CS 4348/5348 Operating Systems -- Project 2,3,4

The goal for implementing the OS simulator, simOS, is to let you get a deeper understanding of the activities going on in computers and operating systems. It may be challenging for you to implement the simOS system completely. Thus, we provide a working simulator "simos.exe", which simulates computing behaviors and implements simple OS resource management policies. Some parts of the source code are cutoff to let you reimplement them. Some components are to be enhanced and you need to write the enhancement code. We hope that simOS will give you concrete hands-on learning experience and reinforce what you have learnt from the lectures. More details about the simOS system and what you need to do are discussed in the following.

You can download the tar file containing the source code files, object files, sample input files, and simos.exe from the course page under my web page: https://personal.utdallas.edu/~ilyen/course/os/home.html. Some sample codes for system calls and some sites with good tutorials are also given in this page. Note that to ensure that you understand the important concepts in simOS, there may be exam questions related to it.

1 Overview of a Simulated Simple OS (SimOS)

1.1 Overview

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 (mainly for swap space), terminal, and clock. CPU drives memory (through load and store instructions) and other I/O devices. The resource management functionalities in SimOS include process management, memory management, swap space management, terminal management, and timer management. The main goal of a computer system is to support program executions. Thus, simOS includes end users (clients) who submit programs for execution. It also supports admin functionalities, allowing administers to issue commands to monitor the system status and make administrative controls.

In the following table, we list the components and the corresponding code modules in simOS. The code file column gives the specific file name for the module, and whether the source code file is complete or incomplete (need you to add code) or not included in the package (for you to implement).

module	code file	Description
simulated CPU	cpu.c, cpu.o (incomplete)	Executes instructions, which operate on simulated CPU registers and memory.
simulated memory	memory.c, (complete)	Support basic memory read and write functions.
process management	process.c + idle.c, process.o (incomplete)	After a program is submitted, a process is created, its instructions and data are loaded, and it will be placed in ready queue. During execution, the process may perform IO, encounter page faults, etc. This module manages the process for all potential activities and states.
memory management	paging.c (only paging.o)	In modern OS, memory mostly uses demand paging and the corresponding management functions are implemented in this module.
loader	loader.c, loader.o (incomplete)	Loader reads in the program (from the program file), checks its validity, composes the memory content for the instructions and data, then loads them into memory, as well as into the swap space.
swap space manager (on disk)	swap.c, swap.o (incomplete)	In demand paging, a process may not have all its instructions and data in memory. The part of the process stored on disk is considered as stored in the swap space. Swap manager manages the swap space.
simulated terminal	term.c, term.o (incomplete)	All the output from the program are sent to the terminal. This module simulates the terminal and its management.

simulated clock/timer	clock.c (complete)	We simulate the clock by counting the instruction cycles. Programs may set timers and the clock manages these timer requests. It sets interrupts upon expiration of timers.
system control	system.c	It reads in the system configuration parameters from "config.sys" and calls the initialization and exiting functions in other components to initialize or exit the system.
administrator interface	admin.c (to be enhanced)	The administrator can issue commands to simOS through this interface to control the system operations or observe the system behaviors.
client	submit.c (to be enhanced)	A client can submit a program for execution. We simulate such interactions and consider multiple clients.

To simulate the parallelized operations of the core computer and its IO devices, simOS implements three threads operating in parallel, the terminal, the disk for swap space, and the CPU-memory-clock. The terminal manager in term.c runs as a separate thread. The swap space management and the simulated underlying disk coded in swap.c runs as another separate thread. The main thread includes all other modules. Currently, admin and client are also included in the main thread, but they should have been separate entities and you will need to code them and separate them from the main thread.

All three threads (main, swap, and terminal) have a similar logical flow in the large. Each maintains a queue for queuing its service requests (ready queue, swap queue, and terminal queue). Let x represent any of the three threads. Requests are inserted to x's queue by some other entities in the system. Each request has an associated process (the work will be performed for that process). x removes a request from its queue and processes it. After processing a request, the process associated to the request is supposed to be placed in some other queue for further processing, unless the process has terminated due to completion or error. For example, after the terminal device finishes servicing a print request for process x, $x$should go back to ready queue.

Some issues in the general procedure discussed above need to be considered. When retrieving a request from x's queue, what if the queue is empty? Do we just skip the current retrieval? If we skip, what should be next. If we loop back to check the queue again, won't we repeatedly get into the same situation without any progress in the system? After processing a request, how shall x transfer the process associated to the request to another queue? Of course, simOS handles all these issues and the details will be discussed when the corresponding components are discussed.

1.2 Simulated Computing Components: CPU, Memory

CPU has a set of registers (defined in simos.h) and it performs computation on these registers. The CPU module in simOS simulates the instruction cycle in real CPU. It first fetches an instruction from memory and recognizes its opcode. For convenience, it also fetches data at the fetch time (should have been done at the instruction execution time) for instructions that do need data from memory (instructions store, sleep, exit do not need memory data). Next is the execute-instruction step and operations on registers are performed. Some special instructions, including print and sleep, are executed somewhat differently. For print, the cpu issues a print request to terminal to print the corresponding data. For sleep, the cpu issues a timer request to clock, expecting to be waken up after the sleep time. For the exit instruction, as well as when errors occur during fetching or execution of the instruction, the cpu sets the status of the process to the corresponding value so that other components of the systems can act accordingly.

The list of instructions, including its opcode, its operands, and how the system should process them, are summarized in the following table.

