INFO0604
PROGRAMMATION MULTI-THREADÉE

COURS 3

GESTION DES SYNCHRONISATIONS
MODÈLES D'UTILISATION DES THREADS

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

Pierre Delisle
Département de Mathématiques, Mécanique et Informatique
Décembre 2019

Plan de la séance

- Invariant et section critique
- Mutex
- Variables de condition
- Visibilité de la mémoire entre les threads
- Modèles d'utilisation des threads

NOTION D'INVARIANT ET SECTION CRITIQUE

Invariant

- Assomption faite par un programme, particulièrement à propos des relations entre des ensembles de données
- Exemple : Liste
  - Possède un pointeur de tête
    - Peut être NULL ou un pointeur sur le premier élément
  - Chaque élément possède un pointeur sur le prochain élément
  - Le pointeur du dernier élément est NULL
- Ces règles sont les invariants de la liste
- Si un programme rencontre un invariant non respecté,
  - Ex : la tête pointe sur quelque chose qui n'est pas un élément valide,
- Le programme produira un résultat incorrect ou se terminera en état d'erreur

Section critique

- Série d'instructions qui affecte un état partagé et qui doit être exécutée de façon strictement séquentielle
- Presque toujours associé à un invariant de donnée
- Exemple : code permettant de retirer un élément de la liste
  - Brise temporairement un invariant
  - Doit être effectué en section critique pour éviter l'accès à la liste pendant la suppression

Synchronisation

- Les synchronisations protègent le programme des « brisures d'invariants »
- Coopération des threads
- Un thread verrouille une ressource pendant la brisure de l'invariant, effectue ses traitements, ramène l'invariant et libère la ressource
- 2 types principaux de synchronisations
  - Mutex
  - Variable de condition

