Utilisation des systèmes UNIX

Benjamin Negrevergne

PSL University – Paris Dauphine – Équipes *MILES*

Programmation parallèle

Plusieurs codes s'exécutent simultanément pour répondre a un seul besoin.

Intérêt:

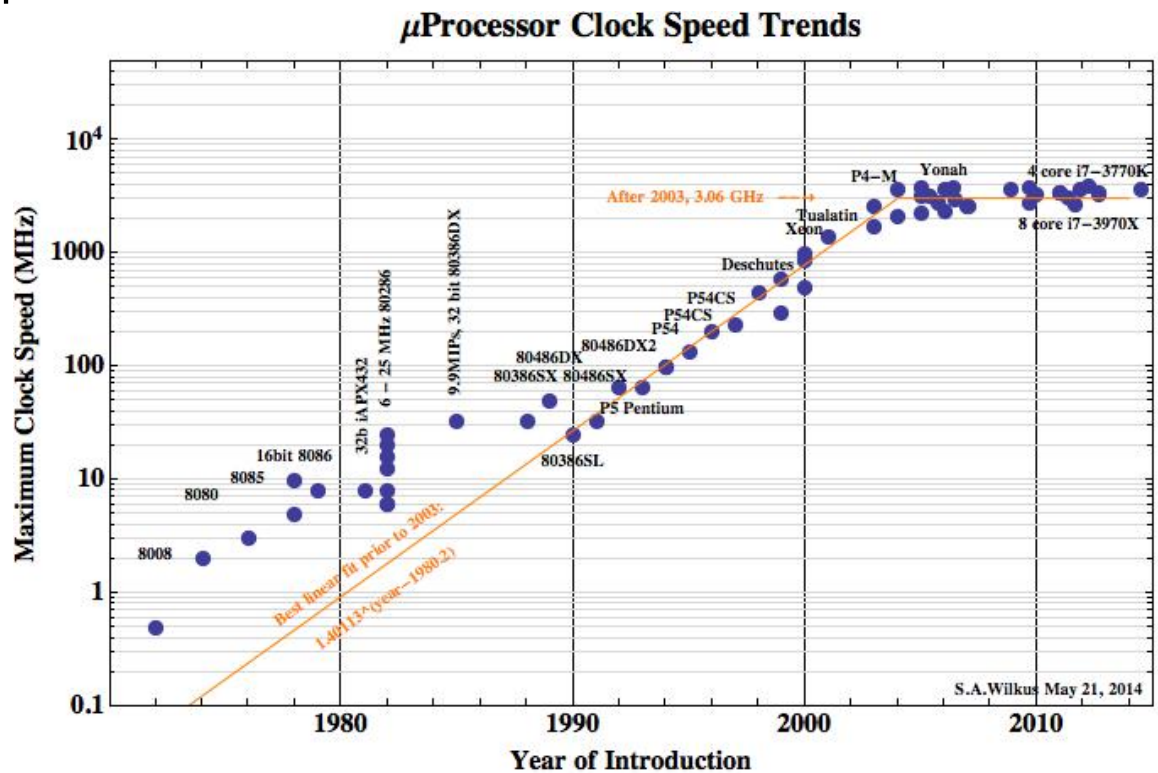
- Pour la performance

Programmation parallèle

Plusieurs codes s'exécutent simultanément pour répondre a un seul besoin.

Intérêt:

- Pour la performance



Programmation parallèle

Plusieurs codes s'exécutent simultanément pour répondre a un seul besoin.

Intérêt:

- Pour la performance
- Pour simplifier l'organisation du code (e.g. serveur web)

Programmation parallèle

Plusieurs codes s'exécutent simultanément pour répondre a un seul besoin.

Intérêt:

- Pour la performance
- Pour simplifier l'organisation du code (e.g. serveur web)

Difficulté:

- Découper le programme en tâches indépendantes
- Assurer la communication entre les tâches
- Assurer la coordination/synchronisation pour:
  - assurer une répartition équitable du travail
  - garantir la cohérence globale des représentations en mémoire

Approche multi-processus

Programmation multi-processus

- Un processus par tâche à accomplir
- Communication: (principalement) explicite
  - Pipe ou fichiers
  - Socket UNIX ou TCP
  - Bibliothèque spécialisées (e.g. MPI)
- Synchronisation: (principalement) via des appels systèmes bloquants

Approche multi-processus

Programmation multi-processus

- Un processus par tâche à accomplir
- Communication: (principalement) explicite
  - Pipe ou fichiers
  - Socket UNIX ou TCP
  - Bibliothèque spécialisées (e.g. MPI)
- Synchronisation: (principalement) via des appels systèmes bloquants

▶ application à **mémoire distribuée**

Approche multi-processus

Programmation multi-processus

- Un processus par tâche à accomplir
- Communication: (principalement) explicite
  - Pipe ou fichiers
  - Socket UNIX ou TCP
  - Bibliothèque spécialisées (e.g. MPI)
- Synchronisation: (principalement) via des appels systèmes bloquants

▶ application à **mémoire distribuée**

Limite:
Communication coûteuse!

