Threads POSIX (1) Création et gestion

Introduction

- Loi de Moore plus trop d'actualité
- La puissance de calcul augmente maintenant (majoritairement) avec la multiplication des coeurs, des processeurs et des machines
- Pour tirer parti des machines multicoeurs : utilisation d'algorithmes parallèles et programmes multithreadés

Introduction

Il est possible d'implémenter des algorithmes parallèles avec ce qu'on a vu jusque là, mais c'est lourd :

- `fork` : appel système lourd
- chaque processus a son espace mémoire (perte de mémoire)
- chaque processus a ses structures systèmes (table des pages, fichiers ouverts...)
- *context switch* lent
- communication inter-processus généralement lente

Solution : les processus légers ("threads")

Processus légers (*threads*)

- Plusieurs visions et implémentations possibles
- Introduction dans la norme POSIX en 1995 (POSIX 1.c)

Différence entre un thread et un processus normal :

- un thread est une "partie" d'un processus
- un processus est l'exécution d'un ensemble (>=1) de threads

Différence entre un ensemble de threads et un ensemble de processus :

- les threads partagent pratiquement tout (mémoire, pid, fichiers ouverts...)
- la synchronisation n'est plus gérée au niveau du système, mais est laissée à l'utilisateur

Threads : avantages et inconvénients

Avantages et inconvénients par rapport à des processus communiquants avec les "anciens" mécanismes

- partage de la mémoire : mécanisme rapide de communication inter-thread
- plus léger : moins de données système à recopier
- plus rapide : le context-switch est plus facile

Les inconvénients sont (uniquement) des "difficultés" de programmation :

- Les threads utilisent les mêmes copies des librairies : les librairies doivent être "MT-safe"
- Il faut gérer la synchronisation (mutex, sémaphores...)

En cas de mauvaise synchronisation :

- Comportement "aléatoire" : bugs, segfaults, exploitations...
- Interblocage

Ordonnancement des threads (et processus) :

- Coopératif :
  Chaque thread doit explicitement rendre la main.
  Problème : si un thread plante ou ne rend pas la main, les autres threads ne s'exécutent plus.
- Préemptif :
  Le système peut arrêter n'importe quel thread, pour switcher à un autre thread (via un mécanisme de temporisation et d'interruption matérielle)

L'ordonnancement des systèmes d'exploitation "modernes" se fait préemptivement (Unix, windows...). Mais l'ordonnancement coopératif peut toujours exister au niveau utilisateur.

Modèle 1 :1 (threads système ou threads noyau)

- Chaque thread correspond à une entité ordonnancée par le noyau.
- L'ordonnancement est alors préemptive.
- Le comportement va être proche d'un ensemble de processus qui partagent leur mémoire et leurs données système.

Avantage : permet à un processus d'utiliser plusieurs coeurs

Implémentations de threads 1 :1 : LinuxThread, NPTL

Modèle N :1 (threads utilisateurs)

- Tous les threads du processus correspondent à une entité ordonnancée par le noyau.
- L'ordonnancement et le *switch* entre les threads se fait au niveau utilisateur

Avantage :

- le *switch* est très rapide (pas d'appel système)
- possibilité d'avoir énormément de threads
- possible même sur des systèmes légers, sans ordonanceur (systèmes embarqués)

Implémentations de threads N :1 : GNU Pth, State Threads

Modèle N :1 (threads utilisateurs)

- Certains langages/paradigmes de programmation utilisent nativement le multi-threading
- C'est le cas notamment de ceux qui utilisent les *coroutines*.
- Une implémentation naïve donnerait trop de threads, et trop de "context switchs", pour être efficacement traités par le système.
- Ces threads utilisateurs légers, exécutions de coroutines, sont appelés des *fibres*

processus / thread système / thread utilisateur / fibre

Modèle M :N ("hybride")

- Tire les avantages des modèles 1 :1 et N :1
- Idéalement, N = nombre de coeurs

Exemples : GHC (Glasgow Haskell compiler), threads NetBSD...

Threads POSIX

Une norme POSIX pour créer des threads, et de les synchroniser

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

Compiler avec l'option `-pthread`

Intègre :

- Fonctions pour créer, attendre et tuer un thread
- Des mécanismes de synchronisation : mutex et variables de condition

L'implémentation Linux de pthread

- L'implémentation actuelle des threads POSIX sous Linux est NPTL (Native POSIX Threads Library). Elle respecte la norme POSIX, et rajoute quelques fonctions non POSIX.
- En interne, il s'agit de threads systèmes (préemptifs et 1 :1).
- NPTL utilise des appels système à `clone()` et `futex()`, et des signaux temps réels.

Threads POSIX : partage

Les pthreads d'un processus partagent :

- le PID, le PPID
- l'espace mémoire
- les descripteurs de fichiers ouverts
- les utilisateurs propriétaires
- les handlers des signaux
- le répertoire courant, le masque de fichiers...

Ne partagent pas :

- les identifiants des threads
- la pile
- le masque des signaux
- `errno` (exercice : comment cela est possible ?)

Créer un thread

Au départ, le processus est constitué d'un unique thread : celui qui exécute la fonction `main`
On peut créer d'autres threads, avec la fonction suivante :

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

- `thread` : l'adresse mémoire où sera copié l'identifiant du nouveau thread
- `attr` : des attributs (NULL = defaut)
- `start_routine` : la fonction d'entrée du thread
- `arg` : l'argument de la fonction

Créer un thread

Exemple :

```
void *fonction(void *a)
{
   ...
   return NULL;
}

...
pthread_t id;
phtread_create(&id,NULL,fonction,NULL)
...
```

Identifiant d'un pthread

- Un pthread n'a pas de PID propre (ils partagent tous le même PID, celui du processus contenant les threads)
- L'identifiant d'un pthread est un objet du type `pthread_t`.
- La norme ne dit pas ce qu'il y a dans `pthread_t` (objet opaque).
- `phtread_self()` renvoie le pthread_t du thread courant.
- `phtread_equal(pthread_t a,pthread_t b)` permet de comparer deux identifiants

Argument et retour d'un pthread

- Il est possible de passer un argument à la fonction qu'on appelle : un pointeur, qu'on peut faire pointer vers la structure de son choix
- Quand start_routine termine, le thread se termine.
- Un thread peut également terminer avec pthread_exit()

- Le thread `main` est spécial, sa terminaison termine le processus, même si d'autres threads sont encore en exécution.
- exit() dans n'importe quel thread termine le processus (i.e. tous les threads)

Fin et attente d'un thread

Similairement à un processus, un thread renvoie un code retour : un pointeur.

Similairement à un fils qui devient zombi, le code retour du thread est gardée en mémoire jusqu'à ce qu'un autre thread le "rejoigne"

Il n'y a pas de notion de père/fils :

- n'importe quel thread peut joindre n'importe quel thread
- si plusieurs attentes du même thread : comportement indéfini

Fin et attente d'un thread

```
int pthread_join(pthread_t thread, void **
    retval);
```

`retval` : un pointeur sur une variable (contenant un pointeur), où le code retour sera copié.

- NULL : ignoré
- thread "annulé" : `PTHREAD_CANCELED`

Il existe des versions non bloquantes dans NPTL (non POSIX !) : `pthread_tryjoin_np`, `pthread_timedjoin_np`

Détacher un thread

Si on ne veut pas avoir à gérer la fin d'un thread, on peut le "détacher".

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

- La structure sera détruite à la terminaison du thread
- Il ne sera pas possible de retrouver son pointeur retour avec `pthread_join`

Arrêter un thread

```
int pthread_exit(void *retval);
```

Arrête (termine) le thread courant.
`retval` sera la valeur retour (récupérée par `pthread_join`)

Attention : dans beaucoup de cas, on ne peut pas simplement terminer un thread sans risquer des fuites mémoires, ou des interblocages.

Il faut faire attention :

- aux fuites mémoires (mémoire dynamique allouée par le thread)
- aux sections critiques (par exemple, si le thread bloque un mutex)

Annuler un thread

```
int pthread_cancel(pthread_t thread);
```

Permet d'"annuler" (de terminer) un thread

Attention : Comme dans le cas de `pthread_exit`, on ne peut pas simplement terminer un thread sans risquer des fuites mémoires, ou des interblocages.

Il ne s'agit pas de "tuer" un thread : le thread doit préparer son annulation.

Annuler un thread

Pour terminer un thread "proprement", on peut rajouter des "handlers" qui seront exécutés à la terminaison/annulation du thread

```
void pthread_cleanup_push(void (*routine)(void
    *), void *arg);
void pthread_cleanup_pop(int execute);
```

- push : rajoute dans une pile une nouvelle fonction à executer en cas d'annulation
- pop : enlève le dernier élement de la pile (et l'execute si `execute` n'est pas 0)

Annuler un thread

Le thread peut (doit) aussi dire quand il peut être annulé

```
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```

- `pthread_setcancelstate` (des)active la possibilité d'annulation du thread courant
- `pthread_setcanceltype` spécifie si l'annulation se fait immédiatement ou à un point d'annulation
- `pthread_testcancel` spécifie un point d'annulation

Attention : beaucoup de fonctions système sont également des points d'annulation...

En pratique : évitez d'annuler les threads avec `pthread_cancel`

Attributs

`pthread_attr_t` : objet (opaque) spécifiant les attributs d'un thread.

- `pthread_attr_init` / `pthread_attr_destroy` : initialise (détruit) un attribut
- `pthread_attr_setstacksize` (`pthread_attr_setstackaddr`) permet de spécifier la taille (l'emplacement) de la pile
- `pthread_attr_setdetachstate` : spécifie l'état détaché
- Il est également possible de spécifier des paramètres d'ordonnancement

Autres choses de pthread

Des parties importantes de pthread seront vus par la suite :

- les *mutex*
- variables de condition

Ces mécanismes permettent de synchroniser les threads lors d'accès concurrents.

Comportement avec les signaux

- Les handlers des signaux sont partagés entre tous les threads.
- Mais chaque thread a son propre masque de signaux !
- On peut modifier le masque d'un thread via `pthread_sigmask` (mêmes arguments que `sigprocmask`).
- On peut envoyer un signal à un thread spécifique via `pthread_kill(pthread_t id,int sig)`
- On peut attendre un signal avec `sigwait`

Comportement avec `exec` et `fork`

- Comportement avec `exec` :
  Si un thread fait un appel à `exec` (qui n'échoue pas), tous les autres threads sont tués.
- Comportement avec `fork` :
  Seul le thread appelant fork est dupliqué.
  Problème : comme les autres threads n'existent plus dans le fils, il se peut qu'un mutex ne soit jamais libéré.
  Une solution est d'utiliser `pthread_atfork` qui rajoute des handlers en cas de `fork`.

Le début des problèmes...

- Les threads partagent leur mémoire. Y compris le segment des données des librairies.
- Les (fonctions des) librairies doivent donc être prévues pour un usage multi-thread
- Lors de la communication par mémoire partagée, il faut aussi faire attention à ce que deux threads ne travaillent pas sur la même zone mémoire en même temps
- Sinon : bug, plantage, exploitation, heisenbug...

Libraires "MT-safe"

- Quand on utilise une librairie (ou une fonction d'une librairie) dans un programme multi-threadé, il faut vérifier si elle a été prévue pour une utilisation multi-threadée (MT-safe)
- Certaines fonctions de la libc sont MT-safe, d'autres non
- Il faut voir le manuel !

Ré-entrance :

- Une fonction est appelée "re-entrante" si elle peut être interrompue, et rappelée (de façon sûre).

Exemple : `strtok`