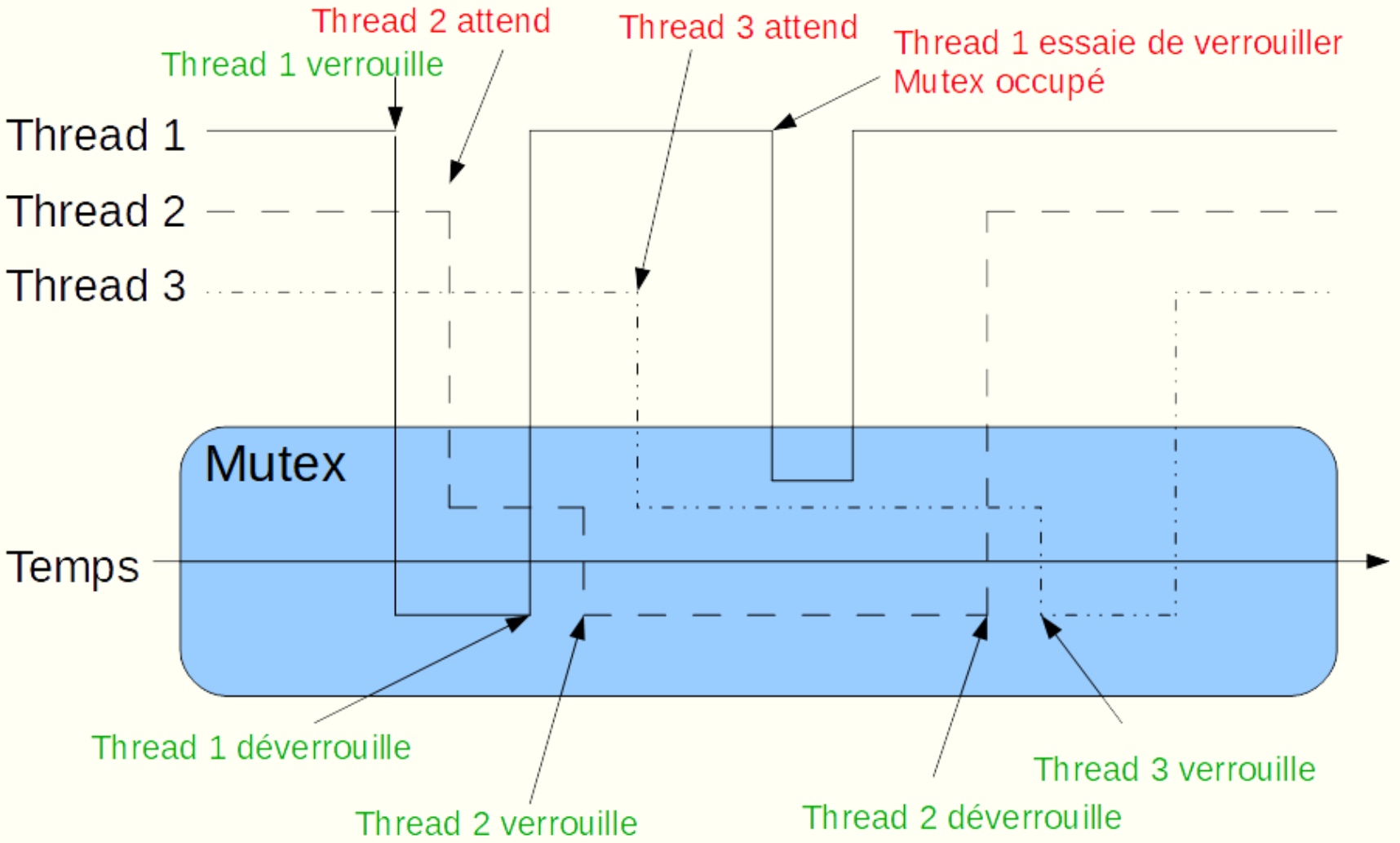
MUTEX

Définition, Création/Destruction, Utilisation

Définition d'un mutex

- MUTual EXclusion
- Assure que les accès à une donnée partagée sont mutuellement exclusifs
  - Seulement un thread à la fois est autorisé à écrire
  - Les autres threads qui veulent lire/écrire attendent
- Cas particulier d'un sémaphore

3 threads et un mutex



Création/Destruction d'un mutex

- pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
- int pthread_mutex_init (pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
- int pthread_mutex_destroy (pthread_mutex_t *mutex);

Mutex statique

- PHTREAD_MUTEX_INITIALIZER : Macro initialisant un mutex statique, attributs par défaut
- Généralement déclaré "file scope"

```
#include <pthread.h>

typedef struct int_protege {
        pthread_mutex_t mutex;    /*protege l'acces a la valeur*/
        int valeur;               /*la valeur en question*/
}  int_p;

int_p element = {PTHREAD_MUTEX_INITIALIZER, 0};

int main(int argc, char *argv) {
        return 0;
}
```

Mutex dynamique

- Utilisé
  - Lorsque la structure contenant le mutex est dynamique
  - Lorsque que le mutex n'est pas initialisé avec les attributs par défaut
- Il faut alors appeler la méthode *phtread_mutex_init()* (une fois seulement)
  - Avant le création de threads ou
  - En appelant *pthread_once()*

```
#include <pthread.h>

typedef struct int_protege {
        pthread_mutex_t mutex;      /*protege l'acces a la valeur*/
        int valeur;                 /*la valeur en question*/
} int_p;

int main(int argc, char *argv) {
        int_p *element;
        int statut;

        element = (int_p *) malloc (sizeof(int_p));
        if (element == NULL)
                fprintf(stderr, "Probleme allocation structure\n");
        statut = pthread_mutex_init(&element->mutex, NULL);
        if (statut != 0)
                fprintf(stderr, "Probleme initialisation mutex\n");
        statut = pthread_mutex_destroy(&element->mutex);
        if (statut != 0)
                fprintf(stderr, "Probleme destruction mutex\n");
        free(element);
        return statut;
}
```

Utilisation des mutex

- Généralement une bonne idée d'associer un mutex directement avec la donnée qu'il protège
- Destruction d'un mutex
  - Dynamique : appel de *pthread_mutex_destroy()*
  - Statique : destruction automatique
- Il ne faut pas détruire un mutex
  - S'il est verrouillé
  - Si un/des thread(s) sont bloqués sur celui-ci

