

# Synchronization

Daniel Hagimont (INPT)

Thanks to Noël De Palma, Fabienne Boyer and  
Arnaud Legrand (UJF)

# Problem statement

(1)  $y := \text{read\_account}(1);$

(2)  $x := \text{read\_account}(2);$

(3)  $x := x + y;$

(4)  $\text{write\_account}(2, x);$

(5)  $\text{write\_account}(1, 0);$

(a)  $v := \text{read\_account}(2);$

(b)  $v = v - 100;$

(c)  $\text{write\_account}(2, v);$

- Account 2 is shared between both executions
- Variables  $x, y, v$  are local
- Executions are performed in parallel and instructions can be intertwined
- (1) (2) (3) (4) (5) (a) (b) (c) is consistent (200/200)(0,300)
- (1) (a) (b) (c) (2) (3) (4) (5) is consistent (200/200)(0,300)
- (1) (2) (a) (3) (b) (4) (c) (5) is not consistent (200/200)(0,100)

# Problem statement

- N processes all competing to use some shared data
  - A critical section is a code fragment, in which the shared data is accessed
- Problem:
  - Ensure shared data consistency
- Ensure mutual exclusion
  - When one process is executing in one critical section, no other process is allowed to execute in this critical section

# Desired properties

- Mutual Exclusion
  - Only one thread can be in a given critical section at a time
- Progress
  - If no process currently in a given critical section, one of the processes trying to enter will eventually get in
- Fairness
- No starvation

# Critical section

- n processes:  $P_0, P_1, \dots, P_n$
- $P_0, P_1, \dots, P_n$  use a set of shared variables  $a, b, c, \dots$
- Structure of a process  $P_i$  :

```
...  
<enter section> // enter mutex  
<access a,b,c,..> // critical section  
<exit section> // leave mutex  
...
```

# Software implementation (1)

Shared data :

```
boolean busy = false;
```

P<sub>i</sub> :

```
while (busy) ;    (1)    // busy waiting  
busy = true;  
<critical section>  
busy = false;
```

- No mutual exclusion if context switch at (1)
  - Test and set are not atomic

# Software implementation (2)

Shared data :

```
int turn = 0; // turn = i : Pi's turn to enter
```

P<sub>i</sub> (0 or 1):

```
while (turn != i); // busy waiting  
<critical section>  
turn = 1-i;
```

- Mutual exclusion
- Can be generalized to N processes
- Progress issue

# Software implementation (3)

Shared data :

```
boolean demand[2] = {false, false}; // Pi ask to enter
```

Pi (0 or 1):

```
demand[i] = true;  
while (demand[1-i]) ;           // busy waiting  
<critical section>  
demand[i] = false;
```

- Mutual exclusion
- Difficult to generalize to N processes
- Can block (deadlock)



# Software implementation (4)

## Peterson algorithm

Shared data :

```
int turn = 0;    // Pi's turn
boolean demand[2] = {false, false}; // Pi ask to enter
```

Pi (0 or 1):

```
demand[i] = true;
turn = 1-i;
while (demand[1-i] && (turn = 1-i)) ;
<critical section>
demand[i] = false;
```

- Difficult to generalize to N processes

# Software implementation (5)

## Lamport algorithm

Shared data :

```
boolean choice[N] = {false, ... false};  
int num[N] = {0, ... 0};
```

P<sub>i</sub>:

```
choice[i] = true;  
int turn = 0;  
for (int k=0;k<N;k++) turn = max(turn, num[k]);  
num[i] = turn + 1;  
choice[i] = false;  
  
for (int k=0;k<N;k++) {  
    while (choice[k]);  
    while ((num[k] != 0) && ((num[k],k) < (num[i],i)));  
}  
<critical section>  
num[i] = 0;
```

# Synthesis

- Software solutions
  - Complex
  - Not very efficient
- Hardware solutions
  - Masking interrupts
  - Test&Set

# Masking interrupts

- Entry section : mask the IT
- Exit section : unmask the IT
- Cannot control the time spent in critical section
- Acceptable if the critical section exec time is short
- Cannot be used with multiprocessors

# Test&Set instruction

- Most CPUs support atomic read-[modify-]write
- Example: `int test_and_set (int *lockp);`
  - Atomically sets `*lockp = 1` and returns old value

```
int Test&Set (int *b) {  
    // set b to 1, and return initial value of b  
    int res = *b;  
    *b = 1;  
    return res;  
}
```

