2. Suppose that two `int` global variables `a` and `b` are shared among several threads. Suppose that `lock_a` and `lock_b` are two mutex locks that guard access to `a` and `b`. Suppose you cannot assume that reads and writes of `int` global variables are atomic. Consider the following code:

```
1   int a, b;
2   pthread_mutex_t lock_a
3       = PTHREAD_MUTEX_INITIALIZER;
4   pthread_mutex_t lock_b
5       = PTHREAD_MUTEX_INITIALIZER;
6
7   void procedure1(int arg) {
8     pthread_mutex_lock(&lock_a);
9     if (a == arg) {
10       procedure2(arg);
11    }
12    pthread_mutex_unlock(&lock_a);
13  }
14
15  void procedure2(int arg) {
16    pthread_mutex_lock(&lock_b);
17    b = arg;
18    pthread_mutex_unlock(&lock_b);
19  }
```

Suppose that to ensure that deadlocks do not occur, the development team has agreed that `lock_b` should always be acquired before `lock_a` by any thread that acquires both locks. Note that the code listed above is not the only code in the program. Moreover, for performance reasons, the team insists that no lock be acquired unnecessarily. Consequently, it would not be acceptable to modify `procedure1` as follows:

```
1   void procedure1(int arg) {
2     pthread_mutex_lock(&lock_b);
3     pthread_mutex_lock(&lock_a);
4     if (a == arg) {
5        procedure2(arg);
6     }
7     pthread_mutex_unlock(&lock_a);
8     pthread_mutex_unlock(&lock_b);
9   }
```

A thread calling `procedure1` will acquire `lock_b` unnecessarily when `a` is not equal to `arg`. [1] Give a design for `procedure1` that minimizes unnecessary acquisitions of `lock_b`. Does your solution eliminate unnecessary acquisitions of `lock_b`? Is there any solution that does this?

Solution: Here is a candidate solution:

```
1   void procedure1(int arg) {
2     int result;
3     pthread_mutex_lock(&lock_a);
4     result = (a == arg);
5     pthread_mutex_unlock(&lock_a);
6
7     if (result) {
8       pthread_mutex_lock(&lock_b);
9       pthread_mutex_lock(&lock_a);
10      if (a == arg) {
```

[1] In some thread libraries, such code is actually incorrect, in that a thread will block trying to acquire a lock it already holds. But we assume for this problem that if a thread attempts to acquire a lock it already holds, then it is immediately granted the lock.

```
11          procedure2(arg);
12        }
13        pthread_mutex_unlock(&lock_a);
14        pthread_mutex_unlock(&lock_b);
15      }
16    }
```

This code avoids acquiring `lock_b` if `a != arg` at the time of the first test. Notice that the first test for `a == arg` needs to be performed while holding `lock_a` because reading `a` is not assured of being atomic. Moreover, there is no need to acquire `lock_b` at that point. The second test for `a == arg` is necessary because the value of `a` may have changed after the first test. Also notice that this solution may acquire `lock_b` unnecessarily. In particular, a thread could acquire `lock_b`, then get suspended, and while suspended, the value of `a` may change so that the second test for `a == arg` yields false.

3. The implementation of `get` in Figure 11.6 permits there to be more than one thread calling `get`.

   However, if we change the code on lines 30-32 to: `pthread_cond_wait`

```
1       if (size == 0) {
2         pthread_cond_wait(&sent, &mutex);
3       }
```

   then this code would only work if two conditions are satisfied:

   - `pthread_cond_wait` returns *only* if there is a matching call to `pthread_cond_signal`, and
   - there is only one consumer thread.

   Explain why the second condition is required.

Solution: Because of the first condition, we would expect that after line 2 above, it must be true that `size != 0`. However, this is not necessarily true if there are two or more consumer threads. Recall that `pthread_cond_wait` temporarily releases the lock on `mutex`, so it could be possible for line 1 above to return true in thread *A*, which will then block on line 2, and while it is blocked, another consumer thread *B* manages to consume the next message sent without executing `pthread_cond_wait`. This could happen if *B* happens to execute line 1 above after a producer has inserted one message into the queue, but before *A* has been scheduled to respond to the signal. When *A* responds to the signal, it will wake up at line 2 above and proceed to consume a message, but there is no message on the queue! Disaster.

5. An alternative form of message passing called rendezvous is similar to the producer/consumer pattern of Example 11.13, but it synchronizes the producer and consumer more tightly. In particular, in Example 11.13, the `send` procedure returns immediately, regardless of whether there is any consumer thread ready to receive the message. In a rendezvous-style communication, the `send` procedure will not return until a consumer thread has reached a corresponding call to `get`. Consequently, no buffering of the messages is needed. Construct implementations of `send` and `get` that implement such a rendezvous.