Verrouiller/Déverrouiller un mutex

- Verrouillage du mutex
  - int pthread_mutex_lock(pthread_mutex_t *mutex);
  - int pthread_mutex_trylock(pthread_mutex_t *mutex);
- Utilisation de la donnée partagée
- Déverrouillage du mutex
  - int pthread_mutex_unlock(pthread_mutex_t *mutex);
- Exemple : AlarmeMutex.c

- sched_yield()
  - Redonne la possibilité au processeur d'exécuter un thread qui est prêt
  - Si aucun thread n'est prêt, le processeur revient à l'exécution du thread appelant
  - Dans le cas du programme d'alarme
    - Permet au thread principal d'accepter une nouvelle entrée au clavier et d'ajouter le nouvel élément dans la liste

Les verrous non-bloquants

- Verrou standard
  - un thread tentant de verrouiller un mutex bloque si le mutex est déjà verrouillé
- On peut plutôt désirer que le thread fasse autre chose d'utile plutôt qu'attendre
- *pthread_mutex_trylock()* retournera un statut d'erreur plutôt plutôt que de bloquer
- Dans ce cas, attention de déverouiller le mutex seulement si trylock a retourné un succès !
- Exemple : compteur.c

La taille adéquate d'un mutex

- Si 2 variables partagées doivent être protégées
  - Un « petit » mutex pour chaque variable ?
  - Un « gros » mutex pour les deux variables ?
- Facteurs à considérer
  - Un mutex n'est pas « gratuit ». Il faut du temps pour le verrouiller et du temps pour le déverrouiller
    - Il faut donc en utiliser aussi peu que possible
  - Les mutex sérialisent l'exécution. Si un gros mutex est verrouillé, les threads vont attendre
    - Il faut donc minimiser l'attente en en créant plusieurs
  - Les 2 premiers facteurs sont contradictoires !
- Que faire ?
  - Intuition
  - Expérimentation
  - Expérience

Utiliser plusieurs mutex

- Dans certains cas, un mutex n'est pas suffisant
  - Protection de la tête d'une file ET de ses éléments
  - Structure d'arbre --> un mutex par noeud
- Attention aux verrous mortels (deadlocks)

Thread 1	Thread 2
pthread_mutex_lock (&mutex_a); pthread_mutex_lock (&mutex_b);	pthread_mutex_lock (&mutex_b); pthread_mutex_lock (&mutex_a);

Utiliser plusieurs mutex

- Il existe différentes façons d'utiliser plusieurs mutex en évitant les verrous mortels
  - Hiérarchie fixe de verrous
    - Le code qui a besoin de mutex_a et de mutex_b doit toujours verrouiller mutex_a d'abord
    - Exemple : Verrouiller la tête de file et ensuite le noeud
  - Essai et retour en arrière
    - Après verrou du premier mutex, effectuer des verrous conditionnels sur les suivants (pthread_mutex_trylock())
    - Si un essai échoue, tout libérer et recommencer
  - Chainage de verrous
    - Verrou de mutex_a
    - Verrou de mutex_b ET libération de mutex_a
    - Utile pour traverser des structures de données comme les listes et les arbres
- Mais attention !
  - Il faut éviter d'écrire des programmes qui passent leur temps à verrouiller/déverrouiller plutôt que de faire du travail utile !

