Multithreading

Partie I

- Concept
- Définition
- Mémoires d'un programme
- Machine à état
- Performance
- Terminologie

Partie II

- Codage performant
- Normalisation
- Mise en œuvre en C/C++
- Mise en œuvre en Java
- IPC
- Thread et temps réel

Partie III

- Gestion de la concurrence
- Section critique
- Algo de Peterson
- Exclusion mutuelle matériel
- Variable atomique
- Opération atomique
- Barrière de synchronisation
- Implémentation en C
- Implémentation en Java

Partie IV

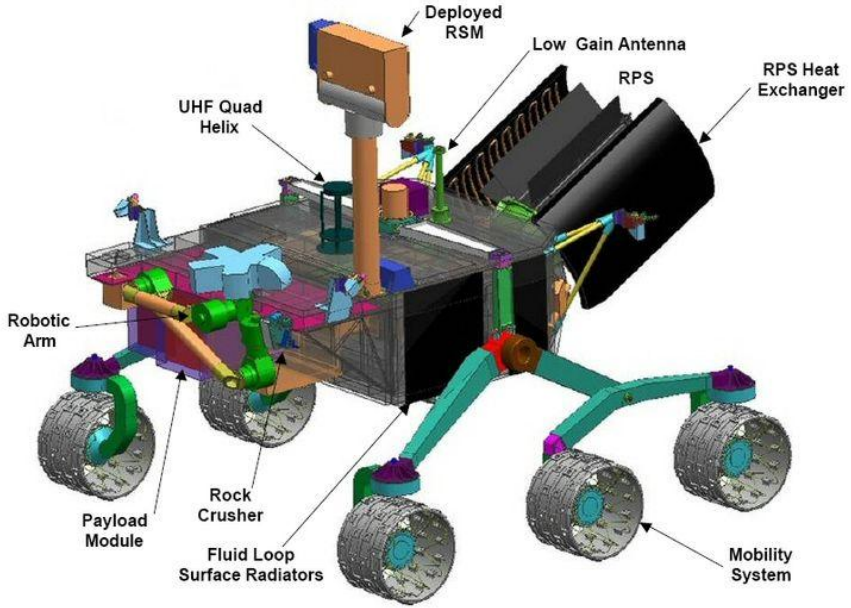
- Risques et problèmes
- Introduction réseau de Pétri
- Patron de conception

Partie V

- Introduction OpenMP

Multithreading

Concept



Le programme principal gère :

- Système de communication,
- Le programme de mobilité,
- Les interfaces d'entrées sorties (capteurs, batterie ...),
- Le laboratoire embarqué ...

Multithreading

Concept

Exemple: Le système de géolocalisation, le programme est décomposé en :

- logiciel de capture de trame satellite et stockage en mémoire,
- logiciel de traitement d'analyse des trames satellites,
- logiciel de calcul,
- logiciel d'affichage
- ...

Multithreading

Concept

Exigences

- Partager les ressources du système afin d'optimiser leur fonctionnement, le coût, le poids ...
- Ne pas multiplier les équipements (surtout dans l'embarqué),
- Nécessité d'exécuter plusieurs programmes pour satisfaire différentes fonctions d'un système (I/O, IHM, réseau …),
- On veut capturer des informations de différentes natures au même instant et exécuter des algorithmes complexes
- ...

Multithreading

Concept

Impacts

- Dégager les différentes fonctionnalités,
- Réduire la complexité du code,
- Augmenter la modularité,
- Augmenter la réutilisabilité des codes,
- Faciliter la maintenance,
- Permettre des améliorations par fonction,
- Augmenter la fiabilité, la sécurité ...

Multithreading

Concept

Mise en oeuvre

- Utilisation d'OS modernes,
- Utilisation dans les systèmes embarqués,
- Utilisation dans les super-calculateurs,
- Utilisation dans des équipements grand public (jeux, multimédia, téléphonie …)

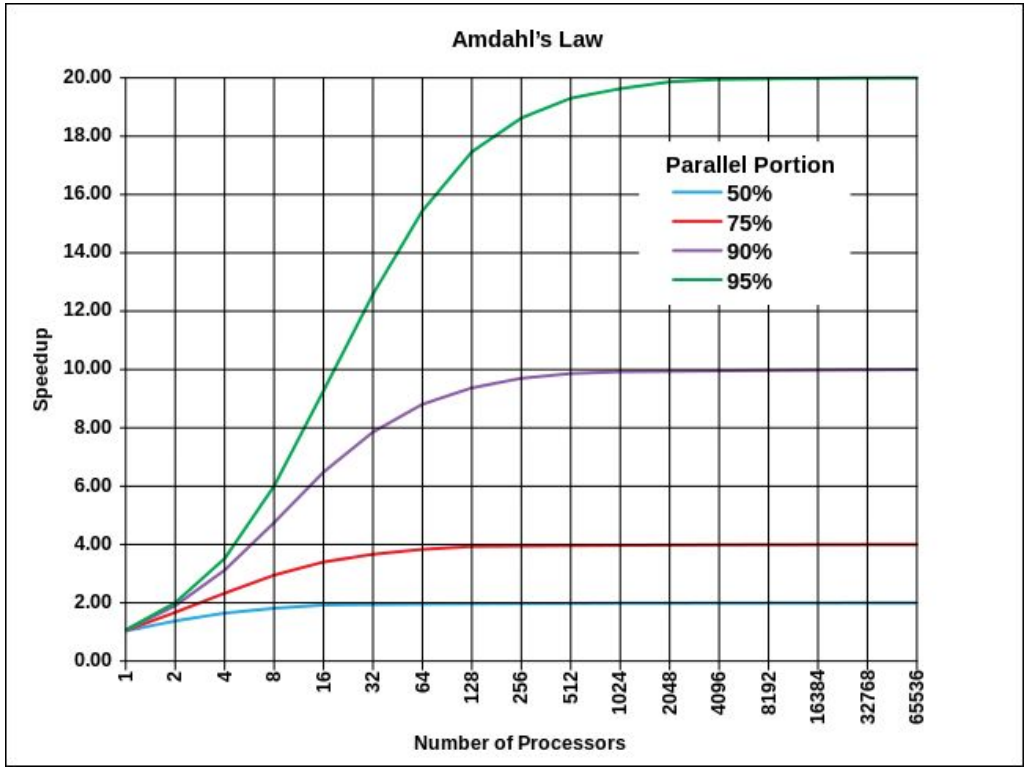
Multithreading

Concept

La loi d'Amdahl donne l'accélération théorique en latence de l'exécution d'une tâche à charge d'exécution constante que l'on peut attendre d'un système dont on améliore les ressources.

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

- $S_{latence}$ est l'accélération théorique en latence de l'exécution de toute la tâche ;
- s est l'accélération en latence de l'exécution de la partie de la tâche bénéficiant de l'amélioration des ressources du système ;
- p est le pourcentage du temps d'exécution de toute la tâche concernant la partie bénéficiant de l'amélioration des ressources du système avant l'amélioration



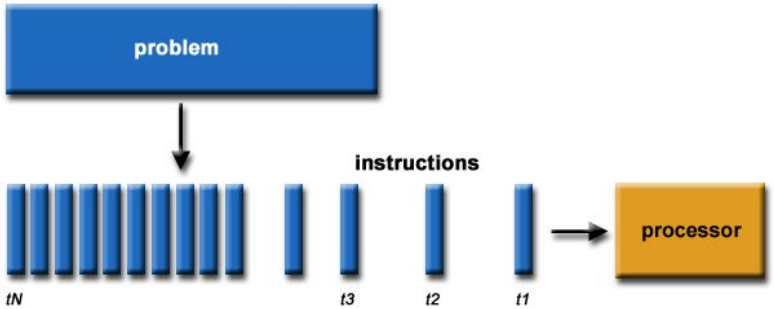
Multithreading

Définition

Calcule en série

Habituellement les logiciels sont écrits avec une suite d'instruction exécutés en série:

- Le système est divisé en en une série d'instruction discrète,
- Les instructions sont exécutées séquentiellement l'une après l'autre,
- Exécution sur un seul processeur,
- Une seule instruction est exécutée en même temps



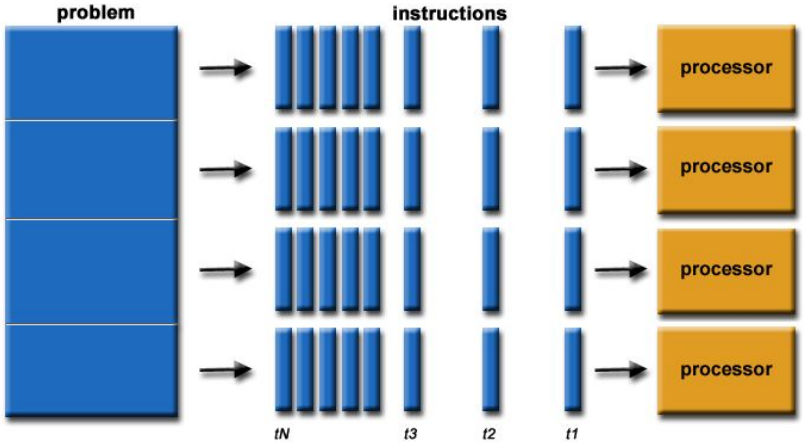
Multithreading

Définition

Calcul parallèle

Le calcul parallèle est l'utilisation simultanée de plusieurs calculateurs pour exécuter un programme:

- Le programme est découpé de façon discrète et peut être résolu de façon concurrente
- Chaque partie est encore divisée en une série d'instruction,
- Chaque instruction de chaque partie est exécutée sur un processeur différent
- Un système de coordination et de contrôle est utilisé pour gérer l'ensemble des calculs.



Multithreading

Définition

Programme = suite d'instructions

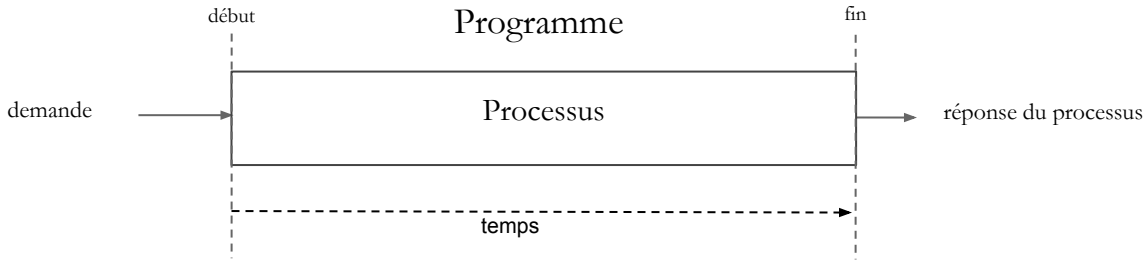
Le processeur exécute une suite d'instruction simple.

- **fetch** : récupère les instructions à partir de la mémoire,
- **decode** : détermine le type d'instruction,
- **exécute** l'instruction,
- **writeback** : enregistre les résultats.

Multithreading

Définition

Processus = Exécution d'un programme

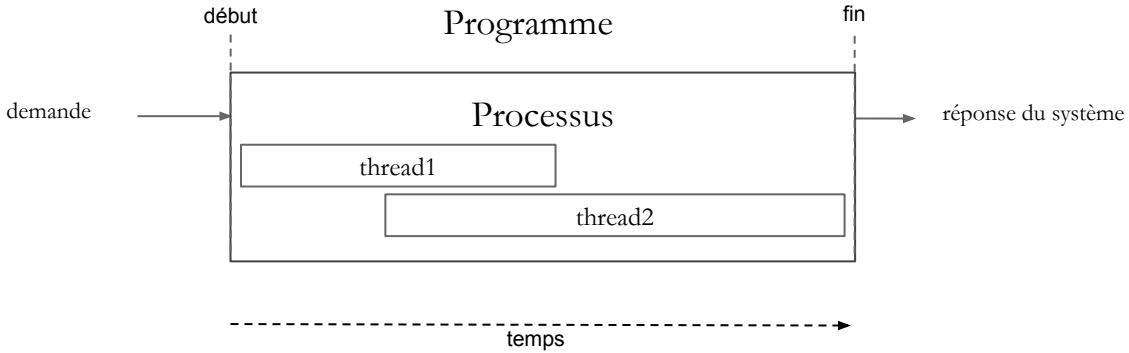


Un **processus** (en anglais, process), en informatique, est un **programme** en cours d'exécution par un ordinateur. Il peut être défini comme un ensemble d'instructions à exécuter, il possède un espace d'adressage en mémoire vive pour stocker la pile, les données de travail.

Multithreading

Définition

Thread = Exécution d'une suite d'instruction dans un processus



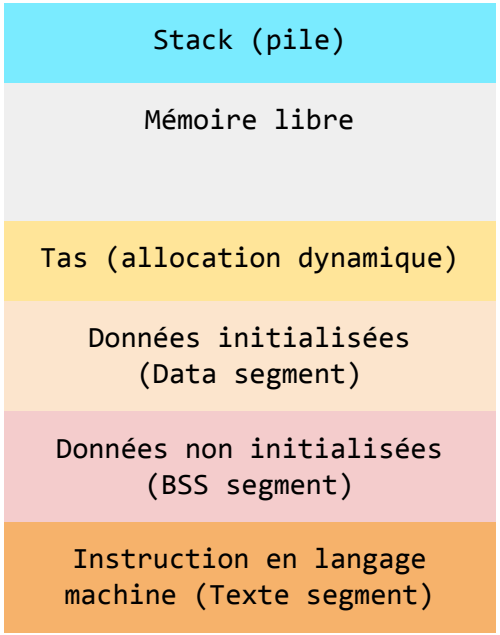
Un ***thread*** ou **fil (d'exécution)** ou **tâche** (normalisés par ISO/CEI 2382-7:2000) autres appellations connues : **processus léger**, **fil d'instruction**, **processus allégé,** représente l'exécution d'un ensemble d'instructions dans un processus.

- Un *threaa* est lancé à partir d'un processus.
- Les *threads* d'un même processus se partagent la mémoire virtuelle du processus.
- Tous les *threads* possèdent leur propre pile d'appel,
- Les threads d'un même processus partagent le même espace d'adresse.

