Operating Systems **Final Exam** ECE344, Fall 2012

Duration: 2.5 hours

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 9 questions on 14 pages (including this page). The value of each part of each question is indicated. The total value of all questions is 90.

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:	_____	(16)
Q3:	_____	(10)
Q4:	_____	(10)
Q5:	_____	(10)
Q6:	_____	(8)
Q7:	_____	(8)
Q8:	_____	(12)
Q9:	_____	(6)
TOTAL:	_____	(90)

Question 1. True / False [10 MARKS]

TRUE FALSE Interrupts should be disabled whenever the processor is in kernel mode.

TRUE FALSE Three processes started running at the same time (e.g., T=0) and have been running for a long time. They have the same fixed priority. A round-robin CPU scheduler will give almost equal CPU allocation to the three processes.

TRUE FALSE Threads in the same process share the heap segment.

TRUE FALSE As the size of the virtual addresss space grows, the size of a page table entry will generally have to grow.

TRUE FALSE A 50-bit virtual address space with 16kB pages and 4-byte page table entries will need a 3-level hierarchical page table structure if each piece of the page table must fit in a single physical page frame.

TRUE FALSE A TLB miss always requires I/O to read the missing virtual page into physical memory.

TRUE FALSE Page buffering eliminates the need to run a page replacement algorithm during a page fault.

TRUE FALSE Of the three main components of disk access time, improvements in disk hardware technology are most likely to reduce transfer time.

TRUE FALSE Assuming that a Unix file consists of 11 blocks, reading the first block of the file will generally be faster than reading the last block of the file.

TRUE FALSE Crash recovery in a journaling file system does not require a full file-system scan.

Question 2. Short Answers [16 MARKS]

Part (a) [4 MARKS] **Protection:** A hardware manufacturer has defined a new feature called User Page Protection (UPP). When the UPP bit is set in the processor status word control register, any attempt to access user-space memory while running in privileged mode causes a page fault. Give one reason why an operating system would use this feature. Describe two OS functions that would need to disable UPP protection.

Part (b) [4 MARKS] **Scheduling:** Give two reasons why a multiprocessor CPU scheduler might assign processes to run on specific processors rather than using a global run queue that serves all the processors.

Part (c) [4 MARKS] **TLB:** Assume the following code snippet:

```
int i;
int p[1024];

for (i = 0;i < 1024;i++) {
    p[i] = 0;
```

How many TLB misses will take place when this code is run for the first time? Provide a justification for each TLB miss. State any assumptions that you make. Assume a 1 KB page size.

Part (d) [4 MARKS] **File Systems:** A program issues the following system calls on a Unix file system:

```
int fd;
fd = open("/Foo", 0); /* Assume that the file exists and is more than 512
                       * bytes. The second argument specifies read-only. */
    read(fd, buf, 512);
```

Show the order in which the blocks are read (and the types of these blocks) to complete the system calls shown above. State the total number of block reads that are performed. You may assume an empty file buffer cache.

Question 3. Synchronization [10 MARKS]

You are implementing an OS with a partner. You ask your partner to write some synchronization functions:

lock(L)	Locks L.
unlock(L)	Unlocks L.
sleep()	Makes the current thread sleep until woken with wake().
wake(T)	Wakes thread T.

Your partner writes the functions above, and then, being really smart, decides to write several combination functions:

lock_sleep(L)	Atomically lock(L) and then sleep().
unlock_sleep(L)	Atomically unlock(L) and then sleep().
lock_wake(L, T)	Atomically lock(L) and then wake(T).
unlock_wake(L, T)	Atomically unlock(L) and then wake(T).

You may assume that all of these functions, but no others, execute atomically.

Part (a) [2 MARKS] You wish to implement the P() and V() semaphore operations using the atomic operations above. Clearly explain why the original synchronization functions (the non-combination functions) are not sufficient for implementing the semaphore operations.

Part (b) [8 MARKS] Implement the semaphore operations using the atomic operations above. Clearly show the shared variables that your code uses. You can assume that a queue implementation is available to you. Clearly indicate the queue functions that your code uses and the type of the arguments provided to the queue functions.

```
// shared variables




// queue functions used and their arguments




P() {









}
V() {









}
```

Question 4. OS161 Process Management [10 MARKS]

Part (a) [4 MARKS] Process management systems calls in Unix-based systems include fork(), exec(), waitpid() and exit(). A student has written code that uses these system calls to launch a child process, have the child execute a program named "hello" (with no arguments), and have the parent wait for the child to complete.

Is the code shown below correct? If not, explain the problem on the right side in one short sentence. Then fix the code by crossing out the lines to be deleted in the code below and then adding new code on the right side.

```
int done = 0;
int rc = fork()

if (rc == 0) { // child
  char *argv[2];
  argv[0] = strdup("hello");
  argv[1] = NULL; // important!
  execv(argv[0], argv);
  done = 1;
} else if (rc > 0) { // parent
  while (done == 0) { // spin
  }
}
```

Part (b) [4 MARKS] Describe how or where an operating system such as OS161 gets the following information about a process.

(a) Where to start executing a program.

(b) The first available heap memory address.

(c) The initial value for the stack pointer for the process.

(d) The arguments passed to a program.

Part (c) [2 MARKS] Process A has issued a system call and is running in kernel mode. Process B is in BLOCKED state. Give one reason why the kernel running in the context of Process A cannot "kill" Process B by invoking thread_exit() and thread_destroy() on Process B.

