Synchronization

Daniel Hagimont (INPT)

hagimont@enseeiht.fr

http://hagimont.perso.enseeiht.fr

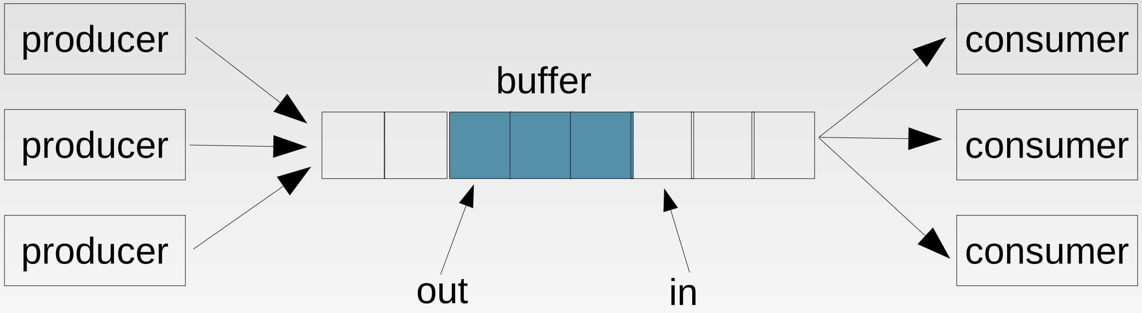
Problem statement

(1) y := read_account(1);
(2) x := read_account(2);
(3) x := x + y;
(4) write_account(2, x);
(5) write_account(1, 0);

(a) v := read_account(2);
(b) v = v - 100;
(c) 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




```
#define BUFFER_SIZE 10

typedef struct {
	char product;
	int amount;
} item;

item buffer [BUFFER_SIZE];
int in = 0; // where to produce
int  out = 0; // where to consume
int nb = 0; // number of items
```

```
void produce(item *i) {
	while (nb == BUFFER_SIZE) {
		// do nothing – no free place in buffer
	}
	memcopy(&buffer[in], i, sizeof(item));
	in = (in+1) % BUFFER_SIZE;
}
```

```
item *consume() {
	item *i = malloc(sizeof(item));
	while (nb == 0) {
		// do nothing – nothing to consume
	}
	memcopy(i, &buffer[out], sizeof(item));
	out = (out+1) % BUFFER_SIZE;
	return i;
}
```

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: P0, P1, .., Pn
- P0, P1, .., Pn use a set of shared variables a, b, c, …

- Structure of a process Pi :

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

Software implementation (1)

Shared data :

```
boolean busy = false;
```

Pi :

```
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
```

Pi (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};
```

Too complex !

Pi:

```
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

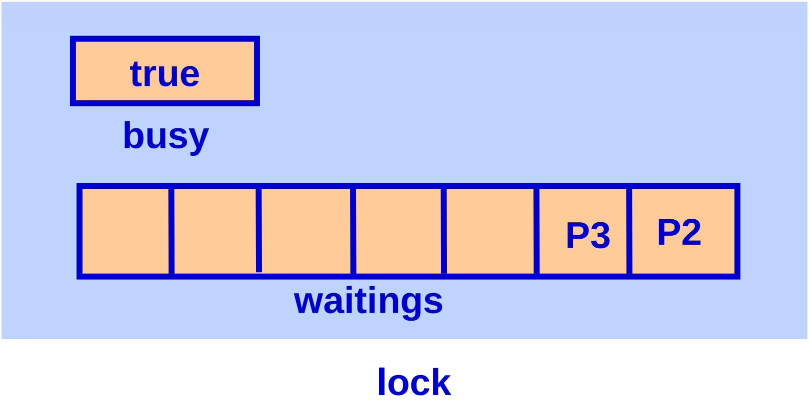
- Previous solution disadvantage :
  - CPU wasting (polling)
- Sleep and wake up solutions :
  - Block a process when it cannot enter a critical section
  - Wake up when the critical section is free
- Different abstractions
  - Lock
  - Semaphore
  - Monitor

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)

Why this loop ?

```
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();
}
```

Not sure which process's going to be waken up

Shared data:

