Name: **Student Id:**

CMPSCI 377: Operating Systems

Exam 1: Processes, Threads, CPU Scheduling and Synchronization

October 9, 2002

General instructions:

- This examination booklet has 10 pages.
- Do not forget to put down your name and student number 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.
- Show your work. Partial credit is possible, but only if you show intermediate steps.
- Good luck.

---

1. **Short answer questions** (20 pts)

   (a) (5 pts) Is the context switch overhead of a user-level thread less than the overhead for processes? Explain briefly.

   Yes. A context switch is implemented by the kernel and involves the copying of state information between the processor and the Process Control Block or Thread Control Block. Because the kernel knows nothing about user-level threads, there cannot technically be a context switch for user-level threads. Note that the user-level scheduler may perform some limited copying of state on behalf of a thread before control is handed to that thread. However, this copy of state information is much smaller than that of a kernel-level process and does not involve dropping into kernel mode (via a system call).

(b) (5 pts) What are the differences between I/O bound and CPU bound processes? Describe one typical application that is I/O bound; describe one that is CPU bound.

An I/O bound process is constantly making system calls for I/O operations, and thus requires little CPU time on any particular time slice. A CPU-bound process makes almost constant use of the CPU.

I/O bound example: a database management system constantly makes requests for disk and interprocess communication services.

CPU bound example: simulating ocean currents requires many floating point operations.

(c) (5 pts) Compare and contrast the use of the Test&Set and the enabling/disabling interrupt techniques for implementing locks.

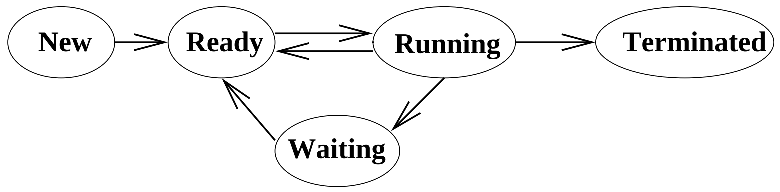
Both are hardware-level mechanisms that can be used for process/thread synchronization. Test&Set requires busy waiting. However, it is easier to get right and programming errors will not lead to system deadlock. With disabling and enabling interrupts, we don't require busy waiting, but if we make a programming error the system may deadlock. Also, we must be concerned with the length of time that we postpone other system activities (especially responses to time-critical interrupts).

(d) (5 pts) Define busy waiting. Under what conditions might it be desirable and undesirable?

Busy waiting means that a process maintains the CPU while it is waiting for some I/O operation to complete (rather than giving up the CPU). This is generally undesirable because the processor could be doing useful work otherwise. This behavior may be desirable under conditions when we know that the process (usually a kernel process) will not be waiting for a long time for the I/O operation to complete and that the process must respond to the completion quickly (within a few processor instructions).

2. **Processes and Threads** (25 pts)

(a) (5 pts) Draw a state transition diagram of process execution states. List two cases under which a process might transition from the waiting state.



A process may transition from the waiting state if an I/O operation has completed, or if the process timer has expired (e.g., if the process made a sleep() system call).

(b) (5 pts) Describe the pros and cons of writing a program with multiple threads (as opposed to a single thread).

Pros: Many programs may easily be thought of as multiple, independent threads of control, which can make programming much simpler.

Cons: As programmers, we must worry about synchronization of the threads during access to common data structures. Under certain circumstances, this can add complexity to a program.

(c) (5 pts) Describe the pros and cons of using kernel- versus user-level threads.

Kernel-level threads pros: kernel scheduler can take the individual threads into account; and a system call by a thread does not block the entire process.

User-level threads pros: management of threads does not require a system call; switching between threads is less expensive than kernel-level threads; and the user may implement custom scheduling algorithms for each process.

(d) (10 pts) Explain in detail what happens when a unix fork() system call is made (at both the kernel and user levels).

The operating system creates an identical copy of the process; this includes copying of: the data segment (the code), the heap, the stack, and the program counter. In general, no memory segments are shared between the child and the parent. However, the child receives a unique process identifier (PID), which is kept in its Process Control Block.

fork() returns a 0 to the child and returns the PID of the child to the parent.

3. **Scheduling** (35 pts)

(a) (8 pts) List and define the different metrics by which we might evaluate a scheduler (list at least 4).

CPU Utilization The percentage of time that the CPU is busy.

Throughput The number of processes completing in a unit of time.

