Started on	Wednesday, 19 April 2023, 6:30 PM
State	Finished
Completed on	Wednesday, 19 April 2023, 8:52 PM
Time taken	2 hours 21 mins
Overdue	21 mins 46 secs
Grade	**23.26** out of 30.00 (**77.54**%)

Question **1**

Partially correct

Mark 0.88 out of 1.00

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

Select one or more:

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

Your answer is partially correct.

You have correctly selected 7.
The correct answers are: Recovery is possible due to redundancy in file system data structures, A recovery program, most typically, builds the file system data structure and checks for inconsistencies, It is possible to lose data as part of recovery, They may take very long time to execute, They can make changes to the on-disk file system, Recovery programs recalculate most of the metadata summaries (e.g. free inode count), Recovery programs are needed only if the file system has a delayed-write policy., Even with a write-through policy, it is possible to need a recovery program.

Question **2**

Partially correct

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

Tell compiler not to reorder memory access beyond this line ✔

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

Disable interrupts to avoid deadlocks ✔

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

Ensure that no printing happens on other processors ✔

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

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

```
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)

✔

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

Disable interrupts to avoid another process's pointer being returned

✔

Your answer is partially correct.

You have correctly selected 7.

The correct answer is: void
acquire(struct spinlock *lk)
{
...
__sync_synchronize();

→ Tell compiler not to reorder memory access beyond this line, 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
acquire(struct spinlock *lk)
{
 pushcli();
→ Disable interrupts to avoid deadlocks, void
panic(char *s)
{
...
 panicked = 1; → Ensure that no printing happens on other processors, void

```
sleep(void *chan, struct spinlock *lk)
{
  ...
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }
```
→ Avoid a self-deadlock,

```
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),

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

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

Question **3**

Partially correct

Mark 0.80 out of 1.00

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

Select one or more:

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

Your answer is partially correct.

You have correctly selected 4.
The correct answers are: many-one model can be implemented even if there are no kernel threads, all three models, that is many-one, one-one, many-many , require a user level thread library, one-one model increases kernel's scheduling load, many-one model gives no speedup on multicore processors, A process blocks in many-one model even if a single thread makes a blocking system call

Question **4**

Correct

Mark 2.00 out of 2.00

Select all the correct statements about synchronization primitives.

Select one or more:

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

Your answer is correct.

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

Question **5**

Correct

Mark 1.00 out of 1.00

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

void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs); | Traverse ebp chain to get sequence of instructions followed in functions calls ✔

void
yield(void)
{
...
release(&ptable.lock);
} | Release the lock held by some another process ✔

void
panic(char *s)
{
...
  panicked = 1; | Ensure that no printing happens on other processors ✔

Your answer is correct.

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

Question **6**

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

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

Question 7

Correct

Mark 1.00 out of 1.00

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

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

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

Question 8

Correct

Mark 1.00 out of 1.00

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

Select all the correct statements about log structured file systems.

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

Your answer is correct.

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

Question **9**

Complete

Mark 1.50 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.

This implementation assumes there is multiple user support for xv6

```
a)  void chown(char* pathname, int owner, int group){
       if(priviledge(currentOwner)  > priviledge(owner)){
             //change owner in inode of file/folder pointed by pathname
            }
       }
```

b) sys_calls to check currentOwner() and filePriviledges(char* filepath)

c) all file related sys_calls like open and read should check for currently logged in user and priviledges for the file

d)

```
struct inode {
uint dev; // Device number
uint inum; // Inode number
int ref; // Reference count
struct sleeplock lock; // protects everything below here
int valid; // inode has been read from disk?
short type; // copy of disk inode
short major;
short minor;
short nlink;
uint size;
uint addrs[NDIRECT+1];
int owner;  --------------> owner
int group;  ---------------> group
};
```

e)

f) no changes in makefile for adding system a sys_call

g)

Comment:

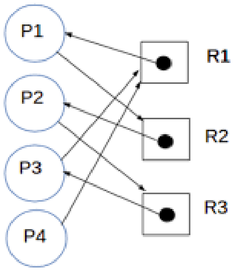
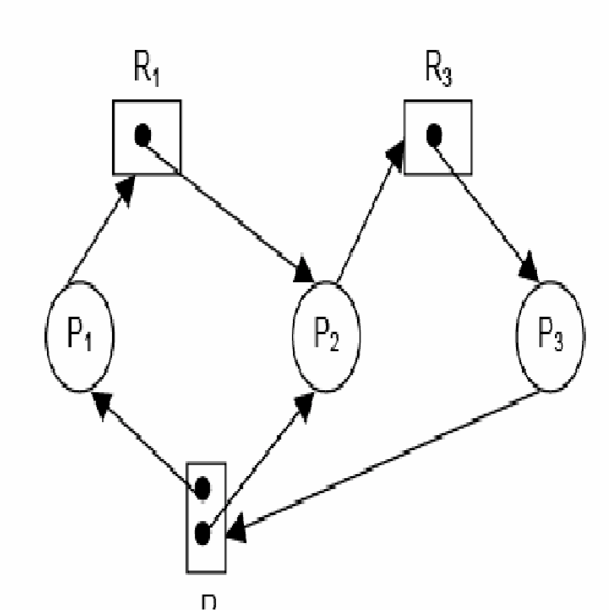
Question **10**

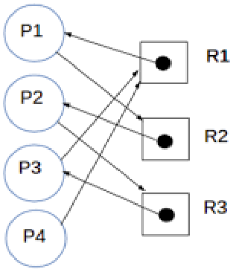
Correct

Mark 1.00 out of 1.00

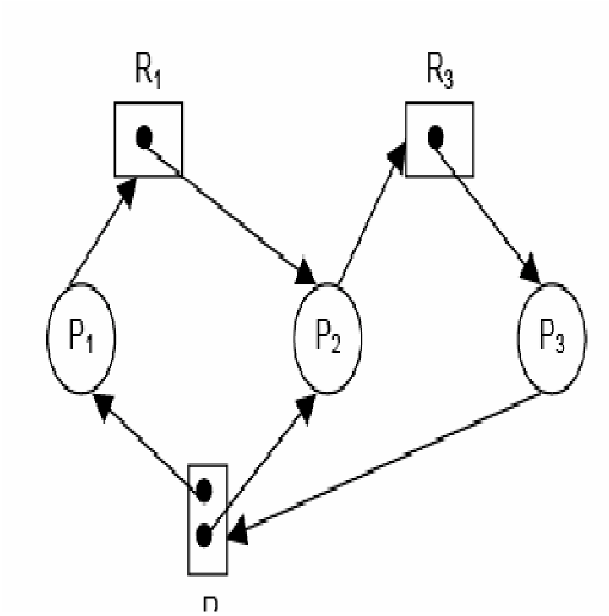
For each of the resource allocation diagram shown,

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

Yes	No		
✔	✖		✔
✔	✖		✔



: Yes



: Yes

Question **11**

Incorrect

Mark 0.00 out of 1.00

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

Answer: 6400030 ✖

The correct answer is: 4800030.00

Question **12**

Correct

Mark 1.00 out of 1.00

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

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: 11011110 ✔

The correct answer is: 11011110

Question **13**

Partially correct

Mark 0.86 out of 1.00

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

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

The volume manager stores additional metadata on the physical disk partitions: True
A logical volume can be extended in size but upto the size of volume group: True
A logical volume may span across multiple physical partitions: 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 volumes: 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 **14**

Incorrect

Mark 0.00 out of 1.00

Assuming a 8- KB page size, what is the page numbers for the address 26583 reference in decimal :

(give answer also in decimal)

Answer: 13 ✖

The correct answer is: 3

Question **15**

Correct

Mark 2.00 out of 2.00

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

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

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

Question **16**

Partially correct

Mark 1.43 out of 2.00

Compare paging with demand paging and select the correct statements.

