



University of Ottawa
School of Information Technology and
Engineering

Course: CSI3131 – Operating Systems
SEMESTER: Winter 2010

Professor:
Date:
Hour:
Room:

Gilbert Arbez
February 12 2010
14:30-15:50 (80 minutes)
CBY B205

Midterm Exam

Please answer all questions on the questionnaire.

The exam consists of three (3) parts:

Part 1	Short Answer Questions	8 points
Part 2	Theory Questions	12 points
Part 3	Problem Solving	15 points
Total		35 points

Part 1: Short Answer Questions (8 questions, each for one point):

1. What part in the PCB allows the OS to setup queues (ready queue, I/O queues, etc.)?

Addresses/pointer variables.

2. Can pipes be used to allow communication between processes on different compute systems (answer yes or no)?

No.

3. List the scheduling criteria that a scheduling algorithm should minimize:

Turnaround, waiting, response.

4. Circle the scheduler that is responsible for swapping processes?

Short term scheduler

Medium Term Scheduler

Long term scheduler

5. Deferred cancellation is used to postpone the cancellation of a thread until its parent is ready to accept the exit value.

Is this statement TRUE ☐ or FALSE ☐ ? (**false *deferred cancellation tells the thread to terminate, it is up to the thread to do that, the parent has nothing to do with that***)

6. Give one example of a resource that is shared by threads within the same process:

open files, the code segment, the data segment, etc.

7. Give an advantage of many to one thread model over the one to one thread model.

Faster

8. What are the three main techniques for hardware input/output? Which one(s) would be most common in operating systems?

Direct I/O

Interrupts

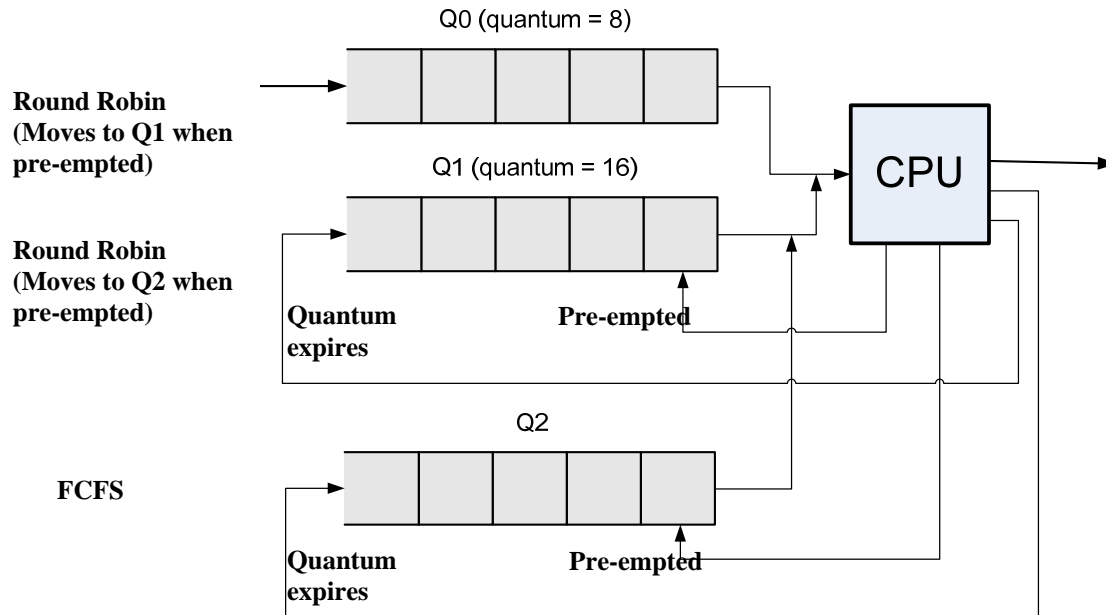
DMA

Part 2: Theory Questions (12 points, answer in the provided space)

1. (2 points) Describe what happens when a system call is made. In particular, explain why the calling user program cannot get control of the system after the CPU has switched to privileged/kernel mode.