# Test&Set critical section

Shared data :

```
int busy = 0; // false
```

Pi:

```
while (Test&Set (&busy));  
<critical section>  
busy = 0;
```

- Busy waiting
- Starvation issue (not FIFO)

# Sleep and wake up solutions

- Previous solution disadvantage :
  - CPU wasting (polling)
- Sleep and wake up solutions :
  - Block a process when it cannot enter a critical section
  - Wake up when the critical section is free
- Different abstractions
  - Lock
  - Semaphore
  - Monitor

# Locks

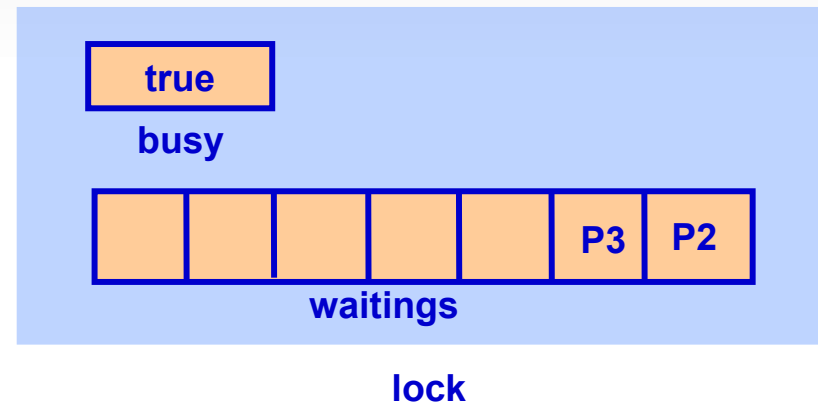
- Simple synchronization primitives
  - Lock/unlock function
  - Only one process can go through a lock at the same time
  - Based on sleep/wakeup
- Different interfaces, implementations, properties (fifo ...)
  - e.g. Thread packages :
    - `void mutex_init (mutex_t *m, ...);`
    - `void mutex_lock (mutex_t *m);`
    - `int mutex_trylock (mutex_t *m);`
    - `void mutex_unlock (mutex_t *m);`



# Lock implementation (1/2)

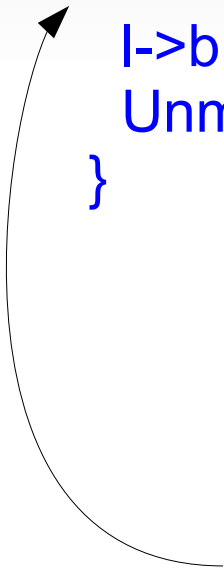
```
typedef struct {  
    int busy; // the lock taken or free  
    proc_ctxt_list waitings; // waiting processes  
} lock;
```

```
void InitLock(lock *l) {  
    Mask();  
    l->busy = false;  
    initList(&(l->waitings));  
    Unmask();  
}
```



# Lock implementation (2/2)

```
void lock(lock *l) {  
    Mask();  
    while (l->busy)  
        Suspend(&(l->waitings));  
    l->busy = true;  
    Unmask();  
}
```



```
void unlock(lock *l) {  
    proc_ctxt waked;  
    Mask();  
    l->busy = false;  
    if (waked = GetFirst((&(l->waitings))))  
        Wakeup(waked);  
    Unmask();  
}
```

- Why this loop ?
- Starvation risk

# Suspend and Wakeup implementation

Shared data:

```
proc_ctxt actif;
```

```
void Suspend( proc_ctxt_list *queue) {  
    proc_ctxt *old = actif;  
    putLast(queue, actif);  
    actif = get(ReadyQueue);  
    ctxtSwap(actif, old);  
}
```

