CS 4348/5348 Operating Systems -- Project 2

Project 2 modifies Project 1 by incorporating more OS components and functionalities in the computer system. Also, in this project, you need to learn to use some important Unix system calls.

The changes in Project 2 include: (1) The system should support the submission and execution of multiple processes. Thus, the shell needs to support a submit command. (2) A computer only terminates when you shutdown, so in your simulated system, you need to implement a terminate command. (3) You need to implement a CPU scheduling algorithm to determine which process to execute next and fetch it from the ready queue. (4) Project 2 introduces a new system component, the printer. For a print instruction, the printout is displayed on the monitor in Project 1. Now we consider to also print to a printer. The printer is a separate entity in the system, not a component of a computer. Thus, we use a separate process to simulate the printer.

1 Shell

Shell commands are not instructions. In Unix, a process is created to run a shell for each user. In Project 2, we consider a single user issuing shell commands to the shell interface. Additional shell commands should be added in shell.c and they are listed in the following table.

Action	Project	System actions
0 (terminate)	Project 2	Terminate the entire system
1 (submit)	Project 2	Submit a new process (need to get the program file name and Base)
2 (register)	Project 1	Dump the values of all registers
3 (memory)	Project 1	Dump the content of the entire memory
4 (readyQ)	Project 2	Dump the information of all processes in the ready queue
5 (PCB)	Project 2	Dump the information of all processes in the list of PCB
6 (printer)	Project 2	Dump the PIDs of all the processes with their spool files open

For each command, you need to implement a function for command processing. Like in Project 1, the main function in shell.c, namely, shell_command(…), simply process the command by calling the corresponding function. The design for each shell command processing function is given as follows.

- Submit. When the user issues a submit command, shell should proceed to prompt the user to get the program file name and its Base (this was implemented in computer.c in Project 1, and should be move to shell.c now). Loading the program into memory should also be moved to shell.c. Subsequent processing, such as creating PCB, etc., can be handled by calling the process_submit() function in scheduler.c.
- Terminate. the shell simply sets a termination flag to indicate that the system should terminate. The shell will stop accepting new shell commands. The CPU will continue its execution, but when the scheduler is in action, it will not fetch processes from the ready queue and will proceed to terminate. A termination request should be sent to the print module and then the printer.
- Dumping commands. The format for displaying the system states will be provided by TA.
- You can leave your original implementation for instruction 9 in cpu.c without any change.

Note that shell is going to be a thread, thus, shell_command() will be a thread function.

2 CPU and Memory

The simulated CPU and memory are the same as your previous cpu.c and memory.c, except for the cpu_operation() function.

In project 1 CPU execution in cpu_operation() loop terminates on program exiting. In Project 2, we consider CPU scheduling. Thus, cpu_operation() loop should also terminate upon time quantum expiration (discussed in the scheduler section). Also cpu_operation() now needs to return with a status: **Exit** or **TQexpiration**. This is to allow the scheduler to take the proper actions accordingly.

Please note, last time I did not put cpu_dump_registers(...) and memory_dump(…) in cpu.c and memory.c (any function names are fine). I expect you knew how to design, if not, please correct your design.

3 CPU Scheduler and Context Switch

Since we incorporate CPU scheduling to the system, initiating the execution of the new process should not be done by shell or computer, but should be handled by a new program **scheduler.c**, which implements the CPU scheduling algorithm. We consider the basic CPU scheduling algorithm, Round Robin, which would require a ready queue. After a program is loaded into memory, it should be inserted into the ready queue. CPU scheduler fetches a process from the ready queue and calls the CPU function cpu_operation() to execute the process. The functions to be implemented in scheduler.c are given in the following table.

