CS 5348 Operating Systems -- Project 3,4

The goals of theses two projects are to let you get a better understanding on how OS works via coding and get familiar with some important Unix system calls via using them. In the following, we first discuss a program that implements a very simple OS and simulates the underlying computer, then discuss how you should change the implementation of the OS components using more sophisticated approaches.

1 Overview of a Simulated Simple OS (SimOS)

In SimOS, we simulate a computer system and then implement a very simple OS that manages the resources of the computer system. The computer system has components: CPU (with registers), memory, disk swap space, and clock. CPU drives memory (through load and store instructions) and other I/O devices (e.g., I/O for page fault). We simulate these by software function calls. When system starts, CPU executes the OS program, which loads in user programs upon submissions and initiates CPU registers for a user program so that CPU (sort of being arranged into) executes the user program. Some more details about the simulated computer components are discussed in 1.1, and the OS that manages the resources is discussed in 1.2. The outer layer user command processing (like shell) is discussed in 1.3.

1.1 Simulated Computer System

CPU has a set of registers (defined in simos.h) and it performs computation on these registers (implemented by function cpu_execution in cpu.c). During one CPU execution cycle, it fetches an instruction from memory and performs the actions according to the instruction. Besides the end-program instruction, the corresponding datum for the instruction is also fetched from memory. The instructions in our simulated computer are listed in Table 1.

Table 1. Instructions in a program.

OP Code	Parameters	System actions
2 (load)	Indx	Load from M[indx] into the AC register, indx (>=0 int)
3 (add)	Indx	Add current AC by M[indx]
4 (mul)	Indx	Multiply current AC by M[indx]
5 (if-goto)	indx, addr	A two-word instruction: indx is the operand of the first word; and addr is the operand of the second word. If M[indx] > 0 then goto addr, else continue to next instruction
6 (store)	Indx	Store current AC into M[indx]
7 (print)	Indx	Print the content of M[indx], AC does not change
8 (sleep)	time	Sleep for the given time in microseconds
1 (end)	Null	End of the current job, null is 0 and is unused

At the end of each instruction execution cycle, CPU checks the interrupt vector. If some interrupt bits are set, then the corresponding interrupt handling actions are performed (by handle_interrupt in cpu.c). Currently, we define 3 interrupt bits (defined in simos.h) and they are discussed in Table 2. The interrupt vector is initialized to 0. After completing interrupt handling, the interrupt vector is reset to 0. (What if interrupt bits get set during interrupt handling?)

Table 2. Interrupt vector.

Interrupt bit position	Description
0 for cpu time quantum (vector value = 1)	Upon activating the execution of a process, process manager sets a one-time timer of duration time quantum. When the time quantum expires, the timer sets this interrupt bit. The handler then sets the

	process to ready state and returns the control to process manager, which will switch process.
1 for age scan timer (vector value = 2)	Memory should set a periodical timer for scan and update of the memory age vectors. Timer sets this interrupt bit when the time is up. The interrupt handler simply invokes the aging scan activity for this interrupt bit.
2 for IO completion (vector value = 4)	When IO of a waiting process completes (such as page fault IO), the manager places the process into an endIO list and sets this interrupt bit. There may be multiple processes with their IO completed but the bit will only be set once. Thus, the interrupt handler fetches all processes in the endIO list and put them in the ready queue.

Memory provides CPU the functions required during CPU execution, including get_instruction(offset), get_data(offset), and put_data(offset) in memory.c. The parameter offset is based on the address space of the process and has to be converted to physical memory address (this conversion is supposed to be done above the physical memory, but for convenience, we put it in physical memory). When a new process is submitted to the system, the system will load the program and the corresponding data into memory. So memory unit also provides functions load_instruction(…) and load_data(…) for this purpose (simulating one type of direct memory access (DMA) without going through CPU).

Terminal (term.c) outputs a string to the monitor. When CPU (cpu.c) processes the print instruction, it puts what is to be printed as a string and sends the string to the terminal queue. When a process ends, process manager (process.c) prepares an end-program message and sends it to terminal queue. At the same time, the running process is switched into the waiting state. Since terminal is an IO device working in parallel with CPU, we need to run terminal manager as a thread. The terminal thread goes through the printing requests in the queue and process them (print the string). Note that inserting requests to the terminal queue is done by the main thread and the terminal thread removes requests from the queue. Thus, they need to be synchronized.

