Sixième partie

Programmation multi-activités

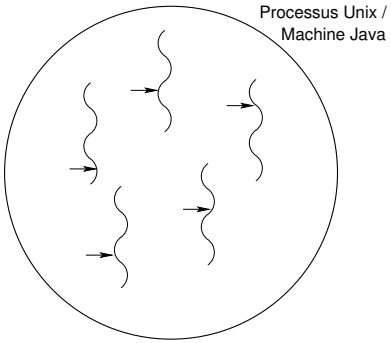
Contenu de cette partie

Préparation aux TPs : présentation des outils de programmation concurrente autour de la plateforme Java

- notion de processus léger
- présentation de la plateforme
- classe `Thread`
- objets de synchronisation : moniteurs, sémaphores...
- régulation des activités : pools d'activités, appels asynchrones, fork/join...
- outils de synchronisation de bas niveau
- autres environnements et modèles : Posix, OpenMP...

Plan

Processus multi-activités

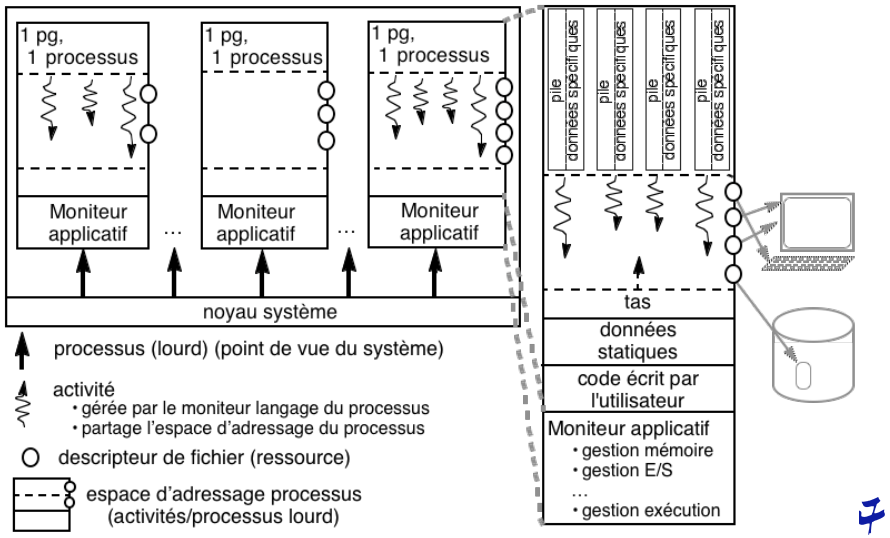


1 espace d'adressage, plusieurs flots de contrôle.
⇒ plusieurs activités (ou processus légers) au sein d'un même processus UNIX / d'une même machine virtuelle Java.

Relation et différences entre processus lourds et légers

- *Processus lourds* : représentent l'exécution d'une application, du point de vue du système
  - → unité d'allocation de ressources
    - espaces d'adressage et ressources distinctes (pas de partage)
    - commutation coûteuse (appels systèmes → passage par le mode superviseur)
- *Processus légers* (threads, activités...) :
  - unité d'exécution : résulte de la décomposition (fonctionnelle) d'un traitement en sous-traitements parallèles, pour tirer profit de la puissance de calcul disponible, ou simplifier la conception
  - les ressources (mémoire, fichiers...) du processus lourd exécutant un traitement sont partagées entre les activités réalisant ce traitement
    - chaque activité a sa pile d'exécution et son contexte processeur, mais les autres éléments sont partagés
  - une bibliothèque applicative (≪ moniteur ≫) gère le partage entre activités du temps processeur alloué au processus lourd → commutation plus efficace.

Mise en œuvre des processus légers



Difficultés de mise en œuvre des processus légers

L'activité du moniteur applicatif est opaque pour le système d'exploitation : le moniteur du langage multiplexe les ressources d'un processus lourd entre ses activités, sans appel au noyau.

→ commutation de contexte plus légère, mais
- appels système usuellement bloquants
  → 1 activité bloquée ⇒ toutes les activités bloquées
  - → utiliser des appels systèmes non bloquants (s'ils existent) au niveau du moniteur applicatif, et gérer l'attente,
- réaction aux événements asynchrones a priori ≪ lente ≫
  - → définir 1 service d'événements au niveau du moniteur applicatif, et utiliser (si c'est possible) le service d'événements système

Remarque : la mise en œuvre des processus légers est directe lorsque le système d'exploitation fournit un service d'activités noyau et permet de coupler activités noyau et activités applicatives

Conception d'applications parallèles en Java

Java permet de manipuler
- les processus (lourds) : classes `java.lang.ProcessBuilder` et `java.lang.Process`
- les activités (processus légers) : classe `java.lang.Thread`

Le degré de parallélisme des applications Java peut être
- contrôlé directement (manipulation des threads)
- ou régulé
  - explicitement : interface `java.util.concurrent.Executor`
  - implicitement : environnement proposé par Java 8 pour la programmation asynchrone/fonctionnelle/événementielle (abordé plus tard)

Plan

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 (Thread)

1/Définir une classe implantant l'interface Runnable (méthode run)

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

2/Utiliser 1 instance de cette classe Runnable pour créer un Thread

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

Remarque : il est aussi possible de créer une activité par héritage de la classe Thread et implantation de la méthode run

Quelques méthodes

Classe Thread :

`static Thread currentThread()`
: fournit (la référence à) l'activité appelante

`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)

`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)

Complément : interruption

Mécanisme élémentaire permettant de signaler un événement asynchrone

- La méthode `interrupt` (appliquée à une activité) positionne un indicateur `interrupted`, testable par :
  `boolean isInterrupted()` qui renvoie la valeur de l'indicateur `interrupted` 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.
- Si l'activité sur laquelle s'applique `interrupt` est bloquée sur une opération de synchronisation qui lève l'exception `InterruptedException` (`Thread.join`, `Thread.sleep`, `Object.wait`...), celle-ci est levée, et `interrupted` est effacé.

Pas d'interruption des entrées-sorties bloquantes → intérêt limité.

Complément : variables localisées

Permet de définir un contexte d'exécution local, non partagé : chaque activité possède sa propre valeur associée à un objet localisé (qui est instance de `ThreadLocal` ou `InheritableThreadLocal`).

```
class Common {
   static ThreadLocal val = new ThreadLocal(); //attribut localisé par thread
   static Integer v = new Integer(0);          // attribut global ''standard''}
```

thread t1 : incrémente v et val

```
Integer o =  new Integer(0);
Integer x = new Integer(0);
Integer y = new Integer(1);
...
Common.val.set(o);
for (int i = 0; i <= 2; i++){
x = (Integer) Common.val.get();
o = Integer.valueOf(x.intValue()+1);
Common.val.set(o);
y = Common.v;
Common.v = Integer.valueOf(y.intValue()+1);
System.out.println("T1 - G: "+y+" / TL: "+x);}
```

thread t2 : incrémente v, autoconcatène val

```
String o = "bip ";
String x ;
Integer y = new Integer(1);
...
Common.val.set(o);
for (int i = 0; i <= 2; i++) {
x = (String) Common.val.get();
o=o+x;
Common.val.set(o);
y = Common.v;
Common.v = Integer.valueOf(y.intValue()+1);
System.out.println("T2 - G: "+y+" / TL: "+x);}
```

```
Résultat :
T1 - G: 0 / TL: 0
T2 - G: 1 / TL: bip
T1 - G: 2 / TL: 1
T2 - G: 3 / TL: bip bip
T1 - G: 4 / TL: 2
T2 - G: 5 / TL: bip bip bip bip
```

Plan

Le paquetage `java.util.concurrent` fournit

- une réalisation des moniteurs
- divers autres objets de synchronisation
  - barrière
  - sémaphore
  - compteur
  - ...
- le contrôle du degré de parallélisme : `Thread`, `Executor`
- des structures de données permettant des accès concurrents (collections « concurrentes ») de manière transparente
  - accès atomiques : `ConcurrentHashMap`...
  - accès non bloquants : `ConcurrentLinkedQueue`

Moniteur Java (5)

- un verrou assurant l'exclusion mutuelle (équité possible)
- variables conditions associées à ce verrou
- pas de priorité au signalé et pas de file des signalés

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

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


Objets de synchronisation (1/3)

Sémaphore

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

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
```


Objets de synchronisation (2/3)

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 barriere = 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();
}
```

Généralisation : la classe `Phaser` permet un rendez-vous (bloquant ou non) pour un *groupe variable* d'activités.

Objets de synchronisation (3/3)

countDownLatch

`init(N)` valeur initiale du compteur

`await()` bloque si strictement positif, rien sinon.

`countDown()` décrémente (si strictement positif). Lorsque le compteur devient nul, toutes les activités bloquées sont débloquées.

interface locks.ReadWriteLock

Fournit des verrous pouvant être acquis en mode

- exclusif (méthode `writeLock()`),
- ou partagé avec les autres non exclusifs (méthode `readLock()`)

→ mise en œuvre du schéma lecteurs/rédacteurs.

→ implémentation : `ReentrantReadWriteLock` (avec/sans équité)

Services de régulation du parallélisme : exécuteurs

Idée

Séparer la création et la gestion des activités des autres aspects (fonctionnels, synchronisation...)

→ définition d'un service de gestion des activités (exécuteur), régulant/adaptant le nombre de threads effectivement actifs, en fonction de la charge courante et du nombre de processeurs physiques disponibles :

- trop de threads → consommation de ressources inutile
- pas assez de threads → capacité de calcul sous-utilisée

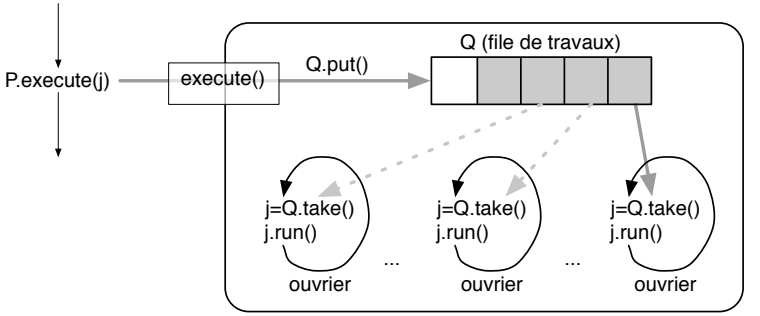
Interfaces d'exécuteurs

- l'interface `Executor` définit la méthode `execute(Runnable r)`,
  - fonctionnellement équivalente à `(new Thread(r)).start()`,
  - avec la différence que `r` ne sera pas forcément exécutée immédiatement/par un thread spécifique.
- la sous-interface `ExecutorService` permet de soumettre (méthode `submit(...)`) des tâches rendant un résultat (`Callable`), lequel pourra être récupéré par la suite, de manière asynchrone.
- l'interface `ScheduledExecutorService` est un `ExecutorService`, avec la possibilité de spécifier un calendrier (départs, périodicité...) pour les tâches exécutées.

Pools de Threads

Schéma de base pour la plupart des implémentations d'exécuteurs

- Une file d'attente de travaux à effectuer
- Un ensemble (fixe ou dynamique) d'activités (ouvriers)
- Une politique de distribution des travaux aux activités (réalisée par un protocole ou par une activité)



Pool P *[sans politique de distribution particulière (file partagée)]*

Principe de fonctionnement : pool de threads « minimal »

```
import java.util.concurrent.*;
public class PlainThreadPool {
  private BlockingQueue<Runnable> queue;
  public PlainThreadPool(int nthr) {
    queue = new LinkedBlockingQueue<Runnable>();
    for (int i=0; i<nthr; i++) { (new Ouvrier()).start(); }
  }
  public void execute(Runnable job) {
    queue.put(job);
  }
  private class Ouvrier extends Thread {
    public void run() {
      while (true) {
        Runnable job = queue.take(); //bloque si nécessaire
        job.run();
      }
    }
  }
}
```

Exécuteurs implantés par des pools de threads prédéfinis

La classe `java.util.concurrent.Executors` est une fabrique pour des stratégies d'exécution classiques :

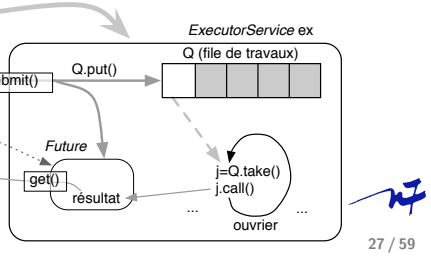
- Nombre fixe d'activités : méthodes `newSingleThreadExecutor()`, `newFixedThreadPool(int nThreads)`
- Nombre d'activités adaptable : méthode `newCachedThreadPool()`
  - Quand il n'y a plus d'activité disponible et qu'un travail est déposé, création d'une nouvelle activité
  - Quand la queue est vide et qu'un délai suffisant (p.ex. 1mn) s'est écoulé, terminaison d'une activité inoccupée
  - Possibilité de définir un calendrier pour la création/activation des threads du pool

La classe `java.util.concurrent.ThreadPoolExecutor` permet de contrôler l'ensemble des paramètres de la stratégie d'exécution : politique de la file (FIFO, priorités...), file bornée ou non bornée, nombre de threads minimum, maximum...

Évaluation asynchrone : `Callable` et `Future`

- Evaluation paresseuse : l'appel effectif d'une fonction peut être différé (éventuellement exécuté en parallèle avec l'appelant)
- `submit(...)` fournit à l'appelant une référence à la valeur future du résultat.
- L'appelant ne se bloque que quand il doit utiliser le résultat de l'appel (si l'évaluation de celui-ci n'est pas terminée).
  → appel de la méthode `get()` sur le `Future`

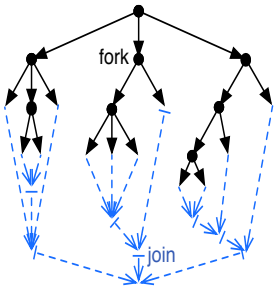
```
class FonctionAsynchrone implements Callable<TypeRetour> {
    public TypeRetour call() { ... return v; }
}
ExecutorService ex = Executors.newCachedThreadPool();
Callable<TypeRetour> fonc = new FonctionAsynchrone();
...
Future<TypeRetour> appel = ex.submit(fonc);
...
TypeRetour ret = appel.get(); //éventuellement bloquant
```



Un peu de Big Data : schéma diviser pour régner (fork/join, map/reduce)

Schéma de base

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

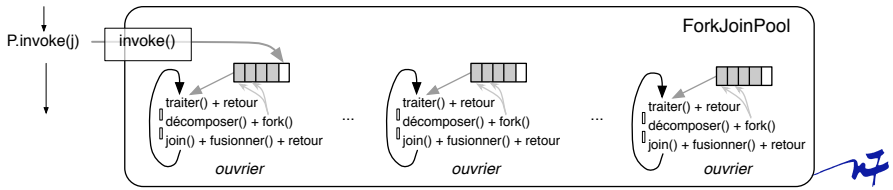


Exécuteur pour le schéma fork/join (1/3)

Difficulté de la stratégie diviser pour régner :
schéma exponentiel + coût de la création d'activités

Classe `ForkJoinPool`

- Ensemble prédéterminé (pool) d'activités, chacune équipée d'une file d'attente de travaux à traiter.
- Les activités gérées sont des instances de `ForkJoinTask` (méthodes `fork()` et `join()`)



Exécuteur pour le schéma fork/join (2/3)

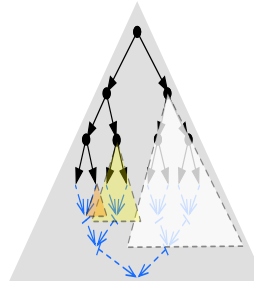
Activité d'un ouvrier du ForkJoinPool

- Un ouvrier traite la tâche placée en tête de sa file
- Un ouvrier appelant `fork()` ajoute les travaux créés en tête de sa propre file

→

Chaque ouvrier traite un arbre de tâches qu'il

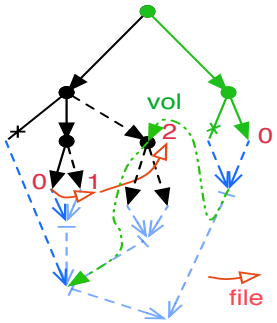
- parcourt (et traite) en profondeur d'abord (en préordre) → économie d'espace
- construit progressivement en largeur, au fur et à mesure de son parcours : lorsqu'un ouvrier descend d'un niveau, les frères de la tâche à traiter sont créés, et placés en tête de la file d'attente



Exécuteur pour le schéma fork/join (3/3)

Régulation

Vol de travail : lorsqu'une activité a épuisé les travaux de sa file, elle prend un travail en queue d'une autre file



La tâche prise correspond au dernier sous-arbre (le plus proche de la racine) qui était affecté à l'ouvrier « volé »

→ pas de conflits si les sous-problèmes sont bien partitionnés

→ pas d'attente inutile pour l'ouvrier « volé » puisque la tâche volée était la dernière à traiter.

Services de synchronisation élémentaire

Fonctions

- exclusion mutuelle
- attendre/signaler un événement

  analogue à un moniteur (priorité au signaleur, sans file des signalés) avec une seule variable condition
- accès élémentaires atomiques

Exclusion mutuelle

Toute instance d'objet Java est munie d'un verrou exclusif

Code synchronisé

```
synchronized (unObj) {
        < Région critique >
}
```

Méthode synchronisée

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

Remarques

- exclusion d'accès de l'objet sur lequel on applique la méthode, pas de la méthode elle-même
- les méthodes de classe (statiques) peuvent aussi être synchronisées : une classe est une instance de la classe `Class`.

Synchronisation associée aux verrous d'objets

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 seule activité bloquée sur l'objet (si aucune activité n'est bloquée, l'appel ne fait rien) ;

`unObj.notifyAll()` réveille toutes les activités bloquées sur l'objet.

Difficultés

- prises multiples de verrous :

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

- pas de priorité au signalé, pas d'ordonnancement sur les déblocages
- une seule notification possible pour une exclusion mutuelle donnée
  → résolution compliquée de problèmes de synchronisation
  - programmer comme avec des sémaphores
  - affecter un objet de blocage distinct à chaque requête et gérer soi-même les files d'attente

Schéma de base

```
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 applicatif
         r.ok = true
         r.notify();
       }
     }
   }
}
```

Atomicité à grain fin

Java fournit des outils pour faciliter la conception d'algorithmes concurrents à grain fin, c'est-à-dire où la coordination sera réalisée par l'accès à des données partagées, plutôt que par suspension/réveil (attente/signal d'événement)

- les lectures et les écritures des références et de la plupart des types primitifs (`long` et `double` exceptés) sont atomiques
- idem pour les variables déclarées `volatile`
- le paquetage `java.util.concurrent.atomic` fournit des classes qui permettent des accès atomiques et offrent en outre des opérations de mise à jour conditionnelle du type `TestAndSet`

Rappel (et mise en garde)

Concevoir et valider des algorithmes de ce type est très ardu. Cette difficulté même a motivé la définition d'objets et de méthodologies de synchronisation (sémaphores,...)

Plan

Posix Threads

Standard de librairie multi-activités, supporté par de nombreuses implantations plus ou moins conformes
(SUN/Solaris 2.5, Linux, FreeBSD, HP-UX 11.0...)
Nom officiel : POSIX 1003.1-1996.
Repris dans X/Open XSH5 .
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.

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.

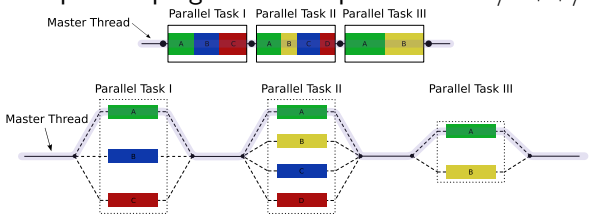
.NET (C#)

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 si code mal conçu
– introduction de bugs en parallélisant du code non parallélisable

Intel Threading Building Blocks

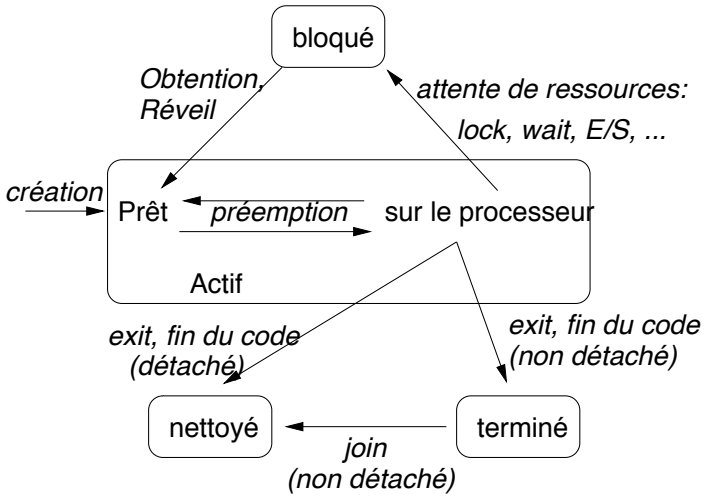
- Bibliothèque pour C++
- Structures de contrôle optimisées `parallel_for`...
- Structures de données optimisées `concurrent_queue`...
- Peu de primitives de synchronisation (exclusion mutuelle, verrou lecteurs/rédacteurs)
- Implantation spécialisée par modèle de processeur
- Partage de tâches par ≪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

Message Passing Interface (MPI)

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

Plan

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 thr);
```

Détache l'activité `thr`.
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 processus 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 réobtenu 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);
```