OP Code	Operand	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 (ifgo)	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 the 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", which is the operand
1 (exit)	Null	End of the current job, null is 0 and is unused

The list of CPU execution status can also be found in simos.h.

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 4 interrupt bits (defined in simos.h) and they are discussed in the table below. 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?)

Interrupt types	Description
cpu time quantum expiration (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 (clock) sets the tqInterrupt bit. The handler then sets the process status to ready and the process manager will subsequently move the process to ready queue.
IO completion (vector value = 2)	When IO of a waiting process completes (such as print or page fault), the manager places the process into an endIO list and sets endIO interrupt. There may be multiple processes with their IO completed but the bit will only be set once. Thus, all processes in the endIO list and should be put to the ready queue (by process.c). Note that when the sleep time is up, we also use endIO interrupt and endIO list, just for convenience, though this is not the case in real systems.
age scan timer (vector value = 4)	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.
page fault (vector value = 8)	If a memory page to be used by the currently executing instruction is not in memory, the at the fetch time, a page fault will be raised by memory.c. The interrupt handler simply calls page_fault_handler (in paging.c) to handle it.

Memory provides CPU the memory access 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 done in paging.c. You will learn more about this address computation when we discuss memory management. And, you will need to implement paging.c in a later project.

1.3 Coding for CPU

The simulated instruction executions in CPU have been removed and you are expected to implement them. For some instructions, you only need to simulate some register level operations, like the micro-instructions we have discussed in our lectures. But there are also some more complex instructions (refer to the table of instructions, including print and sleep), which will put the process to a wait state. For print, an IO request to the terminal should be issued for the process. For sleep, a timer request should be issued for the process so that the process can get awaken after the sleep time is over. During instruction execution, it is necessary to access memory. Retrieving the instruction and the data has been done in fetch_instruction, except for the store instruction because it needs to write to memory, not to read from memory. You need to take care of the memory write when you code for the store instruction. You also need to check the return value (could be normal, error, or page fault) and perform proper actions for various return values.

When there is a page fault or error or when the program executes a special instruction, such as print and sleep, the process can no longer continue execution and the execution status of the process needs to be properly updated to let the system know what subsequent processing steps should be taken.

An interrupt may impact the execution of the running process. One of them is time quantum expiration (tqInterrupt). You need to write the interrupt handling code to handle tqInterrupt. For tqInterrupt, the current running process should be stopped and placed back to the ready queue. Actions such as queue insertion, process status update, etc., should be coded accordingly.

Other processes in the system, not the current running process, may also generate interrupts due to IO completion or sleep time expiration. You also need to write the code for handling endIOinterrupt.

The code in memory.c is complete, though it calls some functions in other modules (e.g., paging.c) and the implementation will be required later.

1.4 Process Management

In process.c, simOS 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 policy.

When a new process is submitted, the submit_process() function (in process.c) is invoked, which calls load_process in loader.c to load the instructions and data of the process into memory, prepares the process control block PCB, and places the process in the ready queue.

The process manager retrieves the first process from the ready queue and executes it. You will need to implement most of the code for process execution. The details will be discussed in the coding section. Here we look at some other issues.

What if the ready queue is empty when the process manager tries to retrieve a process from the ready queue? In real systems, the process manager (cpu scheduler) will let the cpu run an idle process when the ready queue is empty. The process manager in simOS does the same. The idle process is initialized and executed in process.c, but all the coding for the idle process, including initializing, loading, and execution are given in idle.c.

In a realistic system, process retrieval from the ready queue and process execution form an infinite loop, till the system shuts down. But we do not want to let our simOS run forever. Thus, we use admin commands to control the number of execution steps to perform. In each step, execute_process() calls cpu_execution() once and cpu_execution() will continue executing instructions of the running process till it encounters error, exit, page fault, IO operation, or time quantum expiration.

The process manager also manages an endIO list. During program execution, an IO instruction (or a page fault) will cause the process being placed in the queue of the IO device. When the IO for the process has finished, the process is supposed to go back to the ready queue. Also, the system needs to update the PCB to reflect the status of the process. Instead of letting the IO device manager do all these processing voiding modularity, the IO device only raises an **endIO** interrupt and calls insert_endIO_list (provided in process.c) to place the process in the endIO list. When the interrupt handler is invoked, it calls the process manager to move all processes in endIO list to ready queue (endIO_moveto_ready) and perform the corresponding updates in PCB for the process.

1.5 Coding for Process Manager

In the process manager (process.c), you are supposed to implement the main part of execute_process(). This involves performing context switch, invoking cpu_execution, and checking the process execution status in PCB and deciding what to do next. You also need to keep track of the execution time (the number of instructions executed) and add it to the total time used by the process (PCB[?]->timeUsed).

You have learnt context switch and the code for context switch functions is also left for you to implement. Essentially, you need to select the correct set of registers to save and the correct set for restoring.

For both cpu.c and process.c, the string "*** ADD CODE" is added to the cutoff parts. You can easily find out whether there are more coding works to be done by searching for the string.

1.6 Loader

Similar to real systems, a loader loads user programs into memory. In simOS, loader.c implements the loader functionalities.

The first part of loader.c opens the program file, reads in one program instruction or one data, and write the instruction/data to a buffer at the offset position.

In the second part, the loader calls the load_instruction/data function to load the instruction/datum one at a time, to a page buffer, till it is filled or till the program instructions/data are exhausted. Then, the loader writes the filled page buffer to the swap space by inserting a write request to the swap manager. Also, the process page table needs to be updated to record that the page is loaded to the swap space.