A data intensive process may use a lot of memory space for processing a large amount of data. It may not be possible to put all the pages of a process fully in memory. Swap space is used to allow the system to only keep currently needed pages in memory and put the remaining pages for a process on disk (called swap space). We simulate the disk swap space and implement the swap space manager in swap.c.

1.2 Simulated Simple OS

In process.c, the OS implements its process management functions. First, a PCB structure is defined in simos.h. A ready queue is implemented and process scheduling should be done on the ready queue. Currently the scheduling policy is a FIFO based Round Robin. When a process is submitted, the submit_process() function is invoked, which loads the instructions and data of the process into memory, prepares the process control block PCB, and places the process in the ready queue. Execution of processes is supposed to be done continuously under the control of OS, but we use an admin command to directly control the execution to allow easier observation of the behaviors of the system. When executing a process execute_process(), a process is fetched from the ready queue and the CPU function cpu_execution is called for executing the program.

Some code for the simulated OS is mixed with the simulated computer system. In memory.c, two functions allocate_memory(…) and free_memory(…) are offered, which are supposed to be implemented in the OS. In terminal, the function terminal_output simulates terminal printing and the other functions are terminal manager within the OS. Similarly, the swap space manager has the simulated disk IO (read_swap_page and write_swap_page) functions and the OS management code (the remaining functions) mixed in swap.c.

Timer provides the function (add_timer in timer.c) to let other components of the system to setup a timer event. Each timer includes the time (relative, counting from current time), the action to be performed after the time is up, and whether the timer is periodical. When the time is up, the timer manager takes the specified action. The supported actions are discussed in Table 3. Timer manager also automatically inserts the timer back if it is a periodical timer. When a timer is set, the pointer to the timer is returned (casted as an unsigned integer to avoid the necessity of exposing the internal timer structure), which can be used to locate and deactivate the timer by the deactivate_timer function. Note that in a real system, timer is supposed to have its own clock and will check the timer events upon a certain number of clock cycles. In our simulated system, we use CPU cycles as the clock.

Table 3. Actions for the timer.

Action Code	Corresponding action
1 for time quantum action	Set the corresponding interrupt bit and put the process in endWait queue in order for it to go back to the ready queue
2 for age vector scan action	Simply set the corresponding interrupt bit
3 for null action	Do nothing

1.3 Command Processing

We simulate control commands by an adminstrator in admin.c. Available commands to the system are listed in Table 4.

Table 4. Commands in the system.

Action	Parameters	System actions
T	-	Terminate the entire system
s (submit)	fnum	Submit a new process, should be shifted to client program
x (execute)	-	Execute a program from ready queue for one time quantum
r (register)	-	Dump registers
q (queue)	-	Dump ready queue and list of processes completed IO
p (PCB)	-	Dump PCB for every process
e (timer events)	-	Dump timer event list
m (memory)	-	Dump memory related information
d (term & swap)	-	Dump the swap request queue and the terminal queue

2 Project 3 Description

The goal for implementing the computer and OS simulator, simOS, is to let you get a deeper understanding of the activities going on in computer and OS. It may be challenging for you to implement the simOS system completely. Thus, we provide a working simulator “simos.exe” which implements simple OS resource management policies. Some components can be replaced by more advanced resource management algorithms. You will implement a more advanced memory management solution for simOS. Now, to let you get hands-on with the simulator, we removed some parts of the code and ask you to put them back in to complete the system.

cpu.c	The simulated CPU executes instructions and handles interrupts. You need to complete the missing instruction execution code (micro instructions on registers) and understand the interrupt handling functions.
process.c	The process manager manages ready queue and process execution. During process execution, there may be IO instructions, sleep instruction, page fault, or error that will stop the regular execution of the process. Also, some events may occur in the system which interrupts process execution. These are all considered in the process execution procedure. You need to put some code for the execution of an idle process to get a closer look into the process management tasks. You also need to add code for simulated context switch.

term.c	We have removed the synchronization code in this component and you need to add them back. You need to make sure that the termIO function does not busy looping on null terminal queue. You also need to achieve mutual exclusive accesses to terminal queue. Add semaphore controls to the code to achieve the goal.

Files cpu.c and process.c are marked with "*** ADD CODE" strings to indicate where you have to add the desired code. In term.c, your goal is to achieve synchronization. So where to add sync code will be your design.

