pipe.: False

Question **15**

Partially correct

Mark 0.29 out of 0.50

Mark the statements as True/False w.r.t. the basic concepts of memory management.

True	False		
✗	✓	The kernel refers to the page table for converting each virtual address to physical address.	✓
✓	✗	When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.	✓
✗	✓	The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.	✗
✗	✓	The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.	✓
✓	✗	The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.	✗
✗	✓	When a process is executing, each virtual address is converted into physical address by the kernel directly.	✗
✓	✗	The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.	✓

The kernel refers to the page table for converting each virtual address to physical address.: False
When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.: True
The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.: False
The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.: False
The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.: True
When a process is executing, each virtual address is converted into physical address by the kernel directly.: False
The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.: True

Question **16**

Correct

Mark 0.50 out of 0.50

Mark statements True/False w.r.t. change of states of a process. Note that a statement is true only if the claim and argument both are true.

Reference: The process state diagram (and your understanding of how kernel code works). Note - the diagram does not show zombie state!

Figure 3.2 Diagram of process state.

True	False		
◉☑	○✖	Every forked process has to go through ZOMBIE state, at least for a small duration.	✔
○✖	◉☑	A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first	✔
◉☑	○✖	Only a process in READY state is considered by scheduler	✔
◉☑	○✖	A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet	✔
○✖	◉☑	A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.	✔

Every forked process has to go through ZOMBIE state, at least for a small duration.: True
A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first: False
Only a process in READY state is considered by scheduler: True
A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet: True
A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.: False

Question **17**

Partially correct

Mark 0.45 out of 0.50

Select Yes if the mentioned element should be a part of PCB

Select No otherwise.

Yes	No		
✔	✖	Memory management information about that process	✔
✖	✔	Pointer to IDT	✔
✖	✔	PID of Init	✔
✔	✖	PID	✔
✖	✔	Function pointers to all system calls	✔
✔	✖	EIP at the time of context switch	✖
✔	✖	Process state	✔
✔	✖	List of opened files	✔
✔	✖	Process context	✔
✔	✖	Pointer to the parent process	✔

Memory management information about that process: Yes
Pointer to IDT: No
PID of Init: No
PID: Yes
Function pointers to all system calls: No
EIP at the time of context switch: Yes
Process state: Yes
List of opened files: Yes
Process context: Yes
Pointer to the parent process: Yes

Question **18**

Correct

Mark 0.50 out of 0.50

Predict the output of the program given here.

Assume that all the path names for the programs are correct. For example "/usr/bin/echo" will actually run echo command.

Assume that there is no mixing of printf output on screen if two of them run concurrently.

In the answer replace a new line by a single space.

For example::

```
good

output
```

should be written as `good output`

--

```
main() {
    int i;
    i = fork();
    if(i == 0)
        execl("/usr/bin/echo", "/usr/bin/echo", "hi", 0);
    else
        wait(0);
    fork();
    execl("/usr/bin/echo", "/usr/bin/echo", "one", 0);
}
```

Answer: hi one one ✔

The correct answer is: hi one one

Question **19**

Correct

Mark 0.50 out of 0.50

How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security) ?

Select one:

- a. It prohibits one process from accessing other process's memory
- b. It prohibits invocation of kernel code completely, if a user program is running
- c. It prohibits a user mode process from running privileged instructions ✔
- d. It disallows hardware interrupts when a process is running

Your answer is correct.

The correct answer is: It prohibits a user mode process from running privileged instructions

Question **20**

Partially correct

Mark 0.33 out of 0.50

Which of the following parts of a C program do not have any corresponding machine code ?

- [ ] a. pointer dereference
- [x] b. #directives ✔
- [x] c. typedefs ✔
- [ ] d. global variables
- [ ] e. function calls
- [ ] f. expressions
- [ ] g. local variable declaration

Your answer is partially correct.

You have correctly selected 2.
The correct answers are: #directives, typedefs, global variables

◄ Quiz-1 Preparation questions

Jump to...

Quiz - 2 (17 March 2023) ►

Started on	Saturday, 12 February 2022, 10:00:16 AM
State	Finished
Completed on	Saturday, 12 February 2022, 11:48:30 AM
Time taken	1 hour 48 mins
Grade	**8.26** out of 10.00 (**83**%)

Question **1**

Complete

Mark 0.50 out of 0.50

Suppose a program does a scanf() call.

Essentially the scanf does a read() system call.

This call will obviously "block" waiting for the user input.

In terms of OS data structures and execution of code, what does it mean?

Select one:

- a. OS code for read() will move PCB of current process to a wait queue and call scheduler
- b. read() will return and process will be taken to a wait queue
- c. OS code for read() will move the PCB of this process to a wait queue and return from the system call
- d. OS code for read() will call scheduler
- e. read() returns and process calls scheduler()

The correct answer is: OS code for read() will move PCB of current process to a wait queue and call scheduler

Question **2**

Complete

Mark 0.20 out of 1.00

Which parts of the xv6 code in bootasm.S bootmain.c , entry.S and in the codepath related to scheduler() and trap handling() can also be written in some other way, and still ensure that xv6 works properly?

Writing code is not necessary. You only need to comment on which part of the code could be changed to something else or written in another fashion.

Maximum two points to be written.

1. Instead of saving all the registers in trapframe, only necessary register can be saved.
2. Not using paging.

Question **3**

Complete

Mark 0.80 out of 1.00

Mark the statements, w.r.t. the scheduler of xv6 as True or False

True	False	
◉	○	`swtch` is a function that saves old context, loads new context, and returns to last EIP in the new context
○	◉	the control returns to `mycpu()->intena = intena; ();` after `swtch(&p->context, mycpu()->scheduler);` in `sched()`
◉	○	`sched()` and `scheduler()` are co-routines
◉	○	The work of selecting and scheduling a process is done only in `scheduler()` and not in `sched()`
◉	○	`swtch` is a function that does not return to the caller
○	◉	When a process is scheduled for execution, it resumes execution in `sched()` after the call to `swtch()`
◉	○	`sched()` *calls* `scheduler()` and `scheduler()` *calls* `sched()`
○	◉	the control returns to `switchkvm();` after `swtch(&(c->scheduler), p->context);` in `scheduler()`
◉	○	The function `scheduler()` executes using the kernel-only stack
◉	○	The variable c->scheduler on first processor uses the stack allocated entry.S

`swtch` is a function that saves old context, loads new context, and returns to last EIP in the new context: True
the control returns to `mycpu()->intena = intena; ();` after `swtch(&p->context, mycpu()->scheduler);` in `sched()`: False
`sched()` and `scheduler()` are co-routines: True
The work of selecting and scheduling a process is done only in `scheduler()` and not in `sched()`: True
`swtch` is a function that does not return to the caller: True
When a process is scheduled for execution, it resumes execution in `sched()` after the call to `swtch()`
: True
`sched()` *calls* `scheduler()` and `scheduler()` *calls* `sched()`: False
the control returns to `switchkvm();` after `swtch(&(c->scheduler), p->context);` in `scheduler()`: False
The function `scheduler()` executes using the kernel-only stack: True
The variable c->scheduler on first processor uses the stack allocated entry.S: True

Question **4**

Complete

Mark 0.50 out of 0.50

For each line of code mentioned on the left side, select the location of sp/esp that is in use

call bootmain in bootasm.S	0x7c00 to 0
jmp *%eax in entry.S	The 4KB area in kernel image, loaded in memory, named as 'stack'
readseg((uchar*)elf, 4096, 0); in bootmain.c	0x7c00 to 0
cli in bootasm.S	Immaterial as the stack is not used here
ljmp $(SEG_KCODE<<3), $start32 in bootasm.S	Immaterial as the stack is not used here

The correct answer is: call bootmain
in bootasm.S → 0x7c00 to 0, jmp *%eax
in entry.S → The 4KB area in kernel image, loaded in memory, named as 'stack', readseg((uchar*)elf, 4096, 0);
in bootmain.c → 0x7c00 to 0, cli
in bootasm.S → Immaterial as the stack is not used here, ljmp $(SEG_KCODE<<3), $start32
in bootasm.S → Immaterial as the stack is not used here

Question **5**

Complete

Mark 0.08 out of 0.50

The bootmain() function has this code

```
elf = (struct elfhdr*)0x10000;  // scratch space
readseg((uchar*)elf, 4096, 0);
```

Mark the statements as True or False with respect to this code.

In these statements 0x1000 is referred to as ADDRESS

True	False	
◉	○	This line loads the kernel code at ADDRESS
◉	○	This line effectively loads the ELF header and the program headers at ADDRESS
◉	○	If the value of ADDRESS is changed, then the program will not work
○	◉	It the value of ADDRESS is changed to a higher number (upto a limit), the program could still work
○	◉	The value ADDRESS is changed to a 0 the program could still work
○	◉	It the value of ADDRESS is changed to a lower number (upto a limit), the program could still work

This line loads the kernel code at ADDRESS: False
This line effectively loads the ELF header and the program headers at ADDRESS: False
If the value of ADDRESS is changed, then the program will not work: False
It the value of ADDRESS is changed to a higher number (upto a limit), the program could still work: True
The value ADDRESS is changed to a 0 the program could still work: False
It the value of ADDRESS is changed to a lower number (upto a limit), the program could still work: True

Question **6**

Complete

Mark 0.50 out of 0.50

Consider the following programs

exec1.c

```
#include <unistd.h>
#include <stdio.h>
int main() {
   execl("./exec2", "./exec2", NULL);
}
```

exec2.c

```
#include <unistd.h>
#include <stdio.h>
int main() {
   execl("/bin/ls", "/bin/ls", NULL);
  printf("hello\n");
}
```

Compiled as

```
cc    exec1.c  -o exec1
cc    exec2.c  -o exec2
```

And run as

```
$ ./exec1
```

Explain the output of the above command (./exec1)

Assume that /bin/ls , i.e. the 'ls' program exists.

Select one:

- a. Execution fails as the call to execl() in exec2 fails
- b. "ls" runs on current directory
- c. Execution fails as the call to execl() in exec1 fails
- d. Program prints hello
- e. Execution fails as one exec can't invoke another exec

The correct answer is: "ls" runs on current directory

Question **7**

Complete

Mark 0.88 out of 1.00

Select the correct statements about interrupt handling in xv6 code

- [ ] a. Before going to alltraps, the kernel stack contains upto 5 entries.
- [ ] b. xv6 uses the 0x64th entry in IDT for system calls
- [x] c. On any interrupt/syscall/exception the control first jumps in vectors.S
- [ ] d. On any interrupt/syscall/exception the control first jumps in trapasm.S
- [x] e. All the 256 entries in the IDT are filled
- [ ] f. The CS and EIP are changed only immediately on a hardware interrupt
- [x] g. The trapframe pointer in struct proc, points to a location on kernel stack
- [x] h. The CS and EIP are changed only after pushing user code's SS,ESP on stack
- [x] i. Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt
- [ ] j. The function trap() is the called only in case of hardware interrupt
- [x] k. xv6 uses the 64th entry in IDT for system calls
- [ ] l. The trapframe pointer in struct proc, points to a location on user stack
- [x] m. The function trap() is the called irrespective of hardware interrupt/system-call/exception

The correct answers are: All the 256 entries in the IDT are filled, Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt, xv6 uses the 64th entry in IDT for system calls, On any interrupt/syscall/exception the control first jumps in vectors.S, Before going to alltraps, the kernel stack contains upto 5 entries., The trapframe pointer in struct proc, points to a location on kernel stack, The function trap() is the called irrespective of hardware interrupt/system-call/exception, The CS and EIP are changed only after pushing user code's SS,ESP on stack

Question **8**

Complete

Mark 0.50 out of 0.50

What's the trapframe in xv6?

- a. The IDT table
- b. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S
- c. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by code in trapasm.S only
- d. A frame of memory that contains all the trap handler's addresses
- e. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware only
- f. A frame of memory that contains all the trap handler code's function pointers
- g. A frame of memory that contains all the trap handler code

The correct answer is: The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S

Question **9**

Complete

Mark 0.50 out of 0.50

Select Yes if the mentioned element should be a part of PCB

Select No otherwise.

Yes	No	
◉	○	PID
◉	○	Process state
◉	○	Memory management information about that process
◉	○	EIP at the time of context switch
○	◉	Function pointers to all system calls
◉	○	List of opened files
◉	○	Pointer to the parent process
○	◉	Pointer to IDT
◉	○	Process context
○	◉	PID of Init

PID: Yes
Process state: Yes
Memory management information about that process: Yes
EIP at the time of context switch: Yes
Function pointers to all system calls: No
List of opened files: Yes
Pointer to the parent process: Yes
Pointer to IDT: No
Process context: Yes
PID of Init: No

Question **10**

Complete

Mark 0.30 out of 0.50

Select all the correct statements about zombie processes

Select one or more:

- [ ] a. A process can become zombie if it finishes, but the parent has finished before it
- [ ] b. A process becomes zombie when it's parent finishes
- [ ] c. Zombie processes are harmless even if OS is up for long time
- [ ] d. A zombie process remains zombie forever, as there is no way to clean it up
- [x] e. A zombie process occupies space in OS data structures
- [x] f. A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it
- [ ] g. init() typically keeps calling wait() for zombie processes to get cleaned up
- [x] h. If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent

The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it, A process can become zombie if it finishes, but the parent has finished before it, A zombie process occupies space in OS data structures, If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

Question **11**

Complete

Mark 0.50 out of 0.50

Order the sequence of events, in scheduling process P1 after process P0

Process P0 is running	1
context of P1 is loaded from P1's PCB	4
Control is passed to P1	5
Process P1 is running	6
context of P0 is saved in P0's PCB	3
timer interrupt occurs	2

The correct answer is: Process P0 is running → 1, context of P1 is loaded from P1's PCB → 4, Control is passed to P1 → 5, Process P1 is running → 6, context of P0 is saved in P0's PCB → 3, timer interrupt occurs → 2

Question **12**

Complete

Mark 0.50 out of 0.50

Some part of the bootloader of xv6 is written in assembly while some part is written in C. Why is that so?
Select all the appropriate choices

- [ ] a. The code in assembly is required for transition to protected mode, from real mode; but calling convention was applicable all the time
- [x] b. The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C
- [ ] c. The code for reading ELF file can not be written in assembly
- [x] d. The setting up of the most essential memory management infrastructure needs assembly code

The correct answers are: The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C, The setting up of the most essential memory management infrastructure needs assembly code

Question **13**

Complete

Mark 0.50 out of 0.50

Select all the correct statements about code of bootmain() in xv6

```
void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* pa;

  elf = (struct elfhdr*)0x10000;  // scratch space

  // Read 1st page off disk
  readseg((uchar*)elf, 4096, 0);

  // Is this an ELF executable?
  if(elf->magic != ELF_MAGIC)
    return;  // let bootasm.S handle error

  // Load each program segment (ignores ph flags).
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
  for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
  }

  // Call the entry point from the ELF header.
  // Does not return!
  entry = (void(*)(void))(elf->entry);
  entry();
}
```

Also, inspect the relevant parts of the xv6 code. binary files, etc and run commands as you deem fit to answer this question.

- [x] a. The stosb() is used here, to fill in some space in memory with zeroes
- [ ] b. The condition if(ph->memsz > ph->filesz) is never true.
- [x] c. The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it.
- [x] d. The readseg finally invokes the disk I/O code using assembly instructions
- [x] e. The kernel file has only two program headers
- [ ] f. The elf->entry is set by the linker in the kernel file and it's 0x80000000
- [x] g. The kernel file gets loaded at the Physical address 0x10000 in memory.
- [ ] h. The elf->entry is set by the linker in the kernel file and it's 0x80000000
- [ ] i. The kernel file gets loaded at the Physical address 0x10000 +0x80000000 in memory.
- [x] j. The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded
- [x] k. The elf->entry is set by the linker in the kernel file and it's 8010000c

The correct answers are: The kernel file gets loaded at the Physical address 0x10000 in memory., The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it., The elf->entry is set by the linker in the kernel file and it's 8010000c, The readseg finally invokes the disk I/O code using assembly instructions, The stosb() is used here, to fill in some space in memory with zeroes, The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded, The kernel file has only two program headers

Question **14**

Complete

Mark 0.50 out of 0.50

Order the events that occur on a timer interrupt:

Jump to scheduler code	4
Set the context of the new process	6
Jump to a code pointed by IDT	2
Execute the code of the new process	7
Change to kernel stack of currently running process	1
Select another process for execution	5
Save the context of the currently running process	3

The correct answer is: Jump to scheduler code → 4, Set the context of the new process → 6, Jump to a code pointed by IDT → 2, Execute the code of the new process → 7, Change to kernel stack of currently running process → 1, Select another process for execution → 5, Save the context of the currently running process → 3

Question **15**

Complete

Mark 0.50 out of 0.50

In bootasm.S, on the line

```
ljmp    $(SEG_KCODE<<3), $start32
```

The SEG_KCODE << 3, that is shifting of 1 by 3 bits is done because

- a. The code segment is 16 bit and only lower 13 bits are used for segment number
- b. The ljmp instruction does a divide by 8 on the first argument
- c. While indexing the GDT using CS, the value in CS is always divided by 8
- d. The code segment is 16 bit and only upper 13 bits are used for segment number
- e. The value 8 is stored in code segment

The correct answer is: The code segment is 16 bit and only upper 13 bits are used for segment number

Question **16**

Complete

Mark 1.00 out of 1.00

Select the sequence of events that are NOT possible, assuming a non-interruptible kernel code

(Note: non-interruptible kernel code means, if the kernel code is executing, then interrupts will be disabled).

Note: A possible sequence may have some missing steps in between. An impossible sequence will will have n and n+1th steps such that n+1th step can not follow n'th step.

Select one or more:

- [ ] a. P1 running
  P1 makes system call
  system call returns
  P1 running
  timer interrupt
  Scheduler running
  P2 running

- [x] b. P1 running
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again

- [x] c.
  P1 running
  P1 makes sytem call
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again

- [x] d. P1 running
  P1 makes system call
  timer interrupt
  Scheduler
  P2 running
  timer interrupt
  Scheuler
  P1 running
  P1's system call return

- [ ] e. P1 running
  keyboard hardware interrupt
  keyboard interrupt handler running
  interrupt handler returns
  P1 running
  P1 makes sytem call
  system call returns
  P1 running
  timer interrupt
  scheduler
  P2 running

- [ ] f. P1 running
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler

P3 running
Hardware interrupt
Interrupt unblocks P1
Interrupt returns
P3 running
Timer interrupt
Scheduler
P1 running

The correct answers are: P1 running
P1 makes sytem call and blocks
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again, P1 running
P1 makes system call
timer interrupt
Scheduler
P2 running
timer interrupt
Scheuler
P1 running
P1's system call return,
P1 running
P1 makes sytem call
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

◄ Extra Reading on Linkers: A writeup by Ian Taylor (keep changing url string from 38 to 39, and so on)

Jump to...

(Code) IPC - Shm, Messages ►

Started on	Saturday, 16 March 2024, 1:32 PM
State	Finished
Completed on	Saturday, 16 March 2024, 3:47 PM
Time taken	2 hours 14 mins
Grade	**11.19** out of 15.00 (**74.6**%)

Question **1**

Correct

Mark 0.50 out of 0.50

Will this code work for a spinlock() operation? The intention here is to call compare-and-swap() only if the lock is not held (the if condition checks for the same).

```
void spinlock(int *lock) {
{
   while (true) {
      if (*lock == 0) {
         /* lock appears to be available */
         if (!compare_and_swap(lock, 0, 1))
         break
      }
   }
}
```

- a. No, because in the case of both processes succeeding in the "if" condition, both may end up acquiring the lock.
- b. No, because this breaks the atomicity requirement of compare-and-test.
- c. Yes, because no matter in which order the if-check and compare-and-swap run in multiple processes, only one process will succeed in compare-and-swap() and others will keep looping in while-loop. ✔
- d. Yes, because there is no race to update the lock variable

Your answer is correct.

The correct answer is: Yes, because no matter in which order the if-check and compare-and-swap run in multiple processes, only one process will succeed in compare-and-swap() and others will keep looping in while-loop.

Question **2**

Partially correct

Mark 0.25 out of 0.50

Match pairs

semaphore	wait() and signal()	✔
mutex	atomic test and set with loop	✖
spinlock	lock() and unlock()	✖
peterson	per process flag, global turn variable	✔

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: semaphore → wait() and signal(), mutex → lock() and unlock(), spinlock → atomic test and set with loop, peterson → per process flag, global turn variable

Question **3**

Incorrect

Mark 0.00 out of 0.50

In an ext2 file system, if the block size is 4KB and partition size is 24 GB, then the number of block groups will be:

Answer: 183 ✖

The correct answer is: 192.00

Question **4**

Correct

Mark 0.50 out of 0.50

Mark the statements as True or False, w.r.t. passing of arguments to system calls in xv6 code.

True	False		
◉	○	The functions like argint(), argstr() make the system call arguments available in the kernel.	✔
◉	○	Integer arguments are copied from user memory to kernel memory using argint()	✔
◉	○	The arguments to system call originally reside on process stack.	✔
◉	○	String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer	✔
○	◉	The arguments to system call are copied to kernel stack in trapasm.S	✔
◉	○	The arguments are accessed in the kernel code using esp on the trapframe.	✔
○	◉	Integer arguments are stored in eax, ebx, ecx, etc. registers	✔
○	◉	String arguments are first copied to trapframe and then from trapframe to kernel's other variables.	✔

The functions like argint(), argstr() make the system call arguments available in the kernel.: True
Integer arguments are copied from user memory to kernel memory using argint(): True
The arguments to system call originally reside on process stack.: True
String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer: True
The arguments to system call are copied to kernel stack in trapasm.S: False
The arguments are accessed in the kernel code using esp on the trapframe.: True
Integer arguments are stored in eax, ebx, ecx, etc. registers: False
String arguments are first copied to trapframe and then from trapframe to kernel's other variables.: False

Question **5**

Partially correct

Mark 0.46 out of 0.50

Map ext2 data structure features with their purpose

Many copies of Superblock ✔	Redundancy to ensure the most crucial data structure is not lost
A group ✔	Try to keep all the data of a directory and it's file close together in a group
rec_len field in directory entry ✔	allows holes and linking of entries in directory
Used directories count in group descriptor ✔	attempt is made to evenly spread the first-level directories, this count is used there
Mount count in superblock ✔	to enforce file check after certain amount of mounts at boot time
File Name is padded ✔	aligns all memory accesses on word boundary, improving performance
Combining file type and access rights in one variable ✔	saves 1 byte of space
Block bitmap is one block ✖	limits total number of files that can belong to a group
Inode table ✔	All inodes are kept together so that one disk read leads to reading many inodes together, effectively doing a buffering of subsequent inode reads, and to save sp
Inode bitmap is one block ✔	limits total number of files that can belong to a group
Free blocks count in superblock and group descriptor ✔	Redundancy to help fsck restore consistency
Inode table location in Group Descriptor ✔	Obvious, as it's per group and not per file-system

Your answer is partially correct.

You have correctly selected 11.

The correct answer is: Many copies of Superblock → Redundancy to ensure the most crucial data structure is not lost, A group → Try to keep all the data of a directory and it's file close together in a group, rec_len field in directory entry → allows holes and linking of entries in directory, Used directories count in group descriptor → attempt is made to evenly spread the first-level directories, this count is used there, Mount count in superblock → to enforce file check after certain amount of mounts at boot time, File Name is padded → aligns all memory accesses on word boundary, improving performance, Combining file type and access rights in one variable → saves 1 byte of space, Block bitmap is one block → Limits the size of a block group, thus improvising on purpose of a group, Inode table → All inodes are kept together so that one disk read leads to reading many inodes together, effectively doing a buffering of subsequent inode reads, and to save space on disk, Inode bitmap is one

block → limits total number of files that can belong to a group, Free blocks count in superblock and group descriptor → Redundancy to help fsck restore consistency, Inode table location in Group Descriptor → Obvious, as it's per group and not per file-system

Question **6**

Partially correct

Mark 0.33 out of 0.50

Select the correct statements about hard and soft links

Select one or more:

- [ ] a. Deleting a hard link always deletes the file
- [ ] b. Soft links increase the link count of the actual file inode
- [ ] c. Hard links increase the link count of the actual file inode
- [x] d. Soft links can span across partitions while hard links can't ✔
- [x] e. Deleting a soft link deletes the link, not the actual file ✔
- [ ] f. Deleting a soft link deletes both the link and the actual file
- [ ] g. Hard links can span across partitions while soft links can't
- [x] h. Hard links share the inode ✔
- [ ] i. Hard links enforce separation of filename from it's metadata in on-disk data structures.
- [ ] j. Soft link shares the inode of actual file
- [x] k. Deleting a hard link deletes the file, only if link count was 1 ✔
- [ ] l. Deleting a soft link deletes only the actual file

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: Soft links can span across partitions while hard links can't, Hard links increase the link count of the actual file inode, Deleting a soft link deletes the link, not the actual file, Deleting a hard link deletes the file, only if link count was 1, Hard links share the inode, Hard links enforce separation of filename from it's metadata in on-disk data structures.

Question **7**

Correct

Mark 0.50 out of 0.50

Which of the following is DONE by allocproc() ?

- [ ] a. ensure that the process starts in trapret()
- [x] b. allocate kernel stack for the process ✔
- [x] c. setup the trapframe and context pointers appropriately ✔
- [ ] d. setup the contents of the trapframe of the process properly
- [x] e. allocate PID to the process ✔
- [ ] f. setup kernel memory mappings for the process
- [x] g. ensure that the process starts in forkret() ✔
- [x] h. Select an UNUSED struct proc for use ✔

The correct answers are: Select an UNUSED struct proc for use, allocate PID to the process, allocate kernel stack for the process, setup the trapframe and context pointers appropriately, ensure that the process starts in forkret()

Question **8**

Partially correct

Mark 0.38 out of 0.50

The kernel ELF file contains these headers

```
Program Header:
    LOAD off    0x00001000 vaddr 0x80100000 paddr 0x00100000 align 2**12
         filesz 0x00007aab memsz 0x00007aab flags r-x
    LOAD off    0x00009000 vaddr 0x80108000 paddr 0x00108000 align 2**12
         filesz 0x00002516 memsz 0x0000d4a8 flags rw-
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
         filesz 0x00000000 memsz 0x00000000 flags rwx
```

mark the statemetns as True/False

True	False		
◉ ☑	○ ✖	Second header is for Data/Globals	✔
◉ ☑	○ ✖	First header is for the code/text	✔
○ ☑	◉ ✖	in bootmain() the third header leads to allocation of no-memory.	✖
◉ ☑	○ ✖	Third header is for stack	✔

Second header is for Data/Globals: True
First header is for the code/text: True
in bootmain() the third header leads to allocation of no-memory.: True
Third header is for stack: True

Question **9**

Partially correct

Mark 0.19 out of 0.50

It is proposed that when a process does an illegal memory access, xv6 terminate the process by printing the error message "Illegal Memory Access". Select all the changes that need to be done to xv6 for this as True (Note that the changes proposed here may not cover the exhaustive list of all changes required) and the un-necessary/wrong changes as False.

Required	Un-necessary/Wrong		
✖	☑	Add code that checks if the illegal memory access trap was due to an actual illegal memory access.	✖
✖	☑	Change mappages() to set specified permissions on each page table entry	✖
✖	☑	Mark each page as readonly in the page table mappings	✔
☑	✖	Handle the Illegal memory acceess trap in trap() function, and terminate the currently running process.	✔
☑	✖	Ensure that the address 0 is mapped to invalid	✖
☑	✖	Change allocuvm() to call mappages() with proper permissions on each page table entry	✔
☑	✖	Change in the Makefile and instruct cc/ld to start the code of each program at some address other than 0	✖
☑	✖	Change exec to treat text/data sections separately and call allocuvm() with proper flags for page table entries	✖

Add code that checks if the illegal memory access trap was due to an actual illegal memory access.: Un-necessary/Wrong
Change mappages() to set specified permissions on each page table entry: Un-necessary/Wrong
Mark each page as readonly in the page table mappings: Un-necessary/Wrong
Handle the Illegal memory acceess trap in trap() function, and terminate the currently running process.: Required
Ensure that the address 0 is mapped to invalid: Required
Change allocuvm() to call mappages() with proper permissions on each page table entry: Required
Change in the Makefile and instruct cc/ld to start the code of each program at some address other than 0: Required
Change exec to treat text/data sections separately and call allocuvm() with proper flags for page table entries: Required

Question **10**

Partially correct

Mark 0.25 out of 0.50

Suppose a file is to be created in an ext2 file system, in an existing directory /a/b/. Select from below, the list of blocks that may need modification.

Select one or more:

- [x] a. block bitmap in some block group ✔
- [ ] b. data blocks of /a/
- [ ] c. link count on /a/b/ inode
- [ ] d. inode of /a/
- [ ] e. existing data blocks of /a/b/
- [x] f. inode of /a/b/ ✔
- [ ] g. group descriptor(s)
- [ ] h. inode table in some block group
- [x] i. superblock ✔
- [x] j. inode bitmap in some block group ✔
- [ ] k. new data block in some block group
- [ ] l. inode bitmap referrring to /a/b/

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: superblock, group descriptor(s), inode of /a/b/, existing data blocks of /a/b/, inode table in some block group, inode bitmap in some block group, block bitmap in some block group, new data block in some block group

Question **11**

Partially correct

Mark 0.36 out of 0.50

Given below are statements about concurrency and parallelism

Select T/F

A concurrent system can allow more than one task to progress, whereas a parallel system can perform more than one task at the same time.

True	False		
✗	✓	A concurrent system allows more than one task to progress while a parallel system does not.	✓
✓	✗	A concurrent system can allow more than one task to progress, whereas a parallel system can perform more than one task at the same time.	✓
✗	✓	It is not possible to have concurrency without parallelism.	✓
✗	✓	Parallel systems allow more than one task to progress while concurrent systems do not.	✓
✗	✓	It is possible to have parallelism without concurrency	✓
✓	✗	It is possible to have concurrency without parallelism	✗
✗	✓	Both concurrency and parallelism are the same.	✗

A concurrent system allows more than one task to progress while a parallel system does not.: False
A concurrent system can allow more than one task to progress, whereas a parallel system can perform more than one task at the same time.: True
It is not possible to have concurrency without parallelism.: False
Parallel systems allow more than one task to progress while concurrent systems do not.: False
It is possible to have parallelism without concurrency: False
It is possible to have concurrency without parallelism: True
Both concurrency and parallelism are the same.: False

Question **12**

Correct

Mark 0.50 out of 0.50

Consider this program.
Some statements are identified using the // comment at the end.

Assume that = is an atomic operation.

```
#include <stdio.h>
#include <pthread.h>
long c = 0, c1 = 0, c2 = 0, run = 1;
void *thread1(void *arg) {
    while(run == 1) {//E
        c = 10; //A
        c1 = c2 + 5; //B
    }
}
void *thread2(void *arg) {
    while(run == 1) {//F
        c = 20;//C
        c2 = c1 + 3;//D
    }
}
int main() {
    pthread_t th1, th2;
    pthread_create(&th1, NULL, thread1, NULL);
    pthread_create(&th2, NULL, thread2, NULL);
    sleep(2);
    run = 0;
    fprintf(stdout, "c = %ld c1+c2 = %ld c1 = %ld c2 = %ld \n", c, c1+c2, c1, c2);
    fflush(stdout);
}
```

Which statements are part of the critical Section?

Yes	No		
✗	✓	F	✔
✗	✓	E	✔
✓	✗	B	✔
✗	✓	C	✔
✓	✗	D	✔
✗	✓	A	✔

F: No
E: No
B: Yes
C: No
D: Yes
A: No

Question **13**

Correct

Mark 0.50 out of 0.50

Select the statement that most correctly describes what setupkvm() does

- a. creates a 2-level page table for the use of the kernel, as specified in gdtdesc
- b. creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global arrray ✔
- c. creates a 1-level page table for the use by the kernel, as specified in kmap[] global array
- d. creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global arrray and makes kpgdir point to it

The correct answer is: creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global arrray

Question **14**

Correct

Mark 0.50 out of 0.50

Mark the statements as True/False, with respect to the use of the variable "chan" in struct proc.

