

Concurrence

Laurent Pautet

Laurent.Pautet@enst.fr

Version 7.4



Plan

- **Processus légers**
- Bibliothèque POSIX
- Langage Java
- Patrons de conception



Processus légers

Motivations et approches

- Une activité consiste souvent en une suite d'actions:
 - Réception de données (valeurs, événements ...)
 - Traitement des données reçues (calcul / conversion ...)
 - Emission de données en sortie (valeurs, événements ...)
- Toute activité consomme des ressources partagées comme le temps d'exécution ou les données
- Deux approches pour exécuter ces activités:
 - Traitement sérialisé: statiquement entrelacer les activités
 - Il faut tout prévoir pour figer la séquence d'exécution
 - Traitement concurrent : partage temporel des ressources
 - Il faut pouvoir suspendre et reprendre une activité



Processus légers

Avantages et inconvénients

- Coût de la sérialisation pour les systèmes complexes
 - Très déterministe car on sait quelle tâche s'exécute à un instant
 - Peu adaptatif : prévoir à l'avance les enchainements
 - Peu performant : une tâche bloquée bloque les tâches prêtes
- Coût de la concurrence ...
 - Ordonnancement éventuellement préemptif pour déterminer et exécuter les activités prêtes à s'exécuter
- Programmation concurrente :
 - Coordonner des activités partageant leurs ressources
- Programmation répartie :
 - Coordonner des activités sans partage de ressources



Processus légers

Caractéristiques des processus lourds

- Processus lourd Unix « classique » (process)
- Pas de partage de mémoire
 - Copie lors de la création du processus fils
 - Partage coûteux de données par entrées / sorties
- Fonctionnalités rudimentaires
 - fork, exec, exit, wait
 - Synchronisation coûteuse par entrées / sorties
- Pas d'implantations légères possibles
 - Mémoire protégée entre processus par MMU
 - Changement de contexte coûteux (cache, MMU, ...)



Processus légers

Caractéristiques des processus légers

- Les processus légers (threads) :
 - partagent une mémoire commune
 - disposent de plus de fonctionnalités
 - s'accompagnent d'outils de synchronisation
 - peuvent donner lieu à une implantation légère
- Les processus légers ne rendent pas obsolètes les processus lourds classiques qui fournissent
 - Isolation spatiale (espaces d'adressage)
 - Isolation temporelle (ordonnanceurs hiérarchiques)



Processus légers

Architecture

- Un programme se compose de variables globales et de fonctions dont la fonction principale *main*
- Un processus lourd démarre avec un processus léger initial qui exécute la fonction *main*
- Le processus lourd (ou plutôt son processus léger initial) peut créer d'autres processus légers qui exécutent des fonctions de signature similaire à celle de *main*
- Le processus initial et ceux créés ultérieurement s'exécutent en parallèle au sein du processus lourd en partageant les ressources (dont les variables globales)



Processus légers

Composition d'un processus léger

- Un processus léger
 - Exécute une fonction (signature proche de main).
 - Dispose d'une pile privée pour ses données locales,
 - Partage les données globales du processus lourd
- Changer de contexte entre deux processus légers créés par le même processus lourd est plus rapide qu'entre deux processus lourds :
 - Le second cas nécessite plus de mises à jour dans la hiérarchie de mémoire (caches, pages, ...)



Processus légers

Conception logicielle

- Apports pour la programmation système :
 - Partage de mémoire facile
 - Changement de contexte rapide
 - Interface riche comparée à celle des processus lourds
- Apports pour l'ingénierie logicielle :
 - Découpage facilité en activités parallèles
 - Interactions facilitées entre activités parallèles
 - Intégration dans le langage de programmation (Java)



Plan

- Processus légers
- **Bibliothèque POSIX**
- Langage Java
- Patrons de conception



POSIX

Processus léger (Thread)

- Un thread se définit au niveau système par
 - Un identificateur (tid)
- Il possède comme ressources système
 - Une priorité
 - Une sauvegarde de registres
 - Une pile
 - Un masque de signaux
 - Des données privées (clés)
- Ces dernières ne sont disponibles qu'au sein du thread qui l'englobe



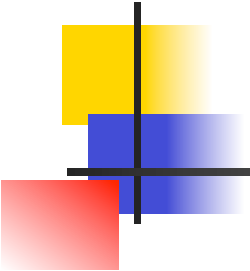
POSIX

Thread

<code>pthread_create (...)</code>	Crée un thread. Par défaut, tous les threads ont la même priorité. On peut changer sa priorité. L'instant de démarrage (activation) dépend de sa priorité.
<code>pthread_exit (...)</code>	Termine le thread uniquement . A la différence d'exit qui termine le processus et ses threads.
<code>pthread_self (...)</code>	Retourne l'identificateur du thread en cours d'exécution
<code>pthread_join (...)</code>	Attend la fin d'un thread dont on donne l'identificateur en paramètre

POSIX

Activation des threads



```
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 */  
}
```

- Les threads sont créés avec une priorité par défaut (en général, celle du créateur)
 - Cette priorité peut être modifiée à la création ou par la suite
- Ci dessus, t0 continue à s'exécuter après création de t1 ... sauf si la priorité de t1 est supérieure à celle de t0 !
 - A priorité égale, t1 peut également s'exécuter dès qu'il est créé, si l'ordonnancement des processus se fait par quantum
- Le contrôle de l'activation d'un thread doit se faire par mécanismes de synchronisation (et non par priorité)

POSIX

Arguments de la procédure principale du thread

- A l'appel de `pthread_create`, on peut fournir un pointeur vers les paramètres passés au *main* du thread
- Il est préférable que les paramètres soient propres au thread. Ne pas les partager entre 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

Utilisation des threads

- L'utilisateur définit le parallélisme logique de l'application en terme de fonctions indépendantes de type *main*
- Le système affecte les threads aux processeurs en appliquant une politique d'ordonnancement
- L'utilisateur se charge de gérer les accès concurrents aux ressources (exclusion mutuelle, réception de signaux)

POSIX

Problème de synchronisation

- La valeur finale de v peut être différente de 20000 !
- L'opération `v++` (ou `v = v+1`) n'est pas forcément insécable, elle peut se décomposer ainsi :
 - Ranger v dans un registre
 - Incrémenter le registre
 - Ranger le registre dans 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

Outils de synchronisation

- Les mécanismes de synchronisation sont nombreux pour les processus légers en POSIX
 - Le verrou (ou mutex)
Offre des mécanismes d'exclusion mutuelle
 - La variable conditionnelle
Offre des mécanismes de file d'attente
 - Le sémaphore
Offre le mécanisme historique de P et V des processus lourds. Peut être construit à l'aide d'un verrou et d'une variable conditionnelle.



POSIX

Mutex ou Verrou

<code>pthread_mutex_init</code>	Crée un verrou dans un état unlocked
<code>pthread_mutex_destroy</code>	Détruit le verrou
<code>pthread_mutex_lock</code>	Prend le verrou si libre, bloque le thread sinon
<code>pthread_mutex_trylock</code>	Prend le verrou si libre sinon renvoie une erreur sans bloquer le thread
<code>pthread_mutex_unlock</code>	Rend le verrou. Doit être <i>généralement</i> être rendu par le thread qui l'a pris



POSIX

Synchronisation avec 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

Bon usage des mutexes

- Le temps passé dans un mutex doit être court
- Seul le thread qui a pris le verrou peut le rendre (comportement par défaut qui peut être modifié)
 - Ceci diffère des sémaphores

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

Variable conditionnelle (VarCond)

<code>pthread_cond_init(&cv,...)</code>	Crée une var. cond.
<code>pthread_cond_destroy</code>	Détruit une var. cond.
<code>pthread_cond_signal(&cv)</code>	Libère un thread bloqué sur var. cond. éventuellement aucun
<code>pthread_cond_broadcast(&cv)</code>	Libère tous les threads bloqués sur var. cond. éventuellement aucun
<code>pthread_cond_wait(&cv, &m)</code>	Bloque (toujours) le thread en rendant le mutex m. Débloque le thread sur signal ou broadcast et reprend m.
<code>pthread_cond_timedwait</code>	Agit comme <code>pthread_cond_wait</code> mais se débloque après un délai absolu . Il s'agit donc de date et non de durée.



POSIX

Exercice : attente sur garde (1/2)

- On souhaite faire bloquer un thread tant qu'une variable ne vaut pas une valeur donnée
- Le code ci-dessous provoque de **l'attente active** et peut **manquer des mises à jour**. Pourquoi ?

```
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

Attente sur garde (2/2)

- La solution ci-dessous règle les deux problèmes
- L'utilisation conjointe d'une VarCond et de son mutex permet de suspendre le thread sans « relâcher » l'exclusion mutuelle (problème classique)

```
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* relie une variable conditionnelle *cv* et un mutex *m*
- On protège l'utilisation de la variable conditionnelle en verrouillant *m* en utilisant *lock*
- Un appel à *wait* relâche le verrou *m* et positionne le thread en état *blocked* dans la file d'attente de *cv*
- Lors d'un appel à *signal*, le thread dans la file d'attente *cv* est déplacé dans la file d'attente de *m*
- Plus tard, il rendra le verrou *m* en utilisant *unlock*

POSIX

Exercice : états et files

```
void *main_t1(void * arg){
```

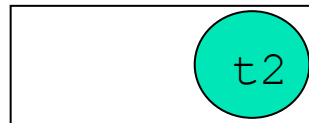
```
05 sleep (2);
```

```
06 pthread_mutex_lock (&m);
```

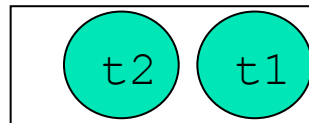
```
11 pthread_mutex_unlock (&m);
```

```
}
```

File
d'attente

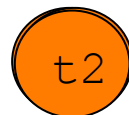


C



M

Statut
(verrouillé)



```
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

VarCond temporisée

- Le code de retour de `pthread_cond_timedwait` détermine la raison de son déblocage.
 - Si le code est 0, la fonction a été débloquée normalement
 - Si le code est `ETIMEDOUT`, le délai a expiré
- Il est important de **vérifier le code de retour** de `pthread_cond_timedwait` et de toute fonction en général



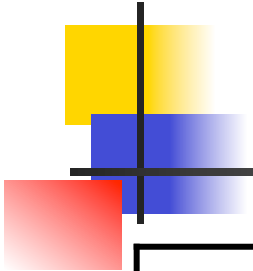
POSIX

Gestion du temps

- Deux structures de temps (absolu)
 - timeval en secondes et micro-secondes pour gettimeofday
 - timespec en seconde s et nano-secondes pour pthread_cond_wait
- Il faut donc les convertir
 - timeval en secondes et micro-secondes pour gettimeofday
 - timespec en seconde s et nano-secondes pour 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

<code>sem_init</code>	Crée un sémaphore anonyme et initialise son compteur (fonction éventuellement disponible)
<code>sem_open</code>	Crée un sémaphore nommé et initialise son compteur (fonction valide aussi pour les processus lourds)
<code>sem_destroy</code>	Détruit un sémaphore
<code>sem_wait</code>	Attend que le compteur soit positif et le décrémente
<code>sem_trywait</code>	Décrémente le compteur si positif sinon retourne une erreur
<code>sem_timedwait</code>	Attend que le compteur soit positif en attendant (toujours) au plus pendant un délai absolu et le décrémente, sinon retourne une erreur
<code>sem_post</code>	Incrémente le compteur (et débloque éventuellement un thread)

POSIX

Tampon circulaire bloquant

- On dispose d'un tampon circulaire
- On bloque lorsque l'on prend dans un tampon vide
- On débloque lorsque l'on dépose un élément
- Un sémaphore pour bloquer, un autre comme verrou

```
#define M 6
sem_t fullSlots, mutex;
sem_t emptySlots;
int first = 0;
int last = M - 1;
int size = 0;
char b[M];
sem_init(&mutex,1);
sem_init(&fullSlots,0);
sem_init(&emptySlots,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)%M;
    b[last]=c; size++;
    sem_post(&mutex);
    sem_post(&fullSlots);
}
```



POSIX

Récapitulatif

- Un thread fournit un fil d'exécution partageant un espace d'adressage avec d'autres threads
- Un mutex sérialise l'accès à des données pendant un temps réduit
- Une variable conditionnelle bloque un thread au sein d'une exclusion mutuelle sans interblocage
- Un sémaphore essaye de prendre une ressource et bloque en cas d'indisponibilité
- **POSIX laisse certaines libertés sémantiques qui rendent parfois les applications non-portables.**
- **Il FAUT vérifier le bon comportement des fonctions par code de retour !**



Plan

- Processus légers
- Bibliothèque POSIX
- **Langage Java**
- Patrons de conception



Java

Processus léger (Thread)

- On hérite de la classe Thread en surchargeant la méthode Run
- On crée l'objet Thread et on le démarre avec la méthode prédéfinie 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

- On hérite de l'interface Runnable en définissant la méthode Run
- On crée un objet Runnable et un objet Thread que l'on associe dans le constructeur du thread
- On délègue l'exécution du Runnable (delegate)
- On démarre le thread par start() ce qui va déclencher la méthode run() du Runnable
- L'objet concurrent ne se trouve donc pas dans l'arbre d'héritage de Thread



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(); // Crée et démarre le thread R2  
    new Thread(r1).start(); // Crée et démarre le thread R1  
}
```



Java

Interface des threads

- `t.start()` sert à activer le thread `t`
- `run()`, exécutée par `start()`, doit être surchargée
- `sleep(d)` bloque le thread courant pour une durée `d` (ms)
- `t.setprio(p)` affecte une priorité `p` à un thread `t`
- `yield()` donne la main au thread suivant de même priorité, ou au premier thread de priorité inférieure
- `t.join()` attend la fin du thread `t`
- `t.join(d)` attend la fin du thread `t` pour une durée `d` (ms)



Java

Synchronisation

- La valeur finale de n.v peut être différente de 20000
 - L'opération add n'est pas forcément insécable, elle peut se décomposer ainsi :
 - Ranger v dans un registre
 - Incrémenter le registre
 - Ranger le registre dans v

```
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);  
    }  
}
```

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



Java

Méthode synchronized

- A chaque objet est associé un verrou. Il est pris:
 - Lors d'un appel à une méthode synchronized
 - Dans un bloc qualifié de synchronized

```
class MyInt {  
    int v;  
    synchronized void add (int i) {  
        v=v+i;  
    }  
    void sub (int i) {  
        // Prend verrou propre à l'objet  
        synchronized (this) {v=v-i;}  
    }  
}
```



Java

Méthode synchronized

- L'exécution d'une méthode synchronized interdit l'exécution de toute méthode synchronized de l'objet effectuée par un **autre** thread
- Le même thread peut rappeler une méthode synchronized de l'objet (verrou réentrant)
- Les appels aux méthodes non-synchronized de l'objet sont toujours autorisées par contre
- En cas de levée d'exception dans une méthode synchronized, le verrou est normalement rendu



Java

Wait et Notify

- wait, notify et notifyAll sont des méthodes prédéfinies de l'objet courant
- Elles ne s'utilisent que dans des méthodes synchronized
- Elles fonctionnent suivant le principe de wait, signal, broadcast des variables conditionnelles de l'API POSIX
- **... mais pas de vérification de code de retour !**
- wait bloque le thread courant et rend le verrou de l'objet. Le thread, une fois débloqué, reprend le verrou.
- notify débloque un thread bloqué sur wait de l'objet
- notifyAll débloque tout thread bloqué sur wait de l'objet



Java

Sem=Mutex+VarCond

- Le sémaphore peut être écrit en Java ...
- Solution incorrecte ... pourquoi ?
 - Exemple de scénario :
 - Au début, un thread T1 est bloqué dans acquire
 - Un thread T2 effectue release et aussitôt un autre, T3, acquiert

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




Java

Sem=Mutex+VarCond

- Solution encore incorrecte ... pourquoi ?
 - Exemple de scénario :
 - Deux threads sont bloqués dans acquire
 - Un thread effectue release, un autre fait de même

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



Java

Sem=Mutex+VarCond

- Solution (presque) correcte (interruptions, ...)
 - Documentation Java :
A thread can also wake up without being notified, interrupted, or timing out, a so-called spurious wakeup.
 - En pratique, il ne faut pas supposer que le thread sort du *wait* suite à un *notify*, un *notifyAll* ou un *timeout*

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



Java

Sem=Mutex+VarCond

- Solution correcte
 - notify ne débloquent pas toujours le thread que l'on souhaite ...
 - Utiliser notifyAll au lieu de notify pour plus de sécurité mais au prix d'une certaine inefficacité

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



Semantique de wait et 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** n'indique en rien s'il s'agit d'une file d'attente. **arbitrary** n'indique rien sur la gestion de la file d'attente (FIFO, FIFO within priority, ...) et l'interaction avec l'ordonnanceur (**thread scheduling**) n'est pas précisé.

Attention aux hypothèses faites sur la sémantique.



Java

Attente temporisé

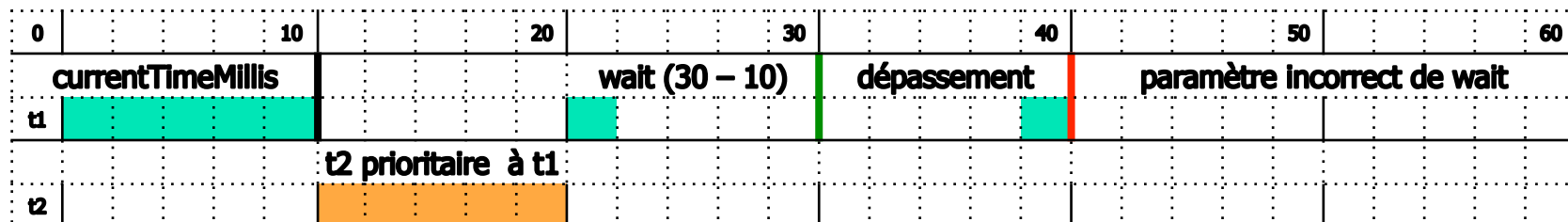
- Comme POSIX, wait dispose d'une version temporisée
- wait(long timeout) retourne sans préciser si le **délai relatif** a expiré ou si une notification a été effectuée
- pthread_cond_timedwait prend comme paramètre temporel un **délai absolu**
- System.currentTimeMillis() permet d'accéder à l'horloge
- wait(0) correspond à wait()
- On peut transformer sous certaines réserves un délai absolu en délai relatif et vice versa

Java

Exemple d'appel temporisé

- POSIX (resp Java) opte pour des délais absolus (resp relatifs)
- Transformer un délai absolu en relatif peut être incorrect
- Exemple : abstime = 30, currentTimeMillis @ 10, wait @ 20

```
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
1						

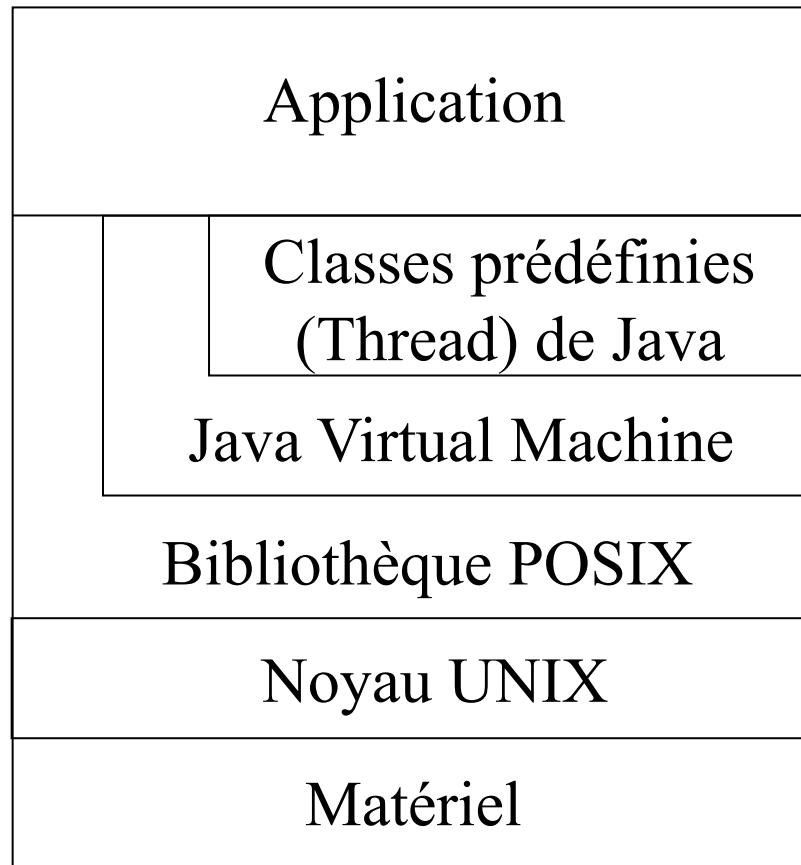
Java

Différences entre POSIX et Java

- En POSIX, wait prend comme paramètre n'importe quels mutex et variable conditionnelle que l'utilisateur a alloués au préalable
- Typiquement, il peut utiliser un même mutex pour deux variables conditionnelles différentes
- En Java, un objet dispose d'un seul mutex et d'une seule variable conditionnelle
- wait prend implicitement comme paramètre le mutex et la variable conditionnelle de l'objet
- La machine Java s'appuie (souvent) sur la bibliothèque POSIX

Java

Interaction entre Java et POSIX



- Chaque couche, si elle existe, fournit un niveau d'abstraction
- Les constructions du langage sont étendues (*expansion*) par le compilateur pour utiliser celles de la biblio du langage ou celles de l'exécutif
- Exemples :
 - Thread Java => Thread POSIX
 - Wait/Notify => Mutex+CondVar
 - CurrentTimeMillis ⇔ Timer



Java

JDK 1.5

- Les outils élémentaires de Java ne permettent pas toujours à l'utilisateur d'implanter efficacement des outils de concurrence de base, comme le sémaphore
- D'autres outils de concurrence sont très répandus comme POSIX mais aussi comme certains patrons de conception pour systèmes concurrents
- JDK 1.5 fournit au travers des bibliothèques complémentaires certains outils de concurrence directement implantés au niveau de la JVM
 - JDK 1.5 fournit un accès plus direct aux fonctions POSIX que le JDK initial ... ainsi que des patrons classiques



Java

JDK 1.5 (POSIX et Patrons)

- POSIX: Verrou, Variable Conditionnelle, Sémaphore
- Patrons ou bibliothèques :
 - BlockingQueue : stockage supportant la concurrence
 - Callable et Future : une tâche Callable correspond à un Runnable qui retourne un résultat Future
 - ExecutorService : stocke des tâches dans un tampon et les exécute en asynchrone grâce à des Threads
 - Barrier: bloque tant que $\# \text{ threads bloquées} < N$
 - Latch: bloque tant que $\# \text{ ressources} > 0$ (join)
 - Exchanger: exécute un rendez-vous entre deux tâches

Java

Utilisation de POSIX pour Java

- Attention au **mélange des concepts** de synchronisation propres à POSIX et ceux propres à Java
- Les méthodes temporisées de POSIX pour Java prennent **toujours des temps relatifs** comme paramètres et non des temps absolus comme en C
- Prendre **garde aux exceptions** qui peuvent laisser les verrous dans un état incohérent
- On peut définir plusieurs files d'attente pour un même verrou ce que ne propose pas les synchronisations natives de Java (un verrou, une var cond par objet)

Java

Attente sur garde (POSIX/Java)

- De nouveau, faire bloquer un thread tant qu'une variable ne vaut pas une valeur donnée

```
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

- Processus légers
- Bibliothèque POSIX
- Langage Java
- **Patrons de conception**



Patrons de conception

Création, Structuration, Comportement

- Problèmes et donc solutions connus et réutilisés
 - Hypothèses d'utilisation
 - Algorithme sous-jacent
 - Ressources utilisées et complexité
- Patrons (Gang of Four)
 - De création
 - De structuration
 - De comportement
- Exemple MVC : Modèle Vue Contrôleur



Patrons de conception

Inspirés de JDK 1.5

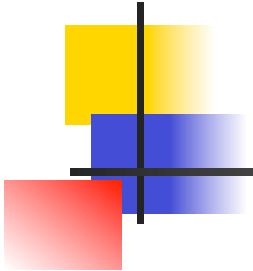
- POSIX offre des mécanismes de concurrence élémentaires et nécessite parfois une écriture parfois ardue pour des solutions complexes
- Java offre des mécanismes de concurrence évolués et nécessite parfois une écriture ardue pour des solutions simples
- Les patrons de conception pour la concurrence ou les bibliothèques complémentaires visent à pallier ces difficultés



Patrons de conception

C/POSIX vs Java/JDK

- Java/JDK offre de nombreux patrons (plus précisément leurs implantations)
- C/POSIX n'offre que peu de patrons
- Connaître ces patrons pour ne pas réinventer les spécifications et les principes
- Savoir correctement implanter ces patrons pour ne pas dépendre du langage (ici Java)
- Connaître les hypothèses d'utilisation, les ressources utilisées et la complexité



Patron de conception

Blocking Queue

- Un patron : boîte aux lettres ou blocking queue.
- put : ajoute un élément, bloque si la boîte pleine
- get: retire un élément, bloque si la boîte vide
- Variantes de patrons:
 - Sémantique bloquante, non-bloquante et temporisée
 - Stockage en tableau, liste ... ordonné ou non
 - Nombre d'objets de synchronisation



Patron de conception

Blocking Queue (Java natif)

- Les tests réveillent intelligemment les threads bloqués
- mais on a une même file pour conditions vide et plein

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

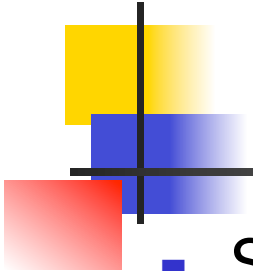
```
    // Wake up if needed  
    if (size == MAX)  
        notifyAll();  
    Object o = b[first];  
    size--;  
    first=(first+1)% MAX;  
    return o;
```

```
}
```

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

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

```
}
```



Patron de conception

Blocking Queue (Java/VarCond)

- Sépare les conditions de blocage et gère les exceptions

```
Lock mutex= new ReentrantLock();
```

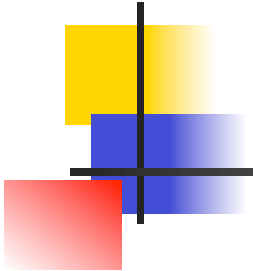
```
Condition notFull  = mutex.newCondition();
```

```
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;}  
}
```

↳ Pautet



Patron de conception

Blocking Queue (Java/POSIX)

- Cette version hybride provoque des **interblocages**
 - Synchronized mal placé
- Intérêt de connaître les patrons et leurs implantations

```
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;  
}
```



Patrons de concurrence

BlockingQueue (JDK)

- BlockingQueue : interface de stockage de données protégées contre les accès concurrents
- ArrayBlockingQueue : implantation avec taille fixe, FIFO
- LinkedBlockingQueue : taille fixe ou dynamique, FIFO
- PriorityBlockingQueue : taille dynamique, 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 ()		



Patrons de concurrence

Tâches sans Threads

- On veut exécuter des tâches en asynchrone : on exécute en parallèle et on récupère le résultat plus tard
- L'interface *ExecutorService* exécute des tâches en asynchrone avec ses Threads internes.
- On soumet une tâche (ie *Callable* ou *Runnable*), et non un Thread, qui est déposée dans une *BlockingQueue*.
- Suivant sa politique de *ThreadPool*, le service alloue un Thread pour exécuter la tâche.
- Le *ThreadPool* définit le nombre et l'instant de création des threads qui exécutent les tâches.
- Le résultat de tâche est stocké dans un *Future*.



Patrons de concurrence

Exécution asynchrone

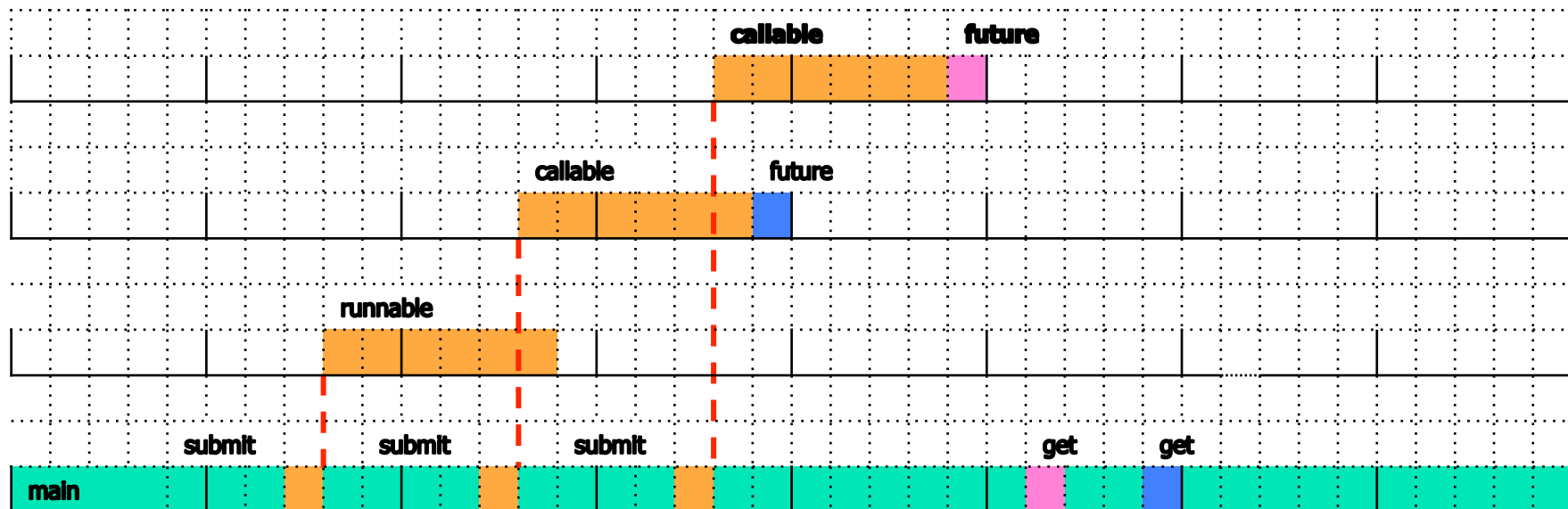
```
ExecutorService executor =  
    Executors.newFixedThreadPool(5);  
Future future = executor.submit(new Callable() {  
    public Object call() throws Exception {  
        System.out.println("Asynchronous Callable");  
        return "Callable Result";}});  
System.out.println("result = " + future.get());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

- *executor* dispose de 5 Threads pour exécuter des objets d'interfaces *Runnable* ou *Callable*
- On exécute de manière asynchrone i.e. ...
- on récupère le résultat plus tard dans *future*

Patrons de conception

Runnable, Callable et Future

- Main demande l'exécution d'un callable grâce à *submit* et obtient un objet *future* pour récupérer le résultat lorsqu'il le souhaite et lorsque celui-ci est disponible (sinon bloque en attendant)





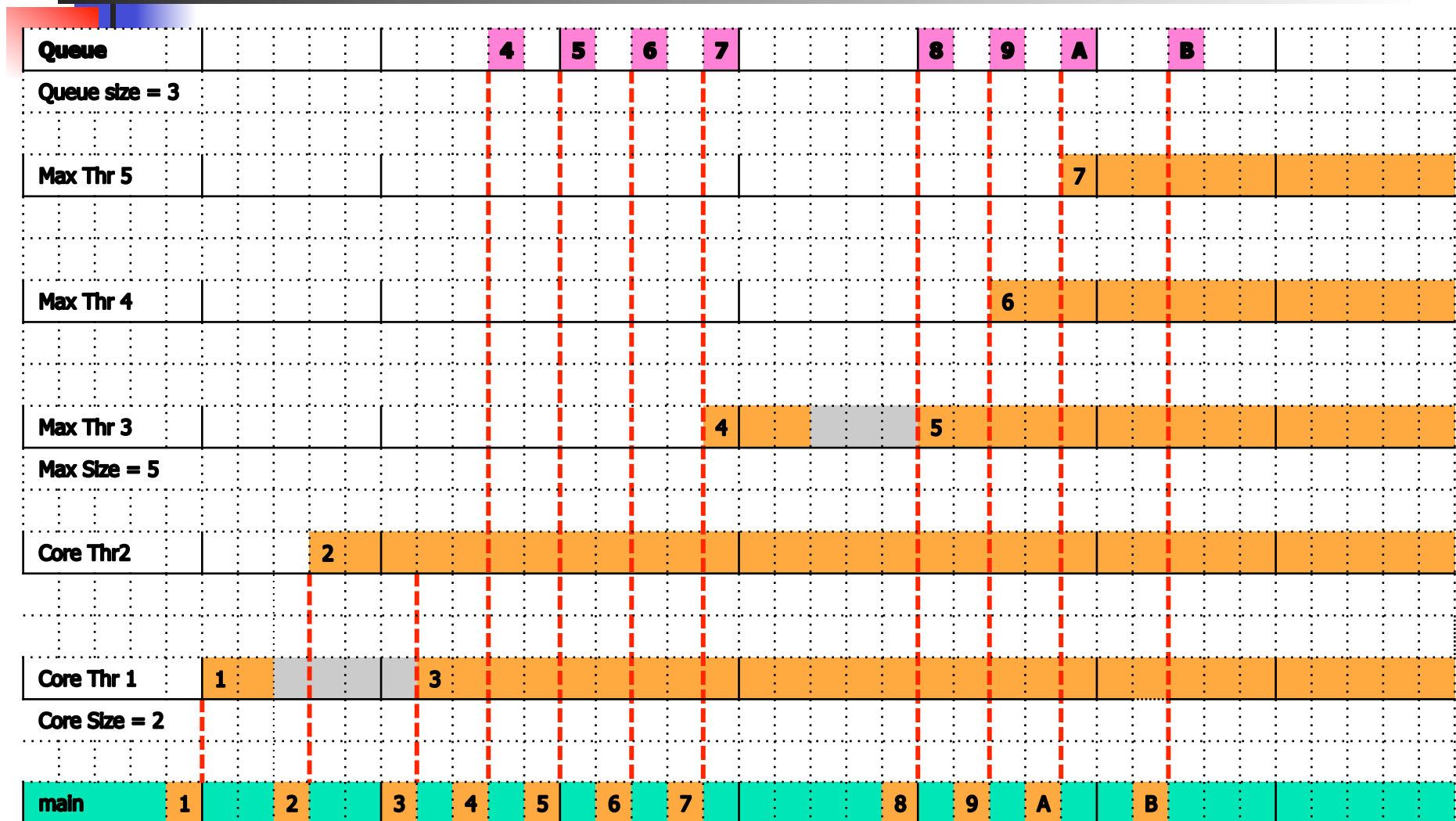
Patrons de concurrence

ThreadPool

- Le *ThreadPool* regroupe les Threads qui exécutent des tâches (*Callable*) stockées dans une *BlockingQueue*
- Les **corePoolSize** premières soumissions entraînent la création de Threads même si celles-ci sont inactives
- Lorsque *corePoolSize* Threads sont créées, elles traitent en boucle les tâches déposées dans la *BlockingQueue*
- Si la *BlockingQueue* vient à être pleine, le nombre de Threads augmente jusqu'à atteindre **maxPoolSize**
- Si la *BlockingQueue* reste pleine malgré cela, les nouvelles soumissions causent des levées d'exception
- Les Threads inactifs sont détruits après **keepAliveTime**

Patrons de concurrence

Threadpool et BlockingQueue





Patrons de concurrence

ThreadPools prédéfinis

- `newFixedThreadPool` :
 - taille fixe, `corePoolSize` = `maxPoolSize`
- `newSingleThreadPool` :
 - taille fixe de 1, `corePoolSize` = `maxPoolSize` = 1
- Paramètre *keepAliveTime*
 - Si *poolSize* est supérieur à *corePoolSize* et si *BlockingQueue* est vide, les threads vont se détruire
 - Ils attendent `keepAliveTime` pour vérifier la propriété et se détruire (immédiat si `keepAliveTime` = 0)



Patrons de concurrence

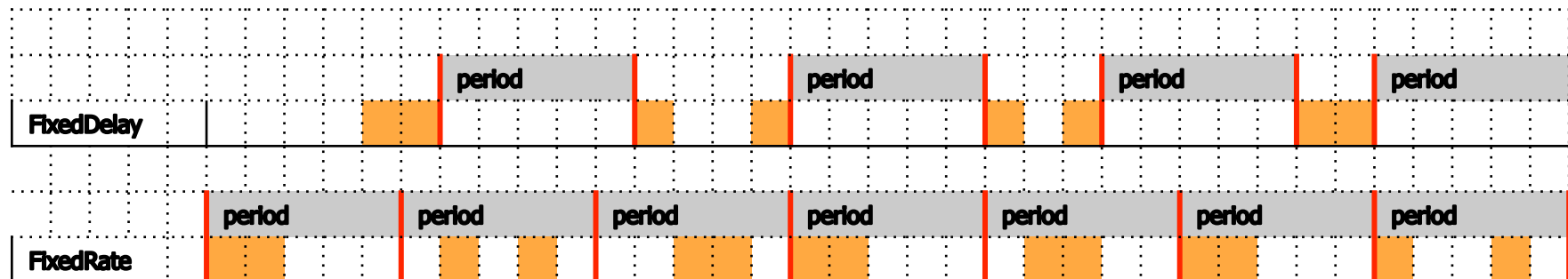
ExecutorService

- Configure *BlockingQueue*, *ThreadPool*, ...
- *execute(Runnable r)* délègue l'exécution de *r* à un thread. Un *Runnable* ne retourne rien.
- *submit(Runnable r)* délègue l'exécution *r*. Retourne *f*, un objet *Future*. *f.get()* bloque tant que l'exécution n'est pas terminée.
- *submit(Callable c)* délègue l'exécution *c*. Retourne *f*, un objet *Future*. *f.get()* retourne le résultat de *c* lorsque l'exécution est terminée

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 l'avance
- En délai fixe, l'activation suivante est déterminée au début de l'activation courante





Patrons de concurrence

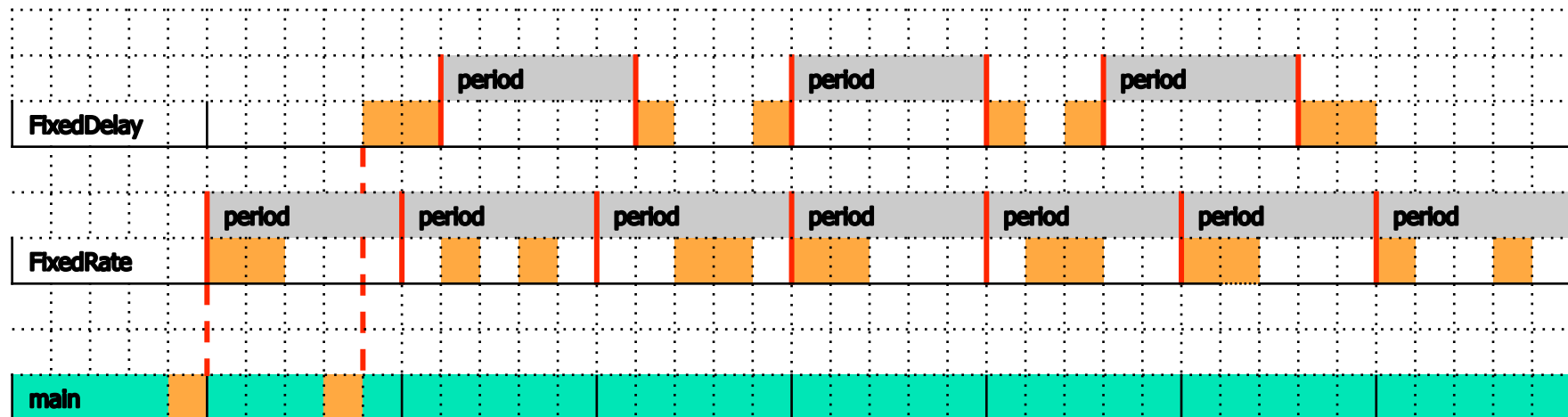
ScheduledExecutorService

- Similaire à *ExecutorService* avec des exécutions temporisées et périodiques
- *schedule* (*Callable c*, *long d*, *TimeUnit u*) délègue à un Thread l'exécution de *c* après *d* exprimé dans l'unité *u*, un délai de démarrage **relatif** .
- *scheduleAtFixedRate* (*Callable c*, *long d*, *long p*, *TimeUnit u*) exécute *c* périodiquement avec la période *p* et un délai relatif *d* exprimés dans l'unité *u*.
- *scheduleWithFixedDelay* (*Callable c*, *long d*, *long p*, *TimeUnit u*) exécute *c* en laissant un temps *p* entre la terminaison d'un travail et l'activation du suivant.

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



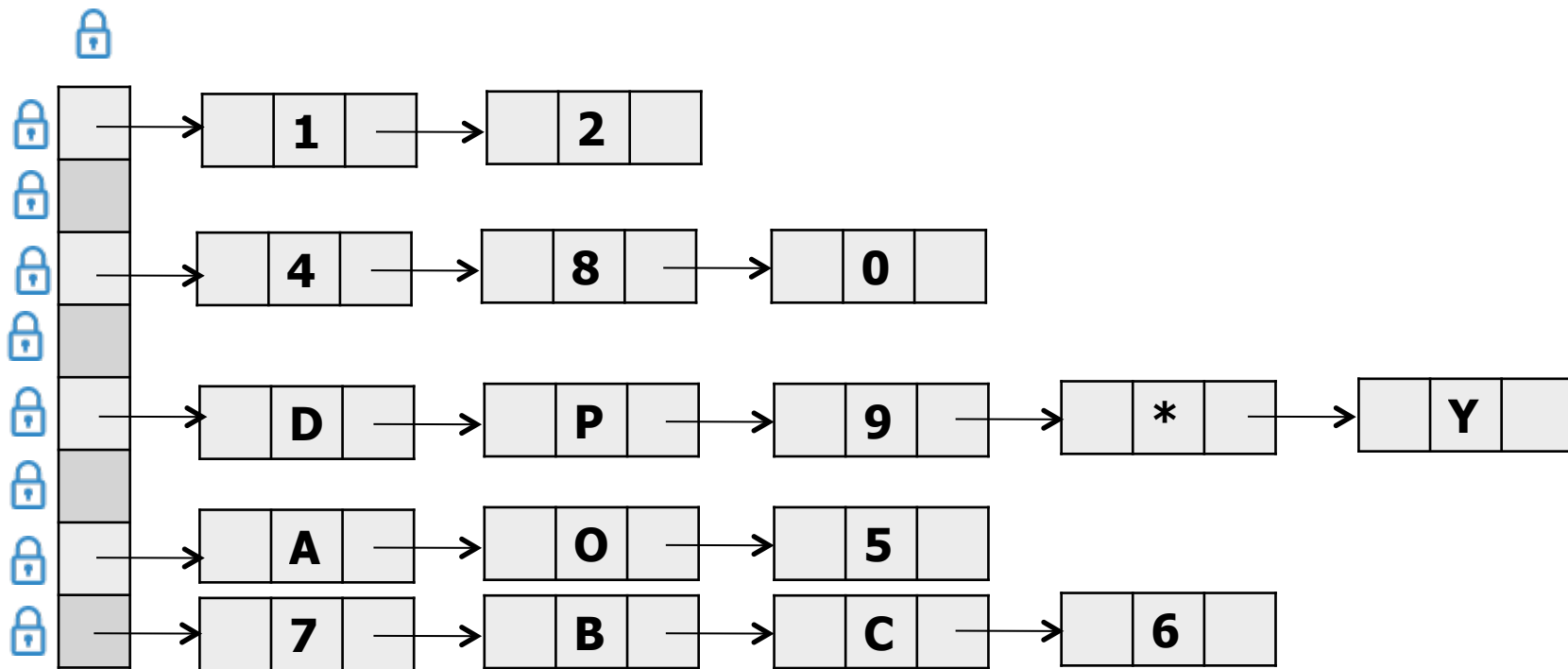


Patron de conception

Verrou lecteur / écrivain

- Besoin : exemple d'une table de hachage
 - A partir d'un élément, on produit un code de hachage
 - On dispose d'une table indexée par les codes
 - Chaque entrée de table contient une liste chaînée
 - Une liste chaînée contient des éléments le même code
- On ne consulte le tableau souvent qu'en lecture
- Les lectures peuvent avoir lieu parallèlement
- Verrouillage différent pour lecture et écriture

Patrons de conception table de hachage (gros grain)





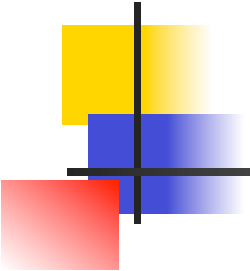
Patrons de concurrence

ReadWriteLock

- Verrou disponible pour des lectures ou des écritures
- Accès en lecture s'il n'y a pas d'écritures ou de demandes d'écriture en cours
- Accès en écriture s'il n'y a pas d'écrivain ou de lecteurs en cours
- Famine possible si on ne prend en compte les demandes d'écriture lors de demandes ininterrompues de lecture

Patrons de conception

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();  
}
```

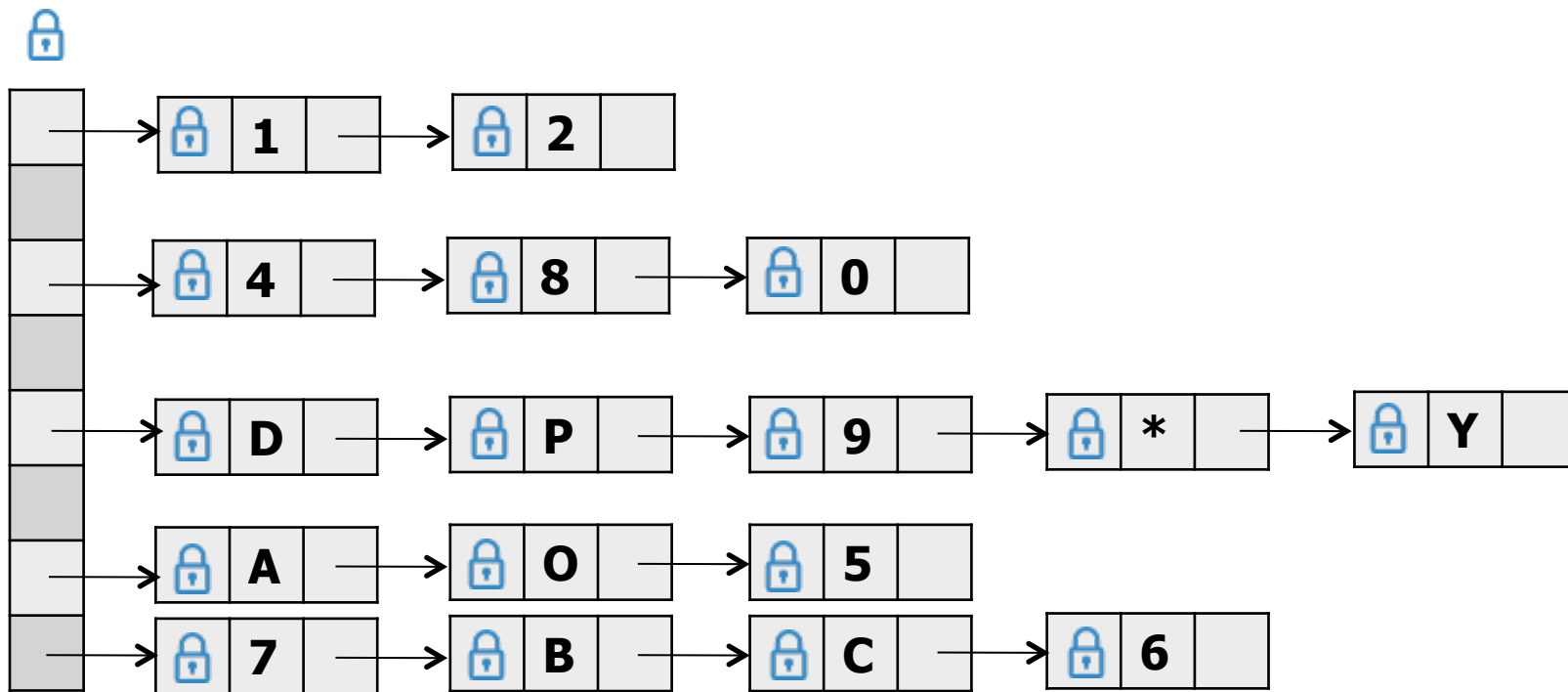


Patrons de conception

ReadWriteLock non reentrant

- La version précédente est non-réentrante
- Prendre garde aux hypothèses d'utilisation
- Le code qui suit explore une liste chaînée avec un contrôle de concurrence à grain fin ...
- Au lieu de verrouiller toute la liste on verrouille un élément de la liste et son précédent
- On doit faire des compromis. Ici, un gain en exécution s'obtient par une perte en mémoire

Patrons de conception table de hachage (grain fin)





Patrons de conception

Liste chaînée (grain fin)

```
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();}
```



Patrons de conception

Liste chaînée (grain fin)

```
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();}
```


Patrons de conception

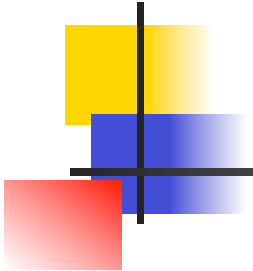
Barrier

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

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

Patrons de conception

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

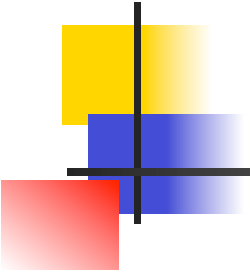
Barrier

- La barrière sert à bloquer N threads et à les débloquent lorsque les N threads sont présents
- Si la barrière est utilisée deux fois de suite, les threads libérés la 1^{ère} fois ne doivent pas franchir la barrière immédiatement la 2^{ème} 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();
}
```

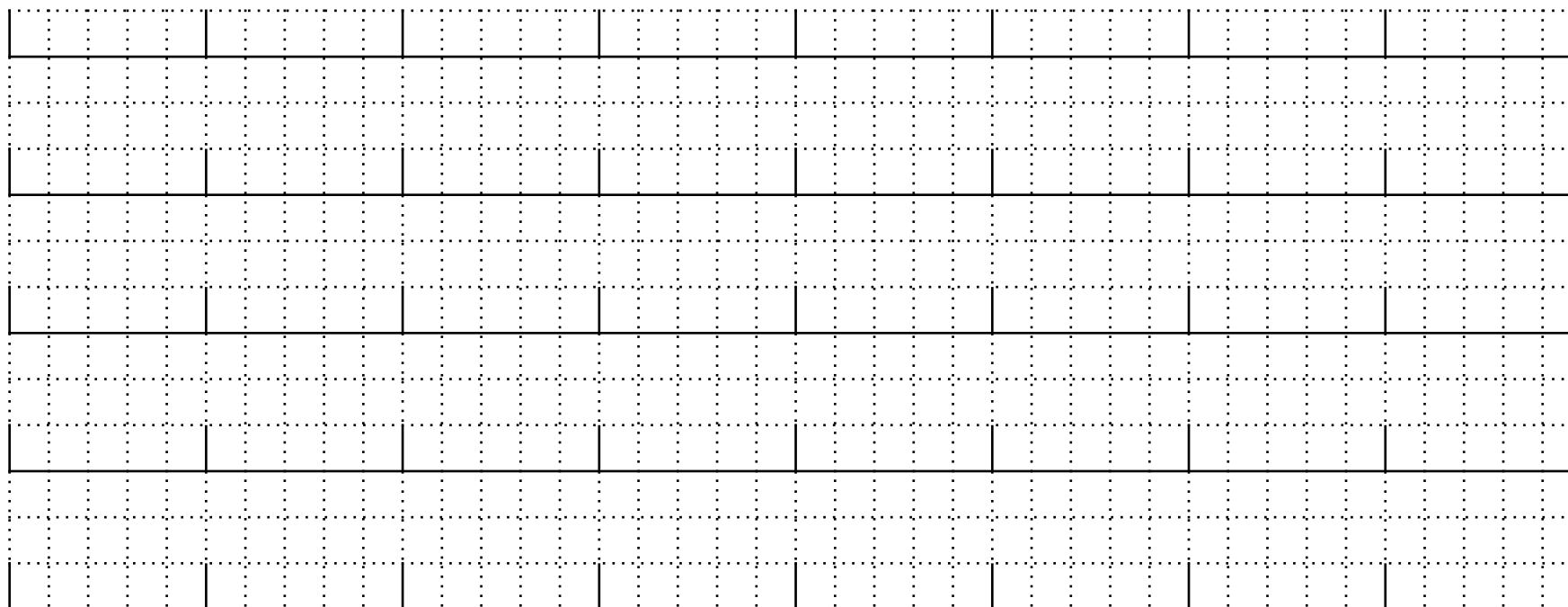


Patrons de conception

Conclusions

- Les patrons de concurrence sont essentiels dans la conception de systèmes concurrents
- Prendre garde aux hypothèses d'utilisation
- Prendre garde à l'algorithme et la complexité
- Prendre garde aux ressources utilisées
- Choisir entre perf. d'exécution et de mémoire

- Gare au copier / coller, surtout pour Java !





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 */
pid=fork();
if (pid==0) {f();}
else        {g();}
```

```
/* threads */
pthread_create (... , f, ...);
pthread_create (... , 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, ...);  
read(f2, ...);  
read(f3, ...);
```

```
pthread_create (... , read_f1, ...);  
pthread_create (... , read_f2, ...);  
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 cœur
 - 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

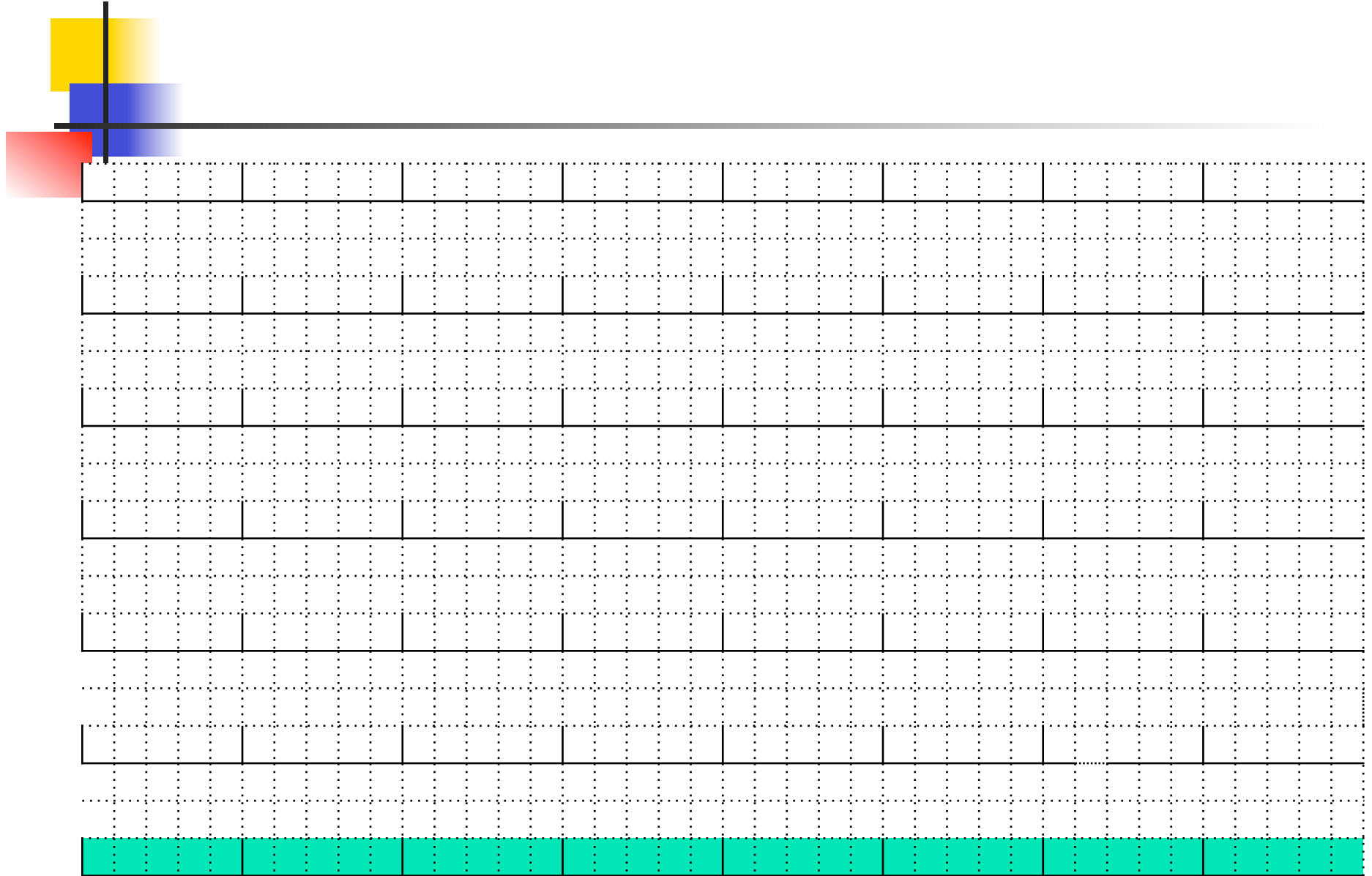
```
/****** wait *****/
```

```
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;
```





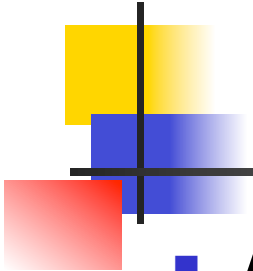
Patron de conception

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;
```