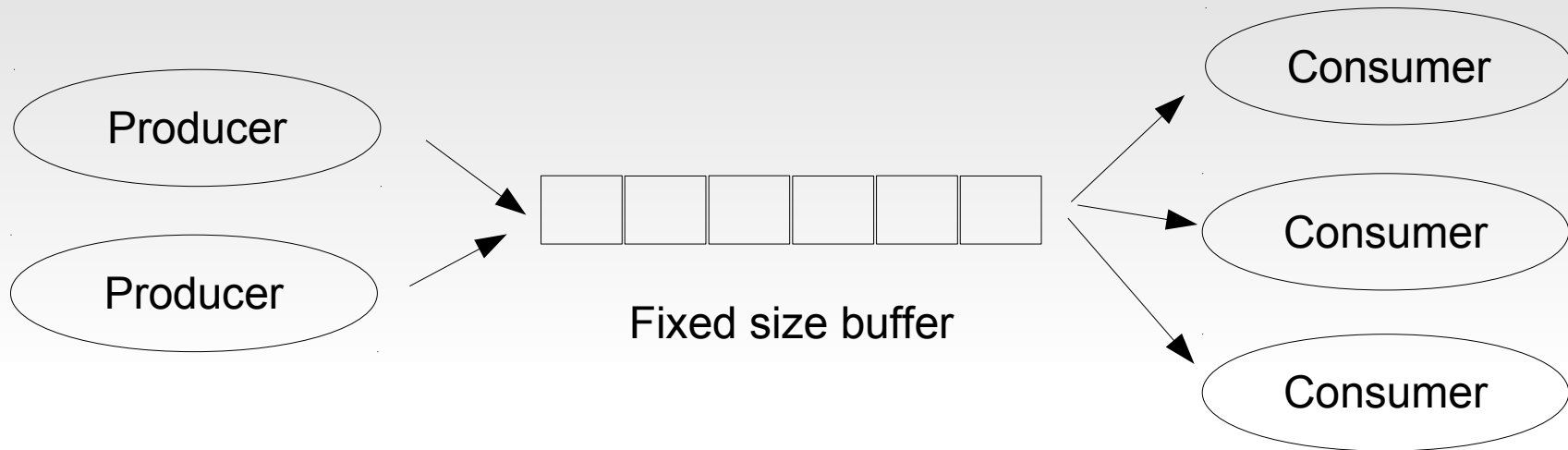
```
void WakeUp(proc_ctxt *p) {  
    put(ReadyQueue, p);  
}
```

# Lock implementation (FIFO)

```
void lock(lock *l) {  
    Mask();  
    → if (l->busy)  
        Suspend(&(l->waitings));  
    l->busy = true;  
    Unmask();  
}
```

```
void unlock(lock *l) {  
    proc_ctxt waked;  
    Mask();  
    if (waked = GetFirst((&(l->waitings))))  
        Wakeup(waked);  
    → else l->busy = false;  
    Unmask();  
}
```

# Producer Consumer example



- Fixed size buffer
- Variable number of producers and consumers

# Producer Consumer example

Shared data:

```
int bufferSize = N;  
int in = 0, out = 0, nb = 0;  
Msg buffer[] = new Msg[bufferSz];
```

```
produce (Msg msg) {  
    buffer[in] = msg;  
    in = in + 1 % bufferSize;  
    nb++;  
}
```

```
Msg Consume {  
    Msg msg = buffer[out];  
    out = out + 1 % bufferSize;  
    nb--;  
    return msg;  
}
```

# Producer Consumer with locks

## locks are not sufficient

Shared data:

```
int bufferSz = N;  
int in = 0, out = 0, nb = 0;  
Msg buffer[] = new Msg[bufferSz];  
Lock mutex = new Lock();
```

```
produce (Msg msg) {  
    mutex.lock();  
    while (nb == bufferSz) {  
        mutex.unlock();  
        yield(); // sleep  
        mutex.lock();  
    }  
    buffer[in] = msg;  
    in = in + 1 % bufferSz;  
    nb++;  
    mutex.unlock();  
}
```

busy  
waiting {

```
Msg Consume {  
    mutex.lock();  
    while (nb == 0) {  
        mutex.unlock();  
        yield(); // sleep  
        mutex.lock();  
    }  
    Msg msg = buffer[out];  
    out = out + 1 % bufferSz;  
    nb--;  
    mutex.unlock();  
    return msg;  
}
```

# Higher synchronization abstractions

- Principles
  - Use application's semantic to suspend/wake up a process that wait for a condition to happen
- Examples
  - Semaphore
  - Monitor



