Operating Systems **Final Exam** ECE344, Fall 2013

Duration: 2 hour 30 min

Examiner: A. Goel

Please fill your student number and name below and then read the instructions below carefully.

Student Number: |__|__|__|__|__|__|__|__|__|

Last Name: |__|__|__|__|__|__|__|__|__|__|__|__|__|__|

First Name: |__|__|__|__|__|__|__|__|__|__|__|__|__|__|

Instructions

Examination Aids: No examination aids are allowed.

Do not turn this page until you have received the signal to start.

Do not remove any sheets from this test book. Answer all questions in the space provided. No additional sheets are permitted. Use the last blank page as scratch space. This page will not be marked.

This exam consists of 8 questions on 16 pages (including this page). The value of each part of each question is indicated. The total value of all questions is 120.

For the written answers, explain your reasoning clearly. Be as brief and specific as possible. Clear, concise answers will be given higher marks than vague, wordy answers. Marks will be deducted for incorrect statements in an answer. Please write legibly!

Work independently.

MARKING GUIDE

Q1:	_______	(10)
Q2:	_______	(10)
Q3:	_______	(10)
Q4:	_______	(20)
Q5:	_______	(18)
Q6:	_______	(20)
Q7:	_______	(12)
Q8:	_______	(20)
TOTAL:	_______	(120)

To the red pill, and the students who spent countless hours
in the rabbit-hole in Wonderland.

Question 1. True / False [10 MARKS]

[✓ TRUE] FALSE A user-space scheduler needs to use a system call to implement blocking synchronization.

TRUE [✓ FALSE] A user-space scheduler uses 2 kernel threads to multiplex 5 user-level threads. When a user-level thread makes a blocking system call, all of the other user-level threads will be blocked as well.

TRUE [✓ FALSE] When a process issues a system call, the OS code starts by executing an instruction to change the processor mode (from user to kernel).

TRUE [✓ FALSE] The OS has a bug if its exception handler code blocks (e.g., goes to sleep) for any reason.

[✓ TRUE] FALSE A scheduler favoring I/O-bound processes does not significantly delay the completion of CPU-bound processes.

TRUE [✓ FALSE] On the x86 architecture, if a given memory reference (load or store) causes a TLB miss, then that memory reference also causes a page fault.

[✓ TRUE] FALSE The paging daemon does not eliminate the need to run a page replacement algorithm during a page fault.

[✓ TRUE] FALSE The Elevator disk scheduling algorithm bounds the waiting time for all disk requests.

TRUE [✓ FALSE] In a system with vast amounts of memory (e.g., greater than the disk size), a file system will issue disk requests initially to read all data from disk and then never issue disk requests.

[✓ TRUE] FALSE An entry in the open file table maintains a pointer to an in-memory inode structure.

Question 2. OS Design [10 MARKS]

Part (a) [3 MARKS] For the `write` system call shown below, list 3 unique checks that the OS must perform to ensure that an application does not crash or corrupt the OS.

```
int write(int fd, const void *buf, size_t count);
```

Check 1	fd is valid (fd points to a valid index in the file descriptor table).
Check 2	buf[0] to buf[count-1] lies in the address space of process.
Check 3	various other checks: file is not writeable/fd is not open for writing, writing to a file beyond the largest supported file size, etc.

Part (b) [3 MARKS] The timer interrupt is a key mechanism used by the OS. Usually, it waits some amount of time (say 10 milliseconds) and then interrupts the CPU. Suppose, the interrupt is not based on time but rather based on the number of TLB misses the CPU encounters. Once a certain number of TLB misses take place, the CPU is interrupted and the OS gets control (e.g., the page fault frequency replacement algorithm might be implemented this way). Will this design affect the timer interrupt and its usefulness?

1. The timer interrupt will no longer be time based, so programs depending on timing would be affected.
2. The timer interrupt may not fire if a program has not TLB misses, so the OS will never get control. This is a much more serious problem.

