York University
Lassonde School of Engineering
Department of Electrical Engineering and Computer Science

Midterm
EECS 3221 Z Operating Systems Fundamentals
Feb. 25, 2016 (2:30-4:00pm)

Family Name: _____

Given Name: _____

EECS Account: _____**Solution**_____

Student Id: _____

Instructions

1. The exam has 7 questions and 10 pages (including this one).
2. No aids permitted.
3. Answer each question in the space provided. If you need more space write on the backs of pages.
4. **Examination time is 90 minutes.**
5. Answers written in pencil or erasable ink will *not* be considered for remarking.
6. Do *not* use red ink. Write legibly. Unreadable answers do *not* count.
7. **No questions re the interpretation, intention, etc. of an exam question will be answered by invigilators. If in doubt, state your interpretation as part of your answer.**

Question	Maximum	Mark
1	8	
2	8	
3	7	
4	11	
5	10	
6	8	
7	8	
Total	60	

1. (8 marks) Interrupt handlers are a critical component of OS kernel. Please complete the following description about how an interrupt is handled in a sequential vectored interrupt system.

MARKING SCHEME: deduct one mark for each wrong or blank spot.

I) CPU normally executes instructions one by one until an interrupt happens. Give two examples of events that may cause an interrupt:

(I.1) [**any hardware events, such as a stroke in keyboard, completion of I/O event, hardware failure, timer, I/O device and so on**]

(I.2) [**software interrupt such as TRAP instruction**]

II) When an interrupt happens:

(II.1) _____ **Hardware** _____ (choose between **Hardware** or **OS kernel**) switches CPU to _____ **kernel** _____ mode.

(II.2) Load PC register according to _____ **interrupt vectors** _____ and jumps to run the corresponding interrupt handler.

III) An interrupt handler typically executes the following steps:

III. 1) _____ **disable interrupt** _____.

III. 2) Save _____ **CPU status or CPU context (registers)**_____.

III. 3) Deal with the interrupt.

III. 4) Restore _____ **CPU status or CPU context (registers)**_____.

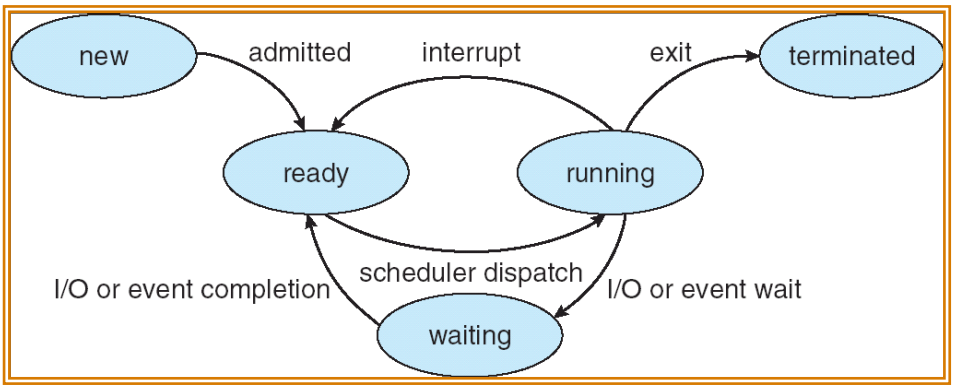
III. 5) _____ **enable interrupt** _____.

III. 6) _____ **OS kernel switches CPU to user mode**_____if it returns to user program.

III. 7) Return from the interrupt handler (restore PC register) .

IV) CPU continues to execute next instruction.

2. (8 marks) Draw a diagram of the five-state process model, showing all states and transitions. Label each transition with possible events that may cause such a transition.



Marking schemes:

*) Diagram (4 marks): label each state correctly; draw all transition correctly, deduct point for any extra transition.

*) Label transitions (4 marks).

3. (7 marks) Indicate whether each of the following statements is **true** or **false**. Write the entire word ``**true**'' or ``**false**'' in the box after each statement. **One mark for each correct answer; No mark deduction for wrong answers.**

(3.1) All system calls are implemented as part of an interrupt handler in kernel space.

[**TRUE**]

(3.2) DMA controller uses an interrupt to inform OS that it has finished data transmission.

[**TRUE**]

(3.3) In multiprogramming system, a process is always running in its CPU burst.

[**FALSE**]

(3.4) Multiple threads that are part of the same process have their own program counters and stack pointer values. [**TRUE**]

(3.5) Multiple threads that are part of the same process have their own copies of global variables.

[**FALSE**]

(3.6) Multiple threads that are part of the same process share all open files.

[**TRUE**]

(3.7) Caching works because memory accesses are totally random.