Select one or more:

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

Your answer is partially correct.

You have correctly selected 5.
The correct answers are: Demand paging requires additional hardware support, compared to paging., Both demand paging and paging support shared memory pages., With demand paging, it's possible to have user programs bigger than physical memory., Demand paging always increases effective memory access time., Paging requires some hardware support in CPU, Calculations of number of bits for page number and offset are same in paging and demand paging., The meaning of valid-invalid bit in page table is different in paging and demand-paging.

Question **17**

Partially correct

Mark 0.80 out of 1.00

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

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

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

Question **18**

Correct

Mark 1.00 out of 1.00

Match the code with it's functionality

S = 5 Wait(S) Critical Section Signal(S)	Counting semaphore ✔
S1 = 0; S2 = 0; P2: Statement1; Signal(S2); P1: Wait(S2); Statemetn2; Signal(S1); P3: Wait(S1); Statement S3;	Execution order P2, P1, P3 ✔
S = 0 P1: Statement1; Signal(S) P2: Wait(S) Statment2;	Execution order P1, then P2 ✔
S = 1 Wait(S) Critical Section Signal(S);	Binary Semaphore for mutual exclusion ✔

Your answer is correct.

The correct answer is: S = 5
Wait(S)
Critical Section
Signal(S) → Counting semaphore, S1 = 0; S2 = 0;
P2:
Statement1;
Signal(S2);

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

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

P2:

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

Question **19**
Correct
Mark 1.00 out of 1.00

Map the technique with it's feature/problem

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

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

Question **20**
Partially correct
Mark 0.25 out of 1.00

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

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

Your answer is partially correct.

You have correctly selected 1.
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 **21**

Complete

Mark 1.00 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.

```
a) buddyAllocate(int requiredSize, string allocatedBitmap, int i, int j ){
        string sizeRequired =  find the least bigger power of 2 than requiredSize in a form of bitmap
        if(sizeRequried ==  allocatedBitmap[i:j]){
                return location in bitmap which match with substring allocatedBitmap[i:j];
        buddyAllocate(requiredSize, 0, sizeof(allocatedBitmap)/2));
        buddyAllocate(requiredSize, sizeof(allocatedBitmap)/2+1, sizeof(allocatedBitmap));
 b)
c)  filealloc()           // remove lines ---->
// for(f = ftable.file; f < ftable.file + NFILE; f++){
// if(f->ref == 0){
// f->ref = 1;
// release(&ftable.lock);
// return f;
// }
// }
and add ----------->   f = getCache();

d) struct memoryBitmap{
        spinlock sl;
        uint allocatedBitmap;
        int sizeMappedToEachBitmap; // like 32 bytes in one of previous questions
e)
f) no changes
g) nope
```

Comment:
checked

Question 22

Correct

Mark 1.00 out of 1.00

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

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

Process	Burst Time
P1	2
P2	6
P3	2
P4	3

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

Answer: 5

P2 waits for 2 units

P3 waits for 2+6 units

P4 waits for 2 + 6 +2 units of time

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

Average waiting time = 20/4 = 5

The correct answer is: 5

Question **23**

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

blocks holding a spinlock ✔

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

too much spinning, bounded wait not guaranteed ✔

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

deadlock ✔

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

not holding lock after unblock

Your answer is correct.

The correct answer is:

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

→ blocks holding a spinlock,

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

→ too much spinning, bounded wait not guaranteed,

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

→ deadlock,

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

→ not holding lock after unblock

◄ Random Quiz - 6 (xv6 file system)

Jump to...

Homework questions: Basics of MM, xv6 booting ►

Started on	Friday, 31 March 2023, 6:18 PM
State	Finished
Completed on	Friday, 31 March 2023, 7:00 PM
Time taken	41 mins 48 secs
Grade	**7.73** out of 15.00 (**51.54**%)

Question **1**

Partially correct

Mark 0.75 out of 1.00

Select all the actions taken by iget()

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

Your answer is partially correct.

You have selected too many options.
The correct answers are: Returns an inode with given dev+inode-number from cache, if it exists in cache, Returns the inode with reference count incremented, Returns a free-inode , with dev+inode-number set, if not found in cache

Question **2**

Partially correct

Mark 0.60 out of 1.00

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

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

Your answer is partially correct.

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

Grade details: 3 / 5 = 60%

Here are the scores for each item in this response:

1. 0 / 1 = 0%
2. 1 / 1 = 100%
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 **3**

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. file system recovery recovers all the lost data
- [x] b. xv6 has a log structured file system✔
- [x] c. ext4 is a log structured file system✖ it's a journaled file system, not log structured
- [ ] d. ext2 is by default a log structured file system
- [x] 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 **4**

Correct

Mark 1.00 out of 1.00

Select all the actions taken by ilock()

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

Your answer is correct.

The correct answers are: Read the inode from disk, if needed, Copy the on-disk inode into in-memory inode, if neeed, Take the sleeplock on the inode, always, Mark the in-memory inode as valid, if needed

Question **5**

Incorrect

Mark 0.00 out of 1.00

Maximum size of a file on xv6 in **bytes** is

(just write a numeric answer)

Answer: 16920576 ✖

The correct answer is: 71680

Question **6**

Partially correct

Mark 1.71 out of 2.00

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

True	False		
◉ ✔	○ ✖	The code supports IDE, and not SATA/SCSI	✔
○ ✖	◉ ✔	only direct blocks are supported	✔
◉ ✔	○ ✖	the superblock does not contain number of free blocks	✔
◉ ✔	○ ✖	only 2 disks are handled by default	✔
○ ✔	◉ ✖	disk driver handles only one buffer at a time	✖
◉ ✔	○ ✖	log is kept on the same device as the file system	✔
○ ✖	◉ ✔	device files are not supported	✔

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

Question **7**

Partially correct

Mark 0.50 out of 1.00

Compare XV6 and EXT2 file systems.

Select True/False for each point.

True	False		
○ ☑	◉ ✖	xv6 contains journal, ext2 does not	✖
○ ✖	◉ ☑	Ext2 contains superblock but xv6 does not.	✔
◉ ✖	○ ☑	In both ext2 and xv6, the superblock gives location of first inode block	✖
○ ✖	◉ ☑	xv6 contains inode bitmap, but ext2 does not	✔
◉ ✖	○ ☑	Both xv6 and ext2 contain magic number	✖
◉ ☑	○ ✖	Ext2 contains group descriptors but xv6 does not	✔

xv6 contains journal, ext2 does not: True
Ext2 contains superblock but xv6 does not.: False
In both ext2 and xv6, the superblock gives location of first inode block: False
xv6 contains inode bitmap, but ext2 does not: False
Both xv6 and ext2 contain magic number: False
Ext2 contains group descriptors but xv6 does not: True

Question **8**

Correct

Mark 1.00 out of 1.00

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

- ○ a. iread
- ○ b. readi
- ○ c. iget
- ○ d. sys_read
- ◉ e. ilock ✔

Your answer is correct.

The correct answer is: ilock

Question **9**

Correct

Mark 2.00 out of 2.00

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

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

The reference count (refcnt) in struct buf is = number of processes accessing the buffer: True
Lock on a buffer is acquired in bget, and released in brelse: True
The "next" pointer chain gives the buffers in LRU order: 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
A buffer can have both B_VALID and B_DIRTY flags set: False
B_VALID means the buffer is empty and can be reused: False
The buffers are maintained in LRU order, in the function brelse: True

Question **10**

Partially correct

Mark 0.17 out of 1.00

Suppose an application on xv6 does the following:

```
int main() {
  char arr[128];
  int fd = open("README, O_RDONLY);
  read(fd, arrr, 100);
}
```

Assume that the code works.

Which of the following things are true about xv6 kernel code, w.r.t. the above C program.