# Semaphores (Dijkstra, 1965)

- Semaphore S :
  - counter S.c; //Model a resource number or a condition
  - waiting queue S.f; //Waiting processes
- Think of a semaphore as a purse with a certain number of tokens
  - Suspend when no more token
  - Wake up when token released
- A Semaphore is initialized with an integer N
- Accessed through P() and V() operations

# Semaphores (Dijkstra, 1965)

wait() ou P ():

```
S.c--  
if S.c < 0 do {  
    // no more free resources  
    put(myself, S.f);  
    suspend(); // suspension  
}
```

signal() ou V ():

```
S.c++;  
if (S.c <= 0) do {  
    // at least 1 waiting process  
    P = get(S.f);  
    wakeup(p);  
}
```

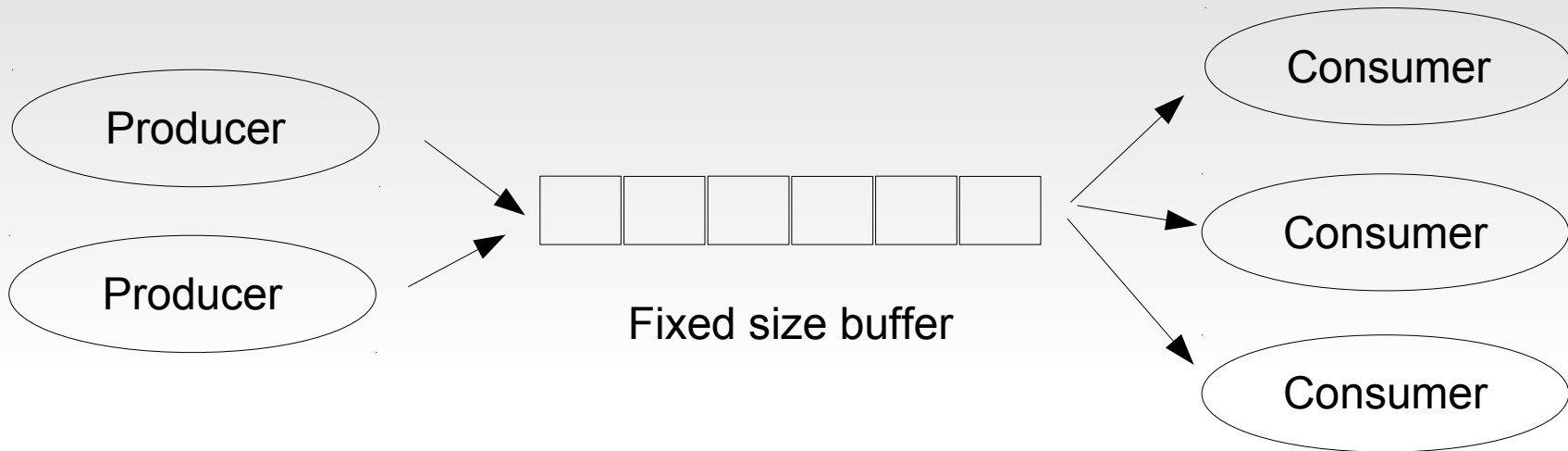


Remember real implementations require lower level locking (e.g. interruption management)

# Semaphores

- Counter  $S.c == S.c \text{ initial} + NV - NP$ 
  - $NV$  is the number of  $V$  operations executed on the semaphore
  - $NP$  is the number of  $P$  operations executed on the semaphore
- Counter  $S.c < 0$ 
  - Correspond to the number of blocked processes
- Counter  $S.c > 0$ 
  - Correspond to the number of available resources
  - Correspond also to the number of processes that can call a  $P$  operation without being blocked
- Counter  $S.c == 0$  :
  - No more resources available and no blocked process
  - The next process that call  $P()$  will be blocked
- Can use semaphores to implement mutual exclusion ( $\text{init} = 1$ )
- Could use semaphores to implement conditions

# Producer Consumer



- Condition to produce/consume
  - Produce: the buffer is not full
  - Consume: the buffer is not empty

# Producer Consumer

- Can write producer/consumer with three semaphores
- Semaphore mutex initialized to 1
  - Used as mutex, protects buffer, in, out. . .
- Semaphore notempty initialized to 0 ( $\approx$  number of items)
  - To block consumer when buffer is empty
- Semaphore notfull initialized to N ( $\approx$  number of free locations)
  - To block producer when queue is full

