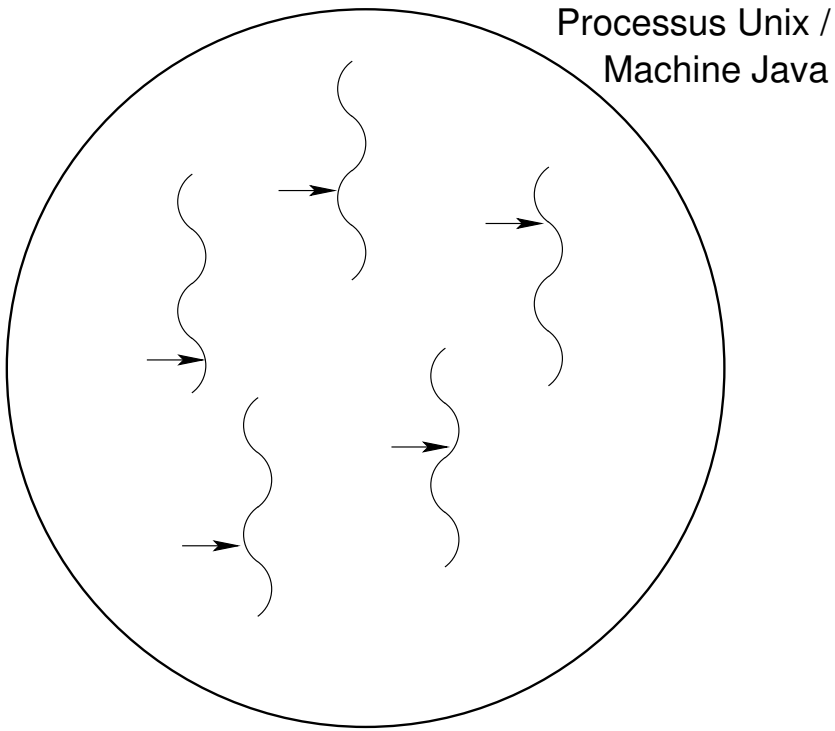
Sixième partie

Programmation multi-activités
Java & Posix Threads

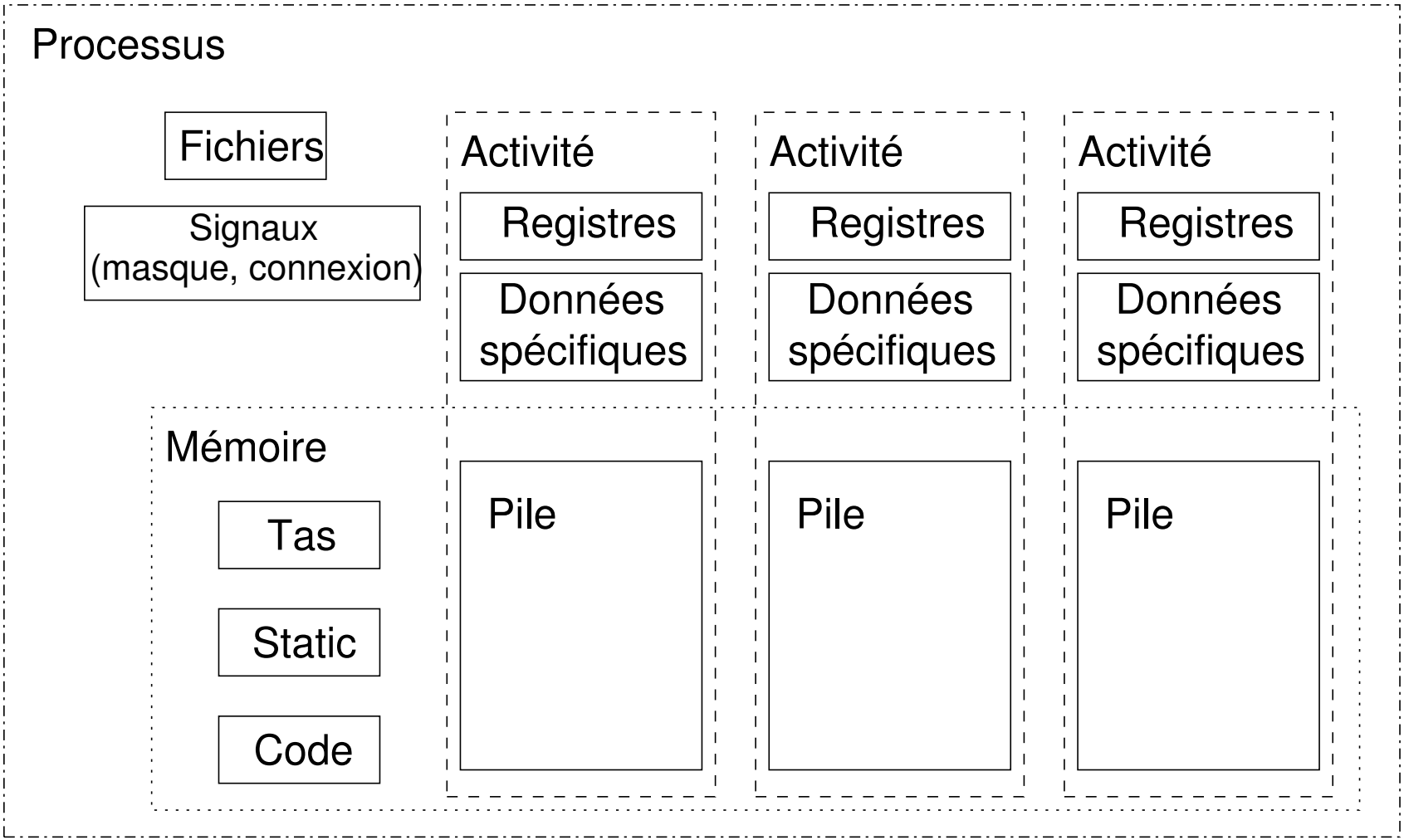
Plan

1. Généralités
2. Java Threads
   - Manipulation des activités
   - Synchronisation (java d'origine)
   - Synchronisation (java moderne)
   - Autres primitives
3. POSIX Threads
   - Manipulation des activités
   - Synchronisation
