11.03.2025 - LECTURE 4

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

CONCURRENT SYSTEMS
LECTURE 4

Prof. Daniele Gorla

→ Concurrent objects

Semaphores

Object: entity with an implementation (hidden) and an interface (visible), made up of a set of operations and a specification of the behaviour (usually specified in a sequential way – e.g., as a set of legal executions).

Concurrent: if the object can be accessed by different processes

Semaphore: is a shared counter S accessed via primitives *up* and *down* s.t.:

1. It is initialized at $s0 \geq 0$
2. It is always ≥ 0
3. *up* atomically increases S
4. *down* atomically decreases S, provided that it is not 0; otherwise, the invoking processes is blocked and waits. → for the counter to be strictly positive

Invariant: $S = s0 + \#(S.up) - \#(S.down)$

Main use: prevent busy waiting (suspend processes that cannot perform *down*)

- **Strong**, if uses a FIFO policy for blocking/unblocking processes, **weak** otherwise
- **Binary**, if it is at most 1 (so, also *up* are blocking) (mutual exclusion as we have seen so far)

2 underlying objects:

- A counter, initialized at s0 that can also become negative
- A data structure (typicaly, a queue), initially empty, to store suspended proc's

Semaphores: ideal implementation

```
S.down() :=
   S.counter--
   if S.counter < 0 then
        enter into S.queue
        SUSPEND
   return
```

```
S.up() :=
   S.counter++
   if S.counter ≤ 0 then
        activate a proc from S.queue
   return
```

Remark 1: if S.counter ≥ 0, then this is the value of the semaphore; otherwise, S.counter tells you how many processes are suspended in S

↳ the absolute value of it

Remark 2: all operations are in MUTEX

Semaphores: actual implementation

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

```
Let t be a test&set register initialized at 0

S.down() :=                          S.up() :=
   Disable interrupts                   Disable interrupts
   wait S.t.test&set() = 0              wait S.t.test&set() = 0
   S.counter--                          S.count++
   if S.counter < 0 then                if S.count ≤ 0 then
        enter into S.queue                   activate a proc from S.queue
        S.t ← 0                         S.t ← 0
        Enable interrupts               Enable interrupts
        SUSPEND                         return
   else S.t ← 0
        Enable interrupts
   return
```

see lecture 3

Test & set object to ensure MUTEX, but it could be with any hardware implementation seen so far one producer and one consumer

Printer example

(Single) Producer/Consumer

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

It is a shared FIFO buffer of size k. Internal representation:

- BUF[0,…,k-1] : generic registers (not even safe) accessed in MUTEX
- IN/OUT : two variables pointing to locations in BUF to (circularly) insert/remove items, both initialized at 0
- FREE/BUSY : two semaphores that count thew number of free/busy cells of BUF, initialized at k and 0 respectively.

Blocking if FREE becomes negative (same with BUSY)

```
B.produce(v) :=
    FREE.down()
    BUF[IN] ← v
    IN ← (IN+1) mod k
    BUSY.up()
    return
```

since it is circular

"there is something to consume"

```
B.consume() :=
    BUSY.down()
    tmp ← BUF[OUT]
    OUT ← (OUT+1) mod k
    FREE.up()
    return tmp
```

This act as a check to see if there is something to be consumed

Remark: reading from/writing into the buffer can be very expensive!

You need two semaphore because you suspend only when the value of the semaphore is zero (with a custom implementation I can use just a single semaphore)

(Multiple) Producers/Consumers

SAPIENZA
Università di Roma
Dipartimento di Informatica

Accessing BUF in MUTEX slows down the implementation

→ we'd like to have the possibility of parallel read/write from different cells

- 2 arrays FULL and EMPTY of atomic boolean registers, initialized at ff and tt, resp
- We have two extra semaphores SP and SC, both initialized at 1

All ones

All zeros

```
B.produce(v) :=
    FREE.down()
    SP.down()
    while ¬EMPTY[IN] do
         IN ← (IN+1) mod k
    i ← IN
    EMPTY[IN] ← ff
    SP.up()
    BUF[i] ← v
    FULL[i] ← tt
    BUSY.up()
    return
```

Eventually an empty location will be found, otherwise the process would be blocked in the semaphore

```
B.consume() :=
    BUSY.down()
    SC.down()
    while ¬FULL[OUT] do
         OUT ← (OUT+1) mod k
    o ← OUT
    FULL[OUT] ← ff
    SC.up()
    tmp ← BUF[o]
    EMPTY[o] ← tt
    FREE.up()
    return tmp
```