Another change needed to make the system work is to port simOS to your client-computer framework. In Project 1, you have Admin as a forked process interacting with Computer via pipe. In this project, we will not have admin as a separate process. There will only be Client and Computer interacting via sockets. You need to rewrite submit.c and term.c for the purpose. Now, simOS is the Computer. Your Computer code of Project 1 can be used in place of submit.c. But some logic in submit.c should be retained to ensure that the system will work.

You used to have a thread in Computer, which reads from multiple Client sockets in turn. You can now replace it by the "select" system call. If you do not wish to make the change, you can specify this in your design document.

Your Client code in Project 1 will remain almost the same, which reads file name from the terminal and sends it to Computer. But Computer may return multiple print out messages to Client. Your can use the message "Process … terminated" as the indicator of process termination.

You also need to send the computation results to Client so that Client can print them on its own window. This requires you to change the code for function "terminal_output" in term.c, which now prints the output to a file. You need to save the socket file descriptor sockfd so that communication to the Clients can be done properly. In simOS, you can save sockfd in PCB (in real OS, PCB has a pointer pointing to an array of file descriptors). You need to add one more parameter to insert_termio function to pass sockfd. When you call insert_termio from cpu.c and process.c, the corresponding sockfd can be retrieved from PCB and passed to the terminal queue.

3 Project 4 Description (almost ready)

In this project, you need to replace the simple implementation of the memory manager with a virtual memory demand paging scheme. You can implement the system in three phases. In the first phase you implement a simple paging scheme. In the second phase, you implement an aging scheme for page replacement and convert the system to demand paging. In the third phase, the process with page fault should be switched out of the CPU, waiting till its page being brought into memory.

you should create a new thread to represent the I/O device which copies a faulted page to the memory while the CPU is executing another process. You only need to turn in the final project and specify which phase you have completed.

3.1 Phase 1: Simple Paging

To manage simple paging, first, you need to implement a free list for the memory manager. In the system initialization time, the free list includes all memory frames besides those occupied by the OS. Next, you implement the page table for each process. For simplicity, the page table for each process can be a fix-sized array (but the array should be dynamically allocated), one entry for each process page. PCB has a pointer pointing to the page table (PTptr). Now, you rewrite the allocate_memory function to allocate pages for newly submitted processes. The pages 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 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 “compute_address” and let get_instruction, get_data, and put_data call it to compute the correct physical memory address for the given offset. If we turn on the debug mode, then upon each invocation of the compute_address function, the offset passed in and the physical memory address computed should be printed.

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

You should also change the dump_memory function to dump 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 then print the memory content of valid memory entries (for that process) in that page.

3.2 Phase 2: Demand Paging

You now have to change the paging scheme to demand paging, i.e., there is no need to load the entire address space of a process to memory. You need to implement the swap space manager. In real OS, the swap space is structured very similarly as the memory. It has a 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 allocate a fixed number of swap pages to each process.

When loading a program upon process submission, the entire address space of the program can be loaded into the swap space, but only the first should be loaded into memory.

Whenever a valid memory address is referenced (by get_data, put_data, get_instruction functions), but the corresponding page is not in the memory, you need to ask the swap space manager to bring the page into memory by calling the insert_swapQ function to start the procedure. But before doing that, the memory manager needs to allocate a new memory page for the process. You can call get_free_page to get a free page for the page fault. If there is no free page, then call select_aged_page to get a to-be-replaced page. In this phase, you will not be able to select a page based on aging policy. So select_aged_page can just return any page.

The swap manager processes the page faults in swap queue. Each request involves two pages, one to be swapped out, one to be swapped in. If the page is obtained from select_aged_page, then we need to write it back to the swap space before loading another page in (later we need to check whether the page is dirty and only write it back to swap space if it is dirty). After finishing handling the page fault, the swap space manager should insert the process to endWait queue and set the IO completion interrupt. Note that the swap space manager handles disk IO, which runs in parallel with the CPU. So swap space manager should be a separate thread.

3.3 Phase 3: Page Replacement Policy

In this phase, we need to implement the page replacement policy, specifically, the aging policy. You should associate an aging vector and a dirty bit to each physical memory frame. Each aging entry is 1 byte (8 bits). The dirty bit should be initialized to 0 and the aging vector should be initialized to 0 as well. When a physical memory frame is written (store instruction is executed), the dirty bit should be set to 1. 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. 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.

