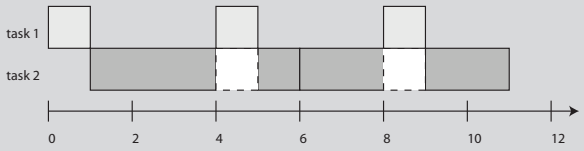
12 Scheduling — Exercises

1. This problem studies fixed-priority scheduling. Consider two tasks to be executed periodically on a single processor, where task 1 has period $p_1 = 4$ and task 2 has period $p_2 = 6$. p.314

   (a) Let the execution time of task 1 be $e_1 = 1$. Find the maximum value for the execution time e_2 of task 2 such that the RM schedule is feasible.

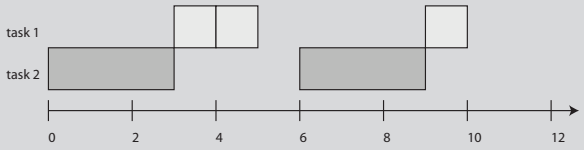
   Solution: The largest execution time for task 2 is $e_2 = 4$. The following figure shows the resulting schedule:

   The schedule repeats every 12 time units.

   (b) Again let the execution time of task 1 be $e_1 = 1$. Let non-RMS be a fixed-priority schedule that is not an RM schedule. Find the maximum value for the execution time e_2 of task 2 such that non-RMS is feasible.

   Solution: The largest execution time for task 2 is $e_2 = 3$. The following figure shows the resulting schedule:

   The schedule repeats every 12 time units.

   (c) For both your solutions to (a) and (b) above, find the processor utilization. Which is better?

Solution: From the figures above, we see that the RM schedule results in the machine being idle for 1 out of 12 time units, so the utilization is 11/12. The non-RM schedule results in the machine being idle for 3 out of 12 time units, so the utilization is 9/12 or 3/4. The RM schedule is better.

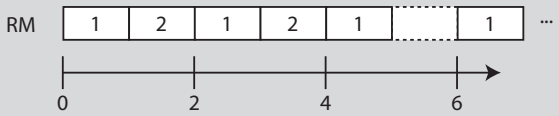
(d) For RM scheduling, are there any values for e_1 and e_2 that yield 100% utilization? If so, give an example.

Solution: Yes. For example, $e_1 = 4$ and $e_2 = 0$.

3. This problem compares RM and EDF schedules. Consider two tasks with periods $p_1 = 2$ and $p_2 = 3$ and execution times $e_1 = e_2 = 1$. Assume that the deadline for each execution is the end of the period.

   (a) Give the RM schedule for this task set and find the processor utilization. How does this utilization compare to the Liu and Layland utilization bound of (12.2)?

Solution: The RM schedule is shown below:



The utilization is given by

$$U = 1 - 1/6 \approx 833\%$$

The utilization bound if $n = 2$ is $n(2^{1/n} - 1) \approx 0.828$. Thus, utilization is larger than the utilization bound, so we have no assurance that the RM schedule is feasible.

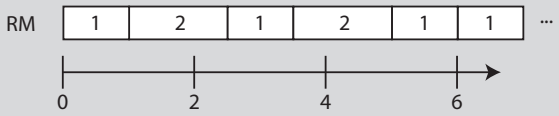
   (b) Show that any increase in e_1 or e_2 makes the RM schedule infeasible. If you hold $e_1 = e_2 = 1$ and $p_2 = 3$ constant, is it possible to reduce p_1 below 2 and still get a feasible schedule? By how much? If you hold $e_1 = e_2 = 1$ and $p_1 = 2$ constant, is it possible to reduce p_2 below 3 and still get a feasible schedule? By how much?

Solution: In the first three time units, the RM schedule must execute task 1 twice, because under the RM principle, it has highest priority and it has become enabled twice in this time period. With $e_1 = 1$, this leaves exactly one time unit to execute task 2 in its first period. Thus, any increase in e_2 will result in task 2 missing its deadline at time 3. Any increase in e_1 will leave less than one time unit for task 2 in its first period, resulting again in a missed deadline.

