

Final Exam

CMSC 412
Operating Systems
Fall 2004

December 16, 2004

Guidelines

This exam has 10 pages (including this one); make sure you have them all. Put your name on each page before starting the exam. Write your answers directly on the exam sheets, using the back of the page as necessary. Bring your exam to the front when you are finished. Please be as quiet as possible.

Errors on the exam will be posted on the board as they are discovered. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn't need to do this at all, so be careful when making assumptions.

Use good test-taking strategy: read through the whole exam first, and answer the questions easiest for you first.

Have a good Holiday!

Question	Points	Score
1	25	
2	25	
3	25	
4	20	
5	10	
6	20	
7	25	
Total	150	

1. (Operating Systems Concepts, 25 points)

In three sentences or less, compare and contrast the following terms.

- (a) Virtual memory vs. Paging

Answer:

It is convenient for each process to have its own logical address space that is distinct from the physical addresses of actual memory. Paging implements the mapping from logical to physical addresses in chunks of memory called pages. Virtual memory allows processes larger than the available physical memory to be executed, and is often implemented using paging.

- (b) Protection vs. Security

Answer:

Protection is the means by which an operating system prevents processes from accessing objects in an unauthorized way. Security addresses external threats to the system, e.g., from the network or humans, which would damage objects directly or by circumventing the protection system.

- (c) Mutual exclusion vs. Progress

Answer:

Mutual exclusion and progress are both criteria of a correct solution of the critical section problem. Mutual exclusion ensures that no more than one process executes within any critical section simultaneously, while progress requires that no process be prevented from executing its critical section if no other process is executing within its critical section.

- (d) Buffer vs. Cache

Answer:

Both are means of temporary storage. A cache is a copy of data stored elsewhere on a slower medium, while a buffer has the only copy of the data, and is typically used for I/O.

- (e) Rotational latency vs. Seek time

Answer:

Both are measures of disk performance. The rotational latency is the time it takes the disk to rotate to the desired sector on the current cylinder, while the seek time is the time it takes the disk head to move to the desired cylinder.

2. (Concurrency, 25 points)

For each program below, indicate whether or not it could exhibit a deadlock, a race condition, or both. If so, explain why. Assume that the function `thread1` runs in one thread and the function `thread2` runs in another thread, and that the following data are shared between threads, and initialized as indicated prior to execution in each case:

```
semaphore m = 1;
semaphore n = 1;
boolean lm = false;
boolean ln = false;
int x = 0;
int y = 0;
```

```
(a) thread1() {
      wait(&m);
      x ++;
      wait(&n);
      y ++;
      signal(&n);
      signal(&m);
    }
    thread2() {
      wait(&n);
      x ++;
      wait(&m);
      y ++;
      signal(&m);
      signal(&n);
    }
```

Answer:

Race condition and deadlock.

```
(b) thread1() {
      while (1) {
        if (!tset(&lm)) {
          if (!tset(&ln)) {
            x ++;
            y ++;
            break;
          }
          else lm = false;
        }
      }
      lm = false;
      ln = false;
    }
    thread2() {
      while (1) {
        if (!tset(&ln)) {
          if (!tset(&lm)) {
            x ++;
            y ++;
            break;
          }
          else ln = false;
        }
      }
      ln = false;
      lm = false;
    }
```

Answer:

No race condition or deadlock.

(Here is the shared data again, for your reference.)

```
semaphore m = 1;
semaphore n = 1;
boolean lm = false;
boolean ln = false;
int x = 0;
int y = 0;
```

```
(c) thread1() {
      wait(&m);
      x ++;
      wait(&n);
      signal(&m);
      y ++;
      signal(&n);
    }
    thread2() {
      wait(&m);
      x ++;
      wait(&n);
      signal(&m);
      y ++;
      signal(&n);
    }
```

Answer:

No race condition or deadlock.

```
(d) thread1() {
      signal(&n);
      wait(&m);
      x ++;
      signal(&m);
      wait(&n);
      y ++;
      signal(&n);
    }
    thread2() {
      wait(&m);
      x ++;
      signal(&m);
      wait(&n);
      y ++;
      signal(&n);
    }
```

Answer:

Race condition.

