CS2106 Operating Systems
2016/17 Semester II
Mid-Term Test

Instructions

1. This paper consists of 15 questions on 8 printed pages including this page. Please check that you have the correct number of questions and printed pages.

2. Shade AND write your matric number on the OCR answer sheet provided. DO NOT WRITE YOUR NAME OR ANY OTHER PERSONAL DETAILS.

3. For each of the questions below, pick the BEST answer possible, and shade the corresponding lozenge on your OCR answer sheet.

4. DO NOT communicate with anyone during the test. If you have any questions please raise your hand HIGH and an invigilator will attend to you as soon as possible.

5. DO NOT borrow any items from anyone. This includes but is not limited to erasers, pens, pencils, calculators, etc.

6. This test is OPEN BOOK, so you can use any PRINTED/HAND-WRITTEN notes, lab reports, textbooks, etc. However no computing devices other than calculators are allowed.

7. This test is 45 minutes long.

Questions

1. Which ONE of the following statements is TRUE? Pick the best answer.

    a. Monolithic kernels are compiled as a whole unit and you cannot load and unload kernel modules on demand. **FALSE. LINUX kernel is a counter-example of a monolithic kernel that can load and unload parts of itself.**
    b. Microkernel architectures have their main kernel routines running in user space. **FALSE. If kernel routines run in user space they are not secure.**
    c. Monolithic kernels run all code in kernel space. **FALSE. If user code runs in kernel space, it compromises the safety of the kernel.**
    d. **+Microkernel architectures run their device drivers in user space. TRUE. Microkernels only keep important kernel functions in kernel space and run everything else in user space, including device drivers.**
    e. None of the options a. to d. above are correct.

2. Why do most operating systems utilize a boot-strap sequence to load and start?
    a. **+It allows the OS to first start and run a simple portion of itself, and as more capabilities are loaded and more of the system is initialized, the bootstrap sequence allows the OS to load and start more and more sophisticated chunks of itself. TRUE. E.g. bootstrap routine in boot sector loads up core disk services, and use this to load up memory and process management routines, and use that to load up services like web servers, ssh servers, etc, and finally load up the shell.**
    b. The bootstrap sequence makes OS start-up more secure. **FALSE. There is nothing in the boot sequence that makes the start-up more secure. E.g. you can replace a module in the kernel with something malicious and the kernel wouldn't known (e.g. lack of digital signatures, etc.)**
    c. The bootstrap sequence allows us to load an operating system that is bigger than the amount of memory available. **FALSE. There is nothing in the process that lets you squeeze in more OS than memory.**
    d. The bootstrap sequence allows an OS to start a large chunk of itself, then selectively unload parts of itself if too much capability has been loaded. **FALSE. This is the opposite of what happens, where the OS loads more and more of itself.**
    e. None of the above options a. to d. are correct.

3. The following 5 jobs are scheduled in a batch system under Shortest Job First. What is their total running time?

   Job 1: 150 clock cycles.
   Job 2: 75 clock cycles.
   Job 3: 270 clock cycles.
   Job 4: 55 clock cycles.
   Job 5: 92 clock cycles.

   Total running time is the time between the start of the first job and the end of the last job, and is independent of job scheduling algorithm.

   Answer = 150 + 75 + 270 + 55 + 92 = 642 cycles
   a. 779 clock cycles.
   b. 156 clock cycles.
   c. +642 clock cycles.
   d. 372 clock cycles.
   e. None of the above options a. to d. are correct.

4. Again using Shortest Job First for the jobs in question 3, what is the worst-case waiting time?

   We sort by job length, then find the waiting time for each job.

   Job 4: 55, Job 2: 75, Job 5:92 Job 1: 150 Job 3: 270

   Waiting time for Job 4: 0
   Waiting time for Job 2: 0+55 = 55
   Waiting time for Job 5: 0 + 55 + 75 = 130
   Waiting time for Job 1: 0+55+75+92 = 222 cycles
   Waiting time for Job 3: 0 + 55 + 75 + 92 + 150 = 372

   So the worst case waiting time is 372 cycles.

   a. 779 clock cycles.
   b. 156 clock cycles.
   c. 642 clock cycles.
   d. +372 clock cycles.
   e. None of the above options a. to d. are correct.

5. Who is the current President of the United States?

   a. +Donald John Trump. Yes, read the news.
   b. Colin K. Y. Tan. **NO. This is me.**
   c. Mohan Kankanhalli. **NO. This is the Dean of SoC**

d. Tsubasa Amami. **NO. This is a Japanese "alternative entertainment" star.**

e. All of the above options a. to d. are correct.

6. Tasks in Rate Monotonic Scheduling Systems are ordered by task periods because:

   a. Safety critical tasks generally have shorter periods. **FALSE. Task periods are determined by other factors (e.g. sensor frequency) and not by safety.**
   b. Tasks with shorter periods tend to be more interactive. **FALSE. RTOS tasks are not typically interactive, and interactivity has no relevance to period.**
   c. +Tasks with shorter periods have shorter deadlines. TRUE. Task deadlines
   d. Tasks with longer periods are not essential.
   e. None of the above options a. to d. are correct.

7. We are given the following sets of tasks in a FIXED PRIORITY scheduling system:

   Task T1: C1 = 1 cycle, P1 = 4 cycles, priority = LOW.
   Task T2: C2 = 2 cycles, P2 = 8 cycles, priority = HIGH.
   Task T3: C3 = 3 cycles, P3 = 12 cycles, priority = MEDIUM

   What is the worst case running time for task T2, including pre-emptions?

   a. 2 cycle. T2 neve gets pre-empted and always gets to run its full number of cycles, which is 2.
   b. 6 cycles
   c. 8 cycles
   d. 12 cycles
   e. None of the options above a. to d. are correct.

