Universitetet i Oslo
Institutt for Informatikk
Research group for Reliable Systems (PSY)
Silvia Lizeth Tapia Tarifa, Riccardo Sieve

INF 5170: Models of Concurrency

Fall 2026 — **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:

```
1 process T{
2   wait for predecessors, if any;
3   body of the task;
4   signal successors, if any;
5 }
```

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,

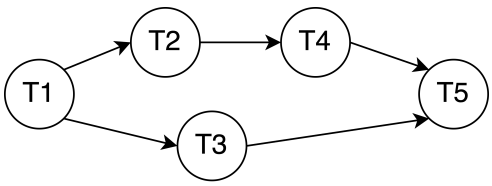


Figure 1: Precedence graph for Exercise 1

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

Solution: [for Exercise 1]

```
1  sem FIN1 = 0, FIN2 = 0, FIN3 = 0, FIN4 = 0;
2
3  process T1 {          process T2 {          process T3 {
4    task 1;               P(FIN1);              P(FIN1);
5    V(FIN1);              task 2;               task 3;
6    V(FIN1);              V(FIN2);              V(FIN3);
7  }                     }                     }
8
9  process T4 {          process T5 {
10   P(FIN2);              P(FIN3);
11   task 4;               P(FIN4);
12   V(FIN4);              task 5;
13 }                     }
```

The trick is to signal 2 times.

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.

Solution: [for Exercise 2] The problem is a variant of the producer/consumer problem in a way. Perhaps best to start thinking is the bees, because there are more than one. The bees have to be under mutex, but bees and the bear are also mutex. When a bee is finished, it can therefore *either* "signal" another bee to enter, or the bear. That can be done by a *split semaphore*, see Listing 1.

Listing 1: The bear and the honeybees

```
1  sem collect = 1          # Mutex semaphore
2  sem eat = 0              # Raise when the bear can eat
3  # collect and eat forms a split binary semaphore
4  int portions = 0         # Number of available portions in the pot
5
6  process bee[i = 1 to n] {
7    while (true) {
8      # collect honey
9      P(collect);
10     portions = portions + 1;
11     if (portions == H) {
12       V(eat);
13     } else {
14       V(collect);
15     } } }
16
17 process bear {
18   while(true) {
19     P(eat);
20     # eat
21     portions = 0;
22     V(collect);
23   }
24 }
```

Exercise 3 (Monitors: Signalling disciplines and SJN) ([1, Exercise 5.3]) Consider the proposed solution to the shortest-job-next allocation problem in Listing 2. Does this solution work correctly for the signal and continue discipline? Does it work correctly for signal and wait?

Solution: [for Exercise 3] The solution works with signal and wait since execution control is handed directly to the signalled process. With signal and continue, the solution might fail since the signalled process cannot directly execute. This might lead to another process "jumping over the `wait` statement": Consider the case where process $P1$ is waiting on the `turn` condition. Then another process $P2$ calls `release()` and therefore sets `free` to true and signals `turn`. However, $P1$ might not be able to execute directly after being signalled because we use signal-and-continue. Therefore, another process $P3$ requiring more time than $P1$ can call the `request` procedure, check that `free` is true and therefore does not need to wait on `turn` but instead directly sets `free` to false. Thereby, it can execute before $P1$.

Listing 2: 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 }
```

Exercise 4 (Monitor solution to the readers/writers problem) The monitor in Listing 3 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.

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`.

Solution: [of Exercise 4]

a) We modify the `release_write` procedure. The solution is straightforward, if we we can test the queue for emptiness, see Listing 4.

b) The "standard" solution gives preferences to readers because reading processes allows further readers to enter, but prohibit writers. For the "precedence to writers" solution, a reader must wait if a writer is active (as before) or if there are any *delayed* writers. We extend the code with a counter that counts the number of delayed writers and modify `request_read`, `release_write` and `request_write`, see Listing 5. It is not necessary to count the number of delayed readers.

Remark 1 The modified versions of `request_read` and `release_write` in Listing **??** do not solve the problem in b).

Listing 3: 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 }
```

Listing 4: Readers/writers with `signal` instead of `signal_all`

```
1  procedure release_write() {
2    nw := nw − 1;
3    signal(oktowrite);
4    while (!empty(oktoread)) signal(oktoread);
5  }
```

Listing 5: Readers/writers, writers precedence

```
1  monitor RW_Controller {
2    int nr := 0, nw := 0; # Invariant: (nr = 0 ∨ nw = 0) ∧ nw ≤ 1
3    int dw := 0;          # number of delayed writers
4    cond oktoread;        # signaled when nw = 0
5    cond oktowrite;       # signaled when nr = 0 and nw = 0
6
7    procedure request_read() {      # reading has less priority
8      while (nw + dw > 0) wait(oktoread);
9      nr := nr + 1;
10   }
11   procedure release_read() {      # unchanged
12     nr := nr − 1;
13     if (nr = 0) signal(oktowrite); # wake up one writer
14   }
15   procedure request_write() {
16     while (nr > 0 ∨ nw > 0) {     # basic invariant unchanged
17       dw := dw + 1;               # indicate writing wish
18       wait(oktowrite);
19       dw := dw − 1;               # writing wish has been granted
20     }
21     nw := nw + 1;
22   }
23   procedure release_write() {
24     nw := nw − 1;
25     if (!empty(oktowrite)) signal(oktowrite); # wake up one writer
26     else signal_all(oktoread);                # wake up all readers
27   }
28 }
```

Listing 6: Readers/writers, writers preference (does NOT solve b)

```
1  procedure request_read() {
2    while (nw > 0 ∨ !empty(oktowrite)) wait(oktoread);
3    nr := nr + 1;
4  }
5
6  procedure release_write() {
7    nw := nw − 1;
8    if (!empty(oktowrite)) signal(oktowrite);
9    else signal_all(oktoread);
10 }
11
```

Even though it may seem like a straightforward solution, it does not guarantee preference to writers.

