The readers-writers problem

Communication through message passing

The readers-writers problem: definition

This problem is a generalization of the mutual exclusion problem. There are two types of processes:

The readers that only read the information and thus can simultaneously access the critical section;

The writers that modify (write) the data and thus must have access in strict mutual exclusion.

Thus, may simultaneous have access to the critical section either

- A single writer and no readers, or
- Several readers and no writers.

The readers-writers problem: Solution structure

- The idea of the solution if to generalize the notion of semaphore into a shared object of type `RWflag` designed to manage the required more elaborate notion of mutual exclusion.

- Four operations are possible on the shared object:
  - `startRead` and `startWrite` that, respectively for the readers and writers, replace the operation `wait` used in the semaphore implementation of mutual exclusion;
  - `endRead` and `endWrite` that, respectively for the readers and writers, replace the operation `signal`.

The readers-writers problem: The reader and the writer processes

The reader and writer processes respectively execute the programs below.

- `rw` is a shared object of type `RWflag`,
- `readthedata()` is the method performing read access to the shared data, and
- `writethedata()` is the method performing write access to the data.

Reader

```
{ while(true)
  { rw.startRead();
    readthedata();
    rw.endRead();
  }
}
```

Writer

```
{ while(true)
  { rw.startWrite();
    writethedata();
    rw.endWrite();
  }
}
```

The class `RWflag` still needs to be defined.

The readers-writers problem: The class RWflag

- The class `RWflag` uses two wait queues: `okw` for the processes waiting to write and `okr` for the processes waiting to read. These queues will be directly implemented with semaphores.
- In order to have immediate resumption, mutual exclusion of the methods of the class `RWflag` is explicitly implemented with a semaphore `mutex`.

```
public class RWflag {
  private int readers;      /* nb of readers */
  private boolean writing; /* writer active ? */
  private SemaphoreFIFO mutex, okw, okr;

  public RWflag()
  { writing = false; readers = 0; mutex = new SemaphoreFIFO(1);
    okw = new SemaphoreFIFO(0); okr = new SemaphoreFIFO(0);
  }
```

The readers-writers problem: The methods `startRead()` and `startWrite()` (1st attempt)

```
public void startRead()
{ mutex.semWait();
    if (writing)
    { mutex.semSignal();
      okr.semWait();
    }
    readers++;
    mutex.semSignal();
}
```

```
public void startWrite()
{ mutex.semWait();
  if ((readers != 0) ||
       writing)
  { mutex.semSignal();
    okw.semWait();
  }
  writing = true;
  mutex.semSignal();
}
```

The readers-writers problem: The methods `endRead()` and `endWrite()` (1st attempt)

```
public void endRead()
{ mutex.semWait();
  readers--;
  if ((readers == 0)
       && (okw.semNbWait() > 0))
    okw.semSignal();
  else mutex.semSignal();
}
```

```
public void endWrite()
{ mutex.semWait();
  writing = false ;
  if (okr.semNbWait() > 0)
    okr.semSignal();
  else
    if (okw.semNbWait() > 0)
      okw.semSignal();
    else mutex.semSignal();
}
```

The readers-writers problem: Notes on the 1st attempt

- In `startRead` (`startWrite`), the process is placed on the corresponding wait queue, if the condition for reading (writing) is not satisfied.
- In `endRead` a waiting writer (if any) is freed when the number of readers reaches 0.
- In `endWrite` one can choose between freeing a waiting reader of a waiting writer. If there is a waiting reader, he is allowed to proceed, if not a waiting writer (if any) is freed.
- A major drawback of the solution is that it allows a group of readers, one of which always being active, to exclude the writers.
- A solution is to block readers as soon as a writer is waiting.

The readers-writers problem: The methods `startRead()` and `startWrite()` (2nd attempt)

```
public void startRead()
{ mutex.semWait();
    if ((writing) ||
       (okw.semNbWait() > 0))
    { mutex.semSignal();
      okr.semWait();
    }
    readers++;
    mutex.semSignal();
}
```

```
public void startWrite()
{ mutex.semWait();
  if ((readers != 0) ||
       writing)
  { mutex.semSignal();
    okw.semWait();
  }
  writing = true;
  mutex.semSignal();
}
```

The methods `endRead` and `endWrite` are not modified.

The readers-writers problem: Notes on the second attempt

- In this second version, all readers wishing to access the shared data are blocked as soon as a writer is waiting.
- The waiting writer will eventually have access to the shared data. Indeed, since no new reader is allowed to proceed, the number of readers will eventually reach 0.
- Once the writer has finished, he yields access to the first waiting reader (if any), or to the next writer if there is one.
- When there is high demand from both readers and writers, access will be alternately given to a writer and and single reader. This is not efficient since several readers could operate simultaneously.
- A better solution would be to free all waiting readers when a writer exits the critical section.