Holding e_1, e_2, and p_2 constant, we can reduce p_1 to 1.5 and still get a feasible schedule. Holding e_1, e_2, and p_1 constant, we can reduce p_2 to 2 and still get a feasible schedule. In both cases, no further reduction is possible because at this point we have 100% utilization.

   (c) Increase the execution time of task 2 to be $e_2 = 1.5$, and give an EDF schedule. Is it feasible? What is the processor utilization?

Solution: The EDF schedule is:

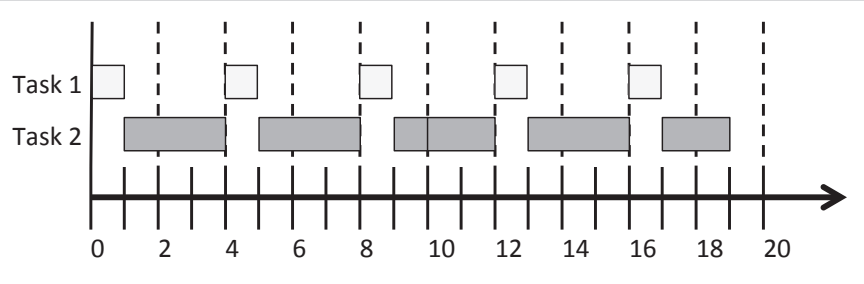


The schedule is feasible and the utilization is 100%.

4. This problem, formulated by Hokeun Kim, also compares RM and EDF schedules. Consider two tasks to be executed periodically on a single processor, where task 1 has period $p_1 = 4$ and task 2 has period $p_2 = 10$. Assume task 1 has execution time $e_1 = 1$, and task 2 has execution time $e_2 = 7$.

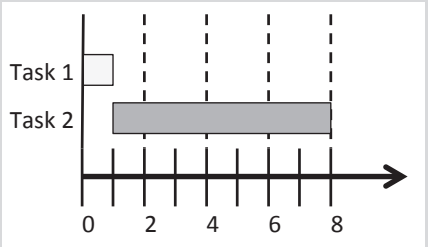
   (a) Sketch a rate-monotonic schedule (for 20 time units, the least common multiple of 4 and 10). Is the schedule feasible?

Solution: The rate monotonic schedule is feasible. The figure below shows the schedule.



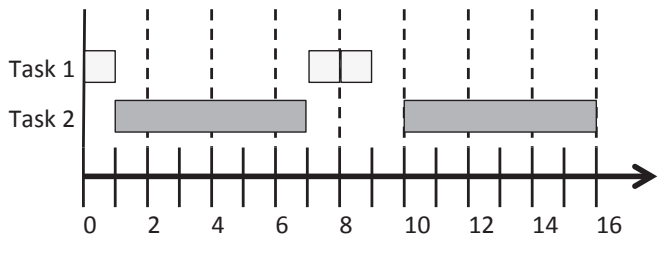
   (b) Now suppose task 1 and 2 contend for a mutex lock, assuming that the lock is acquired at the beginning of each execution and released at the end of each execution. Also, suppose that acquiring or releasing locks takes zero time and the priority inheritance protocol is used. Is the rate-monotonic schedule feasible?

Solution: No. Task 1 misses its deadline at time point 8 (the first deadline, which is met, is at time 4; the second deadline, which is not met, is at time 8).