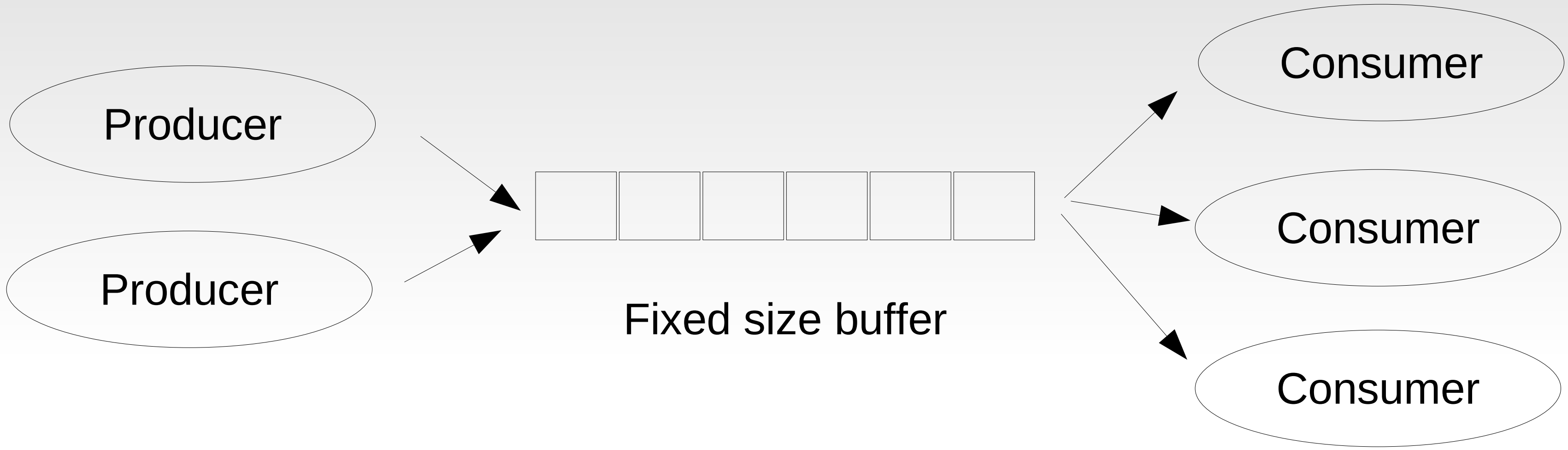
```
        proc_ctxt actif;

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

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

```
Msg Consume {
    Msg msg = buffer[out];
    out = out + 1 % bufferSz;
    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) {

```
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: mutex.unlock(); yield(); // sleep; mutex.lock();

Msg Consume {

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

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

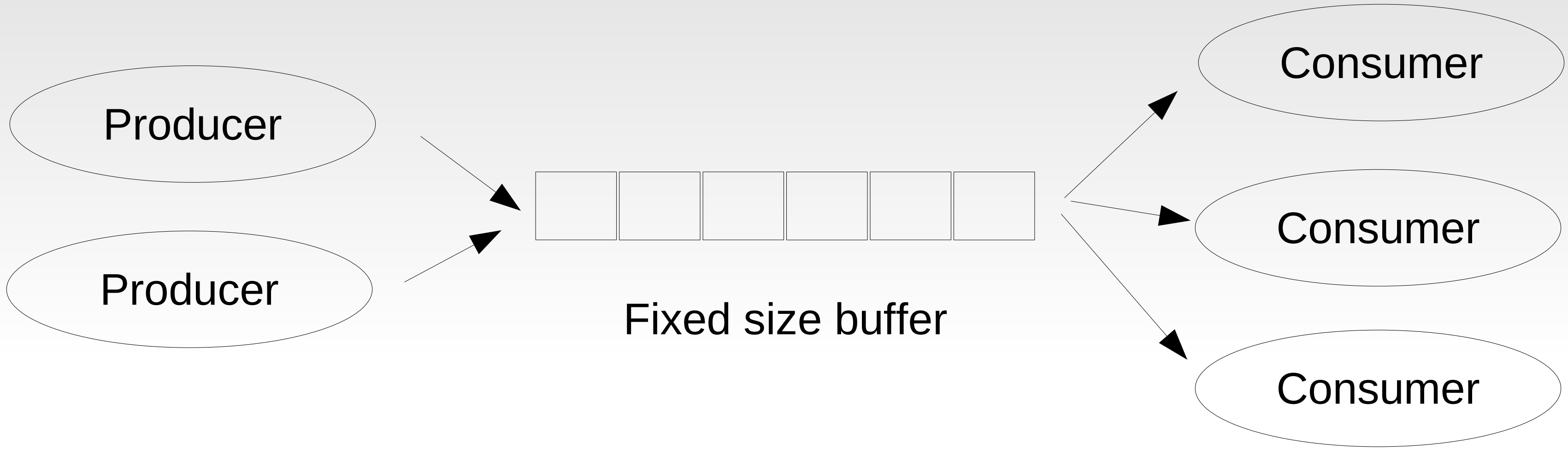
Critical section

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

Semaphores

- Counter S.c == S.c 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 beeing 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 (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 products initialized to 0 ($\approx$ number of items)
  - To block consumer when buffer is empty
- Semaphore places 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 places = new Semaphore(bufferSz);
Semaphore products = new Semaphore(0);
Semaphore mutex = new Semaphore(1);
```