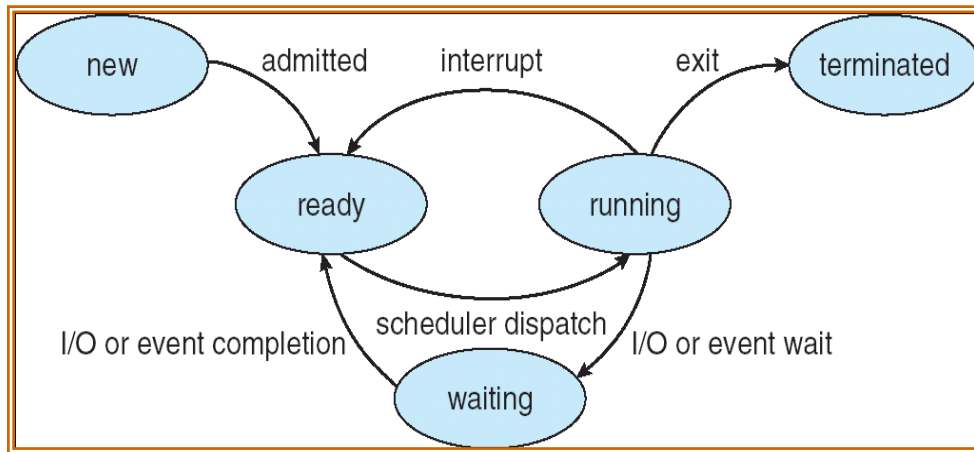
When a system call is made, a software interrupt instruction (called trap) is executed. This switches the CPU to a kernel/privileged mode and the control is given to the location specified in the kernel's trap table containing the addresses of routines for servicing system calls. The calling program can only specify which of the kernel provided routines will be executed, but cannot force the system to execute its (caller's) code.

2. (5 points) The following shows the diagram similar to the one used in class to explain CPU scheduling using multi-level feedback queues.
- Indicate on the diagram at each layer the algorithm used for scheduling.
 - Indicate on each of the four transitions starting at the CPU and ending at a queue which ones occur when a quantum expires and which ones occur when a process is pre-empted when another process enters a higher priority queue.
 - Explain how this approach deals with the convoy effect.



c) Convey effect does not occur since the I/O bound processes have short CPU bursts and will move through the Q0. Any CPU bound process uses up its quantum in Q0/Q1, moving down the queues to Q2. It will be pre-empted by I/O bound processes that enter Q0. Thus the CPU bound processes cannot slow down the I/O bound processes which will keep the I/O hardware busy while the CPU bound process will run on the CPU.

3. (5 points) Draw the 5 state process diagram and explain when each of the following transitions occur:



- a. From *new* state to *ready* state

When a process is admitted to the system by the long-term scheduler

- b. From *wait* state to *ready* state

When a corresponding event has occurred, e.g. i/o event, signal() on a semaphore, etc.

- c. From *running* state to *ready* state

Time quantum expired. Preempted by a higher priority process.

- d. From *waiting* state to *terminated* state

Again, cannot directly happen by the action of the process itself. Can happen if somebody sends kill signal to the process, or when the process is caught in cascading termination

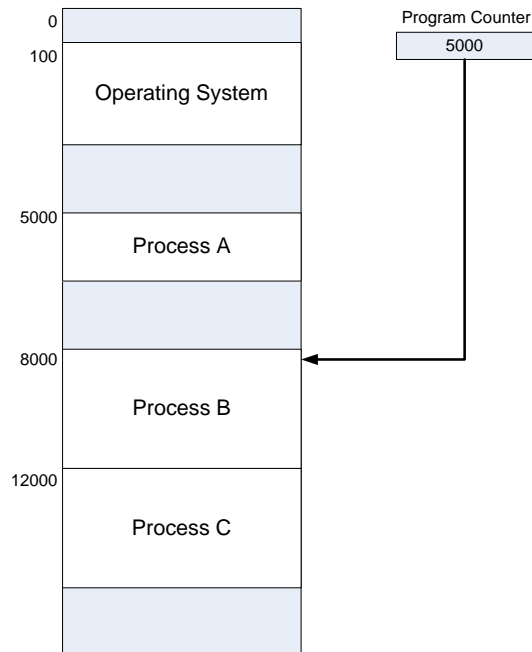
- e. From *running* state to *waiting* state

Called a blocking system call (i.e. blocking I/O operation, wait on a semaphore, wait until child terminates, blocking messaging...)

4. Part 3: Problem Solving (15 points)

Problem 1 (7 points): Simulate Process Execution

The figure below is a simplified diagram showing how three processes occupy memory in a system (assume that all processes are initially in the ready state). Note that part of the memory has been reserved by the operating system. In this problem, two components of the operating system are run: the short term round robin scheduler and the file system that handles I/O requests.