Turnaround time The length of time it takes to run a process from initialization to termination, including all the waiting time.

Waiting time The total amount of time that a process is in the ready queue.

Response time The time between when a process is ready to run and its next I/O request.

(b) (8 pts) Explain the advantages and disadvantages of Shortest-Job-First (SJF) scheduling. Explain why we say that the Multi-Level-Feedback-Queue (MLFQ) is an approximation to SJF. Why does MLFQ not have the disadvantages of SJF?

Advantages: Provably optimal with respect to minimizing waiting time; and can have preemptive or non-preemptive implementations.

Disadvantages: CPU-bound jobs can potentially starve; and it is difficult to know *a priori* the amount of time that a process will require before completion or before its next I/O operation.

MLFQ approximates SJF because it also gives short jobs (or jobs with little CPU time between I/O operations) the highest priority. However, it only differentiates between a small number of job lengths.

MLFQ does not suffer from the "future knowledge" problem of SJF because it uses recent historical information to estimate CPU burst duration.

(c) (19 pts) Compute the completion and waiting times for the following set of jobs. Assume a zero context switch overhead. For the MLFQ, assume 3 levels of time slices of 1, 2, and 4 seconds; For RR, assume a time slice of 1 second. *work before I/O* refers to the duration of work before I/O is performed by the job (assume that this is 1 instruction short of the time slice); *I/O duration* refers to the length of time required to complete the I/O request (during which other jobs may be serviced). When a process arrives at the same time that a context switch occurs, assume that the arriving process is first placed at the end of the queue before the currently-executing process is removed from the CPU and placed on the queue. 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	FCFS	RR	MLFQ	FCFS	RR	MLFQ
1	50	0	-	-	50	90	90	0	40	40
2	30	4	1 sec	1 sec	106	70	61	46	10	1
3	10	8	-	-	112	29	19	102	19	9
Average					89.3	63	56.7	49.3	23	16.7

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

FCFS

Assumption: Job 2 alternates between 1 unit of work and 1 unit of I/O. Job 2 completes before job 3 is initiated (even though I/O is happening half the time).

$1^{50}_{50}\ 2^{30+30}_{110}\ 3^{10}_{120}$

RR

$1^{1..4}_{1..4}\ ''INSERT2''\ 2^{1}_{5}\ 1^{5}_{6}\ 2^{2}_{7}\ 1^{6}_{8}\ ''INSERT3''2^{3}_{9}\ 3^{1}_{10}\ (1^{7..15}_{11..35}\ 2^{4..12}_{12..36}\ 3^{2..10}_{13..37})''COMP3''(1^{16..33}_{38..72}\ 2^{13..30}_{39..73})$
$''COMP2''1^{34..50}_{74..90}$

Note: Although Job 2 completes its CPU usage at time 73, there is an additional 1 second of I/O

MLFQ

$$\begin{array}{l cccccccc|cc|cc|c}
Q1 & 1^{1}_{1} & & & 2^{1}_{5} & & 2^{2}_{7} & & 3^{1}_{9} & 2^{3..11}_{10..26} & & 2^{12..30}_{28..64} & & \\
Q2 & & 1^{3}_{3} & & & & & & & & 3^{2..10}_{11..27} & & & \\
Q3 & & & 1^{4}_{4} & & 1^{5}_{6} & & 1^{6}_{8} & & & & & 1^{7..25}_{29..65} & 1^{26..50}_{66..90}
\end{array}$$

Note: Job 3 never advances to Q3 because it is never given the opportunity to use more than one time unit before it is interrupted by Job 2.

Alternative assumptions: Job 2 alternates between 1 unit of work and 1 unit of I/O. The length in Job 2 includes 15 CPU units and 15 waiting units.

					Completion Time			Wait Time		
Job	length	arrival time	work before I/O	I/O duration	FCFS	RR	MLFQ	FCFS	RR	MLFQ
1	50	0	-	-	50	75		0	25	
2	30	4	1 sec	1 sec	76	40		46	10	
3	10	8	-	-	82	29		72	19	
Average					69.3	48		39.3	18	

FCFS

$1_{50}^{50}\ 2_{80}^{30}\ 3_{90}^{10}$

RR

$1_{1..4}^{1..4}\ ''INSERT2''\ 2_5^1\ 1_6^5\ 2_7^2\ 1_8^6\ ''INSERT3'' 2_9^3\ 3_{10}^1\ (1_{11..35}^{7..15}\ 2_{12..36}^{4..12}\ 3_{13..37}^{2..10}) ''COMP3'' (1_{38..42}^{16..18}\ 2_{39..43}^{13..15})$
$''COMP2'' 1_{44..75}^{19..50}$

Alternative assumptions: Job 2 performs 1 unit of work, then 1 unit of I/O, and then 29 more units of work.