Multithreading

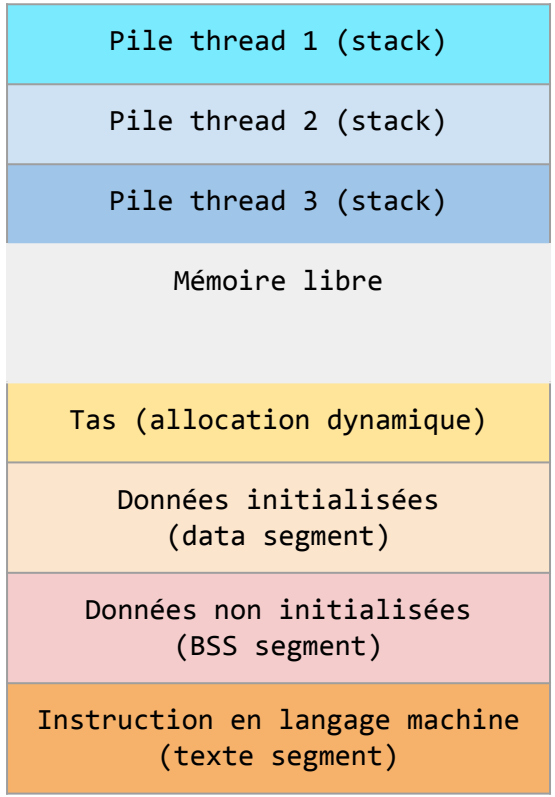
mémoires d'un programme

Processus



BBS: Basic Service Set

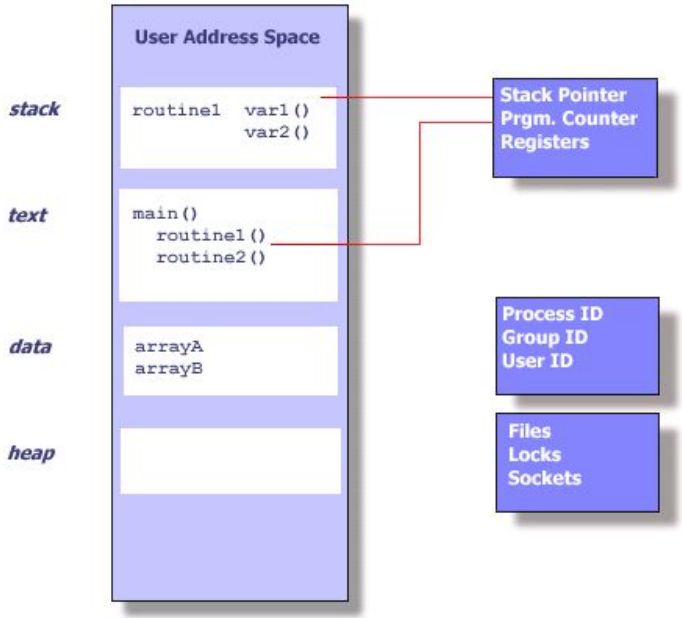
Processus avec plusieurs thread



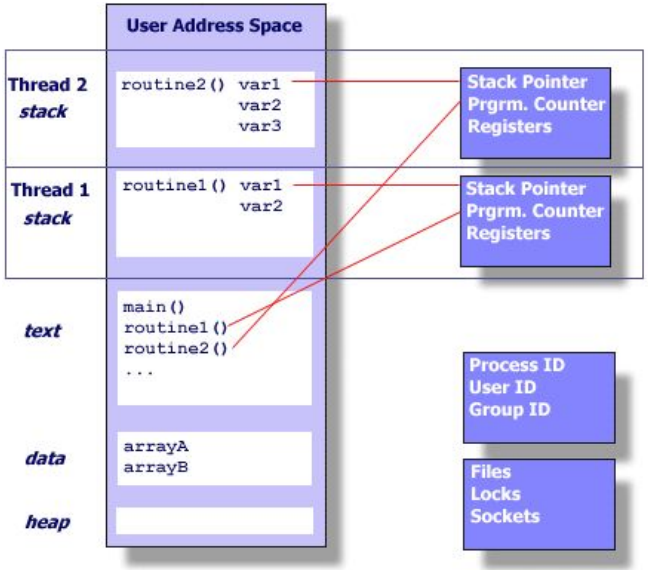
Multithreading

mémoires d'un programme

Processus avec 1 thread



Processus avec plusieurs threads



BBS: Basic Service Set

Multithreading

Graphe état transition d'un thread

Ordonnanceur / scheduler

- Un OS qui supporte le multithreading possède un **ordonnanceur** (*scheduler*) qui est responsable du **séquencement** temporel des processus et thread.

- Les algorithmes d'ordonnancement réalisent la sélection parmi les processus actifs de celui qui va obtenir l'utilisation d'une ressource, que ce soit l'unité centrale, ou bien un périphérique d'entrée-sortie.

- Il existe différents types d'ordonnanceurs.

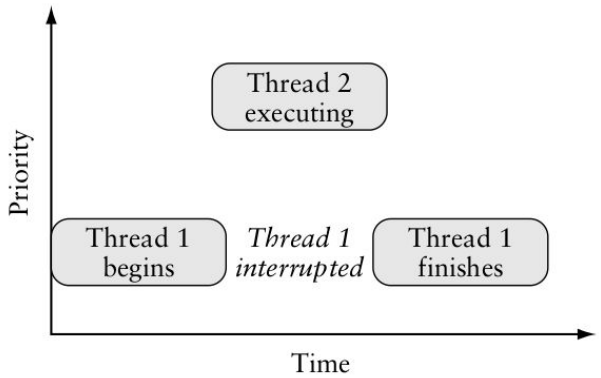
Multithreading

Graphe état transition d'un thread

Ordonnanceur / scheduler /**préemption avec priorité**

La **préemption** est la possibilité qu'a le système de suspendre un processus/thread.
Chaque thread possède une priorité. Un thread de plus haute priorité peut interrompre et suspendre un thread de plus basse priorité en cours d'exécution

Thread preemption.



Le multithreading préemptif est généralement considéré comme la meilleure approche , car il permet au système d'exploitation de déterminer si un changement de contexte doit se produire. L'inconvénient du multithreading préemptif est que le système peut faire un changement de contexte à un moment inapproprié , ce qui peut provoquer un verrouillage , une inversion de priorité ou d'autres effets négatifs , qui peuvent être évitées par le multithreading coopérative.

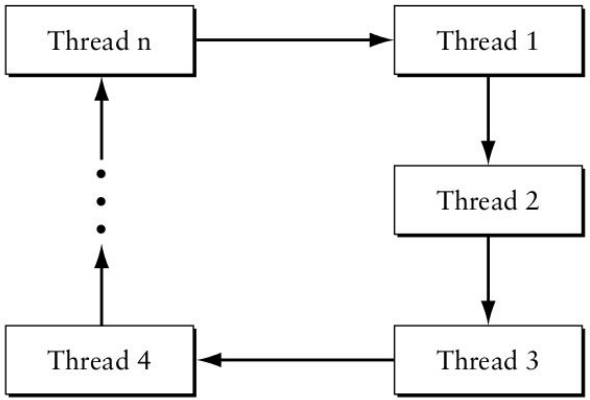
Multithreading

Graphe état transition d'un thread

Ordonnanceur / scheduler / **Round-Robin**

Il existe plusieurs variantes de l'algorithme Round-Robin.

Exemple: Ordonnancement utilisé dans le cas ou plusieurs threads possèdent une même priorité. Chaque thread est exécuté dans un ordre et pendant un quantum de temps.



Multithreading

Graphe état transition d'un thread

Ordonnanceur / scheduler / **Coopératif**

Chaque thread fonctionne jusqu'à ce qu'il passe la main à un autre thread, ou atteint un appel système. C'est le comportement des threads qui permet ce fonctionnement. Ce type de fonctionnement peut être vu dans les applications qui nécessitent un ordre fixe d'exécution

Cela peut créer des problèmes si un thread attend qu'une ressource devienne disponible, car il peut alors se bloquer et entraîner le blocage des autres threads.

Multithreading

Graphe état transition d'un thread

Ordonnanceur / scheduler / **Critique**

Chaque thread fonctionne pendant un quantum de temps fixé par avance avec une marge de sécurité (chemin le plus long à l'exécution).

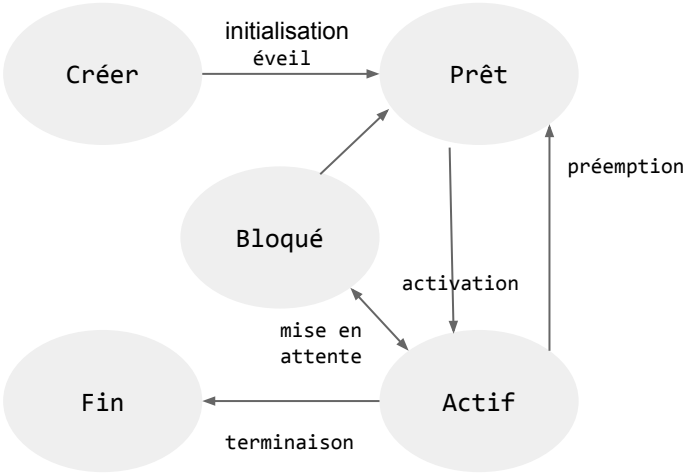
On retrouve ce type de fonctionnement dans les environnements critiques qui nécessite un déterminisme:

- aviation (ARINC 653) https://fr.wikipedia.org/wiki/ARINC_653
- centrale nucléaire
- ferroviaire ...

Multithreading

Graphe état transition d'un thread

Graphe nominal simplifié état transition d'un thread
(avec préemption)



Créer : le thread est créé (un objet)

Prêt : le thread a fini l'initialisation et peut être exécuté par le processeur

Actif : le traitement est en cours dans le processeur (préemption : l'ordonnanceur peut interrompre à tout moment une tâche en cours d'exécution pour permettre à une autre tâche de s'exécuter.)

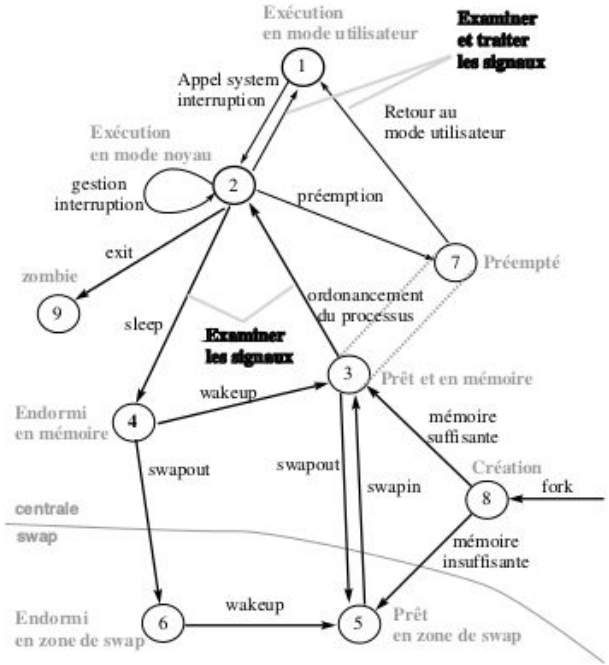
Bloqué : le traitement est suspendu (attente d'un condition, mise au repos ...)

Fin : le thread se termine

Multithreading

Graphe état transition d'un thread

Graphe complet état transition d'un thread



1. le processus s'exécute en **mode utilisateur**

2. le processus s'exécute en **mode noyau**

3. le processus ne s'exécute pas mais est éligible (prêt à s'exécuter)

4. le processus est endormi en mémoire centrale

5. le processus est prêt mais le swappeur doit le transférer en mémoire centrale pour le rendre éligible. (ce mode est différent dans un système à pagination).

6. le processus est endormi en zone de swap (sur disque par exemple).

7. le processus passe du mode noyau au mode utilisateur mais est préempté et a effectué un changement de contexte pour élire un autre processus.

8. naissance d'un processus, ce processus n'est pas encore prêt et n'est pas endormi, c'est l'état initial de tous processus sauf le swappeur.

Multithreading

Graphe état transition d'un thread

Changement de contexte

l'OS possède les informations suivantes, qui lui permettent de changer de contexte d'exécution:

- Thread ID,
- Registres sauvegardés, pointeur de pile, pointeur d'instruction (dans la plupart des processeurs , le pointeur d'instruction (PC : Pointer Counter) est incrémenté après chaque exécution d'une instruction vers la prochaine instruction à exécuter.),
- Pile ou pile (variable locale, variable temporaire, adresse de retour),
- Masque de signaux,
- Priorité (scheduling information).

Multithreading

Graphe état transition d'un thread

Changement de contexte

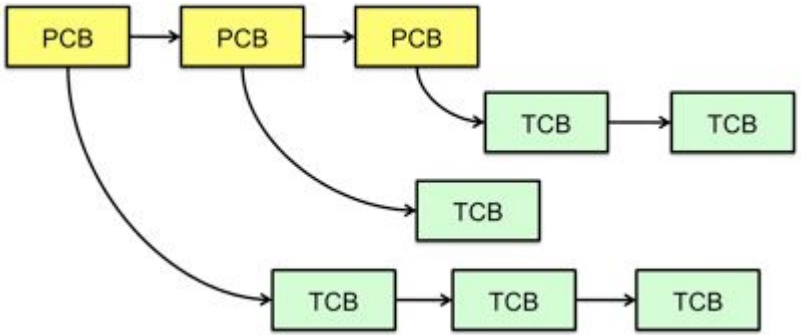
Chaque thread possède les informations suivantes, qui lui permettent de changer de contexte d'exécution:

- Text segment (instructions)
- Data segment (static and global data)
- BSS segment (uninitialized data)
- Open file descriptors
- Signals
- Current working directory
- User and group IDs

Multithreading

Graphe état transition d'un thread

Sauvegarde du contexte des processus et threads

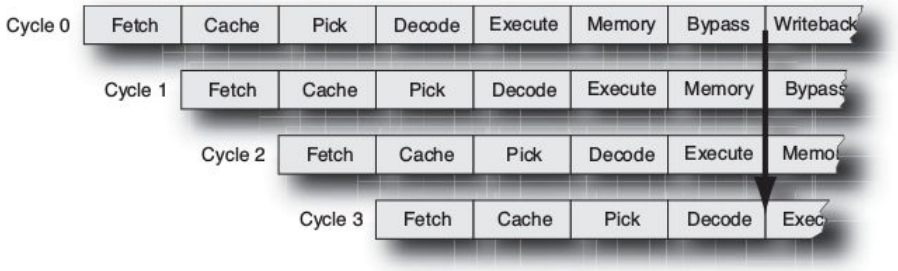


Le système d'exploitation sauve les informations de chaque processus dans un bloc de contrôle (PCB: Process Control Block). Le PCB est organisé en structure de données: table ou liste. Les informations des threads sont sauvegardés dans une structure de données appelée TCB (Thread Control Block). Chaque processus peut avoir un ou plusieurs thread (au moins un). Chaque PCB possède un pointeur sur une structure de TCB.