▶ Inadaptée aux applications qui nécessitent une coordination intensive entre les différentes tâches

Approche multi-threads

Programmation multi-threads

- Un seul processus, un *thread* par tâche à accomplir
  - thread: morceau de code qui s'exécute à l'intérieur du processus
  - les threads d'un même processus partagent l'espace d'adressage mémoire
- Communication: via la mémoire
- Synchronisation: via des primitives de synchronisation dédiées

  sémaphores, mutex, moniteurs

Approche multi-threads

Programmation multi-threads

- Un seul processus, un *thread* par tâche à accomplir
  - thread: morceau de code qui s'exécute à l'intérieur du processus
  - les threads d'un même processus partagent l'espace d'adressage mémoire
- Communication: via la mémoire
- Synchronisation: via des primitives de synchronisation dédiées

  sémaphores, mutex, moniteurs

▶ application à **mémoire partagée**

Créer un thread avec `pthread_create`

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

void *f(void *arg){
  printf("salut, je suis le thread %u\n", pthread_self());
}
int main(int argc, char **argv){
  pthread_t tid;
  pthread_create(&tid, NULL, f, NULL);
  printf("salut, je suis le thread principal (%u)\n", pthread_self());
  /* attention : ce code est incomplet a ce stade (voir slide 8) */
}
```

```
$ gcc pthread0.c -l phthread
$ ./a.out
```

```
salut, je suis le thread principal
      (3047241472)
salut, je suis le thread 3047245632
```

Exécute la fonction `f` dans un nouveau thread.

Non déterminisme

```
$ ./a.out
salut, je suis le thread 3047241472 # thread 1
salut, je suis le thread principal (3047245632) # thread main
$ ./a.out
salut, je suis le thread principal (3047245632) # thread main
salut, je suis le thread 3047241472 # thread 1
# c'est pas le même ordre!
$ ./a.out
salut, je suis le thread principal 3047245632 # thread main
$ # où est le thread 1 ???
```

Non déterminisme (2)

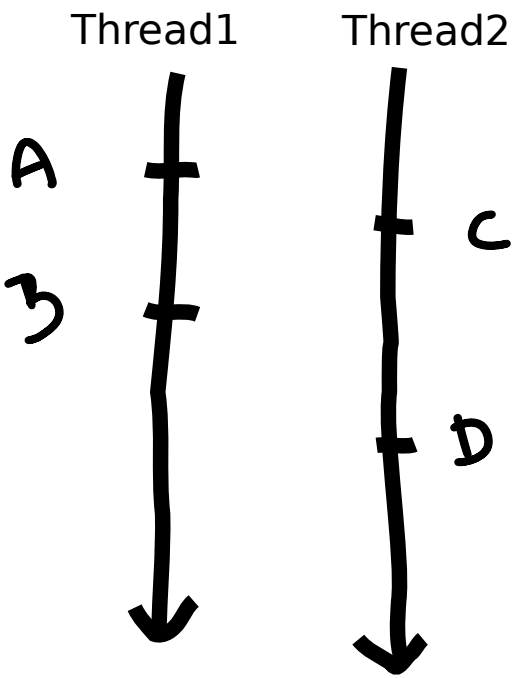
C'est l'ordonnanceur qui pilote l'exécution des différents threads

- on ne peut pas contrôler l'ordre d'exécution des différents threads
- l'ordre peut changer d'une exécution à l'autre
- l'ordonnanceur s'assure que tous les threads ont un accès équitable au CPU

Non déterminisme (2)

C'est l'ordonnanceur qui pilote l'exécution des différents threads

- on ne peut pas contrôler l'ordre d'exécution des différents threads
- l'ordre peut changer d'une exécution à l'autre
- l'ordonnanceur s'assure que tous les threads ont un accès équitable au CPU

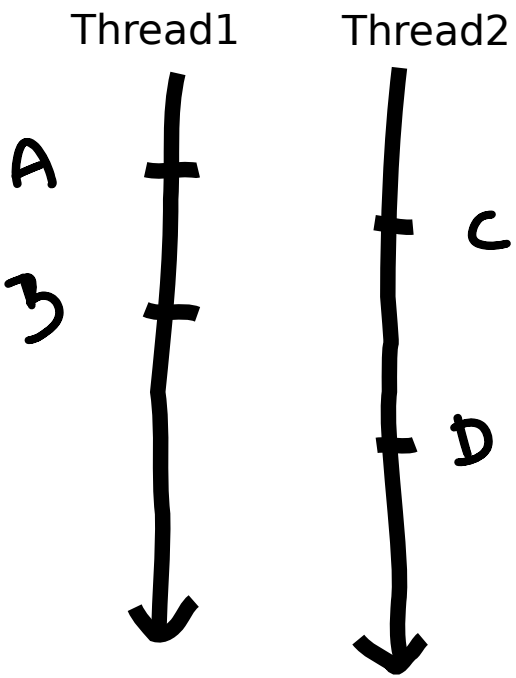


- $A \prec B \rightarrow ?$
- $C \prec D \rightarrow ?$
- $A \prec C \rightarrow ?$
- $B \prec D \rightarrow ?$
- $A \prec D \rightarrow ?$

Non déterminisme (2)

C'est l'ordonnanceur qui pilote l'exécution des différents threads

- on ne peut pas contrôler l'ordre d'exécution des différents threads
- l'ordre peut changer d'une exécution à l'autre
- l'ordonnanceur s'assure que tous les threads ont un accès équitable au CPU

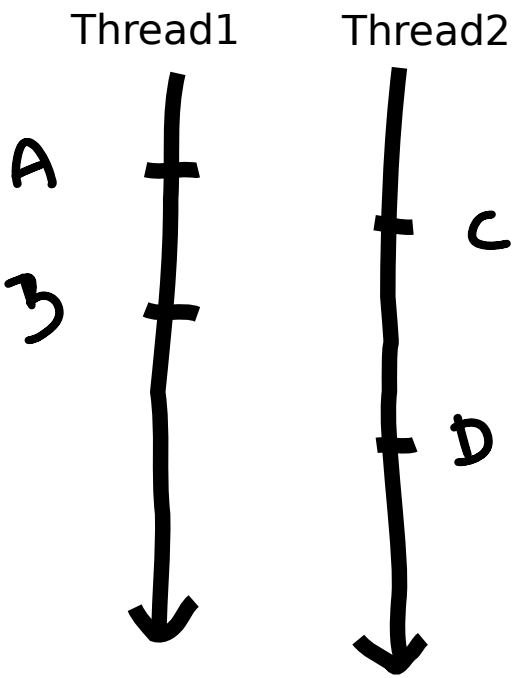


- $A \prec B \rightarrow Vrai$
- $C \prec D \rightarrow Vrai$
- $A \prec C \rightarrow ?$
- $B \prec D \rightarrow ?$
- $A \prec D \rightarrow ?$

Non déterminisme (2)

C'est l'ordonnanceur qui pilote l'exécution des différents threads

- on ne peut pas contrôler l'ordre d'exécution des différents threads
- l'ordre peut changer d'une exécution à l'autre
- l'ordonnanceur s'assure que tous les threads ont un accès équitable au CPU



- $A \prec B \rightarrow Vrai$
- $C \prec D \rightarrow Vrai$
- $A \prec C \rightarrow ?$
- $B \prec D \rightarrow ?$
- $A \prec D \rightarrow ?$

C'est au programmeur de s'assurer que le programme est correct quelque soit l'ordre d'exécution! ▶ primitives de synchronisation

Attendre un thread avec `pthread_join`

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

void *f(void *arg){
  printf("salut, je suis le thread %u\n", pthread_self());
}

int main(int argc, char **argv){
  pthread_t tid;
  pthread_create(&tid, NULL, f, NULL);
  printf("Moi je suis le thread %u\n", pthread_self());
  pthread_join(tid, NULL);
}
```

