Roll No.: \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

Amrita Vishwa Vidyapeetham

Amrita School of Engineering, Coimbatore

B.Tech. Second Assessment Examinations – January 2020

Fourth Semester

Computer Science and Engineering

15CSE213 Operating Systems

Time: Two hours Maximum: 50 Marks

|  |  |  |
| --- | --- | --- |
| **CO** | **Course Outcomes** | |
| CO01 | Understand the architecture and functionalities of modern OS and virtual machines | L2 |
| CO02 | Understand and apply the algorithms for resource management and scheduling | L3 |
| CO03 | Apply semaphores and monitors for classical and real world synchronization scenarios | L4 |
| CO04 | Ability to engage in independent learning as a team to study characteristic features of modern operating systems | L4 |

**Answer all questions**

|  |  |  |
| --- | --- | --- |
|  | **Part A (4 x 5=20 marks)** | |
| 1. | a. Why is switching threads less costly than switching processes? [2]  Less state needs to be saved and restored. Furthermore, switching between threads benefits from caching; whereas, switching between processes invalidates the cache and TLB  b. How are multiple threads managed? Describe what happens in the user space,  and kernel space.  Answer  [2]. | [CO02] |
| 2. | Consider three processes, all arriving at time zero, with total execution time of 10, 20 and 30 units, respectively. Each process spends the first 20% of execution time doing I/O, the next 70% of time doing computation, and the last 10% of time doing I/O again. The operating system uses a shortest remaining compute time first scheduling algorithm and schedules a new process either when the running process gets blocked on I/O or when the running process finishes its compute burst. Assume that all I/O operations can be overlapped as much as possible. For what percentage of ti me does the CPU remain idle? [4]  1.Let three processes be p0, p1 and p2. Their execution time is 10, 20 and 30 respectively. p0 spends first 2 time units in I/O, 7 units of CPU time and finally 1 unit in I/O. p1 spends first 4 units in I/O, 14 units of CPU time and finally 2 units in I/O. p2 spends first 6 units in I/O, 21 units of CPU time and finally 3 units in I/O.  idle p0 p1 p2 idle  0 2 9 23 44 47  Total time spent = 47 Idle time = 2 + 3 = 5 Percentage of idle time = (5/47)\*100 = 10.6 % | [CO02] |
| 3. | a. Mutual exclusion is achieved in TestandSet( ) function only if the initial value of lock variable is False. Justify your answer. [3]  b.What happens if the initial value of lock variable is True? [1]  Ans:  Definition for TestandSet()  booleanTestAndSet(boolean \*target) {  booleanrv = \*target;  \*target = TRUE;  returnrv;  }  Mutual exclusion is achieved as:  do {  while(TestAndSetLock(&lock))  ; // do nothing  // critical section  lock = FALSE;  // remainder section  } while(TRUE);  i) If lock = false initially, Consider two processes P1 and P2 trying to enter CS.  If P1 is going for execution, it gets value false in rv, it sets lock as True, and return false. Due to the false, condition is failed; hence P1 goes for executing critical section.  Now suppose while P1 is in critical section, if P2 goes for execution, its gets Lock value true in rv, assigning true in lock and return true, condition is true, it will wait there only.  Hence Mutual exclusion preserved.  ii) If lock variable is initially true then it will never go forward. It blocks all the two processes there only. | [CO03] |
| 4. | a.What is a fundamental difference between a mutex and a semaphore? How does this enable bounded buffer exclusion? [3]  Answer: Semaphores enable counting with mutual exclusion. After a certain number of counts, then we can prevent further access. For example, for a bounded buffer, we can ensure mutual exclusion and prevent overflow and access to empty buffers   1. Are there priorities in counting semaphores? [1]   Ans: No | [CO03] |