At home: reason on that and find a counterexample

(Multiple) Producers/Consumers

Why is this solution wrong?

```
B.produce(v) :=
    FREE.down()
    SP.down()
    i ← IN
    IN ← (IN+1) mod k
    EMPTY[IN] ← ff
    SP.up()
    BUF[i] ← v
    FULL[i] ← tt
    BUSY.up()
    return
```

```
B.consume() :=
    BUSY.down()
    SC.down()
    o ← OUT
    OUT ← (OUT+1) mod k
    FULL[OUT] ← ff
    SC.up()
    tmp ← BUF[o]
    EMPTY[o] ← tt
    FREE.up()
    return tmp
```

Hint: the problem is related to the relative speed of processes (e.g., consider very quick producers and a few very slow consumers – e.g., the first consumer is very very slow)

The Readers/Writers problem

- Several processes want to access a file
- Readers may simultaneously access the file
- At most one writer at a time
- Reads and writes are mutually exclusive

Remark: this generalizes the MUTEX problem (MUTEX = RW with only writers)

The read/write operations on the file will all have the following shape:

```
conc_read() :=                  conc_write() :=
   begin_read()                    begin_write()
   read()                          write()
   end_read()                      end_write()
```

Weak priority to Readers

Risk of starvation for the writers

- If a reader arrives during a read, it can surpass possible writers already suspended
- When a writer terminates, it activates the first suspended process, irrispectively of whether it is a reader or a writer (so, the priority to readers is said «weak»)

```
GLOB_MUTEX and R_MUTEX semaphores init. at 1
R a shared register init. at 0
```

```
begin_read() :=
    R_MUTEX.down()
    R++
    if R = 1 then GLOB_MUTEX.down()
    R_MUTEX.up()
    return
```

R++ ← currently active readers

R = 1: I'm the first reader

```
end_read() :=
    R_MUTEX.down()
    R--
    if R = 0 then GLOB_MUTEX.up()
    R_MUTEX.up()
    return
```

R = 0: I'm the last reader

```
begin_write() :=
    GLOB_MUTEX.down()
    return
```

```
end_write() :=
    GLOB_MUTEX.up()
    return
```

Strong priority to Readers

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

- When a writer terminates, it activates the first reader, if there is any, or the first writer, otherwise.

```
GLOB_MUTEX, R_MUTEX and W_MUTEX semaphores init. at 1
R a shared register init. at 0
```

```
begin_read() :=                          end_read() :=
```

like before

```
begin_write() :=                         end_write() :=
   W_MUTEX.down()                           GLOB_MUTEX.up()
   GLOB_MUTEX.down()                        W_MUTEX.up()
   return                                   return
```

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Weak priority to Writers

```
GLOB_MUTEX, PRIO_MUTEX, R_MUTEX and W_MUTEX semaphores init. at 1
R and W shared registers init. at 0
```

To prioritize the writers

```
begin_read() :=
    PRIO_MUTEX.down()
    R_MUTEX.down()
    R++
    if R = 1 then GLOB_MUTEX.down()
    R_MUTEX.up()
    PRIO_MUTEX.up()
    return
```

```
end_read() :=    (like weak priority)

    R_MUTEX.down()
    R--
    if R = 0 then GLOB_MUTEX.up()
    R_MUTEX.up()

    return
```

```
begin_write() :=
    W_MUTEX.down()
    W++
    if W = 1 then PRIO_MUTEX.down()
    W_MUTEX.up()
    GLOB_MUTEX.down()
    return
```

```
end_write() :=
    GLOB_MUTEX.up()
    W_MUTEX.down()
    W--
    if W = 0 then PRIO_MUTEX.up()
    W_MUTEX.up()
    return
```

End of lecture 4

Monitors

Semaphores are hard to use in practice because quite low level

Monitors provide an easier definition of concurrent objects at the level of Prog. Lang.

- A concurrent object that guarantees that at most one operation invocation at a time is active inside it
- Internal inter-process synchronization is provided through *conditions*
- **Conditions** are objects that provide the following operations:
  - *wait*: the invoking process suspends, enters into the condition's queue, and releases the mutex on the monitor
  - *signal*: if no process is in the condition's queue, then nothing happens. Otherwise
    - Reactivates the first suspended process, suspends the signaling process that however has a priority to re-enter the monitor (w.r.t. processes that are suspended on conditions)

      → *Hoare semantics*
    - Completes its task and the first process in the condition's queue has priority to enter the monitor (after that the signaling one terminates or suspends)

      → *Mesa semantics*

Rendez-vous through monitors

Rendez-vous is a concurrent object associated to *m* control points (one for every process involved), each of which can be passed when all processes are at their control points.

