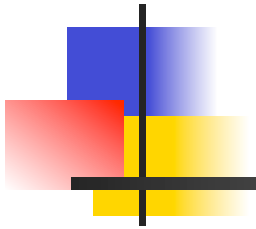
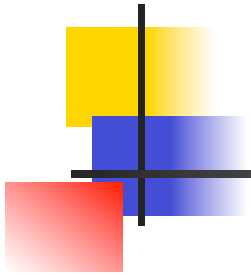


Concurrence

Laurent Pautet
Bertrand Dupouy
Laurent.Pautet@enst.fr

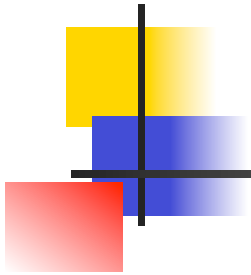
Version 3.0



Plan

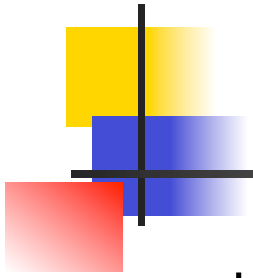
- Introduction
- Processus légers
- Bibliothèque POSIX
- Langage Java

- UEs de spécialités approfondissant ces mécanismes:
  - INF341: Spécifications et modélisation de logiciels
  - INF342: Systèmes Temps Réel
  - INF346: Systèmes Répartis



Processus légers
Plan

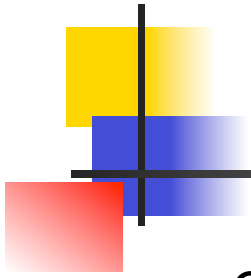
- Problématiques
- Définitions
- Utilisations



Processus légers
Problèmatique

- Un processus classique *(process en anglais)* de type Unix est maintenant qualifié de processus **lourd**
- Les processus lourds ne partagent pas de mémoire
  - Partage coûteux par des entrées / sorties
  - UNIX : Ajout des IPC SystemV
- Fonctionnalités rudimentaires
  - `fork, exec, exit, wait`
  - Pas d'outils de synchronisation
    - Passage coûteux par des entrées / sorties
    - UNIX : Ajout des IPC SystemV
- Pas d'implantations légères possibles
    - Gestion coûteuse en mémoire

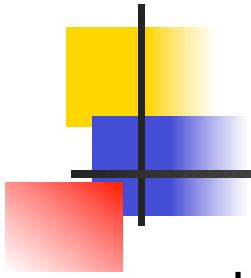
Processus légers
Proposition



- On définit un nouveau type de processus : les processus légers (*threads en anglais*), ceux-ci :
  - partagent une mémoire commune
  - disposent de plus de fonctionnalités
  - s'accompagnent d'outils de synchronisation
  - peuvent ainsi donner lieu à une implantation légère

- Les processus légers ne rendent pas obsolètes les processus lourds classiques
  - Isolation spatiale (espaces d'adressage)
  - Isolation temporelle (ordonnanceurs hiérarchiques)

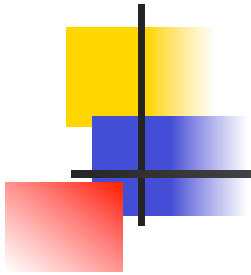
Processus légers
Définitions



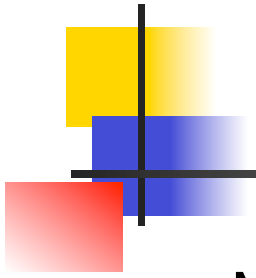
- Un programme se compose notamment de variables globales et de plusieurs sous-programmes dont le sous-programme principal *main*
- Un processus lourd se compose initialement d'un processus léger exécutant le sous-programme principal et de ressources (ex : mémoire pour variables globales)
- Le processus lourd au travers de son processus léger initial pourra créer d'autres processus légers qui exécuteront un des sous-programmes du programme

- Les processus légers (le processus initial et ceux créés ultérieurement) s'exécutent en parallèle au sein du processus lourd en **partageant les ressources**