During any stage of loading, if a program error is detected, the loading procedure terminates and the loader returns "progError".

After all the pages are loaded to the swap space, the loader swaps some initial pages from the swap space to memory by calling function swap_in_page (in paging.c). The number of pages that can be loaded to memory for each process is limited by loadPpages. If a process has less than loadPpages, then all its pages will be loaded to memory. Variable loadPpages is declared in simos.h, its value is given in config.sys, and it is read in by system.c. Since the program validity has already been checked when loading the program to swap space, there will be no program error when swap in the pages to memory.

1.7 Coding for Loader

The code for loading instruction page buffers to swap space is given. You will need to write code to load data page buffers to swap space. Note that since the last page to be filled with instructions may not be full, data loading needs to continue on this partially filled page. If not, the address computation for data will be a problem.

You also need to code for load_pages_to_memory. We have computed the number of pages to be loaded to memory. You need to swap-in pages 0 to lastpage to memory by calling swap_in_page with proper parameters.

1.8 Terminal

Terminal manager (term.c) outputs output strings from the processes to the monitor. Note that the output from simOS components, not from the user programs, do not go through the terminal manager for printing. When CPU (cpu.c) processes the print instruction, it puts what is to be printed as a string and sends the print request together with the string to the terminal queue. When a process finishes, process manager (process.c) prepares an end-program message and also sends it to the terminal queue to be printed. The interface function for inserting the printing requests to the terminal IO queue is **insert_termIO** in "term.c" and it is only invoked from cpu.c and process.c.

The terminal thread manages the termIO queue. It also retrieves the printing requests in the queue and process them. The terminal IO is simulated by printing the given string to a file "**terminal.out**". After issuing a print request, the process owning the print request is switched to the waiting state. When the print request finishes, the process should be sent back to ready queue to continue its execution (unless the process is to exit). As discussed in process management, IO devices cannot directly place a process in ready queue. Thus, the terminal manager raises the endIO interrupt and places the process in the endIO list.

When issuing a terminal output request, the issuer needs to provide the process id (pid) as well as the string to be outputted (outstr). Moreover, the "type" of the output request shall be given, which can be regularIO, indicating that the output is generated by a running program, or exitProgIO, indicating that the output is for an exiting process. The terminal manager, after processing the output request, only puts the process to the endIO list for the former case, not for the latter.

1.9 Swap Space

A large program that has too many instructions may not fully fit in the memory allocated to the process. Similarly, a data intensive process may need a lot of memory space for its data. Thus, it may not be possible to put all the pages of the 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). Swap space manager has a disk segment, simulated by a file named "swap.disk", which is used to store the memory pages of

each process to facilitate their loading into and unloading from the memory. The management of the swap space on disk is implemented in the first part in swap.c. It is an over simplified solution where each process gets a fix-sized swap space with a fixed allocation (starting location is computed by process pid). The main functions offered by swap disk interface include initialize_swap_space, read_swap_page, and write_swap_page. To simulate the high latency in disk IO, we define a diskRWtime (in microseconds). At the end of read/write_swap_page, a sleep time diskRWtime is taken.

The swap space manager is implemented on top of the swap disk in the swap thread (swap.c). In fact, only diskIO should be a separate thread, and the swap space manager should be in OS (main thread running on CPU). But we anyway put the swap space manager in the swap thread.

The swap space manager processes swap requests issued by other components in the system. Naturally, it manages a request queue, swapQ. When a program is submitted, the loader loads the process pages into the swap space and some pages are swapped into the memory. When a memory page is to be used during process execution, but it is not available in memory, a page fault will be raised and the interrupt handler inserts a swap-in request to the swap space manager to ask it to swap in the needed page. Swap manager will load the page from swap space to memory. If the memory space is not sufficient, a page of a process may be swapped out from memory (by paging.c). In this case, a swap-out request will be issued to the swap space manager. The interface function for inserting the swap requests is **insert_swapQ** and it is only invoked by the loader (loader.c) and the memory manager (paging.c). In summary, each swap request can be a write request (for a page being swapped out), or a read request (for a page being swapped in), and is flagged by the field "act", which can be actWrite or actRead, respectively.

In a swap request, we also need to consider another parameter, the "finishact" flag, which specifies the action to be taken after the swap request is done. The finishact flag can be: (a) toReady, which puts the process back to ready queue, (b) freeBuf, which frees the swap input buffer for write (never free the output buffer for read), (c) Both (both toReady and freeBuf), and (c) Nothing (do nothing).

1.10 Coding for Terminal and Swap Space Managers

Since terminal is an IO device working in parallel with CPU, we need to run terminal manager as a thread. Note that inserting requests to the terminal queue is done by cpu and process manager in the main thread and the terminal thread removes requests from the queue. Thus, their accesses may cause conflicts and need to be synchronized. The coding for thread creation and synchronization are left for you to complete. The same thread creation and synchronization need to be done for the swap queue accesses as well. You need to ensure that accesses to the shared queue will be executed mutual exclusively.

Remember the question regarding what to do when the terminal or the swap thread retrieves its queue while the queue is empty. We need to use a semaphore to block the thread when the queue is empty and wake up the thread when the queue is no longer empty. You also need to ensure that the thread will not be blocked when the queue is not empty. You are required to code this synchronization mechanism for both the terminal manager and the swap space manager.