3. (File Systems, 25 points) GeekOS's GOSFS file system simplifies the UNIX approach. Each block is 4096 bytes, and is either a data block for a file, or a directory block, consisting of up to 23 directory entries, defined as follows:

```
struct GOSFS_Dir_Entry {
    ulong_t size;          /* Size of file. */
    ulong_t flags;        /* Flags: used, isdirectory */
    char filename[128];
    ulong_t blockList[10];
    /* Pointers to direct (8), indirect (1), and doubly-indirect (1) blocks. */
};
```

A directory entry describes the contents of the file. The flags indicate whether or not it is in use, and whether or not it describes a file or a directory. The entry specifies the file name (up to 127 characters), and the `blockList` array points to blocks containing the data for the file or directory. A directory consumes at most one disk block (a *directory block*), and thus may contain at most 23 files (since directory entries are 172 bytes). This block is pointed to by `blockList[0]`. Every time a block is read, that block is cached in the buffer cache (which for this problem you can assume is infinitely sized).

Fill out the table below indicating how many disk blocks are read or written on each system call. Assume that we have already read the superblock into memory (when mounting the file system), and that we start with an empty buffer cache. All seek offsets and read/write lengths are in bytes; seek offsets are absolute offsets into the file. The files `/home/mwh/grades.txt` and `/home/mwh/exam.txt` are each size 50 KB at the start.

If you wish, you may state assumptions about your answers, draw pictures, etc. This may get you some partial credit. However, if these things are incorrect, you could lose points.

Answer:

<i>Syscall</i>	<i>Blocks read from disk</i>	<i>Blocks written to disk</i>
<i>open /home/mwh/grades</i>	<i>3</i>	
<i>seek to offset 100</i>		
<i>read length 50</i>	<i>1</i>	
<i>close</i>		
<i>open /home/mwh/exam.txt</i>		
<i>seek to offset 8100</i>		
<i>write length 100</i>	<i>2</i>	<i>2</i>
<i>seek to offset 33000</i>		
<i>read length 120</i>	<i>2</i>	
<i>close</i>		

4. (Disk scheduling, 20 points) Suppose that a disk drive has 10,000 cylinders, numbered 0 to 9999. The driver is currently serving a request at cylinder 1400. The queue of pending requests is, in the order received:

100, 1200, 900, 8000, 8100, 100, 8200, 1000, 4200

Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for the following scheduling algorithms. (For the algorithms in which the head is in constant motion, indicate the direction in which you assume it is moving initially.)

- (a) FCFS

Answer:

36400

- (b) SSTF

Answer:

9400

- (c) SCAN

Answer:

9600 or 18498

- (d) C-LOOK

Answer:

13400 or 16000

5. (Protection, 10 points)

- (a) (5 points) Does UNIX use capabilities, access control lists, or both? Justify your answer.

Answer:

UNIX uses ACLs on files (e.g., the standard read/write/execute permissions). It also uses capabilities: file descriptors, semaphore identifiers, etc. are essentially capabilities.

- (b) (5 points) Say you were running on a system without virtual memory (e.g., on an embedded device). If the operating system could insert into or rewrite code within a binary file before running it, how might it provide virtual-memory-style protection? How would this approach compare to using virtual memory in terms of cost and performance?

Answer:

Virtual memory protection is usually done by hardware: it makes sure that loading/writing/executing a certain address is legal before actually doing it. The operating system could similarly insert software instructions into a program prior to such protected operations, to check whether they are legal; e.g., whether the target address is within the appropriate range. This would be more expensive for execution within a process, but would make interprocess communication much cheaper since it could avoid address space switches.

6. (Virtual Memory, 20 points)

There are two basic performance considerations for virtual memory systems: efficient use of memory, and efficient per-memory accesses (i.e. fewer delays on the CPU). For each of the following design points, explain how the range of possible choices affects these two performance criteria. Present points for and against a given choice.