[**FALSE**]

4. (11 marks) Short questions:

4.1) (4 marks) In a dual-mode CPU architecture, specify which category (**normal** or **previledged**) each of the following instructions belongs to:

(a) TRAP instruction [**normal**]

(b) Turn off interrupt [**previledged**]

(c) Add two general registers and save the result to memory

[**normal**]

(d) Read status of an I/O port [**previledged**]

4.2) (2 marks) Under what situations are context switches allowed to occur in non-preemptive schedulers? What about preemptive schedulers?

Four cases:
when a process
1. Switches from running to waiting state.
2. Switches from running to ready state.
3. Switches from waiting to ready.
4. Terminates.

Preemptive kernels: context switch happens at all the above cases
Non-preemeptive kernels: context switch happens only at cases 1 and 4.

OR

Preemptive kernels: context switches may occur in middle of CPU bursts.
Nonpreemptive kernels: context switches happen only at the end of CPU bursts.

(2 marks)

4.3) (2 marks) Briefly explain why preemptive kernels are harder to design than non-preemptive ones? Use examples to show why preemptive kernels are more complicated.
In preemptive kernels, it is possible to have more than two processes that are concurrently running in kernel space.

P1: running; → make a system call → go to kernel space; in middle of the system call, context switch happens ….

CPU scheduler picks up another process P2

P2: running; → make the same system call → go to kernel space as well

P1 and P2 are running the same code in kernel space. Therefore, kernel space needs protection for preemptive kernels

(2 points)

4.3) (3 marks) Use point form to explain the major steps that OS kernel takes to make context switch from process P0 to process P1?

Step 0: Context switch is triggered by an interrupt (timer, TRAP, etc.). Control goes to CPU scheduler.

Step 1: Dispatcher saves context of P0 into its PCB, including registers, memory space, etc.

Step 2: CPU scheduler selects next process from all ready processes in the ready queue, i.e. P1.

Step 3: Dispatcher restores CPU status from its PCB, including registers, memory space, etc, and resume P1's execution.

(3 points)

5. (10 marks) Please describe all possible outputs for the following two programs when they run **normally** in a Unix system. Also clearly explain how and why each output happens.

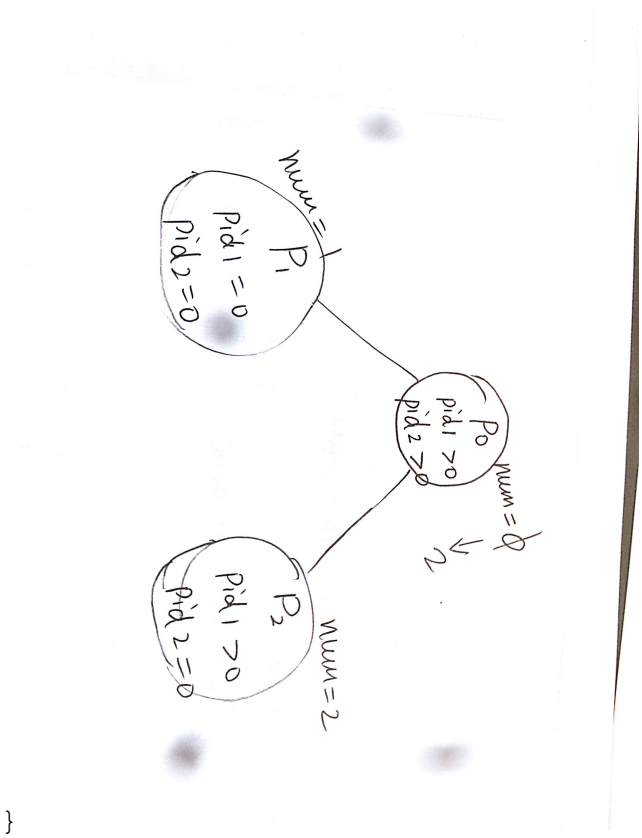
5.1) (5 marks) Program A:

```
#include <stdio.h>
#include <unistd.h>
int num = 0 ;
int main(int argc, char *argv[])
{
  int pid1=0, pid2=0 ;

  pid1 = fork() ;

  printf("%d\n",num) ;

  if (pid1 == 0) {
    num = 1;
  } else if (pid1 > 0) {
    num = 2 ;
    pid2 = fork() ;
  }
  if (pid1>0 || pid2==0)
        printf("%d\n",num) ;
```



}

prints out two '0's, one '1' and two '2'.