Attendre un thread avec `pthread_join`

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

void *f(void *arg){
  printf("salut, je suis le thread %u\n", pthread_self());
}

int main(int argc, char **argv){
  pthread_t tid;
  pthread_create(&tid, NULL, f, NULL);
  printf("Moi je suis le thread %u\n", pthread_self());
  pthread_join(tid, NULL);
}
```

- le thread main ne peut plus terminer avant l'autre thread.

Attention!

```c
#define NUM_THREADS 4
void *f(void *a){
    printf("%u démarre\n", pthread_self());
    sleep(1);   // calcul parallèle
    printf("%u termine\n", pthread_self());
}
int main(int argc, char **argv){
  for(int i = 0; i < NUM_THREADS; i++){
    pthread_t tid;
    pthread_create(&tid, NULL, f, NULL);
    pthread_join(tid, NULL);
  }
}
```

```
$ gcc pthread0.c -l phthread
$ ./a.out
```

```
3208664832 démarre
3208664832 termine
3191879424 démarre
3191879424 termine
3200272128 démarre
3200272128 termine
3183486720 démarre
3183486720 termine
```

Attention!

```c
#define NUM_THREADS 4
void *f(void *a){
    printf("%u démarre\n", pthread_self());
    sleep(1);   // calcul parallèle
    printf("%u termine\n", pthread_self());
}
int main(int argc, char **argv){
  pthread_t tid[NUM_THREADS];
  for(int i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], NULL, f, NULL);
  for(int i = 0; i < NUM_THREADS; i++)
     pthread_join(tid[i], NULL);
}
```

```
$ gcc pthread0.c -l phthread
$ ./a.out
```

```
3208664832 démarre
3191879424 démarre
3200272128 démarre
3183486720 démarre
3200272128 termine
3191879424 termine
3208664832 termine
3183486720 termine
```

Passage d'arguments (générique)

```c
void *f(void *arg){
  float *floatarg = (float*)arg; //cast void* -> float*
  printf("Thread %u, argument : %f\n", pthread_self(), *floatarg);
  return NULL;
}

int main(int argc, char **argv){
  pthread_t tid;
  float pi = 3.1415926;
  pthread_create(&tid, NULL, f, (void*)&pi); // cast float* -> void*
  pthread_join(tid, NULL);
}
```

Passage d'arguments (moins propre)

Fréquent mais à éviter

Note: `sizeof(long long int) == sizeof(void*)`

```c
void *f(void *arg){
  long long arg = (long long)arg; //cast void* -> long long
  printf("Thread %u, argument : %lld\n", pthread_self(), arg);
  return NULL;
}

int main(int argc, char **argv){
  pthread_t tid;
  long long int a = 423;
  pthread_create(&tid, NULL, f, (void*)a); // cast long long -> void*
  pthread_join(tid, NULL);
}
```

Valeur de retour

À éviter aussi.

