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

Midterm
EECS 3221.03Z Operating Systems Fundamentals
Feb 26, 2015 (14:30-16:00)

Section: **EECS3221Z**

Family Name: _____

Given Name: _____

CS Account: _____

Student Id: _____ **Solution** _____

Instructions

1. The exam has 6 questions and 8 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. Generally, 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	12	
5	8	
6	7	
Total	50	

1. (8 marks) It is an important job for OS to protect all sorts of resources in a computer system. The basic protection strategy in multiprogramming OS is to use a dual-mode CPU.

   (a) What are these two CPU modes named? What is the major differences between these two CPU modes?

   Kernel mode (or system mode or monitor mode) and user mode

   In kernel mode, CPU can execute all instructions (both normal and previleged instructions)

   In user mode, CPU can execute only normal instructions and CPU will generate an error if it is foreced to execute a previleged instruction in user mode)

   (2 marks)

   (b) Explain why OS needs such a dual-mode mechanism for effective hardware protection.

   The same CPU sometimes needs to run OS code to manage computer resources (CPU needs super-power in this case). The same CPU sometimes needs to run user programs (CPU power must be limited in order to protect any user program from damaging the system).

   (2 marks)

   (c) How do OS and CPU hardware together guarantee a correct logic in implementing dual-mode operation?

To guarantee that whenever CPU runs OS code, it is in kernel mode; whenever CPU runs user code, it is in user mode.

Consider three cases:

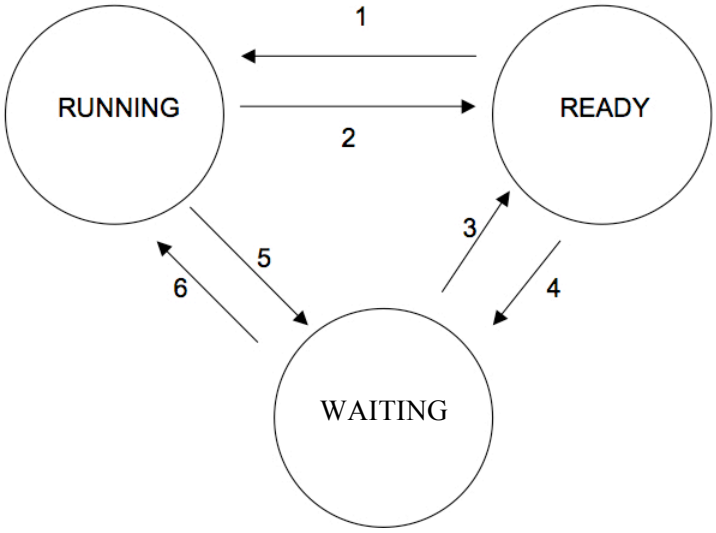
i) After system rebooting, CPU in kernel mode and OS takes control of CPU.

ii) OS needs to explicitly switch CPU to user mode before it passes CPU to any user program.

iii) When an interrupt happens, hardware forces CPU to run interrupt handler program (part of OS kernel) and meanwhile it forces CPU switch to kernel model.

(4 marks)

2. (8 marks) You are given a three-state process model as follows.



For each of the following transitions, answer whether it is allowed or not (circle one). If allowed, show an event that triggers such a transition; if not allowed, briefly explain why in 1-2 sentences.

Marking scheme: 2 marks each; correct choice 1 mark; good explanation 1 more mark

(1) RUNNING → READY [Allowed **X** Not Allowed]

time slice expires may cause this; or another more important process becomes ready

(2) WAITING → RUNNING [Allowed Not Allowed **X**]

all waiting processes will first go to ready first; it needs CPU scheduler to decide who goes from ready to running.

(3) READY → WAITING [Allowed Not Allowed **X**]

a process can't become 'waiting' from 'ready to run' without running in CPU

(4) RUNNING → WAITING [Allowed **X** Not Allowed]

the running process make an IO requestion; or IO-reated system calls

3. (7 marks) Briefly answer the following questions about system calls in OS.

