OS Lab Assignment

Simulation of Virtual Memory—Part II

In this assignment, you will extend your own code of the previous assignment to make a full-fledged virtual memory system.

Overview

In the previous assignment, a fixed amount of RAM was allocated to each process. This policy is now changed. Instead, processes will be allocated RAM from a global pool, on a needs basis. To be consistent with this policy, page replacement will now be done on a **global LRU basis**. Also, the virtual memory manager is structured into two concurrent threads for greater effectiveness.

- The virtual memory manager now contains **two separate threads**:
  - The *page I/O manager* performs writing out of page frames and loading of pages in memory. Operation of the *page I/O manager* is centered around a table named *I/O table*. Each entry in this table describes details of an I/O operation: which page of which process, and which page frame are involved in the operation, and whether the operation is a page-in or page-out operation.
  - The *free frames manager* keeps a sufficient number of free page frames in memory at all times and maintains a *free list* of such page frames. It performs this task by examining pages contained in all page frames and identifying some of them for replacement on a global LRU basis. If the page contained in the page frame is not a dirty page, it adds the page frame to the free list straightaway. If the page is dirty, it requests the *page I/O manager* to perform a page-out operation on the page frame and adds that page frame to *free list* when its page-out operation completes.

  The *free frames manager*, the *page fault handler* and the *page fault handler* share the *I/O table* using appropriate synchronization to avoid race conditions.
- When a page fault arises, the *page fault handler* simply takes of a page frame from the *free list* and loads the required page in it.
- As in previous assignment, the simulation is controlled by the primary thread, which reads its commands from a file named `init`. The `create` command now has one less parameter and the `init` file contains some new commands that govern functioning of the *free frames manager*. Details of the commands are given later in this document.
- As in previous assignment, operation of a user process `i` is controlled by commands in the file `si`.

Details

1. You must have data structures and program logic that is consistent with the Specification given in the Appendix.
2. The *free frames manager* is a thread that maintains a *free list* of page frames and makes sure that a sufficient number of page frames are in the list at every instant of time. This number is some n such that

   $$lower\_threshold \leq n \leq upper\_threshold$$

   where *lower_threshold* and *upper_threshold* are constants.

   To fulfill this requirement, **at appropriate times** the *free frames manager* examines the pages of processes that are in memory and arranges to free a sufficient number of them.

   - The *free frames manager* uses the LRU algorithm on a **global basis** to decide which page frame to add to the free list at any time.
   - The *free frames manager* can add an unmodified page to the free list at any time.
   - If a page has been modified since it was last loaded (it is called a *dirty* page—it is a page whose *modified* or *dirty* bit has been set), the free frames manager informs the *page I/O manager* that a page-out operation should be performed to write the dirty page in the swap space of the process. When the *page I/O manager* informs it that the dirty page has been written out, the *free frames manager* adds the page frame to the free list.
   - The *free frames manager* blocks itself when $n \geq upper\_threshold$. It should be activated when $n < lower\_threshold$.
   - The *free frames manager* must produce report lines to indicate the following events:
     - That it is blocking itself.
     - That it has been activated.
     - That it has added a page frame to the free list.
     - That it has instructed the page I/O manager to write out a dirty page.
3. The *page I/O manager* is another thread. It performs writing out of page frames and loading of pages in memory. Your code **must** perform some dummy I/O operations in the *page I/O manager* to provide a realistic feel to the simulation. (**Note:** The dummy I/O operation would delay the *free frames manager* and page faulting user processes—that is the realistic touch we want.)

   Operation of the *page I/O manager* is centered around a table named *I/O table*. Each entry in this table describes details of an I/O operation: which page

of which process, and which page frame are involved in the operation, and whether the operation is a page-in or page-out operation.

- When the *page fault handler* decides to load a specific page in a specific page frame, it enters details of the page-in operation in the *I/O table* and blocks itself. When the *page I/O manager* completes the page-in operation, it activates the *page fault handler*.
- When the *free frames manager* decides to write a dirty page from a page frame, it enters details of the page-out operation in the *I/O table*. When the *page I/O manager* completes the page-out operation, it activates the *free frames manager* which adds page frame to the *free list*.

You must use appropriate synchronization in your code to avoid race conditions during these operations.

4. The *page fault handler* simply uses one of the free page frames from the list of free page frames to load the page whose reference had caused the page fault.
5. The main thread reads commands from the file named `init`. These commands concern creation of processes, specification of physical memory size, and specification of the number of page frames that should be in the free list. The Appendix contains details of file `init`.
6. As in previous assignment, a process `i`, where `i` is an integer, reads commands from the file named `si` concerning access and modification of specific logical addresses. The Appendix contains details of files `si`.
7. For testing your code, you may decide on the size of memory, the sizes of their logical address spaces, and the number of free page frames. This must be done through commands in the `init` files.
8. Evaluation will be based on performance of your program on a set of test input files, so make sure you use the indicated format for your input files.
9. **For convenience, you may first test the functionality of the *page I/O manager*, and then add the necessary synchronization.**
10. Note the following points regarding coding:

    - **You must use your own code from previous assignment.**
    - **Highest standards of academic honesty are expected and will be enforced.** Hence, you **must not** use any code that is not written by you.

Disclaimer: Right to make refinements/corrections to the specification is reserved.

Appendix : Specification of input files

1. The `init` file contains a list of commands, where each command appears on a separate line. Each command has the format: <action> <parameters>, where a single space is used to separate an action from its parameters and commas are used to separate parameters from one another.

   Following actions are to be supported:

Action_name	Parameters and explanation
Memory_size	Number of page frames (integer)
Lower_threshold	Minimum number of page frames that are expected to be free (integer)
Upper_threshold	Maximum number of page frames that are expected to be free (integer)
Create	Process name (an integer), and process size, which is the max number of pages (integer)
Page_table	No parameters. It displays the page tables of **all** processes.

2. A file `si` contains the following commands

Action_name	Parameters and explanation
Access	page no, word no (both are integers): The indicated process makes a read access to the indicated page.
Modify	page no, word no (both are integers) : The indicated process modifies the indicated page.
End	The process has ended. The number of access and modify operations and the number of page faults should be printed.

3. You must design and use the following data structures:

   (a) Page table: for each process.

   (b) Free list: List of free page frames.

   (c) Table/list of page frames to be written out: The free frames manager would put page frame ids here. The Page I/O manager would perform page-out operations on the pages in these page frames and inform the *free frames manager* when the writing out of a page frame has been completed.

   (d) Any other tables or lists you may need.