Processus légers
Résumé



- Un processus léger
  - exécute un sous-programme (procédure ou fonction).
  - Dispose d'une pile privée pour ses données locales,
  - partage les données globales avec les autres processus légers.
- Le changement de contexte entre deux processus légers **créés par le même processus** lourd s'avère plus rapide qu'entre deux processus lourds :
  - le second cas nécessite plus de mise à jour dans la hiérarchie de mémoire (caches, pages, …)



Processus légers
Processus vs Processeurs

- Multi-processeurs
  - Plusieurs processeurs partagent de la mémoire. On parle de *Symetric Multi Processor* dans le cas d'un ordinateur doté d'une mémoire unique et de plusieurs processeurs
- Multi-processus (lourds ou légers)
  - Plusieurs processus légers partagent des ressources, en particulier la mémoire, pour s'exécuter en parallèle.
- Système réparti :
  - Un système réparti se compose de plusieurs processeurs sans mémoire commune et communiquant par message
- Un multi-processeur offre du vrai parallélisme
- Un mono-processeur offre du pseudo parallélisme

Processus légers
Parallélisation de traitement

- `f` et `g` mettent à jour deux parties **indépendantes** du tableau `Tab`

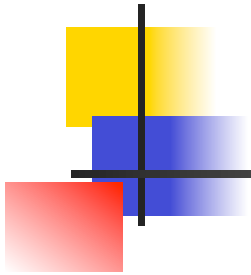
```
int Tab[100,100]
void main(void){f(); g();}
```

- On peut paralléliser le traitement, soit :
  - en créant deux processus lourds indépendants
    Tab doit être alors copié dans un **fichier** partagé (accès par lseek, read, write)
  - en créant deux processus légers (threads) indépendants,
    Tab est rangé en **mémoire** partagée

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

```
/* threads  */
pthread_create (f, …);
pthread_create (g, …);
```

Processus légers
Parallélisation des entrées/sorties

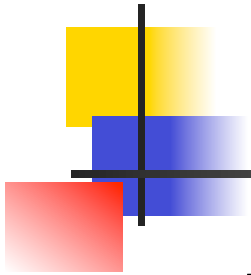


- Chaque processus fait une lecture bloquante indépendante de celles des autres threads
- Dans la solution de gauche, les lectures bloquantes imposent un ordre de réception
- On reçoit les données du fichier f1 avant celles de f2 même si les secondes sont disponibles avant les premières
- Dans la solution de droite, les lectures se font en parallèle

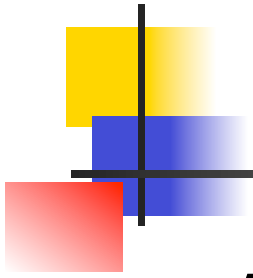
```
read(f1, …);
read(f2, …);
read(f3, …);
```

```
pthread_create (read_f1, …);
pthread_create (read_f2, …);
pthread_create (read_f3, …);
```

Processus légers
Différentes implantations



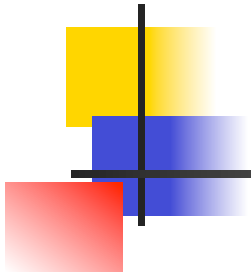
- Au niveau NOYAU : le processus léger est ordonnancé au même niveau que tout processus (lourd ou léger)
- Au niveau PROCESSUS : le processus léger est ordonnancé sur le temps du processus lourd créateur
  - Dans le cas PROCESSUS, on profite moins d'une architecture multi-processeurs et les entrées / sorties des processus légers doivent être redéfinies pour ne pas être bloquantes au niveau processus
- 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



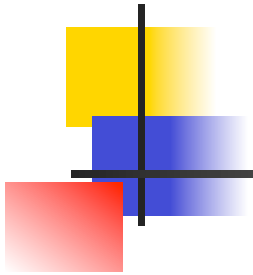
Processus légers
Conclusions

- Apports au niveau programmation système :
  - Partage de mémoire facile
  - Changement de contexte rapide
  - Interface enrichie par rapport à celle des processus lourds
- Apports au niveau ingénierie logicielle :
  - Découpage facilité en activités parallèles
  - Interactions facilitées entre activités parallèles
  - Modélisation de haut niveau autre que API (Java, Ada)
- Problèmes : différentes interfaces, différentes implantations :
  - Plusieurs interfaces (API) sont disponibles (POSIX, Windows, …)
  - Les implantations d'une même interface (Solaris, Linux, …) peuvent différer sur certains points de sémantique

