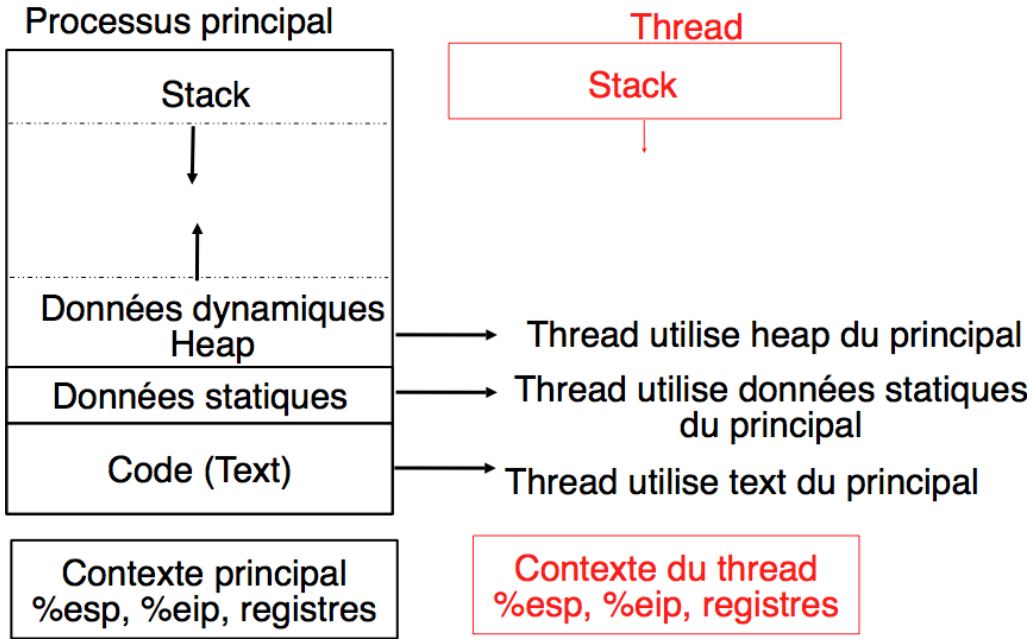
Communication entre threads

Lorsque un programme a été décomposé en plusieurs threads, ceux-ci ne sont en général pas complètement indépendants et ils doivent communiquer entre eux. Cette communication entre threads est un problème complexe comme nous allons le voir. Avant d'aborder ce problème, il est utile de revenir à l'organisation d'un processus et de ses threads en mémoire. La figure ci-dessous illustre schématiquement l'organisation de la mémoire après la création d'un thread POSIX.



Organisation de la mémoire après la création d'un thread POSIX

Le programme principal et le thread qu'il a créé partagent trois zones de la mémoire : le **segment text** qui comprend l'ensemble des instructions qui composent le programme, le **segment de données** qui comprend toutes les données statiques , initialisées ou non (c'est-à-dire les constantes, les variables globales ou encore les chaînes de caractère) et enfin le **heap**. Autant le programme principal que son thread peuvent accéder à n'importe quelle information se trouvant en mémoire dans ces zones. Par contre, le programme principal et le thread qu'il vient de créer ont chacun leur propre contexte et leur propre pile.

La première façon pour un processus de communiquer avec un thread qu'il a lancé est d'utiliser les arguments de la fonction de démarrage du thread et la valeur retournée par le thread que le processus principal peut récupérer via l'appel à **pthread_join(3posix)**. C'est un canal de communication très limité qui ne permet pas d'échange d'information pendant l'exécution du thread.

Il est cependant assez facile pour un processus de partager de l'information avec ses threads ou même de partager de l'information entre plusieurs threads. En effet, tous les threads d'un processus ont accès aux mêmes variables globales et au même **heap**. Il est donc tout à fait possible pour n'importe quel thread de modifier la valeur d'une variable globale. Deux threads qui réalisent un calcul peuvent donc stocker des résultats intermédiaires dans une variable globale ou un tableau global. Il en va de même pour l'utilisation d'une zone de mémoire allouée par **malloc(3)**. Chaque thread qui dispose d'un pointeur vers cette zone mémoire peut en lire le contenu ou en modifier la valeur.

