CS 4348/5348 Operating Systems -- Project 4

The goals of this project are to let you get a better understanding on how OS works through coding and get familiar with some important Unix system calls through using them. In the following, we first discuss a program that implements a very simple OS and 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, and printer. CPU drives memory (through load and store instructions) and other I/O devices (through I/O related instructions). 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 system is discussed in 1.1, and the OS that manage 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 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 (sum)	Indx	Compute $\sum x$, x=1..M[indx], then add $\sum$ to AC
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
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.

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 set

	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 IOdone 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 IOdone 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 from address space 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).

Printer has only one function: printer(…), included in spooler.c, which simply receives a string and some corresponding information and prints them. We also let it check execution status of the process to be printed and print the status information.

1.2 Simulated Simple OS

In process.c, the OS implements its process management functions. First, a PCB is defined in os.h for each process. 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 a user 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 CPU function cpu_execution is called for execution.

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. These are two functions that you have to implement. Current implementation is extremely simple.

The spooler is implemented in spooler.c with the printer. It is in charge of allocating and free up spool space for each process (allocate_spool and free_spool). When a print instruction is executed, the function spool is called, which copies the information to be printed to the spool space. When a process finishes execution, either due to regular termination or due to error, print_spool is called to initialize the printing of the output of the process. The spooler simply sends the spooled content to the printer for printing.

Timer provides the function (set_timer in timer.c) to let other components of the system to set 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 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.

OP Code	Corresponding action
1 for time quantum interrupt	Simply set the corresponding interrupt bit
2 for age vector scan interrupt	Simply set the corresponding interrupt bit
3 for null action	Do nothing

1.3 Command Processing

We simulate shell command processing in command.c. User can issue commands to the system and the command processing component simply process the command by calling corresponding OS functions. 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	New process submitted, the program is in "prog<fnum>"
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
w (swap space)	-	Dump the swap space related information
l (spool)	-	Dump the spool space related information

2 Project Description

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 and 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.

2.1 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. Then, you implement the allocate_memory function to allocate memory for newly submitted processes. After allocation, you implement the page table for each process. The page table should be dynamically allocated and CPU.Mbase should now point to the beginning of the page table. You also need to implement the load_instruction and load_data functions to load instructions and data of a newly submitted process to the memory frames allocated to the process. If 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 observation mode, then upon each invocation of the compute_address function, the offset passed in and the physical memory address computed should be printed.

To allow easy observation, 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, and then print the memory content of valid memory entries (for that process) in that page.

2.2 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 a swap space manager. The swap space is structured very similarly as the memory. The swap space has a number of pages. There should also be a free list to manage all free pages in the swap space. 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 page needs to be loaded to memory. If the swap space is out of free pages, then the submission of the process should be denied.

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 bring the page into memory, including allocating a new memory frame for the process, updating the process page table, and copy the content of the page into memory. After finishing brining in the page, return the memory content (instruction or the data) for the memory access function.

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 being 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). 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, a memory frame with the smallest aging vector should be selected to be swapped out. If the memory frame being selected to be swapped out is dirty, then its content should be written to the swap space. Otherwise, you just need to update the page table to show that the corresponding page of a certain process is no longer in the memory.

To facilitate observing your program behavior, you need to implement a swap space dump function dump_swap to dump the swap space information. You also need to modify the dump memory activities to dump more information about the memory. Also, for each page fault, some information about the page fault should be printed.

2.3 Phase 3: Paging Fault Handling

In this phase, we simulate the IO device by a separate thread and allow page fault to be executed by this IO device thread. The IO device thread should be started in the beginning of the program and it takes IO requests from a queue and serves them.

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 IO request to the IO device 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 by the IO device. When the IO device has completed page fault handling, it should set the corresponding interrupt bit to inform the main thread that the page fault IO has completed. The main thread should then place the process with the page fault back in the ready queue.