Multithreading

Performance

Architecture Pipeline

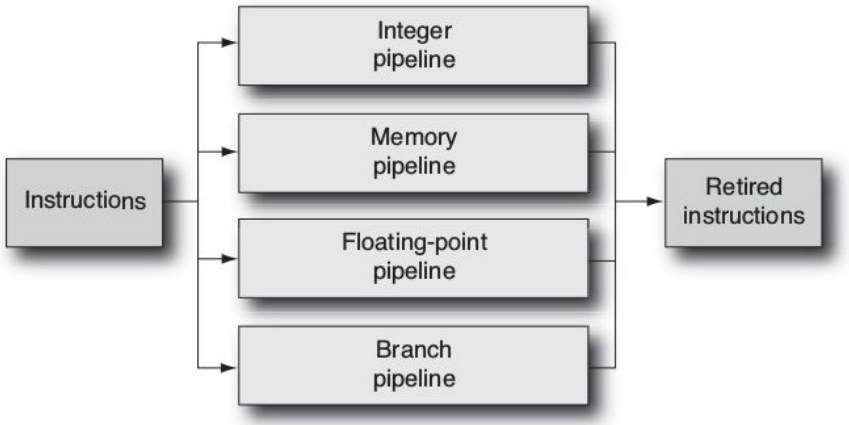


- Exécution de plusieurs instructions en même temps dans une structure “pipeline”,
- L’horloge détermine en combien de fois une instruction a fini l’étage,
- Chaque instruction est exécutée dans le même temps.

Multithreading

Performance

Architecture Pipeline



Les “pipeline” sont spécialisés par type: opération sur entier, flottant, embranchement …

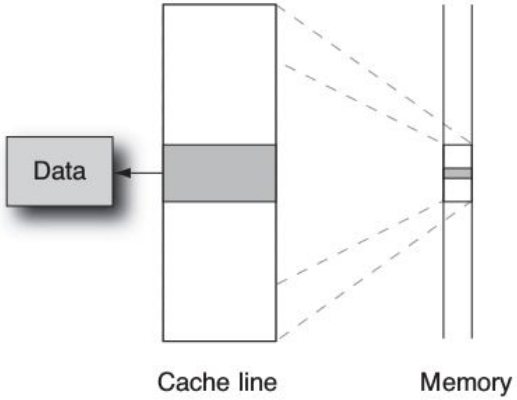
Architecture Pipeline

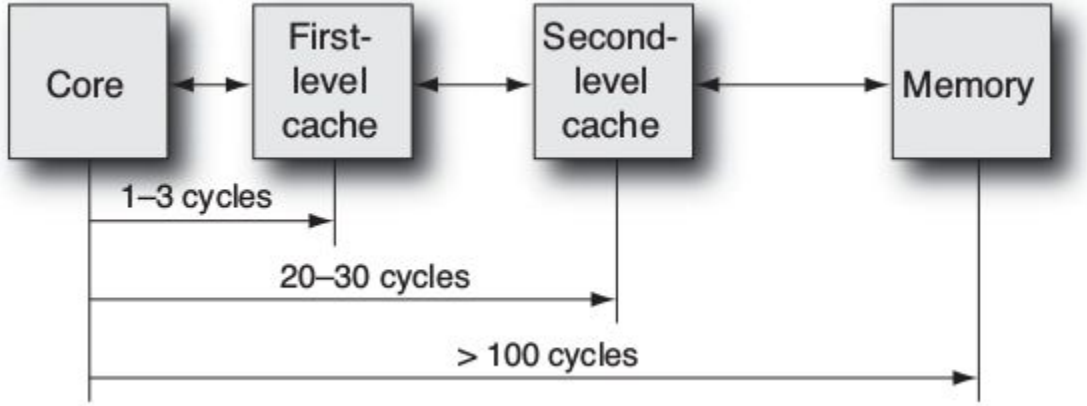
L'exécution du code est anticipé avec un comportement de prédiction (chargement de l'instruction suivante …),

La dépendance dans le code peut entraîner des forts ralentissements,

Une instruction de branchement peut changer l'adresse où l'instruction suivante doit être récupérée.

Utilisation du cache





L'utilisation du cache à un impact immédiat sur les performances. Lorsqu'un processeur demande un ensemble d'octets de mémoire, il ne reçoit pas seulement les octets dont il a besoin. Les données sont extraites de la mémoire avec une taille correspondante à la taille du cache.

Multithreading

Performance

Architecture Pipeline / Exemple avec un processeur ARM

ARM Cortex-M0	3 stage
ARM Cortex-M1	3 stage
ARM Cortex-M4 Specification	3-stage + branch speculation
ARM Cortex-M7	6-stage superscalar + branch prediction

Multithreading

Performance

Comment travailler sans thread ?

- Il est possible de travailler qu'avec des processus !
- Utilisation de la primitive *fork()* qui permet de dupliquer un processus,
- La duplication est plus coûteuse car il faut recopier le tas et les registres,
- L'échange de données est plus complexe car il faut utiliser les IPC pour communiquer,

Multithreading

Performance

Pourquoi travailler avec des threads ?

- Avec plusieurs processeurs, il est possible de réaliser plus de calcul à chaque instant,
- Parallélisation des algorithmes (si possible), acquisition des données,
- Simplification de la structure logiciel (chaque fonctionnalité est gérée par un thread),
- Architecture plus robuste (une partie de l'application ne met plus en péril tout le reste),
- Tâche de surveillance en parallèle d'une tâche de traitement,

Multithreading

Performance

Pourquoi travailler avec des threads ?

- l'OS n'a pas besoin de créer de nouveau emplacement dans la mémoire comme dans le cas des processus,

- L'OS n'a pas besoin également d'allouer de nouvelles structures pour assurer le suivi de l'état des fichiers ouverts et incrémenter le nombre de références sur les descripteurs de fichiers ouverts,

- Il est trivial de partager des données entre les threads . Les mêmes variables globales et statiques peuvent être lues et écrites entre tous les threads d'un processus. Avec un processus mono-thread , le système d'exploitation ne peut rien faire pour faire laisser le processus tirer parti de plusieurs processeurs,

- L'ordonnanceur peut programmer différents threads pour une exécution en parallèle sur différents cœurs de processeurs.

Évaluation du temps d'un processus avec time

La commande **time** permet de mesurer le temps d'exécution d'une commande. Elle fournit les temps ***réels*** (temps total), ***utilisateurs*** (durée nécessaire au processeur pour exécuter les ordres du programme) et ***systèmes*** (durée nécessaire au processeur pour traiter les ordres du système d'exploitation).

Voici un exemple d'utilisation :

time ls -lR / > liste.ls 2> /dev/null

Le résultat s'affiche alors (attention la commande elle même peut être swappée le résultat peut être alors inexacte) :
real 2m39.458s
user 0m9.060s
sys 0m32.330s

Extrait du *man* time:

- **Sys**: Total number of CPU-seconds used by the system on behalf of the process (in kernel mode), in seconds.
- **User**: Total number of CPU-seconds that the process used directly (in user mode), in seconds.
- **Real**: Elapsed real (wall clock) time used by the process, in [hours:]minutes:seconds.

Évaluation du temps d'un processus avec time

On peut ainsi calculer le **facteur d'évaluation** qui traduit la surcharge du système :

$$facteur = \frac{temps_{utilisateur} + temps_{systeme}}{temps_{reel}}$$

Un facteur normal se situe entre **1/5** et **1/10**. Un facteur supérieur à **1/20** traduit une surcharge du système.

Tests de performance

Test1: 50.000 processus, avec fork()

Test2: 50.000 thread, avec pthread_create()

Multithreading

Performance

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

Efficient Communications/Data Exchange:

Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
Intel 2.6 GHz Xeon E5-2670	4.5	51.2
Intel 2.8 GHz Xeon 5660	5.6	32
AMD 2.3 GHz Opteron	1.8	5.3
AMD 2.4 GHz Opteron	1.2	5.3
IBM 1.9 GHz POWER5 p5-575	4.1	16
IBM 1.5 GHz POWER4	2.1	4
Intel 2.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium 2	1.8	6.4

Hyperthreading

L'hyperthreading est une technologie qui a été introduite par Intel, avec l'objectif premier d'améliorer le support du code multithread. Dans certaines conditions de travail, la technologie de l'hyperthreading garantit une utilisation plus efficace des ressources processeurs en exécutant les threads en parallèle sur un seul et unique processeur.

Un processeur équipé en hypertheading prétend faire office de deux processeurs "logiques" pour le système d'exploitation hôte, permettant au système d'exploitation de prévoir deux tâches (threads) ou deux processus simultanément. Les avantages de l'hyperthreading sont les suivants : un support amélioré pour le code multithread, il permet à plusieurs threads de fonctionner simultanément, et offre un temps de réaction et de réponse amélioré.

Multithreading

Terminologie

Système multitâche

Un système multitâche, c'est la possibilité d'avoir plusieurs programmes travaillant en même temps: impression de parallélisme du point de vue de l'utilisateur.

Processus / processus lourd

Un processus (en anglais, *process*), est un programme en cours d'exécution par un ordinateur.

Thread / processus légers / tâche

Les programmes utilisent le concept de thread : un fil d'instructions (un chemin d'exécution) à l'intérieur d'un processus. Un programme peut avoir plusieurs threads.

Multithreading / multitâche

Un programme qui exécute plusieurs threads en même temps est appelé programme multithread ou encore multithreadé.

Préemption

Dans un système d'exploitation multitâche préemptif, les processus ne sont pas autorisés à prendre un temps non-défini pour s'exécuter dans le processeur. Une quantité de temps définie est attribuée à chaque processus ; si la tâche n'est pas accomplie avant la limite fixée, le processus est renvoyé dans la pile pour laisser place au processus suivant dans la file d'attente, qui est alors exécuté par le processeur. Ce droit de préemption peut tout aussi bien survenir avec des interruptions matérielles.

Multithreading

Terminologie

La pile (stack)

Structure de données fondée sur le principe « dernier arrivé, premier sorti ». Stockage des données de travail.

Le tas(heap)

Le tas est une zone où est réalisée l'allocation dynamique avec les fonctions Xalloc().

Ordonnanceur

L'ordonnanceur est un composant du système d'exploitation qui choisit l'ordre d'exécution des processus. En anglais, l'ordonnanceur est appelé *scheduler*.

Commutation des thread

Un processeur qui exécute plusieurs activités donne l'illusion de partager son temps de façon concurrente. Mise en attente du thread actif (courant) dans la liste des threads bloqués ou prêts

- Sauvegarde de son contexte (pointeur sur les variables du thread, compteur d'instruction program counter, priorité du thread, contenu des registres du processeur)
- Recherche du thread ayant la plus haute priorité ou élection à partir d'un quantum de temps
- Restauration du contexte du thread élu (registre du processeur)
- Activation du thread élu

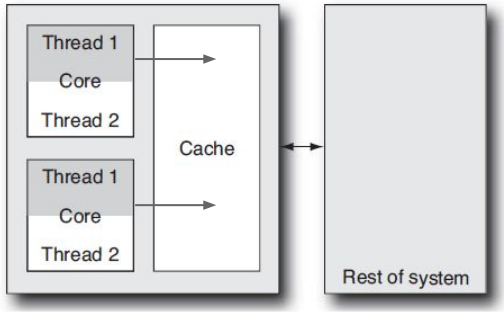
Multithreading

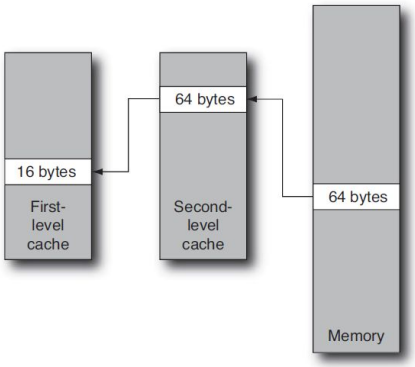
Partie II

Codage performant
Normalisation
Mise en oeuvre en C/C++
Mise en oeuvre Java
IPC
Thread et temps réel

Multithreading

Stratégie de chargement





Les threads accèdent à la mémoire cache du CPU afin d'optimiser les transferts vers les registres. Il existe une latence d'accès. Pour réduire cette latence, on effectue un pré-chargement à partir d'une zone de stockage mémoire (DRAM, …)

Les accès au premier cache du processeur sont plus rapides. Cependant une donnée peut être accessible uniquement en dehors du cache du CPU à cause de sa taille. Afin de réduire la latence, le processeur utilise plusieurs stratégies:

- **out-of-order**: le CPU charge les prochaines données et les données de l'instruction en cours,
- **hardware prefetching**: le processeur détecte et précharge les données à lire en mémoire avant que le processeur le demande,
- **Software prefetching**: préchargement à partir d'une adresse non linéaire.

Multithreading

Préchargement

Par exemple, le préchargement sera en mesure de charger les éléments de A et B dans le code suivant, car la "foulée" (la distance d'accès dans la RAM) est fixée et prédictible:

```
for(int i = 0; i < N; i++)
    A[i*4] = B[i*3];
```

Multithreading

Préchargement

Ici le préchargement n'est pas possible car il faut exécuter avant un rand(), puis exécuter l'opération, la "foulée" n'est pas fixe:

```
for(int i = 0; i < N; i++)
    A[i*4] = B[rand()%N];
```

Multithreading

Impacte des structures sur les performances

L'ordre dans lequel les variables sont déclarées et utilisées en mémoire peut améliorer les performances.

```
struct s
{
 int var1;          // 4 bytes
 int padding1[15];  // 60 bytes
 int var2;          // 4 bytes
}
```

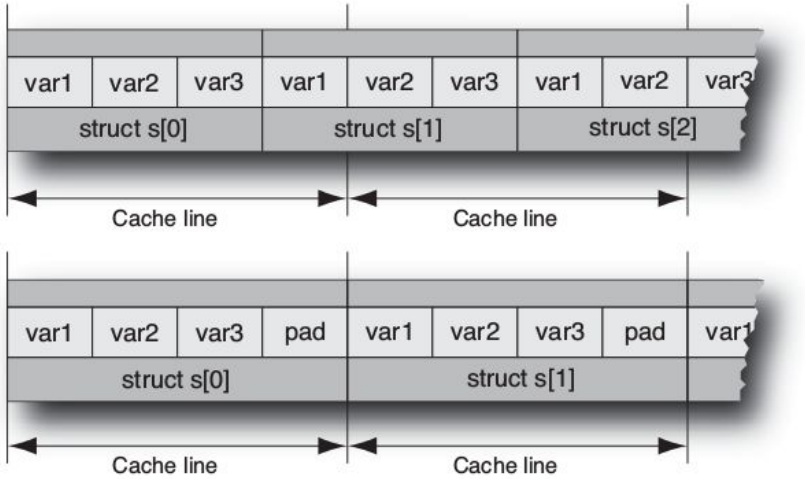
L'accès à **var1** membre de la structure demande un chargement de 64 octets. La taille d'une variable entière est de 4 octets, de sorte que la taille totale de var1, plus padding 1 est de 64 octets. Ceci entraîne le positionnement de var2 sur une autre ligne du cache.

```
struct s
{
 int var1;                // 4 bytes
 int var2;                // 4 bytes
 int padding1[15];        // 60 bytes
 }
```

Si on réordonnance la structure, alors var1 et var2 seront sur la même ligne de cache.