```
#include <string.h>
char *strtok(char *str, const char *delim);
```

`strtok` coupe `str` aux caractères présents dans `delim`

Si `str=NULL`, elle continue de découper la chaîne précédente

```
char w[]="2:5:6:1:2:6:3";
char *p=strtok(w,":");
while (p) {
  printf("%s ",p);
  p=strtok(NULL,":");
}
```

Sortie : 2 5 6 1 2 6 3

Problème (avec les programmes multithreads) : strtok garde en interne un pointeur sur la position courante dans le chaîne.

Exemple : `strtok`

Version ré-entrante : `strtok_r`

```
char *strtok_r(char *str, const char *delim,
    char **saveptr)
```

`strtok_r` ne garde pas un pointeur interne : il faut lui en fournir un

```
char *tmp;
char w[]="2:5:6:1:2:6:3";
char *p=strtok_r(w,":",&tmp);
while (p) {
 printf("%s ",p);
 p=strtok_r(NULL,":",&tmp);
}
```

Problème d'optimisation du compilateur

```
int i;

void *thread(void *a) {
   sleep(1);
   i=1;
}
...
pthread_create(id,NULL,thread,NULL);
i=0;
while(i==0) { usleep(1000); }
...
```

Avec trop d'optimisations, le compilateur peut considérer que `i` reste à 0, car jamais affectée à autre chose que 0 dans main.

Modificateur du C : `volatile`. Indique au compilateur qu'une variable peut changer entre ses différents accès.

Problèmes de synchronisation

Un thread peut être arrêté n'importe quand pour laisser sa place à un autre thread :

- y compris au milieu d'une ligne
- y compris au milieu d'une instruction basique en C (ex : `i++`)

Problème : si un thread A travaille sur une zone mémoire M, et est arrêté alors que M est inconsistante, le thread B trouvera M en état inconsistant.

- Comportement imprévisible !

Problèmes de synchronisation

```
long long z=0;

void* th(void *r) {
  for(long long a=0;a<1000000;a++)
    z++;
}

int main() {
  pthread_t id1,id2;
  pthread_create(&id1,NULL,th,NULL);
  pthread_create(&id2,NULL,th,NULL);
  pthread_join(id1,NULL);
  pthread_join(id2,NULL);
  printf("%Ld\n",z);
}
```

sortie : 948249

Atomicité

Code assembleur de z++ :

```
movq    z(%rip), %rax
addq    $1, %rax
movq    %rax, z(%rip)
```

On aimerait que ces 3 instructions ne puissent être interrompues.

Instruction(s) atomique : instruction(s) ne pouvant être interrompues.
Mais : il n'est pas possible de bloquer les interruptions dans le mode utilisateur.

Pour rendre atomique (vis à vis des autres threads) une accès sur une zone mémoire : mécanisme d'exclusion mutuelle (mutex)
C'est le sujet du prochain cours !

Threads (2) Synchronisation des processus concurrents : mutex

Introduction

Les threads partagent leur mémoire.

Le partage de mémoire est généralement voulu et avantageux :

- cela évite de gaspiller de la mémoire
- c'est un mécanisme de communication inter-thread (et inter-processus) très rapide

L'important est de bien savoir gérer l'accès concurrent à la mémoire

(Rappel : il faut faire attention aux fonctions/librairies non réentrantes, ou non "MT-safe")

Un exemple pour commencer...

Un thread peut être arrêté n'importe quand pour laisser sa place à un autre thread :

- y compris au milieu d'une ligne
- y compris au milieu d'une instruction basique en C (ex : `i++`)

Problème : si un thread A travaille sur une zone mémoire M, et est arrêté alors que M est inconsistante, le thread B trouvera M en état inconsistant.

- Comportement imprévisible ! (bug, plantage, exploitation, mauvaise note)

Un exemple pour commencer...

```
long long z=0;

void* th(void *r) {
  for(long long a=0;a<1000000;a++)
    z++;
}

int main() {
  pthread_t id1,id2;
  pthread_create(&id1,NULL,th,NULL);
  pthread_create(&id2,NULL,th,NULL);
  pthread_join(id1,NULL);
  pthread_join(id2,NULL);
  printf("%Ld\n",z);
}
```

sortie : 1020102 ou 1271305 ou 948249...

Atomicité

Code assembleur de z++ :

```
movq    z(%rip), %rax
addq    $1, %rax
movq    %rax, z(%rip)
```

On aimerait que ces 3 instructions ne puissent être interrompues.

Instruction(s) atomique : instruction(s) ne pouvant être interrompues.

Pour que l'exemple précédent soit correct, il faudrait rentre l'instruction z++ atomique.

Atomicité d'une instruction

L'atomicité peut se faire au niveau du processeur.

Il faut distinguer 2 choses :

- Atomicité vis à vis d'une interruption
  Une instruction processeur est atomique vis à vis d'une interruption.
- Atomicité vis à vis des autres coeurs
  Dans les ordinateurs multiprocesseurs et/ou multicoeurs, du fait de la pipeline, une variable peut changer entre sa lecture et son écriture. Elle n'est donc pas (par défaut) atomique vis à vis des autres coeurs .
  Sur x86 : on peut rendre atomique une instruction machine avec le préfixe `lock`.

Atomicité d'une instruction : exemple

Pour que le programme précédent fonctionne comme souhaité

- On peut incrémenter `z` avec l'instruction assembleur `incq`
  $\Rightarrow$ l'opération sera atomique vis a vis des interruptions.
  $\Rightarrow$ le programme fonctionnera correctement si la machine a un unique coeur.
- Pour que le programme soit "correct" dans le cas général :
  Il faut rendre atomique l'instruction : `lock incq`
  $\Rightarrow$ le programme fonctionne comme souhaité.

  Problème : Du fait que la pipeline ne sert presque plus, c'est beaucoup plus lent...

Atomicité d'un ensemble d'instructions

En pratique, les opérations sur une zone mémoire prennent généralement plusieurs instructions

Problèmes :

- Il n'est pas possible (ni raisonnable) de bloquer les interruptions / les autres coeurs dans le mode utilisateur.
- Pourquoi ? Un processus malveillant/planté/bogué pourrait bloquer tous les autres processus...
- Il n'est pas possible d'utiliser un mécanisme du type `lock` sur un ensemble d'instructions

Atomicité d'un ensemble d'instructions

De toutes façons : on ne veut pas l'atomicité d'un ensemble d'instructions...
Cela bloquerait tous les coeurs pour accéder à une zone mémoire, alors que les autres ne travaillent pas forcement sur cette zone...

La bonne solution n'est pas d'avoir des sections atomiques, mais des sections où on a l'exclusivité sur une partie de la mémoire.

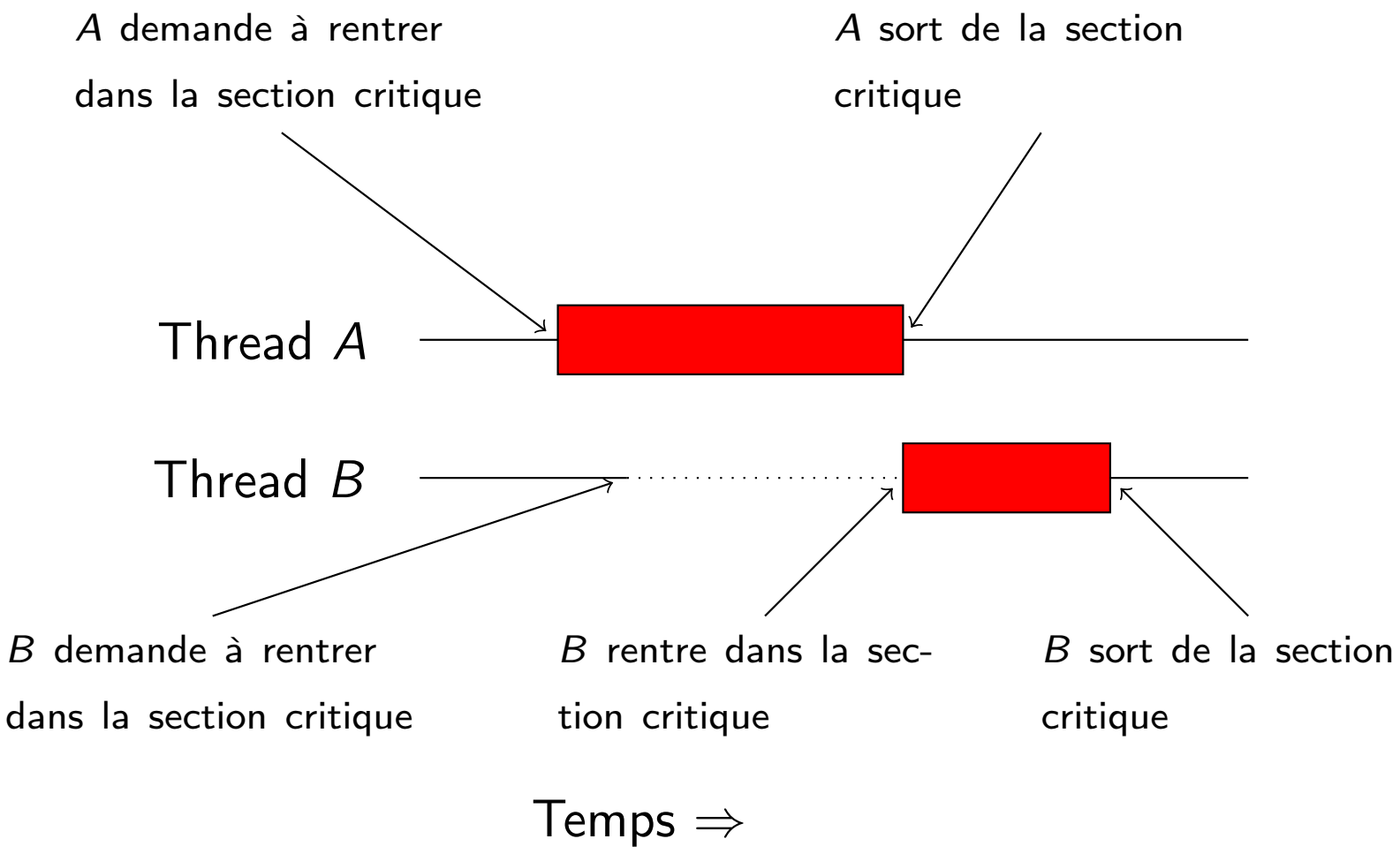
Cela s'appelle une section critique .

Sections critiques

Une section critique est une section du code où il n'y a au maximum qu'un thread à la fois

Si un thread B veut rentrer dans une section critique, et qu'un autre thread A est déjà dans la section critique, on doit faire attendre le thread B jusqu'à ce que le thread A sorte de la section critique.

Sections critiques



Sections critiques

Avantage :

- on ne bloque pas les autres threads qui ne travaillent pas sur la section critique
- il peut y avoir plusieurs sections critiques différentes, qui n'interfèrent pas entre elles.

Sections critiques

Il faut un mécanisme pour entrer et sortir des sections critiques.

Ce qu'on attend d'un mécanisme de gestion des sections critiques :

- l'exclusion mutuelle : deux threads ne sont pas en même temps dans la section critique
- la progression : le processus continue de progresser dans tous les cas possibles d'exécution
- l'attente bornée : un thread qui demande à rentrer dans une section critique ne va pas attendre indéfiniment

Sections critiques

Ce n'est pas un problème trivial. Plusieurs solutions fausses ont été publiées

Les problèmes en cas de mauvaise gestion des sections critiques :

- Si l'exclusion mutuelle est pas respectée :
  Situation de compétition (race condition) : deux threads sont dans une section critique en même temps : non déterminisme (bug, plantage, exploitation...)
- Si la progression n'est pas respectée :
  Interblocage (deadlock) : le processus est bloqué.
- Si l'attente bornée n'est pas respectée :
  Famine (starvation) : un thread ne voit jamais sa demande aboutir

Exemple simple : 2 threads

Solution 1 (?)