| 5. | a. List the scheduling mechanisms supported by the operating system considered  for case study. [3] b. Scheduler Activations attempts to provide the best of both kernel and user-level threads, while avoiding the disadvantages of both. How? [1]   * Each application is provided with a virtual multiprocessor. * Each application has its own thread scheduler, and decides which threads should run on physical processors that it is assigned by the kernel. * The kernel is told how many threads an application would like to run so it can try to allocate that many physical processors for it. * The application needs is told when the kernel gives (and takes away) physical processors from it.is done by having the kernel do an up-call to the application when such events need to take place.    A scheduler activation is the kernel mechanism that makes an up-call to the applications thread scheduler when events (processor allocations and deallocations) need to take place. | [CO04] |
| **Part B (6 x 5=30 marks)** | | |
| 6. | a. FastTree wants to implement a web-server for video streaming using multithreading, where each thread serves one incoming request by loading a video file from the disk. Assume the operating system only provides the normal blocking read system call for disk reads. Do you think user-level threads or kernel-level threads are being used? Why? [3]  Ans: Kernel-level threads. Because each thread will make blocking I/O calls. With kernel-level thread, one thread won’t block others. But if user-level thread is used, then one thread will block all other threads.  b. FastTree wants to include social networking as well. Implement a webserver to serve each user’s “Home” page (the first page you see after you log in). This time web-server needs to perform many tasks: load the news feeds from each of your friends, load the advertisement, check for new messages, etc. Implement the webserver by using multithreading, and have one thread to perform each of the tasks, and later these threads will cooperate with each other to collectively construct the “Home” page. For performance reasons, FastTree makes sure that all the data these threads need is already cached in the memory (so they don’t have to perform any disk I/O). Do you think user-level threads or kernel-level threads should be used? Why? [3]  Ans: User-level thread. Here since the concern of user-level thread, namely “one thread can block all other threads within the same process”, no longer exists (as threads won’t make blocking I/O calls), so we can use user-level thread for its efficiency. This is in particular beneficial since these threads needs to communicate frequently with each other. If kernel-thread is used, everytime such communication needs to go through the kernel, which is more expensive. | [CO02] |
| 7. | Assume that the jobs arrive at different times as follows:   |  |  |  |  | | --- | --- | --- | --- | | **Process** | **Burst Time** | **Priority** | **Arrival Time** | | P1 | 8 | 4 | 0 | | P2 | 6 | 1 | 2 | | P3 | 1 | 2 | 2 | | P4 | 9 | 2 | 1 | | P5 | 3 | 3 | 3 |  1. Calculate the turnaround and wait times for the following algorithms: [4]    1. Preemptive Priority Scheduling    2. Round Robin (assume a quantum of 1ms) 2. Which of the two algorithms has the shortest wait time? [1] 3. Which has the fastest average turnaround time? [1]     Ans:  file:///C:/Users/bindukr/AppData/Local/Temp/IMG_20200206_110005.jpgfile:///C:/Users/bindukr/AppData/Local/Temp/IMG_20200206_110129.jpg | [CO02] |
| 8. | Two processes, P1 and P2, need to access a critical section of code. Consider the following synchronization construct used by the processes.  ***wants1*** and ***wants2*** are shared variables, which are initialized to ***false***. Which one of the following statements is TRUE about the above construct? Justify your answer. [Marks awarded only for justification] [6]   1. Mutual exclusion in achieved 2. Progress is achieved 3. Bounded Wait is achieved 4. Deadlock is prevented  |  |  | | --- | --- | | /\* P1 \*/  while (true) {  wants1 = true;  while (wants2 == true);  /\* Critical Section \*/  wants1=false;  }  /\* Remainder section \*/ | /\* P2 \*/  while (true) {  wants2 = true;  while (wants1==true);  /\* Critical Section \*/  wants2 = false;  }  /\* Remainder section \*/ |   i)When P1 initially enters, it makes the wants1 to true, and enters CS as wants2 is initially true. Now if P2 enters, it makes wants2 to true, and enters an infinite loop in the while condition as while (wants1==true); is true. Therefore Mutual exclusion is satisfied  ii) As wants2 is initially false, P1 can enter CS as many times as it wants. Therefore progress is also achieved.  