

Started on	Wednesday, 19 April 2023, 6:17 PM
State	Finished
Completed on	Wednesday, 19 April 2023, 8:27 PM
Time taken	2 hours 10 mins
Overdue	10 mins 8 secs
Grade	19.68 out of 30.00 (65.61%)

Question **1**

Correct

Mark 1.00 out of 1.00

Given that the memory access time is 100 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: ✓

The correct answer is: 4800020.00

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.

- ☒ a. log may be kept on same block device or another block device ✓
- ☒ b. file system recovery may end up losing data ✓
- ☒ c. even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery ✓
- ☒ d. a transaction is said to be committed when all operations are written to file system ✗
- ☐ 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 **3**

Correct

Mark 1.00 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)
        ;
    (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);
    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;
    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(semaphore *s) {
    spinlock(&(s->sl));
    (s->val)++;
    x = dequeue(s->sl) and enqueue(readyq, x);
    spinunlock(&(s->sl));
}

```

not holding lock after unblock



Your answer is correct.

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)
        ;
    (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);
    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;
    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(semaphore *s) {
    spinlock(&(s->sl));
    (s->val)++;
    x = dequeue(s->sl) and enqueue(readyq, x);
    spinunlock(&(s->sl));
}
```

→ not holding lock after unblock

Question **4**

Partially correct

Mark 0.67 out of 1.00

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

Select one or more:

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

Your answer is partially correct.

You have selected too many options.

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 **5**

Partially correct

Mark 1.50 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 ✓
- ☒ b. All synchronization primitives are implemented essentially with some hardware assistance. ✓
- ☐ c. Blocking means one process passing over control to another process
- ☒ d. Spinlocks are good for multiprocessor scenarios, for small critical sections ✓
- ☐ e. Thread that is going to block should not be holding any spinlock
- ☐ f. Blocking means moving the process to a wait queue and spinning
- ☒ g. Mutexes can be implemented using blocking and wakeup ✓
- ☒ h. Semaphores can be used for synchronization scenarios like ordered execution ✓
- ☐ i. Semaphores are always a good substitute for spinlocks
- ☐ j. Mutexes can be implemented using spinlock
- ☒ k. Spinlocks consume CPU time ✓
- ☐ l. Mutexes can be implemented without any hardware assistance

Your answer is partially correct.

You have correctly selected 6.

The correct answers are: Spinlocks are good for multiprocessor scenarios, for small critical sections, Spinlocks consume CPU time, Semaphores can be used for synchronization scenarios 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 **6**

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

Release the lock held by some another process



```
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
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 deadlocks



```

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

```

Tell compiler not to reorder memory access beyond this line



```

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**

```

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

```

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

```

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)
{
    pushcli();
    → Disable interrupts to avoid deadlocks

```

Question **7**

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: 1101001000010000



The correct answer is: 11011110

Question 8

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:

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

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.

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

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

The working set model is an attempt at approximating the locality of a process.: True

Thrashing can be limited if local replacement is used.: True

Thrashing can occur even if entire memory is not in use.: False

Thrashing is particular to demand paging systems, and does not apply to pure paging systems.: True

Thrashing occurs when the total size of all processes's locality exceeds total memory size.: True

Thrashing occurs because some process is doing lot of disk I/O.: False

Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.: False

mmap() solves the problem of thrashing.: False

Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.: True

During thrashing the CPU is under-utilised as most time is spent in I/O: True

Select T/F for statements about Volume Managers.

Do pay attention to the use of the words physical partition and physical volume.

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

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

A physical partition should be initialized as a physial volume, before it can be used by volume manager.: True

A logical volume may span across multiple physical partitions: True

The volume manager stores additional metadata on the physical disk partitions: True

The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.: True

A volume group consists of multiple physical volumes: True

Question **11**

Complete

Mark 0.50 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.
1. Calculate the nearest power of 2 to size. iterate over the free list, if found, remove the block from the free list and return it else, split a larger block and repeat
 2. `int sys_buddy_alloc(void);`
 3. In the function used by `kmalloc` to allocate memory from free list if `kalloc` returns `NULL`, return
 4. Name and a one-line description of new userland functionality to be added: `buddy_alloc()` - A userland function that returns a pointer to a block of memory allocated by the buddy allocator.
 5. Changes to Makefile: Add the source file containing the implementation of the buddy allocator to the list of source files to be compiled. Any other change in a maximum of 20 words per change. Modify page allocation to be aligned with powers of 2. Add a function to split a larger block into two smaller ones.

Comment:

Question **12**

Correct

Mark 1.00 out of 1.00

Map the technique with it's feature/problem

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

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

Question **13**

Complete

Mark 1.00 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.

1. int chown(char *path, int owner_id);

2. int sys_chown(void);

3. int sys_chown(void){

return sys_chown();

}

in syscall.h

#define SYS_chown 22

in user.h

int chown(char*, int);

in usys.S

SYSCALL(chown)

in sysproc.c

int sys_chown(void) {

char *path;

int owner_id;

if (argstr(0, &path) < 0 || argint(1, &owner_id) < 0) {

return -1;

}

return chown(path, owner_id);

}

d) no new ds added

e) chown() is a new system call that allows the owner of a file to be changed to a new user.

f) changes to makefile

Add sysfile.o and sysproc.o to the object files list.

g) no other changes req

Comment:

Question **14**

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:



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 **15**

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:

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

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 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. ✓
- ☒ b. 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()) ✓
- ☒ 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() ✓
- ☒ d. 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. ✓
- ☒ e. The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories ✓
- ☒ f. A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/" ✓
- ☒ g. The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems. ✓
- ☒ h. The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup. ✓