True	False		
● ✔	○ ✖	in xv6, the address of an appropriate variable is used as a "condition" for a waiting process.	✔
● ✔	○ ✖	When chan is not NULL, the 'state' in struct proc must be SLEPING	✔
○ ✖	● ✔	Changing the state of a process automatically changes the value of 'chan'	✔
○ ✖	● ✔	when chan is NULL, the 'state' in proc must be RUNNABLE.	✔
● ✔	○ ✖	chan stores the address of the variable, representing a condition, for which the process is waiting.	✔
○ ✖	● ✔	chan is the head pointer to a linked list of processes, waiting for a particular event to occur	✔
● ✔	○ ✖	'chan' is used only by the sleep() and wakeup1() functions.	✔
● ✔	○ ✖	The value of 'chan' is changed only in sleep()	✔

in xv6, the address of an appropriate variable is used as a "condition" for a waiting process.: True
When chan is not NULL, the 'state' in struct proc must be SLEPING: True
Changing the state of a process automatically changes the value of 'chan': False
when chan is NULL, the 'state' in proc must be RUNNABLE.: False
chan stores the address of the variable, representing a condition, for which the process is waiting.: True
chan is the head pointer to a linked list of processes, waiting for a particular event to occur: False
'chan' is used only by the sleep() and wakeup1() functions.: True
The value of 'chan' is changed only in sleep(): True

Question **15**

Partially correct

Mark 0.25 out of 0.50

Mark statements about deadlocks as True or false

True	False		
✖	☑	A deadlock must involve at least two processes	✖
☑	✖	Deadlocks are not possible if there is no race	✔
☑	✖	A deadlock is possible only if all the 4 conditions of mutual exclusion, cyclic wait, hold and wait, and no preemption are satisfied	✔
☑	✖	A deadlock necessarily requires a cycle in the resource allocation graph	✔
☑	✖	Cycle in the resource allocation graph does not necessarily mean a deadlock	✖
✖	☑	Deadlocks are the same as livelocks	✖

A deadlock must involve at least two processes: False
Deadlocks are not possible if there is no race: True
A deadlock is possible only if all the 4 conditions of mutual exclusion, cyclic wait, hold and wait, and no preemption are satisfied: True
A deadlock necessarily requires a cycle in the resource allocation graph: True
Cycle in the resource allocation graph does not necessarily mean a deadlock: True
Deadlocks are the same as livelocks: False

Question **16**

Correct

Mark 0.50 out of 0.50

Why V2P_WO is used in entry.S and not V2P ?

- a. Because the processor can not do a type casting at run time
- b. The typecasting has the effect of creating virtual address, while without typecast we get physical address.
- c. Because entry.S is an assembly code file and assemblers do not know about data types and type casting. ✔
- d. It's a mistake. They could have used the same macro in both places.
- e. The two macros are different. They lead to different calculations.

Your answer is correct.

The correct answer is: Because entry.S is an assembly code file and assemblers do not know about data types and type casting.

Question **17**

Correct

Mark 0.50 out of 0.50

The variable $stack in entry.S is

- a. a memory region allocated as a part of entry.S ✔
- b. located at the value given by %esp as setup by bootmain()
- c. located at 0
- d. located at 0x7c00
- e. located at less than 0x7c00

The correct answer is: a memory region allocated as a part of entry.S

Question **18**

Partially correct

Mark 0.13 out of 0.50

Match the code with it's functionality

S = 1
Wait(S)
Critical Section
Signal(S);

Counting semaphore ✖

S = 5
Wait(S)
Critical Section
Signal(S)

Execution order P2, then P1 ✖

S = 0
P1:
Statement1;
Signal(S)

Execution order P1, then P2 ✔

P2:
Wait(S)
Statment2;

S1 = 0; S2 = 0;
P2:
Statement1;
Signal(S2);

P1:
Wait(S2);
Statemetn2;
Signal(S1);

Binary Semaphore for mutual exclusion ✖

P3:
Wait(S1);
Statement S3;

Your answer is partially correct.

You have correctly selected 1.
The correct answer is: S = 1
Wait(S)
Critical Section
Signal(S); → Binary Semaphore for mutual exclusion, S = 5
Wait(S)
Critical Section
Signal(S) → Counting semaphore, S = 0
P1:
Statement1;
Signal(S)

P2:
Wait(S)
Statment2; → Execution order P1, then P2, S1 = 0; S2 = 0;
P2:
Statement1;
Signal(S2);

P1:
Wait(S2);
Statemetn2;
Signal(S1);

P3:

Wait(S1);
Statement S3; → Execution order P2, P1, P3

Question **19**
Correct
Mark 0.50 out of 0.50

The "push 0" in vectors.S is

- a. A placeholder to match the size of struct trapframe
- b. To be filled in as the return value of the system call
- c. Place for the error number value ✔
- d. To indicate that it's a system call and not a hardware interrupt

The correct answer is: Place for the error number value

Question **20**
Correct
Mark 0.50 out of 0.50

Mark statements as True/False w.r.t. the creation of free page list in xv6.

True	False			
◉	○	The pointers that link the pages together are in the first 4 bytes of the pages themselves	✔	

◉	○	if(kmem.use_lock)
		acquire(&kmem.lock);
		is not done when called from kinit1() because there is no need to take the lock when kinit1() is running because interrupts are disabled and only one processor is running
			✔	

◉	○	the kmem.lock is used by kfree() and kalloc() only.	✔	
○	◉	free page list is a singly circular linked list.	✔	it's singly linked NULL terminated list.

○	◉	if(kmem.use_lock)
		acquire(&kmem.lock);
		this "if" condition is true, when kinit2() runs because multi-processor support has been enabled by now.
			✔	No. kinit2() calls kfree() and then initializes use_lock.

◉	○	kmem.use_lock is set to 1 after free page list is created, so that kmem.lock is taken before accessing kmem.freelist.	✔	

The pointers that link the pages together are in the first 4 bytes of the pages themselves: True
if(kmem.use_lock)
acquire(&kmem.lock);
is not done when called from kinit1() because there is no need to take the lock when kinit1() is running because interrupts are disabled and only one processor is running: True
the kmem.lock is used by kfree() and kalloc() only.: True
free page list is a singly circular linked list.: False
if(kmem.use_lock)
acquire(&kmem.lock);
this "if" condition is true, when kinit2() runs because multi-processor support has been enabled by now.: False
kmem.use_lock is set to 1 after free page list is created, so that kmem.lock is taken before accessing kmem.freelist.: True

Question **21**

Incorrect

Mark 0.00 out of 0.50

In the code below assume that each function can be executed concurrently by many threads/processes.
Ignore syntactical issues, and focus on the semantics.

This program is an example of

```
spinlock a, b;  // assume initialized
thread1() {
    spinlock(a);
    //some code;
    spinlock(b);
    //some code;
    spinunlock(b);
    spinunlock(a);
}
thread2() {
    spinlock(a);
    //some code;
    spinlock(b);
    //some code;
    spinunlock(b);
    spinunlock(a);
}
```

- a. Deadlock or livelock depending on actual race ✖
- b. Livelock
- c. None of these
- d. Self Deadlock
- e. Deadlock

Your answer is incorrect.

The correct answer is: None of these

Question **22**

Correct

Mark 0.50 out of 0.50

The variable 'end' used as argument to kinit1 has the value

- a. 801154a8 ✔
- b. 8010a48c
- c. 80110000
- d. 80000000
- e. 80102da0
- f. 81000000

The correct answer is: 801154a8

Question **23**

Partially correct

Mark 0.17 out of 0.50

Which of the following is done by mappages()?

- [ ] a. create page table mappings to the range given by "pa" and "pa + size"
- [ ] b. allocate page frame if required
- [ ] c. allocate page table if required
- [x] d. create page table mappings for the range given by "va" and "va + size" ✔
- [ ] e. allocate page directory if required

The correct answers are: create page table mappings for the range given by "va" and "va + size", allocate page table if required, create page table mappings to the range given by "pa" and "pa + size"

Question **24**

Correct

Mark 0.50 out of 0.50

Mark statements as T/F

All statements are in the context of preventing deadlocks.

True	False		
◉ ☑	○ ✖	Hold and wait means a thread/process holding some locks and waiting for acquiring some.	✔
◉ ☑	○ ✖	Circular wait is avoided by enforcing a lock ordering	✔
○ ✖	◉ ☑	If a resource allocation graph contains a cycle then there is a guarantee of a deadlock	✔
○ ✖	◉ ☑	The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order	✔
◉ ☑	○ ✖	Deadlock is possible if all the conditions are met at the same time: Mutual exclusion, hold and wait, no pre-emption, circular wait.	✔
◉ ☑	○ ✖	A process holding one resources and waiting for just one more resource can also be involved in a deadlock.	✔
◉ ☑	○ ✖	Mutual exclusion is a necessary condition for deadlock because it brings in locks on which deadlock happens	✔

Hold and wait means a thread/process holding some locks and waiting for acquiring some.: True
Circular wait is avoided by enforcing a lock ordering: True
If a resource allocation graph contains a cycle then there is a guarantee of a deadlock: False
The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order: False
Deadlock is possible if all the conditions are met at the same time: Mutual exclusion, hold and wait, no pre-emption, circular wait.: True
A process holding one resources and waiting for just one more resource can also be involved in a deadlock.: True
Mutual exclusion is a necessary condition for deadlock because it brings in locks on which deadlock happens: True

Question **25**

Correct

Mark 0.50 out of 0.50

Doing a lookup on the pathname /a/b/b/c/d for opening the file "d" requires reading 6 ✔ no. of inodes. Assume that there are no hard/soft links on the path.

Write the answer as a number.

The correct answer is: 6

Question **26**

Correct

Mark 0.50 out of 0.50

Mark statements as True/False, w.r.t. the given diagram

True	False			
○ ✗	● ☑	This is a diagram of swtch() called from scheduler()	✔	No. diagram of swtch() called from sched()
○ ✗	● ☑	The diagram is wrong because it shows the user stack and kernel stack together (continuous), but in practice they are separate	✔	diagram shows only kernel stack
● ☑	○ ✗	The blue shaded part in "kernel stack of a process(4k)" refers to remaining part of stack (not used yet)	✔	
● ☑	○ ✗	The "ESP" (second entry from top) is stack pointer of user-stack of process, while the "ESP" (first entry below pink region) is the trapframe pointer on kernel stack of process.	✔	
● ☑	○ ✗	The diagram is correct	✔	
○ ✗	● ☑	The "context" yellow coloured box, pointed to by cpu->scheduler is on the kernel stack of the scheduler.	✔	

This is a diagram of swtch() called from scheduler(): False
The diagram is wrong because it shows the user stack and kernel stack together (continuous), but in practice they are separate: False
The blue shaded part in "kernel stack of a process(4k)" refers to remaining part of stack (not used yet): True
The "ESP" (second entry from top) is stack pointer of user-stack of process, while the "ESP" (first entry below pink region) is the trapframe pointer on kernel stack of process.: True
The diagram is correct: True
The "context" yellow coloured box, pointed to by cpu->scheduler is on the kernel stack of the scheduler.: False

Question **27**

Partially correct

Mark 0.44 out of 0.50

Select the correct statements about sched() and scheduler() in xv6 code

- [x] a. scheduler() switches to the selected process's context ✔
- [x] b. After call to swtch() in scheduler(), the control moves to code in sched() ✔
- [x] c. After call to swtch() in sched(), the control moves to code in scheduler() ✔
- [x] d. When either sched() or scheduler() is called, it results in a context switch ✔
- [x] e. sched() and scheduler() are co-routines ✔
- [ ] f. sched() switches to the scheduler's context
- [x] g. When either sched() or scheduler() is called, it does not return immediately to caller ✔
- [x] h. Each call to sched() or scheduler() involves change of one stack inside swtch() ✔

Your answer is partially correct.

You have correctly selected 7.

The correct answers are: sched() and scheduler() are co-routines, When either sched() or scheduler() is called, it does not return immediately to caller, When either sched() or scheduler() is called, it results in a context switch, sched() switches to the scheduler's context, scheduler() switches to the selected process's context, After call to swtch() in scheduler(), the control moves to code in sched(), After call to swtch() in sched(), the control moves to code in scheduler(), Each call to sched() or scheduler() involves change of one stack inside swtch()

Question **28**

Incorrect

Mark 0.00 out of 0.50

Select the correct statements about paging (not demand paging) mechanism

Select one or more:

- [x] a. Page table is accessed by the OS as part of execuation of an instruction ✖
- [ ] b. User process can update it's own page table entries
- [x] c. An invalid entry on a page means, either it was illegal memory reference or the page was not present in memory. ✖
- [ ] d. The PTBR is loaded by the OS
- [ ] e. User process can update it's own PTBR
- [ ] f. An invalid entry on a page means, it was an illegal memory reference
- [ ] g. Page table is accessed by the MMU as part of execution of an instruction
- [ ] h. OS creates the page table for every process

Your answer is incorrect.

The correct answers are: OS creates the page table for every process, The PTBR is loaded by the OS, Page table is accessed by the MMU as part of execution of an instruction, An invalid entry on a page means, it was an illegal memory reference

Question **29**

Correct

Mark 0.50 out of 0.50

Which of the following is not a task of the code of swtch() function

- [x] a. Jump to next context EIP ✗
- [ ] b. Save the old context
- [ ] c. Load the new context
- [x] d. Change the kernel stack location ✓
- [ ] e. Switch stacks
- [x] f. Save the return value of the old context code ✓

The correct answers are: Save the return value of the old context code, Change the kernel stack location

Question **30**

Correct

Mark 0.50 out of 0.50

Consider the image given below, which explains how paging works.

Figure 9.8 Paging hardware.

Mention whether each statement is True or False, with respect to this image.

True	False		
◉☑	○✗	The page table is indexed using page number	✔
◉☑	○✗	Maximum Size of page table is determined by number of bits used for page number	✔
○✗	◉☑	The page table is indexed using frame number	✔
◉☑	○✗	The physical address may not be of the same size (in bits) as the logical address	✔
○✗	◉☑	Size of page table is always determined by the size of RAM	✔
◉☑	○✗	The page table is itself present in Physical memory	✔
○✗	◉☑	The locating of the page table using PTBR also involves paging translation	✔
◉☑	○✗	The PTBR is present in the CPU as a register	✔

The page table is indexed using page number: True
Maximum Size of page table is determined by number of bits used for page number: True
The page table is indexed using frame number: False
The physical address may not be of the same size (in bits) as the logical address: True
Size of page table is always determined by the size of RAM: False
The page table is itself present in Physical memory: True
The locating of the page table using PTBR also involves paging translation: False
The PTBR is present in the CPU as a register: True

◄ Quiz-1 (15 Marks)

Jump to...

Link to UBuntu VM for this course (VDI File, 15+GB) ►

Started on	Tuesday, 22 March 2022, 2:02:01 PM
State	Finished
Completed on	Tuesday, 22 March 2022, 5:26:37 PM
Time taken	3 hours 24 mins
Grade	**34.52** out of 40.00 (**86**%)

Question **1**

Correct

Mark 2.00 out of 2.00

W.r.t. Memory management in xv6,

xv6 uses physical memory upto 224 MB onlyMark statements True or False

True	False		
✗	✓	The kernel's page table given by kpgdir variable is used as stack for scheduler's context	✓
✓	✗	PHYSTOP can be increased to some extent, simply by editing memlayout.h	✓
✗	✓	The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context	✓
✓	✗	The process's address space gets mapped on frames, obtained from ~2MB:224MB range	✓
✓	✗	The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir	✓
✓	✗	The kernel code and data take up less than 2 MB space	✓
✓	✗	The free page-frame are created out of nearly 222 MB	✓
✗	✓	The switchkvm() call in scheduler() changes CR3 to use page directory of new process	✓
✓	✗	xv6 uses physical memory upto 224 MB only	✓
✓	✗	The stack allocated in entry.S is used as stack for scheduler's context for first processor	✓
✓	✗	The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context	✓

The kernel's page table given by kpgdir variable is used as stack for scheduler's context: False
PHYSTOP can be increased to some extent, simply by editing memlayout.h: True
The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context: False
The process's address space gets mapped on frames, obtained from ~2MB:224MB range: True
The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir: True
The kernel code and data take up less than 2 MB space: True
The free page-frame are created out of nearly 222 MB: True
The switchkvm() call in scheduler() changes CR3 to use page directory of new process: False
xv6 uses physical memory upto 224 MB only: True
The stack allocated in entry.S is used as stack for scheduler's context for first processor: True
The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context: True

Question **2**

Correct

Mark 2.00 out of 2.00

Consider the reference string
6 4 2 0 1 2 6 9 2 0 5

If the number of page frames is 3, then total number of page faults (including initial), using FIFO replacement is:

Answer: 10 ✔

#6# 6,4# 6,4,2 #0,4,2# 0,1,2 #0,1,6 #9,1,6# 9,2,6# 9,2,0 #5,2,0

The correct answer is: 10

Question **3**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about linking and loading.

Select one or more:

- [ ] a. Static linking leads to non-relocatable code
- [ ] b. Loader is last stage of the linker program
- [x] c. Loader is part of the operating system ✔
- [x] d. Continuous memory management schemes can support static linking and static loading. (may be inefficiently) ✔
- [x] e. Dynamic linking and loading is not possible without demand paging or demand segmentation. ✔
- [x] f. Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently) ✔
- [ ] g. Dynamic linking is possible with continous memory management, but variable sized partitions only.
- [x] h. Dynamic linking essentially results in relocatable code. ✔
- [ ] i. Continuous memory management schemes can support dynamic linking and dynamic loading.

Your answer is correct.

The correct answers are: Continuous memory management schemes can support static linking and static loading. (may be inefficiently), Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently), Dynamic linking essentially results in relocatable code., Loader is part of the operating system, Dynamic linking and loading is not possible without demand paging or demand segmentation.

Question **4**

Partially correct

Mark 0.80 out of 1.00

Select all the correct statements w.r.t user and kernel threads

Select one or more:

- [ ] a. one-one model can be implemented even if there are no kernel threads
- [x] b. A process blocks in many-one model even if a single thread makes a blocking system call ✔
- [ ] c. A process may not block in many-one model, if a thread makes a blocking system call
- [ ] d. many-one model can be implemented even if there are no kernel threads
- [x] e. all three models, that is many-one, one-one, many-many , require a user level thread library ✔
- [x] f. many-one model gives no speedup on multicore processors ✔
- [x] g. one-one model increases kernel's scheduling load ✔

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: many-one model can be implemented even if there are no kernel threads, all three models, that is many-one, one-one, many-many , require a user level thread library, one-one model increases kernel's scheduling load, many-one model gives no speedup on multicore processors, A process blocks in many-one model even if a single thread makes a blocking system call

Question **5**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about MMU and it's functionality (on a non-demand paged system)

Select one or more:

- [x] a. Logical to physical address translations in MMU are done in hardware, automatically ✔
- [ ] b. Illegal memory access is detected by operating system
- [x] c. Illegal memory access is detected in hardware by MMU and a trap is raised ✔
- [x] d. MMU is inside the processor ✔
- [ ] e. MMU is a separate chip outside the processor
- [ ] f. The operating system interacts with MMU for every single address translation
- [ ] g. Logical to physical address translations in MMU are done with specific machine instructions
- [x] h. The Operating system sets up relevant CPU registers to enable proper MMU translations ✔

Your answer is correct.

The correct answers are: MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

Question **6**

Incorrect

Mark 0.00 out of 1.00

Consider a computer system with a 32-bit logical address and 4- KB page size. The system supports up to 512 MB of physical memory. How many entries are there in each of the following?

Write answer as a decimal number.

A conventional, single-level page table:

2097152

✖

An inverted page table:

262144

✖

Question **7**

Correct

Mark 1.00 out of 1.00

Figure 9.8 Paging hardware.

Mark the statements as True or False, w.r.t. the above diagram (note that the diagram does not cover all details of what actually happens!)

True	False		
✗	✓	Using the offset d in the physical page-frame is done by MMU	✔
✓	✗	The split of logical address into p and d is done by MMU	✔
✓	✗	The logical address issued by CPU is the same one generated by compiler	✔
✗	✓	There are total 3 memory references in this diagram	✔
✓	✗	The combining of f and d is done by MMU	✔
✓	✗	The page table is in physical memory and must be continuous	✔

Using the offset d in the physical page-frame is done by MMU: False
The split of logical address into p and d is done by MMU: True
The logical address issued by CPU is the same one generated by compiler: True
There are total 3 memory references in this diagram: False
The combining of f and d is done by MMU: True
The page table is in physical memory and must be continuous: True

Question **8**

Partially correct

Mark 0.83 out of 1.00

Select all correct statements w.r.t. Major and Minor page faults on Linux

- [x] a. Minor page faults are an improvement of the page buffering techniques ✔
- [ ] b. Thrashing is possible only due to major page faults
- [x] c. Major page faults are likely to occur in more numbers at the beginning of the process ✔
- [x] d. Minor page fault may occur because the page was a shared memory page ✔
- [x] e. Minor page fault may occur because of a page fault during fork(), on code of an already running process ✔
- [x] f. Minor page fault may occur because the page was freed, but still tagged and available in the free page list ✔

The correct answers are: Minor page fault may occur because the page was a shared memory page, Minor page fault may occur because of a page fault during fork(), on code of an already running process, Minor page fault may occur because the page was freed, but still tagged and available in the free page list, Major page faults are likely to occur in more numbers at the beginning of the process, Thrashing is possible only due to major page faults, Minor page faults are an improvement of the page buffering techniques

Question **9**

Correct

Mark 1.00 out of 1.00

W.r.t. xv6 code, match the state of a process with a code that sets the state

EMBRYO	fork()->allocproc() before setting up the UVM	✔
ZOMBIE	exit(), called by process itself	✔
RUNNING	scheduler()	✔
UNUSED	wait(), called by parent process	✔
RUNNABLE	wakeup(), called by an interrupt handler	✔
SLEEPING	sleep(), called by any process blocking itself	✔

The correct answer is: EMBRYO → fork()->allocproc() before setting up the UVM, ZOMBIE → exit(), called by process itself, RUNNING → scheduler(), UNUSED → wait(), called by parent process, RUNNABLE → wakeup(), called by an interrupt handler, SLEEPING → sleep(), called by any process blocking itself

Question **10**

Partially correct

Mark 0.75 out of 1.00

Select the correct points of comparison between POSIX and System V shared memory.

- [ ] a. System V is more prevalent than POSIX even today
- [x] b. POSIX allows giving name to shared memory, System V does not ✔
- [x] c. POSIX shared memory is "thread safe", System V is not ✔
- [x] d. POSIX shared memory is newer than System V shared memory ✔

The correct answers are: POSIX shared memory is newer than System V shared memory, POSIX shared memory is "thread safe", System V is not, POSIX allows giving name to shared memory, System V does not, System V is more prevalent than POSIX even today

Question **11**

Correct

Mark 1.00 out of 1.00

Map the functionality/use with function/variable in xv6 code.

Array listing the kernel memory mappings, to be used by setupkvm()	kmap[] ✔
Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed	mappages() ✔
return a free page, if available; 0, otherwise	kalloc() ✔
Setup kernel part of a page table, and switch to that page table	kvmalloc() ✔
Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary	walkpgdir() ✔
Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices	setupkvm() ✔

Your answer is correct.

The correct answer is: Array listing the kernel memory mappings, to be used by setupkvm() → kmap[], Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed → mappages(), return a free page, if available; 0, otherwise → kalloc(), Setup kernel part of a page table, and switch to that page table → kvmalloc(), Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary → walkpgdir(), Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices → setupkvm()

Question **12**

Correct

Mark 1.00 out of 1.00

For the reference string
3 4 3 5 2

using LRU replacement policy for pages,

consider the number of page faults for 2, 3 and 4 page frames.
Select the most correct statement.

Select one:

- a. Exhibit Balady's anomaly between 2 and 3 frames
- b. This example does not exhibit Balady's anomaly
- c. LRU will never exhibit Balady's anomaly ✔
- d. Exhibit Balady's anomaly between 3 and 4 frames

Your answer is correct.

The correct answer is: LRU will never exhibit Balady's anomaly

Question **13**

Correct

Mark 1.00 out of 1.00

Given that a kernel has 1000 KB of total memory, and holes of sizes (in that order) 300 KB, 200 KB, 100 KB, 250 KB. For each of the requests on the left side, match it with the chunk chosen using the specified algorithm.

Consider each request as first request.

100 KB, worst fit	300 KB	✔
150 KB, first fit	300 KB	✔
200 KB, first fit	300 KB	✔
150 KB, best fit	200 KB	✔
50 KB, worst fit	300 KB	✔
220 KB, best fit	250 KB	✔

The correct answer is: 100 KB, worst fit → 300 KB, 150 KB, first fit → 300 KB, 200 KB, first fit → 300 KB, 150 KB, best fit → 200 KB, 50 KB, worst fit → 300 KB, 220 KB, best fit → 250 KB

Question **14**

Correct

Mark 1.00 out of 1.00

The data structure used in kalloc() and kfree() in xv6 is

- a. Double linked NULL terminated list
- b. Doubly linked circular list
- c. Singly linked circular list
- d. Singly linked NULL terminated list ✔

Your answer is correct.

The correct answer is: Singly linked NULL terminated list

Question **15**

Correct

Mark 1.00 out of 1.00

Select the correct statements about interrupt handling in xv6 code

- [x] a. On any interrupt/syscall/exception the control first jumps in vectors.S ✔
- [ ] b. The trapframe pointer in struct proc, points to a location on user stack
- [ ] c. On any interrupt/syscall/exception the control first jumps in trapasm.S
- [x] d. The function trap() is the called irrespective of hardware interrupt/system-call/exception ✔
- [x] e. The CS and EIP are changed only after pushing user code's SS,ESP on stack ✔
- [ ] f. xv6 uses the 0x64th entry in IDT for system calls
- [x] g. Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt ✔
- [ ] h. The CS and EIP are changed only immediately on a hardware interrupt
- [ ] i. The function trap() is the called only in case of hardware interrupt
- [x] j. Before going to alltraps, the kernel stack contains upto 5 entries. ✔
- [x] k. All the 256 entries in the IDT are filled ✔
- [x] l. The trapframe pointer in struct proc, points to a location on kernel stack ✔
- [x] m. xv6 uses the 64th entry in IDT for system calls ✔

Your answer is correct.

The correct answers are: All the 256 entries in the IDT are filled, Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt, xv6 uses the 64th entry in IDT for system calls, On any interrupt/syscall/exception the control first jumps in vectors.S, Before going to alltraps, the kernel stack contains upto 5 entries., The trapframe pointer in struct proc, points to a location on kernel stack, The function trap() is the called irrespective of hardware interrupt/system-call/exception, The CS and EIP are changed only after pushing user code's SS,ESP on stack

Question **16**

Correct

Mark 1.00 out of 1.00

If one thread opens a file with read privileges then

Select one:

- a. other threads in the same process can also read from that file ✔
- b. other threads in the another process can also read from that file
- c. none of these
- d. any other thread cannot read from that file

Your answer is correct.

The correct answer is: other threads in the same process can also read from that file

Question **17**

Correct

Mark 1.00 out of 1.00

For each function/code-point, select the status of segmentation setup in xv6

entry.S	gdt setup with 3 entries, at start32 symbol of bootasm.S	✔
after seginit() in main()	gdt setup with 5 entries (0 to 4) on one processor	✔
bootmain()	gdt setup with 3 entries, at start32 symbol of bootasm.S	✔
kvmalloc() in main()	gdt setup with 3 entries, at start32 symbol of bootasm.S	✔
after startothers() in main()	gdt setup with 5 entries (0 to 4) on all processors	✔
bootasm.S	gdt setup with 3 entries, at start32 symbol of bootasm.S	✔

Your answer is correct.

The correct answer is: entry.S → gdt setup with 3 entries, at start32 symbol of bootasm.S, after seginit() in main() → gdt setup with 5 entries (0 to 4) on one processor, bootmain() → gdt setup with 3 entries, at start32 symbol of bootasm.S, kvmalloc() in main() → gdt setup with 3 entries, at start32 symbol of bootasm.S, after startothers() in main() → gdt setup with 5 entries (0 to 4) on all processors, bootasm.S → gdt setup with 3 entries, at start32 symbol of bootasm.S

Question **18**

Correct

Mark 1.00 out of 1.00

The complete range of virtual addresses (after main() in main.c is over), from which the free pages used by kalloc() and kfree() is derived,are:

- a. P2V(end), P2V(PHYSTOP)
- b. end, PHYSTOP
- c. end, P2V(4MB + PHYSTOP)
- d. end, 4MB
- e. end, (4MB + PHYSTOP)
- f. P2V(end), PHYSTOP
- g. end, P2V(PHYSTOP) ✔

Your answer is correct.

The correct answer is: end, P2V(PHYSTOP)

Question **19**

Partially correct

Mark 0.56 out of 1.00

Mark the statements as True or False, w.r.t. mmap()

True	False		
✖	☑	mmap() results in changes to buffer-cache of the kernel.	✖
✖	☑	MAP_SHARED leads to a mapping that is copy-on-write	✖
☑	✖	mmap() can be implemented on both demand paged and non-demand paged systems.	✔
☑	✖	mmap() is a system call	✔
☑	✖	MAP_PRIVATE leads to a mapping that is copy-on-write	✖
☑	✖	mmap() results in changes to page table of a process.	✔
✖	☑	on failure mmap() returns NULL	✔
✖	☑	MAP_FIXED guarantees that the mapping is always done at the specified address	✖
☑	✖	on failure mmap() returns (void *)-1	✔

mmap() results in changes to buffer-cache of the kernel.: False
MAP_SHARED leads to a mapping that is copy-on-write: False
mmap() can be implemented on both demand paged and non-demand paged systems.: True
mmap() is a system call: True
MAP_PRIVATE leads to a mapping that is copy-on-write: True
mmap() results in changes to page table of a process.: True
on failure mmap() returns NULL: False
MAP_FIXED guarantees that the mapping is always done at the specified address: False
on failure mmap() returns (void *)-1: True

Question **20**

Correct

Mark 1.00 out of 1.00

Given below is a sequence of reference bits on pages before the second chance algorithm runs. Before the algorithm runs, the counter is at the page marked (x). Write the sequence of reference bits after the second chance algorithm has executed once. In the answer write PRECISELY one space BETWEEN each number and do not mention (x).

0 0 1(x) 1 0 1 1

Answer: 0 0 0 0 0 1 1 ✔

The correct answer is: 0 0 0 0 0 1 1

Question **21**

Partially correct

Mark 0.75 out of 1.00

Consider a demand-paging system with the following time-measured utilizations:
CPU utilization : 20%
Paging disk: 97.7%
Other I/O devices: 5%

For each of the following, indicate whether it will (or is likely to) improve CPU utilization (even if by a small amount). Explain your answers.

a. Install a faster CPU : No ✔

b. Install a bigger paging disk. : No ✔

c. Increase the degree of multiprogramming. : No ✔

d. Decrease the degree of multiprogramming. : Yes ✔

e. Install more main memory.: Yes ✔

f. Install a faster hard disk or multiple controllers with multiple hard disks. : Yes ✔

g. Add prepaging to the page-fetch algorithms. :
May be ✖

h. Increase the page size. : No ✖

Question **22**

Correct

Mark 1.00 out of 1.00

Suppose a kernel uses a buddy allocator. The smallest chunk that can be allocated is of size 32 bytes. One bit is used to track each such chunk, where 1 means allocated and 0 means free. The chunk looks like this as of now:

10011010

Now, there is a request for a chunk of 50 bytes.

After this allocation, the bitmap, indicating the status of the buddy allocator will be

Answer: 11111010 ✔

The correct answer is: 11111010

Question **23**

Partially correct

Mark 0.80 out of 1.00

Select all the correct statements about process states.

Note that in this question you lose marks for every incorrect choice that you make, proportional to actual number of incorrect choices.

- [x] a. Process state is changed only by interrupt handlers ✖ -> no exit() also changes it, and so does fork() during it's execution
- [x] b. The scheduler can change state of a process from RUNNALBE to RUNNING ✔
- [ ] c. Process state is stored in the processor
- [x] d. Process state is stored in the PCB ✔
- [ ] e. A process becomes ZOMBIE when another process bites into it's memory
- [ ] f. Process state is implemented as a string
- [ ] g. The scheduler can change state of a process from RUNNALBE to RUNNING and vice-versa
- [x] h. Process state can be implemented as just a number ✔
- [x] i. A process becomes ZOMBIE when it calls exit() ✔

Your answer is partially correct.