   - Autres primitives
4. Autres approches
   - Microsoft Windows API
   - Microsoft .NET
   - OpenMP
   - Intel TBB

Processus multi-activités



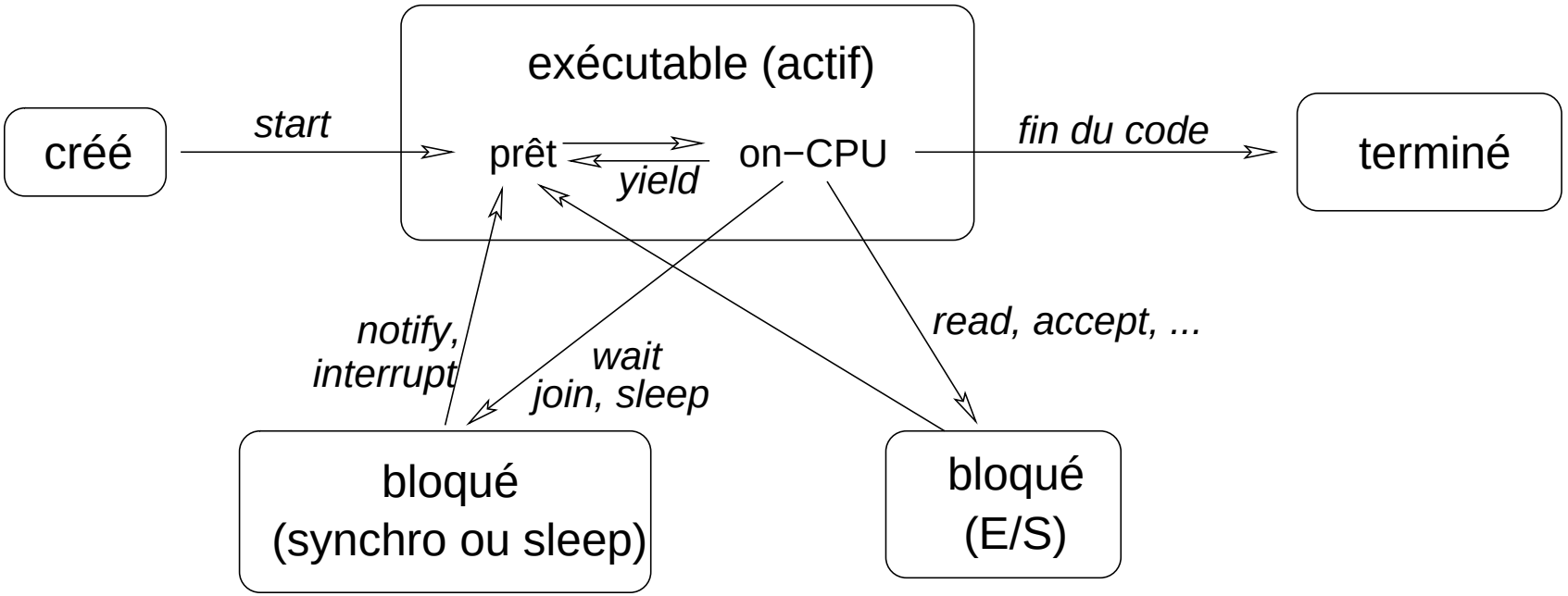
1 espace d'adressage, plusieurs flots de contrôle.
$\Rightarrow$ plusieurs activités (ou processus légers) au sein d'un même processus UNIX / d'une même machine virtuelle Java.