Multithreading

Impacte des structures sur les performances

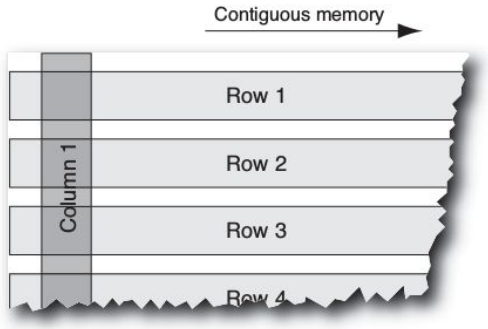


Multithreading

Utilisation des tableaux

Le tableau est un modèle standard de stockage de données. La performance de l'application est optimale si le tableau est agencé de telle sorte que l'accès aux éléments sélectionnés soient contigus. Listing ci-dessous montre un exemple d'accès à un tableau avec une distance non contiguë.

```
{
double ** array;
double total=0;
...
for (int i=0; i<cols; i++)
for (int j=0; j<rows; j++)
total += array[j][i];
...
}
```



En C/C ++, les tableaux sont disposés dans la mémoire de telle sorte que les éléments d'une ligne soient adjacents (ici indexé par la variable i); ce qu'on appelle "ligne-major". Cependant, la boucle interne itère avec **J** et accède à des éléments qui ne sont pas contiguës dans ma mémoire. La plupart des compilateurs arrivent à corriger (mais pas toujours!) l'ordre d'accès et à améliorer les accès mémoires.

Multithreading

Accès natif

Accès au thread natif du système d'exploitation:

-C

-C++

Pas d'accès au thread natif du système d'exploitation:

-Java

-Python

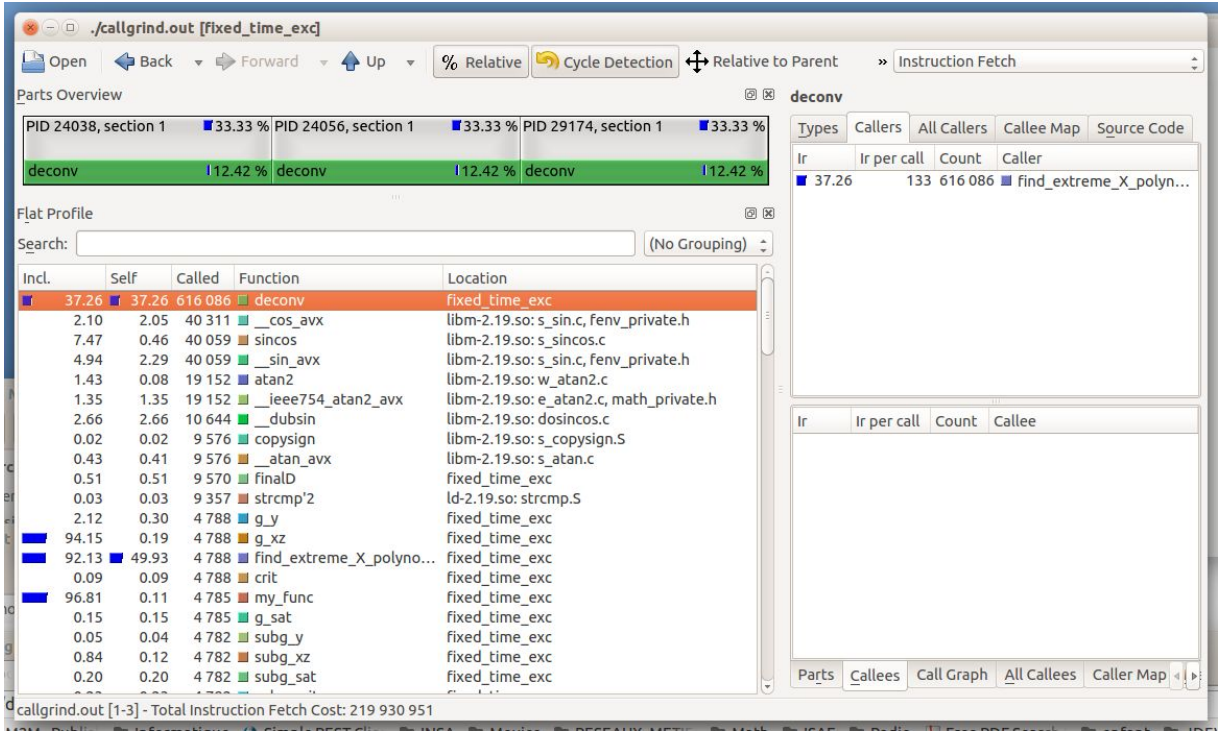
-NET Framework

Multithreading

Profilage

valgrind

`valgrind --tool=callgrind mon_programme` → génération d'un rapport callgrind.out.29174 → affichage du rapport: kcachegrind



Multithreading

Profilage

gprof

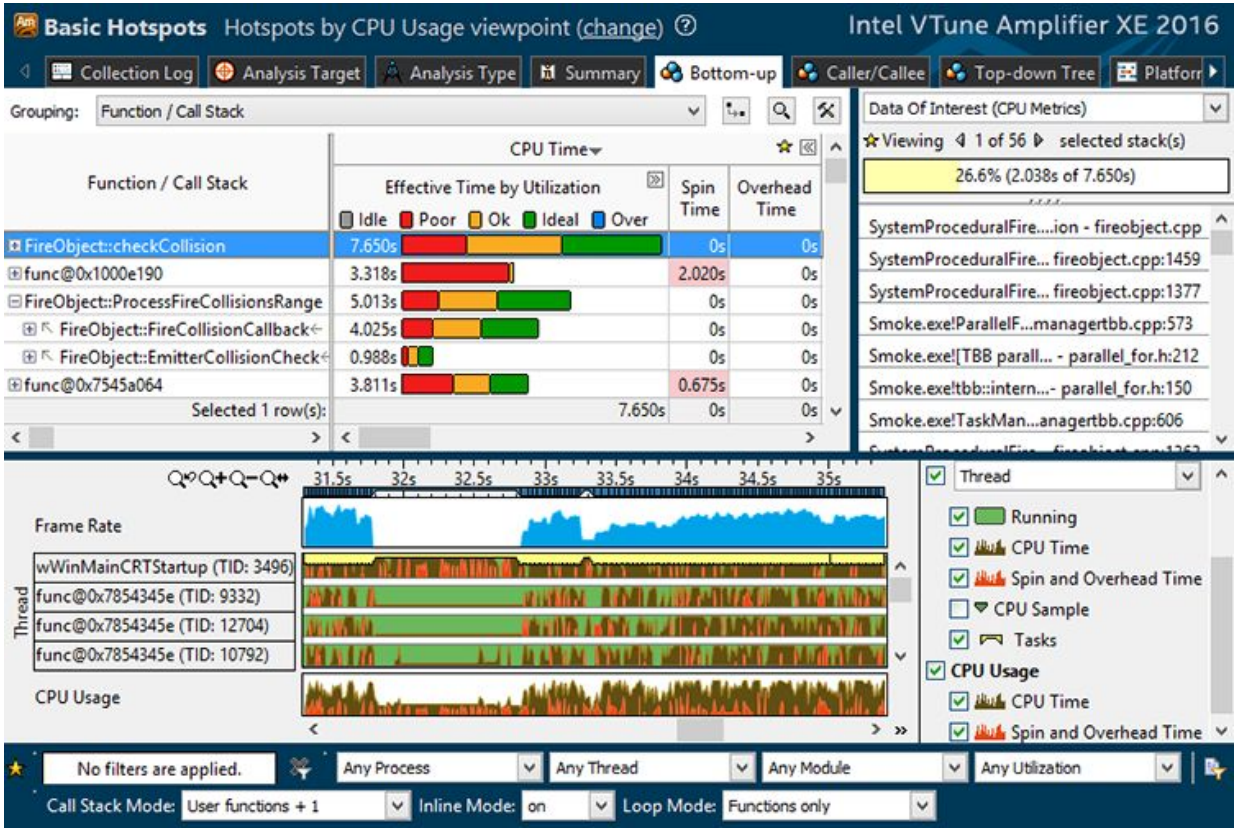
option qui permet la génération du fichier de profilage

```
$ gcc -Wall -pg test_gprof.c -o test_gprof

$ gprof test_gprof gmon.out > analysis.txt


$ vi analysis.txt

Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 34.43     14.26    14.26        1    14.26    28.05  func1
 33.31     28.05    13.79        1    13.79    13.79  func2
 33.31     41.84    13.79        1    13.79    13.79  new_func1
  0.12     41.89     0.05                             main
….
```

Multithreading

Profilage

vtune



Multithreading

Compilation

gcc/g++

option d'optimisation permet un gain important dans certains cas

```
$ gcc -Wall -pg -O2 test_gprof.c -o test_gprof

...
```

Multithreading

Outils

gprof: https://sourceware.org/binutils/docs/gprof/

valgrind: http://valgrind.org/info/tools.html

oprofile: http://oprofile.sourceforge.net/news/

gperftools: https://github.com/gperftools/gperftools

vtune: https://software.intel.com/en-us/intel-vtune-amplifier-xe

Multithreading

Normalisation

Afin de tirer pleinement parti des capacités fournies par des threads, une interface de programmation standardisée est nécessaire. Pour les systèmes UNIX, cette interface a été spécifiée par la norme de l'**IEEE POSIX** (1995). Les implémentations qui adhèrent à cette norme sont appelés threads **POSIX** ou **Pthreads**. La plupart des fournisseurs de matériel offrent maintenant Pthreads en plus de leur API propriétaire.

La plupart des systèmes d'exploitation UNIX et UNIX-like respectent les principales caractéristiques de POSIX. Par conséquent, une application codée avec cette norme norme sera portable entre les implémentations UNIX et les systèmes d'exploitation de type UNIX, tels que Linux, FreeBSD et Mac OS X, qui sont également basés sur un noyau UNIX.

Microsoft Windows ne met pas en œuvre la normes POSIX directement, bien qu'il existe quelques solutions qui permettent aux programmes POSIX de fonctionner sur les plateformes Windows (CygWin). Le support Multithreading sous Microsoft Windows est largement similaire à l'appui fourni par les threads POSIX. Les différences sont en grande partie dans le nom des fonctions de l'API, plutôt que dans les fonctionnalités.

Multithreading

Normalisation POSIX

POSIX.1c, Threads extensions ⇔ IEEE Std 1003.1c-1995

POSIX.1c

Threads extensions

- Thread Creation, Control, and Cleanup
- Thread Scheduling
- Thread Synchronization
- Signal Handling

Multithreading

OS à faible empreinte

En dehors de Linux, il existe de nombreux support pour l'embarqué, qui se présente comme une librairie et pas vraiment comme un OS. Le code de l'OS est compilé en même temps que l'application embarqué. Le support n'est pas toujours POSIX, le plus souvent il est propriétaire.

Ces OS présentent une très faible mémoire de l'ordre d'une dizaine Kb.

Il n'est pas nécessaire d'embarquer Linux systématiquement pour avoir un très bon support multithread !

Bien que très réduit, les composants de l'OS présentent les outils nécessaires pour gérer la concurrence.

Multithreading

OS à faible empreinte

threadX : développé et commercialisé par Logic Express pour RTOS:
http://rtos.com/products/threadx/

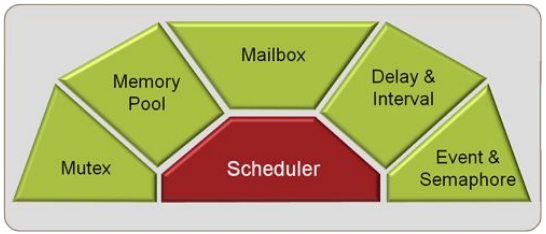
Minimal Kernel Size: Under 2K bytes
Queue Services: 900 bytes
Semaphore Services: 450 bytes
Mutex Services: 1200 bytes
Block Memory Services: 550 bytes
Minimal RAM requirement: 500 bytes
Minimal ROM requirement: 2K bytes

Multithreading

OS à faible empreinte

ARM / RTX : API de multithread pour CORTEX M, Round-Robin, Preemptive, Cooperative
http://www.keil.com/rl-arm/rtx_specs.asp
http://www.keil.com/rl-arm/kernel.asp

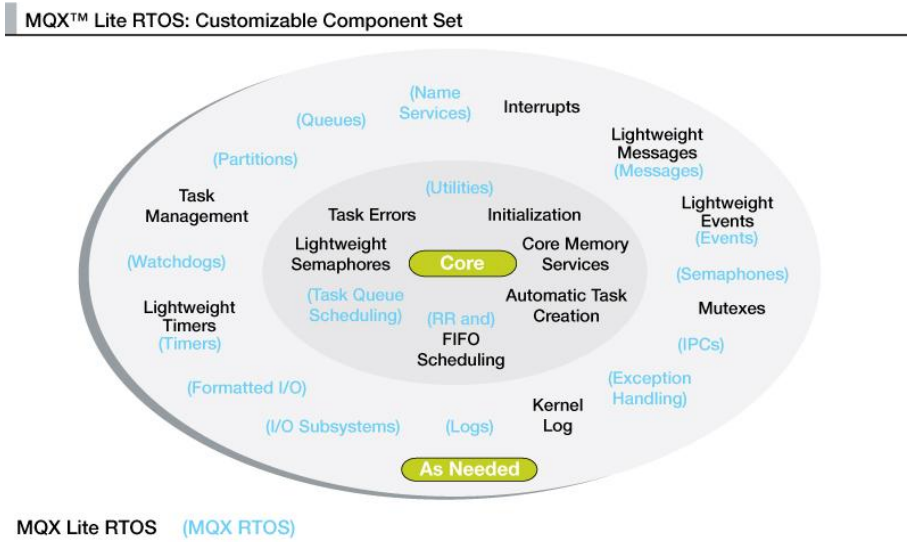
CODE Size	< 4.0 KBytes
RAM Space for Kernel	< 300 Bytes + 128 Bytes User Stack
RAM Space for a Task	TaskStackSize + 52 Bytes
RAM Space for a Mailbox	MaxMessages*4 + 16 Bytes
RAM Space for a Semaphore	8 Bytes
RAM Space for a Mutex	12 Bytes
RAM Space for a User Timer	8 Bytes
Hardware Requirements	SysTick timer



Multithreading

OS à faible empreinte

NXP: NXP MQX™ Lite RTOS (4 KB RAM)
http://www.nxp.com/pages/mqx-lite-rtos:MQXLITERTOS



Multithreading

Mise en oeuvre en C / Posix

```
gcc-Wall  stack.c -lpthread
```

```
#include <pthread.h>
#include <stdio.h>

void* thread_code( void * param )
{
      printf( "In thread code\n" );
}


int main()
{
      pthread_t thread;
      pthread_create( &thread, 0, &thread_code, 0 );
      printf( "In main thread\n" );
      pthread_join( thread, 0 );
}
```

procédure exécutée par le thread

création du thread

attente de la fin du thread

```
android@pc3-18:~$ ./thread
In main thread
In thread code
```

Multithreading

Mise en oeuvre en C / Posix

Détaché un thread avec pthread_detach()

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

void *slave_entry(void *arg)
{
	int i;
	pthread_t self;

	self = pthread_self();
	pthread_detach(self);

	for (i = 0; i < 3; i++) {
	printf("slave thread: %d\n", i);
	sleep(1);
	}
	return NULL;
}

int main()
{
	int error;
	pthread_t slave_tid;
	pthread_create(&slave_tid, NULL, slave_entry, NULL);
	printf("main thread exit\n");
	pthread_exit(NULL);
	return 0;
}
```

Dans le précédent exemple, le main attend la fin du thread. Il est possible d'utiliser des threads détachés qui s'exécutent même si le *main* se termine.