```c
void *f(void *arg){
  float *retval = malloc(sizeof(float));
  *retval = 3.1415926 * 2;
  return (void*)retval ; // cast float* -> void*
}

int main(int argc, char **argv){
  pthread_t tid;
  pthread_create(&tid, NULL, f, NULL);
  float *retvalp;
  pthread_join(tid, (void*)&retvalp); // cast float** -> void*
  printf("Le thread a terminé, valeur de retour: %f\n", *retvalp);
  free(retvalp);
}
```

Variables partagées?

▶ Une variable est “partagée” si son identifiant dans tous les threads référence la même adresse mémoire.

Variables partagées?

▶ Une variable est “partagée” si son identifiant dans tous les threads référence la même adresse mémoire.

```
1
2  int globale;
3
4  void *f(void *arg){
5    int locale = 2;
6    printf("&globale %p\n", &globale);
7    printf("&locale %p\n", &locale);
8    printf("&arg %p\n", &arg);
9    printf("arg %p\n", arg);
10
11 }
12
13 void main(){
14   pthread_t tid1, tid2;
15   int *x = malloc(sizeof(int));
16   pthread_create(&tid1, NULL, f, (void*)x);
17   pthread_create(&tid2, NULL, f, (void*)x);
18   pthread_join(tid1, NULL);
19   pthread_join(tid2, NULL);
20 }
```

Variable partagées? (2)

```
1 void *f(void *arg){
2   int a = 2;
3   printf("&globale %p\n", &globale);
4   printf("&locale %p\n", &locale);
5   printf("&arg %p\n", &arg);
6   printf("arg %p\n", arg);
7   printf("\n");
8 }
```

Thread1

```
1 &globale 0x56496941f040
2 &locale 0x7faa056e8eec
3 &arg 0x7faa056e8ed8
4 arg 0x557185de92a0
```

Thread2

```
1 &globale 0x56496941f040
2 &locale 0x7faa05ee9eec
3 &arg 0x7faa05ee9ed8
4 arg 0x557185de92a0
```

Variable partagées? (2)

```c
void *f(void *arg){
  int a = 2;
  printf("&globale %p\n", &globale);
  printf("&locale %p\n", &locale);
  printf("&arg %p\n", &arg);
  printf("arg %p\n", arg);
  printf("\n");
}
```

Thread1

```
&globale 0x56496941f040
&locale 0x7faa056e8eec
&arg 0x7faa056e8ed8
arg 0x557185de92a0
```

Thread2

```
&globale 0x56496941f040
&locale 0x7faa05ee9eec
&arg 0x7faa05ee9ed8
arg 0x557185de92a0
```

- globale est unique et partagée

Variable partagées? (2)

```c
void *f(void *arg){
  int a = 2;
  printf("&globale %p\n", &globale);
  printf("&locale %p\n", &locale);
  printf("&arg %p\n", &arg);
  printf("arg %p\n", arg);
  printf("\n");
}
```

Thread1

```
&globale 0x56496941f040
&locale 0x7faa056e8eec
&arg 0x7faa056e8ed8
arg 0x557185de92a0
```

Thread2

```
&globale 0x56496941f040
&locale 0x7faa05ee9eec
&arg 0x7faa05ee9ed8
arg 0x557185de92a0
```

- globale est unique et partagée
- autant de variables `locale` que de threads

Variable partagées? (2)

```c
void *f(void *arg){
  int a = 2;
  printf("&globale %p\n", &globale);
  printf("&locale %p\n", &locale);
  printf("&arg %p\n", &arg);
  printf("arg %p\n", arg);
  printf("\n");
}
```

Thread1

```
&globale 0x56496941f040
&locale 0x7faa056e8eec
&arg 0x7faa056e8ed8
arg 0x557185de92a0
```

Thread2

```
&globale 0x56496941f040
&locale 0x7faa05ee9eec
&arg 0x7faa05ee9ed8
arg 0x557185de92a0
```

- globale est unique et partagée
- autant de variables `locale` que de threads
- pareil pour `arg`

Variable partagées? (2)

```c
void *f(void *arg){
  int a = 2;
  printf("&globale %p\n", &globale);
  printf("&locale %p\n", &locale);
  printf("&arg %p\n", &arg);
  printf("arg %p\n", arg);
  printf("\n");
}
```

Thread1

```
&globale 0x56496941f040
&locale 0x7faa056e8eec
&arg 0x7faa056e8ed8
arg 0x557185de92a0
```

Thread2