```
int intA=0,intB=0;
```

Thread A :

```
while(1) {
  while(intB) /*wait*/;
  intA=1;
  //sect. critique
  intA=0;
  //sect. normale
}
```

Thread B :

```
while(1) {
  while(intA) /*wait*/;
  intB=1;
  //sect. critique
  intB=0;
  //sect. normale
}
```

Correct ?

Non ! Les 2 threads peuvent être dans la section critique en même temps (situation de compétition)

Exemple simple : 2 threads

Solution 2 (?)

```
int intA=0,intB=0;
```

Thread A :

```
while(1) {
  intA=1;
  while(intB) /*wait*/;
  //sect. critique
  intA=0;
  //sect. normale
}
```

Thread B :

```
while(1) {
  intB=1;
  while(intA) /*wait*/;
  //sect. critique
  intB=0;
  //sect. normale
}
```

Correct ?

Non ! Les 2 threads peuvent se bloquer mutuellement (interblocage)

Exemple simple : 2 threads

Solution 3 (?)

```
int rnd=0;
```

Thread A :

```
while(1) {
  while(rnd!=0) /*wait*/;
  //sect. critique
  rnd=1;
  //sect. normale
}
```

Thread B :

```
while(1) {
  while(rnd!=1) /*wait*/;
  //sect. critique
  rnd=0;
  //sect. normale
}
```

Correct ?
Non ! Quand le thread A termine, B est bloqué indéfiniment (famine)

Exemple simple : 2 threads

Solution 4 (?)

```
int rnd=0;
int intA=0,intB=0;
```

Thread A :

```
while(1) {
 intA=1;
 rnd=1;
 while(intB && rnd==1)
     /*wait*/;
 //sect. critique
 intA=0;
 //sect. normale
}
```

Thread B :

```
while(1) {
 intB=1;
 rnd=0;
 while(intA && rnd==0)
     /*wait*/;
 //sect. critique
 intB=0;
 //sect. normale
}
```

Correct ?

Oui ! (Solution de Peterson)

Attente active et passive

L'attente avant d'entrer dans une section critique peut être :

- active (*spinlock*) : le thread continue de tourner jusqu'à ce qu'il a le droit d'entrer dans la section critique
  Dans l'exemple précédent (Peterson), l'attente est active
- passive : le thread est mis en pause jusqu'à ce qu'il a la possibilité de rentrer dans la section critique
  Dans ce cas, il faut interférer avec l'ordonnanceur
  Avantage : on laisse le temps CPU aux autres threads qui peuvent faire des choses plus constructives

En général, on préfère l'attente passive. (Mais dans certains cas très particuliers, l'attente active peut être plus avantageuse.)

Les primitives

En général, on ne reprogramme pas soi même les tests d'entrée dans une section critique.

- C'est fastidieux
- le risque d'erreur est très important
- on ne tire pas parti des possibilités de l'OS.

On passe par des primitives (du langage, de librairies et/ou du système) : des verrous .

Les primitives

Il existe différents types de verrous :

- sémaphores (POSIX 1.b)
- mutex (pthreads)
- rwlocks (pthreads)
- barrières (pthreads)
- variables de condition / moniteurs (pthreads)

Attention ! Les verrous proposés par les langages/systèmes ne sont que des primitives (pour simplifier la vie).

Une mauvaise utilisation peut toujours mener à des problèmes : situation de competition, interblocage ou famine...

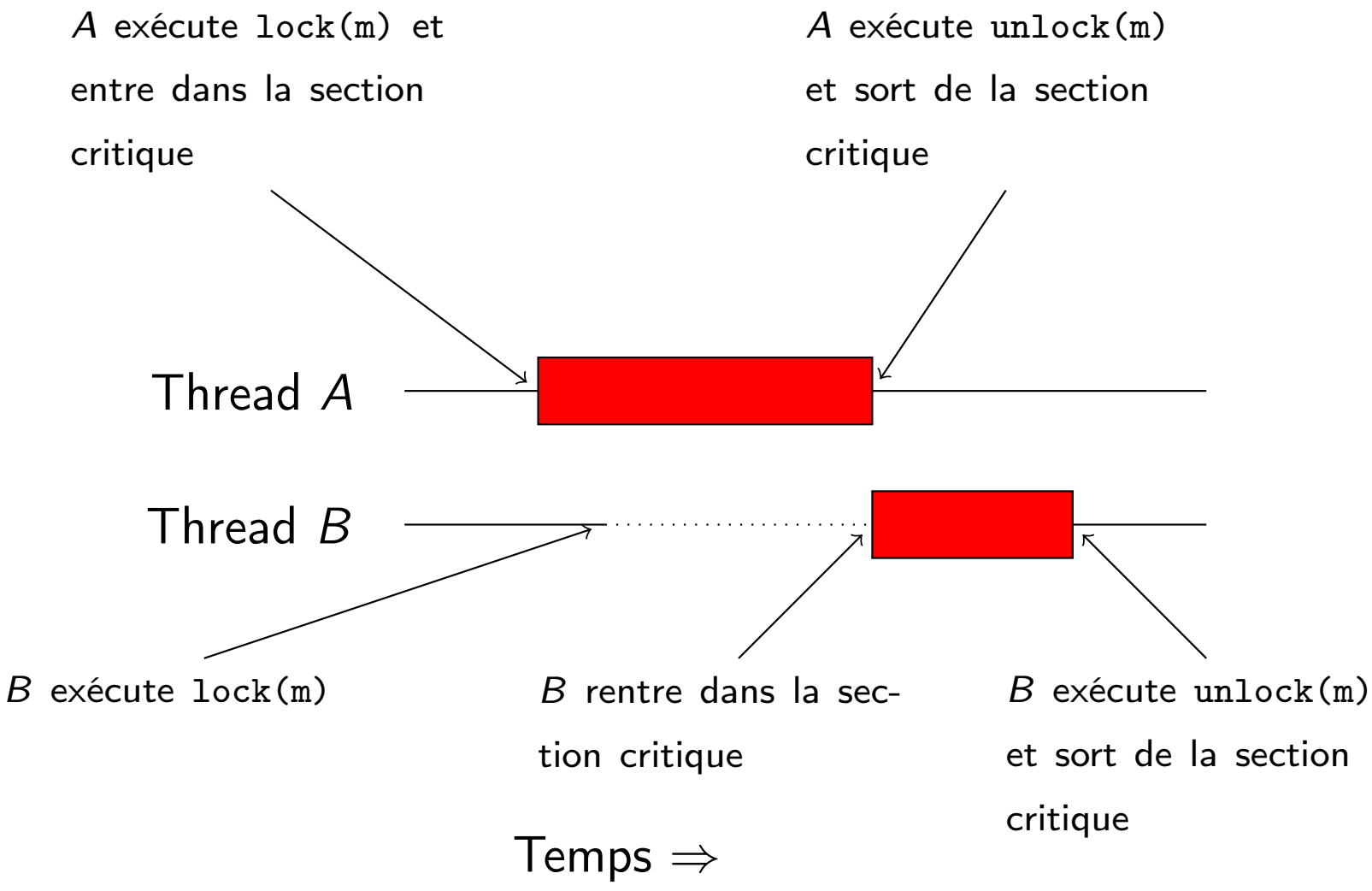
Verrou le plus simple : le mutex

Mutex : assure qu'au plus un thread est dans la section critique à un moment donné.

Pseudo-code :

```
mutex m;

//section non critique
lock(m);
//section critique;
unlock(m);
//section non critique
..
```

Verrou le plus simple : le mutex



Les mutex de pthreads

```c
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_init(pthread_mutex_t *mutex, const
    pthread_mutexattr_t *mutexattr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Exemple :

```c
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&mutex);
//section critique
pthread_mutex_unlock(&mutex);
```

Les mutex de pthreads

Note :

- Deux façons d'initialiser un mutex :

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

ou

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex,NULL);
```

- `mutexattr` permet d'initialiser un mutex avec d'autres attributs que ceux par défaut (récursif, partagé...).
  NULL = défaut
- `trylock` essaye de bloquer le mutex. Si il échoue, il renvoie directement la main avec une erreur (retour $\neq 0$)

Implémentation d'un mutex (?)

Ici, un mutex est un entier.

```
int a=1;

lock(int &a) {
  while(a==0) /*wait*/;
  a=0;
}

unlock(int &a) {
  a=1;
}
```

Problème : pour que ce soit correct, le test et l'affectation doivent se faire atomiquement

Implémentation possible d'un mutex

```
int a=0;

lock(int &a) {
 int tmp=0;
 while(1) {
  xchg(a,tmp);
  if(tmp==1) break;
 }
}

unlock(int &a) {
 a=1;
}
```

`xchg(a,b)` échange (atomiquement) a et b. (instruction x86)

- Problème : attente active et possible famine

Pseudo-implémentation idéale d'un mutex

```
int libre=1;
queue liste_attente;

lock() { //atomiquement
 while(1) {
  if(libre) {
   libre=0;
   return;
  }
  liste_attente.push(thread_courant());
  wait();
 }
}

unlock() { //atomiquement
 if(liste_attente.non_vide())
   signal(liste_attente.pop());
 else
   libre=1;
}
```

Où sont implémentés les verrous ?

Problème des verrous en mode utilisateur :

- pour éviter l'attente active, il faut jouer avec l'ordonnanceur
- pour éviter les famines, il faut une file d'attente

Ces taches sont souvent laissées aux OS

Problème des verrous en mode noyau : les appels systèmes sont très lents !

Bon compromis : combiner les deux

Implémentation dans NPTL

- `lock()` sur un verrou libre : opération atomique
- `lock()` sur un verrou non libre : opération atomique + appel système (`futex()`) qui met le thread en pause, et rajoute à une liste d'attente
- `unlock()` : opération atomique +
  (si la liste d'attente est non vide) appel système à `futex` pour libérer le thread suivant.

- coté utilisateur : un boléen (libre ou non) et le nombre de threads en attente
- coté système : une liste d'attente

Mutex récursif

NPTL (et d'autres implémetations) introduisent d'autres possibilités (non POSIX, "non portables"). Par exemple :

- mutex récursif : l'action de bloquer un mutex déjà bloqué par le thread courant ne bloque pas. Exemple :

