CSci 4061: Introduction to Operating Systems (Spring 2013)
Second Midterm Exam (April 11, 2013) (100 points)
Open Book and Lecture Notes
(Bring Your U Photo Id to the Exam)

This exam paper contains 5 questions (12 pages)
Total 100 points.

Suggested guideline: A 20 point question should take about 15 minutes to answer.

Please put your official name and NOT your assumed name.

First Name:

Last Name:

Student ID:

Q #	1	2	3	4	5
Max Points	16	24	20	20	20
Points Scored					

Question 1 (16 points):

Indicate by circling YES or NO if the given statement is correct or incorrect.

(a) For each kernel-level thread, a separate stack is maintained in the address-space of the process.
YES **NO**

(b) For each kernel-level thread, a separate stack is maintained in the kernel memory space.
YES **NO**

(c) For a kernel-level thread, data related to scheduling and execution context is maintained by the kernel. **YES** **NO**

(d) When a process masks a certain signal, then that signal is ignored and discarded by the kernel.
YES **NO**

(e) Critical sections implemented using the Test-and-Set instruction should be short in terms of execution time. **YES** **NO**

(f) A signal handler function can return any type of data as its return value. **YES** **NO**

(g) When a process is swapped out to disk, some of its threads can still execute if threads are supported at the kernel level. **YES** **NO**

(h) When a process is swapped out to disk, signals can still be sent to that process. **YES** **NO**

(i) When a process is swapped out to disk, all of its open file descriptors are closed. **YES** **NO**

(j) When a process is swapped out to disk, memory allocated for its address-space is freed.
YES **NO**

(k) When a process is swapped out to disk, process state transition from READY state to WAITING state can still happen. **YES** **NO**

(l) When a process is swapped out to disk, process state transition from WAITING state to READY state can still happen. **YES** **NO**

(m) All threads in a process share all open file descriptors of that process. **YES** **NO**

(n) Using the wait() function process can wait for the termination of any process, including its “grand-children”, i.e. the children of any of its child processes. **YES** **NO**

(o) FCFS scheduling policy is not suitable for interactive systems. **YES** **NO**

(p) CPU schedulers for interactive systems tend to give higher priority to processes that use CPU a lot without doing much input/output operations. **YES** **NO**

Question 2 (24 points): Consider four jobs (A, B, C, and D) waiting in a queue to be processed. Their respective total service times, as they appear in the queue (starting with the job at the head of the queue), are 10, 15, 5 and 30 msec. Assume that the jobs arrived at the same time.

(a) (8 points) For the following scheduling policies, draw a time chart showing when each of the five jobs will be completed.

- FCFS (with jobs' order in the queue)
- Shortest Job First
- Round-Robin (quantum = 10 msec)
- Round-Robin (quantum = 5 msec)

(b) (8 points) Determine the average turnaround time for each of the scheduling policies of part (a) above.

(c) (8 points) Compute the average waiting time for the four cases in part (a).

(continue your answer for Question 2…)

Question 3: (20 points)
Consider the following program which will create a set of processes, and answer the questions given below the code. Assume that there are no errors in creating processes.

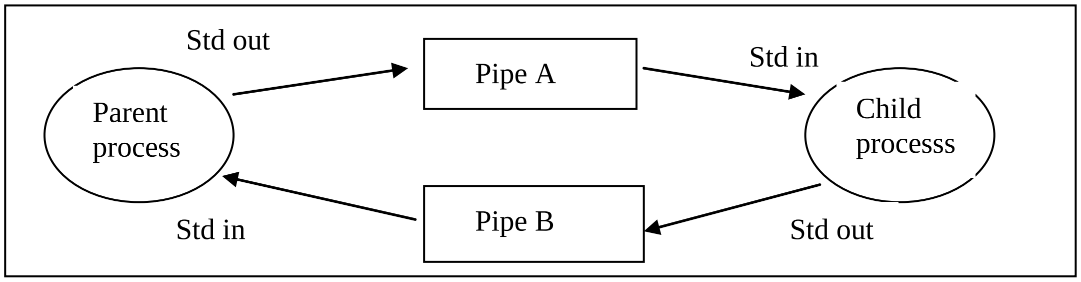
```
void main ( void ) {
  int  i,  stat;         int k = 1;     int n = 3;
  pid_t childpid;
  for ( i=0;  i < n; i++ )  {
      printf ( “i = %d k = %d “, i, k);
      childpid = fork();
      if ( childpid != 0 )  {   /* I am parent; I just created a child */
               break;
      }
      else {
               k= k*2;
      }
  }   // end of for loop
  if ( i==n) {
      exit(k);
  }
  else {    wait ( &stat );
              if ( WIFEXITED(stat) ) {
                 printf ("Exit status %d \n", WEXITSTATUS(stat) );
                exit( k + WEXITSTATUS(stat) );
             }
  }
}
```