```
&globale 0x56496941f040
&locale 0x7faa05ee9eec
&arg 0x7faa05ee9ed8
arg 0x557185de92a0
```

- globale est unique et partagée
- autant de variables `locale` que de threads
- pareil pour `arg`
- la région pointée par arg réside dans le tas, et est partagée

Partage de variables

Qu'est ce qu'il se passe ?

- Une pile par thread $\rightarrow$ toutes les variables locales restent locales
  - Il y a autant de variables `locale` et `arg` que de threads
  - Écrire dans la variable `locale`, ne change pas la valeur de `locale` dans un autre thread (heureusement!)

- Il est possible de partager des variables via des pointeurs ou des variables globales

Partage de variables

Qu'est ce qu'il se passe ?

- Une pile par thread $\rightarrow$ toutes les variables locales restent locales
  - Il y a autant de variables `locale` et `arg` que de threads
  - Écrire dans la variable `locale`, ne change pas la valeur de `locale` dans un autre thread (heureusement!)

- Il est possible de partager des variables via des pointeurs ou des variables globales

▶ Exactement la même logique que pour les appels de fonction classiques!!

Variables partagées

L'ordonnanceur (scheduleur) peut interrompre l'exécution d'un thread, et donner la main à un autre thread à tout moment

- Au milieu d'une boucle
- Au milieu d'un test.
- Au milieu d'un affichage.
- Au milieu d'un changement de valeurs...

Variables partagées

L'ordonnanceur (scheduleur) peut interrompre l'exécution d'un thread, et donner la main à un autre thread à tout moment

- Au milieu d'une boucle
- Au milieu d'un test.
- Au milieu d'un affichage.
- Au milieu d'un changement de valeurs...

Conséquences, du point d'une vue d'un thread:

- la valeur d'une variable peut changer d'un moment à l'autre
- un test vrai a un instant t ne l'est plus forcément à t+1
- les variables/objets peuvent être dans des états erronés ou incohérents lorsque plusieurs threads y accèdent simultanément

Variables partagées

L'ordonnanceur (scheduleur) peut interrompre l'exécution d'un thread, et donner la main à un autre thread à tout moment

- Au milieu d'une boucle
- Au milieu d'un test.
- Au milieu d'un affichage.
- Au milieu d'un changement de valeurs...

Conséquences, du point d'une vue d'un thread:

- la valeur d'une variable peut changer d'un moment à l'autre
- un test vrai a un instant t ne l'est plus forcément à t+1
- les variables/objets peuvent être dans des états erronés ou incohérents lorsque plusieurs threads y accèdent simultanément

▶ Il faut:

- faut attendre l'arrivée de certains évènements
- protéger l'accès aux variables partagées

Accès non protégé (1)

```
1  int globale = 0;
2
3  void *f(void *a){
4    for (int i = 0; i < 10; i++)
5      globale++;
6  }
7
8  int main(int argc, char **argv){
9    pthread_t tid1;
10   pthread_create(&tid1, NULL, f, NULL);
11   printf("globale = %d\n", globale);
12   pthread_join(tid1, NULL);
13 }
```

Résultat?

Accès non protégé (1)

```
1  int globale = 0;
2
3  void *f(void *a){
4    for (int i = 0; i < 10; i++)
5      globale++;
6  }
7
8  int main(int argc, char **argv){
9    pthread_t tid1;
10   pthread_create(&tid1, NULL, f, NULL);
11   printf("globale = %d\n", globale);
12   pthread_join(tid1, NULL);
13 }
```

Résultat?

- N'importe quoi entre 0 et 10.

Accès non protégé (2)

```
1  int globale = 0;
2
3  void *f(void *a){
4    for(;;)
5      globale = 1 - globale;
6  }
7
8  void *g(void *a){
9    if(globale == 0)
10     printf("globale = %d\n", globale)
11 }
12
13 int main(int argc, char **argv){
14   pthread_t tid1, tid2;
15   pthread_create(&tid1, NULL, f, NULL);
16   pthread_create(&tid2, NULL, g, NULL);
17   pthread_join(tid1, NULL);
18   pthread_join(tid1, NULL);
19 }
```

Affichages possibles:

Accès non protégé (2)

```
1  int globale = 0;
2
3  void *f(void *a){
4    for(;;)
5      globale = 1 - globale;
6  }
7
8  void *g(void *a){
9    if(globale == 0)
10     printf("globale = %d\n", globale)
11 }
12
13 int main(int argc, char **argv){
14   pthread_t tid1, tid2;
15   pthread_create(&tid1, NULL, f, NULL);
16   pthread_create(&tid2, NULL, g, NULL);
17   pthread_join(tid1, NULL);
18   pthread_join(tid1, NULL);
19 }
```

Affichages possibles:

- `globale = 0`
- `globale = 1`

Accès non protégé (3)

```
1  int globale = 0;
2  void *f(void *a){
3    for (int i = 0; i < 10000; i++)
4      globale++;
5
6  }
7  int main(int argc, char **argv){
8    pthread_t tid1,tid;
9    pthread_create(&tid1, NULL, f, NULL);
10   pthread_create(&tid2, NULL, f, NULL);
11   pthread_join(tid1, NULL);
12   pthread_join(tid2, NULL);
13   printf("globale = %d\n", globale);
14 }
```

Accès non protégé (3)

```
1  int globale = 0;
2  void *f(void *a){
3    for (int i = 0; i < 10000; i++)
4      globale++;
5
6  }
7  int main(int argc, char **argv){
8    pthread_t tid1,tid;
9    pthread_create(&tid1, NULL, f, NULL);
10   pthread_create(&tid2, NULL, f, NULL);
11   pthread_join(tid1, NULL);
12   pthread_join(tid2, NULL);
13   printf("globale = %d\n", globale);
14 }
```

Résultat:

```
1  $ ./a.out
2  globale = 19840
3  $ ./a.out
4  globale = 16294
5  $ ./a.out
6  globale = 16352
7  $ ./a.out
8  globale = 18352
```

Pourquoi?

Code C (exécuté par les 2 threads)

```
1 for(int i = 0; i < 10; i++){
2   globale++;
3 }
```

Code assembleur:

Thread 1

```
1 begin:
2   LOAD @i, R1
3   ADD R1, 1
4   STORE R1, @globale
5   JMP begin
```

Thread 2

```
1 begin:
2   LOAD @i, R1
3   ADD R1, 1
4   STORE R1, @globale
5   JMP begin
```

Pourquoi? (2)

Thread 1

```
1 begin:
2   LOAD @globale, R1
3   #
4   #
5   #
6   ADD R1, 1
7   STORE R1, @globale # globale : 1->1 !!
```

Thread 2

```
1 begin:
2   #
3   LOAD @globale, R1
4   ADD R1, 1
5   STORE R1, @globale
6   #
7   #
```

Pourquoi? (3)

- Un thread peut rendre la main après avoir incrémenté la valeur (insctruction ADD) mais avant de l'avoir enregistré en mémoire (instruction STORE)
- Si l'autre thread prend la main, incrémente la variable et enregistre le résultat, le résultat sera perdu lorsque le premier thread va reprendre la main.
- `i++` n'est pas `une opération atomique` , mais une séquences de 3 instructions (LOAD/ADD/STORE) qui peuvent être interrompues à tout moment.

Valeur finale dans le pire cas?

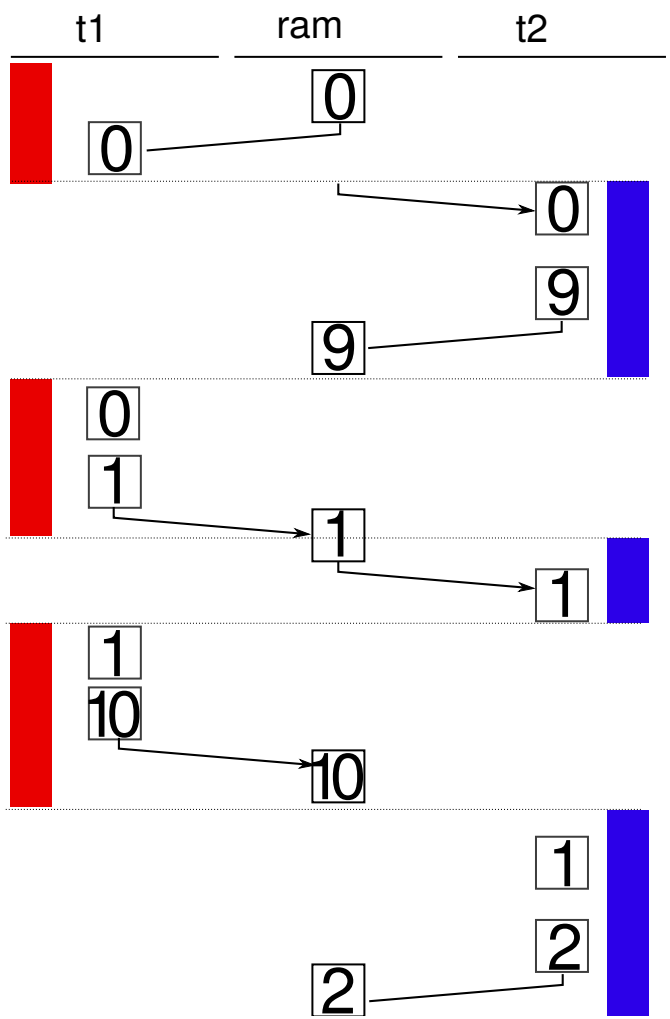
Pourquoi? (3)

- Un thread peut rendre la main après avoir incrémenté la valeur (insctruction ADD) mais avant de l'avoir enregistré en mémoire (instruction STORE)
- Si l'autre thread prend la main, incrémente la variable et enregistre le résultat, le résultat sera perdu lorsque le premier thread va reprendre la main.
- `i++` n'est pas `une opération atomique` , mais une séquences de 3 instructions (LOAD/ADD/STORE) qui peuvent être interrompues à tout moment.

Valeur finale dans le pire cas?

▶ 2!!

Pire cas



Solution

- Rendre l'opération i++ atomique? ▶ pas possible
- Utiliser une *section critique* pour s'assurer qu'un seul thread est en train de modifier la variable à la fois.

Mutex

Mutex = *mutual exclusion*

```
1  int globale = 0;
2  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
3  // ^ mutex partagé par tous les threads
4
5  void *f(void *arg){
6     for(int i = 0; i < 10000; i++){
7       pthread_mutex_lock(&mutex);
8       globale++; // un seul thread a la fois
9       pthread_mutex_unlock(&mutex);
10    }
11 }
```

