

Midterm #2

CMSC 412
Operating Systems
Fall 2005

November 22, 2004

Guidelines

This exam has 7 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.

If you have a question, raise your hand and I will come to you. Errors on the exam will be posted on the board if 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.

Question	Points	Score
1	15	
2	20	
3	25	
4	15	
5	25	
Total	100	

1. (Fragmentation, 15 points)

- (a) (6 points) Which of the following schemes for *memory allocation* exhibit internal fragmentation? Which exhibit external fragmentation? Check all the boxes that apply.

Answer:

<i>Allocation Scheme</i>	<i>External</i>	<i>Internal</i>
<i>Contiguous</i>	×	
<i>Segmented</i>	×	(×)
<i>Paged</i>		×

Segmentation may or not have internal fragmentation, depending on whether there is a minimum segment size. On the x86, for example, segments are in increments of bytes, or increments of 4K pages, with the latter having a much larger maximum segment size.

- (b) (6 points) Which of the following schemes for *file allocation* exhibit internal fragmentation? Which exhibit external fragmentation? Check all the boxes that apply.

Answer:

<i>Allocation Scheme</i>	<i>External</i>	<i>Internal</i>
<i>Contiguous</i>	×	×
<i>Linked</i>		×
<i>Indexed</i>		×

All schemes exhibit internal fragmentation because files are allocated in blocks, rather than bytes.

- (c) (3 points) Of the schemes in part 1b that exhibit internal fragmentation, what is the maximum internal fragmentation possible per file if using a block size of B bytes?

Answer:

The simplest reasonable answer is $B - 1$ bytes: a file with no data needs no blocks at all, while a file with 1 byte needs one block, thus wasting $B - 1$ bytes. However, the answer can be more precise. Under the indexed scheme, there is also the overhead of the index block, which if we assume the block is B bytes and block pointers are 4 bytes, the overhead is $B - 4$ bytes, so the total becomes $2B - 5$ bytes. Having multiply-indirected blocks would make it worse!

2. (Page Replacement, 20 points)

- (a) (4 points) Suppose a program issues a request for a page not currently in main memory. Given that main memory is composed of 4 frames, use the following table to determine which frame's contents should be swapped out:

Frame #	Time When Loaded	Time Last Referenced	Referenced Bit	Modified Bit
0	126	279	0	0
1	230	280	1	0
2	120	282	1	1
3	160	290	1	1

- i. When using FIFO page replacement. **Answer:**

2

- ii. When using LRU page replacement. **Answer:**

0

- iii. When using Most-Recently Used (MRU) page replacement. **Answer:**

3

- iv. When using Least-Frequently Used (LFU) page replacement. **Answer:**

0. Note that since we do not know when the reference bits were last reset, this may not be accurate. So another reasonable answer is simply "we can't tell."

- (b) For each of the next two questions, assuming no pages are in memory at the start, indicate which pages are in memory in the indicated frames *following* each reference in the sequence, using the given page replacement algorithm. If a fault occurs, put a check the "fault?" box. If an algorithm needs to evict a frame, but could choose equally from among multiple frames (i.e., there is a tie according to the algorithm), choose the frame for the page that was brought in to memory least recently. (Next page, please.)

i. (8 points) FIFO page replacement.

Answer:

<i>page reference</i>	<i>0</i>	<i>1</i>	<i>7</i>	<i>2</i>	<i>3</i>	<i>2</i>	<i>7</i>	<i>1</i>	<i>0</i>	<i>3</i>
<i>frame 1</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>3</i>	<i>3</i>	<i>3</i>	<i>3</i>	<i>3</i>	<i>3</i>
<i>frame 2</i>		<i>1</i>	<i>1</i>	<i>1</i>	<i>1</i>	<i>1</i>	<i>1</i>	<i>1</i>	<i>0</i>	<i>0</i>
<i>frame 3</i>			<i>7</i>	<i>7</i>	<i>7</i>	<i>7</i>	<i>7</i>	<i>7</i>	<i>7</i>	<i>7</i>
<i>frame 4</i>				<i>2</i>	<i>2</i>	<i>2</i>	<i>2</i>	<i>2</i>	<i>2</i>	<i>2</i>
<i>fault?</i>	×	×	×	×	×				×	

ii. (8 points) LRU page replacement.

Answer:

<i>page reference</i>	<i>0</i>	<i>1</i>	<i>7</i>	<i>2</i>	<i>3</i>	<i>2</i>	<i>7</i>	<i>1</i>	<i>0</i>	<i>3</i>
<i>frame 1</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>3</i>	<i>3</i>	<i>3</i>	<i>3</i>	<i>0</i>	<i>0</i>
<i>frame 2</i>		<i>1</i>	<i>1</i>	<i>1</i>	<i>1</i>	<i>1</i>	<i>1</i>	<i>1</i>	<i>1</i>	<i>1</i>
<i>frame 3</i>			<i>7</i>	<i>7</i>	<i>7</i>	<i>7</i>	<i>7</i>	<i>7</i>	<i>7</i>	<i>7</i>
<i>frame 4</i>				<i>2</i>	<i>2</i>	<i>2</i>	<i>2</i>	<i>2</i>	<i>2</i>	<i>3</i>
<i>fault?</i>	×	×	×	×	×				×	×

3. (File System Implementation, 25 points)

Background: Say we have a file system that uses 1000 byte blocks. In the file system's metadata, block numbers are expressed as 2 byte integers.

Assume that all accesses to the disk are through a buffer cache that contains at most 4 blocks, using an LRU replacement strategy. Writes to disk should only occur when the user indicates this explicitly with a `sync()`, or when a dirty buffer needs to be evicted. Do not assume that you have any other caches in the system (e.g., a name cache).