```
gcc stack.c -lpthread
```

```
pc@pc3-18:~$ ./thread
main thread exit
slave thread: 0
slave thread: 1
slave thread: 2
```

Multithreading

Mise en oeuvre en C++11 - Posix

```
#include <iostream>
#include <thread>

//This function will be called from a thread

void call_from_thread()
{
std::cout << "Hello, World" << std::endl;
}

int main()
{
  //Launch a thread
  std::thread t1(call_from_thread);
  //Join the thread with the main thread
  t1.join();
  return 0;
}
```

```
android@pc3-18:~$ g++ -std=c++11 -pthread thread.c -o thread
android@pc3-18:~$ ./thread
Hello, World
```

Multithreading

Mise en oeuvre en C / Priorité

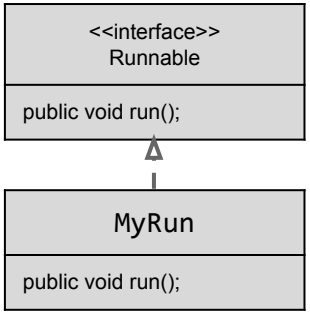
La modification de le priorité nécessite le compte root. Les primitives sont les suivantes:

```
#include <pthread.h>

pthread_setschedparam(pthread_t thread, int policy,
                      const struct sched_param *param);
pthread_getschedparam(pthread_t thread, int *policy,
                      struct sched_param *param);
```

En ligne de commande, il est possible d'utiliser "nice":

```
fcamps@enterprise:~/dev/captronic$ nice -n13 emacs toto.txt
```

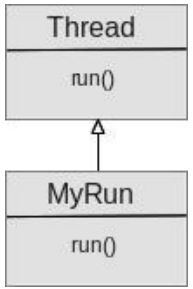
Thread et API Java - interface Runnable




```
public class MyRun implements Runnable {
  public void run() {
    while(true) { // Boucle infinie ou conditionnelle
      // Traitement à faire
    }
  }

  public static void main(String args[ ]) {
     // le constructeur prend en paramètre un thread
     (new Thread(new MyRun())).start();
  }
}
```

L'interface Runnable définit une seule méthode, run(), destiné à contenir le code exécuté. La méthode run() est exécutée par la machine virtuelle Java.

Thread et API Java - extends



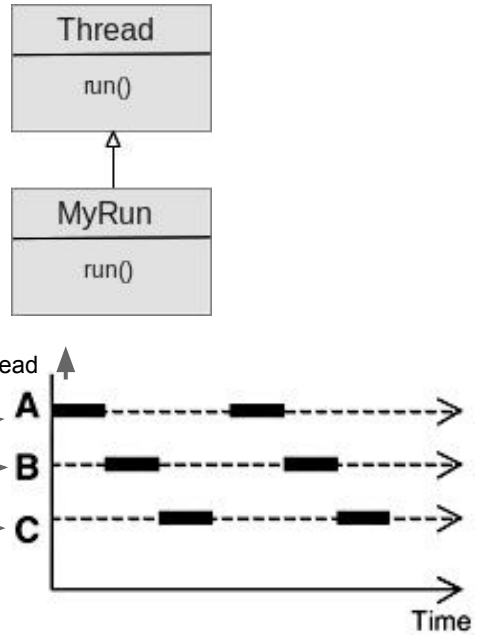

```
class MyRun extends Thread {
      public void run() {
            while (true) {
                  // Boucle infinie ou conditionnelle
                  // pour effectuer des traitements.
            }
      }

      public static void main(String args[]) {
            (new MyRun()).start();
            // ou encore
            // MyRun p = new MyRun();
            // p.start();
      }
}
```

Sous-classe Thread. La classe Thread elle-même implémente Runnable.

Plusieurs Threads

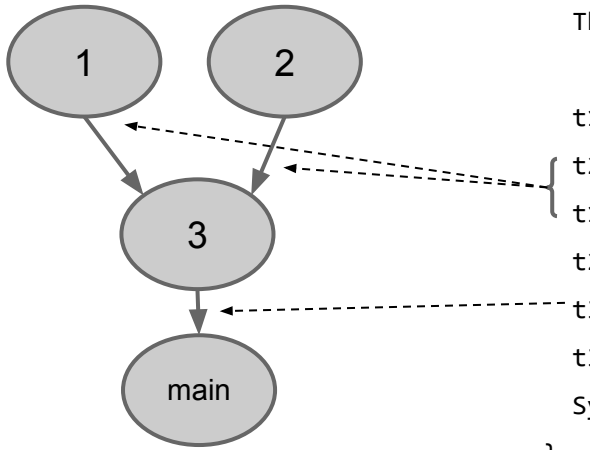
```
public class MyRun extends Thread {

	public void run() {
		// traitement
	}

	/**
	 * @param args
	 */
	public static void main(String[] args) {

		MyRun A = new MyRun();
		MyRun B = new MyRun();
		MyRun C = new MyRun();
		A.start();
		B.start();
		C.start();
	}
}
```

Pour exécuter plusieurs threads :



Graphe de dépendance de threads

Dans certains programmes, il est nécessaire d'activer des threads selon un graphe:




```
public class MonThread8 extends Thread {
  public void run() {
    // code à exécuter
  }
  public static void main(String[] args) {
    Thread t1 = new MonThread8();
    Thread t2 = new MonThread8();
    Thread t3 = new MonThread8();

    t1.start();
    t2.start();
    t1.join();
    t2.join();
    t3.start();
    t3.join();
    System.out.println("Fin");
  }
}
```

attente de 1 et 2 pour démarrer 3

attente de 3 par le main puis fin

Java : Priorité d'un thread

- La priorité donne une indication à l'ordonnanceur Java pour augmenter le niveau d'éligibilité d'un thread et par conséquent augmenter son temps d'exécution par rapport aux autres threads.

- Les ordonnanceurs des systèmes d'exploitation s'appliquent à fournir un ordonnancement bien plus évolué que celui qui est requis par la spécification du langage Java. Un mapping est réalisé entre les threads de la machine virtuelle et le système hôte : les niveaux de priorité Java ne sont pas toujours respectés, voir complètement ignorés.

- L'API Java définit dix niveaux de priorités qui peuvent correspondre aux priorités d'ordonnancement du système d'exploitation (mais pas toujours).

- Dans la plupart des applications Java, tous les threads ont la même priorité, Thread.NORM_PRIORITY. Par défaut un thread a la priorité Thread.NORM_PRIORITY.

- Le mécanisme de priorité des threads est un instrument tranchant et il n'est pas toujours évident de savoir quels seront les effets d'un changement de priorité ; augmenter celle d'un thread peut ne rien donner ou faire en sorte qu'un seul thread soit planifié de préférence à l'autre, d'où une famine de ce dernier.

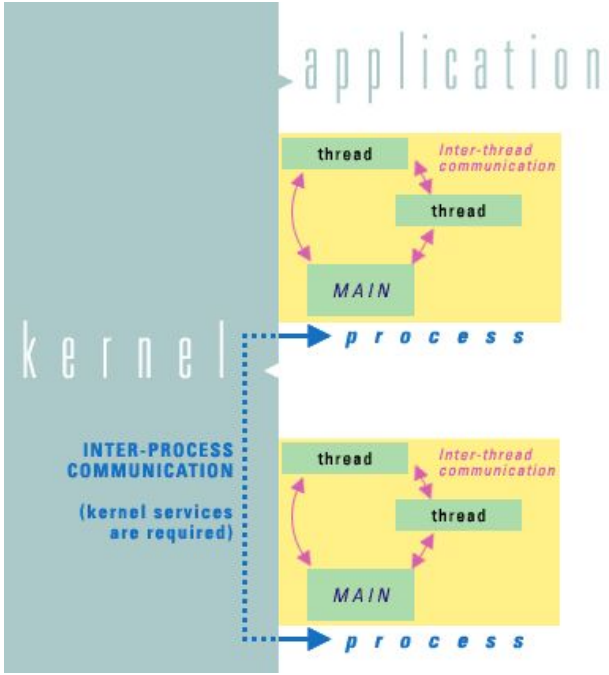
IPC

Les **IPC** (Inter-Process Communication) permettent la communication entre les processus et les threads. On distingue principalement les techniques suivantes:

- mémoire partagée
- tube et tube nommé
- signaux
- mailbox

En langage C/C++, il existe deux normes (+ propriétaire) : System V et POSIX. La plupart des OS complexes proposent les deux possibilités.

La programmation des IPC peut être complexe à mettre en oeuvre.



IPC
Normalisation POSIX

POSIX.1b

Real-time extensions

- Priority Scheduling
- Real-Time Signals
- Clocks and Timers
- Semaphores
- Message Passing
- Shared Memory
- Asynch and Synch I/O
- Memory Locking

IPC

Mise en oeuvre en langage C/Signaux

```
...
void execFils()
{
  struct sigaction action;
  int pid = getpid();
  int pidPere = getppid();

  printf("Processus Fils : %d  ->  Pere : %d\n", pid, pidPere);

  // Mask de signal pour lequel correspond un signal pour chaque bit
  sigset_t maskSignal;

  /* mask avec tous les signaux */
  if(sigfillset(&maskSignal))
    {
      printf("Probleme vidage du mask de signaux\n");
      exit(-1);
    }

  /* On enleve SIGUSR1 du mask car il doit passer*/
  if (sigdelset(&maskSignal, SIGUSR1))
    {
      printf("Probleme suppression du signal SIGUSR1 dans le mask\n");
      exit(-2);
    }

  // On applique  : (tous les signaux - SIGUSR1)
  sigprocmask(SIG_SETMASK, &maskSignal, NULL);

  /* mise en place du handler */
  action.sa_handler = &handlerFils;
  sigaction(SIGUSR1, &action, NULL);

  // Attente du signal SIGUSR1
  sigsuspend(&maskSignal);
}
```

sensibilité aux signaux

réception

traitement

IPC

Mise en oeuvre en langage C/Mémoire partagée

```
int main()
{
        const char *name = "OS";
        const int SIZE = 4096;

        int shm_fd;
        void *ptr;

        /* open the shared memory segment */
        shm_fd = shm_open(name, O_RDONLY, 0666);
        if (shm_fd == -1) {
                printf("shared memory failed\n");
                exit(-1);
        }

        /* now map the shared memory segment in the address space of the process */
        ptr = mmap(0,SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
        if (ptr == MAP_FAILED) {
                printf("Map failed\n");
                exit(-1);
        }

        /* now read from the shared memory region */
        printf("%s",(char *)ptr);

        /* remove the shared memory segment */
        if (shm_unlink(name) == -1) {
                printf("Error removing %s\n",name);
                exit(-1);
        }

        return 0;
}
```

IPC

Mise en oeuvre en langage C/Mailbox

```
int main(int argc,char * argv[]) // Recepteur
{
    mqd_t mq;
    int taille;
    char * buffer;
    long int duree;
    struct mq_attr attr;
    struct timeval heure;
    struct timeval * recue;

    if (argc != 2) {
        fprintf(stderr, "usage: %s nom_file_messagen", argv[0]);
        exit(EXIT_FAILURE);
    }

    mq = mq_open(argv[1], O_RDONLY | O_CREAT, 0600, NULL);
    if (mq == (mqd_t) -1) {
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }

    if (mq_getattr(mq, & attr) != 0) {
        perror("mq_getattr");
        exit(EXIT_FAILURE);
    }
    taille = attr.mq_msgsize;
    buffer = malloc(taille);

    if (buffer == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    recue = (struct timeval *) buffer;
    while (1) {
        mq_receive(mq, buffer, taille, NULL);
        gettimeofday(& heure, NULL);
        duree  = heure.tv_sec - recue->tv_sec;
        duree *= 1000000;
        duree += heure.tv_usec - recue->tv_usec;
        fprintf(stdout, "%ld usecn", duree);
    }
    return EXIT_SUCCESS;
}
// gcc -Wall recepteur-01.c -o recepteur-01 -lrt -pthread
// ./recepteur-01 /msg
```

```
#include <fcntl.h>
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>

int main(int argc,char * argv[]) // Emetteur
{
    mqd_t mq;
    struct timeval heure;

    if (argc != 2) {
        fprintf(stderr, "usage: %s nom_file_messagen", argv[0]);
        exit(EXIT_FAILURE);
    }

    mq = mq_open(argv[1], O_WRONLY | O_CREAT, 0600, NULL);
    if (mq == (mqd_t) -1) {
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }
    while (1) {
        gettimeofday(& heure, NULL);
        mq_send(mq, (char *) & heure, sizeof(heure), 1);
        sleep(1);
    }
    return EXIT_SUCCESS;
}