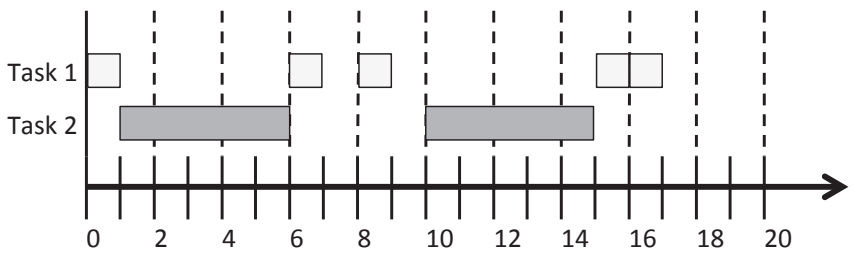


   (c) Assume still that tasks 1 and 2 contend for a mutex lock, as in part (b). Suppose that task 2 is running an **anytime algorithm**, which is an algorithm that can be terminated early and still deliver useful results. For example, it might be an image processing algorithm that will deliver a lower quality image when terminated early. Find the maximum value for the execution time e_2 of task 2 such that the rate-monotonic schedule is feasible. Construct the resulting schedule, with the reduced execution time for task 2, and sketch the schedule for 20 time units. You may assume that execution times are always positive integers.

Solution: If we reduce the execution time of task 2 to $e_2 = 6$, task 1 still misses its deadline at time point 16, as shown below:



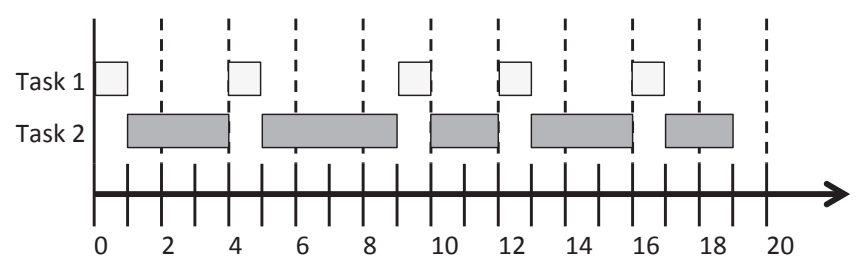
Reducing further to $e_2 = 5$ makes the schedule feasible, as shown below:



Therefore, the maximum value for the execution time of task 2 that makes the rate monotonic schedule feasible is $e_2 = 5$.

(d) For the original problem, where $e_1 = 1$ and $e_2 = 7$, and there is no mutex lock, sketch an EDF schedule for 20 time units. For tie-breaking among task executions with the same deadline, assume the execution of task 1 has higher priority than the execution of task 2. Is the schedule feasible?

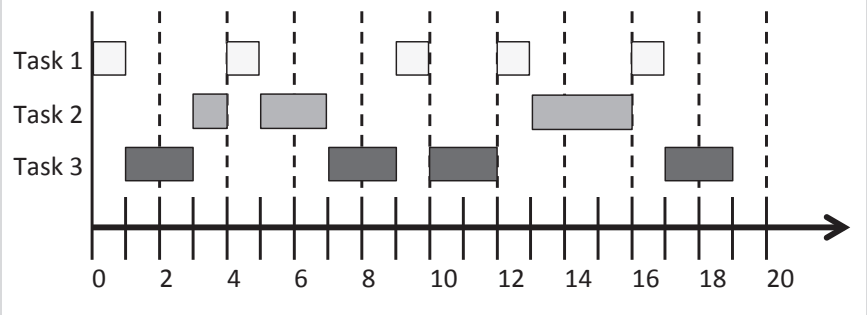
Solution: The EDF schedule is feasible. The figure below shows the schedule.



(e) Now consider adding a third task, task 3, which has period $p_3 = 5$ and execution time $e_3 = 2$. In addition, assume as in part (c) that we can adjust execution time of task 2.

Find the maximum value for the execution time e_2 of task 2 such that the EDF schedule is feasible and sketch the schedule for 20 time units. Again, you may assume that the execution times are always positive integers. For tie-breaking among task executions with the same deadline, assume task i has higher priority than task j if $i < j$.)