```
foo(int i)
{
   lock(mutex);
   // section critique
   if(i>0) foo(i-1);
   // section critique
   unlock(mutex);
}
```

Read-write locks

On pourrait permettre à plusieurs threads qui ne modifient pas la mémoire, de travailler (en lecture seule) sur une zone mémoire.

Une solution : read-write locks (rwlocks)

Deux types de sections critiques

- les sections critiques en lecture seule (celles des lecteurs)
- les sections critiques en lecture/écriture (celles des écrivains)

Read-write locks

Garantie des rwlocks :

- si un thread est dans une section critique en lecture/écriture, il n'y a aucun autre thread dans une section critique (ni en lecture seule, ni en lecture/écriture)
- (si aucun thread est dans une section critique en lecture/écriture, il n'y a pas de limite sur le nombre de threads dans une section critique en lecture seule)

Attention : on peut facilement arriver à des famines

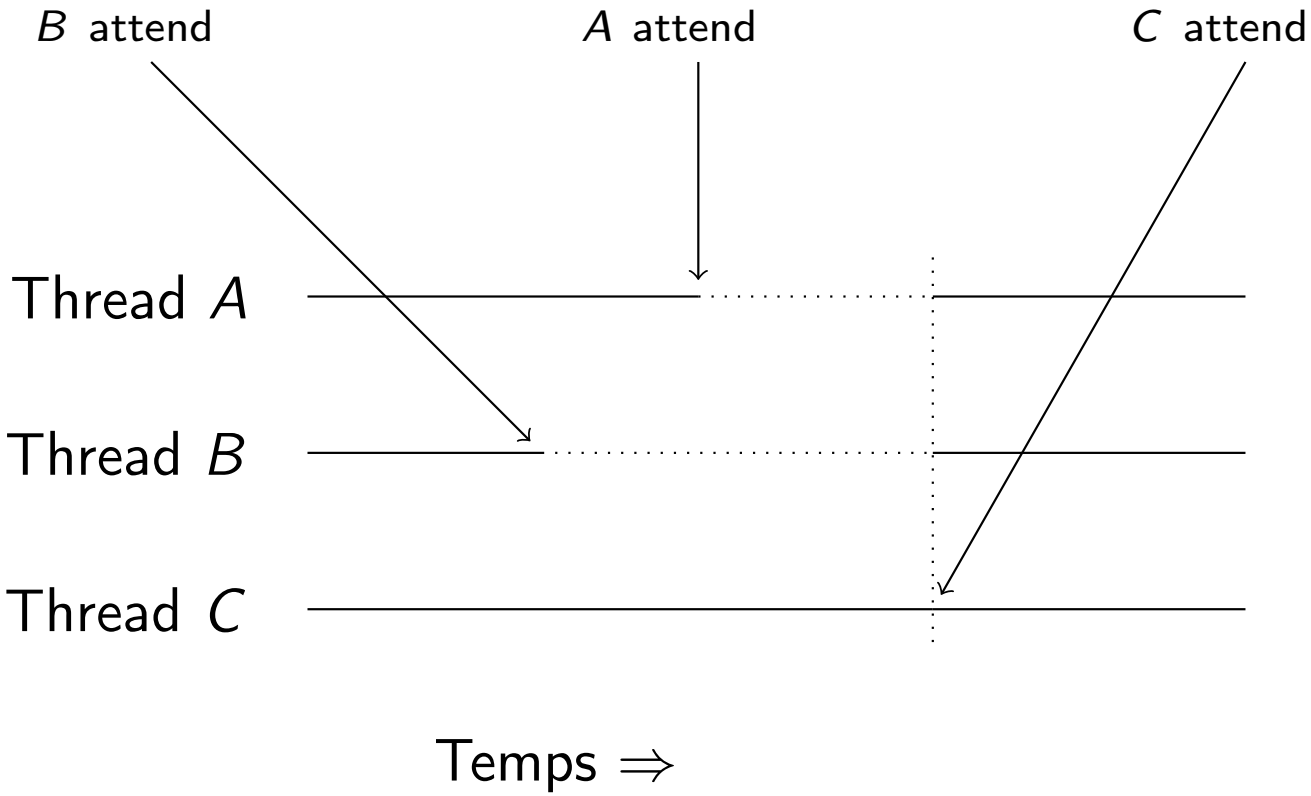
- préférer les lecteurs : il peut y avoir famine des écrivains
- préférer les écrivains : il peut y avoir famine des lecteurs

Les rwlocks de pthreads

```
#include <pthread.h>

pthread_rwlock_t lock = PTHREAD_RWLOCK_INITIALIZER;

int pthread_rwlock_init(pthread_rwlock_t * restrict
    lock, const pthread_rwlockattr_t * restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t *lock);

int pthread_rwlock_rdlock(pthread_rwlock_t *lock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *lock);
int pthread_rwlock_timedrdlock(pthread_rwlock_t *
    restrict lock, const struct timespec * restrict
    abstime);

int pthread_rwlock_wrlock(pthread_rwlock_t *lock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *lock);
int pthread_rwlock_timedwrlock(pthread_rwlock_t *
    restrict lock, const struct timespec * restrict
    abstime);

int pthread_rwlock_unlock(pthread_rwlock_t *lock);
```

Les barrières

Les barrières permettent de synchroniser les threads



Les barrières POSIX

```
int pthread_barrier_init(pthread_barrier_t *restrict
    barrier, const pthread_barrierattr_t *restrict attr
    , unsigned count);
int pthread_barrier_destroy(pthread_barrier_t *barrier)
    ;
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

- `count` : nombre de threads qui doivent attendre à la barrière

Exemple :

```
pthread_barrier_t barrier;
pthread_barrier_init(&barrier,NULL,3);
```

Puis dans chaque thread (A, B et C) :

```
// section non synchronisee
pthread_barrier_wait(&barrier);
// section synchronisee
```

Problèmes : vitesse

Le but des programmes parallèles est de gagner en vitesse.

En utilisant les mécanismes précédents (mutex...) un processus peut perdre du temps :

- dans les attentes qu'un verrou se libère (attente active ou passive)
- dans les instructions atomiques (plus lentes à cause de problèmes de cache)
- dans les appels systèmes relatifs aux (dé)blocage de verrous

Problèmes : vitesse

Solutions :

- limiter la taille des sections critiques
- séparer les zones mémoires partagés
  (idéalement : 1 zone mémoire = 1 verrou)
- mais sans faire trop d'entrées/sorties de sections critiques

Un problème d'échelle (de granularité) peut parfois se poser

- trouver le bon compromis

Problèmes : vitesse

Exemple (bidon)

$$\sigma(n) = \sum_{1 \leq i \leq n : i | n} i$$

```
long long n,s=0;
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;

void* th(void *r) {
  for(long long i=(long long)r;i<=n;i+=2)
    if(i%n==0) {
      pthread_mutex_lock(&m);
      s+=i;
      pthread_mutex_unlock(&m);
    }
  return NULL;
}
```

Problèmes : vitesse

Mieux :

```
long long n,s=0;
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;

void* th(void *r) {
  long long tmp=0;
  for(long long i=(long long)r;i<=n;i+=2)
      if(i%n==0)
        tmp+=i;
  pthread_mutex_lock(&m);
  s+=tmp;
  pthread_mutex_unlock(&m);
  return NULL;
}
```

- plus d'opérations de calcul (ici, 2 additions en plus)
- mais beaucoup moins de sections critiques

Gros problèmes

En cas de mauvaise gestion des sections critiques :

- situation de compétition : bugs, plantages, morts (Therac-25)
- interblocage

Si on protège chaque section critique par un verrou adéquat, l'exclusion mutuelle devrait être respectée. Le problème sera généralement l'interblocage

Suite :

- Les 5 philosophes...
- Sémaphores
- Variables de condition (Moniteurs)
- Gérer les interblocages
- Concurrence en C++11

Threads (3) : Sémaphores et variables de condition

Introduction

- Les threads partagent leur mémoire
- Il faut protéger les accès mémoires concurrents
- Les mutex sont des verrous qui permettent de délimiter des sections critiques
- Mais des fois, les mutex ne suffisent pas...

On va voir :

- Les sémaphores
- Les variables de conditions

Problème classique : 5 philosophes

- 5 philosophes sont autour d'une table ronde
- Il y a une baguette entre chaque philosophe (i.e. : une baguette pour deux philosophes)
- Un plat de sushis pour tout le monde est au centre
- La vie d'un philosophe se résume à deux actions : penser et manger
- Pour manger, il doit prendre les 2 baguettes (à sa gauche et à sa droite), puis il peut commencer à manger
- Quand il a fini, il repose les baguettes et peut commencer à penser
- Chaque baguette : une ressource (un mutex)

But :

- Tout le monde doit pouvoir manger (pas de famine)
- Limiter les attentes

Problème classique : 5 philosophes

```
mutex baguette[5];

void *philosophe(void *a)
{
  int i=(long long) a;
  while(1) {
    printf("%d␣pense\n",i);
    lock(&baguette[i]);
    lock(&baguette[(i+1)%5]);
    printf("%d␣mange\n",i);
    unlock(&baguette[i]);
    unlock(&baguette[(i+1)%5]);
  }
}
```

Correct ?

Non : interblocage. Si tout le monde a la baguette à gauche, tout le monde est bloqué

5 philosophes : solution ?

```
mutex baguette[5],general;

void *philosophe(void *a)
{
  int i=(long long) a;
  while(1) {
    printf("%d␣pense\n",i);
    lock(&general);
    lock(&baguette[i]);
    lock(&baguette[(i+1)%5]);
    printf("%d␣mange\n",i);
    unlock(&baguette[i]);
    unlock(&baguette[(i+1)%5]);
    unlock(&general);
  }
}
```

Correct... mais qu'un seul philosophe mange à la fois. Les mutex baguette[i] ne servent à rien $\rightarrow$ une seule section critique

5 philosophes : solution ?

```
mutex baguette[5],general;

void *philosophe(void *a)
{
  int i=(long long) a;
  while(1) {
    printf("%d␣pense\n",i);
    lock(&general);
    lock(&baguette[i]);
    lock(&baguette[(i+1)%5]);
    unlock(&general);
    printf("%d␣mange\n",i);
    unlock(&baguette[i]);
    unlock(&baguette[(i+1)%5]);
  }
}
```

Mieux... mais un philosophe doit des fois attendre inutilement pour manger.

5 philosophes : solution ?

```
void *philosophe(void *a)
{
  int i=(long long) a;
  while(1) {
    printf("%d pense\n",i);
    while(1) {
      lock(&general);
      if(baguette[i]==1 && baguette[(i+1)%5]==1) {
       baguette[i]=baguette[(i+1)%5]=0;
       unlock(&general);
       break;
      }
      unlock(&general);
    }
    printf("%d mange\n",i);
    baguette[i]=baguette[(i+1)%5]=1;
  }
}
```

Attente active et famine

Parenthèse : Famine ?

On peut distinguer deux types de famines :

- la famine "avérée" : un thread est indéfiniment bloqué (quelle que soit le déroulement de la suite)
- la famine "probabiliste" : il y a une probabilité non nulle qu'un thread reste bloqué indéfiniment longtemps.

Si on s'autorise la famine "avérée", il existe une solution simple aux 5 philosophes sans interblocage, ni situation de compétition :

- On désigne un philosophe qui n'a pas le droit de prendre de baguettes (donc ni de manger et penser)

Parenthèse : Famine ?

Si on s'autorise la famine "probabiliste" (mais pas "avérée"), la solution précédente est bonne (modulo le fait qu'elle soit en attente active)

Note :

- La famine est, des fois, pas très grave (e.g. si les threads font le même travail).
- Beaucoup considèrent que la famine probabiliste n'est pas un vrai problème.

Ordre d'importance :
famine "probabiliste" / famine "avérée" / interblocage / situation de compétition

5 philosophes : solution ?

Plusieurs (idées) de solutions

- Contre l'attente active : rajouter des temporisations aléatoires
  - bricolage : pas optimal (et toujours possible famine)
- Prendre une baguette (lock), essayer de prendre l'autre (trylock). Si la deuxième n'est pas disponible, reposer la première (unlock) et recommencer.
  - attente active (et possible famine)
- Prendre une baguette (lock), essayer de prendre l'autre (trylock). Si la deuxième n'est pas disponible, reposer la première (unlock) et recommencer dans le sens contraire .
  - (possible famine)

5 philosophes : solution ?

Note : si les philosophes sont sur une table linéaire (5 philosophes, 6 baguettes), la solution triviale marche.

Ce qui pose problème : les cycles

Solution : casser les cycles

5 philosophes : solution ?

```
void *philosophe(void *a)
{
  int i=(long long) a;
  while(1) {
    printf("%d␣pense\n",i);
    if(i==0) {
     lock(&baguette[(i+1)%5]);
     lock(&baguette[i]);
    } else {
     lock(&baguette[i]);
     lock(&baguette[(i+1)%5]);
    }
    printf("%d␣mange\n",i);
    unlock(&baguette[i]);
    unlock(&baguette[(i+1)%5]);
  }
}
```

Correct. Mais pas très joli, et l'asymétrie introduit des biais (des philosophes attendent plus que d'autres en moyenne)

5 philosophes : Solution de Dijkstra

On limite à 4 le nombre de philosophes qui peuvent rentrer dans la phase "prendre des baguettes"

- au plus 4 arcs dans le graphe $\Rightarrow$ pas de cycle.

Principe des sémaphores .

(Exercice : est il possible que le 5ème philosophe attende inutilement ?)

(Exercice : combien, au minimum, faut-il autoriser de philosophes dans la phase "prendre des baguettes" pour qu'il n'y ait pas d'attente inutile ?)

Les sémaphores

Un autre type de verrou est le sémaphore

- Introduit par Dijkstra ($\sim$ 1963)
- Premier verrou à apparaître dans un vrai système
- Plus général qu'un mutex
- (Mais il a plutôt un intérêt historique et, ici, didactique)

- 3 opérations :
  - init(N) : initialise le sémaphore à N "ressources"
  - P() (ou wait(), ou down()) : demande une ressource. Si il n'y a plus de ressource libre, attend jusqu'à ce qu'une ressource se libère
  - V() (ou signal(), ou post(), ou up()) : libère une ressource
- garantie : au plus N threads possèdent une "ressource"

Sémaphore : différence avec un mutex

- Un sémaphore avec N=1 simule (un peu près) un mutex
- (Un sémaphore avec N=1 est un sémaphore binaire)
- Mais : le mutex est un booléen, le sémaphore est un entier
  - unlock() ;unlock() ; n'aura pas le même comportement que V() ;V() ;
- Mais : un mutex doit être débloqué par le thread qui l'a bloqué.
  - On peut se servir d'un sémaphore comme d'un "signal" pour débloquer un autre thread

```
init(sem,0);
```

Thread A :

```
sleep(2);
// pas syncro
V(sem);
//synchro
```

Thread B :

```
sleep(1);
// pas syncro
P(sem);
//synchro
```

Sémaphores POSIX

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned
    int value);
int sem_destroy(sem_t *sem);

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct
    timespec *abs_timeout);

int sem_post(sem_t *sem);
```

- `pshared` = 0 : le sémaphore n'est pas partagé avec un autre processus (uniquement entre les threads de ce processus)
- `pshared` = 1 : le sémaphore est partagé. `sem` doit être dans une zone mémoire partagé entre les différents processus

5 philosophes avec sémaphore

```
mutex baguette[5];
semaphore sem;
init(sem,4);

void *philosophe(void *a) {
  int i=(long long) a;
  while(1) {
    printf("%d pense\n",i);
    wait(sem);
    lock(baguette[i]);
    lock(baguette[(i+1)%5]);
    post(sem);
    printf("%d mange\n",i);
    unlock(baguette[i]);
    unlock(baguette[(i+1)%5]);
  }
}
```

Parfait ! pas de famine (même "probabiliste"), pas d'attente inutile.

Sémaphore : Implémentation ?

La première solution (Dijkstra) utilisait des blocages d'interruptions. Cela n'est plus valable sur des machines multiprocesseurs, mais cela peut être simulé avec un mutex

```
init(int &s,int N) {
  lock(mutex);
  s=N;
  unlock(mutex);
}

V(int &s) {
  lock(mutex);
  s++;
  unlock(mutex);
}
```

```
P(int &s) {
  while(1) {
    lock(mutex);
    if(s>0) {
      s--;
      unlock(mutex);
      return;
    }
    unlock(mutex);
  }
}
```

- Attente active et famine

Sémaphore : Implémentation ?

Comment implémenter un sémaphore avec une attente passive ?

C'est les mêmes problèmes qu'on avait déjà vu avec les mutex :

- il faut une file (éviter la famine)
- il faut communiquer avec l'ordonnanceur (pour éviter l'attente active)

(Les sémaphores sont disponibles dans POSIX, mais imaginons que ce ne soit pas le cas.)

Comment peut on faire pour les reprogrammer correctement ?

On ne peut pas simuler un sémaphore avec des mutex seuls

Mais on peut le faire avec un mutex + une variable de condition

Variable de condition

Plus généralement, supposons qu'on veut qu'un thread bloque jusqu'à ce qu'une condition (qui peut être compliquée) soit vérifiée

```
//...
while(1) {
   lock();
   if(condition()==1) {
      // operations d'entree de section critique
      unlock();
      break;
   }
   unlock();
}
//section critique
lock();
// operations de sortie de section critique
unlock();
//...
```

On voudrait avoir le même comportement que le code ci-dessus, mais sans les famines et dans l'attente active. Comment faire ?

Variable de condition

Variable de condition : primitive qui permet de mettre en attente (dans une queue) un thread en débloquant (atomiquement) un mutex

```
mutex m;
cond c;

//...
lock(m);
while(condition()==0)
   wait(c,m);
// operations d'entree de section critique
unlock(m);
//section critique
lock(m);
// operations de sortie de section critique
signal(c);
unlock(m);
//...
```

Variable de condition

Une variable de condition fonctionne toujours de pair avec un mutex

3 primitives :

- wait(c,m) : (atomiquement) débloquer m, mettre le thread en pause, et le rajouter dans la queue de c
  - au moment de l'appel, m doit être bloqué !
- signal(c) : débloque le premier thread de la queue de c
- broadcast(c) : débloque tous les threads de la queue de c

signal ou broadcast ?

Si tous les threads en attente attendent sur la même condition : signal()

Si les threads ont des conditions différentes : broadcast()

Sinon : un thread est débloqué, mais si sa condition n'est pas validée, il va devoir re-entrer en sommeil. Si un autre thread a sa condition validée, il ne sera pas réveillé par défaut.

Variables de condition dans pthread

```
#include <pthread.h>

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_init(pthread_cond_t *cond,
    pthread_condattr_t *cond_attr);
int pthread_cond_destroy(pthread_cond_t *cond);

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

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);
```

5 philosophes avec mutex+cond

```
void *philosophe(void *a)
{
  int i=(long long) a;
  while(1) {
    printf("%d pense\n",i);

    pthread_mutex_lock(&mutex);
    while(baguette[i]==0 || baguette[(i+1)%5]==0)
      pthread_cond_wait(&cond,&mutex);
    baguette[i]=baguette[(i+1)%5]=0;
    pthread_mutex_unlock(&mutex);

    printf("%d mange\n",i);

    pthread_mutex_lock(&mutex);
    baguette[i]=baguette[(i+1)%5]=1;
    pthread_cond_broadcast(&cond);
    pthread_mutex_unlock(&mutex);
  }
}
```

Sémaphores avec mutex+cond

```
typedef struct {
  int n;
  pthread_mutex_t m;
  pthread_cond_t c;
} sem_t;

void sem_init(sem_t *s,
              int n)
{
  s->n=n;
  pthread_mutex_init(
        &s->m,NULL);
  pthread_cond_init(
        &s->c,NULL);
}
```

```
void sem_post(sem_t *s)
{
 pthread_mutex_lock(&s->m);
 s->n++;
 pthread_cond_signal(&s->c);
 pthread_mutex_unlock(&s->m);
}

void sem_wait(sem_t *s)
{
 pthread_mutex_lock(&s->m);
 while(s->n<=0)
  pthread_cond_wait(&s->c,
                    &s->m);
 s->n--;
 pthread_mutex_unlock(&s->m);
}
```

Pour finir...

- Moniteurs = mutex + variable de condition associée au mutex
- Généralise les autres primitives
- Par exemple, en C++11 : les seules primitives introduites dans la STD sont `<mutex>` et `<condition_variable>`

Interlude : Concurrence C++11

Le C++11 introduit plusieurs classes pour gérer les threads et la concurrence :

- threads
- mutex
- variables de condition

<thread>

- `std::thread` représente un thread (similaire à `pthread_t`)
- le thread est lancé à la construction :

```
void fct(int a, double b) {
 //...
}

std::thread th(fct,42,3.14);
```

- grâce à la "magie" des templates, pas besoin de jouer avec un argument `void*`

<thread>

- `std::thread` est remplaçable (mais pas copiable). `operator=` déplace le thread.

```
std::thread th[42];

for(int i=0;i<42;i++)
  th[i]=std::thread(foo,i);
```

- Fonctions membres : `join()`, `detach()`, `swap()`
- si on détruit un std : :thread alors qu'il correspond à un thread en exécution, non détaché : exception
- pas de code retour (voir <future>)

<mutex>

- `std::mutex` et `std::recursive_mutex`
- Fonctions membres : `lock()`, `try_lock`, `unlock`
- `std::lock_guard` est un conteneur pour un mutex (il le bloque à sa construction, et le débloque à sa destruction)

```
std::mutex m;
//..
{
 std::lock_guard<std::mutex> l(m);
 //section critique
}
// section normale
```

- particulièrement utile pour que le mutex soit débloqué automatiquement en cas d'exception

<mutex>

- `std::unique_lock` est un conteneur plus évolué.
- Il a notamment les mêmes fonctions membres qu'un mutex, ce qui permet de s'en servir comme un mutex (avec l'assurance qu'il sera débloqué en cas de destruction)

```
std::mutex m;
std::uniquelock<std::mutex> l(m, std::defer_lock);

l.lock();
//section critique
l.unlock();
```

`std::lock()`

- `std::lock(m1,m2,...)` permet de bloquer plusieurs mutex à la fois
- Algorithme :
  - bloque le premier mutex m_1,
  - puis essaye de bloquer les autres avec `try_lock()`.
  - Si un mutex m_i, $i > 1$ échoue, il débloque tous les autres : $m_1, \ldots m_{i-1}$,
  - puis recommence en commençant par m_i.
- Sans deadlock, et sans attente active.

5 philosophes en C++11

```
#include <thread>
#include <mutex>
std::mutex baguette[5];

void philosophe(int i) {
  while(1) {
    printf("%d␣pense\n",i);
    lock(baguette[i],baguette[(i+1)%5]);
    printf("%d␣mange\n",i);
    baguette[i].unlock();
    baguette[(i+1)%5].unlock();
  }
}

int main() {
  std::thread id[5];
  for(int i=0;i<4;i++)
    id[i]=std::thread(philosophe,i);
  id[0].join();
}
```

<condition_variable>

- `std::condition_variable` est une variable de condition
- constructeur sans argument

Fonctions membres :

- `wait(l)` (`l` doit être un `unique_lock<mutex>`)
- `notify_one()` = signal
- `notify_all()` = broadcast

<future>

Permet d'accéder au résultat de procédures asynchrones

```
int carre(int x) {
  for(long long i=0;i<1000000000;i++);
  return x*x;
}

int main()
{
  std::future<int> res=std::async(carre,42);
  printf("resultat=%d\n",res.get());
  return 0;
}
```

<future>

Ou bien avec des promesses :

```
void carre(int x,std::promise<int> pr) {
  for(long long i=0;i<1000000000;i++);
  pr.set_value(x*x);
}

int main()
{
  std::promise<int> pr;
  std::future<int> res=pr.get_future();
  std::thread th(carre,42,std::move(pr));
  printf("resultat=%d\n",res.get());
  th.join();
  return 0;
}
```

<atomic>

Permet de faire des opérations atomiques sur une variable

```
std::atomic<int> atint;

void th() {
  for(int i=0;i<10000000;i++)
    atint++;
}

int main() {
  atint.store(0);
  std::thread th1(th);
  std::thread th2(th);
  th1.join();
  th2.join();
  printf("%d\n",atint.load());
  return 0;
}
```

Variables `thread_local`

En C/C++ : il y a deux types de variables :

- les automatiques (ou locales) : dans la pile (défaut, mot clef `auto`)
- les statiques (ou globales) : dans le segment de données (mot clef `static`)

Le C++11 rajoute le type `thread_local`

- la variable sera locale au thread

Gestion des interblocages

Conditions pour avoir un interblocage

Un interblocage arrive quand les 4 conditions suivantes sont vérifiées en même temps : [Coffman 1971]

- exclusion mutuelle : la/les ressource(s) n'est pas partageable
- "hold and wait" : le(s) thread(s) a déjà bloqué une ressource, et en demande une autre
- non-préemption : c'est le thread qui libère par lui même les ressources
- attente circulaire : il existe une chaîne de processus $P_1 \dots P_k$ telle que chaque processus P_i bloque une ressource R_i et chaque processus P_i demande la ressource $R_{(i+1)\%k}$.

Exemple : les 5 philosophes ont chacun la fourchette de gauche, et attendent la fourchette de droite.

Prévenir et éviter les interblocages

Briser une des 4 conditions :

- Enlever l'exclusion mutuelle : par exemple
  - remplacer par des opérations atomiques.
  - algorithmes "sans blocages" : utilisation d'opérations atomiques "lecture-modification-ecriture"
- Enlever "hold and wait" : Exemple :
  - s'imposer de demander au plus 1 ressource à la fois.
  - bloquer plusieurs mutex atomiquement en même temps
  - si on demande plusieurs mutex, on fait attention à ne pas bloquer si on tient déjà un mutex
- Préemption : Difficile à faire... faudrait que cela soit prévu par les primitives et le programme

Prévenir et éviter les interblocages

Contre l'attente circulaire : ne pas créer de cycles dans le graphe

- Par exemple : Solution de Dijkstra pour les 5 philosophes
- Solution générale possible : mettre un ordre total sur toutes les ressources, et demander les ressources dans l'ordre
- Le graphe de dépendance sera toujours un sous graphe d'un graphe acyclique
- Exemple : 5 philosophes asymétriques : un des philosophes prend les fourchettes dans l'autre sens.
- Une solution simple dans le cas général : adresse mémoire du mutex

Prévenir et éviter les interblocages

Si on a (en plus) les informations de quelles ressources, ou combinaisons de ressources, peuvent être demandés, il est possible de prévenir dynamiquement les cycles.

Idée : On connaît le graphe des dépendances possibles, et on connaît le sous graphe des dépendances actuelles.
Il suffit de bloquer l'accès à une ressource qui pourrait créer un cycle. (Rester dans des états "sains")

Exemple : algorithme du Banquier de Dijkstra

Prévenir et éviter les interblocages

Exemple (5 philosophes)

- P_0 prend f_0, P_1 prend f_1, P_2 prend f_2, P_3 prend f_3
- P_4 demande f_4. Lui donner ? Non !
- P_0 possède f_0 : l'arc $f_0 \to f_1$ est possible.
- Etc. $f_0 \to f_1 \to f_2 \to f_3 \to f_4$ est possible
- Donner f_4 à P_4 créerait un cycle : un interblocage est possible.
- La demande de P_4 pour f_4 bloque jusqu'à ce que cette possibilité soit écartée

Détecter les interblocages

Il est théoriquement possible (pour l'OS) de détecter les interblocages.

Par exemple, si on a que des mutex (exclusion mutuelle, et pas de préemption), il suffit de construire le graphe de dépendance :

- Pour tout thread t bloqué dans un lock(R_t), tous les arcs entre $x \rightarrow R_t$, pour tout les x bloqués par t.

Et de tester ce graphe contient un cycle. Si oui : il y a un interblocage...

Mais que peut-on faire si on détecte un interblocage ?

Sous Linux ?

Le noyau Linux :

- S'occupe qu'il n'y ait pas d'interblocage dans le noyau
- Mais ne détecte/résout pas les interblocages des processus.

Pourquoi ?

- Quand on est dans une situation d'interblocage, on ne peut pas débloquer en respectant l'exclusion mutuelle
  - à part "tuer" quelque chose...
- On ne peut pas savoir à l'avance si on va arriver à une situation d'interblocage :
  Il faudrait analyser le code pour savoir les dépendances possibles...
  On frise avec des problèmes indécidables

Il faudrait des primitives de verrouillage beaucoup plus compliquées.
Cela en vaut il la peine ?

Livelock

Un autre type d'interblocage : le livelock

Aucun thread n'est bloqué, mais aucun thread n'avance (le code exécuté ne sert qu'à la gestion de concurrence)

Exemple : excès de politesse. Un thread veut laisser sa place pour une ressource à un autre thread si il le demande. Chaque thread se passe la main.

Concurrence : Ordonnancement

Ordonnancement : rappels

(Ici thread = thread ou processus non multi-threadé)

L'ordonnancement est préemptif.

Travail de l'ordonnanceur :

- Choisir à quel thread (prêt) il doit donner la main, et
- (pour les ordonanceurs préemptifs) combien de temps il lui donne.

État des threads

États des threads considérés par l'ordonnanceur :

- exécution : le thread est en exécution (sur un des coeurs)
- prêt : le thread attend que l'ordonnanceur lui donne la main
- en attente : le thread ne peut/doit pas être exécuté pour le moment (en attente d'une IO ou d'un mutex, sleep...)
- terminé

Changement d'état

Les threads "vivants" changent constamment d'état (exécution, prêt, attente)

- exécution $\rightarrow$ attente : appel système sur une ressource bloquante
- attente $\rightarrow$ prêt : la ressource se libère
- prêt $\rightarrow$ exécution : l'ordonnanceur donne la main au thread (dispatch)
- exécution $\rightarrow$ prêt :
  - le thread décide par lui même de rendre la main (POSIX : `sched_yield()`),ou
  - (ordo préemptif) le temps accordé au thread est dépassé (interruption)

Les différents temps

- temps total ("réel") : temps total d'un processus (de son création à sa terminaison)
- temps user : temps passé en mode utilisateur (temps passé en "exécution")
- temps système : temps passé en mode système (dans les appels systèmes)
- temps d'attente : temps passé en état "prêt"

(Note : pour voir les temps réels, user et sys : `time` *`commande`*).

Objectifs d'un ordonnanceur

Un ordonnanceur doit avoir plusieurs objectifs :

- Utiliser le(s) CPU à 100%
- Respecter l'équité
- Respecter les priorités ("nice")
- Minimiser le temps total d'une tache courte. (Réactivité. Par exemple : une commande.)
- Minimiser le temps total pour un long travail
- Éviter de faire trop de context-switch (par exemple, accorder des temps trop courts)
- Éviter de passer trop de temps dans l'ordonnanceur
- ...

Algorithmes l'ordonnancement

- Problème difficile
- Il y en a plusieurs possibles, en fonctions des objectifs principaux
- Cela pourrait être le sujet de tout un cours...
- Les algorithmes simples ont souvent des problèmes.

Stratégies "typiques" :

- First-Come, First-Served (FCFS) : (coopératif). Algo "minimal", on ne réfléchit pas. Peu de context switch. Problème : le temps de réponse peut être long (infini). Pas de gestion des priorités.
- Shortest-Job-First (SJF) : résout le problème de la réactivité. Problème : il faut connaître (ou prédire) le temps d'un processus a priori.

Algorithmes d'ordonnancement : Round-Robin

Round-Robin (RR) : (ordo préemptif)

- L'ordonnanceur a une liste (ou queue) L de threads "prêt".
- L'ordonnanceur prend (et retire) le premier thread (T) de la liste
- L'ordonnanceur exécute T, avec une limite de temps déterminé (ex : 0.1 seconde).
- Quand T rend la main (ex : IO bloquant), ou a épuisé son temps autorisé, l'ordonnanceur le rajoute à la fin de L.

Problèmes :

- Pas de gestion de la priorité.
- Si T fait souvent des appels à des I/O bloquantes, il est laissé.

Avec les priorités ?

Solutions :

- Avoir plusieurs listes (une par priorité)
  Si beacoup de priorités, un peu lourd...
  Décisions à prendre : quelle liste dispatcher ?

- Utilisation de files de priorités (plutôt que des listes/queues)
  (permet aussi de dépénaliser les threads qui font beaucoup d'I/O bloquantes)

Et avec plusieurs processeurs/coeurs ?

Un thread est généralement associé à un coeur
Pourquoi ?

- histoire de cache
- histoire de mémoire...

Mais si un coeur est moins utilisé qu'un autre, l'ordonnanceur peut décider de déplacer un thread.

L'ordonnanceur doit choisir quel coeur associer à un thread
Et décider quand le changer de coeur (load balance)

Parenthèse : UMA / NUMA

Architecture UMA (uniform memory access)

- une mémoire centrale partagée par plusieurs processeurs/coeurs

Exemples : vos machines (Intel Core ix, smartphones...)

Problème :

- si il y a beaucoup de coeurs, goulot d'étranglement pour l'accès mémoire
- les contrôleurs de mémoire sont (maintenant) souvent intégrés aux processeurs, et il difficile de concevoir des processeurs avec beaucoup de coeurs (plus de pertes, problème de dissipation thermique)

Parenthèse : UMA / NUMA

Architecture NUMA (non-uniform memory access)

- Chaque processeur (ou groupe de coeurs) dispose de sa mémoire (et forme un noeud)
- Mais chaque noeud peut accéder à toute la mémoire de la machine.
- Si c'est une mémoire d'un autre noeud, il faut passer par un bus qui inter-connecte les différents noeud (bus très rapide, mais quand l'accès est quand même plus lent)

Exemples : Systèmes multiprocesseurs courants (Intel Xeon, AMD Opterons...)

- Optimalement, il faudrait qu'un thread soit exécute dans le noeud où est sa mémoire

Contrainte supplémentaire pour l'ordonnanceur...

- Il faut interférer avec l'ordonnanceur mémoire
- déplacer un thread d'un noeud à un autre est plus contraignant

Ordonnanceur(s) de Linux

Plusieurs ordonanceurs au cours de l'histoire de Linux

- O(n) scheduler (Linux 2.4 : 2001-2011)
  - le temps est divisé en "epoch". À chaque epoch, chaque thread a droit à un certain nombre de temps CPU (basé sur la priorité)
  - Si un thread n'a pas épuise tout son temps CPU au cours d'une epoch, il rajoute la moitié du temps restant à son temps à l'epoch suivante.
  - Problème : temps linéaire en le nombre de processus entre chaque epoch
- O(1) scheduler (Linux 2.6, avant 2.6.23 : 2003-2007)
  - 140 niveaux de priorités (0-99 pour le système et temps réel, 100-139 pour les utilisateurs)
  - 2 queues par priorité : actif/inactif
  - pénalité si temps écoulé, récompense si I/O bloquante

Ordonnanceur(s) de Linux

Actuellement : Completely Fair Scheduler

- une file de priorité : arbre rouge-noir, indexé par le temps utilisé par le processus
- temps accordé : le temps que le thread a attendu $\times$ le ratio du temps qu'il aurait utilisé sur un processeur "idéal"
- l'ordonnanceur donne la main au thread qui a moins utilisé son temps (le plus petit dans la liste)
- quand le thread rend la main, l'ordonnanceur réinsère le thread dans l'arbre rouge-noir, avec son nouveau temps
- les processus avec beaucoup d'attente sont naturellement "récompensés"

Politiques et priorités POSIX

(Voir `man sched`)

On peut interférer avec l'ordonnanceur pour changer les politiques ou les priorités

(Généralement, plutôt utile pour les applications temps réel.)

Déjà vu : `nice()` pour un processus

Il y a aussi `getpriority()` / `setpriority()`. (Priorité d'un processus, d'un groupe de processus, et d'un utilisateur.)

Différentes politiques POSIX

Différentes politiques d'ordonnancement sont prévues dans POSIX (pour les processus et threads) :

- SCHED_OTHER : politique "standard"

Des politiques "temps réel" :

- SCHED_FIFO : (First-in-First-Out)
- SCHED_RR : (Round-Robin)

Dans ce cas, il y a une priorité supplémentaire (généralement entre 0 et 99) pour le thread
(Et quelques autres politiques, apparues plus récemment)

Pour changer la politique d'un processus : `sched_setscheduler()`

Pour changer la politique d'un thread :
`pthread_setschedparam()`

Affinités

On peur vouloir contrôler sur quel noeud un processus tourne, ou répartir ses threads sur différents noeuds, pour avoir de meilleurs performances.

Il est possible de choisir (sous Linux) sur quel(s) coeur(s) un thread peut tourner, avec `sched_setaffinity`.

Il est aussi possible de choisir des affinités mémoires. Voir `man numa`

Problème : Inversion de priorité

- A de forte priorité
- B de priorité moyenne
- C de faible priorité

Scénario :

- C bloque une ressource R
- A demande (et bloque) sur la ressource R
- B arrive
- B prend 100% CPU, car il a la priorité sur C
- C ne relâche pas R
- A ne peut pas être exécuté

B bloque A (par l'intermédiaire de C).
(problème réel : Mars pathfinder)

Problème : Inversion de priorité

Solutions :

- ne pas trop prioriser les threads de priorité supérieure. (Pas valable en temps réel)
- aléatoirement, donner du temps CPU à des taches de priorité basse (bricolage...)
- (priority ceiling) donner une priorité à un mutex. Si un thread bloque le mutex, il hérite (temporairement) de la priorité du mutex (si elle est plus haute)
- (priority inherance) si un thread C bloque le mutex, et un thread A de plus haute priorité demande le mutex, C hérite (temporairement) de la priorité de A.

Problème : Inversion de priorité

Dans pthread :