Malheureusement, permettre à tous les threads de lire et d'écrire simultanément en mémoire peut être une source de problèmes. C'est une des difficultés majeures de l'utilisation de threads. Pour s'en convaincre, considérons l'exemple ci-dessous **[1]**.

```
long global=0;
int increment(int i) {
  return i+1;
}
void *func(void * param) {
  for(int j=0;j<1000000;j++) {
    global=increment(global);
 }
  return(NULL);
}
```

Dans cet exemple, la variable `global` est incrémentée `1000000` de fois par la fonction `func`. Après l'exécution de cette fonction, la variable `global` contient la valeur `1000000`. Sur une machine multiprocesseurs, un programmeur pourrait vouloir en accélérer les performances en lançant plusieurs threads qui exécutent chacun la fonction `func`. Cela pourrait se faire en utilisant par exemple la fonction `main` ci-dessous.

```
int main (int argc, char *argv[])  {
  pthread_t thread[NTHREADS];
  int err;
  for(int i=0;i<NTHREADS;i++) {
   err=pthread_create(&(thread[i]),NULL,&func,NULL);
   if(err!=0)
```

```
    error(err,"pthread_create");
  }
  /*...*/
  for(int i=NTHREADS-1;i>=0;i--) {
    err=pthread_join(thread[i],NULL);
    if(err!=0)
      error(err,"pthread_join");
  }
  printf("global: %ld\n",global);
  return(EXIT_SUCCESS);
}
```

Ce programme lance alors 4 threads d'exécution qui incrémentent chacun un million de fois la variable `global`. Celle-ci étant initialisée à `0`, la valeur affichée par **printf(3)** à la fin de l'exécution doit donc être `4000000`. L'exécution du programme ne confirme malheureusement pas cette attente.

```
$ for i in {1..5}; do ./pthread-test;  done
global: 3408577
global: 3175353
global: 1994419
global: 3051040
global: 2118713
```

Non seulement la valeur attendue (`4000000`) n'est pas atteinte, mais en plus la valeur change d'une exécution du programme à la suivante. C'est une illustration du problème majeur de la découpe d'un programme en threads. Pour bien comprendre le problème, il est utile d'analyser en détails les opérations effectuées par deux threads qui exécutent la ligne `global=increment(global);`.

accessible

La variable `global` est stockée dans une zone mémoire qui est accessibles aux deux threads. Appelons-les *T1* et *T2*. L'exécution de cette ligne par un thread nécessite l'exécution de plusieurs instructions en assembleur. Tout d'abord, il faut charger la valeur de la variable `global` depuis la mémoire vers un registre. Ensuite, il faut placer cette valeur sur la pile du thread puis appeler la fonction `increment`. Cette fonction récupère son argument sur la pile du thread, la place dans un registre, incrémente le contenu du registre et sauvegarde le résultat dans le registre `%eax`. Le résultat est retourné dans la fonction `func` et la variable globale peut enfin être mise à jour.

Malheureusement les difficultés surviennent lorsque deux threads exécutent en même temps la ligne `global=increment(global);`. Supposons qu'à cet instant, la valeur de la variable `global` est `1252`. Le premier thread charge une copie de cette variable sur sa pile. Le second fait de même. Les deux threads ont donc chacun passé la valeur `1252` comme argument à la fonction `increment`. Celle-ci s'exécute et retourne la valeur `1253` que chaque thread va récupérer dans `%eax`. Chaque thread va ensuite transférer cette valeur dans la zone mémoire correspondant à la variable `global`. Si les deux threads exécutent l'instruction assembleur correspondante exactement au même moment, les deux écritures en mémoire seront sérialisées par les processeurs sans que l'on ne puisse a priori déterminer quelle écriture se fera en premier **[McKenney2005]**. Alors que les deux threads ont chacun exécuté un appel à la fonction `increment`, la valeur de la variable n'a finalement été incrémentée qu'une seule fois même si cette valeur a été transférée deux fois en mémoire. Ce problème se reproduit fréquemment. C'est pour cette raison que la valeur de la variable `global` n'est pas modifiée comme attendu.

