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.

1 CPU Scheduling and Context Switch

In Project 2, you need to implement a CPU scheduling algorithm. First, the system should support the submission and execution of multiple processes. To enable process submission, a submit shell command should be implemented. 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). The subsequent processing, including creating a PCB for the new process and load the program into memory, should also be shifted to shell.c. To allow the identification of the processes, you need to give each process an ID, PID, and store it in the corresponding PCB.

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 (may not be the newly inserted one) and calls the CPU function (some function in cpu.c) to execute the fetched process.

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. For simplicity, for each invocation of CPU from the scheduler, the CPU breaks the instruction fetch and execution loop and returns to the scheduler either when it has executed TQ instructions or it encounters the exit instruction (current process terminates). You need to change the CPU loop control accordingly. Note that in real OS, CPU execution loop terminates when detecting the time quantum expiration interrupt (or encountering IO or exit and we do not consider IO).

Context switch is a very important step in process switch. You need to perform context switch when a process get time quantum expiration and another process is switched in for execution on CPU.

The scheduler will continue fetching processes and let CPU execute them forever (like real computers). The system terminates only after the user issues a terminate command to the shell (will be elaborated later). 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 can be set to 1.

2 Printer Process

Another change in Project 2 is the addition of a printer. For a print instruction, the printout is displayed on the monitor in Project 1. Now we consider to also print to a printer (monitor display remains). The printer is a separate entity in the system, not a component of a computer. Thus, in Project 2, we use a separate process to simulate the 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. When the printer process starts, it opens the printer.out file. When a user process is created, a request is sent to the printer to initiate

its spooling (i.e., a file is created for the process spool). For each print instruction, what are to be printed is sent to the printer and get spooled. 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) and a notification should be sent from the printer to the shell to allow the shell to display a notification message to the user (e.g., Process <pid> has finished printing). You should use pipe for communication with the printer (with computer and with shell). Since shell cannot wait for the printer to send the notification message (it already waits for the user input), printer should signal the shell to allow the shell to go and read the notification from the printer upon finishing printing.

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 printer.out by adding a sleep to each line of printing. The sleep time ***PT*** will be a system parameter.

3 Shell and Computer Threads

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 supported in Project 2 and are listed in the following table. By the way, you can leave your original implementation for instruction 9 in cpu.c without any change.

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 (processes)	Project 2	Dump the information of all processes in the ready queue
5 (printer)	Project 2	Dump the spooled contents (file) of each process on the printer

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.

3.1 Shell Thread

The shell thread runs in a loop to read shell commands and process them when a command is received. Most of the commands in the shell are for dumping system states, except for terminate (0) and submit (1). We have discussed submit command earlier (in CPU scheduling). For termination, 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 the scheduler will no longer fetch processes from the ready queue and will proceed to terminate. A termination request should be sent to the printer and the printer can close the printer.out file and terminate. The processes that have not finished will not have printer output.

The new shell commands 4 and 5 are for dumping the process information and printer information. In real systems, a process may be in the ready queue or in one of the IO queues or wait queues. Since we do not consider waiting on IO queues (no read instruction) or other waiting states (like sleep), all active processes will be in the ready queue. Thus, for 4, you only need to go through the ready queue to print the PCB information of the processes. Note that, all information about a process should be stored in its PCB or have a pointer from the PCB. You need to print all the information about the process, including those in PCB and those pointed to from PBC.

In Project 2, we only consider 1 printer, thus, for shell command 5, you only need to dump the spooled contents of each process (together with the process ID) on the single printer. Since there is the computer thread that will continue to send printing content, retrieval of the spooled content will be an issue. You cannot close the spool file. Instead, you can either use "cat" to display the file content at the Unix shell level or support shared file accesses to allow read and write to be performed by different threads. In either case, you need to flush the write buffer before retrieving the file content.