Solution: The total execution times of task 1 and 2 within the 20 time units window are 5 and 8, respectively. Thus, the total execution time of task 3 should be less than $20 - (5 + 8) = 7$ within the 20 time units window. Therefore, the EDF schedule becomes feasible when $e_2 = 3$. The figure below shows the schedule.


Task 1
Task 2
Task 3
0
2
4
6
8
10
12
14
16
18
20

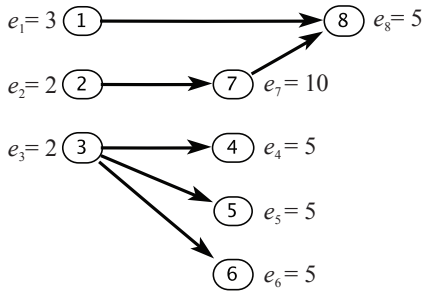
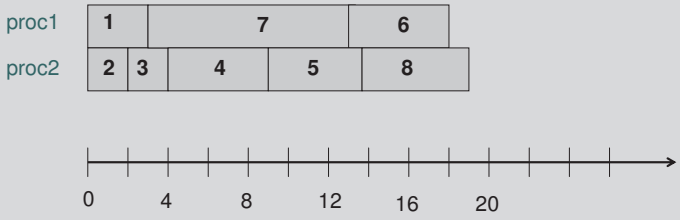


Figure 12.1: Precedence Graph for Exercise 6.

6. This problem studies scheduling anomalies. Consider the task precedence graph depicted in Figure 12.1 with eight tasks. In the figure, e_i denotes the execution time of task i. Assume task i has higher priority than task j if $i < j$. There is no preemption. The tasks must be scheduled respecting all precedence constraints and priorities. We assume that all tasks arrive at time $t = 0$.

   (a) Consider scheduling these tasks on two processors. Draw the schedule for these tasks and report the makespan.

Solution: The schedule is as shown below with a makespan of 19 units:

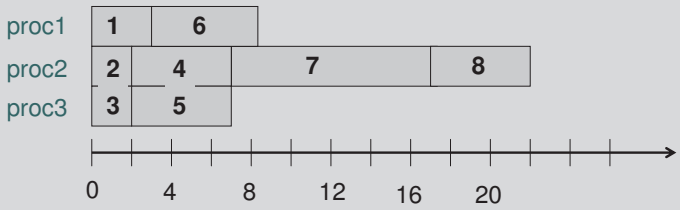


Schedule length = 19

   (b) Now consider scheduling these tasks on three processors. Draw the schedule for these tasks and report the makespan. Is the makespan bigger or smaller than that in part (a) above?

Solution: The schedule for 3 processors is as shown below with the makespan increasing to 22 units:

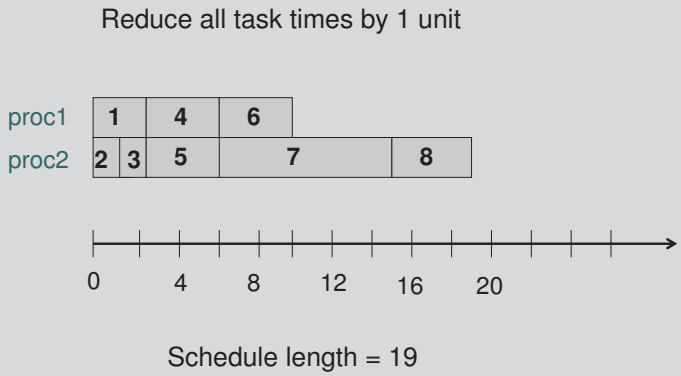
Add a processor



Schedule length = 22

(c) Now consider the case when the execution time of each task is reduced by 1 time unit. Consider scheduling these tasks on two processors. Draw the schedule for these tasks and report the makespan. Is the makespan bigger or smaller than that in part (a) above?

Solution: If each task time is reduced by 1 unit, the schedule is as shown with the makespan staying at 19 units below:



7. This problem studies the interaction between real-time scheduling and mutual exclusion, and was formulated by Kevin Weekly.

