Concurrency Programming

Laurent Pautet

Laurent.Pautet@enst.fr

Version EN, 1.0

Plan

- **Lightweight processes**
- POSIX Library
- Java Language
- Design Patterns

Lightweight Processes
Motivations and approaches

- An activity often consists of a sequence of actions:
  - Receipt of data (values, events, etc)
  - Processing of data received (computation / conversion, etc.)
  - Emission of output data (values, events, etc.)
- All activity consumes shared resources like CPU or data
- Two approaches to performing these activities
  - Serialized processing: statically interleaving the activities
    - Everything must be planned to serialize the execution sequence
  - Concurrent processing: temporal sharing of resources (CPU)
    - It is necessary to be able to suspend and resume an activity

Lightweight Processes
Benefits and Drawbacks

- Cost of sequential programming
  - Very deterministic because we know which task is being executed at a given time
  - Not very adaptive: plan the sequences in advance
  - Poor performance: a blocked task blocks ready tasks
- Cost of concurrency programming
  - Possibly preemptive scheduling to determine and execute the ready activities
- Concurrent systems programming:
  - Coordinate activities sharing their resources
- Distributed systems programming:
  - Coordinate activities without sharing resources

Lightweight Processes
Characteristics of heavyweight processes

- Classic heavyweight Unix processes
  - Copy or clone of the parent process
  - No memory sharing
  - Expensive data sharing through I/O
- Basic functionality
  - fork, exec, exit, wait
  - Costly synchronization by I/O
- No light OS implementation possible
  - Memory protected between processes by MMU
  - Costly change of context (cache, MMU, etc.)

Lightweight Processes
Characteristics of lightweight processes

- Lightweight processes (threads) :
  - Share a common memory
  - Have enriched features such as synchronization tools
  - May result in a light implementation
- Lightweight processes do not obsolete classic heavyweight processes that provide
  - Spatial isolation (address spaces)
  - Time isolation (hierarchical schedulers)

Lightweight Processes
Architecture

- A program is made up of global variables and functions in particular the *main* function
- A heavy process starts with an initial lightweight process which executes the *main* function
- The heavy process (or its initial light process) can create other light processes whose main function signature is similar to the *main* function one
- The initial process and those created later run in parallel and share resources (including global variables) within the heavy process

Lightweight Processes
Composition of a lightweight process

- A light process
  - Executes a main function (signature close to main)
  - Has a private stack for its local data,
  - Shares the overall data of the heavy process
- Context switch between two light processes created by the same heavy process is faster than between two heavy processes:
  - The second case requires more updates in the memory hierarchy (caches, pages, ...)

Lightweight Processes
Software Design

- Benefits for system programming:
  - Easy memory sharing
  - Rapid context change
  - Rich interface compared to the heavy process one
- Benefits for software engineering:
  - Easier design into parallel activities
  - Facilitated interactions between parallel activities
  - Integration into the programming language (Java)as a first class citizen

Plan

- Lightweight processes
- **POSIX Library**
- Java Language
- Design Patterns

POSIX
Lightweight Processes (Thread)

- A thread is defined at the system level by
  - An identifier (tid)
- It has as system resources
  - A priority
  - A copy of its registers
  - A stack
  - A signal mask
  - Private data (keys)
- These resources are only visible within the thread which encompasses it

POSIX
Thread

`pthread_create (…)`	Creates a thread. By default, all threads have the same priority. One can change its priority. **The start time (activation) depends on its priority** .
`pthread_exit (…)`	Completes the thread **only** (not the process). Different from exit that ends the process and its threads.
`pthread_self (…)`	Returns the identifier of the current thread execution
`pthread_join (…)`	Waits for the completion of a thread which is given the identifier as parameter

POSIX Thread activation

```
int main (void) {
   pthread_t t0, t1, t2;
   t0 = pthread_self();                       /* thread t0 */
   pthread_create (&t1, NULL, f, NULL); /* thread t1 */
   pthread_create (&t2, NULL, g, NULL); /* thread t2 */
}
```

- Threads are created with a default priority (parent’s one)
  - This priority can be modified at creation or later on
- Above, t0 continues to execute after creation of t1 … Unless the priority of t1 is greater than that of t0!
  - With equal priority, t1 can also run as soon as it is created, if the scheduling of the processes is done by quantum
- The control of the activation of a thread must be done by synchronization mechanisms (not by priority)

POSIX
Arguments of the thread *main* function

- When calling pthread_create, one can provide a pointer to the parameters passed to the *main* thread function
- It is safer to have the parameters specific to the thread. Do not share them between threads.

```
int main (void) {
 pthread_t t[2];
 int id;
 int * arg;
for (id = 0; id < 2; id++){
   arg = malloc(sizeof(int));
   *arg = id;
   pthread_create(&t[id], NULL, f, arg);
 }
}
```

```
void f (void * arg) {
 int id = (int) *arg;
 printf(«id = %d\n », id);
}
```

POSIX
Using threads

- The user defines the logical parallelism of the application in terms of independent *main* functions
- The system assigns threads to processors in applying a scheduling policy
- The user is responsible for managing concurrent access to resources (mutual exclusion, signal operations, etc)

POSIX
Synchronisation problem

- Below, the final value of V can be different from 20000 !
- The operation v++ (or v = v+1) is not atomic and can be broken down as :
  - Load v in a register
  - Increment register
  - Store the register in v

```
int main (void) {
 pthread_t t1, t2;
 pthread_create(&t1, NULL, f, NULL);
 pthread_create(&t2, NULL, f, NULL);
 pthread_join (t1, …);
 pthread_join (t2, …);
 printf («v = %d\n », v);
 return 0;
}
```

```
int v = 0;
void f (void) {
 int i;
 pthread_t t;
 t = pthread_self();
 for (i=0; i<10000; i++) v++;
 printf(«%d: v = %d\n », t, v);
}
```

POSIX
Synchronization tools

- The synchronization mechanisms are many for lightweight POSIX processes
  - The lock (or mutex) offers mutual exclusion mechanisms. **A critical section must be released as soon as possible**
  - The conditional variable offers queuing mechanisms
  - The semaphore offers the historical P and V mechanisms of heavy processes. Can be built using a lock and a conditional variable.

POSIX
Mutex or Lock

`pthread_mutex_init`	Create a mutex in an unlocked state
`pthread_mutex_destroy`	Destroy mutex
`pthread_mutex_lock`	Take lock if it is free, block the thread otherwise. **The duration of a critical section must be short.**
`pthread_mutex_trylock`	Take lock if free otherwise returns an error without blocking the thread. **The duration of a critical section must be short.**
`pthread_mutex_timedlock`	Take lock if free otherwise wait for an absolute delay (date). **The duration of a critical section must be short.**
`pthread_mutex_unlock`	Release the lock. Must be usually released by the thread that took it

POSIX
Synchronisation with a mutex

```
int v = 0;
pthread_mutex_t m;
void f(void){
   int i;
   pthread_t t;
   t = pthread_self();
   for (i=0; i<10000; i++){
      pthread_mutex_lock(&m);
      v = v + 1;
      pthread_mutex_unlock(&m);
   }
   printf(«%d: v = %d\n », t, v);
}
```

```
void main (void){
   pthread_t t1, t2;
   pthread_mutex_init(&m, …);
   pthread_create(&t1, …, f, …);
   pthread_create(&t2, …, f, …);
   pthread_join(t1, …);
   pthread_join(t2, …);
   printf (« v = %d\n », v);
  }
```

POSIX
Good programming practices with mutexes

- The time spent in a mutex must be short
  - No need for trylock or timedlock in practice
- Only the thread that took the lock can release it
  - Can be configured. This differs from semaphores.

NON

```
…
pthread_mutex_init (&m);
pthread_create (f1, …);
pthread_create (f2, …);
…
void f1(){
   pthread_mutex_lock(&m);
}
void f2(){
   pthread_mutex_unlock(&m);
}
```

OUI

```
…
pthread_mutex_init (&m);
pthread_create (f, …);
…
void f (){
   pthread_mutex_lock (&m);
   …
   pthread_mutex_unlock (&m);
}
```

POSIX
Difference trylock & timedlock(0 us)

- trylock () tries to take the lock and otherwise continues executing
- timelock () tries to take the lock and otherwise waits an absolute timeout possibly 0 us ... but it waits for whatever happens.
- It stays into the blocked state to possibly resume execution if no other thread of the same priority is ready

POSIX
Conditional Variable (CondVar)

`pthread_cond_init(&cv,…)`	Create a conditional variable.
`pthread_cond_destroy`	Destroy a conditional variable.
`pthread_cond_signal(&cv)`	Release a thread blocked on a conditional variable. Possibly none.
`pthread_cond_broadcast(&cv)`	Release all threads blocked on a conditional variable. Possibly none.
`pthread_cond_wait(&cv, &m)`	Block (**always**) the thread by releasing mutex m, unblocking thread on signal or broadcast and at last taking m back.
`pthread_cond_timedwait`	Proceed as `pthread_cond_wait` but wait only for an **absolute** delay (date).