Solution:

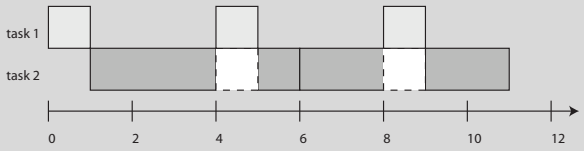
```
1  // Condition variables to signal ready and complete.
2  int send_ready = 0;
3  pthread_cond_t send_ready_c = PTHREAD_COND_INITIALIZER;
4  int receive_ready = 0;
5  pthread_cond_t receive_ready_c = PTHREAD_COND_INITIALIZER;
6  int complete = 1;
7  pthread_cond_t complete_c = PTHREAD_COND_INITIALIZER;
8
9  // Procedure to send a message.
10 void send(int message) {
11     pthread_mutex_lock(&mutex);
12     complete = 0;
13     // Wait until the receiver is ready.
14     while (receive_ready == 0) {
15         pthread_cond_wait(&receive_ready_c, &mutex);
16     }
17     buffer = message;
18     send_ready = 1;
19     pthread_cond_signal(&send_ready_c);
20     // Wait until communication completes.
21     while (complete == 0) {
22         pthread_cond_wait(&complete_c, &mutex);
23     }
24     pthread_mutex_unlock(&mutex);
25 }
26
27 // Procedure to get a message.
28 int get() {
29     int result;
30     pthread_mutex_lock(&mutex);
31     // Signal that the receiver is ready.
32     receive_ready = 1;
33     pthread_cond_signal(&receive_ready_c);
34     // Wait until the sender is ready.
35     while (send_ready == 0) {
36         pthread_cond_wait(&send_ready_c, &mutex);
37     }
38     receive_ready = 0;
39     result = buffer;
40     send_ready = 0;
41     complete = 1;
42     pthread_cond_signal(&complete_c);
43     pthread_mutex_unlock(&mutex);
44     return result;
45 }
```

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.

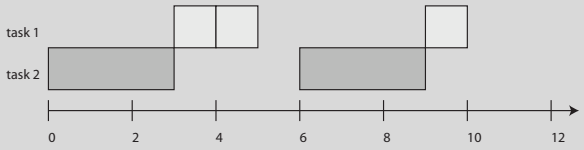
   Solution: The largest execution time for task 2 is $e_2 = 4$. The following figure shows the resulting schedule:

   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.

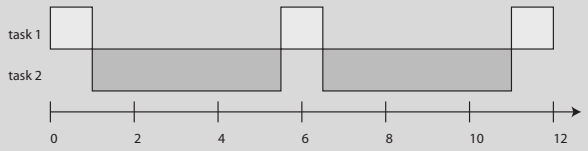
(d) For RM scheduling, are there any values for e_1 and e_2 that yield 100% utilization? If so, give an example.

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

2. This problem studies dynamic-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 $p_2 = 6$. Let the deadlines for each invocation of the tasks be the end of their period. That is, the first invocation of task 1 has deadline 4, the second invocation of task 1 has deadline 8, and so on. 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 EDF is feasible.

Solution: The largest execution time for task 2 is $e_2 = 4.5$. The following figure shows the resulting schedule:



The schedule repeats every 12 time units.

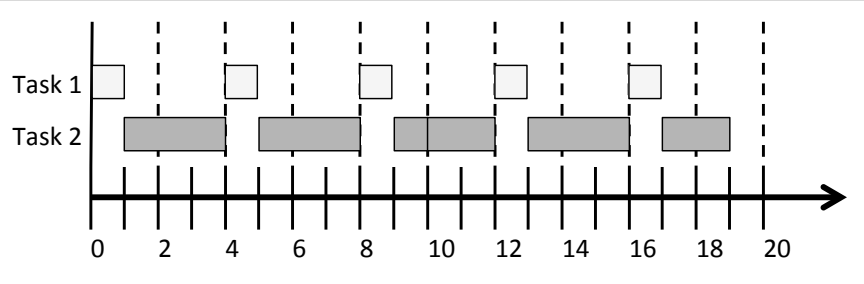
   (b) For the value of e_2 that you found in part (a), compare the EDF schedule against the RM schedule from Exercise 1 (a). Which schedule has less preemption? Which schedule has better utilization?

Solution: Comparing the schedule in (a) with the schedule in Exercise 1(a), we see that EDF has no preemption at all, while RM performs two preemptions every 12 time units. Moreover, EDF has 100% utilization, whereas RM has less.

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 $e_2 = 7$.

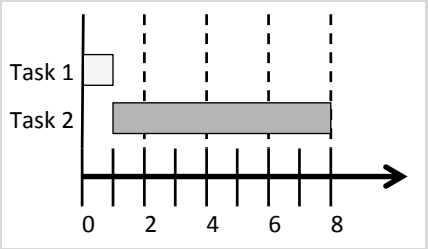
   (a) Sketch a rate-monotonic schedule (for 20 time units, the least common multiple of 4 and 10). Is the schedule feasible?