You have selected too many options.

The correct answers are: Process state is stored in the PCB, Process state can be implemented as just a number, The scheduler can change state of a process from RUNNALBE to RUNNING, A process becomes ZOMBIE when it calls exit()

Question **24**

Correct

Mark 1.00 out of 1.00

Select the most common causes of use of IPC by processes

- [ ] a. More security checks
- [x] b. More modular code ✔
- [ ] c. Get the kernel performance statistics
- [x] d. Sharing of information of common interest ✔
- [x] e. Breaking up a large task into small tasks and speeding up computation, on multiple core machines ✔

The correct answers are: Sharing of information of common interest, Breaking up a large task into small tasks and speeding up computation, on multiple core machines, More modular code

Question **25**

Correct

Mark 1.00 out of 1.00

Mark statements True/False w.r.t. change of states of a process. Note that a statement is true only if the claim and argument both are true.

Reference: The process state diagram (and your understanding of how kernel code works). Note - the diagram does not show zombie state!

Figure 3.2 Diagram of process state.

True	False		
✗	✓	A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.	✔
✗	✓	A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first	✔
✓	✗	A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet	✔
✓	✗	Every forked process has to go through ZOMBIE state, at least for a small duration.	✔
✓	✗	Only a process in READY state is considered by scheduler	✔

A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.: False
A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first: False
A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet: True
Every forked process has to go through ZOMBIE state, at least for a small duration.: True
Only a process in READY state is considered by scheduler: True

Question **26**

Correct

Mark 1.00 out of 1.00

Choice of the global or local replacement strategy is a subjective choice for kernel programmers. There are advantages and disadvantages on either side. Out of the following statements, that advocate either global or local replacement strategy, select those statements that have a logically CONSISTENT argument. (That is any statement that is logically correct about either global or local replacement)

Consistent	Inconsistent		
		Global replacement can be preferred when greater throughput (number of processes completing per unit time) is a concern, because each process tries to complete at the expense of others, thus leading to overall more processes completing (unless thrashing occurs).	✔
		Local replacement can lead to under-utilisation of memory, because a process may not use all the pages allocated to it all the time.	✔
		Local replacement can be preferred when avoiding thrashing is a major concern because with local replacement and minimum number of frames allocated, a process is always able to progress and cascading inter-process page faults are avoided.	✔
		Local replacement results in more predictable per-process completion time because number of page faults can be better predicted.	✔
		Global replacement may give highly variable per process completion time because number of page faults become un-predictable.	✔

Global replacement can be preferred when greater throughput (number of processes completing per unit time) is a concern, because each process tries to complete at the expense of others, thus leading to overall more processes completing (unless thrashing occurs).: Consistent
Local replacement can lead to under-utilisation of memory, because a process may not use all the pages allocated to it all the time.: Consistent
Local replacement can be preferred when avoiding thrashing is a major concern because with local replacement and minimum number of frames allocated, a process is always able to progress and cascading inter-process page faults are avoided.: Consistent
Local replacement results in more predictable per-process completion time because number of page faults can be better predicted.: Consistent
Global replacement may give highly variable per process completion time because number of page faults become un-predictable.: Consistent

Question **27**

Correct

Mark 1.00 out of 1.00

Mark the statements about named and un-named pipes as True or False

True	False		
✗	✓	The buffers for named-pipe are in process-memory while the buffers for the un-named pipe are in kernel memory.	✔
✓	✗	Un-named pipes are inherited by a child process from parent.	✔
✓	✗	Both types of pipes are an extension of the idea of "message passing".	✔
✗	✓	The pipe() system call can be used to create either a named or un-named pipe.	✔
✓	✗	Both types of pipes provide FIFO communication.	✔
✗	✓	A named pipe has a name decided by the kernel.	✔
✓	✗	Named pipes can exist beyond the life-time of processes using them.	✔
✓	✗	Un-named pipes can be used for communication between only "related" processes, if the common ancestor created it.	✔
✓	✗	Named pipe exists as a file	✔
✗	✓	Named pipes can be used for communication between only "related" processes.	✔

The buffers for named-pipe are in process-memory while the buffers for the un-named pipe are in kernel memory.: False
Un-named pipes are inherited by a child process from parent.: True
Both types of pipes are an extension of the idea of "message passing".: True
The pipe() system call can be used to create either a named or un-named pipe.: False
Both types of pipes provide FIFO communication.: True
A named pipe has a name decided by the kernel.: False
Named pipes can exist beyond the life-time of processes using them.: True
Un-named pipes can be used for communication between only "related" processes, if the common ancestor created it.: True
Named pipe exists as a file: True
Named pipes can be used for communication between only "related" processes.: False

Question **28**

Partially correct

Mark 0.80 out of 1.00

Mark the statements as True or False, w.r.t. thrashing

True	False		
✖	☑	mmap() solves the problem of thrashing.	✖
✖	☑	Thrashing can occur even if entire memory is not in use.	✖
☑	✖	Thrashing occurs when the total size of all processe's locality exceeds total memory size.	✔
☑	✖	The working set model is an attempt at approximating the locality of a process.	✔
✖	☑	Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.	✔
✖	☑	Thrashing occurs because some process is doing lot of disk I/O.	✔
☑	✖	Thrashing can be limited if local replacement is used.	✔
☑	✖	Thrashing is particular to demand paging systems, and does not apply to pure paging systems.	✔
☑	✖	During thrashing the CPU is under-utilised as most time is spent in I/O	✔
☑	✖	Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.	✔

mmap() solves the problem of thrashing.: False
Thrashing can occur even if entire memory is not in use.: False
Thrashing occurs when the total size of all processe's locality exceeds total memory size.: True
The working set model is an attempt at approximating the locality of a process.: True
Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.: False
Thrashing occurs because some process is doing lot of disk I/O.: False
Thrashing can be limited if local replacement is used.: True
Thrashing is particular to demand paging systems, and does not apply to pure paging systems.: True
During thrashing the CPU is under-utilised as most time is spent in I/O: True
Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.: True

Question **29**

Correct

Mark 2.00 out of 2.00

For the reference string
3 4 3 5 2

using FIFO replacement policy for pages,

consider the number of page faults for 2, 3 and 4 page frames.
Select the correct statement.

Select one:

- a. Exhibit Balady's anomaly between 3 and 4 frames
- b. Do not exhibit Balady's anomaly ✔
- c. Exhibit Balady's anomaly between 2 and 3 frames

Your answer is correct.

The correct answer is: Do not exhibit Balady's anomaly

Question **30**

Correct

Mark 1.00 out of 1.00

Mark the statements as True or False, w.r.t. passing of arguments to system calls in xv6 code.

True	False		
✗	✓	The arguments to system call are copied to kernel stack in trapasm.S	✔
✗	✓	String arguments are first copied to trapframe and then from trapframe to kernel's other variables.	✔
✓	✗	The arguments to system call originally reside on process stack.	✔
✓	✗	Integer arguments are copied from user memory to kernel memory using argint()	✔
✓	✗	The functions like argint(), argstr() make the system call arguments available in the kernel.	✔
✓	✗	The arguments are accessed in the kernel code using esp on the trapframe.	✔
✗	✓	Integer arguments are stored in eax, ebx, ecx, etc. registers	✔
✓	✗	String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer	✔

The arguments to system call are copied to kernel stack in trapasm.S: False
String arguments are first copied to trapframe and then from trapframe to kernel's other variables.: False
The arguments to system call originally reside on process stack.: True
Integer arguments are copied from user memory to kernel memory using argint(): True
The functions like argint(), argstr() make the system call arguments available in the kernel.: True
The arguments are accessed in the kernel code using esp on the trapframe.: True
Integer arguments are stored in eax, ebx, ecx, etc. registers: False
String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer: True

Question **31**

Correct

Mark 1.00 out of 1.00

Mark whether the given sequence of events is possible or not-possible. Also, select the reason for your answer.

For each sequence it's a not-possible sequence if some important event is not mentioned in the sequence.

Assume that the kernel code is non-interruptible and uniprocessor system.

```
Process P1, user code executing
Timer interrupt
Context changes to kernel context
Generic interrupt handler runs
Generic interrupt handler calls Scheduler
Scheduler selects P2 for execution
After scheduler, Process P2 user code executing
```

This sequence of events is: not-possible ✔

Because

Process P2 has to return from interrupt context before it's user code executes ✔

Question **32**

Partially correct

Mark 1.33 out of 2.00

Match the description of a memory management function with the name of the function that provides it, in xv6

Setup and load the user page table for initcode process	inituvm()	✔
Load contents from ELF into existing pages	loaduvm()	✔
Copy the code pages of a process	copyuvm()	✖
Create a copy of the page table of a process	copyuvm()	✔
Load contents from ELF into pages after allocating the pages first	loaduvm()	✖
Mark the page as in-accessible	clearpteu()	✔
Switch to user page table	switchuvm()	✔
Switch to kernel page table	setupkvm()	✖
setup the kernel part in the page table	setupkvm()	✔

The correct answer is: Setup and load the user page table for initcode process → inituvm(), Load contents from ELF into existing pages → loaduvm(), Copy the code pages of a process → No such function, Create a copy of the page table of a process → copyuvm(), Load contents from ELF into pages after allocating the pages first → No such function, Mark the page as in-accessible → clearpteu(), Switch to user page table → switchuvm(), Switch to kernel page table → switchkvm(), setup the kernel part in the page table → setupkvm()

Question **33**

Partially correct

Mark 0.40 out of 2.00

Order the following events, in the creation of init() process in xv6:

1. ✔ userinit() is called
2. ✔ empty struct proc is obtained for initcode
3. ✖ kernel stack is allocated for initcode process
4. ✖ Stack is allocated for "/init" process
5. ✖ the header of "/init" ELF file is ready by kernel
6. ✖ kernel memory mappings are created for initcode
7. ✖ code is set to start in forkret() when process gets scheduled
8. ✖ trapframe and context pointers are set to proper location
9. ✔ values are set in the trapframe of initcode
10. ✖ initcode process is set to be runnable
11. ✖ trap() runs
12. ✖ initcode is selected by scheduler for execution
13. ✖ name of process "/init" is copied in struct proc
14. ✖ Arguments on setup on process stack for /init
15. ✖ memory mappings are created for "/init" process
16. ✖ page table mappings of 'initcode' are replaced by makpings of 'init'
17. ✖ initcode calls exec system call
18. ✔ function pointer from syscalls[] array is invoked
19. ✖ sys_exec runs
20. ✖ initcode process runs

Your answer is partially correct.

Grading type: Relative to the next item (including last)

Grade details: 4 / 20 = 20%

Here are the scores for each item in this response:

1. 1 / 1 = 100%
2. 1 / 1 = 100%
3. 0 / 1 = 0%
4. 0 / 1 = 0%
5. 0 / 1 = 0%
6. 0 / 1 = 0%
7. 0 / 1 = 0%
8. 0 / 1 = 0%
9. 1 / 1 = 100%
10. 0 / 1 = 0%
11. 0 / 1 = 0%
12. 0 / 1 = 0%
13. 0 / 1 = 0%

14. 0 / 1 = 0%
15. 0 / 1 = 0%
16. 0 / 1 = 0%
17. 0 / 1 = 0%
18. 1 / 1 = 100%
19. 0 / 1 = 0%
20. 0 / 1 = 0%

The correct order for these items is as follows:

1. userinit() is called
2. empty struct proc is obtained for initcode
3. kernel stack is allocated for initcode process
4. trapframe and context pointers are set to proper location
5. code is set to start in forkret() when process gets scheduled
6. kernel memory mappings are created for initcode
7. values are set in the trapframe of initcode
8. initcode process is set to be runnable
9. initcode is selected by scheduler for execution
10. initcode process runs
11. initcode calls exec system call
12. trap() runs
13. function pointer from syscalls[] array is invoked
14. sys_exec runs
15. the header of "/init" ELF file is ready by kernel
16. memory mappings are created for "/init" process
17. Stack is allocated for "/init" process
18. Arguments on setup on process stack for /init
19. name of process "/init" is copied in struct proc
20. page table mappings of 'initcode' are replaced by makpings of 'init'

Question **34**

Correct

Mark 1.00 out of 1.00

After virtual memory is implemented

(select T/F for each of the following)One Program's size can be larger than physical memory size

True	False		
○ ✗	◉ ✓	Virtual addresses become available to executing process	✔
◉ ✓	○ ✗	Code need not be completely in memory	✔
◉ ✓	○ ✗	Cumulative size of all programs can be larger than physical memory size	✔
◉ ✓	○ ✗	Logical address space could be larger than physical address space	✔
○ ✗	◉ ✓	Virtual access to memory is granted to all processes	✔
◉ ✓	○ ✗	Relatively less I/O may be possible during process execution	✔
◉ ✓	○ ✗	One Program's size can be larger than physical memory size	✔

Virtual addresses become available to executing process: False
Code need not be completely in memory: True
Cumulative size of all programs can be larger than physical memory size: True
Logical address space could be larger than physical address space: True
Virtual access to memory is granted to all processes: False
Relatively less I/O may be possible during process execution: True
One Program's size can be larger than physical memory size: True

Question **35**

Partially correct

Mark 0.50 out of 1.00

Select all the correct statements about signals

Select one or more:

- [ ] a. Signals are delivered to a process by another process
- [ ] b. The signal handler code runs in kernel mode of CPU
- [x] c. SIGKILL definitely kills a process because it can't be caught or ignored, and it's default action terminates the process ✔
- [x] d. A signal handler can be invoked asynchronously or synchronously depending on signal type ✔
- [x] e. Signal handlers once replaced can't be restored ✖
- [x] f. Signals are delivered to a process by kernel ✔
- [x] g. SIGKILL definitely kills a process because it's code runs in kernel mode of CPU ✖
- [x] h. The signal handler code runs in user mode of CPU ✔

Your answer is partially correct.

You have selected too many options.
The correct answers are: Signals are delivered to a process by kernel, A signal handler can be invoked asynchronously or synchronously depending on signal type, The signal handler code runs in user mode of CPU, SIGKILL definitely kills a process because it can't be caught or ignored, and it's default action terminates the process

◄ (Optional Assignment) lseek system call in xv6

Jump to...

Feedback on Quiz-2 ►

Started on	Monday, 16 January 2023, 9:00 PM
State	Finished
Completed on	Monday, 16 January 2023, 10:07 PM
Time taken	1 hour 6 mins
Grade	**13.25** out of 15.00 (**88.33**%)

Question **1**

Correct

Mark 1.00 out of 1.00

When you turn your computer ON, you are often shown an option like "Press F9 for boot options". What does this mean?

- a. The choice of booting slowly or fast
- b. The choice of which OS to boot from
- c. The BIOS allows us to choose the boot device, the device from which the boot loader will be loaded ✔
- d. The choice of the boot loader (e.g. GRUB or Windows-Loader)

The correct answer is: The BIOS allows us to choose the boot device, the device from which the boot loader will be loaded

Question **2**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about bootloader.

Every wrong selection will deduct marks proportional to 1/n where n is total wrong choices in the question.

You will get minimum a zero.

- [x] a. Modern Bootloaders often allow configuring the way an OS boots ✔
- [ ] b. Bootloader must be one sector in length
- [ ] c. The bootloader loads the BIOS
- [x] d. Bootloaders allow selection of OS to boot from ✔
- [x] e. LILO is a bootloader ✔

Your answer is correct.

The correct answers are: LILO is a bootloader, Modern Bootloaders often allow configuring the way an OS boots, Bootloaders allow selection of OS to boot from

Question **3**

Correct

Mark 2.00 out of 2.00

What will this program do?

```
int main() {
fork();
execl("/bin/ls", "/bin/ls", NULL);
printf("hello");
}
```

- a. one process will run ls, another will print hello
- b. run ls once
- c. run ls twice and print hello twice
- d. run ls twice and print hello twice, but output will appear in some random order
- e. run ls twice ✔

Your answer is correct.

The correct answer is: run ls twice

Question **4**

Correct

Mark 1.00 out of 1.00

Order the following events in boot process (from 1 onwards)

Login interface	5	✔
Init	4	✔
OS	3	✔
BIOS	1	✔
Shell	6	✔
Boot loader	2	✔

Your answer is correct.

The correct answer is: Login interface → 5, Init → 4, OS → 3, BIOS → 1, Shell → 6, Boot loader → 2

Question **5**

Partially correct

Mark 0.50 out of 1.00

Select the correct statements about hard and soft links

Select one or more:

- [ ] a. Soft link shares the inode of actual file
- [x] b. Deleting a hard link deletes the file, only if link count was 1 ✔
- [ ] c. Soft links increase the link count of the actual file inode
- [x] d. Soft links can span across partitions while hard links can't ✔
- [ ] e. Deleting a soft link deletes only the actual file
- [x] f. Deleting a hard link always deletes the file ✖
- [ ] g. Hard links can span across partitions while soft links can't
- [x] h. Deleting a soft link deletes the link, not the actual file ✔
- [ ] i. Hard links increase the link count of the actual file inode
- [ ] j. Deleting a soft link deletes both the link and the actual file
- [x] k. Hard links share the inode ✔
- [ ] l. Hard links enforce separation of filename from it's metadata in on-disk data structures.

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: Soft links can span across partitions while hard links can't, Hard links increase the link count of the actual file inode, Deleting a soft link deletes the link, not the actual file, Deleting a hard link deletes the file, only if link count was 1, Hard links share the inode, Hard links enforce separation of filename from it's metadata in on-disk data structures.

Question **6**

Partially correct

Mark 1.75 out of 3.00

Select correct statements about mounting

Select one or more:

- [ ] a. Mounting makes all disk partitions available as one name space
- [ ] b. Mounting deletes all data at the mount-point
- [x] c. Even in operating systems with a pluggable kernel module for file systems, the code for mounting any particular file system must be already present in the operating system system kernel ✖
- [x] d. It's possible to mount a partition on one computer, into namespace of another computer. ✔
- [x] e. Mounting is attaching a disk-partition with a filesystem on it, into another file system name-space ✔
- [ ] f. On Linuxes mounting can be done only while booting the OS
- [x] g. The existing name-space at the mount-point is no longer visible after mounting ✔
- [ ] h. The mount point can be a file as well
- [x] i. The mount point must be a directory ✔
- [x] j. In operating systems with a pluggable kernel module for file systems, the code for mounting a particular file system is provided by the module of that file system. ✔

Your answer is partially correct.

You have correctly selected 5.

The correct answers are: Mounting is attaching a disk-partition with a filesystem on it, into another file system name-space, The mount point must be a directory, The existing name-space at the mount-point is no longer visible after mounting, Mounting makes all disk partitions available as one name space, In operating systems with a pluggable kernel module for file systems, the code for mounting a particular file system is provided by the module of that file system., It's possible to mount a partition on one computer, into namespace of another computer.

Question **7**

Correct

Mark 1.00 out of 1.00

Compare multiprogramming with multitasking

- ( ) a. A multitasking system is not necessarily multiprogramming
- (•) b. A multiprogramming system is not necessarily multitasking ✔

The correct answer is: A multiprogramming system is not necessarily multitasking

Question **8**

Correct

Mark 1.00 out of 1.00

Is the terminal a part of the kernel on GNU/Linux systems?

- a. yes
- b. no ✔ wrong

The correct answer is: no

Question **9**

Correct

Mark 1.00 out of 1.00

Consider the following programs

exec1.c

```
#include <unistd.h>
#include <stdio.h>
int main() {
   execl("./exec2", "./exec2", NULL);
}
```

exec2.c

```
#include <unistd.h>
#include <stdio.h>
int main() {
   execl("/bin/ls", "/bin/ls", NULL);
  printf("hello\n");
}
```

Compiled as

```
cc    exec1.c   -o exec1
cc    exec2.c   -o exec2
```

And run as

```
$ ./exec1
```

Explain the output of the above command (./exec1)

Assume that /bin/ls , i.e. the 'ls' program exists.

Select one:

- a. Execution fails as the call to execl() in exec2 fails
- b. Execution fails as the call to execl() in exec1 fails
- c. "ls" runs on current directory ✔
- d. Execution fails as one exec can't invoke another exec
- e. Program prints hello

Your answer is correct.

The correct answer is: "ls" runs on current directory

Question **10**

Correct

Mark 1.00 out of 1.00

A process blocks itself means

- a. The kernel code of an interrupt handler, moves the process to a waiting queue and calls scheduler
- b. The kernel code of system call calls scheduler
- c. The application code calls the scheduler
- d. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler ✔

The correct answer is: The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

Question **11**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about two modes of CPU operation

Select one or more:

- [x] a. Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode ✔
- [x] b. The two modes are essential for a multitasking system ✔
- [x] c. There is an instruction like 'iret' to return from kernel mode to user mode ✔
- [x] d. The two modes are essential for a multiprogramming system ✔
- [x] e. The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously ✔

Your answer is correct.

The correct answers are: The two modes are essential for a multiprogramming system, The two modes are essential for a multitasking system, There is an instruction like 'iret' to return from kernel mode to user mode, The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously, Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode

Question **12**

Correct

Mark 1.00 out of 1.00

which of the following is not a difference between real mode and protected mode

- a. in real mode the addressable memory is less than in protected mode
- b. in real mode the addressable memory is more than in protected mode ✔
- c. processor starts in real mode
- d. in real mode general purpose registers are 16 bit, in protected mode they are 32 bit
- e. in real mode the segment is multiplied by 16, in protected mode segment is used as index in GDT

The correct answer is: in real mode the addressable memory is more than in protected mode

◄ Random Quiz - 1 (Pre-Requisite Quiz)

Jump to...

Random Quiz - 3 (processes, memory management, event driven kernel), compilation-linking-loading, ipc-pipes ►

Started on	Thursday, 2 February 2023, 9:01 PM
State	Finished
Completed on	Thursday, 2 February 2023, 10:55 PM
Time taken	1 hour 53 mins
Grade	**16.46** out of 20.00 (**82.28**%)

Question **1**

Complete

Mark 1.00 out of 1.00

Which of the following are NOT a part of job of a typical compiler?

- [x] a. Check the program for logical errors
- [ ] b. Convert high level langauge code to machine code
- [ ] c. Invoke the linker to link the function calls with their code, extern globals with their declaration
- [x] d. Suggest alternative pieces of code that can be written
- [ ] e. Check the program for syntactical errors
- [ ] f. Process the # directives in a C program

The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

Question **2**

Complete

Mark 1.00 out of 1.00

Consider the following code and MAP the file to which each fd points at the end of the code.

```
int main(int argc, char *argv[]) {
   int fd1, fd2 = 1, fd3 = 1, fd4 = 1;

   fd1 = open("/tmp/1", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
   fd2 = open("/tmp/2", O_RDDONLY);
   fd3 = open("/tmp/3", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
   close(0);
   close(1);
   dup(fd2);
   dup(fd3);
   close(fd3);
   dup2(fd2, fd4);
   printf("%d %d %d %d\n", fd1, fd2, fd3, fd4);
   return 0;
}
```

fd4	/tmp/2
fd2	/tmp/2
1	/tmp/3
fd1	/tmp/1
fd3	closed
0	/tmp/2
2	stderr

The correct answer is: fd4 → /tmp/2, fd2 → /tmp/2, 1 → /tmp/3, fd1 → /tmp/1, fd3 → closed, 0 → /tmp/2, 2 → stderr

Question **3**

Complete

Mark 0.75 out of 1.00

Select the sequence of events that are NOT possible, assuming an interruptible kernel code

Select one or more:

- [ ] a. P1 running
  keyboard hardware interrupt
  keyboard interrupt handler running
  interrupt handler returns
  P1 running
  P1 makes sytem call
  system call returns
  P1 running
  timer interrupt
  scheduler
  P2 running
- [ ] b. P1 running
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P3 running
  Hardware interrupt
  Interrupt unblocks P1
  Interrupt returns
  P3 running
  Timer interrupt
  Scheduler
  P1 running
- [ ] c. P1 running
  P1 makes system call
  system call returns
  P1 running
  timer interrupt
  Scheduler running
  P2 running
- [x] d. P1 running
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again
- [x] e.
  P1 running
  P1 makes sytem call
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again

f. P1 running
P1 makes system call
timer interrupt
Scheduler
P2 running
timer interrupt
Scheuler
P1 running
P1's system call return

The correct answers are: P1 running
P1 makes sytem call and blocks
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again,
P1 running
P1 makes sytem call
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

Question **4**

Complete

Mark 0.00 out of 1.00

Select all the correct statements about named pipes and ordinary(unnamed) pipe

Select one or more:

- [x] a. named pipes can be used between multiple processes but ordinary pipes can not be used
- [x] b. named pipe can be used between any processes
- [ ] c. both named and unnamed pipes require some kind of agreed protocol to be effectively used among multiple processes
- [x] d. named pipes are more efficient than ordinary pipes
- [x] e. ordinary pipe can only be used between related processes
- [x] f. a named pipe exists as a file on the file system
- [ ] g. named pipe exists even if the processes using it do exit()

The correct answers are: ordinary pipe can only be used between related processes, named pipe can be used between any processes, a named pipe exists as a file on the file system, named pipe exists even if the processes using it do exit(), both named and unnamed pipes require some kind of agreed protocol to be effectively used among multiple processes

Question **5**

Complete

Mark 4.50 out of 5.00

Following code claims to implement the command

/bin/ls -l | /usr/bin/head -3 | /usr/bin/tail -1

Fill in the blanks to make the code work.

Note: Do not include space in writing any option. x[1][2] should be written without any space, and so is the case with [1] or [2]. Pay attention to exact syntax and do not write any extra character like ';' or = etc.

```
int main(int argc, char *argv[]) {
   int pid1, pid2;
   int pfd[
2
][2];

   pipe(
pfd[0]
);
   pid1 =
fork()
;
   if(pid1 != 0) {
      close(pfd[0]
[0]
);
      close(
1
);
      dup(
pfd[0][1]
);
      execl("/bin/ls", "/bin/ls", "
-l
", NULL);
   }
   pipe(
pfd[1]
);

pid2
= fork();
   if(pid2 == 0) {
      close(
pfd[0][1]
;
```

```
        close(0);
        dup(
[ pfd[1][0] ]
);
        close(pfd[1]
[ [0] ]
);
        close(
[ 1 ]
);
        dup(
[ pfd[1][1] ]
);
        execl("/usr/bin/head", "/usr/bin/head", "
[ -3 ]
", NULL);
    } else {
        close(pfd
[ [1][1] ]
);
        close(
[ 0 ]
);
        dup(
[ pfd[1][0] ]
);
        close(pfd
[ [0][0] ]
);
        execl("/usr/bin/tail", "/usr/bin/tail", "
[ -1 ]
", NULL);
  }
}
```

Question **6**

Complete

Mark 1.00 out of 1.00

Select the compiler's view of the process's address space, for each of the following MMU schemes:
(Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

Paging	one continuous chunk
Segmentation, then paging	many continuous chunks of variable size
Segmentation	many continuous chunks of variable size
Relocation + Limit	one continuous chunk

The correct answer is: Paging → one continuous chunk, Segmentation, then paging → many continuous chunks of variable size, Segmentation → many continuous chunks of variable size, Relocation + Limit → one continuous chunk

Question **7**

Complete

Mark 1.40 out of 2.00

Match the elements of C program to their place in memory

Local Variables	Stack
Malloced Memory	Heap
Global Static variables	Data
Global variables	Stack
Code of main()	Code
Local Static variables	Data
Function code	Data
Arguments	Stack
#define MACROS	No memory needed
#include files	No memory needed

The correct answer is: Local Variables → Stack, Malloced Memory → Heap, Global Static variables → Data, Global variables → Data, Code of main() → Code, Local Static variables → Data, Function code → Code, Arguments → Stack, #define MACROS → No Memory needed, #include files → No memory needed

Question **8**

Complete

Mark 0.63 out of 1.00

Consider the image given below, which explains how paging works.

Figure 9.8 Paging hardware.

Mention whether each statement is True or False, with respect to this image.

True	False	
◉	○	The physical address may not be of the same size (in bits) as the logical address
◉	○	The locating of the page table using PTBR also involves paging translation
◉	○	Size of page table is always determined by the size of RAM
○	◉	The page table is indexed using frame number
◉	○	The PTBR is present in the CPU as a register
◉	○	The page table is indexed using page number
○	◉	Maximum Size of page table is determined by number of bits used for page number
◉	○	The page table is itself present in Physical memory

The physical address may not be of the same size (in bits) as the logical address: True
The locating of the page table using PTBR also involves paging translation: False
Size of page table is always determined by the size of RAM: False
The page table is indexed using frame number: False
The PTBR is present in the CPU as a register: True
The page table is indexed using page number: True
Maximum Size of page table is determined by number of bits used for page number: True
The page table is itself present in Physical memory: True

Question **9**

Complete

Mark 0.67 out of 1.00

Select the sequence of events that are NOT possible, assuming a non-interruptible kernel code

(Note: non-interruptible kernel code means, if the kernel code is executing, then interrupts will be disabled).

Note: A possible sequence may have some missing steps in between. An impossible sequence will will have n and n+1th steps such that n+1th step can not follow n'th step.

Select one or more:

- [ ] a. P1 running
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P3 running
  Hardware interrupt
  Interrupt unblocks P1
  Interrupt returns
  P3 running
  Timer interrupt
  Scheduler
  P1 running
- [ ] b. P1 running
  P1 makes system call
  system call returns
  P1 running
  timer interrupt
  Scheduler running
  P2 running
- [x] c. P1 running
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again
- [ ] d. P1 running
  keyboard hardware interrupt
  keyboard interrupt handler running
  interrupt handler returns
  P1 running
  P1 makes sytem call
  system call returns
  P1 running
  timer interrupt
  scheduler
  P2 running
- [x] e. P1 running
  P1 makes system call
  timer interrupt
  Scheduler
  P2 running

timer interrupt
Scheuler
P1 running
P1's system call return

f.

P1 running
P1 makes sytem call
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

The correct answers are: P1 running
P1 makes sytem call and blocks
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again, P1 running
P1 makes system call
timer interrupt
Scheduler
P2 running
timer interrupt
Scheuler
P1 running
P1's system call return,
P1 running
P1 makes sytem call
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

Question **10**

Complete

Mark 1.00 out of 1.00

Consider the two programs given below to implement the command (ignore the fact that error checks are not done on return values of functions)

$ ls . /tmp/asdfksdf >/tmp/ddd 2>&1

Program 1

```
int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(1);
    dup(fd);
    close(2);
    dup(fd);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Program 2

```
int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    close(1);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(2);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Select all the correct statements about the programs

Select one or more:

- [ ] a. Both programs are correct
- [ ] b. Program 2 is correct for > /tmp/ddd but not for 2>&1
- [ ] c. Program 1 is correct for > /tmp/ddd but not for 2>&1
- [ ] d. Program 2 ensures 2>&1 and does not ensure > /tmp/ddd
- [ ] e. Program 2 makes sure that there is one file offset used for '2' and '1'
- [ ] f. Program 1 ensures 2>&1 and does not ensure > /tmp/ddd
- [ ] g. Only Program 2 is correct
- [ ] h. Program 1 does 1>&2
- [x] i. Program 1 makes sure that there is one file offset used for '2' and '1'
- [ ] j. Program 2 does 1>&2
- [ ] k. Both program 1 and 2 are incorrect
- [x] l. Only Program 1 is correct

The correct answers are: Only Program 1 is correct, Program 1 makes sure that there is one file offset used for '2' and '1'

Question **11**

Complete

Mark 0.71 out of 1.00

Order the events that occur on a timer interrupt:

Jump to a code pointed by IDT	2
Save the context of the currently running process	3
Execute the code of the new process	6
Jump to scheduler code	4
Change to kernel stack of currently running process	1
Select another process for execution	5
Set the context of the new process	7

The correct answer is: Jump to a code pointed by IDT → 2, Save the context of the currently running process → 3, Execute the code of the new process → 7, Jump to scheduler code → 4, Change to kernel stack of currently running process → 1, Select another process for execution → 5, Set the context of the new process → 6

Question **12**

Complete

Mark 0.80 out of 1.00

Select all the correct statements about zombie processes

Select one or more:

- [ ] a. A process becomes zombie when it's parent finishes
- [x] b. A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it
- [x] c. If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent
- [ ] d. Zombie processes are harmless even if OS is up for long time
- [x] e. A process can become zombie if it finishes, but the parent has finished before it
- [ ] f. A zombie process occupies space in OS data structures
- [x] g. init() typically keeps calling wait() for zombie processes to get cleaned up
- [ ] h. A zombie process remains zombie forever, as there is no way to clean it up

The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it, A process can become zombie if it finishes, but the parent has finished before it, A zombie process occupies space in OS data structures, If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

Question **13**

Complete

Mark 1.00 out of 1.00

Select the state that is not possible after the given state, for a process:

New: Running

Ready : Waiting

Running: : None of these

Waiting: Running

Question **14**

Complete

Mark 1.00 out of 1.00

Select the order in which the various stages of a compiler execute.

