CS39002 Operating Systems Laboratory
Spring 2024

Lab Assignment: 8
Date of submission: 07–Apr –2024

Simulating Virtual Memory through (Pure) Demand Paging

1. Problem Overview

We simulate a demand-paged virtual memory in this assignment. In a typical multiprogramming environment, processes arrive in the system and they are scheduled by the scheduler. Once a process is allocated to CPU, it starts execution and hence CPU starts generating the virtual addresses. The sequence in which the virtual addresses are generated, is known 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 order to determine the frame number, it consults the page table. If the page reference is illegal, the current process gets terminated, whereas if the page is present in the main memory, the current process can continue its execution. Otherwise a page-fault occurs. At that point of time, an I/O request needs to be initiated to bring the page from disk to main memory. As a result, the scheduler then takes the CPU away from current process and schedules another process. Meanwhile the page-fault handler routine of 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, then it loads the page into this frame. Otherwise it identifies a victim page using LRU page-replacement algorithm 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 it can be scheduled again by the scheduler.

We simulate these activities using **four** modules

(a) **Master** is responsible for creating other modules as well as for the initialization of the required 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 faults.
(d) **Process** generates the page number from reference string.

After Process completes execution, Scheduler notifies Master. Master terminates Scheduler, MMU, and finally itself.

Next, we describe each module in detail with respect to its specific tasks, data structures and interaction with other modules. We also specify the input and output of the system.

2. Input

- Total number of process (k)
- Virtual address space – Maximum number of pages required per process (m)
- Physical address space – Total number of frames (f)

3. Master Process

Master process performs the following tasks.

(a) Creates and initializes different data structures as mentioned below
   (i) Page Table: Master process creates one page table for each process, where the size of each page table is same as the virtual address space. This is implemented using shared memory (SM1).
   (ii) Free Frame List: Master creates this list, which is used by the MMU. This is implemented using shared memory (SM2).
   (iii) Ready Queue: Master creates this queue, which is used by the scheduler. This is implemented using message queue (MQ1).
   (iv) In order to communicate between different modules, it creates two more message queues.
      (1) MQ2 is used for communication between scheduler and MMU.
      (2) MQ3 is used for communication between processes and MMU.

(b) Creates scheduler for scheduling the processes from ready queue (MQ1). It passes following parameters (MQ1, MQ2) via command-line arguments.

(c) Creates MMU for translating page number to frame number and to handle page-faults. It passes following parameters (MQ2,MQ3,SM1,SM2) via command-line arguments.

(d) Creates k number of 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. It also shares the message queue (MQ1, MQ3) with every process.

3.1 Implementation Details

Master module is implemented in Master.c file. It reads 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 to that process and allocates frames proportionately.

Reference String Generation

- For a process p_i, if total number of pages required is m_i as randomly selected by master, then
  - Length of reference string for p_i is $\geq 2 \times m\_i$ and $\leq 10 \times m\_i$
  - Select a random number (x) from this interval and that is the length of the reference string
  - While generating the reference string, it is to be ensured that it contains invalid (but legal) addresses as well
  - With a low probability, illegal addresses can be generated (so a few processes encounter segmentation faults)
- Master initializes a table containing the number of pages required by every process. This is used for checking if the process is requesting any invalid page reference.
- Master first creates and initializes all data structures, then creates scheduler and next creates MMU. Then it creates all the k processes.
- After performing all process creation tasks, master will wait() until scheduler notifies completion of all the process execution. Master will terminate scheduler and MMU first and then terminates itself.

4. Scheduler

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 and removes it from the queue and sends it a signal for starting execution. Then the scheduler blocks itself until it 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 schedule 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

4.1 Implementation

Scheduler is implemented in sched.c file. Master executes the sched.c via exec() with the proper arguments as explained in the implementation of Master. Once all the processes are executed, scheduler informs the master to terminate all the modules.

5. Processes

Execution of process means generation of page number 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 go 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 PFH routine to handle the page fault. Please note 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. Please note that 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).

