Exercise Session
Synchronization / Resource allocation

Operating Systems
Chalmers University of Technoloy – Gothenburg University

Question 1

We have three threads A, B, and C taking care of operations opA, opB, and opC respectively. The threads use three semaphores with the following initial value: semA=1, semB=1, and semC=0.

Thread A:

```
wait(semC);
wait(semB);
opA; // some operation
signal(semB);
signal(semA);
```

Thread B:

```
wait(semA);
wait(semB);
opB; //some operation
signal(semB);
```

Thread C:

```
wait(semA);
wait(semB);
opC; //some operation
signal(semB);
signal(semA);
signal(semC);
```

Are the following executions possible or not and why?

(i) opA opB opC

(ii) opB opC opA

(iii) opC opA opB

(iv) opB opA opC

Question 1 - Solution

Initial values: semA=1, semB=1, semC=0

Thread A:
```
  wait(semC);
  wait(semB);
  opA; // some operation
  signal(semB);
  signal(semA);
```

Thread B:
```
  wait(semA);
  wait(semB);
  opB; //some operation
  signal(semB);
```

Thread C:
```
  wait(semA);
  wait(semB);
  opC; //some operation
  signal(semB);
  signal(semA);
  signal(semC);
```

Are the following executions possible or not and why?

(i) opA opB opC

no: A blocks at semC until C has executed opC

Question 1 - Solution

Initial values: semA=1, semB=1, semC=0

Thread A:
```
  wait(semC);
  wait(semB);
  opA; // some operation
  signal(semB);
  signal(semA);
```

Thread B:
```
  wait(semA);
  wait(semB);
  opB; //some operation
  signal(semB);
```

Thread C:
```
  wait(semA);
  wait(semB);
  opC; //some operation
  signal(semB);
  signal(semA);
  signal(semC);
```

Are the following executions possible or not and why?

(ii) opB opC opA

no: B "consumes" from semA but does not produce in it, hence C will block

Question 1 - Solution

Initial values: semA=1, semB=1, semC=0

Thread A:
```
wait(semC);
wait(semB);
opA; // some operation
signal(semB);
signal(semA);
```

Thread B:
```
wait(semA);
wait(semB);
opB; //some operation
signal(semB);
```

Thread C:
```
wait(semA);
wait(semB);
opC; //some operation
signal(semB);
signal(semA);
signal(semC);
```

Are the following executions possible or not and why?

(iii) opC opA opB

Yes, it is possible.

Question 1 - Solution

Initial values: semA=1, semB=1, semC=0

Thread A:

```
  wait(semC);
  wait(semB);
  opA; // some operation
  signal(semB);
  signal(semA);
```

Thread B:

```
  wait(semA);
  wait(semB);
  opB; //some operation
  signal(semB);
```

Thread C:

```
  wait(semA);
  wait(semB);
  opC; //some operation
  signal(semB);
  signal(semA);
  signal(semC);
```

Are the following executions possible or not and why?
(iv) opB opA opC

no: same as (ii), now A blocks at semC until C has executed opC; ie if B executes first, no other thread can proceed.

Question 2

(a) Consider two threads, A and B. Thread B must execute operation opB only after thread A has completed operation opA. How can you guarantee this synchronization using semaphores?

(b) Consider two threads, A and B which must forever take turns executing operation opA and operation opB, respectively. Thread A must be the one that executes opA first. How can you guarantee that using semaphores?

Question 2 - Solution

(a) Consider two threads, A and B. Thread B must execute operation opB only after thread A has completed operation opA. How can you guarantee this synchronization using semaphores?

Use semaphore *flag*, initialized to 0

```
Thread A:
    ...
    opA
    signal(flag)
```

```
Thread B:
    ...
    wait(flag)
    opB
```

Thread B will be able to proceed from wait(*flag*) only after signal(*flag*) is executed by thread A

Question 2 - Solution

(a) Consider two threads, A and B. Thread B must execute operation opB only after thread A has completed operation opA. How can you guarantee this synchronization using semaphores?

(b) Consider two threads, A and B which must forever take turns executing operation opA and operation opB, respectively. Thread A must be the one that executes opA first. How can you guarantee that using semaphores?

Now the threads work in a loop, using semaphores SA and SB. Thread A must wait for opB, except the first time, hence SB is initialized to 1. Thread B must wait for opA with SA initialized to 0. After opA (resp opB), A executes signal(SA) (resp B executes signal(SB), to generate the required signal to enable the other one to proceed.

Initial values:
SA = 0, SB = 1

```
Thread A:
    while(true):
        wait(SB)
        opA
        signal(SA)
```

```
Thread B:
    while(true):
        wait(SA)
        opB
        signal(SB)
```

Question 3

Servers can be designed to limit the number of open connections. For example, a server may wish to have only N active socket connections at any point in time. As soon as N connections are made, the server will not accept a new connection until an existing one is released. With pseudocode, describe how semaphores can be used to limit the number of concurrent connections.

Question 3 - Solution

Servers can be designed to limit the number of open connections. For example, a server may wish to have only N active socket connections at any point in time. As soon as N connections are made, the server will not accept a new connection until an existing one is released. With pseudocode, describe how semaphores can be used to limit the number of concurrent connections.

General semaphore S, initialized to N; execute wait(S) and signal(s) before and after the connection between each client (socket) and server.

Question 4

Show a method that solves the critical section problem for arbitrary number of threads using the atomic TestAndSet instruction that is available in several processor architectures. Use pseudocode in the description and argue about the properties of the solution, with respect to mutual exclusion, progress and fairness. (It is not necessary to describe a solution that guarantees fairness in this question, but if you can, of course it is ok).

Question 4 - Solution

Show a method that solves the critical section problem for arbitrary number of threads using the atomic TestAndSet instruction that is available in several processor architectures. Use pseudocode in the description and argue about the properties of the solution, with respect to mutual exclusion, progress and fairness. (It is not necessary to describe a solution that guarantees fairness in this question, but if you can, of course it is ok).

Without fairness:

```
boolean TAS(Boolean *target)
{
    boolean rv = *target;
    *target  = true;
    return rv;
}
```

Shared Boolean variable *lock* initialized to false

Threads:

```
    repeat
        while (TAS(&lock)); //busywait
        [critical section]
        lock = false;
        …
    forever
```

Question 4 - Solution

With fairness:

Shared Boolean variable *lock,* initialized to false
Shared Boolean array waiting[0…n-1], initialized to false

Threads:

```
repeat
    waiting[i] = true;
    while (TAS(&lock) && waiting[i]); //busywait
    waiting[i] = false;
    [critical section]
    j = (i+1)%n;
    while(j!=i && !waiting[j]) //find the next one waiting to hand over the lock
        j = (j+1)%n;
    if (i==j)
        lock = false; //completed one round without handing over; release lock
    else
        waiting[j] = false; //handover the lock
forever
```