True	False			
○	○	The loop in readi() will always read a different block using bread()	✖	
◉	○	value of fd will be 3	✔	
○	○	The "memmove(dst, bp->data + off%BSIZE, m);" in readi() will copy the data from the disk to the kernel buffers	✖	
○	○	The process will be made to sleep only once	✖	
○	○	The ONLY function that gets called on return devsw[ip->major].read(ip, dst, n); is consoleread	✖	
◉	○	The data is transferred from disk to kernel buffers first, and then address of arr is maped to the kernel buffers	✖	No. data is copied into arr.

The loop in readi() will always read a different block using bread(): False
value of fd will be 3: True
The "memmove(dst, bp->data + off%BSIZE, m);" in readi() will copy the data from the disk to the kernel buffers: False
The process will be made to sleep only once: True
The 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 maped to the kernel buffers: False

Question **11**

Not answered

Marked out of 1.00

The lines

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

in namex() function

mean

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

Your answer is incorrect.

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

Question **12**

Not answered

Marked out of 1.00

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

sys_chdir()	Choose...
skipelem	Choose...
ialloc	Choose...
namei	Choose...
bmap	Choose...
filestat()	Choose...
dirlookup	Choose...
balloc	Choose...
ideintr	Choose...
bread	Choose...
stati	Choose...
commit	Choose...

Your answer is incorrect.

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

Question **13**

Not answered

Marked out of 1.00

Match function with it's functionality

dirlookup	Choose...
namex	Choose...
nameiparent	Choose...
dirlink	Choose...

Your answer is incorrect.

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

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

Jump to...

(Random Quiz - 7) Pre-Endsem Quiz ►

Started on	Thursday, 9 March 2023, 6:20 PM
State	Finished
Completed on	Thursday, 9 March 2023, 7:23 PM
Time taken	1 hour 3 mins
Overdue	7 mins 42 secs
Grade	**5.46** out of 10.00 (**54.56**%)

Question **1**

Partially correct

Mark 0.15 out of 1.00

Consider the following command and it's output:

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

Following code in bootmain()

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

and following selected lines from Makefile

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

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

Also read the code of bootmain() in xv6 kernel.

Select the options that describe the meaning of these lines and their correlation.

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

Your answer is partially correct.

You have correctly selected 2.

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

Question **2**

Partially correct

Mark 0.20 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	0x10000 to 0x7c00	✖
jmp *%eax in entry.S	Immaterial as the stack is not used here	✖
cli in bootasm.S	Immaterial as the stack is not used here	✔
readseg((uchar*)elf, 4096, 0); in bootmain.c	The 4KB area in kernel image, loaded in memory, named as 'stack'	✖
call bootmain in bootasm.S	0x10000 to 0x7c00	✖

Your answer is partially correct.

You have correctly selected 1.
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', cli
in bootasm.S → Immaterial as the stack is not used here, readseg((uchar*)elf, 4096, 0);
in bootmain.c → 0x7c00 to 0, call bootmain
in bootasm.S → 0x7c00 to 0

Question **3**

Incorrect

Mark 0.00 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 ljmp instruction does a divide by 8 on the first argument
- b. While indexing the GDT using CS, the value in CS is always divided by 8 ✖
- c. The value 8 is stored in code segment
- d. The code segment is 16 bit and only lower 13 bits are used for segment number
- e. The code segment is 16 bit and only upper 13 bits are used for segment number

Your answer is incorrect.

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

Question **4**

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

Your answer is correct.

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

Question **5**

Partially correct

Mark 0.57 out of 1.00

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

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

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

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

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

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

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

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

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

Your answer is partially correct.

You have correctly selected 4.
The correct answers are: The kernel file gets loaded at the Physical address 0x10000 in memory., The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it., The elf->entry is set by the linker in the kernel file and it's 8010000c, The readseg finally invokes the disk I/O code using assembly instructions, The stosb() is used here, to fill in some space in memory with zeroes, The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded, The kernel file has only two program headers

Question **6**

Partially correct

Mark 0.50 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 setting up of the most essential memory management infrastructure needs assembly code
- [ ] b. The code in assembly is required for transition to protected mode, from real mode; but calling convention was applicable all the time
- [x] 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 partially correct.

You have correctly selected 1.
The correct answers are: The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C, The setting up of the most essential memory management infrastructure needs assembly code

Question **7**

Partially correct

Mark 0.50 out of 1.00

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

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

Your answer is partially correct.

You have correctly selected 3.
The correct answer is: 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, 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, bootmain() → gdt setup with 3 entries, at start32 symbol of bootasm.S

Question **8**

Partially correct

Mark 0.91 out of 1.00

W.r.t. Memory management in xv6,

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

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

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

Question **9**

Partially correct

Mark 0.88 out of 1.00

Select the correct statements about interrupt handling in xv6 code

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

Your answer is partially correct.

You have correctly selected 7.

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 **10**

Partially correct

Mark 0.75 out of 1.00

xv6.img: bootblock kernel

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

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

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

Your answer is partially correct.

You have correctly selected 3.
The correct answers are: xv6.img is the virtual processor used by the qemu emulator, The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk., The size of the kernel file is nearly 5 MB, The size of xv6.img is exactly = (size of bootblock) + (size of kernel)

◄ Random Quiz 4 : Scheduling, signals, segmentation, paging, compilation, process-state

Jump to...

Random Quiz - 6 (xv6 file system) ►

Started on	Thursday, 16 February 2023, 9:00 PM
State	Finished
Completed on	Thursday, 16 February 2023, 9:54 PM
Time taken	53 mins 39 secs
Grade	**12.88** out of 15.00 (**85.86**%)

Question **1**

Correct

Mark 1.00 out of 1.00

Mark whether the concept is related to scheduling or not.

Yes	No		
◉ ✔	○ ✖	ready-queue	✔
◉ ✔	○ ✖	context-switch	✔
◉ ✔	○ ✖	timer interrupt	✔
◉ ✔	○ ✖	runnable process	✔
○ ✖	◉ ✔	file-table	✔

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

Question **2**

Partially correct

Mark 0.67 out of 1.00

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

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

Your answer is partially correct.

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

Question **3**

Partially correct

Mark 0.50 out of 1.00

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

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

Your answer is partially correct.

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

Question **4**

Partially correct

Mark 0.80 out of 1.00

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

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

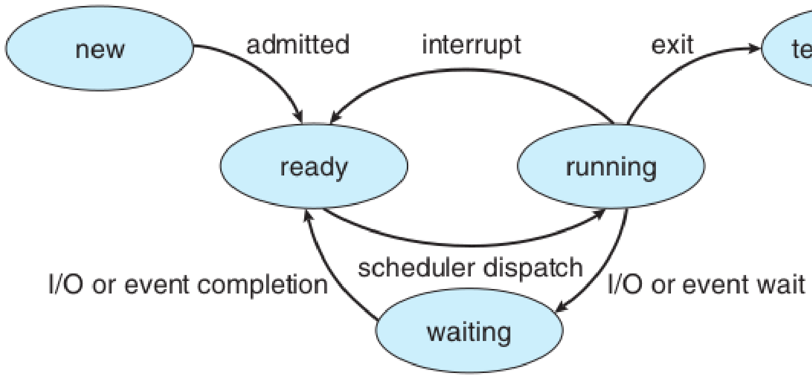


Figure 3.2 Diagram of process state.

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

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

Question **5**

Correct

Mark 1.00 out of 1.00

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

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

Your answer is correct.