8. Repeat the question above: What is the worst case running time for task T1?

   Do CIA:

   S1,0 = C2 + C3 + C1 = 6 cycles
   Higher priority tasks are T2 and T3 with periods of 8 and 12. Since S1,0 is less than any of these, there are no further pre-emptions.

   a. 2 cycle
   b. +6 cycles
   c. 8 cycles
   d. 12 cycles
   e. None of the options above a. to d. are correct.

9. Which ONE of the following statements is TRUE above the set of tasks in Question 7?

   a. +This set of tasks is NOT SCHEDULABLE, with task T1 missing its deadline.
   b. This set of tasks is NOT SCHEDULABLE, with task T2 missing its deadline.
   c. This set of tasks is NOT SCHEDULABLE, with tasks T1 and T3 missing their deadlines.
   d. This set of tasks IS SCHEDULABLE.
   e. None of the above options a. to d. above are correct.

10. Which ONE of the following statements is true above the LINUX scheduler?

   a. LINUX Real-Time FIFO (RT-FIFO) tasks are guaranteed to meet their deadlines. **FALSE. RT-FIFO queues are fixed priority queues, and a higher priority task can hold on to the CPU indefinitely causing lower priority tasks to miss deadlines.**
   b. +Without the expired set, only tasks in the highest priority queue will get to run. TRUE. Tasks finish their time quanta have to be placed "somewhere" to be run again, and since there's no expired set, that "somewhere" is to the back of the queue. When these tasks reach the front again they will be run, starving all the other queues.
   c. Tasks in the RT-FIFO queue are given time quanta like all other tasks. **FALSE, RT-FIFO tasks run until pre-empted by higher priority tasks.**
   d. Tasks that have been blocked on I/O for a long time are given less CPU time, since they don't require much CPU time. **FALSE. In fact they are boosted and given MORE CPU time to catch up.**

e. None of the above options a. to d. are correct.

The code fragment below shows a very misguided attempt to coordinate between two processes by using mutexes. Use this fragment to answer questions 11 and 12.

```
//Initialize mutex to an unlocked state.
mutex_t coord = MUTEX_UNLOCKED;

void wait()
{
     lock_mutex(&coord);
}

void blockProcess()
{   // Note: No mistake; code is identical to wait()
    lock_mutex(&coord);
}

void unblockProcess()
{
    unlock_mutex(&coord);
}

int x = 0; // This variable is shared between processes

int main()
{
      //spawnThread works like fork() except that global
      // variables like x and coord are shared.
      if(spawnThread())
      {
             // Parent
             //  Block waiting child process to update x
              blockProcess();
              x = x + 5;
             unblockProcess();
      }
      else
     {
             // Child
             // Wait for x to be updated
             wait();

           // Use result put into x by parent.
           y = x * 2;
     }
}
```

11. This piece of code does not work. Why? (When analyzing the answers, you should only consider the code as given above.)

    a. There is a possibility that unblockProcess unlocks the mutex before blockProcess locks it, leading to the mutex going into an undefined state. **NO. Code shows blockProcess called before unblockProcess. Note: It is TRUE though that unlocking a mutex before it is locked can put it into an undefined state in POSIX systems.**
    b. There is a possibility that blockProcess() in the parent are executed before wait() in the child, leading to incorrect execution. **NO. This is the intended behavior – Parent calls blockProcess, and when child calls wait(), it is blocked until the parent calls unblockProcess.**
    c. **+There is a possibility that wait() in the child is executed before blockProcess() in the parent, leading to incorrect execution. YES. Not the intended answer but is a superset that can include the actual answer d. So this is accepted.**
    d. **+There's a possibility of wait() in the child executing before blockProcess() in the parent, leading to deadlock.**
    e. None of the above options a. to d. are correct.

12. Which ONE of the following options can effectively fix the code in Question 11?

    a. +

```
// Binary semaphore sema initialized to 0
     bsem_t sema = 0;

     void wait()
     {
           sem_pend(&sema);
     }

     void blockProcess()
     {
          // Nothing
     }

     void unblockProcess()
     {
            sem_post(&sema);
     }
```

YES. The wait will block until unblockProcess is called. If unblockProcess is called first, then wait won't block at all, which is the intended behavior.

b.
```
   // Counting semaphore sema initialized to 1
        csem_t sema = 1;
        void wait()
        {
             sem_pend(&sema);
        }

        void blockProcess()
        {
             // Nothing
        }

        void unblockProcess()
        {
            sem_post(&sema);
        }
```

NO. If unblockProcess is called more than once before a wait, wait will fail to block multiple times, which in this context is unlikely to be correct.

c.
```
       sleepvar c; // Sleep&Wake variable c.

        void wait()
        {
           sleep(&c);
        }

        void blockProcess()
        {
            //Nothing
        }

        void unblockProcess()
        {
              wake(&c);
        }
```

NO. If unblockProcess is called first, the wake will be lost.

d.
```
        condvar c;
        mutex m;

        void wait();
        {
            mutex_lock(&m);
            wait(&c);
            mutex_unlock(&m);
        }

         void blockProcess()
        {
```

```
    mutex_lock(&m);
}

void unblockProcess()
{
   signal(&c);
   mutex_unlock(&m);
}
```

NO. The mutex lock in wait can deadlock with the mutex lock in blockProcess.

e. None of the options above a. to d. are correct.