Processus
Fichiers
Signaux
(masque, connexion)
Activité
Registres
Données
spécifiques
Activité
Registres
Données
spécifiques
Activité
Registres
Données
spécifiques
Mémoire
Tas
Static
Code
Pile
Pile
Pile

Plan

1. Généralités
2. Java Threads
   - Manipulation des activités
   - Synchronisation (java d'origine)
   - Synchronisation (java moderne)
   - Autres primitives
3. POSIX Threads
   - Manipulation des activités
   - Synchronisation
   - Autres primitives
4. Autres approches
   - Microsoft Windows API
   - Microsoft .NET
   - OpenMP
   - Intel TBB

Cycle de vie d'une activité



Création d'une activité 1

Implantation de l'interface `Runnable` (méthode `run`)

Définition d'une activité

```
class X implements Runnable {
     public void run() { /* code du thread */ }
}
```

Utilisation

```
X x = new X(...);
Thread t = new Thread(x);    // activité créée
t.start();                   // activité démarrée
:
t.join();                    // attente de la terminaison
```

Création d'activités – exemple

```
class Compteur implements Runnable {
    private int max;
    private int step;
    public Compteur(int max, int step) {
        this.max = max; this.step = step;
    }
    public void run() {
        for (int i = 0; i < max; i += step)
          System.out.println(i);
    }
}

public class DemoThread {
    public static void main (String[] a) {
        Compteur c2 = new Compteur(10, 2);
        Compteur c3 = new Compteur(15, 3);
        new Thread(c2).start();
        new Thread(c3).start();
    }
}
```

Création d'une activité 2

Héritage de la classe `Thread` et implantation de la méthode `run` :

Définition d'une activité

```
class X extends Thread {
     public void run() { /* code du thread */ }
}
```

Utilisation

```
X x = new X();    // activité créée
x.start();        // activité démarrée
...
x.join();         // attente de la terminaison
```

Déconseillé : risque d'erreur de redéfinition de `Thread.run`.

Quelques méthodes

Classe `Thread` :

`static Thread currentThread()`
: obtenir l'activité appelante

`static void sleep(long ms) throws InterruptedException`
: suspend l'exécution de l'activité appelante pendant la durée indiquée (ou jusqu'à ce que l'activité soit interrompue)

`void join() throws InterruptedException`
: suspend l'exécution de l'activité appelante jusqu'à la terminaison de l'activité sur laquelle `join()` est appliquée (ou jusqu'à ce que l'activité appelante soit interrompue)

Interruption

Mécanisme minimal permettant d'interrompre une activité.
La méthode `interrupt` (appliquée à une activité) provoque

soit la levée de l'exception `InterruptedException` si l'activité est bloquée sur une opération de synchronisation (`Thread.join`, `Thread.sleep`, `Object.wait`)

soit le positionnement d'un indicateur `interrupted`, testable par :

`boolean isInterrupted()` qui renvoie la valeur de l'indicateur de l'activité sur laquelle cette méthode est appliquée ;

`static boolean interrupted()` qui renvoie et *efface* la valeur de l'indicateur de l'activité appelante.

Pas d'interruption des entrées-sorties bloquantes ⇒ peu utile.

Synchronisation (Java 1)

Principe

- exclusion mutuelle
- attente/signalement sur un objet
- équivalent à un moniteur avec une seule variable condition

Obsolète : la protection par exclusion mutuelle (`syncronized`) sert encore, mais éviter la synchronisation sur objet et préférer les véritables moniteurs introduits dans Java 5.

Exclusion mutuelle

Tout objet Java est équipé d'un verrou d'exclusion mutuelle.

Code synchronisé

```
synchronized (unObj) {
   < Excl. mutuelle vis-à-vis des autres
     blocs synchronized(unObj) >
}
```

Méthode synchronisée

```
synchronized T uneMethode(...) { ... }
```

(exclusion d'accès de l'objet sur lequel on applique la méthode, pas de la méthode elle-même)

Exclusion mutuelle

Chaque classe possède aussi un verrou exclusif qui s'applique aux méthodes de classe (méthodes statiques) :

```
class X {
     static synchronized T foo() { ... }
     static synchronized T' bar() { ... }
}
```

`synchronized` assure l'exécution en exclusion mutuelle pour toutes les méthodes statiques synchronisées de la classe `X`.
Ce verrou ne concerne pas l'exécution des méthodes d'objets.