Besides thread creation and synchronization by semaphores, one more work to be done for terminal and swap space managers is to complete the code in request processing. Upon finishing processing a request, the associated process should be pushed to another queue and some corresponding interrupt should be set. You need to write code for it and the "*** ADD CODE" string is there to indicate where the code is missing.

Note that for synchronization, you need to know where to insert the semaphore code. So, there are no "*** ADD CODE" directives to indicate where to insert the synchronization code.

1.11 Clock

Clock keeps track of system time. To make the control of the system execution easier, we use the number of CPU cycles as the clock time in simOS. If we use real time, we will not be able to control the time quantum. So, in simOS, CPU calls clock to advance the time after each instruction cycle. In real systems, clock, after a certain

number of its own ticks, updates a clock register in CPU. In simOS, CPU.numCycles is the clock register and it is incremented by clock after each instruction cycle.

Clock provides a function (**add_timer** in clock.c) to let other components of the system to setup a timer event. Each timer request includes the **time** (relative, counting from current time), the process id, the action to be performed after the time is up (**action**), and the periodicity of the timer (**recurperiod**). When the time is up, the timer manager takes the specified action. The supported actions are discussed in the following table. Clock also automatically inserts the timer request 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). This returned pointer may be used for locating and deactivating the timer when needed (by deactivate_timer).

The actions that may be performed after the timer expires are given in the following table

Action Code	Corresponding action
1 actTQinterrupt for time quantum	Set the tqInterrupt bit and put the process in endIO list, to be sent back to the ready queue
2 actAgeInterrup for age vector scan	Simply set the ageInterrupt bit, to let the memory age scan be performed periodically (system activity, no process involved)
3 actReadyInterrupt for regular situations	After timer expiration, the process goes back to ready queue, this is mainly used by the sleep instruction
0 actNull no action is needed	Do nothing, this is mainly used for timer deactivation, to avoid more expensive actions of removing the event from within the data structure

The recurperiod for the timer is simply the period that the corresponding event should recur repeatedly. In most of the cases, it should be the same as the time given for the timer. If a timer request is not periodical, then oneTimeTimer (= 0) can be used when add the timer request by add_timer.

There is nothing to be done for clock.c because the topics for clock will be covered at the end of the semester. But you do need some information given in clock.c and in this section when you code other modules which interacts with clock.

1.12 System Configuration, Initialization, and Termination

All the definitions that are needed in multiple code files are defined in one header file, simos.h. Each global variable and each global function are briefly explained, which probably can provide some high level view of the overall system and the interactions between system components.

In system.c, all the system configuration parameters are read in from “config.sys”. This is to provide a centralized initialization of important system parameters, instead of letting each system component inputs its own parameters. These system parameters are defined and briefly elaborated in simos.h under system.c section. Function initialize_system in system.c calls the initialization functions of individual components to initialize the system. Function system_exit in system.c calls the exit functions of individual components to cleanly exit the system.

There are three sleep time parameters read in by system.c from config.sys, including **instrTime**, **termPrintTime**, and **diskRWtime**, which defines the delays in CPU instruction execution (cpu.c), terminal printing (term.c), and disk read/write (swap.c) to allow us to control the speed of the system execution when needed.

All the simOS system outputs go to two channels, the normal information file descriptor “**infF**” and the debugging log file descriptor “**bugF**”. The information file includes important system messages, such as uncommon simOS errors, notification of system activities, and printouts for requesting for input data. It is currently set to “**stdout**”. The debugging log file is used for all system components and it is set to “**debug.tmp**”. The assignment is hardcoded in system.c. You can change the files associated to infF and bugF in system.c. Note that admin.c also assigns an output channel for all its dump requests, which is set to stdout currently and can be reassigned to other output channels.

There are 6 debugging control variables (read in from config.sys) to control whether to print debugging messages for related system components. For example, memDebug controls whether to print debugging

information from memory.c and paging.c. These debugging control parameters are also read in from config.sys at the initialization time in system.c.

2 Project 2 Description

Besides getting familiar with simOS and completing the missing code, you are also expected to learn Unix system calls and get a better understanding of them. In Project 2, you need to be able to fork new processes and use pipes to communicate between the processes. You also need to learn Unix system signals and signal handlers. The three phases of Project 2. Unix signals are interrupts. To differentiate from the interrupts in simOS, we use signals to refer to Unix signals (interrupts).

2.1 Phase 1. Implement Code for CPU and Process Manager

As introduced in Section 1, you need to complete the code for simOS so that it runs the same way as the original simulator and supports correct program executions. In this phase, you need to study how the instructions operate on registers, how interrupts work, and the concept of the process control block and process states. Then, you can complete the code in cpu.c and process.c. You also have admin.c, system.c, submit.c, memory.c, and clock.c that are complete without needing any modification. You can compile your cpu.c and process.c with admin.c, system.c, submit.c, memory.c, and clock.c and use the loader.o, paging.o, swap.o, and term.o files to generate simos.exe. The makefile given to you for the simOS system will generate simos.exe if all source code files are complete. You need to modify it to exclude the incomplete and missing source code files and generate simos.exe.

2.2 Phase 2. Input Files to simOS and Testing

The executable simos.exe has no command line input (just type "./simos.exe" to execute it). There will be run time inputs for it and you can reference admin.c or Section 2.3 to see the available commands.

At initialization time, simOS reads in "config.sys" to configure the system. You need to get familiar with the systems parameters given in config.sys and change them during testing. You need to read the code to understand the impacts of these parameters to the system.