# Producer Consumer

## Shared data:

```
int bufferSz = N;  
int in = 0, out = 0;  
Msg buffer[] = new Msg[bufferSz];  
Semaphore notFull = new Semaphore(bufferSz);  
Semaphore notEmpty = new Semaphore(0);  
Semaphore mutex = new Semaphore(1);
```

```
produce (Msg msg) {  
    notFull.P();  
    mutex.P();  
    buffer[in] = msg;  
    in = in + 1 % bufferSz;  
    mutex.V();  
    notEmpty.V();  
}
```

```
Msg Consume {  
    notEmpty.P();  
    mutex.P();  
    Msg msg = buffer[out];  
    out = out + 1 % bufferSz;  
    mutex.V();  
    notFull.V();  
    return msg;  
}
```

# Producer Consumer (improve parallelism)

Shared data:

```
int bufferSz = N;  
int in = 0, out = 0;  
Msg buffer[] = new Msg[bufferSz];  
Semaphore notFull = new Semaphore(bufferSz);  
Semaphore notEmpty = new Semaphore(0);  
Semaphore mutexIn = new Semaphore(1);  
Semaphore mutexOut = new Semaphore(1);
```

```
produce (Msg msg) {  
    notFull.P();  
    mutexIn.P();  
    buffer[in] = msg;  
    in = in + 1 % bufferSz;  
    mutexIn.V();  
    notEmpty.V();  
}
```

```
Msg Consume {  
    notEmpty.P();  
    mutexOut.P();  
    Msg msg = buffer[out];  
    out = out + 1 % bufferSz;  
    mutexOut.V();  
    notFull.V();  
    return msg;  
}
```

# Thread and Semaphore

- Thread packages typically provide semaphores.
  - `int sem_init(sem_t *sem, int pshared, unsigned int value);`
  - `int sem_post(sem_t *sem);`
  - `int sem_wait(sem_t *sem);`
  - `int sem_trywait(sem_t *sem);`
  - `int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);`
  - `int sem_getvalue(sem_t *sem, int *sval);`



# Semaphore conclusion

- They are quite error prone
  - If you call P instead of V, you'll have a deadlock
  - If you forget to protect parts of your code, you end up with a mutual exclusion violation
  - If you have “tokens” of different types, it may be hard to reason about
  - If by mistake you interchange the order of the P and V, you may violate mutual exclusion or end up with a deadlock.
- That is why people have proposed higher-level language constructs

# Deadlock ??

- A correct solution is not always ensured by the semaphore :

```
P(mutex);  
if ...  
    P(S);  
    ...  
else  
    ...  
    V(S);  
V(mutex);
```

← Possible deadlock

**RULE:** never block in a critical section without releasing the section

# Monitor

- Programming language construct
- A Monitor contains
  - Data
  - Function (f1,...,fn)
  - Init function
  - Conditions
- Functions are executed in mutual exclusion
- A “condition variable” is a synchronization structure (a queue) associated to a “logical condition”
  - wait() suspends the caller
  - signal() wakes up a waiting process if any, else the signal is LOST
- In general, condition queues are FIFO

# Monitor

```
monitor <monitor-name> {  
    <shared variables + conditions declarations>  
  
    procedure init { initialization code }  
  
    procedure f1 (...) {  
        ...  
    }  
    procedure f2 (...) {  
        ...  
    }  
    procedure Pn (...) {  
        ...  
    }  
}
```

# Monitor

- Only one process is running inside the monitor at a time
- On a signal
  - Either the signal sender keep the monitor (signal sender priority) = Signal and continue
  - Or the signal receiver acquires the monitor (signal receiver priority) = Signal and wait
- Monitor release
  - When the current procedure completes
  - When calling a wait operation

# Producer Consumer with monitors

```
Monitor ProdConsMonitor {
  int bufferSz, nb, in, out;
  Msg buffer[];
  Condition notEmpty, notFull;

  procedure init() {
    bufferSz = N;
    nb = in = out = 0;
    buffer = new Msg[bufferSz];
  }
}
```

- Signal receiver priority

```
procedure produce(Msg msg) {
  if (nb==bufferSz)
    notFull.wait();
  buffer[in] = msg;
  in = in + 1 % bufferSz;
  nb++;
  notEmpty.signal();
}
```

```
procedure consume() : Msg {
  if (nb==0)
    notEmpty.wait();
  Msg msg = buffer[out];
  out = out + 1 % bufferSz;
  nb--;
  notFull.signal();
}
```