The root directory is stored at block 1, with all directory entries fitting in this single block. This directory contains a file `file`, consisting of 10 data blocks. `file`'s FCB and data blocks are stored on disk at the following blocks:

File block #	FCB	0	1	2	3	4	5	6	7	8	9
Disk block #	10	4	12	15	16	17	2	3	20	21	22

Problem: Say a user program issues the following system calls for accessing the filesystem. Assuming we start with an empty buffer cache, fill out the table below indicating how many disk blocks are read or written on each system call. For the first column, assume you are using a file allocation table (FAT). Assume the FAT is stored at block 0 on the disk. For the second column, assume you are using an indexed allocation strategy, where the index block for `file` is stored at block 23.

If you wish, you may state assumptions about your answers, draw pictures, etc. It may be useful for you to keep a picture of the contents of the buffer cache next to each line in the table below. This may get you some partial credit. However, if these things are incorrect, you could lose points.

Answer:

Operation	FAT		Indexed	
	# reads	# writes	# reads	# writes
<i>open /file</i>	2		2	
<i>read 50 bytes at offset 100</i>	1		2	
<i>write 200 bytes at offset 400</i>				
<i>write 50 bytes at offset 1100</i>	2		1	
<i>write 50 bytes at offset 8100</i>	1	1	1	1
<i>read 50 bytes at offset 250</i>	1	1	1	1
<i>sync</i>		1		1
<i>read 50 bytes at offset 3400</i>	1		1	

The *open* requires reading in the root directory block; the answer also includes reading in the file control block, though that could be delayed. The first read requires reading the first block of the file; in both cases, for this you need the file control block, and for the indexed case you also need the index block. To write to a block, we have to first read it in. However, in the case of the first write, the block is already in the cache. The write marks this block as dirty. For the next write, the block has to be read in again; in the case of FAT, this requires reading in the FAT, which is stored in a single block, to get the link to the file's second block. The next two actions cause the dirty blocks to get evicted from the cache, and so these must be written. Therefore, at the time of *sync*, the only dirty block is the ninth file block, which gets written. Finally, the last read accesses a block not in the cache, so it must be read from disk.

4. (Disk Scheduling, 15 points)

- (a) (5 points) In the last problem, the file system implementation only interacts with the buffer cache, and it is the buffer cache that actually reads from or writes to the disk. Using this scheme, when can the disk scheduling algorithm make a difference in system performance? Does it make any difference for this particular trace?

Answer:

*The only place a disk scheduling algorithm can make a difference is when there are multiple read and write requests available at once. This could happen under high load to the file system, where potentially many processes are accessing the file system through the shared buffer cache. The only way it could arise in the prior problem is due to **sync**: multiple dirty blocks could be in the cache, and they could be reordered to improve throughput. However, since we only ever write one block to disk at a time in the prior problem, this never comes up.*

- (b) (10 points) Assume you have a disk with 100 blocks (numbered 0–99), and the operating system's queue to read or write from the device contains the blocks 5, 50, 25, 95, 10. If the disk head is currently at block 45, what is the total *seek distance* when using

- i. Shortest Seek Time First (SSTF) **Answer:**

The order the blocks are written is 50, 25, 10, 5, 95, and since the head begins at 45 the seek distance is 140.

- ii. First-Come First-Served (FCFS) **Answer:**

The order the blocks are written is 5, 50, 25, 95, 10, and since the head begins at 45 the seek distance is 265.

5. (Short Answer, 25 points)

(a) (5 points) What is a virtual filesystem (VFS)? Why is it useful? **Answer:**

A virtual file system is an abstraction of a file system's implementation from its interface, as viewed by the rest of the operating system. System calls interact with the VFS interface, rather than a particular implementation. This makes the implementation of file systems more modular, so that it's easier to add support for new file systems, and easier to port and maintain existing ones.

(b) (10 points) Say I have a paged memory system that uses a hash-table-based page table and a TLB, where the time ϵ to access the TLB is .001 memory references, and page table lookups take, on average, 2 memory references (not including the actual read to the requested address). Say memory references hit the TLB 70% of the time, on average.

If as a hardware designer I had the choice to increase the size of the TLB to improve the hit rate to 80%, or to use a smarter hashing algorithm to reduce the average page table lookup time to 1.8 references, which choice should I make if I want to improve the effective access time? Justify your answer.

Answer:

We can justify our answer by calculating the effective access time (EAT) in each case. For the modified page table:

$$EAT_{pt} = 0.7(.001 + 1) + 0.3(.001 + 1.8 + 1) = 1.541$$

Here: 0.7 is the frequency that requests hit the TLB; the cost of accessing the TLB (.001) is borne whether we hit the TLB or not. The 1 appearing in both parts is the time to read the actual memory, while the 1.8 is the time to access the page table.

For the modified TLB:

$$EAT_{tlb} = 0.8(.001 + 1) + 0.2(.001 + 2 + 1) = 1.401$$

Since the latter is smaller, this is the preferred choice.

(c) (5 points) Can virtual memory be implemented using segmentation? Why or why not?

Answer:

Yes. Rather than swapping pages to and from disk, we would swap segments. The segment table would need valid/present bits, so that when a segment is swapped out, its valid bit would be set to false. The ensuing segmentation fault would result in the OS swapping the segment back in. This could be difficult or inefficient if segments are very large in practice.

(d) (5 points) Would it make sense to implement a file system for a tape drive using techniques we have discussed in class (e.g., that one uses to implement FAT, UFS, extfs, etc.)? Why or why not? **Answer:**

No. The performance would be abysmal. The main problems are that tapes are bad at random access, since they must wind or unwind to get to a specific spot, and they must truncate after each write. This would result in bad seek times for reads, and massive amounts of recopying if files were ever written after they were created.