Synchronisation par objet

Méthodes `wait` et `notify[All]` applicables à tout objet, pour lequel l'activité a obtenu l'accès exclusif.

`unObj.wait()` libère l'accès exclusif à l'objet et bloque l'activité appelante en attente d'un réveil via une opération `unObj.notify`

`unObj.notify()` réveille une unique activité bloquée sur l'objet, et la met en attente de l'obtention de l'accès exclusif (si aucune activité n'est bloquée, l'appel ne fait rien) ;

`unObj.notifyAll()` réveille toutes les activités bloquées sur l'objet, qui se mettent toutes en attente de l'accès exclusif.

Synchronisation basique – exemple

```
class VeloToulouse {
    private int nbVelos = 0;

    public void prendre() throws InterruptedException {
        synchronized(this) {
            while (this.nbVelos == 0) {
                this.wait();
            }
            this.nbVelos--;
        }
    }

    public void rendre() {
        // assume : toujours de la place
        synchronized(this) {
            this.nbVelos++;
            this.notify();
        }
    }
}
```

Synchronisation basique – exemple

```
class BarriereBasique {
    private final int N;
    private int nb = 0;
    private boolean ouverte = false;
    public BarriereBasique(int N) { this.N = N; }

    public void franchir() throws InterruptedException {
        synchronized(this) {
            this.nb++;
            this.ouverte = (this.nb >= N);
            while (! this.ouverte)
              this.wait();
            this.nb--;
            this.notifyAll();
        }
    }
    public synchronized void fermer() {
        if (this.nb == 0)
          this.ouverte = false;
    }
}
```

Difficultés

- prises multiples de verrous :

```
synchronized(o1) { synchronized(o2) { o1.wait(); } }
```

- une seule notification possible pour une exclusion mutuelle donnée $\rightarrow$ résolution difficile de problèmes de synchronisation

Pas des moniteurs de Hoare !

- programmer comme avec des sémaphores
- affecter un objet de blocage distinct à chaque requête et gérer soit-même les files d'attente

- pas de priorité au signalé, pas d'ordonnancement sur les déblocages

```
class Requête {
    bool ok;
    // paramètres d'une demande
}
List<Requête> file;
```

demande bloquante

```
req = new Requête(...)
synchronized(file) {
   if (satisfiable(req)) {
     // + maj état applicatif
     req.ok = true;
   } else {
     file.add(req)
   }
}
synchronized(req) {
   while (! req.ok)
     req.wait();
```

libération

```
synchronized(file) {
   // + maj état applicatif
   for (Requête r : file) {
     synchronized(r) {
       if (satisfiable(r)) {
         // + maj état applicati
         r.ok = true
         r.notify();
       }
     }
   }
```

Java 5

Nouveau paquetage `java.util.concurrent` avec

- moniteurs
- autres objets de synchronisation (barrière, sémaphore, compteur...)
- contrôle du parallélisme
- structures de données autorisant/facilitant l'accès concurrent

Principe des moniteurs

- 1 verrou assurant l'exclusion mutuelle
- plusieurs variables conditions associées à ce verrou
- attente/signalement de ces variables conditions
- = un moniteur
- pas de priorité au signalé

JDK 5 moniteur

```
import java.util.concurrent.locks.*;
class ProdCon {
  Lock verrou = new ReentrantLock();
  Condition pasPlein  = verrou.newCondition();
  Condition pasVide = verrou.newCondition();
  Object[] items = new Object[100];
  int depot, retrait, nbElems;

  public void deposer(Object x) throws InterruptedException {
        verrou.lock();
        while (nbElems == items.length) pasPlein.await();
        items[depot] = x;
        depot = (depot + 1) % items.length;
        nbElems++;
        pasVide.signal();
        verrou.unlock();
    }
    ⋮
```

Paquetage java.util.concurrent

Sémaphore

```
Semaphore s = new Semaphore(1); // nb init. de jetons
s.acquire();                    // = down
s.release();                    // = up
```

BlockingQueue

`BlockingQueue` = producteurs/consommateurs (interface)

`LinkedBlockingQueue` = prod./cons. à tampon non borné

`ArrayBlockingQueue` = prod./cons. à tampon borné

```
BlockingQueue bq;
bq.put(m);      // dépôt (bloquant) d'un objet en queue
x = bq.take(); // obtention (bloquante) de l'objet en tête
```

Paquetage java.util.concurrent (2)

CyclicBarrier

Rendez-vous bloquant entre N activités : passage bloquant tant que les N activités n'ont pas demandé à franchir la barrière ; passage autorisé pour toutes quand la N-ième arrive.

```
CyclicBarrier barrier = new CyclicBarrier(3);
for (int i = 0; i < 8; i++) {
    Thread t = new Thread() {
        public void run() {
            barriere.await();
            System.out.println("Passé !");
        }};
    t.start();
}
```

Paquetage `java.util.concurrent` (3)

countDownLatch

init(N) valeur initiale du compteur

countDown() décrémente (si strictement positif)

await() bloque si strictement positif, rien sinon.

locks.ReadWriteLock

Verrou acquis soit en mode exclusif (write), soit partagé avec les autres non exclusifs (read). Analogue aux lecteurs/rédacteurs.