Part (c) [4 MARKS] In a Unix file system, the `/A/B/C/D.txt` file uses one block. Assume that each directory in this file system uses one block as well. To read the file `D.txt`, list the maximum and the minimum number of disk reads for different types of blocks (e.g., inode, data, etc.). In both cases, assume that the buffer cache is initially empty.

Maximum	
Block Type	# of reads
super block	1
inode block	5
directory block	4
data block	1

Minimum	
Block Type	# of reads
super block	1
inode block	1
directory block	4
data block	1

When all the five inodes are located in the same block, only one inode block read is required.

Question 3. Synchronization [10 MARKS]

A user has written the snippet of code shown below. Assume that a main thread (code not shown) invokes `Init_thread` once and `Worker_thread` multiple times. All these threads are invoked concurrently.

Use blocking locks and condition variables to ensure that there are no races in this code and to ensure two conditions: 1) the initialization code ($x = 0$) in `Init_thread` runs before x is incremented ($x = x + 1$) by any `Worker_thread`, 2) after 10 worker threads have each incremented x, `Init_thread` calls `exit` so that the program exits.

Make sure to clearly declare and initialize any synchronization variables. You will not need any additional non-synchronization variables. Since you are using blocking synchronization primitives, do not use `sleep`, `wakeup`, tight loops, or interrupt disabling.

```
int x = -1;
lock l = unlocked;
cv init;
cv end;

Init_thread() {
        lock(l);



        x = 0;
        signal(l, init);
        wait(l, end);
        unlock(l);

        exit(0);


}

Worker_thread() {
        lock(l);
        while (x == -1) {
          wait(l, init);
        }
        x = x + 1;
        if (x == 10)
          signal(l, end);
        unlock(l);

}
```

A similar check can also be placed in Init_thread().

Question 4. **Multiprocessor Synchronization** [20 MARKS]

We have seen how the semaphore primitives P and V are implemented for uniprocessors. In this problem, we will implement these primitives for multiprocessors. The relevant code for the uniprocessor version of these primitives is shown below. The `schedule()` function removes the current thread from the run queue and chooses another thread to run.

```
P(sem) {
    spl = splhigh();
    while (sem->count==0) {
        thread_sleep(sem);
    }
    sem->count--;
    splx(spl);
}
```

```
V(sem) {
    spl = splhigh();
    sem->count++;
    thread_wakeup(sem);
    splx(spl);
}
```

```
thread_sleep(void *cond) {
    add_to_wait_queue(cond);
    schedule();
}
```

```
thread_wakeup(void *cond) {
    thread = remove_from_wait_queue(cond);
    add_to_run_queue(thread);
}
```

Part (a) [16 MARKS] In the four questions shown below, circle all (one or more) correct answers. Be careful, two marks will be deducted for each incorrect answer. In the answers below, "corrupt" means an incorrect update.

1) Would the semaphore code shown above work for multiprocessors?

[✓ 1.] No, it could corrupt the `sem->count` variable.

[✓ 2.] No, it could corrupt the wait queue.

[✓ 3.] No, it could corrupt the run queue.

[✓ 4.] No, it could cause a thread to wait forever.

5. Yes, it would work.

2) Your partner argues that the code would clearly not work for multiprocessors and suggests a minimal change: add spin locks in the `thread_sleep` and `thread_wakeup` implementation:

```
thread_sleep(void *cond) {
    spin_lock(schedlock);
    add_to_wait_queue(cond);
    schedule();
    spin_unlock(schedlock);
}
```

```
thread_wakeup(void *cond) {
    spin_lock(schedlock);
    thread = remove_from_wait_queue(cond);
    add_to_run_queue(thread);
    spin_unlock(schedlock);
}
```

Would the semaphore code shown earlier now work for multiprocessors?

[✓ 1.] No, it could corrupt the `sem->count` variable.

