(1 pt) Name: **(1 pt) User Id:**

CMPSCI 377: Operating Systems

Exam 1: Processes, Threads, CPU Scheduling and Synchronization

March 7, 2003

General instructions:

- This examination booklet has 9 pages.
- Do not forget to put down your name and user ID on the exam books.
- The exam is closed book. You are allowed one page of your own notes for reference during the exam.
- Explain your answers clearly and be concise. Do not write long essays (even if there is a lot of open space on the page).
- You have 50 minutes to complete the exam. Be a smart test taker: if you get stuck on one problem go on to the next. Don't waste your time giving details that the question does not request. Points will be deducted if too many unnecessary details are given.
- Questions worth 5 points can be answered with about 1.5 sentences
- Show your work. Partial credit is possible, but only if you show intermediate steps.
- Good luck.

---

1. **Hardware and OS Design** (25 pts)

   (a) (5 pts) Define Direct Memory Access (DMA). Give two examples in which it is useful.

   With DMA, an I/O device transfers data directly between it and memory without the data first being copied to the CPU. DMA is useful for performing large data transfers (e.g., disk or video I/O transfers).

(b) (10 pts) List one advantage and one disadvantage to including some functional module of the OS within the kernel (i.e. so that it executes in kernel mode). What choice does a micro-kernel tend to make in this regard?

Advantage: calls between OS modules do not involve system calls (and so is more efficient).

Disadvantage: the module runs in kernel mode, even if it does not require this functionality. This means that serious bugs in the module can affect the entire system.

Micro-kernels attempt to push as much functionality into user space as possible, with the hope of making the kernel as efficient and as small as possible (and hopefully it is also more reliable).

(c) (5 pts) Give one reason for preferring Test&Set over enabling/disabling interrupts in implementing a lock in a uniprocessor system. Give one reason for preferring enabling/disabling interrupts.

Test&Set is an operation that we would trust user-level processes to execute; we would not want to trust user-level processes with altering the interrupt behavior.

Test&Set requires busy waiting, whereas enabling/disabling interrupts does not.

(d) (5 pts) Give three examples of an explicit hardware mechanism that is motivated by specific OS services.

Atomic operations for synchronization.

Kernel/user mode, base/limit registers, protected instructions for various forms of protection.

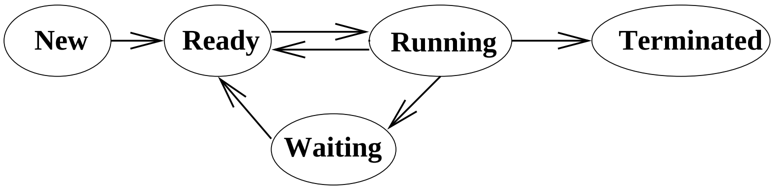
Interrupt vectors for handling interrupts.

Traps and trap vectors for handling internal errors and system calls.

Interrupts and memory-mapped communication for I/O

2. **Processes and Threads** (20 pts)

(a) (10 pts) Draw a state transition diagram of process execution states. Can a process transition from waiting for an I/O operation to the terminated state? Why or why not?



No. A process waiting for I/O must first transition to the ready queue and then to the running state before it may terminate.

(b) (5 pts) What is the essential cause in the difference in cost between a context switch for processes and kernel-level threads (assume the threads are part of the same process)?

Processes have their own memory configuration. Thus, a process context switch requires reconfiguration of the related hardware (and storage of the out-going configuration). Threads, on the other hand, share their memory space and do not require reconfiguration between switches.

(c) (5 pts) What is the essential cause of the difference in cost between a context switch for kernel-level threads and a switch that occurs between user-level threads?

Kernel-level threads require a system call for the switch to occur; user-level threads do not.

3. **Scheduling** (29 pts)

(a) (5 pts) What is the benefit to using a Round Robin scheduling algorithm? What is the main disadvantage of this algorithm?

Round Robin is a fair scheduling algorithm in that it gives each process an equal opportunity at the CPU. The price we pay for this fairness is in average waiting and response time.

(b) (5 pts) How would one approximate Round Robin scheduling with a lottery scheduling algorithm?

Distribute the tickets uniformly to all processes.