POSIX
Plan



- Threads
- Verrous
- Variables conditionnelles
- Sémaphores



POSIX
Thread ou processus léger

- 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 données ne sont disponibles qu'au sein du processus qui l'englobe

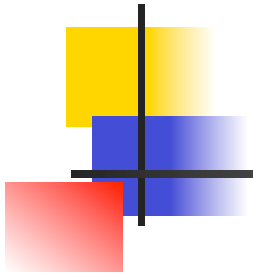
POSIX Thread

`pthread_create (…)`	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.**
`pthread_exit (…)`	Termine le thread **uniquement**. A la différence d'exit qui termine le processus et ses threads.
`pthread_self (…)`	Retourne l'identificateur du thread en cours d'exécution
`pthread_join (…)`	Attend la fin d'un thread dont on donne l'identificateur en paramètre

POSIX
Activation des threads

```
int main (void) {                  /* thread t0 */
   …
   pthread_create (f, …);          /* thread t1 */
   pthread_create (g, …);          /*thread  t2 */
   …
}
```

- Les threads sont créées avec une priorité par défaut (en général, celle du créateur)
  - Cette priorité peut être modifiée à la création ou par la suite
- Dans l'exemple, le thread 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 grâce à des mécanismes de **synchronisation** (et non par sa priorité)



POSIX
Utilisation des threads

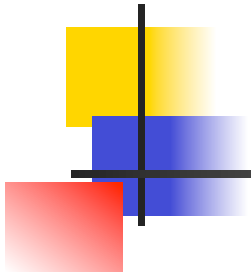
- L'utilisateur définit le parallélisme logique de l'application en terme de sous-programmes
- Le **système** se charge d'affecter les threads aux processeurs en appliquent la politique **d'ordonnancement**
- **L'utilisateur** se charge de gérer les accès **concurrents** aux ressources (exclusion mutuelle, réception de signaux)

POSIX
Problème de synchronisation

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

```
static 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);
}
```

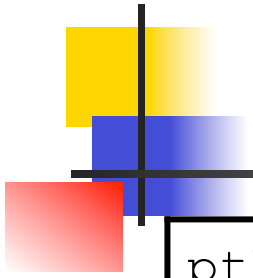
- La valeur calculée pour v peut être différente de 20000 !
- L'opération v++ (ou v = v+1) n'est pas forcément insécable, elle peut se décomposer ainsi :
  - Ranger v dans un registre
  - Incrémenter registre
  - Ranger registre dans v



POSIX
Synchronisation

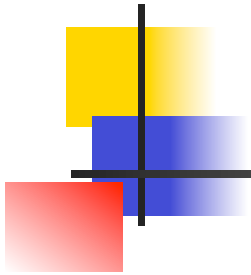
- Les mécanismes de synchronisation sont plus nombreux pour les processus légers notamment en POSIX
  - Le verrou (ou mutex)
    Offre des mécanismes d'exclusion mutuelle
  - La variable conditionnelle
    Offre des mécanismes de file d'attente
  - Le sémaphore
    Offre le mécanisme historique de P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

POSIX
Mutex ou Verrou



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

POSIX : Mutex
Exemple



```
static 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 Mutex Programmation

- 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, …);
…
fonc1(){
pthread_mutex_lock(&m);
}
fonc2(){
pthread_mutex_unlock(&m);
}
```

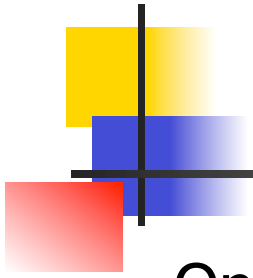
OUI

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

POSIX
Variable conditionnelle

`pthread_cond_init(&cv,…)`	Crée une var. cond.
`pthread_cond_destroy`	Détruit une var. cond.
`pthread_cond_signal(&cv)`	Libère un thread bloqué sur var. cond. éventuellement aucun
`pthread_cond_broadcast(&cv)`	Libère tous les threads bloqués sur var. cond. éventuellement aucun
`pthread_cond_wait(&cv, &m)`	Bloque (**toujours**) le thread en rendant le mutex m. Débloque le thread sur signal ou broadcast et reprend m.
`pthread_cond_timed_wait`	Agit comme `pthread_cond_wait` mais se débloque après un délai. Un délai de 0 induit une attente infinie

