Memory
OS Lab 6

Make sure to review Lab 5 before doing this lab.

Flexible operating systems vs specific operating systems. In other words, should an OS handle many memory models or just one. If it handles many memory models, then the OS is viewed to be flexible. Flexible operating systems often require a lot of system resources, like RAM and special CPU registers and larger space on disk for a backing store (swap space). One of the initial decisions an OS manufacturer must make is in how much of RAM should be reserved for the operating system. This impacts what the user can do since there is less available memory.

When the computer boots, RAM is automatically in the flat memory model. This makes sense since the hardware is created flat by the engineer. Flat means that RAM is one contiguous space from address 0 to address n-1, indexed by the address like a one-dimensional array. There is no hardware present in RAM to form another model. At boot the OS kernel is loaded into RAM under the default flat memory model. It is now the job of the kernel to decide on how to treat the remaining memory: fixed, paged, segmented, virtual, or hybrid.

QUESTION 1: Loading Programs in the Flat Memory Model

For this discussion assume a RAM (addresses 0 – n-1) has the OS loaded into a portion of RAM (say addresses 0 to p, where p<n-1). Assume further that addresses p+1 to n-1 is space available for users. Assume this OS is a multi-processing operating system, which means that more than one process can be loaded into RAM and executed concurrently. Assume there is only one core. So, concurrently means time-sharing with quanta.

How does the OS manage multiple processes using the Flat Memory Model?

There is no built-in way to handle big processes when using the Flat Memory Model. This means the entire process must be loaded into RAM. We don't use a backing store. If there is not enough RAM then the program cannot load into RAM, it cannot run. Processes that can fit into RAM, after time passes, gradually terminate. Each terminated process leaves a hole in RAM. For example, when the computer first boots there is nothing in RAM except the operating system, therefore RAM = OS + one-big-hole (RAM= OS[0 to p] + FREE[p+1 to n-1]). Over time, the user runs and quits programs, which results in a RAM that might look like this, RAM = OS + p1 + hole_a + p2 + p3 + hole_b + p4 + remainder-of-initial-one-big-hole. The little holes, hole_a and hole_b, are locations where a process had terminated.

When the user launches a new program, given the above, the key optimization problems become: (a) finding an optimal hole for the given process when launching, and (b) reducing the number of holes through merging (aka. "defragging").

Try the following problems:

1. Assume the flat memory model. Assume RAM goes from address 0 to 199. Assume the OS uses addresses 0 to 99. The remaining memory, 100 to 199, is for user processes. Which of the following loading strategies is better with respect to Big Oh: (a) insert into biggest available hole,

(b) insert into best fitting hole, or (c) inserting into the first hole that is big enough? Try each strategy on the following sequence of events to help you determine optimality: at time t0 launch process p1 with size 50 bytes, at t1 launch p2 with 20 bytes, at t2 launch p3 with 20 bytes, at t3 p2 terminates, at t4 p3 and p4 launch each 10 bytes, at t5 p1 terminates, at t6 p5 launches with 20 bytes, at t7 p6 launches with 20 bytes.

2. Assume problem 1, from above, but consider hole management. Which hole management strategy is the best with respect to Big Oh **and** useful degradation (can we fit stuff into it) of RAM: (a) do nothing, (b) merge adjacent holes (if two or more holes happen to be adjacent to one other then, redefine them as a single bigger hole), (c) pause process execution after n processes have terminated and then move all the processes to new locations in RAM such that all the little holes disappear and is replaced by a single big hole ("defragging"). For (a) (b) and (C) use the same sequence of events from problem 1 to help you think through this problem.

Discuss the above two cases with your friends, the TA or try to solve them alone.

QUESTION 2: Loading Programs in the Paged Memory Model

At boot the RAM is in the default flat memory mode. The kernel is loaded into RAM at addresses 0 to p in flat mode. Then the OS takes over and creates data structures that effectively chop the remaining parts of RAM, p+1 to n-1, which is the user's space, into equal sized pieces of m bytes, called frames. If P+1 to n-1 equals 100 bytes and m = 5 bytes, then the user's space is divided into 20 equal sized frames. The kernel OS space is still in flat mode, but the remaining flat memory is treated by the kernel as if it is paged.

