University ofOttawa
School of Information Technology and Engineering

Course: CSI3310 – Operating Systems Principles
SEMESTER: Winter 2007

Professor: Stefan Dobrev
Date: February 26 2007
Hour: 14:30-16:00 (1.5 hours)
Room: Montpetit 202

Midterm Exam

The exam consists of three (3) parts:

Part 1	**Multiple Choice**	**8 points**
Part 2	**Short Answers**	**10 points**
Part 3	**Problem Solving**	**10 points**
Total		**25+3 points**

The exam is worth 28 points, 3 of them are essentially bonus (25 points represent 25% of the total mark).

Part 1: Multiple choice (8 questions, each for one point):

1. **One** of the following statements are **false** about interrupts:
   a. If the interrupts are enabled, the CPU checks for interrupts after the execution of every instruction
   b. If an interrupt is detected, the currently running program is terminated and replaced by an *interrupt handler*
   c. Interrupt vector is a table containing addresses of the interrupt handlers.
   d. Interrupts may have different priorities
   e. Interrupts can be disabled only when the CPU is in kernel mode.

2. **One** of the following statements is **false:**
   a. The only way a user process can enter a kernel mode is by performing system call, which will give the control to the OS kernel.
   b. I/O hardware can be directly accessed only by a process running in the kernel mode.
   c. If a privileged instruction is executed while the CPU is in the user mode, an interrupt is raised and the current process is terminated.
   d. Any process can change from user mode to kernel(privileged) mode and continue executing its own code, but when its kernel mode work is done, the control is given to the OS kernel to check for any unauthorized access.
   e. If a user process needs to perform I/O operation, it has to make a system call to ask the OS kernel to perform it in its behalf.

3. **Two** of the following statements are **false** about process states and state transitions:
   a. A process state changes from NEW to READY when it is admitted to the system by the long term scheduler
   b. Process state changes from RUNNING to WAITING when its time slice expires
   c. Only processes in the READY state can change into RUNNING state.
   d. If a process is preempted, its state changes to READY.
   e. There can be several processes in the WAITING state, but only one in the READY state.

4. **One** of the following statements is **false** about processes and threads:
   a. Context switching for threads requires less overhead that for processes.
   b. When a process invokes an I/O system call, its state may be changed from *executing* to *waiting*.
   c. With the many to one model, user threads are managed by the kernel.
   d. One task of context switching is to store the CPU state in the PCB.
   e. Overhead in creating and terminating threads can be reduced through the use of thread pools.

5. **One** of the following statements is **true** about processes and threads:
   a. Threads share the same address space, including code segment, data segment, stack pointer and program counter
   b. User threads do not share memory, only kernel threads do
   c. Thread library implementing many-to-one model can make use of at most 4 CPUs.
   d. When one thread of a process makes exec() call, all threads are terminated.
   e. The thread library is responsible for thread scheduling, regardless of which model (many-to-one, one-to-one, many-to-many) is used.

6. **Two** of the following statements about scheduling are **false:**
   a. Preemptive scheduler lets the process finish its CPU burst, but preempts the process when it starts blocking I/O operation
   b. Round Robin is a preemptive scheduler and does not suffer from starvation
   c. Non-preemptive schedulers suffer from the convoy effect.
   d. Shortest-Job-First scheduler is only of theoretical interest, as we cannot know the future CPU burst lengths
   e. Multilevel Feedback Queue scheduler gives priority to I/O intensive processes

7. Solution requirements for the critical section problem are:
   a. Progress, Mutual exclusion and Bounded Waiting.
   b. Progress, Interleaved execution, and Mutual Exclusion
   c. Bounded waiting, Simultaneous memory access, and Efficiency.
   d. Efficiency, Progress, and Bounded Waiting.
   e. Simultaneous memory access, Mutual Exclusion, and Progress.

8. **One** of the following statements is **true**:
   a. Peterson's algorithm does not have busy waiting
   b. Unlike Peterson's algorithm, using hardware instructions (test&set, xchng) eliminates busy waiting.
   c. Unlike hardware instructions (test&set, xchng), semaphores allow elimination of busy waiting.
   d. If a process/thread invokes the `wait()` method of a semaphore, it will block until some other process/thread calls the `signal()` method
   e. Interrupt disabling ensures mutual exclusion on single- and multi-CPU computers, but on a multi-CPU computer a process can be preempted even if the interrupts are disabled.

Part 2: Short answer questions (10 points, answer in the provided space)

1. (1 point) Explain the difference between symmetric multiprocessing and asymmetric multiprocessing.

2. (2 points) What is the difference between the microkernel OS structure and the modular OS structure?

3. (1 point) Explain what *deferred thread cancellation* is and why it is used.

4. (2 points) Explain (draw a diagram and explain transitions) how PCB queues are used in process scheduling.