Part A: (10 points) Show the tree of parent-child relationships among the processes that are created by this program. Number the processes in the order in which they are created, starting with the process executing the above code as process number 0.

Part B (10 points): Show the output produced by this program.

Question 4 (20 points):

Part (a) (10 points): Write the missing parts of the code to create the two pipes between the parent and its child process, as shown in the figure below. These processes are connected in a ring structure. The standard out of the parent process is connected to the write end of the pipe A, and the standard input of the child is connected the read end of pipe A. Similarly pipe B connects the standard out of the child process to the standard input of the parent process. In your code you MUST close all unnecessary file descriptors.



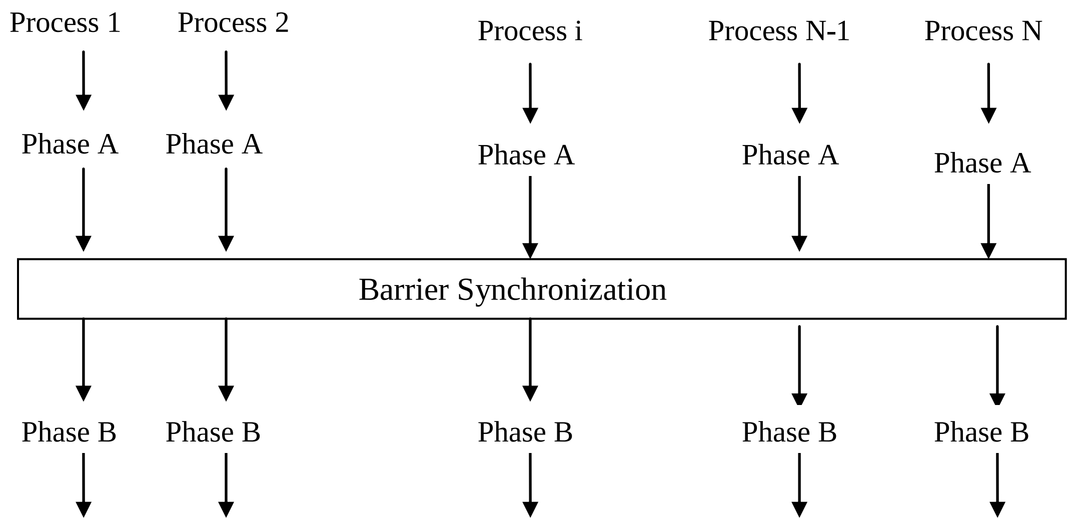
```
Parent Process:  /* No error checking code needed */
  int fdA[2];
  int fdB[2];
  pipe( fdA );     // create pipe A
  pipe( fdB );     // create pipe B

 if ( fork() ) {











} else {











}
```

Part (b) (10 points): What will be the output produced by the following program?

```
int i = 0;
void myalarm ( int signum ) {
  i = 4;
}
void main () {
   signal( SIGALRM,   myalarm );
   alarm(3);
   for (i=0; i<7; i++ ){
      printf("Hello %d\n", i);
      sleep( 2 );
   }
}
```

Question 4 (20 points): *Barrier Synchronization Problem:* A system contains N processes. Each of these processes executes certain computation phase A, it then reaches the barrier synchronization point and waits for all other processes to finish their execution of phase A and reach the barrier. A process can proceed to execute phase B only after all N processes have reached the barrier and executed the barrier synchronization code.

<u>You are asked to write barrier synchronization code using semaphores.</u>
<u>Your code should be written in pseudo-code notation as used in class examples.</u>



/* Declare semaphores and other shared variables here */

Procedure BarrierSychronization( ) {
/* Write barrier synchronization code here - using pseudo code notation */

Continue here answer for Question 5…

Blank page

Blank page