Problem Set 4
Synchronization with semaphores

Reminders

A *critical resource* is a resource shared between multiple processes and whose access must be controlled to maintain the consistency of its contents (variable, queue, etc ...). In particular, this access control may involve *mutual exclusion*, that is to say that the resource is never accessed by more than one process at a time.
A *critical section* is a sequence of instructions in which the access of a process to a critical resource must be performed *indivisibly*.
Indivisibility means that there can be no concurrent executions of critical sections for the same resource.
Mutual exclusion is a special case of *synchronization*. A synchronization is an operation affecting the progress of a set of processes:

- to establish an order of occurrence among operations (send / receive),
- and/or to enforce a condition (rendez-vous, mutual exclusion).

Busy waiting is the mobilization of a processor for the repetitive execution of a synchronization primitive until the synchronization condition is verified.
A *semaphore* is a construct that helps control access to a resource by avoiding busy waiting. It consists of a counter and a queue. When positive, the value of the counter represents the number of resources available; when it is negative it represents the number of processes waiting for a resource access. The queue management policy is defined by the system designer. Upon creating a semaphore, the queue is empty. A semaphore is manipulated using the following indivisible operations:

`SEM* CS(cpt)`	Creates a semaphore whose counter is initialized to `cpt`
`DS(sem)`	Destroying a semaphore. Returns an error if the queue is not empty.
`P(sem)`	Requests an access to a resource. If there is no available access, the calling process is blocked
`V(sem)`	Releases an access to a resource. If the queue is not empty, one of the processes is unblocked.

1. Studying the anatomy of a semaphore

1.1.

Write the code of primitives P and V using the following operations and types:

- type `TASK` has a field `state` whose value is in `{READY, BLOCKED}`.
- type `SEM` has a field `count` of type `int` and a field `queue` of type `QUEUE`.
- `TASK* current()` returns the current running task.
- `void insert(TASK t, QUEUE* queue)` inserts a job in the queue.
- `TASK* extract(QUEUE* queue)` extracts a task from the queue.
- `sched()` triggers a context switch.

```
P(SEM *s) {
      s->count--;
      if (s->count < 0) {
            TASK *t = current()
            t->state = BLOCKED;
            insert(t, s->queue);
            sched();
      }
}

V(SEM *s) {
      s->count++;
      if (s->count <= 0) {
            TASK *t = extract(s->queue);
            t->state = READY;
      }
}
```

1.2.

Explain why these primitives must be made indivisible and how we can achieve this indivisibility.

A semaphore is a critical resource by definition, which makes the code of primitives P and V into critical sections.

The following examples show what might go wrong. The counter of a semaphore S has value 1, so calling P(S) a first time should be non-blocking, and calling it a second time should be blocking. Two processes P_0 and P_1 call P(S) concurrently. P_0 runs first and executes the first line which decrements the counter of S to 0. At this point there is a context switch: P0 loses the CPU, and P_1 gets elected. P_1 executes the first line of P(S), thus decrementing the counter to value -1, and then blocks and gets inserted in the queue. When P_0 gets elected again, it also blocks and gets inserted in the queue because of the counter value…

One solution for making these primitives indivisible is to execute them in system mode, mask interrupts while the counter and the queue are being manipulated.

1.3.

Why not just mask/unmask interrupts to perform a critical section in user mode?

Because there is no limit to the size and duration of the critical section. A process could acquire the critical section and then get stuck into an infinite loop, which would violate the liveness property of critical sections. Not only that, but there would be no way to hand the control of the CPU over to any other process.

1.4.

Consider a process currently executing a critical section. Can this process be interrupted by another process? If not, explain why; otherwise, explain what happens upon the interruption.
This spirit of this question is more or less the same as (1.3)
A process in a critical section MUST remain interruptible.
When a process P_0 that executes a critical section gets interrupted by another process P_1, it is evicted from the CPU, and the kernel starts the interrupt routine. Upon returning from the interrupt routine, the kernel hands the CPU back to P_0.

