Course: CIS-452 Assignment: Program Three Design Documentation

Author: Joshua Engelsma
Instructor: Dr. Greg Wolffe

Date: 4/16/2015

System Overview:

The main goal of this project is to simulate the effects of virtual memory, using LRU as a replacement algorithm. The program acts by reading in a reference line from a data file. This line contains a process, and a page number that it is trying to access. A processes logical address space is 64k, and each page must be mapped to the 16k of physical memory. When the is no room left to map in physical memory, the page in memory that was least recently used is removed from its frame, in order to make room for another page to get in.

System Commands:

The User Interface that I built allows for several basic commands. They are as follows: *Run To Next Reference* - This command allows the user to read in one reference line from the file at a time. *Run To Completion* - This button runs through all the lines of the file at once, and reports total statistics at the end of the run. *Run To Next Fault* - Feature lets you run through page references in the file until you have reached a page fault at which point it stops. *Reset* - Starts the simulator over again from the beginning of the file. *Quit* - Command to quit the program

Implementation:

In order to implement virtual memory using lru, I had to make use of several very important structures.

The first important structure/object to mention is that of a PCB. This custom object is made for every process when they make their first page reference. The object contains the logical address space size of a process, the number of references made to pages by that process, the number of page faults that process has endured, and the page table of the process. In order to track the logical address space size, the number of faults, and the number of references made to the process, I can update these fields appropriately every time I reference a page (for page references), have a page fault (for total faults), and reference a larger page than the current logical address space size (for logical address space size). I stored all of the PCBs in a python dictionary where the key was the pid of the process, and the value was the PCB object.

As mentioned, the PCB object also contains the page table of the process. I implemented the page table as a python dictionary where the key is the page of the process, and the value is a list of the frame that page resides in physical memory, a reference time stamp (needed for LRU algorithm), and resident bit that lets us know if the page is currently in physical memory.

The last two important structures to talk about are physical memory, and the free frame list. Physical memory is a list where each index contains a tuple with a process id, and page number. The free list is a list of indices that do not currently hold any content in physical memory.

Now that the various structure and types have been discussed, I can easier walk through how the program is algorithmically implemented. The first step of course is to read in a line from the file. At this point, I first grab out the pid of the reference. Then I convert the binary page number to a decimal page number. The next question I ask is, is the pid of this page reference in my dictionary of PCBs. If its not, I have a page fault because if the process was never created, obviously this page was never referenced. In this situation, I first need to create a PCB for the entering process. I then increment its number of page faults, memory references, and adjust its logical address size based on the page number read in. The next step would involve updating the page table, however, in order to update the page table, I need to have the frame the page will be stored in physical memory. To get this frame, I just grab the first free frame available on the free list. If there is a free frame this is just an easy pop a frame off the free list, however, if the free frame list is empty, then I have to remove a page from physical memory. This is done using the page least recently used. Because I keep a timestamp of the latest reference time of each page in the page table, I can loop through my PCB structure and grab all the page tables. Then I can look at all that pages in the page tables and determine which one (of the ones where resident bit is a 1 i.e. in memory), has the reference time longest ago. This page is the page that was least recently used. As such, I can grab the frame that this page resides in physical memory, remove it from physical memory, and add the free frame to the free frame list. Now I have a free frame to give my process who can finally update his page table with a new page at the page number referenced with the frame the page resides at, the reference time, and a resident bit of 1. Finally, I can add this PCB record to the dictionary of PCBs.

After checking if the PCB record already exists, the next question is, does the page referenced for this pid exist in the page table of the PCB. If it does not, then we have a page fault situation, and the same logic as discussed above is applied to update the existing PCB record, and update the page table by adding the new page which will get a frame number to map to using the free frame list (which may have to grab a frame using LRU).

In the scenario where the PCB record already exists and the page referenced already exists, the logic is very similar to that described above, however, we don't need to create a new PCB record or a new page. Instead, we grab the current PCB record using the pid of the reference line in the file. Immediately the number of memory references can be incremented, and the logical address space can be updated if the page is larger than the current address space. Next, we need to determine if the page referenced is already in memory. This is a simple check. We simply grab the page table from the PCB record. Then we check the page table where the key is the page number of the reference line. If the resident bit is a 1, then this page is in memory already, no page fault occurs, and so we just need to update the reference time of this page in the page table. If the resident bit is a 0, then we know that the page is not in memory at the moment, and a page fault occurs. Again the same logic as discussed above for page fault scenarios applies. A free frame is granted from the free list - which may have to use LRU (found using timestamps of all resident pages) to remove a page from full memory, and grant it to the process. The process, then updates its number of references, faults, and logical address space size in the PCB. Finally, it updates its page table at the page referenced to contain the list of the frame (given by the free frame list using LRU) that page maps to in physical memory, the reference time, and a resident bit value of 1.

In this manner, I simulate virtual memory using the LRU algorithm. Using the structures of PCB (and various fields associated), PCB dictionary, Page Table dictionary, free frame list, physical memory frames list, reference datetimes to implement LRU, and the three scenarios of PCB does not exist, page does not exist in page table, and page does exist, so determine if its in memory or not before faulting, I have implemented full functionality.

Quick UI Overview:

The UI is implemented using the MVC paradigm. I have encapsulated all of the manager logic in a “manager” object. This object contains public methods that I can call which will run the manager to the next line, or run to the next fault and so on. The manager object contains all fields such as PCB records, page tables etc. After calling the method to update my model, I can grab model data from these fields, and update my UI accordingly.