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 **17**

Partially correct

Mark 0.67 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
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;
```

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, void

acquire(struct spinlock *lk)

{

...

getcallerpcs(&lk, lk->pcs); → Traverse ebp chain to get sequence of instructions followed in functions calls, void

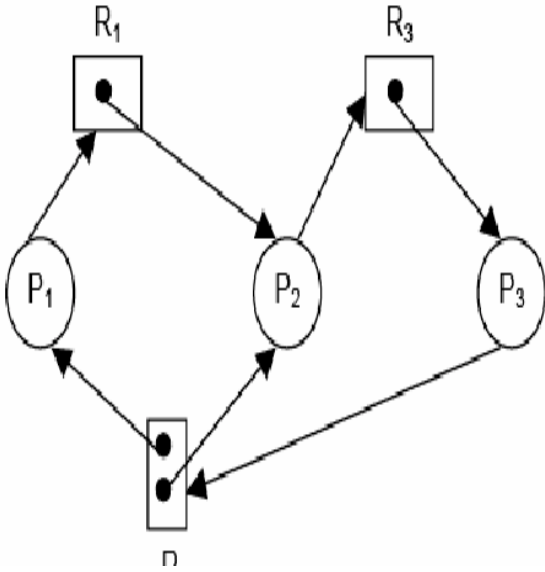
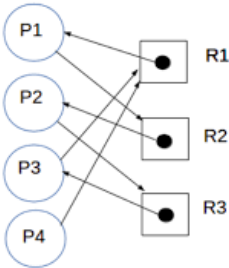
panic(char *s)

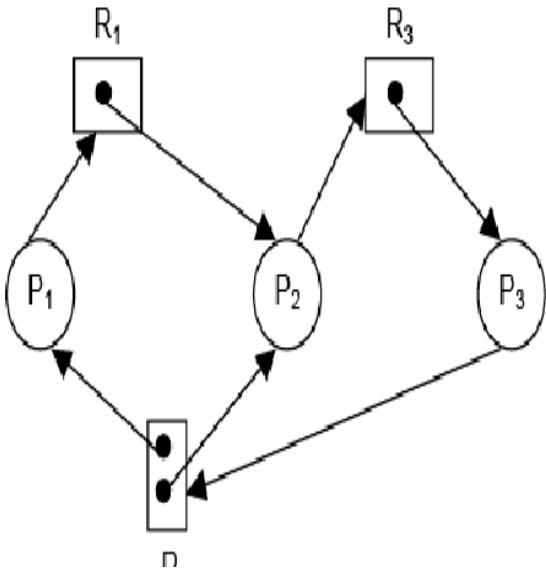
{

...

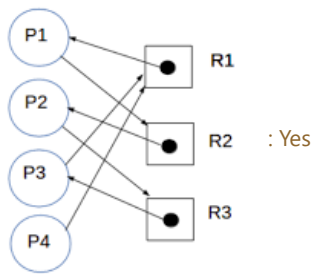
panicked = 1; → Ensure that no printing happens on other processors

For each of the resource allocation diagram shown, infer whether the graph contains at least one deadlock or not.

Yes	No
<div><div><input checked="" type="radio"/> <input type="radio"/></div><div><input type="radio"/> <input checked="" type="radio"/></div></div> <div></div>	<div><input checked="" type="radio"/></div>
<div><div><input checked="" type="radio"/> <input type="radio"/></div><div><input type="radio"/> <input checked="" type="radio"/></div></div> <div></div>	<div><input checked="" type="radio"/></div>



: Yes



Question **19**

Correct

Mark 2.00 out of 2.00

Compare paging with demand paging and select the correct statements.

Select one or more:

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

Your answer is correct.

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 20

Partially correct

Mark 0.50 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		
<input checked="" type="radio"/>	<input type="radio"/>	The arguments to system call originally reside on process stack.	✗
<input checked="" type="radio"/>	<input type="radio"/>	The functions like <code>argint()</code> , <code>argstr()</code> make the system call arguments available in the kernel.	✓
<input checked="" type="radio"/>	<input type="radio"/>	Integer arguments are copied from user memory to kernel memory using <code>argint()</code>	✓
<input checked="" type="radio"/>	<input type="radio"/>	String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer	✗
<input type="radio"/>	<input checked="" type="radio"/>	String arguments are first copied to trapframe and then from trapframe to kernel's other variables.	✓
<input type="radio"/>	<input checked="" type="radio"/>	Integer arguments are stored in <code>eax</code> , <code>ebx</code> , <code>ecx</code> , etc. registers	✗
<input type="radio"/>	<input checked="" type="radio"/>	The arguments to system call are copied to kernel stack in <code>trapasm.S</code>	✗
<input checked="" type="radio"/>	<input type="radio"/>	The arguments are accessed in the kernel code using <code>esp</code> on the trapframe.	✓

The arguments to system call originally reside on process stack.: True

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

String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer: True

String arguments are first copied to trapframe and then from trapframe to kernel's other variables.: False

Integer arguments are stored in `eax`, `ebx`, `ecx`, etc. registers: 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

Question **21**

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

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

Question **22**

Incorrect

Mark 0.00 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: ✗

The correct answer is: 134

Question **23**

Correct

Mark 1.00 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);

Execution order P2, P1, P3



P3:

Wait(S1);

Statement S3;

S = 0

P1:

Statement1;

Signal(S)

Execution order P1, then P2



P2:

Wait(S)

Statment2;

S = 5

Wait(S)

Critical Section

Signal(S)

Counting semaphore



S = 1

Wait(S)

Critical Section

Signal(S);

Binary Semaphore for mutual exclusion



Your answer is correct.

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

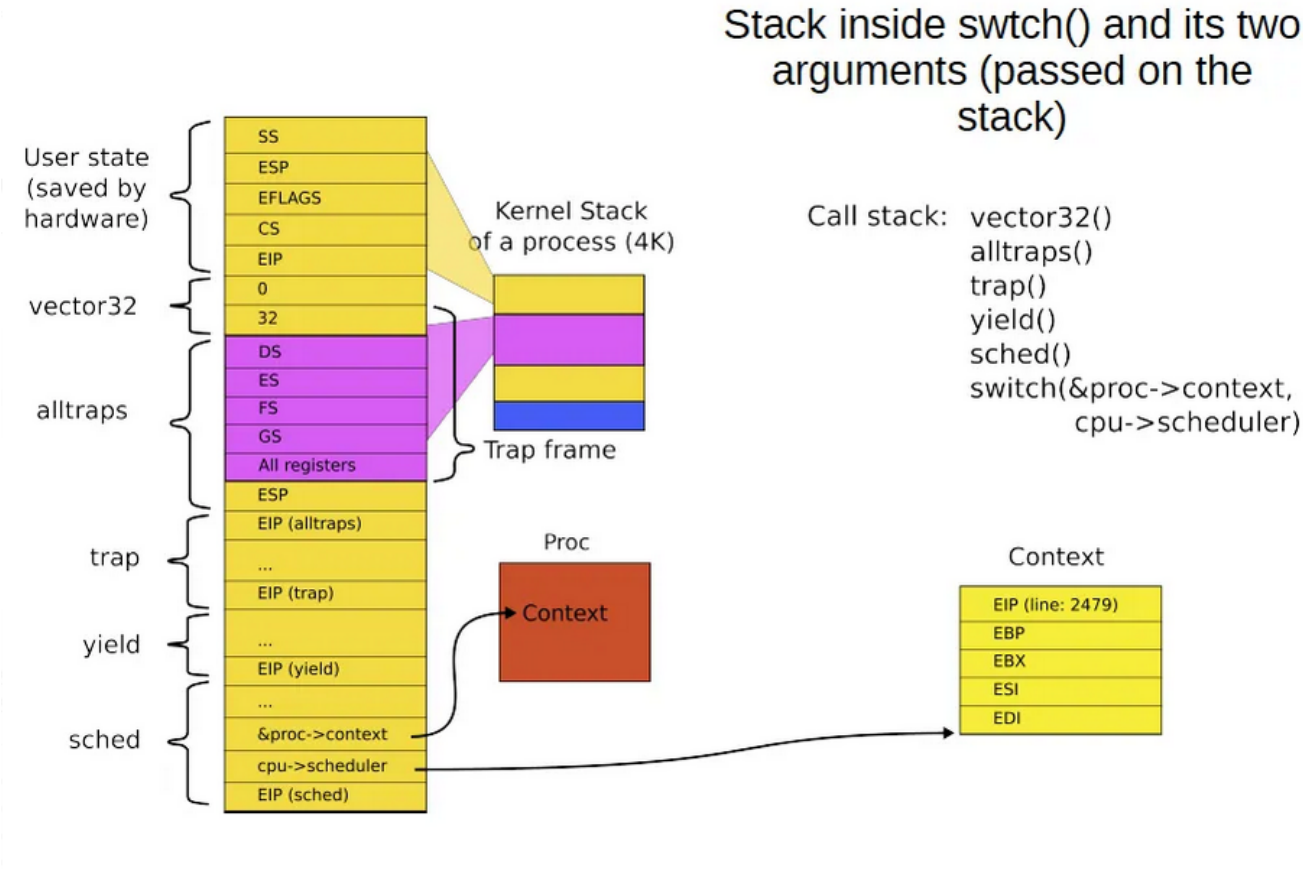
◀ Random Quiz - 6 (xv6 file system)

Jump to...

Homework questions: Basics of MM, xv6 booting ►

Started on	Friday, 17 March 2023, 2:29 PM
State	Finished
Completed on	Friday, 17 March 2023, 4:41 PM
Time taken	2 hours 12 mins
Grade	5.05 out of 10.00 (50.47%)

Mark statements as True/False, w.r.t. the given diagram



True	False		
<input type="radio"/>	<input checked="" type="radio"/>	This is a diagram of switch() called from scheduler()	✗
<input checked="" type="radio"/>	<input type="radio"/>	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.	✓
<input type="radio"/>	<input checked="" type="radio"/>	The "context" yellow coloured box, pointed to by cpu->scheduler is on the kernel stack of the scheduler.	✗
<input checked="" type="radio"/>	<input type="radio"/>	The diagram is correct	✗
<input type="radio"/>	<input checked="" type="radio"/>	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
<input checked="" type="radio"/>	<input type="radio"/>	The blue shaded part in "kernel stack of a process(4k)" refers to remaining part of stack (not used yet)	✗

This is a diagram of switch() called from scheduler(): False

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 "context" yellow coloured box, pointed to by cpu->scheduler is on the kernel stack of the scheduler.: False

The diagram is correct: True

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

Question **2**

Incorrect

Mark 0.00 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. readseg() reads first 4k bytes of kernel in memory
- ☒ 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. ❌
- ☒ c. The kernel.ld file contains instructions to the linker to link the kernel properly ✔️
- ☐ d. Although 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.
- ☒ e. The bootmain() code does not read the kernel completely 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 read using program headers in bootmain().
- ☒ g. The kernel.asm file is the final kernel file ❌
- ☒ h. Although 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. ✔️
- ☒ i. The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files ✔️

Your answer is incorrect.

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, Although 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 **3**

Incorrect

Mark 0.00 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 region of memory which is currently used as stack by processor
- ☐ b. The "stack" variable declared in "stack.S" in xv6
- ☐ c. The stack variable used in the program being discussed
- ☐ d. The region of memory where the kernel remembers all the function calls made
- ☐ e. The region of memory allocated by kernel for storing the parameters of functions
- ☒ f. The stack segment ❌
- ☐ g. The ss:esp pair

Your answer is incorrect.

The correct answer is: The ss:esp pair

Question **4**

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 **5**

Partially correct

Mark 0.13 out of 0.50

when is each of the following stacks allocated?

kernel stack of process

during fork() in allocproc()



kernel stack for the scheduler, on other processors

Choose...

kernel stack for scheduler, on first processor

Choose...

user stack of process

during exec()



Your answer is partially correct.

You have correctly selected 1.

The correct answer is: kernel stack of process → during fork() in allocproc(), kernel stack for the scheduler, on other processors → in main()->startothers(), kernel stack for scheduler, on first processor → in entry.S, user stack of process → during fork() in copyuvm()

Question **6**
 Partially correct
 Mark 0.38 out of 0.75

Mark statements as True/False w.r.t. ptable.lock

True	False		
<input type="radio"/>	<input checked="" type="radio"/>	ptable.lock is acquired but never released	how is that possible?
<input checked="" type="radio"/>	<input type="radio"/>	ptable.lock protects the proc[] array and all struct proc in the array	
<input checked="" type="radio"/>	<input type="radio"/>	It is taken by one process but released by another process, running on same processor	
<input type="radio"/>	<input type="radio"/>	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)	
<input type="radio"/>	<input checked="" type="radio"/>	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.
<input type="radio"/>	<input checked="" type="radio"/>	ptable.lock can be held by different processes on different processors at the same time	
<input type="radio"/>	<input checked="" type="radio"/>	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
<input checked="" type="radio"/>	<input type="radio"/>	A process can sleep on ptable.lock if it can't aquire it.	It's a spinlock!

ptable.lock is acquired but never released: False
 ptable.lock protects the proc[] array and all struct proc in the array: True
 It is taken by one process but released by another process, running on same processor: 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
 The swtch() in scheduler() is called without holding the ptable.lock when control jumps to it from sched(): False
 ptable.lock can be held by different processes on different processors at the same time: False
 the rule of "never block holding a spinlock" does not apply to ptable.lock in xv6: True
 A process can sleep on ptable.lock if it can't aquire it.: False

Question **7**
 Correct
 Mark 0.25 out of 0.25

Which of the following is not a task of the code of swtch() function

- ☒ a. Save the return value of the old context code
- ☒ b. Change the kernel stack location
- ☐ c. Switch stacks
- ☐ d. Jump to next context EIP
- ☐ e. Save the old context
- ☐ f. Load the new context

The correct answers are: Save the return value of the old context code, Change the kernel stack location

Question 8

Partially correct

Mark 0.11 out of 0.75

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

<code>inb \$0x64,%al</code>	real mode	✓
<code>movl \$(V2P_WO(entrypgdir)), %eax</code>	protected mode with segmentation and 4 MB pages	✗
<code>orl \$CR0_PE, %eax</code>	protected mode with segmentation and 4 MB pages	✗
<code>jmp *%eax</code>	real mode	✗
<code>ljmp \$(SEG_KCODE<<3), \$start32</code>	protected mode with only segmentation	✗
<code>readseg((uchar*)elf, 4096, 0);</code>	protected mode with segmentation and 4 MB pages	✗
<code>movw %ax, %gs</code>	real mode	✗

The correct answer is: `inb $0x64,%al` → real mode, `movl $(V2P_WO(entrypgdir)), %eax` → protected mode with only segmentation, `orl $CR0_PE, %eax` → real mode, `jmp *%eax` → protected mode with segmentation and 4 MB pages, `ljmp $(SEG_KCODE<<3), $start32` → real mode, `readseg((uchar*)elf, 4096, 0);` → protected mode with only segmentation, `movw %ax, %gs` → protected mode with only segmentation

Question 9

Partially correct

Mark 0.10 out of 0.25

Match function with it's meaning

<code>iderw</code>	tell disc controller to start I/O for the first buffer on idequeue	✗
<code>idestart</code>	Issue a disk read/write for a buffer, block the issuing process	✗
<code>ideintr</code>	disk interrupt handler, transfer data from controller to buffer for read-request, wake up processes waiting for this buffer	✗
<code>ideinit</code>	Initialize the disc controller	✓
<code>idewait</code>	Wait for disc controller to be ready	✓

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: `iderw` → Issue a disk read/write for a buffer, block the issuing process, `idestart` → tell disc controller to start I/O for the first buffer on idequeue, `ideintr` → disk interrupt handler, transfer data from controller to buffer, wake up processes waiting for this buffer, `iderw` → start I/O for next buffer, `ideinit` → Initialize the disc controller, `idewait` → Wait for disc controller to be ready

Question 10

Partially correct

Mark 0.50 out of 1.00

Mark the statements as True/False w.r.t. switch()

True	False		
<input checked="" type="radio"/>	<input type="radio"/>	switch stores the old context on new stack, and restores new context from old stack.	<input checked="" type="radio"/> old goes on old, new comes from new stack
<input type="radio"/>	<input checked="" type="radio"/>	sched() is the only place when p->context is set	<input checked="" type="radio"/>
<input type="radio"/>	<input checked="" type="radio"/>	p->context used in scheduler()->switch() was Generally set when the process was interrupted earlier, and came via sched()->switch()	<input checked="" type="radio"/> That's the only place when p->context is changed.
<input type="radio"/>	<input checked="" type="radio"/>	movl %esp, (%eax) means, *(c->scheduler) = contents of esp When switch() is called from scheduler()	<input checked="" type="radio"/>
<input type="radio"/>	<input checked="" type="radio"/>	switch() called from scheduler() changes the stack from the process's kernel stack to the scheduler's kernel stack.	<input checked="" type="radio"/> it does reverse!
<input checked="" type="radio"/>	<input type="radio"/>	switch() is written in assembly language, because it violates calling convention, by changing the stack itself.	<input checked="" type="radio"/>
<input type="radio"/>	<input checked="" type="radio"/>	switch() is written in assembly language because it violates the calling convention by pushing parameters on the stack on its own.	<input checked="" type="radio"/>
<input checked="" type="radio"/>	<input type="radio"/>	switch() is called only from sched() or scheduler()	<input checked="" type="radio"/>
<input checked="" type="radio"/>	<input type="radio"/>	push in switch() happens on old stack, while pop happens from new stack	<input checked="" type="radio"/>
<input checked="" type="radio"/>	<input type="radio"/>	switch() changes the context from "old" to "new"	<input checked="" type="radio"/> yeah, that's the definition

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

p->context used in scheduler()->switch() was **Generally** set when the process was interrupted earlier, and came via sched()->switch(): True

movl %esp, (%eax)

means, *(c->scheduler) = contents of esp

When switch() is called from scheduler(): False

switch() called from scheduler() changes the stack from the process's kernel stack to the scheduler's kernel stack.: False

switch() is written in assembly language, because it violates calling convention, by changing the stack itself.: True

switch() is written in assembly language because it violates the calling convention by pushing parameters on the stack on its own.: False

switch() is called only from sched() or scheduler(): True

push in switch() happens on old stack, while pop happens from new stack: True

switch() changes the context from "old" to "new": True

Question 11

Correct

Mark 0.25 out of 0.25

Why is there a call to kinit2? Why is it not merged with kinit1?

- ☐ 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 kinit2() is called to serve need of more page frames
- ☐ c. Because there is a limit on the values that the arguments to kinit1() can take.
- ☒ d. kinit2 refers to virtual addresses beyond 4MB, which are not mapped before kalloc() is called ✓

The correct answer is: kinit2 refers to virtual addresses beyond 4MB, which are not mapped before kalloc() is called

Question 12

Partially correct

Mark 0.19 out of 0.50

Mark the statements as True/False, with respect to the use of the variable "chan" in struct proc.

True	False		
<input checked="" type="radio"/>	<input type="radio"/>	When chan is not NULL, the 'state' in struct proc must be SLEEPING	✓
<input checked="" type="radio"/>	<input type="radio"/>	The value of 'chan' is changed only in sleep()	✗
<input checked="" type="radio"/>	<input type="radio"/>	in xv6, the address of an appropriate variable is used as a "condition" for a waiting process.	✗
<input checked="" type="radio"/>	<input type="radio"/>	chan stores the address of the variable, representing a condition, for which the process is waiting.	✓
<input type="radio"/>	<input checked="" type="radio"/>	chan is the head pointer to a linked list of processes, waiting for a particular event to occur	✗
<input type="radio"/>	<input checked="" type="radio"/>	when chan is NULL, the 'state' in proc must be RUNNABLE.	✗
<input type="radio"/>	<input checked="" type="radio"/>	Changing the state of a process automatically changes the value of 'chan'	✗
<input checked="" type="radio"/>	<input type="radio"/>	'chan' is used only by the sleep() and wakeup1() functions.	✓

When chan is not NULL, the 'state' in struct proc must be SLEEPING: True

The value of 'chan' is changed only in sleep(): True

in xv6, the address of an appropriate variable is used as a "condition" for a waiting process.: True

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

when chan is NULL, the 'state' in proc must be RUNNABLE.: False

Changing the state of a process automatically changes the value of 'chan': False

'chan' is used only by the sleep() and wakeup1() functions.: True

Question 13

Partially correct

Mark 0.33 out of 0.50

Mark statements as True/False w.r.t. the creation of free page list in xv6.

True	False		
<input checked="" type="radio"/>	<input type="radio"/>	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	✓
<input type="radio"/>	<input checked="" type="radio"/>	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.
<input checked="" type="radio"/>	<input type="radio"/>	kmem.use_lock is set to 1 after free page list is created, so that kmem.lock is taken before accessing kmem.freelist.	✓
<input checked="" type="radio"/>	<input type="radio"/>	the kmem.lock is used by kfree() and kalloc() only.	✓
<input type="radio"/>	<input checked="" type="radio"/>	free page list is a singly circular linked list.	✗ it's singly linked NULL terminated list.
<input checked="" type="radio"/>	<input type="radio"/>	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: True

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

the kmem.lock is used by kfree() and kalloc() only.: True

free page list is a singly circular linked list.: False

The pointers that link the pages together are in the first 4 bytes of the pages themselves: True

Question **14**

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		
<input checked="" type="radio"/>	<input type="radio"/>	The spinlock lk->lk is held when the process comes out of sleep()	✓
<input checked="" type="radio"/>	<input type="radio"/>	Sleeplock() will ensure that either the process gets the lock or the process gets blocked.	✓
<input type="radio"/>	<input checked="" type="radio"/>	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()	✗
<input checked="" type="radio"/>	<input type="radio"/>	the 'spinlock lk' is needed in a sleeplock, because access to the sleeplock for locking/unlocking itself creates a critical section	✓
<input checked="" type="radio"/>	<input type="radio"/>	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()

True	False		
<input checked="" type="radio"/>	<input type="radio"/>	Wakeup() will wakeup the first process waiting for the lock	✗ Wakeup() will wakeup all processes waiting for the lock
<input checked="" type="radio"/>	<input type="radio"/>	the 'spinlock lk' protects 'locked' variable, but not the 'name' nor the 'pid'	✗
<input checked="" type="radio"/>	<input type="radio"/>	<pre> 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); } </pre>	✗ loop is required because other process might have obtained the lock before this process returns from sleep().
<input type="radio"/>	<input checked="" type="radio"/>	sleep() is the function which blocks a process.	✓
<input checked="" type="radio"/>	<input type="radio"/>	A process has acquired the sleeplock when it comes out of sleep()	✗
<input type="radio"/>	<input checked="" type="radio"/>	All processes waiting for the sleeplock will have a race for acquiring lk->lk spinlock, because all are woken up	✓ wakeup() wakes up all processes, and they "thunder" to take the spinlock.

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

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

the 'spinlock lk' is needed in a sleeplock, because access to the sleeplock for locking/unlocking itself creates a critical section: True

The process which called acquiresleep() and then got blocked, is woken up by the timer interrupt: True

Wakeup() will wakeup the first process waiting for the lock: False

the 'spinlock lk' protects 'locked' variable, but not the 'name' nor the 'pid': 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

```

sleep() is the function which blocks a process.: True

A process has acquired the sleeplock when it comes out of sleep(): False

All processes waiting for the sleeplock will have a race for acquiring lk->lk spinlock, because all are woken up: True

Question **15**

Correct

Mark 0.50 out of 0.50

The struct buf has a sleeplock, and not a spinlock, because

- ☐ a. struct buf is used for disk I/O which takes a lot of time, so sleeping/blocking is the only option available.
- ☒ b. struct buf is used for disk I/O which takes a lot of time, so sleeping/blocking is preferred to spinning/busy-wait for the desired buf. ✓
- ☐ c. struct buf is used as a general purpose cache by kernel and cache operations take a lot of time, so better to use sleeplock rather than spinlock
- ☐ d. It could be a spinlock, but xv6 has chosen sleeplock for purpose of demonstrating how to use a sleeplock.
- ☐ e. sleeplock is preferable because it is used in interrupt context and spinlock can not be used in interrupt context

Your answer is correct.

The correct answer is: struct buf is used for disk I/O which takes a lot of time, so sleeping/blocking is preferred to spinning/busy-wait for the desired buf.

Question **16**

Correct

Mark 0.25 out of 0.25

The variable 'end' used as argument to kinit1 has the value

- ☐ a. 80110000
- ☐ b. 80000000
- ☐ c. 80102da0
- ☐ d. 81000000
- ☒ e. 801154a8 ✓
- ☐ f. 8010a48c

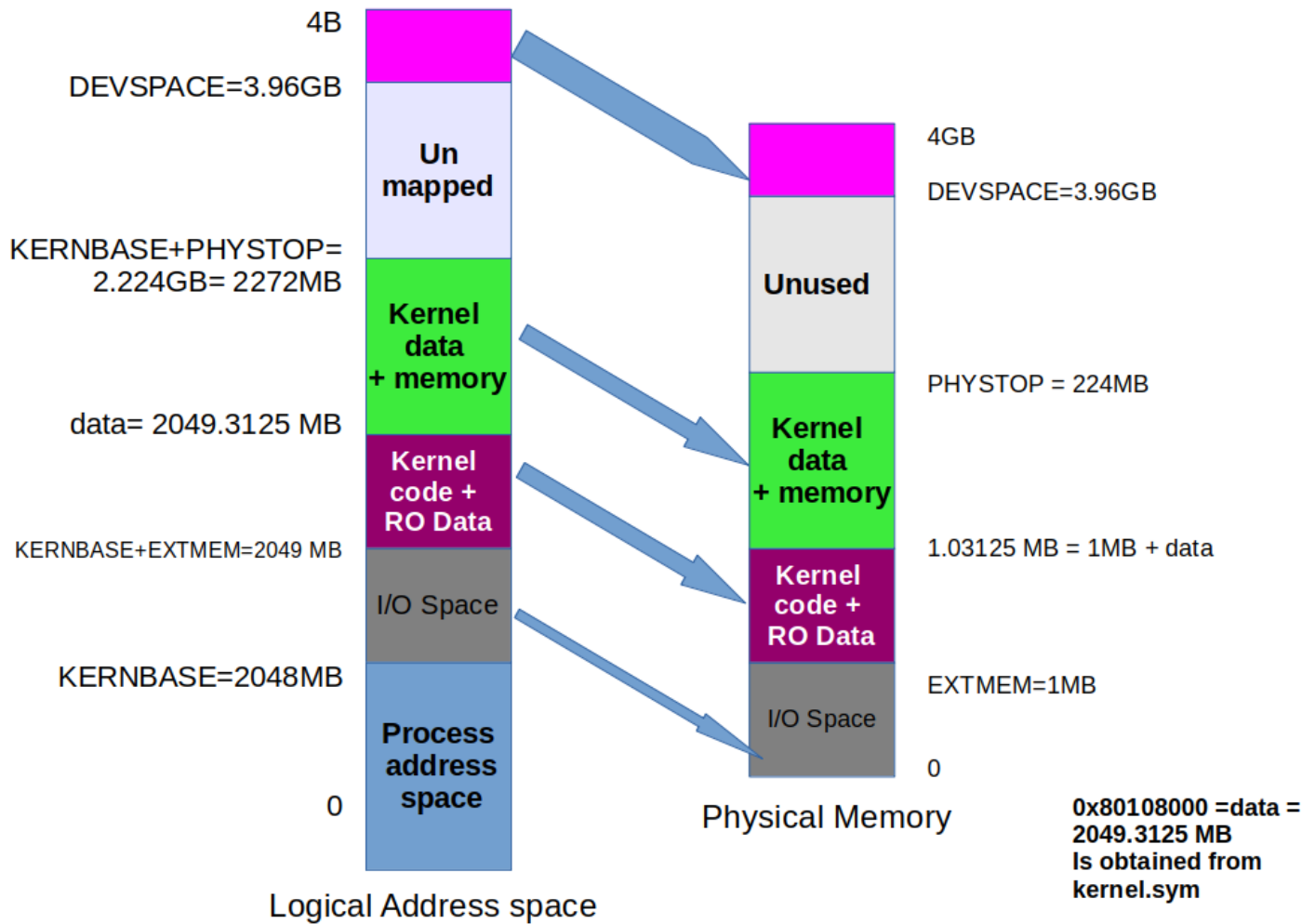
The correct answer is: 801154a8

Question 17

Partially correct

Mark 0.36 out of 0.50

With respect to this diagram, mark statements as True/False.