					Completion Time			Wait Time		
Job	length	arrival time	work before I/O	I/O duration	FCFS	RR	MLFQ	FCFS	RR	MLFQ
1	50	0	-	-	50	90	90	0	40	40
2	30	4	1 sec	1 sec	77	69	69	46	38	38
3	10	8	-	-	83	29	28	73	19	18
Average										

FCFS

$1_{50}^{50}\ 2_{81}^{30+1}\ 3_{91}^{10}$

RR

$1_{1..4}^{1..4}\ ''INSERT2''\ 2_5^1\ 1_6^5\ 2_7^2\ 1_8^6\ ''INSERT3'' 2_9^3\ 3_{10}^1\ (1_{11..35}^{7..15}\ 2_{12..36}^{4..12}\ 3_{13..37}^{2..10}) ''COMP3'' (1_{38..72}^{16..33}\ 2_{39..73}^{13..30})$
$''COMP2'' 1_{74..90}^{34..50}$

MLFQ

Q1	1_1^1			2_5^1		2_7^2		3_9^1		
Q2		1_3^3					2_8^3		$2_{11}^5\ 3_{13}^3$	
Q3			1_4^4		1_6^5					$1_{17}^9\ 2_{21}^9 3_{25}^7\ 1_{29}^{13}\ 2_{33}^{13}\ 3_{36}^{10} \mid 1_{40..64}^{17..29}\ 2_{44..68}^{17..29} \mid 1_{72}^{33}\ 2_{73}^{30}\ 1_{74..90}^{37..50}$

4. **Synchronization** (20 pts)

(a) (5 pts) What is the difference between a binary and a counting semaphore? Give a **short** example of where a counting semaphore is useful.

A binary semaphore is either locked or unlocked. A counting semaphore can keep track of the locked/unlocked status of several identical resources; specifically, it counts the number of unlocked resources.

Counting semaphores are useful, for example, for keeping track of the number of free slots in a finite-sized buffer. A process inserting a new element can wait on this semaphore, an operation that will block until a slot is free.

(b) (15 pts)

Three threads (B, C, and D) cooperate to add up the contents of an array x of size 20 as follows. B adds up the even elements, x[0], x[2],... . C adds up the odd elements, x[1], x[3],... . D adds up the results of B and C.

Supply the necessary variables and code below so that:

- Each thread terminates after finishing its work; they do not wait for other threads to complete their execution.
- B and C are able to access (different entries of) the array concurrently.
- Counting semaphores are the only synchronization constructs (no access to interrupts, process queues, etc.). Assume the java Semaphore class that we discussed in class (and is to be used for lab 2).
- The threads do not busy wait.
- The code for B, C and D is of minimal length (pseudo-java is all that is necessary).

You may assume that all of the necessary thread creation and starting code already exists (so, you do not need to supply it).

```
class Distributed_Adder
{
    int x[];
    int sum;
    // Supply any additional variables here

    Semaphore sem;
    int sumB, sumC;

    Distributed_Adder()
    {
        x = get_some_array_of_integers();
        // Supply any necessary initializations here

        da.sem = new Semaphore(-1);

    };
};
```

```
class ThreadB extends Thread
{
   Distributed_Adder da;
   ThreadB(Distributed_Adder ptr){da = ptr;};
   void run()
   {
        //  Supply code here

        int i;
        da.sumB = 0;
        for(i=0; i<20; i+=2)
          da.sumB += da.x[i];
        da.sem.Signal();

    };
};

class ThreadC extends Thread
{
   Distributed_Adder da;

   ThreadC(Distributed_Adder ptr){da = ptr;};

   void run()
   {
        //  Supply code here

        int i;
        da.sumC = 0;
        for(i=1; i<20; i+=2)
          da.sumC += da.x[i];
        da.sem.Signal();

    };
};
```

```
class ThreadD extends Thread
{
   Distributed_Adder da;

   ThreadD(Distributed_Adder ptr){da = ptr;};

   void run()
   {
        //  Supply code here

        da.sem.Wait();
        da.sum = da.sumB + da.sumC;

   };
};
```