POSIX Variable Conditionnelle Exemple 1/2

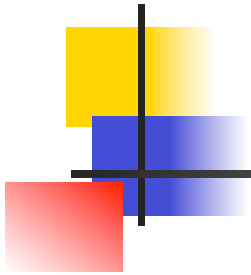


- On souhaite faire bloquer un thread tant qu'une variable ne vaut pas une valeur donnée
- La solution ci-dessous provoque de l'attente active et peut manquer des mises à jour si plus de deux d'entre elles surviennent pendant la période t

```
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);
        int r = x;
        pthread_mutex_unlock(&m);
        if (r == y) break;
        sleep (t);
    }
}
```

POSIX Variable Conditionnelle Exemple 2/2



- La solution ci-dessous règle les deux problèmes
- L'utilisation conjointe d'une var. cond. et de son verrou permet de bloquer sans « sortir » de 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
Sémaphore

`sem_init`	Crée un sémaphore et initialise son compteur
`sem_destroy`	Détruit un sémaphore
`sem_wait`	Attend que le compteur soit positif et le décrémente
`sem_trywait`	Décrémente le compteur si positif sinon retourne une erreur
`sem_post`	Incrémente le compteur (et débloque éventuellement un thread

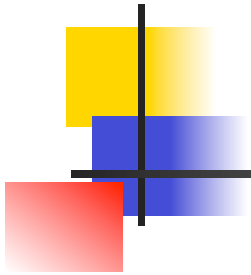
POSIX Sémaphore Exemple

Un producteur et un consommateur

```
sem_t  emptySlots;
sem-_t fullSlots;
int first, last, n;
char b[size];
sem_init
  (&emptySlots,size);
sem_init
  (&fullSlots,0);
first=0;
last=size;
n=0;
```

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

```
void set (char c){
sem_wait(&emptySlots);
last=(last+1)%size;
b[last]=c;
n--;
sem_post(&fullSlots);
}
```



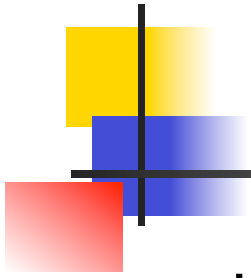
POSIX
Limitations du sémaphore

- 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, `wait` peut se trouver dans une fonction et `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

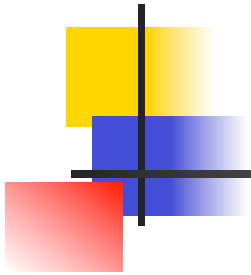
```
/********* wait *********/
void wait (my_sem_t *s){
  pthread_mutex_lock(s.mutex);
  s.count--;
  if(s.count < 0)
  pthread_cond_wait(s.varcond,s.mutex);
  pthread_mutex_unlock(s.mutex);
}

/********* 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);
}
```

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



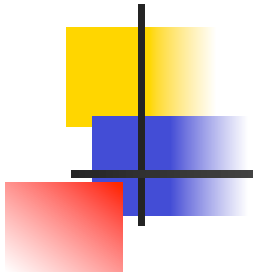
POSIX
Récapitulatif

- 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 (union d'un mutex et d'une var. cond.)
- **POSIX laisse certaines libertés sémantiques qui rendent parfois les applications non-portables.**



Java
Plan

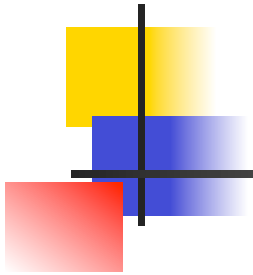
- Manipulation de processus
- Méthode synchronized de l'utilisateur
- Méthodes prédéfinies de synchronisation
- Limitations du modèle de concurrence Java
- Extensions apportées par JDK 1.5



JavaThread

- 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
}
```



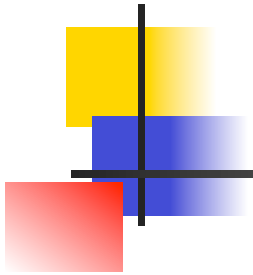
Java
Runnable 1/2

- 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
Utilisation des threads



- t.start() sert à activer le thread t
- run(), exécutée par start(), vide par défaut, doit être surchargée
- sleep(d) bloque le thread courant pendant une durée d en 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 pendant une durée d en ms

Java Synchronisation