True	False		
<input checked="" type="radio"/>	<input type="radio"/>	This diagram only shows the absolutely defined virtual->physical mappings, not the mappings defined at run time by kernel.	✓
<input checked="" type="radio"/>	<input type="radio"/>	When bootloader loads the kernel, then physical memory from EXTMEM upto EXTMEM + data is occupied.	✓
<input type="radio"/>	<input checked="" type="radio"/>	The kernel virtual addresses start from KERNLINK = KERNBASE + EXTMEM	✗
<input checked="" type="radio"/>	<input type="radio"/>	The process's pages are mapped into physical memory from 1.03125 MB to PHYSTOP.	✓
<input checked="" type="radio"/>	<input type="radio"/>	PHYSTOP can be changed , but that needs kernel recompilation and re-execution.	✓
<input type="radio"/>	<input checked="" type="radio"/>	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	
<input checked="" type="radio"/>	<input type="radio"/>	"Kernel data + memory" on right side, here refers to the region from which pages are allocated to the kernel and process both.
	<input checked="" type="radio"/>	"Kernel data + memory" on LEFT side, here refers to the virtual addresses of kernel used at run time.

This diagram only shows the absolutely defined virtual->physical mappings, not the mappings defined at run time by kernel.: True

When bootloader loads the kernel, then physical memory from EXTMEM upto EXTMEM + data is occupied.: 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

PHYSTOP can be changed , but that needs kernel recompilation and re-execution.: 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

"Kernel data + memory" on right side, here refers to the region from which pages are allocated to the kernel and process both.: True

Question 18

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" clears the pipeline of the CPU so that it is as good as "fresh" CPU
- ☐ b. "cli" stands for clear screen and the screen should be cleared before OS boots.
- ☒ c. "cli" disables interrupts. It is required because as of now there are no interrupt handlers available ✖
- ☐ d. "cli" enables interrupts, it is required because the kernel must handle interrupts.
- ☐ e. "cli" enables interrupts, it is required because the kernel supports interrupts.
- ☐ f. "cli" clears all registers and makes them zero, so that processor is as good as "new"
- ☐ g. It disables interrupts. It is required because the interrupt handlers of kernel are not yet installed.
- ☐ h. "cli" that is Command Line Interface needs to be enabled first

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 19

Correct

Mark 0.25 out of 0.25

Select the odd one out

- ☐ a. Process stack of running process to kernel stack of running process
- ☒ b. Kernel stack of new process to kernel stack of scheduler ✔
- ☐ 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 Process stack of new process

The correct answer is: Kernel stack of new process to kernel stack of scheduler

Question **20**

Correct

Mark 0.50 out of 0.50

Which of the following is DONE by allocproc() ?

- ☐ a. ensure that the process starts in trapret()
- ☐ b. setup the contents of the trapframe of the process properly
- ☒ c. setup the trapframe and context pointers appropriately ✓
- ☒ d. allocate kernel stack for the process ✓
- ☒ e. ensure that the process starts in forkret() ✓
- ☒ f. allocate PID to the process ✓
- ☒ g. Select an UNUSED struct proc for use ✓
- ☐ h. setup kernel memory mappings for 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()

[◀ Quiz-1\(24 Feb 2023\)](#)

Jump to...

[Pre-requisite Quiz \(old\) - use it for practice ▶](#)

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

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

scheduling order (Matching)

- Order the events that occur on a timer interrupt:
 - Jump to a code pointed by IDT
 - > 2
 - Change to kernel stack of currently running process
 - > 1
 - Save the context of the currently running process
 - > 3
 - Jump to scheduler code
 - > 4
 - Select another process for execution
 - > 5
 - Set the context of the new process
 - > 6
 - Execute the code of the new process
 - > 7

timer interrupt - order events (Matching)

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

- cli
 - in bootasm.S
 - > Immaterial as the stack is not used here
 - > 0x7c00 to 0x10000
- ljmp \$(SEG_KCODE<<3), \$start32
 - in bootasm.S
 - > Immaterial as the stack is not used here
- call bootmain
 - in bootasm.S
 - > 0x7c00 to 0
- 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'
 - > 0x10000 to 0x7c00

xv6, esp values (Matching)

Select Yes if the mentioned element should be a part of PCB Select No otherwise.

part of PCB? (Multiple True False (ETH))

- 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
 - (50%) 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
 - (50%) The setting up of the most essential memory management infrastructure needs assembly code
 - (-50%) The code in assembly is required for transition to protected mode, from real mode; but calling convention was applicable all the time
 - (-50%) The code for reading ELF file can not be written in assembly

bootloader in assembly,C, why? (Multiple choice)

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

- (14.28571%) The kernel file gets loaded at the Physical address 0x10000 in memory.
- (14.28571%) The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it.
- (14.28571%) The elf->entry is set by the linker in the kernel file and it's 8010000c
- (14.28571%) The readseg finally invokes the disk I/O code using assembly instructions
- (14.28571%) The stosb() is used here, to fill in some space in memory with zeroes
- (14.28571%) The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded
- (14.28571%) The kernel file has only two program headers
- (-25%) The elf->entry is set by the linker in the kernel file and it's 0x80000000
- (-25%) The elf->entry is set by the linker in the kernel file and it's 0x80000000
- (-25%) The condition if(ph->memsz > ph->filesz) is never true.
- (-25%) The kernel file gets loaded at the Physical address 0x10000 +0x80000000 in memory.

correct stmt: bootmain() (Multiple choice)

In bootasm.S, on the line ljmp \$(SEG_KCODE<<3), \$start32The SEG_KCODE << 3, that is shifting of 1 by 3 bits is done because

- (0%) The value 8 is stored in code segment
- (100%) The code segment is 16 bit and only upper 13 bits are used for segment number
- (0%) The ljmp instruction does a divide by 8 on the first argument
- (0%) The code segment is 16 bit and only lower 13 bits are used for segment number
- (0%) While indexing the GDT using CS, the value in CS is always divided by 8

SEG_KCODE (Multiple choice / One answer only)

What's the trapframe in xv6?

- (100%) The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S
- (0%) The IDT table
- (0%) The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware only
- (0%) The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by code in trapasm.S only
- (0%) A frame of memory that contains all the trap handler code
- (0%) A frame of memory that contains all the trap handler code's function pointers
- (0%) A frame of memory that contains all the trap handler's addresses

What's the trapframe in xv6? (Multiple choice / One answer only)

Select all the correct statements about zombie processes

- (20%) A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it
- (20%) A process can become zombie if it finishes, but the parent has finished before it
- (20%) A zombie process occupies space in OS data structures
- (20%) If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent
- (20%) init() typically keeps calling wait() for zombie processes to get cleaned up
- (-33.33333%) A process becomes zombie when it's parent finishes
- (-33.33333%) A zombie process remains zombie forever, as there is no way to clean it up
- (-33.33333%) Zombie processes are harmless even if OS is up for long time

correct stmt- zombie (Multiple choice)

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 exec1cc 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.

- (100%) "ls" runs on current directory
- (0%) Execution fails as one exec can't invoke another exec
- (0%) Execution fails as the call to execl() in exec1 fails
- (0%) Execution fails as the call to execl() in exec2 fails
- (0%) Program prints hello

exec-then-exec (Multiple choice / One answer only)

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?

- (100%) OS code for read() will move PCB of current process to a wait queue and call scheduler
- (0%) OS code for read() will call scheduler
- (0%) OS code for read() will move the PCB of this process to a wait queue and return from the system call
- (0%) read() will return and process will be taken to a wait queue
- (0%) read() returns and process calls scheduler()

Process blocks - meaning (Multiple choice / One answer only)

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

bootmain scratch space (Multiple True False (ETH))

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.

options in boot, scheduler code (Essay)

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 have n and n+1th steps such that n+1th step can not follow n'th step.

- (-33.33333%) P1 running P1 makes system callsystem call returnsP1 runningtimer interruptScheduler runningP2 running
- (-33.33333%) P1 runningkeyboard hardware interruptkeyboard interrupt handler runninginterrupt handler returnsP1 runningP1 makes sytem call system call returnsP1 runningtimer interruptschedulerP2 running
- (-33.33333%) P1 runningP1 makes sytem call and blocksSchedulerP2 runningP2 makes sytem call and blocksSchedulerP3 runningHardware interruptInterrupt unblocks P1Interrupt returnsP3 runningTimer interruptSchedulerP1 running
- (33.33333%) P1 runningP1 makes sytem call and blocksSchedulerP2 runningP2 makes sytem call and blocksSchedulerP1 running again
- (33.33333%) P1 runningP1 makes system calltimer interruptSchedulerP2 runningtimer interruptScheulerP1 runningP1's system call return
- (33.33333%) P1 runningP1 makes sytem callSchedulerP2 runningP2 makes sytem call and blocksSchedulerP1 running again

Sequence not possible (Multiple choice)

Select the correct statements about interrupt handling in xv6 code

- (12.5%) All the 256 entries in the IDT are filled
- (12.5%) Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt
- (12.5%) xv6 uses the 64th entry in IDT for system calls
- (12.5%) On any interrupt/syscall/exception the control first jumps in vectors.S
- (12.5%) Before going to alltraps, the kernel stack contains upto 5 entries.
- (12.5%) The trapframe pointer in struct proc, points to a location on kernel stack
- (12.5%) The function trap() is the called irrespective of hardware interrupt/system-call/exception
- (12.5%) The CS and EIP are changed only after pushing user code's SS,ESP on stack
- (-20%) xv6 uses the 0x64th entry in IDT for system calls
- (-20%) On any interrupt/syscall/exception the control first jumps in trapasm.S
- (-20%) The trapframe pointer in struct proc, points to a location on user stack
- (-20%) The function trap() is the called only in case of hardware interrupt
- (-20%) The CS and EIP are changed only immediately on a hardware interrupt

correct stmt: xv6 interrupt handler (Multiple choice)

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

Scheduler code: T/F (Multiple True False (ETH))

Started on	Friday, 24 February 2023, 2:45 PM
State	Finished
Completed on	Friday, 24 February 2023, 4:57 PM
Time taken	2 hours 12 mins
Grade	8.34 out of 10.00 (83.4%)

Question **1**

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"

Data blocks of the file	<input type="text" value="Yes"/>	✗
One or more data bitmap blocks for the parent directory	<input type="text" value="Yes"/>	✗
One or multiple data blocks of the parent directory	<input type="text" value="No"/>	✓
Block bitmap(s) for all the blocks of the file	<input type="text" value="No"/>	✗
Possibly one block bitmap corresponding to the parent directory	<input type="text" value="Yes"/>	✓
Superblock	<input type="text" value="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: Data blocks of the file → No, One or more data bitmap blocks for the parent directory → No, One or multiple data blocks of 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, Superblock → Yes

Question **2**

Correct

Mark 0.50 out of 0.50

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

- ☒ a. typedefs ✓
- ☒ b. #directives ✓
- ☐ c. expressions
- ☐ d. pointer dereference
- ☐ e. local variable declaration
- ☒ f. global variables ✓
- ☐ g. function calls

Your answer is correct.

The correct answers are: #directives, typedefs, global variables

Question **3**

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	✓
Paging	one continuous chunk	✓
Relocation + Limit	one continuous chunk	✓
Segmentation, then paging	many continuous chunks of variable size	✓

Your answer is correct.

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

Question **4**

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 disallows hardware interrupts when a process is running
- ☐ c. It prohibits invocation of kernel code completely, if a user program 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 **5**

Correct

Mark 0.50 out of 0.50

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

True	False		
<input checked="" type="radio"/>	<input type="radio"/>	It's possible that a particular hardware has multiple device drivers available for it.	✓
<input type="radio"/>	<input checked="" type="radio"/>	Device driver is part of hardware	✓
<input checked="" type="radio"/>	<input type="radio"/>	Device driver is an intermediary between the hardware controller and OS	✓
<input checked="" type="radio"/>	<input type="radio"/>	Writing a device driver mandatorily demands reading the technical documentation about the hardware.	✓
<input type="radio"/>	<input checked="" type="radio"/>	Different devices of the same type (e.g. 2 IDE hard disks) must need different device drivers.	✓
<input checked="" type="radio"/>	<input type="radio"/>	Device driver is part of OS code	✓

It's possible that a particular hardware has multiple device drivers available for it.: True

Device driver is part of hardware: False

Device driver is an intermediary between the hardware controller and OS: True

Writing a device driver mandatorily demands reading the technical documentation about the hardware.: True

Different devices of the same type (e.g. 2 IDE hard disks) must need different device drivers.: False

Device driver is part of OS code: True

Question **6**

Partially correct

Mark 0.40 out of 0.50

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

Select No otherwise.

Yes	No		
<input checked="" type="radio"/>	<input type="radio"/>	Process context	✓
<input type="radio"/>	<input checked="" type="radio"/>	PID of Init	✓
<input checked="" type="radio"/>	<input type="radio"/>	Pointer to the parent process	✓
<input checked="" type="radio"/>	<input type="radio"/>	List of opened files	✓
<input checked="" type="radio"/>	<input type="radio"/>	PID	✓
<input checked="" type="radio"/>	<input type="radio"/>	Memory management information about that process	✓
<input type="radio"/>	<input checked="" type="radio"/>	Pointer to IDT	✗
<input checked="" type="radio"/>	<input type="radio"/>	EIP at the time of context switch	✓
<input type="radio"/>	<input checked="" type="radio"/>	Function pointers to all system calls	✗
<input checked="" type="radio"/>	<input type="radio"/>	Process state	✓

Process context: Yes

PID of Init: No

Pointer to the parent process: Yes

List of opened files: Yes

PID: Yes

Memory management information about that process: Yes

Pointer to IDT: No

EIP at the time of context switch: Yes

Function pointers to all system calls: No

Process state: Yes

Question **7**

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:



The correct answer is: hi one one

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!