Thread pools

- Une file d'attente de travaux à effectuer
- Un ensemble (fixe ou dynamique) d'activités
- Un distributeur des travaux aux activités (`ThreadPoolExecutor`)

Thread pool naïf

```
public class NaiveThreadPool {
  private List<Runnable> queue;
  public NaiveThreadPool(int nthr) {
    queue = new LinkedList<Runnable>();
    for (int i=0; i<nthr; i++) { (new Worker()).start(); }
  }
  public void execute(Runnable job) {
    synchronized(queue) { queue.add(job); queue.notify(); }
  }
  private class Worker extends Thread {
    public void run() {
      Runnable job;
      while (true) {
        synchronized(queue) {
            while (queue.isEmpty()) { queue.wait(); }
            job = queue.remove(0);
        }
        job.run();
      }
  }}
}
```

Thread pool tout aussi naïf

```
import java.util.concurrent.*;
public class NaiveThreadPool2 {
  private BlockingQueue<Runnable> queue;
  public NaiveThreadPool2(int nthr) {
    queue = new LinkedBlockingQueue<Runnable>();
    for (int i=0; i<nthr; i++) { (new Worker()).start(); }
  }

  public void execute(Runnable job) {
    queue.put(job);
  }

  private class Worker extends Thread {
    public void run() {
      while (true) {
        Runnable job = queue.take(); // bloque si nécessaire
        job.run();
      }
  }}
}
```

Thread pool moins naïf

- Nombre minimal d'activités
- Nombre maximal d'activités
- Quand il n'y a plus d'activités disponibles et qu'un travail est déposé, création d'une nouvelle activité (sans dépasser le max)
- Quand la queue est vide et qu'un délai suffisant s'est écoulé, terminaison d'une activité inoccupée (sans dépasser le min)
- Politique de la file :
  - priorité FIFO, autre
  - file bornée ou non bornée
  - travail rejeté ?

Évaluation parallèle (futurs)

Similaire à l'évaluation paresseuse : l'invocation d'une fonction a lieu en retardé (éventuellement en parallèle) avec l'invocateur, qui ne se bloquera que quand il voudra le résultat de l'appel (et si celui-ci n'est pas terminé). Pour garder trace de l'appel, il obtient une valeur future.

```
class MyFunction implements Callable<TypeRetour> {
  public TypeRetour call() { ... return v; }
}

ExecutorService executor = Executors.newCachedThreadPool();
Callable<TypeRetour> fun = new MyFunction();
Futur<TypeRetour> appel = executor.submit(fun);
...
TypeRetour ret = appel.get(); // éventuellement bloquant
```

Fork/join (java 7)

Fork/join basique

```
Résultat résoudre(Problème pb) {
  si (pb est petit) {
     résoudre directement le problème
  } sinon {
     décomposer le problème en parties indépendantes
     fork : créer des sous-tâches (sous-travaux)
                  pour résoudre chaque partie
     join : attendre la réalisation de ces sous-tâches
     fusionner les résultats partiels
  }
```

Rq : tâche/travail (passif) $\neq$ activité/thread (actif)

Fork/join évolué

- Schéma exponentiel
- Coût de la création d'activité
- $\Rightarrow$ Ensemble prédéterminé d'activités (thread pool), chacune équipée d'une file d'attente de travaux à faire.
- Lors d'un fork, ajout en tête de sa file
- Vol de travail : lorsqu'une activité est inactive, elle prend un travail en queue d'une autre file.
- (java 7 n'a qu'une file commune au pool)

Variables localisées

Chaque activité possède sa propre valeur associée à un même objet localisé.