Note

Contrôler la pile d'un thread POSIX

La taille de la pile d'un thread POSIX est l'un des attributs qui peuvent être modifiés lors de l'appel à **pthread_create(3)** pour créer un nouveau thread. Cet attribut peut être fixé en utilisant la fonction **pthread_attr_setstackaddr(3posix)** comme illustré dans l'exemple ci-dessous **[2]** (où `thread_first` est la fonction qui sera appelée à la création du thread). En général, la valeur par défaut choisie par le système suffit, sauf lorsque le programmeur sait qu'un thread devra par exemple allouer un grand tableau auquel il sera le seul à avoir accès. Ce tableau sera alors alloué sur la pile qui devra être suffisamment grande pour le contenir.

```
pthread_t first;
pthread_attr_t attr_first;
size_t stacksize;

int err;

err= pthread_attr_init(&attr_first);
if(err!=0)
  error(err,"pthread_attr_init");

err= pthread_attr_getstacksize(&attr_first,&stacksize);
if(err!=0)
  error(err,"pthread_attr_getstacksize");

printf("Taille par défaut du stack : %ld\n",stacksize);

stacksize=65536;

err= pthread_attr_setstacksize(&attr_first,stacksize);
if(err!=0)
  error(err,"pthread_attr_setstacksize");

err=pthread_create(&first,&attr_first,&thread_first,NULL);
```

```
  if(err!=0)
    error(err,"pthread_create");
```

Ce problème d'accès concurrent à une zone de mémoire par plusieurs threads est un problème majeur dans le développement de programmes découpés en threads, que ceux-ci s'exécutent sur des ordinateurs mono-processeurs ou multiprocesseurs. Dans la littérature, il est connu sous le nom de problème de la **section critique** ou **exclusion mutuelle**. La **section critique** peut être définie comme étant une séquence d'instructions qui ne peuvent *jamais* être exécutées par plusieurs threads simultanément. Dans l'exemple ci-dessus, il s'agit de la ligne `global=increment(global);`. Dans d'autres types de programmes, la section critique peut être beaucoup plus grande et comprendre par exemple la mise à jour d'une base de données. En pratique, on retrouvera une section critique chaque fois que deux threads peuvent modifier ou lire la valeur d'une même zone de la mémoire.

Le fragment de code ci-dessus présente une autre illustration d'une section critique. Dans cet exemple, la fonction `main` (non présentée), créée deux threads. Le premier exécute la fonction `inc` qui incrémente la variable `global`. Le second exécute la fonction `is_even` qui teste la valeur de cette variable et compte le nombre de fois qu'elle est paire. Après la terminaison des deux threads, le programme affiche le contenu des variables `global` et `even`.

```
long global=0;
int even=0;
int odd=0;

void test_even(int i) {
  if((i%2)==0)
    even++;
}

int increment(int i) {
  return i+1;
}

void *inc(void * param) {
  for(int j=0;j<1000000;j++) {
    global=increment(global);
  }
  pthread_exit(NULL);
}

void *is_even(void * param) {
  for(int j=0;j<1000000;j++) {
    test_even(global);
  }
  pthread_exit(NULL);
}
```

L'exécution de ces deux threads donne, sans surprise des résultats qui varient d'une exécution à l'autre.

```
$ for i in {1..5}; do ./pthread-test-if;  done
global: 1000000, even:905140
global: 1000000, even:919756
global: 1000000, even:893058
global: 1000000, even:891266
global: 1000000, even:895043
```

Coordination entre threads

L'utilisation de plusieurs threads dans un programme fonctionnant sur un seul ou plusieurs processeurs nécessite l'utilisation de mécanismes de coordination entre ces threads. Ces mécanismes ont comme objectif d'éviter que deux threads ne puissent modifier ou tester de façon non coordonnée la même zone de la mémoire.

