Started on	Wednesday, 19 April 2023, 6:16 PM
State	Finished
Completed on	Wednesday, 19 April 2023, 8:03 PM
Time taken	1 hour 47 mins
Grade	**21.42** out of 30.00 (**71.39**%)

Question **1**

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

not holding lock after unblock ✔

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

deadlock ✔

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

→ not holding lock after unblock,

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

→ deadlock

Question **2**

Complete

Mark 0.25 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.

pseudo-code for a new function buddy_alloc()

```
struct buddy_block {
  struct buddy_block *next;
   int size;
};

struct buddy_list {
  struct buddy_block *head;
};

struct buddy {
  void *mem;
  int size;
  struct buddy_list *free_lists;
};
```

prototypes for the new functions:

void init_buddy(struct buddy *b, void *mem, int size, int min_block_size);

void *buddy_alloc(struct buddy *b, int size);

void buddy_free(struct buddy *b, void *ptr);

in syscall.h we need to add

int sys_buddy_init(void);

int sys_buddy_alloc(void);

int sys_buddy_free(void);

we also need to change kalloc() function

Comment:
checked

Question **3**

Partially correct

Mark 0.67 out of 1.00

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

Select one or more:

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

Your answer is partially correct.

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

Question **4**

Correct

Mark 2.00 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)
{
...
  getcallerpcs(&lk, lk->pcs);
```

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

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

Ensure that no printing happens on other processors ✔

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

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

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

Disable interrupts to avoid deadlocks ✔

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

Avoid a self-deadlock ✔

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

Release the lock held by some another process ✔

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

Tell compiler not to reorder memory access beyond this line

```
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)

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
panic(char *s)
{
...
  panicked = 1; → Ensure that no printing happens on other processors, struct proc*
myproc(void) {
...
  pushcli();
  c = mycpu();
  p = c->proc;
  popcli();
...
}

  → Disable interrupts to avoid another process's pointer being returned, void
acquire(struct spinlock *lk)
{
  pushcli();
  → Disable interrupts to avoid deadlocks, void
sleep(void *chan, struct spinlock *lk)
{
  ...
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  } → Avoid a self-deadlock, void
yield(void)
{
...

```
release(&ptable.lock);
}
```

→ Release the lock held by some another process, void

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

→ Tell compiler not to reorder memory access beyond this line, 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)

Question **5**

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.

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

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

Question **6**

Correct

Mark 1.00 out of 1.00

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

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

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

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 arguments to system call originally reside on process stack.	✔
☑	✗	The arguments are accessed in the kernel code using esp on the trapframe.	✔
✗	☑	Integer arguments are stored in eax, ebx, ecx, etc. registers	✔
☑	✗	String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer	✔
☑	✗	The functions like argint(), argstr() make the system call arguments available in the kernel.	✔
✗	☑	The arguments to system call are copied to kernel stack in trapasm.S	✔
✗	☑	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()	✔

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

Question **8**

Incorrect

Mark 0.00 out of 1.00

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

Select one or more:

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

Your answer is incorrect.

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

Question **9**

Partially correct

Mark 0.75 out of 1.00

Match the code with it's functionality

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

Binary Semaphore for mutual exclusion ✔

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

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

P3:
Wait(S1);
Statement S3;

Execution order P3, P2, P1 ✖

S = 0
P1:
Statement1;
Signal(S)

P2:
Wait(S)
Statment2;

Execution order P1, then P2 ✔

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

Counting semaphore ✔

Your answer is partially correct.

You have correctly selected 3.
The correct answer is: S = 1
Wait(S)
Critical Section
Signal(S); → Binary Semaphore for mutual exclusion, S1 = 0; S2 = 0;
P2:
Statement1;
Signal(S2);

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

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

P2:

Wait(S)

Statment2; → Execution order P1, then P2, S = 5

Wait(S)

Critical Section

Signal(S) → Counting semaphore

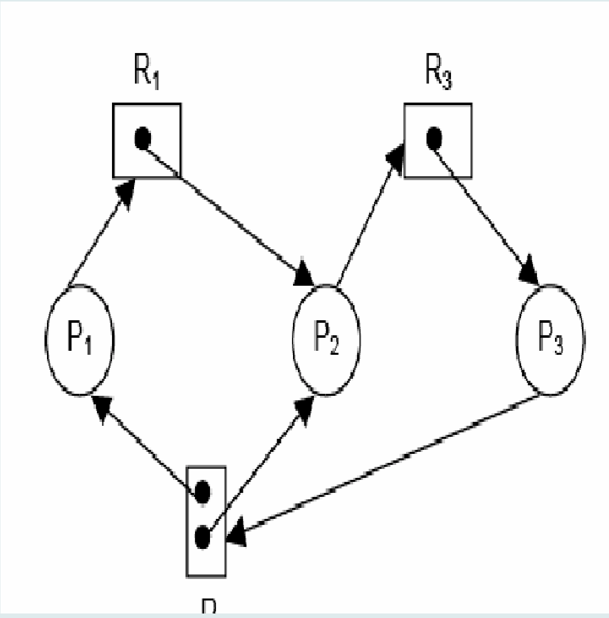
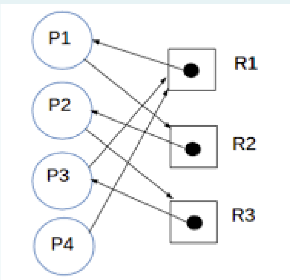
Question **10**