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



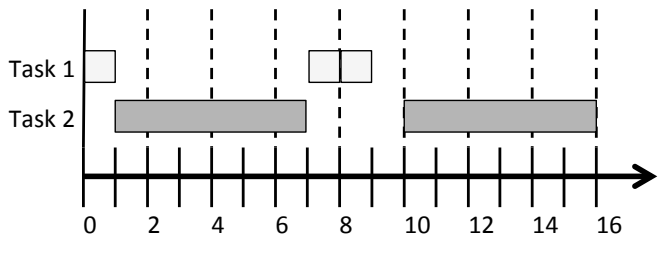
   (b) Now suppose task 1 and 2 contend for a mutex lock, assuming that the lock is acquired at the beginning of each execution and released at the end of each execution. Also, suppose that acquiring or releasing locks takes zero time and the priority inheritance protocol is used. Is the rate-monotonic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

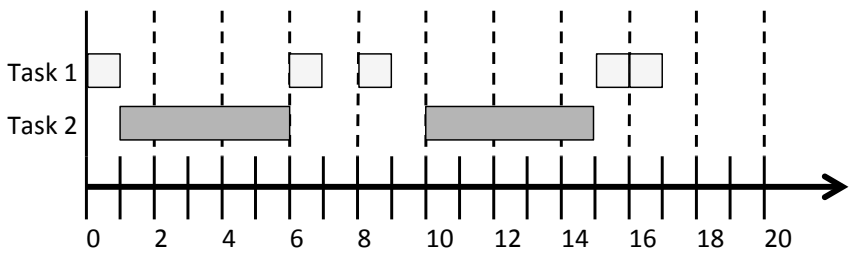


   (c) Assume still that tasks 1 and 2 content for a mutex lock, as in part (b). Suppose that task 2 is running an **anytime algorithm**, which is an algorithm that can be terminated early and still deliver useful results. For example, it might be an image processing algorithm that will deliver a lower 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:



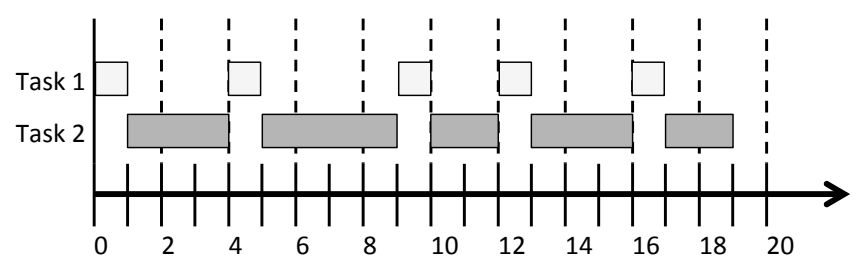
Reducing further to $e_2 = 5$ makes the schedule feasible, as shown below:



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?

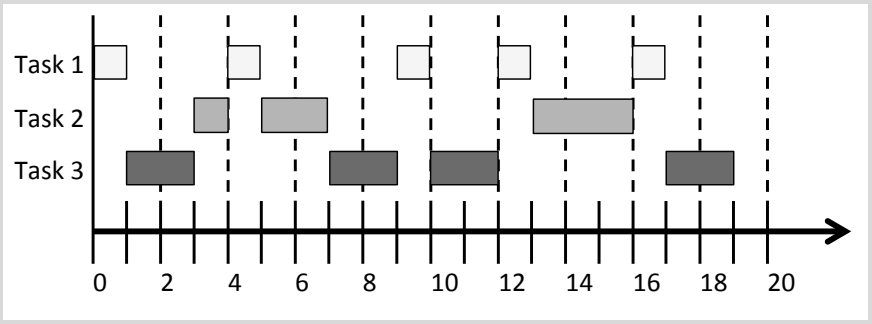
Solution: The EDF schedule is feasible. The figure below shows the schedule.



(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

3. The notion of reachability has a nice symmetry. Instead of describing all states that are reachable from some initial state, it is just as easy to describe all states from which some state can be reached. Given a finite-state system M, the **backward reachable states** of a set F of states is the set B of all states from which some state in F can be reached. The following algorithm computes the set of backward reachable states for a given set of states F:

Input : A set F of states and transition relation δ for closed finite-state system M
Output: Set B of backward reachable states from F in M

1 **Initialize:** $B := F$
2 $B_{\text{new}} := B$
3 **while** $B_{\text{new}} \neq \emptyset$ **do**
4 $\quad B_{\text{new}} := \{s \mid \exists s' \in B \text{ s.t. } s' \in \delta(s) \wedge s \notin B\}$
5 $\quad B := B \cup B_{\text{new}}$
6 **end**

Explain how this algorithm can check the property $\mathbf{G}p$ on M, where p is some property that is easily checked for each state s in M. You may assume that M has exactly one initial state s_0.

Solution: Let the input F to the algorithm be the set of all states where p does not hold. Then $\mathbf{G}p$ is true if and only if the output B of the algorithm does not contain the initial state s_0.