Exclusion mutuelle

Le premier problème important à résoudre lorsque l'on veut coordonner plusieurs threads d'exécution d'un même processus est celui de l'**exclusion mutuelle**. Ce problème a été initialement proposé par Dijkstra en 1965 **[Dijkstra1965]**. Il peut être reformulé de la façon suivante pour un programme décomposé en threads.

Considérons un programme décomposé en N threads d'exécution. Supposons également que chaque thread d'exécution est cyclique, c'est-à-dire qu'il exécute régulièrement le même ensemble d'instructions, sans que la durée de ce cycle ne soit fixée ni identique pour les N threads. Chacun de ces threads peut être décomposé en deux parties distinctes. La première est la partie non-critique. C'est un ensemble d'instructions qui peuvent être exécutées par le thread sans nécessiter la moindre coordination avec un autre thread. Plus précisément, tous les threads peuvent exécuter simultanément leur partie non-critique. La seconde partie du thread est appelée sa **section critique**. Il s'agit d'un ensemble d'instructions qui ne peuvent être exécutées que par un seul et unique thread. Le problème de l'**exclusion mutuelle** est de trouver un algorithme qui permet de garantir qu'il n'y aura jamais deux threads qui simultanément exécuteront les instructions de leur section critique.

Cela revient à dire qu'il n'y aura pas de violation de la section critique. Une telle violation pourrait avoir des conséquences catastrophiques sur l'exécution du programme. Cette propriété est une propriété de **sûreté** (**safety** en anglais). Dans un programme découpé en threads, une propriété de **sûreté** est une propriété qui

doit être vérifiée à tout instant de l'exécution du programme.

En outre, une solution au problème de l'**exclusion mutuelle** doit satisfaire les contraintes suivantes **[Dijkstra1965]** :

a. La solution doit considérer tous les threads de la même façon et ne peut faire aucune hypothèse sur la priorité relative des différents threads.
b. La solution ne peut faire aucune hypothèse sur la vitesse relative ou absolue d'exécution des différents threads. Elle doit rester valide quelle que soit la vitesse d'exécution non nulle de chaque thread.
c. La solution doit permettre à un thread de s'arrêter en dehors de sa section critique sans que cela n'invalide la contrainte d'exclusion mutuelle
d. Si un ou plusieurs threads souhaitent entamer leur section critique, aucun de ces threads ne doit pouvoir être empêché indéfiniment d'accéder à sa section critique.

La troisième contrainte implique que la terminaison ou le crash d'un des threads ne doit pas avoir d'impact sur le fonctionnement du programme complet et le respect de la contrainte d'exclusion mutuelle pour la section critique.

La quatrième contrainte est un peu plus subtile mais tout aussi importante. Toute solution au problème de l'exclusion mutuelle contient nécessairement un mécanisme qui permet de bloquer l'exécution d'un thread pendant qu'un autre exécute sa section critique. Il est important qu'un thread puisse accéder à sa section critique si il le souhaite. C'est un exemple de propriété de **vivacité** (**liveness** en anglais). Une propriété de **vivacité** est une propriété qui ne peut pas être éternellement invalidée. Dans notre exemple, un thread ne pourra jamais être empêché d'accéder à sa section critique.

Exclusion mutuelle sur monoprocesseurs

Même si les threads sont très utiles dans des ordinateurs multiprocesseurs, ils ont été inventés et utilisés d'abord sur des processeurs capables d'exécuter un seul thread d'exécution à la fois. Sur un tel processeur, les threads d'exécution sont entrelacés plutôt que d'être exécutés réellement simultanément. Cet entrelacement est réalisé par le système d'exploitation.