# Producer Consumer with monitors

```
Monitor ProdConsMonitor {
  int bufferSz, nb, in, out;
  Msg buffer[];
  Condition notEmpty, notFull;

  procedure init() {
    bufferSz = N;
    nb = in = out = 0;
    buffer = new Msg[bufferSz];
  }
}
```

- Signal sender priority

```
procedure produce(Msg msg) {
  while (nb==bufferSz)
    notFull.wait();
  buffer[in] = msg;
  in = in + 1 % bufferSz;
  nb++;
  notEmpty.signal();
}

procedure consume() : Msg {
  while (nb==0)
    notEmpty.wait();
  Msg msg = buffer[out];
  out = out + 1 % bufferSz;
  nb--;
  notFull.signal();
}
```

# pthread synchronization

- `pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;`
- `int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutex_attr *attr);`
- `int pthread_mutex_destroy (pthread_mutex_t *m);`
- `int pthread_mutex_lock (pthread_mutex_t *m);`
- `int pthread_mutex_trylock (pthread_mutex_t *m);`
- `int pthread_mutex_unlock (pthread_mutex_t *m);`



# pthread synchronization

- `pthread_cond_t vc = PTHREAD_COND_INITIALIZER;`
- `int pthread_cond_init (pthread_cond_t *vc, const pthread_cond_attr *attr);`
- `int pthread_cond_destroy (pthread_cond_t *vc);`
- `int pthread_cond_wait (pthread_cond_t*, pthread_mutex_t*);`
- `int pthread_cond_timedwait (pthread_cond_t*, pthread_mutex_t*, const struct timespec *abstime);`
- `int pthread_cond_signal (pthread_cond_t *vc);`
- `int pthread_cond_broadcast (pthread_cond_t *vc);`

# Java synchronization

- For each object
  - a lock
  - a condition
- Monitor principles
  - Synchronized methods = executed in mutual exclusion
  - wait and notify/notifyAll to manage conditions

```
class Example {  
  
    int cpt; // shared data  
  
    public void synchronized get() {  
        if (cpt <= 0) wait();  
        cpt--;  
    }  
    public void synchronized put() {  
        cpt++;  
        notify();  
    }  
}
```

# Java synchronization

- Synchronize a chunk of code

```
synchronized (oneObj) {  
    < critical section >  
}
```

- Also one lock per class

```
class X {  
    static synchronized T foo() { ... }  
}
```

# Exercise

## reader/writer with semaphores

- A shared document
- Users can read/write the document

beginRead()  
    < reading >  
endRead()

beginWrite()  
    < writing >  
endWrite()

- Multiple readers / single writer

# Exercise

## reader/writer with semaphores

Shared data:

```
int nbReaders = 0;  
Semaphore reader = new Semaphore(1);  
Semaphore writer = new Semaphore(1);
```

```
beginRead () {  
    reader.P();  
    nbReaders ++;  
    If (nbReaders == 1)  
        writer.P();  
    reader.V();  
}
```

```
endRead () {  
    reader.P();  
    nbReaders --;  
    If (nbReaders == 0)  
        writer.V();  
    reader.V();  
}
```

```
beginWrite () {  
    writer.P();  
}
```

```
endWrite () {  
    writer.V();  
}
```

- Potential starvation of writers

# Exercise

## reader/writer with semaphores (and fairness)

Shared data:

```
int nbReaders = 0;  
Semaphore reader = new Semaphore(1);  
Semaphore writer = new Semaphore(1);  
Semaphore fifo = new Semaphore(1);
```

```
beginRead () {  
    fifo.P();  
    reader.P();  
    nbReaders ++;  
    if (nbReaders == 1)  
        writer.P();  
    reader.V();  
    fifo.V();  
}  
  
beginWrite () {  
    fifo.P();  
    writer.P();  
    fifo.V();  
}  
  
endRead () {  
    reader.P();  
    nbReaders --;  
    if (nbReaders == 0)  
        writer.V();  
    reader.V();  
}  
  
endWrite () {  
    writer.V();  
}
```