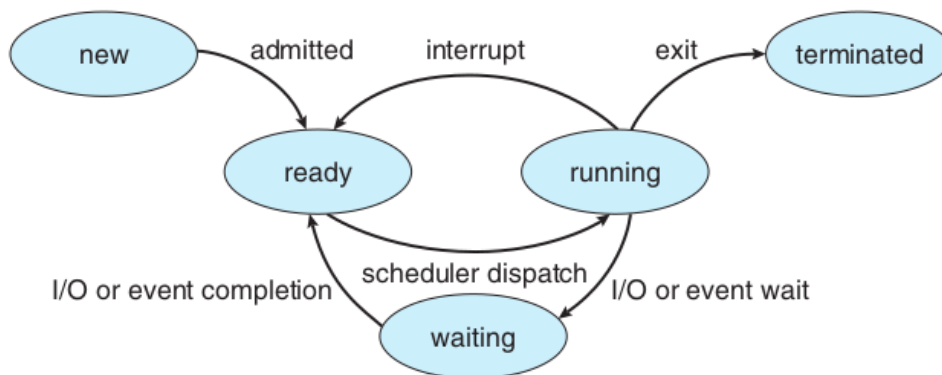


Figure 3.2 Diagram of process state.

True	False		
<input checked="" type="radio"/>	<input type="radio"/>	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	✓
<input type="radio"/>	<input checked="" type="radio"/>	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.	✓
<input checked="" type="radio"/>	<input type="radio"/>	Only a process in READY state is considered by scheduler	✓
<input type="radio"/>	<input checked="" type="radio"/>	A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first	✓
<input checked="" type="radio"/>	<input type="radio"/>	Every forked process has to go through ZOMBIE state, at least for a small duration.	✓

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

Only a process in READY state is considered by scheduler: True

A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first: False

Every forked process has to go through ZOMBIE state, at least for a small duration.: True

Question **9**

Correct

Mark 0.50 out of 0.50

Map each signal with it's meaning

SIGUSR1	User Defined Signal	✓
SIGPIPE	Broken Pipe	✓
SIGSEGV	Invalid Memory Reference	✓
SIGALRM	Timer Signal from alarm()	✓
SIGCHLD	Child Stopped or Terminated	✓

The correct answer is: SIGUSR1 → User Defined Signal, SIGPIPE → Broken Pipe, SIGSEGV → Invalid Memory Reference, SIGALRM → Timer Signal from alarm(), SIGCHLD → Child Stopped or Terminated

Question **10**

Correct

Mark 0.50 out of 0.50

Match the elements of C program to their place in memory

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

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

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. Program 1 ensures 2>&1 and does not ensure > /tmp/ddd
- ☐ b. Program 2 makes sure that there is one file offset used for '2' and '1'
- ☐ c. Program 2 does 1>&2
- ☐ d. Both programs are correct
- ☒ e. Only Program 1 is correct ✓
- ☐ f. Program 1 does 1>&2
- ☐ g. Both program 1 and 2 are incorrect
- ☐ h. Program 2 ensures 2>&1 and does not ensure > /tmp/ddd
- ☐ i. Program 1 is correct for > /tmp/ddd but not for 2>&1
- ☒ j. Program 1 makes sure that there is one file offset used for '2' and '1' ✓
- ☐ k. Program 2 is correct for > /tmp/ddd but not for 2>&1
- ☐ 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 **12**

Correct

Mark 0.50 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		
<input checked="" type="radio"/>	<input type="radio"/>	The process has definitely performed illegal memory access.	✓
<input checked="" type="radio"/>	<input type="radio"/>	The core file can be analysed later using a debugger, to determine what went wrong.	✓ use gdb ./core ./executable-filename
<input checked="" type="radio"/>	<input type="radio"/>	On Linux, the process was sent a SIGSEGV signal and the default handler for the signal is "Term", so the process is terminated.	✓
<input type="radio"/>	<input checked="" type="radio"/>	The illegal memory access was detected by the kernel and the process was punished by kernel.	✓ "detection" is done by CPU, not kernel.
<input type="radio"/>	<input checked="" type="radio"/>	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.
<input checked="" type="radio"/>	<input type="radio"/>	The image of the process is stored in a file called "core", if the ulimit allows so.	✓ see ulimit -a
<input type="radio"/>	<input checked="" type="radio"/>	The term "core" refers to the core code of the kernel.	✓ core means memory, all memory for the process.

The process has definitely performed illegal memory access.: True

The core file can be analysed later using a debugger, to determine what went wrong.: 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

On Linux, the message is printed only because the memory management scheme is segmentation: False

The image of the process is stored in a file called "core", if the ulimit allows so.: True

The term "core" refers to the core code of the kernel.: False

Question **13**

Incorrect

Mark 0.00 out of 0.50

Doing a lookup on the pathname /a/b/b/c/d for opening the file "d" requires reading 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 **14**

Partially correct

Mark 0.42 out of 0.50

Which of the following instructions should be privileged?

Select one or more:

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

Your answer is partially correct.

You have correctly selected 5.

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**

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	✓
Linked allocation	Too many seeks	✓
Continuous allocation	need for compaction	✓

Your answer is correct.

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

Question **16**

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		
<input checked="" type="radio"/>	<input type="radio"/>	When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.	✓
<input type="radio"/>	<input checked="" type="radio"/>	When a process is executing, each virtual address is converted into physical address by the kernel directly.	✓
<input checked="" type="radio"/>	<input type="radio"/>	The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.	✗
<input checked="" type="radio"/>	<input type="radio"/>	The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.	✓
<input type="radio"/>	<input checked="" type="radio"/>	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.	✗
<input type="radio"/>	<input checked="" type="radio"/>	The kernel refers to the page table for converting each virtual address to physical address.	✗
<input type="radio"/>	<input checked="" type="radio"/>	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.: 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 ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.: 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 kernel refers to the page table for converting each virtual address to physical address.: 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

Question **17**

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. writing zeroes on all sectors
- ☒ c. creating layout of empty directory tree/graph data structure ✓
- ☐ d. storing all the necessary programs on the disk/partition

The correct answer is: creating layout of empty directory tree/graph data structure

Question **18**

Partially correct

Mark 0.45 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];

```
    pipe(
```

✓);

```
    pid1 =
```

✓ ;

```
    if(pid1 != 0) {
```

```
        close(pfd[0]
```

✓);

```
        close(
```

✓);

```
        dup(
```

✓);

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

✓ ", NULL);

```
    }
```

```
    pipe(
```

✓);

✓ = fork();

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

```
        close(
```

✗ ;

```
        close(0);
```

```
        dup(
```

```
✓ );
    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
[1][0]
✗ );
    execl("/usr/bin/tail", "/usr/bin/tail", "
-1
✓ ", NULL);
}
}
```

Question **19**

Partially correct

Mark 0.35 out of 0.50

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

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

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 exist beyond the life-time of processes using them.: True

Both types of pipes provide FIFO communication.: True

The pipe() system call can be used to create either a named or un-named pipe.: False

Un-named pipes are inherited by a child process from parent.: True

Un-named pipes can be used for communication between only "related" processes, if the common ancestor created it.: True

Named pipes can be used for communication between only "related" processes.: False

Named pipe exists as a file: 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

Question **20**

Partially correct

Mark 0.19 out of 0.50

How does the compiler calculate addresses for the different parts of a C program, when paging is used?

Global variables	starting with 0	✗
typedef	Immediately after the text, along with globals	✗
#include files	No memory allocated, they are handled by pre-processor	✗
malloced memory	Heap (handled by the malloc-free library, using OS's system calls)	✓
Local variables	An offset with respect to stack pointer (esp)	✓
Text	starting with 0	✓
#define	No memory allocated, as they are not variables, but only conceptual definition of a type	✗
Static variables	Immediately after the text	✗

Your answer is partially correct.

You have correctly selected 3.

The correct answer is: Global variables → Immediately after the text, typedef → No memory allocated, as they are not variables, but only conceptual definition of a type, #include files → No memory allocated for the file, but if it contains variables, then variables may be allocated memory, malloced memory → Heap (handled by the malloc-free library, using OS's system calls), Local variables → An offset with respect to stack pointer (esp), Text → starting with 0, #define → No memory allocated, they are handled by pre-processor, Static variables → Immediately after the text, along with globals

[◀ Quiz-1 Preparation questions](#)

Jump to...

[Quiz - 2 \(17 March 2023\) ▶](#)

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

- a. (25%) MMU is inside the processor
- b. (25%) Logical to physical address translations in MMU are done in hardware, automatically
- c. (25%) The Operating system sets up relevant CPU registers to enable proper MMU translations
- d. (25%) Illegal memory access is detected in hardware by MMU and a trap is raised
- e. (-25%) MMU is a separate chip outside the processor
- f. (-25%) Logical to physical address translations in MMU are done with specific machine instructions
- g. (-25%) The operating system interacts with MMU for every single address translation
- h. (-25%) Illegal memory access is detected by operating system

correct stmt: MMU (Multiple choice)

• **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.**

- a. (100%) Do not exhibit Balady's anomaly
- b. (0%) Exhibit Balady's anomaly between 2 and 3 frames
- c. (0%) Exhibit Balady's anomaly between 3 and 4 frames

fifo balady's anomaly (Multiple choice / One answer only)

• **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.**

- a. (50%) This example does not exhibit Balady's anomaly
- b. (0%) Exhibit Balady's anomaly between 2 and 3 frames
- c. (0%) Exhibit Balady's anomaly between 3 and 4 frames
- d. (100%) LRU will never exhibit Balady's anomaly

LRU balady's anomaly (Multiple choice / One answer only)

• **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:10011010Now, 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

buddy allocated-1 (Short answer)

• **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:**

- 10
- 9

FIFO page faults (Numerical)

• **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 executingTimer interruptContext changes to kernel contextGeneric interrupt handler runsGeneric interrupt handler calls SchedulerScheduler selects P2 for executionAfter scheduler, Process P2 user code executing This sequence of events is:{#1} Because {#2}**

- {1:MULTICHOICE:%100%not-possible#~%0%possible#}
- {1:MULTICHOICE:%100%Process P2 has to return from interrupt context before it's user code executes#~%0%Timer interrupt is not possible as kernel is non-interruptible#~%0%The scheduler will not run in this scenario#~%0%Generic interrupt handler can not call scheduler#~%0%Process P1's user code will prohibit any interrupts#~%0%Interrupts are not possible when kernel code is running#~%0%On uniprocessor systems timer is not needed#}

impossible-sequence-of-events-1 (Embedded answers (Cloze))

• **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. (25%) Process state is stored in the PCB
- b. (25%) Process state can be implemented as just a number
- c. (25%) The scheduler can change state of a process from RUNNALBE to RUNNING
- d. (25%) A process becomes ZOMBIE when it calls exit()
- e. (-20%) Process state is changed only by interrupt handlers
- f. (-20%) Process state is stored in the processor
- g. (-20%) Process state is implemented as a string
- h. (-20%) The scheduler can change state of a process from RUNNALBE to RUNNING and vice-versa
- i. (-20%) A process becomes ZOMBIE when another process bites into it's memory

process state correct stmt (Multiple choice)

• **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!**

process-state-change - 1 (Multiple True False (ETH))

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

- a. (20%) many-one model can be implemented even if there are no kernel threads
- b. (20%) all three models, that is many-one, one-one, many-many , require a user level thread library
- c. (20%) one-one model increases kernel's scheduling load
- d. (20%) many-one model gives no speedup on multicore processors
- e. (20%) A process blocks in many-one model even if a single thread makes a blocking system call
- f. (-50%) one-one model can be implemented even if there are no kernel threads
- g. (-50%) A process may not block in many-one model, if a thread makes a blocking system call

correct stmt-thread models (Multiple choice)

• **If one thread opens a file with read privileges then**

- a. (0%) other threads in the another process can also read from that file
- b. (100%) other threads in the same process can also read from that file
- c. (0%) any other thread cannot read from that file
- d. (0%) none of these

Thread (Multiple choice / One answer only)

• **Select all the correct statements about signals**

- a. (25%) Signals are delivered to a process by kernel
- b. (25%) A signal handler can be invoked asynchronously or synchronously depending on signal type
- c. (25%) The signal handler code runs in user mode of CPU
- d. (25%) SIGKILL definitely kills a process because it can't be caught or ignored, and it's default action terminates the process
- e. (-25%) Signals are delivered to a process by another process
- f. (-25%) The signal handler code runs in kernel mode of CPU
- g. (-25%) SIGKILL definitely kills a process because it's code runs in kernel mode of CPU
- h. (-25%) Signal handlers once replaced can't be restored

correct stmt - signal handling (Multiple choice)

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

- 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[]`
- Setup kernel part of a page table, and switch to that page table
-> `kvmalloc()`
 - -> `kinit1()`
 - -> `kinit2()`
 - -> `kfree()`

functionality/use with function/variable in xv6 code. (Matching)

• **After virtual memory is implemented (select T/F for each of the following)One Program's size can be larger than physical memory size**

After VM implementation (Multiple True False (ETH))

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

- `bootasm.S`
-> `gdt setup with 3 entries, at start32 symbol of bootasm.S`
- `bootmain()`
-> `gdt setup with 3 entries, at start32 symbol of bootasm.S`
- `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`
- after `seginitt()` 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`
- -> `gdt setup with 3 entries, right from first line of code of bootloader`

code,segmentation status,xv6 (Matching)

• **W.r.t. Memory management in xv6, xv6 uses physical memory upto 224 MB onlyMark statements True or False**

xv6 memory management T/F (Multiple True False (ETH))

• **Select the correct statements about interrupt handling in xv6 code**

- (12.5%) All the 256 entries in the IDT are filled
- (12.5%) Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt
- (12.5%) xv6 uses the 64th entry in IDT for system calls
- (12.5%) On any interrupt/syscall/exception the control first jumps in `vectors.S`
- (12.5%) Before going to `alltraps`, the kernel stack contains upto 5 entries.
- (12.5%) The `trapframe` pointer in `struct proc`, points to a location on kernel stack
- (12.5%) The function `trap()` is the called irrespective of hardware interrupt/system-call/exception
- (12.5%) The CS and EIP are changed only after pushing user code's SS,ESP on stack
- (-20%) xv6 uses the 0x64th entry in IDT for system calls
- (-20%) On any interrupt/syscall/exception the control first jumps in `trapasm.S`
- (-20%) The `trapframe` pointer in `struct proc`, points to a location on user stack
- (-20%) The function `trap()` is the called only in case of hardware interrupt
- (-20%) The CS and EIP are changed only immediately on a hardware interrupt

correct stmt: xv6 interrupt handler (Multiple choice)

- **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. **(100%)** end, P2V(PHYSTOP)
 - b. **(0%)** end, PHYSTOP
 - c. **(0%)** P2V(end), P2V(PHYSTOP)
 - d. **(0%)** P2V(end), PHYSTOP
 - e. **(0%)** end, 4MB
 - f. **(0%)** end, (4MB + PHYSTOP)
 - g. **(0%)** end, P2V(4MB + PHYSTOP)

free pages VA range (Multiple choice / One answer only)

- **The data structure used in kalloc() and kfree() in xv6 is**
 - a. **(100%)** Singly linked NULL terminated list
 - b. **(0%)** Singly linked circular list
 - c. **(0%)** Double linked NULL terminated list
 - d. **(0%)** Doubly linked circular list

kalloc, kfree data structure (Multiple choice / One answer only)

- **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)**

Global/local arguments (Multiple True False (ETH))

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

Named vs unnamed pipe (Multiple True False (ETH))

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

T/F about mmap() (Multiple True False (ETH))

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

T/F argument passing xv6 (Multiple True False (ETH))

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

T/F multiple about Thrashing (Multiple True False (ETH))

- **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
- UNUSED-> **wait()**, called by parent process
- SLEEPING-> **sleep()**, called by any process blocking itself
- RUNNABLE-> **wakeup()**, called by an interrupt handler
- RUNNING-> **scheduler()**
- ZOMBIE-> **exit()**, called by process itself
- -> **exit()**, called by an interrupt handler
- -> **wait()** called by the exiting process itself

xv6 process state changes (Matching)

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

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

xv6 VM functions (Matching)

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

- a. (16.66667%) Minor page fault may occur because the page was a shared memory page
- b. (16.66667%) Minor page fault may occur because of a page fault during fork(), on code of an already running process
- c. (16.66667%) Minor page fault may occur because the page was freed, but still tagged and available in the free page list
- d. (16.66667%) Major page faults are likely to occur in more numbers at the beginning of the process
- e. (16.66667%) Thrashing is possible only due to major page faults
- f. (16.66667%) Minor page faults are an improvement of the page buffering techniques

Major & Minor page faults (Multiple choice)

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

- a. (25%) POSIX shared memory is newer than System V shared memory
- b. (25%) POSIX shared memory is "thread safe", System V is not
- c. (25%) POSIX allows giving name to shared memory, System V does not
- d. (25%) System V is more prevalent than POSIX even today

POSIX & SYSV SHM (Multiple choice)

• **Select the most common causes of use of IPC by processes**

- a. (33.33333%) Sharing of information of common interest
- b. (33.33333%) Breaking up a large task into small tasks and speeding up computation, on multiple core machines
- c. (33.33333%) More modular code
- d. (-50%) Get the kernel performance statistics
- e. (-50%) More security checks

Use of IPC (Multiple choice)

• **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

Second chance (Short answer)

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

- userinit() is called
- empty struct proc is obtained for initcode
- kernel stack is allocated for initcode process
- trapframe and context pointers are set to proper location
- code is set to start in forkret() when process gets scheduled
- kernel memory mappings are created for initcode
- values are set in the trapframe of initcode
- initcode process is set to be runnable
- initcode is selected by scheduler for execution
- initcode process runs
- initcode calls exec system call
- trap() runs
- function pointer from syscalls[] array is invoked
- sys_exec runs
- the header of "/init" ELF file is ready by kernel
- memory mappings are created for "/init" process
- Stack is allocated for "/init" process
- Arguments on setup on process stack for /init
- name of process "/init" is copied in struct proc
- page table mappings of 'initcode' are replaced by mappings of 'init'

order: creation of init process (Ordering)

• **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!)**

T/F basics of paging (Multiple True False (ETH))

• 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.