Instances de `ThreadLocal` ou `InheritableThreadLocal`.

```
class MyValue extends ThreadLocal {
    // surcharger éventuellement initValue
}
class Common {
   static MyValue val = new MyValue();
}
```

thread t1	*thread t2*
`o = new Integer(1);`	`o = "machin";`
`Common.val.set(o);`	`Common.val.set(o);`
`x = Common.val.get();`	`x = Common.val.get();`

Plan

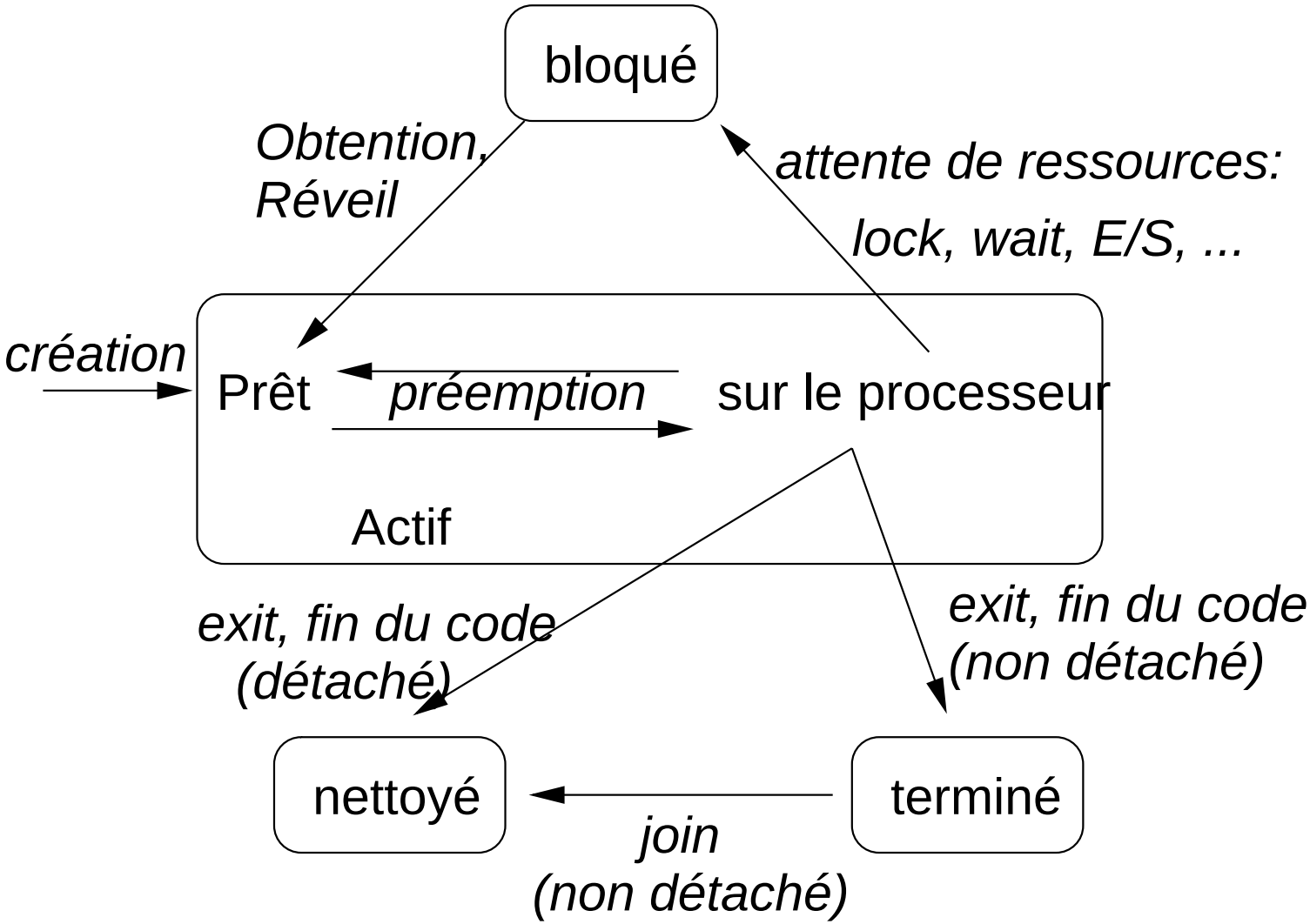
1. Généralités
2. Java Threads
   - Manipulation des activités
   - Synchronisation (java d'origine)
   - Synchronisation (java moderne)
   - Autres primitives
3. POSIX Threads
   - Manipulation des activités
   - Synchronisation
   - Autres primitives
4. Autres approches
   - Microsoft Windows API
   - Microsoft .NET
   - OpenMP
   - Intel TBB

Posix Threads

Standard de librairie multi-activités pour le C, supporté par de nombreuses implantations plus ou moins conformantes.
Contenu de la bibliothèque :

- manipulation d'activités (création, terminaison...)
- synchronisation : verrous, variables condition.
- primitives annexes : données spécifiques à chaque activité, politique d'ordonnancement...
- ajustement des primitives standard : processus lourd, E/S, signaux, routines réentrantes.

