# CS 5348 Operating Systems -- Project 4

For Project 4, you need to replace the simple memory manager with a simulalted virtual memory demand paging scheme. You should complete the implementation in “swap.c”, “paging.c”, and “loader.c” to realize the demand paging scheme (memory.c can be deleted). You can implement the system in three phases. But you only need to turn in the final project and specify which phase you have completed. The detailed description about the three phases are as follows.

A new tar file, including the new code files, is provided. You can either make changes from the new source code or make changes starting from your previous code. If you start from the new source code, you do not have to port it to socket clients. To ease your debugging effort, you can simply use the admin submission to submit programs.

For this project, you can form a group of 2 members. Your code should have specific comments on who implemented which function (if an entire module is implemented by one student, then state at the beginning of the file, instead of at each function). You should also specify in the readme file the contributions of the group members at a high level. A recommended division is to have one member implement the paging functions and another implement the swap and loader functions.

## 1 Phase 1: Simple Paging

To manage simple paging, first, you need to implement a free list for the memory manager. At the system initialization time, the free list includes all memory frames besides those occupied by the OS (# OS pages = OSpages, which is read in from config.sys).

Next, you need to implement the page table for each process. For simplicity, the page table for each process can be a fix-sized array (#entries = maxPpages, which is read in from config.sys), one entry for each process page. The page table for each process should be dynamically allocated when the process is created. PCB of each process has a pointer pointing to this page table (PTptr). Note that in the page table, you need to clearly indicate whether each page is in-use (actually has content). If you read from an unused page, there should be access violation. It is ok to write to an unused page and upon such write, you need to change the status of the page to indicate that it is in-use.

Next, you shall implement the loading functions. In the original program, loading is done in memory.c. In real operating systems, there is a loader program to load executables. We now simulate the loader program by loader.c. In this phase, the loader loads all pages of the program to the memory. The memory frames for a new process can be obtained from the free list (get_free_page). After allocation, you need to load instructions and data into these pages (the load_instruction and load_data functions). If, during allocate_memory, the memory does not have a sufficient number of free frames to load a process, the submission should be denied and a message should be printed to indicate that.

During program execution, the addressing scheme needs to be implemented to allow the system to fetch the correct physical memory. You can implement an addressing algorithm “calculate_memory_address” and let get_instruction, get_data, and put_data call it to compute the correct physical memory address for the given offset.

Do not forget to implement the free_process_memory function so that pages of a terminated process will be returned to the free list of pages. Function free_process_memory is called by process.c after process termination.

You should also change the dump_memory functions to dump various memory related information, such as the list of free memory frames, the page tables of all the active processes, and the actual memory contents

for each process. When printing the actual memory content for each process, you should print page by page, first print the page number and the corresponding frame number, and then print the memory content of valid memory entries (for that process) in that page. When printing the page table for the current running process, you need to print all the non-null entries in the page table and their detailed attributes. You also need to change the admin.c to include additional dump functions.

# 2 Phase 2: Demand Paging

You now have to change the paging scheme to demand paging, i.e., the address space of a process does not have to be fully in memory. You need to implement the swap space manager. In real OS, the swap space management is very similar to the memory frame management. It has a large number of pages and a free list to provide free pages upon requests. PCB needs to know where in the swap space each of its page is. Our implementation is a greatly simplified version. We create a swap space in a file "swap.disk" and allocate a fixed number of swap pages to each process (like the original simple memory implementation). The starting address for page $p$ of process $pid$ can be computed by $(pid - 2) * swpp * ps + p * ps$, where $swpp$ is the number of swap pages per process (which is maxPpages read in from config.sys) and $ps$ is the page size (which is pageSize * dataSize).

When a valid memory address is referenced (by get_data, put_data, get_instruction functions), but the corresponding page is not in the memory (a page in the swap space, or a newly written page), you need to create a page fault. You need to define a page fault interrupt bit in "simos.h". You need to change the code of handle_interrupt function in "cpu.c" to process this interrupt (simply call "page_fault_handler", which should be defined in "memory.c"). Note: the old code has page_fault_handler, which is not correct. I just forgot to remove the half-coded function body.