VARIABLES DE CONDITION

Définition, Création/Destruction, Utilisation

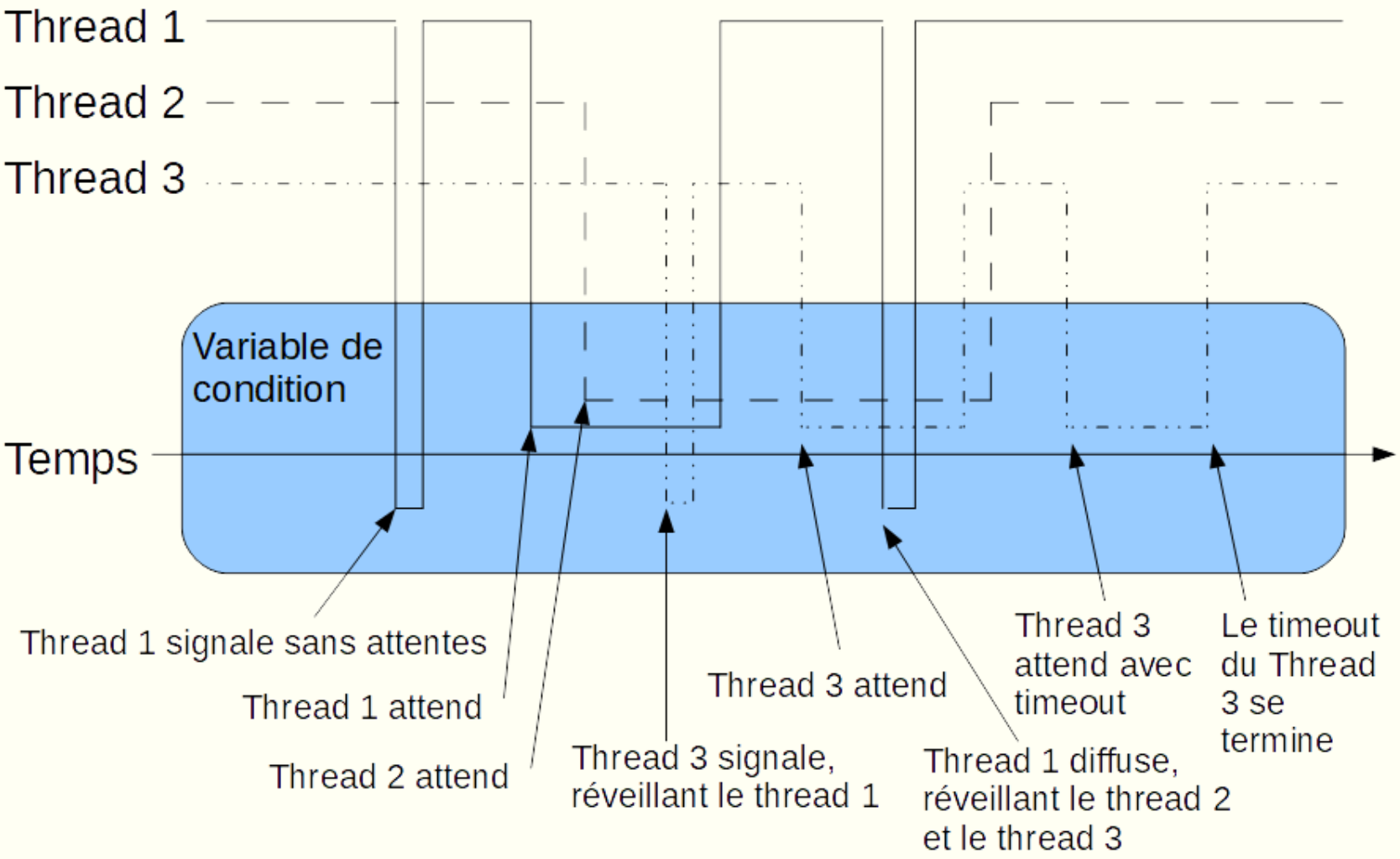
Définition d'une variable de condition

- Permet de communiquer de l'information à propos de l'état de données partagées …
  - La file n'est plus vide
  - La file est maintenant vide
- …et de synchroniser les threads en conséquence
- Marins à la dérive
  - Sommeil, coups d'épaule et cris
- Suspend l'exécution des threads lorsqu'ils n'ont rien d'utile à faire
- Exemple : Si un thread n'a rien à faire avec une file vide
  - Il attendra qu'un élément soit ajouté à la file
- Dans ce cas, ce thread doit
  - Verrouiller le mutex pour déterminer l'état de la file
  - Déverouiller le mutex et se mettre en attente du changement de l'état de la file
- Le déverrouillage et l'attente sont *atomiques* (indivisibles)

