McGill University

COMP-310 – Operating Systems – ECSE-427

Assignment #3

Due: November 17, 2008 at 23:55 on Web CT

About Processes and Memory

QUESTION 1: Virtual Memory

This question builds on the program you created in Assignment #2. Most of the functionality you created will be carried over to this assignment. What will change is how memory is implemented. We will also make changes to the RUN command.

In assignment #2 the RUN command launched a series of scripts in memory. It were left to your discretion on how you loaded these scripts into memory. In this assignment we will implement a virtual memory paging system and the memory manager that will support it. As we discussed in class, from the point of view of the programmer the process, in a virtual memory environment, has full access to all of the computer's virtual memory. To do that you will have to implement a 'backing store'. To make things a bit easier we will implement some simplifications.

FRAMES AND RAM

Assignment #2's main memory is currently implemented as an array of character pointers. We will keep this. Adding frames to it is not too hard since it is a logical constant/fixed division of RAM. To define the size of a frame you will make a global integer variable called FRAMESIZE. Its default value will be 4. This number represents the number of cells in the RAM array that constitutes a frame. Frames will follow the rule that they always start at cell 0. This means that the first frame includes cells 0, 1, 2 and 3. The next frame includes cells 4, 5, 6 and 7; and so on. Your RAM array must be a multiple of 4. Define you RAM to be 100 cells.

Frame sizes can be modified. In our case we will do this through the command-line prompt and argc/argv. When your OS is launched you can optionally supply the following command-line argument to change the default frame size.

-framesize n

Where: -framesize is the command-line switch and 'n' is the integer number representing the size of the frame. If the user inputs a frame size <=0 or >max RAM array size, the program ignores the switch and sets the frame size to its default size of 4. To make our programming easier we will add a restriction here; Frame sizes can only be multiples of 4. If the user asks for a size that is not a multiple of 4 then the program ignores the user's selection and uses the default frame size of 4.

THE BACKING STORE

All the scripts launched by RUN can now be of any size. We do not care if the script is larger than RAM since we have a backing store. But RUN needs to be modified to take advantage of the virtual memory system. For each scripts listed in the RUN command, RUN will copy them into the backing store and convert them into pages. Originally they were simple text files of arbitrary length, but now in the backing store, they will be pages of fixed frame size.

To simplify this process we will make some restrictions to our OS. The first restriction is that a command in a script cannot be longer than 99 characters. A script can be of any length – I.e. have as many commands in the text file as they like ignoring the size of RAM, but, each of these commands are restricted to a maximum of 99 characters. Given this restriction created our backing store pages will be easier.

The backing store will use the following data structure:

```
struct bs_rec
{
        char command1[100];
        char command2[100];
        char command3[100];
        char command4[100];
};
```

Above is a data structure that defines what a backing store page looks like. Even though our RAM array is defined as char *, permitting a variable length string, we will restrict this feature to 99 characters. The hundredth character is reserved for the '\0'.

In assignment #2 RUN loaded each script into RAM. Now, RUN will copy the script into the backing store using the following method:

For each script:

1. Create a PCB attached to a temporary pointer.
2. Load the script line-by-line into the page_rec structure.
3. fwrite the struct in write mode into a text file having the same name as the script file but with the extension .BS (for Backing Store). Normally this file would be saved in a hidden area called the backing store, but your backing store will be a subdirectory named BACKINGSTORE. You can put this subdirectory in the current directory.
4. Repeat steps 2 & 3 for all the lines in the script. If the last page did not use all 4 arrays in the struct, make sure to initialize the unused arrays to '\0'.
5. When done, create a field in the PCB called FILE *BACKINGSTORE that will point to that .BS file. Created another field in the PCB called long int MAXPAGES. Save the the total number of created pages for that .BS file.
6. Insert the PCB into the Loading queue.

THE PAGING SYSTEM

To implement paging we will need a paging data structure:

```
struct page_rec
{
        int inRAM; // boolean true if page in RAM, false if page in backing store
        int wasModified; // will not be used in this assignment
        int baseAddress; // the index number of the RAM array cell of the frame
};
```

The page_rec structure will be defined as a field within the PCB. To make things easy for us, this field will be an array of fixed length:

```
struct page_rec pageTable[100]; // max limit of 100 pages
```

Note that the pageTable index IS the page number. Stored within the cell referenced by the page number is the actual frame baseAddress. This too is an array index number but in this case it refers to the RAM array.

Optionally, if you want to be more professional, you can implement this as a pointer and malloc the array to the size specified by the MAXPAGES variable. There will be no bonus points for doing this – there will only be glory!

READY QUEUE, PCBs, PAGES AND FRAMES

If you have not noticed already the paging architecture is implemented through the PCB. Each PCB has its own page table. The OS is aware of the process and hence its page table through the ready queue. Since the OS obeys, through code, the frame size variable, then everything should be fine.

MEMORY MANAGEMENT

Just because we copied everything into the backing store correctly formatted and have a PCB in the ready list with a properly constructed page table, tells us nothing about what has actually been loaded in RAM and how we should deal with RAM.

