University of Newcastle

School of Electrical Engineering and Computer Science

**COMP2240 - Operating Systems**

**Workshop 4 - Solution**

**Topics: Real-time and Multiprocessor Scheduling**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1.** | Consider a set of five aperiodic tasks with the execution profiles of Table 1. Develop scheduling diagrams for Earliest Deadline, Earliest Deadline with unforced idle time and FCFS algorithms for this set of tasks.   |  |  |  |  | | --- | --- | --- | --- | | **Process** | **Arrival Time** | **Execution Time** | **Starting Deadline** | | A | 10 | 20 | 100 | | B | 20 | 20 | 20 | | C | 40 | 20 | 60 | | D | 50 | 20 | 80 | | E | 60 | 20 | 70 |   Table 1: Execution profile for Problem 1  **Answer:** |
| **2.** | **Least laxity first (LLF)** is a real-time scheduling algorithm for periodic tasks. Slack time, or laxity, is the amount of time between when a task would complete if it started now and its next deadline. This is the size of the available scheduling window. Laxity can be expressed as  *Laxity = (deadline time) – (current time) – (processor time needed)*  LLF selects the task with the minimum laxity to execute next. If two or more tasks have the same minimum laxity value, they are serviced on a FCFS basis.   1. Suppose a task currently has a laxity of *t*. By how long may the scheduler delay starting this task and still meet its deadline? 2. Suppose a task currently has a laxity of *0*. What does this mean? 3. What does it mean if a task has negative laxity? 4. Consider a set of three periodic tasks with the execution profiles of Table . Develop scheduling diagrams for this set of tasks that compare rate monotonic, earliest-deadline first, and LLF. Assume preemption may occur at 5-ms intervals. Comment on the results.  |  |  |  | | --- | --- | --- | | **Task** | **Period** | **Execution Time** | | A | 6 | 2 | | B | 8 | 2 | | C | 12 | 3 |   Table 2: Light load execution profile  **Answer:**   1. The task may be delayed up to an interval of *t* and still meet its deadline. 2. A laxity of 0 means that the task must be executed now or will fail to meet its deadline. 3. A task with negative laxity cannot meet its deadline. 4. The task set has a total load of 83% (load=20/24). The resulting schedules differ in the number of (potentially costly) context switches: 13, 11, and 13, respectively. Any timeline repeats itself every 24-time units. Although the total load (83%) is higher than the RM limit for this case (78%), RM can schedule the set satisfactorily. |
| **3.** | This problem demonstrates that although Equation for rate monotonic scheduling is a sufficient condition for successful scheduling, it is not a necessary condition (i.e., sometimes successful scheduling is possible even if Equation is not satisfied).   1. Consider a task set with the following independent periodic tasks:   **Task P1**:*C1 = 20*; *T1 = 100*  **Task P2**:*C2 = 30*; *T2 = 145*  Can these tasks be successfully scheduled using rate monotonic scheduling?   1. Now add the following task to the set:   **Task P3**:*C3 = 68; T3 = 150*  Is Equation satisfied?   