The readers-writers problem: The methods `startRead()` and `startWrite()`

```
public void startRead()
{ mutex.semWait();
  if ((writing) ||
       (okw.semNbWait() > 0))
    { mutex.semSignal();
      okr.semWait();
    }
  readers++;
  if (okr.semNbWait() > 0)
    okr.semSignal();
  else mutex.semSignal();
}
```

```
public void startWrite()
{ mutex.semWait();
  if ((readers != 0) ||
       writing)
  { mutex.semSignal();
    okw.semWait();
  }
  writing = true;
  mutex.semSignal();
}
```

The readers-writers problem:
The methods `endRead()` and `endWrite()`

```
public void endRead()
{ mutex.semWait();
  readers--;
  if ((readers == 0)
       && (okw.semNbWait() > 0))
    okw.semSignal();
  else mutex.semSignal();
}
```

```
public void endWrite()
{ mutex.semWait();
  writing = false ;
  if (okr.semNbWait() > 0)
    okr.semSignal();
  else
    if (okw.semNbWait() > 0)
      okw.semSignal();
    else mutex.semSignal();
}
```

The readers-writers problem: Notes on the final solution

- The only change with respect to the previous version can be found at the end of the method `startRead`.

- A *cascade wake up* is used: when a reader exits `startRead`, it wakes up the next reader, if one is waiting.

- The result is that, when a writer finishes, all waiting readers are freed.

- However, readers arriving after the reader has started executing `endWrite` have to wait on the semaphore `mutex` as long as the cascade wake up is not finished and will thus not be involved in it.

Communication through message passing

Communication through message passing: Motivation

- So far, *shared memory*, in one form or another, has been used as an interprocess communication mechanism.

- Communicating through shared memory is not adapted to situations in which processes do not share physical memory (machines connected through a network, parallel machines without shared memory).

- In these situations, interprocess communication through *message passing* is used.

Communication through message passing: Concept

- Communication through message passing is done with *communication channels* or *message queues*. A communication channel appears as a FIFO queue to which (from which) messages of a given type can be added (removed). A channel can be declared as follows:

  ```
  chan q type;
  ```

- There are then two possible operations on the communication channel.
  - `q!expr` which places the value of `expr` in the channel.
  - `q?var` which removes the head element of the channel and copies it to the variable `var`.

- Clearly, the types of the channel and of the expressions being placed on the channel (the variable receiving an element from the channel) must match. If this is not the case, the operation is not executed and there is an error.

- In some circumstances, it is useful to allow a receive operation `q?const` in which a constant rather than a variable is used.
  - No value is then received, but the operation is only possible if the value to be received is identical to the constant.
  - This amounts to testing the value at the head of the queue and to remove it only if it matches the expected value.

- It is not *a priori* necessary to impose a bound on channel capacity, but one will exist in practice in any implementation.

The producer-consumer problem in a message passing framework

With message passing, the solution to this problem is immediate.

```
                    chan q int;

         append(v)                    int take()
         int v                         { int v;
          { q!v;                          q?v;
          }                               return v;
                                       }
```

Implementing message queues

- In a shared memory machine, message queues can simply be implemented by an appropriate data structure in kernel memory, operation in the queues being implemented by system calls.

- In a distributed environment, communication goes through a network and relies on network communication protocols (such as TCP/IP).

- We will limit our study to message queues implemented by a data structure held in kernel memory.

- It is still necessary to specify what happens when a message is placed in a queue that is full, or read from a queue that is empty.

Full queues - empty queues

- When the queue to which one writes (from which one reads) is full (empty), the process executing the operation can
  - be suspended, or
  - receive an error message indicating that the operation has not been executed.
- Blocking the process has the advantage of allowing process synchronization, but is a constraint on the use of queues.
- Signaling that the operation is not possible leaves the responsibility of handling errors to the programmer.

An implementation of message queues in the context of ULg03

- The message queues are managed with a table kept in kernel memory. A queue is characterized by its number that corresponds to its position in this table.
- There are system calls for creating and releasing queues, as well for executing the operations `q!` et `q?`
- In ULg03, to execute for instance the operation `q!`, the arguments to the system call (the code `MSGSND`, the queue number, and the value to be sent) are placed on the stack. Assuming that the queue number is in `r0` and the value to be sent is in `r1`, this can be done as follows.

```
PUSH(r1)          | 3rd argument
PUSH(r0)          | 2nd argument
CMOVE(MSGSND,r2)  | 1st argument
PUSH(r2)
SVC()             | system call
```

The SVC handler

Remember that executing `SVC()` leads to the following stub.

```
h_stub:  ST(r0, User, r31)     | save
         ST(r1, User+4, r31)   | the registers
              . . .
         ST(r30, User+30*4, r31)
         CMOVE(KStack, SP)     | Load the system SP
         BR(Svc_handler,LP)    | Call to the Handler
         LD(r31, User, r0)     | restore
         LD(r31, User+4, r1)
         LD(r31, User+30*4, r30)
         JMP(XP)               | return to application
```

`Svc_handler` collects the arguments and calls the handler for the requested operation.