```
1    int main(int argc, char **argv){
2      pthread_t tid1, tid2;
3      pthread_create(&tid1, NULL, f, NULL);
4      pthread_create(&tid2, NULL, f, NULL);
5      pthread_join(tid1, NULL);
6      pthread_join(tid2, NULL);
7      printf("globale = %d\n", globale);
8   }
```

lock/unlock

`pthread_mutex_lock`

- prend le mutex, s'il est disponible, ou
- bloque le thread si un autre thread détient déjà le mutex

lock/unlock

`pthread_mutex_lock`

- prend le mutex, s'il est disponible, ou
- bloque le thread si un autre thread détient déjà le mutex

`pthread_mutex_unlock`

- libère le mutex
- réveille le(s) autre(s) thread(s) en attente.

Comment implémenter mutex_lock?

`pthread_mutex_lock`

- prend le mutex, s'il est disponible, ou
- bloque le thread si un autre thread détient déjà le mutex

Comment implémenter mutex_lock?

`pthread_mutex_lock`

- prend le mutex, s'il est disponible, ou
- bloque le thread si un autre thread détient déjà le mutex

```
1    mutex_pris = 0;
2    void pthread_mutex_lock(pthread_mutex_t *mutex){
3      while (mutex_pris){
4        sleep();
5      }
6      // ...
7      mutex_pris = 1;
8    }
```

Problème?

Comment implémenter mutex_lock?

`pthread_mutex_lock`

- prend le mutex, s'il est disponible, ou
- bloque le thread si un autre thread détient déjà le mutex

```
1    mutex_pris = 0;
2    void pthread_mutex_lock(pthread_mutex_t *mutex){
3      while (mutex_pris){
4        sleep();
5      }
6      // ...
7      mutex_pris = 1;
8    }
```

Problème?

▶ le test n'est toujours pas atomique!

test & set

Instruction test & set:

- écrit 1 dans une variable
- retourne l'ancienne valeur

▶ 2 opération en une instruction atomique!

test & set

Instruction test & set:

- écrit 1 dans une variable
- retourne l'ancienne valeur

▶ 2 opération en une instruction atomique!

```
1    mutex_pris = 0;
2    void pthread_mutex_lock(pthread_mutex_t *mutex){
3      while ( test_and_set(&mutex_pris) == 1){
4          sleep();
5      }
6      // mutex_pris est passée de 0 à1 une et une seule fois
7    }
```

(attention: le vrai code est plus compliqué!)

Exemple

```
1
2  #define DATA_SIZE 10e6
3  #define NUM_THREADS 10
4  #define SLICE_SIZE 10e6 / NUM_THREADS
5
6  double *data = ...;
7
8  double f(double d){
9     // calcul compliqué
10 }
11
12 void *parallel_map(void *arg){
13   long long id = (long long)arg; // sizeof(long long) == sizeof(void*)
14   int begin = id * SLICE ;
15   int end = (id+1) * SLICE;
16   for(int i = begin; i < end; i++){
17       data[i] = f(data[i]);
18   }
19 }
20
21 int main(int argc, char **argv){
22  // creer NUM_THREADS threads qui exécutent la fonction parallel map
23 }
```

Échange d'informations

- Problème : échanger un résultat intermédiaire entre deux threads
  - Résultat échangé via une variable partagée
  - Utilisation de mutex pour éviter les états incohérents

Suffisant ?

Échange d'informations

- Problème : échanger un résultat intermédiaire entre deux threads
  - Résultat échangé via une variable partagée
  - Utilisation de mutex pour éviter les états incohérents

Suffisant ?

▶ Non, il faut aussi un mecanisme pour qu'un thread puisse attendre des donnees.

Exemple

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

float result = 0;

void *compute_result(void *){
    result = 42;
}

void *print_result(void *){
    printf("result %f\n", result); // affiche?
}


int main(){
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, compute_result, NULL);
  pthread_create(&tid2, NULL, print_result, NULL);

  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
}
```

Affiche ?

Exemple

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

float result = 0;

void *compute_result(void *){
    result = 42;
}

void *print_result(void *){
    printf("result %f\n", result); // affiche?
}


int main(){
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, compute_result, NULL);
  pthread_create(&tid2, NULL, print_result, NULL);

  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
}
```

Affiche ?

- 0 ou 42.

Corrigé?

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


int result_ready = 0;
float result = 0;

void *compute_result(void *){
    result = 42;
    result_ready = 1;
}

void *print_result(void *){
    while( ! reasult_ready );
    printf("result %d\n", result);
}
int main(){
    ....
}
```

Problème?

- Attente active du thread (utilise des ressources)
- `result_ready` peut ne pas avoir été mis à jour en RAM

Moniteur

```
1 pthread_cond_t cond;
2 pthread_mutex_t mutex;
3 pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
4 pthead_cond_signal(pthread_cond_t *cond)
5 pthead_cond_broadcast(pthread_cond_t *cond)
```

Moniteur

```
1 pthread_cond_t cond;
2 pthread_mutex_t mutex;
3 pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
4 pthead_cond_signal(pthread_cond_t *cond)
5 pthead_cond_broadcast(pthread_cond_t *cond)
```

- `pthread_cond_wait()`: libère le mutex, et met le thread en veille
  ▶ permet de mettre un thread en attente, jusqu'à l'arrivée d'un signal
- `pthead_cond_signal()`: réveille un thread en attente sur `cond`
- `pthead_cond_broadcast()`: réveille tous les threads en attente sur `cond`.