//  gcc -Wall recepteur-01.c -o recepteur-01 -lrt -pthread
// ./recepteur-01 /msg
```

Thread et temps réel

- Les threads RT* s'appuient sur un OS et un ordonnanceur RT,
- Linux par exemple est par défaut par préemption, donc pas RT. Sinon il faut soit utiliser un patch de type Xenomai ou Preempt RT ou utiliser un OS RTOS,
- Les priorités permettent de gérer le quantum de temps dans le CPU,
- Dans le cas d'un Linux RTOS l'ordonnanceur peut être du type suivant: **Normal, FIFO and Round Robin**.

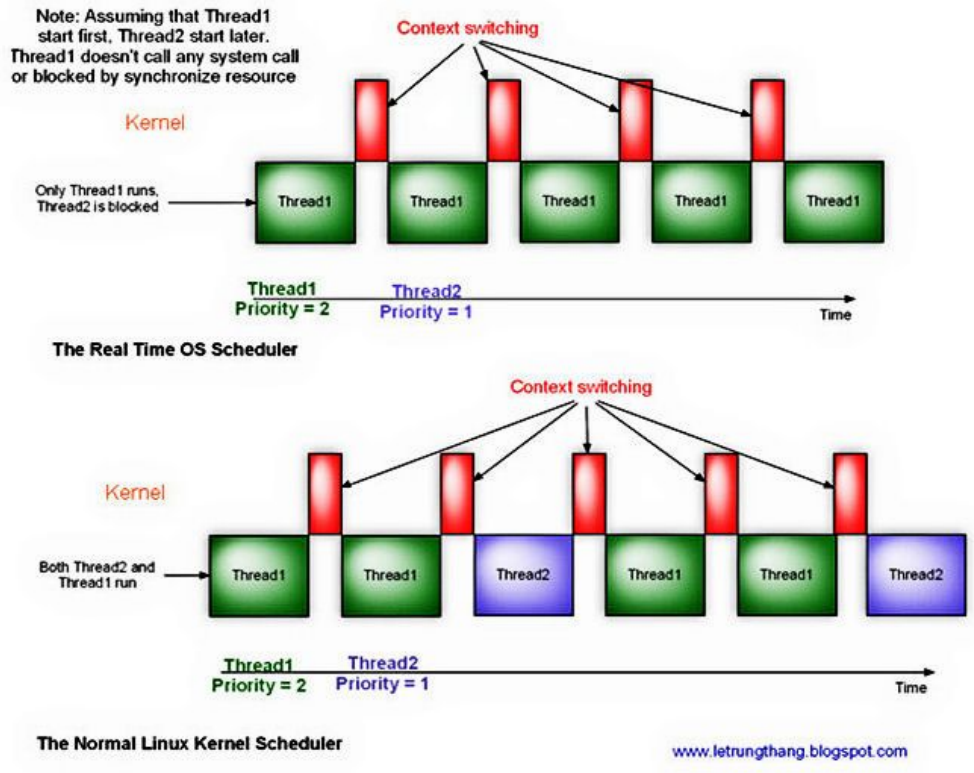
*RT:Real Time

Thread et temps réel

Hypothèse: Thread1 a une priorité de 2 et thread1 une priorité de 1

Dans l'ordonnanceur RTOS , Thread1 , avec une priorité plus élevée , fonctionne toujours . Thread2 avec une priorité inférieure n'a jamais une chance d' être exécuté.

Dans un ordonnanceur non RTOS: Par comparaison, un système Linux par défaut fait exactement le contraire , ce qui autorise le fonctionnement de Thread1 et Thread2. Thread1 avec une priorité plus élevée aura plus de temps d'exécution par rapport à Thread2 qui a une priorité inférieure .



Thread et temps réel

Dans le cas d'un **ordonnancement "normal"**, un thread peut être stoppé (suspendu) lorsque l'un de ces trois conditions apparaît:

1- Le thread est bloqué lors d'un accès à une ressource (I/O block, mutex, semaphone...)

2- Le thread abandonne son quantum de temps (call sleep() or pthread_yield())

3- L'ordonnanceur suspend le thread lorsque son quantum de temps est atteint. Le temps d'exécution du thread dépend alors de sa priorité

Thread et temps réel

Ordonnancement FIFO: un thread peut être stoppé (suspendu) lorsque l'un de ces trois conditions apparaît:

1- Le thread est bloqué lors d'un accès à une ressource (I/O block, mutex, semaphone...)

2- Le thread est préempté par un thread de haute priorité,

3- Le thread abandonne son quantum de temps (call sleep() or pthread_yield())

Thread et temps réel

Ordonnancement Round Robin: un thread peut être stoppé (suspendu) lorsque l'un de ces trois conditions apparaît:

1- Le thread est bloqué lors d'un accès à une ressource (I/O block, mutex, semaphone...)

2- Le thread est préempté par un thread de haute priorité,

3- Le thread abandonne son quantum de temps (call sleep() or pthread_yield())

4- Le quantum de temps (time slice) d'exécution est expiré

Note: la plupart des RTOS utilisent Round Rbin (noté aussi RR).

Partie II

Gestion de la concurrence
Section critique
Algo de Peterson
Exclusion mutuelle matériel
Variable atomique
Opération atomique
Barrière de synchronisation
Implémentation en C
Implémentation en Java

Gestion de la concurrence

Accès concurrent à une ressource

Que se passe-t-il dans cet exemple ?

- Le code est exécuté par deux threads
- Le problème : Toutes ces opérations doivent se faire **en une seule fois, sans interruption**, sinon l'état devient incohérent.

  On parle de **section critique**

  - Exemple : exécution par deux threads
  - Le thread 1 exécute les 2 premières lignes avec les valeurs x=300 et y=1 et est interrompu avec x=301
  - Le thread 2 s'était arrêté juste avant d'exécuter dessine(x,y) et dessine(301,y) au lieu de dessine(0,y). L'algorithme n'est pas respecté car x ne devrait pas dépasser 300 pour l'affichage !

```
x=x+1;
if (x>300) {
    x=0 ;
    y=y+1;
}
dessiner (x,y);
```

Section critique, atomicité

Notion garantissant qu'un thread est seul à exécuter un morceau de code

On parle aussi de zone d'**exclusion mutuelle.**

Zone de section critique / exclusion mutuelle :

Un ensemble d'instructions qui doit être exécuté de manière **atomique** (sans être interrompu) par au plus un **unique thread** à la fois

Forme générale d'une section critique

```
PROTOCOLE ENTRÉE (PARAMÈTRES)    -----> entrée en section critique

            lire variable x             -----> section critique
            incrémenter la variable x   -----> section critique

PROTOCOLE DE SORTIE (PARAMÈTRES) -----> sortie de section critique
```

Protocole d'entrée : il est responsable de garantir qu'au plus un thread entre en section critique.

--> Si la section critique est occupée, un thread doit être bloqué en attendant la libération de la section critique.

Protocole de sortie : il permet généralement de faire entrer un thread qui était bloqué en attente de la libération de la section critique.

Section critique : le code qui doit être exécuté par un unique thread à la fois

--> En général, le code sera simple et court, sans boucle, pour éviter de faire attendre inutilement les autres threads.

Section critique

La définition précédente est trop faible : il suffit de ne jamais autoriser l'entrée en section critique...

Un mécanisme d'exclusion mutuelle doit vérifier **3 propriétés** :

Sûreté (*safety*)

Exclusion Mutuelle: au plus un thread est dans la section critique

Vivacité (*liveness*)

Progrès: Si un thread demande à entrer en section critique et qu'aucun autre thread ne l'occupe, il finira par rentrer

Attente bornée

Un thread qui demande à entrer en section critique finira par y être autorisé au bout d'un temps fini, quelles que soient les requêtes des autres threads.

Implémentation de l'exclusion mutuelle : l'algorithme de Peterson


```
//flag[] tableau de boolean ;  turn un entier
flag[0]   = false;
flag[1]   = false;
turn;
```

P0:

```
flag[0] = true;
turn = 1;
while (flag[1] == true && turn == 1)
{
   // busy wait
}
// critical section
...
// end of critical section
flag[0] = false;
```

P1:

```
flag[1] = true;
turn = 0;
while (flag[0] == true && turn == 0)
{
   // busy wait
}
// critical section
...
// end of critical section
flag[1] = false;
```

Questions :

- Identifier les 3 zones (entrée, sortie, section critique)

- Vérifier que ce protocole implémente bien les propriétés:
  - exclusion mutuelle
  - progrès
  - attente bornée

- Quel problème identifiez vous ?

Hypothèse :

- Les opérations sur les variables sont toutes atomiques.

Implémentation de l'exclusion mutuelle par le matériel

Les processeurs conçus pour les opérations en parallèle disposent d'instructions spéciales pour gérer les accès concurrents aux variables partagées. Les instructions du processeur peuvent réaliser des opérations atomiques :

- ***testAndSet*** : pour les accès mémoire

- ***fetch-and-add*** : récupérer une valeur et ajouter une valeur (utilisé pour les accès mémoires et les exclusions mutuelles)

- ***compare-and-swap*** (CAS) : instruction atomique qui permet de comparer une valeur et de réaliser un échange

Autres instructions (en fonction du processeur : ARM, x86, PowerPC ...) :

- **CompareExchange**
-**Compare-And-Swap**

Variables atomiques

Performance : Les variables atomiques offrent une sémantique mémoire qui permet de réaliser des opérations non interruptibles sans utiliser d'appel système. Le code est très performant.

Interblocage : Les algorithmes qui utilisent les variables atomiques sont non bloquants mais plus complexes à mettre en oeuvre, par contre ils sont immunisés contre les interblocages et autres problèmes de vivacité. Avec les algorithmes qui reposent sur les verrous, les autres threads ne peuvent pas progresser si un thread se met en sommeil ou en boucle pendant qu'il détient le verrou, alors que les algorithmes non bloquants sont imperméables aux échecs des différents threads.

Variables atomiques

Le problème des instructions non atomiques, provient du fait qu'elles peuvent être décomposées en instructions élémentaires que chaque thread peut alors exécuter à n'importe quel instant :

counter = counter + 1;

Cette instruction peut s'écrire :

1- Charger counter dans un registre de processeur
2- Ajouter +1 au registre
3- Placer le résultat dans counter

Opérations atomiques

Une suite d'opération avec des variables atomiques ne garantit pas un comportement déterministe. Il faut intégrer les opérations dans une section critique:

Protocole d'entrée

counter1 = counter1 + 1;

counter2 = counter1 + 1;

Protocole de sortie

Variable atomiques / section critiques ?

Les variables atomiques:

- plus performant car très simple
- toutes les opérations arithmétiques ne sont pas atomiques

Section critiques:

- pour une suite d'opération atomique dépendante
- pour une suite d'opération non atomique

Barrière de synchronisation

Rappels :

- Les mécanismes d'exclusion mutuelle en attente active

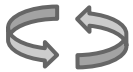
Protocole d'entrée qui boucle en attente de libération de ressource

Section critique

Protocole de sortie

- Problèmes :
  - Consommateurs de temps CPU pour attendre
  - Le temps processeur n'est pas optimisé
  - Problème de ressource si trop de processus en attente

Barrière de synchronisation passive

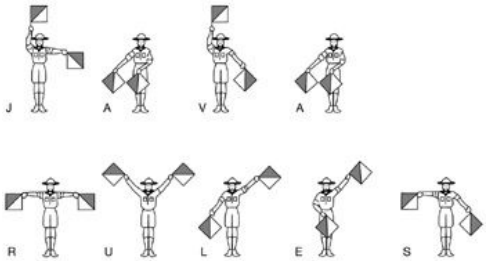


Protocole d'entrée : autorisation entrée section critique ? **non --> thread en attente passive jusqu'à autorisation**

Section critique

Protocole de sortie

Utilisation des sémaphores pour encadrer la section critique. Les sémaphores constituent une généralisation du mécanisme des verrous. Ils ont été proposés par E. W. Dijkstra.



Sémaphore : Propriétés

Propriété 0 : Le sémaphore est constitué d'une variable entière e (nombre de jeton), d'une file d'attente associée fifo et de deux primitives : P() ou acquire() pour prendre un jeton ; et V() ou release() pour libérer un jeton. Ces deux primitives sont **atomiques**.

Pseudo code :

```
function P(semaphore S, integer e):  // "Puis-je"
   if S >= e:
         [S ← S - 1]
   fi

function V(semaphore S, integer e): // "Vas-y"
   [S ← S + 1]
```

Sémaphore : Propriétés

Propriété 1 : Le sémaphore est initialisé avec une valeur entière, supérieure ou égale à 0.

Propriété 2 : Si valeur d'initialisation est e = n > 0 alors le sémaphore autorise à n threads l'accès à une section critique, la ressource est dite n-partageable.

Propriété 3 : Si l'on désire mettre en oeuvre une exclusion mutuelle pour l'accès à une section critique, alors il suffit d'initialiser le sémaphore à 1. On obtiendra alors une protection similaire au verrou. Ce type de sémaphore est appelé, sémaphore d'exclusion mutuelle : **mutex**.
On trouve aussi le nom de **sémaphore binaire**, qui prend que deux valeurs: 0 ou 1.

Sémaphore : Propriétés

Propriété 4 : Si le sémaphore est initialisé à 0, alors il est bloqué.

Propriété 5 : Tant que e est positif alors la section critique peut être accédée par les threads ; si e=0 après l'appel de P() alors le prochain thread sera bloqué ; le protocole de sortie de la section critique incrémente e, un thread en attente pourra alors entrer.

Mutex

Le **mutex** se présente comme un **sémaphore binaire** qui ne peut prendre que deux valeurs, 0 ou 1. Si le sémaphore est “pris” il possède la valeur 0 sinon 1.

Les différents langages implémentent ou pas ce principe. Par exemple, en Java la notion de mutex n’existe par directement, il faut créer un sémaphore binaire. En C et C++, la notion de mutex est implémentée.

Mutex

Implémentation en C++11

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex m;
int i = 0;

void makeACallFromPhoneBooth(int num)
{
    m.lock();
        std::cout << num << " " << i << " Hello " << std::endl;
        i++;
    m.unlock();
}

int main()
{
int a=1,b=2,c=3;
std::thread man1(makeACallFromPhoneBooth,a);
std::thread man2(makeACallFromPhoneBooth,b);
std::thread man3(makeACallFromPhoneBooth,c);

man1.join();//man1 finished his phone call and joins the crowd
man2.join();//man2 finished his phone call and joins the crowd
man3.join();//man3 finished his phone call and joins the crowd
return 0;
}
```

```
android@pc3-18:~$ g++ -std=c++11 -pthread thread.c -o
thread
android@pc3-18:~$ ./thread
1 0 Hello
2 1 Hello
3 2 Hello
```

Variable atomic

Implémentation en C++

```cpp
#include <atomic>
struct AtomicCounter {
    std::atomic<int> value;

    void increment(){
        ++value;
    }

    void decrement(){
        --value;
    }

    int get(){
        return value.load();
    }
};
```

Mutex

Implémentation en Java

```
public class Example {
    private int value = 0;

    private final Semaphore mutex = new Semaphore(1);

    public int getNextValue() throws InterruptedException {
        try {
            mutex.acquire();
            return value++;
        } finally {
            mutex.release();
        }
    }
}
```

Synchronisation en Java

Exclusion mutuelle en Java :

- Mot clef **`synchronized`**
  Garantit l'atomicité d'une portion de code :

```
boolean synchronized testAndSet( ) {
      boolean oldValue = occupee;
      occupee = true;
       return oldValue;
}
```

- Sémantique :
  Si un thread exécute une méthode protégée par synchronized, tout autre thread voulant exécuter la méthode est mis en attente
  - Attente passive
    (pas d'utilisation de ressource CPU)

Exclusion mutuelle Java : cas de sections critiques multiples

Exemple : implémentation d'un compteur :

- une variable partagée : int compteur
- 3 méthodes :

```
public void synchronized incremente() {
   compteur++;}

public void synchronized decremente(){
   compteur--;}

int getCompteur(){
   return compteur;}
```

- Sémantique : un thread au plus se trouve dans incremente() ou decremente()
  compteur++ est exécuté de manière atomique (alors que l'opération ++ n'est pas atomique en soit)