```
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 ();}
```

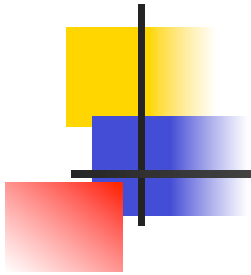
- n.v peut être différente de 20000
  - L'opération add n'est pas forcément insécable, elle peut se décomposer ainsi :
    - Ranger v dans un registre
    - Incrémenter registre
    - Ranger registre dans v

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**
- L'appel à une méthode synchronized interdit l'accès à toute autre méthode synchronized de l'objet
- Les méthodes non-synchronized continuent à accéder à l'objet
- En cas de levée d'exception dans une méthode synchronized, le verrou est normalement rendu

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

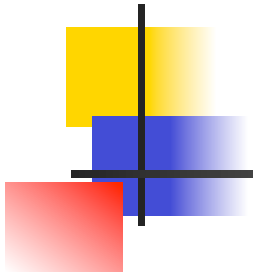
Java
Wait et Notify



- `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

- `wait` bloque le thread courant et rend le verrou de l'objet (synchronized) de façon atomique. 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

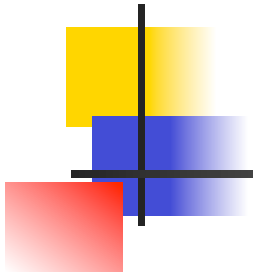
Java
Sem=Mutex+VarCond



- 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, *acquire*

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

Java
Sem=Mutex+VarCond

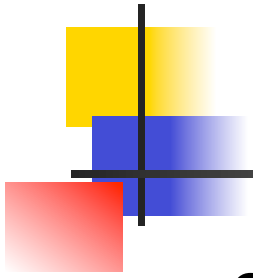


- Solution encore incorrecte … pourquoi ?
  - Exemple de scénario :
    - Deux threads sont bloqués dans *acquire*
    - Un thread effectue *release* puis un autre fait de même

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

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

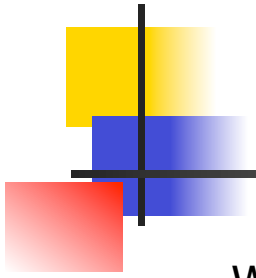
Java
Sem=Mutex+VarCond



- Solution (presque) correcte (interruptions, …)
- *notify* ne débloque pas toujours le thread que l'on souhaite … (voir plus loin)
- Utiliser *notifyAll* au lieu de *notify* au prix d'une certain inefficacité …

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

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



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
Tampon circulaire (inefficace)

- On implante une boîte aux lettres sous forme de tampon circulaire :
  - put *(*resp. get*)* bloque s'il est plein *(*resp. vide*)*.
  - n : taille courante,
  - first et last : début et fin de tampon
    - Cette solution recalcule (**inefficace**) les conditions de sortie de get et put pour chaque thread en attente mais ne nécessite qu'une seule file d'attente

```
class Mailbox {

synchronized Object get() {
  while (n == 0) wait();
  Object o = b[first];
  n--;
  first=(first+1)%size;
  notifyAll();
}
```

```
synchronized put (E o) {
  while (n == size) wait();
  last=(last+1)%size;
  n++;
  b[last] = o;
  notifyAll();
  }
 }
```

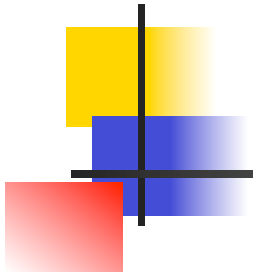
Java
Tampon circulaire (incorrect)

- objet Java = un mutex (+ une variable conditionnelle)
  - Or tampon circulaire = un mutex + deux var. conditionnelles
  - Cette solution imbrique deux objets sémaphores dans un objet tampon ce qui peut poser un problème **d'interblocage**
    - *Exemple : le tampon est vide, un consommateur entre dans get (donc prend le verrou associé à l'objet), se bloque sur fullSots. Si un producteur arrive, il se bloquera sur put (verrou pris, libérable par le consommateur)*

