Operating Systems Laboratory (CS39002)
Spring Semester 2018-2019

Assignment 5: Simulation of Virtual Memory using Demand Paging

Assignment given on: March 11, 2019

Assignment deadline: March 25, 2019, 2:15 PM

Problem Overview:

It is required to simulate a demand-paged virtual memory management system. In a typical operating system, processes arrive in the system and they are scheduled for execution by the scheduler. Once a process is allocated CPU, it starts execution whereby the CPU generates a sequence of virtual addresses. The corresponding sequence of virtual page numbers is referred to as the ***page reference string***. Once a virtual address (page number) is generated, it is given to the Memory Management Unit (MMU) to determine the corresponding physical address (frame number) in memory. In order to determine the frame number, it consults the ***translation lookaside buffer (TLB)***, *in the event of a miss, it consults the* ***page table***. If the page is present in the main memory the current process can continue its execution; otherwise, a ***page fault*** occurs and the requested page must be brought from disk to main memory (i.e. an I/O operation). During this time, the scheduler takes the CPU away from current process, puts it to waiting state, and schedules another process. Meanwhile the page-fault handling interrupt service routine **(ISR)** associated with the MMU performs the following activities to bring the requested page into main memory. First, it checks if there is any free frame in main memory. If so, it loads the page into this frame. Otherwise it identifies a ***victim page*** using Least Recently Used (LRU) page-replacement algorithm, removes the victim page from memory, and brings in the new page. Once the page-fault is handled, the process comes out of the waiting state and gets added to the ready queue so that the scheduler can select it again.

The whole program has to be structured into the following four modules:

a) **Master** is responsible for creating other modules as well initialization of the data structures.

b) **Scheduler** is responsible for scheduling various processes using FCFS algorithm.

c) **MMU** translates the page number to frame number and handles page-fault, and also manages the page table.

d) **Process** generates the page numbers from page reference string.

After all the **Process**es have finished execution, the **Scheduler** will notify **Master**. Master will terminate Scheduler, MMU and then will terminate itself.

The inputs to the program will be the following:

i) Total number of processes (k)

ii) Virtual address space – maximum number of pages required per process (m)

iii) Physical address space – total number of frames in maim memory (f)

iv) Size of the TLB (s)

1. The Master module:

The Master module performs the following tasks.

a) Creates and initializes different data structures as mentioned below:

   i) **Page Table** – There will be one page table for each process, where the size of each page table is same as the virtual address space. This is to be implemented using shared memory (SM1).

   ii) **Free Frame List** – This will contain a list of all free frames in main memory, and will be used by MMU. This isto be implemented using shared memory (SM2).

   iii) **Ready Queue** – This queue is used by the scheduler, and is to beimplemented using message queue (MQ1).

   iv) In order to communicate between different modules it creates two more message queues MQ2 (for communication between Scheduler and MMU), and MQ3 (for communication between Process-es and MMU).

b) Creates the **Scheduler** module as a separate Linux process for scheduling the Process-es from ready queue (MQ1). It passes the parameters (MQ1, MQ2) via command-line arguments during process creation.

c) Creates **MMU** module as a separate Linux process for translating page numbers to frame numbers and to handle page faults. It passes the parameters (MQ2, MQ3, SM1, SM2) via command line arguments during process creation.

d) Creates k number of **Process**-es as separate Linux processes at fixed interval of time (250 ms). **Master** generates the page reference string (R_i) for every process (P_i) and passes the same via command line argument during process creation. It also passes the parameters (MQ1, MQ3) to every process.

The **Master** module is to be implemented in Master.c or Master.cpp file. It will read 3 inputs (k, m, f) as mentioned above. For every process, it selects a random number between (1,m) and assigns it as the required number of pages of that process and allocates frames proportionately.

The page reference string will be generated as follows:

- For a process P_i, if total number of pages required is m_i as randomly selected by **Master**, then
  - o Length of reference string for P_i is >=2* m_i and <=10*m_i
  - o Select a random number (x) from this interval and that is the length of the reference string
- **Master** first creates and initializes all data structures, then creates **Scheduler** and next creates **MMU**. Then it creates all the ‘k’ **Process**-es.
- After performing all process creation tasks, **Master** will wait() till scheduler notifies completion of all the process execution. **Master** will terminate **Scheduler** and **MMU** first, and then terminates itself.

2. The Scheduler module

Scheduler is responsible for scheduling all the ‘k’ processes. It continuously scans the ready queue and selects processes in FCFS order for scheduling. Initially when the ready queue is empty, it waits on the ready queue (MQ1). Once a process gets added, it starts scheduling.

Scheduler selects the first process from ready queue, removes it from the queue and sends it a signal for starting execution. Then the scheduler blocks itself till gets notification message from MMU. It can receive two types of message from MMU:

- Type I message "***PAGE FAULT HANDLED***" – After successful page-fault handling

  After getting this signal, it enqueues the current process and schedules the first process from Ready queue.

- Type II message "***TERMINATED***" – After successful termination of process

  After getting this signal, it schedules the first process from Ready queue.

The **Scheduler** module is to be implemented in sched.c or sched.cpp file. **Master** executes the program via exec() with the proper arguments as explained in the implementation of **Master**. Once all the **Process**-es are executed, **Scheduler** informs the **Master** to terminate all the modules.