Les systèmes d'exploitation de la famille Unix permettent d'exécuter plusieurs programmes *en même temps* sur un ordinateur, même si il est équipé d'un processeur qui n'est capable que d'exécuter un thread à la fois. Cette fonctionnalité est souvent appelée le **multitâche** (ou **multitasking** en anglais). Cette exécution simultanée de plusieurs programmes n'est en pratique qu'une illusion puisque le processeur ne peut qu'exécuter qu'une séquence d'instructions à la fois.

Pour donner cette illusion, un système d'exploitation multitâche tel que Unix exécute régulièrement des changements de contexte entre threads. Le **contexte** d'un thread est composé de l'ensemble des contenus des registres qui sont nécessaires à son exécution (y compris le contenu des registres spéciaux tels que `%esp`, `%eip` ou `%eflags`). Ces registres définissent l'état du thread du point de vue du processeur. Pour passer de l'exécution du thread *T1* à l'exécution du thread *T2*, le système d'exploitation doit initier un **changement de contexte**. Pour réaliser ce changement de contexte, le système d'exploitation initie le transfert du contenu des registres utilisés par le thread *T1* vers une zone mémoire lui appartenant. Il transfère ensuite depuis une autre zone mémoire lui appartenant le contexte du thread *T2*. Si ce changement de contexte est effectué fréquemment, il peut donner l'illusion à l'utilisateur que plusieurs threads ou programmes s'exécutent simultanément.

Sur un système Unix, il y a deux types d'événements qui peuvent provoquer un changement de contexte.

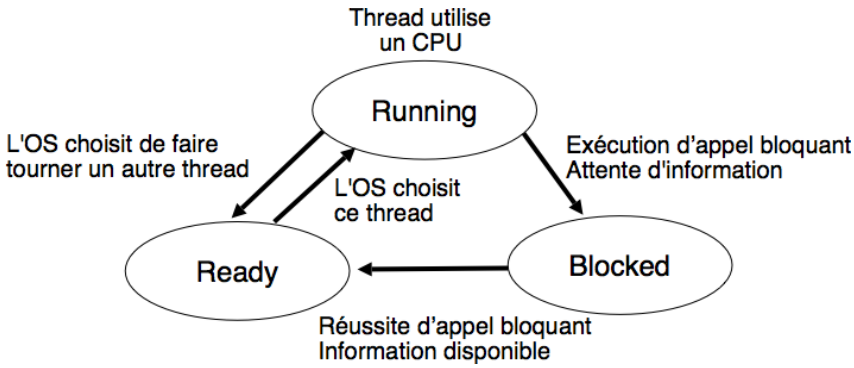
1. Le hardware génère une **interruption**
2. Un thread exécute un **appel système bloquant**

Ces deux types d'événements sont fréquents et il est important de bien comprendre comment ils sont traités par le système d'exploitation.

Une **interruption** est un signal électronique qui est généré par un dispositif connecté au microprocesseur. De nombreux dispositifs d'entrées-sorties comme les cartes réseau ou les contrôleurs de disque peuvent générer une interruption lorsqu'une information a été lue ou reçue et doit être traitée par le processeur. En outre, chaque ordinateur dispose d'une horloge temps réel qui génère des interruptions à une fréquence déterminée par le système d'exploitation mais qui est généralement comprise entre quelques dizaines et quelques milliers de *Hz*. Ces interruptions nécessitent un traitement rapide de la part du système d'exploitation. Pour cela, le processeur vérifie, à la fin de l'exécution de *chaque* instruction si un signal d'interruption **[3]** est présent. Si c'est le cas, le processeur sauvegarde en mémoire le contexte du thread en cours d'exécution et lance une routine de traitement d'interruption faisant partie du système d'exploitation. Cette routine analyse l'interruption présente et lance les fonctions du système d'exploitation nécessaires à son traitement. Dans le cas d'une lecture sur disque, par exemple, la routine de traitement d'interruption permettra d'aller chercher la donnée lue sur le contrôleur de disques.