```
MySem emptySlots = new MySem(n);
MySem fullSlots  = new MySem(0);
synchronized put(Object o){
   emptySlots.acquire();
   last=(last+1)%size;
   b[last] = o;
   fullSlots.release();}
```

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

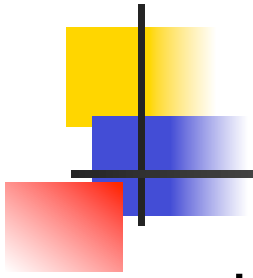
Java
Tampon circulaire (correct)



- Dans un objet Java, la variable conditionnelle est obligatoirement liée au mutex de l'objet associé
- Dans POSIX, la variable conditionnelle est liée au mutex passé en paramètre à wait().

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

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



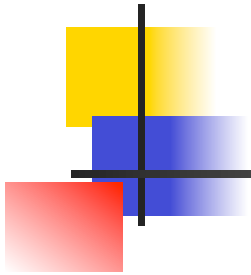
Java
Appel temporisé

- Un appel temporisé sur une méthode bloquante ne peut se faire au niveau du thread appelant
  - L'objet auquel appartient la méthode bloquante doit la dupliquer pour en faire une version temporisée
  - wait(long timeout) retourne sans préciser si le délai a expiré ou si une notification a été effectuée
  - Solution :
    - System.currentTimeMillis() permet d'accéder à l'horloge
    - On note l'horloge avant et après le wait.
    - L'expiration est détectée si la différence excède le délai
    - Le thread peut être suspendu entre ces deux mesures. L'expiration peut survenir en dépit d'une notification sans que cela influe sémantiquement

Java
Exemple d'appel temporisé

```
synchronized boolean acquire(long t){
  long s = System.currentTimeMillis();     // suspend
  while (count==0){
     wait(t);
     long r  = System.currentTimeMillis(); // resume
     if ( (t != 0) && (r - s >= t))
         return false; // échéance ou non?
   };
  count--;
  return true;
}
```

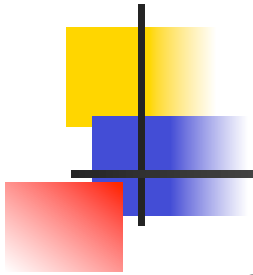
- wait(0) correspond à wait(), donc pour que acquire(0) ait le même comportement que acquire() il faut traiter ce cas particulier



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
- JDK 1.5 fournit au travers de bibliothèques complémentaires certains outils de concurrence directement implantés au niveau de la JVM
  - Pour faire simple : JDK 1.5 fournit un accès plus direct aux fonctions POSIX que le JDK initial …

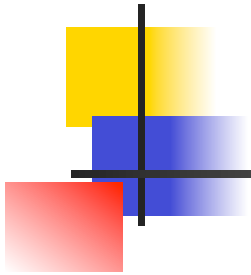
Java
JDK 1.5 ou POSIX en Java



- Semaphore, Verrou, Variable Conditionnelle: implantés correctement directement sur JVM