Cycle de vie d'une activité



Création d'une activité

```
int pthread_create (pthread_t *thread,
                    const pthread_attr_t *attr,
                    void * (*start_routine)(void *),
                    void *arg);
```

Crée une nouvelle activité pour exécuter la routine indiquée, appelée avec l'argument `arg`. Les attributs sont utilisés pour définir la priorité et la politique d'ordonnancement (scheduling policy). `thread` contient l'identificateur de l'activité créée.

```
pthread_t pthread_self (void);
int pthread_equal (pthread_t thr1, pthread_t thr2);
```

`self` renvoie l'identificateur de l'activité appelante.
`pthread_equal` : vrai si les arguments désignent la même activité.

Terminaison

```
void pthread_exit (void *status);
```

Termine l'activité appelante en fournissant un code de retour.
`pthread_exit(NULL)` est automatiquement exécuté en cas de terminaison du code de l'activité sans appel de `pthread_exit`.

```
int pthread_join (pthread_t thr, void **status);
```

Attend la terminaison de l'activité et récupère le code retour.
L'activité ne doit pas être détachée ou avoir déjà été « jointe ».

Terminaison – 2

```
int pthread_detach (pthread_t thread);
```

Détache l'activité `thread`.
Les ressources allouées pour l'exécution d'une activité (pile...) ne sont libérées que lorsque l'activité s'est terminée et que :

- *ou* `join` a été effectué,
- *ou* l'activité a été détachée.

L'activité initiale

Au démarrage, une activité est automatiquement créée pour exécuter la procédure `main`. Elle exécute une procédure de démarrage qui contient le code :

```
{ int r = main(argc,argv); exit(r); }
```

Si la procédure `main` se termine, le process unix est ensuite terminé (par l'appel à `exit`), et non pas seulement l'activité initiale. Pour éviter que la procédure `main` ne se termine alors qu'il reste des activités :

- bloquer l'activité initiale sur l'attente de la terminaison d'une ou plusieurs autres activités (`pthread_join`) ;
- terminer explicitement l'activité initiale avec `pthread_exit`, ce qui court-circuite l'appel de `exit`.

Synchronisation

Principe

Moniteur de Hoare élémentaire avec priorité au signaleur :

- verrous
- variables condition
- pas de transfert du verrou à l'activité signalée

Verrou

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_init (pthread_mutex_t *mutex,
                        const pthread_mutex_attr *attr);

int pthread_mutex_destroy (pthread_mutex_t *m);
```

Verrouillage/déverrouillage

```
int pthread_mutex_lock (pthread_mutex_t *m);
int pthread_mutex_trylock (pthread_mutex_t *m);
int pthread_mutex_unlock (pthread_mutex_t *m);
```

`lock` verrouille le verrou, avec blocage en attente si déjà verrouillé. Renvoie 0 si ok.

`trylock` verrouille le verrou si possible et renvoie 0, sinon renvoie `EBUSY` si le verrou est déjà verrouillé.

`unlock` déverrouille. Seule l'activité qui a verrouillé `m` a le droit de le déverrouiller.

Variable condition

```
pthread_cond_t vc = PTHREAD_COND_INITIALIZER;

int pthread_cond_init (pthread_cond_t *vc,
                       const pthread_cond_attr *attr);

int pthread_cond_destroy (pthread_cond_t *vc);
```

Attente/signal

```
int pthread_cond_wait (pthread_cond_t*,
                       pthread_mutex_t*);
int pthread_cond_timedwait (pthread_cond_t*,
                    pthread_mutex_t*,
                    const struct timespec *abstime);
```

`cond_wait` l'activité appelante doit posséder le verrou spécifié. L'activité se bloque sur la variable condition après avoir libéré le verrou. L'activité reste bloquée jusqu'à ce que `vc` soit signalée et que l'activité ait réacquis le verrou.

`cond_timedwait` comme `cond_wait` avec délai de garde. À l'expiration du délai de garde, le verrou est reobtenu et la procédure renvoie `ETIMEDOUT`.

Attente/signal

```
int pthread_cond_signal (pthread_cond_t *vc);
int pthread_cond_broadcast (pthread_cond_t *vc);
```

`cond_signal` signale la variable condition : une activité bloquée sur la variable condition est réveillée et tente de réacquérir le verrou de son appel de `cond_wait`. Elle sera effectivement débloquée quand elle le réacquerra.

`cond_broadcast` toutes les activités en attente sont réveillées, et tentent d'obtenir le verrou correspondant à leur appel de `cond_wait`.

Ordonnancement

Par défaut : ordonnancement arbitraire pour l'acquisition d'un verrou ou le réveil sur une variable condition.

Les activités peuvent avoir des priorités, et les verrous et variables conditions peuvent être créés avec respect des priorités.

Données spécifiques

Données spécifiques

Pour une clef donnée (partagée), chaque activité possède sa propre valeur associée à cette clef.