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

Question **6**

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

Question **7**

Partially correct

Mark 1.56 out of 2.00

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

- [ ] a. A process waiting for any condition is woken up by another process only
- [ ] b. A waiting process starts running after the wait is over
- [x] c. A process that is running is not on the ready queue ✔
- [x] d. A process waiting for I/O completion is typically woken up by the particular interrupt handler code ✔
- [ ] e. It is not maintained in the data structures by kernel, it is only for conceptual understanding of programmers
- [x] f. A running process may terminate, or go to wait or become ready again ✔
- [ ] g. Typically, it's represented as a number in the PCB
- [x] h. A process can self-terminate only when it's running ✔
- [x] i. Processes in the ready queue are in the ready state ✔
- [ ] j. A process changes from running to ready state on a timer interrupt or any I/O wait
- [x] k. A process in ready state is ready to be scheduled ✔
- [ ] l. A process in ready state is ready to receive interrupts
- [ ] m. Changing from running state to waiting state results in "giving up the CPU"
- [x] 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 **8**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about zombie processes

Select one or more:

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

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

Question **9**

Partially correct

Mark 0.50 out of 1.00

Select all the correct statements about signals

Select one or more:

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

Your answer is partially correct.

You have selected too many options.

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

Question **10**

Correct

Mark 1.00 out of 1.00

Map each signal with it's meaning

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

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

Question **11**

Correct

Mark 1.00 out of 1.00

Match the names of PCB structures with kernel

xv6	struct proc ✔
linux	struct task_struct ✔

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

Question **12**

Partially correct

Mark 0.86 out of 1.00

Mark True/False

Statements about scheduling and scheduling algorithms

True	False			
◉ ✔	○ ✖	Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.	✔	
○ ✖	◉ ✔	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-premptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.	✔	
◉ ✔	○ ✖	Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.	✔	
◉ ✔	○ ✖	xv6 code does not care about Processor Affinity	✔	
◉ ✔	○ ✖	Response time will be quite poor on non-interruptible kernels	✔	
○ ✔	◉ ✖	Processor Affinity refers to memory accesses of a process being stored on cache of that processor	✖	

Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.: True
On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread: False
A scheduling algorithm is non-premptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.: True
Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.: True
xv6 code does not care about Processor Affinity: 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

Question **13**

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 **14**

Correct

Mark 1.00 out of 1.00

Which of the following statements is false ?

Select one:

- a. Real time systems generally use non preemptive CPU scheduling. ✔
- b. A process scheduling algorithm is preemptive if the CPU can be forcibly removed from a process.
- c. Time sharing systems generally use preemptive CPU scheduling.
- d. Response time is more predictable in preemptive systems than in non preemptive systems.

Your answer is correct.

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

◄ Random Quiz - 3 (processes, memory management, event driven kernel), compilation-linking-loading, ipc-pipes

Jump to...

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

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	**14.19** out of 20.00 (**70.93**%)

Question **1**

Complete

Mark 0.50 out of 1.00

Select the sequence of events that are NOT possible, assuming an interruptible kernel code

Select one or more:

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

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

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

P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

The correct answers are: P1 running
P1 makes sytem call and blocks
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again,
P1 running
P1 makes sytem call
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

Question **2**

Complete

Mark 0.00 out of 1.00

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

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

Program 1

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

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

Program 2

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

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

Select all the correct statements about the programs

Select one or more:

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

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 **3**

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 **4**

Complete

Mark 4.75 out of 5.00

Following code claims to implement the command

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

Fill in the blanks to make the code work.

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

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

2

][2];

pipe(

pfd[0]

);
pid1 =

fork()

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

[0]

);
close(

1

);
dup(

pfd[0][1]

);
execl("/bin/ls", "/bin/ls", "

-l

", NULL);
}
pipe(

pfd[1]

);

pid2

= fork();
if(pid2 == 0) {
close(

pfd[0][1]

;
close(0);
dup(

pfd[0][0]

);
    close(pfd[1]

[0]

);
    close(

1

);
    dup(

pfd[1][1]

);
    execl("/usr/bin/head", "/usr/bin/head", "

-3

", NULL);
  } else {
    close(pfd

[1][1]

);
    close(

0

);
    dup(

pfd[1][0]

);
    close(pfd

[0][0]

);
    execl("/usr/bin/tail", "/usr/bin/tail", "

-1

", NULL);
  }
}

Question **5**

Complete

Mark 1.00 out of 1.00

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

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

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

Question **6**

Complete

Mark 0.00 out of 1.00

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

Select one or more:

- [x] a. both named and unnamed pipes require some kind of agreed protocol to be effectively used among multiple processes
- [x] b. named pipes are more efficient than ordinary pipes
- [x] c. named pipe exists even if the processes using it do exit()
- [x] d. ordinary pipe can only be used between related processes
- [x] e. named pipes can be used between multiple processes but ordinary pipes can not be used
- [x] f. a named pipe exists as a file on the file system
- [x] 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 **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 will have n and n+1th steps such that n+1th step can not follow n'th step.

Select one or more:

- [ ] a. P1 running
  P1 makes system call
  system call returns
  P1 running
  timer interrupt
  Scheduler running
  P2 running
- [ ] 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 and blocks
  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

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

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

Question **8**
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 langauge code to machine code
- [ ] b. Process the # directives in a C program
- [x] c. Check the program for logical errors
- [ ] d. Check the program for syntactical errors
- [x] e. Suggest alternative pieces of code that can be written
- [ ] f. Invoke the linker to link the function calls with their code, extern globals with their declaration

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

Question **9**

Complete

Mark 1.40 out of 2.00

Match the elements of C program to their place in memory

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

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

Question **10**

Complete

Mark 0.67 out of 1.00

Select all the correct statements about zombie processes

Select one or more:

- [x] a. If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent
- [x] b. A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it
- [ ] c. A process becomes zombie when it's parent finishes
- [x] d. A zombie process occupies space in OS data structures
- [x] 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
- [x] g. Zombie processes are harmless even if OS is up for long time
- [x] h. init() typically keeps calling wait() for zombie processes to get cleaned up

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

Question **11**

Complete

Mark 0.29 out of 1.00

Order the events that occur on a timer interrupt:

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

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

Question **12**

Complete

Mark 1.00 out of 1.00

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

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

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

fd1	/tmp/1
fd2	/tmp/2
fd4	/tmp/2
0	/tmp/2
2	stderr
1	/tmp/3
fd3	closed

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

Question **13**

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)

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

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

Question **14**

Complete

Mark 0.75 out of 1.00

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

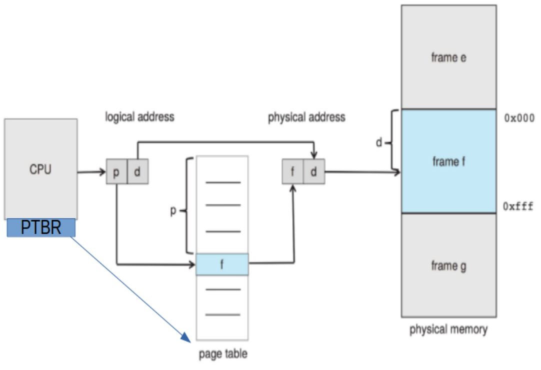


Figure 9.8 Paging hardware.

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

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

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

Question **15**

Complete

Mark 1.00 out of 1.00

A process blocks itself means

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

The correct answer is: The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

Started on	Monday, 16 January 2023, 9:00 PM
State	Finished
Completed on	Monday, 16 January 2023, 10:05 PM
Time taken	1 hour 4 mins
Grade	**11.52** out of 15.00 (**76.78**%)

Question **1**

Correct

Mark 1.00 out of 1.00

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

a. yes

b. no ✔ wrong

The correct answer is: no

Question **2**

Partially correct

Mark 0.67 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. Bootloaders allow selection of OS to boot from ✔

b. LILO is a bootloader ✔

c. The bootloader loads the BIOS

d. Modern Bootloaders often allow configuring the way an OS boots

e. Bootloader must be one sector in length

Your answer is partially correct.

You have correctly selected 2.

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

Question **3**

Correct

Mark 2.00 out of 2.00

What will this program do?