The hard disk needs to be partitioned to hold the backing store (or swap space). If the frame size m is 5 bytes, then, when the user launches a program, the OS chops that program in multiple 5-byte pages storing each page in the backing store. The program is then considered by the OS to be executing (even though it is still on disk in the backing store). If the program is 20 bytes long and the frame size m is 5 bytes, then the process has 4 pages in the backing store.

For paged implementations, the RAM plus the backing store is considered to be process memory, so the entire program does not need to be loaded into RAM (unlike the flat memory model). Only the parts of the algorithm that are currently being executed need to be in RAM. The remaining parts of the algorithm could be in the backing store. This means that the OS needs to support a memory swapping routine. When we run out of algorithm to execute in RAM we need to fetch the next page from the backing store and load it into RAM. We need to find a free frame. If there are no free frames, we need to select a victim and swap out the victim and swap in the page we need.

Try the following problems:

1. Assume the paged memory model. Assume RAM is divided into 5 frames: f0, f1, f2, f3, and f4. Assume two processes are running, p1 and p2. Assume p1 has 3 pages and p2 has 4 pages in total. Assume p1's pages are page0, page1, and page2. Assume p1's pages are in RAM f0=page0, f1=page1, and f2=page2. In other words, all the code from p1 is in RAM. Assume p2 has its first page in f3 and the other pages are in the backing store. What should the OS do if P2

has requested to load two more of its pages from the backing store into RAM? Consider the following:

a. Where should the pages be loaded? In other words, is one of the page-loads a simpler case than the other? In what way is It simpler?
b. Is there an optimal way of selecting a victim (discuss with your classmates or Lab TA)? Remember the following: a victim is removed from a frame by overwriting it. If a victim is dirty, then we need to save the frame to the backing store. If we select a victim that happens to be needed by another process, then at the next quanta a request will be made to reload the page we just threw out.

2. Create a text file called `backingstore.txt` and a C program called `loader.c`.

Populate the text file with:

aaaaaaaaaabbbbbbbbbbccccccccccddddddddddeeeeeeeeeeffffffffffgggggggggghhhhhhhhhhiiiiiii
iiijjjjjjjjjjkkkkkkkkkkllllllllllmmmmmmmmmmnnnnnnnnnnoooooooooopppppppppppqqqqqqqqqq
rrrrrrrrrrssssssssssttttttttttuuuuuuuuuuvvvvvvvvvvwwwwwwwwwwxxxxxxxxxxyyyyyyyyyyzzzzzz
zzzz

The above is a single line of text without carriage returns or any other white characters. It is the entire alphabet with each letter appearing 10 times. We will assume that each sequence of 10 characters is a page in the backing store, and that this entire list of characters represents a single executing process. The process has 26 pages.

In the C program:

In a loop, ask the user to input an integer number between 0 and 25. For example, the user inputs the number 3. Using `fseek` and `fread`, jump to the fourth page, read the page's 10 characters and display that on the screen. In our example that would have been "`ddddddddddd`". Repeat this 5 times, asking the user for a page number each time. After that the program terminates.

Doing this properly means that you fopened the text file only once and fclosed it only once. In the loop you used `fseek` and `fread` each time.

If you want a challenge: after reading the page, make it dirty by changing one character. Ask the user for the character and the location where the single character change occurs in the page. Then write that page back into the text file. To be clear: ask user for page, read page, make the read in page dirty, write dirty page back into file overwriting the previous page. After your program has terminated, open the text file to see if the write was successful. Do this in the loop so that 5 dirty writes occur.

Example:

```
p = fopen("backingstore.txt","r+wt");

scanf("%d", &n);

fseek(p, n*10, SEEK_SET);
fread(buffer, sizeof(char), 10, p);
```

PROBLEMS FROM TEXTBOOK

Try the following textbook end of chapter problems:

Chapter 3: 4, 5, 7, 11, 16

YOU HAVE FINISHED THE LAB