Définition d'une variable de condition

- Donc, une variable de condition est
  - Un mécanisme de signalement associé à un mutex
  - Par association, associé à une donnée partagée
- L'*attente* d'une variable de condition
  - Libère le mutex et place le thread en attente
    - D'un signalement de la variable de condition (qui réveille un thread en attente) ou
    - D'une diffusion de la variable de condition (qui réveille tous les threads en attente)
  - Le thread se réveille alors avec le mutex **verrouillé**
- Une variable de condition est toujours associée à un mutex
  - Mais un mutex n'est pas toujours associé à une variable de condition !
- Un mutex peut être associé à une ou plusieurs variables de condition
  - Un seul mutex synchronise tous les accès à une file
  - Une variable de condition pour signaler la file vide
  - Une variable de condition pour signaler la file pleine

Fonctionnement d'une variable de condition avec 3 threads



Création/Destruction d'une variable de condition

- pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
- int pthread_cond_init (pthread_cond_t *cond, pthread_condattr_t *condattr);
- int pthread_cond_destroy (pthread_cond_t *cond);

Variable de condition statique

- PHTREAD_COND_INITIALIZER
  - Macro initialisant une variable de cond. statique, attributs par défaut
  - Généralement déclaré "file scope"

```
#include <pthread.h>

typedef struct int_protege {
        pthread_mutex_t mutex;   /*protege l'acces a la valeur*/
        pthread_cond_t cond;      /*Signale un changement a la valeur*/
        int valeur;               /*la valeur en question*/
}  int_p;

int_p element = {PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, 0};

int main(int argc, char *argv[]) {
        return 0;
}
```

Variable de condition dynamique

- Utilisée
  - Lorsque la structure contenant la variable de condition est dynamique
  - Lorsque que la variable de condition n'est pas initialisée avec les attributs par défaut
- Il faut alors appeler la méthode *phtread_cond_init()* (une fois seulement)
  - Avant le création de threads ou
  - En appelant *pthread_once()*

```
typedef struct int_protege {
        pthread_mutex_t mutex;          /*protege l'acces a la valeur*/
        pthread_cond_t cond; /*Signale un changement a la valeur*/
        int valeur;             /*la valeur en question*/
}  int_p;
int main(int argc, char *argv[]) {
        int_p *element;
        int statut;
        element = (int_p *) malloc(sizeof(int_p));
        if (element == NULL)
                fprintf(stderr, "Probleme allocation structure\n");
        statut = pthread_mutex_init(&element->mutex, NULL);
        if (statut != 0)
                fprintf(stderr, "Probleme initialisation mutex\n");
        statut = pthread_cond_init(&element->cond, NULL);
        if (statut != 0)
                fprintf(stderr, "Probleme initialisation variable cond\n");
        statut = pthread_cond_destroy(&element->cond);
        if (statut != 0) fprintf(stderr, "Probleme destruction variable cond\n");
        statut = pthread_mutex_destroy(&element->mutex);
        if (statut != 0)   fprintf(stderr, "Probleme destruction mutex\n");
        free(element); return statut;
}
```

Utilisation d'une variable de condition

- Généralement une bonne idée d'associer une variable de condition avec un seul prédicat
  - Et à l' « encapsuler » avec son mutex associé
- Destruction d'une variable de condition
  - Dynamique : appel de *pthread_cond_destroy()*
  - Statique : destruction automatique
- Il ne faut pas détruire une variable de condition
  - Si un/des thread(s) sont bloqués ou en attente sur celle-ci

Attendre sur une variable de condition

- int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t * mutex);
- int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mutex,

  struct timespec *expiration);
- À l'appel
  - Déverouillage du mutex avant blockage du thread
- Au réveil
  - Re-verrouillage du mutex avant exécution du thread