5.1 Implementation

This is implemented in process.c. Processes are created by Master with proper argument (as mentioned in master section). The processes will put them in the ready queue and pause itself. 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), –2 (when tries to access invalid page reference). 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 notifies the scheduler (see MMU). Scheduler terminates the process and removes it from ready queue.

6. Memory Management Unit (MMU)

Master creates MMU and then MMU gets suspended. MMU wakes up after receiving the page number via message queue (MQ3) from process. It receives the page number from the process and checks in the page table for the corresponding process. There can be following two cases:

- If the page is already in page table, MMU sends back the corresponding frame number to the process.
- Else in case of page fault
  - Sends –1 to the process as frame number
  - Invoke the PageFaultHandler (PFH) to handle the page fault
    - If a free frame is available, then update the page table, and also update the corresponding free-frame list.
    - If no free frame is available, then do local page replacement. Select victim page using LRU, replace it, bring in a new frame, and update the page table.
  - Intimate the scheduler by sending Type I message to enqueue the current process and schedules the next process
- 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.
- If MMU receives –2, it prints "TRYING TO ACCESS INVALID PAGE REFERENCE" and sends Type II message to the scheduler for scheduling the next process.
- Please note that MMU maintains a global timestamp (count), which is incremented by 1 after every page reference.

6.1 Implementation

MMU is implemented in MMU.c. 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 Matser module. It will implement PFH as a function inside it and will call it whenever a page fault occurs. As mentioned in scheduler section after resolving the page fault or in case of invalid page reference, it sends corresponding notification messages (Type I or Type II) to the scheduler.

7. Data Structures

- 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

8. Output

Code Master.c comprising the subroutines sched.c, process.c, mmu.c etc

Please produce the following output as mentioned below.

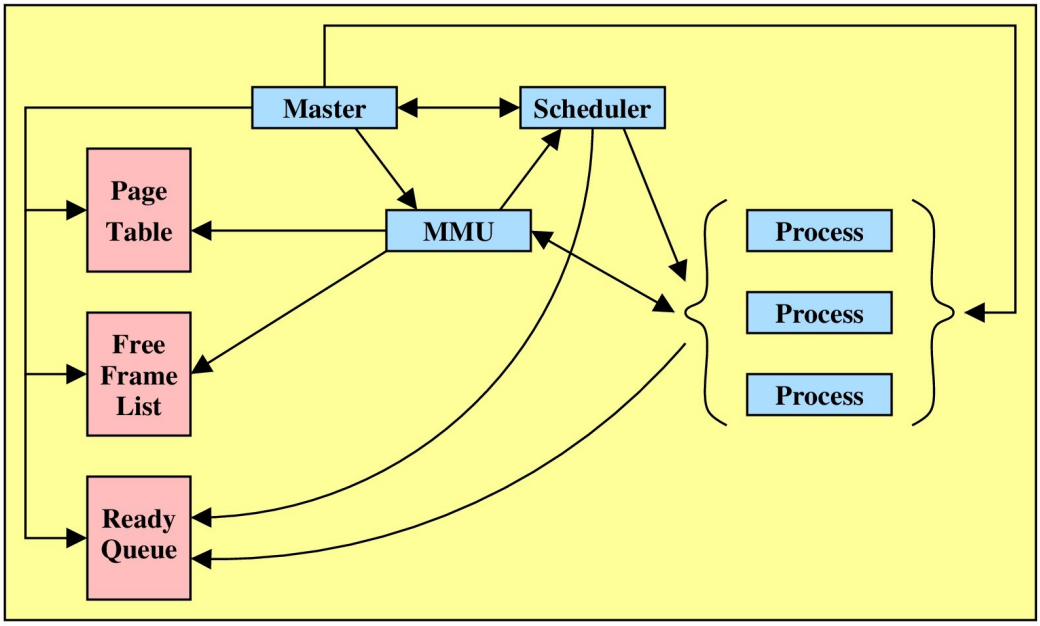
(a) MMU.c runs in xterm and prints the page fault information

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

(ii) Invalid page reference - (pi,xi) - where pi indicates the process number and xi indicates the invalid page number. Also prints total number of invalid page references for every process.

(iii) 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) Also write all these outputs in an output file result.txt.



Data Structures and interaction diagram among different modules