(1) (1 mark) What system calls are used for in OS?

system calls are used for OS to provide services to other applications or programs;

OR

normal program can make system calls to request OS services

(2) (2 marks) What differences are between system calls and function calls?

function calls are used for requesting services in the same space (either user or kernel)

system calls are used for requesting services in kernel space from user space; it crosses the space boundary so that it needs to switch CPU mode (user→kenerl) at the same time

(3) Briefly expalin how sytem calls are implemented in OS? Consider its implementation in both user space and kernel space.

a. (2 marks) In User Space:

i) save system call number and all parameters

ii) execute software interrupt instruction (TRAP)

b. (2 marks) In Kernel Space:

i) All system calls are implemented in the interrupt handler program correponding to TRAP

ii) where jump to dfferent parts based on the saved system call number; after it is done, switch back to user mode and return to user program

4. (12 marks) Assume the following programs runs **normally** in Unix.

4.1) (6 marks) Program A:

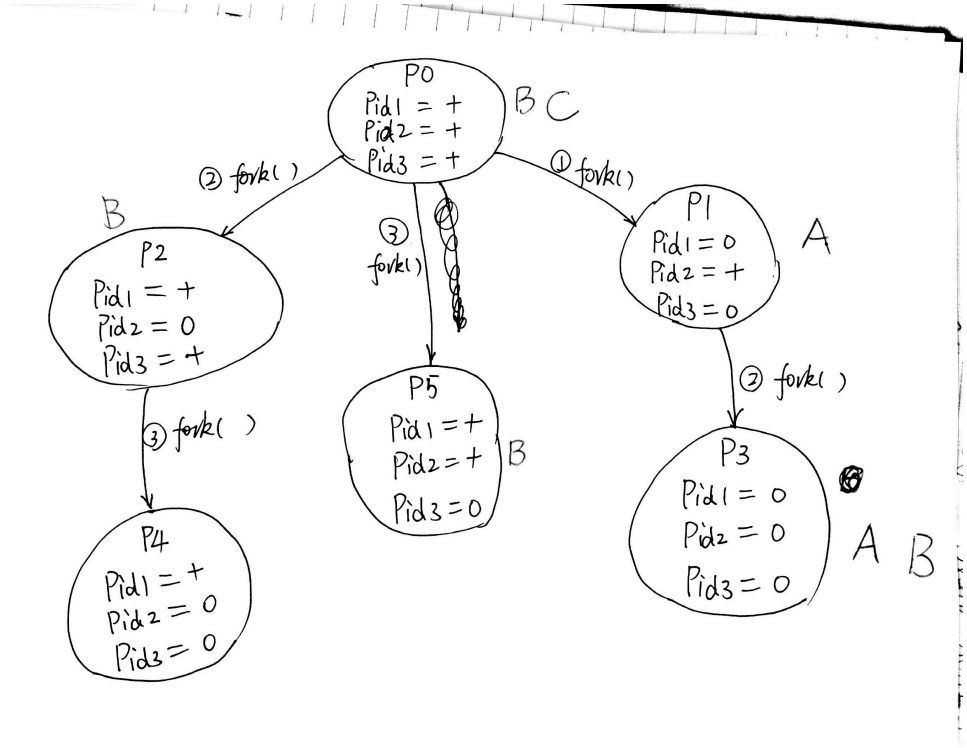
```
#include <stdio.h>
int num = 0 ;
int main(int argc, char *argv[])
{
  int pid1=0, pid2=0, pid3=0;

  pid1 = fork() ;
  pid2 = fork() ;
  if(pid1 == 0 ) {
      printf("A\n") ;
  } else {
      pid3 = fork() ;
      printf("B\n") ;
  }
  if (pid2 != 0 && pid3 != 0) {
    printf("C\n") ;
  }
}
```

What is the total number of processes that will be created by runnng this program? How many of each character 'A' to 'C' will be printed out? Briefly explain why.