The submit command ("s") will require you to give the name of an input user program to simOS. A user programs can have any name of your choice. Currently, you have prog1 in the tar package. To thoroughly test your systems in this and subsequent phases, you need to write new programs for submissions. You can check the sample program "prog1" and the first part of "loader.c" to see how to write the user programs. The first line of the program gives three integers: the memory size for the program, the number of instructions, and the number of data entries. We have #instructions + #data-entries = memory-size. In fact, memory-size is redundant, but we will just leave it there.

After the first line in a program are the instructions with each instruction on a line. The first integer of each instruction is the opcode (all opcodes are defined in cpu.c) and the second integer is the operand, which gives the address of the data to be operated on by the instruction.

In a real executable, all the addresses in an instruction is relative to the beginning of the address space (some system consider the offset from the current instruction, but it is not the case in simOS). In simOS, the addresses are offsets from the beginning of the address space. Considering that the first instruction is of address 0, the address of the first data would equal to #instructions. Thus, the address in the operand may be any number between #instructions to memory-size – 1. Each data occupies a line. If a data is not defined yet before the execution of the program (e.g., a memory word to be written to in the program), you need to put "0" or any arbitrary number in the corresponding data line.

You need to submit your testing user programs together with your system on elearning and your input programs will also be evaluated. The sample program prog1 only performs some arbitrary actions. You can write a simOS program that is more meaningful. But since there is no input statement in the instruction set, the capability of the simOS program is limited.

2.3 Phase 3. Admin Command Processing via Fork, Pipe, and Signal

We simulate the administrator interface and activities in admin.c. Specific admin commands can be found in admin.c and are also listed in the following table.

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 till some event stops it
y (execute)	r	Repeat what is done for x r times
r (register)	-	Dump registers
q (queue)	-	Dump ready queue and endIO list
p (PCB)	-	Dump PCB for every process in the system
m (memory)	-	Dump the page table of each process, for all processes
f (memory frame)	-	Dump the metadata of all memory frames, frame by frame
n (memory)	-	Dump the contents of the entire memory, frame by frame
e (timer events)	-	Dump timer event list
t (term)	-	Dump the terminal request queue
w (swap)	-	Dump the swap request queue

In current admin.c, the input command is read in by scanf. This means if the admin issues "y 100000", then it is not possible to issue any observation command while the system is in execution for 100000 rounds. We would like to change the admin code so that it runs asynchronously with the execution of the processes so that during process execution, admin can issue commands to observe the system behaviors.

The goal above shall be achieved by writing a new admin interface process (**adminUI.c** which compiles to **admin.exe**). Upon initialization, adminUI.c **fork**s a child process which runs simOS (better not to use exec). After fork, the child simOS process should initialize the system, run execute_process for a large number of rounds, and call system_exit to exit simOS. To simulate real computer systems, we would like to let simOS execute processes in an infinite loop. However, if your program is not designed properly, you may end up having a process that never terminates and you need to kill it externally. So, we give a command line input to admin.exe, **numR**, which is the number of rounds execute_process should be activated (instead of having an infinite loop).

When the adminUI.c reads a command from the administrator, it should send the command to simOS. We use pipe to pass the command from admin interface to simOS. But this way, simOS still need to be blocked on a read from pipe. We use Unix system interrupt (signal and signal handler) to achieve the goal. Before admin forks, two pipes shall be created to support two-way communication between the admin interface process and the simOS process. The admin commands will be issued with a signal (not simOS interrupt) and a pipe to simOS and processed by a Unix signal handler (not simOS interrupt handler) in simOS. The flow of the admin.exe program is: (1) read admin command from administrator, (2) sends a signal to simOS process via **kill**(…) system call, (3) sends the command to simOS process via the parent write pipe, (4) waits for the response from simOS via the parent read pipe, and (5) display the simOS output to the monitor.

The interface for admin.exe should be the same as the original simOS.exe, except that the administrator can no longer issue "x" or "y" commands.

You need to make a few changes to simOS to enable the communication with adminUI.c and to allow admin commands being executed while the processes are in execution.

(A) In simOS, you need to define a new interrupt type and (say adcmdInterrupt) and give it a proper value in simos.h. The purpose of this interrupt is to let the system switch to serve the admin command when it is set. Then, you need to modify handle_interrupt in cpu.c to handle adcmdInterrupt. The interrupt handler simply needs to activate the current admin command processing function. Thus, you should implement the handler in admin.c.

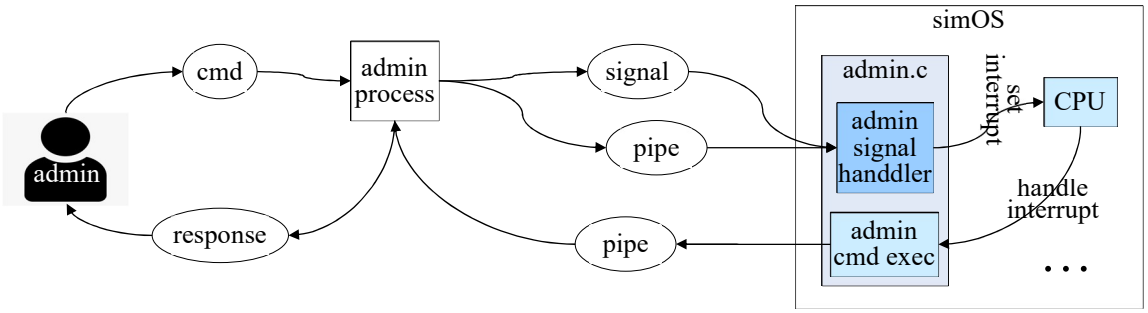
(B) In admin.c (in simOS), you should remove the admin commands "x" and "y". You do not need to call execute_process for a single round. The function execute_process_iteratively should now be invoked at the system initialization time. Also, you need to capture a Unix system signal (not simOS interrupt) by associating a

signal handler to a selected signal number (use the **signal** system call). You can choose either of the signal numbers: SIGINT, SIGRTMIN, or SIGRTMAX for the purpose. This can also be incorporated in admin.c.