In page_fault_handler, you need to ask the swap space manager to bring the page into memory by calling the insert_swapQ function to start the swap space request. But before doing that, the memory manager needs to allocate a new memory frame for the process. You can call get_free_page to get a free memory frame for the page fault. If there is no free frame, then you can call select_agest_frame to get a frame and the corresponding to-be-replaced page. In this phase, you will not be able to select a page based on aging policy. So, select_agest_frame can just return any page.

After a frame is chosen for the page fault, you need to update the page tables of the processes who have the swapped in and swapped out pages. You also need to update the metadata of the selected frame to have the new process information. Then the swap request is inserted to the swap queue. At this time, the process with page fault should be put into wait state. Memory should not handle this directly. The page fault is raised by the get_data, put_data, or get_instruction function and they should return mPFault back (to cpu.c). cpu.c will check the memory return value and set the exeStatus to ePFault and set page fault interrupt. The page fault interrupt will be caught by the handle_interrupt function and subsequently, page_fault_handler will be called. Subsequently, process.c should check exeStatus and put the process into wait state.

Both swapping in the page with page fault and the dirty page write are diskIO and should be processed by the swap space manager, and diskIO should happen in parallel with CPU execution. Thus, we simulate the swap space manager by running it as a separate thread (in fact, only diskIO should be a separate thread, but we anyway put the entire swap space manager as a separate thread). To simulate the high latency in disk IO, we define a diskRWtime and at the end of read_swap_page and write_swap_page, you should put a sleep for diskRWtime in microseconds.

The swap manager thread processes the requests in its swap queue. Each swap request can be a write request for a page being swapped out, or a read request for a page being swapped in (which is flagged by the act field and can have values actRead and actWrite). If a selected frame contains a dirty page, then we need to write it back to the swap space before loading another page in. This means you need to keep track of the dirty status for each memory frame. When a physical memory frame is written, the dirty bit should be set to 1. You should only write a dirty page of an active process (not a terminated process) back to swap space. A

swap request should also specify the action to be performed by the swap space manager after the swap request is fulfilled. These actions include to put back the process to the ready queue (more precisely, insert the process to endWait queue and set the IO completion interrupt). We also consider another action item, free the buffer passed to the swap manager (which is needed by the loader in the next phase). All together, the "finishact" flag (specifying the action after swap is done) can be toReady (put the process back to ready queue), freeBuf (free the swap input buffer for write, never the output buffer for read), Both (both toReady and freeBuf), and Nothing (do nothing).

You need to complete the code in swap.c for swap queue management and for thread creation/disposal. You also need to do the similar sync/mutex as in term.c to sync/protect the swap space activities. The swap space sync is a little more complex than that of term.c. You need to analyze carefully to ensure the program correctness.

# 3 Phase 3: Page Replacement Policy and Loader

In this phase, we need to implement the page replacement policy, specifically, the aging policy. You should associate an aging vector to each physical memory frame. Each aging entry is 1 byte (8 bits). The aging vector should be initialized to 0 (zeroAge). When a physical memory is being accessed, including when a page is loaded to it, the leftmost bit of the aging vector should be set to 1 (highestAge). The aging vectors for all memory frames should be scanned periodically (a timer is set to the desired scan period, when the time is up, the interrupt bit will be set, then the scan activity will be invoked by the interrupt handler calling memory_agescan). During each scan, aging vector value should be right shifted. When the aging vector of a memory frame becomes 0, the physical memory should be returned to the free list and if it is dirty, its content should be written back to the swap space. We consider a lazy write-back policy, i.e., swap out is only done when the freed frame is allocated to another process. The original owner process of the frame can continue to use the frame till it is selected to be used by another process. In case the original owner did use a frame after it is freed, we should revert the state of the frame, i.e., the frame should be returned to the original owner process, unless the age is again 0.