Pre-processing	1
Intermediate code generation	3
Linking	4
Syntatical Analysis	2
Loading	does not exist

The correct answer is: Pre-processing → 1, Intermediate code generation → 3, Linking → 4, Syntatical Analysis → 2, Loading → does not exist

Question **15**

Complete

Mark 1.00 out of 1.00

A process blocks itself means

a. The application code calls the scheduler
b. The kernel code of system call calls scheduler
c. The kernel code of an interrupt handler, moves the process to a waiting queue and calls scheduler
d. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

The correct answer is: The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

◄ Random Quiz - 2: bootloader, system calls, fork-exec, open-read-write, linux-basics, processes

Jump to...

Random Quiz 4 : Scheduling, signals, segmentation, paging, compilation, process-state ►

Started on	Thursday, 9 March 2023, 6:20 PM
State	Finished
Completed on	Thursday, 9 March 2023, 7:30 PM
Time taken	1 hour 10 mins
Overdue	14 mins
Grade	**5.55** out of 10.00 (**55.53**%)

Question **1**

Partially correct

Mark 0.18 out of 1.00

W.r.t. Memory management in xv6,

xv6 uses physical memory upto 224 MB onlyMark statements True or False

True	False		
☐ ☑	☐ ✗	xv6 uses physical memory upto 224 MB only	✗
☐ ☑	☐ ✗	The stack allocated in entry.S is used as stack for scheduler's context for first processor	✗
☐ ☑	☐ ✗	The free page-frame are created out of nearly 222 MB	✗
☐ ☑	☐ ✗	The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context	✗
☐ ☑	☐ ✗	PHYSTOP can be increased to some extent, simply by editing memlayout.h	✗
☐ ✗	◉ ☑	The switchkvm() call in scheduler() changes CR3 to use page directory of new process	✔
☐ ☑	☐ ✗	The process's address space gets mapped on frames, obtained from ~2MB:224MB range	✗
◉ ✗	☐ ☑	The kernel's page table given by kpgdir variable is used as stack for scheduler's context	✗
☐ ☑	☐ ✗	The kernel code and data take up less than 2 MB space	✗
☐ ✗	◉ ☑	The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context	✔
☐ ☑	☐ ✗	The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir	✗

xv6 uses physical memory upto 224 MB only: True
The stack allocated in entry.S is used as stack for scheduler's context for first processor: True
The free page-frame are created out of nearly 222 MB: True
The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context: True
PHYSTOP can be increased to some extent, simply by editing memlayout.h: True
The switchkvm() call in scheduler() changes CR3 to use page directory of new process: False
The process's address space gets mapped on frames, obtained from ~2MB:224MB range: True
The kernel's page table given by kpgdir variable is used as stack for scheduler's context: False
The kernel code and data take up less than 2 MB space: True

The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context: False
The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir: True

Question **2**

Correct

Mark 1.00 out of 1.00

What's the trapframe in xv6?

- a. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S ✔
- b. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by code in trapasm.S only
- c. The IDT table
- d. A frame of memory that contains all the trap handler's addresses
- e. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware only
- f. A frame of memory that contains all the trap handler code
- g. A frame of memory that contains all the trap handler code's function pointers

Your answer is correct.

The correct answer is: The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S

Question **3**

Partially correct

Mark 0.75 out of 1.00

Select the correct statements about interrupt handling in xv6 code

- [x] a. The trapframe pointer in struct proc, points to a location on process's kernel stack ✔
- [x] b. The CS and EIP are changed only after pushing user code's SS,ESP on stack ✔
- [ ] c. On any interrupt/syscall/exception the control first jumps in trapasm.S
- [x] d. All the 256 entries in the IDT are filled in xv6 code ✔
- [ ] e. Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt
- [ ] f. Before going to alltraps, the kernel stack contains upto 5 entries.
- [ ] g. The trapframe pointer in struct proc, points to a location on user stack
- [ ] h. The function trap() is the called only in case of hardware interrupt
- [ ] i. The CS and EIP are changed immediately (as the first thing) on a hardware interrupt
- [ ] j. xv6 uses the 0x64th entry in IDT for system calls
- [x] k. xv6 uses the 64th entry in IDT for system calls ✔
- [x] l. On any interrupt/syscall/exception the control first jumps in vectors.S ✔
- [x] m. The function trap() is the called even if any of the hardware interrupt/system-call/exception occurs ✔

Your answer is partially correct.

You have correctly selected 6.

The correct answers are: All the 256 entries in the IDT are filled in xv6 code, Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt, xv6 uses the 64th entry in IDT for system calls, On any interrupt/syscall/exception the control first jumps in vectors.S, Before going to alltraps, the kernel stack contains upto 5 entries., The trapframe pointer in struct proc, points to a location on process's kernel stack, The function trap() is the called even if any of the hardware interrupt/system-call/exception occurs, The CS and EIP are changed only after pushing user code's SS,ESP on stack

Question **4**

Partially correct

Mark 0.55 out of 1.00

Consider the following command and it's output:

```
$ ls -lht xv6.img kernel
-rw-rw-r-- 1 abhijit abhijit 4.9M Feb 15 11:09 xv6.img
-rwxrwxr-x 1 abhijit abhijit 209K Feb 15 11:09 kernel*
```

Following code in bootmain()

```
  readseg((uchar*)elf, 4096, 0);
```

and following selected lines from Makefile

```
xv6.img: bootblock kernel
    dd if=/dev/zero of=xv6.img count=10000
    dd if=bootblock of=xv6.img conv=notrunc
    dd if=kernel of=xv6.img seek=1 conv=notrunc

kernel: $(OBJS) entry.o entryother initcode kernel.ld
    $(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode entryother
    $(OBJDUMP) -S kernel > kernel.asm
    $(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > kernel.sym
```

Also read the code of bootmain() in xv6 kernel.
Select the options that describe the meaning of these lines and their correlation.

- [ ] a. Althought the size of the kernel file is 209 Kb, only 4Kb out of it is the actual kernel code and remaining part is all zeroes.
- [x] b. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is not read as it is user programs. ✖
- [x] c. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain(). ✔
- [ ] d. The kernel.asm file is the final kernel file
- [ ] e. Althought the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.
- [x] f. The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files ✔
- [x] g. The kernel.ld file contains instructions to the linker to link the kernel properly ✔
- [x] h. readseg() reads first 4k bytes of kernel in memory ✔
- [ ] i. The bootmain() code does not read the kernel completely in memory

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain()., readseg() reads first 4k bytes of kernel in memory, The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files, The kernel.ld file contains instructions to the linker to link the kernel properly, Althought the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.

Question **5**

Correct

Mark 1.00 out of 1.00

For each function/code-point, select the status of segmentation setup in xv6

bootmain()	gdt setup with 3 entries, at start32 symbol of bootasm.S ✔
kvmalloc() in main()	gdt setup with 3 entries, at start32 symbol of bootasm.S ✔
entry.S	gdt setup with 3 entries, at start32 symbol of bootasm.S ✔
bootasm.S	gdt setup with 3 entries, at start32 symbol of bootasm.S ✔
after startothers() in main()	gdt setup with 5 entries (0 to 4) on all processors ✔
after seginit() in main()	gdt setup with 5 entries (0 to 4) on one processor ✔

Your answer is correct.

The correct answer is: bootmain() → gdt setup with 3 entries, at start32 symbol of bootasm.S, kvmalloc() in main() → gdt setup with 3 entries, at start32 symbol of bootasm.S, entry.S → gdt setup with 3 entries, at start32 symbol of bootasm.S, bootasm.S → gdt setup with 3 entries, at start32 symbol of bootasm.S, after startothers() in main() → gdt setup with 5 entries (0 to 4) on all processors, after seginit() in main() → gdt setup with 5 entries (0 to 4) on one processor

Question **6**

Correct

Mark 1.00 out of 1.00

Some part of the bootloader of xv6 is written in assembly while some part is written in C. Why is that so?
Select all the appropriate choices

- [ ] a. The code in assembly is required for transition to protected mode, from real mode; but calling convention was applicable all the time
- [x] b. The setting up of the most essential memory management infrastructure needs assembly code ✔
- [ ] c. The code for reading ELF file can not be written in assembly
- [x] d. The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C ✔

Your answer is correct.

The correct answers are: The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C, The setting up of the most essential memory management infrastructure needs assembly code

Question **7**

Incorrect

Mark 0.00 out of 1.00

xv6.img: bootblock kernel

```
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
```

Consider above lines from the Makefile. Which of the following is INCORRECT?

- [x] a. The kernel is located at block-1 of the xv6.img ✗
- [x] b. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk. ✓
- [x] c. The size of the kernel file is nearly 5 MB ✓
- [x] d. The bootblock is located on block-0 of the xv6.img ✗
- [x] e. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies 10,000 blocks on the disk. ✗
- [x] f. Blocks in xv6.img after kernel may be all zeroes. ✗
- [ ] g. xv6.img is the virtual processor used by the qemu emulator
- [x] h. The size of the xv6.img is nearly 5 MB ✗
- [x] i. The xv6.img is the virtual disk that is created by combining the bootblock and the kernel file. ✗
- [x] j. The bootblock may be 512 bytes or less (looking at the Makefile instruction) ✗
- [ ] k. The size of xv6.img is exactly = (size of bootblock) + (size of kernel)

Your answer is incorrect.

The correct answers are: xv6.img is the virtual processor used by the qemu emulator, The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk., The size of the kernel file is nearly 5 MB, The size of xv6.img is exactly = (size of bootblock) + (size of kernel)

Question **8**

Partially correct

Mark 0.07 out of 1.00

Select all the correct statements about code of bootmain() in xv6

```
void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* pa;

  elf = (struct elfhdr*)0x10000;  // scratch space

  // Read 1st page off disk
  readseg((uchar*)elf, 4096, 0);

  // Is this an ELF executable?
  if(elf->magic != ELF_MAGIC)
    return;  // let bootasm.S handle error

  // Load each program segment (ignores ph flags).
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
  for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
  }

  // Call the entry point from the ELF header.
  // Does not return!
  entry = (void(*)(void))(elf->entry);
  entry();
}
```

Also, inspect the relevant parts of the xv6 code. binary files, etc and run commands as you deem fit to answer this question.

- [x] a. The kernel file has only two program headers ✔
- [x] b. The kernel file gets loaded at the Physical address 0x10000 in memory. ✔
- [x] c. The elf->entry is set by the linker in the kernel file and it's 0x80000000 ✖
- [ ] d. The readseg finally invokes the disk I/O code using assembly instructions
- [ ] e. The elf->entry is set by the linker in the kernel file and it's 8010000c
- [ ] f. The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded
- [x] g. The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it. ✔
- [x] h. The elf->entry is set by the linker in the kernel file and it's 0x80000000 ✖
- [ ] i. The kernel file gets loaded at the Physical address 0x10000 +0x80000000 in memory.

- [ ] j. The condition if(ph->memsz > ph->filesz) is never true.
- [x] k. The stosb() is used here, to fill in some space in memory with zeroes ✔

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: The kernel file gets loaded at the Physical address 0x10000 in memory., The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it., The elf->entry is set by the linker in the kernel file and it's 8010000c, The readseg finally invokes the disk I/O code using assembly instructions, The stosb() is used here, to fill in some space in memory with zeroes, The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded, The kernel file has only two program headers

Question **9**

Correct

Mark 1.00 out of 1.00

In bootasm.S, on the line

```
ljmp    $(SEG_KCODE<<3), $start32
```

The SEG_KCODE << 3, that is shifting of 1 by 3 bits is done because

- a. While indexing the GDT using CS, the value in CS is always divided by 8
- b. The code segment is 16 bit and only upper 13 bits are used for segment number ✔
- c. The value 8 is stored in code segment
- d. The code segment is 16 bit and only lower 13 bits are used for segment number
- e. The ljmp instruction does a divide by 8 on the first argument

Your answer is correct.

The correct answer is: The code segment is 16 bit and only upper 13 bits are used for segment number

Question **10**

Incorrect

Mark 0.00 out of 1.00

For each line of code mentioned on the left side, select the location of sp/esp that is in use

ljmp $(SEG_KCODE<<3), $start32 in bootasm.S	0x7c00 to 0	✖
readseg((uchar*)elf, 4096, 0); in bootmain.c	Immaterial as the stack is not used here	✖
jmp *%eax in entry.S	0x7c00 to 0	✖
cli in bootasm.S	Choose...	
call bootmain in bootasm.S	Immaterial as the stack is not used here	✖

Your answer is incorrect.

The correct answer is: ljmp $(SEG_KCODE<<3), $start32 in bootasm.S → Immaterial as the stack is not used here, readseg((uchar*)elf, 4096, 0); in bootmain.c → 0x7c00 to 0, jmp *%eax in entry.S → The 4KB area in kernel image, loaded in memory, named as 'stack', cli in bootasm.S → Immaterial as the stack is not used here, call bootmain in bootasm.S → 0x7c00 to 0

◄ Random Quiz 4 : Scheduling, signals, segmentation, paging, compilation, process-state

Jump to...

Random Quiz - 6 (xv6 file system) ►

Started on	Friday, 31 March 2023, 6:18 PM
State	Finished
Completed on	Friday, 31 March 2023, 7:00 PM
Time taken	41 mins 52 secs
Grade	**4.92** out of 15.00 (**32.83**%)

Question **1**

Partially correct

Mark 0.67 out of 1.00

Select all the actions taken by iget()

- [x] a. Returns an inode with given dev+inode-number from cache, if it exists in cache ✔
- [x] b. Returns the inode with reference count incremented ✔
- [ ] c. Returns a free-inode , with dev+inode-number set, if not found in cache
- [ ] d. Panics if inode does not exist in cache
- [ ] e. Returns a valid inode if not found in cache
- [ ] f. Returns the inode with inode-cache lock held
- [ ] g. Returns the inode locked

Your answer is partially correct.

You have correctly selected 2.

The correct answers are: Returns an inode with given dev+inode-number from cache, if it exists in cache, Returns the inode with reference count incremented, Returns a free-inode , with dev+inode-number set, if not found in cache

Question **2**

Incorrect

Mark 0.00 out of 1.00

Note: for this question you get full marks if you select all and only correct options, you get ZERO if at least one option is wrong or not selected.

Select all the correct statements about log structured file systems.

- [x] a. ext4 is a log structured file system ✖ it's a journaled file system, not log structured
- [x] b. file system recovery recovers all the lost data ✖
- [x] c. log structured file systems considerably improve the recovery time ✔
- [ ] d. xv6 has a log structured file system
- [x] e. ext2 is by default a log structured file system ✖

Your answer is incorrect.

The correct answers are: xv6 has a log structured file system, log structured file systems considerably improve the recovery time

Question **3**

Partially correct

Mark 0.86 out of 2.00

Select T/F w.r.t physical disk handling in xv6 code

True	False		
✗	✓	device files are not supported	✗
✓	✗	The code supports IDE, and not SATA/SCSI	✗
✓	✗	log is kept on the same device as the file system	✓
✓	✗	disk driver handles only one buffer at a time	✓
✓	✗	the superblock does not contain number of free blocks	✓
✗	✓	only direct blocks are supported	✗
✓	✗	only 2 disks are handled by default	✗

device files are not supported: False
The code supports IDE, and not SATA/SCSI: True
log is kept on the same device as the file system: True
disk driver handles only one buffer at a time: True
the superblock does not contain number of free blocks: True
only direct blocks are supported: False
only 2 disks are handled by default: True

Question **4**

Partially correct

Mark 1.00 out of 2.00

Marks the statements as True/False w.r.t. "struct buf"

True	False			
✗	✓	B_VALID means the buffer is empty and can be reused	✓	No. it means it contains data, same as the data on disk
✓	✗	Lock on a buffer is acquired in bget, and released in brelse	✗	
✓	✗	The reference count (refcnt) in struct buf is = number of processes accessing the buffer	✓	
✗	✓	A buffer can have both B_VALID and B_DIRTY flags set	✗	only one will be set
✗	✓	The "next" pointer chain gives the buffers in LRU order	✗	No. MRU order.
✓	✗	A buffer can be both on the MRU/LRU list and also on idequeue list.	✗	
✓	✗	The buffers are maintained in LRU order, in the function brelse	✓	
✓	✗	B_DIRTY flag means the buffer contains modified data	✓	

B_VALID means the buffer is empty and can be reused: False
Lock on a buffer is acquired in bget, and released in brelse: True
The reference count (refcnt) in struct buf is = number of processes accessing the buffer: True
A buffer can have both B_VALID and B_DIRTY flags set: False
The "next" pointer chain gives the buffers in LRU order: False
A buffer can be both on the MRU/LRU list and also on idequeue list.: True
The buffers are maintained in LRU order, in the function brelse: True
B_DIRTY flag means the buffer contains modified data: True

Question **5**

Partially correct

Mark 0.40 out of 1.00

Arrange the following in their typical order of use in xv6.

1. ✔ iget
2. ✖ ilock
3. ✔ iunlock
4. ✖ iput
5. ✖ use inode

Your answer is partially correct.

Grading type: Relative to the next item (including last)

Grade details: 2 / 5 = 40%

Here are the scores for each item in this response:

1. 1 / 1 = 100%
2. 0 / 1 = 0%
3. 1 / 1 = 100%
4. 0 / 1 = 0%
5. 0 / 1 = 0%

The correct order for these items is as follows:

1. iget
2. ilock
3. use inode
4. iunlock
5. iput

Question **6**

Partially correct

Mark 0.50 out of 1.00

Compare XV6 and EXT2 file systems.

Select True/False for each point.

True	False		
✖	☑	In both ext2 and xv6, the superblock gives location of first inode block	✖
☑	✖	xv6 contains journal, ext2 does not	✖
✖	☑	Ext2 contains superblock but xv6 does not.	✔
✖	☑	xv6 contains inode bitmap, but ext2 does not	✔
✖	☑	Both xv6 and ext2 contain magic number	✖
☑	✖	Ext2 contains group descriptors but xv6 does not	✔

In both ext2 and xv6, the superblock gives location of first inode block: False
xv6 contains journal, ext2 does not: True
Ext2 contains superblock but xv6 does not.: False
xv6 contains inode bitmap, but ext2 does not: False
Both xv6 and ext2 contain magic number: False
Ext2 contains group descriptors but xv6 does not: True

Question **7**

Incorrect

Mark 0.00 out of 1.00

Maximum size of a file on xv6 in **bytes** is

(just write a numeric answer)

Answer: 512 ✖

The correct answer is: 71680

Question **8**

Partially correct

Mark 0.17 out of 1.00

Select all the actions taken by ilock()

- [ ] a. Mark the in-memory inode as valid, if needed
- [x] b. Read the inode from disk, if needed ✔
- [ ] c. Take the sleeplock on the inode, always
- [ ] d. Get the inode from the inode-cache
- [x] e. Copy the on-disk inode into in-memory inode, if neeed ✔
- [x] f. Take the sleeplock on the inode, optionally ✖
- [ ] g. Lock all the buffers of the file in memory

Your answer is partially correct.

You have correctly selected 2.

The correct answers are: Read the inode from disk, if needed, Copy the on-disk inode into in-memory inode, if neeed, Take the sleeplock on the inode, always, Mark the in-memory inode as valid, if needed

Question **9**

Incorrect

Mark 0.00 out of 1.00

The lines

```
if(ip->type != T_DIR){
  iunlockput(ip);
  return 0;
}
```

in namex() function

mean

- [ ] a. The last path component (which is a file, and not a directory) has been resolved, so release the lock (using iunlockput) and return
- [x] b. One of the sub-components on the given path name, was a directory, but it was not supposed to be a directory, hence an error ✖
- [ ] c. There was a syntax error in the pathname specified
- [ ] d. One of the sub-components on the given path name, was not a directory, hence it's an error
- [x] e. One of the sub-components on the given path name, did not exist, hence it's an error ✖
- [x] f. ilock is held on the inode, and hence it's an error if it is a directory ✖
- [ ] g. No directory entry was found for the file to be opened, hence an error

Your answer is incorrect.

The correct answer is: One of the sub-components on the given path name, was not a directory, hence it's an error

Question **10**

Partially correct

Mark 0.50 out of 1.00

Suppose an application on xv6 does the following:

```
int main() {
  char arr[128];
  int fd = open("README, O_RDONLY);
  read(fd, arrr, 100);
}
```

Assume that the code works.

Which of the following things are true about xv6 kernel code, w.r.t. the above C program.

True	False			
✖	☑	The "memmove(dst, bp->data + off%BSIZE, m);" in readi() will copy the data from the disk to the kernel buffers	✖	It will transfer from kernel buffer to user memory (arr)
☑	✖	The process will be made to sleep only once	✔	Yes. Reading 100 bytes means reading only one disk block. So bread()->iderw() is called only once.
✖	☑	The data is transferred from disk to kernel buffers first, and then address of arr is maped to the kernel buffers	✖	No. data is copied into arr.
☑	✖	The ONLY function that gets called on return devsw[ip->major].read(ip, dst, n); is consoleread	✔	There is no other device file in xv6
✖	☑	The loop in readi() will always read a different block using bread()	✖	
☑	✖	value of fd will be 3	✔	

The "memmove(dst, bp->data + off%BSIZE, m);" in readi() will copy the data from the disk to the kernel buffers: False
The process will be made to sleep only once: True
The data is transferred from disk to kernel buffers first, and then address of arr is maped to the kernel buffers: False
The ONLY function that gets called on return devsw[ip->major].read(ip, dst, n); is consoleread: True
The loop in readi() will always read a different block using bread(): False
value of fd will be 3: True

Question **11**

Partially correct

Mark 0.83 out of 1.00

Map the function in xv6's file system code, to it's perceived logical layer.

ideintr	disk driver	✔
filestat()	file descriptor	✔
ialloc	inode	✔
bread	buffer cache	✔
balloc	system call	✖
skipelem	pathname lookup	✔
bmap	pathname lookup	✖
sys_chdir()	system call	✔
stati	inode	✔
commit	logging	✔
namei	pathname lookup	✔
dirlookup	directory	✔

Your answer is partially correct.

You have correctly selected 10.

The correct answer is: ideintr → disk driver, filestat() → file descriptor, ialloc → inode, bread → buffer cache, balloc → block allocation on disk, skipelem → pathname lookup, bmap → inode, sys_chdir() → system call, stati → inode, commit → logging, namei → pathname lookup, dirlookup → directory

Question **12**

Incorrect

Mark 0.00 out of 1.00

Match function with it's functionality

nameiparent	Write a new entry in a given directory	✖
dirlink	Link a directory with another directory	✖
namex	Search a given name in a given directory	✖
dirlookup	Lookup (search) for a given directory	✖

Your answer is incorrect.

The correct answer is: nameiparent → return in-memory inode for parent directory of a given pathname, dirlink → Write a new entry in a given directory, namex → return in-memory inode for a given pathname, dirlookup → Search a given name in a given directory

Question **13**

Incorrect

Mark 0.00 out of 1.00

An inode is read from disk as a part of this function

- a. ilock
- b. sys_read ✖
- c. iget
- d. readi
- e. iread

Your answer is incorrect.

The correct answer is: ilock

◄ Random Quiz - 5: xv6 make, bootloader, interrupt handling, memory management

Jump to...

(Random Quiz - 7) Pre-Endsem Quiz ►

Started on	Thursday, 16 February 2023, 9:00 PM
State	Finished
Completed on	Thursday, 16 February 2023, 10:43 PM
Time taken	1 hour 43 mins
Grade	**13.00** out of 15.00 (**86.68**%)

Question **1**

Correct

Mark 1.00 out of 1.00

Which of the following statements is false ?

Select one:

- a. A process scheduling algorithm is preemptive if the CPU can be forcibly removed from a process.
- b. Time sharing systems generally use preemptive CPU scheduling.
- c. Real time systems generally use non preemptive CPU scheduling. ✔
- d. Response time is more predictable in preemptive systems than in non preemptive systems.

Your answer is correct.

The correct answer is: Real time systems generally use non preemptive CPU scheduling.

Question **2**

Correct

Mark 1.00 out of 1.00

Order the sequence of events, in scheduling process P1 after process P0

timer interrupt occurs	2 ✔
Process P1 is running	6 ✔
context of P0 is saved in P0's PCB	3 ✔
Control is passed to P1	5 ✔
context of P1 is loaded from P1's PCB	4 ✔
Process P0 is running	1 ✔

Your answer is correct.

The correct answer is: timer interrupt occurs → 2, Process P1 is running → 6, context of P0 is saved in P0's PCB → 3, Control is passed to P1 → 5, context of P1 is loaded from P1's PCB → 4, Process P0 is running → 1

Question **3**

Correct

Mark 1.00 out of 1.00

Select the compiler's view of the process's address space, for each of the following MMU schemes:
(Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

Paging	one continuous chunk ✔
Relocation + Limit	one continuous chunk ✔
Segmentation	many continuous chunks of variable size ✔
Segmentation, then paging	many continuous chunks of variable size ✔

Your answer is correct.

The correct answer is: Paging → one continuous chunk, Relocation + Limit → one continuous chunk, Segmentation → many continuous chunks of variable size, Segmentation, then paging → many continuous chunks of variable size

Question **4**

Correct

Mark 1.00 out of 1.00

Match the names of PCB structures with kernel

linux struct task_struct ✔

xv6 struct proc ✔

The correct answer is: linux → struct task_struct, xv6 → struct proc

Question **5**

Correct

Mark 1.00 out of 1.00

Select the state that is not possible after the given state, for a process:

New: Running ✔

Ready : Waiting ✔

Running: : None of these ✔

Waiting: Running ✔

Question **6**

Partially correct

Mark 0.47 out of 1.00

Select all the correct statements about zombie processes

Select one or more:

- [ ] a. A zombie process occupies space in OS data structures
- [x] b. init() typically keeps calling wait() for zombie processes to get cleaned up ✔
- [x] c. If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent ✔
- [x] d. A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it ✔
- [x] e. Zombie processes are harmless even if OS is up for long time ✖
- [x] f. A process can become zombie if it finishes, but the parent has finished before it ✔
- [ ] g. A process becomes zombie when it's parent finishes
- [ ] h. A zombie process remains zombie forever, as there is no way to clean it up

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it, A process can become zombie if it finishes, but the parent has finished before it, A zombie process occupies space in OS data structures, If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

Question **7**

Partially correct

Mark 0.86 out of 1.00

Mark True/False

Statements about scheduling and scheduling algorithms

True	False			
◉☑	○✖	xv6 code does not care about Processor Affinity	✔	
○☑	◉✖	Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.	✖	
◉☑	○✖	Response time will be quite poor on non-interruptible kernels	✔	
◉☑	○✖	Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.	✔	
◉☑	○✖	Processor Affinity refers to memory accesses of a process being stored on cache of that processor	✔	
◉☑	○✖	A scheduling algorithm is non-premptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.	✔	
○✖	◉☑	On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread	✔	It's the negation of this. Time NOT spent in idle thread.

xv6 code does not care about Processor Affinity: True
Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.: True
Response time will be quite poor on non-interruptible kernels: True
Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.: True
Processor Affinity refers to memory accesses of a process being stored on cache of that processor: True
A scheduling algorithm is non-premptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.: True
On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread: False

Question **8**

Correct

Mark 1.00 out of 1.00

Mark whether the concept is related to scheduling or not.

Yes	No		
✗	✓	file-table	✓
✓	✗	context-switch	✓
✓	✗	timer interrupt	✓
✓	✗	ready-queue	✓
✓	✗	runnable process	✓

file-table: No
context-switch: Yes
timer interrupt: Yes
ready-queue: Yes
runnable process: Yes

Question **9**

Partially correct

Mark 0.50 out of 1.00

Select all the correct statements about signals

Select one or more:

- [ ] a. Signals are delivered to a process by another process
- [ ] b. The signal handler code runs in user mode of CPU
- [x] c. A signal handler can be invoked asynchronously or synchronously depending on signal type ✔
- [x] d. SIGKILL definitely kills a process because it can't be caught or ignored, and it's default action terminates the process ✔
- [ ] e. Signal handlers once replaced can't be restored
- [x] f. The signal handler code runs in kernel mode of CPU ✖
- [x] g. Signals are delivered to a process by kernel ✔
- [ ] h. SIGKILL definitely kills a process because it's code runs in kernel mode of CPU

Your answer is partially correct.

You have correctly selected 3.
The correct answers are: Signals are delivered to a process by kernel, A signal handler can be invoked asynchronously or synchronously depending on signal type, The signal handler code runs in user mode of CPU, SIGKILL definitely kills a process because it can't be caught or ignored, and it's default action terminates the process

Question **10**

Correct

Mark 1.00 out of 1.00

Map each signal with it's meaning

SIGPIPE	Broken Pipe	✔
SIGCHLD	Child Stopped or Terminated	✔
SIGUSR1	User Defined Signal	✔
SIGALRM	Timer Signal from alarm()	✔
SIGSEGV	Invalid Memory Reference	✔

The correct answer is: SIGPIPE → Broken Pipe, SIGCHLD → Child Stopped or Terminated, SIGUSR1 → User Defined Signal, SIGALRM → Timer Signal from alarm(), SIGSEGV → Invalid Memory Reference

Question **11**

Partially correct

Mark 0.80 out of 1.00

Mark statements True/False w.r.t. change of states of a process. Note that a statement is true only if the claim and argument both are true.

Reference: The process state diagram (and your understanding of how kernel code works). Note - the diagram does not show zombie state!

Figure 3.2 Diagram of process state.

True	False		
◉☑	○✖	Every forked process has to go through ZOMBIE state, at least for a small duration.	✔
◉☑	○✖	Only a process in READY state is considered by scheduler	✔
◉☑	○✖	A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet	✔
◉✖	○☑	A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first	✖
○✖	◉☑	A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.	✔

Every forked process has to go through ZOMBIE state, at least for a small duration.: True
Only a process in READY state is considered by scheduler: True
A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet: True
A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first: False
A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.: False

Question **12**

Correct

Mark 1.00 out of 1.00

Which of the following parts of a C program do not have any corresponding machine code ?

- [ ] a. pointer dereference
- [ ] b. function calls
- [x] c. global variables ✔
- [ ] d. local variable declaration
- [x] e. #directives ✔
- [x] f. typedefs ✔
- [ ] g. expressions

Your answer is correct.

The correct answers are: #directives, typedefs, global variables

Question **13**

Partially correct

Mark 1.38 out of 2.00

Select all the correct statements about the state of a process.

- [x] a. A process in ready state is ready to be scheduled ✔
- [ ] b. A waiting process starts running after the wait is over
- [ ] c. A process changes from running to ready state on a timer interrupt or any I/O wait
- [x] d. A process waiting for I/O completion is typically woken up by the particular interrupt handler code ✔
- [x] e. Processes in the ready queue are in the ready state ✔
- [ ] f. A process changes from running to ready state on a timer interrupt
- [ ] g. It is not maintained in the data structures by kernel, it is only for conceptual understanding of programmers
- [x] h. A process waiting for any condition is woken up by another process only ✖
- [x] i. A running process may terminate, or go to wait or become ready again ✔
- [x] j. A process can self-terminate only when it's running ✔
- [x] k. Typically, it's represented as a number in the PCB ✔
- [x] l. A process that is running is not on the ready queue ✔
- [x] m. Changing from running state to waiting state results in "giving up the CPU" ✔
- [ ] n. A process in ready state is ready to receive interrupts

Your answer is partially correct.