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

first-best-worst fit(2) (Matching)

• Select all the correct statements about linking and loading.

- a. (20%) Continuous memory management schemes can support static linking and static loading. (may be inefficiently)
- b. (20%) Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently)
- c. (20%) Dynamic linking essentially results in relocatable code.
- d. (20%) Loader is part of the operating system
- e. (20%) Dynamic linking and loading is not possible without demand paging or demand segmentation.
- f. (-25%) Loader is last stage of the linker program
- g. (-25%) Static linking leads to non-relocatable code
- h. (-25%) Continuous memory management schemes can support dynamic linking and dynamic loading.
- i. (-25%) Dynamic linking is possible with continous memory management, but variable sized partitions only.

correct stmt: linking loading (Multiple choice)

• 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: {#1} An inverted page table: {#2}

- {1:SHORTANSWER:%100%1048576#}
- {1:SHORTANSWER:%100%131072#}

number of page table entries (Embedded answers (Cloze))

• 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 :{#1} b. Install a bigger paging disk. :{#2}c. Increase the degree of multiprogramming. :{#3}d. Decrease the degree of multiprogramming. :{#4}e. Install more main memory.:{#5}f. Install a faster hard disk or multiple controllers with multiple hard disks. :{#6}g. Add prepaging to the page-fetch algorithms. :{#7} h. Increase the page size. :{#8}

- {1:MULTICHOICE:%100%No#~%0%Yes#}
- {1:MULTICHOICE:%100%No#~%0%Yes#}
- {1:MULTICHOICE:%100%No#~%0%Yes#}
- {1:MULTICHOICE:%100%Yes#~%0%No#}
- {1:MULTICHOICE:%100%Yes#~%0%No#}
- {1:MULTICHOICE:%100%Yes#~%0%No#}
- {1:MULTICHOICE:%100%Yes#~%0%No#~%0%May be#}
- {1:MULTICHOICE:%100%Yes#~%0%No#~%0%May be#}

increase cpu utilisation (Embedded answers (Cloze))

[Dash...](#) / [My...](#) / [Computer Eng...](#) / [CEIT-even...](#) / [OS-even...](#) / [Theory: rand...](#) / [Random Quiz - 2: bootloader, system calls, fork-exec, op...](#)

Started on Monday, 16 January 2023, 9:00 PM

State Finished

Completed on Monday, 16 January 2023, 10:09 PM

Time taken 1 hour 9 mins

Grade 9.27 out of 15.00 (61.78%)

Question **1**

Incorrect

Mark 0.00 out of 1.00

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 **2**

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 BIOS allows us to choose the boot device, the device from which the boot loader will be loaded ✔️
- ☐ b. The choice of the boot loader (e.g. GRUB or Windows-Loader)
- ☐ c. The choice of booting slowly or fast
- ☐ d. The choice of which OS to boot from

The correct answer is: The BIOS allows us to choose the boot device, the device from which the boot loader will be loaded

Question 3

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

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 4

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. Bootloader must be one sector in length
- ☒ b. Bootloaders allow selection of OS to boot from ✓
- ☐ c. The bootloader loads the BIOS
- ☒ d. Modern Bootloaders often allow configuring the way an OS boots ✓
- ☒ 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 5

Incorrect

Mark 0.00 out of 3.00

Select correct statements about mounting

Select one or more:

- ☒ a. 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 ✖
- ☒ b. The existing name-space at the mount-point is no longer visible after mounting ✔
- ☒ c. Mounting is attaching a disk-partition with a filesystem on it, into another file system name-space ✔
- ☐ d. On Linuxes mounting can be done only while booting the OS
- ☐ e. Mounting makes all disk partitions available as one name space
- ☐ f. Mounting deletes all data at the mount-point
- ☐ g. The mount point must be a directory
- ☐ h. It's possible to mount a partition on one computer, into namespace of another computer.
- ☐ i. 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.
- ☒ j. The mount point can be a file as well ✖

Your answer is incorrect.

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 6

Correct

Mark 1.00 out of 1.00

A process blocks itself means

- ☒ a. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler ✔
- ☐ b. The application code calls the scheduler
- ☐ c. The kernel code of system call calls scheduler
- ☐ d. The kernel code of an interrupt handler, 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 7

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 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 segment is multiplied by 16, in protected mode segment is used as index in GDT
- ☐ 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 8

Partially correct

Mark 0.67 out of 1.00

Select the correct statements about hard and soft links

Select one or more:

- ☐ a. Deleting a soft link deletes only the actual file
- ☐ b. Hard links increase the link count of the actual file inode
- ☐ c. Hard links can span across partitions while soft links can't
- ☐ d. Soft links increase the link count of the actual file inode
- ☒ e. Deleting a hard link deletes the file, only if link count was 1 ✓
- ☒ f. Soft links can span across partitions while hard links can't ✓
- ☒ g. Hard links share the inode ✓
- ☐ h. Soft link shares the inode of actual file
- ☐ i. Hard links enforce separation of filename from it's metadata in on-disk data structures.
- ☐ j. Deleting a soft link deletes both the link and the actual file
- ☒ k. Deleting a soft link deletes the link, not the actual file ✓
- ☐ l. Deleting a hard link always deletes the 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 **9**

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

Your answer is correct.

The correct answer is: run ls twice

Question **10**

Correct

Mark 1.00 out of 1.00

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

- ☒ a. no ✓ wrong
- ☐ b. yes

The correct answer is: no

Question 11

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. Program prints hello
- ☒ d. "ls" runs on current directory ✓
- ☐ e. Execution fails as the call to `execl()` in `exec1` fails

Your answer is correct.

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

Question **12**

Not answered

Marked out of 1.00

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

Shell	<input data-bbox="280 324 422 374" type="text" value="Choose..."/>
Boot loader	<input data-bbox="280 378 422 427" type="text" value="Choose..."/>
BIOS	<input data-bbox="280 432 422 481" type="text" value="Choose..."/>
Login interface	<input data-bbox="280 486 422 535" type="text" value="Choose..."/>
OS	<input data-bbox="280 539 422 589" type="text" value="Choose..."/>
Init	<input data-bbox="280 593 422 647" type="text" value="Choose..."/>

Your answer is incorrect.

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

[◀ Random Quiz - 1 \(Pre-Requisite Quiz\)](#)

Jump to...

[Random Quiz - 3 \(processes, memory management, event driven kernel\), compilation-linking-loading, ipc-pipes ▶](#)

[Das...](#) / [My...](#) / [Computer En...](#) / [CEIT-even...](#) / [OS-even...](#) / [Theory: ran...](#) / [Random Quiz - 3 \(processes, memory management, event driv...](#)

Started on	Thursday, 2 February 2023, 9:00 PM
State	Finished
Completed on	Thursday, 2 February 2023, 11:00 PM
Time taken	1 hour 59 mins
Grade	13.90 out of 20.00 (69.52%)

Question 1

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];
```

```
    pipe(
```

```
);
```

```
    pid1 =
```

```
;
```

```
    if(pid1 != 0) {
```

```
        close(pfd[0]
```

```
);
```

```
        close(
```

```
);
```

```
        dup(
```

```
);
```

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

```
", NULL);
```

```
    }
```

```
    pipe(
```

```
);
```

```
= fork();
```

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

```
        close(
```

```
;
```

```
        close(0);
```

```
        dup(
```

```
);
```

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

```

);
    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
            [1][0]
        );
        execl("/usr/bin/tail", "/usr/bin/tail", "
            -1
        ", NULL);
    }
}

```

Question **2**

Complete

Mark 1.00 out of 1.00

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

- ☐ a. Convert high level language code to machine code
- ☐ b. Check the program for syntactical errors
- ☒ c. Check the program for logical errors
- ☐ d. Invoke the linker to link the function calls with their code, extern globals with their declaration
- ☒ e. Suggest alternative pieces of code that can be written
- ☐ 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 **3**

Complete

Mark 1.00 out of 1.00

A process blocks itself means

- ☒ a. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler
- ☐ b. The application code calls the 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 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 **4**

Complete

Mark 0.50 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)

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

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

Question 5

Complete

Mark 0.88 out of 1.00

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

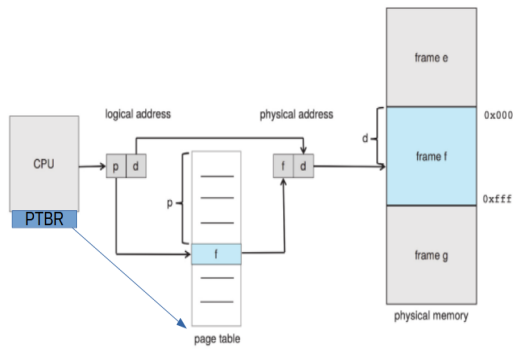


Figure 9.8 Paging hardware.

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

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

The page table is indexed using frame number: False

The page table is itself present in Physical memory: True

The page table is indexed using page number: 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 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 PTBR is present in the CPU as a register: True

Question **6**

Complete

Mark 0.71 out of 1.00

Order the events that occur on a timer interrupt:

Set the context of the new process

Select another process for execution

Jump to a code pointed by IDT

Save the context of the currently running process

Jump to scheduler code

Execute the code of the new process

Change to kernel stack of currently running process

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

Question 7

Complete

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 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
P1 running again
- ☐ 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
- ☐ d.
P1 running
P1 makes sytem call
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again
- ☒ 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
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

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 8

Complete

Mark 0.00 out of 1.00

Select all the correct statements about named pipes and ordinary(unnamed) pipe

Select one or more:

- ☒ a. named pipes are more efficient than ordinary pipes
- ☒ b. a named pipe exists as a file on the file system
- ☒ c. named pipes can be used between multiple processes but ordinary pipes can not be used
- ☐ d. both named and unnamed pipes require some kind of agreed protocol to be effectively used among multiple processes
- ☐ e. named pipe exists even if the processes using it do exit()
- ☒ f. ordinary pipe can only be used between related processes
- ☒ g. named pipe can be used between any processes

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 9

Complete

Mark 1.00 out of 1.00

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

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

The correct answer is: Intermediate code generation → 3, Syntactical Analysis → 2, Loading → does not exist, Pre-processing → 1, Linking → 4

Question **10**

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 **11**

Complete

Mark 0.47 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
- ☒ b. `init()` typically keeps calling `wait()` for zombie processes to get cleaned up
- ☒ c. A zombie process occupies space in OS data structures
- ☐ 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 can become zombie if it finishes, but the parent has finished before it
- ☐ f. A zombie process remains zombie forever, as there is no way to clean it up
- ☒ g. Zombie processes are harmless even if OS is up for long time
- ☒ h. A process becomes zombie when it finishes, and remains zombie until parent calls `wait()` on it

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**

Complete

Mark 0.71 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_RDONLY);  
    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;  
}
```

fd2	<input type="text" value="/tmp/3"/>
fd3	<input type="text" value="closed"/>
2	<input type="text" value="stderr"/>
fd1	<input type="text" value="/tmp/1"/>
0	<input type="text" value="/tmp/2"/>
fd4	<input type="text" value="None of these"/>
1	<input type="text" value="/tmp/3"/>

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

Question **13**

Complete

Mark 1.60 out of 2.00

Match the elements of C program to their place in memory

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

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

Question **14**

Complete

Mark 0.20 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. Program 2 does 1>&2
- ☒ b. Program 1 does 1>&2
- ☒ c. Both programs are correct
- ☒ d. Program 1 makes sure that there is one file offset used for '2' and '1'
- ☐ e. Only Program 1 is correct
- ☐ f. Only Program 2 is correct
- ☐ g. Program 2 is correct for > /tmp/ddd but not for 2>&1
- ☐ h. Program 2 makes sure that there is one file offset used for '2' and '1'
- ☐ i. Both program 1 and 2 are incorrect
- ☐ j. Program 1 ensures 2>&1 and does not ensure > /tmp/ddd
- ☒ k. Program 1 is correct for > /tmp/ddd but not for 2>&1
- ☐ l. Program 2 ensures 2>&1 and does not ensure > /tmp/ddd

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 **15**

Complete

Mark 0.00 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
P1 running again
- ☐ 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
timer interrupt
Scheduler
P2 running
timer interrupt
Scheuler
P1 running
P1's system call return
- ☐ d. P1 running
P1 makes sytem call
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again
- ☒ 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

◀ [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, 7:26 PM
Time taken	1 hour 6 mins
Overdue	10 mins 33 secs
Grade	7.07 out of 10.00 (70.73%)

Question **1**

Correct

Mark 1.00 out of 1.00

What's the trapframe in xv6?

- ☐ a. The IDT table
- ☐ b. A frame of memory that contains all the trap handler's addresses
- ☐ c. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware 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 code in trapasm.S only
- ☒ f. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S ✓
- ☐ 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 1.00 out of 1.00

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

bootmain()	<input type="text" value="gdt setup with 3 entries, at start32 symbol of bootasm.S"/>	✓
after startothers() in main()	<input type="text" value="gdt setup with 5 entries (0 to 4) on all processors"/>	✓
bootasm.S	<input type="text" value="gdt setup with 3 entries, at start32 symbol of bootasm.S"/>	✓
entry.S	<input type="text" value="gdt setup with 3 entries, at start32 symbol of bootasm.S"/>	✓
kvmalloc() in main()	<input type="text" value="gdt setup with 3 entries, at start32 symbol of bootasm.S"/>	✓
after seginit() in main()	<input type="text" value="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, 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, 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, after seginit() in main() → gdt setup with 5 entries (0 to 4) on one processor

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?

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

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 **4**

Incorrect

Mark 0.00 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. The kernel.asm file is the final kernel file ❌
- ☒ b. The bootmain() code does not read the kernel completely in memory ❌
- ☒ 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 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.
- ☐ e. The kernel.ld file contains instructions to the linker to link the kernel properly
- ☒ f. Although 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. ❌
- ☒ g. Although 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. ✓
- ☒ h. The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files ✓
- ☐ i. readseg() reads first 4k bytes of kernel in memory

Your answer is incorrect.

