CIS657 Principles of Operating Systems

Zhen Ren

1 Homework3

Ex. 3.1 — (10pts) The CPU (most of them) assumes that programs are running from address 0 and the address space of the programs are contiguous. MMU enable the CPU make such an assumption. Consider a memory management for a system that only allows one application program to be loaded in the main memory and run. This means that at any given time, only the OS and one application program are loaded in the main memory. If we want to support the CPU's assumption without an MMU, how would you load the program and OS in the main memory? Explain your idea by drawing the memory map.

If a system only one user program to run at a time. In this case, we have only two programs loaded in the main memory at most:the OS and the user program. So the computer is similar to the following table. The following table is the memory map.

	Available Memory
	A User Program
0x80-1 0000	Operating System

We will first load the OS into the main memory. And then we will load the user Program into the main memory, as the upper map. However, without MMU we can not keep protection from accessing OS.

Ex. 3.2 — (15 pts) Memory hierarchy is given by the speed of memory devices and the size of the devices along the hierarchy. What if we build a giant memory device made of the fastest materials available that is as big as some terabytes? Would that eliminate the need for having intermediate memory devices as in memory hierarchy? Why or why not?

No.

1. Performance problem. While the memory space is giant, the access time of the memory increases significantly, like finding a book in a large library. We still to set up a memory hierarchy to find the balance of access time and the space. With the help of memory hierarchy, we achieve the balance of access time and space.
The memory hierarchy includes three types: cache, main memory, and disk storage. For the disk storage, we can store much more than some terabytes. So the fastest materials can not replace the disk storage in the computer. And with the help of memory hierarchy we could reach a balance of access time and the space. We could access lots of bytes within reasonable time.
Then there are also Principle of Locality: Temporal locality and Spatial locality. And with the help of localities, we could
Temporal locality has features: The program instruction that was executed recently by the CPU would most likely be executed again
Spatial locality has features: The program instructions that are close to the recently executed instructions in source code would mostly likely be executed soon.

2. Cost problem. Even the speed of a memory device depends on both the device's hardware characteristics and the size. And fastest material will improve hardware. But the growing size will take a long time to accessing the exact element out of a larger number of elements. And the cost of fastest materials for memory must more expensive than the slower materials. If all memories are made up with fastest materials, the cost expands to huge number that people can't afford it. So the intermediate memory memory can not be replaced by the fastest materials.

The main purpose of hierarchical memory is to narrow the speed gap between two hierarchies. So in order to deal with meet the requirements of large, fast, not volatile memory. We need a memory hierarchy: including small amounts of fast, expensive memory- cache, some medium-speed, medium price main memory, gigabytes of slow, cheap disk storage.

Ex. 3.3 — (10 pts) What does OS have to do when all page frames are being used and there is at least one page to be brought in from hard disk to the main memory?

If there is no unused page frames available in the main memory, we need to find a victim page

to be replaced. And the method to choose the victim page is the page that would be least likely to be used in the near future among all the pages in the main memory. And there are some page replacement algorithms: Optimal, NRU, FIFO, second, clock, LRU, NFU, aging, WSClock
And it has the following steps:
1.Hardware traps to kernel
2.General registers saved
3.OS determines which virtual page needed
4.OS checks validity of address, seeks page frame
5.If selected frame is dirty, write it to disk
6.OS schedules disk read to bring the new page in from disk; schedule another process
7.Disk interrupt; Page tables updated
8.Faulting instruction backed up to where it began
9.Faulting process scheduled
10.Registers restored
11.Program continues

Ex. 3.4 — (10 pts) Why would you use NFU over LRU?

Because LRU is clearly expensive because for each memory reference, OS or a designated hardware component must update information about page usages. NFU is an algorithm that approximate it. NFU uses a reference bit, can increment the counter less often and updates the linked list less frequently(NFU using a linked list). If a page is referenced since the last clock interrupt, at the current clock interrupt the counter is incremented by 1. The detailed NFU and LRU algorithm is put in the answer to Ex.3.8

Ex. 3.5 — (10 pts) Why would you use NFU with aging over NFU or LRU?