```
int pthread_mutexattr_getprotocol(
    const pthread_mutexattr_t *attr,
    int *protocol);
int pthread_mutexattr_setprotocol(
    pthread_mutexattr_t *attr,
    int protocol);
int pthread_mutex_setprioceiling(
    pthread_mutex_t* mutex,
    int prioceiling, int* old_ceiling );
```

`protocol` :

- PTHREAD_PRIO_NONE : le fait de bloquer le mutex n'affecte pas la priorité
- PTHREAD_PRIO_PROTECT : (priority ceiling) bloquer le mutex donne au thread la priorité du mutex (s'il est supérieur)
- PTHREAD_PRIO_INHERIT : (priority inherance) bloquer le mutex donne au thread le maximum des priorités des threads demandant le même mutex

Partage de mémoire entre processus (POSIX)

Mémoire partagée inter-processus

Plusieurs processus peuvent partager leur mémoire : c'est un IPC très rapide

Attention : comme pour les threads, il faut gérer la concurrence, soit :

- avec des sémaphores (pshared=1)
- avec des mutex partagés (attribut pshared)

Mémoire partagée inter-processus : cas simple

Entre père et fils :

```
char *x=mmap(NULL,65536,PROT_READ|PROT_WRITE,
    MAP_SHARED|MAP_ANONYMOUS,0,0);
if(fork()==0) {
   /* fils */
   strcpy(x,"COUCOU");
   return 0;
}
/* pere */
sleep(1);
printf("%s\n",x);
```

Affiche : COUCOU

Sans lien de parenté :

```
int fd=open("partage",O_RDWR|O_CREAT,0644);
ftruncate(fd,65536);
char *x=mmap(NULL,65536,PROT_READ|PROT_WRITE,
    MAP_SHARED,fd,0);
```

Processus 1 :

```
while(1) {
 snprintf(x,999,"COUCOU␣%d",rand());
 sleep(1);
}
```

Processus 2 :

```
while(1) {
 sleep(1);
 printf("%s\n",x);
}
```

- Cela marche, mais on utilise le disque (plus lent que la RAM)
- On voudrait mimer ce comportement, sans passer par le disque

Mémoire partagée inter-processus : `shm_open`

Solution : objets mémoire partagé (`man shm_overview`)

```
int fd=shm_open("/partage",O_RDWR|O_CREAT,0644);
ftruncate(fd,65536);
char *x=mmap(NULL,65536,PROT_READ|PROT_WRITE,
    MAP_SHARED,fd,0);
```

(Processus 1 et 2 comme précédemment)

- OK !
- En fait `/partage` sera dans un système de fichier en RAM, dans `/dev/shm/`

Objet mémoire POSIX

```c
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>

int shm_open(const char *name, int oflag,
    mode_t mode);
int shm_unlink(const char *name);
```

(compiler avec `-lrt`)

Renvoie un descripteur de fichier, sur lequel on peut faire les opérations qu'on a déjà vues :
`ftruncate`, `mmap`, `munmap`, `close`...

Sémaphore partagé entre processus

(Voir `man shm_overview`)

Deux façons de créer un sémaphore partagé :

- Sémaphore anonyme
- Sémaphore nommé

Sémaphore anonyme : le sémaphore doit être crée avec `sem_init`, avec `pshared=1`, dans une zone mémoire partagée (via `mmap`)

Sémaphore partagé entre processus

Sémaphores nommés (mécanisme similaire à `shm_open`) :

```
#include <semaphore.h>
#include <sys/stat.h>
#include <fcntl.h>

sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,
    mode_t mode, unsigned int value);
```

(compiler avec `-pthread`)

Mutex partagé entre processus

```
pthread_mutexattr_t mutexattr;
pthread_mutexattr_init(&mutexattr);
pthread_mutexattr_setpshared(&mutexattr,
    PTHREAD_PROCESS_SHARED);

pthread_mutex_init(pmutex,&mutexattr);
//...
```

`pmutex` doit être dans une zone mémoire partagée

Similairement, les variables de condition, rwlocks, barrières...
peuvent aussi être partagés entre processus.

Sécurité (vue rapide)

Introduction

Plusieurs types d'attaques à considérer :

- attaques locales
  - accès, ouverture de la machine
  - exploitation et escalade de privilège depuis un compte utilisateur
- attaques depuis l'extérieur (réseau)
  - accès, exploitation, escalade de privilège depuis une connexion réseau
  - écoute / modification des connexions réseau
  - DoS
  - facteur humain
- attaques passives (sans altération) :
  - écoute des paquets réseaux, scan de ports...
- attaques actives (altération) :
  - DoS, man in the middle, exploit

Accès physique à la machine

Attaque "depuis l'intérieur" : l'attaquant a accès à la machine (accès régulier en tant qu'utilisateur, vol...)

Solutions :

- empêcher le boot depuis USB, CD, réseau.
- mot de passe dans le BIOS et grub
  problème : si on peut ouvrir la machine, on peut effacer le mot de passe du BIOS (enlever la pile)
- fermer et attacher la machine à clef
  (attention, cela se crochète facilement)
- chiffrer le disque dur (et de préférence, tout le disque dur et le swap)
- tous les disques durs de portables devraient être chiffrés !

Exploitation de failles

"exploit" : L'attaquant utilise un bug du programme pour lui faire exécuter une fonction qu'il n'aurait pas dû en temps normal

Conséquences possibles :

- plantages
- appel d'une fonction dans la libc (ex : "return to libc")
- accès à un shell au niveau de privilège du processus ("shellcode")
- ...

"Escalade de privilège"
Exemple : on est utilisateur lambda, et on utilise exploite une faille dans un programme setuid root pour pouvoir lancer une commande/fonction avec les privilèges root

Bugs couramment exploités

Problèmes de mémoire

- dépassement de tableaux (buffer, stack, heap overflow...)
- problèmes de pointeurs...

Problèmes d'entrées

- fonctions non sûres (sprintf...)
- dépassement d'entiers
  - attention au négatif !
- chaîne de formatage