2. No, it could corrupt the wait queue.

3. No, it could corrupt the run queue.

[✓ 4.] No, it could cause a thread to wait forever.

5. Yes, it would work.

wait queue and run queue have no races now but P and V still have a race.

3) Your friend from Barkly University argues that the code would clearly still not work for multiprocessors and suggests another minimal change: replace interrupt disabling with spin locks in the `P` and `V` implementation:

```
P(sem) {
    spin_lock(sem->spinlock);
    while (sem->count==0) {
        thread_sleep(sem);
    }
    sem->count--;
    spin_unlock(sem->spinlock);
}
```

```
V(sem) {
    spin_lock(sem->spinlock);
    sem->count++;
    thread_wakeup(sem);
    spin_unlock(sem->spinlock);
}
```

Would this semaphore code work for multiprocessors?

1. No, it could corrupt the `sem->count` variable.

2. No, it could corrupt the wait queue.

3. No, it could corrupt the run queue.

[✓ 4.] No, it could cause a thread to wait forever.

5. Yes, it would work.

spin_lock is not released in P when sleeping. deadlock is possible.

4) Your partner looks at the code, mutters something about Barkly, and suggests adding `spin_unlock` and `spin_lock` around `thread_sleep` in the `P` implementation:

```
P(sem) {
    spin_lock(sem->spinlock);
    while (sem->count==0) {
        spin_unlock(sem->spinlock);
        thread_sleep(sem);
        spin_lock(sem->spinlock);
    }
    sem->count--;
    spin_unlock(sem->spinlock);
}
```

unlock done before sleep. wake up be lost, causing a thread to wait forever.

sem->count is protected by sem->spinlock, so it will not be corrupted.

You are convinced this code will not work for multiprocessors?

1. No, it could corrupt the `sem->count` variable.
2. No, it could corrupt the wait queue.
3. No, it could corrupt the run queue.
4. [✓ No, it could cause a thread to wait forever.]

Part (b) [4 MARKS] You know that the previous code is almost correct, but it does cause problems occasionally. How would you go about fixing the problem? Use the bullet points shown below to state your solution. Be as precise as possible. Hint: the implementation of the semaphore primitives should be tightly coupled with the sleep/wakeup primitives.

1. Add a new spinlock parameter to thread_sleep.
2. Move spin_unlock(sem->spinlock) from the P() code to right AFTER spin_lock(schedlock). In this case, wake up will not be lost because unlocking the semaphore spin lock is done after the scheduler spin lock is acquired (and wakeup will block on acquiring the scheduler lock).
3.
4.

Question 5. TLB [18 MARKS]

Part (a) [4 MARKS] An early RISC processor uses a software-managed TLB with a 4 KB page size, and 4 byte integer size. It only provides a single control bit in the TLB entry. This control bit is the valid bit. You wish to implement swapping on this processor and would like to track referenced and dirty pages. Can you simulate each of these bits in the OS? If so, explain how. If not, explain why. If you make any assumptions, state it clearly below.

Bits	Yes/No	How/Why
Referenced	Yes	On a TLB fault, mark the page table entry as referenced, and load the TLB entry. Periodically (e.g., timer interrupt) mark the page table entry as unreferenced and evict the TLB entry, so that next time the page is accessed, a TLB fault will occur, and the page can be marked referenced again.
Dirty	Yes	The TLB does not have a writeable bit, so we cannot use a TLB fault to determine that a page that was initially read has been dirtied. Instead, when a page is read, we will need to mark it referenced and not load the TLB entry. When a page is written, mark the page dirty and load the TLB. This approach does not take advantage of the TLB for read-only accesses, and hence is very inefficient.

If the answer mentioned disabling/not using the TLB for both read and write, we gave full marks.