Fonctionnement

- Tous les threads qui attendent concurremment sur une variable de condition
  - Doivent spécifier le même mutex associé
- Les threads peuvent attendre des variables de condition différentes associées au même mutex
- Il est important de tester le prédicat
  - Après verrouillage du mutex et avant attente
  - Lorsque le thread se réveille
- Exemple : VariableCond.c
- Lors du réveil d'un thread (et verrou du mutex), un autre thread peut déjà avoir acquis le mutex
  - Après le déverrouillage du mutex par cet autre thread, le prédicat peut avoir été remis à faux
- Il est souvent utile d'utiliser des « prédicats relaxés »
  - «La file peut être vide» plutôt que «la file est vide»

Réveiller les threads en attente

- Réveiller un seul thread en attente : signal
  - int pthread_cond_signal (pthread_cond_t *cond);
- Réveiller tous les threads en attente : diffusion
  - int pthread_cond_broadcast (pthread_cond_t *cond);

- Exemple : AlarmeCond.c

VISIBILITÉ DE LA MÉMOIRE ENTRE THREADS

Un peu de recul face à la programmation multi-thread

Threads et mémoire

- Les mutex et variables de condition permettent de synchroniser les instructions des threads
- Il ne s'agit pas seulement de s'assurer que 2 threads n'écrivent pas dans la même zone mémoire
- Il faut comprendre comment les threads « voient » la mémoire de l'ordinateur
  - Les règles à respecter ne sont pas celles de l'architecture matérielle, mais celles de Pthreads

Visibilité mémoire de Pthreads

Règle 1

- Les valeurs de la mémoire qu'un thread voit lorsqu'il appelle *pthread_create* sont aussi vues par le nouveau thread lorsqu'il démarre
  - Une donnée écrite à la mémoire après l'appel de *pthread_create* n'est pas nécessairement vue par le nouveau thread, **même si l'écriture se produit avant le démarrage du thread**

Règle 2

- Les valeurs de la mémoire qu'un thread voit lorsqu'il déverrouille un mutex (directement ou par l'attente d'une variable de condition) sont aussi vues par n'importe quel thread qui verrouille plus tard le même mutex
  - Une donnée écrite à la mémoire après le déverrouillage du mutex n'est pas nécessairement vue par le thread qui verrouille le mutex, **même si l'écriture se produit avant le verrouillage**

Visibilité mémoire de Pthreads

Règle 3

- Les valeurs de la mémoire qu'un thread voit lorsqu'il se termine (annulation, retour de la fonction, appel de *pthread_exit*) sont aussi vues par un thread qui le joint en appelant *pthread_join*

Règle 4

- Les valeurs de la mémoire qu'un thread voit lorsqu'il signale ou diffuse (broadcast) une variable de condition sont aussi vues par un thread qui est réveillé par ce signal ou diffusion
  - Une donnée écrite à la mémoire après le signal ou la diffusion n'est pas nécessairement vue par le thread qui se réveille, **même si l'écriture se produit avant le réveil**

Exemple 1 – Exécution correcte

Thread 1	Thread 2

pthread_mutex_lock (&mutex1);
variableA = 1;
variableB = 2;
pthread_mutex_unlock (&mutex1);
	pthread_mutex_lock (&mutex1);
	localA = variableA;
	localB = variableB;
	pthread_mutex_unlock (&mutex1);

- Lorsque le thread 2 retourne de *pthread_mutex_lock*, il verra les mêmes valeurs que le thread 1 a vu au moment où il a appelé *pthread_mutex_unlock*

Exemple 2 – Exécution incorrecte

Thread 1	Thread 2

pthread_mutex_lock (&mutex1);
variableA = 1;
pthread_mutex_unlock (&mutex1);
variableB = 2;
	pthread_mutex_lock (&mutex1);
	localA = variableA;
	localB = variableB;
	pthread_mutex_unlock (&mutex1);

- Lorsque le thread 2 retourne de *pthread_mutex_lock*, il verra la même valeur de *variableA* que le thread 1 a vu au moment où il a appelé *pthread_mutex_unlock,* mais pas nécessairement celle de *variableB*

Quoi faire en tant que programmeur ?

- Si possible, s'assurer qu'un seul thread accèdera à une donnée spécifique
  - Chaque thread est synchrone avec lui-même
  - Moins il y a de données partagées, moins il y a de travail à faire (autant pour le programmeur que pour le système !)
- À chaque fois que 2 threads doivent accéder à une donnée commune, appliquer une des règles de visibilité de Pthreads
  - Dans la plupart des cas, signifie d'utiliser un mutex

Quelques éléments de réflexion…

- Dans un programme séquentiel synchrone (avec un seul thread), il est toujours sécuritaire de lire ou écrire à n'importe quel emplacement de la mémoire
  - Si un programme écrit une valeur et la lit plus tard, il lira toujours la dernière valeur qu'il a écrit
- Dans un programme multi-thread, ce n'est pas aussi simple !
  - Vous ne pouvez jamais être certain de toujours savoir ce que chaque thread est en train de faire !
  - Le temps d'exécution de chaque thread peut ne pas être prévisible
  - Les threads peuvent s'exécuter simultanément sur plusieurs processeurs/coeurs

Quelques éléments de réflexion…

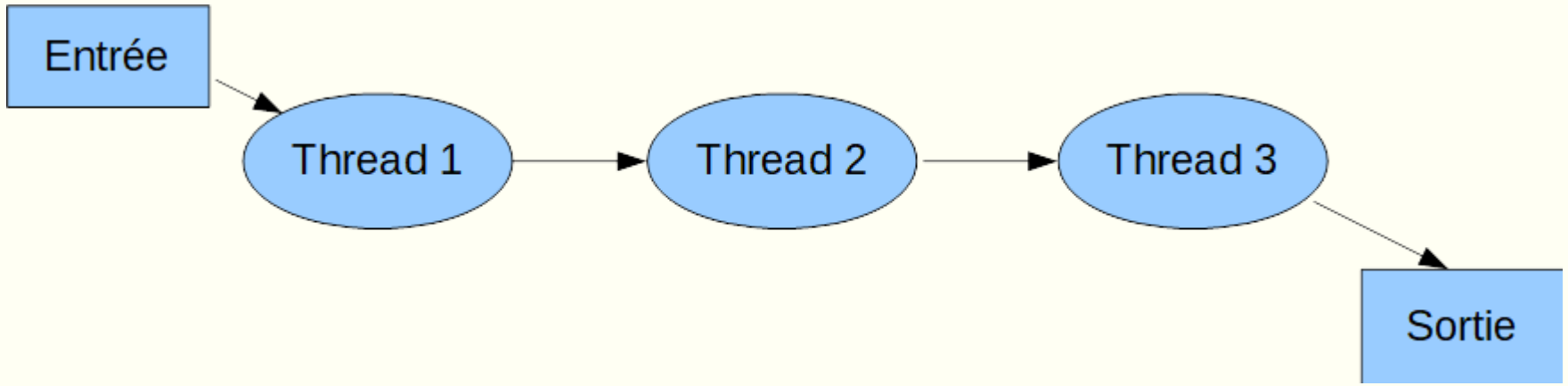
- Une adresse mémoire ne peut contenir qu'une seule valeur à la fois
  - Si 2 threads écrivent des valeurs différentes à une meme adresse, une seule valeur restera au final
  - Même si thread1 écrit après thread2, la valeur finale ne sera pas nécessairement celle de thread1
    - Systèmes de caches et de bus mémoire
    - Fonctionnement pouvant être différent d'une exécution à l'autre, ou d'un ordinateur à un autre
- Les synchronisations garantissent la visibilité
- Fonction réentrante
  - Peut être interrompue au milieu de son exécution et appelée à nouveau avant que les appels précédents soient terminés
- Fonction *thread-safe*
  - Manipule les données partagées en garantissant une exécution « sécuritaire » par plusieurs threads en même temps

