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

## Midterm
## EECS 3221E Operating Systems Fundamentals
## Nov 1, 2018 (4:00-5:30pm)

Family Name: ____________**Solution** ______________________________

Given Name: ______________________________________________

EECS Account: ____________________________________________

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 | 7 | |
| 3 | 8 | |
| 4 | 12 | |
| 5 | 12 | |
| 6 | 9 | |
| 7 | 9 | |
| Total | 65 | |

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 what events 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**_____ (Hardware or OS kernel) switches CPU to _____**kernel**___________ mode.

(II.2) Load PC register according to _______**interrupt vectors** ___________________ and jumps to run an 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)**___________.

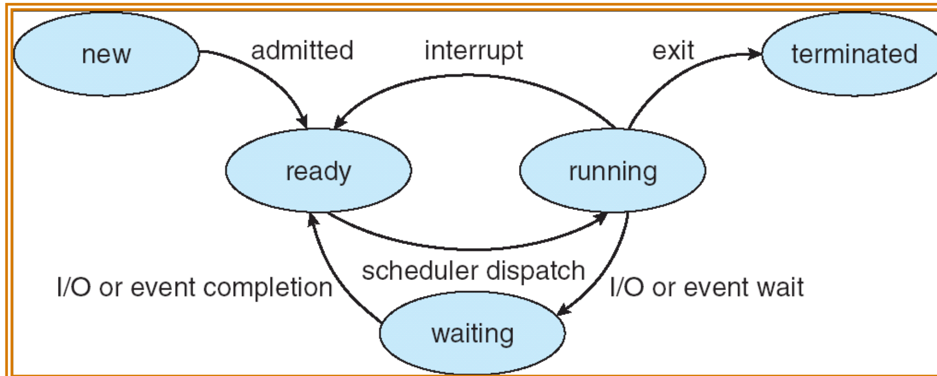
III. 5) ____**enable interrupt** _____________.

III. 6) ______**OS kernel switches CPU to user mode**______ if it returns to user programs.

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

**IV)** CPU continues to execute next instruction.

2. (7 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: deduct at least ONE point for each mistake

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

*) Label transitions (3 marks): at least one event for centred transitions; deduct one point for any extra transition.

3. (8 marks) 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; One mark deduction for each wrong answer. No mark for a blank answer. DO NOT GUESS.**

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

[ **TRUE** ]

(3.2) DMA controller uses an interrupt to inform CPUs after every byte of data is sucessfully transferred to memory. [ **FALSE** ]

(3.3) In multiprogramming systems, a process or thread is always in the "running" state during its CPU bursts. [ **FALSE** ]

(3.4) Multiple threads that are a part of the same process keep track of their own program counters and stack pointers. [ **TRUE** ]

(3.5) Multiple threads that are a 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 CPUs' memory accesses are totally random.

[ **FALSE** ]

(3.8 ) In a non-preemptive CPU scheduler, CPU is assigned to run a process and the process will hold CPU until this process terminates. [ **FALSE** ]

4. (12 marks) Short questions:

4.1) (5 marks) In a dual-mode CPU architecture, specify which category (**normal** or **previledged**) each of the following instructions belongs to (**DO NOT GUESS. One mark deduction for each wrong answer. No mark for a blank answer.**):

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

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

(c) Bit shifting left [ **normal** ]

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

(e) Change the memory map [ **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) detect one mark for each error.**

4.3) (2 marks) Briefly explain why preemptive schedulers 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. (1 point)**

**(1 point for examples)**

**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**

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, detect one mark for each error. )**

5. (12 marks) Please describe all possible outputs for the following two programs when they run **normally** in a Unix system. Also clearly explain how each output happens.

5.1) (6 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;
    pid2 = fork() ;
  } else if (pid1 > 0) {
    num = 2 ;

  }
  if (pid2==0)
        printf("%d\n",num) ;
}
```

**prints out two ‘0’s, one ‘1’ and one ‘2’.**

**0 1 0 2 (4 possible orders)**

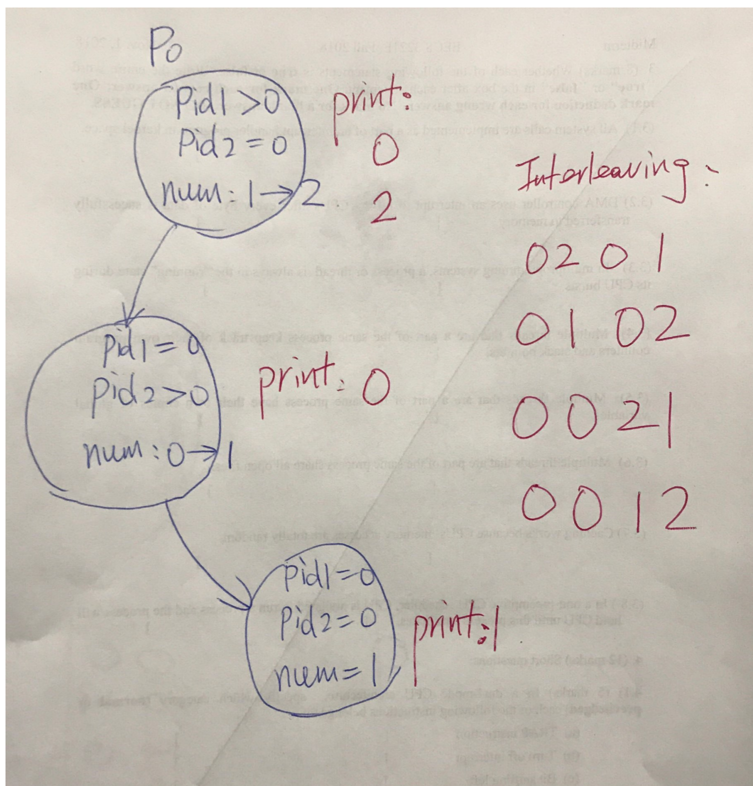
**0 0 1 2**

**0 0 2 1**

**0 2 0 1**

**(4 points: deduct one point for one missing case; deduct 2 points for each extra case)**

**explanation: (2 points)**




Scanned with CamScanner

5.2) (6 marks) Program B:

```
#include <pthread.h>
#include <stdio.h>
```

```
int X = 0, Y = 0 ;
void *runner1()
{
  for(; X<4; X++) {
   sleep(1);   /* goes to waiting for 1 second */
   Y = 1 ;
   printf("%d",X) ;
   Y = 0;
  }
}
void *runner2()
{
  while (X<4) {
     if ( Y != 0) printf("%d",X) ;
  }
}
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 … 0 1 … 1 2 …2 3 … 3**

**(3 points for perfect results; deduct 1 point for each minor errors; 0 points for any major error)**

**Explanation (3 points)**

**1) interleaving of two threads due to unknown execution order**

**2) thread1 prints 0, 1, 2, 3 in order; thread2 prints a number of X=0/1/2/3 only when thread1 turns Y to 1 (when printing the same X).**

**answer “0 a…a 1 a … a 2 a … a 3 a … a” get 0 mark.**

**answer “0 4 … 4 1 4 … 4 2 4 … 4 3 4 … 4” get 0 mark.**

6. (9 marks) Here is a table of processes and their arrival times and CPU burst lengths:

| Process ID | Arrival Time | CPU burst |
|---|---|---|
| P1 | 0 | 10 |
| P2 | 1 | 5 |
| P3 | 2 | 3 |
| P4 | 3 | 3 |

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.*

**6 marks: 2 marks for each column; detect ½ point for each error**

| Time | FCFS | SJF | RR |
|---|---|---|---|
| 0 | **P1** | **P1** | **P1** |
| 1 | **P1** | **P2** | **P1** |
| 2 | **P1** | **P3** | **P1** |
| 3 | **P1** | **P3** | **P2** |
| 4 | **P1** | **P3** | **P2** |
| 5 | **P1** | **P4** | **P2** |
| 6 | **P1** | **P4** | **P3** |
| 7 | **P1** | **P4** | **P3** |
| 8 | **P1** | **P2** | **P3** |
| 9 | **P1** | **P2** | **P1** |
| 10 | **P2** | **P2** | **P1** |
| 11 | **P2** | **P2** | **P1** |
| 12 | **P2** | **P1** | **P4** |
| 13 | **P2** | **P1** | **P4** |
| 14 | **P2** | **P1** | **P4** |
| 15 | **P3** | **P1** | **P2** |
| 16 | **P3** | **P1** | **P2** |
| 17 | **P3** | **P1** | **P1** |
| 18 | **P4** | **P1** | **P1** |
| 19 | **P4** | **P1** | **P1** |
| 20 | **P4** | **P1** | **P1** |
| 21 | | | |

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

**1.5 marks:**

FCSF: ____ **(0+9+13+15)/4 =9.25**___ SJF: ___**(11+6+0+2)/4 =4.75**_____

RR: ___**(11+11+4+9)/4 =8.75**____

Count the total number of incurred context switches for each algorithm:

**1.5 marks:**

FCSF: ______**4**_____ SJF: _______**6**________ RR: _______**8**___________

7. (9 marks) The following method is proposed to keep track of how many times `runner()` is exucuted.

```
int Counter = 0;
void *runner()
 {
   Counter++;

       /* Does its work here */
       …

   return ;
 }
```

Consider the following three different ways to use the above method and answer whether you can print the correct value of how many times `runner()` is called in each case. For case **(A)** or **(B)** or **(C)**, please answer YES or NO first and then briefly justify your answer.

**Marking scheme: 0 mark if YES/NO is wrong; otherwise Y/N -> 1 point; explanation->2 points**

**(A)**

```
int main(int argc, char *argv[])
 {
   pthread_t tid[MAXSIZE]; /* the thread identifiers */

   for(int i=0; i<MAXSIZE; i++)
      pthread_create(&tid[i],NULL,runner,NULL);

   for(int i=0; i<MAXSIZE; i++) pthread_joint(tid[i],NULL) ;


   printf("runner()is called %d times in total.\n", Counter);
 }
```

**NO, it can't always print a correct value because of race condition.**

**Multiple threads run concurrently and are modifying the same variable 'Counter' at the same time without any protection. Context switch may cause execution sequences of these threads interleaving so that the resultant value becomes unexpected.**

**Need to use line 4 and line 5 to explain how the execution sequences are actually interleaved.**

**(B)**
```
int main(int argc, char *argv[])
 {
   pthread_t tid[MAXSIZE]; /* the thread identifiers */

   for(int i=0; i<MAXSIZE; i++) {
       pthread_create(&tid[i],NULL,runner,NULL);
       pthread_joint(tid[i],NULL) ;
   }
   printf("runner()is called %d times in total.\n", Counter);
 }
```

**YES, no race condition happens because all child threads are created and run sequentially. They will modify 'Counter' one by one so that no race condition happens in this case. It keeps the correct value of 'Counter'.**

**(C)**
```
int main(int argc, char *argv[])
 {
   for(int i=0; i<MAXSIZE; i++)  runner() ;

   printf("runner()is called %d times in total.\n",Counter);
 }
```

**YES, no race condition happens because ONLY one process(or thread) is invloved in this case. The only process/thread modifies 'Counter' from time to time so that no race condition happens in this case. It keeps the correct value of 'Counter'.**