If you want to investigate semaphore implementation further, you can take a look at Peter Druschel's lecture notes: https://people.mpi-sws.org/~druschel/courses/os/lectures/proc4.pdf

2. First steps with semaphores

Consider the three following processes A, B, and C.

Process A	Process B	Process C
(a) `P(Mutex);` (b) `x = x + 1;` (c) `V(Mutex);`	(d) `P(Mutex);` (e) `x = x * 2;` (f) `V(Mutex);`	(g) `P(Mutex);` (h) `x = x - 4;` (i) `V(Mutex);`

A, B, and C run concurrently, and the value of semaphore `Mutex` is initialized to 1.

2.1.

Let the instructions be executed in the following sequence: a d b g c e f h i
Perform a *hand execution* of this sequence. After every instruction, give:
- the value of the counter,
- the contents of the queue of the semaphore,
- the status of each process (ready, blocked).

Is this sequence valid (ie. can the operations actually occur in this order)?

Instruction	Counter	Queue	Statuses
	1	Ø	Ar Br Cr
a	0	Ø	Ar Br Cr
d	-1	B	Ar Bb Cr
b	-1	B	Ar Bb Cr
g	-2	B C	Ar Bb Cb
c	-1	C	Ar Br Cb
e	-1	C	Ar Br Cb
f	0	Ø	Ar Br Cr
h	0	Ø	Ar Br Cr
i	1	Ø	Ar Br Cr

Consider another sequence: a d e b c f g h i. Is this sequence valid?
This sequence is not valid, because after the execution of (a) and (d), the counter of MUTEX has value -1 and process B is blocked. Therefore instruction (e) cannot get executed.

Now consider the three following processes running concurrently on one processor. Variable `a` is a shared variable, and it gets initialized along with semaphores `S1` and `S2` as follows:

```
int a = 6; S1 = CS (1); S2 = CS (0);
```

Process A	Process B	Process C
(a) `P(S1);` (1) `a = a + 7;` (b) `V(S2);`	(2) `a = a - 5;`	(c) `V(S1);` (d) `P(S2);` (3) `a = a * 3;`

2.2.

Give all the valid sequences of (numbered) instructions, and the value of `a` associated with every valid sequence.

The counter of S2 is initialized at 0, so instruction (1) is always executed before instruction (3).
Instruction (2) can get executed at any point.
Therefore there are 3 valid sequences:

(2) (1) (3) => a = 24
(1) (2) (3) => a = 24
(1) (3) (2) => a = 34

2.3.

Which semaphore could be deleted without any consequence for the execution of all processes? Justify your answer.

The counter of S1 is initialized at 1, and P(S1) is only called once so no process will ever block on S1.

2.4.

Add one semaphore (give its initialization value) and change the code of the processes to make sure that any execution will result with `a` equal to 34.

As shown in (2.2), the only valid sequence that ensures a = 34 is (1) (3) (2)
Instruction (2) must get executed after instruction (3)
This can be obtained by adding a semaphore S3 (init. 0), an instruction P(S3) in process B before instruction (2), and an instruction V(S3) in process C after instruction (3)

2.5.

Repeat question (2.4), this time to make sure that any execution will result with `a` equal to 24.

This time instruction (2) must get executed *before* instruction (3)
Add:

- a semaphore S3 (init. 0),
- an instruction V(S3) in process B after instruction (2),
- and an instruction P(S3) in process C before instruction (3)

We modify the code of processes A, B, and C as follows. Variable `a` is still a shared variable, and it gets initialized along with semaphores `S1` and `S2` as follows:

```
int a = 6; S1 = CS (1); S2 = CS (1);
```

Process A	Process B	Process C
(a) `P(S1);` (b) `P(S2);` (1) `a = a + 7;` (c) `V(S1);` (d) `V(S2);`	(e) `P(S2);` (2) `a = a - 5;` (f) `V(S2);` (g) `V(S1);`	(h) `P(S2);` (i) `P(S1);` (3) `a = a * 3;` (j) `V(S2);`

2.6.