```
Lock lock = new ReentrantLock();
Condition notFull =lock.newCondition();
Condition notEmpty=lock.newCondition();
void put(Object o) {
   try {
   lock.lock();
   while (count == size)
      notFull.await();
   b[last] = o; count++;
   last=(last+1)%size;
   notEmpty.signal();
  } finally {lock.unlock();}
}
```

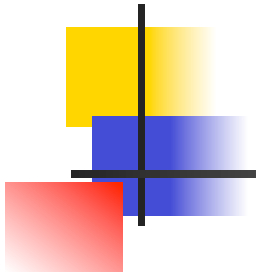
```
Object get(){
   try {
   lock.lock();
   while (count == 0)
      notEmpty.await();
   Object o = b[first];
   first=(first+1)%size;
   count--;
   notFull.signal();
   return o;
   } finally {lock.unlock();}
}
```



Java
JDK 1.5 ou interfaces prédéfinies

- `Callable` : exécute une fonction en asynchrone
- `Future`: renvoie le résultat d'une exécution asynchrone
- `Executor`: alloue et manipule des threads dans un pool
- `Barrier`: bloque tant que # threads bloquées < N
- `Latch`: bloque tant que # ressources > 0 (`join`)
- `Exchanger`: exécute un rendez-vous entre deux tâches

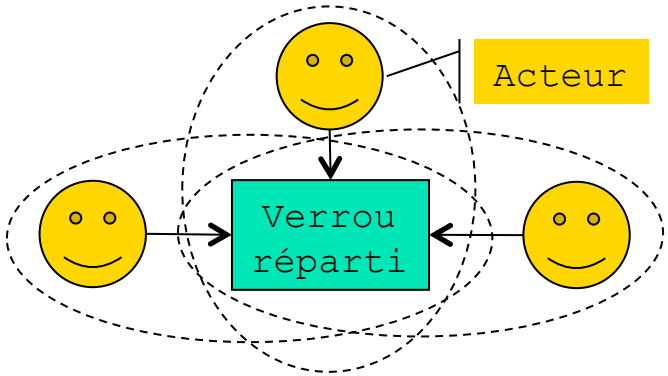
```
ExecutorService executor = ...
void lookUp(String name){
Future<String> future = executor.submit(new Callable<int>() {
  int call() {return operator.lookUp(n);}});
  printSomething();
  print(future.get());}
```



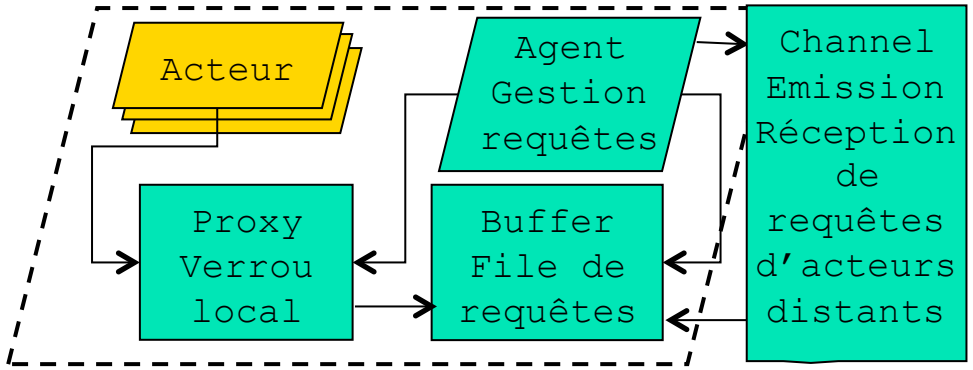
Conclusions

- POSIX offre des mécanismes de concurrence à la sémantique élémentaire et nécessite une écriture parfois ardue de mécanismes plus complexes
- Java offre des mécanismes de concurrence plus évolués qui rendent parfois complexes l'implémentation de constructions simples
  - JDK 1.5 vise à pallier ces difficultés grâce à des bibliothèques complémentaires
- Les mécanismes de Java sont traduits de manière automatique sous forme de constructions de types processus légers en masquant l'hétérogénéité des implémentations de ceux-ci.

Retour sur le cas d'étude