2 A's 4 B's 1 C (2 marks)

6 processes (incldluing the original one) (2 marks)

Need a brief explanation about the following hierarchical structure (2 marks)



4.2) (6 marks) Program B:

```
#include <pthread.h>
#include <stdio.h>
int value = 0 ;
void *runner(void *param) {
     value = 5 ;
     pthread_exit(0) ;
 }
int main( int argc, char *argv[]) {
   int pid;
   pthread_t tid ;
   pthread_attr_t attr ;

   pid = fork() ;
   if (pid == 0)  {
     pthread_attr_init(&attr) ;
```

```
    pthread_create(&tid, &attr, runner, NULL) ;
    pthread_join(tid,NULL) ;
    printf("value = %d\n", value) ;
  } else if (pid > 0)  {
    value = 10 ;
    wait(NULL) ;
    printf("value = %d\n", value) ;
  }
}
```

Please describe all possible outputs for the following two programs and explain how each output happens.

Correct output:

value = 5
value = 10

Explanation:

***) fork() creates two processes and gets two copies of value;**

***) Child process create a new thread -> two threads in child process and both threads share one copy of value in child process. The new thread change it from 0 to 5 ; the other thread wait and then print its value as "value=5".**

***) Parent process changes its own copy of value from 0 to 10 and wait and then print its value as "value=10"**

***) since parent wait(NULL), the child process always print before parent process.**

5. (8 marks) Here is a table of processes and their arrival times and CPU burst lengths:

Process ID	Arrival Time	CPU burst
P1	0	4
P2	2	6
P3	3	2
P4	7	4

Assume there is one CPU available, show the scheduling order for these processes under FCFS, preemptive Shortest-Job First (SJF), and Round-Robin (RR) (quantum=1 time unit). Fill the table to show which process CPU is running at each time. *Assume that context switch overhead is 0. In RR, assume new arrival process is given priority over time-quantum-expired process.*

Time	FCFS	SJF	RR
0	**P1**	**P1**	**P1**
1	**P1**	**P1**	**P1**
2	**P1**	**P1**	**P2**
3	**P1**	**P1**	**P1**
4	**P2**	**P3**	**P3**
5	**P2**	**P3**	**P2**
6	**P2**	**P2**	**P1**
7	**P2**	**P4**	**P3**
8	**P2**	**P4**	**P2**
9	**P2**	**P4**	**P4**
10	**P3**	**P4**	**P2**
11	**P3**	**P2**	**P4**
12	**P4**	**P2**	**P2**
13	**P4**	**P2**	**P4**
14	**P4**	**P2**	**P2**
15	**P4**	**P2**	**P4**

Calculate the waiting time for these three scheduling algorithms:

FCSF: _____**14**_____

SJF: _____**9**_____

RR: _____**18**_____

6. (7 marks) The following method is proposed to keep track of how many child threads are currently active in a multi-threaded program.

```
/*1*/ int NumThreads = 0;
/*2*/ void *runner()
/*3*/ {
/*4*/    NumThreads++;

        /* child thread does its work here */
        …

/*5*/    NumThreads--;
/*6*/    return ;
/*7*/ }

/*8*/  int main(int argc, char *argv[])
/*9*/  {
/*10*/   pthread_t tid1[MAXSIZE]; /* the thread identifiers */

/*11*/   for(i=0; i<MAXSIZE; i++) {
/*12*/      pthread_create(&tid[i],NULL,runner,NULL);
/*13*/   }

/*14*/  printf("Currently %d threads are working\n", NumThreads);

/*17*/ }
```

Explain whether `line 14` always prints out the correct number of active threads that are currently running between `lines 4 and 5` (not including `lines 4 and 5`). Please answer YES or NO first and then justify your answer.

(answer Yes → 0 mark; answer No but without justification, 2 marks maximum)

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

Multiple threads are modifying the same variable 'NumThreads' 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.