Before activating the IO device to perform page copying, the necessary processing on the memory side should be done. After selecting a memory frame for the faulted page, if the selected frame is not from the free list, then the page table for the process that loses a page should be updated. If the selected frame is dirty then the content should be copied back to the swap space. The page table for the process with page fault should be updated. Of course, all these should have been implemented in the second phase.

When the main thread forwards the page copying request to the IO device, it should give the IO device thread the following information:

- A flag to indicate the IO action to be performed. In current case, we have two actions: swapInPage and swapInOutPages. In the case of swapInPage, the selected frame for the page fault is not dirty, so only need to copy the page from the swap space to the memory frame. In the swapInOutPages, the selected frame is dirty and needs to be copied to the swap space.
- The swap space address of the faulted page, and the memory frame allocated to the faulted page.
- The memory frame number that is to be swapped out, and the swap space address for the to-be-swapped-out frame (if the swapped-out frame is not dirty, then these will be dummy data)

The IO device thread should provide an insert_IO function to allow other components to insert page fault IO requests. Each request should contain the information listed above. The IO device thread should be inactive if the the IO request queue is empty. Whenever an IO request is inserted in the IO request queue, the main thread should activate the IO thread to process the IO requests in the queue. Once the IO request thread is activated, it gets an IO request from the queue and performs the necessary processing (copying).

Note that most of the processing for a page fault should be taken care of right when the page fault occurs, instead of being processed during interrupt handling after IO completion. The processing during interrupt handling should be minimized to not to disturb the execution of the running process too much. Too much processing during interrupt handling also can result in a full context switch, which is not desirable. Right after a page fault, there will be a process switch anyway, so OS takes full control and take care of all the necessary processing at that time.

Due to the introduction of an IO thread, the copying between memory frame and swap space should all be done by the IO thread. So, when a page becomes free after its aging vector turns to 0, the memory manager should send an IO request to copy the page content to the swap space if the page is dirty. To avoid copying unnecessarily (if the freed frame is never used again before the system shutdown), we delay the copying till it is selected during a page fault. Thus, when a frame is selected for a page fault, no matter whether the frame is in the free list or in a process page table, the dirty bit of the frame should be checked and the copying should be done if the frame is dirty.

2.4 Input

A system configuration file “config.sys” should be used to set the configuration 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 command.c in the initialize_system function.

<observation mode>
-- if the observation mode is 1, it is on, more information should be printed;
-- otherwise, it is 0, and the observation mode is off
<quantum> <idle-quantum>
-- <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 size> <swap space size> <OS size>
-- page size is the size of each page in physical memory
-- 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>
-- specify the period in number of instruction cycles for the system to perform aging vector scan
<spool space size per process in bytes>

The system let you issue commands like a system administrator. The list of commands has been given in Table 4. One command is to submit programs and the program will be loaded to the system. The program is in a simulated assembly language. 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

Note: in the above, when multiple inputs are put in one line, it means the parameters should be read from one line.

2.5 Output

Your program should print all the required information very clearly. Do not print debugging information to garble the output. If any required information is not printed, we consider you did not implement that feature and will not give you credit for the implementation of that component.

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. Note that you need to modify command.c to call proper dump functions to perform the desired printing.

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. For each page, you can print the page number, its location in the swap space, and its content (only the parts that are in use). Note that you need to modify command.c to add a case to handle the dump swap command (w).

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.

If we turn on the observation mode, then additional information should be printed. First, upon each invocation of the compute_address function, the offset passed in and the physical memory address computed should be printed. You should also print the information regarding the aging vector and dirty bit updates before and after the updates.

The IO device thread should provide a dump_IO function. The dump_IO function prints the IO queue. Under observation mode, whenever the IO performs the processing of a request, it should print the information regarding the request it is processing, including all attributes of the IO request.

3 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 *Makefile* that generates the executable "simos.exe" from your source code files. If you are not familiar with *Makefile*, please check the man page.
- ✓ 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.