iii) When P1 initially enters, it makes the wants1 to true, and enters CS as wants2 is initially true. Now if P2 enters, it makes wants2 to true, and enters an infinite loop in the while condition as while (wants1==true); is true. If P1 completes CS and exits and tries to enter CS again, it cannot enter as P2 had made wants2 to true. Here bounded waiting condition is also satisfied as there is a bound on the number of process which gets access to critical section after a process request access to it.  iv) When P1 initially enters, it makes the wants1 to true. At this point, if a context switch happens, P2 enters and changes wants2 to true. P2 cannot enter CS as the while condition is true, and if again a context switch happens, P1 also cannot enter CS as while condition is true. So both process p1 and p2 enter in while loop and waiting for each other to finish. This while loop run indefinitely which leads to deadlock. | [CO03] |
| 9. | Ms Iodine has asked students to help her out with the chemical reaction to form water, which she does not seem to be able to get right due to synchronization problems. The trick is to get two H atoms and one O atom all together at the same time. The atoms are threads. Each H atom invokes a procedure hReady when it is ready to react, and each O atom invokes a procedure oReady when it is ready. The procedures must delay until there are at least two H atoms and one O atom present, and then one of the threads must call the procedure makeWater (which just prints out a debug message that water was made). After the makeWater call, two instances of hReady and one instance of oReady should return. Studens may assume that the semaphore implementation enforces FIFO order for wakeups—the thread waiting longest in P() is always the next thread woken up by a call to V().  Answer the following questions:   1. Is the proposed code for hReady() and oReady() safe or dangerous? If dangerous write the modified version of the code. Your solution should avoid starvation and busy-waiting. [3]     Semaphore h\_wait = 0;  oReady(){  P(o\_wait);  makeWater();  V(h\_wait);  V(h\_wait);  return;  }  Semaphore o\_wait = 0;  int count = 0;  hReady(){  count++;  if(count %2 == 1)  { P(h\_wait); }  else  { V(o\_wait);  P(h\_wait); }  return; }  Ans: This solution is dangerous. Threads calling hReady() access shared data without holding a lock! For example, you could have N threads increment count, and because they do so without a lock, the result could be 1 instead of N -- in other words, no water would be made regardless of how many H’s arrived.  The solution is to put a lock acquire before the first line in hReady, and release before the first P(h\_wait) and after the second P(h\_wait). Some put the lock acquire after the increment, and that simply doesn’t work!   1. Another proposed solution to H2O problem. Is this solution safe or dangerous? Justify your answer. [3]   Semaphore h\_wait = 0;  oReady(){  P(o\_wait);  P(o\_wait);  makeWater();  V(h\_wait);  V(h\_wait);  return; }  Semaphore o\_wait = 0;  hReady(){  V(o\_wait)  P(h\_wait)  return; }  Answer: This is dangerous, since it may lead to starvation. If two H’s arrive, then the value of the o\_wait semaphore will be 2. If two O’s arrive, then they can each decrement o\_wait, before either can decrement it twice. So no water is made, even though enough atoms have arrived. The fix is to put a lock acquire before the first line in oReady, and a lock release after the two V(h\_wait)’s. This way, only one oxygen looks for waiting H’s at a time -- if there aren’t enough H’s for the first oxygen, there won’t be enough for any of the later oxygens either. |  |
| 10. | a. Performance of broadcasting to a condition variable decreases as the number of  waiting threads increases. True or False? Justify. [1]  True; with more waiting threads, more threads are awoken which must be scheduled (all incurring a context-switch). | [CO03] |
|  | b. Passengers come to a bus stop and wait for a bus. When the bus arrives, all the waiting passengers invoke boardBus, but anyone who arrives while the bus is boarding has to wait for the next bus. The capacity of the bus is forty people; if there are more than forty people waiting, some will have to wait for the next bus. When all the waiting passengers have boarded, the bus can invoke depart. If the bus arrives when there are no passengers, it should depart immediately. Write synchronization code that enforces all of these constraints with pthread synchronization primitives. [5]    Solution 1 |  |

****

**Solution 2**

****