POSIX
Exercise : waiting on guard (1/2)

- We want to block a thread as long as a variable is not equal to a given value (passed as a parameter)
- The code below uses **polling** and may **miss updates**. Why ?

```
int x;
pthread_mutex_t m;
void set_x(int y) {
   pthread_mutex_lock(&m);
   x = y;
   pthread_mutex_unlock(&m);
}
```

```
void wait_for_x_equal(int y){
    while (true) {
       pthread_mutex_lock(&m);
       if (x == y) break;
       pthread_mutex_unlock(&m);
       sleep (t);
     }
   pthread_mutex_unlock(&m);
}
```

POSIX
Waiting on guard (2/2)

- The solution below fixes both problems
- The use of both a VarCond and its mutex allows to suspend the thread without "releasing" mutual exclusion (classic problem)

```
int x;
mutex_t m;
pthread_cond_t v;
void set_x (int y){
   pthread_mutex_lock(&m);
   x = y;
   pthread_cond_broadcast(&v);
   pthread_mutex_unlock(&m);
}
```

```
void wait_for_x_equal(int y) {
    pthread_mutex_lock (&m);
    while (x != y)
       pthread_cond_wait(&v, &m);
    pthread_mutex_unlock(&m);
}
```

POSIX
Interaction entre Mutex et VarCond

- *wait* binds a conditional variable *cv* to a mutex *m*
- The access to the conditional variable *cv* is protected thanks to a lock *m*
- A call to *wait* releases the mutex *m* and sets the thread in *blocked* state in the *cv* queue
- During a *signal or broadcast* call, the thread in the *cv*'s queue is moved to *m*'s queue in order for the thread to get lock *m* back
- Later it will return the lock *m* using *unlock*

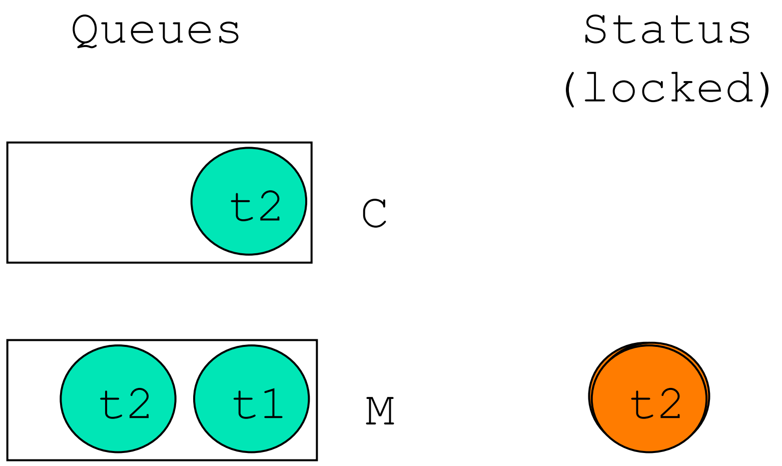
POSIX

Exercise : states and queues

```
void *main_t1(void * arg){
 05 sleep (2);
 06 pthread_mutex_lock (&m);
 11 pthread_mutex_unlock (&m);
}
```

```
void *main_t2(void *arg){
 01 pthread_mutex_lock (&m);
 02 pthread_cond_wait(&c,&m);
 09 pthread_mutex_unlock (&m);
}
```

```
void *main_t3(void * arg){
 03 sleep(1);
 04 pthread_mutex_lock (&m);
 07 sleep (2);
 08 pthread_cond_signal (&c);
 09 sleep (1);
 10 pthread_mutex_unlock (&m);
}
```



POSIX
Timed Conditional Variable

- The return code of pthread_cond_timedwait determines the reason for its release.
  - If the code is 0, the function has been released normally
  - If the code is ETIMEDOUT, the time has expired
- It is important to check the return code of pthread_cond_timedwait and any POSIX function in general

POSIX
Time Management

- Two (absolute) time structures
  - timeval in seconds and microseconds for gettimeofday
  - timespec in seconds and nano-seconds for pthread_cond_wait
- It is therefore necessary to convert them
  - timeval in seconds and microseconds for gettimeofday
  - timespec in seconds and nano-seconds for pthread_cond_wait

```
int                   rc;
struct timespec       ts;
struct timeval        tv;
gettimeofday(&tv, NULL);
ts.tv_sec  = tv.tv_sec + delay;
ts.tv_nsec = tv.tv_usec*1000;
```

```
mutex_lock(&m);
while (x != y){
  rc = cond_timedwait(&v,&m,&ts);
  if (rc == ETIMEDOUT) break;
}
mutex_unlock(&m);
```

POSIX Sémaphore

`sem_open`	Creates a named semaphore and initializes its counter (function also valid for heavy processes)
`sem_init`	Creates an anonymous semaphore and initializes its counter (possibly deprecated function)
`sem_destroy`	Destroy a semaphore
`sem_wait`	Wait for the counter to be positive and decrease it
`sem_trywait`	Decrements the counter if positive otherwise returns a fault
`sem_timedwait`	Wait for the counter to be positive while waiting (always) at most for an absolute period and the decrease, otherwise return an error
`sem_post`	Increments the counter and possibly unblocks a thread

POSIX
Circular blocking buffer

- We have a circular buffer
- We block when we get from an empty buffer
- We block when we put in a full buffer
- Two semaphores to block, third one for mutual exclusion

```
sem_t fullSlots, mutex;
sem_t emptySlots;
int first, size = 0;
int last = MAX - 1;
char b[MAX];
mutex=sem_open(«m»,…,1);
fullSlots=
   sem_open(«fs», …, 0);
emptySlots=
   sem_open(«es», …,M);
```

```
char get (void){
 char c;
 sem_wait(&fullSlots);
 sem_wait(&mutex);
 c=b[first]; size--;
 first=(first+1)%MAX;
 sem_post(&mutex);
 sem_post(&emptySlots);
 return c;
}
```

```
void put(char c){
 sem_wait(&emptySlots);
 sem_wait(&mutex);
 last=(last+1)%MAX;
 b[last]=c; size++;
 sem_post(&mutex);
 sem_post(&fullSlots);
}
```

POSIX Summary

- A thread provides a parallel sequence of execution sharing a address space with other threads
- A mutex serializes access to data for a short time
- A conditional variable blocks a thread within mutual exclusion without introducing deadlocks
- A semaphore tries to take a resource and blocks in case of unavailability
- **POSIX leaves certain semantic variabilities which sometimes make applications non-portable**
- **You MUST check the correct behavior of the functions by return code! This is C programming !**

Plan

- Lightweight processes
- POSIX Library
- **Java Language**
- Design Patterns

Java
Lightweight process (Thread)

- We inherit from the Thread class by overloading the Run method
- We create the Thread object and start it with the predefined method start ()

```
class MyThread extends Thread {
   public void run(){System.out.println("Execute"+ getName());}
}
public static void main (String args[]) {
  Thread t1 = new MyThread("T1"); // Crée l'objet T1
  Thread t2 = new MyThread("T2"); // Crée l'objet T2
  t2.start();   // start appelle la méthode run() de T2;
  t1.start();   // start appelle la méthode run() de T1
}
```

Java
Runnable 1/2

- We inherit the Runnable interface and override the Run method
- We create a Runnable object and a Thread object that we associate in the thread constructor
- We delegate the execution of the Runnable (delegate)
- We start the thread with start () which activates the run () method of the Runnable object
- The Runnable object is therefore not in the Thread inheritance tree

Java
Runnable 2/2

```
class MyRunnable implements Runnable {
    String name;
    public MyRunnable (String s) {name = s;}
    public void run() {
      System.out.println("Execute "  + name);
    }
}
void main (String args[]) {
    MyRunnable r1 = new MyRunnable("R1");
    MyRunnable r2 = new MyRunnable("R2");
    new Thread(r2).start(); // Create and start object R2
    new Thread(r1).start(); // Create and start object R1
}
```

Java
Threads operations

- t.start() is used to activate thread t
- run(), launched by start (), must be overloaded
- sleep(d) suspends current thread for a duration d (ms)
- t.setprio(p) sets a priority p to a thread t
- yield() gives control to the next thread of the same priority, or to the first thread of lower priority
- t.join() waits for the end of thread t
- t.join(d) waits for the end of thread t for a period d (ms)

Java
Synchronisation

- The final value of n.v can be different from 20,000
  - The add operation is not necessarily atomic, it can be broken downas follows:
    - Load v in a register
    - Increment register
    - Store register in v

```java
class MyInt {
   int v;
   void add (int i) {v=v+i;}
}
class MyThread extends Thread {
   static MyInt n = new MyInt(0);
   public void run() {
      for(int i=0; i<10000; i++) n.add(1);
   }
}
```

```java
static void main(String args[]){
   Thread t1, t2;
   t1 = new MyThread ("T1");
   t2 = new MyThread ("T2");
   t1.start ();
   t2.start ();
}
```

Java synchronized method

- Each object is associated with its own lock. The object is locked
  - When calling a synchronized method
  - In a block qualified as synchronized

```
class MyInt {
   int v;
   synchronized void add (int i){
      v=v+i;
   }
   void sub (int i) {
       synchronized (this){v=v-i;}
   }
}
```

Java synchronized method

- The execution of a synchronized method forbids execution of any synchronized method of the object executed by **another** thread
- The same thread can recall a method synchronized object (reentrant lock)
- Calls to non-synchronized methods of the object are always allowed
- When an exception is raised in a synchronised method, the lock is automatically released

Java
Wait, Notify and Broadcast

- wait, notify and notifyAll are predefined methods
- They should only be used in synchronized methods
- They work according to the specifications of wait, signal,
- broadcast of POSIX API conditional variables
- **… but provide no return code !**
- wait() suspends the current thread and releases the object lock. The thread, once resumed, gets lock back.
- notify() resumes a thread suspended on wait()
- notifyAll() resumes any thread suspended on wait()

Java Sem=Mutex+VarCond

- A semaphore can be implemented in Java …
- This implementation is incorrect … why ?
  - Consider the following scenario:
  - First, thread T1 is blocked on acquire()
  - Thread T2 executes release() and thread T3 acquire()

```
class MySem {
  int count;
  synchronized acquire(){          synchronized release() {
    if (count == 0)wait();          if (count == 0)notify();
    count--;                        count++;
  }                               }
                                }
```

Java
Sem=Mutex+VarCond

- Implementation still incorrect … why ?
  - Consider the following scenario :
  - First, two threads are blocked in acquire()
  - A thread executes release(), then another one does the same

```
class MySem {
  int count;
  synchronized acquire() {
    while (count == 0)
      wait();
    count--;
  }
```

```
synchronized release() {
    if (count == 0)notify();
    count++;
  }
}
```

Java Sem=Mutex+VarCond

- An implementation (almost) correct (without interrupts)
  - Java documentation :
    A thread can also wake up without being notified, interrupted, or timing out, a so-called spurious wakeup.
  - In other words, do not assume that a thread always returns from *wait* due to *notify, notifyAll* or a *timeout*

```
class MySem {
   int count;
   synchronized acquire(){
      count--;
      if (count < 0)wait();
   }
                              synchronized release() {
                               count++;
                               if (count <= 0)notify();
                              }
                            }
```

Java Sem=Mutex+VarCond

- Correct implementation (catch exceptions / interrupts)
  - Advice: do not assume notify() resume the thread you expected unless you can prove it …
  - Use notifyAll() instead of notify(): less efficient but safer

```
class MySem {
  int count;
  synchronized acquire(){
    while (count <= 0)
      try {wait();}
      catch(Exception e{})
    count--;
  }
  synchronized release(){
    count++;
    notifyAll();
  }
}
```

Semantic behaviour of wait and notify

- Wait() causes the current thread (T) to place itself in the **wait set** for this object (O) and then to relinquish any synchronization claims on O. T becomes **disabled for thread scheduling purposes** and lies dormant until [notified or interrupted]
- T is then removed from the **wait set** for O and **re-enabled for thread scheduling.** It then competes in the usual manner with other threads for the right to synchronize on the object

- Notify wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is **arbitrary** and occurs at the **discretion of the implementation**.

- **wait set** does not indicate how the queue is managed. **arbitrary** does not indicate nothing on queue management (FIFO, FIFO within priority, etc.) and the interaction with the scheduler (**thread scheduling**) is not specified .

Be careful with the assumptions made about semantics

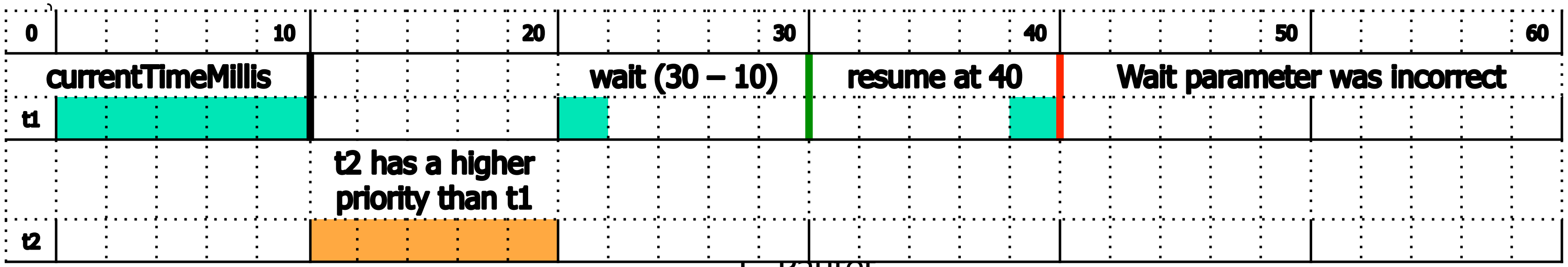
Java
Timed wait

- Like POSIX, wait has a timed version
- wait (long timeout) returns without specifying whether the **relative delay** (duration) has expired or a notification has been made
- pthread_cond_timedwait takes as parameter temporal an **absolute delay** (date)
- System.currentTimeMillis () provides access to the clock
- wait (0) corresponds to wait ()
- We can transform, with precautions, in relative delay and vice versa

Java
absolute delay vs relative delay

- POSIX (resp Java) uses absolute delays (resp relative delays)
- Transforming an absolute delay into a relative one may be incorrect
- In the following, abstime = 30 but the thread is resumed at 40
  - wait(abstime - System.currentTimeMillis()) is incorrect

```
synchronized boolean acquire(long abstime){
  while (count == 0){
    try {wait(abstime - System.currentTimeMillis());break;
    } catch (InterruptedException e) {};
  };
  if (count > 0) {count--; return true;}
  else return false;
```

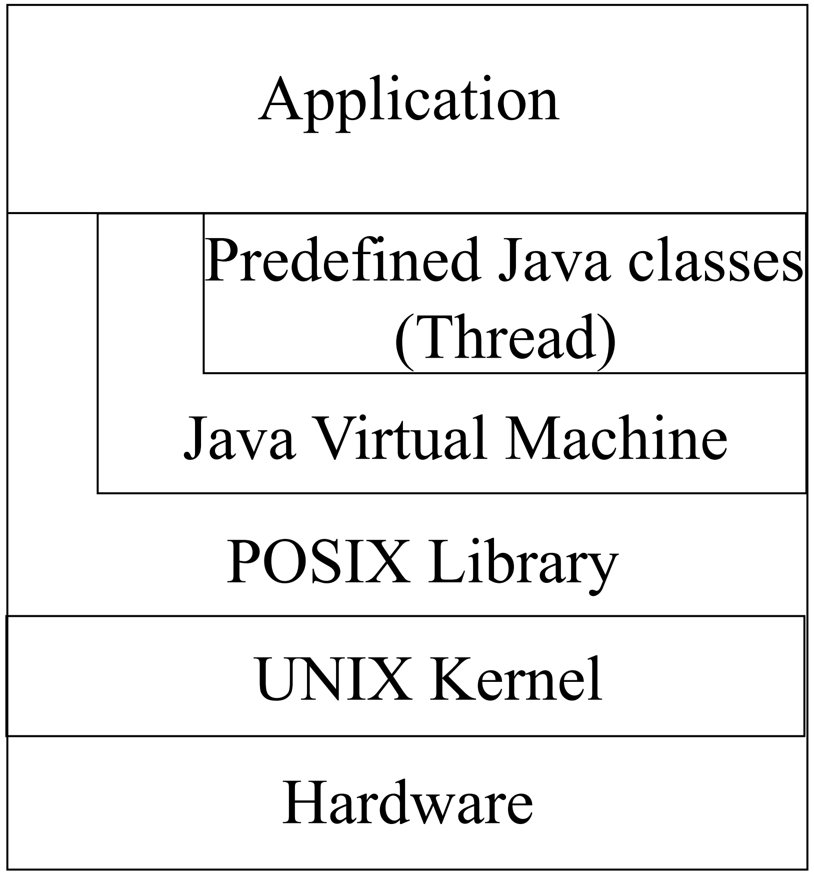


0
10
20
30
40
50
60
currentTimeMillis
wait (30 – 10)
resume at 40
Wait parameter was incorrect
t1
t2 has a higher priority than t1
t2

Java
Differences between POSIX and Java

- In POSIX, wait() takes as a parameter any mutex and conditional variable that the user has allocated
- Typically, it can use the same mutex for two different conditional variables and it is easy to let a thread block on two queues without introducing deadlocks
- In Java, an object has a single mutex and a single conditional variable and it may be tricky to let a thread block on two queues without introducing deadlocks
- In Java, wait implicitly takes as a parameter the simutex and the conditional variable of the object
- The Java machine oftenly relies on the library POSIX

Java
Interaction between Java and POSIX



- Each layer provides its abstraction level and semantics
- Language constructs are expanded by the compiler to fit those of the lower layer
- Examples :
  - Thread Java => Thread POSIX
  - Wait/Notify => Mutex+CondVar
  - CurrentTimeMillis ⇔ Timer

Java
JDK 1.5

- Basic Java mechanisms do not always allow the user to effectively implement more complex mechanisms such as semaphore, barrier, etc.
- Other concurrency mechanisms are very common like POSIX ones but also like concurrency design patterns
- JDK1.5 provides through libraries additional concurrency mechanisms directly implemented at JVM level
- In particular, JDK1.5 provides more direct access to POSIX functions than the original JDK

Java
JDK 1.5 (POSIX and Patterns)

- POSIX:
  - (Reentrant) Lock
  - Conditional Variable
  - Semaphore
- (Many) Patterns:
  - BlockingQueue : already presented
  - Callable and Future : a Callable request corresponds to a Runnable but also returns a result stored in a Future object
  - ExecutorService : manage a pool of threads and execute asynchronously requests through threads from the pool
  - Barrier: block threads as long as # blocked threads < N
  - Latch: block threads while # ressources > 0 (join)

Java
Using POSIX for Java

- Be careful with the **combined use** of synchronization mechanisms specific to POSIX and those specific to Java
- POSIX for Java timed methods take **always relative times** as parameters and not absolute times as in C
- Beware **of exceptions** that may leave POSIX for Java locks in an inconsistent state
- POSIX for Java can define several queues for the same lock while native Java synchronizations do not offer (a single lock and a single conditional variable per object)

Java
Waiting on guard (POSIX/Java)

- Block a thread while a variable is not equal to a given value passed as parameter

```
Lock mutex = new ReentrantLock();
Condition update= mutex.newCondition();
```

```
void setX(int y){
  try {
    mutex.lock();
    x = y;
    update.signalAll();
  } finally {mutex.unlock();}
}
```

```
void waitForXEqual(int y){
  try {
    mutex.lock();
     while (x != y) update.await();
  } finally {mutex.unlock();}
}
```

Plan

- Lightweight processes
- POSIX Library
- Java Language
- **Design Patterns**

Design patterns
Creational, Structural, Behavioural, …

- Well-known problems and well-known solutions
  - Hypothèses d’utilisation
  - Algorithme sous-jacent
  - Ressources utilisées et complexité
- Design patterns (Gang of Four)
  - Creational (Factory, Singleton, …)
  - Structural (Adaptor, Proxy, Facade, …)
  - Behaviour (Strategy, Observer, …)
- Typical example : MVC for Model View Control

Design Patterns JDK 1.5

- POSIX offers basic mechanisms and requires sometimes complex implementations to solve common problems
- Java offers native mechanisms and induces sometimes inefficient implementations for common problems
- Design patterns for concurrency offer additional libraries aiming to overcome these difficulties.
- Design Patterns and POSIX like mechanisms have been introduced in Java libraries (JDK 1.5).

Patrons de conception C/POSIX vs Java/JDK

- C / POSIX offers only few patterns
- Java / JDK offers many patterns
- Important to know these patterns so as not to reinvent their specification and implementation
- Important to know how to correctly implement them independently from the language (in C)
- Important to know their assumptions, used resources and implementation complexity

Patron de conception Blocking Queue

- Pattern: mailbox or blocking queue.
- put : add an element, block if queue is full
- get: remove an element, block if queue is empty
- Semantic variations:
  - Blocking, non-blocking, timed semantics
  - Data structure : array, list … sorted or not
  - Synchronisation mechanisms

Design Patterns
Blocking Queue (native Java)

- Notification mechanisms should be smart …
  - … only when the queue is no longer full or empty
- 2 blocking conditions => 1 queue only (inside object)

```
synchronized Object get() {
  while (size == 0)
     wait();

  // Wake up if needed
  if (size == MAX)
     notifyAll();
  Object o = b[first];
  size--;
  first=(first+1)% MAX;
  return o;
}
```

```
synchronized put(Object o) {
  while (size == MAX)
     wait();

  // Wake up if needed
  if (size == 0)
     notifyAll();
  last=(last+1)% MAX;
  size++;
  b[last] = o;
}
```

Design Patterns
Blocking Queue (POSIX for Java)

- 2blocking conditions => 2 queues (conditions variables)

```
Lock mutex= new ReentrantLock();
Condition notFull  = mutex.newCondition();
Condition notEmpty = mutex.newCondition();
```

```
void put(Object o){
  try {mutex.lock()
     while (size == MAX)
        notFull.await();
     if (size == 0)
        notEmpty.signalAll();
    b[last] = o; size++;
    last=(last+1)%MAX;
  } finally {mutex.unlock();}
}
```

```
Object get(){
  try {mutex.lock();
     while (size == 0)
       notEmpty.await();
    if (size == MAX)
       notFull.signalAll();
    Object o = b[first];
    first=(first+1)%MAX; size--;
  } finally {mutex.unlock();
 return o;}
}
```

Design Patterns
Blocking Queue (Java + POSIX)

- This hybrid implementation causes **deadlocks**
  - Synchronized badly used

```
Semaphore emptySlots = new Semaphore(MAX);
Semaphore fullSlots  = new Semaphore(0);
```

```
synchronized put(Object o){
  emptySlots.acquire();
  last=(last+1)% MAX;
  size++;
  b[last] = o;
  fullSlots.release();
}
```

```
synchronized Object get(){
  fullSlots.acquire();
  Object o = b[first];
  size--;
  first=(first+1)% MAX;
  emptySlots.release();
  return o;
}
```

Design Patterns BlockingQueue (JDK)

- BlockingQueue : data storage interface protected against concurrent access
- ArrayBlockingQueue : fixed size, FIFO
- LinkedBlockingQueue : fixed or dynamic size, FIFO
- PriorityBlockingQueue : dynamic size, Comparables

	Exception	**Value**	**Block**	**Timeout**
Insert	add (o)	offer (o)	put (o)	offer (o, timeout, timeunit)
Remove	remove (o)	poll ()	take ()	poll (timeout, timeunit)
Read	element ()	peek ()		

Design Patterns
Tasks without defining Threads

- We want to execute tasks (or requests) asynchronously: we execute them in parallel and get the result later.
- The ExecutorService offers a way to submit tasks and execute them asynchronously with its internal threads
- A task is an object (not a thread) implementing an interface such as Callable or Runnable. It may be stored in a BlockingQueue while waiting for being executed
- According to its policy, the ThreadPool allocates an internal thread or reuse an existing one to execute tasks
- It defines the number and lifetime of internal threads
- The task result is stored in a Future object.

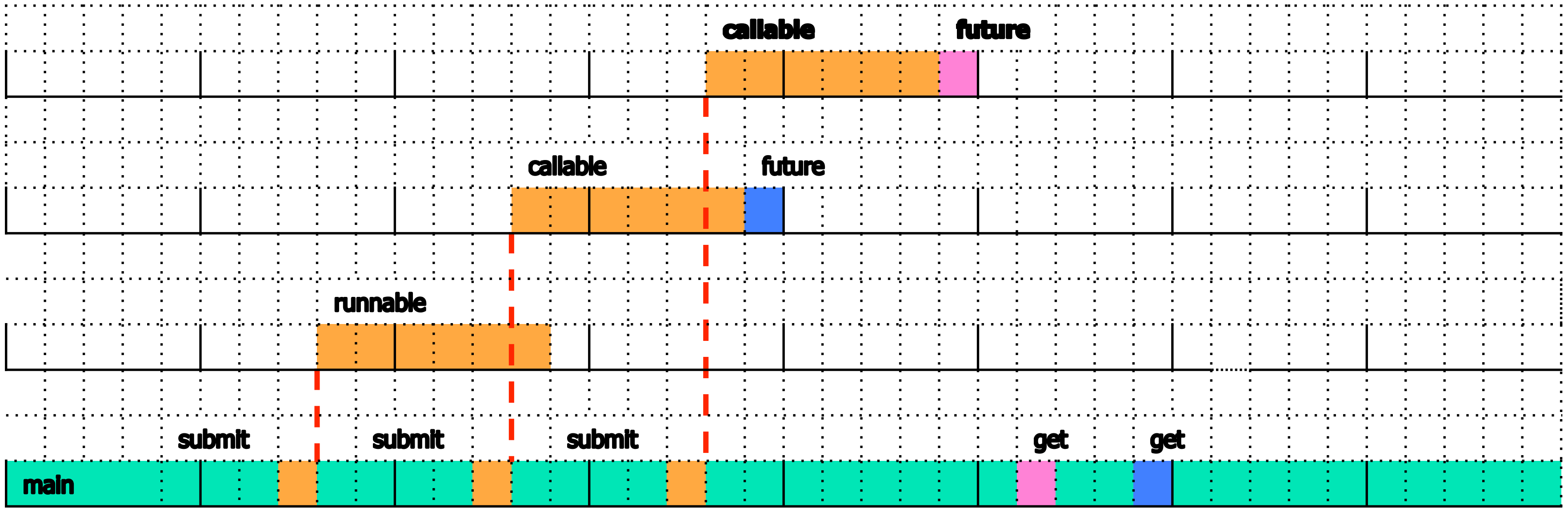
Design Patterns Asynchronous execution

```
ExecutorService executor =
  Executors.newFixedThreadPool(5);
Future future = executor.submit(new Callable(){
  public Object call() throws Exception {
    System.out.println("Asynchronous Callable");
    return "Callable Result";}});
System.out.println("result = " + future.get());
```

- *executor* provides 5 threads to asynchronously execute objects of interfaces *Runnable* or *Callable*
- Submitting a Callable returns a Future object that will be used to get the computation result back

Design Patterns Runnable, Callable and Future

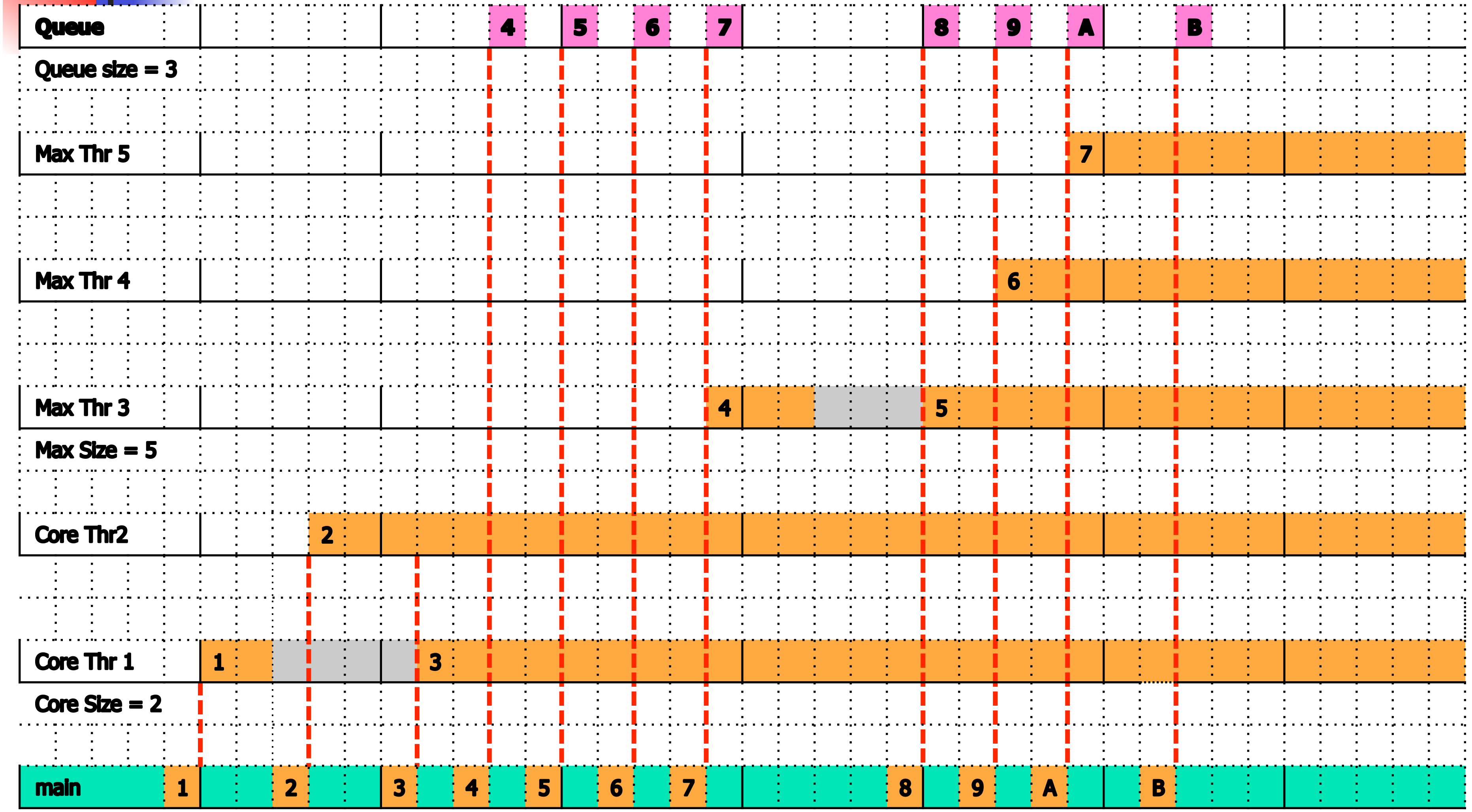
- We submit Callable and obtain a Future object to get the result back when desired and available (otherwise block)
- Unlike a Callable object, a Runnable object does not return any results



Design Patterns ThreadPool

- The **ThreadPool** manages Threads that execute tasks (Callable) stored in a BlockingQueue
- The first **corePoolSize** submissions lead to creation of Threads even if the already created ones remain inactive
- When corePoolSize Threads are created, they execute the tasks stored in the **BlockingQueue**
- When the BlockingQueue becomes full, the number of Threads increases until it reaches **maxPoolSize**
- If it remains full, new submissions throws exception
- Inactive Threads are destroyed after **keepAliveTime,** the number of Threads stays greater than **corePoolSize**

Design Patterns
Threadpool and BlockingQueue



Design Patterns
Predefined ThreadPools

- newFixedThreadPool :
  - Fixed size, corePoolSize = maxPoolSize
- newSingleThreadPool :
  - Fixed size, corePoolSize = maxPoolSize = 1
- Parameter *keepAliveTime*
  - If *poolSize* is greater than *corePoolSize* and if *BlockingQueue* is empty, threads may be destroyed
  - They wait for keepAliveTime and check the above conditions before terminating
  - keepAliveTime = 0 means immediate

Design Patterns ExecutorService

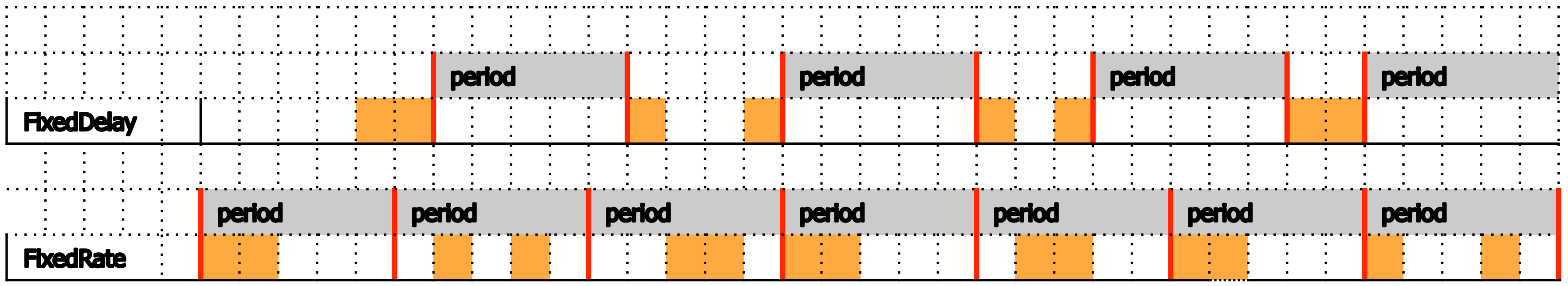
- Configure BlockingQueue, ThreadPool , …
- execute (Runnable r) delegates the execution of r to a thread. A Runnable returns nothing.
- submit (Runnable r) delegates the execution of r. It returns a Future object f. f.get () blocks as long as the execution is not completed.
- submit (Callable c) delegates the execution of c. It returns a Future object f. f.get () returns the result of c when execution is completed.

Design Patterns ScheduledExecutorService

- ScheduledExecutorService is similar to ExecutorService enriched with timed and periodic executions
- *schedule (Callable c, long d, TimeUnit u)* delegates to a Thread the execution of *c* which happens after a relative startup delay *d* expressed in unit *u*.
- *scheduleAtFixedRate (Callable c, long d, long p, TimeUnit u)* executes task *c* periodically with period *p* and a relative delay *d* expressed in unit *u*.
- *scheduleWithFixedDelay (Callable c, long d, long p, TimeUnit u)* executes *c* leaving a time *p* between the end of the previous job and the start of the next one.

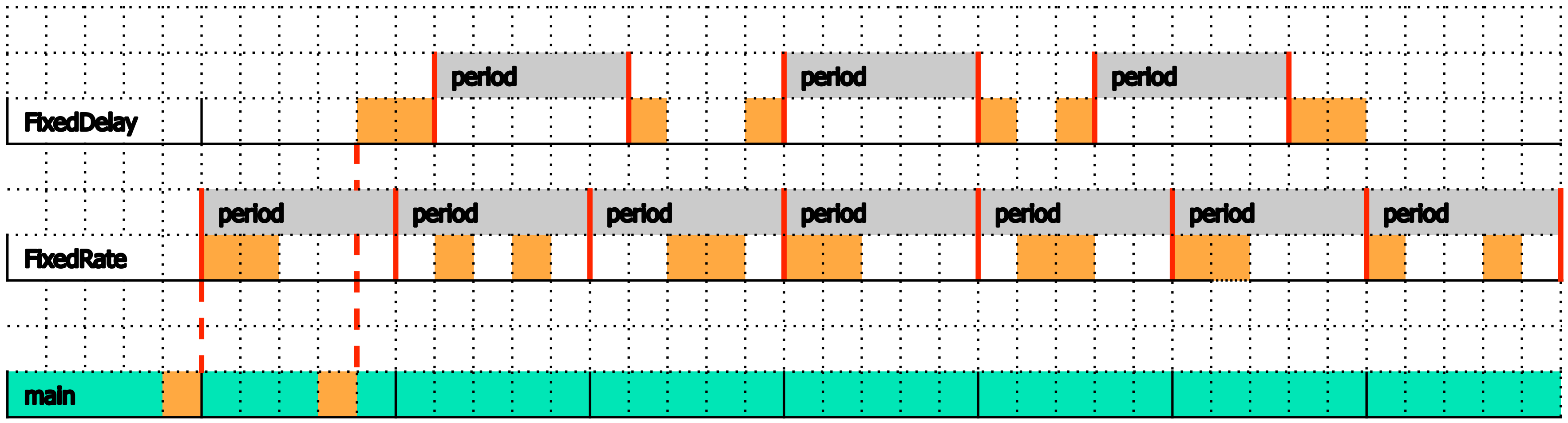
Design Patterns Periodic Tasks

- Periodic tasks do not return Future objects, they are reactivated either with a fixed rate or delay
- With a fixed rate, the activations are predetermined at precise instants separated by a fixed time, the period
- With a fixed delay, the next activation is dynamically determined at the end of current activation + the period



Patrons de conception Thread périodique

- Les threads périodiques ne renvoient pas d'objets Future, elles ne s'arrêtent jamais
- En période fixe, l'activation est déterminée à date fixe
- En délai fixe, l'activation suivante est déterminée à la fin de l'activation courante

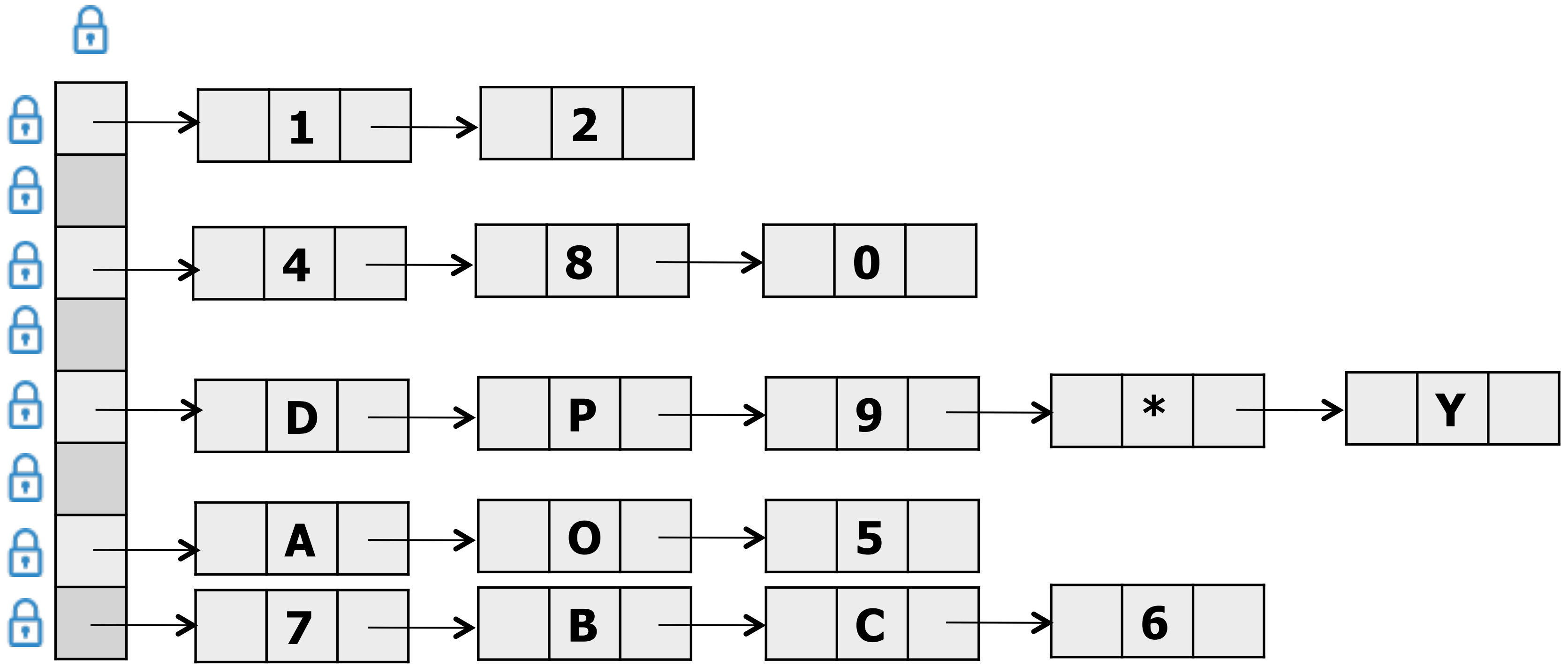


Design Patterns
Lock for Readers / Writers

- Protect a hashtable against concurrent accesses
  - We have a set of key-value pairs
  - From a key, we produce an (hash) index of a table
  - At a given index, the table contains a linked list
  - Each element in the list has the same index produced by different key–value pairs
- Frequent read operations, unfrequent write ones
- Read operations can be executed in parallel
- Write operations must be serialized

Design Patterns
Coarse Grained Lock

- Lock the table in write mode even for read operations and then the lock of linked list.



Design Patterns
ReadWriteLock

- Lock for read and write operations
- Read access if there are no write operations or write requests in progress
- Write access if there are no write or read operations in progress
- Possible famine if write request occurs during an uninterrupted sequence of read operations
- The following implementation is non-reentrant
  - Beware of usage assumptions

Design Patterns
ReadWriteLock

```
int readers = 0;
int writers  = 0;
int writeRequests = 0;

synchronized void lockRead(){
  while (writers > 0 ||
           writeRequests > 0) wait();
  readers++;
}
synchronized void unlockRead(){
  readers--;
  notifyAll();
}
```

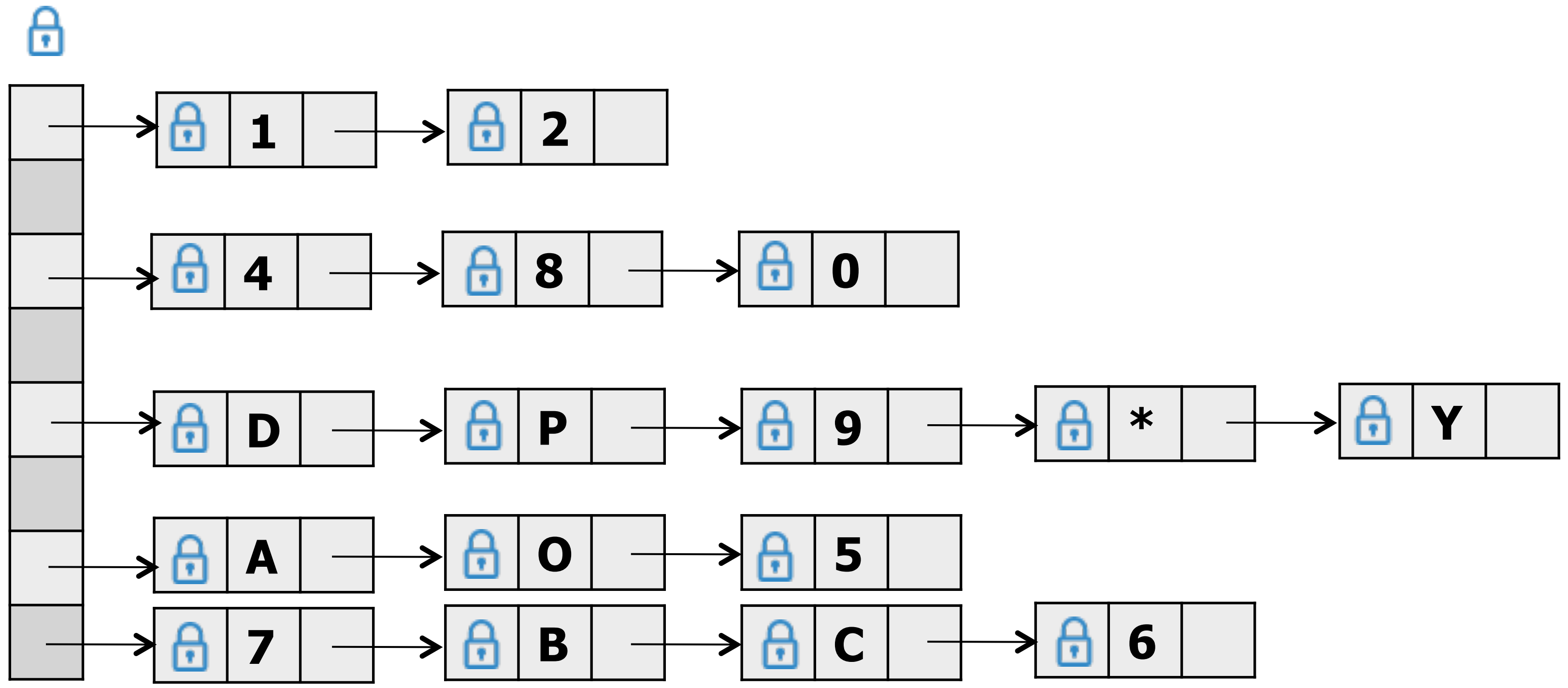
```
synchronized void lockWrite(){
  writeRequests++;
  while (readers > 0 ||
           writers > 0) wait();
  writeRequests--;
  writers++;
}

synchronized void unlockWrite(){
  writers--;
  notifyAll();
}
```

Design Patterns
Fine Grained Lock

- The following code explores a linked list with a fine-grained control ...
- Instead of locking a whole linked list, we lock an element of the list and its precedent
- We have to make compromises. Here, a gain in execution is obtained by a loss of memory
- It is always a matter of balance between CPU usage and memory usage

Design Patterns
Fine Grained Lock



Design Patterns
Fine Grained Lock

```
public boolean add(T item) {
  Node pred, current;
  int key = item.hashCode();
  head.mutex.lock();
  pred = head;
  try {
    Node current = pred.next;
    current.mutex.lock();
    try {
      while (current.key < key){
        pred.mutex.unlock();
        pred = current;
        current = current.next;
        current.mutex.lock();
      }
      if (current.key == key) return false;
      Node newNode = new Node(item);
      newNode.next = current;
      pred.next = newNode;
      return true;
    } finally {current.mutex.unlock();}
  } finally {pred.mutex.unlock();}
```

Design Patterns
Fine Grained Lock

```
public boolean remove(T item) {
  Node pred = null, current = null;
  int key = item.hashCode();
  head.mutex.lock();
  try {
    pred = head;
    current = pred.next;
    current.mutex.lock();
    try {
      while (current.key < key){
        pred.mutex.unlock();
        pred = current;
        current = current.next;
        current.mutex.lock();
      }
      if (current.key == key) {
        pred.next = current.next;
        return true;
      }
      return false;
    } finally {current.mutex.unlock();}
  } finally {pred.mutex.unlock();}
```

Design Patterns
Barrier

- The barrier is used to block N threads and unblock them when the N threads are present
- If the barrier is used twice in a row, the threads released the first round must not cross the barrier immediately for a second round

```
typedef struct _barrier_t {
   pthread_mutex_t mutex;
   pthread_cond_t cv;
   int threshold;
   int counter;
   int cycle;
} barrier_t;
```

Design Patterns
Barrier

```
pthread_mutex_lock (&barrier->mutex);
cycle = barrier->cycle
if (--barrier->counter == 0) {
   barrier->cycle = !barrier->cycle;
   barrier->counter = barrier->threshold;
   pthread_cond_broadcast (&barrier->cv);
} else
   while (cycle == barrier->cycle)
      pthread_cond_wait ( &barrier->cv, &barrier->mutex);
pthread_mutex_unlock (&barrier->mutex);
```

Patrons de conception Barrier

- La barrière sert à bloquer N threads et à les débloquer lorsque les N threads sont présents
- Si la barrière est utilisée deux fois de suite, les threads libérés la 1ère fois ne doivent pas franchir la barrière immédiatement la 2ème fois

```
class Barrier {
   Lock mutex;
   Condition cv;
   int threshold;
   int counter;
   int cycle;};
```

Patrons de conception Barrier

```
void await(){
  this.mutex.lock();
  int cycle = this.cycle
  if (this.counter == 0) {
    this.cycle = !this.cycle;
    this.counter = this.threshold;
    this.cv.signalAll();
  } else
    while (cycle == this.cycle)
      this.cv.await();
  this.mutex.unlock();
}
```

```
Barrier (int threshold) {
  this.threshold = threshold;
  this.counter = threshold;
  this.cycle = 0;
  this.mutex =
     new ReentrantLock();
  this.cv = mutex.newCondition();
}
```

Design Patterns Conclusions

- Design patterns are essential in the design of concurrent systems
- Beware of usage assumptions
- Beware of the algorithm and the complexity
- Pay attention to the resources used
- Balance between execution and memory perf.
- Beware of copy / paste, especially for Java!
- Internet is your friend only if you use it wisely

Processus légers
Mono/Multi Processeurs

- Multi-processeurs
  - Les processeurs avec mémoire commune communiquent par partage de données en mémoire.