3. The Process modules:

Execution of process means generation of page numbers from reference string. Process sends the page number to MMU using message queue (MQ3) and receives the frame number from MMU.

- If MMU sends a valid frame number
  - It parses the next page number in the reference string and goes in loop
- Else, in case of page fault
  - It gets -1 as frame number from MMU and saves the current element of the reference string for continuing its execution when it is scheduled next and goes into wait(). MMU invokes page fault handling routine to handle the page fault. It may be noted that the current process is out of Ready queue and Scheduler enqueues it to the Ready queue once page fault is resolved.
- Else, in case of invalid page reference
  - It gets -2 as frame number from MMU and terminates itself. The MMU informs the scheduler to schedule the next process.

When the Process completes the scanning of the reference string, it sends -9 (marker to denote end of page reference string) to the MMU and MMU will notify the Scheduler (see MMU).

The **Process** module is to be implemented in process.c or process.cpp. Processes are created by **Master** with proper argument (as mentioned in Master section). The processes will put them in the Ready queue and pause themselves. Whenever the Scheduler schedules a process, only then it will come out of this pause state and will start execution. The process reads the reference string one by one and sends them to the MMU and receives the corresponding frame number (when available), -1 (when page faults occurs). When the process completes the scanning of the reference string, it sends -9 (to denote end of page reference string) to the MMU, and MMU will notify the Scheduler. Scheduler will terminate the process and removes from Ready queue.

4. **The Memory Management Unit (MMU) Process:**

   Master creates **MMU** and then it pauses. **MMU** wakes up after receiving the page number via message queue (MQ3) from **Process**. It receives the page number from the process and checks the **TLB**, failing which, it checks the page table for the corresponding process. There can be following two cases:

   i) If the page is already in page table, **MMU** sends back the corresponding frame number to the process.

   ii) Else in case of page fault

      - Sends -1 to the process as frame number
      - Invokes the page fault handler to handle the page fault
         - If free frame available - update the page table and also updates the corresponding free-frame list.
         - If no free frame available – do local page replacement. Select victim page using LRU, replace it and brings in a new frame and update the page table.
      - Intimate the Scheduler by sending Type I message to enqueue the current process and schedule the next process.

   The data transfer of the page contents between the main memory and the disk may be assumed to happen in the background via DMA, without blocking the execution of the running process. However, the execution of the ISR will occupy the CPU, for a short time, which may be ignored for simplicity.

   If MMU receives the page number -9 via message queue then it infers that the process has completed its execution and it updates the free-frame list and releases all allocated frames. After this, MMU sends Type II message to the Scheduler for scheduling the next process.

   The MMU maintains a global timestamp (count), which is incremented by 1 after every page reference.

   MMU is implemented in MMU.c or MMU.cpp. MMU will be executed via the **Master** process with four command line arguments: page table (SM1), free frame list (SM2), MQ2 and MQ3 as mentioned in **Master** module. It will implement page fault handling as a function inside it and call it whenever a page fault occurs.

Data Structures Required:

- Page Table

  Implemented as shared memory

  For each process there is a page table

  Size of each page table is same as the size of virtual space

  Each entry in page table contains < frame_number, valid/invalid bit >

  Initially, all frame numbers are equal to -1

- Free Frame List

Implemented as shared memory

Created by Master and maintained by MMU

- Process to Page Number Mapping

  Can be implemented using shared memory

  It contains the number of pages required by every process

Required Outputs:

The following outputs are to be generated.

a) MMU.c runs in xterm and produce the page fault information:

   i. Page fault sequence (pi,xi) - where pi indicates the process number and xi indicate the page number for which a page fault is generated. Also prints total number of pagefault for every process.

   ii. Global ordering (ti,pi,xi) - where ti indicates the global timestamp (which is incremented by 1 after every page reference) maintained by MMU, pi is the process number and xi is the page number.

b) All these outputs are written into an output file “result.txt”.

Evaluation Guidelines:

While entering marks, the partwise break up should also be entered according to the marking guidelines given below. There is a separate component for individual assessment, based on how the student answers questions.

Sl	**Items for the Master**	**Marks**
(1a)	Spawning the processes of the other modules	5
(1b)	Creating the page table	10
(1c)	Creating the free frame list	10
(1d)	Creating the ready queue	10
(1e)	Creating the message queues	5
(1f)	Other operational responsibilities of the Master	5
	Total	45

Sl	**Items for the Scheduler**	**Marks**
(2a)	Monitoring and maintaining the ready queue	5
(2b)	Selecting processes for scheduling	5
(2c)	Scheduling processes on receiving the Type I message	5
(2d)	Scheduling processes on receiving the Type II message	5
	Total	20

Sl	Items for the Process	Marks
(3a)	Handling valid page numbers from the MMU	5
(3b)	Handling page fault from the MMU	5
(3c)	Handling invalid page reference from the MMU	5
	Total	15

Sl	Items for the MMU	Marks
(4a)	Maintaining and simulating the TLB	5
(4b)	Maintaining and working on the page table	5
(4c)	Handling page faults	10
(4d)	Maintaining the global timestamp	5
	Total	25