Le deuxième type d'événement est l'exécution d'un appel système bloquant. Un thread exécute un **appel système** chaque fois qu'il doit interagir avec le système d'exploitation. Ces appels peuvent être exécutés directement ou via une fonction de la librairie **[4]**. Il existe deux types d'appels systèmes : les appels bloquants et les appels non-bloquants. Un appel système non-bloquant est un appel système que le système d'exploitation peut exécuter immédiatement. Cet appel retourne généralement une valeur qui fait partie du système d'exploitation lui-même. L'appel **gettimeofday(2)** qui permet de récupérer l'heure actuelle est un exemple d'appel non-bloquant. Un appel système bloquant est un appel système dont le résultat ne peut pas toujours être fourni immédiatement. Les lectures d'information en provenance de l'entrée standard (et donc généralement du clavier) sont un bon exemple simple d'appel système bloquant. Considérons un thread qui exécute la fonction de la librairie **getchar(3)** qui retourne un caractère lu sur **stdin**. Cette fonction utilise l'appel système **read(2)** pour lire un caractère sur **stdin**. Bien entendu, le système d'exploitation est obligé d'attendre que l'utilisateur presse une touche sur le clavier pour pouvoir fournir le résultat de cet appel système à l'utilisateur. Pendant tout le temps qui s'écoule entre l'exécution de **getchar(3)** et la pression d'une touche sur le clavier, le thread est bloqué par le système d'exploitation. Plus aucune instruction du thread n'est exécutée tant que la fonction **getchar(3)** ne s'est pas terminée et le contexte du thread est mis en attente dans une zone mémoire gérée par le système d'exploitation. Il sera redémarré automatiquement par le système d'exploitation lorsque la donnée attendue sera disponible.

Ces interactions entre les threads et le système d'exploitation sont importantes. Pour bien les comprendre, il est utile de noter qu'un thread peut se trouver dans trois états différents du point de vue de son interaction avec le système d'exploitation. Ces trois états sont illustrés dans la figure ci-dessous.



Etats d'un thread d'exécution

Lorsqu'un thread est créé avec la fonction **pthread_create(3)**, il est placé dans l'état *Ready*. Dans cet état, les instructions du thread ne s'exécutent sur aucun processeur mais il est prêt à être exécuté dès qu'un processeur se libérera. Le deuxième état pour un thread est l'état *Running*. Dans cet état, le thread est exécuté sur un des processeurs du système. Le dernier état est l'état *Blocked*. Un thread est dans l'état *Blocked* lorsqu'il a exécuté un appel système bloquant et que le système d'exploitation attend l'information permettant de retourner le résultat de l'appel système. Pendant ce temps, les instructions du thread ne s'exécutent sur aucun processeur.

Les transitions entre les différents états d'un thread sont gérées par le système d'exploitation. Lorsque plusieurs threads d'exécution sont simultanément actifs, le système d'exploitation doit arbitrer les demandes d'utilisation du CPU de chaque thread. Cet arbitrage est réalisé par l'ordonnanceur (ou **scheduler** en anglais). Le **scheduler** est un ensemble d'algorithmes qui sont utilisés par le système d'exploitation pour sélectionner le ou les threads qui peuvent utiliser un processeur à un moment donné. Il y a souvent plus de threads qui sont dans l'état *Ready* que de processeurs disponibles et le scheduler doit déterminer quels sont les threads à exécuter.

Une description détaillée du fonctionnement d'un scheduler relève plutôt d'un cours sur les systèmes d'exploitation que d'un premier cours sur le langage C, mais il est important de connaître les principes de base de fonctionnement de quelques schedulers.