process_init_PCBs(…)	Initialize a PCB data structure for PCBs of multiple processes.
process_init_PCB(…)	Create a PCB entry for a submitted process and give the process a unique PID (generally PID is in sequential order).
process_dispose_PCB(…)	Dispose a PCB entry for an exiting_process.
Process_dump_PCB(…)	Print out the information of all the PCBs in the PCB list.
process_init_readyQ(…)	Initialize a ready queue data structure.
process_insert_readyQ (…)	Insert the PCB of a newly submitted process into the ready queue.
process_fetch_readyQ(…)	Fetch a process to execute from ready queue based on the CPU scheduling algorithm. It returns the fetched PCB pointer.
process_dump_readyQ (…)	Print out the list of all processes in the readyQ (just the PIDs is fine)
process_context_switch (…)	Context switch two processes. Pass 2 PCBs as parameters. Need to consider the case of only switching one process in/out.
process_init (…)	Initialize the module by calling other init functions in the module, including process_init_PCBs and process_init_readyQ.
process_submit (…)	Handle process submission by calling other functions in the module, including process_init_PCB, process_insert_readyQ, and print_init_spool. (called by shell.c)
process_execute (…)	Handle process execution by calling other functions, including process_fetch_readyQ and cpu_operation (in cpu.c). If cpu_operation returns as Exit, then call process_exit.
process_exit (…)	Clean up for the exiting process, including calling dispose_PCB and calling print to print_end_spool, etc. This should be done for all remaining processes upon termination.

The round robin algorithm has a time quantum. Instead of considering real time, we use number of instructions for the purpose. A system parameter $\boldsymbol{TQ}$ defines the time quantum and should be given in "config.sys".

Context switch is a very important step in process switch. You need to perform context switch when a process is switched in or out (not exit).

The scheduler will continue fetching processes and let CPU execute them forever (like real computers). The scheduler terminates only when the termination flag is on (turned on by the shell). The check of termination flag is right before fetching from ready queue. Upon termination, scheduler needs to inform the printer to also terminate by calling print_terminate().

In your PCB, besides other necessary information, also include the file name of the program for this process. This is not the case in real systems, but for grading purpose. This will allow us to know which program is in execution and observe the behaviors more easily.

If the ready queue is empty and the scheduler cannot fetch a user process, it should assign an **idle process** for the CPU to execute (same in real OS). You need to prepare an idle program **prog-idle** for the purpose. There should be an infinite loop in prog-idle so that it can continue execution whenever scheduled. Also, prog-idle should be loaded upon system initialization so that it is ready for execution any time, but it should never be placed

in the ready queue so that it will not compete for CPU with user programs. For convenience in showing the system states, PID for prog-idle should be set to 1.

4 Printer Process

In Project 1, for a print instruction, the printout is displayed on the monitor in Project 1. In project 2, we consider to also print it to a printer (monitor display remains). The printer is a separate entity in the system, not a component of a computer. Thus, we use a separate process to simulate the printer. You need to implement two components for the printer, one is the simulated printer (**printer.c**) process, the other is the component in the original computer process to initiate and interact with the printer process (**print.c**).

4.1 Print Component

The functions in print.c are given below.

print_init(…)	Fork the printer process and establish the pipe communication. Wait for the ACK from printer via pipe.
print_init_spool(…)	When a process is created, the computer can call this function to initialize the spooling for the process. This function simply sends the PID of the process to the corresponding printer function via the pipe.
print_end_spool(…)	When a process exits, the computer can call this function to indicate the termination of the spool of the process and print outputs to the simulated paper.
print_print(…)	Whenever cpu executes a print instruction, it calls this function. This function should at least have input parameters "buffer" and PID. CPU passes what is to be printed to print() via buffer. Function print() sends the buffer and PID to printer via the pipe.
print_terminate(…)	Before the system terminates, this function is called to inform the printer to terminate.

4.2 Simulated Printer

Note that, the printout on the monitor is in the order of the print instructions being executed in different programs, but the printout on the paper by the printer should be continuous for each process. OS generally implements a spooler to spool the printout of each process on disk. The entire printout of a process is sent to a printer only after the process terminates. Of course, the spooler software runs on computer, not on printer. But here we push the spooling task to the printer.

We assume that every process has some printing instruction(s). For simplicity, we use a file to store (spool) the printout of each process. And, we use a file **printer.out** to simulate the paper printer output.

printer_init(…)	Initialize the printer, including opening the simulated printing paper, i.e., the "printer.out" file. Sends an ACK to the print component to indicate that the initialization is done.
printer_init_spool(…)	Opening a spool file for the process. You can use PID to differentiate the spool file names.
printer_end_spool(…)	The printer prints the contents in the spool file of the process to the simulated paper and close the spool file.
Printer_dump_spool(…)	Print out the PIDs of all the processes with their spool files open.
printer_print(…)	This function is for handling the regular print instructions. It should determine which spool file to use and writes the to-be-printed content to the spool file.
printer_terminate(…)	For any process that has not terminated, print its partial output in its spool file to the simulated paper, and add a message at the end to indicate that the process did not finish yet. Then, clean up and terminate the printer process. Do not forget to close the file that simulates the printer paper.
printer_main(…)	This is the printer main function. It calls printer_init() to initialize the printer and then waits on the pipe. Once a message is received, it analyzes the message and determines the function to call.