```
int main(int ac,char **av) {
   printf(av[1]);
   return 0;
}
```

- injection (injection SQL...)

...

Bugs couramment exploités

Prévention (voir cours "Bonnes pratiques, débogage et optimisation") :

- Bonne pratiques de codage
- Éviter les fonctions réputés dangereuses
- Vérifier les codes retours
- Vérifier les dépassements
- Compiler avec -Wall
- `valgrind`
- ...

Exploitations de failles depuis l'extérieur (Réseau)

L'attaquant se connecte à une application via le réseau, et exploite une faille de l'application

Solutions (partielles) :

- garder le système à jour (mise à jour de sécurités)
- limiter les logiciels accessibles (et surtout ceux qui peuvent mener à une escalade de privilèges) à ceux nécessaires
- filtrer les paquets ("firewall") (Linux : `iptables` ou autres)
- mots de passes robustes (pas de mots de passe vide, bidon, défaut, même pour les tests !)...

Détecter :

- analyse de logs (/var/log/)
  - attention, cela peut être effacé par l'attaquant
- systèmes de détection d'intrusion...

Attaques sur le réseau :

- Écoute des messages réseaux
- Manipulation des messages réseaux
- Spoofing (usurpation d'une adresse)
- DoS (Denial of Service)
- ...

Sécurité sur le réseau

Les liens réseaux sont généralement considérés comme non sûrs :

- Tout le monde peut écouter ce qui passe sur le wifi
- Il est possible d'écouter ce qui passe sur l'ethernet (même si c'est routé)
- Un routeur IP peut être compromis, ou espionné
- ...

Solutions :

- Chiffrer/Signer les messages
- Certificats
- (Généralement tous les protocoles ont une version chiffrée)

Cryptographie : chiffrement

Deux grands types de systèmes cryptographiques :

Cryptographie symétrique :

- il faut la même clef pour crypter et décrypter
- rapide

Cryptographie asymétrique :

- les messages peuvent être chiffrés par tout le monde, via une clef publique
- ils ne peuvent être décryptés que via une clef privée
- problèmes mathématiques, plus lent

Souvent, les protocoles utilisent les deux : une phase asymétrique pour donner/échanger une clef privée, puis le reste en symétrique.

Principe de Kerckhoffs

Principe de Kerckhoffs :

- "La sécurité d'un cryptosystème ne doit reposer que sur le secret de la clef"

Maxime de Shannon :

- "L'adversaire connaît le système"

Plus un algorithme de cryptographie est publique et connu, plus il sera testé et sûr...

Cryptographie symétrique :

- Même clef pour crypter et décrypter
- Généralement, travaille sur des blocs de $b = 32 \cdot k$ bits .
- La fonction de chiffrement et une bijection de 2^b vers 2^b

Exemples :

- DES (Data Encryption Standard) :
  - Blocs de 64 bits, clef de 56 bits
  - Ancien standard, mais devenu bien trop faible. Ne plus utiliser !
- AES (Advanced Encryption Standard) :
  - Blocs de taille 128, clefs de taille 128, 192 ou 256 bits

Avantages : rapide (opérations simples), souvent en hardware
Inconvénient : les clefs doivent être partagés sur un canal sécurisé

Cryptographie symétrique : Mode d'opération

Si on chiffre une suite de blocs $m_0, m_1 \ldots$, on n'utilise généralement pas la fonction telle quelle sur chaque bloc.

Sinon :

- Deux blocs identiques seront encodés de la même manière
- On peut facilement dupliquer et supprimer des bouts de messages...

"Mode d'opération" sur les blocs :

- Cipher Block Chaining (CBC) :
  - $c_0 = f_e(m_0 \oplus IV)$
  - $c_i = f_e(m_i \oplus c_{i-1})$
- Cipher Feedback (CFB) :
  - $c_0 = m_0 \oplus f_e(IV)$
  - $c_i = m_i \oplus f_e(c_{i-1})$
- ...

(IV : initialisation vector)

Cryptographie asymétrique

Les messages sont chiffrés via une clef publique, et déchiffrés via une clef privée

Exemples :

- RSA, El-Gamal, ECC, DH

Problème :

- plus lent (opérations compliquées)
- clefs généralement grandes
- souvent basés sur des problèmes qu'on suppose difficiles...

Cryptographie asymétrique

Basés sur des problèmes mathématiques difficiles :

- Décomposition en facteurs premiers (RSA, Rabin...)
- Logarithme discret : (ElGamal, Diffie-Hellman)
  - Z_p
  - Courbes elliptiques (ECC) : clefs plus petites
  - ...
- ...

"Post-quantique" :

- plus court vecteur dans un réseau (NTRU)
- ...

Exemple : Chiffrement RSA (Rivest-Shamir-Adleman)

- Alice choisi deux nombres premiers p et q, et un entier e
- La clef publique est (pq, e)
- La clef privée est (p, q, e)
- Bob chiffre le message m en $c = m^e \pmod{pq}$
- Alice déchiffre $m = c^f \pmod{pq}$, où $ef = 1 \pmod{\varphi(pq)}$
- $\varphi(pq) = (p-1)(q-1)$ (Indicatrice d'Euler)
- $m^{ef} = m^{1+k\varphi(pq)} = m \times (m^{\varphi(pq)})^k = m \pmod{pq}$
- (souvent) difficile de retrouver p et q à partir de pq

Exemple : Chiffrement RSA (Rivest-Shamir-Adleman)

Problèmes :

- Si m est petit, ou si $log(m) \times e < log(pq)$, on peut facilement retrouver m à partir de c
- Si l'ensemble des possibilités pour m est petit (ex "OUI" ou "NON"), on peut tout essayer
- Deux blocs identiques sont codés de la même manière.

Solution : "Padding"

- Une partie importante du message (au moins 8 octets) sont remplis aléatoirement

Exemple 2 : Échange de clef de Diffie-Hellman

- Alice et Bob se mettent d'accord (publiquement) sur un groupe G (ex : $(Z/pZ, \times)$, p premier), et sur un générateur $g \in G$
- Alice choisi a et Bob choisi b (secrets)
- Alice envoie g^a à Bob
- Bob envoie g^b à Alice
- Alice calcule la clef $c = (g^b)^a$
- Bob calcule la clef $c = (g^a)^b$
- Alice et Bob peuvent se servir de c pour chiffrer avec un système symétrique
- (souvent) difficile de retrouver a depuis g^a et g (Logarithme discret)
- ECC : encore plus difficile...

Taille des clefs & records

NIST (National Institute of Standards and Technology) (2012) :

Sym.	RSA / DH	ECC
80	1024	160-233
112	2048	224-255
128	3072	256-383
192	7680	384-512
256	15360	512+

Record de clef RSA cassée : 768 bits (2010)
Record de clef ECC cassée : 113 bits (2015)

Fonction de hachage cryptographique

Fonction de hachage :
associe à une donnée de taille arbitraire une image de taille fixe

Généralement, $f : 2^k \rightarrow 2^b$ où k est quelconque, et b est un multiple de la taille des mots machine (64, 128, 256...)

Applications :

- Généralise la somme de contrôle (vérifier qu'un message/fichier n'a pas été modifié)
- Table de hachage

Fonction de hachage cryptographique

Fonction de hachage cryptographique :

- rapide à calculer
- sens-unique : étant donné y, difficile de trouver x tel que $f(x) = y$
- résistante faible aux collisions : étant donné x, difficile de trouver $x' \neq x$ tel que $f(x) = f(x')$
- résistante forte aux collisions : difficile de trouver $x \neq x'$ tels que $f(x) = f(x')$

Exemples :

- MD5 : 128 bits
- SHA-1 : 160 bits, SHA-256 : 256 bits

Application :

- Sommes de contrôles (vérifier qu'un message/fichier n'a pas été modifié volontairement) (`md5`, `sha1`...)
- Signatures

Signature numérique

Permet au destinataire d'un message :

- d'identifier l'émetteur
- de vérifier que le message n'a pas été modifié

Même principe que la cryptographie asymétrique

Généralement, utilisé conjointement avec une fonction de hachage :

- On signe le haché du message.

Signature numérique

Exemple : signature RSA :

- $enc(m) = m^e$ et $dec(c) = c^f$ modulo pq
  avec $ef = 1 \pmod{(p-1)(q-1)}$
- enc et dec sont commutatives :
  $enc(dec(x)) = dec(enc(x)) = x$.

- Bob envoie un message m à Alice
- Il calcule le haché $h(m)$ de m
- Bob envoie m concaténé à $s = dec(h(m))$

- Alice vérifie si $enc(s) = h(m)$

Chiffrement + Signature RSA :

Alice et Bob ont chacun leur clef

- Bob envoie un message m à Alice
- $s = \text{enc}_b(h(m))$.
- Bob envoie $m' = \text{enc}_a(m|s))$

- Alice déchiffre m' : $m|s = \text{dec}_a(m')$
- Alice vérifie si $\text{dec}_b(s) = h(m)$

"Man in the middle"

Alice et Bob doivent préalablement échanger leur clef publiques.

Si cela se fait sur un canal non sécurisé : un attaquant peut modifier tous les messages entre Alice et Bob, il peut substituer les clefs publiques (dont il ne connaît pas clef privées) par des nouvelles clefs publiques qu'il a généré.

Attaque "Man in the middle".

"Man in the middle"

- Bob envoie sa clef publique pub_b à Alice
- Oscar intercepte le message, et remplace pub_b par $pub_{b'}$, avant de le renvoyer à Alice
- Alice pense que la clef de Bob est $pub_{b'}$, et lui envoie un message en chiffrant avec $pub_{b'}$
- Oscar intercepte le message, le déchiffre avec la clef $priv_{b'}$ et envoie à Bob le message chiffré avec pub_b
- Bob reçoit un message (éventuellement signé par Alice), et ne s'aperçoit de rien.

"Man in the middle"

Solutions :

- Échanger les clefs par un canal sûr (par exemple en personne)
- Certifications de clefs et autorités de confiances :
  - faire certifier sa clef par une autorité de confiance. La certification réside dans la signature par l'autorité de confiance

Exemple :

- Certificat : $cert$ = "Michael Rao, 857736C8"
- Certification par l'autorité CA : $cert|sig_{CA}(hash(cert))$ (après vérification de l'identité)
- On accepte tous les certificats de CA.

En pratique...

- Des fonctions de cryptographie sont cassés ou trop faibles : MD4, DES, WEP...
- Souvent, les problèmes ne viennent pas des fonctions cryptographie, mais de protocoles mal faits
- Généralement, évitez de designer vous même vos fonctions/protocoles de chiffrement
- Utilisez si possibles des librairies/protocoles existants et éprouvés !

SSL/TLS

SSL = Secure Sockets Layer
TLS = Transport Layer Security

Protocoles de sécurisation des échanges. ("Couche supplémentaire" dans le modèle par couche)

Permet de :

- chiffrer
- authentifier le client et le serveur
- vérifier l'intégrité des données

Exemple : HTTPS = HTTP sur SSL/TLS,

Programmes / Librairies

- Librairies implémentant les fonctions cryptographiques courantes : openssl : (implémente SSL et les fonctions cryptographiques bas niveau), libgcrypt...
- PGP/GPG pour chiffrer/signer les mails.
- SSH : session, copie de fichiers, système de fichier réseau...
- Chiffrer les disques : LUKS, encfs (user-space)