   Consider the following excerpt of code:

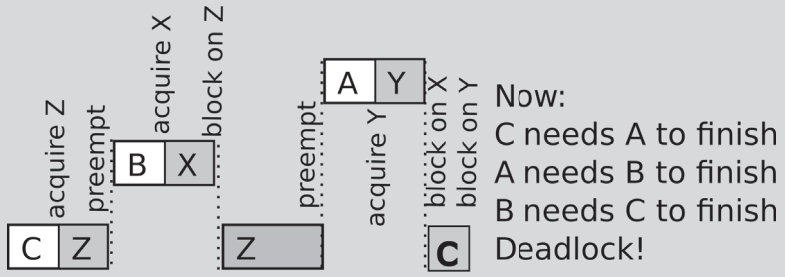
```
1  pthread_mutex_t X; // Resource X: Radio communication
2  pthread_mutex_t Y; // Resource Y: LCD Screen
3  pthread_mutex_t Z; // Resource Z: External Memory (slow)
4
5  void ISR_A() { // Safety sensor Interrupt Service Routine
6    pthread_mutex_lock(&Y);
7    pthread_mutex_lock(&X);
8    display_alert(); // Uses resource Y
9    send_radio_alert(); // Uses resource X
10   pthread_mutex_unlock(&X);
11   pthread_mutex_unlock(&Y);
12 }
13
14 void taskB() { // Status recorder task
15   while (1) {
16     static time_t starttime = time();
17     pthread_mutex_lock(&X);
18     pthread_mutex_lock(&Z);
19     stats_t stat = get_stats();
20     radio_report( stat ); // uses resource X
21     record_report( stat ); // uses resource Z
22     pthread_mutex_unlock(&Z);
23     pthread_mutex_unlock(&X);
24     sleep(100-(time()-starttime)); // schedule next excecution
25   }
26 }
27
28 void taskC() { // UI Updater task
29   while(1) {
30     pthread_mutex_lock(&Z);
31     pthread_mutex_lock(&Y);
32     read_log_and_display(); // uses resources Y and Z
33     pthread_mutex_unlock(&Y);
34     pthread_mutex_unlock(&Z);
35   }
36 }
```

   You may assume that the comments fully disclose the resource usage of the procedures. That is, if a comment says ”uses resource X”, then the relevant procedure uses only resource X. The scheduler running aboard the system is a priority-based preemptive scheduler, where taskB is higher priority than taskC. In this problem, ISR_A can be thought of as an asynchronous task with the highest priority.

   The intended behavior is for the system to send out a radio report every 100ms and for the UI to update constantly. Additionally, if there is a safety interrupt, a radio report is sent immediately and the UI alerts the user.

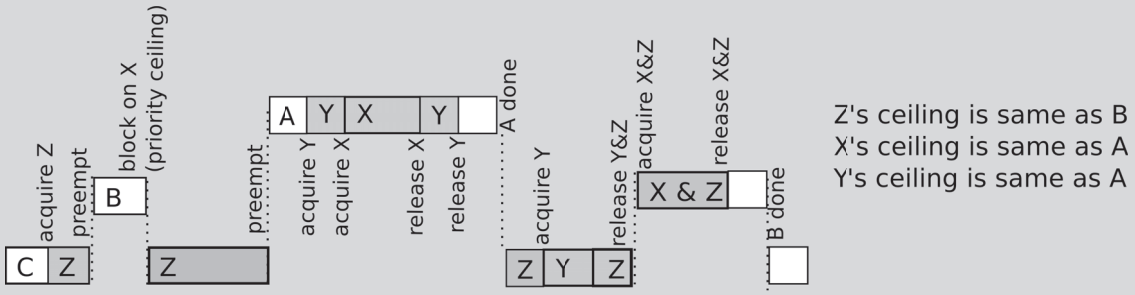
   (a) Occasionally, when there is a safety interrupt, the system completely stops working. In a scheduling diagram (like Figure 12.11 in the text), using the tasks {A,B,C}, and resources {X,Y,Z}, explain the cause of this behavior. Execution times do not have to be to scale in your diagram. Label your diagram clearly. You will be graded in part on the clarity of your answer, not just on its correctness.