You have correctly selected 8.

The correct answers are: Typically, it's represented as a number in the PCB, A process in ready state is ready to be scheduled, Processes in the ready queue are in the ready state, A process that is running is not on the ready queue, A running process may terminate, or go to wait or become ready again, A process changes from running to ready state on a timer interrupt, Changing from running state to waiting state results in "giving up the CPU", A process can self-terminate only when it's running, A process waiting for I/O completion is typically woken up by the particular interrupt handler code

Question **14**

Correct

Mark 1.00 out of 1.00

Which of the following are NOT a part of job of a typical compiler?

- [ ] a. Invoke the linker to link the function calls with their code, extern globals with their declaration
- [x] b. Check the program for logical errors ✔
- [ ] c. Convert high level langauge code to machine code
- [ ] d. Check the program for syntactical errors
- [x] e. Suggest alternative pieces of code that can be written ✔
- [ ] f. Process the # directives in a C program

Your answer is correct.

The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

◄ Random Quiz - 3 (processes, memory management, event driven kernel), compilation-linking-loading, ipc-pipes

Jump to...

Homework questions: Basics of MM, xv6 booting ►

Started on	Tuesday, 16 January 2024, 5:05 PM
State	Finished
Completed on	Tuesday, 16 January 2024, 5:56 PM
Time taken	50 mins 58 secs
Grade	**11.48** out of 15.00 (**76.56**%)

Question **1**

Partially correct

Mark 0.80 out of 1.00

Select all the correct statements about two modes of CPU operation

Select one or more:

- [ ] a. The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously
- [x] b. There is an instruction like 'iret' to return from kernel mode to user mode ✔
- [x] c. The two modes are essential for a multiprogramming system ✔
- [x] d. The two modes are essential for a multitasking system ✔
- [x] e. Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode ✔

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: The two modes are essential for a multiprogramming system, The two modes are essential for a multitasking system, There is an instruction like 'iret' to return from kernel mode to user mode, The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously, Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode

Question **2**

Incorrect

Mark 0.00 out of 1.00

Write the possible contents of the file /tmp/xyz after this program.

In the answer if you want to mention any non-text character, then write \0 For example abc\0\0 means abc followed by any two non-text characters

```
int main(int argc, char *argv[]) {
   int fd1, fd2, n, i;
   char buf[128];

   fd1 = open("/tmp/xyz", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
   write(fd1, "hello", 5);
   fd2 = open("/tmp/xyz", O_WRONLY, S_IRUSR|S_IWUSR);
   write(fd2, "bye", 3);
   close(fd1);
   close(fd2);
   return 0;
}
```

Answer: hello bye ✖

The correct answer is: byelo

Question **3**

Correct

Mark 0.50 out of 0.50

Select all the correct statements about bootloader.

Every wrong selection will deduct marks proportional to 1/n where n is total wrong choices in the question.

You will get minimum a zero.

- [x] a. Bootloaders allow selection of OS to boot from ✔
- [x] b. LILO is a bootloader ✔
- [ ] c. The bootloader loads the BIOS
- [ ] d. Bootloader must be one sector in length
- [x] e. Modern Bootloaders often allow configuring the way an OS boots ✔

Your answer is correct.

The correct answers are: LILO is a bootloader, Modern Bootloaders often allow configuring the way an OS boots, Bootloaders allow selection of OS to boot from

Question **4**

Partially correct

Mark 0.75 out of 1.00

Given below is the output of "ps -eaf".

Answer the questions based on it.

```
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 Jan05 ?        00:01:08 /sbin/init splash
root           2       0  0 Jan05 ?        00:00:00 [kthreadd]
root           3       2  0 Jan05 ?        00:00:00 [rcu_gp]
root           4       2  0 Jan05 ?        00:00:00 [rcu_par_gp]
root           9       2  0 Jan05 ?        00:00:00 [mm_percpu_wq]
root          10       2  0 Jan05 ?        00:00:00 [rcu_tasks_rude_]
root          11       2  0 Jan05 ?        00:00:00 [rcu_tasks_trace]
root          12       2  0 Jan05 ?        00:00:22 [ksoftirqd/0]
root          13       2  0 Jan05 ?        00:06:29 [rcu_sched]
root          14       2  0 Jan05 ?        00:00:02 [migration/0]
root          15       2  0 Jan05 ?        00:00:00 [idle_inject/0]
root          16       2  0 Jan05 ?        00:00:00 [cpuhp/0]
root          17       2  0 Jan05 ?        00:00:00 [cpuhp/1]
root          18       2  0 Jan05 ?        00:00:00 [idle_inject/1]
root          19       2  0 Jan05 ?        00:00:03 [migration/1]
root          20       2  0 Jan05 ?        00:00:13 [ksoftirqd/1]
root          22       2  0 Jan05 ?        00:00:00 [kworker/1:0H-events_highpri]
root          23       2  0 Jan05 ?        00:00:00 [cpuhp/2]
root          24       2  0 Jan05 ?        00:00:00 [idle_inject/2]
root          25       2  0 Jan05 ?        00:00:01 [migration/2]
root          26       2  0 Jan05 ?        00:00:09 [ksoftirqd/2]
root          28       2  0 Jan05 ?        00:00:00 [kworker/2:0H-kblockd]
root          29       2  0 Jan05 ?        00:00:00 [cpuhp/3]
root          30       2  0 Jan05 ?        00:00:00 [idle_inject/3]
root          31       2  0 Jan05 ?        00:00:02 [migration/3]
root          32       2  0 Jan05 ?        00:00:07 [ksoftirqd/3]
root          34       2  0 Jan05 ?        00:00:00 [kworker/3:0H-events_highpri]
root          35       2  0 Jan05 ?        00:00:00 [cpuhp/4]
root          36       2  0 Jan05 ?        00:00:00 [idle_inject/4]
root          37       2  0 Jan05 ?        00:00:02 [migration/4]
root          38       2  0 Jan05 ?        00:00:06 [ksoftirqd/4]
root          40       2  0 Jan05 ?        00:00:00 [kworker/4:0H-events_highpri]
root          41       2  0 Jan05 ?        00:00:00 [cpuhp/5]
root          42       2  0 Jan05 ?        00:00:00 [idle_inject/5]
root          43       2  0 Jan05 ?        00:00:02 [migration/5]
root          44       2  0 Jan05 ?        00:00:05 [ksoftirqd/5]
root          46       2  0 Jan05 ?        00:00:00 [kworker/5:0H-events_highpri]
root          47       2  0 Jan05 ?        00:00:00 [cpuhp/6]
root          48       2  0 Jan05 ?        00:00:00 [idle_inject/6]
root          49       2  0 Jan05 ?        00:00:02 [migration/6]
root          50       2  0 Jan05 ?        00:00:05 [ksoftirqd/6]
root          52       2  0 Jan05 ?        00:00:00 [kworker/6:0H-events_highpri]
root          53       2  0 Jan05 ?        00:00:00 [cpuhp/7]
root          54       2  0 Jan05 ?        00:00:00 [idle_inject/7]
root          55       2  0 Jan05 ?        00:00:02 [migration/7]
root          56       2  0 Jan05 ?        00:00:06 [ksoftirqd/7]
root          58       2  0 Jan05 ?        00:00:00 [kworker/7:0H-events_highpri]
root          59       2  0 Jan05 ?        00:00:00 [kdevtmpfs]
root          60       2  0 Jan05 ?        00:00:00 [netns]
root          61       2  0 Jan05 ?        00:00:00 [inet_frag_wq]
root          62       2  0 Jan05 ?        00:00:00 [kauditd]
root          63       2  0 Jan05 ?        00:00:00 [khungtaskd]
root          64       2  0 Jan05 ?        00:00:00 [oom_reaper]
root          65       2  0 Jan05 ?        00:00:00 [writeback]
```

```
root          66       2  0 Jan05 ?        00:01:58 [kcompactd0]
root          67       2  0 Jan05 ?        00:00:00 [ksmd]
root          68       2  0 Jan05 ?        00:00:04 [khugepaged]
root         115       2  0 Jan05 ?        00:00:00 [kintegrityd]
root         116       2  0 Jan05 ?        00:00:00 [kblockd]
root         117       2  0 Jan05 ?        00:00:00 [blkcg_punt_bio]
root         118       2  0 Jan05 ?        00:00:00 [tpm_dev_wq]
root         119       2  0 Jan05 ?        00:00:00 [ata_sff]
root         120       2  0 Jan05 ?        00:00:00 [md]
root         121       2  0 Jan05 ?        00:00:00 [edac-poller]
root         122       2  0 Jan05 ?        00:00:00 [devfreq_wq]
root         123       2  0 Jan05 ?        00:00:00 [watchdogd]
root         129       2  0 Jan05 ?        00:00:00 [irq/25-AMD-Vi]
root         131       2  0 Jan05 ?        00:04:33 [kswapd0]
root         132       2  0 Jan05 ?        00:00:00 [ecryptfs-kthrea]
root         134       2  0 Jan05 ?        00:00:00 [kthrotld]
root         135       2  0 Jan05 ?        00:00:00 [irq/27-pciehp]
root         139       2  0 Jan05 ?        00:00:00 [acpi_thermal_pm]
root         140       2  0 Jan05 ?        00:00:00 [vfio-irqfd-clea]
root         141       2  0 Jan05 ?        00:00:00 [ipv6_addrconf]
root         144       2  0 Jan05 ?        00:00:03 [kworker/6:1H-kblockd]
root         151       2  0 Jan05 ?        00:00:00 [kstrp]
root         154       2  0 Jan05 ?        00:00:00 [zswap-shrink]
root         162       2  0 Jan05 ?        00:00:00 [charger_manager]
root         164       2  0 Jan05 ?        00:00:03 [kworker/4:1H-kblockd]
root         197       2  0 Jan05 ?        00:00:03 [kworker/3:1H-kblockd]
root         213       2  0 Jan05 ?        00:00:03 [kworker/7:1H-kblockd]
root         215       2  0 Jan05 ?        00:00:00 [nvme-wq]
root         216       2  0 Jan05 ?        00:00:00 [nvme-reset-wq]
root         217       2  0 Jan05 ?        00:00:00 [nvme-delete-wq]
root         223       2  0 Jan05 ?        00:00:00 [irq/42-ELAN2513]
root         224       2  0 Jan05 ?        00:04:20 [irq/41-ELAN071B]
root         225       2  0 Jan05 ?        00:00:00 [cryptd]
root         226       2  0 Jan05 ?        00:00:00 [amd_iommu_v2]
root         249       2  0 Jan05 ?        00:00:00 [ttm_swap]
root         250       2  0 Jan05 ?        00:20:29 [gfx]
root         251       2  0 Jan05 ?        00:00:00 [comp_1.0.0]
root         252       2  0 Jan05 ?        00:00:00 [comp_1.1.0]
root         253       2  0 Jan05 ?        00:00:00 [comp_1.2.0]
root         254       2  0 Jan05 ?        00:00:00 [comp_1.3.0]
root         255       2  0 Jan05 ?        00:00:00 [comp_1.0.1]
root         256       2  0 Jan05 ?        00:00:00 [comp_1.1.1]
root         257       2  0 Jan05 ?        00:00:00 [comp_1.2.1]
root         258       2  0 Jan05 ?        00:00:00 [comp_1.3.1]
root         259       2  0 Jan05 ?        00:00:27 [sdma0]
root         260       2  0 Jan05 ?        00:00:00 [vcn_dec]
root         261       2  0 Jan05 ?        00:00:00 [vcn_enc0]
root         262       2  0 Jan05 ?        00:00:00 [vcn_enc1]
root         263       2  0 Jan05 ?        00:00:00 [jpeg_dec]
root         265       2  0 Jan05 ?        00:00:00 [card0-crtc0]
root         266       2  0 Jan05 ?        00:00:00 [card0-crtc1]
root         267       2  0 Jan05 ?        00:00:00 [card0-crtc2]
root         268       2  0 Jan05 ?        00:00:00 [card0-crtc3]
root         271       2  0 Jan05 ?        00:00:03 [kworker/5:1H-kblockd]
root         277       2  0 Jan05 ?        00:00:03 [kworker/1:1H-kblockd]
root         320       2  0 Jan05 ?        00:00:00 [raid5wq]
root         380       2  0 Jan05 ?        00:00:11 [jbd2/nvme0n1p5-]
root         381       2  0 Jan05 ?        00:00:00 [ext4-rsv-conver]
root         432       2  0 Jan05 ?        00:00:03 [kworker/2:1H-kblockd]
root         446       1  0 Jan05 ?        00:01:16 /lib/systemd/systemd-journald
root         470       2  0 Jan05 ?        00:00:00 [rpciod]
root         473       2  0 Jan05 ?        00:00:00 [xprtiod]
root         474       2  0 Jan05 ?        00:00:00 bpfilter_umh
```

```
root         503       1  0 Jan05 ?        00:00:07 /lib/systemd/systemd-udevd
root         517       2  0 Jan05 ?        00:00:00 [loop0]
root         554       2  0 Jan05 ?        00:00:00 [loop1]
root         563       2  0 Jan05 ?        00:00:00 [loop2]
root         586       2  0 Jan05 ?        00:00:00 [loop3]
root         588       2  0 Jan05 ?        00:00:00 [loop4]
root         589       2  0 Jan05 ?        00:00:00 [loop5]
root         612       2  0 Jan05 ?        00:00:00 [loop6]
root         613       2  0 Jan05 ?        00:00:00 [loop7]
root         629       2  0 Jan05 ?        00:00:00 [loop8]
root         637       2  0 Jan05 ?        00:00:00 [cfg80211]
root         676       2  0 Jan05 ?        00:05:00 [irq/75-iwlwifi:]
root         678       2  0 Jan05 ?        00:01:07 [irq/76-iwlwifi:]
root         682       2  0 Jan05 ?        00:01:27 [irq/77-iwlwifi:]
root         688       2  0 Jan05 ?        00:00:49 [irq/78-iwlwifi:]
root         695       2  0 Jan05 ?        00:01:39 [irq/79-iwlwifi:]
root         700       2  0 Jan05 ?        00:01:22 [irq/80-iwlwifi:]
root         703       2  0 Jan05 ?        00:01:13 [irq/81-iwlwifi:]
root         704       2  0 Jan05 ?        00:01:38 [irq/82-iwlwifi:]
root         708       2  0 Jan05 ?        00:00:44 [irq/83-iwlwifi:]
root         713       2  0 Jan05 ?        00:00:00 [loop9]
root         715       2  0 Jan05 ?        00:00:00 [irq/84-iwlwifi:]
root         782       2  0 Jan05 ?        00:00:00 [loop10]
root         797       2  0 Jan05 ?        00:00:00 [loop11]
root         811       2  0 Jan05 ?        00:00:00 [loop12]
root         838       2  0 Jan05 ?        00:00:00 [loop13]
root         847       2  0 Jan05 ?        00:00:00 [loop14]
root         879       2  0 Jan05 ?        00:00:00 [loop15]
root         884       2  0 Jan05 ?        00:00:00 [loop16]
root         885       2  0 Jan05 ?        00:00:00 [loop17]
root         945       2  0 Jan05 ?        00:00:00 [loop18]
root         946       2  0 Jan05 ?        00:00:00 [loop19]
root         947       2  0 Jan05 ?        00:00:00 [loop20]
root        1012       2  0 Jan05 ?        00:00:00 [jbd2/nvme0n1p8-]
root        1013       2  0 Jan05 ?        00:00:00 [ext4-rsv-conver]
root        1015       2  0 Jan05 ?        00:01:09 [jbd2/nvme0n1p7-]
root        1016       2  0 Jan05 ?        00:00:00 [ext4-rsv-conver]
_rpc        1062       1  0 Jan05 ?        00:00:00 /sbin/rpcbind -f -w
systemd+    1063       1  0 Jan05 ?        00:01:24 /lib/systemd/systemd-resolved
systemd+    1064       1  0 Jan05 ?        00:00:00 /lib/systemd/systemd-timesyncd
root        1144       1  0 Jan05 ?        00:00:46 /usr/sbin/acpid
avahi       1146       1  0 Jan05 ?        00:00:06 avahi-daemon: running [abhijit-laptop.local]
root        1149       1  0 Jan05 ?        00:00:01 /usr/lib/bluetooth/bluetoothd
message+    1150       1  0 Jan05 ?        00:04:21 /usr/bin/dbus-daemon --system --address=systemd: --nofork --nopidfile -
-systemd-activation --syslog-only
root        1152       1  0 Jan05 ?        00:03:12 /usr/sbin/NetworkManager --no-daemon
root        1157       1  0 Jan05 ?        00:01:02 /usr/sbin/iio-sensor-proxy
root        1159       1  0 Jan05 ?        00:00:27 /usr/sbin/irqbalance --foreground
root        1162       1  0 Jan05 ?        00:00:01 /usr/bin/lxcfs /var/lib/lxcfs
root        1165       1  0 Jan05 ?        00:00:00 /usr/bin/python3 /usr/bin/networkd-dispatcher --run-startup-triggers
root        1170       1  0 Jan05 ?        00:00:29 /usr/lib/policykit-1/polkitd --no-debug
syslog      1175       1  0 Jan05 ?        00:00:20 /usr/sbin/rsyslogd -n -iNONE
root        1182       1  0 Jan05 ?        00:01:13 /usr/lib/snapd/snapd
root        1187       1  0 Jan05 ?        00:00:12 /usr/lib/accountsservice/accounts-daemon
root        1192       1  0 Jan05 ?        00:00:00 /usr/sbin/cron -f
root        1198       1  0 Jan05 ?        00:00:00 /usr/libexec/switcheroo-control
root        1201       1  0 Jan05 ?        00:00:12 /lib/systemd/systemd-logind
root        1202       1  0 Jan05 ?        00:00:07 /lib/systemd/systemd-machined
root        1203       1  0 Jan05 ?        00:01:28 /usr/lib/udisks2/udisksd
root        1204       1  0 Jan05 ?        00:00:15 /sbin/wpa_supplicant -u -s -O /run/wpa_supplicant
daemon      1209       1  0 Jan05 ?        00:00:00 /usr/sbin/atd -f
avahi       1216    1146  0 Jan05 ?        00:00:00 avahi-daemon: chroot helper
docker-+    1279       1  0 Jan05 ?        00:00:22 /usr/bin/docker-registry serve /etc/docker/registry/config.yml
```

```
root        1282       1  0 Jan05 ?        00:00:00 /usr/bin/python3 /usr/bin/twistd3 --nodaemon --pidfile= epoptes
jenkins     1285       1  0 Jan05 ?        00:15:01 /usr/bin/java -Djava.awt.headless=true -jar /usr/share/java/jenkins.war
--webroot=/var/cache/jenkins/war --httpPort=8080
root        1296       1  0 Jan05 ?        00:00:18 php-fpm: master process (/etc/php/7.4/fpm/php-fpm.conf)
vnstat      1314       1  0 Jan05 ?        00:00:15 /usr/sbin/vnstatd -n
root        1326       1  0 Jan05 ?        00:00:02 /usr/sbin/ModemManager
root        1327       1  0 Jan05 ?        00:00:31 /usr/bin/anydesk --service
colord      1359       1  0 Jan05 ?        00:00:01 /usr/libexec/colord
root        1374       1  0 Jan05 ?        00:00:00 /usr/sbin/gdm3
root        1420       1  0 Jan05 ?        00:00:14 /usr/sbin/apache2 -k start
www-data    1436    1296  0 Jan05 ?        00:00:00 php-fpm: pool www
www-data    1437    1296  0 Jan05 ?        00:00:00 php-fpm: pool www
mysql       1461       1  0 Jan05 ?        00:38:52 /usr/sbin/mysqld
root        1490       1  0 Jan05 ?        00:00:04 /usr/sbin/libvirtd
root        1491       1  0 Jan05 ?        00:00:00 /usr/bin/python3 /usr/share/unattended-upgrades/unattended-upgrade-
shutdown --wait-for-signal
rtkit       1593       1  0 Jan05 ?        00:00:07 /usr/libexec/rtkit-daemon
libvirt+    1766       1  0 Jan05 ?        00:00:00 /usr/sbin/dnsmasq --conf-file=/var/lib/libvirt/dnsmasq/default.conf --
leasefile-ro --dhcp-script=/usr/lib/libvirt/libvirt_leaseshelper
root        1767    1766  0 Jan05 ?        00:00:00 /usr/sbin/dnsmasq --conf-file=/var/lib/libvirt/dnsmasq/default.conf --
leasefile-ro --dhcp-script=/usr/lib/libvirt/libvirt_leaseshelper
root        1859       1  0 Jan05 ?        00:00:18 /usr/lib/upower/upowerd
root        1995       1  0 Jan05 ?        00:00:00 /opt/saltstack/salt/run/run minion
root        2041    1995  0 Jan05 ?        00:05:18 /opt/saltstack/salt/run/run minion MultiMinionProcessManager
MinionProcessManager
root        2278       1  0 Jan05 ?        00:00:50 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
root        2282       1  0 Jan05 ?        00:00:17 /usr/sbin/inetd
whoopsie    2302       1  0 Jan05 ?        00:00:01 /usr/bin/whoopsie -f
kernoops    2330       1  0 Jan05 ?        00:00:19 /usr/sbin/kerneloops --test
kernoops    2341       1  0 Jan05 ?        00:00:19 /usr/sbin/kerneloops
root        2366       2  0 Jan05 ?        00:00:00 [iprt-VBoxWQueue]
lxc-dns+    2370       1  0 Jan05 ?        00:00:00 dnsmasq --conf-file=/dev/null -u lxc-dnsmasq --strict-order --bind-
interfaces --pid-file=/run/lxc/dnsmasq.pid --listen-address 10.0.3.1 --dhcp-range 10.0.3.2,10.0.3.254 --dhcp-lease-max=253
--dhcp-no-override --except-interface=lo --interface=lxcbr0 --dhcp-leasefile=/var/lib/misc/dnsmasq.lxcbr0.leases --dhcp-
authoritative
root        2404       2  0 Jan05 ?        00:00:00 [iprt-VBoxTscThr]
root        3508       1  0 Jan05 ?        00:00:01 /usr/lib/postfix/sbin/master -w
root        3615    1374  0 Jan05 ?        00:00:01 gdm-session-worker [pam/gdm-password]
abhijit     3629       1  0 Jan05 ?        00:00:17 /lib/systemd/systemd --user
abhijit     3630    3629  0 Jan05 ?        00:00:00 (sd-pam)
abhijit     3636    3629  1 Jan05 ?        04:31:21 /usr/bin/pulseaudio --daemonize=no --log-target=journal
abhijit     3638    3629  0 Jan05 ?        00:20:35 /usr/libexec/tracker-miner-fs
abhijit     3642    3629  0 Jan05 ?        00:01:11 /usr/bin/dbus-daemon --session --address=systemd: --nofork --nopidfile
--systemd-activation --syslog-only
abhijit     3644       1  0 Jan05 ?        00:00:03 /usr/bin/gnome-keyring-daemon --daemonize --login
abhijit     3662    3629  0 Jan05 ?        00:00:01 /usr/libexec/gvfsd
abhijit     3667    3629  0 Jan05 ?        00:00:00 /usr/libexec/gvfsd-fuse /run/user/1000/gvfs -f -o big_writes
abhijit     3673    3629  0 Jan05 ?        00:00:02 /usr/libexec/gvfs-udisks2-volume-monitor
abhijit     3681    3629  0 Jan05 ?        00:00:01 /usr/libexec/gvfs-mtp-volume-monitor
abhijit     3685    3629  0 Jan05 ?        00:00:11 /usr/libexec/gvfs-afc-volume-monitor
abhijit     3690    3629  0 Jan05 ?        00:00:01 /usr/libexec/gvfs-gphoto2-volume-monitor
abhijit     3695    3629  0 Jan05 ?        00:00:01 /usr/libexec/gvfs-goa-volume-monitor
abhijit     3700    3629  0 Jan05 ?        00:00:01 /usr/libexec/goa-daemon
root        3701       2  0 Jan05 ?        00:00:00 [krfcommd]
abhijit     3708    3629  0 Jan05 ?        00:00:04 /usr/libexec/goa-identity-service
abhijit     3724    3615  0 Jan05 tty2     00:00:00 /usr/lib/gdm3/gdm-x-session --run-script env
GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --systemd --session=ubuntu
abhijit     3726    3724  1 Jan05 tty2     02:47:57 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority
-background none -noreset -keeptty -verbose 3
abhijit     3747    3724  0 Jan05 tty2     00:00:00 /usr/libexec/gnome-session-binary --systemd --systemd --session=ubuntu
abhijit     3816    3747  0 Jan05 ?        00:00:01 /usr/bin/ssh-agent /usr/bin/im-launch env
GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --systemd --session=ubuntu
abhijit     3845    3629  0 Jan05 ?        00:00:00 /usr/libexec/at-spi-bus-launcher
```

```
abhijit     3850    3845  0 Jan05 ?        00:00:07 /usr/bin/dbus-daemon --config-file=/usr/share/defaults/at-
spi2/accessibility.conf --nofork --print-address 3
abhijit     3869    3629  0 Jan05 ?        00:00:00 /usr/libexec/gnome-session-ctl --monitor
abhijit     3876    3629  0 Jan05 ?        00:00:04 /usr/libexec/gnome-session-binary --systemd-service --session=ubuntu
abhijit     3890    3629  1 Jan05 ?        03:43:47 /usr/bin/gnome-shell
abhijit     3927    3890  0 Jan05 ?        00:31:49 ibus-daemon --panel disable --xim
abhijit     3931    3927  0 Jan05 ?        00:00:00 /usr/libexec/ibus-dconf
abhijit     3932    3927  0 Jan05 ?        00:01:51 /usr/libexec/ibus-extension-gtk3
abhijit     3934    3629  0 Jan05 ?        00:00:04 /usr/libexec/ibus-x11 --kill-daemon
abhijit     3937    3629  0 Jan05 ?        00:00:02 /usr/libexec/ibus-portal
abhijit     3949    3629  0 Jan05 ?        00:00:27 /usr/libexec/at-spi2-registryd --use-gnome-session
abhijit     3953    3629  0 Jan05 ?        00:00:00 /usr/libexec/xdg-permission-store
abhijit     3955    3629  0 Jan05 ?        00:00:01 /usr/libexec/gnome-shell-calendar-server
abhijit     3964    3629  0 Jan05 ?        00:00:00 /usr/libexec/evolution-source-registry
abhijit     3973    3629  0 Jan05 ?        00:00:02 /usr/libexec/evolution-calendar-factory
abhijit     3986    3629  0 Jan05 ?        00:00:01 /usr/libexec/dconf-service
abhijit     3992    3629  0 Jan05 ?        00:00:01 /usr/libexec/evolution-addressbook-factory
abhijit     4007    3629  0 Jan05 ?        00:00:00 /usr/bin/gjs /usr/share/gnome-shell/org.gnome.Shell.Notifications
abhijit     4023    3629  0 Jan05 ?        00:00:00 /usr/libexec/gsd-a11y-settings
abhijit     4025    3629  0 Jan05 ?        00:00:08 /usr/libexec/gsd-color
abhijit     4029    3629  0 Jan05 ?        00:00:00 /usr/libexec/gsd-datetime
abhijit     4032    3629  0 Jan05 ?        00:00:21 /usr/libexec/gsd-housekeeping
abhijit     4033    3629  0 Jan05 ?        00:00:05 /usr/libexec/gsd-keyboard
abhijit     4036    3629  0 Jan05 ?        00:00:12 /usr/libexec/gsd-media-keys
abhijit     4037    3629  0 Jan05 ?        00:00:11 /usr/libexec/gsd-power
abhijit     4038    3629  0 Jan05 ?        00:00:00 /usr/libexec/gsd-print-notifications
abhijit     4039    3629  0 Jan05 ?        00:00:01 /usr/libexec/gsd-rfkill
abhijit     4041    3629  0 Jan05 ?        00:00:01 /usr/libexec/gsd-screensaver-proxy
abhijit     4042    3876  0 Jan05 ?        00:01:06 /usr/lib/x86_64-linux-gnu/libexec/kdeconnectd
abhijit     4045    3629  0 Jan05 ?        00:00:35 /usr/libexec/gsd-sharing
abhijit     4047    3629  0 Jan05 ?        00:00:00 /usr/libexec/gsd-smartcard
abhijit     4051    3629  0 Jan05 ?        00:00:00 /usr/libexec/gsd-sound
abhijit     4057    3629  0 Jan05 ?        00:00:00 /usr/libexec/gsd-usb-protection
abhijit     4063    3629  0 Jan05 ?        00:00:05 /usr/libexec/gsd-wacom
abhijit     4071    3629  0 Jan05 ?        00:00:00 /usr/libexec/gsd-wwan
abhijit     4072    3876  0 Jan05 ?        00:01:00 baloo_file
abhijit     4075    3876  0 Jan05 ?        00:00:00 /usr/libexec/gsd-disk-utility-notify
abhijit     4076    3629  0 Jan05 ?        00:00:08 /usr/libexec/gsd-xsettings
abhijit     4078    3876  0 Jan05 ?        00:00:10 /usr/bin/python3 /usr/bin/blueman-applet
abhijit     4082    3876  0 Jan05 ?        00:00:14 /usr/bin/anydesk --tray
abhijit     4108    3876  0 Jan05 ?        00:00:00 /usr/lib/x86_64-linux-gnu/indicator-messages/indicator-messages-service
abhijit     4109    3876  0 Jan05 ?        00:00:06 /usr/libexec/evolution-data-server/evolution-alarm-notify
abhijit     4129    3629  0 Jan05 ?        00:00:50 /snap/snap-store/959/usr/bin/snap-store --gapplication-service
abhijit     4191    3629  0 Jan05 ?        00:00:00 /usr/libexec/gsd-printer
abhijit     4219    3629  0 Jan05 ?        00:00:03 /usr/libexec/xdg-document-portal
abhijit     4265    3927  0 Jan05 ?        00:03:20 /usr/libexec/ibus-engine-simple
abhijit     4291    3629  0 Jan05 ?        00:00:10 /usr/bin/python3 /usr/bin/blueman-tray
abhijit     4301    3629  0 Jan05 ?        00:00:00 /usr/lib/bluetooth/obexd
abhijit     4377    3662  0 Jan05 ?        00:00:02 /usr/libexec/gvfsd-trash --spawner :1.3 /org/gtk/gvfs/exec_spaw/0
abhijit     4395    3629  0 Jan05 ?        00:00:12 /usr/libexec/xdg-desktop-portal
abhijit     4399    3629  0 Jan05 ?        00:01:08 /usr/libexec/xdg-desktop-portal-gtk
abhijit     4480    3629  0 Jan05 ?        00:00:21 /usr/libexec/gvfsd-metadata
abhijit     5687    3662  0 Jan05 ?        00:00:00 /usr/libexec/gvfsd-network --spawner :1.3 /org/gtk/gvfs/exec_spaw/1
abhijit     5701    3662  0 Jan05 ?        00:00:01 /usr/libexec/gvfsd-dnssd --spawner :1.3 /org/gtk/gvfs/exec_spaw/3
root        6228       2  0 Jan05 ?        00:00:00 [kdmflush]
root        6236       2  0 Jan05 ?        00:00:00 [kcryptd_io/253:]
root        6237       2  0 Jan05 ?        00:00:00 [kcryptd/253:0]
root        6238       2  0 Jan05 ?        00:00:23 [dmcrypt_write/2]
root        6260       2  0 Jan05 ?        00:00:24 [jbd2/dm-0-8]
root        6261       2  0 Jan05 ?        00:00:00 [ext4-rsv-conver]
abhijit     6421    3927  0 Jan05 ?        00:00:36 /usr/lib/ibus/ibus-engine-m17n --ibus
abhijit     6434    3876  0 Jan05 ?        00:00:13 update-notifier
abhijit    30565    3629  0 Jan05 ?        00:08:56 /usr/libexec/gnome-terminal-server
```

```
abhijit    30576   30565  0 Jan05 pts/0    00:00:00 bash
abhijit   131017  364845  1 Jan09 ?        03:12:27 /usr/lib/virtualbox/VirtualBoxVM --comment ubuntu 18.04 --startvm
45993a5c-3ded-452f-941e-4579d12c1ad9 --no-startvm-errormsgbox
abhijit   159668   30565  0 Jan09 pts/10   00:00:00 bash
abhijit   161637   30565  0 Jan05 pts/1    00:00:01 bash
abhijit   171109  159668  0 Jan09 pts/10   00:00:33 evince......
abhijit   171114    3629  0 Jan09 ?        00:00:00 /usr/libexec/evinced
abhijit   204055    3629  0 Jan10 ?        00:00:13 /usr/bin/python3 /usr/bin/update-manager --no-update --no-focus-on-map
abhijit   270190    3629  0 Jan05 ?        00:56:34 /usr/lib/x86_64-linux-gnu/libexec/kactivitymanagerd
abhijit   270199    3629  0 Jan05 ?        00:00:24 /usr/bin/kglobalaccel5
abhijit   270207    3629  0 Jan05 ?        00:00:00 kdeinit5: Running...
abhijit   270208  270207  0 Jan05 ?        00:00:19 /usr/lib/x86_64-linux-gnu/libexec/kf5/klauncher --fd=8
abhijit   279971    3629  0 Jan05 ?        01:20:12 telegram-desktop
abhijit   280011  279971  0 Jan05 ?        00:00:00 sh -c /usr/lib/x86_64-linux-gnu/libproxy/0.4.15/pxgsettings
org.gnome.system.proxy org.gnome.system.proxy.http org.gnome.system.proxy.https org.gnome.system.proxy.ftp
org.gnome.system.proxy.socks
abhijit   280015  280011  0 Jan05 ?        00:00:00 /usr/lib/x86_64-linux-gnu/libproxy/0.4.15/pxgsettings
org.gnome.system.proxy org.gnome.system.proxy.http org.gnome.system.proxy.https org.gnome.system.proxy.ftp
org.gnome.system.proxy.socks
abhijit   325756    3629  0 Jan12 ?        00:01:09 /usr/libexec/tracker-store
root      326592       1  0 Jan12 ?        00:00:00 sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups
abhijit   347912    3644  0 Jan06 ?        00:00:00 /usr/bin/ssh-agent -D -a /run/user/1000/keyring/.ssh
root      348716       1  0 Jan12 ?        00:00:28 /usr/bin/containerd
abhijit   351306    3629  3 Jan12 ?        03:32:42 /usr/lib/firefox/firefox
abhijit   351429  351306  0 Jan12 ?        00:00:00 /usr/lib/firefox/firefox -contentproc -parentBuildID 20240108143603 -
prefsLen 37272 -prefMapSize 247458 -appDir /usr/lib/firefox/browser {83364ade-74ec-4bcc-94f8-9f3779a869f1} 351306 true
socket
abhijit   351458  351306  0 Jan12 ?        00:29:34 /usr/lib/firefox/firefox -contentproc -childID 1 -isForBrowser -
prefsLen 37337 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {4dbb54ee-e1a2-41e4-b10b-587526532b78} 351306
true tab
abhijit   351495  351306  0 Jan12 ?        00:04:06 /usr/lib/firefox/firefox -contentproc -childID 2 -isForBrowser -
prefsLen 38090 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {02f1a4e7-d831-4a5f-a9fd-59a809bb03e8} 351306
true tab
abhijit   351717  351306  0 Jan12 ?        00:01:14 /usr/lib/firefox/firefox -contentproc -parentBuildID 20240108143603 -
sandboxingKind 0 -prefsLen 42823 -prefMapSize 247458 -appDir /usr/lib/firefox/browser {a1e46009-cfd1-46c8-ac9c-
19ba8bf3f46a} 351306 true utility
abhijit   351844  351306  0 Jan12 ?        00:07:23 /usr/lib/firefox/firefox -contentproc -parentBuildID 20240108143603 -
prefsLen 43306 -prefMapSize 247458 -appDir /usr/lib/firefox/browser {91008bef-b6d6-452f-b4a6-e78d887b3569} 351306 true rdd
abhijit   353500    3662  0 Jan06 ?        00:00:00 /usr/libexec/gvfsd-http --spawner :1.3 /org/gtk/gvfs/exec_spaw/4
abhijit   364809    3890  0 Jan06 ?        00:19:20 /usr/lib/virtualbox/VirtualBox
abhijit   364837    3629  0 Jan06 ?        00:16:49 /usr/lib/virtualbox/VBoxXPCOMIPCD
abhijit   364845    3629  0 Jan06 ?        00:29:44 /usr/lib/virtualbox/VBoxSVC --auto-shutdown
root      364957       2  0 Jan06 ?        00:00:00 [dio/dm-0]
abhijit   369749   30565  0 Jan06 pts/3    00:00:00 bash
abhijit   369879   30565  0 Jan06 pts/4    00:00:00 bash
abhijit   379620   30565  0 Jan06 pts/6    00:00:00 bash
abhijit   381917    3890  0 Jan06 ?        00:00:46 flameshot
root      386201       2  0 Jan06 ?        00:00:02 [kworker/0:2H-acpi_thermal_pm]
postfix   389291    3508  0 Jan06 ?        00:00:00 qmgr -l -t unix -u
root      486171       2  0 Jan12 ?        00:00:01 [kworker/0:0H-kblockd]
abhijit   527395    3629  0 Jan12 ?        00:00:49 /usr/bin/gedit --gapplication-service
abhijit   551299  351306  0 Jan12 ?        00:01:54 /usr/lib/firefox/firefox -contentproc -childID 438 -isForBrowser -
prefsLen 35109 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {c4ebd70d-bad7-4e7c-9066-39827103c84e} 351306
true tab
abhijit   552002    3890  0 Jan12 ?        00:03:32 /opt/Signal/signal-desktop --no-sandbox
abhijit   552005  552002  0 Jan12 ?        00:00:00 /opt/Signal/signal-desktop --type=zygote --no-zygote-sandbox --no-
sandbox
abhijit   552006  552002  0 Jan12 ?        00:00:00 /opt/Signal/signal-desktop --type=zygote --no-sandbox
abhijit   552037  552005  0 Jan12 ?        00:03:28 /opt/Signal/signal-desktop --type=gpu-process --no-sandbox --enable-
crash-reporter=18887fa1-4d37-46f0-bf9f-b37513c81cf9,no_channel --user-data-dir=/home/abhijit/.config/Signal --gpu-
```