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

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

Producer Consumer (improve parallelism)

Shared data:

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

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

```
Msg Consume {
    products.P();
    mutexOut.P();
    Msg msg = buffer[out];
    out = out + 1 % bufferSz;
    mutexOut.V();
    places.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 places, products;

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

- Signal <u>receiver</u> priority

```
procedure produce(Msg msg) {

  if (nb==bufferSz)
     places.wait();
  buffer[in] = msg;
  in = in + 1 % bufferSz;
  nb++;
  products.signal();
}

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

Producer Consumer with monitors

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

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

- Signal <u>sender</u> priority

```
procedure produce(Msg msg) {

  while (nb==bufferSz)
     places.wait();
  buffer[in] = msg;
  in = in + 1 % bufferSz;
  nb++;
  products.signal();
}

procedure consume() : Msg {
  while (nb==0)
     products.wait();
  Msg msg = buffer[out];
  out = out + 1 % bufferSz;
  nb--;
  places.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 *vc, pthread_mutex_t *m);
- int pthread_cond_timedwait (pthread_cond_t *vc, pthread_mutex_t *m, 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
  - one lock
  - one condition
- Monitor principles
  - Synchronized methods = executed in mutual exclusion
  - wait and notify/notifyAll to manage the condition

```
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()                 beginWrite()
    < reading >                 < writing >
endRead()                   endWrite()
```

- Multiple readers / single writer

Exercise
reader/writer with semaphores

Shared data:

```
int nbReaders = 0;
Semaphore mutex = new Semaphore(1);
Semaphore exclusive = new Semaphore(1);
```

```
beginRead () {
        mutex.P();
        If (nbReaders == 0)
            exclusive.P();
        nbReaders ++;
        mutex.V();
}
```

```
endRead () {
        mutex.P();
        nbReaders --;
        If (nbReaders == 0)
            exclusive.V();
        mutex.V();
}
```

```
beginWrite () {

exclusive.P();
}
```

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

- Potential starvation of writers

Exercise
reader/writer with semaphores
(and fairness)

Shared data:

```
int nbReaders = 0;
Semaphore mutex = new Semaphore(1);
Semaphore exclusive = new Semaphore(1);
Semaphore fifo = new Semaphore(1);
```

```
beginRead () {
        fifo.P();
        mutex.P();
        if (nbReaders == 0)
            exclusive.P();
        nbReaders ++;
        mutex.V();
        fifo.V();
}
```

```
endRead () {
        mutex.P();
        nbReaders --;
        if (nbReaders == 0)
            exclusive.V();
        mutex.V();
}
```

```
beginWrite () {
        fifo.P();
        exclusive.P();
        fifo.V();
}
```

```
endWrite () {
        exclusive.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;
   if (! canRead.empty())
            canRead.signal();
   else canWrite.signal();
}
```

- Priority to signal receiver
- Priority to readers

Exercise
semaphore with monitor

```
monitor Semaphore () {
   int count;
   Condition positive;

procedure init(int v0) {
   count = v0;
}

procedure P() {
   count--;
   if (count < 0) positive.wait();
}

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

Resources you can read

- Operating System Concepts, 10th Edition, Abraham Silberschatz, Peter B. Galvin, Greg Gagne
  - http://os-book.com/
  - Chapters 6, 7
- Modern Operating Systems, Andrew Tanenbaum
  - http://www.cs.vu.nl/~ast/books/mos2/
  - Chapter 2 (2.3 & 2.4)