Universitetet i Oslo
Institutt for Informatikk

Research group for Reliable Systems (PSY)
Silvia Lizeth Tapia Tarifa, Andrea Pferscher, Juliane Päßler, Tobias John

INF 5170: Models of Concurrency

Fall 2024 **Group Session 3** 18.09.2024

Topic: Semaphores and Monitors

Exercise 1 (Semaphores: Precedence graph) ([1, Exercise 4.4a]) A precedence graph is a directed, acyclic graph. Nodes represent tasks, and arcs indicate the order in which tasks are to be accomplished. In particular, a task can execute as soon as all its predecessors have been completed. Assume that the tasks are processes and that each process has the following outline:

```
process T{
  wait for predecessors, if any;
  body of the task;
  signal successors, if any;
}
```

Using semaphores, show how to synchronize five processes whose permissible execution is specified by the precedence graph in Figure 1. Minimize the number of semaphores that you use,

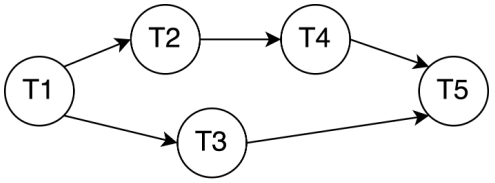

Figure 1: Precedence graph for Exercise 1

and do not impose constraints not specified in the graph. For example, `T2` and `T3` can execute concurrently after `T1` completes.

Exercise 2 (Semaphores: Bear and honeybees) ([1, Exercise 4.36]) Given are `n` honeybees and a hungry bear. They share a pot of honey. The pot is initially empty; its capacity is `H` portions of honey. The bear sleeps until the pot is full, then eats all the honey and goes back to sleep. Each bee repeatedly gathers one portion of honey and puts it in the pot; the bee who fills the pot awakens the bear.

Represent the bear and honeybees as processes and develop code that simulates their actions. Use semaphores for synchronization.

Exercise 3 (Monitors: Signalling disciplines and SJN) ([1, Exercise 5.3]) Consider the proposed solution to the shortest-job-next allocation problem in Listing 1. Does this solution work correctly for the signal and continue discipline? Does it work correctly for signal and wait?

Exercise 4 (Monitor solution to the readers/writers problem) The monitor in Listing 2 is used to control reader's and writer's access to a shared resource ([1, Fig. 5.5, page 216]). You may assume that signaling is handled by the *signal and continue* discipline.

Listing 1: SJN

```
1  monitor Shortest_Job_Next {
2    bool free := true;
3    cond turn;
4
5    procedure request(int time) {
6      if (free = false)
7          wait(turn, time);
8      free := false;
9    }
10   procedure release() {
11     free := true;
12     signal(turn);
13   }
14 }
```

a) In the monitor, the primitive `signal_all` is used in `release_write()`. Modify the monitor so that it uses `signal` instead but still awakes all readers waiting on `oktoread`.

b) In the given monitor, readers take precedence over writers. Modify the monitor such that writers take precedence over readers.

c) Modify the monitor so that readers and writers are allowed to access the resource *in turns* if both readers and writers want to access the resource.

d) Modify the monitor such that both readers and writers access the resource in a first-come-first-served (FCFS) manner. Allow more than one reader to access the resource as long as the FCFS-principle is satisfied. You may assume to have a (FIFO) queue `q` with the following operations:

- `enqueue(q,X)` returns `q` with the element `X` added at the end of the queue;
- `dequeue(q)` returns `q` with the first element removed;
- `inspect(q)` returns the first element of the queue without altering `q`;
- `empty(q)` returns true only when `q` is empty;
- an empty queue is declared by the statement `queue q := empty`.

References

[1] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.

Listing 2: A monitor for the readers/writers problem

```
1  monitor RW_Controller { # Invariant: (nr = 0 ∨ nw = 0) ∧ nw ≤ 1
2    int nr := 0, nw := 0   # number of readers, number of writers
3    cond oktoread;         # signalled when nw = 0
4    cond oktowrite;        # signalled when nr = 0 and nw = 0
5
6    procedure request_read() {
7      while (nw > 0) wait(oktoread);
8      nr := nr + 1;
9    }
10
11   procedure release_read() {
12     nr := nr − 1;
13     if nr = 0 signal (oktowrite);        # wake up one writer
14   }
15
16   procedure request_write() {
17     while (nr > 0 ∨ nw > 0) wait(oktowrite);
18     nw := nw + 1;
19   }
20
21   procedure release_write() {
22     nw := nw − 1;
23     signal(oktowrite);      # wake up one writer
24     signal_all(oktoread);   # wake up all readers
25   }
26 }
```