Un premier scheduler simple est le **round-robin**. Ce scheduler maintient en permanence une liste circulaire de l'ensemble des threads qui se trouvent dans l'état *Ready* et un pointeur vers l'élément courant de cette liste. Lorsqu'un processeur devient disponible, le scheduler sélectionne le thread référencé par ce pointeur. Ce thread passe dans l'état *Running*, est retiré de la liste et le pointeur est déplacé vers l'élément suivant dans la liste. Pour éviter qu'un thread ne puisse monopoliser éternellement un processeur, un scheduler **round-robin** limite généralement le temps qu'un thread peut passer dans l'état *Running*. Lorsqu'un thread a utilisé un processeur pendant ce temps, le scheduler vérifie si il y a un thread en attente dans l'état *Ready*. Si c'est le cas, le scheduler force un changement de contexte, place le thread courant dans l'état *Ready* et le remet dans la liste circulaire tout en permettant à un nouveau thread de passer dans l'état *Running* pour s'exécuter. Lorsqu'un thread revient dans l'état *Ready*, soit parce qu'il vient d'être créé ou parce qu'il vient de quitter l'état *Blocked*, il est placé dans la liste afin de pouvoir être sélectionné par le scheduler. Un scheduler **round-robin** est équitable. Avec un tel scheduler, si N threads sont actifs en permanence, chacun recevra $\frac{1}{N}$ de temps CPU disponible.

Un second type de scheduler simple est le scheduler à priorités. Une priorité est associée à chaque thread. Lorsque le scheduler doit sélectionner un thread à exécuter, il commence d'abord par parcourir les threads ayant une haute priorité. En pratique, un scheduler à priorité maintiendra une liste circulaire pour chaque niveau de priorité. Lorsque le scheduler est appelé, il sélectionnera toujours le thread ayant la plus haute priorité et se trouvant dans l'état *Ready*. Si plusieurs threads ont le même niveau de priorité, un scheduler de type **round-robin** peut être utilisé dans chaque niveau de priorité. Sous Unix, le scheduler utilise un scheduler à priorité avec un round-robin à chaque niveau de priorité, mais la priorité varie dynamiquement en fonction du temps de façon à favoriser les threads interactifs.

Connaissant ces bases du fonctionnement des schedulers, il est utile d'analyser en détails quels sont les événements qui peuvent provoquer des transitions entre les états d'un thread. Certains de ces événements sont provoqués par le thread lui-même. C'est le cas de la transition entre l'état *Running* et l'état *Blocked*. Elle se produit lorsque le thread exécute un **appel système bloquant**. Dans ce cas, un processeur redevient disponible et le scheduler peut sélectionner un autre thread pour s'exécuter sur ce processeur. La transition entre l'état *Blocked* et l'état *Running* dépend elle du système d'exploitation, directement lorsque le thread a été bloqué par le système d'exploitation ou indirectement lorsque le système d'exploitation attend une information venant d'un dispositif d'entrées-sorties. Les transitions entre les états *Running* et *Ready* dépendent elles entièrement du système d'exploitation. Elles se produisent lors de l'exécution du scheduler. Celui-ci est exécuté lorsque certaines interruptions surviennent. Il est exécuté à chaque interruption d'horloge. Cela permet de garantir l'exécution régulière du scheduler même si les seuls threads actifs exécutent une boucle infinie telle que `while(true);`. A l'occasion de cette interruption, le scheduler mesure le temps d'exécution de chaque thread et si un thread a consommé beaucoup de temps CPU alors que d'autres threads sont dans l'état *Ready*, le scheduler forcera un changement de contexte pour permettre à un autre thread de s'exécuter. De la même façon, une interruption relative à un dispositif d'entrées-sorties peut faire transiter un thread de l'état *Blocked* à l'état *Ready*. Cette modification du nombre de threads dans l'état *Ready* peut forcer le scheduler à devoir effectuer un changement de contexte pour permettre à ce thread de poursuivre son exécution. Sous Unix, le scheduler utilise des niveaux de priorité qui varient en fonction des opérations d'entrées sorties effectuées. Cela a comme conséquence de favoriser les threads qui effectuent des opérations d'entrées sorties par rapport aux threads qui effectuent uniquement du calcul.