The memory manager will follow these rules:

1. As already stated, RUN copies the script into the backing store and inserts the new PCB into the tail of the Loading queue.
2. The memory manager extracts PCBs from the Loading queue, and..
    1. Loads only the first page of the script into RAM
    2. The page is loaded using the first fit strategy. If all of RAM is full then the manager selects a victim and removes the page from RAM. Since we are not

implementing write-backs we only need to updated the victim's page table and overwrite the frame with the new commands. The victim will be the PCB at the tail of the ready list. If that PCB has NO pages in RAM we go to the second to last PCB, and so on until we find a PCB with a page in RAM. We will overwrite the oldest cell, which should be cell 0. If that page is already not in RAM, we will then overwrite the next oldest which is cell 1, and so on until we fine one. We only overwrite one page since we only need one page.

3. The new PCB is removed from the Loading queue and is added to the tail of the Ready queue.

THE MEMORY MANAGER AND RUN-TIME

As your script executes it will come to a time when it has executed all 4 cells of its frame. When that has happened the PCB page table needs to be consulted for the subsequent page. If that page is already in memory and there is still quanta then the CPU pointer is updated to the cell of the next page and things continue as normally. If the next page is not in memory then the PCB looses the CPU and quanta, the memory manager is called to load the missing page into RAM using the rules stated previously (in MEMORY MANAGER), after that the PCB is put back at the tail of the ready list.

When your process terminates, the PCB is deleted and so is the script in the backing store.

Optionally, for the glory again (not points), you could implement an interrupt feature. When a process no longer has pages in RAM it looses the CPU and its quanta, but it is not processed automatically as described above. Instead it is put into a Waiting queue. Then, part of the kernel overhead, the memory manager will look at the Waiting queue, selecting someone (FIFO or Priority) to be processed and reinserted into the Ready queue as described above.

Have fun with this one. This OS is beginning to look real!

QUESTION 2: Interrupt Handling Problem

When an OS needs to interrupt an executing process the process looses control of the CPU. In addition to that, all the processes waiting for the CPU cannot access the CPU because the OS is now executing on the CPU. This is called overhead. It is important to the OS developer to reduce the overhead time so that the processes experiences maximal CPU usage (minimal processing interruptions).

Consider a producer/consumer situation where the producer must never be blocked. In other words, when the producer has data, that data must be processed to completion. If the producer does not have data then it does not need to execute. Producers often write their data to some type of buffer. Since our producer cannot be blocked we do not know how much data will be sent to the buffer. One solution would be to implement the buffer as a linked list. To avoid the overhead of implementing the buffer as a linked list, the

following scheme is used. The buffer is implemented as a fixed size array, A[n], where n is large enough *most of the time*. In the case where the producer finds the buffer full, the array size n is temporarily extended to accommodate the spike in the production by mallocing a bigger array and copying the contents of the previous array into the bigger array and then freeing the smaller array. The extension is removed when the number of elements falls again below n (to preserve memory). Compare the effectiveness of this scheme with a linked list implementation, given the following elements:

- TL is the time to perform one insert or remove operation in a linked list implementation
- TA is the time to perform one insert or remove operating in the proposed array implementation
- OH is the overhead time to temporarily extend the array
- P is the probability that any given insert operation will overrun the normal array size n.

Answer the following:

A) Derive a formula for computing the value of P, below which the proposed scheme will outperform the linked list implementation
B) What is the value of P when TL = 10 * TA and OH = 100 * TA?

QUESTION 3: OS Data Structures

Assume at time 5000 there are four processes, p1 through p4, waiting for a timeout signal. The four processes are scheduled to wake up at time 520, 645, 695, 710 respectively.

A) Assuming an implementation using a priority queue with time differences, show the queue and the contents of the countdown timer at time 500.
B) After p1 wakes up, it immediately issues another call to timer() for 70 units of time. Assuming that processing the call takes no time, show the priority queue after the call.
C) Assuming p1 issues the same call again (timer(70)) immediately after it wakes up for the second time. Show the new priority queue.

QUESTION 4: Consider a banking system with many different accounts. Processes may transfer money between any two accounts, Ai and Aj, by executing the following instruction:

Lock Ai; Lock Aj; Update Ai; Update Aj; Unlock Ai; Unlock Aj;

A) Show how a deadlock can occur in such a system.
B) How can the ordered resource policy be implemented to prevent deadlock if the set of possible accounts is not known a priori or changes dynamically?

WHAT TO HAND IN

On Web CT in electronic form:

- A single WORD, or PDF document answering the non-programming questions.
- The source files for your OS and its make file.
- Two ASCII programs to run on your OS
- Also include a text file describing what operating system your program was developed under. It would be good if you tested your program on our machines on the third floor of Trottier.
- Please provide comments in your code so that it will be easier to grade.

HOW IT WILL BE GRADED

This assignments is worth 20 points, distributed as follows:

- Question 1: Total of 13 points
  - Frames in RAM 1 points
  - Backing Store 3 point
  - Paging System 3 points
  - Run-time management 3 point
  - Memory management 3 points
- Question 2: 2 points (one point for each sub question)
- Question 3: 3 points (one point for each sub question)
- Question 4: 2 points (one point for each sub question)