5. (2 points) The following algorithm proposes an approach for dealing with the critical section problem. Does it provide a suitable solution? Justify your answer (argue why it is a solution, or identify which requirements of the CSP are violated and show how).

```
Process Pi:
repeat
  while(turn!=i){};
     Critical Section
  turn:=j;
     Remainder Section
forever
```

6. (2 points) Fill in the algorithms for the semaphore's methods, explaining what they do.

```
wait(S):


signal(S):
```

Part 3: Problem Solving (10 points)

Problem 1 (4 points): Simulate Round Robin scheduler

System specification:

- Each CPU burst is followed by an I/O operation taking 12 time units
- The I/O operations of different processes overlap and do not interfere, i.e. if P1 starts to wait at time 10 and P2 starts to wait at time 20, P1 will become ready at time 10+12 = 22 and P2 will become ready at time 20+12=32.

Scheduler specification:

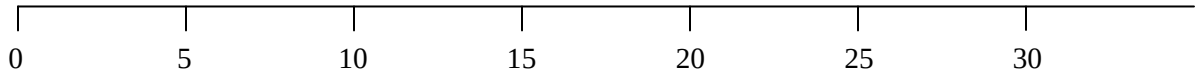
- Round Robin scheduler, one priority level
- The time slice is 5 time units

Process specification:

- There are four processes P0, P1, P2 and P3
- Arrival time of processes are as follows:
  - P0: 0, P1: 2, P2: 18, P3: 24
- The CPU burst times of processes are as follows:
  P0: 8, 9, 10, 7
  P1: 12, 14, 10, 13
  P2: 2, 4, 2, 1, 4, 3, 2, 3
  P3: 6, 4, 7, 4, 8

Your task:

- Simulate the Round Robin scheduler scheduling these 4 processes for the first 30 time steps
- In the following line, mark the times of the context switches and mark which process is executing at a given moment:

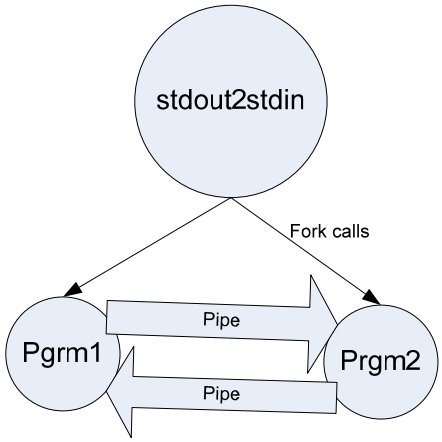


Problem 2 (6 points): Connecting standard output to standard input: *stdout2stdin*.

Detailed specification: Complete the following main routine with the appropriate `pipe`, `fork`, `dup2`, and `execlp` calls so that two processes are created each with their standard output attached to the standard input of the other process as shown in the diagram below. The synopsis for the program is:

```
stdout2stdin <pgrm1> <pgrm2>
```

Where <*pgrm1*> and <*pgrm2*> are two executable files.



Synopsis of system calls available:

```
int fork()
int dup2(int oldfd, int newfd)
int execlp(char *file, char *arg,…,NULL)
int pipe(int filedes[2])
```

```
int main(int argc, char *argv[])
{

   char *pgrm1;         /*pointer to first program */
   char *pgrm2;         /*pointer to second program */


   if(argc != 3)
   {
     printf("Usage: stdout2stdin <pgrm1> <pgrm2> \n");
     exit(1);
   }

   /* get programs */
   pgrm1 = argv[1];
   pgrm2 = argv[2];
```

University ofOttawa
School of Information Technology and Engineering

Course: CSI3310 – Operating Systems Principles
SEMESTER: Winter 2007

Professor: Stefan Dobrev
Date: February 26 2007
Hour: 14:30-16:00 (1.5 hours)
Room: Montpetit 202

Midterm Exam
Solution

The exam consists of three (3) parts:

Part 1	**Multiple Choice**	**8 points**
Part 2	**Short Answers**	**10 points**
Part 3	**Problem Solving**	**10 points**
Total		**25+3 points**

The exam is worth 28 points, 3 of them are essentially bonus (25 points represent 25% of the total mark).

Part 1: Multiple choice:

1. b 2. d 3. b,e 4. c 5. d 6. a, d 7. a 8. c

Part 2: Short answer questions

Question 1 :

Symmetric multiprocessing: All CPU can perform scheduling; partitioning of tasks; and all OS functions.
Asymmetric multiprocessing: These functions are allocated to a single CPU; the other CPUs are dedicated to executing user processes.

Question 2;

Microkernel structure:

- The kernel of the first OSs were monolithique that contained all OS functions.
- Later an effort was made to include only essential functions in a smaller kernel (e.g. scheduling and message passing), and use system processes for other OS functions (e.g. file systems).