You should use pipe for communication between the print component and the simulated printer. To avoid the computer starts execution and sends print commands to the printer before the printer is ready, the printer should send an ACK to the print component to indicate its readiness.

When a process terminates, a notification is sent to the printer and the printer starts to print out the entire spool of the process to the "paper" (i.e., the printer.out file). After finishing printing the output of a process, its spool can be freed (delete file).

CPU and IO have very different speeds. Spooling writes to a disk file, so it properly reflects the disk access time. Printing on a printer is extremely slow compared to CPU and disk accesses. Thus, we simulate the slow writes to the simulated paper printer.out by adding a sleep to each line of printing. The sleep time ***PT*** will be a system parameter given in config.sys file.

5 Shell and Computer Threads

Your program needs to wait for reading shell commands from the user and execute the user programs at the same time. Obviously, this is not possible and we can use two threads, one is the shell thread that waits on input commands, and the other is the computer thread that executes user programs.

5.1 Shell Thread

Function shell_command() implements the shell thread function.

5.2 Computer Thread

The computer.c main program will be somewhat different from that in Project 1.

- Now we have 3 system parameters, memory size M, time quantum TQ, and printing time PT. These parameters are read from **config.sys** in the order given here.
- Call init functions in each component in proper order, including printer initialization from print.c.
- Initialize the shell thread.
- Start the CPU scheduler, which will start CPU execution of processes.

In real systems, CPU runs forever till shut down. Scheduler should also run on CPU. But this is infeasible in our simulated system. In our simulated system, scheduler manages CPU execution, but runs on real CPU, not simulated CPU. The same for shell. In a real system, shell is also a process and should be scheduled to run on CPU by the same scheduler. But in our simulated system, shell is running on real CPU. Only the user programs are in the simulated memory and the simulated CPU.

5.3 Concurrency Control

Note that the shell thread and the main thread (the computer) access shared system states concurrently. Proper concurrency control should be coded to avoid potential problems due to concurrent accesses. The shell reads system states for dumping commands, which may result in the display of stale states, but it is not an issue. The concurrent access problem on files for command 5 has been discussed earlier. For terminate, the shell sets the terminate flag, but the computer components simply read the flag and there will be no problem. The submit command causes the insertion to the ready queue and the scheduler removes processes from the queue. This write-write conflict on the ready queue could cause problem and queue accesses should be protected. But at this stage, you do not know enough about concurrency control and can ignore the problem for time being.

6 Run Time Output and Testing

For testing and grading purpose, you need to provide clear outputs to show the behaviors of your program. Below are some situations that you definitely should display the corresponding messages:

- When the printer process starts
- When a process switch occurs (display the PIDs and PCs of the switched-in and switched-out processes)

- When the printer finishes printing for a process
- When any other important events happen

The list above may not be complete. You can add more situations for displaying messages. TA may add additional ones as well.

Since the simulated execution of user programs may go very fast and it may be hard for you to test your simulated system with multiple processes running, you can do the following when testing your system:

- Prepare several user programs that loops for a long time.
- If necessary, add some sleep time to each iteration. Do not add a long sleep time; otherwise, your system will run very slowly.
- Redirect input from stdin to a file (like what you did in the first project). Put many submit commands in the file. This way the system will be able to read in submissions much faster than your typing speed.

7 Project Submission

You need to submit your program **before** midnight (11:59pm) of the due date (check the web page for the due date). Use UTD elearning system for submission: "elearning.utdallas.edu".

Your submission should include the following:

- All source code files making up your solutions to this assignment.
  - All the source code files (modified or unmodified) from Project 1
  - "scheduler.c"
  - "print.c" and "printer.c"
  - Modified "makefile" for compiling your source code into "computer.exe" (no "execute" script)
  - Any other source code and user/idle program files you may have
- A "readme" file
  - Indicate how to compile and run your programs in case it deviates from the specification
  - If you did not finish the project, you need to specify which part(s) you have completed and which part(s) are missing
- You can zip or tar all your submission files together and then submit it on elearning