0 1 0 2 2 (7 possible orders)

0 0 1 2 2

0 0 2 1 2

0 0 2 2 1

0 2 0 1 2

0 2 0 2 1

0 2 2 0 1

5.2) (5 marks) Program B:

```
#include <pthread.h>
#include <stdio.h>
int X = 0, Y = 0 ;
void *runner1()
{
  for(; X<4; X++) {
   Y = 0 ;
   printf("%d",X) ;
   Y = 4;
```

```
   }
}
void *runner2()
{
  while (X<4) {
     if ( Y != 0) printf("%d",Y) ;
  }
}
int main(int argc, char *argv[])
{
   pthread_t tid1, tid2; /* the thread identifier */
    /* create the thread 1 & 2*/
   pthread_create(&tid2,NULL,runner2,NULL);
   pthread_create(&tid1,NULL,runner1,NULL);
   pthread_join(tid2,NULL) ;
   pthread_join(tid1,NULL) ;
}
```

0 4 … 4 1 4 … 4 2 4 … 4 3 4 … 4

1) interleaving of two threads due to unknown execution order

2) thread1 prints 0, 1, 2, 3 in order; thread2 prints a number of 4 only when thread1 turns Y to 4.

answer "0 a…a 1 a … a 2 a … a 3 a … a" get 0 mark.

6. (8 marks) Here is a table of processes and their arrival times and CPU burst lengths:

Process ID	Arrival Time	CPU burst
P1	0	5
P2	1	3
P3	2	8
P4	3	6

Assume one CPU is available, show the scheduling order for these processes under FCFS, preemptive Shortest-Job First (SJF), and Round-Robin (RR) (quantum=3 time units). Fill the table to show which process CPU is running at each time. *Assume that context switch overhead is 0. In RR, assume time-quantum-expired process is given priority over new arrival process.*

Time	FCFS	SJF	RR
0	**P1**	**P1**	**P1**
1	**P1**	**P2**	**P1**
2	**P1**	**P2**	**P1**
3	**P1**	**P2**	**P2**

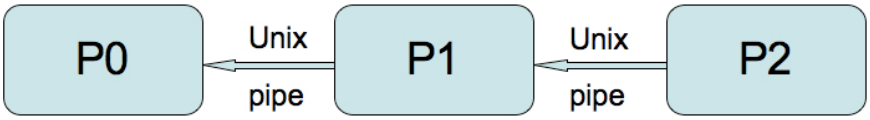
4	**P1**	**P1**	**P2**
5	**P2**	**P1**	**P2**
6	**P2**	**P1**	**P3**
7	**P2**	**P1**	**P3**
8	**P3**	**P4**	**P3**
9	**P3**	**P4**	**P1**
10	**P3**	**P4**	**P1**
11	**P3**	**P4**	**P4**
12	**P3**	**P4**	**P4**
13	**P3**	**P4**	**P4**
14	**P3**	**P3**	**P3**
15	**P3**	**P3**	**P3**
16	**P4**	**P3**	**P3**
17	**P4**	**P3**	**P4**
18	**P4**	**P3**	**P4**
19	**P4**	**P3**	**P4**
20	**P4**	**P3**	**P3**
21	**P4**	**P3**	**P3**

Calculate the average waiting time for each of these three scheduling algorithms:

FCSF: ____**(0+4+6+13)/4 =5.75**_____ SJF: ___**(3+0+5+12)/4 =5.0**___

RR: _____**(6+2+12+11)/4 =7.75**_____

7. (8 marks) Write a C program to implement the following three processes in a Unix system:



where `P0` is parent of `P1` and `P1` is parent of `P2`. Two Unix pipes are created from child proces to parent process as above. In your program, `P2` sends its pid to `P1` through the pipe; `P1` is waiting to read whatever message from this pipe and passes it to `P0` through another pipe; `P0` is waiting to read the message and print onto the screen.

System calls for reference:

```
#include <unistd.h>
  ssize_t read(int fd, void *buf, size_t count);
  ssize_t write(int fd, const void *buf, size_t count);
  int close(int fd);
```

```
  int pipe( int filedes[2] ) ;
  pid_t fork(void);
  pid_t getpid(void);

#include <unistd.h>
#include <stdlib.h>
int main() {
  int n, fd1[2], fd2[2];
  pid_t  pid ;
  long int num ;

  if( pipe(fd1) < 0 )   err_sys("pipe1 error") ;
  if ( (pid = fork()) < 0 ) err_sys("fork error") ;
  else if ( pid > 0 )  { /* parent P0 */
      pid_t num ;
      close(fd1[1]) ;
      read(fd1[0], &num, sizeof(num)) ;
      print("%d\n", num) ;
 } else  {     /* child process P1 */
   close(fd1[0]) ;
   if (pipe(fd2) <0)  err_sys("pipe2 error") ; ;
   if ( (pid = fork()) < 0)  err_sys("fork error") ;
   else if ( pid == 0 )  {  /* grand-child process P2 */
      close(fd2[0]) ;
      pid = getpid(void);
      write(fd2[1], &pid, sizeof(pid)) ;
   } else {  /* Process P1 */
      pid_t num;
      close(fd2[1]) ;
      while (read(fd2[0], &num, sizeof(num))==0) ;
      write(fd1[1], &num, sizeof(num) ) ;
  }
```