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

- a. run ls twice ✔
- b. run ls twice and print hello twice
- c. run ls once
- d. one process will run ls, another will print hello
- 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 **4**

Correct

Mark 1.00 out of 1.00

Compare multiprogramming with multitasking

- a. A multitasking system is not necessarily multiprogramming
- b. A multiprogramming system is not necessarily multitasking ✔

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

Question **5**

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 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 **6**

Correct

Mark 1.00 out of 1.00

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

- a. in real mode the segment is multiplied by 16, in protected mode segment is used as index in GDT
- b. processor starts in real mode
- c. in real mode the addressable memory is more than in protected mode ✔
- d. in real mode general purpose registers are 16 bit, in protected mode they are 32 bit
- 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 **7**

Correct

Mark 1.00 out of 1.00

When you turn your computer ON, you are often shown an option like "Press F9 for boot options". What does this mean?

- a. The choice of booting slowly or fast
- b. The choice of the boot loader (e.g. GRUB or Windows-Loader)
- c. The BIOS allows us to choose the boot device, the device from which the boot loader will be loaded ✔
- 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 **8**

Partially correct

Mark 0.83 out of 1.00

Select the correct statements about hard and soft links

Select one or more:

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

Your answer is partially correct.

You have correctly selected 5.
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 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. "ls" runs on current directory✔
- c. Execution fails as the call to execl() in exec2 fails
- d. Program prints hello
- 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 **10**

Partially correct

Mark 0.60 out of 1.00

Select all the correct statements about two modes of CPU operation

Select one or more:

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

Your answer is partially correct.

You have correctly selected 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 **11**

Partially correct

Mark 0.67 out of 1.00

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

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

Your answer is partially correct.

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

Question **12**

Partially correct

Mark 0.75 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 mount point must be a directory
- [ ] c. Mounting deletes all data at the mount-point
- [ ] d. Mounting makes all disk partitions available as one name space
- [ ] e. On Linuxes mounting can be done only while booting the OS
- [x] f. It's possible to mount a partition on one computer, into namespace of another computer. ✔
- [x] g. Mounting is attaching a disk-partition with a filesystem on it, into another file system name-space ✔
- [ ] h. The existing name-space at the mount-point is no longer visible after mounting
- [x] i. The mount point can be a file as well ✖
- [x] j. In operating systems with a pluggable kernel module for file systems, the code for mounting a particular file system is provided by the module of that file system. ✔

Your answer is partially correct.

You have correctly selected 3.

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.

◄ Random Quiz - 1 (Pre-Requisite Quiz)

Jump to...

Random Quiz - 3 (processes, memory management, event driven kernel), compilation-linking-loading, ipc-pipes ►

Started on	Friday, 17 March 2023, 2:29 PM
State	Finished
Completed on	Friday, 17 March 2023, 4:33 PM
Time taken	2 hours 3 mins
Grade	**6.75** out of 10.00 (**67.46**%)

Question **1**

Correct

Mark 0.50 out of 0.50

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

a. sleeplock is preferable because it is used in interrupt context and spinlock can not be used in interrupt context

b. struct buf is used as a general purpose cache by kernel and cache operations take lot of time, so better to use sleeplock rather than spinlock

c. It could be a spinlock, but xv6 has chosen sleeplock for purpose of demonstrating how to use a sleeplock.

d. struct buf is used for disk I/o which takes lot of time, so sleeping/blocking is preferred to spinning/busy-wait for the desired buf. ✔

e. struct buf is used for disk I/o which takes lot of time, so sleeping/blocking is the only option available.

Your answer is correct.

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

Question **2**

Partially correct

Mark 0.91 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	Statement		Feedback
◉ ✔	○ ✘	The spinlock lk->lk is held when the process comes out of sleep()	✔	
○ ✘	◉ ✔	sleep() is called holding a spinlock. This could be avoided by releasing the lock before calling sleep() and acquiring it again after call to sleep()	✔	

○ ✘	◉ ✔	acquire(&lk->lk);
		while (lk->locked) {
		sleep(lk, &lk->lk);
		}
		could also be written as
		acquire(&lk->lk);
		if (lk->locked) {
		sleep(lk, &lk->lk);
		}
			✔	loop is required because other process might have obtained the lock before this process returns from sleep().

◉ ✔	○ ✘	All processes waiting for the sleeplock will have a race for aquiring lk->lk spinlock, because all are woken up	✔	wakeup() wakes up all processes, and they "thunder" to take the spinlock.

True	False			
○ ✖	◉ ✔	the 'spinlock lk' protects 'locked' variable, but not the 'name' nor the 'pid'	✔	
◉ ✔	○ ✖	sleep() is the function which blocks a process.	✔	
◉ ✔	○ ✖	the 'spinlock lk' is needed in a sleeplock, because access to the sleeplock for locking/unlocking itself creates a critical section	✔	
◉ ✔	○ ✖	Sleeplock() will ensure that either the process gets the lock or the process gets blocked.	✔	
○ ✖	◉ ✔	Wakeup() will wakeup the first process waiting for the lock	✔	Wakeup() will wakeup all processes waiting for the lock
○ ✖	◉ ✔	A process has acquired the sleeplock when it comes out of sleep()	✔	
○ ✔	◉ ✖	The process which called acquiresleep() and then got blocked, is woken up by the timer interrupt	✖	it's woken up by another process which called releasesleep() and then wakeup()

The spinlock lk->lk is held when the process comes out of sleep(): 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

```
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
```

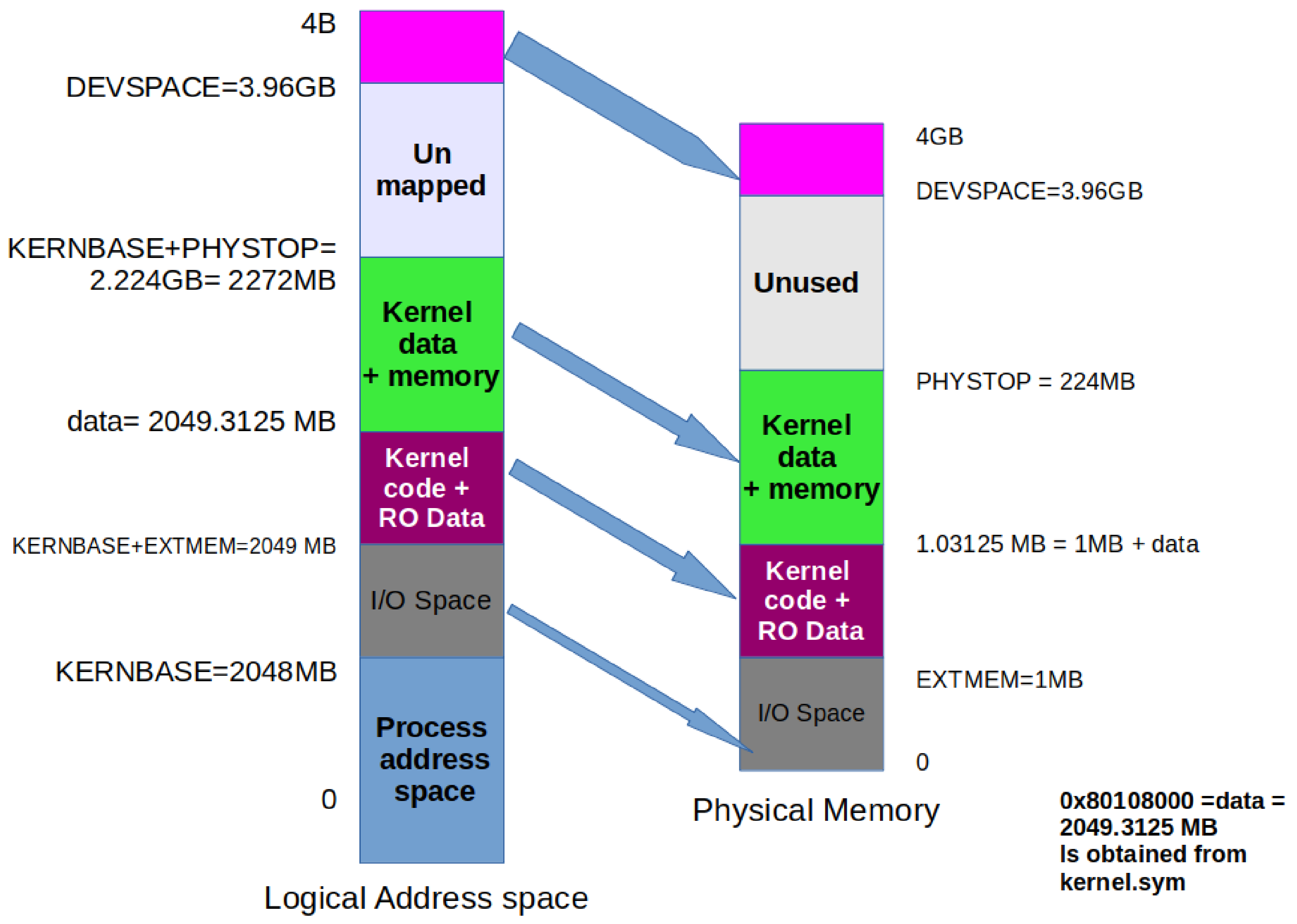
All processes waiting for the sleeplock will have a race for aquiring lk->lk spinlock, because all are woken up: True
the 'spinlock lk' protects 'locked' variable, but not the 'name' nor the 'pid': False
sleep() is the function which blocks a process.: True
the 'spinlock lk' is needed in a sleeplock, because access to the sleeplock for locking/unlocking itself creates a critical section: True
Sleeplock() will ensure that either the process gets the lock or the process gets blocked.: True
Wakeup() will wakeup the first process waiting for the lock: False
A process has acquired the sleeplock when it comes out of sleep(): False
The process which called acquiresleep() and then got blocked, is woken up by the timer interrupt: True