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, Although 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

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 upper 13 bits are used for segment number ✓
- ☐ b. The value 8 is stored in code segment
- ☐ 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 correct.

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

Question 6

Correct

Mark 1.00 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. xv6 uses the 0x64th entry in IDT for system calls
- ☒ c. The CS and EIP are changed only after pushing user code's SS,ESP on stack ✓
- ☐ d. The function trap() is the called only in case of hardware interrupt
- ☒ e. xv6 uses the 64th entry in IDT for system calls ✓
- ☐ f. On any interrupt/syscall/exception the control first jumps in trapasm.S
- ☒ g. On any interrupt/syscall/exception the control first jumps in vectors.S ✓
- ☒ h. Before going to alltraps, the kernel stack contains upto 5 entries. ✓
- ☒ i. All the 256 entries in the IDT are filled in xv6 code ✓
- ☒ j. The trapframe pointer in struct proc, points to a location on process's kernel stack ✓
- ☒ k. Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt ✓
- ☐ l. The CS and EIP are changed immediately (as the first thing) on a hardware interrupt
- ☒ m. The function trap() is the called even if any of the hardware interrupt/system-call/exception occurs ✓

Your answer is correct.

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 **7**

Partially correct

Mark 0.27 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		
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The free page-frame are created out of nearly 222 MB	✗
<input checked="" type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The stack allocated in entry.S is used as stack for scheduler's context for first processor	✓
<input checked="" type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The kernel's page table given by kpgdir variable is used as stack for scheduler's context	✗
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/> <input checked="" type="checkbox"/>	The switchkvm() call in scheduler() changes CR3 to use page directory of new process	✓
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/> <input checked="" type="checkbox"/>	PHYSTOP can be increased to some extent, simply by editing memlayout.h	✗
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The process's address space gets mapped on frames, obtained from ~2MB:224MB range	✗
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context	✗
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/> <input checked="" type="checkbox"/>	xv6 uses physical memory upto 224 MB only	✗
<input checked="" type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir	✓
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context	✗
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The kernel code and data take up less than 2 MB space	✗

The free page-frame are created out of nearly 222 MB: True

The stack allocated in entry.S is used as stack for scheduler's context for first processor: True

The kernel's page table given by kpgdir variable is used as stack for scheduler's context: False

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 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 switchkvm() call in scheduler() changes CR3 to use page directory kpgdir: 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 kernel code and data take up less than 2 MB space: True

Question 8

Partially correct

Mark 0.80 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	Immaterial as the stack is not used here	✓
jmp *%eax in entry.S	0x10000 to 0x7c00	✗
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	✓

Your answer is partially correct.

You have correctly selected 4.

The correct answer is: ljmp \$(SEG_KCODE<<3), \$start32

in bootasm.S → Immaterial as the stack is not used here, jmp *%eax

in entry.S → The 4KB area in kernel image, loaded in memory, named as 'stack', 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

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

Your answer is correct.

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 **10**

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
- ☒ 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; after that calling convention applies, hence code can be written in C ✓
- ☐ d. The code for reading ELF file can not be written in assembly

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

[◀ 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 41 secs
Grade	8.41 out of 15.00 (56.08%)

Question **1**

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.

- ☐ a. ext2 is by default a log structured file system
- ☒ b. xv6 has a log structured file system ✓
- ☒ c. ext4 is a log structured file system ✗ it's a journaled file system, not log structured
- ☒ d. file system recovery recovers all the lost data ✗
- ☒ e. log structured file systems considerably improve the recovery time ✓

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 **2**

Partially correct

Mark 0.50 out of 1.00

Select all the actions taken by ilock()

- ☐ a. Mark the in-memory inode as valid, if needed
- ☐ b. Get the inode from the inode-cache
- ☒ c. Read the inode from disk, if needed ✓
- ☒ d. Take the sleeplock on the inode, always ✓
- ☐ e. Lock all the buffers of the file in memory
- ☐ f. Take the sleeplock on the inode, optionally
- ☐ g. Copy the on-disk inode into in-memory inode, if needed

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 needed, Take the sleeplock on the inode, always, Mark the in-memory inode as valid, if needed

Question 3

Partially correct

Mark 1.50 out of 2.00

Marks the statements as True/False w.r.t. "struct buf"

True	False		
<input type="radio"/>	<input checked="" type="radio"/>	B_VALID means the buffer is empty and can be reused	✓ No. it means it contains data, same as the data on disk
<input checked="" type="radio"/>	<input type="radio"/>	B_DIRTY flag means the buffer contains modified data	✓
<input checked="" type="radio"/>	<input type="radio"/>	A buffer can be both on the MRU/LRU list and also on idequeue list.	✓
<input type="radio"/>	<input checked="" type="radio"/>	The "next" pointer chain gives the buffers in LRU order	✗ No. MRU order.
<input checked="" type="radio"/>	<input type="radio"/>	The reference count (refcnt) in struct buf is = number of processes accessing the buffer	✓
<input type="radio"/>	<input checked="" type="radio"/>	A buffer can have both B_VALID and B_DIRTY flags set	✓ only one will be set
<input checked="" type="radio"/>	<input type="radio"/>	Lock on a buffer is acquired in bget, and released in brelse	✓
<input checked="" type="radio"/>	<input type="radio"/>	The buffers are maintained in LRU order, in the function brelse	✗

B_VALID means the buffer is empty and can be reused: False

B_DIRTY flag means the buffer contains modified data: True

A buffer can be both on the MRU/LRU list and also on idequeue list.: True

The "next" pointer chain gives the buffers in LRU order: False

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

Lock on a buffer is acquired in bget, and released in brelse: True

The buffers are maintained in LRU order, in the function brelse: True

Question **4**

Correct

Mark 1.00 out of 1.00

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

namei	pathname lookup	✓
bmap	inode	✓
stati	inode	✓
ialloc	inode	✓
bread	buffer cache	✓
filestat()	file descriptor	✓
skipelem	pathname lookup	✓
ideintr	disk driver	✓
commit	logging	✓
sys_chdir()	system call	✓
dirlookup	directory	✓
balloc	block allocation on disk	✓

Your answer is correct.

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

Question **5**

Partially correct

Mark 0.75 out of 1.00

Match function with it's functionality

namex	return on-disk inode for a given pathname	✗
nameiparent	return in-memory inode for parent directory of a given pathname	✓
dirlookup	Search a given name in a given directory	✓
dirlink	Write a new entry in a given directory	✓

Your answer is partially correct.

You have correctly selected 3.

The correct answer is: namex → return in-memory inode for a given pathname, nameiparent → return in-memory inode for parent directory of a given pathname, dirlookup → Search a given name in a given directory, dirlink → Write a new entry in a given directory

Question **6**

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 **7**

Partially correct

Mark 1.43 out of 2.00

Select T/F w.r.t physical disk handling in xv6 code

True	False		
<input checked="" type="radio"/>	<input type="radio"/>	the superblock does not contain number of free blocks	✓
<input checked="" type="radio"/>	<input type="radio"/>	disk driver handles only one buffer at a time	✓
<input checked="" type="radio"/>	<input type="radio"/>	only 2 disks are handled by default	✓
<input checked="" type="radio"/>	<input type="radio"/>	The code supports IDE, and not SATA/SCSI	✓
<input type="radio"/>	<input checked="" type="radio"/>	only direct blocks are supported	✗
<input type="radio"/>	<input checked="" type="radio"/>	device files are not supported	✗
<input checked="" type="radio"/>	<input type="radio"/>	log is kept on the same device as the file system	✓

the superblock does not contain number of free blocks: True

disk driver handles only one buffer at a time: True

only 2 disks are handled by default: True

The code supports IDE, and not SATA/SCSI: True

only direct blocks are supported: False

device files are not supported: False

log is kept on the same device as the file system: True

Question 8

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, arr, 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		
<input checked="" type="radio"/>	<input type="radio"/>	The ONLY function that gets called on <code>return devsw[ip->major].read(ip, dst, n);</code> is <code>consoleread</code>	<input type="checkbox"/> There is no other device file in xv6
<input type="radio"/>	<input checked="" type="radio"/>	The data is transferred from disk to kernel buffers first, and then address of <code>arr</code> is mapped to the kernel buffers	<input type="checkbox"/> No. data is copied into <code>arr</code> .
<input checked="" type="radio"/>	<input type="radio"/>	The process will be made to sleep only once	<input checked="" type="checkbox"/> Yes. Reading 100 bytes means reading only one disk block. So <code>bread()->iderw()</code> is called only once.
<input checked="" type="radio"/>	<input type="radio"/>	value of <code>fd</code> will be 3	<input checked="" type="checkbox"/>
<input type="radio"/>	<input checked="" type="radio"/>	The loop in <code>readi()</code> will always read a different block using <code>bread()</code>	<input checked="" type="checkbox"/> No. the offsets can overlap over the same block
<input type="radio"/>	<input checked="" type="radio"/>	The <code>"memmove(dst, bp->data + off%BSIZE, m);"</code> in <code>readi()</code> will copy the data from the disk to the kernel buffers	<input type="checkbox"/> It will transfer from kernel buffer to user memory (<code>arr</code>)

The ONLY function that gets called on `return devsw[ip->major].read(ip, dst, n);` is `consoleread`: True

The data is transferred from disk to kernel buffers first, and then address of `arr` is mapped to the kernel buffers: False

The process will be made to sleep only once: True

value of `fd` will be 3: True

The loop in `readi()` will always read a different block using `bread()`: False

The `"memmove(dst, bp->data + off%BSIZE, m);"` in `readi()` will copy the data from the disk to the kernel buffers: False

Question **9**

Partially correct

Mark 0.40 out of 1.00

Arrange the following in their typical order of use in xv6.

1. ☒ iget
2. ☒ use inode
3. ☒ ilock
4. ☒ iunlock
5. ☒ iput

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. 0 / 1 = 0%
2. 0 / 1 = 0%
3. 0 / 1 = 0%
4. 1 / 1 = 100%
5. 1 / 1 = 100%

The correct order for these items is as follows:

1. iget
2. ilock
3. use inode
4. iunlock
5. iput

Question **10**

Incorrect

Mark 0.00 out of 1.00

An inode is read from disk as a part of this function

- ☒ a. iget
- ☐ b. iread
- ☐ c. sys_read
- ☐ d. ilock
- ☐ e. readi

Your answer is incorrect.

The correct answer is: ilock

Question **11**

Correct

Mark 1.00 out of 1.00

The lines

```
if(ip->type != T_DIR){  
    iunlockput(ip);  
    return 0;  
}
```

in namex() function

mean

- ☒ a. One of the sub-components on the given path name, was not a directory, hence it's an error ✓
- ☒ b. One of the sub-components on the given path name, did not exist, hence it's an error ✗
- ☒ c. 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 ✗
- ☒ d. ilock is held on the inode, and hence it's an error if it is a directory ✗
- ☒ e. No directory entry was found for the file to be opened, hence an error ✗
- ☒ f. The last path component (which is a file, and not a directory) has been resolved, so release the lock (using iunlockput) and return ✗
- ☒ g. There was a syntax error in the pathname specified ✗

Your answer is correct.

The correct answer is: One of the sub-components on the given path name, was not a directory, hence it's an error

Question **12**

Partially correct

Mark 0.67 out of 1.00

Select all the actions taken by iget()

- ☐ a. Returns the inode locked
- ☒ b. Returns a free-inode , with dev+inode-number set, if not found in cache ✓
- ☒ c. Returns an inode with given dev+inode-number from cache, if it exists in cache ✓
- ☐ d. Returns the inode with reference count incremented
- ☐ e. Returns a valid inode if not found in cache
- ☐ f. Returns the inode with inode-cache lock held
- ☐ g. Panics if inode does not exist in cache

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

Compare XV6 and EXT2 file systems.

Select True/False for each point.

True	False		
<input type="radio"/>	<input checked="" type="radio"/>	Both xv6 and ext2 contain magic number	✓
<input type="radio"/>	<input checked="" type="radio"/>	xv6 contains inode bitmap, but ext2 does not	✓
<input checked="" type="radio"/>	<input type="radio"/>	Ext2 contains group descriptors but xv6 does not	✓
<input checked="" type="radio"/>	<input type="radio"/>	In both ext2 and xv6, the superblock gives location of first inode block	✗
<input checked="" type="radio"/>	<input type="radio"/>	xv6 contains journal, ext2 does not	✓
<input checked="" type="radio"/>	<input type="radio"/>	Ext2 contains superblock but xv6 does not.	✗

Both xv6 and ext2 contain magic number: False

xv6 contains inode bitmap, but ext2 does not: False

Ext2 contains group descriptors but xv6 does not: True

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

◀ Random Quiz - 5: xv6 make, bootloader, interrupt handling, memory management

Jump to...

[Dash...](#) / [My c...](#) / [Computer Engi...](#) / [CEIT-even-...](#) / [OS-even-s...](#) / [Theory: rand...](#) / [Random Quiz 4 : Scheduling, signals, segmentation, p...](#)

Started on Thursday, 16 February 2023, 9:01 PM

State Finished

Completed on Thursday, 16 February 2023, 10:21 PM

Time taken 1 hour 19 mins

Grade 12.83 out of 15.00 (85.51%)

Question **1**

Correct

Mark 1.00 out of 1.00

Match the names of PCB structures with kernel

xv6 ✓

linux ✓

The correct answer is: xv6 → struct proc, linux → struct task_struct

Question **2**

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. Response time is more predictable in preemptive systems than in non preemptive systems.
- ☒ d. Real time systems generally use non preemptive CPU scheduling. ✓

Your answer is correct.

The correct answer is: Real time systems generally use non preemptive CPU scheduling.

Question 3

Partially correct

Mark 1.56 out of 2.00

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

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

Your answer is partially correct.

You have correctly selected 7.

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 4

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)

- | | | |
|---------------------------|---|---|
| Segmentation, then paging | many continuous chunks of variable size | ✓ |
| Paging | one continuous chunk | ✓ |
| Segmentation | many continuous chunks of variable size | ✓ |
| Relocation + Limit | one continuous chunk | ✓ |

Your answer is correct.

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

Question 5

Correct

Mark 1.00 out of 1.00

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

Process P0 is running

 ✓

context of P1 is loaded from P1's PCB

 ✓

context of P0 is saved in P0's PCB

 ✓

Process P1 is running

 ✓

timer interrupt occurs

 ✓

Control is passed to P1

 ✓

Your answer is correct.

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

Question 6

Correct

Mark 1.00 out of 1.00

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

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

Your answer is correct.

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

Question 7

Partially correct

Mark 0.57 out of 1.00

Mark True/False

Statements about scheduling and scheduling algorithms

True	False		
<input checked="" type="radio"/>	<input type="radio"/>	A scheduling algorithm is non-preemptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.	✓
<input checked="" type="radio"/>	<input type="radio"/>	xv6 code does not care about Processor Affinity	✗
<input checked="" type="radio"/>	<input type="radio"/>	Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.	✓
<input checked="" type="radio"/>	<input type="radio"/>	Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.	✓
<input checked="" type="radio"/>	<input type="radio"/>	Response time will be quite poor on non-interruptible kernels	✗
<input checked="" type="radio"/>	<input type="radio"/>	Processor Affinity refers to memory accesses of a process being stored on cache of that processor	✓
<input type="radio"/>	<input checked="" type="radio"/>	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.

A scheduling algorithm is non-preemptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.: True

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

Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.: True

Response time will be quite poor on non-interruptible kernels: True

Processor Affinity refers to memory accesses of a process being stored on cache of that processor: True

On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread: False

Question 8

Partially correct

Mark 0.80 out of 1.00

Mark whether the concept is related to scheduling or not.

Yes	No	
<input checked="" type="radio"/>	<input type="radio"/>	timer interrupt
<input checked="" type="radio"/>	<input type="radio"/>	context-switch
<input checked="" type="radio"/>	<input type="radio"/>	ready-queue
<input type="radio"/>	<input checked="" type="radio"/>	file-table
<input checked="" type="radio"/>	<input type="radio"/>	runnable process

timer interrupt: Yes

context-switch: Yes

ready-queue: Yes

file-table: No

runnable process: Yes

Question 9

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. expressions
- ☒ b. typedefs ✓
- ☐ c. function calls
- ☒ d. #directives ✓
- ☒ e. global variables ✓
- ☐ f. local variable declaration
- ☐ g. pointer dereference

Your answer is correct.