Upon receiving a signal, your signal handler should: (1) read the admin command from the child read pipe, (2) raise adcmdInterrupt.

(C) The output from simOS should now be directed to a pipe and sent back to the admin process for displaying. The admin process can simply create a pipe for the response communication and use the pipe as the file descriptor for all the dump functions. The admin process can then receive the responses and print them out on the monitor.

2.4 Illustration



2.5 Submission

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

- ✓ All new source code files making up your solutions to this assignment. In this project, it should include cpu.c, process.c, system.c (optional), admin.c, and adminUI.c.
- ✓ If you implemented the admin interface process with multiple source code files, you need include them. Also, you should list these files in the "readme" document and specify the functionalities of each.
- ✓ Your updated makefile that will generate admin.exe.
- ✓ A "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 have changed source code files we provided (other than those that you are supposed to change), then list the files. For each file being changed, discuss what changes are made and why you need to make those changes.

3 Project 3 Description

There are two major focuses in this project. First, you need to study the pthread library and the use of semaphores for synchronization. The first two phases are about using semaphores to achieve different synchronization goals. Then, you need to learn to program sockets to enable user-simOS communications.

3.1 Phase 1. Implement Code for Swap.c, Term.c, and Loader.c

Youo need to study the code for swap space manager and terminal manager and complete the code for swap.c and term.c, including completing the end of request processing actions as well as thread creation and synchronization. First, you insert code for performing necessary actions after a swap/terminal request has

completed processing. Second, in start_swap_manager and start_terminal, you need to create threads so that the managers run as separate threads. You also need to define and initialize semaphores. Finally, you add sem_wait and sem_post functions in the code to ensure proper synchronization between the threads.

Your synchronization solution should ensure that the accesses to termIO queue and swapQ are mutual exclusive. You also need to properly handle the problem of waiting for an empty queue. Note that handling this synchronization issues is not just to learn how to program for concurrent threads, but also to see how the concurrent control examples introduced in the lectures can be applied to practical problems. So, you are required to use integer semaphores only (with sem_wait and sem_post and sem_init only). You are not allowed to use more advanced synchronization functions offered in pthread library (of course it will be fun to know what other functions are available in thread libraries and you can study them). You can reference the bounded buffer problem and modify the solution for this synchronization and mutual exclusion problem.

Next, you should study how the loader works with swap space manager to accomplish the loading of a program. There are a few questions you should think over regarding loader.c and swap.c interactions. (a) The loader continuously sends requests to swap manager, asking swap manager to write the buffer content to swap space. Before the swap space manager finishes all these requests (write), the loader issues requests to swap manager to load the program from swap space (read) to memory. Will this cause synchronization problem and mess up program loading? (b) The loader, before ensuring that the program pages are loaded to the swap space, calls memory manager to update the process page table to reflect that the to-be-loaded page is already in the swap space. Will this cause any problem? (c) When loading a page to swap space, we first load instructions/data to a page buffer, and then give the page buffer to swap space manager for loading. Why does the loader use a new buffer for every page, instead of reusing the same buffer after loading? Also, how is the buffer space freed? (d) How does loader know when the loading is done so that it can send the newly loaded process to ready queue? (e) Is there any better way to code the loader considering that disk IO has to be asynchronous?

You can complete loader.c after all these issues are clear to you. Do not just code loader.c without understanding these points. After completing loader.c, you can compile all your C code files with paging.o to generate simos.exe.

3.2 Phase 2. Correct Sync for EndIO List Accesses

In process.c, the functions for accessing the endIO list have been implemented, including function insert_endIO_list, which is called by swap.c, term.c, and clock.c, and function endIO_moveto_ready, which is invoked locally by process.c. Similar to the case in swap.c and term.c, you need to ensure mutual exclusive accesses to the endIO list. But in addition, we prioritize these functions.

Since function endIO_moveto_ready may move multiple processes from endIO list to ready queue, which may be slower, we give it a lower priority than insert_endIO_list. In other words, if two threads are trying to perform endIO list insertion (insert_endIO_list) and endIO list removal (endIO_moveto_ready), then the insertion thread gets to go first, unless the removal thread has already started its access. If multiple threads are trying to perform list insertion simultaneously, then the first comer gets the right to do its insertion first (no priority between insertion threads). You need to use integer semaphores to achieve this prioritized synchronization and mutual exclusion. Same as Phase 1, you are not allowed to use more advanced synchronization functions offered in pthread library, just semaphores and its sem_wait and sem_post and sem_init functions. You can reference the reader-writer problem introduced in the lectures and come up with a specific solution for this case.

3.3 Phase 3. Multiple Clients Submitting Programs via Sockets

Another change to simOS is to make it support multiple clients. Each client can submit his/her program to simOS via his/her own window. Even though simOS is now executing processes in a large number of iterations, the submit request from clients should be taken care of without much delay. The needed changes to simOS are as follows.

(A) Now you need to remove the "s" command from process_admin_command in admin.c and let client submissions be handled by submit.c.