    Un calculateur Symmetric Multi Processors est doté d’une mémoire unique et de plusieurs processeurs identiques
- Système réparti
  - Les processeurs sans mémoire commune communiquent par échange de message au travers du réseau.
- Un multi-processeur offre du vrai parallélisme
- Un mono-processeur ou un cœur de multi-processeur offre du pseudo parallélisme (grâce à un noyau)

Processus légers
Parallélisme de traitement

- *f* et *g* modifient 2 parties indépendantes *tab*

```
int tab[100,100]
void main(void){f(); g();}
```

- On peut paralléliser le traitement, soit :
  - Par deux processus lourds (process) indépendants *tab* stocké dans un fichier (accès par *lseek, read, write*)
  - Par deux processus légers (thread) indépendants, *tab* rangé en mémoire partagée

```
/* processus */              /* threads  */
pid=fork();                  pthread_create (…, f, …);
if (pid==0)  {f();}          pthread_create (…, g, …);
else         {g();}
```

Processus légers
Parallélisme des entrées/sorties

- Chaque processus fait une lecture bloquante indépendante de celles des autres processus
- A gauche, les données de *f1* sont lues avant celles de *f2* même si celles de *f2* sont disponibles avant celles de *f1*
- A droite, les lectures se font en parallèle peu importe l'ordre dans lequel les données arrivent

```
read(f1, …);          pthread_create (…, read_f1, …);
read(f2, …);          pthread_create (…, read_f2, …);
read(f3, …);          pthread_create (…, read_f3, …);
```

Processus légers
Différentes implantations

- Au niveau NOYAU
  - Le processus léger est ordonnancé au même niveau que tout processus (lourd ou léger)
- Au niveau PROCESSUS (approche obsolète) :
  - Le noyau ordonnance un processus lourd
  - Le processus lourd ordonnance le processus léger
  - On ne profite pas d'une architecture multi-cœurs car le processus lourd se trouve sur un seul cœur et donc les processus légers qu'il ordonnance sont aussi sur un seul coeur
  - Les entrées / sorties des processus légers doivent être redéfinies pour ne pas être bloquantes au niveau processus

Processus légers Interfaces et sémantiques

- Le standard POSIX fournit une interface de manipulation des processus légers que nous allons étudier
- Ce standard n’est pas toujours précis de sorte que les implantations divergent sur certains points sémantiques
- Il existe d’autres interfaces avec d’autres sémantiques comme celle du système Windows
- Les langages de programmation offrent accès à ces API notamment pour le langage C
- … mais peuvent également fournir des constructeurs propres pour exprimer le parallélisme comme en Java

POSIX
Limitations du sémaphore

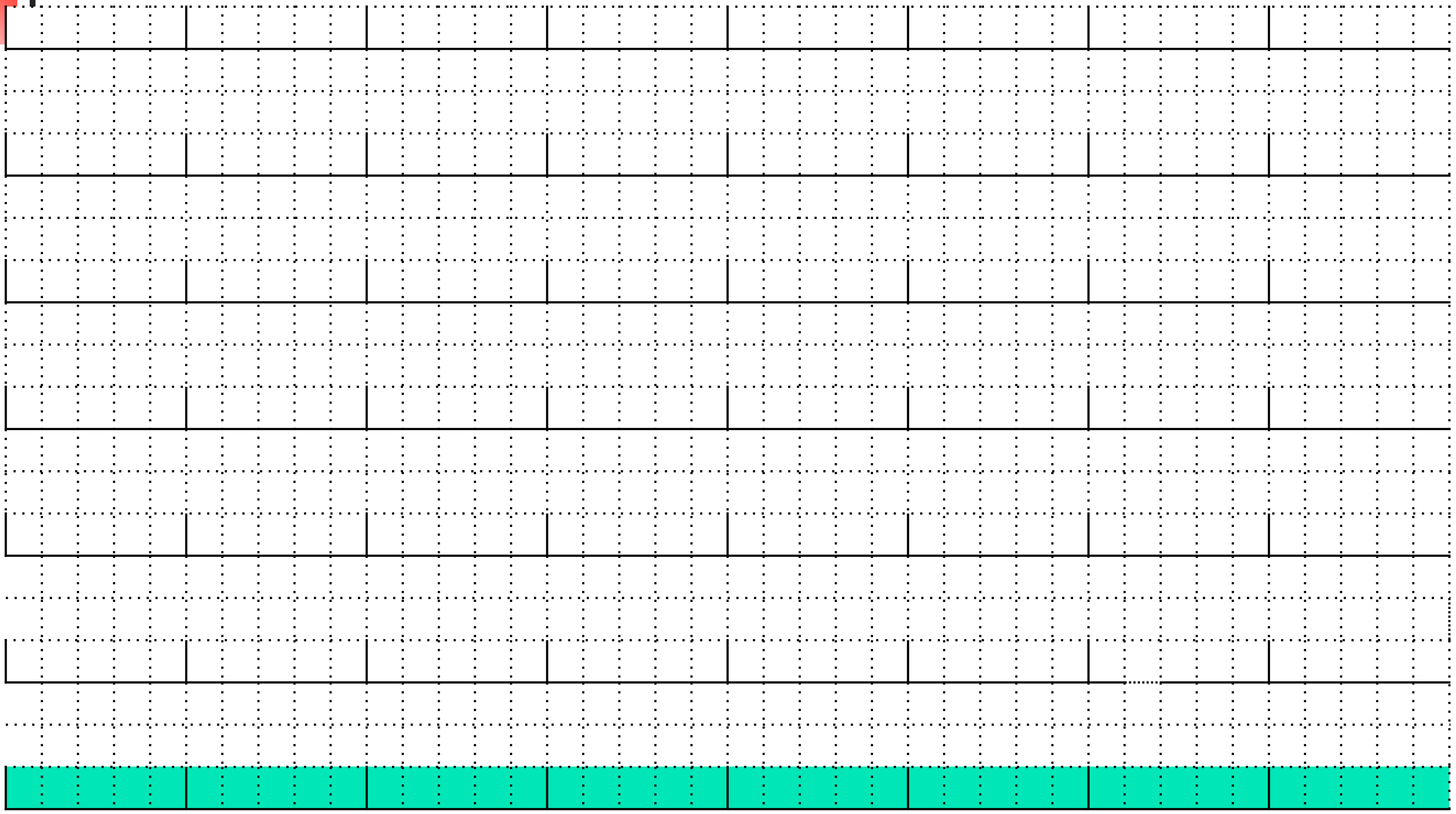
- Le sémaphore est un outil de base peu structurant et difficile à manipuler sur des problèmes complexes
- On prend une ressource du sémaphore sans savoir où on la rendra (à comparer à `goto`). Par exemple, `sem_wait` peut se trouver dans une fonction et `sem_post` dans une autre.
- Un sémaphore peut être mis en œuvre à l'aide d'un mutex et d'une variable conditionnelle

POSIX Sémaphore Mutex + VarCond

```
/********* wait *********/
void wait (my_sem_t *s){
  pthread_mutex_lock(s.mutex);
  s.count--;
  if(s.count < 0)
     pthread_cond_wait(s.varcond,s.mutex);
  pthread_mutex_unlock(s.mutex);
}

/********* post *********/
void post (my_sem_t *s) {
   pthread_mutex_lock (s.mutex);
   s.count++;
   if (s.count <= 0)
      pthread_cond_signal (s.varcond);
   pthread_mutex_unlock (s.mutex);
}
```

```
typedef struct {
   int count;
   pthread_mutex_t *mutex;
   pthread_cond_t *varcond;
} my_sem_t;
```



Patron de conception Blocking Queue (Java/Sem)

```
Semaphore emptySlots = new Semaphore(MAX);
Semaphore fullSlots  = new Semaphore (0);
```

```
void put(Object o){
  emptySlots.acquire();
  synchronized (this) {
    b[last] = o;
    size++;
    last=(last+1)%MAX;
  }
  fullSlots.release();
}
```

```
Object get(){
  Object o;
  fullSlots.acquire();
  synchronized (this) {
   Object o = b[first];
   first=(first+1)%MAX;
    size--;
  }
  emptySlots.release()
  return o;
}
```

Patrons de conception Blocking Queue (C/VarCond)

- Avec une même file pour tampon vide et plein

```
mutex_t m; mutex_init(&m,NULL);
cond_t cv; cond_init(&cv,NULL);

void put(char c){
  mutex_lock(&m);
  while (size == MAX)
    cond_wait(&cv,&m);
  if (size == 0)
    cond_broadcast(&cv);
  last=(last+1)%MAX;
  b[last]=c; size++;
  mutex_unlock(&m);
}
```

```
void * get (){
  void *o;
  mutex_lock(&m);
  while (size == 0)
    cond_wait(&cv,&m);
  if (size == MAX)
   cond_broadcast(&cv);
  o=b[first]; size--;
  first=(first+1)%MAX;
  mutex_unlock(&m);
  return o;
}
```

Patrons de conception
Blocking Queue (C/sem)

- 3 sémaphores et donc séparation des conditions

```
sem_t emptySlots; sem_init(&emptySlots,MAX);
sem_t fullSlots; sem_init(&fullSlots,0);
sem_t mutex; sem_init(&mutex,1);
```

```
void set(void * o){
  sem_wait(&emptySlots);
  sem_wait(&mutex);
  last=(last+1)% MAX;
  b[last]=o;
  size++;
  sem_post(&mutex);
  sem_post(&fullSlots);
}
```

```
void *get(){
  void *o;
  sem_wait(&fullSlots);
  sem_wait(&mutex);
  o=b[first];
  size--;
  first=(first+1)% MAX;
  sem_post(&mutex);
  sem_post(&emptySlot);
  return o;
}
```