Example moniteur

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


int result_ready = 0;
float result = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Thread 1

```c
void *compute_result(void *){
    phtread_mutex_lock(&mutex);
    result = 42;
    result_ready = 1;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}
```

Thread 2

```c
void *print_result(void *){
    pthread_mutex_lock(&mutex);
    while( ! reasult_ready )
       pthread_cond_wait(&cond, &mutex);
    //  condition satisfaite!
    printf("result %d\n", result);
    pthread_mutex_unlock(&mutex);
}
```

Lecteurs / redacteurs

```
1   list_t *list;
2
3   void *t1(void *){
4      begin_read();
5      int val = list_get_first(list);
6      end_read();
7      return val;
8   }
9
10  void *t2(void *arg){
11   int *int_arg = (int *)arg;
12   begin_write();
13   list_add(int_arg);
14   end_write();
15  }
```

- Opération de lecture protégées par `begin_read()` en `end_read()`
- Opération d'écriture protégées par `begin_write()` en `end_write()`

Lecteurs / redacteurs

```
1   list_t *list;
2
3   void *t1(void *){
4      begin_read();
5      int val = list_get_first(list);
6      end_read();
7      return val;
8   }
9
10  void *t2(void *arg){
11   int *int_arg = (int *)arg;
12   begin_write();
13   list_add(int_arg);
14   end_write();
15  }
```

- Opération de lecture protégées par `begin_read()` en `end_read()`
- Opération d'écriture protégées par `begin_write()` en `end_write()`

Problème:

- Comment implémenter `begin_read()`, `begin_write()`, `end_read()`, `end_write()`

Lecteurs / rédacteurs avec mutex

```
1  void begin_read(){
2    pthread_mutex_lock(&mutex);
3  }
4  void end_read(){
5    pthread_mutex_unlock(&mutex);
6  }
7  void begin_write(){
8    pthread_mutex_lock(&mutex);
9  }
10 void end_write(){
11   pthread_mutex_unlock(&mutex); }
```

Problème:

- 1 seul lecteur / rédacteur à la fois
- Pas de parallélisme

Lecteurs / rédacteurs avec mutex

```
1  void begin_read(){
2    pthread_mutex_lock(&mutex);
3  }
4  void end_read(){
5    pthread_mutex_unlock(&mutex);
6  }
7  void begin_write(){
8    pthread_mutex_lock(&mutex);
9  }
10 void end_write(){
11   pthread_mutex_unlock(&mutex); }
```

Problème:

- 1 seul lecteur / rédacteur à la fois
- Pas de parallélisme

Solution:

- Bloquer les lecteurs lorsqu'il existe un rédacteur
- Bloquer les rédacteurs lorsqu'il existe un lecteur ou un redacteur
- Impossible avec des mutex seulement

Lecteurs/redacteurs avec cond. (1)

```
1
2  pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
3  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
4  int num_readers = 0;
5  int num_writers = 0;
6
7  void begin_read(){
8    pthread_mutex_lock(&mutex);
9    while(num_writers > 0)
10       pthread_cond_wait(&cond, &mutex);
11   num_readers++;
12   pthread_mutex_unlock(&mutex);
13 }
14
15 void end_read(){
16   pthread_mutex_lock(&mutex);
17   num_readers--;
18   if(num_readers == 0)
19     pthread_cond_broadcast(&cond);
20   pthread_mutex_unlock(&mutex);
21 }
```

Lecteurs/redacteurs avec cond. (2)

```
1
2  void begin_write(){
3      pthread_mutex_lock(&mutex);
4      while(num_readers > 0 || num_writers > 0)
5          pthread_cond_wait(&cond, &mutex);
6      num_writers++;
7      pthread_mutex_unlock(&mutex);
8  }
9
10 void end_write(){
11     pthread_mutex_lock(&mutex);
12     num_writers--;
13     if(num_writers == 0)
14       pthread_cond_broadcast(&cond, &mutex);
15     pthread_mutex_unlock(&mutex);
16 }
```

Équité de la solution

- Est ce que tous les threads vont avoir accès à la ressource partagée? Non! Si on a un flot continu de lecteurs, les rédacteurs n'auront jamais la main

```
1 Reader 1 enters critical section
2 Writer enters critical section ... on hold
3 Reader 2 enters critical section
4 Reader 1 got value 3
5 Reader 3 enters critical section
6 Reader 2 got value 1
7 Reader 4 enters critical section
8 Reader 3 got value 4
9 ...
```

Équité de la solution

- Est ce que tous les threads vont avoir accès à la ressource partagée? Non! Si on a un flot continu de lecteurs, les rédacteurs n'auront jamais la main

```
1 Reader 1 enters critical section
2 Writer enters critical section ... on hold
3 Reader 2 enters critical section
4 Reader 1 got value 3
5 Reader 3 enters critical section
6 Reader 2 got value 1
7 Reader 4 enters critical section
8 Reader 3 got value 4
9 ...
```

▶ Solution : bloquer les lecteurs qui arrivent **après** la mise en attente du redacteur