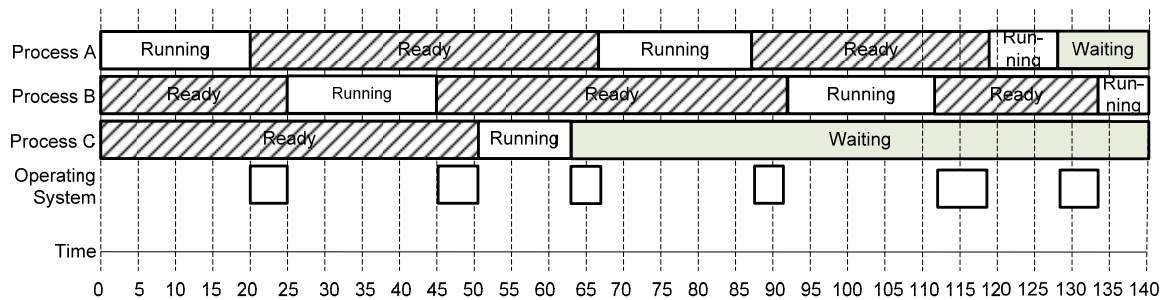
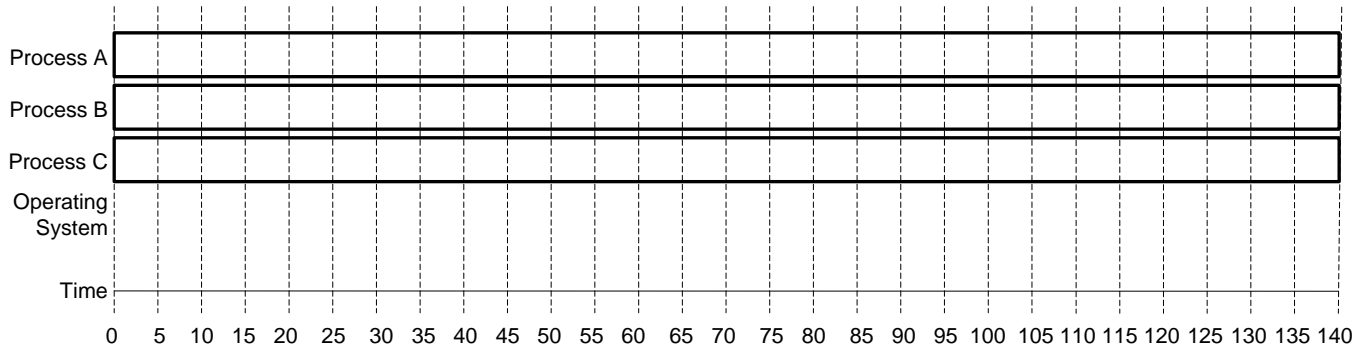
System execution proceeds as follows:

Time 0 to 20: Execution of Process A
Time 20: Process A quantum expires.
Time 20 to 25: Context switch between Process A and Process B
Time 25 to 45: Execution of process B
Time 45: Process B quantum expires
Time 45 to 51: Context switch between Process B and Process C
Time 51 to 63: Execution of Process C
Time 63: Process C requests a read from the hard drive
Time 63 to 67: Context switch between Process C and Process A
Time 67 to 87: Execution of Process A
Time 87: Process A quantum expires
Time 87 to 92: Context switch between process A and Process B
Time 92 to 112: Execution of Process B
Time 112: Process B quantum expires
Time 112 to 118: Context switch between process B and Process A
Time 118 to 128: Execution of Process A
Time 128: Process A requests a write to the hard drive
Time 128 to 133: Context switch between process A and Process B
Time 133 to ...: Execution of Process B

Complete the figure below to show for each process how its state changes during the system execution. For each process fill in the bar to show its state at different times using the following legend:



For the operating system, indicate the times that its code is running. (To make things simple, assume that the a process moves to the ready or wait states as soon as its execution is terminated, that is, when the OS starts executing; and to the running state when the OS has completed the execution of its code).