3.2 Computer Thread

The computer.c main program will be somewhat different from that in Project 1. Note that we have 3 system parameters, memory size M, time quantum TQ, and printing time PT. We put these parameters in **config.sys** and let the computer reads them in at the starting time. We also add another system parameter in config.sys, state file name, **StateF**. If StateF is Null, then the state information from shell should be printed on screen; otherwise, the print out should go to the given StateF. This is to ease the testing and grading efforts, making your program execution information being observed more clearly.

After read in the system parameters and made proper initialization for various data structures in the system, Computer needs to fork the printer process and create the shell thread. Then computer calls the scheduler to start the CPU scheduling and program executions.

3.3 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.

4 Run Time Output

For testing and grading purpose, you need to provide clear outputs to show the behaviors of your program. Here are some situations that you should display the corresponding message: When the printer process starts, when the shell thread prompt to the user for inputs, when a process switch occurs (display the pids and PCs of the switched-in and switched-out processes), and when the printer finishes printing for a process (on both the printer side and shell side).

CS 4348/5348 Operating Systems -- Project 3

The goal of Project 3 is for you to practice programming using sockets. You need to modify the system to include multiple computers and multiple printers with a printer manager.

Your printer program will

Your computer program in Project 2 will not require changes, except for the communication mechanism

just need to run multiple versions of it on multiple windows on multiple computers (this means the real physical computer).

1 Printer Manager and Printers

The printer in Project 3 will simulate the network printers. It will be a very simple program, which reads the input text data from the printer manager and print the received text on simulated "paper". When starting, the printer sends a registration message to the printer manager. The printer manager responds to the printer with a printerID. After registration, the printer manager will send the text to be printed to the printer with a maximal buffer size ***PB***, which is given in the config.sys file. The communication between the printer and the printer manager are via sockets.

The code for the printer manager will be similar to the printer program in Project 2. Spooling for the printouts of the processes are done in the printer manager. The major change is the communication mechanism. You need to create a server socket in the printer manager for communicating with multiple computers and multiple printers. When a process finishes execution (computer informs printer manager), the printer manager chooses one printer and sends the printouts of the process to the selected printer. The selection can be based on workloads on the printer (or you can simply use round robin).

2 Shell Commands

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 (processes)	Project 2	Dump the information of all processes in the ready queue
5 (printers)	Project 2&3	Dump the spooled contents (file) of each process on each printer
6 (printer manager)	Project 3	Dump the allocation of print jobs to printers

3 Computer

Your computer program will be similar to that in Project 2, except that socket (instead of pipe) should be used for the communication between the computer and the printer.

4 System Setup

The printer manager creates the server socket and the printers and computers need to know its IP and port number. Thus, for Project 3, config.sys will have three new parameters (from Project 2), printer manager's IP and port, and ***PB***.

CS 4348/5348 Operating Systems -- Project 4

In earlier projects, we use external control of "Base" for each process to ensure non-overlapping memory allocation for multiple processes. In Project 5, you need to implement a memory manager to ensure proper memory allocation for processes. For program submission, the user only needs to provide the program file name to shell and Base is no longer needed.

There are two choices for this project. If you wish to implement a simple memory manager, you will work alone and choose the best-fit algorithm with dynamic allocation. If you wish to implement the simple paging mechanism, you can form a group of 2 members and implement it. If you would like to implement the demand paging algorithm, you can form a group of 3 members and implement it. For demand paging, you need to handle page faults and implement the aging scheme for page replacement decisions.

In the case of dynamic allocation, there is not need to change the loader program. For the case of simple paging, your loader needs to load the input user program to non-continuous memory pages. For demand paging, you need to have an additional system parameter in config.sys, namely, the working set size $\boldsymbol{WS}$. Also, you need to implement the swap space. The swap space can simply be an array. You can use external Base to control the starting location of the swap space for each process (just the same as what we did for memory in earlier projects). When loading, you need to copy the user program to the swap space, and only load WS pages to memory. When there is a page fault, you need to write the replaced page back to the swap space and read the demanded page into memory from the swap space.