Operating systems Project #2: Multi-process execution with Virtual Memory (paging)

Introduction

In the previous project, we did some scheduling simulation; that enables multiple processes to run concurrently. Based upon that, we will add virtual memory to each process. Virtual memory is a huge notion that enables us to 1. develop program easily, 2. protect our program from the other-malicious- programs. With virtual memory, each process can access memory as it owns the entire memory (although it is not, in reality). Thus, software engineers do not have to worry about physical memory location of variables, code, stack, etc. Instead, it runs with its own address space, which is the set of address that a process can point to. To work with address space, we should take architectural support for virtual memory that is called paging and/or segmentation.

Paging is a concept that implements address translation between physical address and logical address (virtual address). For a process to run with an address space, program should be written with virtual address. That is, CPU directly sees virtual address. On the other hand, your computer should work with the actual location of data (physical address). OS maintains a table that maps virtual address onto physical address, so that CPU sees virtual address, but it accesses corresponding physical address.

Paging defines the concept around address space, with page. A page is a small subset of address space. To map a page to an actual memory location, we chop the entire main memory into small pieces, called page frames. Then, OS maps every page with page frame. That is, page is a unit of address mapping. In usual 32-bit computers, the size of pages is 4 kilo-bytes. That is, 4-KB-sized address space and memory pieces are mapped.

Each page frame is numbered with integer, and the number is called as page frame number. The number begins with 0 according to the physical address. (i.e. page frame number: 0, 1, 2, 3, … for physical memory address 0~0xFFF, 0x1000~0x1FFF, 0x2000~0x2FFF, 0x3000~0x3FFF, respectively). OS maintains a table that maps VA to PA, which is called page table.

CPU only sees virtual address, and MMU translates the virtual address into physical address so that CPU can get the actual value from physical memory location. MMU is a hardware that resides in a CPU; when CPU accesses memory, it translates address (from VA to PA), according to a page table that records a VA-to-PA mapping. A page table resides in memory. At runtime, MMU (inside CPU) has to remember the beginning address of page table. Then, it calculates the virtual address from page table index; and it accesses the page table to fetch the physical address (corresponding to the virtual address). Finally, it accesses physical address.

Note that address space is given per-process in multi-process environment. When scheduling is completed, scheduler restores the context of the next running process. At this time, process's page table information is also restored, so that MMU can point to the next process's page table.

due by November 30th (The last min. of Nov.)

Page table can be updated, at runtime. We can fill all the entries in page table at the process initialization. However, the actual utilization of address space is quite low, and the utilization could be vary among all the processes. Thus, memory address can be allocated at runtime. Namely, pages are allocated at runtime. Accordingly, page frames should be allocated at runtime. That is, virtual memory mapping can be updated at runtime. This is called demand paging.

To support demand paging, kernel has to prepare free pages list, at booting time (bootstrapping). When a process is created, then the corresponding page table (the page table for the process) is allocated; however, the table is initialized with empty mapping at the beginning. Therefore, all the page table entries are set to 'invalid'. Then, when a process accesses a memory (address), CPU tries to access PA by accessing the process page table. When the page table entry is invalid, page fault is occurred. The page fault is caught by the OS.

When OS catches page fault, OS stops the currently running process, until the proper mapping is made. To make a proper mapping, OS finds the free page frame. Then, OS updates the page table by filling in the page table entry with free page frame number. When a process completes, OS reclaims all the free pages that it has consumed. After that, OS resumes the process with the instruction that generates fault.

———————————————————————————————————————

In this programming assignment, you will simulate virtual memory mapping with paging, fully in software. You have to implement above-mentioned demand paging with the proper assumptions.

Since this program assignment could be one of your major take-outs from this course, so please work hard to complete the job/semester. If you need help, please ask for the help. (I am here for that specific purpose.) I, of course, welcome for any questions on the subject. Note for the one strict rule that do not copy code from any others. Deep discussion on the subject is okay (and encouraged), but same code (or semantics) will result in sad ending.

