Midterm #2

CMSC 412
Operating Systems
Fall 2004

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. However, to minimize interruptions, you may only do this once for the entire exam. Therefore, wait until you are sure you don't have any more questions before raising your hand. 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.

Question	Points	Score
1	20	
2	25	
3	15	
4	25	
5	10	
Total	100	

1. (Memory Management, 20 points)

   (a) (2 points) If an architecture has a 32-bit address space. defines 4 kilobyte pages, and each page table entry consumes 4 bytes, how much memory would be required to keep the entire page table in memory?

   Answer:

   2^{32} *bytes* $/2^{12}$ *bytes per page* $= 2^{20}$ *pages* $\times 4$ *bytes per entry* $= 4$ *megabytes*

   (b) (5 points) To avoid keeping the entire page table in memory, the Intel 386 (and later) uses a two-level paging scheme in which a per-process page directory points to individual page tables. Page tables can themselves be paged. In GeekOS, most of you did not implement pageable page tables: is the two-level scheme actually saving you any physical memory in this case? Why or why not?

   Answer:

   Yes, it can save you memory when a process is not using the entire virtual address space. A page table does not have to be created for an address range that is not in use by the process; i.e., page tables are allocated on-demand. A single-level page table would allocate all page tables, even those for addresses not in use.

   (c) (8 points) Explain the difference between internal and external fragmentation. Which of these is exhibited by a paged memory scheme? Which is exhibited by a segmented memory scheme?

   Answer:

   Fragmentation *is a situation in which a request for* N *bytes cannot be satisfied despite the fact that* M *bytes are unused, where* $M > N$. External fragmentation *occurs when available memory is split into many discontiguous chunks* C_i *where* $size(C_i) < N$. *A segmented memory scheme can exhibit external fragmentation because it usually allocates variable-sized chunks (depending on the size of the segment), and allocated memory must be contiguous.* Internal fragmentation *occurs when extra memory is provided for a request; i.e., a request of size* S *is satisfied with a chunk of size* $C > S$, *so* $C - S$ *bytes are unused but not allocatable. This occurs in a paged memory scheme, which satisfies all requests with an integral number of pages. This scheme does not exhibit external fragmentation because allocated pages need not be contiguous.*

   (d) (5 points) What allocation strategy (contiguous, non-contiguous) did GeekOS use for processes in project 2? Could you have used the other strategy instead? How? What allocation strategy was used in project 4?

   Answer:

   Project 2 used contiguous allocation (segmentation). You could have used a non-contiguous strategy by allocating separate memory for each segment. Project 4 used non-contiguous allocation (paging).

2. (Page Replacement, 25 points) Suppose we have a virtual address space consisting of 5 pages, numbered 1 to 5, and physical memory to hold 4 frames.

   (a) For each of the next three 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.

      i. (7 points) FIFO page replacement.

Answer:

page reference	*2*	*1*	*3*	*4*	*1*	*2*	*5*	*1*	*2*	*3*	*4*	*5*
frame 1	**2**	*2*	*2*	*2*	*2*	*2*	**5**	*5*	*5*	*5*	*5*	*5*
frame 2		**1**	*1*	*1*	*1*	*1*	*1*	*1*	**2**	*2*	*2*	*2*
frame 3			**3**	*3*	*3*	*3*	*3*	*3*	*3*	*3*	*3*	*3*
frame 4				**4**	*4*	*4*	*4*	*4*	*4*	*4*	*4*	*4*
fault?	×	×	×	×			×		×			

      ii. (7 points) LRU page replacement.

Answer:

page reference	*2*	*1*	*3*	*4*	*1*	*2*	*5*	*1*	*2*	*3*	*4*	*5*
frame 1	**2**	*2*	*2*	*2*	*2*	*2*	*2*	*2*	*2*	*2*	*2*	*2*
frame 2		**1**	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	**5**
frame 3			**3**	*3*	*3*	*3*	**5**	*5*	*5*	*5*	**4**	*4*
frame 4				**4**	*4*	*4*	*4*	*4*	*4*	**3**	*3*	*3*
fault?	×	×	×	×			×			×	×	×

(next page)

iii. (7 points) Pseudo-LRU page replacement as you implemented GeekOS.