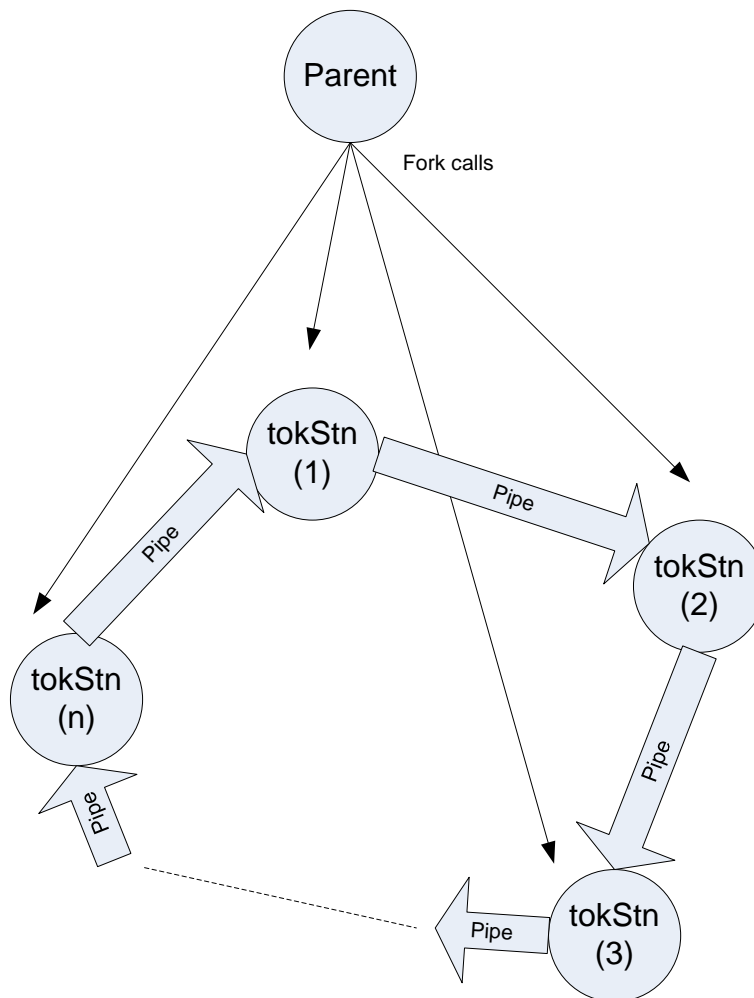


Problem 2 (8 points): Create nr child processes connected in a ring.

Detailed specification: The function `void createRing(int nr)` is called in a parent process to create nr child processes. A pipe connects the standard output of a child process to the standard input of another process to form a ring with all child processes as shown in the diagram below. Complete the function with the appropriate `pipe`, `fork`, `dup2`, and `execlp` calls to create such a ring of child processes. Child processes should only have `stdin`, `stdout`, and `stderr` file descriptors open when the `execlp()` is called. The child processes should run the `tokStn` program (with no arguments).

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])
```




```

void createRing(int nr) {
    int oriPipe[2];    // the first pipe created
    int childPipe[2]; // the next pipe connecting to the next child
    int fdRead;        // the read end of the last pipe created
    int fdWrite;       // the write end of the very first pipe created
    int i;
    char stnnum[20];

    pipe(oriPipe);
    fdRead = oriPipe[0];
    fdWrite = oriPipe[1];
    // create nr-1 children in a line
    for(i=0 ; i<nr-1 ; i++) {
        sprintf(stnnum,"%d",i);
        pipe(childPipe);
        if (fork()==0) { /* In child */
            /* the reads will be from the last pipe created */
            dup2(fdRead, 0);
            // 0 fd will work from now, no need for fdRead
            // fdWrite is inherited but unneeded
            close(fdRead);
            close(fdWrite);

            /* the writes will be into the new pipe */
            dup2(childPipe[1],1);
            // will use only the write end of the childPipe
            // but that is already copied to fd 1
            close(childPipe[1]);
            close(childPipe[0]);

            // execute the tokstn program
            execlp("tokstn", "tokstn", stnnum, NULL);
        }
        // the main program after creating this chain of children,
        // needs only the read end of the last pipe
        close(fdRead); /* no longer needed */
        fdRead = childPipe[0]; /* set to read end of new pipe */
        close(childPipe[1]); /* do no need this any more */
    }

    // Now create the last child and close the cycle
    /* use pipe saved fds (write end of original pipe */
    /* and read end of last pipe created */
    sprintf(stnnum,"%d",i);
    if (fork()==0) { /* In child */
        dup2(fdRead,0); /* read from the last pipe created */
        dup2(fdWrite,1); /* write to the first pipe created */
        close(fdRead);
        close(fdWrite);
        execlp("tokstn", "tokstn", stnnum, NULL);
    }
    close(fdRead);
    /* the main program writes into the cycle, to prime the pump */
    write(fdWrite,"t",1);
    close(fdWrite);
}

```