Extra implementation/analysis is highly encouraged. For example, you can implement two-level paging, or swapping with LRU algorithms. (Unique /creative approaches are more appreciated, even trial has significant credit.)

Lastly, an advice: Begin as early as possible. Ask for help as early as possible. Try as early as possible. They are for your happy-ending.

The followings are specific requirements for your program.

1. The objective: understand address translation with paging, efficient page table management using demand paging.
   A. Simulator has to work correctly. (should not break down, VA->PA translation has

to be properly performed by kernel, OS updates page table at runtime)
   B. One process (parent process) acts as kernel, the other 10 processes act as user process.
2. At OS initialization:
   A. At booting time, physical memory has to be fragmented in page size, and OS must maintain the free page frame list (or simply free page list).
   B. Total physical memory size can be assumed. PFN begins with zero.
3. At user process initialization:
   A. OS maintains a page table for each process. When OS creates a new process, OS allocates an empty-initialized page table.
4. At user process execution:
   A. When a process gets a time slice (tick), it should access some (10) memory addresses (pages).
      i. To simulate memory access, a user process sends ipc message that contains memory access request for 10 pages.
   B. Then, OS checks the page table.
      i. If the page table entry is valid, the PA is accessed.
      ii. If the page table entry is invalid, the OS takes a new free page frame from the free page list. Then, the page table is updated with page frame number.
   C. To check page table, we need to separate VM address into two parts: VM page number, and VM page offset.
      i. MMU points to the beginning address of page table.
      ii. VM page number is used as page table index.
      iii. At the the page table index, page table entry is located.
      iv. Inside the page table entry, page frame number, and FLAG are also stored.
5. Extra implementation (page with data)
   A. You can simulate paging with real data. We can assume that physical memory is filled with some numbers or patterns. When a user process accesses memory, then you can get the real data.
   B. Memory access is either read or write. You can simulate read/write. Note that write requires some data to write; and a read request returns with read data.
6. Extra implementation (Two-level paging)
   A. Actual page utilization is usually low; thus, most page table entries are filled with zero, for most of the time.
   B. To minimize page table size, we can reduce the page table structure size. Address bits are separated in first-level VM address, second-level VM address, and VM page offset.
   C. MMU then points to the first-level page table. First-level VM address is used as first-level page table index. Second-level VM address is used as second-level page table index.
   D. Second-level page table does not exist when the first-level entry is null. When first-level page fault occurs, OS then allocates a free page frame. Then, the first-level page table entry is filled with page frame number.
   E. Next, Second-level page table is accessed; then page fault occurs again. The page fault is caught by OS, Then OS takes another free page frame, so that it fills page

table with page frame number.

F. When process completes, it should clean up all the page tables, and frees the used memory.

7. Extra implementation (swapping)
   A. Because the capacity of physical memory is limited; and virtual memory address space is more than physical memory.
   B. When the OS lacks free page frames, it moves some page frames in main memory to secondary memory (or disk) to obtain free pages.
   C. To pick some pages to evict from main memory, we can use LRU algorithm. LRU selects the least frequently used pages. To select the least frequently used page, we need to use some counter for each page.
   D. When a page is selected, it is moved to disk. We assume a infinite disk size. Disk is assumed to be a big file. To distinguish active pages in main memory from passive pages in secondary memory, we use additional flag bit. To save multiple pages in disk, you need to save process id, page number along with the page data. For brevity, page data is assumed to be null.
   E. When a swapped-out page is accessed, OS stops the current process, and moves data from disk to main memory, refilling the page table. Page frame number can be different from previous one.
8. Output:
   A. At each time tick, access VA, PA, page fault event, page table changes, and read/write value can be logged.
9. Program completion/terminal condition:
   A. All simulation should be terminated when time tick is over 10,000.
10. Evaluation:
   A. Different credits are given for implementations, and demos.

Happy hacking!