Answer:

page reference	*2*	*1*	*3*	*4*	*1*	*2*	*5*	*1*	*2*	*3*	*4*	*5*
frame 1	**2**	*2*	*2*	*2*	*2*	*2*	*2*	*2*	*2*	*2*	*2*	**5**
frame 2		**1**	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*
frame 3			**3**	*3*	*3*	*3*	**5**	*5*	*5*	*5*	**4**	*4*
frame 4				**4**	*4*	*4*	*4*	*4*	*4*	**3**	*3*	*3*
fault?	×	×	×	×			×			×	×	×

The strategy in GeekOS worked by tagging each page with a "last referenced" clock value. The page's clock value is set to the current time when it is first is brought into memory. Whenever a page is accessed, its reference bit is set. When a page fault occurs, all those pages whose reference bit is set have their clock value set to the current time, and the reference bits are cleared.

(b) (4 points) FIFO is known to exhibit Belady's anomaly. Could either LRU or GeekOS pseudo-LRU exhibit Belady's anomaly? Why or why not?

Answer:

Belady's anomaly is the situation in which adding more physical frames increases the number of page faults for a particular reference string. LRU cannot exhibit this because it is a stack-based algorithm. That is, adding more frames simply adds to the set of choices for a possible victim page, but does not replace one set of choices for another. Pseudo-LRU can exhibit Belady's anomaly because it degenerates to FIFO in the case that all pages are accessed between page faults (i.e., all the reference bits are set, so we must pick the oldest page).

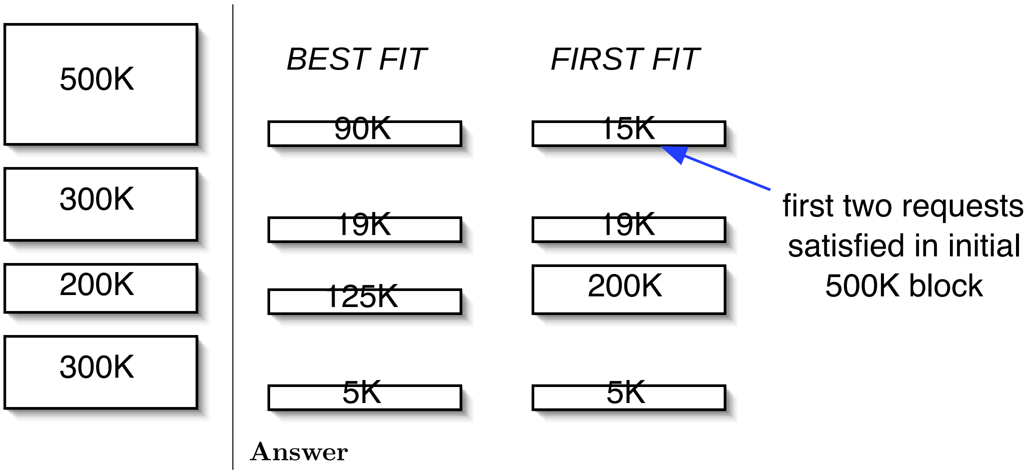
3. (Free space Management, 15 points)

(a) (3 points) A bit vector can be used to keep track of which blocks are free in a file system's partition on disk. Assuming a 1 kilobyte block size, and a disk size of 40 gigabytes, how big would the bit vector have to be?

Answer:

40×2^{30} *bytes* / 2^{10} *bytes per block* = 40×2^{20} *blocks. At one bit per block, this is* 5 *MB of space.*

(b) (12 points) Consider the free memory layout shown below (the blocks are ordered from top to bottom). To the right, draw to the what the free memory layouts would be after satisfying requests for (in order) memory blocks of size 75K, 410K, 281K, and 295K. Draw one memory layout for the *best-fit* algorithm, and one for *first-fit*.



4. (File System Implementation, 25 points) Say we have a file system that uses 4096 byte blocks. Assume that each time a file block is read, it is cached in the buffer cache. For this file system, each file is associated with a file control block (FCB, e.g., a UNIX *inode*) that is read into memory when the file is opened, and that this block points to those blocks that make up the file's contents (how this is implemented depends on the allocation strategy, see below).

   Say a user program issues the following system calls for accessing the file `grades`. This file is 50K (51200 bytes) in size, and is stored in the root (`/`) directory. Assuming we start with an empty buffer cache, fill out the table below indicating how many disk blocks are read on each system call. You will compare different file allocation strategies, one per column in the table. The first line of the table is filled in for you.

   For the first column, assume you are using the linked allocation strategy. For the second column, assume you are using a hybrid indexed allocation strategy as with UNIX: the first 12 blocks of the file are linked directly from the FCB, followed by a single-indirect link, a double-indirect link, and a triple indirect link.

   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.

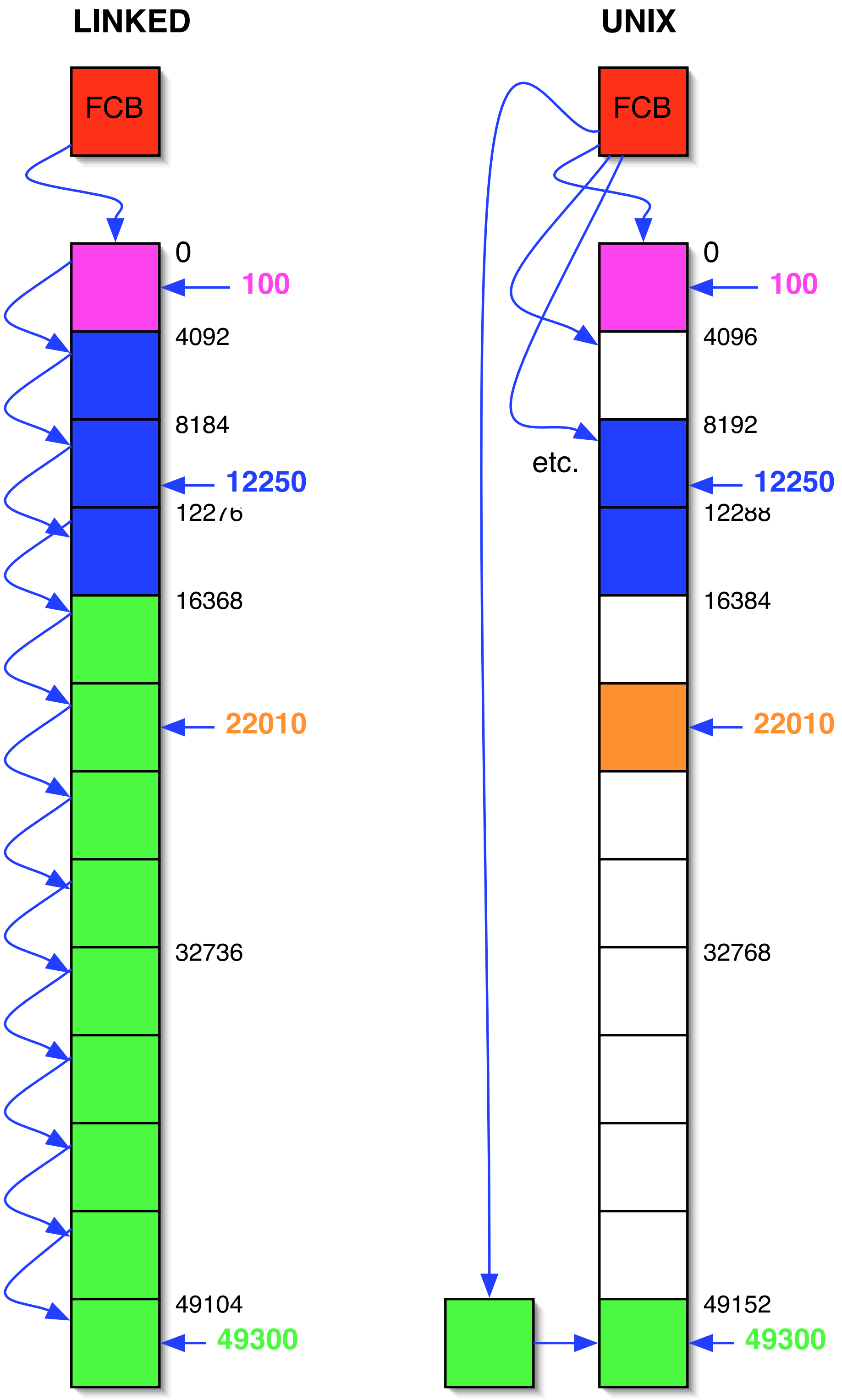
Answer:

Syscall	*Linked*	*UNIX*
open `/grades`	*2*	*2*
seek to offset 100	n/a	n/a
read length 50	*1*	*1*
seek to offset 12250	n/a	n/a
read length 60	*3*	*2*
seek to offset 49300	n/a	n/a
read length 120	*9*	*2*
seek to offset 22010	n/a	n/a
read length 80	*0*	*1*

The next page shows a diagram of the way the files are laid out in both approaches. Each read labeled with a different color (magenta for the read at offset 100, blue for the read at offset 12250, etc.). The blocks are colored depending on which read brought them into the buffer cache. Once in the cache, a block need not be read again for future requests. The FCB, colored red, was put in the buffer cache when opening the file.

In both cases, to satisfy the read at offset 100, only the first block of the file need be read in. For the second read, the linked case needed to follow the link from the first block to the third, in which 12250 was located, necessitating 3 reads, one of which (for the first block) was already in the buffer cache. In both cases, the request went over a block boundary, so the fourth block needed to be read as well. For the third request, the linked case needed to iterate to the end of the file. For UNIX, an index block needed to be read in first, and then the actual block. Finally, for the fourth request, no new reads were needed for the linked case since they were already in the buffer cache, whereas one was needed for the UNIX case.

This problem illustrates well the advantage of UNIX for "random access" reads over the linked strategy.


LINKED
FCB
0
100
4092
8184
12250
12276
16368
22010
32736
49104
49300
UNIX
FCB
0
100
4096
8192
etc.
12250
12288
16384
22010
32768
49152
49300

5. (File System Interface, 10 points)

   (a) (3 points) What is the *setuid* bit used in UNIX?

   Answer:

   When set for an executable program, this bit indicates that the process should be run with the permissions of the file owner, rather than those of the user who started the execution. For directories, it indicates that new files created within that directory should be owned by the owner of the directory, rather than the creator of the file.

   (b) (7 points) *Links* can be used to create a file system namespace as a directed acyclic graph, meaning that files or directories can be shared. What is the different between a *hard link* and a *symbolic link* in UNIX? Why does UNIX forbid hard links to directories?

   Answer:

   A hard link is a duplicate pointer to the same file contents. A reference count on the file contents is used to determine when it is safe to delete the file (only when the last owner deletes the file, and the count goes to 0*, are its contents recycled). A symbolic link is simply a file path (a string) that is followed to the actual file automatically when (most) operations are performed on the link. Deleting the link has no effect on the file's contents, and deleting the file has no effect on the link. Symbolic links can cross partitions, and be to directories, whereas hard links cannot.*

   A hard link to a directory is disallowed because it could create cycles in the directory graph. That is, we could cause directory `a` *to contain a link that points to* `b`*, which itself contains a link that points to* `a` *(meaning the reference count on each is at least* 1*). If all other links to these two directories were deleted, the filesystem wouldn't be able to recycle them, even though they could no longer be accessed, because the reference count is greater than* 0*.*