The correct answers are: #directives, typedefs, global variables

Question 10

Partially correct

Mark 0.40 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!

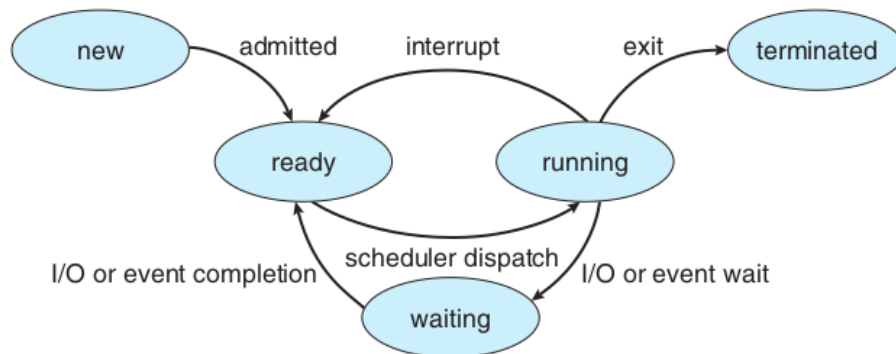


Figure 3.2 Diagram of process state.

True	False		
<input checked="" type="radio"/>	<input type="radio"/>	A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first	✗
<input type="radio"/>	<input checked="" type="radio"/>	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.	✓
<input checked="" type="radio"/>	<input type="radio"/>	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	✗
<input checked="" type="radio"/>	<input type="radio"/>	Every forked process has to go through ZOMBIE state, at least for a small duration.	✓
<input checked="" type="radio"/>	<input type="radio"/>	Only a process in READY state is considered by scheduler	✗

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

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 11

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 12

Correct

Mark 1.00 out of 1.00

Map each signal with it's meaning

SIGALRM Timer Signal from alarm() ✓

SIGPIPE Broken Pipe ✓

SIGCHLD Child Stopped or Terminated ✓

SIGUSR1 User Defined Signal ✓

SIGSEGV Invalid Memory Reference ✓

The correct answer is: SIGALRM → Timer Signal from alarm(), SIGPIPE → Broken Pipe, SIGCHLD → Child Stopped or Terminated, SIGUSR1 → User Defined Signal, SIGSEGV → Invalid Memory Reference

Question 13

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

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 14

Correct

Mark 1.00 out of 1.00

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

Your answer is correct.

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

◀ [Random Quiz - 3 \(processes, memory management, event driven kernel\), compilation-linking-loading, ipc-pipes](#)

Jump to...

[Homework questions: Basics of MM, xv6 booting](#) ►

• Match the elements of C program to their place in memory

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

C program to segment (Matching)

• Match the File descriptors to their meaning

- 0-> Standard Input
- 1-> Standard output
- 2-> Standard error

FDs to meaning (Matching)

• Match the MACRO with it's meaning

- PHYSTOP-> 224 MB
- KERNBASE-> 2 GB
- KERNLINK-> 2.224 GB
- -> 2.1 GB
- -> 2 MB

Meaning of MACROS in MM (wrong choice 2.224) (Matching)

• Match the names of PCB structures with kernel

- xv6-> struct proc
- linux-> struct task_struct
- -> struct process
- -> struct task_structure
- -> struct process_struct

PCB names (Matching)

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

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

Syscall order correctly (Matching)

• A process blocks itself means

- a. (100%) The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler
- b. (0%) The application code calls the scheduler
- c. (0%) The kernel code of system call calls scheduler
- d. (0%) The kernel code of an interrupt handler, moves the process to a waiting queue and calls scheduler

Blocking means (Multiple choice / One answer only)

• 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. (100%) 3 1 1
- b. (0%) 3 4 5
- c. (0%) 3 1 2
- d. (0%) 1 1 1
- e. (0%) 2 2 2
- f. (0%) 3 3 3

FD output (Multiple choice / One answer only)

• Which of the following is not a task of the code of swtch() function

- a. (50%) Save the return value of the old context code
- b. (50%) Change the kernel stack location
- c. (0%) Save the old context
- d. (0%) Load the new context
- e. (0%) Jump to next context EIP
- f. (0%) Switch stacks

Not done by swtch() (Multiple choice)

• Which of the following state transitions are not possible?

- a. (33.33333%) Ready -> Terminated
- b. (33.33333%) Waiting -> Terminated
- c. (-100%) Running -> Waiting
- d. (33.33333%) Ready -> Waiting

Not possible state transition (Multiple choice)

• Select the odd one out

- a. (100%) Kernel stack of new process to kernel stack of scheduler
- b. (0%) Process stack of running process to kernel stack of running process
- c. (0%) Kernel stack of running process to kernel stack of scheduler
- d. (0%) Kernel stack of scheduler to kernel stack of new process
- e. (0%) Kernel stack of new process to Process stack of new process

Odd (stack transition) out (Multiple choice / One answer only)

• The "push 0" in vectors.S is

- a. (100%) Place for the error number value
- b. (0%) To be filled in as the return value of the system call
- c. (0%) A placeholder to match the size of struct trapframe
- d. (0%) To indicate that it's a system call and not a hardware interrupt

push 0 for errno (Multiple choice / One answer only)

• The trapframe, in xv6, is built by the

- a. (100%) hardware, vectors.S, trapasm.S
- b. (0%) vectors.S, trapasm.S
- c. (0%) hardware, vectors.S
- d. (0%) hardware, trapasm.S
- e. (0%) hardware, vectors.S, trapasm.S, trap()

Who builds trapframe? (Multiple choice / One answer only)

- Calculate the EAT in NANO-seconds (upto 2 decimal points) w.r.t. a page fault, given Memory access time = {m} ns Average page fault service time = {t} ms Page fault rate = {p}

Answer: $(1-\{p\}) \cdot \{m\} + \{p\} \cdot \{t\} \cdot 1000000$

EAT (page fault) (Calculated)

- Map the parts of a C code to the memory regions they are related to

- global initialized variables-> data
- static variables-> data
- global un-initialized variables-> bss
- functions-> code
- local variables-> stack
- function arguments-> stack
- mallocced memory-> heap
- -> buffers

C code parts, region mapping (Matching)

- 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 3-> 9
- Page table of P1, Page 4-> 23
- Page table of P1, Page 5-> 7
- Page table of P2, Page 0-> 9
- Page table of P2, Page 1-> 15
- Page table of P2, Page 3-> 4
- Page table of P2, Page 4-> 7
- -> 2
- -> 3
- -> 0
- -> 5
- -> 6

Find page table entries (Matching)

- 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

static/dynamic link/load (Matching)

- 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.basic55a435049000-55a435248000 r-xp 0002e000 103:05 917529 /usr/bin/vim.basic55a435248000-55a4352b6000 r--p 0022d000 103:05 917529 /usr/bin/vim.basic55a4352c5000-55a4352e2000 rw-p 002a9000 103:05 917529 /usr/bin/vim.basic55a4352e2000-55a4352f0000 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.so7f275b0a6000-7f275b0ad000 r-xp 00003000 103:05 917901 /usr/lib/x86_64-linux-gnu/libnss_files-2.31.so7f275b0ad000-7f275b0af000 r--p 0000a000 103:05 917901 /usr/lib/x86_64-linux-gnu/libnss_files-2.31.so7f275b0af000-7f275b0b0000 r--p 0000b000 103:05 917901 /usr/lib/x86_64-linux-gnu/libnss_files-2.31.so7f275b0b0000-7f275b0b1000 rw-p 0000c000 103:05 917901 /usr/lib/x86_64-linux-gnu/libnss_files-2.31.so7f275b0b1000-7f275b0b7000 rw-p 00000000 00:00 0 7f275b0b7000-7f275b8f5000 r--p 00000000 103:05 925247 /usr/lib/locale/locale-archive7f275b8f5000-7f275b8fa000 rw-p 00000000 00:00 0 7f275b8fa000-7f275b8fc000 r--p 00000000 103:05 924216 /usr/lib/x86_64-linux-gnu/libogg.so.0.8.47f275b8fc000-7f275b901000 r-xp 00002000 103:05 924216 /usr/lib/x86_64-linux-gnu/libogg.so.0.8.47f275b901000-7f275b904000 r--p 00007000 103:05 924216 /usr/lib/x86_64-linux-gnu/libogg.so.0.8.47f275b904000-7f275b905000 ---p 0000a000 103:05 924216 /usr/lib/x86_64-linux-gnu/libogg.so.0.8.47f275b905000-7f275b906000 r--p 0000a000 103:05 924216 /usr/lib/x86_64-linux-gnu/libogg.so.0.8.47f275b906000-7f275b907000 rw-p 0000b000 103:05 924216 /usr/lib/x86_64-linux-gnu/libogg.so.0.8.47f275b907000-7f275b90a000 r--p 00000000 103:05 924627 /usr/lib/x86_64-linux-gnu/libvorbis.so.0.4.87f275b90a000-7f275b921000 r-xp 00003000 103:05 924627 /usr/lib/x86_64-linux-gnu/libvorbis.so.0.4.87f275b921000-7f275b932000 r--p 0001a000 103:05 924627 /usr/lib/x86_64-linux-gnu/libvorbis.so.0.4.87f275b932000-7f275b933000 ---p 0002b000 103:05 924627 /usr/lib/x86_64-linux-gnu/libvorbis.so.0.4.87f275b933000-7f275b934000 r--p 0002b000 103:05 924627 /usr/lib/x86_64-linux-gnu/libvorbis.so.0.4.87f275b934000-7f275b935000 rw-p 0002c000 103:05 924627 /usr/lib/x86_64-linux-gnu/libvorbis.so.0.4.87f275b935000-7f275b937000 rw-p 00000000 00:00 0 7f275b937000-7f275b938000 r--p 00000000 103:05 917914 /usr/lib/x86_64-linux-gnu/libutil-2.31.so7f275b938000-7f275b939000 r-xp 00001000 103:05 917914 /usr/lib/x86_64-linux-gnu/libutil-2.31.so7f275b939000-7f275b93b000 r--p 00002000 103:05 917914 /usr/lib/x86_64-linux-gnu/libutil-2.31.so7f275b93b000-7f275b93c000 rw-p 00003000 103:05 917914 /usr/lib/x86_64-linux-gnu/libutil-2.31.so7f275b93c000-7f275b93e000 r--p 00000000 103:05 915906 /usr/lib/x86_64-linux-gnu/libz.so.1.2.117f275b93e000-7f275b94f000 r-xp 00002000 103:05 915906 /usr/lib/x86_64-linux-gnu/libz.so.1.2.117f275b94f000-7f275b950000 r--p 00013000 103:05 915906 /usr/lib/x86_64-linux-gnu/libz.so.1.2.117f275b95000-7f275b956000 ---p 00019000 103:05 915906 /usr/lib/x86_64-linux-gnu/libz.so.1.2.117f275b956000-7f275b957000 r--p 00019000 103:05 915906 /usr/lib/x86_64-linux-gnu/libz.so.1.2.117f275b957000-7f275b958000 rw-p 0001a000 103:05 915906 /usr/lib/x86_64-linux-gnu/libz.so.1.2.117f275b958000-7f275b95c000 r--p 00000000 103:05 923645 /usr/lib/x86_64-linux-gnu/libexpat.so.1.6.117f275b95c000-7f275b978000 r-xp 00004000 103:05 923645 /usr/lib/x86_64-linux-gnu/libexpat.so.1.6.117f275b978000-7f275b982000 r--p 00020000 103:05 923645 /usr/lib/x86_64-linux-gnu/libexpat.so.1.6.117f275b982000-7f275b983000 ---p 0002a000 103:05 923645 /usr/lib/x86_64-linux-gnu/libexpat.so.1.6.117f275b983000-7f275b985000 r--p 0002a000 103:05 923645 /usr/lib/x86_64-linux-gnu/libexpat.so.1.6.117f275b985000-7f275b986000 rw-p 0002c000 103:05 923645