The set of all control points is called ***barrier***.

```
monitor RNDV :=
        cnt ∈ {0,…,m} init at 0

        condition B

        operation barrier() :=
                cnt++
                if cnt < m then B.wait()
                            else cnt ← 0
                B.signal()
                return
```

Implementation through semaphores

- A semaphore MUTEX init at 1 (to guarantee mutex in the monitor)
- For every condition C, a semaphore SEM_C init at 0 and an integer N_C init at 0 (to store and count the number of suspended processes on the given condition)
- A semaphore PRIO init at 0 and an integer N_{PR} init at 0 (to store and count the number of processes that have performed a signal, and so have priority to re-enter the monitor)

1. Every monitor operation starts with `MUTEX.down()` and ends with
   `if N_PR > 0 then PRIO.up() else MUTEX.up()`
2. `C.wait() :=`
```
N_C++
if N_PR > 0 then PRIO.up() else MUTEX.up()
SEM_C.down()
N_C--
return
```
3. `C.signal() :=`
```
if N_C > 0 then  N_PR++
                 SEM_C.up()
                 PRIO.down()
                 N_PR--
return
```

Monitors for Rs/Ws: Strong Priority to Readers


```
monitor RW_READERS :=
   AR, WR, AW, WW init at 0
   condition CR, CW

   operation begin_read() :=
         WR++
         if AW≠0 then CR.wait()
                      CR.signal()
         AR++
         WR--

   operation end_read() :=
         AR--
         if AR+WR=0 then CW.signal()

   operation begin_write() :=
         if (AR+WR≠0 OR AW≠0) then
                   CW.wait()
         AW++

   operation end_write() :=
         AW--
         if WR > 0 then
               CR.signal()
         else CW.signal()
```

Remark: possible starvation for writers!

Monitors for Rs/Ws: Strong Priority to Writers


```
monitor RW_WRITERS :=
   AR, WR, AW, WW init at 0
   condition CR, CW

   operation begin_read() :=
        if WW+AW≠0 then CR.wait()
                        CR.signal()
        AR++

   operation end_read() :=
        AR--
        if AR=0 then CW.signal()

   operation begin_write() :=
        WW++
        if AR+AW≠0 then CW.wait()
        AW++
        WW--

   operation end_write() :=
        AW--
        if WW > 0  then CW.signal()
                   else CR.signal()
```

Remark: possible starvation for readers!