preferences=WAAAAAAAAAAgAAAEAAAAAAAAAAAAAAAAAAABgAAAAAAA4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABAAAAGAAAAAAAAAAYAAAAAAAA
--use-gl=angle --use-angle=swiftshader-webgl --shared-files --field-trial-
handle=0,i,3213315393136307567,15155041764863120512,262144 --disable-
features=HardwareMediaKeyHandling,SpareRendererForSitePerProcess
abhijit 552045 552002 0 Jan12 ?   00:00:04 /opt/Signal/signal-desktop --type=utility --utility-sub-
type=network.mojom.NetworkService --lang=en-GB --service-sandbox-type=none --no-sandbox --enable-crash-reporter=18887fa1-
4d37-46f0-bf9f-b37513c81cf9,no_channel --user-data-dir=/home/abhijit/.config/Signal --shared-
files=v8_context_snapshot_data:100 --field-trial-handle=0,i,3213315393136307567,15155041764863120512,262144 --disable-
features=HardwareMediaKeyHandling,SpareRendererForSitePerProcess
abhijit 552103 552002 1 Jan12 ?   00:57:51 /opt/Signal/signal-desktop --type=renderer --enable-crash-
reporter=18887fa1-4d37-46f0-bf9f-b37513c81cf9,no_channel --user-data-dir=/home/abhijit/.config/Signal --app-
path=/opt/Signal/resources/app.asar --no-sandbox --no-zygote --enable-blink-features=CSSPseudoDir,CSSLogical --disable-
blink-features=Accelerated2dCanvas,AcceleratedSmallCanvases --first-renderer-process --no-sandbox --disable-gpu-compositing
--lang=en-GB --num-raster-threads=4 --enable-main-frame-before-activation --renderer-client-id=4 --time-ticks-at-unix-
epoch=-1704847690836396 --launch-time-ticks=218630770368 --shared-files=v8_context_snapshot_data:100 --field-trial-
handle=0,i,3213315393136307567,15155041764863120512,262144 --disable-
features=HardwareMediaKeyHandling,SpareRendererForSitePerProcess
abhijit 552149 552002 0 Jan12 ?   00:00:12 /opt/Signal/signal-desktop --type=utility --utility-sub-
type=audio.mojom.AudioService --lang=en-GB --service-sandbox-type=none --no-sandbox --enable-crash-reporter=18887fa1-4d37-
46f0-bf9f-b37513c81cf9,no_channel --user-data-dir=/home/abhijit/.config/Signal --shared-files=v8_context_snapshot_data:100
--field-trial-handle=0,i,3213315393136307567,15155041764863120512,262144 --disable-
features=HardwareMediaKeyHandling,SpareRendererForSitePerProcess
abhijit 554660 351306 0 Jan13 ?   00:04:20 /usr/lib/firefox/firefox -contentproc -childID 442 -isForBrowser -
prefsLen 35109 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {2cffa6f3-6f8c-4a3f-880d-01cc71baf983} 351306
true tab
abhijit 554855 351306 4 Jan13 ?   03:24:06 /usr/lib/firefox/firefox -contentproc -childID 445 -isForBrowser -
prefsLen 35109 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {abcba8e9-5b97-4281-ad64-1356841dcf34} 351306
true tab
abhijit 556349 351306 0 Jan13 ?   00:01:18 /usr/lib/firefox/firefox -contentproc -childID 451 -isForBrowser -
prefsLen 35109 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {6318d453-643c-4b85-9f0f-bfd0a429cd41} 351306
true tab
abhijit 557600   3629 0 Jan13 ?   00:00:08 /usr/lib/speech-dispatcher-modules/sd_espeak-ng /etc/speech-
dispatcher/modules/espeak-ng.conf
abhijit 557607   3629 0 Jan13 ?   00:00:08 /usr/lib/speech-dispatcher-modules/sd_dummy /etc/speech-
dispatcher/modules/dummy.conf
abhijit 557610   3629 0 Jan13 ?   00:00:08 /usr/lib/speech-dispatcher-modules/sd_generic /etc/speech-
dispatcher/modules/mary-generic.conf
abhijit 557613   3629 0 Jan13 ?   00:00:00 /usr/bin/speech-dispatcher --spawn --communication-method unix_socket -
-socket-path /run/user/1000/speech-dispatcher/speechd.sock
abhijit 571320 351306 0 Jan13 ?   00:01:16 /usr/lib/firefox/firefox -contentproc -childID 486 -isForBrowser -
prefsLen 35110 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {71966a59-b1df-4c44-975c-62d7cd41188c} 351306
true tab
abhijit 571410 351306 0 Jan13 ?   00:01:34 /usr/lib/firefox/firefox -contentproc -childID 488 -isForBrowser -
prefsLen 35110 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {bfbc15ff-6ca5-4837-b97e-334e0bfc8855} 351306
true tab
abhijit 571582 351306 0 Jan13 ?   00:03:32 /usr/lib/firefox/firefox -contentproc -childID 492 -isForBrowser -
prefsLen 35110 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {04e791e5-9cde-4e03-9211-141bdb440139} 351306
true tab
abhijit 591339 4139495 0 Jan13 pts/9   00:00:00 vi mh-pyt.txt
root   594356   2 0 Jan13 ?   00:00:00 [dio/nvme0n1p7]
abhijit 594726   30565 0 Jan13 pts/5   00:00:00 bash
abhijit 791421 351306 1 Jan14 ?   00:40:27 /usr/lib/firefox/firefox -contentproc -childID 839 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {b6f69be3-7ebe-43c9-9d55-c72161180b5e} 351306
true tab
abhijit 794226 351306 0 Jan14 ?   00:02:03 /usr/lib/firefox/firefox -contentproc -childID 848 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -

```
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {28fddc81-ea39-4496-a1b6-b70d9a8a9c31} 351306
true tab
abhijit   797871  351306  0 Jan14 ?        00:02:41 /usr/lib/firefox/firefox -contentproc -childID 851 -isForBrowser -
prefsLen 35226 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {b254a3c7-e1f7-4245-9fcf-820074a4e69f} 351306
true tab
abhijit   799607   30565  0 Jan14 pts/11   00:00:00 bash
abhijit   803503   30565  0 Jan14 pts/12   00:00:00 bash
abhijit   815513  799607  0 Jan15 pts/11   00:00:00 vi timetable-todo2
abhijit   821738  351306  0 Jan15 ?        00:07:30 /usr/lib/firefox/firefox -contentproc -childID 995 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {90cad852-941e-44b3-8fa4-0d44cbcfda7a} 351306
true tab
abhijit   895193  351306  0 Jan15 ?        00:00:04 /usr/lib/firefox/firefox -contentproc -childID 1131 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {218262bd-4f9e-423a-9368-4d2e44f33125} 351306
true tab
root      942398       1  0 10:40 ?        00:00:00 /usr/sbin/cupsd -l
root      942399       1  0 10:40 ?        00:00:00 /usr/sbin/cups-browsed
www-data  942465    1420  0 10:40 ?        00:00:00 /usr/sbin/apache2 -k start
www-data  942466    1420  0 10:40 ?        00:00:00 /usr/sbin/apache2 -k start
www-data  942468    1420  0 10:40 ?        00:00:00 /usr/sbin/apache2 -k start
www-data  942469    1420  0 10:40 ?        00:00:00 /usr/sbin/apache2 -k start
www-data  942470    1420  0 10:40 ?        00:00:00 /usr/sbin/apache2 -k start
www-data  942471    1420  0 10:40 ?        00:00:00 /usr/sbin/apache2 -k start
abhijit   954109   30565  0 11:21 pts/2    00:00:00 bash
abhijit   961628  351306  3 12:36 ?        00:05:34 /usr/lib/firefox/firefox -contentproc -childID 1704 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {cb82b978-ef9d-4a76-b59c-5b13b652a0ec} 351306
true tab
root      961877       2  0 12:36 ?        00:00:02 [kworker/u32:5-events_unbound]
root      962113       2  0 12:39 ?        00:00:08 [kworker/u33:3-hci0]
abhijit   968060  351306  0 13:09 ?        00:00:05 /usr/lib/firefox/firefox -contentproc -childID 1749 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {c0ba3673-966e-48a6-b029-91346e348342} 351306
true tab
root      968299       2  0 13:11 ?        00:00:02 [kworker/u32:4-events_unbound]
root      969560       2  0 13:23 ?        00:00:01 [kworker/5:1-cgroup_destroy]
root      969608       2  0 13:24 ?        00:00:00 [kworker/u32:7-events_unbound]
root      969648       2  0 13:24 ?        00:00:01 [kworker/2:0-inet_frag_wq]
abhijit   969715  379620  0 13:25 pts/6    00:00:00 ssh root@10.1.101.41
root      970435       2  0 13:27 ?        00:00:01 [kworker/3:2-rcu_gp]
root      971921       2  0 13:33 ?        00:00:00 [kworker/1:2-rcu_gp]
root      972048       2  0 13:35 ?        00:00:00 [kworker/7:2-rcu_gp]
root      972127       2  0 13:37 ?        00:00:01 [kworker/0:1-events_long]
root      972207       2  0 13:37 ?        00:00:01 [kworker/4:1-events]
root      972378       2  0 13:39 ?        00:00:00 [kworker/1:0-cgroup_destroy]
root      972435       2  0 13:39 ?        00:00:00 [kworker/6:2-events]
root      972964       2  0 13:41 ?        00:00:00 [kworker/7:0-events]
root      973166       2  0 13:41 ?        00:00:00 [kworker/u32:0-events_unbound]
root      973193       2  0 13:42 ?        00:00:00 [kworker/2:1-events]
root      973282       2  0 13:43 ?        00:00:00 [kworker/4:2-events]
root      973384       2  0 13:44 ?        00:00:00 [kworker/3:0-rcu_gp]
root      973389       2  0 13:44 ?        00:00:01 [kworker/u33:0-hci0]
root      973391       2  0 13:44 ?        00:00:00 [kworker/6:0-rcu_gp]
root      973392       2  0 13:44 ?        00:00:00 [kworker/5:2-events]
root      973655       2  0 13:45 ?        00:00:00 [kworker/1:1-events]
root      973664       2  0 13:46 ?        00:00:00 [kworker/7:1-events]
root      973965       2  0 13:47 ?        00:00:00 [kworker/2:2-events]
root      974126       2  0 13:48 ?        00:00:00 [kworker/u32:1-kcryptd/253:0]
root      974189       2  0 13:48 ?        00:00:00 [kworker/4:0-events]
root      974190       2  0 13:48 ?        00:00:00 [kworker/0:2-events]
root      974369       2  0 13:49 ?        00:00:00 [kworker/u32:2-nvme-wq]
```

```
root      974539       2  0 13:49 ?        00:00:00 [kworker/u32:3-events_unbound]
root      974655       2  0 13:50 ?        00:00:00 [kworker/6:1-events]
root      974690       2  0 13:50 ?        00:00:00 [kworker/5:0-events]
root      974740       2  0 13:50 ?        00:00:00 [kworker/7:3-events]
root      974742       2  0 13:50 ?        00:00:00 [kworker/3:1-pm]
root      974743       2  0 13:50 ?        00:00:00 [kworker/u32:6-events_unbound]
root      974744       2  0 13:50 ?        00:00:00 [kworker/u32:8-kcryptd/253:0]
root      974745       2  0 13:50 ?        00:00:00 [kworker/u32:9-events_unbound]
root      974746       2  0 13:50 ?        00:00:00 [kworker/u32:10-events_unbound]
root      974747       2  0 13:50 ?        00:00:00 [kworker/u32:11-events_unbound]
root      974748       2  0 13:50 ?        00:00:00 [kworker/u32:12-kcryptd/253:0]
root      974749       2  0 13:50 ?        00:00:00 [kworker/u32:13-events_unbound]
root      974750       2  0 13:50 ?        00:00:00 [kworker/u32:14-events_unbound]
root      974751       2  0 13:50 ?        00:00:00 [kworker/u32:15-events_unbound]
root      974752       2  0 14:18 ?        00:00:00 [kworker/u32:16-events_unbound]
root      974753       2  0 14:18 ?        00:00:00 [kworker/u32:17-events_unbound]
root      974754       2  0 14:18 ?        00:00:00 [kworker/u32:18-events_unbound]
root      974755       2  0 14:18 ?        00:00:00 [kworker/u32:19-events_unbound]
root      974756       2  0 14:18 ?        00:00:00 [kworker/u32:20-kcryptd/253:0]
root      974757       2  0 14:18 ?        00:00:00 [kworker/u32:21-events_unbound]
root      974758       2  0 14:18 ?        00:00:00 [kworker/u32:22-events_unbound]
root      974759       2  0 14:18 ?        00:00:00 [kworker/u32:23-events_unbound]
root      974760       2  0 14:18 ?        00:00:00 [kworker/u32:24-events_unbound]
root      974761       2  0 14:18 ?        00:00:00 [kworker/u32:25-events_unbound]
root      974762       2  0 14:18 ?        00:00:00 [kworker/u32:26-events_unbound]
root      974763       2  0 14:18 ?        00:00:00 [kworker/u32:27-kcryptd/253:0]
root      974764       2  0 14:18 ?        00:00:00 [kworker/u32:28-kcryptd/253:0]
root      974765       2  0 14:18 ?        00:00:00 [kworker/u32:29-events_unbound]
root      974766       2  0 14:18 ?        00:00:00 [kworker/u32:30+events_unbound]
root      974767       2  0 14:18 ?        00:00:00 [kworker/u32:31-events_unbound]
root      974768       2  0 14:18 ?        00:00:00 [kworker/u32:32-events_unbound]
root      974769       2  0 14:18 ?        00:00:00 [kworker/u32:33-events_unbound]
root      974770       2  0 14:18 ?        00:00:00 [kworker/u32:34-events_unbound]
root      974771       2  0 14:18 ?        00:00:00 [kworker/u32:35-events_unbound]
root      974772       2  0 14:18 ?        00:00:00 [kworker/u32:36-events_unbound]
root      974773       2  0 14:18 ?        00:00:00 [kworker/u32:37-events_unbound]
root      974774       2  0 14:18 ?        00:00:00 [kworker/u32:38-events_unbound]
root      974775       2  0 14:18 ?        00:00:00 [kworker/u32:39-events_unbound]
root      974776       2  0 14:18 ?        00:00:00 [kworker/u32:40-kcryptd/253:0]
root      974778       2  0 14:18 ?        00:00:00 [kworker/u32:42-events_unbound]
root      974779       2  0 14:18 ?        00:00:00 [kworker/3:3-events]
root      974780       2  0 14:18 ?        00:00:00 [kworker/3:4-events]
root      974798       2  0 14:18 ?        00:00:00 [kworker/3:5-events]
root      974995       2  0 14:18 ?        00:00:00 [kworker/u33:2-rb_allocator]
root      975505       2  0 14:20 ?        00:00:00 [kworker/6:3-events]
root      975656       2  0 14:20 ?        00:00:00 [kworker/5:3-events]
root      975657       2  0 14:29 ?        00:00:00 [kworker/1:3-pm]
root      975658       2  0 14:29 ?        00:00:00 [kworker/1:4-events]
abhijit   975722  351306  0 14:29 ?        00:00:02 /usr/lib/firefox/firefox -contentproc -childID 1836 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {d9ea12ea-5b3f-477c-9c08-9a4196ea8d04} 351306
true tab
root      976242       2  0 15:16 ?        00:00:00 [kworker/0:0-events]
root      976243       2  0 15:16 ?        00:00:00 [kworker/0:3-events]
root      976244       2  0 15:16 ?        00:00:00 [kworker/0:4-events]
postfix   976248    3508  0 15:16 ?        00:00:00 pickup -l -t unix -u -c
abhijit   976770  351306  0 15:18 ?        00:00:00 /usr/lib/firefox/firefox -contentproc -childID 1840 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {cb567d37-1c03-46db-966c-632f4b708354} 351306
true tab
abhijit   976836  351306  0 15:19 ?        00:00:00 /usr/lib/firefox/firefox -contentproc -childID 1841 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {b62e1986-221e-481e-bda4-62fb37caaa67} 351306
```

```
true tab
abhijit   977138  351306  0 15:20 ?        00:00:00 /usr/lib/firefox/firefox -contentproc -childID 1842 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {989da731-7bd5-44ce-abd0-200ca4c84b9b} 351306
true tab
abhijit   977184  594726  0 15:21 pts/5    00:00:00 ps -eaf
abhijit  1664880    3629  0 Jan07 ?        00:00:06 /usr/bin/gnome-calendar --gapplication-service
abhijit  1665340    3629  0 Jan07 ?        00:00:00 /usr/bin/gpg-agent --supervised
abhijit  3872409    3629  0 Jan07 ?        00:00:05 /usr/bin/seahorse --gapplication-service
root     3873244       1  0 Jan07 ?        00:00:55 /sbin/mount.ntfs /dev/nvme0n1p3 /media/abhijit/windows -o
rw,nodev,nosuid,windows_names,uid=1000,gid=1000,uhelper=udisks2
abhijit  3884359   30565  0 Jan07 pts/7    00:00:00 bash
abhijit  4108623   30565  0 Jan08 pts/8    00:00:00 bash
abhijit  4136834    3890  0 Jan08 ?        00:00:00 /usr/lib/libreoffice/program/oosplash --calc
abhijit  4136869 4136834  0 Jan08 ?        00:32:11 /usr/lib/libreoffice/program/soffice.bin --calc
abhijit  4139495   30565  0 Jan08 pts/9    00:00:00 bash
```

The PID of the grand-parent of the process with PID 4108623 is : 3629 ✔

The two processes which were created by kernel have PIDs (in increasing order) 1 ✔ and 2 ✔

The process that created most of the "graphical" processes is having PID 30565 ✖

Question **5**

Correct

Mark 1.00 out of 1.00

How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security) ?

Select one:

- a. It disallows hardware interrupts when a process is running
- b. It prohibits invocation of kernel code completely, if a user program is running
- c. It prohibits a user mode process from running privileged instructions ✔
- d. It prohibits one process from accessing other process's memory

Your answer is correct.

The correct answer is: It prohibits a user mode process from running privileged instructions

Question **6**

Correct

Mark 1.00 out of 1.00

Select all statements that correctly explain the use/purpose of system calls.

Select one or more:

- [x] a. Allow I/O device access to user processes ✔
- [x] b. Provide services for accessing files ✔
- [ ] c. Handle ALL types of interrupts
- [ ] d. Handle exceptions like division by zero
- [x] e. Switch from user mode to kernel mode ✔
- [x] f. Provide an environment for process creation ✔
- [ ] g. Run each instruction of an application program

Your answer is correct.

The correct answers are: Switch from user mode to kernel mode, Provide services for accessing files, Allow I/O device access to user processes, Provide an environment for process creation

Question **7**

Incorrect

Mark 0.00 out of 1.00

Select all the correct statements about the process init on Linuxes/Unixes.

Select one or more:

- [ ] a. no process can exec 'init'
- [ ] b. init is created by kernel 'by hand'
- [ ] c. init can not be killed with SIGKILL
- [x] d. init is created by kernel by forking itself ✘
- [x] e. init typically has a pid=1 ✔
- [ ] f. any user process can fork and exec init
- [ ] g. only a process run by 'root' user can exec 'init'

Your answer is incorrect.

The correct answers are: init is created by kernel 'by hand', init typically has a pid=1, init can not be killed with SIGKILL, only a process run by 'root' user can exec 'init'

Question **8**

Correct

Mark 1.00 out of 1.00

Order the following events in boot process (from 1 onwards)

Login interface	5	✔
Shell	6	✔
BIOS	1	✔
OS	3	✔
Init	4	✔
Boot loader	2	✔

Your answer is correct.

The correct answer is: Login interface → 5, Shell → 6, BIOS → 1, OS → 3, Init → 4, Boot loader → 2

Question **9**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about bootloader.

Every wrong selection will deduct marks proportional to 1/n where n is total wrong choices in the question.

You will get minimum a zero.

- [ ] a. The bootloader loads the BIOS
- [x] b. Bootloaders allow selection of OS to boot from ✔
- [x] c. Modern Bootloaders often allow configuring the way an OS boots ✔
- [x] d. LILO is a bootloader ✔
- [ ] e. Bootloader must be one sector in length

Your answer is correct.

The correct answers are: LILO is a bootloader, Modern Bootloaders often allow configuring the way an OS boots, Bootloaders allow selection of OS to boot from

Question **10**

Correct

Mark 0.50 out of 0.50

Is the terminal a part of the kernel on GNU/Linux systems?

- ( ) a. yes
- (•) b. no ✔ wrong

The correct answer is: no

Question **11**

Correct

Mark 0.50 out of 0.50

Compare multiprogramming with multitasking

- a. A multitasking system is not necessarily multiprogramming
- b. A multiprogramming system is not necessarily multitasking ✔

The correct answer is: A multiprogramming system is not necessarily multitasking

Question **12**

Correct

Mark 1.00 out of 1.00

Consider the following programs

exec1.c

```
#include <unistd.h>
#include <stdio.h>
int main() {
    execl("./exec2", "./exec2", NULL);
}
```

exec2.c

```
#include <unistd.h>
#include <stdio.h>
int main() {
    execl("/bin/ls", "/bin/ls", NULL);
   printf("hello\n");
}
```

Compiled as

```
cc    exec1.c   -o exec1
cc    exec2.c   -o exec2
```

And run as

```
$ ./exec1
```

Explain the output of the above command (./exec1)

Assume that /bin/ls , i.e. the 'ls' program exists.

Select one:

- a. Execution fails as one exec can't invoke another exec
- b. Execution fails as the call to execl() in exec2 fails
- c. "ls" runs on current directory ✔
- d. Execution fails as the call to execl() in exec1 fails
- e. Program prints hello

Your answer is correct.

The correct answer is: "ls" runs on current directory

Question **13**

Partially correct

Mark 0.60 out of 1.00

Select all the correct statements about two modes of CPU operation

Select one or more:

- [ ] a. The two modes are essential for a multitasking system
- [ ] b. The two modes are essential for a multiprogramming system
- [x] c. The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously ✔
- [x] d. Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode ✔
- [x] e. There is an instruction like 'iret' to return from kernel mode to user mode ✔

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: The two modes are essential for a multiprogramming system, The two modes are essential for a multitasking system, There is an instruction like 'iret' to return from kernel mode to user mode, The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously, Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode

Question **14**

Correct

Mark 0.50 out of 0.50

When you turn your computer ON, on BIOS based systems, you are often shown an option like "Press F9 for boot options". What does this mean?

- ( ) a. The choice of booting slowly or fast
- ( ) b. The choice of the boot loader (e.g. GRUB or Windows-Loader)
- ( ) c. The choice of which OS to boot from
- (•) d. The BIOS allows us to choose the boot device, the device from which the boot loader will be loaded ✔

The correct answer is: The BIOS allows us to choose the boot device, the device from which the boot loader will be loaded

Question **15**

Correct

Mark 1.00 out of 1.00

Predict the output of the program given here.

Assume that all the path names for the programs are correct. For example "/usr/bin/echo" will actually run echo command.

Assume that there is no mixing of printf output on screen if two of them run concurrently.

In the answer replace a new line by a single space.

For example::

```
good

output
```

should be written as `good output`

```
--

main() {
    int i;
    i = fork();
    if(i == 0)
        execl("/usr/bin/echo", "/usr/bin/echo", "hi", 0);
    else
        wait(0);
    fork();
    execl("/usr/bin/echo", "/usr/bin/echo", "one", 0);
}
```

Answer: hi one one ✔

The correct answer is: hi one one

Question **16**

Partially correct

Mark 0.33 out of 1.00

Select the sequence of events that are NOT possible, assuming a non-interruptible kernel code

(Note: non-interruptible kernel code means, if the kernel code is executing, then interrupts will be disabled).

Note: A possible sequence may have some missing steps in between. An impossible sequence will will have n and n+1th steps such that n+1th step can not follow n'th step.

Select one or more:

- [ ] a. P1 running
  P1 makes system call
  system call returns
  P1 running
  timer interrupt
  Scheduler running
  P2 running
- [x] b. ✔
  P1 running
  P1 makes sytem call
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again
- [x] c. ✖
  P1 running
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P3 running
  Hardware interrupt
  Interrupt unblocks P1
  Interrupt returns
  P3 running
  Timer interrupt
  Scheduler
  P1 running
- [ ] d. P1 running
  P1 makes system call
  timer interrupt
  Scheduler
  P2 running
  timer interrupt
  Scheuler
  P1 running
  P1's system call return
- [ ] e. P1 running
  keyboard hardware interrupt
  keyboard interrupt handler running
  interrupt handler returns
  P1 running
  P1 makes sytem call
  system call returns
  P1 running
  timer interrupt
  scheduler
  P2 running
- [x] f. ✔
  P1 running
  P1 makes sytem call and blocks
  Scheduler
  P2 running

P2 makes sytem call and blocks
Scheduler
P1 running again

Your answer is partially correct.

You have correctly selected 2.
The correct answers are: P1 running
P1 makes sytem call and blocks
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again, P1 running
P1 makes system call
timer interrupt
Scheduler
P2 running
timer interrupt
Scheuler
P1 running
P1's system call return,
P1 running
P1 makes sytem call
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

Question **17**

Correct

Mark 1.00 out of 1.00

What will this program do?

```
int main() {
fork();
execl("/bin/ls", "/bin/ls", NULL);
printf("hello");
}
```

- a. run ls twice and print hello twice
- b. run ls once
- c. run ls twice and print hello twice, but output will appear in some random order
- d. one process will run ls, another will print hello
- e. run ls twice ✔

Your answer is correct.

The correct answer is: run ls twice

◄ Surprise Quiz - 1 (pre-requisites)

Jump to...

Surprise Quiz - 3 (processes, memory management, event driven kernel), compilation-linking-loading) ►

Started on	Wednesday, 7 February 2024, 6:19 PM
State	Finished
Completed on	Wednesday, 7 February 2024, 7:14 PM
Time taken	54 mins 55 secs
Grade	**15.60** out of 20.00 (**78**%)

Question **1**

Correct

Mark 1.00 out of 1.00

Select the order in which the various stages of a compiler execute.