The handler for the MSGSND system call

In kernel memory, the (bounded) queues are kept in the following data structure.

```
struct Qdescr {int count, in, out = 0; int content[N];} Qtbl[K];
```

An the handler then is

```
Msgsnd_h(qno,v)
int qno,v;
{ if (Qtbl[qno].count < N)
   {Qtbl[qno].content[Qtbl[qno].in] = v;
    Qtbl[qno].in = (Qtbl[qno].in + 1) % N;
    Qtbl[qno].count++;
    return 1;
   }
  else return 0;
}
```

The returned value, which can be found in `r0` by the process having executed the call, indicates whether the sent value could be placed in the queue or not.

A blocking implementation ?

- If a blocking implementation is desired, one could use the semaphore-based implementation of a buffer studied previously.
- But, can systems calls be used in supervisor mode?
- Not in the context we have seen so far, because what is done when a system call occurs does not make sense if the kernel is already running.
- It is however possible to directly call the corresponding handler. However, if a process is suspended upon a system call, it is only the process state that is preserved, the state of the handler is not.
- One must thus make sure that everything is correct if, when a process is reactivated, the system call is re-executed from the beginning.

Blocking queues of capacity 0: **the** *rendez-vous*

- A special case of blocking queues are queues of capacity 0.
- In this case, the process placing an element in the queue is always blocked until another process takes this element from the queue.
- Conversely, a process receiving an element from a queue is always blocked until a process places an element in the queue.
- The operations `q!` and `q?` executed by the processes are thus synchronized: the first that is ready to execute its operation waits for the other. This is called a *rendez-vous*

Implementing *rendez-vous*

To implement *rendez-vous*, it is sufficient to adapt the implementation of a buffer as follows.

```
/* mémoire partagée */
int buf;
semaphore placed = 0;
semaphore taken = 0;
```

```
msgsnd(v)
int v
 { buf = v;
   signal(placed);
   wait(taken);
  }
```

```
int msgrcv()
 { int v;
   wait(placed);
   v = buf;
   signal(taken);
   return v;
 }
```

In `msgsnd(v)`, signaling that the element is placed must be done before waiting for it to be taken; if not there would be a deadlock.

Programming with *rendez-vous*

- Since with a *rendez-vous*, the process waiting to receive an element is blocked, if is necessary to be able to simultaneously wait on several queues, for example used by different processes; or to wait for different types of messages on the same queue.

- Languages in which *rendez-vous* is used (CSP, OCCAM, ADA) include a special instruction for this.

- This instruction also allows accepting a message from a queue only if a given condition is satisfied.

An instruction for waiting on several *rendez-vous*

We will use the instruction below, which is inspired by ADA, but written with a C-style syntax.

```
select
{ when (cond1) q1?m1 ; statement1;
  when (cond2) q2?m2 ; statement2;
    ........
  when (condk) qk?mk ; statementk;
  else defaultstatement;
}
```

This instruction is interpreted as follows.

- First determine for which alternatives the condition is true and the input operation executable (i.e. there is a process ready to execute the corresponding send operation).

- For one of these alternatives, the receive operation and the following statement are executed.

- If there is no alternative for which this is possible,
  - wait for a send operation on one of the queues in an alternative for which the condition is true; or
  - execute the `else` clause if it is present.

Simulating a semaphore using *rendez-vous*

```
#define wait = 1
#define signal = 0

semsimul(inival)
  int inival;
  { int value;
    value = inival
    while (true)
    select
    { when (value > 0) q?wait ; value--;
      when (true) q?signal ; value++;
    }
  }
```

Mutual exclusion with the simulated semaphore

```
chan q int [0];
```

Process 1 :

```
while (True)
{   nc1: /* non critical
              section */ ;
          q!wait;
  crit1: /* critical section */ ;
          q!signal;
}
```

Process 2 :

```
while (True)
{   nc2: /* non critical
              section */
          q!wait;
  crit2: /* critical section */
          q!signal;
}
```

Processus 0 :

```
semsimul(1);
```

In this setting, the semaphore is embodied by a process, not just a shared object.