If there is a page fault but no free memory frame in the free list (from get_free_page), then you need to call select_agest_frame to select a memory frame with the smallest aging vector. In select_agest_frame, you need to scan all frames. If there are frames with age vector = 0, then you need to free all of them and return one of them to the caller. During the first scan, you should also find out what is the lowest age value. In the second scan, you need to free all the memory frames with the lowest age value (put all of them to free list) and return one of them to the caller. You can differentiate dirty frames from clean ones. In case a frame is dirty with the lowest age (but != 0) and there are other clean frames with the lowest age, then you can skip the dirty frame without selecting it and even without freeing it. If the dirty frame is the only frame with the lowest age, then select it to be swapped out.

Now you have the swap manager, you can rewrite your loader to load a program without having to load all its pages. In a real operating system, the loader loads some pages of a program into memory and maps the remaining pages from disk to memory (the disk pages are as though memory pages). We will use a different approach. We load all the pages of a program to its swap space and then load some initial pages from the swap space to memory. All these have to be done through swap manager, i.e., you need to insert the write requests for writing the loaded pages to the swap space to the swapQ and also insert read requests for reading the swap pages to memory to the swapQ. The purpose of the "finishact" field of the swapQ (discussed in Phase 2) is for the proper control of what to do after swap action is finished. You can set the field properly when issuing the swap request to make sure that the system will operate properly.

# 4 Printout

To facilitate observing your program behavior, you need to implement a swap space dump function to dump the swap space information, including dumping the swapQ and dumping the actual swap content for a

certain page of a certain process. You also need to modify the dump memory functions to dump information about the memory, including dumping the metadata for the frames (age, free, dirty, the owner process and owner page number, etc.), the free list, and the actual memory content. Also, for each process, you need to have functions to print the process page table, the actual memory content of each page. If the page is on disk, you need to print the actual page content in the swap space.

For each page fault, some information about the page fault and its handling steps should be printed. When there is a page fault, you should print a statement showing that there is a page fault and give the process pid and the faulted page number. During page fault handling, you need to print the information of the frame being selected for the faulted page, whether it is selected directly from the free list or it is selected by using the lowest age criteria, the original frame metadata and the updated frame metadata, etc. When you insert the swap operations into swap queue, you should print what you have inserted, either by the swap manager, or by the memory manager, or by both.

When printing the actual memory content for each process, you should print page by page. First print the page number, its age vector, and its dirty bit and free bit. Then print the memory content of valid pages of the process. Always give proper headers about what you are printing.

To observe the ordering of the execution of the main thread and the swap thread, you need to print swap activities as well. When a swap request is processed, print the corresponding information about the swap request.

# 5 Testing

You have to generate your own input programs and change the parameters specified in config.sys to thoroughly test your memory manager implementation.

Note that the input program no longer works with the new simos.exe provided in the new tar file. In the original input program, we start the data addressing from 0. In the new simos.exe, the addresses for data have to continue from the address of the last instruction. You have to change the sample input programs. You also need to generate some new input programs for testing your memory manager, especially for special accesses and access violations.

The config.sys file for the new simos.exe has some changes also. A new version of config.sys is provided in the tar file. You need to change the parameters in config.sys to explore your implementation. We will use some boundary conditions to test your new memory manager implementation.

# 6 Submission

You need to electronically submit your program **before** midnight of the due date (check the web page for the due date). Your submission should be a zip file or tar file including the following

- ✓ All source code files making up your solutions to this assignment.
- ✓ The *ReadMe* file that contains:
  - ✓ Which phase you have completed and whether there are some features in that phase that do not work yet, or some features in the next phase that have been realized.
  - ✓ Deviations in your system from the specifications, including added features as well as missing and changed features.
  - ✓ If you use the new tar source code, which files you have changed from the source code we provided and what changes are made (be very specific on the changes you made) on files other than paging.c, swap.c, and loader.c.
  - ✓ The contribution of the group members at a high level.

You should submit your project through elearning.