Consider the situation where exactly one writer process is delayed on `oktowrite` when another writer starts execution of `release_write`. After `signal(oktowrite)`, the waiting writer is moved back into the entry queue of the monitor. Now, `empty(oktowrite)` is **true** and `nw = 0`. Thus, a newly arriving reader is given access to the shared resource.

Therefore, `!empty(oktowrite)` is *not* a sufficient condition to guarantee absence of waiting writers. Therefore, we follow the outline from [1, Exercise 5.7] and use counters to count the number of delayed processes.

c) When a reader releases the critical section, the critical section must be handed over to a waiting writer (if any). When a writer leaves the critical section, access must be given to a waiting reader. The solution in Listing 7 awakens all waiting readers when a writer

finishes. With signal and continue, the signalled processes are moved over to the entry queue where they compete with other processes for monitor access. We must therefore program in order to avoid that a new writer gets access to the critical section before the awakened readers. We achieve this by having two counters `dr` and `dw` counting the number of delayed readers and writers, respectively. A new reader must wait if there are active or delayed writers. A new writer must wait if there are active or delayed readers or active or delayed writers. For the readers, we need a separate counter ensuring that all signalled processes start execution before any new signalling happens by using `startup` because `signal_all` is used.

Listing 7: Readers/writers with turns

```
1  monitor RW_Controller {
2    int nr := 0, nw := 0;   # Invariant: (nr = 0 ∨ nw = 0) ∧ nw ≤
3    int dr := 0; dw := 0;   # number of delayed readers and writers
4    int startup := 0;       # number of readers signalled to read
5    cond oktoread;          # signaled when nw = 0
6    cond oktowrite;         # signaled when nr = 0 and nw = 0
7
8    procedure request_read() {
9      if (nw + dw > 0) {  # active writer or delayed processes
10       dr := dr + 1;       # now we cound the delayed readers, too
11       wait(oktoread);
12       dr := dr − 1;
13       startup := startup − 1;
14     }
15     nr := nr + 1;
16   }
17   procedure release_read() {
18     nr := nr − 1;
19     if (nr = 0 ∧ startup = 0) signal(oktowrite);  # wake up one writer
20   }
21   procedure request_write() {
22     if (nw + nr + dr + dw > 0) { # active or waiting processes
23       dw := dw + 1;
24       wait(oktowrite);
25       dw := dw − 1;
26     }
27     nw := nw + 1;
28   }
29   procedure release_write() {
30     nw := nw − 1;
31     if (dr > 0) {
32       signal_all(oktoread);
33       startup := dr;
34     }
35     else signal(oktowrite);
36   }
37 }
```

c) If we can check the queue, it is easy to enqueue all enter-requests. The basic "entry-conditions" from the original formulation from Listing 3 on page 4 are unchanged, except that here we do not use the while-loop. Besides that, we use the queue as additional condition: if the queue `q` is non-empty, the process has to "enqueue itself". By enqueue-itself, it enters an entry into the queue where it marks whether it is a reader or a writer. Besides that, it enqueues its identity in the queue associated with the condition variable. Since we have two condition variables, there are two different cv-process queues. However, there is only one "handmade" queue `q`.

The operation `inspect(q)` will be used to control the signaling. We represent the readers with the value `false` and the writers with the value `true` on the queue. The solution uses "*passing the condition*" in the sense that `q` will never be empty after we have signaled a condition variable, but before the process has entered the monitor. (In contrast to queues on condition variables which become empty if only one process waited on the queue.) By this, we avoid that a new process can sneak past the signaled process.

References

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

Listing 8: Readers/writers: FCFS

```
1  monitor RW_Controller {
2    bool READER := false;          # these two are constants
3    bool WRITER := true;
4    int nr := 0, nw := 0;
5    cond oktoread;
6    cond oktowrite;
7    queue q := empty;              # declaration of empty queue
8
9    procedure request_read() {
10     if (nw > 0 ∨ q != empty) {   # active writer or waiting process
11       q := enqueue(q,READER);    # update the queue with waiting reader
12       wait(oktoread);
13       q := dequeue(q);           # remove the first element of the queue
14     }
15     nr := nr + 1;
16     if (q != empty ∧ inspect(q)=READER)      # still reader first on the queue;
17       signal(oktoread);          # wake up next reader on the queue
18   }
19
20   procedure release_read() {     # At this point, we may signal a
21     nr := nr − 1;                # writer if it is in the front of the
22     if (nr = 0 ∧ inspect(q)=WRITER)   # queue.
23       signal(oktowrite);
24   }
25
26   procedure request_write() {
27     if (nr > 0 ∨ nw > 0 ∨ q != empty) { # active or waiting processes
28      q := enqueue(q,WRITER);     # update the queue with waiting writer
29      wait(oktowrite);
30      q := dequeue(q);            # remove first element of the queue
31     }
32     nw := nw + 1;
33   }
34
35   procedure release_write() {
36     nw := nw − 1;
37     if (q != empty) {            # some process is waiting
38       if (inspect(q)=READER)     # the next one is a READER
39         signal(oktoread);
40       else                       # the next one is a WRITER
41         signal(oktowrite);
42     }
43   }
44 }
```