CS 537: Introduction to Operating Systems (Summer 2017)

University of Wisconsin-Madison
Department of Computer Sciences

Final Exam

Friday, August 11th 2017

3 pm - 5:30 pm

There are **eighteen (18) total numbered pages** with **thirteen (13) questions**.

PLEASE READ ALL QUESTIONS CAREFULLY!

There are many easy questions and a few hard questions in this exam. You may want to use a **easiest-question-first scheduling policy**. This will help you to answer most questions on this exam without getting stuck on a single hard question.

Good luck with your exam!

Please write your **FULL NAME** and **UW ID** below.

NAME:_____

UW ID:_____

Grading Page

Question	**Points Scored**	**Maximum Points**
1		10
2		10
3		10
4		10
5		10
6		10
7		10
8		10
9		10
10		10
11		10
12		10
13		10
Total		**130**

1. Remember Virtualization?

The Process Control Block (PCB) for a process in xv6 is shown below.

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Kernel stack for this process
  enum procstate state;        // Process state
  volatile int pid;            // Process ID
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  ...                          // more fields...
};
```

a. Can you name any **TWO** fields in the PCB that is used for **virtualizing the CPU** among processes?

b. Can you name any **TWO** fields in the PCB that is used for **virtualizing the memory** among processes?

2. Semaphores

Consider the following implementation of lock and condition variable (CV) with semaphore.

Lock Implementation

```
typedef struct {
    sem_t s;
} lock;

void lock_init(lock* lk) {
    sem_init(&lk->s, 1);
}

void lock_acquire(lock* lk) {
    sem_wait(&lk->s);
}

void lock_release(lock* lk) {
    sem_post(&lk->s);
}
```

CV Implementation

```
typedef struct {
    sem_t s;
} cond;

void cond_init(cond* cv) {
    sem_init(&cv->s, 0);
}

void cond_wait(cond* cv, lock* lk) {
    lock_release(lk);
    sem_wait(&cv->s);
    lock_acquire(lk);
}

void cond_signal(cond* cv) {
    sem_post(&cv->s);
}
```

a. What is the difference between this lock implementation and a standard spinlock?

b. What is the difference between this CV implementation and a standard CV?

3. Deadlocks

Assume Thread#1 tries to `acquire(&v1->lock, &v2->lock)` and Thread#2 tries to `acquire(&v2->lock, &v1->lock)`. Four conditions need to hold for a deadlock to occur:

- Mutual exclusion
- Hold-and-wait
- No preemption
- Circular wait

a. The following implementations of `acquire(lock* L1, lock* L2)` try to prevent deadlock by breaking one of the conditions above. Write the corresponding condition (on the side of the three code snippets) that each solution is trying to break.

`if (L1 > L2) {`
`  pthread_mutex_lock(L1);`
`  pthread_mutex_lock(L2);`
`} else {`
`  pthread_mutex_lock(L2);`
`  pthread_mutex_lock(L1);`
`}`	

`pthread_mutex_lock(prevention);`
`pthread_mutex_lock(L1);`
`pthread_mutex_lock(L2);`
`pthread_mutex_unlock(prevention);`	

`top:`
`  pthread_mutex_lock(L1);`
`  if (pthread_mutex_trylock(L2) != 0) {`
`  pthread_mutex_unlock(L1);`
`  goto top;`
`  }`	

b. The three code snippets above prevent a deadlock by breaking one of the four required conditions for a deadlock to happen. How can you break the deadlock condition that is not attacked by the above three solutions?

c. For one of the solutions above, it is still possible that no thread can get both locks and proceed. Which of the above solutions has this issue? What is the problem here? How can you solve it?

4. Disks

We have a disk with the following parameters:

- Capacity = 1 TB (NOTE: 1TB = 1024 GB)
- RPM = 10000
- **Average** Seek = 9 ms
- Maximum Transfer Rate = 10^8 B/s

Assume there is **no cache or buffer**, and you will need to wait for a whole rotation if you want to access the same sector twice. We are always reading or writing a whole sector of size **512 bytes**.

a. How many sectors do we have?

b. How long would it take to serve 10 random reads on average?

c. How long would it take to serve 10 random updates on average? An update is a read followed by a write to the same sector, so the access pattern will be R0W0R1W1...R9W9 rather than R0R1...R9W0W1...W9.

d. How long would it take to serve 10 sequential reads? You may assume they are on the same track.

e. How long would it take to serve 10 sequential updates? Note that the access pattern is R0W0R1W1...R9W9, and all 10 sectors are on the same track.

5. Disk Scheduling

In this question, we will perform some calculations on a simplified disk. Assume the **maximum rotational delay** on this disk is **R**, the **time to seek between 2 adjacent tracks** is **S**, and the **transfer time** is so fast that we just consider it to be **free**. Also assume $\mathbf{S} \gg \mathbf{R}$. **All requests are already received.** The disk head can start from any position.

a. Assume the disk has only a **single track**, and a **FIFO** (First In First Out) scheduling policy. What is the (approximate) **worst case** execution time for **3 different requests** (i.e., the three requests are issued to **three different sectors** on the disk)?

b. Assume the disk has only a **single track**, and a **SATF** (Shortest Access Time First) scheduling policy. What is the **worst case** execution time for **3 different requests**?

c. Assume the disk has **3 tracks**, and a **FIFO** scheduling policy. What is the **worst case** execution time for **3 different requests**?

d. Assume the disk has **3 tracks**, and a **SATF** scheduling policy. What is the **worst case** execution time for **3 different requests**?

e. Assume the disk has **3 tracks**, and a **C-SCAN** scheduling policy. What is the **worst case** execution time for **3 different requests**? Recall that in C-SCAN the disk head is only allowed to scan in one direction.

6. RAID

This question is about the flow of I/Os in a RAID system. In RAIDs, some I/Os can happen in parallel, whereas some happen in sequence. To indicate two I/Os (to blocks 0 and 1, for example) in a flow can happen at the **same time**, we write "**(0 1)**"; to indicate they must happen in **sequence**, we write "**0, 1**".

These flows can be built into larger chains; for example, consider the sequence â(0 1), (2 3)â, which would indicate I/Os to blocks 0 and 1 could be issued in parallel, followed by I/Os to 2 and 3 in parallel.

We can also indicate the **read** and **write** operations in a flow with "**r**" and "**w**". Thus, "(r0 r1), (w2 w3)" is used to indicate we are reading blocks 0 and 1 in parallel, and, when that is finished, writing blocks 2 and 3 in parallel.

Assume we have the following **RAID-4**, with a **single parity disk**. You may also assume that we use **the most efficient method** for computing the parity.