Part (b) [4 MARKS] For the processor described above, consider the following code snippet:

```
int a[4096];
int b[4096];

int simple() {
    int c = 0; int i = 0;
cswitch:    /* <- context switch takes place */
    for (i = 0; i < 4096; i++) {
        c = a[i] + b[i];
    }
    return c;
}
```

How many TLB misses will take place when the `simple()` function is run for the first time? Ignore the context switch comment in the code. Provide a justification for each TLB miss. State any assumptions that you make.

page size = 4KB, size of integer is 4 bytes.

Total number of misses = 10.

1 TLB miss for code section (assuming all code fits in 4 KB).
1 TLB miss for stack section (for variables i and c).
8 TLB misses for data section (Array 'a' takes 4 pages (4096 * 4), similarly 'b').

Part (c) [2 MARKS] Now suppose a context switch occurs at the `cswitch:` label in **Part (b)**. At this context switch, the TLB is flushed completely. Will there be additional TLB misses when this code is run? If so, how many? Clearly explain why.

There will be 2 additional TLB misses for the code and stack sections. These entries would have been loaded previously but would have been evicted on a context switch. The data section TLB entries (for the two arrays) would not have been loaded at the time of the context switch, and so there will be no additional TLB misses for the data section.

Part (d) [2 MARKS] Suppose you decide to use a linear (single-level) page table in your OS for the processor described earlier. How many memory accesses would occur to the page table when the code shown in **Part (b)** is run without any context switches?

There would be 10 memory accesses to the linear page table for the 10 TLB misses.

Part (e) [4 MARKS] Now let's imagine that the processor didn't have a TLB and accessed your linear page table directly when translating addresses. How many memory accesses would occur to the page table when the code shown in **Part (b)** is run without any context switches? Circle the best answer below and provide a justification.

1. Same as the answer to **Part (d)**
2. Somewhere between 4K-8K accesses
3. Somewhere between 8K-12K accesses
4. Somewhere between 12K-16K accesses
5. Somewhere between 16K-20K accesses
6. [✓ More than 20K accesses]

The for loop runs 4096 (4K) times. In the loop, there are memory accesses to a[i], b[i], c, and i. In addition there are at least two instructions that are executed in the loop, e.g., the load and store instructions for loading a, b, i, and storing c and i, and a jump instruction. So the total number of accesses to the page table are 4K * (4 data accesses + 2 or more instructions) > 20K accesses.

Part (f) [2 MARKS] Now suppose a context switch occurs at the `cswitch:` label. Would your answer in **Part (e)** be the same or not? Why?

The context switch does not change the number above because the context switch affects the TLB accesses, and in Part (e), there is no TLB.

Question 6. **Page Tables** [20 MARKS]

Suppose that a computer system consists of a processor that uses 50 bit virtual addresses and its paging hardware supports 16 KB page frames. The size of each page table entry is 4 bytes. Each entry has a valid bit, a referenced bit, a dirty bit and two bits for protection that are located in the high order bits of the entry. Please answer the questions shown below. If a question below asks you to calculate a size, show the size in a human readable unit (e.g., KB=2^{10} bytes, MB=2^{20} bytes, GB=2^{30} bytes or TB=2^{40} bytes), unless indicated otherwise.

Part (a) [3 MARKS] If the paging hardware uses a single-level page table, calculate the total size of the page table.

page size = 16 KB, so page offset has 14 bits
# of pages = 2^(50-14) = 2^36
page table size = 2^36 * 4 bytes per entry = 2^38 = 2^8 GB = 256 GB

Part (b) [3 MARKS] Calculate the maximum size of physical memory that can be supported by this system.

frame offset = 14 bits (see above)
PTE has 32 bits. It has 5 control bits, so the maximum frame number can have 27 bits.
So maximum physical memory supported = 2^(27 + 14) = 2^41 = 2 TB

Part (c) [4 MARKS] You realize that a single-level page table is not the best design for this processor. Design a multi-level page table for the paging hardware by answering the following questions: 1) how many levels does your page table design need, 2) how many page table entries will each level have? Ensure that each page table fits within a page frame.