# Exercise

## reader/writer with monitors


```
monitor ReaderWriter () {  
    int nbReaders;  
    boolean writer;  
    Condition canRead, canWrite;
```

```
    procedure init() {  
        nbReaders = 0;  
        writer = false;  
    }  
}
```

```
    procedure beginRead() {  
        if (writer) canRead.wait();  
        nbReader++;  
        canRead.signal();  
    }  
}
```

```
    procedure endRead() {  
        nbReader--;  
        if (nbReaders == 0) canWrite.signal();  
    }  
}
```

```
    procedure beginWrite() {  
        if ((nbReaders > 0) || (writer))  
            canWrite.wait();  
        writer = true;  
    }  
}
```

```
    procedure endWrite() {  
        writer = false;  
        canRead.signal();   
        if (nbReaders == 0) canWrite.signal();  
    }  
}
```

- Priority to signal receiver

- Priority to readers

# Exercise

## reader/writer with monitors

```
monitor ReaderWriter () {  
    int nbReaders;  
    boolean writer;  
    Condition canRead, canWrite;
```

```
    procedure init() {  
        nbReaders = 0;  
        writer = false;  
    }  
}
```

```
    procedure beginRead() {  
        if (writer) canRead.wait();  
        nbReader++;  
        canRead.signal();  
    }  
}
```

```
    procedure endRead() {  
        nbReader--;  
        if (nbReaders == 0) canWrite.signal();  
    }  
}
```

```
    procedure beginWrite() {  
        if ((nbReaders > 0) || (writer))  
            canWrite.wait();  
        writer = true;  
    }  
}
```

```
    procedure endWrite() {  
        writer = false;  
        canWrite.signal();  
        If (!write) canRead.signal();  
    }  
}
```

- Priority to writers



# Exercise

## reader/writer with monitors

```
monitor ReaderWriter () {  
    int nbReaders;  
    boolean writer;  
    Condition queue, nextWriter;
```

```
    procedure init() {  
        nbReaders = 0;  
        writer = false;  
    }
```

```
    procedure beginRead() {  
        if ((writer) || (!queue.empty())  
            || (!nextWriter.empty()))  
            queue.wait();  
        nbReader++;  
        queue.signal();  
    }
```

```
    procedure endRead() {  
        nbReader--;  
        if (nbReaders == 0) {  
            if (nextWriter.empty()) queue.signal();  
            else nextWriter.signal();  
        }
```

```
    procedure beginWrite() {  
        if ((nbReaders > 0) || (writer))  
            queue.wait();  
        if (nbReaders > 0) nextWriter.wait();  
        writer = true;  
    }
```

```
    procedure endWrite() {  
        writer = false;  
        queue.signal();  
    }
```

- FIFO

# Exercise

## semaphore with monitor

```
monitor Semaphore () {  
    int count;  
    Condition positive;  
  
    procedure init(int v0) {  
        count = v0;  
    }  
  
    procedure P() {  
        if (count == 0) positive.wait();  
        count--;  
    }  
  
    procedure V() {  
        count++;  
        positive.signal();  
    }  
}
```

# Exercise

## monitor with semaphore

### Shared data:

```
Semaphore mutex = new Semaphore(1);    // mutex of the monitor  
Semaphore cond = new Semaphore(0);    // one per condition in the monitor
```

### monitor\_procedure

```
mutex.P();  
< procedure body >  
mutex.V();
```

### cond.wait in a procedure

```
mutex.V();  
cond.P();  
mutex.P();
```

### cond.signal in a procedure

```
if (!cond.empty()) {  
    cond.V();  
}
```

- Priority to the sender

# Exercise

## monitor with semaphore

### Shared data:

```
Semaphore mutex = new Semaphore(1);    // mutex of the monitor
Semaphore waitings = new Semaphore(0);  // to block signal senders
Semaphore cond = new Semaphore(0);     // one per condition in the monitor
```

### monitor\_procedure

```
mutex.P();
< procedure body >
if (!waitings.empty())
    waitings.V();
else mutex.V();
```

### cond.wait in a procedure

```
if (!waitings.empty())
    waitings.V();
else mutex.V();
cond.P();
```

### cond.signal in a procedure

```
if (!cond.empty()) {
    cond.V();
    waitings.P();
}
```

- Priority to the receiver