1. Suppose that the first instance of the preceding three tasks arrives at time. Assume that the first deadline for each task is the following:   *D1 = 100; D2 = 145; D3 = 150*  Using rate monotonic scheduling, will all three deadlines be met? What about deadlines for future repetitions of each task?  **Answer:**   1. The total utilization of P1 and P2 is (0.41, which is less than the bound 0.828, given for two tasks by Equation 10.2. Therefore, these two tasks are schedulable. 2. The utilization of all the tasks is ( 0.86, which exceeds the bound of 0.779. 3. Observe that P1 and P2 must execute at least once before P3 can begin executing. Therefore, the completion time of the first instance of P3 can be no less than 20 + 30 + 68 = 118. However, P1 is initiated one additional time in the interval (0, 118). Therefore, P3 does not complete its first execution until 118 + 20 = 138. This is within the deadline for P3. By continuing this reasoning, we can see that all deadlines of all three tasks can be met. |
| **4.** | Consider two processes, *P1* and *P2*, where *T1 = 50*, *C1 = 25*, *T2 = 75*, and *C2 = 30*.   1. Can these two processes be scheduled using rate-monotonic scheduling? Illustrate your answer using a Gantt chart. 2. Illustrate the scheduling of these two processes using earliest deadline- first (EDF) scheduling.   **Answer:**   1. The utilisation will be:   The sufficient condition for 2 processes, under which we can conclude the system is schedulable is:  Since 0.90 > 0.828, the system is not schedulable using RMS algorithm.  Consider when *P*1 is assigned a higher priority than *P*2 with the rate monotonic scheduler. *P*2 is not scheduled early enough to meet its deadline. On the other hand, if *P*1 is assigned a lower priority than *P*2, then *P*1 does not meet its deadline since it will not be scheduled in time.    Figure 1: Given tasks are not schedulable  b) The EDF scheduling was able to meet the deadline of both processes as shown in the Gantt chart above. |
| **5.** | Suppose that an application has three threads *T1, T2* and *T3* having decreasing priority. Give a scenario that may cause unbounded priority inversion.  **Answer:**  Given that the application has three threads *T1, T2*, and *T3*, such that thread *T1* has high priority, thread *T2* has medium priority, and thread *T3* has low priority, the following scenario gives a sequence for unbounded priority inversion:  Let us assume that initially Thread *T1* and Thread *T2* are sleeping or blocked  *t1* : Thread *T3* runs and enters a critical section.  *t2:* Thread *T2* starts running, preempting Thread *T3* because Thread *T2* has a higher priority. So, Thread *T3* continues to own the critical section.  *t3:* Thread *T1* starts running, preempting Thread 2. Thread *T1* tries to enter the critical section that Thread *T3* owns, but because it is owned by another thread, Thread *T1* blocks, waiting for the critical section.  *t4:* Thread *T2* starts running because it has a higher priority than Thread *T3* and Thread *T1* is not running. Thread 3 never releases the critical section that Thread *T1* is waiting for because Thread *T2* continues to run.  Therefore, the highest-priority thread in the system, Thread *T1*, becomes blocked waiting for lower-priority threads to run. |
|  |  |