The number of pages is 2^36 (see Part a). So we need 36 bits to index a page number. The page size is 16 KB. The number of page table entries that can fit in a page is 16 KB/4B= 4K = 2^12. So each level can support 12 bits of the page index. So we need a three-level page table, with each level indexing 12 bits of the page number (this accounts for the 36 bits in the page number).

Part (d) [2 MARKS] Suppose a process is using 4 MB of memory and no pages are swapped to disk. What is the minimum amount of memory that your multi-level page table could be using at this time? Show your answer in pages.

A page is 16 KB. Nr.of pages needed for 4 MB is 2^22/2^14 = 2^8 = 256 pages.
A page table has 2^12 = 4096 entries. So a single third-level page table can support all the 256 pages above. The minimum amount of memory needed is a page for each level of the page table or 3 pages.

Part (e) [4 MARKS] Suppose a process is using 4 MB of memory and no pages are swapped to disk. What is the maximum amount of memory that your multi-level page table could be using at this time? Show your answer in pages.

The maximum amount of memory needed for page tables would be as follows:
1. Top-level page table: 1 page
2. Second-level page table: 256 pages, each page pointing to one third-level page
3. Third-level page table: 256 pages, with a single page table entry in each page pointing to a physical frame.
So total number of pages = 513 pages.

Part (f) [4 MARKS] Given the pages being accessed by the process in **Part (e)**, how would you redesign your page table to reduce the memory needed for page tables?

The main idea is to increase the number of levels in the page table and have smaller size page table at each level.

For example, we could have a 4 level page table. Each table would index 9 bits (9 * 4 = 36 bits for indexing all pages). Each page table would be 2^(9 + 2 bits per entry) = 2 ^11 = 2 KB. In this scheme, the total memory requirement is:

1 + 256 + 256 + 256 = 769 pages (but the page size is 1/8 that of Part e).

Another option might be to use an inverted page table. Nr of frames = 2^27. Page table entry size = 8 bytes (4 bytes for next field). So total size of the page table is 2^30 bytes = 2^30 bytes/2^14 bytes/page = 2^16 pages = 65536 pages. So this doesn't really work unless there were many (> 65536 / 513) processes in the system.

Question 7. Disks [12 MARKS]

The shortest-access-time-first (SATF) scheduler is a variant of the shortest-seek-first (SSF) algorithm. It picks the disk IO request from the IO scheduler queue that has the shortest access time from the current head position and services this request. In this question, we'll perform some simple calculations on a highly-simplified disk.

Part (a) [3 MARKS] Assume a simple disk that has only a single track, and a simple FIFO scheduling policy. The rotational delay on this disk is R. There is no seek cost (only one track!), and transfer time is so fast that we just consider it to be free. What is the (approximate) worst case execution time for three (3) requests (to different blocks)?

3R, each request takes roughly R time.

Part (b) [3 MARKS] Now assume that a SATF scheduler is being used by the disk (but it still only has a single track). What is the worst-case time for three requests (to different blocks) now?

R, in one rotation, all the three requests can be served.

Some students wrote that each request could come right after the previous one was served and so the worst case execution time would be 3R. We accepted that answer.

Part (c) [3 MARKS] Now assume the disk has three tracks. The time to seek between two adjacent tracks is S; it takes twice that to seek across two tracks (e.g., from the outer to the inner track). Given a FIFO scheduler, what is the worst-case time for three requests?

3 * (2S + R), each request takes 2S + R time.

Part (d) [3 MARKS] The same question above, but for a SATF scheduler.

3S + 3R. In the worst case, there will be three seeks required and 3 rotations to get to all the requests. The three seeks will occur when the disk head is in the center track, and the head needs to move to out the outer and then the inner track.

We also accepted answers like 4S + R (in one rotation, get to all the blocks by seeking).