Question **3**

Partially correct

Mark 0.36 out of 0.50

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



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

True	False		
☑	✗	The kernel file, after compilation, has maximum virtual address up to "data" as shown in the diagram, which is equal to "end" variable	✔

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

Question **4**

Incorrect

Mark 0.00 out of 0.25

Select the odd one out

- a. Kernel stack of new process to Process stack of new process
- b. Process stack of running process to kernel stack of running process ✗
- c. Kernel stack of scheduler to kernel stack of new process
- d. Kernel stack of running process to kernel stack of scheduler
- e. Kernel stack of new process to kernel stack of scheduler

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

Question **5**

Partially correct

Mark 0.80 out of 1.00

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

True	False			
◉ ✔	○ ✖	swtch() is written in assembly language, because it violates calling convention, by changing the stack itself.	✔	
◉ ✔	○ ✖	push in swtch() happens on old stack, while pop happens from new stack	✔	
○ ✖	◉ ✔	swtch() called from scheduler() changes the stack from the process's kernel stack to the scheduler's kernel stack.	✔	it does reverse!
○ ✖	◉ ✔	switch stores the old context on new stack, and restores new context from old stack.	✔	old goes on old, new comes from new stack
◉ ✔	○ ✖	p->context used in scheduler()->swtch() was **Generally** set when the process was interrupted earlier, and came via sched()->swtch()	✔	That's the only place when p->context is changed.
◉ ✔	○ ✖	swtch() changes the context from "old" to "new"	✔	yeah, that's the definition
◉ ✖	○ ✔	sched() is the only place when p->context is set	✖	no. allocproc() also sets it.
○ ✖	◉ ✔	swtch() is written in assembly language because it violates the calling convention by pushing parameters on the stack on its own.	✔	any function can push anything on stack, but remove it properly, that will not affect calling convention.
◉ ✖	○ ✔	movl %esp, (%eax) means, *(c->scheduler) = contents of esp When swtch() is called from scheduler()	✖	No. it means c->scheduler = contents of esp.
◉ ✔	○ ✖	swtch() is called only from sched() or scheduler()	✔	

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

Question **6**

Partially correct

Mark 0.44 out of 0.50

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

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

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

Question **7**
Partially correct
Mark 0.15 out of 0.25

Match function with it's meaning

ideintr	disk interrupt handler, transfer data from conroller to buffer, wake up processes waiting for this buffer, start I/O for next buffer	✔
idewait	Wait for disc controller to be ready	✔
ideinit	Initialize the disc controller	✔
iderw	tell disc controller to complete I/O for all pending requests	✖
idestart	Issue a disk read/write for a buffer, block the issuing process	✖

Your answer is partially correct.

You have correctly selected 3.
The correct answer is: ideintr → disk interrupt handler, transfer data from conroller to buffer, wake up processes waiting for this buffer, start I/O for next buffer, idewait → Wait for disc controller to be ready, ideinit → Initialize the disc controller, 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

Question **8**
Partially correct
Mark 0.25 out of 0.50

when is each of the following stacks allocated?

kernel stack of process	during fork() in allocproc()	✔
kernel stack for scheduler, on first processor	in entry.S	✔
user stack of process	during exec()	✖
kernel stack for the scheduler, on other processors	in entry.S	✖

Your answer is partially correct.

You have correctly selected 2.
The correct answer is: kernel stack of process → during fork() in allocproc(), kernel stack for scheduler, on first processor → in entry.S, user stack of process → during fork() in copyuvm(), kernel stack for the scheduler, on other processors → in main()->startothers()

Question **9**
Correct
Mark 0.25 out of 0.25

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

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

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

Question **10**

Correct

Mark 0.25 out of 0.25

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

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

The correct answer is: 801154a8

Question **11**

Partially correct

Mark 0.23 out of 0.50

Which of the following is DONE by allocproc() ?

- a. setup the trapframe and context pointers appropriately
- b. setup the contents of the trapframe of the process properly
- c. allocate PID to the process
- d. ensure that the process starts in forkret()
- e. allocate kernel stack for the process
- f. ensure that the process starts in trapret()
- 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()

Question **12**

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

Your answer is incorrect.