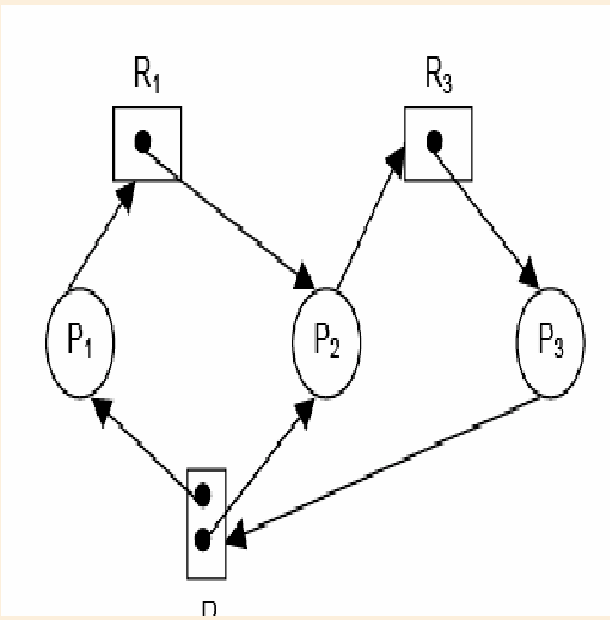
Correct

Mark 1.00 out of 1.00

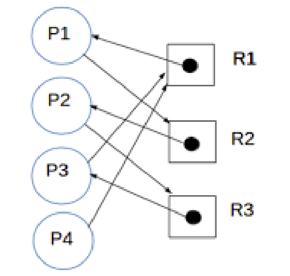
For each of the resource allocation diagram shown,

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

Yes	No		
◉ ✔	○ ✖		✔
◉ ✔	○ ✖		✔



: Yes



: Yes

Question **11**

Complete

Mark 0.50 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.

Pseudo-code

int chown(char *path, int owner_id);

in syscall.h

#define SYS_chown 22

in syscall.c

int sys_chown(void);

in user.h

int chown(char*, int);

in usys.S

SYSCALL(chown)

in sysproc.c

```
int sys_chown(void) {
  char *path;
  int owner_id;
if (argstr(0, &path) < 0 || argint(1, &owner_id) < 0) {
  return -1;
}
  return chown(path, owner_id);
}
```

Comment:

Question **12**

Correct

Mark 1.00 out of 1.00

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

Select all the correct statements about log structured file systems.

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

Your answer is correct.

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

Question **13**

Correct

Mark 1.00 out of 1.00

Map the technique with it's feature/problem

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

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

Question **14**

Correct

Mark 1.00 out of 1.00

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

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

Process	Burst Time
P1	2
P2	6
P3	2
P4	3

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

Answer: 5.00

P2 waits for 2 units

P3 waits for 2+6 units

P4 waits for 2 + 6 +2 units of time

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

Average waiting time = 20/4 = 5

The correct answer is: 5

Question **15**

Partially correct

Mark 1.75 out of 2.00

Select all the correct statements about synchronization primitives.

Select one or more:

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

Your answer is partially correct.

You have correctly selected 7.
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 **16**

Incorrect

Mark 0.00 out of 1.00

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

11010010

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

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

Answer: 1101001000010000 ✘

The correct answer is: 11011110

Question **17**

Correct

Mark 1.00 out of 1.00

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

Select one or more:

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

Your answer is correct.

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 **18**

Partially correct

Mark 1.50 out of 2.00

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

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

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 **19**

Correct

Mark 2.00 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. Calculations of number of bits for page number and offset are same in paging and demand paging. ✔
- [x] d. Demand paging requires additional hardware support, compared to paging. ✔
- [ ] e. Paging requires NO hardware support in CPU
- [x] f. Paging requires some hardware support in CPU ✔
- [x] g. Both demand paging and paging support shared memory pages. ✔
- [x] h. The meaning of valid-invalid bit in page table is different in paging and demand-paging. ✔
- [x] i. With demand paging, it's possible to have user programs bigger than physical memory. ✔
- [x] j. Demand paging always increases effective memory access time. ✔

Your answer is correct.

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

Question **20**

Correct

Mark 1.00 out of 1.00

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

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

A logical volume may span across multiple physical volumes: True
The volume manager stores additional metadata on the physical disk partitions: True
A physical partition should be initialized as a physial volume, before it can be used by volume manager.: True
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
A logical volume may span across multiple physical partitions: True
A logical volume can be extended in size but upto the size of volume group: True

Question **21**

Correct

Mark 1.00 out of 1.00

Assuming a 8- KB page size, what is the page numbers for the address 1014396 reference in decimal :

(give answer also in decimal)

Answer: 123 ✔

The correct answer is: 124

Question **22**

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

Release the lock held by some another process ✔

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

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

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

Ensure that no printing happens on other processors ✔

Your answer is correct.

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

Question **23**

Incorrect

Mark 0.00 out of 1.00

Given that the memory access time is 150 ns, probability of a page fault is 0.9 and page fault handling time is 8 ms,
The effective memory access time in nanoseconds is:

Answer: 8700000 ✖

The correct answer is: 7200015.00

◄ Random Quiz - 6 (xv6 file system)

Jump to...

Homework questions: Basics of MM, xv6 booting ►