(c) (19 pts) Compute the completion and waiting times for the following set of jobs under the Shortest-Job-First (SJF), Round Robin (RR), and Multi-Level Feedback Queue (MLFQ) scheduling algorithms. Make the following assumptions:

- Zero context switch overhead.
- Job 1 performs 1.5 sec of work, followed by 1 sec of I/O. This is then repeated 2 more times (so a total of 4.5 secs of work and 3 secs of I/O).
- If otherwise not defined, then assume that job 2 is scheduled before job 3.
- For RR, assume a 2 sec time-slice, and when a job arrives in the middle of execution of another job, the new job is placed at the end of the RR queue.
- SJF is preemptive.
- For MLFQ, assume 3 levels of time slices of 1, 2, and 4 seconds.

Clearly state any additional assumptions that you make. Show your work.

					Completion Time			**Wait Time**		
Job	length	arrival time	work before I/O	I/O duration	SJF	RR	MLFQ	SJF	RR	MLFQ
1	4.5	1	1.5 sec	1 sec	8.5	12.5	11.5	1	5	4
2	10	0	-	-	16.5	16.5	16.5	6.5	6.5	6.5
3	2	0	-	-	2	4	6	0	2	4

Notation: $Job^{job\ units}_{total\ units}$

Note: Completion time = Turnaround time (if you assumed Completion time = time of completion, then we will take that into account).

SJF

$3^2_2\ 1^{1.5}_{3.5}\ 2^1_{4.5}\ 1^3_6\ 2^2_7\ 1^{4.5}_{8.5}\ 2^{10}_{16.5}$

Note: Job 1 completes its last unit of I/O at time 9.5

RR

$2^2_2\ 3^2_4\ 1^{1.5}_{5.5}\ 2^4_{7.5}\ 1^3_9\ 2^6_{11}\ 1^{4.5}_{12.5}\ 2^{10}_{16.5}$

Note: Job 1 completes its last unit of I/O at time 13.5

MLFQ

Q1	$2^1_1\ 3^1_2\ 1^1_3$			$1^{2.5}_{8.5}$			1^4_{11}		
Q2		$2^3_5\ 3^2_6\ 1^{1.5}_{6.5}$			1^3_9			$1^{4.5}_{11.5}$	
Q3			$2^4_{7.5}$			2^5_{10}			$2^9_{15.5}\ 2^{10}_{16.5}$

Note: Job 1 completes its last unit of I/O at time 12.5

4. **Synchronization** (24 pts)

(a) (10 pts) Define *process synchronization*. What are the two conditions (correctness properties) that a proper implementation of a lock needs to satisfy?

Synchronization is a technique by which we ensure that data structures shared by multiple processes/threads are not left in an inconsistent state due to the arbitrary execution order of the processes.

1) the lock must ensure that only one process can enter the critical section at one time, and

2) the lock must ensure that progress in a computation is ultimately made (assuming proper use).

(b) (14 pts) Suppose we have a world in which there are 3 robots (red, blue, green) – each is controlled by its own thread. You must ensure that the robots only move in the following order: red, blue, green, red, blue, green, etc. Add the necessary code below that performs the appropriate initializations and enforces this execution order. Use only semaphores for your synchronization. You may use pseudo-code for your solution.

```
class Robot_Game
{
    // Supply variables here

    Semaphore sem[3];
    R_Robot red;
    G_Robot green;
    B_Robot blue;



    Robot_Game()
    {
        // Supply any necessary initializations here

        ra.sem = new Semaphore[3];
        ra.sem[0] = new Semaphore(1);
        ra.sem[1] = new Semaphore(0);
        ra.sem[2] = new Semaphore(0);


        red = new R_Robot(this);
        green = new G_Robot(this);
        blue = new B_Robot(this);

        red.start();
        green.start();
        blue.start();


    };
};
```

```
class R_Robot extends Thread
{
   Robot_Game rg;
   R_Robot(Robot_Game ptr){rg = ptr;};
   void run()
   {
     while(true) {
        //  Supply only synchronization code

        da.sem[0].Wait();


        <Make Move>


        da.sem[1].Signal();

      };
    };
};

class G_Robot extends Thread
{
   Robot_Game rg;

   G_Robot(Robot_Game ptr){rg = ptr;};

   void run()
   {
      while(true) {
        //  Supply only synchronization code

        da.sem[1].Wait();


        <Make Move>

        da.sem[2].Signal();

      };
    };
};
```

```
class B_Robot extends Thread
{
   Robot_Game rg;

   B_Robot(Robot_Game ptr){rg = ptr;};

   void run()
   {
      while(true) {
        //  Supply only synchronization code

        da.sem[2].Wait();


        <Make Move>

        da.sem[0].Signal();

     };
   };
};
```