When there is a page fault but no free memory frame in the free list (from get_free_page), select_aged_page should select a memory frame with the smallest aging vector. If the memory frame being selected to be swapped out is dirty, then its content should be written to the swap space; otherwise, no need to do so. For either case, we need to update the page table and then insert the swap request to swap queue. After handing the page fault to swap space manager, the process 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 return (to process.c). 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 to invoke diskIO. Thus, the swap space manager has to be executed as a separate thread.

To facilitate observing your program behavior, you need to implement a swap space dump function to dump the swap space information. You also need to modify the dump memory activities to dump more information about the memory, including aging vector and dirty bit. Also, for each page fault, some information about the page fault should be printed. All major dumping should use the Debug flag to allow the printing being disabled.

In summary, when a page fault occurs to the process in execution, the OS performs actions to handle the page fault. After the initial page fault processing, the memory manager (the main thread) forwards the page fault request to the swap manager thread for copying the faulted page from swap space into memory (and possibly copying the original content of the selected memory frame to the swap space). After that, the process manager (the main thread) switches out of the process in execution and another process is scheduled to run. At the same time, the process that is switched out should be in a waiting state, waiting for the faulted page to be brought in. When the disk has completed page fault handling, it should set the corresponding interrupt bit to inform the main thread that the page fault has completed. The main thread (process.c) should then place the process back in the ready queue.

3.4 Input

A system configuration file “config.sys” is used to set the configuration parameters of the system. The content of the configuration file is given in the following. The configuration parameters are defined in simos.h and read in system.c in the initialize_system function.

<max-process> <quantum> <idle-quantum>
    -- max-process: max number of processes allowed
    -- quantum: time quantum, given as number of instructions
    -- idle-quantum: time quantum for the idle process, cloud be less than <quantum>
<page size> <total memory pages>
    -- page size is the size of each page in physical memory
    -- total memory pages is the total number of pages in the memory
<number of load pages> <max number of pages per process> <OS pages>
    -- number of load pages is the #pages given to each process upon loading the process
    -- max number of pages per process: used to determine the page table size for each process
    -- OS size is the size of the address space of the OS which has to stay permanently in memory
    -- all these sizes are in number of words
<age scan period> <terminal output time> <disk rw time>
    -- age scan period specifies how frequently the system should perform aging vector scan
    -- terminal output time and disk rw time are for simulating the slow IO (by usleep)

The program is in a simulated machine code. The format for a program is specified as follows.

<memory size> N M
-- memory size is the size of the memory the program requests, since we do not have instructions to allocate dynamic memory, the memory size for each program is fixed
-- N is the number of instructions in the program
-- M is the number of static data in the program
<instructions>
-- following the first line, there should be N lines of instructions
each instruction should have an opcode and an operand, both are integers in one line
the last instruction should always be end-program, with a dummy operand
the descriptions for opcode and operand are given in Table 1
<data>
-- following the instructions are N lines of data, initialized in the program file
every data should be initialized, for those without initial value, set it to 0 anyway

3.5 Output

Your program should print all the required information very clearly. You should change the dump_memory function to dump 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, its age vector, and its dirty bit. Then print the memory content of valid memory entries in that page. Always give proper headers about what you are printing. Also, when printing memory content of each process, you should only print the pages that are actually in the memory.

You also need to implement a dump_swap function to allow the administrator to dump the swap space. The dump should be similar to memory dump. You need to print the free page in the swap space and the swap space for each process. When printing the swap space for each process, you need to print the process id and all its pages that are in the swap space.

Whenever there is a page fault, after you processed the page fault, you need to print out a message to show that a page fault has occurred, print the process id and page number that had incurred a page fault, and print the free frame that has been allocated to the faulted page. If the free list was empty and the page fault causes a memory frame being swapped out, then the swapped out memory frame number and the corresponding process id and its page number that was swapped out should be printed.

3.6 Testing

You have to generate your own input programs and change the parameters specified in config.sys to thoroughly test your memory manager implementation. The sample prog.1 and config.sys are only for illustration purpose. We will use very different parameters and simOS programs to do testing.

4 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 *DesignDoc* file that contains the description of the major features of your program that is not specified in the project specification, including all major design decisions with respect to data and program structures.

You should submit your project through elearning.