Question 5. Memory Management [10 MARKS]

Part (a) [4 MARKS] You are updating the operating system for an older CPU architecture that provides base and limit registers for address translation and memory protection. You have been asked to implement paged memory allocation for this system. Is this possible? If so, give a high-level description of how it would work. If not, explain why it would not work.

Part (b) [4 MARKS] A particular system has 17-bit virtual addresses, 20-bit physical addresses, and a 1024 byte page size.

a) How many bits are needed for the offset? _____

b) How many virtual pages can a process have? _____

c) How many physical page frames can the system have? _____

d) If page table entries include Valid, Dirty, Referenced, Read, Write and Execute bits, what is the minimum size of a page table entry? _____

Part (c) [2 MARKS] For the system in part 2 above, should we implement a two-level page table? Why or why not?

Question 6. Virtual Memory Page Replacement [8 MARKS]

Given the following stream of page references by an application, calculate the number of page faults the application would incur with the following page replacement algorithms. Assume that all pages are initially free.

Reference Stream: A B C D A B E A B C D E B A B

Part (a) [2 MARKS] FIFO page replacement with 3 physical pages available.

Part (b) [2 MARKS] LRU page replacement with 3 physical pages available.

Part (c) [2 MARKS] Optimal page replacement with 3 physical pages available.

Part (d) [2 MARKS] If we increase the number of physical pages from 3 to 4, will the number of page faults always decrease using FIFO page replacement for any reference stream? If so, briefly explain your answer. Otherwise, show an example of a reference stream for which the number of page faults with 3 physical pages is less than the number of page faults with 4 physical pages. State the number of page faults in both cases.

Question 7. File Systems [8 MARKS]

For each of the following techniques, briefly explain how it is intended to improve I/O performance, and then describe one case where the technique can be expected to have little or no effect on performance.

Part (a) [2 MARKS] Disk scheduling

Part (b) [2 MARKS] File system block caching

Part (c) [2 MARKS] File block placement (allocation)

Part (d) [2 MARKS] Increasing the file-system block size

Question 8. File System Design [12 MARKS]

The Unix file system uses several types of indirect blocks (e.g., singly indirect, doubly indirect, etc.) for storing metadata information for large files. Thus, accessing a large file requires accessing several additional indirect blocks.

You decide to implement an extent-based file system, *efs*, to reduce metadata storage and access overhead for large files. An extent consists of a contiguous sequence of sectors on disk. Your file system uses the following representation for an inode:

```
#define NEXTENT 12

struct efs_inode {
        uint64_t filesize;
        uint16_t type;
        uint16_t nlink;
        uint32_t extent_size;
        uint32_t extents[NEXTENT];
}
```

The `extent_size` is a 32-bit unsigned number that indicates the size of each extent, in terms of the number of consecutive 512-byte sectors. All extents of a file are the same size. The `extents` array holds the starting disk sector number for each extent.

Files are created with an initial `extent_size` of 1 (that is, each extent is a single 512-byte sector). Whenever the file grows larger than the maximum file size supported by the current extent-size, the extent-size is doubled.

Part (a) [3 MARKS] What is the maximum file size supported by efs? Express your answer as $n * 2^m$ and explain your reasoning.

Part (b) [3 MARKS] Assuming that the inode is already in memory, how many disk accesses are required to read the byte at offset 1,000,000 in a file? *Briefly* explain your answer.

Part (c) [4 MARKS] What operations are required when the file grows larger than the maximum file size supported by the current extent-size (in addition to doubling the extent-size field in the inode)?

Part (d) [2 MARKS] Describe two drawbacks of the efs design over a traditional Unix file system.

Question 9. Journaling File System [6 MARKS]

Your friend has designed a new file system called the Confused Journaling File System (CJFS). CJFS tries to keep all of its meta-data consistent through the use of journaling. CJFS updates the file system when an application has appended a single block to a file as follows:

1 - write begin transaction to journal
2 - write journal descriptor to journal (describing the update)
3 - write data bitmap, inode, data block to journal
4 - write end transaction to journal
5 - write data bitmap, inode, data block to their final in-place locations
6 - free transaction in journal

However, CJFS has to wait for various I/Os to complete in order to work correctly. For example, if CJFS issues I/Os 1 and 2, waits for them to complete, and then issues I/O 3, it knows that I/Os 1 and 2 have completed before 3.

The following are some suggestions as to where to place such waits. Your job is to figure out whether the suggested placements work. Circle whether CJFS is either too slow (because it spends too much time waiting for I/Os to complete), broken (because it doesn't actually guarantee that meta-data is kept consistent), or just fine, for the following options:

(a) Wait between 3 and 4:	Too slow	Broken	Just Fine
(b) Wait between 3 and 4, Wait between 5 and 6:	Too slow	Broken	Just Fine
(c) Wait between 3 and 4, Wait between 4 and 5, Wait between 5 and 6:	Too slow	Broken	Just Fine
(d) Wait between 1 and 2, Wait between 3 and 4, Wait between 4 and 5, Wait between 5 and 6:	Too slow	Broken	Just Fine
(e) Wait between 1 and 2, Wait between 5 and 6:	Too slow	Broken	Just Fine
(f) Wait between 3 and 4, Wait between 4 and 5	Too slow	Broken	Just Fine

[Use the space below for rough work.]

END OF EXAMINATION