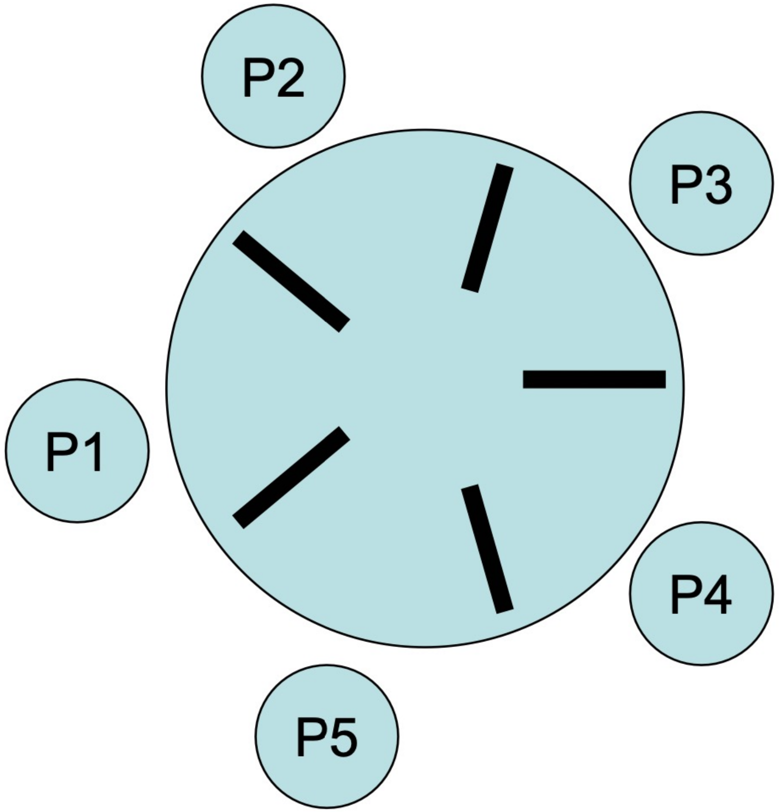
Monitors for Rs/Ws: a fair solution

- After a write, all waiting readers are enabled
- During a read, new readers must wait if writers are waiting

```
monitor RW_FAIR :=
   AR, WR, AW, WW init at 0
   condition CR, CW

   operation begin_read() :=
        WR++
        if WW+AW≠0 then CR.wait()
                        CR.signal()
        AR++
        WR--

   operation end_read() :=
        AR--
        if AR=0 then CW.signal()

   operation begin_write() :=
        WW++
        if AR+AW≠0 then CW.wait()
        AW++
        WW--

   operation end_write() :=
        AW--
        if WR > 0  then CR.signal()
                   else CW.signal()
```

Dining Philosophers (Dijkstra, 1965)

- *N* philosophers seated around a circular table
- There is one chopstick between each pair of philosophers
- A philosopher must pick up its two nearest chopsticks in order to eat
- A philosopher must pick up first one chopstick, then the second one, not both at once



PROBLEM: Devise a deadlock-free algorithm for allocating these limited resources (chopsticks) among several processes (philosophers).

A non-deadlock-free solution

A simple algorithm for protecting access to chopsticks:

each chopstick is governed by a mutual exclusion semaphore that prevents any other philosopher from picking up the chopstick when it is already in use by another philosopher

```
semaphore chopstick[5] initialized to 1
Philosopher(i) :=
        while(1) do
                chopstick[i].down()
                chopstick[(i+1)%N].down()
                // eat
                chopstick[(i+1)%N].up()
                chopstick[i].up()
```

Guarantees that no two neighbors eat simultaneously, i.e. a chopstick can only be used by one its two neighboring philosophers

We can have deadlock if all philosophers simultaneously grab their right chopstick

Deadlock-free solutions

Break the symmetry of the system:

- All philosophers first grab their left-most chopstick, apart from one (e.g., the last one) that first tries to grab the right-most one
- odd philosophers pick first left then right, while even philosophers pick first right then left
- allow at most 4 philosophers at the same table when there are 5 resources

We shall also see a solution where symmetry is not broken

- allow a philosopher to pick up chopsticks only if both are free. This requires protection of critical sections to test if both chopsticks are free before grabbing them.

  → this will be easily implemented through a monitor

Solution 1

Give a number to forks and always try with the smaller

→ all philosophers first pick left and then right, except for the last one that first picks right and then left.

```
semaphores fork[N] all initialized at 1;
Philosopher(i) :=
   Repeat
         think;
         if (i < N-1) then
                 fork[i].down();
                 fork[i+1].down();
         else
                 fork[0].down();
                 fork[N-1].down();
         eat;
         fork[(i+1)%N].up();
         fork[i].up();
```

Solution 2

Odd philosophers first pick left and then right, even philosophers first pick right and then left.

```
semaphores fork[N] all initialized at 1;
Philosopher(i) :=
   Repeat
        think;
        if (i % 2 == 0) then
                fork[i].down();
                fork[(i+1)%N].down();
        else
                fork[(i+1)%N].down();
                fork[i].down();
        eat;
        fork[(i+1)%N].up();
        fork[i].up();
```

Solution 3

Allow at most N-1 philosophers at a time sitting at the table

```
semaphores fork[N] all initialized at 1
semaphore table initialized at N-1

Philosopher(i) :=
   Repeat
         think;
         table.down();
         fork[i].down();
         fork[(i+1)%N].down();
         eat;
         fork[(i+1)%N].up();
         fork[i].up();
         table.up()
```

Solution 4

Pick up 2 chopsticks only if both are free

- a philosopher moves to his/her eating state only if both neighbors are not in their eating states

  → need to define a state for each philosopher

- if one of my neighbors is eating, and I'm hungry, ask them to signal me when they're done

  → thus, states of each philosopher are: thinking, hungry, eating

  → need condition variables to signal waiting hungry philosopher(s)

This solutoin very well fits with the features of monitors!

Solution 4

```
monitor DP
   status state[N] all initialized at thinking;
   condition self[N];

   Pickup(i) :=
        state[i] = hungry;
        test(i);
        if (state[i] != eating) then self[i].wait;

   Putdown(i) :=
        state[i] = thinking;
        test((i+1)%N);
        test((i-1)%N);

   test(i) :=
        if (state[(i+1)%N] != eating && state[(i-1)%N] != eating
            && state[i] == hungry)
        then    state[i] = eating;
                self[i].signal();
```