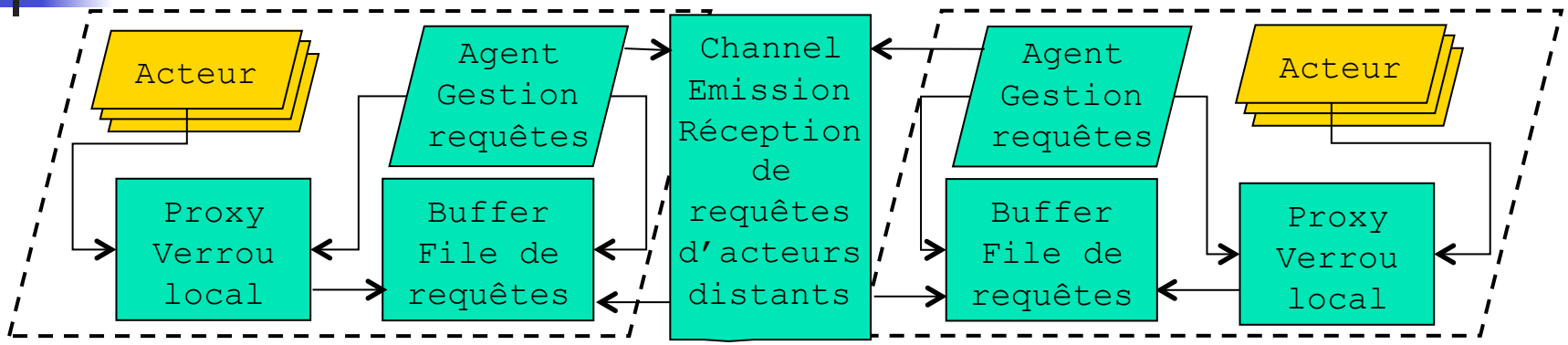


Etude de cas (Java threads) Modélisation des Acteurs



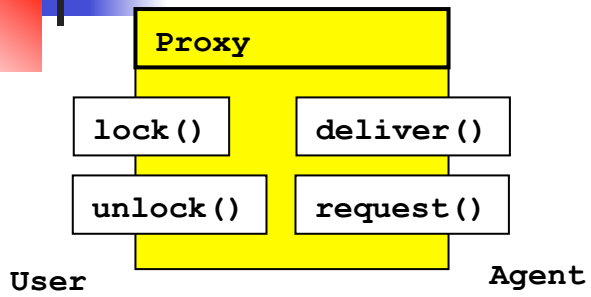
- Tous les acteurs modélisés par des processus légers sont regroupés sur un même processus lourd
- Un acteur prend le verrou, attend un temps aléatoire, rend le verrou pour le recommencer après un temps aléatoire
- Chaque agent A_n modélisé par un processus léger dispose d'un tampon circulaire T_n de requêtes
- Le composant Channel remplit le tampon T_n de requêtes émises vers le nœud *n*
- L'agent A_n consiste en un processus cyclique qui lit les requêtes de T_n, interagit avec le proxy P_n et peut envoyer des requêtes au travers du composant Channel

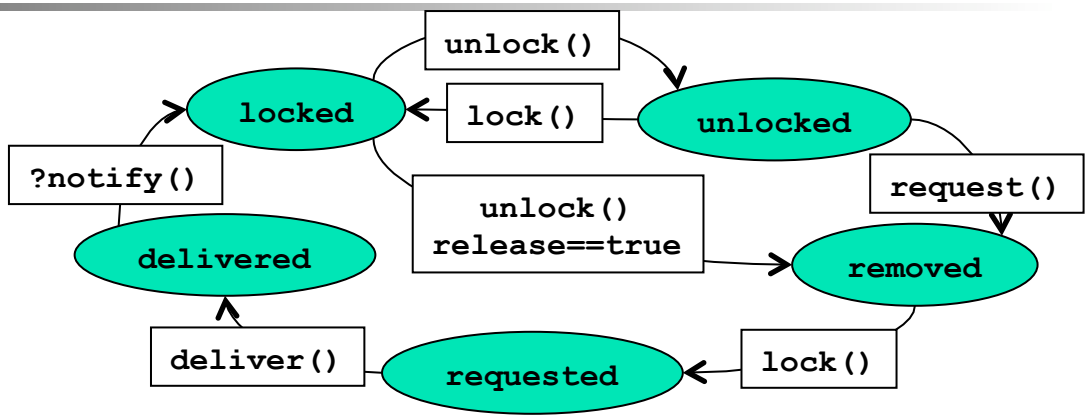
Etude de cas (Java threads) Modélisation de Channel



- Le composant Channel se charge de la communication et disposera de la même interface (*send, recv*) pour nos différents cas d'étude (threads, sockets, CORBA)
- Dans le cas présent, les tampons sont partagés sur le processus lourd et une émission vers *n* se fait par copie de la requête dans le tampon P_n
- Channel dispose de deux fonctions *initialize* pour construire un maillage entre nœuds et *activate* pour autoriser le démarrage du système (opérations triviales ici)

Etude de cas (Java threads) Modélisation de Proxy





- Le proxy doit respecter un enchainement d'états que l'on assurera à l'aide de pré et post conditions
  - Les pré-conditions vérifient que l'on rentre dans une méthode en étant uniquement dans un des états prévus
  - Les post-conditions vérifient que l'on sort d'une méthode en étant uniquement dans un des états prévus
- Les méthodes sont synchronisées pour éviter les accès concurrents et on utilise parfois les mécanismes de wait, notify ou notifyAll pour assurer la progression de l'automate