- Pour getCompteur(), Java garantit qu'une lecture ou une écriture sur un type simple est atomique

Exclusion mutuelle Java : synchronisation sur un Object

Java généralise ce mécanisme à tout objet :

```
synchronized (obj) {
 // section critique
}
```

`synchronized` est valide sur tout objet (héritant de Object)

- `public synchronized int method(...)`
  - un seul thread peut appeler la méthode à un instant donné (tout autre appel concurrent place le thread en attente)
- `synchronized(obj) { code }`
  - dans tous tous les blocs de code gardés par `synchronized(obj)` (il peut y en avoir plusieurs avec le même objet `obj`) il y a au plus un seul thread
- `verrou avec this : (code équivalent)`

```
public void methode() {
    synchronized(this) {
        //section critique
    }
}
```

```
synchronized void methode() {
    //section critique
}
```

Partage d'une mémoire entre objet


```
static int compteur;

public static void synchronized incremente() {
   compteur++;
}
```

la méthode `static` évite des méthodes d'instance, le verrou se fait sur le même objet (même barrière pour tous les threads)

Attention aux deadlocks (verrous mortels)

- ```
  void virementBancaire(Compte a, Compte b, int montant) {
    synchronized(a) {
       synchronized(b) {
          b.credite(montant);
          a.debite(montant);
  }}} // libération verrou a et b à la sortie
  ```

- Problème si on a en même temps :
  - viremmentBancaire(a,b,100);
  - viremmentBancaire(b,a,100);
  - "en attendant Godot" :
    - le premier thread a le verrou sur a et attend pour le b
    - le second thread a le verrou sur b et attend pour le a

- Ce problème arrive ici car la synchronisation ne se fait pas au niveau de la méthode mais au niveau des barrières de verrous
  ```

Synchronisation de méthodes ou d'objet ?

- Intérêt de la synchronisation de méthode
  - plus simple
  - naturel

- Intérêt de la synchronisation d'objet
  - permet de réduire la taille des blocs synchronisés
  - le code exécuté en synchronized est beaucoup plus lent

- **En pratique**
  - `synchronized` est nécessaire dès qu'une variable est accédée en lecture et écriture par plus d'un thread.
  - Si deux variables sont correllées, leurs modifications doivent se faire en même temps dans le même bloc synchronisé.

Réentrance

Lorsqu'un thread demande un verrou qui est déjà verrouillé par un autre thread, le thread demandeur est bloqué. Les verrous internes étant **réentrants**, si un thread tente de prendre un verrou qu'il détient déjà, la requête réussit. La réentrance signifie que les verrous sont acquis par appel :

```
public class Widget {
public synchronized void doSomething() {
...
}
}

public class LoggingWidget extends Widget {
public synchronized void doSomething() {
System.out.println(toString() + ": calling doSomething");
super.doSomething();
}
}
```

Les méthodes doSomething() de Widget et LoggingWidget étant toutes les deux synchronized, chacune tente d'obtenir le verrou sur le Widget avant de continuer. Si les verrous internes n'étaient pas réentrants, l'appel à super.doSomething() ne pourrait jamais obtenir le verrou puisque ce dernier serait considéré comme déjà pris : le thread serait bloqué en permanence en attente d'un verrou qu'il ne pourra jamais obtenir.

Variables et Opérations Atomiques

Certains langages de programmation dont Java, proposent des variables et opérations dîtes atomiques (ressemble aux opérations TAS). Le but est le suivant (package java.util.concurrent.atomic) :

- **Garantir des opérations atomiques de ces variables**
  - exemple x=x+1 devient complétement atomique avec x.getAndIncrement()
  - x=x-1 devient x.getAndDecrement()
- **Simplifier le code** : on peut supprimer les synchronized pour les opérations simples
- **Proposer des opérateurs et types** : int, boolean ...

Exemple pour un int :

- AtomicInteger count = new AtomicInteger(0);
- count.getAndIncrement(); // --> ++
- count.get() // --> retourne la valeur

Verrous explicites avec délai

Les modes d'acquisition de verrou scrutable et avec délai offerts par tryLock() autorisent une gestion des erreurs plus sophistiquée qu'avec une acquisition inconditionnelle. Avec les verrous internes, un interblocage est fatal – la seule façon de s'en sortir est de relancer l'application et le seul moyen de s'en prémunir consiste à construire le programme pour empêcher un ordre d'acquisition incohérent. Le verrouillage scrutable et avec délai offre une autre possibilité : l'évitement probabiliste des interblocages.

```
final Lock lock = new ReentrantLock();