```
int pthread_key_create (pthread_key_t *clef,
                        void (*destructeur)(void *));

int pthread_setspecific (pthread_key_t clef,
                         void *val);
void *pthread_getspecific (pthread_key_t clef);
```

Plan

1. Généralités
2. Java Threads
   - Manipulation des activités
   - Synchronisation (java d'origine)
   - Synchronisation (java moderne)
   - Autres primitives
3. POSIX Threads
   - Manipulation des activités
   - Synchronisation
   - Autres primitives
4. Autres approches
   - Microsoft Windows API
   - Microsoft .NET
   - OpenMP
   - Intel TBB

Windows API (C, C++)

Plus de 150 (?) fonctions, dont :

- création d'activité : `CreateThread`
- exclusion mutuelle : `InitializeCriticalSection`, `EnterCriticalSection`, `LeaveCriticalSection`
- synchronisation basique : `WaitForSingleObject`, `WaitForMultipleObjects`, `SetEvent`
- synchronisation « évoluée » : `SleepConditionVariableCS`, `WakeConditionVariable`

Note : l'API Posix Threads est aussi supportée (ouf).

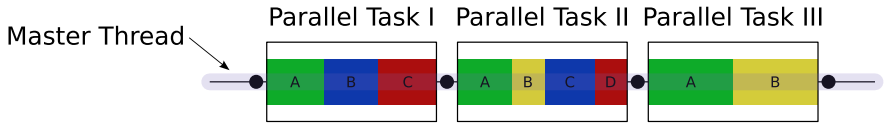
.NET (C#)

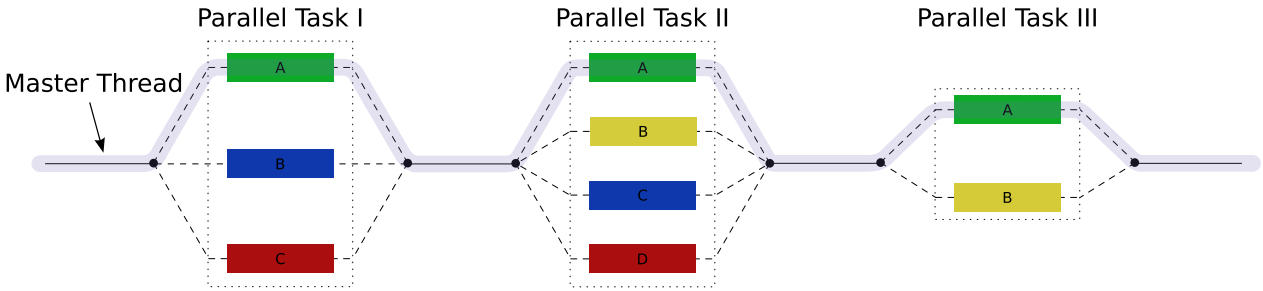
Très similaire à Java :

- Création d'activité :
  `t = new System.Threading.Thread(méthode);`
- Démarrage : `t.Start();`
- Attente de terminaison : `t.Join();`
- Exclusion mutuelle : `lock(objet) { ... }`
  (mot clef du langage)
- Synchronisation élémentaire :
  `System.Threading.Monitor.Wait(objet);`
  `System.Threading.Monitor.Pulse(objet);` (= notify)
- Sémaphore :
  `s = new System.Threading.Semaphore(nbinit,nbmax);`
  `s.Release(); s.WaitOne();`

OpenMP

- API pour la programmation parallèle en C/C++/Fortran





- Annotations dans le code, interprétées par le compilateur

Boucle parallèle

```
int i, a[N];
#pragma omp parallel for
for (i = 0; i < N; i++)
    a[i] = 2 * i;
```

OpenMP avantages/inconvénients

+ simple
+ amélioration progressive du code
+ une seule version séquentielle / parallèle
+ peu de modifications sur le code séquentiel d'origine
– exclusivement multiprocesseurs à mémoire partagée
– compilateur dédié
– peu de primitives de synchronisation (atomicité uniquement)
– gros travail sur du code mal conçu
– introduction de bugs en parallélisant du code non parallélisable

Intel Threading Building Blocks

- Bibliothèque pour C++
- Structures de contrôles optimisées `parallel_for`...
- Structures de données optimisées `concurrent_queue`...
- Peu de primitives de synchronisation (exclusion mutuelle, verrou lecteurs/rédacteurs)
- Implantation spécialisée par modèle de processeur
- Partage de tâches par « vol de travail »
- Inconvénient : portabilité (compilateur + matériel)

Message Passing Interface

- Originellement, pour le calcul haute performance sur clusters de supercalculateurs
- D'un point de vue synchronisation, assimilable aux processus communicants
- Vu en 3ème année