Modify submit.c to run it as a submit thread in simOS (similar to what you do for swap.c and term.c). The submit thread should be able to receive requests from multiple clients. We use socket to achieve this task. In the submit thread, a server socket is created. It listens on a port. When a submission request is sent from any client to the port, the server socket receives the submission requests (input message) and process it. For each submission request, the submit thread raises the **submitInterrupt** (or any name you want to use). The request is then processed by the interrupt handler, which calls "**submit_process**" (provided in process.c) to create a PCB for the process and to load the program into swap space and memory.

Note that you need to keep the submission information somewhere so that upon interrupt handling, you can access the corresponding submission information. Since there may be multiple clients, multiple submission requests may be issued at the same time. But submitInterrupt is only one bit and multiple submissions will only raise it once. This is different from admin's case since there will only be one admin. Thus, when handling submitInterrupt, you need to consider processing multiple submissions. Specifically, you need to maintain all the awaiting submissions in some data structure. You can implement the data structure by yourself, or copy and modify one of the queues implemented in simOS, or use a library for the purpose.

For the program submission requests, you are not supposed to use signal to handle the request in a signal handler. You need to use socket and a **single** submit thread to achieve the task. To make your code flexible, and to make TA testing easier, you need to give the port number for your server socket as a command line input. After Project 2, your main program becomes admin.c and we run admin.exe for the simOS system. Remember that admin.exe had a numR command line input. Now, you add the port number as the second command line input to admin.exe. E.g.,

./admin.exe 1000000 21234

(B) similar to the change for admin, you need to add the definition for submitInterrupt in simos.h and add the call to the submission interrupt handler in handle_interrupt().

(C) You also need to implement a separate submission client program, **client.c** (compiled to **client.exe**), to interface with the clients for program submissions. Each client runs the client program on its own window. The submission client program reads in the client submission information and wraps it as a request and sends the submission request to the server socket established by the submit thread (in A). Note that since the submission client can run on any computer, the program a client submits should be transferred through the communication channel to the submit thread. You cannot just send a file name and assume that the file is accessible by simOS.

The submission client program can be implemented in any language, as long as you can communicate with the server socket properly. In Java and Python, some types of output streams may be formatted differently from those in c or c++ and you need to use the proper data stream and you need to make sure that the messages communicated between the client and simOS are compatible.

(D) With the submission client program, each client can have its own window for submission. At the same time, any printout from the program should be sent back to the client and get printed in the client window. Thus, you need to change the simulated terminal in term.c also. First, the "**terminal_output**" function, which originally outputs the output strings from programs to file "terminal.out", now should send the output string to each specific client via a specific socket. Upon accepting a client connection, the submit thread should retain the socket information that is dedicated to the client. The client and socket information can be maintained in PCB (remember PCB has a pointer pointing to files and devices in use). This way, terminal manager can easily access the socket information and use it for the corresponding client.

(E) Your client program needs to know the IP address and port number of the server socket in simOS (or admin.exe). Thus, the IP address and port number should be provided as command line input to client.exe. The sample client execution and program submission for client.exe is as follows:

```
> ./client.exe 129.110.242.180 21234        (IP and port are samples)
> s prog1                                   (s: submit, prog1 is the program file name)
pid=2, M[10]=10.00
```

pid=2, M[11]=60.00
Process 2 had completed successfully: Time=15, PF=5

The blue text shows the sample outputs from simOS, sent from term.c. Only outputs for the program should be returned to the client via the established socket. The client program should wait on its socket for outputs and print the outputs to monitor. But the client program cannot have an infinite loop to wait on the socket for the outputs. You need to find out all the termination outputs from process.c and detect them to terminate the output waiting loop. After received all the outputs, the client can close its connection and terminate. The server, knowing that the client program has terminated, will close the specific client socket and clean up for process exiting.

3.4 Phase 4. Continuous Client Submission and Select System Call

For convenience, we will allow each client connection to act like a session and allow the user to submit multiple programs. The connection should be closed after the user decides not to submit any new programs. Thus, your client program should accept commands as follows:

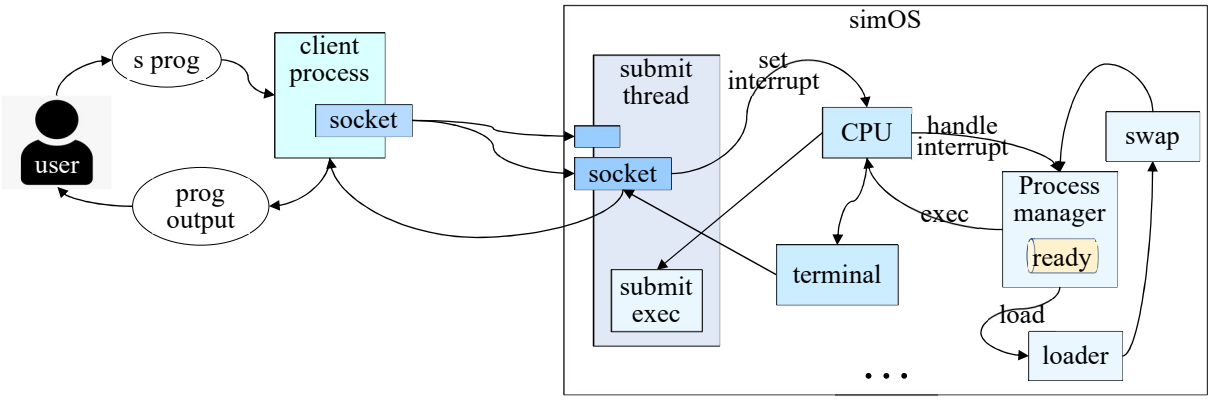
```
> ./client.exe 129.110.242.180 21234          (IP and port are samples)
> s prog1                                     (s: submit, prog1 is the program file name)
pid=2, M[10]=10.00
pid=2, M[11]=60.00
Process 2 had completed successfully: Time=15, PF=5
> s prog1.err
Program prog.err has loading problem!!!
> s prog2.err
pid=3, M[10]=10.00
Process 4 had encountered error in execution!!!
> T                                           (terminate, the client will no longer submit any more programs)
```