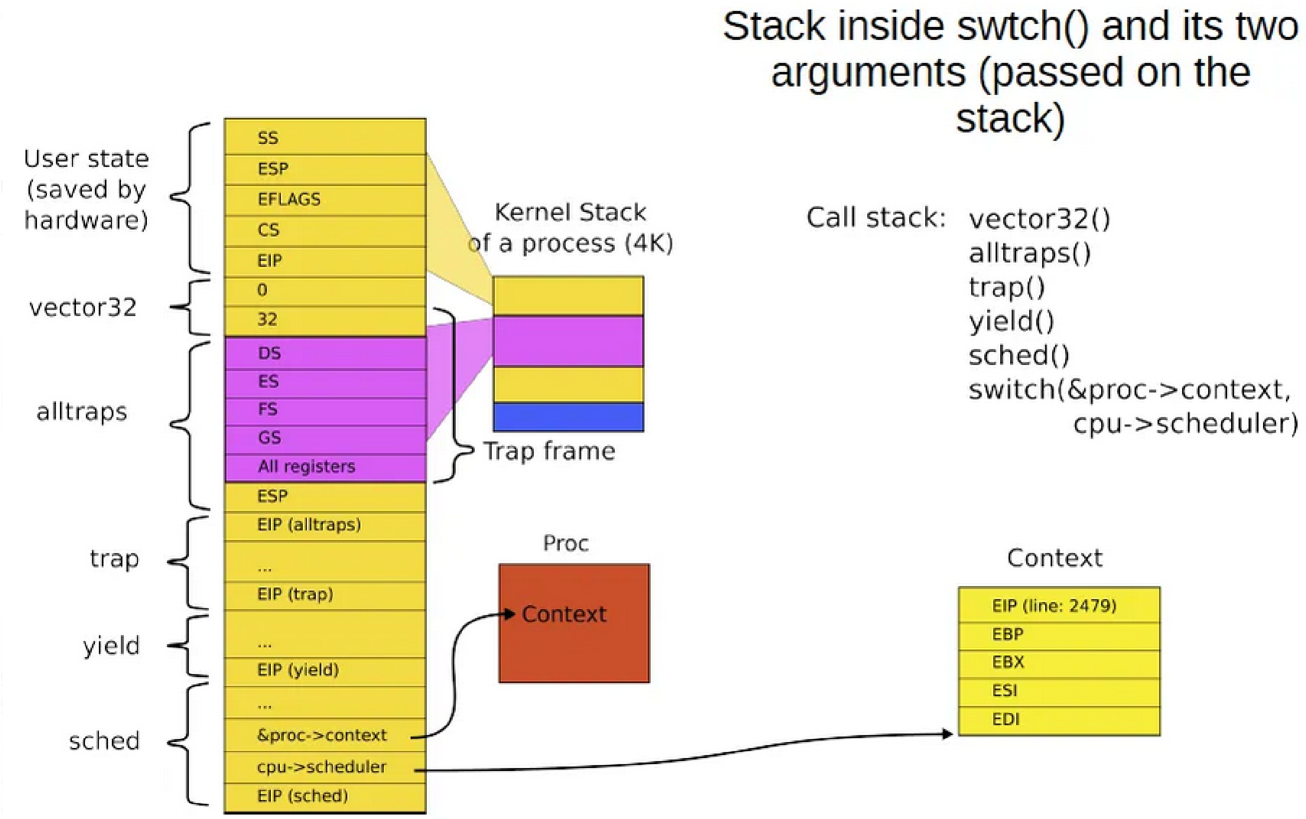
The correct answer is: It disables interrupts. It is required because the interrupt handlers of kernel are not yet installed.

Question 13

Partially correct

Mark 0.42 out of 0.50

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



True	False	Statement		Feedback
✗	☑ (selected)	This is a diagram of swtch() called from scheduler()	✔	No. diagram of swtch() called from sched()
☑ (selected)	✗	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.	✔	
✗	☑ (selected)	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
☑ (selected)	✗	The diagram is correct	✔	
☑ (selected)	✗	The blue shaded part in "kernel stack of a process(4k)" refers to remaining part of stack (not used yet)	✔	
✗ (selected)	☑	The "context" yellow coloured box, pointed to by cpu->scheduler is on the kernel stack of the scheduler.	✗	

This is a diagram of swtch() called from scheduler(): False
The "ESP" (second entry from top) is stack pointer of user-stack of process, while the "ESP" (first entry below pink region) is the trapframe pointer on kernel stack of process.: True
The diagram is wrong because it shows the user stack and kernel stack together (continuous), but in practice they are separate: False
The diagram is correct: True

The blue shaded part in "kernel stack of a process(4k)" refers to remaining part of stack (not used yet): True
The "context" yellow coloured box, pointed to by cpu->scheduler is on the kernel stack of the scheduler.: False

Question **14**

Partially correct

Mark 0.66 out of 0.75

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

True	False			
◉ ✔	○ ✘	One sequence of function calls which takes and releases the ptable.lock is this: iderw->sleep, acquire(ptable.lock)->sched->swtch()->scheduler()->swtch()->yield(), release(ptable.lock)	✔	One process slept, another was scheduled and it came out of timer interrupt.
◉ ✔	○ ✘	It is taken by one process but released by another process, running on same processor	✔	
◉ ✘	○ ✔	A process can sleep on ptable.lock if it can't aquire it.	✘	It's a spinlock!
◉ ✔	○ ✘	ptable.lock protects the proc[] array and all struct proc in the array	✔	
○ ✘	◉ ✔	ptable.lock is acquired but never released	✔	how is that possible?
○ ✘	◉ ✔	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.
○ ✘	◉ ✔	ptable.lock can be held by different processes on different processors at the same time	✔	No lock can be held like this!
◉ ✔	○ ✘	the rule of "never block holding a spinlock" does not apply to ptable.lock in xv6	✔	sched() is called only if you hold ptable.lock

One sequence of function calls which takes and releases the ptable.lock is this:
iderw->sleep, acquire(ptable.lock)->sched->swtch()->scheduler()->swtch()->yield(), release(ptable.lock): True
It is taken by one process but released by another process, running on same processor: True
A process can sleep on ptable.lock if it can't aquire it.: False
ptable.lock protects the proc[] array and all struct proc in the array: True
ptable.lock is acquired but never released: False
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

Question **15**

Incorrect

Mark 0.00 out of 0.25

Why is there a call to kinit2? Why is it not merged with knit1?

- a. call to seginit() makes it possible to actually use PHYSTOP in argument to kinit2()
- b. Because there is a limit on the values that the argumets to knit1() can take.
- c. When kinit1() is called there is a need for few page frames, but later knit2() is called to serve need of more page frames ✖
- d. knit2 refers to virtual addresses beyond 4MB, which are not mapped before kvalloc() is called

The correct answer is: knit2 refers to virtual addresses beyond 4MB, which are not mapped before kvalloc() is called

Question **16**

Partially correct

Mark 0.54 out of 0.75

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

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

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

Question **17**

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 segment
- c. The ss:esp pair
- d. The region of memory where the kernel remembers all the function calls made
- e. The "stack" variable declared in "stack.S" in xv6
- f. The stack variable used in the program being discussed
- g. The region of memory allocated by kernel for storing the parameters of functions

Your answer is incorrect.

The correct answer is: The ss:esp pair

Question **18**

Incorrect

Mark 0.00 out of 0.25

Which of the following call sequence is impossible in xv6?

- a. Process 1: timer interrupt -> trap() -> yield() -> sched() -> switch() -> scheduler()-> Process 2 runs -> write -> sys_write() -> trap()-> ...
- 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: 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

Your answer is incorrect.

The correct answer is: Process 1: timer interrupt -> trap() -> yield() -> sched() -> switch() -> scheduler()-> Process 2 runs -> write -> sys_write() -> trap()-> ...

Question **19**

Correct

Mark 0.50 out of 0.50

Consider the following command and it's output:

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

Following code in bootmain()

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

and following selected lines from Makefile

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


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

Also read the code of bootmain() in xv6 kernel.

Select the options that describe the meaning of these lines and their correlation.

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

Your answer is correct.

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

Question **20**

Correct

Mark 0.50 out of 0.50

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

True	False			
☑	✖	the kmem.lock is used by kfree() and kalloc() only.	✔	

✖	☑	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.

✖	☑	free page list is a singly circular linked list.	✔	it's singly linked NULL terminated list.
☑	✖	kmem.use_lock is set to 1 after free page list is created, so that kmem.lock is taken before accessing kmem.freelist.	✔	
☑	✖	The pointers that link the pages together are in the first 4 bytes of the pages themselves	✔	

☑	✖	if(kmem.use_lock)
		    acquire(&kmem.lock);
		is not done when called from kinit1() because there is no need to take the lock when kinit1() is running because interrupts are disabled and only one processor is running
			✔	

the kmem.lock is used by kfree() and kalloc() only.: 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
free page list is a singly circular linked list.: 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 pointers that link the pages together are in the first 4 bytes of the pages themselves: True
if(kmem.use_lock)
    acquire(&kmem.lock);
is not done when called from kinit1() because there is no need to take the lock when kinit1() is running because interrupts are disabled and only one processor is running: True

◄ Quiz-1(24 Feb 2023)

Jump to...