MODÈLES D'UTILISATION DES THREADS

Pipeline, Équipe de travail, Client/serveur

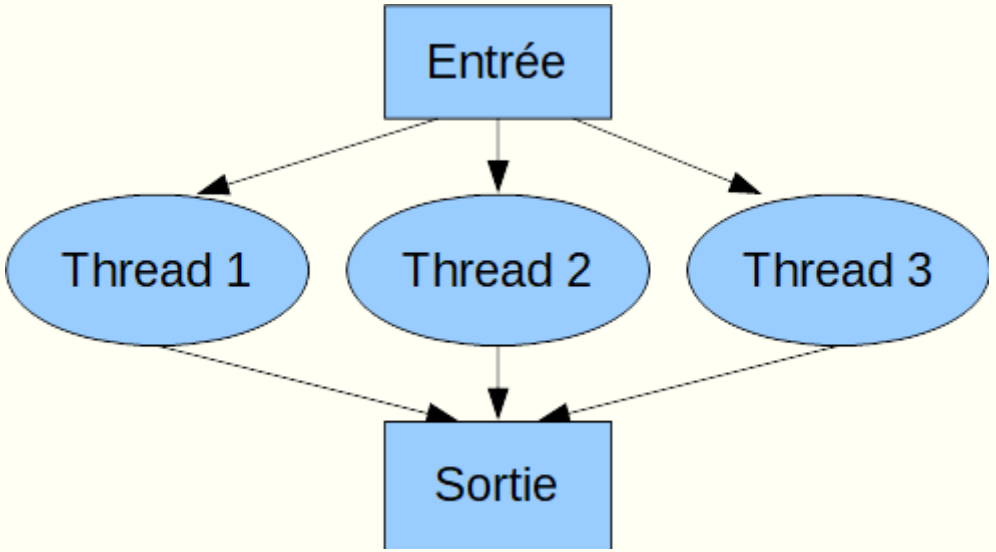
Pipeline

- Chaque thread exécute la même opération sur une séquence d'éléments de données
  - Le résultat de l'opération sur une donnée est passé à un autre thread pour la prochaine opération
- Semblable à une ligne d'assemblage
- Exemple : Pipeline.c



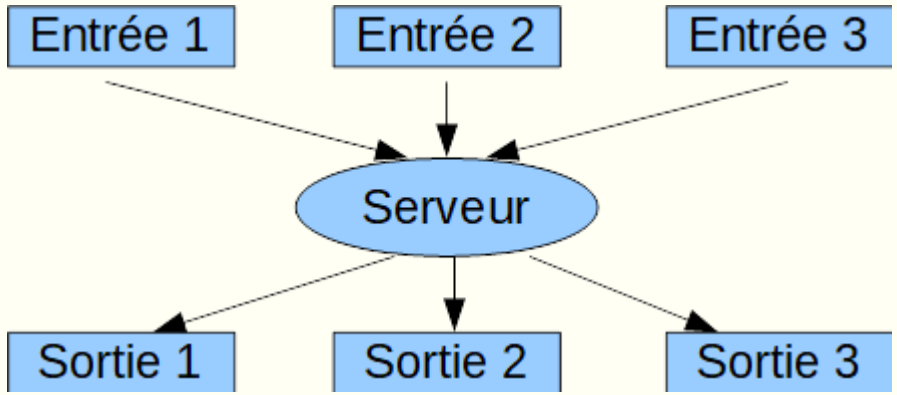
Équipe de travail (work crew)

- Les données sont traitées indépendamment par un ensemble de threads
- Décomposition « parallèle »
  - Ex. : chaque thread traite une ligne d'une matrice
- Chaque thread peut faire la même opération ou une opération différente
  - SIMD ou MIMD
- Ex. : Equipe.c



Client/serveur

- Le client fait une requête au serveur de traiter une donnée
- Le client peut
  - Attendre le résultat du serveur
  - Continuer ses propres traitements et vérifier le résultat plus tard lorsque requis
- Ex. : ClientServeur.c



PROCHAIN COURS

PROGRAMMATION MULTI-THREADÉE AVANCÉE