gnu/libexpat.so.1.6.117f275b986000-7f275b988000 r--p 00000000 103:05 924057 /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.17f275b988000-7f275b98d000 r-xp 00002000 103:05 924057 /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.17f275b98d000-7f275b98f000 r--p 00007000 103:05 924057 /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.17f275b98f000-7f275b990000 r--p 00008000 103:05 924057 /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.17f275b990000-7f275b991000 rw-p 00009000 103:05 924057 /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.17f275b991000-7f275b995000 r--p 00000000 103:05 921934 /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.37f275b995000-7f275b9a3000 r-xp 00004000 103:05 921934 /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.37f275b9a3000-7f275b9a9000 r--p 00012000 103:05 921934 /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.37f275b9a9000-7f275b9aa000 r--p 00017000 103:05 921934 /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.37f275b9aa000-7f275b9ab000 rw-p 00018000 103:05 921934 /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.37f275b9ab000-7f275b9ad000 rw-p 00000000 00:00 0 7f275b9ad000-7f275b9af000 r--p 00000000 103:05 924631 /usr/lib/x86_64-linux-gnu/libvorbisfile.so.3.3.77f275b9af000-7f275b9b4000 r-xp 00002000 103:05 924631 /usr/lib/x86_64-
linux-gnu/libvorbisfile.so.3.3.77f275b9b4000-7f275b9b5000 r--p 00007000 103:05 924631 /usr/lib/x86_64-
linux-gnu/libvorbisfile.so.3.3.77f275b9b5000-7f275b9b6000 ---p 00008000 103:05 924631 /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.77f275b9b6000-7f275b9b7000 r--p 00008000 103:05 924631 /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.77f275b9b7000-7f275b9b8000 rw-p 00009000 103:05 924631 /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.77f275b9b8000-7f275b9ba000 r--p 00000000 103:05 924277 /usr/lib/x86_64-linux-gnu/libpcr2-
8.so.0.9.07f275b9ba000-7f275ba1e000 r-xp 00002000 103:05 924277 /usr/lib/x86_64-linux-gnu/libpcr2-
8.so.0.9.07f275ba1e000-7f275ba46000 r--p 00066000 103:05 924277 /usr/lib/x86_64-linux-gnu/libpcr2-
8.so.0.9.07f275ba46000-7f275ba47000 r--p 0008d000 103:05 924277 /usr/lib/x86_64-linux-gnu/libpcr2-
8.so.0.9.07f275ba47000-7f275ba48000 rw-p 0008e000 103:05 924277 /usr/lib/x86_64-linux-gnu/libpcr2-
8.so.0.9.07f275ba48000-7f275ba6d000 r--p 00000000 103:05 917893 /usr/lib/x86_64-linux-gnu/libc-2.31.so7f275ba6d000-
7f275bbe5000 r-xp 00025000 103:05 917893 /usr/lib/x86_64-linux-gnu/libc-2.31.so7f275bbe5000-7f275bc2f000 r--p 0019d000 103:05 917893 /usr/lib/x86_64-linux-gnu/libc-2.31.so7f275bc2f000-7f275bc30000 ---p 001e7000 103:05 917893 /usr/lib/x86_64-linux-
gnu/libc-2.31.so7f275bc30000-7f275bc36000 rw-p 001ea000 103:05 917893 /usr/lib/x86_64-linux-gnu/libc-
2.31.so7f275bc36000-7f275bc3a000 rw-p 00000000 00:00 0 7f275bc3a000-7f275bc41000 r--p 00000000 103:05 917906 /usr/lib/x86_64-linux-
gnu/libc-2.31.so7f275bc41000-7f275bc52000 r-xp 00007000 103:05 917906 /usr/lib/x86_64-linux-gnu/libpthread-
2.31.so7f275bc52000-7f275bc57000 r--p 00018000 103:05 917906 /usr/lib/x86_64-linux-gnu/libpthread-
2.31.so7f275bc57000-7f275bc58000 r--p 0001c000 103:05 917906 /usr/lib/x86_64-linux-gnu/libpthread-2.31.so7f275bc58000-
7f275bc59000 rw-p 0001d000 103:05 917906 /usr/lib/x86_64-linux-gnu/libpthread-2.31.so7f275bc59000-7f275bc5d000 rw-p 00000000 00:00 0 7f275bc5d000-7f275bcce000 r--p 00000000 103:05 917016 /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.07f275bcce000-7f275bf29000 r-xp 00071000 103:05 917016 /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.07f275bf29000-7f275c142000 r--p 002cc000 103:05 917016 /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.07f275c142000-7f275c143000 ---p 004e5000 103:05 917016 /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.07f275c143000-7f275c149000 r--p 004e5000 103:05 917016 /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.07f275c149000-7f275c190000 rw-p 004eb000 103:05 917016 /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.07f275c190000-7f275c1b3000 rw-p 00000000 00:00 0 7f275c1b3000-7f275c1b4000 r--p 00000000 103:05 917894 /usr/lib/x86_64-linux-gnu/libdl-2.31.so7f275c1b4000-7f275c1b6000 r-xp 00001000 103:05 917894 /usr/lib/x86_64-linux-
gnu/libdl-2.31.so7f275c1b6000-7f275c1b7000 r--p 00003000 103:05 917894 /usr/lib/x86_64-linux-gnu/libdl-
2.31.so7f275c1b7000-7f275c1b9000 rw-p 00004000 103:05 917894 /usr/lib/x86_64-linux-gnu/libdl-2.31.so7f275c1b9000-
7f275c1bb000 rw-p 00000000 00:00 0 7f275c1bb000-7f275c1c0000 r-xp 00000000 103:05 923815 /usr/lib/x86_64-linux-
gnu/libgpm.so.27f275c1c0000-7f275c3bf000 ---p 00005000 103:05 923815 /usr/lib/x86_64-linux-gnu/libgpm.so.27f275c3bf000-
7f275c3c0000 r--p 00004000 103:05 923815 /usr/lib/x86_64-linux-gnu/libgpm.so.27f275c3c0000-7f275c3c1000 rw-p 00005000 103:05 923815 /usr/lib/x86_64-linux-gnu/libgpm.so.27f275c3c1000-7f275c3c3000 r--p 00000000 103:05 923315 /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.22537f275c3c3000-7f275c3c8000 r-xp 00002000 103:05 923315 /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.22537f275c3c8000-7f275c3ca000 r--p 00007000 103:05 923315 /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.22537f275c3ca000-7f275c3cb000 r--p 00008000 103:05 923315 /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.22537f275c3cb000-7f275c3cc000 rw-p 00009000 103:05 923315 /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.22537f275c3cc000-7f275c3cf000 r--p 00000000 103:05 923446 /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.57f275c3cf000-7f275c3d9000 r-xp 00003000 103:05 923446 /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.57f275c3d9000-7f275c3dd000 r--p 0000d000 103:05 923446 /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.57f275c3dd000-7f275c3de000 r--p 00010000 103:05 923446 /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.57f275c3de000-7f275c3df000 rw-p 00011000 103:05 923446 /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.57f275c3df000-7f275c3e5000 r--p 00000000 103:05 924431 /usr/lib/x86_64-linux-
gnu/libselinux.so.17f275c3e5000-7f275c3fe000 r-xp 00006000 103:05 924431 /usr/lib/x86_64-linux-
gnu/libselinux.so.17f275c3fe000-7f275c405000 r--p 0001f000 103:05 924431 /usr/lib/x86_64-linux-
gnu/libselinux.so.17f275c405000-7f275c406000 ---p 00026000 103:05 924431 /usr/lib/x86_64-linux-
gnu/libselinux.so.17f275c406000-7f275c407000 r--p 00026000 103:05 924431 /usr/lib/x86_64-linux-
gnu/libselinux.so.17f275c407000-7f275c408000 rw-p 00027000 103:05 924431 /usr/lib/x86_64-linux-
gnu/libselinux.so.17f275c408000-7f275c40a000 rw-p 00000000 00:00 0 7f275c40a000-7f275c418000 r--p 00000000 103:05 924540 /usr/lib/x86_64-linux-gnu/libtinfo.so.6.27f275c418000-7f275c427000 r-xp 0000e000 103:05 924540 /usr/lib/x86_64-linux-gnu/libtinfo.so.6.27f275c427000-7f275c435000 r--p 0001d000 103:05 924540 /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.27f275c435000-7f275c439000 r--p 0002a000 103:05 924540 /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.27f275c439000-7f275c43a000 rw-p 0002e000 103:05 924540 /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.27f275c43a000-7f275c449000 r--p 00000000 103:05 917895 /usr/lib/x86_64-linux-gnu/libm-
2.31.so7f275c449000-7f275c4f0000 r-xp 0000f000 103:05 917895 /usr/lib/x86_64-linux-gnu/libm-2.31.so7f275c4f0000-
7f275c587000 r--p 000b6000 103:05 917895 /usr/lib/x86_64-linux-gnu/libm-2.31.so7f275c587000-7f275c588000 r--p 0014c000 103:05 917895 /usr/lib/x86_64-linux-gnu/libm-2.31.so7f275c588000-7f275c589000 rw-p 0014d000 103:05 917895 /usr/lib/x86_64-linux-gnu/libm-2.31.so7f275c589000-7f275c58b000 rw-p 00000000 00:00 0 7f275c58b000-7f275c5af000 r--p 00000000 103:05 917889 /usr/lib/x86_64-linux-gnu/libm-2.31.so7f275c5af000-7f275c5d2000 r-xp 00001000 103:05 917889 /usr/lib/x86_64-linux-gnu/libm-2.31.so7f275c5d2000-7f275c5da000 r--p 00024000 103:05 917889 /usr/lib/x86_64-linux-gnu/libm-2.31.so7f275c5da000-7f275c5db000 r--p 0002c000 103:05 917889 /usr/lib/x86_64-linux-gnu/libm-2.31.so7f275c5db000-7f275c5dc000 r--p 0002c000 103:05 917889 /usr/lib/x86_64-linux-gnu/libm-2.31.so7f275c5dc000-7f275c5dd000 rw-p 0002d000 103:05 917889 /usr/lib/x86_64-linux-gnu/libm-2.31.so7f275c5dd000-7f275c5de000 rw-p 00000000 00:00 0 7ffd22d2f000-7ffd22d50000 r-w 00000000 00:00 0 [stack]7ffd22db0000-7ffd22db4000 r--p 00000000 00:00 0 [vdso]ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0 [vvar]7ffd22db4000-7ffd22db6000 r-xp 00000000 00:00 0 [vsyscall]

Meaning of "maps" file (Multiple True False (ETH))

• W.r.t the figure given below, mark the given statements as True or False.

• **Select all the correct statements, w.r.t. Copy on Write**

- a. (25%) Fork() used COW technique to improve performance of new process creation.
- b. (25%) If either parent or child modifies a COW-page, then a copy of the page is made and page table entry is updated
- c. (25%) COW helps us save memory
- d. (25%) Vfork() assumes that there will be no write, but rather exec()
- e. (-50%) use of COW during fork() is useless if exec() is called by the child
- f. (-50%) use of COW during fork() is useless if child called exit()

COW T/F (Multiple choice)

• **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. (25%) Firefox has likely been running for a large amount of time
- b. (25%) Apache web-server has not been doing much work
- c. (25%) The bash shell is mostly busy doing work within a particular locality
- d. (25%) All of the processes here exhibit some good locality of reference

meaning of ps -eo min_flt, etc. (Multiple choice)

• **which of the following, do you think, are valid concerns for making the kernel pageable?**

- a. (25%) The kernel's own page tables should not be pageable
- b. (25%) The page fault handler should not be pageable
- c. (25%) The kernel must have some dedicated frames for it's own work
- d. (25%) The disk driver and disk interrupt handler should not be pageable
- e. (-50%) No data structure of kernel should be pageable
- f. (-50%) No part of kernel code should be pageable.

pageable kernel (Multiple choice)

• **Order the following events, related to page fault handling, in correct order**

- MMU detects that a page table entry is marked "invalid"
- Page fault interrupt is generated
- Page fault handler in kernel starts executing
- Page fault handler detects that it's a page fault and not illegal memory access
- Empty frame is found
- Disk read is issued
- Page faulting process is made to wait in a queue
- Other processes scheduled by scheduler
- Disk Interrupt occurs
- Disk interrupt handler runs
- Page table of page faulted process is updated
- Page faulted process is moved to ready-queue

Demand paging : order (Ordering)

• **Assuming a 8- KB page size, what is the page numbers for the address {address} reference in decimal : (give answer also in decimal)**

Answer: {address} / (8*1024)

page number calculation (Calculated)

• 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
- first fit 500 KB
-> 600 KB
- best fit 115 KB
-> 125 KB
- best fit 500 KB
-> 600 KB
- worst fit 115 KB
-> 750 KB
- worst fit 500 KB
-> 635 KB
- -> 200 KB
- -> 350 KB

first/worst/best fit (Matching)

• 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 : {#1} FIFO replacement and 3 page frames is : {#2}

- {1:SHORTANSWER:%100%4}
- {1:SHORTANSWER:%100%4}

#page faultts (Embedded answers (Cloze))

• Page sizes are a power of 2 because

- a. (100%) Certain bits are reserved for offset in logical address. Hence page size = 2^(no.of offset bits)
- b. (0%) operating system calculations happen using power of 2
- c. (0%) MMU only understands numbers that are power of 2
- d. (0%) Power of 2 calculations are highly efficient
- e. (0%) Certain bits are reserved for offset in logical address. Hence page size = 2^(32 - no.of offset bits)

Page sizes are a power of 2 because (Multiple choice / One answer only)

• Compare paging with demand paging and select the correct statements.

- a. (14.28571%) Demand paging requires additional hardware support, compared to paging.
- b. (14.28571%) Both demand paging and paging support shared memory pages.
- c. (14.28571%) With demand paging, it's possible to have user programs bigger than physical memory.
- d. (14.28571%) Demand paging always increases effective memory access time.
- e. (14.28571%) Paging requires some hardware support in CPU
- f. (14.28571%) Calculations of number of bits for page number and offset are same in paging and demand paging.
- g. (14.28571%) The meaning of valid-invalid bit in page table is different in paging and demand-paging.
- h. (-33.33333%) With paging, it's possible to have user programs bigger than physical memory.
- i. (-33.33333%) Paging requires NO hardware support in CPU
- j. (-33.33333%) TLB hit ration has zero impact in effective memory access time in demand paging.

paging vs demand paging (Multiple choice)

• Shared memory is possible with which of the following memory management schemes ?

- a. (33.33333%) paging
- b. (33.33333%) segmentation
- c. (-100%) continuous memory management
- d. (33.33333%) demand paging

shared memory - possible (Multiple choice)

• **Arrange the following in the correct order of execution (w.r.t. 'init')**

- userinit() is called-> 1
- 'initcode' struct proc is created-> 2
- 'initcode' process is marked RUNNABLE-> 3
- mpmain() calls scheduler()-> 4
- scheduler() schedules initcode() process-> 5
- initcode() returns in forkret()-> 6
- initcode() returns from trapret()-> 7
- initcode() calls exec("/init", ...) -> 8

init related execution sequence (Matching)

• **Map the virtual address to physical address in xv6**

- KERNBASE-> 0
- KERNLINK-> 0x100000
- 80108000-> 0x108000
- 0xFE000000-> 0xFE000000
- -> 0x80000000

kernel memory mappings (Matching)

• **The approximate number of page frames created by kinit1 is**

- a. (100%) 3000
- b. (0%) 1000
- c. (0%) 2000
- d. (0%) 4000
- e. (0%) 10
- f. (0%) 4
- g. (0%) 16

#kinit1's pages (Multiple choice / One answer only)

• **Select all the correct statements about initcode**

- a. (25%) code of 'initcode' is loaded along with the kernel during booting
- b. (25%) the size of 'initcode' is 2c
- c. (25%) The data and stack of initcode is mapped to one single page in userinit()
- d. (25%) initcode essentially calls exec("/init",...)
- e. (-33.33333%) initcode is the 'init' process
- f. (-33.33333%) code of initcode is loaded in memory by the kernel during userinit()
- g. (-33.33333%) code of initcode is loaded at virtual address 0

correct about initcode (Multiple choice)

• **Which of the following is DONE by allocproc() ?**

- a. (20%) Select an UNUSED struct proc for use
- b. (20%) allocate PID to the process
- c. (20%) allocate kernel stack for the process
- d. (20%) setup the trapframe and context pointers appropriately
- e. (20%) ensure that the process starts in forkret()
- f. (-33.33333%) ensure that the process starts in trapret()
- g. (-33.33333%) setup kernel memory mappings for the process
- h. (-33.33333%) setup the contents of the trapframe of the process properly

not done by allocproc() (Multiple choice)

• **Which of the following is done by mappages()?**

- a. (33.33333%) create page table mappings for the range given by "va" and "va + size"
- b. (33.33333%) allocate page table if required
- c. (33.33333%) create page table mappings to the range given by "pa" and "pa + size"
- d. (-50%) allocate page directory if required
- e. (-50%) allocate page frame if required

not done by mappages (Multiple choice)

• **What does seginit() do?**

- a. (100%) Adds two additional entries to GDT corresponding to Code and Data segments, but to be used in privilege level 3
- b. (0%) Adds two additional entries to GDT corresponding to Code and Data segments, but to be used in privilege level 0
- c. (0%) Nothing significant, just repetition of earlier GDT setup but with 2-level paging setup done
- d. (0%) Nothing significant, just repetition of earlier GDT setup but with free frames list created now
- e. (0%) Nothing significant, just repetition of earlier GDT setup but with kernel page table allocated now

seginit() does? (Multiple choice / One answer only)

• **Select the statement that most correctly describes what setupkvm() does**

- a. (100%) creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global array
- b. (0%) creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global array and makes kpgdir point to it
- c. (0%) creates a 2-level page table for the use of the kernel, as specified in gtdesc
- d. (0%) creates a 1-level page table for the use by the kernel, as specified in kmap[] global array

setupkvm()'s job (Multiple choice / One answer only)

• **What does userinit() do ?**

- a. (100%) sets up the 'initcode' process to start execution in forkret()
- b. (0%) sets up the 'init' process to start execution in forkret()
- c. (0%) sets up the 'initcode' process to start execution in trapret()
- d. (0%) sets up the 'initcode' process to start execution in forkret ()
- e. (0%) initializes the users
- f. (0%) initializes the process 'init' and starts executing it

userinit() does? (Multiple choice / One answer only)

• **The variable 'end' used as argument to kinit1 has the value**

- a. (100%) 801154a8
- b. (0%) 80110000
- c. (0%) 80000000
- d. (0%) 81000000
- e. (0%) 80102da0
- f. (0%) 8010a48c

value of end (Multiple choice / One answer only)

• **Does exec() code around clearptau() lead to wastage of one page frame?**

- a. (100%) yes
- b. (0%) no

wastage in exec? (Multiple choice / One answer only)

• **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. (33.33333%) exec() loads from ELF file and the address of first instruction to be executed is given by 'entry'
- b. (33.33333%) In userinit() the function inituvm() has mapped the code of 'initcode' to be starting at virtual address 0
- c. (33.33333%) the initcode is created using objcopy, which discards all relocation information and symbols (like entry)
- d. (-33.33333%) the 'entry' in initcode is anyways 0
- e. (-33.33333%) the code of 'initcode' is loaded at physical address 0
- f. (-33.33333%) elf.entry is anyways 0, so both statements mean the same

why different eip settings? (Multiple choice)

• **Why is there a call to kinit2? Why is it not merged with kinit1?**

- a. (100%) knit2 refers to virtual addresses beyond 4MB, which are not mapped before kalloc() is called
- b. (0%) Because there is a limit on the values that the arguments to kinit1() can take.
- c. (0%) 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. (0%) call to seginit() makes it possible to actually use PHYSTOP in argument to kinit2()

why kinit2()? (Multiple choice / One answer only)