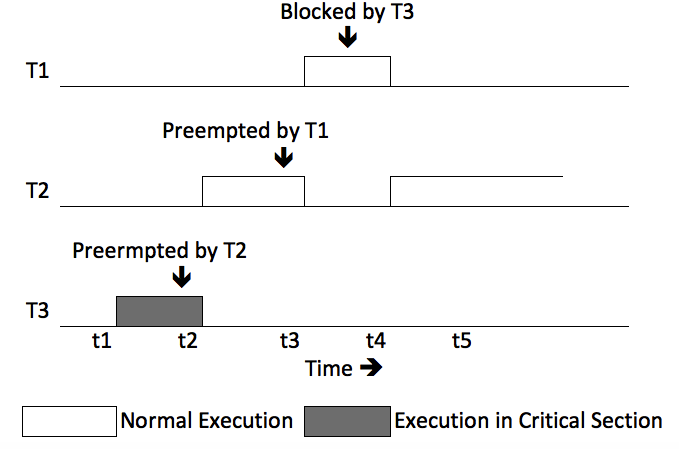


Figure: An example of Unbounded Priority Inversion

**Supplementary problems:**

|  |  |
| --- | --- |
| **S1.** | What happens if three CPUs in a multiprocessor attempt to access exactly the same word of memory at exactly the same instant?  **Answer:**  Depending on how CPUs are connected to memory, one of them gets through first, for example, seizes the bus first. It completes its memory operation, then another one happens, etc. It is not predictable which one goes first, but if the system has been designed for sequential consistency, it should not matter. |
| **S2.** | Explain why interrupt and dispatch latency times must be bounded in a hard real-time system.  **Answer:**  The tasks that follow an interrupt request are: save the program counter, determine the type of interrupt, save the current process state, and then invoke the appropriate interrupt service routine. Dispatch latency is the cost associated with stopping one process and starting another. Both interrupt and dispatch latency needs to be minimized in order to ensure that real-time tasks receive immediate attention. Furthermore, sometimes interrupts are disabled when kernel data structures are being modified, so the interrupt does not get serviced immediately. For hard real-time systems, the time-period for which interrupts are disabled must be bounded in order to guarantee the desired quality of service. |
| **S3.** | The Linux scheduler implements “soft” real-time scheduling. What features necessary for certain real-time programming tasks are missing? How might they be added to the kernel? What are the costs (downsides) of such features?  **Answer:**  Linux’s “soft” real-time scheduling provides ordering guarantees concerning the priorities of runnable processes: real-time processes will always be given a higher priority by the scheduler than normal time-sharing processes, and a real-time process will never be interrupted by another process with a lower real-time priority.  However, the Linux kernel does not support “hard” real-time functionality. That is, when a process is executing a kernel service routine, that routine will always execute to completion unless it yields control back to the scheduler either explicitly or implicitly (by waiting for some asynchronous event). There is no support for preemptive scheduling of kernel-mode processes. As a result, any kernel system call that runs for a significant amount of time without rescheduling will block execution of any real-time processes. Many real-time applications require such hard real-time scheduling. In particular, they often require guaranteed worst-case response times to external events. To achieve these guarantees, and to give user-mode real time processes a true higher priority than kernel-mode lower-priority processes, it is necessary to find a way to avoid having to wait for low-priority kernel calls to complete before scheduling a real-time process.  For example, if a device driver generates an interrupt that wakes up a high-priority real-time process, then the kernel needs to be able to schedule that process as soon as possible, even if some other process is already executing in kernel mode. Such preemptive rescheduling of kernel-mode routines comes at a cost. If the kernel cannot rely on non-preemption to ensure atomic updates of shared data structures, then reads of or updates to those structures must be protected by some other, finer-granularity locking mechanism. This fine-grained locking of kernel resources is the main requirement for provision of tight scheduling guarantees. Manyother kernel features could be added to support real-time programming. Deadline-based scheduling could be achieved by making modifications to the scheduler. |
| **S4.** | Some descriptions of UNIX would indicate that it is unsuitable for real-time applications because a process executing in kernel mode may not be preempted. Elaborate. Then consider and discuss if it is possible that UNIX may be modified to accommodate real-time applications at all?  **Answer:**  Preemption of a kernel process is not allowed in the UNIX operating system. A high-priority process cannot preempt a low priority process that is executing a system call in the kernel mode.  In real-time applications, for example factory automation, detection systems, air traffic control systems, and process control applications, it is important that a critical event be attended to immediately. The scheduling algorithm in UNIX does not provide a means of preempting a running kernel process or supereceding other kernel processes and placing this event to the head of the run queue. It is also important that a real-time process once started not be interrupted until it completes the operation.  It might be possible to handle real-time applications by enhancing the scheduling algorithm. The user should be able to define one or two processes as being real-time, making them always able to operate at the highest priority level and able to preempt any other process at any time except during critical system functions. (Note: the *nice* system call is not capable of achieving this.) It is also very important that any real-time process is locked into memory so that no page fault delays occur and so that it can be scheduled very rapidly. AT&T UNIX System V provides a *plock*  system call that provides for locking a user process in memory. |
| **S5.** | Discuss ways in which the priority inversion problem could be addressed in a real-time system. Also discuss whether the solutions could be implemented within the context of a proportional share scheduler.  **Answer**:  The priority inversion problem could be addressed by temporarily changing the priorities of the processes involved. Processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. When they are finished, their priority reverts to its original value. This solution can be easily implemented within a proportional share scheduler; the shares of the high-priority processes are simply transferred to the lower-priority process for the duration when it is accessing the resources. |