Intermediate code generation	3	✔
Syntatical Analysis	2	✔
Pre-processing	1	✔
Loading	does not exist	✔
Linking	4	✔

The correct answer is: Intermediate code generation → 3, Syntatical Analysis → 2, Pre-processing → 1, Loading → does not exist, Linking → 4

Question **2**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about MMU and it's functionality (on a non-demand paged system)

Select one or more:

- [ ] a. Logical to physical address translations in MMU are done with specific machine instructions
- [ ] b. Illegal memory access is detected by operating system
- [x] c. Illegal memory access is detected in hardware by MMU and a trap is raised ✔
- [x] d. Logical to physical address translations in MMU are done in hardware, automatically ✔
- [ ] e. The operating system interacts with MMU for every single address translation
- [x] f. The Operating system sets up relevant CPU registers to enable proper MMU translations ✔
- [ ] g. MMU is a separate chip outside the processor
- [x] h. MMU is inside the processor ✔

The correct answers are: MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

Question **3**

Correct

Mark 2.00 out of 2.00

Order the events that occur on a timer interrupt:

Jump to a code pointed by IDT	2	✔
Select another process for execution	5	✔
Set the context of the new process	6	✔
Jump to scheduler code	4	✔
Change to kernel stack of currently running process	1	✔
Save the context of the currently running process	3	✔
Execute the code of the new process	7	✔

The correct answer is: Jump to a code pointed by IDT → 2, Select another process for execution → 5, Set the context of the new process → 6, Jump to scheduler code → 4, Change to kernel stack of currently running process → 1, Save the context of the currently running process → 3, Execute the code of the new process → 7

Question **4**

Incorrect

Mark 0.00 out of 2.00

Consider the two programs given below to implement the command (ignore the fact that error checks are not done on return values of functions)

$ ls . /tmp/asdfksdf >/tmp/ddd 2>&1

Program 1

```
int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(1);
    dup(fd);
    close(2);
    dup(fd);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Program 2

```
int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    close(1);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(2);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Select all the correct statements about the programs

Select one or more:

- [x] a. Program 1 ensures 2>&1 and does not ensure > /tmp/ddd ✖
- [ ] b. Program 2 is correct for > /tmp/ddd but not for 2>&1
- [ ] c. Program 1 makes sure that there is one file offset used for '2' and '1'
- [ ] d. Program 2 does 1>&2
- [ ] e. Program 2 makes sure that there is one file offset used for '2' and '1'
- [x] f. Program 1 is correct for > /tmp/ddd but not for 2>&1 ✖
- [ ] g. Only Program 1 is correct
- [ ] h. Program 1 does 1>&2
- [ ] i. Only Program 2 is correct
- [x] j. Both program 1 and 2 are incorrect ✖
- [ ] k. Program 2 ensures 2>&1 and does not ensure > /tmp/ddd
- [ ] l. Both programs are correct

The correct answers are: Only Program 1 is correct, Program 1 makes sure that there is one file offset used for '2' and '1'

Question **5**

Partially correct

Mark 0.75 out of 1.00

Select the compiler's view of the process's address space, for each of the following MMU schemes:
(Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

Paging	Many continuous chunks each of page size	✖
Segmentation	many continuous chunks of variable size	✔
Relocation + Limit	one continuous chunk	✔
Segmentation, then paging	many continuous chunks of variable size	✔

The correct answer is: Paging → one continuous chunk, Segmentation → many continuous chunks of variable size, Relocation + Limit → one continuous chunk, Segmentation, then paging → many continuous chunks of variable size

Question **6**

Correct

Mark 1.00 out of 1.00

A process blocks itself means

- a. The kernel code of system call calls scheduler
- b. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler ✔
- c. The kernel code of an interrupt handler, moves the process to a waiting queue and calls scheduler
- d. The application code calls the scheduler

The correct answer is: The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

Question **7**

Correct

Mark 1.00 out of 1.00

Select the state that is not possible after the given state, for a process:

New:	Running	✔
Ready :	Waiting	✔
Running: :	None of these	✔
Waiting:	Running	✔

Question **8**

Partially correct

Mark 0.25 out of 1.00

Select the correct statements about paging (not demand paging) mechanism

Select one or more:

- [ ] a. User process can update it's own page table entries
- [x] b. Page table is accessed by the MMU as part of execution of an instruction ✔
- [x] c. The PTBR is loaded by the OS ✔
- [ ] d. User process can update it's own PTBR
- [x] e. An invalid entry on a page means, either it was illegal memory reference or the page was not present in memory. ✖
- [ ] f. An invalid entry on a page means, it was an illegal memory reference
- [x] g. Page table is accessed by the OS as part of execuation of an instruction ✖
- [x] h. OS creates the page table for every process ✔

The correct answers are: OS creates the page table for every process, The PTBR is loaded by the OS, Page table is accessed by the MMU as part of execution of an instruction, An invalid entry on a page means, it was an illegal memory reference

Question **9**

Correct

Mark 1.00 out of 1.00

Consider the following code and MAP the file to which each fd points at the end of the code. Assume that files/folders exist when needed with proper permissions and open() calls work.

```
int main(int argc, char *argv[]) {
   int fd1, fd2 = 1, fd3 = 1, fd4 = 1;

   fd1 = open("/tmp/1", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
   fd2 = open("/tmp/2", O_RDDONLY);
   fd3 = open("/tmp/3", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
   close(0);
   close(1);
   dup(fd2);
   dup(fd3);
   close(fd3);
   dup2(fd2, fd4);
   printf("%d %d %d %d\n", fd1, fd2, fd3, fd4);
   return 0;
}
```

0	/tmp/2	✔
fd2	/tmp/2	✔
fd1	/tmp/1	✔
1	/tmp/3	✔
fd4	/tmp/2	✔
fd3	closed	✔
2	stderr	✔

The correct answer is: 0 → /tmp/2, fd2 → /tmp/2, fd1 → /tmp/1, 1 → /tmp/3, fd4 → /tmp/2, fd3 → closed, 2 → stderr

Question **10**

Correct

Mark 1.00 out of 1.00

Predict the output of the program given here.

Assume that there is no mixing of printf output on screen if two of them run concurrently.

In the answer replace a new line by a single space.

For example::

```
good
output
```

should be written as `good output`

--

```
int main() {
  int pid;
  printf("hi\n");
  pid = fork();
  if(pid == 0) {
    exit(0);
  }
  printf("bye\n");
  fork();
  printf("ok\n");
}
```

Answer: hi bye ok ok

The correct answer is: hi bye ok ok

Question **11**

Partially correct

Mark 0.80 out of 1.00

Select all the correct statements about zombie processes

Select one or more:

- [x] a. A zombie process occupies space in OS data structures ✔
- [x] b. A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it ✔
- [ ] c. A zombie process remains zombie forever, as there is no way to clean it up
- [x] d. If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent ✔
- [ ] e. A process becomes zombie when it's parent finishes
- [x] f. A process can become zombie if it finishes, but the parent has finished before it ✔
- [ ] g. Zombie processes are harmless even if OS is up for long time
- [ ] h. init() typically keeps calling wait() for zombie processes to get cleaned up

The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it, A process can become zombie if it finishes, but the parent has finished before it, A zombie process occupies space in OS data structures, If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

Question **12**

Correct

Mark 1.00 out of 1.00

Which of the following are NOT a part of job of a typical compiler?

- [x] a. Check the program for logical errors ✔
- [ ] b. Process the # directives in a C program
- [ ] c. Convert high level langauge code to machine code
- [ ] d. Check the program for syntactical errors
- [x] e. Suggest alternative pieces of code that can be written ✔
- [ ] f. Invoke the linker to link the function calls with their code, extern globals with their declaration

The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

Question **13**

Partially correct

Mark 1.80 out of 2.00

Match the elements of C program to their place in memory

Global Static variables	Data	✔
Global variables	Data	✔
Function code	Code	✔
Arguments	Stack	✔
#define MACROS	No Memory needed	✔
Malloced Memory	Heap	✔
#include files	No Memory needed	✖
Code of main()	Code	✔
Local Variables	Stack	✔
Local Static variables	Data	✔

The correct answer is: Global Static variables → Data, Global variables → Data, Function code → Code, Arguments → Stack, #define MACROS → No Memory needed, Malloced Memory → Heap, #include files → No memory needed, Code of main() → Code, Local Variables → Stack, Local Static variables → Data

Question **14**

Correct

Mark 1.00 out of 1.00

Mark the statements as True/False w.r.t. the basic concepts of memory management.

True	False		
◉	○	When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.	✔
○	◉	The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.	✔
◉	○	The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.	✔
○	◉	The kernel refers to the page table for converting each virtual address to physical address.	✔
◉	○	The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.	✔
○	◉	The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.	✔
○	◉	When a process is executing, each virtual address is converted into physical address by the kernel directly.	✔

When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.: True
The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.: False
The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.: True
The kernel refers to the page table for converting each virtual address to physical address.: False
The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.: True
The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.: False
When a process is executing, each virtual address is converted into physical address by the kernel directly.: False

Question **15**

Partially correct

Mark 0.25 out of 1.00

Select the sequence of events that are NOT possible, assuming an interruptible kernel code

Select one or more:

- [ ] a. P1 running
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P3 running
  Hardware interrupt
  Interrupt unblocks P1
  Interrupt returns
  P3 running
  Timer interrupt
  Scheduler
  P1 running
- [x] b. ✔
  P1 running
  P1 makes sytem call
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again
- [ ] c. P1 running
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again
- [ ] d. P1 running
  P1 makes system call
  timer interrupt
  Scheduler
  P2 running
  timer interrupt
  Scheuler
  P1 running
  P1's system call return
- [ ] e. P1 running
  keyboard hardware interrupt
  keyboard interrupt handler running
  interrupt handler returns
  P1 running
  P1 makes sytem call
  system call returns
  P1 running

timer interrupt
scheduler
P2 running

f. P1 running ✖
P1 makes system call
system call returns
P1 running
timer interrupt
Scheduler running
P2 running

The correct answers are: P1 running
P1 makes sytem call and blocks
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again,
P1 running
P1 makes sytem call
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

Question **16**

Partially correct

Mark 0.75 out of 1.00

Consider the image given below, which explains how paging works.

Figure 9.8 Paging hardware.

Mention whether each statement is True or False, with respect to this image.

True	False		
◉☑	○✖	The page table is indexed using page number	✔
◉☑	○✖	The PTBR is present in the CPU as a register	✔
◉☑	○✖	The page table is itself present in Physical memory	✔
○☑	◉✖	The physical address may not be of the same size (in bits) as the logical address	✖
○☑	◉✖	Maximum Size of page table is determined by number of bits used for page number	✖
○✖	◉☑	The page table is indexed using frame number	✔
○✖	◉☑	Size of page table is always determined by the size of RAM	✔
○✖	◉☑	The locating of the page table using PTBR also involves paging translation	✔

The page table is indexed using page number: True
The PTBR is present in the CPU as a register: True
The page table is itself present in Physical memory: True
The physical address may not be of the same size (in bits) as the logical address: True
Maximum Size of page table is determined by number of bits used for page number: True
The page table is indexed using frame number: False
Size of page table is always determined by the size of RAM: False
The locating of the page table using PTBR also involves paging translation: False

Question **17**

Correct

Mark 1.00 out of 1.00

which of the following is not a difference between real mode and protected mode

- a. in real mode the addressable memory is more than in protected mode ✔
- b. in real mode general purpose registers are 16 bit, in protected mode they are 32 bit
- c. in real mode the addressable memory is less than in protected mode
- d. in real mode the segment is multiplied by 16, in protected mode segment is used as index in GDT
- e. processor starts in real mode

The correct answer is: in real mode the addressable memory is more than in protected mode

◄ Surprise Quiz - 2

Jump to...

Questions for test on kalloc/kfree/kvmalloc, etc. ►

Started on	Monday, 24 January 2022, 7:03:45 PM
State	Finished
Completed on	Monday, 24 January 2022, 9:26:15 PM
Time taken	2 hours 22 mins
Grade	**17.90** out of 20.00 (**90**%)

Question **1**

Complete

Mark 1.00 out of 1.00

How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security) ?

Select one:

- a. It disallows hardware interrupts when a process is running
- b. It prohibits invocation of kernel code completely, if a user program is running
- c. It prohibits one process from accessing other process's memory
- d. It prohibits a user mode process from running privileged instructions

The correct answer is: It prohibits a user mode process from running privileged instructions

Question **2**

Complete

Mark 2.00 out of 2.00

Match the program with it's output (ignore newlines in the output. Just focus on the count of the number of 'hi')

main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }	hi
main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }	hi
main() { int i = NULL; fork(); printf("hi\n"); }	hi hi
main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }	hi hi

The correct answer is: main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi, main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi, main() { int i = NULL; fork(); printf("hi\n"); } → hi hi, main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi hi

Question **3**

Complete

Mark 1.00 out of 1.00

Rank the following storage systems from slowest (first) to fastest(last)

You can drag and drop the items below/above each other.

- Magnetic tapes
- Optical disk
- Hard-disk drives
- Nonvolatile memory
- Main memory
- Cache
- Registers

Question **4**

Complete

Mark 1.00 out of 1.00

Why should a program exist in memory before it starts executing ?

- a. Because the variables of the program are stored in memory
- b. Because the hard disk is a slow medium
- c. Because the memory is volatile
- d. Becase the processor can run instructions and access data only from memory

The correct answer is: Becase the processor can run instructions and access data only from memory

Question **5**

Complete

Mark 2.00 out of 2.00

xv6.img: bootblock kernel

```
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
```

Consider above lines from the Makefile. Which of the following is incorrect?

- [x] a. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk.
- [x] b. The size of the kernel file is nearly 5 MB
- [x] c. xv6.img is the virtual processor used by the qemu emulator
- [ ] d. Blocks in xv6.img after kernel may be all zeroes.
- [ ] e. The xv6.img is the virtual disk that is created by combining the bootblock and the kernel file.
- [ ] f. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies 10,000 blocks on the disk.
- [ ] g. The size of the xv6.img is nearly 5 MB
- [ ] h. The kernel is located at block-1 of the xv6.img
- [ ] i. The bootblock is located on block-0 of the xv6.img
- [x] j. The size of xv6.img is exactly = (size of bootblock) + (size of kernel)
- [ ] k. The bootblock may be 512 bytes or less (looking at the Makefile instruction)

The correct answers are: xv6.img is the virtual processor used by the qemu emulator, The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk., The size of the kernel file is nearly 5 MB, The size of xv6.img is exactly = (size of bootblock) + (size of kernel)

Question **6**

Complete

Mark 0.60 out of 1.00

Select all the correct statements about two modes of CPU operation

Select one or more:

- [ ] a. Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode
- [x] b. There is an instruction like 'iret' to return from kernel mode to user mode
- [ ] c. The two modes are essential for a multiprogramming system
- [x] d. The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously
- [x] e. The two modes are essential for a multitasking system

The correct answers are: The two modes are essential for a multiprogramming system, The two modes are essential for a multitasking system, There is an instruction like 'iret' to return from kernel mode to user mode, The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously, Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode

Question **7**

Complete

Mark 0.50 out of 0.50

Order the following events in boot process (from 1 onwards)

BIOS	1
Init	4
Boot loader	2
OS	3
Login interface	5
Shell	6

The correct answer is: BIOS → 1, Init → 4, Boot loader → 2, OS → 3, Login interface → 5, Shell → 6

Question **8**

Complete

Mark 2.00 out of 2.00

Which of the following are NOT a part of job of a typical compiler?

- [ ] a. Invoke the linker to link the function calls with their code, extern globals with their declaration
- [ ] b. Check the program for syntactical errors
- [x] c. Suggest alternative pieces of code that can be written
- [x] d. Check the program for logical errors
- [ ] e. Process the # directives in a C program
- [ ] f. Convert high level langauge code to machine code

The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

Question **9**

Complete

Mark 0.50 out of 0.50

Is the terminal a part of the kernel on GNU/Linux systems?

- ( ) a. yes
- (•) b. no

The correct answer is: no

Question **10**

Complete

Mark 0.50 out of 0.50

Is the command "cat README > done &" possible on xv6? (Note the & in the end)

- (•) a. yes
- ( ) b. no

The correct answer is: yes

Question **11**

Complete

Mark 0.50 out of 0.50

Compare multiprogramming with multitasking

- (●) a. A multiprogramming system is not necessarily multitasking
- ( ) b. A multitasking system is not necessarily multiprogramming

The correct answer is: A multiprogramming system is not necessarily multitasking

Question **12**

Complete

Mark 0.80 out of 1.00

Match the register with the segment used with it.

esi	ds
eip	cs
edi	ds
esp	ss
ebp	ss

The correct answer is: esi → ds, eip → cs, edi → es, esp → ss, ebp → ss

Question **13**

Complete

Mark 0.83 out of 2.00

Select all statements that correctly explain the use/purpose of system calls.

Select one or more:

- [ ] a. Run each instruction of an application program
- [x] b. Provide an environment for process creation
- [x] c. Allow I/O device access to user processes
- [x] d. Provide services for accessing files
- [x] e. Handle ALL types of interrupts
- [ ] f. Switch from user mode to kernel mode
- [ ] g. Handle exceptions like division by zero

The correct answers are: Switch from user mode to kernel mode, Provide services for accessing files, Allow I/O device access to user processes, Provide an environment for process creation

Question **14**

Complete

Mark 1.67 out of 2.00

Which of the following instructions should be privileged?

Select one or more:

- [x] a. Set value of timer.
- [ ] b. Read the clock.
- [x] c. Access I/O device.
- [x] d. Access memory management unit of the processor
- [ ] e. Access a general purpose register
- [x] f. Turn off interrupts.
- [x] g. Modify entries in device-status table
- [ ] h. Set value of a memory location
- [ ] i. Switch from user to kernel mode.

The correct answers are: Set value of timer., Access memory management unit of the processor, Turn off interrupts., Modify entries in device-status table, Access I/O device., Switch from user to kernel mode.

Question **15**

Complete

Mark 1.00 out of 1.00

```
int value = 5;
int main()
{
   pid_t pid;
   pid = fork();
   if (pid == 0) { /* child process */
      value += 15;
      return 0;
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("%d", value); /* LINE A */
   }
   return 0;
}
```

What's the value printed here at LINE A?

Answer: 5

The correct answer is: 5

Question **16**

Complete

Mark 2.00 out of 2.00

Select all the correct statements about calling convention on x86 32-bit.

- [ ] a. Paramters are pushed on the stack in left-right order
- [x] b. Return address is one location above the ebp
- [ ] c. Space for local variables is allocated by substracting the stack pointer inside the code of the caller function
- [x] d. Space for local variables is allocated by substracting the stack pointer inside the code of the called function
- [ ] e. The two lines in the beginning of each function, "push %ebp; mov %esp, %ebp", create space for local variables
- [x] f. during execution of a function, ebp is pointing to the old ebp
- [ ] g. The return value is either stored on the stack or returned in the eax register
- [x] h. Parameters may be passed in registers or on stack
- [x] i. Compiler may allocate more memory on stack than needed
- [x] j. The ebp pointers saved on the stack constitute a chain of activation records
- [x] k. Parameters may be passed in registers or on stack

The correct answers are: Compiler may allocate more memory on stack than needed, Parameters may be passed in registers or on stack, Parameters may be passed in registers or on stack, Return address is one location above the ebp, during execution of a function, ebp is pointing to the old ebp, Space for local variables is allocated by substracting the stack pointer inside the code of the called function, The ebp pointers saved on the stack constitute a chain of activation records

◄ (Task) Compulsory xv6 task

Jump to...

(Optional Assignment) Shell Programming(Conformance tests) ►

Started on	Wednesday, 9 February 2022, 7:01:59 PM
State	Finished
Completed on	Wednesday, 9 February 2022, 8:01:50 PM
Time taken	59 mins 51 secs
Grade	**10.00** out of 11.00 (**91**%)

Question **1**
Complete
Mark 1.00 out of 1.00

The ljmp instruction in general does

- (selected) a. change the CS and EIP to 32 bit mode, and jumps to new value of EIP
- b. change the CS and EIP to 32 bit mode, and jumps to next line of code
- c. change the CS and EIP to 32 bit mode
- d. change the CS and EIP to 32 bit mode, and jumps to kernel code

The correct answer is: change the CS and EIP to 32 bit mode, and jumps to new value of EIP

Question **2**
Complete
Mark 1.00 out of 1.00

The kernel is loaded at Physical Address

- a. 0x0010000
- b. 0x80000000
- (selected) c. 0x00100000
- d. 0x80100000

The correct answer is: 0x00100000

Question **3**

Complete

Mark 1.00 out of 1.00

The number of GDT entries setup during boot process of xv6 is

- a. 0
- b. 3
- c. 256
- d. 2
- e. 255
- f. 4

The correct answer is: 3

Question **4**

Complete

Mark 1.00 out of 1.00

The variable $stack in entry.S is

- a. located at less than 0x7c00
- b. a memory region allocated as a part of entry.S
- c. located at the value given by %esp as setup by bootmain()
- d. located at 0x7c00
- e. located at 0

The correct answer is: a memory region allocated as a part of entry.S

Question **5**

Complete

Mark 1.00 out of 1.00

which of the following is not a difference between real mode and protected mode

- a. in real mode the segment is multiplied by 16, in protected mode segment is used as index in GDT
- b. in real mode general purpose registers are 16 bit, in protected mode they are 32 bit
- c. in real mode the addressable memory is more than in protected mode
- d. processor starts in real mode
- e. in real mode the addressable memory is less than in protected mode

The correct answer is: in real mode the addressable memory is more than in protected mode

Question **6**

Complete

Mark 1.00 out of 1.00

The kernel ELF file contains how many Program headers?

- a. 2
- b. 4
- c. 9
- d. 3
- e. 10

The correct answer is: 3

Question **7**

Complete

Mark 1.00 out of 1.00

ELF Magic number is

- a. 0xELF
- b. 0xFFFFFFFFF
- c. 0
- d. 0x0x464CELF
- e. 0xELFELFELF
- f. 0x464C457FL
- g. 0x464C457FU

The correct answer is: 0x464C457FU

Question **8**

Complete

Mark 1.00 out of 1.00

x86 provides which of the following type of memory management options?

- a. segmentation and two level paging
- b. segmentation and one level paging
- c. segmentation only
- d. segmentation or one or two level paging
- e. segmentation and one or two level paging
- f. segmentation or paging

The correct answer is: segmentation and one or two level paging

Question **9**

Complete

Mark 1.00 out of 1.00

Why is the code of entry() in Assembly and not in C?

- a. There is no particular reason, it could also be in C
- b. Because the kernel code must begin in assembly
- c. Because it needs to setup paging
- d. Because the symbol entry() is inside the ELF file

The correct answer is: Because it needs to setup paging

Question **10**

Not answered

Marked out of 0.50

code line, MMU setting: Match the line of xv6 code with the MMU setup employed

Answer:

The correct answer is: inb $0x64,%al

Question **11**

Not answered

Marked out of 0.50

Match the pairs of which action is taken by whom

Answer:

The correct answer is: kernel

Question **12**

Complete

Mark 1.00 out of 1.00

The right side of line of code "entry = (void(*)(void))(elf->entry)" means

- a. Get the "entry" in ELF structure and convert it into a function pointer accepting no arguments and returning nothing
- b. Get the "entry" in ELF structure and convert it into a function void pointer
- c. Convert the "entry" in ELF structure into void
- d. Get the "entry" in ELF structure and convert it into a void pointer

The correct answer is: Get the "entry" in ELF structure and convert it into a function pointer accepting no arguments and returning nothing

◀ Homework questions: Basics of MM, xv6 booting

Jump to...

(Code) Files, redirection, dup, (IPC)pipe ▶

Started on	Monday, 21 February 2022, 7:00:44 PM
State	Finished
Completed on	Monday, 21 February 2022, 7:55:40 PM
Time taken	54 mins 56 secs
Grade	**9.00** out of 10.00 (**90**%)

Question **1**
Complete
Mark 0.50 out of 0.50

Which of the following state transitions are not possible?

- [x] a. Waiting -> Terminated
- [ ] b. Running -> Waiting
- [x] c. Ready -> Terminated
- [x] d. Ready -> Waiting

The correct answers are: Ready -> Terminated, Waiting -> Terminated, Ready -> Waiting

Question **2**
Complete
Mark 1.00 out of 1.00

Select the odd one out

- a. Kernel stack of running process to kernel stack of scheduler
- b. Kernel stack of new process to Process stack of new process
- c. Kernel stack of new process to kernel stack of scheduler
- d. Kernel stack of scheduler to kernel stack of new process
- e. Process stack of running process to kernel stack of running process

The correct answer is: Kernel stack of new process to kernel stack of scheduler

Question **3**

Complete

Mark 0.50 out of 0.50

Match the names of PCB structures with kernel

linux	struct task_struct
xv6	struct proc

The correct answer is: linux → struct task_struct, xv6 → struct proc

Question **4**

Complete

Mark 1.00 out of 1.00

The "push 0" in vectors.S is

- a. To indicate that it's a system call and not a hardware interrupt
- b. To be filled in as the return value of the system call
- c. Place for the error number value
- d. A placeholder to match the size of struct trapframe

The correct answer is: Place for the error number value

Question **5**

Complete

Mark 0.50 out of 0.50

Match the MACRO with it's meaning

KERNLINK	2.1 GB
PHYSTOP	224 MB
KERNBASE	2 GB

The correct answer is: KERNLINK → 2.224 GB, PHYSTOP → 224 MB, KERNBASE → 2 GB

Question **6**

Complete

Mark 0.50 out of 0.50

Match the File descriptors to their meaning

1	Standard output
0	Standard Input
2	Standard error

The correct answer is: 1 → Standard output, 0 → Standard Input, 2 → Standard error

Question **7**

Complete

Mark 1.00 out of 1.00

Match the elements of C program to their place in memory

Code of main()	Code
Function code	Code
Local Static variables	Data
#define MACROS	No Memory needed
#include files	No memory needed
Global variables	Data
Malloced Memory	Heap
Arguments	Stack
Global Static variables	Data
Local Variables	Stack

The correct answer is: Code of main() → Code, Function code → Code, Local Static variables → Data, #define MACROS → No Memory needed, #include files → No memory needed, Global variables → Data, Malloced Memory → Heap, Arguments → Stack, Global Static variables → Data, Local Variables → Stack

Question **8**

Complete

Mark 1.00 out of 1.00

A process blocks itself means

- (selected) a. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler
- b. The kernel code of system call calls scheduler
- c. The kernel code of an interrupt handler, moves the process to a waiting queue and calls scheduler
- d. The application code calls the scheduler

The correct answer is: The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

Question **9**

Complete

Mark 0.00 out of 1.00

The trapframe, in xv6, is built by the

- a. hardware, vectors.S
- b. hardware, vectors.S, trapasm.S
- (selected) c. hardware, trapasm.S
- d. vectors.S, trapasm.S
- e. hardware, vectors.S, trapasm.S, trap()

The correct answer is: hardware, vectors.S, trapasm.S

Question **10**

Complete

Mark 1.00 out of 1.00

Which of the following is not a task of the code of swtch() function

- [ ] a. Load the new context
- [x] b. Save the return value of the old context code
- [ ] c. Save the old context
- [ ] d. Switch stacks
- [ ] e. Jump to next context EIP
- [x] f. Change the kernel stack location

The correct answers are: Save the return value of the old context code, Change the kernel stack location

Question **11**

Complete

Mark 1.00 out of 1.00

What will be the output of this program

```
int main() {
  int fd;
  printf("%d ",   open("/etc/passwd", O_RDONLY));
  close(1);
  fd = printf("%d ", open("/etc/passwd", O_RDONLY));
  close(fd);
   fd = printf("%d ", open("/etc/passwd", O_RDONLY));
}
```

- a. 3 3 3
- b. 3 4 5
- c. 2 2 2
- d. 3 1 2
- e. 1 1 1
- f. 3 1 1

The correct answer is: 3 1 1

Question **12**

Complete

Mark 1.00 out of 1.00

Arrange in correct order, the files involved in execution of system call

trapasm.S	3
vectors.S	2
trap.c	4
usys.S	1

The correct answer is: trapasm.S → 3, vectors.S → 2, trap.c → 4, usys.S → 1

◄ Description of some possible course mini projects

Jump to...

(Code) mmap related programs ►

Started on	Saturday, 26 February 2022, 5:18:28 PM
State	Finished
Completed on	Saturday, 26 February 2022, 6:33:29 PM
Time taken	1 hour 15 mins
Grade	**13.71** out of 15.00 (**91**%)

Question **1**

Complete

Mark 0.50 out of 0.50

Map the technique with it's feature/problem

static linking	large executable file
dynamic linking	small executable file
static loading	wastage of physical memory
dynamic loading	allocate memory only if needed

The correct answer is: static linking → large executable file, dynamic linking → small executable file, static loading → wastage of physical memory, dynamic loading → allocate memory only if needed

Question **2**

Complete

Mark 0.86 out of 1.00

Compare paging with demand paging and select the correct statements.

Select one or more:

- [x] a. Both demand paging and paging support shared memory pages.
- [x] b. Paging requires some hardware support in CPU
- [ ] c. With paging, it's possible to have user programs bigger than physical memory.
- [x] d. Demand paging always increases effective memory access time.
- [x] e. The meaning of valid-invalid bit in page table is different in paging and demand-paging.
- [ ] f. Demand paging requires additional hardware support, compared to paging.
- [x] g. Calculations of number of bits for page number and offset are same in paging and demand paging.
- [x] h. With demand paging, it's possible to have user programs bigger than physical memory.
- [ ] i. TLB hit ration has zero impact in effective memory access time in demand paging.
- [ ] j. Paging requires NO hardware support in CPU

The correct answers are: Demand paging requires additional hardware support, compared to paging., Both demand paging and paging support shared memory pages., With demand paging, it's possible to have user programs bigger than physical memory., Demand paging always increases effective memory access time., Paging requires some hardware support in CPU, Calculations of number of bits for page number and offset are same in paging and demand paging., The meaning of valid-invalid bit in page table is different in paging and demand-paging.

Question **3**

Complete

Mark 1.00 out of 1.00

Page sizes are a power of 2 because

Select one:

- (●) a. Certain bits are reserved for offset in logical address. Hence page size = 2^(no.of offset bits)
- ( ) b. MMU only understands numbers that are power of 2
- ( ) c. Power of 2 calculations are highly efficient
- ( ) d. operating system calculations happen using power of 2
- ( ) e. Certain bits are reserved for offset in logical address. Hence page size = 2^(32 - no.of offset bits)

The correct answer is: Certain bits are reserved for offset in logical address. Hence page size = 2^(no.of offset bits)

Question **4**

Complete

Mark 1.00 out of 1.00

which of the following, do you think, are valid concerns for making the kernel pageable?

- [ ] a. No part of kernel code should be pageable.
- [x] b. The disk driver and disk interrupt handler should not be pageable
- [ ] c. No data structure of kernel should be pageable
- [x] d. The kernel's own page tables should not be pageable
- [x] e. The page fault handler should not be pageable
- [x] f. The kernel must have some dedicated frames for it's own work

The correct answers are: The kernel's own page tables should not be pageable, The page fault handler should not be pageable, The kernel must have some dedicated frames for it's own work, The disk driver and disk interrupt handler should not be pageable

Question **5**

Complete

Mark 1.00 out of 1.00

Suppose two processes share a library between them. The library consists of 5 pages, and these 5 pages are mapped to frames 9, 15, 23, 4, 7 respectively. Process P1 has got 6 pages, first 3 of which consist of process's own code/data and 3 correspond to library's pages 0, 2, 4. Process P2 has got 7 pages, first 3 of which consist of processe's own code/data and remaining 4 correspond to library's pages 0, 1, 3, 4. Fill in the blanks for page table entries of P1 and P2.