LRU is very expensive in the page replacement algorithm. NFU is an approximation algorithm and reduce the cost. However both LRU and NFU never forget anything, which means that when a page first used a lot, but not used frequently in the future will stay in the memory for a long time, even this page is not used anymore.
NFU with aging can solve this problem by recording used age of a page. Thus this algorithm could select the not frequently used page, even it first used frequently, as victim page. As a result, using NFU with aging will be over NFU or LRU.

Ex. 3.6 — (10 pts) In what purpose the two-handed clock algorithm useful?

When the pages used recently will be used again soon, and we can throw out page that has been

unused for longest time. So this situation is the reason of using two-handed clock algorithm. Because two-handed clock algorithm tries to leave a certain number of page frames available at all times so that when a page fault happens, there is no need to call a page replacement algorithm reactively, instead of NFU and LRU addressing a page fault as it happens when a victim page is needed
And if the angle is really wide, the second hand would take longer to point to a page after the first hand. This means there is more time for the pages to get their R bits set. If two-hands are overlapping (the angle is zero), then every page is evicted.

Ex. 3.7 — (10 pts) Why do you have to back up the instruction after page fault handling? And which instruction is that?

When a page fault occurs, the process cannot continue to run until the desired page is brought in from the hard disk. Notice that in this case, a page fault caused a hard disk I/O. The process caused a page fault and consequently it needs to be blocked and wait for the I/O operation to be completed (which means the desired page is now in the memory). After the I/O is completed, an interrupt is raised to notify the CPU that the I/O for the page fault has been completed. The CPU then runs the page fault I/O exception handler, in which the current processes's program counter is one step backed up to attempt to rerun the instruction that caused the page fault. The process is now waken up and inserted back in the ready queue. When the scheduler picks the process, the process will start running from the instruction that caused the page fault.
Because a page fault causes an I/O but the programmer of the process didn't intend the I/O, we call this an involuntary I/O. A page fault causes an involuntary I/O to the process that caused it!

The instruction is that causes the page fault.

Ex. 3.8 — (25 pts, 5pts for each algorithm) Describe page replacement algorithms including FIFO, First-in Last-out (FILO), NRU, LRU, NFU?

The FIFO (First-in First-Out) algorithm keeps track of the order of the pages as they are brought into the memory. Then it replaces the oldest one in the memory. As an implementation, we can use a sorted linked-list to keep track of the order. This algorithm may work well in some cases, but when older pages are more frequently used, this algorithm will replace these frequently used pages only to bring them back to memory in the near future.

The FILO (First-in Last-Out) algorithm keeps track of the order of the pages as they are brought into the memory. Then it replaces the newest one in the memory. We can use a sorted linked-list which is similar to FIFO to keep track of the order. This algorithm may work well in some cases, but when older pages are less frequently used, this algorithm will be bad because it will store these less-frequently used pages a long time without replacing.

In NRU, two additional bits are used to keep track of each page's usage pattern. These two bits can be part of the page table entry for each page. The tow bits will be added at the end of the virtual and physical page number pairs.The two bits are called Referenced bit and Dirty bit. If a page was referenced since the last clock interrupt, the bit is set. When a clock interrupt occurs, the Reference bit is reset. A page's dirty bit is set when the page has been modified (by the CPU most likely). If a page is selected to be replaced and its dirty bit is set, we know the page must be written back to the hard disk for updating. If the dirty bit is not set, the page can be simply replaced by the incoming page from hard disk.

LRU keeps track of how many times each page is used. A couple of different implementations exists: one using a counter associated with each page table entry and the other using a linked list of page numbers that is updated for each page reference. LRU is clearly expensive because for each memory reference, OS or a designated hardware component must update information about page usages. When a page fault occurs and a victim page needs to be chosen, LRU will choose the page with the smallest counter value associated, if counters are used for implementation. If a linked list is used, the page that is at the end of the linked list will be replaced.

NFU is an approximate algorihtm to LRU. NFU uses a reference bit. If a page is referenced since the last clock interrupt, at the current clock interrupt the counter is incremented by 1. If the reference bit is clear, the counter for the page is not incremented. Notice that between two clock interrupts, a page can be referenced multiple times, but the counter will only be incremented by 1, therefore it is an approximation of LRU. The obvious advantage of NFU algorithm is its comparatively lighter overhead.