if (lock.tryLock(500, NANOSECONDS))
{
  try {
          // do something;
        }
  finally
  {
       lock.unlock();
  }
}
else
{
   // do something;
}
```

Partie IV

- Risques et problèmes
- Patron de conception
- Introduction réseau de Pétri

Risques et problèmes du multithreading

Blocage et interblocage d'un thread

La programmation multithread est délicate car un thread mal conçu peut rester bloqué dans un état et entraîner des dysfonctionnements du programme, voire du système. Le blocage d'un thread peut causer le blocage d'un autre thread, on appelle ceci l'interblocage.

Complexité des algorithmes

L'utilisation des threads rend un système plus complexe, car ce sont des programmes concurrents pour l'utilisation des ressources matérielles.

Parallélisation des algorithmes

Les threads sont souvent utilisés pour paralléliser l'exécution d'un algorithme (calcul, affichage, système ...). La parallélisation pose de nombreux problèmes de synchronisation et peut entraîner des blocages / interblocages.

Modéliser les accès aux ressources
Réseau de Pétri

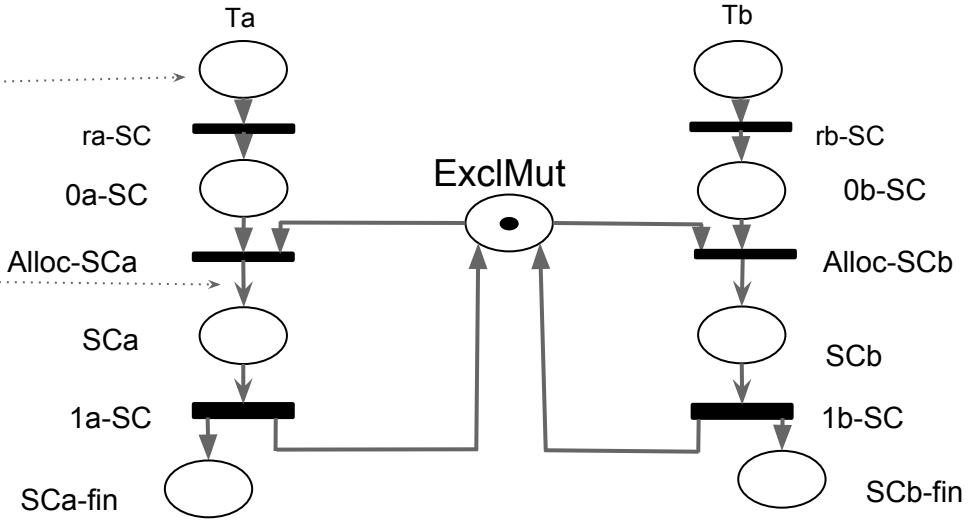
Introduction : La conception d'une architecture complexe est une tâche très délicate, car il est nécessaire de prendre en compte les traitements internes afin d'éviter les interblocages. Les langages de programmation fournissent des API qui permettent de construire un système qui partage ses ressources entre ses différents constituants, cependant les langages ne fournissent pas de méthodologie d'analyse pour concevoir des systèmes complexes. Il est donc nécessaire d'avoir une approche méthodologique qui permet la compréhension, l'analyse d'un système sur laquelle s'appuiera la phase de conception.

Les réseaux de Pétri est un concept créé par Carl Adam Petri en 1962 afin de modéliser la composition et la communication entre automates. Ce concept permet de dégager au plus tôt les interblocage dans les systèmes les plus complexes.

Analyse des systèmes complexes

Les systèmes complexes doivent être abordés avec méthode. En effet si un système possède un grand nombre d'interblocages possible, l'analyse du code ou la définition d'un algorithme ne permet pas facilement de détecter les comportements complexes. L'utilisation des réseaux de Petri peut faciliter l'analyse.

- un jeton dans une place permet de définir l'état du système,
- pour passer d'un état à un autre il faut que la place possède un jeton, alors on peut tirer la transition,
- le fonctionnement dépend fortement de l'état initial.



Modélisation de l'exclusion mutuelle en réseau de Petri

Ti = Tâche n° i
SC = Section Critique
Ri = Requête de demande d'entrée en section critique par la tâche Ti
Li = Libération par Ti

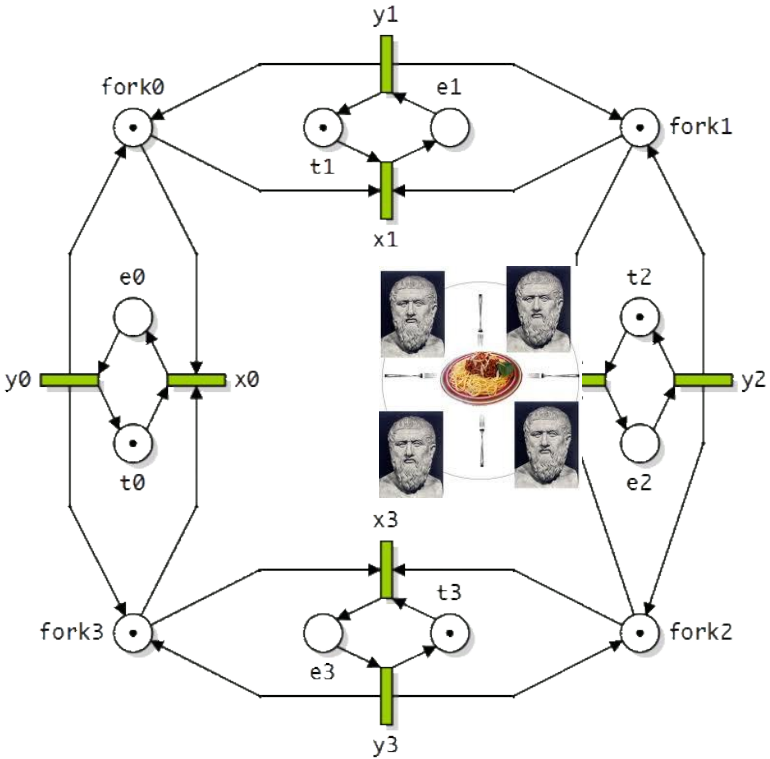
Analyse des systèmes complexes

Introduction : La conception d'une architecture complexe est une tâche très délicate, car il est nécessaire de prendre en compte les traitements internes afin d'éviter les interblocages. Les langages de programmation fournissent des API qui permettent de construire un système qui partage ses ressources entre ses différents constituants, cependant les langages ne fournissent pas de méthodologie d'analyse pour concevoir des systèmes complexes. Il est donc nécessaire d'avoir une approche méthodologique qui permet la compréhension, l'analyse d'un système sur laquelle s'appuiera la phase de conception.

Les réseaux de Pétri est un concept créé par Carl Adam Petri en 1962 afin de modéliser la composition et la communication entre automates. Ce concept permet de dégager dès la conception les interblocages dans les systèmes les plus complexes.

Analyse des systèmes complexes

Exemple du paradigme du "*dîner des philosophes*" :

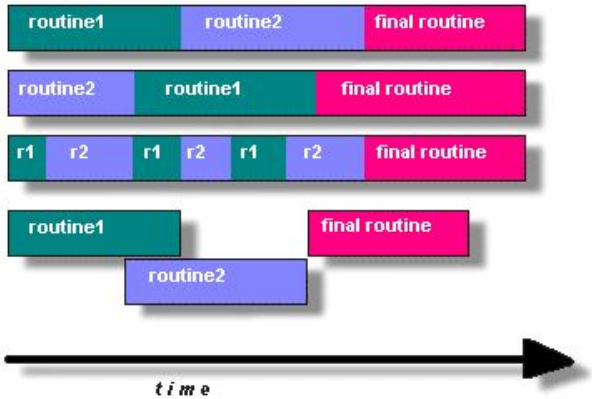


Multithreading

Conception

Éléments à prendre en considération lors de la conception d'un système multithread:

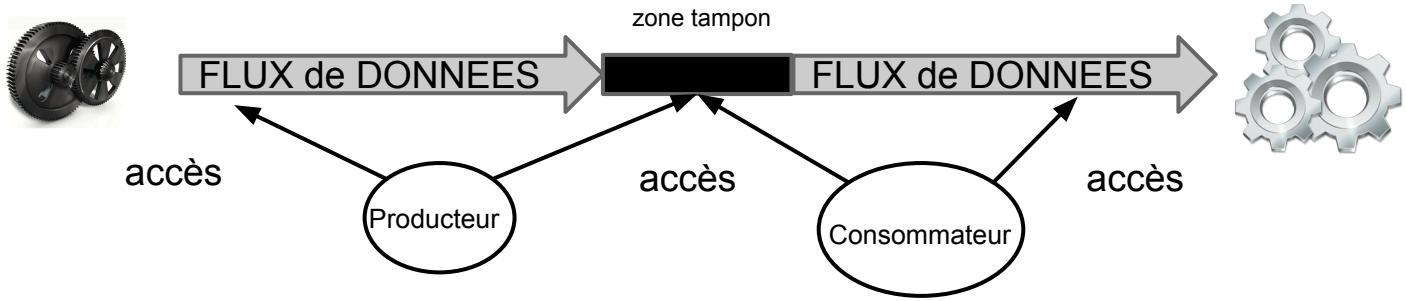
- Quel type de modèle utilisé ?
- Quel type de partitionnement temporel
- Équilibrage de charge
- Communication (périphérique …)
- Dépendance des données
- Synchronisation et "race conditions"
- Accès mémoire
- Accès I/O
- Complexité des algorithmes
- Programmation effort/costs/time
- Choix de l'OS



Producteur / Consommateur

Le système producteur / consommateur est un paradigme classique qui nous permet de présenter les principaux problèmes qui se posent dans une coopération entre threads lors de l'accès à des variables communes avec synchronisation.

Supposons pour commencer une paire de threads : un seul thread producteur, un seul thread consommateur. Etant donné que les threads s'exécutent concurremment et généralement à des vitesses différentes, une zone tampon est nécessaire pour stocker la production qui n'est pas encore consommée.

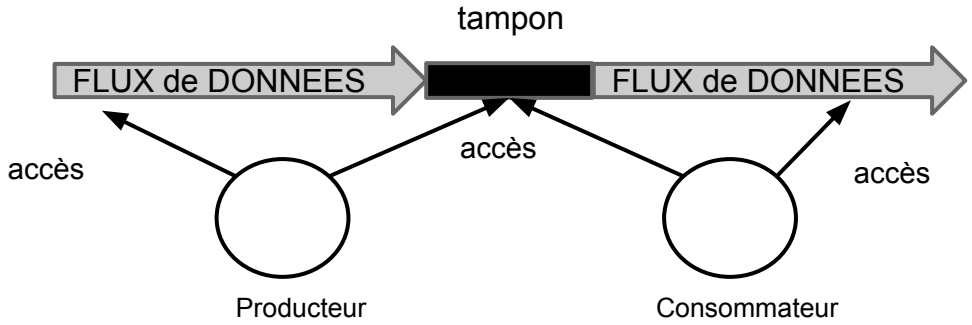


Producteur / Consommateur

Définition du modèle

Afin que la solution soit valide, il faut donc que les **propriétés** suivantes soient respectées :

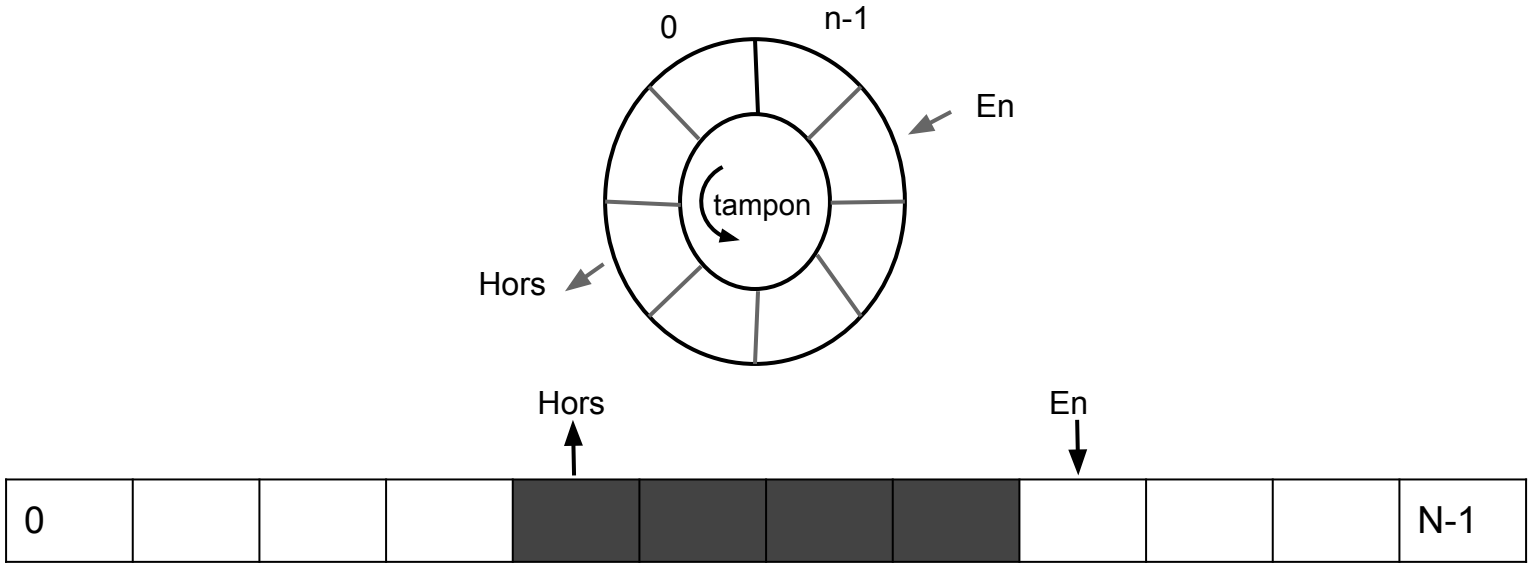
- **Propriété 1** : Le thread consommateur ne peut pas prélever un message que le thread producteur est en train de déposer (exclusion mutuelle de la zone tampon)
- **Propriété 2** : Le thread producteur ne peut pas placer un message dans la zone tampon si celui-ci est plein (interdiction de surimpression d'une valeur). Si le tampon est plein, le thread producteur est mis en attente passive, et ce jusqu'à ce que le tampon ait au moins un emplacement disponible
- **Propriété 3** : Le consommateur ne consomme qu'une seule fois le message. Si le tampon est vide, il est mis en attente jusqu'à ce qu'un nouveau message soit disponible dans le tampon.
- **Propriété 4** : Producteur et consommateur ne doivent pas être bloqués en même temps
- **Propriété 5** : Les messages sont consommés dans l'ordre de leur production; de plus le producteur et le consommateur n'opèrent jamais simultanément sur le même message ou la même zone dans le tampon.



Producteur / Consommateur

Modèle tampon borné

Les vitesses des threads sont généralement distinctes. Il peut se produire différents cas de fonctionnement : un thread peut avoir une production plus élevée, un thread peut être bloqué, un thread consommateur n'arrive pas à suivre une cadence. Afin d'éviter que de tels comportements n'aient des effets négatifs sur la progression et la vivacité du système dans son ensemble, une solution consiste à étendre la capacité du tampon à N messages pour absorber les écarts de débit (tampon circulaire).



Lecteur / Rédacteur

Des objets ou des ressources tels qu'un enregistrement, un fichier, un répertoire ou même un sous-ensemble de registres peuvent avoir la particularité d'être sujets au partage entre plusieurs threads ou processus. Plus précisément, plusieurs threads sont susceptibles de lire ou de consulter les données que concrétisent ces objets, ce sont des **threads de type lecteur** (*reader* dans la littérature).

D'autres threads sont susceptibles de modifier ces mêmes données ou informations, se sont des threads de type **rédacteur** (*writer*).

Les opérations d'écritures s'avèrent toutefois critiques. Il convient en effet de les réaliser en exclusion mutuelle, c'est-à-dire une à la fois. Car les résultats qui découleraient d'une exécution concurrente de threads rédacteurs seraient vraisemblablement imprédictibles.

Les lectures et écritures doivent donc être mutuellement exclusives. Par contre l'on peut imaginer que plusieurs threads lecteurs puissent simplement et concurremment accéder en lecture à une même ressource, en d'autre terme, consulter simultanément une information : un fichier, un répertoire.

Le problème ici décrit est connu sous le paradigme des **lecteur-rédacteur** ou **lecteur-écrivain**.

Lecteur / Rédacteur

Propriétés des systèmes lecteur/rédacteur

Propriété 1 : Un nombre quelconque de threads peut accéder en lecture à la ressource, donc l'accès est concurrent mais pas exclusif en lecture

Propriété 2 : Un seul thread à la fois peut accéder en écriture à la ressource, donc l'accès est exclusif en écriture

Propriété 3 : Pendant une écriture, aucun thread ou processus lecteur ne peut accéder à la ressource et vice versa, en d'autres termes aucune écriture n'est autorisée pendant une lecture : les accès en lecture et en écriture sont mutuellement exclusifs.

On considère : qu'un thread n'appartient, au moment où il accède à l'objet (ressource), qu'à une des deux catégories lecteur/rédacteur. Soit il consulte la ressource (lecteur), soit il la modifie (écriture).

Lecteur / Rédacteur

Priorités des accès

Un certain nombre de politique de priorités peuvent généralement être pris en considération avec le paradigme Lecteur/Rédacteur :

- Les lecteurs ont la priorité sur les rédacteurs
- Les lecteurs ont la priorité sur les rédacteurs uniquement si un lecteur accède déjà à la ressource
- Les lecteurs et les rédacteurs ont la même priorité
- Les rédacteurs ont la priorité sur les lecteurs
- Les lecteurs et rédacteurs ont une priorité indéfinie qui dépend du système (?)

Multithreading

Conception - pattern de développement

D’une façon générique, on remarque les architectures suivantes:

Simple tâche

Création d’un thread pour répondre à une tâche spécifique , habituellement de façon asynchrone à partir du processus principal du programme. Lorsque le thread a terminée, il sort .

Worker threads

Dans ce modèle, un processus peut avoir un certain nombre de tâches distinctes qui peuvent être exécutées simultanément les uns avec les autres. Un thread est créé pour chacun de ces éléments de travail. Par exemple, dans un programme de traitement de texte , vous pouvez avoir un thread séparé qui est responsable du traitement d'entrée et d'autres commandes de l'utilisateur , tandis qu'un autre thread est chargé de générer la mise en page à l'écran.

Thread pools

Le travail est partagé par un ensemble de thread qui sont préparés au démarrage du dispositif. Chacun réalise un travail identifié dans une file d’attente.

Patrons de conception / Design Patterns

Leader/Followers

Prenons un exemple, imaginons une station de taxi, tous les conducteurs de taxi dorment sauf un, le taxi leader. Un pool de threads représente ici l'ensemble des taxis :

Le leader attend un client comme un thread peut attendre un événement.

Lorsqu'un client arrive alors le leader réveille un autre taxi qui sera à son tour le nouveau leader, puis répond à la demande du client.

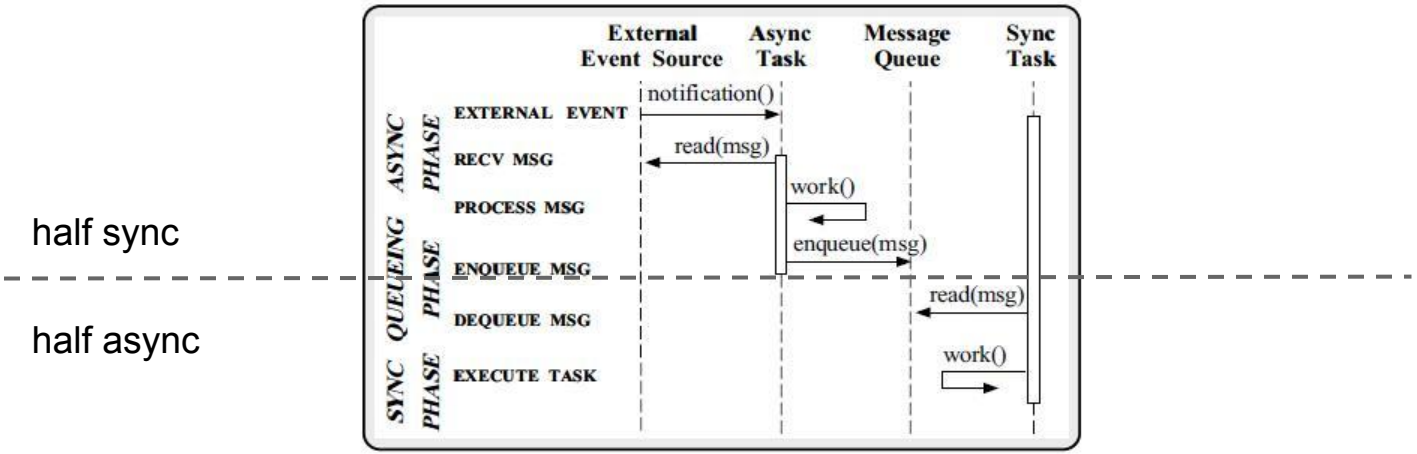
Lorsque le taxi a fini sa course, il revient dans le pool de taxi et attend d'être sollicité comme leader si besoin.

Patrons de conception / Design Patterns

Half-Sync/Half-Async

Le modèle architectural Half-Sync/Half-Async découple le traitement asynchrone* et synchrone* dans les systèmes concurrents, pour simplifier la programmation sans réduire les performances. Ce modèle introduit deux couches communicantes, l'une pour les traitements asynchrones et l'autre pour le traitement de service synchrone. Une couche intermédiaire avec une file d'attente joue le rôle de médiateur entre les couches synchrone et asynchrone.

* En mode **synchrone**, le processus appelant attend que le processus appelé ait renvoyé sa réponse pour continuer à s'exécuter,
* En mode **asynchrone**, le processus appelant continue à travailler pendant que le processus appelé exécute le traitement demandé et gère via un événement - ou éventuellement via une instruction de synchronisation - le(s) retour(s) du processus appelé.



Partie V

Introduction OpenMP

Introduction OpenMP

OpenMP (Open Multi-Processing) est une interface de programmation pour le calcul parallèle sur architecture à mémoire partagée. Cette API est supportée sur de nombreuses plateformes, incluant **GNU/Linux**, **OS X** et **Windows**, pour les langages de programmation **C, C++ et Fortran**. Il se présente sous la forme d'un ensemble de directives, d'une bibliothèque logicielle et de variables d'environnement.

OpenMP est **portable** et **dimensionnable**. Il permet de développer rapidement des applications parallèles à petite granularité en restant proche du code séquentiel.

Introduction OpenMP

```
#include <omp.h>

  main ()  {

  int var1, var2, var3;

  Serial code
      .
      .
      .

  Beginning of parallel region. Fork a team of threads.
  Specify variable scoping

  #pragma omp parallel private(var1, var2) shared(var3)
    {

    Parallel region executed by all threads
          .
    Other OpenMP directives
          .
    Run-time Library calls
          .
    All threads join master thread and disband

    }

  Resume serial code
      .
      .
      .

  }
```

Introduction OpenMP

Nombre de processeur:

```
enterprise:~> grep 'processor.*:' /proc/cpuinfo | wc -l
8
```

```
#include "stdio.h"

int main(int argc, char *argv[])
{
  #pragma omp parallel
  {
    printf("hello multicore user!\n");
  }
  return(0);
}
```

1

```
### Basic Makefile for OpenMP

CC = /usr/bin/gcc
CFLAGS = -g -O0 -fopenmp
LD = /usr/bin/gcc
LDFLAGS = -g -fopenmp

PROGRAM = opnmp

all:    ${PROGRAM}.exe

${PROGRAM}.exe: ${PROGRAM}.o
	${LD} ${LDFLAGS} $< -o ${PROGRAM}.exe

${PROGRAM}.o: ${PROGRAM}.c
	${CC} ${CFLAGS} -c $< -o ${PROGRAM}.o

clean:
	rm -f ${PROGRAM}.o ${PROGRAM}.exe
```

2

```
fcamps@enterprise:~/dev/captronic$ make
/usr/bin/gcc -g -O0 -fopenmp -c opnmp.c -o
opnmp.o
/usr/bin/gcc -g -fopenmp opnmp.o -o opnmp.exe

./opnmp.exe
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
```

3

Introduction OpenMP

Augmenter le nombre de processeur

```
fcamps@enterprise:~/dev/captronic$ export OMP_NUM_THREADS="16"
fcamps@enterprise:~/dev/captronic$ ./opnmp.exe
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
hello multicore user!
```

Multithreading

Sources

https://fr.wikipedia.org/wiki/Thread_(informatique)

https://en.wikipedia.org/wiki/Program_counter

https://computing.llnl.gov/tutorials/parallel_comp/

https://www.cs.rutgers.edu/~pxk/416/notes/05-threads.html

http://www-igm.univ-mlv.fr/~dr/NCSPDF/chapitre07.pdf

https://fr.wikipedia.org/wiki/Multit%C3%A2che_pr%C3%A9emptif

https://en.wikipedia.org/wiki/Amdahl%27s_law