Pre-requisite Quiz (old) - use it for practice ►

Started on	Friday, 24 February 2023, 2:44 PM
State	Finished
Completed on	Friday, 24 February 2023, 4:26 PM
Time taken	1 hour 42 mins
Grade	**9.53** out of 10.00 (**95.27**%)

Question **1**

Correct

Mark 0.50 out of 0.50

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

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

Program 1

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

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

Program 2

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

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

Select all the correct statements about the programs

Select one or more:

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

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 **2**

Partially correct

Mark 0.36 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			
○ ✗	◉ ✓	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.
○ ✓	◉ ✗	On Linux, the process was sent a SIGSEGV signal and the default handler for the signal is "Term", so the process is terminated.	✗	
○ ✗	◉ ✓	The term "core" refers to the core code of the kernel.	✓	core means memory, all memory for the process.
◉ ✗	○ ✓	The illegal memory access was detected by the kernel and the process was punished by kernel.	✗	"detection" is done by CPU, not kernel.
◉ ✓	○ ✗	The image of the process is stored in a file called "core", if the ulimit allows so.	✓	see ulimit -a
◉ ✓	○ ✗	The core file can be analysed later using a debugger, to determine what went wrong.	✓	use gdb ./core ./executable-filename
◉ ✓	○ ✗	The process has definitely performed illegal memory access.	✓	

On Linux, the message is printed only because the memory management scheme is segmentation: False
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 term "core" refers to the core code of the kernel.: False
The illegal memory access was detected by the kernel and the process was punished by kernel.: False
The image of the process is stored in a file called "core", if the ulimit allows so.: True
The core file can be analysed later using a debugger, to determine what went wrong.: True
The process has definitely performed illegal memory access.: True

Question **3**

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

Your answer is correct.

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

Question **4**

Correct

Mark 0.50 out of 0.50

Doing a lookup on the pathname /a/b/b/c/d for opening the file "d" requires reading [6] ✔ no. of inodes. Assume that there are no hard/soft links on the path.

Write the answer as a number.

The correct answer is: 6

Question **5**

Correct

Mark 0.50 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"

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

Your answer is correct.

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

Question **6**

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)

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

Your answer is correct.

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

Question **7**

Partially correct

Mark 0.36 out of 0.50

Mark the statements as True/False w.r.t. the basic concepts of memory management.

True	False		
○ ✗	◉ ✓	The kernel refers to the page table for converting each virtual address to physical address.	✓
◉ ✓	○ ✗	When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.	✓
◉ ✗	○ ✓	The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.	✗
○ ✗	◉ ✓	The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.	✓
○ ✓	◉ ✗	The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.	✗
◉ ✓	○ ✗	The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.	✓
○ ✗	◉ ✓	When a process is executing, each virtual address is converted into physical address by the kernel directly.	✓

The kernel refers to the page table for converting each virtual address to physical address.: False
When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.: True
The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.: False
The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.: False
The compiler generates 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
When a process is executing, each virtual address is converted into physical address by the kernel directly.: False

Question **8**

Correct

Mark 0.50 out of 0.50

Map each signal with it's meaning

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

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

Question **9**

Partially correct

Mark 0.45 out of 0.50

Match the elements of C program to their place in memory

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

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

Question **10**

Correct

Mark 0.50 out of 0.50

Predict the output of the program given here.

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

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

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

For example::

`good`

`output`

should be written as `good output`

```
--

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

Answer: hi one one ✔

The correct answer is: hi one one

Question **11**

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)

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

Your answer is correct.

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

Question **12**

Partially correct

Mark 0.44 out of 0.50

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

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

Your answer is partially correct.

You have correctly selected 7.

The correct answer is: Global variables → Immediately after the text, Static variables → Immediately after the text, along with globals, #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), typedef → No memory allocated, as they are not variables, but only conceptual definition of a type, #define → No memory allocated, they are handled by pre-processor, Local variables → An offset with respect to stack pointer (esp), Text → starting with 0

Question **13**

Correct

Mark 0.50 out of 0.50

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

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

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

Question **14**

Partially correct

Mark 0.45 out of 0.50

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

Select No otherwise.

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

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

Question **15**

Correct

Mark 0.50 out of 0.50

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

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

Your answer is correct.

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

Question **16**

Correct

Mark 0.50 out of 0.50

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

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

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

Question **17**

Partially correct

Mark 0.48 out of 0.50

Following code claims to implement the command

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

Fill in the blanks to make the code work.

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

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

2

✔][2];

pipe(

pfd[0]

✔);
pid1 =

fork()

✔ ;
if(pid1 != 0) {
close(pfd[0]

[0]

✔);
close(

1

✔);
dup(

pfd[0][1]

✔);
execl("/bin/ls", "/bin/ls", "

-l

✔ ", NULL);
}
pipe(

pfd[1]

✔);

pid2

✔ = fork();
if(pid2 == 0) {
close(

pfd[0][1])

✖ ;
close(0);
dup(

pfd[0][0]

✔);
close(pfd[1]

[0]

✔);
close(

1

✔);
dup(

pfd[1][1]

✔);
execl("/usr/bin/head", "/usr/bin/head", "

-3

✔ ", NULL);
} else {
close(pfd

[1][1]

✔);
close(

0

✔);
dup(

pfd[1][0]

✔);
close(pfd

[0][0]

✔);
execl("/usr/bin/tail", "/usr/bin/tail", "

-1

✔ ", NULL);
}
}

Question **18**

Correct

Mark 0.50 out of 0.50

What is meant by formatting a disk/partition?

- a. storing all the necessary programs on the disk/partition
- b. erasing all data on the disk/partition
- c. writing zeroes on all sectors
- d. creating layout of empty directory tree/graph data structure ✔

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

Question **19**

Correct

Mark 0.50 out of 0.50

Which of the following instructions should be privileged?

Select one or more:

- [ ] a. Read the clock.
- [x] b. Access memory management unit of the processor✔
- [ ] c. Set value of a memory location
- [x] d. Turn off interrupts.✔
- [x] e. Set value of timer.✔
- [ ] f. Access a general purpose register
- [x] g. Access I/O device.✔
- [x] h. Switch from user to kernel mode. ✔ This instruction (like INT) is itself privileged - and that is why it not only changes the mode, but also ensures a jump to an ISR (kernel code)
- [x] i. Modify entries in device-status table✔

Your answer is correct.

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

Question **20**

Correct

Mark 0.50 out of 0.50

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

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

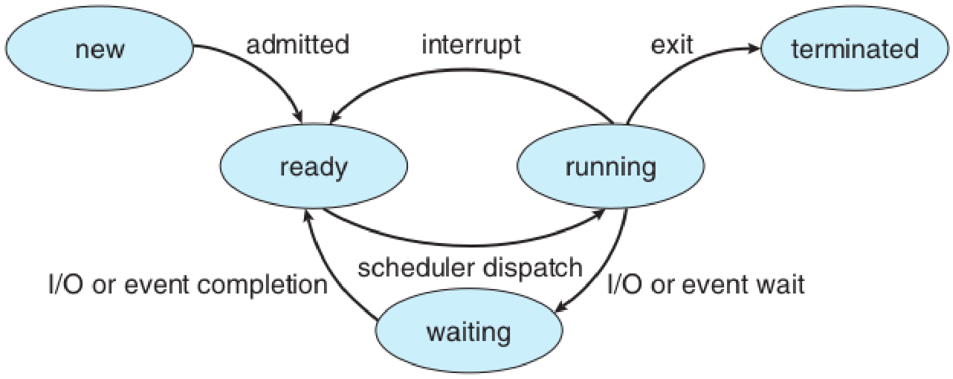


Figure 3.2 Diagram of process state.

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

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

◄ Quiz-1 Preparation questions

Jump to...

Quiz - 2 (17 March 2023) ►