Les threads : introduction

Bertrand Dupouy

Plan

Plan

– Présentation

Historique
Définitions
Implantations
Apports
Problèmes

– Ordonnancement

– Synchronisation

Verrous
Variables conditionnelles
Sémaphores

Pourquoi les threads ?

• Inconvénients du processus classique:

– Changement de contexte long (notamment pour les applications du type "temps réel" ou "multi média"),
– Pas de partage de mémoire (communications lentes, pb dans le cas des architectures *Symetric multi processor* ou SMP)
– Manque d'outils de synchronisation
– Interface rudimentaire (`fork, exec,exit,wait`)

• Pourquoi le changement de contexte est-il "long":

- 90% de ce temps est consacré à la gestion de la mémoire,

• On introduit une nouvelle forme de processus : ceux-ci **partagent** la mémoire, ainsi on **résout aussi celui du changement de contexte « long »**

• Ce nouveau type d'activité s'appelle un *thread*

• Les threads ne rendent pas obsolètes les processus classiques en particulier dans le cas où la séparation des espaces d'adressage entre applications s'impose, pour des raisons de sécurité, par exemple.

Terminologie

- Thread : fil en anglais, un thread est un fil d'exécution, plusieurs traductions :
  - activité,
  - fil d'exécution,
  - *lightweight process* (*lwp*) ou processus léger (par opposition au processus créé par fork, qualifié de processus lourd),
- Par la suite nous utiliserons le terme « thread » ou « processus léger »

Définitions

A un instant donné, un processus classique, tel le processus Unix créé par `fork`, ne comporte qu'un seul fil d'exécution, un seul **thread**.

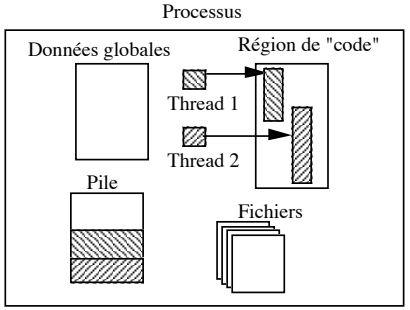
Les threads permettent de dérouler plusieurs suites d'instructions, en PARALLELE, à l'intérieur du même processus.

- Le processus devient la structure d'ALLOCATION des ressources (fichiers, mémoire) pour les threads. Ces ressources, en particulier **l'espace d'adressage,** peuvent être partagées par plusieurs threads,
- On peut donc envisager les threads comme des processus qui partagent toutes leurs ressources, sauf la pile et quelques registres (compteur ordinal, pointeur de pile).

Thread et processus

• Un **thread** exécute une **fonction**. Donc, un thread :

- ne "voit" qu'une partie de la région de code du processus qui l'héberge,
- dispose de sa propre pile pour implanter les variables locales,



- partage les données globales avec les autres threads.

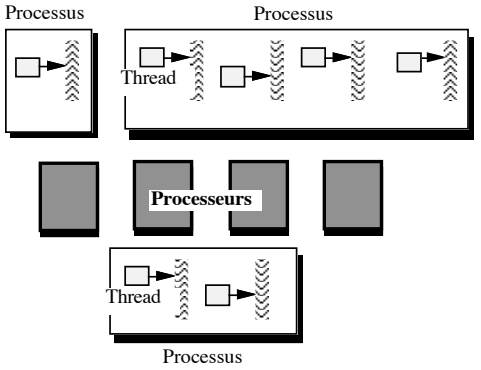
• Attention :

Changement de contexte court et partage de mémoire ne sont vrais qu'entre threads créés dans le **même processus .**

Multithreading Multiprocesseur

• Multiprocesseur : plusieurs processeurs partagent de la mémoire. On parle de SMP (*symetric multi processor*) dans le cas d'un ordinateur doté d'une mémoire unique et de plusieurs processeurs

• Multithreading : plusieurs processus légers (threads) partagent des ressources, en particulier la mémoire, pour s'exécuter en parallèle. Si on dispose de plusieurs processeurs, vrai parallélisme; sinon parallélisme logique du multitâches.



Exemple 1

• Les fonctions `f` et `g` mettent à jour deux parties **indépendantes** d'un tableau `Tab` (exemple idéal d'utilisation des threads …) :

```
int Tab[100,100];

void main (void)
{
  …
  f();
  g();
  …
}
```

On veut paralléliser le traitement, deux solutions :

1- fork : deux processus indépendants exécutent l'un f, l'autre g.

Pb: duplication des zones de mémoire, il faut passer par un fichier ou de la mémoire partagée. Dans ce cas la séparation des espaces d'adressage est pénalisante.

2- thread : thread (f) thread(g) : transformation simple du code, parce que, ici, il n'y a pas à gérer de synchronisation.

Utilisation de fork	Utilisation de thread

…
f=fork();
if (f == 0) { … f(); … }
f=fork();
if (f == 0) { … g(); … }
…
	…
	pthread_create (f...);
	pthread_create (g...);
	…

Exemple 2

• Parallèlisation simple des entrées-sorties :

• Chaque thread fait un appel bloquant, mais chaque appel est indépendant des autres :

Programmation classique	Programmation threads

`read(periph1, …);`
`read(periph2, …);`
`read(periph3, …);`
sauf option spéciale, chacun des appels bloque le suivant, empêchant le parallélisme des exécutions, imposant ainsi un **ordre**.
	`pthread_create (read_periph1...);`
	`pthread_create (read_periph2...);`
	`pthread_create (read_periph3...);`
	les trois requêtes se font en parallèle, sans ordre.

• serveur réseau : certains serveurs se parallélisent plus facilement avec des threads qu'avec un appel à `fork` (**`nfsd`**).

Implantation

- Un thread peut être implanté :
  - au niveau du NOYAU, il est alors ordonnancé indépendamment du processus dans lequel il a été créé,
  - au niveau du PROCESSUS qui l'accueille, il accède alors au processeur dans les quanta alloués à ce processus
- Dans le premier cas, le thread est l'unité d'ordonnancement. Quand un processus est lancé, on exécute en fait un thread associé à `main`
- La seconde méthode sollicite moins le noyau (pas d'appel à celui-ci pour les changements de contexte), mais :
  - problème de gestion des E/S qui vont bloquer tout le processus (les appels systèmes bloquants doivent être redéfinis)
  - pas de réel contrôle sur l'ordonnancement, en particulier dans le cas des MP
- On présentera ici l'API POSIX. Celle-ci est proposée par la plupart des systèmes , en particulier par SunOS.

Principaux apports

Aspects système :

- la mémoire est partagée donc le changement de contexte est simple : il suffit de commuter quelques registres (d'où le synonyme *lightweight process*, processus léger).
- fonctionnement du noyau en parallèle sur plusieurs processeurs,

Aspects utilisateur :

- Découpage facile de l'application en activités parallèles, donc utilisation simple des différentes unités d'un multiprocesseur,
- Les threads permettent de passer d'un modèle de programmation asynchrone à modèle synchrone (les e/s bloquantes sont gérées par des threads).
- API puissante : outils de synchronisation variés,…

Aspects langage:

- Les threads permettent d'implanter le threading java, le tasking Ada

Notions de base

• Voici les ressources propres à un thread :

- un identificateur (le *thread identifier*, ou `tid`, équivalent du `pid`),
- une priorité,
- une configuration de registres, une pile
- un masque de signaux,
- d'éventuelles données privées,

• Le nombre et l'identité des threads d'un processus sont invisibles depuis un autre processus,

Attention : à l'utilisation par les threads des appels concernant tout le processus comme `exit`, ...

Principales fonctions de Manipulation

Nom de la fonction (POSIX)	Rôle de la fonction
`pthread_create (...)`	**Création** d'un thread. Par défaut, tous les threads ont la même priorité. Le moment de son démarrage dépend de sa priorité. On peut changer sa priorité.
`pthread_exit (...)`	**Termine le** thread, et le thread seulement, à la différence de exit qui termine le processus et tous ses threads.
`pthread_self (...)`	Renvoie le numéro (`tid`) du thread courant, équivalent de `getpid().`
`pthread_join ()`	Pour **attendre** la **fin** d'un thread dont on donne le `tid`.

Analogies Système/langage

• On peut faire les analogies suivantes :

Langage	Système
`new` objet allocation mémoire pour une nouvelle instance.	`malloc ( …)` allocation mémoire pour une variable
`new` thread Les structures de données associées au thread sont créées. Le thread n'est pas démarré.	`pthread_create (…)` Le thread est créé. L'instant de son démarrage dépend de sa priorité : ce démarrage peut être immédiat.

• Attention : lors d'un `pthread_create` ou d'un `T.start()`, le thread peut démarrer immédiatement si sa priorité est supérieure à celle du thread courant..

Développer des application En utilisant les threads

• Les threads définissent le parallélisme logique maximum attendu par l'application. Le système se charge **d'adapter** cette demande à la configuration matérielle courante (nombre de processeurs).

• La principale difficulté dans l'utilisation des threads est la gestion des accès aux ressources communes, citons :

- les données partagées (aucune protection entre threads),
- les signaux (qui reçoit un signal : tous les threads, un seul ?),

• les threads permettent de paralléliser les applications sur multiprocesseur, mais le programmeur reste responsable de la synchronisation des accès à la mémoire

Plan

– Présentation

Historique
Définitions
Implantations
Apports
Problèmes

– Ordonnancement

– Synchronisation

Verrous
Variables conditionnelles
Sémaphores

Ordonnancement : les priorités

• On reprend l'exemple précédent :

```
…
void main (void)
{
  …
  pthread_create (f, …);
  pthread_create (g, …);
  …
}
```

• Attention, de nombreuses combinaisons d'ordonnancement sont possibles :

- Si le thread (appelé t1) créé pour exécuter f a une priorité supérieure à celle de celui qui exécute main (appelé t0), on passe la main à t1. La création du second thread est différée…

- Dans le cas d'un ordonnancement à base de quantum de temps (algorithme du tournique ou round robin), si t1 a une priorité égale à celle de t0, t0 continue et crée t2, sauf si la fin de quantum du processus lourd intervient avant l'appel `pthread_create (g, …)!`

• Comme dans le cas des processus, pour contrôler l'exécution des threads on est conduit à utiliser des outils de synchronisation, plutôt que de jouer sur les priorités.

Plan

– Présentation

- Historique
- Définitions
- Implantations
- Apports
- Problèmes

– Ordonnancement

– Synchronisation

- Verrous
- Variables conditionnelles
- Sémaphores

Synchronisation : Exemple

• Nous illustrons ici le problème que pose le partage de mémoire par les threads : quelle sera la valeur affichée par main ?

```
static int valeur = 0;
void main(void)
{
  pthread_t tid1, tid2;
  void fonc(void);
  pthread_set_concurrency(2);
  pthread_create(&tid1, NULL, (void *(*)()) fonc,
NULL);
  pthread_create(&tid2, NULL, (void *(*)()) fonc,
NULL);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  printf("valeur = %d\n", valeur);
}
void fonc(void)
{
    int i;
    pthread_t tid;
    for (i=0;  i <1000000; i++) valeur = valeur + 1;
    tid = pthread_self();
    printf("tid : %d valeur = %d\n", tid, valeur);
}
```

• Remarque : `join` est nécessaire. Sans `join` le processus n'attend pas la fin des threads, il se termine et entraîne la fin de tous les threads créés.

Synchronisation : Exemple

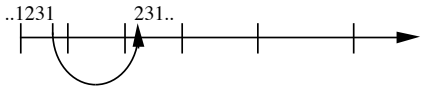
• Voici les résultats de quelques exécutions différentes, pourquoi ces différences ?

```
tid : 4 valeur = 1342484
tid : 5 valeur = 1495077
valeur = 1495077

tid : 4 valeur = 1279685
tid : 5 valeur = 1596260
valeur = 1596260

tid : 5 valeur = 1958900
tid : 4 valeur = 2000000
valeur = 2000000
```

• L'incrémentation de `valeur` se fait en plusieurs instructions machine :

1-Ranger `valeur` dans un registre
2-Incrémenter le registre
3-Ranger le registre dans `valeur`

• Si la fin de quantum intervient après la phase 1, le contexte mémorise l'état du registre à cet instant. Lors de la restauration du contexte, le registre reprend cette valeur, les incréments effectués entre-temps par l'autre thread seront donc écrasés...



Fonctions de synchronisation

• Plusieurs types d'outils :

– le plus simple : les verrous (*locks*),

– les sémaphores pour résoudre l'exclusion mutuelle, pour synchroniser. Problème : interblocage possible.

– les variables conditionnelles (*condition variables*). Elles résolvent le problème de l'interblocage. Associées à un verrou, elles servent à gérer l'attente d'un événement par un ou plusieurs threads.

Synchronisation : verrous

- le verrou doit être rendu par le thread qui l'a PRIS.
- les verrous peuvent être utilisés par des threads appartenant à différents processus,
- la libération des threads bloqués se fait de façon aléatoire.

Nom de la fonction	Rôle de la fonction
`pthread_mutex_init (...)`	le verrou est créé et mis à l'état "unlock"
`pthread_mutex_destroy(...)`	le verrou est détruit
`pthread-mutex_lock (...)`	si le verrou est déja pris, le thread est bloqué
`pthread_mutex_trylock(...)`	renvoie une erreur si le verrou est déja pris, le thread n'est PAS bloqué
`pthread_mutex_unlock (...)`	rend le verrou et libère un thread

Verrous : Exemple 1

- Reprenons l'exemple précédent. Pour en assurer le bon fonctionnement, il faudrait modifier la boucle ainsi :

Avant modification:

```
for (i=0;  i <1000000; i++) valeur = valeur + 1;
```

Après modification :

```
for (i=0;  i <1000000; i++)  {
     pthread_mutex_lock (&verrou);
     valeur = valeur + 1;
     pthread_mutex_unlock (&verrou);
}
```

Verrous : Exemple 2

• Autre exemple :

Deux threads T_1,et T_2 doivent assurer une contrainte de cohérence sur les données x et y , qui est ici : **x = y**.

```
pthread_mutex_t verrou;
int x, y;
```

Thread T_1	Thread T_2

...
pthread_mutex_lock(&verrou);
x = x + 1;
y = y + 1;
pthread_mutex_unlock(&verrou
)
...
	...
	pthread_mutex_lock
	(&verrou);
	x = 2 * x;
	y = 2 * y;
	pthread_mutex_unlock(&verro
	u)
	...

Verrous : Programmation

• Le temps passé dans une section critique doit être court,

• Seul le thread qui a pris un verrou peut le rendre (à la différence des sémaphores) :

non	oui

...
pthread_create (f1, ···);
pthread_create (f2, ···);
...
 	
/* fonction f1 */
f1()
{
pthread_mutex_lock (&Verrou);
...
}
 	
/* fonction f2 */
f2()
{
...
pthread_mutex_unlock (&Verrou);
}
	/* f1 et f2 ne contiennent pas
	d'appel a lock et unlock */···
	pthread_create (f, ···);
	...
	
	f()
	{
	pthread_mutex_lock (&Verrou);
	f1() ;
	f2() ;
	pthread_mutex_unlock (&Verrou);
	...
	}

Synchronisation : sémaphores

- les sémaphores peuvent être utilisés par des threads appartenant à différents processus,
- la libération des threads bloqués se fait de façon aléatoire (!).

Nom de la fonction	Rôle de la fonction
`sem_init (&S, , Compt,)`	initialisation du sémaphore
`sem_destroy (&S)`	suppression du sémaphore
`sem_wait (&S)`	attend que la valeur du compteur du sémaphore soit positive, puis la décrémente
`sem_trywait (&S)`	décrémente le compteur s'il est positif, sinon erreur
`sem_post (&S)`	incrémente le compteur et libère éventuellement un thread.

Sémaphores : exemple

- On reprend l'exemple des threads producteur et consommateur :

```
sem_t     SP, SC;

int    Ind_Prod;
int    Ind_Cons;
char       Tampon [DIM];
char       Depot = 'a';

sem_init (&SP, , DIM, );
sem_init (&SC, , 0);
```

Thread T_1	Thread T_2

`while (Travail) {`
`  sem_wait (&SP);`
`  Tampon[Ind_Prod] = Depot++;`
`  Ind_Prod = (Ind_Prod+1)%DIM;`
`  sem_post (&SC);`
`}`
	`while (Travail) {`
	`  sem_wait (&SC);`
	`  Conso(Tampon[Ind_Prod]);`
	`  Ind_Cons = (Ind_Cons+1)%DIM;`
	`  sem_post (&SP);`
	`}`

Synchronisation : variables conditionnelles

- les variables conditionnelles (*conditions variables*), s'utilisent conjointement à un verrou, elles évitent les interblocages.

Nom de la fonction	Rôle de la fonction
`pthread_cond_init (&cv)`	La var.cond. `cv` est créée
`pthread_cond_destroy (&cv)`	La var.cond. `cv` est détruite
`pthread_cond_wait (&cv, &V)`	Fait passer le thread à l'état bloqué ET rend le verrou `V` de façon **atomique**. Sort de l'état bloqué et **essaie de reprendre** `V` sur un cond_signal ou cond_broadcast
`pthread_cond_signal (&cv)`	Libère un des threads bloqués sur `cv`.
`pthread_cond_broadcast (&cv)`	Libère tous les threads bloqués sur `cv`.
`pthread_cond_timedwait ()`	attend un signal de libération pendant un certain temps

Variables conditionnelles : Exemple 1-1

• Soit deux threads T_1 et T_2 et une ressource R protégée par un verrou :

- T_1 met à jour R,
- T_2 attend que R soit mise à jour

• Scénario :

T_1 prend le verrou et consulte R, elle n'a pas la valeur souhaitée. Il va donc rendre le verrou et attendre que R atteigne la valeur souhaitée. Il va ensuite revenir périodiquement essayer de reprendre le verrou pour consulter R. On se retrouve donc dans un schéma du type attente active.

Thread T_1	Thread T_2

...
while (true)
{
lock() ;
...
tester(R) ;
...
unlock() ;
sleep(t) ;
}
...
	...
	lock() ;
	...
	mettre-a-jour(R) ;
	...
	unlock() ;

Variables conditionnelles : Exemple 1-2

• L'utilisation d'un variable conditionnelle résout élégamment ce problème :

T_1 fait appel à cond_wait, il passe dans l'état bloqué, rend le verrou (ces deux actions sont atomiques) et sera réveillé par un cond_signal émis par T2.

• Extrait du programme :

```
int Etat = 1 ;

pthread_cond_t VarC ;       /* la var. cond.  */
pthread_mutex_t Verrou ;    /* la verrou associé  */
```

Thread T_1	Thread T_2
... pthread_mutex_lock (&Verrou); while (Etat == 1) pthread_cond_wait(&VarC,&Verrou) ; ... pthread_mutex_unlock (&Verrou); ...	... pthread_mutex_lock (&Verrou); Etat = 0; ... pthread_cond_broadcast(&VarC); pthread_mutex_unlock (&Verrou); ...

Variables conditionnelles : Exemple 2

• Accès à une base de données avec priorité aux lecteurs, on donne les outils de synchronisation et la donnée partagée :

```
pthread_mutex_t  V;         /*Verrou */
pthread_cond_t       VarC ; /*Var. cond. */

int     Nb_lect ;   /*Nb_lect >0 -> nbre de lecteurs
                      Nb_lect= -1 -> un ecrivain   */
   ...
```

Lecteur	Ecrivain

`void lecteur ()`
`{`
`pthread_mutex_lock (&V);`
`while (Nb_lect < 0)`
`  pthread_cond_wait`
`(&VarC,&V);`
`Nb_lect++;`
`pthread_mutex_unlock(&V);`
`  **LireBdD ();**`
`pthread_mutex_lock(&V);`
`Nb_lect--;`
`if (Nb_lect == 0)`
 	
`pthread_cond_signal(&VarC);`
`pthread_mutex_unlock(&V);`
`}`
	`void ecrivain ()`
	`{`
	`pthread_mutex_lock (&V);`
	`while (Nb_lect != 0)`
	`  pthread_cond_wait(&VarC,&V);`
	`Nb_lect--;`
	`pthread_mutex_unlock(&V);`
	`  **EcrireBdD ();**`
	`pthread_mutex_lock(&V);`
	`Nb_lect = 0;`
	`pthread_cond_broadcast(&VarC);`
	`pthread_mutex_unlock(&V);`
	`}`

Synchronisation : Récapitulatif

• **le verrou (*lock*)**

l'outil le plus rapide et le moins consommateur de mémoire. **Il doit être rendu par le thread qui l'a pris** . Il s'utilise surtout pour sérialiser l'accès à une ressource ou assurer la cohérence des données.

• **le sémaphore**

consomme plus de mémoire et s'utilise dans les cas où on se synchronise sur l'état d'une variable, plutôt qu'en attendant une commande venue (`cond_signal`) d'un autre thread. Il n'exige pas que le verrouillage/déverrouillage soit fait par le même thread.

• **les variables conditionnelles (*condition variables*)**

Une variable conditionnelle s'utilise pour attendre l'occurence d'un événement.

Le verrou qu'on doit lui associer sert à gérer l'exclusion mutuelle sur les variables internes liées à cette variable conditionnelle (compteur).

Ce verrou est automatiquement rendu par le système lors de l'appel à cond_wait et repris dès la sortie de cette fonction. Il doit être rendu par l'utilisateur après cette sortie.

Synchronisation : récapitulatif

• Visibilité :

les outils de synchronisation peuvent être vus par des threads appartenant à des processus DIFFERENTS :

• Passage à l'état bloqué, sortie de cet état :

- L'appel à cond_wait est **toujours** bloquant (à la différence de `lock` ou `sem_wait`)
- cond_signal ne fait qu'essayer de débloquer un thread, le signal est **perdu** si aucun thread n'est bloqué lors de son émission (alors que V incrémente un compteur)
- pour éviter les interblocages, utiliser les verrous ou les sémaphores suivant un ordre fixe, ou utiliser les variables conditionnelles

• Performances :

attention à l'inversion de priorité

Quelques fonctions POSIX

• Pour créer et initialiser un verrou visible par des threads appartenant à des processus différents :

```
#include <pthread.h>
pthread_mutex_t          Verrou ;
pthread_mutexattr_t      Attributs;

pthread_mutexattr_init (&Attributs);
pthread_mutexattr_setpshared(&Attributs,PTHREAD_PROCESS_SHARED);
pthread_mutex_init (&Verrou, & Attributs);
```

• Pour créer et initialiser une variable conditionnelle (privée ou visible à l'extérieur du processus) :

```
pthread_cond_t        VarCond;
pthread_condattr_t    Attributs;

pthread_condattr_init (&Attributs) ;
pthread_condattr_setpshared (&Attributs,
PTHREAD_PROCESS_PRIVATE);
                                    ou _PROCESS_SHARED);
pthread_cond_init (&VarCond, &Attributs);
```

Les threads Java

Créer et démarrer un thread (extends Thread)

• Hériter de la classe `Thread` et surcharger la méthode `run` :

```
// main va créer deux threads et, ensuite, les
démarrer.
class essai {
    public static void main (String args[]) {

   Thread T1 = new MonThread("  T1  ");
   Thread T2 = new MonThread("  T2  ");

   System.out.println("Lancement de T2");
   T2.start();   // start appelle la methode run de T2
   System.out.println("Lancement de T1");
   T1.start();    // start appelle la methode run de T1
   while (true) ;
    }
}
// La classe MonThread hérite de Thread
class MonThread extends Thread {
    public MonThread(String str) {
        super(str);
    }
    public void run() {
   System.out.println("Execution de " + getName());
    }
}
```

Exécution 1	Exécution 2
Lancement de T2 Execution de T2 Lancement de T1 Execution de T1	Lancement de T2 Lancement de T1 Execution de T2 Execution de T1

Créer et démarrer un thread (Runnable) -1

• On utilise des objets qui ne sont pas des threads. On invoque leur méthode run :

```
class essai2 {
    public static void main (String args[]) {

   Action A1 = new Action("  A1  ");
   Action A2 = new Action("  A2  ");
      System.out.println("Lancement de A2");
   A2.run();
   System.out.println("Lancement de A1");
   A1.run();
   while (true) ;
    }
}

class Action
{
    String local;
    public Action (String str) {
   local = str;
    }
    public void run() {

   System.out.println("Execution de "  + local );
    }
}
```

• Résultat :

```
Lancement de A2
Execution de   A2
Lancement de A1
Execution de   A1
```


Créer et démarrer un thread (Runnable) -2

• On transforme ces objets en threads. On appelle start qui appellera run :

```
class essai2 {
    public static void main (String args[]) {

   Action A1 = new Action("  A1  ");
   Action A2 = new Action("  A2  ");

   System.out.println("Lancement de A2");
   new Thread(A2).start();
   System.out.println("Lancement de A1");
   new Thread(A1).start();
   while (true) ;
    }
}

class Action    implements Runnable
{
    String local;
    public Action (String str) {
   local = str;
    }
    public void run() {

   System.out.println("Execution de "  + local );
    }
}
```

• Résultat :

```
Lancement de A2
Lancement de A1
Execution de   A2
Execution de   A1
```

Java : Ordonnancement

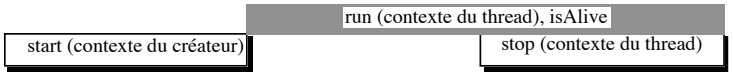
• Principales méthodes associées à la classe `Thread` :

- `T.start()`, pour démarrer le thread T,
- `run()`, exécutée par `start`, vide par défaut, à surcharger,
- `sleep(t)`, le thread courant se bloque pendant t millisecondes, (gérer les exceptions),
- `setprio()`, pour affecter une priorité à un thread,
- `yield()`, donne la main au thread suivant de la même priorité, ou au premier thread de priorité inférieure,
- `T.join()`, pour attendre la fin du thread T,
- `T.join(t)`, pour attendre la fin de T pendant au plus t millisecondes,

Java : Etats des threads

• Un thread `T` passe dans l'état prêt lors de l'appel à `T.start()` qui appelle la méthode `run` de `T`.

• `start` est exécutée dans le contexte du thread créateur de `T` , `run` dans le contexte de `T` :



Java : Synchronisation

• On crée puis on lance deux threads :.

```
...
    Thread T1, T2;
    // Creation des threads
    T1 = new MonThread ("Thread_1");
    T2 = new MonThread ("Thread_2");
    // Demarrage des threads
    T1.start();
    T2.start();
...

//
class MonThread extends Thread
{
  static int valeur = 0;
  ...
public void run()
    {
    System.out.println(getName() + " demarre");
    for (int i =0; i < 1000000 ; i++)valeur = valeur + 1;
    System.out.println("\t\t"+getName()+"s arrete  " + valeur );
    }
}
```


Java : Synchronisation

• Voici les résultats de quelques exécutions différentes, pourquoi ces différences ? Réponse : l'incrément de valeur se fait en plusieurs instructions machine.

```
Resultats :

Thread_1 demarre
Thread_1 s arrete   1000000
Thread_2 demarre
Thread_2 s arrete   2000000

Thread_1 demarre
Thread_2 demarre
Thread_1 s arrete   1026351
Thread_2 s arrete   1491741

Thread_1 demarre
Thread_2 demarre
Thread_1 s arrete   1377972
Thread_2 s arrete   1103782

Thread_1 demarre
Thread_2 demarre
Thread_1 s arrete   1922435
Thread_2 s arrete   2000000
```

Java : Synchronisation

• A chaque objet est associé un verrou, ce verrou est pris par :

- l'appel à une méthode qualifiée de `synchronized`
- l'entrée dans un bloc qualifié de `synchronized`

• Conséquences :

- l'appel à méthode `synchronized` interdit l'accès à **toute autre méthode `synchronized`**,
- les méthodes non synchronized continuent à **accéder** à l'objet

Synchronisation `Wait/notify`

• `Wait, Notify , NotifyAll` (à utiliser **dans** des méthodes **`synchronized`**) fonctionnent suivant le modèle le wait/signal/broadcast des variables conditionnelles :

- `Wait` bloque le thread appelant et rend le verrou, ceci de façon atomique. Lorsque le thread est débloqué, il reprend le verrou,
- `Notify` débloque le thread bloqué sur `wait` (atomique),
- `NotifyAll` débloque tous les threads bloqués sur `wait` (atomique),

Java : Synchronisation, mise en œuvre

• Utilisation de `synchronized` :

- coûteux en temps : **le contrôle d'accès** multiplie le temps passé pour la gestion d'un appel de méthode par un facteur 5

• attention à la cohérence des données : si le thread est détruit alors qu'il se trouve dans une méthode `synchronized`, la libération des verrous est bien gérée par Java, pas les éventuelles **données** protégées,

• attention aux appels de méthodes non "synchronisées" dans des méthodes "synchronisées",

`wait/notify` Exemple (1)

• Scénario : un thread doit attendre qu'une variable passe à zéro, cette variable est décrémentée par plusieurs autres threads,

• Implémentation, un objet propose les méthodes suivantes :

- `Init` qui initialise la variable,
- `Attendre` qui bloque le thread tant que la valeur zéro n'est pas atteinte,
- `Decrementer` qui décrémente la variable et réveille le thread bloqué si elle passe à zéro

wait/notify
Exemple (2)

```
class Synchro
{
  int Valeur;

// Initialisation de la variable partagee

  public synchronized void Init (int Ma_Valeur)
    {
      Valeur = Ma_Valeur;
    }

//  Décrementer  la  variable  et  réveiller  le  thread
 bloqué

  public synchronized int Decrementer ( )
    {
      Valeur = Valeur - 1;
      if ( Valeur == 0)
          {
           notify();
           return (0);
          }
      return (Valeur);
    }

// Attendre

   public synchronized void Attendre ( )
    {
     ...
     wait();
     ...
    }
}
```


JDK 1.5

- Runnable -> callable :
  - Callable <V> renvoie une valeur de type V,
  - quand cette valeur est-elle retournée ? -> Future<V>
- Future<V>:
- get : attente de la fin du thread
- isDone pour vérifier si le résultat est arrivé ou non
- Executors pour gérer un pool de threads:
- ScheduledExecutor
- isDone pour vérifier si le résultat est arrivé ou non

JDK 1.5

• Sémaphores, exemple :

```
import java.util.concurrent.*;

public void Gerer_RdV(){
   Verrou.acquire();
   Nb_Arrives=Nb_Arrives+1;
   if (Nb_Arrives < N){
      Verrou.release();
      Sem_RDV.acquire();
   }
   else{
   Verrou.release();
   for (int i=1; i<=N-1; i++) Sem_RDV.release();
   Nb_Arrives = 0;
   }
}
```

• Variables conditionnelles (cf. API threads POSIX ...)

JDK 1.5

• Pool de threads (exemple, ici N vaut 3):

```
import java.util.concurrent.*;

class Mon_Thread extends Thread
{
  public void run()
 {
    for (int i=0; i<0xFFFFFFF; i++);
    System.out.println("Ici :" +
Thread.currentThread().getName() );

…
Executor Execute_Thread = Executors.newFixedThreadPool(N);
…
  for (int i=0; i<2*N; i++)
      Execute_Thread.execute(new Mon_Thread());

  for (int i=0; i<N; i++) {
       Les_Threads[i]= new Mon_Thread();
       Les_Threads[i].setName("Mon_Thread"+i);
       Les_Threads[i].start();
   }
 for (int i=0; i<2*N; i++)
     Execute_Thread.execute(new Mon_Thread());

…

// frechou% java Exec_Thread 3
// Ici :pool-1-thread-1
// Ici :Mon_Thread2
// Ici :pool-1-thread-2
// Ici :pool-1-thread-3
// Ici :Mon_Thread0
// Ici :Mon_Thread1
// Ici :pool-1-thread-1
// Ici :pool-1-thread-3
// Ici :pool-1-thread-2
// Ici :pool-1-thread-1
// Ici :pool-1-thread-2
// Ici :pool-1-thread-3
// Ici :pool-1-thread-1
// Ici :pool-1-thread-2
// Ici :pool-1-thread-3
```