Solution: The scheduling diagram is shown below:



(b) Using the priority ceiling protocol, show the scheduling diagram for the same sequence of events that you gave in part (a). Be sure to show all resource locks and unlocks until all tasks are finished or reached the end of an iteration. Does execution stop as before?

Solution: Execution does not stop, as illustrated with the following diagram:



(c) Without changing the scheduler, how could the code in taskB be reordered to fix the issue? Using an exhaustive search of all task/resource locking scenarios, prove that this system will not encounter deadlock. (Hint: There exists a proof enumerating 6 cases, based on reasoning that the 3 tasks each have 2 possible resources they could block on.)

Solution: The following could replace the body of the while loop in TaskB:

```
stats_t stat = get_stats();
pthread_mutex_lock(&X);
radio_report( stat ); // uses resource X
pthread_mutex_unlock(&X);
pthread_mutex_lock(&Z);
record_report( stat ); // uses resource Z
pthread_mutex_unlock(&Z);
```

1: If A blocks on X, it is dependent on B. While it has X, B will not block on any other resource and is guaranteed to release X.
2: If A blocks on Y, it is dependent on C. While holding Y, C already has both the locks it needs and will release both.
1+2: Thus, A will always finish.

3: If C blocks on Z, it depends on B. While it holds Z, B will not block any other resource and is guaranteed to release Z.
4: If C blocks on Y, it is dependent on A. We have shown that A always finishes.

3+4: Thus, C will always finish.

5: If B blocks on X, it is dependent on A finishing. We have shown that A always finishes.
6: If B blocks on Z, it depends on C. We have shown that C always finishes.
5+6: Thus, B always finishes.

1-6: Since, A, B, and C, always finish execution, this does not encounter deadlocks.
Note that the change of code allows us to guarantee that B releases it's locks. Otherwise, we would have a circular dependency and could not prove this.

Another solution is to swap the order that B requests its resources in (but still nested). The proof is similar to above, except line 3, in which you must show that B always finishes.

Name: _____ UCB Email: _____

Grants in the Tock OS. The Tock Operating Systems[1] ("TockOS") uses a mechanism called *grants* to avoid trade-offs between memory efficiency and concurrency. Recall that grants are physically allocated within a process's memory region, but processes are not allowed to access the grant region. Provide, in your own words, answers to the following questions. Make sure that your answers are sufficiently detailed so that someone who is unfamiliar with Tock would understand the essential ideas.

(a) What specific problem does Tock solve with grants?

Grants solve the problem of how to allocate dynamic memory for the kernel. They support dynamic resource needs (due to changing or multiple applications) without requiring dynamic allocation within the kernel itself.

(b) Why is this problem not usually an issue on more traditional computers?

One reason is because they have so much memory available. It's unlikely that they will run out of memory for use in the kernel.

Another reason is because if the kernel runs out of resources to perform an operation on traditional computers, it can inform the user, who can then make a decision about whether to kill a task and free up resources.

(c) Why is this problem quite important in embedded systems?

One reason is because they have so little memory available. It's quite possible that an attempt to allocate memory in the kernel will fail.

Another reason is that embedded systems run without user interaction for long durations (often their whole lifetime). They need to be able to robustly handle application needs.

Another reason is that it isn't possible to statically allocate all the memory you will possibly need. You will either end up with too little, or else waste a bunch.

(d) How does Tock prevent a process from accessing grant memory?

By using the Memory Protection Unit (MPU) to prevent read or write access.

[1] http://web.eecs.umich.edu/~prabal/pubs/papers/levy17tock.pdf