Page table of P2, Page 3	9
Page table of P2, Page 1	3
Page table of P1, Page 3	9
Page table of P2, Page 0	2
Page table of P1, Page 5	7
Page table of P1, Page 4	23
Page table of P2, Page 4	15

The correct answer is: Page table of P2, Page 3 → 4, Page table of P2, Page 1 → 15, Page table of P1, Page 3 → 9, Page table of P2, Page 0 → 9, Page table of P1, Page 5 → 7, Page table of P1, Page 4 → 23, Page table of P2, Page 4 → 7

Question **6**

Complete

Mark 1.00 out of 1.00

Order the following events, related to page fault handling, in correct order

1. MMU detects that a page table entry is marked "invalid"
2. Page fault interrupt is generated
3. Page fault handler in kernel starts executing
4. Page fault handler detects that it's a page fault and not illegal memory access
5. Empty frame is found
6. Disk read is issued
7. Page faulting process is made to wait in a queue
8. Other processes scheduled by scheduler
9. Disk Interrupt occurs
10. Disk interrupt handler runs
11. Page table of page faulted process is updated
12. Page faulted process is moved to ready-queue

Question **7**

Complete

Mark 0.83 out of 1.00

Given six memory partitions of 300 KB , 600 KB , 350 KB , 200 KB , 750 KB , and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB and 500 KB (in order)?

first fit 115 KB	300 KB
worst fit 115 KB	750 KB
best fit 500 KB	600 KB
worst fit 500 KB	600 KB
best fit 115 KB	125 KB
first fit 500 KB	600 KB

The correct answer is: first fit 115 KB → 300 KB, worst fit 115 KB → 750 KB, best fit 500 KB → 600 KB, worst fit 500 KB → 635 KB, best fit 115 KB → 125 KB, first fit 500 KB → 600 KB

Question **8**

Complete

Mark 1.00 out of 1.00

For the reference string
3 4 3 5 2

the number of page faults (including initial ones) using

FIFO replacement and 2 page frames is :

4

FIFO replacement and 3 page frames is :

4

Question **9**

Complete

Mark 1.00 out of 1.00

Select all the correct statements, w.r.t. Copy on Write

- [x] a. COW helps us save memory
- [x] b. If either parent or child modifies a COW-page, then a copy of the page is made and page table entry is updated
- [ ] c. use of COW during fork() is useless if exec() is called by the child
- [x] d. Vfork() assumes that there will be no write, but rather exec()
- [ ] e. use of COW during fork() is useless if child called exit()
- [x] f. Fork() used COW technique to improve performance of new process creation.

The correct answers are: Fork() used COW technique to improve performance of new process creation., If either parent or child modifies a COW-page, then a copy of the page is made and page table entry is updated, COW helps us save memory, Vfork() assumes that there will be no write, but rather exec()

Question **10**

Complete

Mark 1.00 out of 1.00

W.r.t the figure given below, mark the given statements as True or False.

Figure 10.9 Need for page replacement.

True	False	
◉	○	Global replacement means chose any of the frame from 2 to 7
◉	○	Handling this scenario demands two disk I/Os
◉	○	The kernel's pages can not used for replacement if kernel is not pageable.
◉	○	Kernel occupies two page frames
◉	○	Local replacement means chose any of the frames 2, 3, 6
◉	○	Page 1 of process 1 needs a replacement
○	◉	Local replacement means chose any of the frame from 2 to 7
○	◉	Global replacement means chose any of the frame from 0 to 7

Global replacement means chose any of the frame from 2 to 7: True
Handling this scenario demands two disk I/Os: True
The kernel's pages can not used for replacement if kernel is not pageable.: True
Kernel occupies two page frames: True
Local replacement means chose any of the frames 2, 3, 6: True
Page 1 of process 1 needs a replacement: True
Local replacement means chose any of the frame from 2 to 7: False
Global replacement means chose any of the frame from 0 to 7: False

Question **11**

Complete

Mark 0.60 out of 1.00

Given below is the "maps" file for a particular instance of "vim.basic" process.

Mark the given statements as True or False, w.r.t. the contents of the map file.

```
55a43501b000-55a435049000 r--p 00000000 103:05 917529                    /usr/bin/vim.basic
55a435049000-55a435248000 r-xp 0002e000 103:05 917529                    /usr/bin/vim.basic
55a435248000-55a4352b6000 r--p 0022d000 103:05 917529                    /usr/bin/vim.basic
55a4352b7000-55a4352c5000 r--p 0029b000 103:05 917529                    /usr/bin/vim.basic
55a4352c5000-55a4352e2000 rw-p 002a9000 103:05 917529                    /usr/bin/vim.basic
55a4352e2000-55a4352f0000 rw-p 00000000 00:00 0
55a436bc9000-55a436e5b000 rw-p 00000000 00:00 0                          [heap]
7f275b0a3000-7f275b0a6000 r--p 00000000 103:05 917901                    /usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so
7f275b0a6000-7f275b0ad000 r-xp 00003000 103:05 917901                    /usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so
7f275b0ad000-7f275b0af000 r--p 0000a000 103:05 917901                    /usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so
7f275b0af000-7f275b0b0000 r--p 0000b000 103:05 917901                    /usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so
7f275b0b0000-7f275b0b1000 rw-p 0000c000 103:05 917901                    /usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so
7f275b0b1000-7f275b0b7000 rw-p 00000000 00:00 0
7f275b0b7000-7f275b8f5000 r--p 00000000 103:05 925247                    /usr/lib/locale/locale-archive
7f275b8f5000-7f275b8fa000 rw-p 00000000 00:00 0
7f275b8fa000-7f275b8fc000 r--p 00000000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b8fc000-7f275b901000 r-xp 00002000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b901000-7f275b904000 r--p 00007000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b904000-7f275b905000 ---p 0000a000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b905000-7f275b906000 r--p 0000a000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b906000-7f275b907000 rw-p 0000b000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b907000-7f275b90a000 r--p 00000000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b90a000-7f275b921000 r-xp 00003000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b921000-7f275b932000 r--p 0001a000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b932000-7f275b933000 ---p 0002b000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b933000-7f275b934000 r--p 0002b000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b934000-7f275b935000 rw-p 0002c000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b935000-7f275b937000 rw-p 00000000 00:00 0
7f275b937000-7f275b938000 r--p 00000000 103:05 917914                    /usr/lib/x86_64-linux-
gnu/libutil-2.31.so
7f275b938000-7f275b939000 r-xp 00001000 103:05 917914                    /usr/lib/x86_64-linux-
gnu/libutil-2.31.so
7f275b939000-7f275b93a000 r--p 00002000 103:05 917914                    /usr/lib/x86_64-linux-
gnu/libutil-2.31.so
7f275b93a000-7f275b93b000 r--p 00002000 103:05 917914                    /usr/lib/x86_64-linux-
```

```
gnu/libutil-2.31.so
7f275b93b000-7f275b93c000 rw-p 00003000 103:05 917914                    /usr/lib/x86_64-linux-
gnu/libutil-2.31.so
7f275b93c000-7f275b93e000 r--p 00000000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b93e000-7f275b94f000 r-xp 00002000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b94f000-7f275b955000 r--p 00013000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b955000-7f275b956000 ---p 00019000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b956000-7f275b957000 r--p 00019000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b957000-7f275b958000 rw-p 0001a000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b958000-7f275b95c000 r--p 00000000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b95c000-7f275b978000 r-xp 00004000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b978000-7f275b982000 r--p 00020000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b982000-7f275b983000 ---p 0002a000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b983000-7f275b985000 r--p 0002a000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b985000-7f275b986000 rw-p 0002c000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b986000-7f275b988000 r--p 00000000 103:05 924057                    /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b988000-7f275b98d000 r-xp 00002000 103:05 924057                    /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b98d000-7f275b98f000 r--p 00007000 103:05 924057                    /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b98f000-7f275b990000 r--p 00008000 103:05 924057                    /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b990000-7f275b991000 rw-p 00009000 103:05 924057                    /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b991000-7f275b995000 r--p 00000000 103:05 921934                    /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b995000-7f275b9a3000 r-xp 00004000 103:05 921934                    /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b9a3000-7f275b9a9000 r--p 00012000 103:05 921934                    /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b9a9000-7f275b9aa000 r--p 00017000 103:05 921934                    /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b9aa000-7f275b9ab000 rw-p 00018000 103:05 921934                    /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b9ab000-7f275b9ad000 rw-p 00000000 00:00 0
7f275b9ad000-7f275b9af000 r--p 00000000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9af000-7f275b9b4000 r-xp 00002000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b4000-7f275b9b5000 r--p 00007000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b5000-7f275b9b6000 ---p 00008000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b6000-7f275b9b7000 r--p 00008000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b7000-7f275b9b8000 rw-p 00009000 103:05 924631                    /usr/lib/x86_64-linux-
```

```
gnu/libvorbisfile.so.3.3.7
7f275b9b8000-7f275b9ba000 r--p 00000000 103:05 924277                    /usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0
7f275b9ba000-7f275ba1e000 r-xp 00002000 103:05 924277                    /usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0
7f275ba1e000-7f275ba46000 r--p 00066000 103:05 924277                    /usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0
7f275ba46000-7f275ba47000 r--p 0008d000 103:05 924277                    /usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0
7f275ba47000-7f275ba48000 rw-p 0008e000 103:05 924277                    /usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0
7f275ba48000-7f275ba6d000 r--p 00000000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275ba6d000-7f275bbe5000 r-xp 00025000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275bbe5000-7f275bc2f000 r--p 0019d000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275bc2f000-7f275bc30000 ---p 001e7000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275bc30000-7f275bc33000 r--p 001e7000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275bc33000-7f275bc36000 rw-p 001ea000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275bc36000-7f275bc3a000 rw-p 00000000 00:00 0
7f275bc3a000-7f275bc41000 r--p 00000000 103:05 917906                    /usr/lib/x86_64-linux-
gnu/libpthread-2.31.so
7f275bc41000-7f275bc52000 r-xp 00007000 103:05 917906                    /usr/lib/x86_64-linux-
gnu/libpthread-2.31.so
7f275bc52000-7f275bc57000 r--p 00018000 103:05 917906                    /usr/lib/x86_64-linux-
gnu/libpthread-2.31.so
7f275bc57000-7f275bc58000 r--p 0001c000 103:05 917906                    /usr/lib/x86_64-linux-
gnu/libpthread-2.31.so
7f275bc58000-7f275bc59000 rw-p 0001d000 103:05 917906                    /usr/lib/x86_64-linux-
gnu/libpthread-2.31.so
7f275bc59000-7f275bc5d000 rw-p 00000000 00:00 0
7f275bc5d000-7f275bcce000 r--p 00000000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275bcce000-7f275bf29000 r-xp 00071000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275bf29000-7f275c142000 r--p 002cc000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275c142000-7f275c143000 ---p 004e5000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275c143000-7f275c149000 r--p 004e5000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275c149000-7f275c190000 rw-p 004eb000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275c190000-7f275c1b3000 rw-p 00000000 00:00 0
7f275c1b3000-7f275c1b4000 r--p 00000000 103:05 917894                    /usr/lib/x86_64-linux-
gnu/libdl-2.31.so
7f275c1b4000-7f275c1b6000 r-xp 00001000 103:05 917894                    /usr/lib/x86_64-linux-
gnu/libdl-2.31.so
7f275c1b6000-7f275c1b7000 r--p 00003000 103:05 917894                    /usr/lib/x86_64-linux-
gnu/libdl-2.31.so
7f275c1b7000-7f275c1b8000 r--p 00003000 103:05 917894                    /usr/lib/x86_64-linux-
gnu/libdl-2.31.so
7f275c1b8000-7f275c1b9000 rw-p 00004000 103:05 917894                    /usr/lib/x86_64-linux-
gnu/libdl-2.31.so
7f275c1b9000-7f275c1bb000 rw-p 00000000 00:00 0
```

```
7f275c1bb000-7f275c1c0000 r-xp 00000000 103:05 923815                    /usr/lib/x86_64-linux-
gnu/libgpm.so.2
7f275c1c0000-7f275c3bf000 ---p 00005000 103:05 923815                    /usr/lib/x86_64-linux-
gnu/libgpm.so.2
7f275c3bf000-7f275c3c0000 r--p 00004000 103:05 923815                    /usr/lib/x86_64-linux-
gnu/libgpm.so.2
7f275c3c0000-7f275c3c1000 rw-p 00005000 103:05 923815                    /usr/lib/x86_64-linux-
gnu/libgpm.so.2
7f275c3c1000-7f275c3c3000 r--p 00000000 103:05 923315                    /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3c3000-7f275c3c8000 r-xp 00002000 103:05 923315                    /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3c8000-7f275c3ca000 r--p 00007000 103:05 923315                    /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3ca000-7f275c3cb000 r--p 00008000 103:05 923315                    /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3cb000-7f275c3cc000 rw-p 00009000 103:05 923315                    /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3cc000-7f275c3cf000 r--p 00000000 103:05 923446                    /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3cf000-7f275c3d9000 r-xp 00003000 103:05 923446                    /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3d9000-7f275c3dd000 r--p 0000d000 103:05 923446                    /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3dd000-7f275c3de000 r--p 00010000 103:05 923446                    /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3de000-7f275c3df000 rw-p 00011000 103:05 923446                    /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3df000-7f275c3e5000 r--p 00000000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c3e5000-7f275c3fe000 r-xp 00006000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c3fe000-7f275c405000 r--p 0001f000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c405000-7f275c406000 ---p 00026000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c406000-7f275c407000 r--p 00026000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c407000-7f275c408000 rw-p 00027000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c408000-7f275c40a000 rw-p 00000000 00:00 0
7f275c40a000-7f275c418000 r--p 00000000 103:05 924540                    /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c418000-7f275c427000 r-xp 0000e000 103:05 924540                    /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c427000-7f275c435000 r--p 0001d000 103:05 924540                    /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c435000-7f275c439000 r--p 0002a000 103:05 924540                    /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c439000-7f275c43a000 rw-p 0002e000 103:05 924540                    /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c43a000-7f275c449000 r--p 00000000 103:05 917895                    /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c449000-7f275c4f0000 r-xp 0000f000 103:05 917895                    /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c4f0000-7f275c587000 r--p 000b6000 103:05 917895                    /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c587000-7f275c588000 r--p 0014c000 103:05 917895                    /usr/lib/x86_64-linux-
gnu/libm-2.31.so
```

```
7f275c588000-7f275c589000 rw-p 0014d000 103:05 917895                    /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c589000-7f275c58b000 rw-p 00000000 00:00 0
7f275c5ae000-7f275c5af000 r--p 00000000 103:05 917889                    /usr/lib/x86_64-linux-gnu/ld-
2.31.so
7f275c5af000-7f275c5d2000 r-xp 00001000 103:05 917889                    /usr/lib/x86_64-linux-gnu/ld-
2.31.so
7f275c5d2000-7f275c5da000 r--p 00024000 103:05 917889                    /usr/lib/x86_64-linux-gnu/ld-
2.31.so
7f275c5db000-7f275c5dc000 r--p 0002c000 103:05 917889                    /usr/lib/x86_64-linux-gnu/ld-
2.31.so
7f275c5dc000-7f275c5dd000 rw-p 0002d000 103:05 917889                    /usr/lib/x86_64-linux-gnu/ld-
2.31.so
7f275c5dd000-7f275c5de000 rw-p 00000000 00:00 0
7ffd22d2f000-7ffd22d50000 rw-p 00000000 00:00 0                          [stack]
7ffd22db0000-7ffd22db4000 r--p 00000000 00:00 0                          [vvar]
7ffd22db4000-7ffd22db6000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

True	False	
◉	○	This is a virtual memory map (not physical memory map)
◉	○	vim.basic uses the math library
◉	○	The size of the stack is one page
◉	○	The 5th entry 55a4352c5000-55a4352e2000 **may** correspond to "data" of the vim.basic
◉	○	The size of the heap is one page

This is a virtual memory map (not physical memory map): True
vim.basic uses the math library: True
The size of the stack is one page: False
The 5th entry 55a4352c5000-55a4352e2000 **may** correspond to "data" of the vim.basic: True
The size of the heap is one page: False

Question **12**
Complete
Mark 1.00 out of 1.00

Calculate the EAT in NANO-seconds (upto 2 decimal points) w.r.t. a page fault, given

Memory access time = 191 ns

Average page fault service time = 11 ms

Page fault rate = 0.5

Answer: 5500095

The correct answer is: 5500095.50

Question **13**

Complete

Mark 1.00 out of 1.00

Assuming a 8- KB page size, what is the page numbers for the address 428517 reference in decimal :

(give answer also in decimal)

Answer: 52

The correct answer is: 52

Question **14**

Complete

Mark 0.67 out of 1.00

Shared memory is possible with which of the following memory management schemes ?

Select one or more:

- [x] a. demand paging
- [ ] b. continuous memory management
- [x] c. paging
- [ ] d. segmentation

The correct answers are: paging, segmentation, demand paging

Question **15**

Complete

Mark 0.50 out of 0.50

Map the parts of a C code to the memory regions they are related to

global un-initialized variables	bss
static variables	data
local variables	stack
function arguments	stack
global initialized variables	data
malloced memory	heap
functions	code

The correct answer is: global un-initialized variables → bss, static variables → data, local variables → stack, function arguments → stack, global initialized variables → data, malloced memory → heap, functions → code

Question **16**

Complete

Mark 0.75 out of 1.00

Given below is the output of the command "ps -eo min_flt,maj_flt,cmd" on a Linux Desktop system. Select the statements that are consistent with the output

```
626729 482768 /usr/lib/firefox/firefox -contentproc -parentBuildID 20220202182137 -prefsLen 9256 -prefMapSize 264738 -appDir /usr/lib/firefox/browser 6094 true rdd
2167 687 /usr/sbin/apache2 -k start
1265185 222 /usr/bin/gnome-shell
102648 111 /usr/sbin/mysqld
9813 0 bash
15497 370 /usr/bin/gedit --gapplication-service
```

- [ ] a. Apache web-server has not been doing much work
- [x] b. The bash shell is mostly busy doing work within a particular locality
- [x] c. Firefox has likely been running for a large amount of time
- [x] d. All of the processes here exihibit some good locality of reference

The correct answers are: Firefox has likely been running for a large amount of time, Apache web-server has not been doing much work, The bash shell is mostly busy doing work within a particular locality, All of the processes here exihibit some good locality of reference

◄ (Code) mmap related programs

Jump to...

Points from Mid-term feedback ►

Started on	Monday, 7 March 2022, 7:00:14 PM
State	Finished
Completed on	Monday, 7 March 2022, 8:20:00 PM
Time taken	1 hour 19 mins
Grade	**12.33** out of 15.00 (**82**%)

Question **1**

Complete

Mark 1.00 out of 1.00

Why is there a call to kinit2? Why is it not merged with knit1?

- a. Because there is a limit on the values that the argumets to knit1() can take.
- b. knit2 refers to virtual addresses beyond 4MB, which are not mapped before kvalloc() is called
- c. When kinit1() is called there is a need for few page frames, but later knit2() is called to serve need of more page frames
- d. call to seginit() makes it possible to actually use PHYSTOP in argument to kinit2()

The correct answer is: knit2 refers to virtual addresses beyond 4MB, which are not mapped before kvalloc() is called

Question **2**

Complete

Mark 1.00 out of 1.00

Map the virtual address to physical address in xv6

80108000	0x108000
KERNLINK	0x100000
0xFE000000	0xFE000000
KERNBASE	0

The correct answer is: 80108000 → 0x108000, KERNLINK → 0x100000, 0xFE000000 → 0xFE000000, KERNBASE → 0

Question **3**

Complete

Mark 1.50 out of 1.50

Arrange the following in the correct order of execution (w.r.t. 'init')

initcode() returns from trapret()	7
mpmain() calls scheduler()	4
initcode() returns in forkret()	6
userinit() is called	1
'initcode' process is marked RUNNABLE	3
initcode() calls exec("/init", ...)	8
'initcode' struct proc is created	2
scheduler() schedules initcode() process	5

The correct answer is: initcode() returns from trapret() → 7, mpmain() calls scheduler() → 4, initcode() returns in forkret() → 6, userinit() is called → 1, 'initcode' process is marked RUNNABLE → 3, initcode() calls exec("/init", ...) → 8, 'initcode' struct proc is created → 2, scheduler() schedules initcode() process → 5

Question **4**

Complete

Mark 1.00 out of 1.00

What does seginit() do?

- a. Nothing significant, just repetition of earlier GDT setup but with free frames list created now
- b. Adds two additional entries to GDT corresponding to Code and Data segments, but to be used in privilege level 3
- c. Nothing significant, just repetition of earlier GDT setup but with 2-level paging setup done
- d. Nothing significant, just repetition of earlier GDT setup but with kernel page table allocated now
- e. Adds two additional entries to GDT corresponding to Code and Data segments, but to be used in privilege level 0

The correct answer is: Adds two additional entries to GDT corresponding to Code and Data segments, but to be used in privilege level 3

Question **5**

Complete

Mark 1.00 out of 1.00

Which of the following is done by mappages()?

- [ ] a. allocate page directory if required
- [x] b. create page table mappings to the range given by "pa" and "pa + size"
- [x] c. allocate page table if required
- [ ] d. allocate page frame if required
- [x] e. create page table mappings for the range given by "va" and "va + size"

The correct answers are: create page table mappings for the range given by "va" and "va + size", allocate page table if required, create page table mappings to the range given by "pa" and "pa + size"

Question **6**

Complete

Mark 1.00 out of 1.00

What does userinit() do ?

- a. initializes the process 'init' and starts executing it
- b. sets up the 'initcode' process to start execution in trapret()
- c. initializes the users
- d. sets up the 'initcode' process to start execution in forkret ()
- e. sets up the 'initcode' process to start execution in forkret()
- f. sets up the 'init' process to start execution in forkret()

The correct answer is: sets up the 'initcode' process to start execution in forkret()

Question **7**

Complete

Mark 1.00 out of 1.00

The variable 'end' used as argument to kinit1 has the value

- a. 801154a8
- b. 80110000
- c. 80000000
- d. 8010a48c
- e. 80102da0
- f. 81000000

The correct answer is: 801154a8

Question **8**

Complete

Mark 0.00 out of 1.00

The approximate number of page frames created by kinit1 is

- a. 1000
- b. 4
- c. 16
- d. 2000
- e. 4000
- f. 3000
- g. 10

The correct answer is: 3000

Question **9**

Complete

Mark 0.67 out of 1.00

Select all the correct statements about initcode

- [x] a. initcode essentially calls exec("/init",...)
- [x] b. The data and stack of initcode is mapped to one single page in userinit()
- [x] c. the size of 'initcode' is 2c
- [ ] d. code of initcode is loaded at virtual address 0
- [x] e. code of 'initcode' is loaded along with the kernel during booting
- [x] f. code of initcode is loaded in memory by the kernel during userinit()
- [ ] g. initcode is the 'init' process

The correct answers are: code of 'initcode' is loaded along with the kernel during booting, the size of 'initcode' is 2c, The data and stack of initcode is mapped to one single page in userinit(), initcode essentially calls exec("/init",...)

Question **10**

Complete

Mark 1.00 out of 1.00

Does exec() code around clearptau() lead to wastage of one page frame?

- a. no
- b. yes

The correct answer is: yes

Question **11**

Complete

Mark 1.50 out of 1.50

Which of the following is DONE by allocproc() ?

- [ ] a. ensure that the process starts in trapret()
- [x] b. allocate kernel stack for the process
- [ ] c. setup kernel memory mappings for the process
- [ ] d. setup the contents of the trapframe of the process properly
- [x] e. ensure that the process starts in forkret()
- [x] f. setup the trapframe and context pointers appropriately
- [x] g. Select an UNUSED struct proc for use
- [x] h. allocate PID to the process

The correct answers are: Select an UNUSED struct proc for use, allocate PID to the process, allocate kernel stack for the process, setup the trapframe and context pointers appropriately, ensure that the process starts in forkret()

Question **12**

Complete

Mark 0.67 out of 2.00

exec() does this: curproc->tf->eip = elf.entry, but userinit() does this: p->tf->eip = 0; Select all the statements from below, that collectively explain this

- [ ] a. elf.entry is anyways 0, so both statements mean the same
- [ ] b. the 'entry' in initcode is anyways 0
- [ ] c. the initcode is created using objcopy, which discards all relocation information and symbols (like entry)
- [x] d. exec() loads from ELF file and the address of first instruction to be executed is given by 'entry'
- [x] e. In userinit() the function inituvm() has mapped the code of 'initcode' to be starting at virtual address 0
- [x] f. the code of 'initcode' is loaded at physical address 0

The correct answers are: exec() loads from ELF file and the address of first instruction to be executed is given by 'entry', In userinit() the function inituvm() has mapped the code of 'initcode' to be starting at virtual address 0, the initcode is created using objcopy, which discards all relocation information and symbols (like entry)

Question **13**

Complete

Mark 1.00 out of 1.00

Select the statement that most correctly describes what setupkvm() does

- ( ) a. creates a 2-level page table for the use of the kernel, as specified in gdtdesc
- (•) b. creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global arrray
- ( ) c. creates a 1-level page table for the use by the kernel, as specified in kmap[] global array
- ( ) d. creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global arrray and makes kpgdir point to it

The correct answer is: creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global arrray

◄ Questions for test on kalloc/kfree/kvmalloc, etc.

Jump to...

(Optional Assignment) Slab allocator in xv6 ►

Started on	Monday, 7 March 2022, 7:01:03 PM
State	Finished
Completed on	Monday, 7 March 2022, 8:20:01 PM
Time taken	1 hour 18 mins
Grade	**13.42** out of 15.00 (**89**%)

Question **1**

Complete

Mark 1.00 out of 1.00

Map the virtual address to physical address in xv6

80108000	0x108000
KERNLINK	0x100000
0xFE000000	0xFE000000
KERNBASE	0

The correct answer is: 80108000 → 0x108000, KERNLINK → 0x100000, 0xFE000000 → 0xFE000000, KERNBASE → 0

Question **2**

Complete

Mark 1.00 out of 1.00

What does seginit() do?

- a. Nothing significant, just repetition of earlier GDT setup but with free frames list created now
- b. Nothing significant, just repetition of earlier GDT setup but with kernel page table allocated now
- c. Adds two additional entries to GDT corresponding to Code and Data segments, but to be used in privilege level 3
- d. Nothing significant, just repetition of earlier GDT setup but with 2-level paging setup done
- e. Adds two additional entries to GDT corresponding to Code and Data segments, but to be used in privilege level 0

The correct answer is: Adds two additional entries to GDT corresponding to Code and Data segments, but to be used in privilege level 3

Question **3**

Complete

Mark 1.50 out of 1.50

Arrange the following in the correct order of execution (w.r.t. 'init')

'initcode' struct proc is created	2
userinit() is called	1
mpmain() calls scheduler()	4
initcode() calls exec("/init", ...)	8
'initcode' process is marked RUNNABLE	3
scheduler() schedules initcode() process	5
initcode() returns from trapret()	7
initcode() returns in forkret()	6

The correct answer is: 'initcode' struct proc is created → 2, userinit() is called → 1, mpmain() calls scheduler() → 4, initcode() calls exec("/init", ...) → 8, 'initcode' process is marked RUNNABLE → 3, scheduler() schedules initcode() process → 5, initcode() returns from trapret() → 7, initcode() returns in forkret() → 6

Question **4**

Complete

Mark 1.00 out of 1.00

Select the statement that most correctly describes what setupkvm() does

- a. creates a 1-level page table for the use by the kernel, as specified in kmap[] global array
- b. creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global arrray
- c. creates a 2-level page table for the use of the kernel, as specified in gdtdesc
- d. creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global arrray and makes kpgdir point to it

The correct answer is: creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global arrray

Question **5**

Complete

Mark 1.00 out of 1.00

Which of the following is done by mappages()?

- [x] a. create page table mappings to the range given by "pa" and "pa + size"
- [x] b. allocate page table if required
- [x] c. create page table mappings for the range given by "va" and "va + size"
- [ ] d. allocate page frame if required
- [ ] e. allocate page directory if required

The correct answers are: create page table mappings for the range given by "va" and "va + size", allocate page table if required, create page table mappings to the range given by "pa" and "pa + size"

Question **6**

Complete

Mark 1.00 out of 1.00

The approximate number of page frames created by kinit1 is

- ( ) a. 16
- ( ) b. 2000
- ( ) c. 1000
- ( ) d. 4
- ( ) e. 10
- ( ) f. 4000
- (•) g. 3000

The correct answer is: 3000

Question **7**

Complete

Mark 1.33 out of 2.00

exec() does this: curproc->tf->eip = elf.entry, but userinit() does this: p->tf->eip = 0; Select all the statements from below, that collectively explain this

- [ ] a. the code of 'initcode' is loaded at physical address 0
- [x] b. exec() loads from ELF file and the address of first instruction to be executed is given by 'entry'
- [x] c. In userinit() the function inituvm() has mapped the code of 'initcode' to be starting at virtual address 0
- [ ] d. the initcode is created using objcopy, which discards all relocation information and symbols (like entry)
- [ ] e. the 'entry' in initcode is anyways 0
- [ ] f. elf.entry is anyways 0, so both statements mean the same

The correct answers are: exec() loads from ELF file and the address of first instruction to be executed is given by 'entry', In userinit() the function inituvm() has mapped the code of 'initcode' to be starting at virtual address 0, the initcode is created using objcopy, which discards all relocation information and symbols (like entry)

Question **8**

Complete

Mark 0.08 out of 1.00

Select all the correct statements about initcode

- [x] a. The data and stack of initcode is mapped to one single page in userinit()
- [x] b. the size of 'initcode' is 2c
- [x] c. code of initcode is loaded in memory by the kernel during userinit()
- [ ] d. initcode is the 'init' process
- [x] e. code of initcode is loaded at virtual address 0
- [ ] f. code of 'initcode' is loaded along with the kernel during booting
- [x] g. initcode essentially calls exec("/init",...)

The correct answers are: code of 'initcode' is loaded along with the kernel during booting, the size of 'initcode' is 2c, The data and stack of initcode is mapped to one single page in userinit(), initcode essentially calls exec("/init",...)

Question **9**

Complete

Mark 1.50 out of 1.50

Which of the following is DONE by allocproc() ?

- [ ] a. setup kernel memory mappings for the process
- [x] b. allocate kernel stack for the process
- [ ] c. ensure that the process starts in trapret()
- [x] d. Select an UNUSED struct proc for use
- [x] e. setup the trapframe and context pointers appropriately
- [ ] f. setup the contents of the trapframe of the process properly
- [x] g. ensure that the process starts in forkret()
- [x] h. allocate PID to the process

The correct answers are: Select an UNUSED struct proc for use, allocate PID to the process, allocate kernel stack for the process, setup the trapframe and context pointers appropriately, ensure that the process starts in forkret()

Question **10**

Complete

Mark 1.00 out of 1.00

Does exec() code around clearptau() lead to wastage of one page frame?

- (•) a. yes
- ( ) b. no

The correct answer is: yes

Question **11**

Complete

Mark 1.00 out of 1.00

The variable 'end' used as argument to kinit1 has the value

- ( ) a. 81000000
- ( ) b. 80000000
- ( ) c. 80102da0
- ( ) d. 80110000
- ( ) e. 8010a48c
- (•) f. 801154a8

The correct answer is: 801154a8

Question **12**

Complete

Mark 1.00 out of 1.00

What does userinit() do ?

- a. initializes the users
- b. initializes the process 'init' and starts executing it
- c. sets up the 'init' process to start execution in forkret()
- d. sets up the 'initcode' process to start execution in forkret ()
- e. sets up the 'initcode' process to start execution in trapret()
- f. sets up the 'initcode' process to start execution in forkret()

The correct answer is: sets up the 'initcode' process to start execution in forkret()

Question **13**

Complete

Mark 1.00 out of 1.00

Why is there a call to kinit2? Why is it not merged with knit1?

- a. call to seginit() makes it possible to actually use PHYSTOP in argument to kinit2()
- b. Because there is a limit on the values that the argumets to knit1() can take.
- c. When kinit1() is called there is a need for few page frames, but later knit2() is called to serve need of more page frames
- d. knit2 refers to virtual addresses beyond 4MB, which are not mapped before kvalloc() is called

The correct answer is: knit2 refers to virtual addresses beyond 4MB, which are not mapped before kvalloc() is called

◄ Questions for test on kalloc/kfree/kvmalloc, etc.

Jump to...

(Optional Assignment) Slab allocator in xv6 ►