- (a) (7 points) Single-level page table vs. multi-level page table.

Answer:

Single-level page table: Address translations are reasonably efficient, but the table could require a lot of space per process.

Multi-level page table: Address translations are more expensive, since they require multiple lookups, but the table will require much less space, since lower-level page tables can be paged out.

- (b) (7 points) Small vs. large page size.

Answer:

A small page size results in less internal fragmentation and finer-grained access control to pages, but can result in more page-outs to disk, fewer addresses within the TLB, and larger page tables. Large page sizes are the opposite.

- (c) (6 points) Choice of page replacement algorithm.

Answer:

A page replacement algorithm like LRU requires more overhead per memory access to keep track of the order pages were accessed, but reduce the total number of page-outs. In contrast, an algorithm like FIFO imposes no overhead on memory accesses, but may do a poorer job of keeping pages in memory.

7. (Process Scheduling, 25 points)

- (a) For normal processes, Linux uses a *credit-based*, preemptive, prioritized scheduling algorithm. Each process is associated with a counter of *credits*. The credits are initialized to the process priority when it enters the system, and the credits are reduced by 1 each time the quantum expires while the process is running. The process with the most credits left is chosen to run. (If the scheduler must choose between processes with the same number of credits, the one waiting the longest is run first; if more than one has been waiting the same amount of time, it chooses the one with the lowest process ID.) When no runnable process has any credits left (i.e. all processes with non-zero credits are blocked, say on I/O), the credits of *all* processes are increased by the formula $credits = (credits/2) + priority$ (truncating the fraction).
- i. (5 points) How does this scheme tend to favor I/O-bound processes over CPU-bound ones? Why is this a good idea for Linux?

Answer:

Because processes with the most credits are scheduled first, and because I/O-bound processes are more likely to gain credits over time, I/O bound processes become preferred scheduling choices. This is good for Linux, which is used both interactively via the keyboard and via the web as a server (either way, end users are I/O-bound).

- ii. (5 points) How does this scheme compare with the multi-level prioritized scheduler you wrote for GeekOS? Are the goals the same? How are the mechanisms different?

Answer:

The goals are the same, but the method of prioritization was different: queue level as opposed to priority number. The main difference is the criteria used to move to different priorities; the Linux approach is a bit more fine-grained.

(next page)

- (b) Linux also supports real-time scheduling, using a round-robin scheduler. If any real-time process is ready to run, the one with the highest priority is scheduled (a real-time process always is favored over a normal process). If multiple processes could run, the one that has waited the longest is chosen, else the one with the lowest process ID.

Consider the following workload:

Process	Priority	Burst Times
p_1	2	6,6,4
p_2	1	2,2,3,2
p_3	1	16

Draw a Gantt chart to illustrate how these processes would be scheduled using Linux's normal process scheduling algorithm, and its prioritized round-robin algorithm. *You do not need to include context-switching time in your picture.* Assume that the quantum is set to 5 time units, and that the time spent doing I/O between bursts is 1 time unit. For each algorithm, calculate the waiting time for each process.

- (8 points) Linux normal process scheduling.

Answer:

Here is the process scheduling order, with the amount of time each process takes in parentheses: $P1(5)$, $P2(2)$, $P3(5)$, $P1(1)$, $P2(2)$, $P1(5)$, $P2(3)$, (recredit here), $P1(1)$, $P2(2)$, $P1(4)$, $P3(11)$.

This answer assumes you reset waiting times after the recredit. If you don't do this, then the answer is $P1(5)$, $P2(2)$, $P3(5)$, $P1(1)$, $P2(2)$, $P1(5)$, $P2(3)$, (recredit here), $P1(5)$, $P3(5)$, $P1(4)$, $P2(2)$, $P3(6)$.

- (7 points) Linux real-time round robin scheduling.

Answer:

$P1(5)$, $P1(1)$, $P2(2)$, $P1(5)$, $P1(1)$, $P3(5)$, $P1(4)$, $P2(2)$, $P3(5)$, $P2(3)$, $P3(5)$, $P2(2)$, $P3(1)$.