Modular structure :

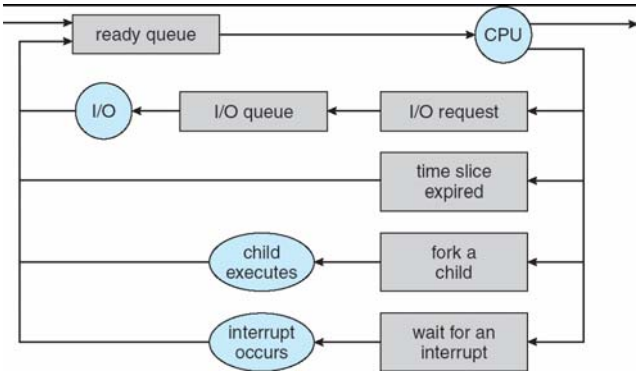
- Many modern OSs used an approach that uses modules :
  - Object oriented approach
  - Each component is separate
  - Standard interfaces are used to access functions
  - Modules are dynamically loaded by the kernel according to system needs.
- The structure is similar to the layered structure, but much more flexible.

Question 3 :

- Cancellation refers to termination of a thread requested by another thread. In the case of deferred cancellation, a flag is set to indicate termination; the thread verifies the flag to see if it should terminate its execution. This allows the thread to clean up its resources. This approach gives a graceful termination.

Question 4 :

The PCBs are placed in the different queues according to their state. For example, PCBs whose processes are in the Ready state are placed in the Ready queue. PCBs of processes in the wait state are placed in the I/O queues. CPU scheduling selects a process in the Ready queue for execution (allocated to the CPU). Note that the Ready queue and the I/O queues are typically implemented as multiple queues.



Question 5 :

No, it is not valid. It does note satisfy the progress requirement, since each process must wait for the other process to pas through its CS, even if the other process is in its remainder section.

Question 6 :

```
wait(S):    S.value --;
            if S.value < 0
            { // Semaphore is busy if value < 0
                    Add the process/thread to S.L;
                  block() // the process/thread is placed in the wait state
            }
```

The wait() is used to bloc a process until a signal is received from another process.

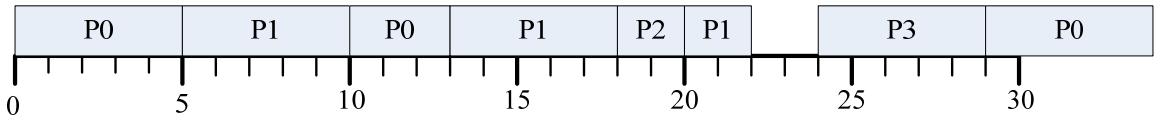
```
signal(S): S.value ++;
      if S.value ≤ 0
      {  // Processes/threads are waiting in the queue if value ≤ 0
            Remove a process/thread P from S.L;
          wakeup(P)  // selected process/thread becomes ready
      }
```

The signal() call is used to allow blocked processes to continue their execution.

Part 3: Problem Solving

Question 1 :

Time	Wait	Ready	CPU
0			P0(8), 5
2		P1(12)	P0(6), 3
5		P0(3)	P1(12), 5
10		P1(7)	P0(3), 5
13	P0(12)		P1(7), 5
18	P0(7)	P1(2)	P2(2), 5
20	P2(12), P0(5)		P1(2), 5
22	P1(12), P2(10), P0(3)		
24	P1(10), P2(8), P0(1)		P3(6), 5
25	P1(9), P2(7)	P0(9)	P3(5), 4
29	P3(12), P1(5), P2(3)		P0(9), 5
32	P3(9), P1(2)	P2(4)	P0(6), 2



Question 2:

```
int main(int argc, char *argv[])
{

   int i,j;             /*indexes into arrays */
   char *pgrm1;         /*pointer to first program */
   char *pgrm2;         /*pointer to second program */
   int pipe1to2[2];  /* pipe to prc1 to prc2 */
   int pipe2to1[2];  /* pipe to prc2 to prc1 */
   int pid;

   if(argc != 3)
   {
     printf("Usage: stdout2stdin <pgrm1> <pgrm2> \n");
     exit(1);
   }

   /* get programs */
   pgrm1 = argv[1];
   pgrm2 = argv[2];

   /* create the pipes */
   pipe(pipe1to2);
   pipe(pipe2to1);
   /* create process 1 */
   pid = fork();
   if(pid == 0)
   {
      dup2(pipe1to2[0], 0);
      dup2(pipe2to1[1], 1);
      close(pipe1to2[0]);
      close(pipe1to2[1]);
      close(pipe2to1[0]);
      close(pipe2to1[1]);
      execlp(pgrm1, pgrm1, NULL);
   }

   /* create process 2 */
   pid = fork();
   if(pid == 0)
   {
      dup2(pipe2to1[0], 0);
      dup2(pipe1to2[1], 1);
      close(pipe1to2[0]);
      close(pipe1to2[1]);
      close(pipe2to1[0]);
      close(pipe2to1[1]);
      execlp(pgrm2, pgrm2, NULL);
   }
}
```