Find a sequence of (lettered) instructions which leads to a *deadlock*.
If (h) gets executed before (e) and (b), and if (a) gets executed before (i), then all three instructions (b), (e), and (h) become blocking. This leads to a deadlock because all three processes will eventually become blocked.

3. Synchronization and deadlock

Consider N user processes U_1 to U_N that share a set of X servers providing access to Y resources (X> 0, Y> 0, N> X + Y). The processes are synchronized by means of 2 semaphores S and R, the respective representations of the availability of servers and resources, along with the following program:

```
S = CS(X); R = CS(Y);

Begin loop
      P (S); P (R); V (S);  /* Prologue */
      workworkwork();       /* Resource usage */
      P (S); V (R); V (S);  /* Epilogue */
End loop
```

3.1.

What is the maximum number of processes that can simultaneously access a resource?
In order to gain access to a resource, a process acquires a server with P(S) and then releases it with V(S). The same goes for releasing a resource. It follows that semaphore S does not control access to the resources, only semaphore R does. Since the counter of R is initialized to Y, there can be at most Y concurrent accesses to resources by processes.

3.2.

Show that, for N = 2 and X = Y = 1, concurrent executions of this program can lead to a deadlock.

Let's number the instructions that affect semaphores:

```
(1) P(S); (2) P(R); (3) V(S);
/* Resource usage */
(4) P(S); (5) V(R); (6) V(S);
```

N is 2 so we have 2 processes P_1 and P_2

Both S.counter and R.counter are initially set to 1.

Let P_2 execute the sequence (1) (2) (3) and spend time using the unique resource.

S.counter is now at value 1 and R.counter at value 0.

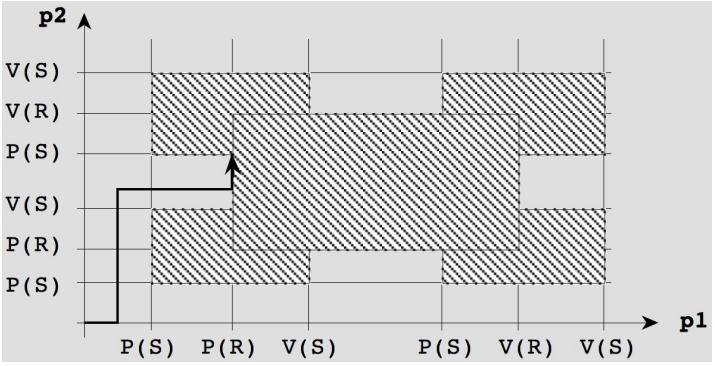
While P_2 uses the resource, P_1 executes (1); this decrements S.counter to 0.

It leaves P_1 free to execute (2); this decrements R.counter to -1 and blocks P_1.

When P_2 is done using the resource, it tries to acquire a server with instruction (4); this decrements S.counter to -1 and blocks P_0.

Both P_2 and P_1 are now blocked in a deadlock.

The following figure illustrates this situation:



3.3.

Let X and Y be arbitrary values strictly greater than 0. What is the minimum number of processes which may lead to a deadlock?

As long as at least 1 server is accessible, processes that have acquired a resource can release it. The deadlock becomes possible when there are Y processes concurrently using a resource after successfully executing the sequence (1)(2)(3), and X processes that are blocked on instruction (2). In this situation, none of the Y processes can acquire a server to release its resource.

It follows that a deadlock can occur if there are at least X+Y concurrent processes involved.

3.4.

The deadlock can be prevented by introducing a new semaphore L to limit the number of processes allowed into the prologue. Insert the operations P(L) and V(L) where necessary in the prologue and the epilogue, and give the initialization value of L.

The deadlock we've identified can be prevented by limiting the number of concurrent processes that are allowed to acquire servers and resources. Inserting P(L) before instruction (1) and V(L) after instruction (6) provides control over this; now we need to set the initial value of L. Even if all the resources are locked, there is no deadlock as long as at least one server can be acquired to release a resource. Therefore L should be initialized to X+Y-1

We can increase the concurrency of our solution by releasing L even earlier, ie. in between instructions (4) and (5).

The following figure illustrates this solution: