CSCI 4320 (Principles of Operating Systems), Fall 2005

Homework 3

Assigned: November 4, 2005.

Due: November 14, 2005, at 5pm.

Credit: 30 points.

1 Reading

Be sure you have read chapters 3 and 4 (through 4.3).

2 Problems

Answer the following questions. You may write out your answers by hand or using a word processor or other program, but please submit hard copy, either in class or in my mailbox in the department office.

1. (5 points) Suppose you are designing an electronic funds transfer system, in which there will be many identical processes that work as follows: Each process accepts as input an amount of money to transfer, the account to be credited, and the account to be debited. It then locks both accounts (one at a time), transfers the money, and releases the locks when done. Many of these processes could be running at the same time, and it must be possible to do multiple transfers at the same time, provided no two concurrent transfers affect the same account(s). The available locking mechanism is fairly primitive: It acquires locks one at a time, and there is no provision for testing a lock to find out whether it is available (you must simply attempt to acquire it, and wait if it's not available). A friend proposes a simple scheme for locking the accounts: First lock the account to be credited; then lock the account to be debited. Can this scheme lead to deadlock?

   If you think it cannot, briefly explain why not. If you think it can, first give an example of a possible deadlock situation, and then design a scheme that avoids deadlocks, keeping to the constraints in the problem description.

2. (5 points) Consider a computer system that uses variable partitions and swapping to manage memory (as described in chapter 4 pages 197–198); suppose it has 10,000 bytes of main memory, currently allocated as shown by the following table. (In this problem, amounts and addresses are in decimal, and unrealistically small, to keep the arithmetic simple.)

Locations	Contents
7500 – 9999	free
4500 – 7499	process B
4000 – 4499	free
3000 – 3999	process A
2000 – 2999	free
0 – 1999	operating system

Answer the following questions:

(a) Suppose we now need to allocate space first for process C, which needs 400 bytes, and then for process D, which needs 200 bytes. At what addresses (locations) will memory be allocated for these two processes if we use

i. a best-fit strategy for allocating memory?

ii. a worst-fit strategy for allocating memory?

iii. a first-fit strategy for allocating memory?

(b) Suppose the MMU for this system uses the simple base register / limit register scheme described in chapter 1 of the textbook (pages 26–27).

i. What value would need to be loaded into the base register if we performed a context switch to restart process B?

ii. What memory locations would correspond to the following virtual (program) addresses in process B?

- 100
- 4000

3. (5 points) Now consider a computer system using paging to manage memory; suppose it has 64K (2^{16}) bytes of memory and a page size of 4K bytes, and suppose the page table for some process (call it process A) looks like the following.

Page number	Present/absent bit	Page frame number
0	1	5
1	1	4
2	1	2
3	0	?
4	0	?
5	1	7
6	0	?
...	0	?
15	0	?

Answer the following questions about this system.

(a) If the only processes in main memory were process A and an operating system using page frame 0, how many free page frames would there be?

(b) How many bits are required to represent a physical address (memory location) on this system? If each process has a maximum address space of 64K bytes, how many bits are required to represent a virtual (program) address?

(c) What memory locations would correspond to the following virtual (program) addresses for process A? (Here, the addresses will be given in hexadecimal, i.e., base 16, to make the needed calculations simpler. Your answers should also be in hexadecimal. Notice that if you find yourself converting between decimal and hexadecimal, *you are doing the problem the hard way.* Stop and think whether there is an easier way.)

- 0x1420
- 0x2ff0

- 0x4008
- 0x0010

(d) If we want to guarantee that this system could support 16 concurrent processes and give each an address space of 64K bytes, how much disk space would be required for storing out-of-memory pages? Justify your answer. (If your answer depends on making additional assumptions, state what they are — e.g., you might assume that the operating system will always use the first page frame of memory and will never be paged out.)

4. (5 points) Now consider a much bigger computer system, one in which addresses (both physical and virtual) are 32 bits and the system has 2^{32} bytes of memory. (You can express your answers in terms of powers of 2, if that is convenient.)

   (a) What is the maximum size in bytes of a process's address space on this system?

   (b) Is there a logical limit to how much main memory this system can make use of? That is, could we buy and install as much more memory as we like, assuming no hardware constraints? (Assume that the sizes of physical and virtual addresses don't change.)

   (c) If page size is 4K (2^{12}) and each page table entry consists of a page frame number and four additional bits (present/absent, referenced, modified, and read-only), how much space is required for each process's page table? If we allow a maximum of 64 (2^6) processes, how much space in total is required for all their page tables? (You should express the size of each page table entry in bytes, not bits, assuming 8 bits per byte and rounding up if necessary.)

   (d) Suppose instead we use a single inverted page table (as described in chapter 4, pages 213–214), in which each entry consists of a page number, a process ID, and four additional bits (free/in-use, referenced, modified, and read-only), and we allow at most 64 processes, how much space is needed for this inverted page table? (You should express the size of each page table entry in bytes, not bits, assuming 8 bits per byte and rounding up if necessary.)

3 Programming Problems

Do the following programming problems. You will end up with at least one code file per problem, plus some text. Turn in your code by sending mail to `bmassing@cs.trinity.edu`, with each of your code files as an attachment. You can turn in the text by e-mail as well, or in hardcopy. You can develop your programs on any system that provides the needed functionality, but I will test them on one of the department's Fedora Core 4 Linux machines, so you should probably make sure they work in that environment before turning them in.

1. (10 points) Write a program or programs to demonstrate that how long it takes to access all elements of a large data structure can depend on whether they're accessed in contiguous order (i.e., one after another in the order in which they're stored in memory, or in some other order — see the minute essay for November 4). Turn in your program(s) and output showing differences in execution time. (It's probably simplest to just put this output in a text file and send that together with your source code file(s).) I'd prefer programs in C, C++, or Java, but anything that can be compiled and executed on one of the FC4 lab machines is fine (as long as you tell me how to compile and execute what you turn in, if it's not C/C++ or Java). You don't have to develop and run your programs on one of the FC4 lab machines,

but if you don't, (1) tell me what system you used instead, and (2) be sure your programs at least compile and run on one of the lab machines, even if they don't necessarily give the same timing results as on the system you used.

Possibly helpful hints:

- An easy way to measure how long program `mypgm` takes on a Linux system is to run it by typing `time mypgm`. Another way is to run it with `/usr/bin/time mypgm`. (This gives more/different information — try it.)
- If your program takes the approach suggested by the minute essay — accessing elements of a 2D array in row order versus column order — keep in mind the following:

  To the best of my knowledge, C and C++ allocate local variables on the stack, which may be limited in size. Dynamically allocated variables (i.e., those allocated with `malloc` or `new`) aren't subject to this limit.

  Dynamic allocation of 2D arrays in C is full of pitfalls. It may be easier to just allocate a 1D array and fake accessing it as a 2D array (e.g., the element in `x[i][j]`, if `x` is a 2D array, is at offset `i*ncols+j`).