After a user submits a program, the client program will not prompt to the user till the received output indicates the termination of the program. The client uses command “s” for program submission, and “T” for terminating the submission activities.

In Phase 3, we only consider one submission from each client. Thus, after accepting a client connection, the submit thread can go ahead and read the client submission message. Since it only needs to read once from the client, we let the thread be blocked by the receive function. In other words, the submit thread can only receive one client submission at a time, even though the main thread may be executing other programs and/or responding to other programs in parallel.

We do not want a multi-thread solution for this case. The required solution is to use the **select()** system call, which allows you to wait on multiple sockets (or any other types of file descriptors) at the same time and receives some message(s) when one or more are available. Now you have a complete simOS that can handle multiple clients.

3.5 Illustration



3.6 If you skip Project 2

In case you did not complete Project 2 successfully, you can work on Project 3 without it. But you have to complete the first phase of Project 2, i.e., complete the code in cpu.c and process.c. Otherwise, you will not be able to change the interrupt handling capability for this project.

Instead of running admin.exe, you just need to run simos.exe and do the same modifications as discussed in this section (Section 3). You do need to give simOS.exe one command line input, the port number for the server socket.

If you wish to skip the early phases of Project 3, you can choose to skip Phase 2, which does not impact Phases 3 and 4. But you do need to have a complete term.c in order to proceed to Phases 3 and 4 because you need to change the code in term.c and you need a complete version to start with.

3.7 Submission

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

- ✓ All new source code files making up your solutions to this and previous assignments. In other words, any files that are new or different from the downloaded tar file should be submitted. It should include cpu.c, process.c, admin.c, system.c (optional), swap.c, term.c, loader.c, admin.c, adminUI.c, submit.c, and client.c.
- ✓ If you implemented the admin interface process with multiple source code files, you need include them.
- ✓ If you implemented the client program with multiple source code files, you need include them. Also, you need to list the files in the "readme" document and specify the functionalities of each.
- ✓ Your updated makefile that will generate admin.exe and client.exe.
- ✓ A "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 have changed source code files we provided (other than those that you are supposed to change), then list the files. For each file being changed, discuss what changes are made and why you need to make those changes.

4 Project 4 Description

For Project 4, you need to implement a simulated virtual memory with demand paging in paging.c. 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. You can also directly go for the final implementation without going through the phases, but it may be more challenging and phase by phase implementation does not really require additional coding. The detailed description about the three phases are as follows.

4.1 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 is given in variable 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.

During program execution, the addressing scheme needs to be implemented to allow the system to fetch the correct physical memory. You can implement "calculate_memory_address" 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 implement the memory dump functions to dump various memory related information. …

4.2 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. 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 and raise the page fault interrupt pFaultException. The page fault interrupt handler, **page_fault_handler**, is called in cpu.c, but the detailed implementation should be given in paging.c.

In page_fault_handler, you need to ask the swap space manager to bring the page into memory by inserting a request to swap queue. But before doing that, the memory manager needs to allocate a new memory frame for the process at page fault. 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 free 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 and subsequently, page_fault_handler will be called. Subsequently, process.c checks exeStatus and put the process into wait state.

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. This should be done in the calculate_memory_address

function. You should only write a dirty page of an active process (not a terminated process) back to swap space. Also, you need to be careful with the order of the swap requests, i.e., you need to send a request to write the dirty swap-out page first, and then read the swap-in page to replace the content of the selected memory frame.

4.3 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. 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 without freeing it. If the dirty frame is the only frame with the lowest age, then select it to be swapped out.

At the system initialization time, you need to set a timer with the desired scan period, agescanPeriod (the period for the periodical age scan). Also, the timer should be of type “periodical”, which is indicated by setting

When the time is up, the ageInterrupt bit shall be set (by clock.c). Then, the scan activity will be invoked by the interrupt handler calling memory_agescan. At the same time, clock will automatically insert a new timer to its event list for the next age scan activity.

4.4 Printout

To facilitate observing your program behavior, you 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 can call the dump page function in swap.c 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.

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.

4.5 Testing

Memory management can be complex. So, you need to generate many different cases to thoroughly test your paging.c code. You should also change the relevant parameters in config.sys in your tests.

4.6 If you skip Projects 2 and 3

In case you did not complete Project 2 or Project 3 successfully, you can work on Project 4 without them. In fact, if you wish to simplify your testing, you can download the tar file again and start your project from the original package. You just need to write the code for paging.c for this project. In fact, to simplify testing, we will test your system by integrating your paging.c with the original simOS.

4.7 Submission

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

- ✓ The new source code files making up your solutions to this project. It should include paging.c and other code files that are for memory management.
- ✓ Your updated makefile.
- ✓ A "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 have changed source code files we provided, then list the files. For each file being changed, discuss what changes are made and why you need to make those changes.