We also accepted 3 * (2S + R) if the answer mentioned that each request could come right after the previous one was served.

Question 8. **Feature Full File System (FFFS)** [20 MARKS]

You have just joined a storage startup that has designed a high-end, one petabyte (1024 TB = 2^{50} bytes) storage array for a modern data center. You are in charge of designing a modern, feature-full file system (FFFS) for this storage system. To do so, you start with the traditional Unix file system (UFS) and address many of its limitations.

Part (a) [3 MARKS] Assume the standard UFS supports an 8KB block size, and an inode has 12 direct pointers, one single indirect block pointer, and one double indirect block pointer. Assume also that each pointer is 32 bits in size. What is the approximate maximum file size supported in UFS? State your number in the most appropriate human readable unit (e.g., MB, TB).

Block size = 8 KB.
Nr of pointers in each block = 2^13 / 4 (bytes per pointer) = 2^11 = 2048.
Maximum file size = 8 KB * (12 + 2048 + 2048 * 2048)
Approximately, 8 KB * 2048 * 2048 = 2^13 * 2^11 * 2^11 = 2^35 = 2^5 GB = 32 GB.

Part (b) [3 MARKS] In FFFS, we want to support files that can be as large as 128 TB. What would be the minimal changes you would make to UFS to support these huge files.

128 TB = 2^7 * 2^40 = 2^47
Unfortunately, since the pointer size is 32 bits, and block size is 13 bits, the maximum file size that could be supported with 8 KB blocks is 2^45. So first, we need to increase the block size to 15 bits (32 KB).

Now the maximum file size that can be supported is 32 KB * (2048 * 4) * (2048 * 4) = 2^15 * 2^13 * 2^13 = 2^41.

This is insufficient, so we also need to add a triple indirect block pointer.

Some answers mentioned adding two triple indirect block pointers. We accepted that answer although it doesn't solve the problem associated with the limits of the pointer size * block size.

Part (c) [2 MARKS] In UFS, the number of inodes that can be allocated is fixed when the file system is first created. Describe why it is fixed (i.e., describe how the allocation/deallocation of inodes is managed).

Inodes are allocated from a fixed size linear inode table (an array). As a result, the inode table size cannot be increased after the file system is created.

Part (d) [6 MARKS] In FFFS, we want to support growing the number of inodes over time. Describe the changes you would make to UFS to implement this feature, as precisely as possible.

We could create a multi-level inode table similar to page tables or an inode pointing to blocks using a multi-level scheme. A multi-level inode table can be implemented in may ways. One way would be to have all inode blocks be allocated from the dynamically allocated blocks. A pointer in the super block would point to a single block, which is the first level inode table. The first-level inode table would point to multiple second-level inode blocks, that would be allocated from the dynamically allocated region as more inodes are needed. This scheme can be extended to an arbitrary level inode table.

Some answers mentioned that the inode table should be made really big. That was not an acceptable answer.

Some answers mentioned that new inode tables could be allocated dynamically and linked together in a linked list manner. We accepted that answer although allocating an inode table dynamically may not work if the file system is fragmented.

Part (e) [6 MARKS] In UFS, the file system is created in a fixed size partition or disk. In FFFS, we would like to add new 1 PB storage arrays to the file system dynamically (e.g., starting with a 1 PB file system, grow it to a 8 PB file system over time). Describe the changes you would make to UFS to implement this feature, as precisely as possible.

One way to allow adding disks to UFS while making minimal changes to UFS is to assume that each disk has the same format as UFS. Files can be stored on one disk only (otherwise, block pointers would have to contain a tuple consisting of disk#, block #).

There would be one change required to the disk format: the super block would contain a pointer to the super block on the next disk in the chain.

inode numbers in directory entries could refer to the inode tables on other disks (there would be no change to the inode number format). By caching the super blocks of all the disks, it would be easy to convert an inode number to the inode table on the appropriate disk.

[Use the space below for rough work.]