```
  D0  D1  D2  D3  D4
   0   1   2   3  P0
   4   5   6   7  P1
   8   9  10  11  P2
  12  13  14  15  P3
.. (and so forth) ..
```

What is the flow for the following requests?

a. Read blocks 0, 1, 2, and 3

b. Read blocks 0, 4, 8, and 12

c. Read blocks 0, 5, 10, and 15

d. Write block 0

e. Write blocks 0, 1, 2, and 3

f. Write blocks 0, 4, 8, and 12

g. Write blocks 0, 5, 10, and 15

7. File System: Files & Directories

Imagine the following commands are run on an FFS-like file system that supports both **soft** and **hard** links. **Note:** the symbols â>>â are used to redirect stdout to ***append*** to the specified file; if the file does not exist, it will ***create*** this file; `echo` will print a newline after the string.

```
1  mkdir dir1;
2  ln -s dir1 dir2;
3  echo "hello" >> dir2/file1;
4  ln dir2/file1 dir1/file2;
5  echo "world" >> dir1/file1;
6  rm dir1/file2;
7  echo "hello world 1" >> dir2/file1;
8  echo "hello world 2" >> dir1/file2;
```

a. Draw the directory tree after executing line 8.

b. How many inodes are created after executing line 5? Which files or directories share the same inode?

c. How many inodes are created after executing line 8? Which files or directories share the same inode?

d. What will be printed if we run `cat dir1/file1` after executing line 8? If there is no such file, write N/A.

e. What will be printed if we run `cat dir2/file2` after executing line 8? If there is no such file, write N/A.

8. File System Implementation

- Assuming a very simple file system that supports 7 operations: **`mkdir()`** : creates a new directory; **`creat()`** : creates a new (empty) file; **`open()`** and **`close()`** : opens and closes a file, respectively; **`write()`** : appends a block to a file; **`link()`** : creates a hard link to a file; **`unlink()`** : unlinks a file (removing it if `linkcnt==0`).

- The state of the file system is indicated by the contents of two data structures, i.e., **inodes** and **data**.

- The inodes each have three fields: the first field indicates the type of file (`f` for a regular file, `d` for a directory); the second indicates which data block belongs to a file (here, files can only be empty, which have the address of the data block set to -1, or one block in size, which would have a non-negative address); the third shows the reference count for the file or directory.

  - For example, the following inode is a regular file, which is empty (address field set to -1), and has just one link in the file system: `[f a:-1 r:1]`. If the same file had a block allocated to it (say block 10), it would be shown as follows: `[f a:10 r:1]`. If someone then created a hard link to this inode, it would then become `[f a:10 r:2]`.
  - Note: the reference count of directory here is different from xv6. Here we need to account for the parent ".." as well, e.g. for an empty root directory, the reference count should be 2 because both "." and ".." refer to it.

- Data blocks can either retain user data or directory data.

  - An empty root directory looks like this, assuming the root inode is 0: `[(.,0) (..,0)]`. If we add a single file named `f` to the root directory, which has been allocated inode number 1, the root directory contents would then become: `[(.,0) (..,0) (f,1)]`.
  - If a data block contains user data, it is shown as just a single character within the block, e.g., `[D]`.
  - If it is empty and unallocated, just a pair of empty brackets (`[]`) are shown.

- Assume when allocating a new inode or data block, the first empty inode or block will be used.

Now the initial state of the file system is as follows:

```
inodes [d a:0 r:2] [] [] []; data [(.,0) (..,0)] [] [] []
```

Write down the file system states after each of the following operations. You should assume that the following operations are performed **sequentially**. i.e., The state of the file system builds on top of the previous operations.

a. `int fd1 = creat("/a");`

b. `write(fd1, "A", BLOCKSIZE);`

c. `mkdir("/b");`

d. `close(fd1); unlink("/a");`

e. `int fd2 = creat("/b/c"); write(fd2, "C", BLOCKSIZE);`

9. Berkeley Fast File System (FFS)

a. Draw a diagram of the file system layout of FFS.

b. Assume that all **leaf nodes** of the directory trees represent **regular files**. For instance the directory tree below has 3 leaf nodes (`b, c, d`), so it represents three files `/a/b, /a/c, /d`:

```
/
|-- a
|   |-- b
|   +-- c
|
+-- d
```

Assume that there are only 10 inodes in each group, use the table below to demonstrate how the inodes are spread across the groups given a specific directory tree as per the FFS policies. You should write the **name of the file(s)** in the space provided for the inodes. You need not worry about the data blocks of the files.

```
/
|-- a
|   |-- e
|   |-- f
|   |-- g
|   |-- h
|   +-- i
|
|-- b
|-- c
|   +-- j
|
+-- d
```

Groups	Inodes									
1										
2										
3										
4										
5										
6										
7										
8										

```
/
|-- a
|   +-- b
|       +-- c
|           +-- d
|                   |-- e
|                   +-- f
|
+-- g
```

Groups	Inodes									
1										
2										
3										
4										
5										
6										
7										
8										

c. Without the large-file exception, a single large file would be placed as follows in FFS.

```
Group Inodes      Data
0     /a--------  /aaaaaaaaa aaaaaaaaaa aaaaaaaaaa a---------
1     ----------  ---------- ---------- ---------- ----------
2     ----------  ---------- ---------- ---------- ----------
```

With the large-file exception, FFS spreads the file across groups, resulting in the following diagram

```
Group Inodes      Data
0     /a--------  /aaaaa---- ---------- ---------- ----------
1     ----------  aaaaa----- ---------- ---------- ----------
2     ----------  aaaaa----- ---------- ---------- ----------
3     ----------  aaaaa----- ---------- ---------- ----------
4     ----------  aaaaa----- ---------- ---------- ----------
5     ----------  aaaaa----- ---------- ---------- ----------
6     ----------  ---------- ---------- ---------- ----------
```

a. What is the **benefit** of splitting a large file across different groups in FSS?

b. What is the **issue** with splitting a large file across different groups in FSS?

10. Log-structured File Systems (LFS)

The log-structured file system buffers updates in memory (in segments) and then writes them out to disk sequentially. In this question, youâll be able to watch the traffic stream of writes to disk performed by LFS. Your task is to figure out what the inputs to the system (i.e., the **system call(s)** that took place) that caused these writes to happen.

- The system calls include `open(), close(), read(), write(), lseek()`.
- Assume the file system was basically empty (except the root directory).
- Assume that a single inode takes up an entire block (for simplicity). The LFS inode map is called the âimapâ below and of course is also updated as needed.
- The following file system operations are performed **sequentially**. i.e., The state of the file system builds on top of the previous operations.

a. Segment written starting at disk address 100, in a segment of size 4:

```
block 100: [("." 0), (".." 0), ("foo" 1)] // a data block
block 101: [size=1,ptr=100,type=d] // an inode
block 102: [size=0,ptr=-,type=r] // an inode
block 103: [imap: 0->101,1->102] // a piece of the imap
```

What file system operation(s) led to this segment write?

b. Segment written to disk address 104, in a segment of size 4:

```
block 104: [SOME DATA]
block 105: [SOME DATA]
block 106: [size=2,ptr=104,ptr=105,type=r]
block 107: [imap: 0->101,1->106]
```

What file system operation(s) led to this segment write?

c. Segment written to disk address 108, in a segment of size 4:

```
block 108: [SOME DATA]
block 109: [SOME DATA]
block 110: [size=2,ptr=108,ptr=109,type=r]
block 111: [imap: 0->101,1->110]
```

What file system operation(s) led to this segment write?

d. Segment written to disk address 112, in a segment of size 4:

```
block 112: [SOME DATA]
block 113: [SOME DATA]
block 114: [size=4,ptr=108,ptr=109,ptr=112,ptr=113,type=r]
block 115: [imap: 0->101,1->114]
```

What file system operation(s) led to this segment write?

e. After all of those writes, starting from block 100, how much garbage was left on the disk? (i.e., write down which blocks are filled with garbage, if any)

11. Journaling

- Assume we have a basic implementation of **data journaling** in our Very Simple File System (VSFS).
- Assume we are **creating a new empty file** in an existing directory. This operation must update **6 blocks**: the directory inode, two data blocks (directory & file), the file inode, the inode bitmap, and the data bitmap.
- Assume the directory inode and the file inode are in **different** on-disk blocks.
- Assume a **transaction begin block** and a **transaction end block** are written at the beginning and the end of each transaction respectively.
- Assume each block is written **synchronously** (i.e., a barrier is performed after every write and blocks are pushed out of the disk cache).

If the system crashes after the following number of blocks have been synchronously written to disk, what will happen when the **system reboots**? Fill **True (T)** or **False (F)** into the table below. (**Hint:** the first disk write is the transaction begin block written to the journal and the second disk write is one block (among the six blocks to be updated) written to the journal.)

# of disk writes	Transaction replayed during recovery?	File system in new state?
1		
4		
7		
8		
9		
10		
14		
15		

12. Flash-based SSDs

There following statements about flash-based SSDs may have a few mistakes. Can you **CIRCLE** the mistake and **CORRECT** them? Some statements may NOT contain any mistakes in which case you may just check it with a **TICK MARK**.

a. Before writing to a page within a flash, the nature of the device requires that you first erase just that page.

b. Time(Read a page) $<$ Time(Erase a block) $<$ Time(Program a page)

c. The **Flash Translation Layer (FTL)** takes read and write requests on logical blocks and turns them into low-level read, erase, and program commands on the underlying physical blocks and physical pages (that comprise the actual flash device).

d. In a log-structured FTL, sometimes a block will be filled with long-lived data that does not get over-written; in this case, garbage collection will never reclaim this block.

e. Rate(Random Reads in SSD) $<$ Rate(Random Writes in SSD) because writes just happen to the page cache (in physical memory) and NOT to the flash.

f. Rate(Random Writes in SSD) $\gg$ Rate(Random Writes in HDD) because SSD converts all random writes to sequential writes to the log-structured flash storage.

13. Threads and Processes (again!)

Assume that the code snippet below compiles successfully, all the APIs like **fork()**, **pthread_create**() do not fail, and the values in the **malloc**'ed memory are all **initialized to 0**.

```
void worker(int *foo) {
    int *bar = malloc(sizeof(int));
    for (int i = 0; i < 1000000; i++) {
        ( *foo )++;
        ( *bar )++;
    }
    printf("pid = %d, foo : val = %d, addr = %p\n", getpid(), *foo, foo);
    printf("pid = %d, bar : val = %d, addr = %p\n", getpid(), *bar, bar);
}

int main() {
    int *foo;
    pthread_t t;
    foo = malloc(sizeof(int));
    fork();
    pthread_create(&t, NULL, worker, foo);
    pthread_join(t, NULL);
}
```

a. How many process are created when the code snippet shown above is executed?

b. How many threads (including main thread) are created when the code snippet shown above is executed?

c. What is the **output** of the above code snippet? You may **assume** some **values** for **pid** of each process (e.g., pid = 1) and the **virtual addresses** of foo (e.g., 0x1000) and bar (e.g., 0xFFF0). Assume that the **threads share the same pid** with their parent process. You may also assume that **malloc behaves similarly** for all processes and threads. Remember, there may be multiple correct answers and it's enough to write just ONE correct answer.

Congratulations on finishing the CS 537 Final Exam! :)