Note

Un thread peut demander de passer la main.

Dans la plupart de nos exemples, les threads cherchent en permanence à exécuter des instructions. Ce n'est pas nécessairement le cas de tous les threads d'un programme. Par exemple, une application de calcul scientifique pourrait être découpée en $N+1$ threads. Les N premiers threads

réalisent le calcul tandis que le dernier calcule des statistiques. Ce dernier thread ne doit pas consommer de ressources et être en compétition pour le processeur avec les autres threads. La librairie thread POSIX contient la fonction **pthread_yield(3)** qui peut être utilisée par un thread pour indiquer explicitement qu'il peut être remplacé par un autre thread. Si un thread ne doit s'exécuter qu'à intervalles réguliers, il est préférable d'utiliser des appels à **sleep(3)** ou **usleep(3)**. Ces fonctions de la librairie permettent de demander au système d'exploitation de bloquer le thread pendant un temps au moins égal à l'argument de la fonction.

Sur une machine monoprocesseur, tous les threads s'exécutent sur le même processeur. Une violation de section critique peut se produire lorsque le scheduler décide de réaliser un changement de contexte alors qu'un thread se trouve dans sa section critique. Si la section critique d'un thread ne contient ni d'appel système bloquant ni d'appel à **pthread_yield(3)**, ce changement de contexte ne pourra se produire que si une interruption survient. Une solution pour résoudre le problème de l'exclusion mutuelle sur un ordinateur monoprocesseur pourrait donc être la suivante :

```
disable_interrupts();
// début section critique
// ...
// fin section critique
enable_interrupts();
```

Cette solution est possible, mais elle souffre de plusieurs inconvénients majeurs. Tout d'abord, une désactivation des interruptions perturbe le fonctionnement du système puisque sans interruptions, la plupart des opérations d'entrées-sorties et l'horloge sont inutilisables. Une telle désactivation ne peut être que très courte, par exemple pour modifier une ou quelques variables en mémoire. Ensuite, la désactivation des interruptions, comme d'autres opérations relatives au fonctionnement du matériel, est une opération privilégiée sur un microprocesseur. Elle ne peut être réalisée que par le système d'exploitation. Il faudrait donc imaginer un appel système qui permettrait à un thread de demander au système d'exploitation de désactiver les interruptions. Si un tel appel système existait, le premier programme qui exécuterait `disable_interrupts();` sans le faire suivre de `enable_interrupts();` quelques instants après pourrait rendre la machine complètement inutilisable puisque sans interruption plus aucune opération d'entrée-sortie n'est possible et qu'en plus le scheduler ne peut plus être activé par l'interruption d'horloge. Pour toutes ces raisons, la désactivation des interruptions n'est pas un mécanisme utilisable par les threads pour résoudre le problème de l'exclusion mutuelle **[5]**.

Coordination par Mutex

Le premier mécanisme de coordination entre threads dans la librairie POSIX sont les **mutex**. Un **mutex** (abréviation de *mutual exclusion*) est une structure de données qui permet de contrôler l'accès à une ressource. Un **mutex** qui contrôle une ressource peut se trouver dans deux états :

- *libre* (ou *unlocked* en anglais). Cet état indique que la ressource est libre et peut être utilisée sans risquer de provoquer une violation d'exclusion mutuelle.
- *réservée* (ou *locked* en anglais). Cet état indique que la ressource associée est actuellement utilisée et qu'elle ne peut pas être utilisée par un autre thread.

Un **mutex** est toujours associé à une ressource. Cette ressource peut être une variable globale comme dans les exemples précédents, mais cela peut aussi être une structure de données plus complexe, une base de données, un fichier, ... Un mutex s'utilise par l'intermédiaire de deux fonctions de base. La fonction *lock* permet à un thread d'acquérir l'usage exclusif d'une ressource. Si la ressource est libre, elle est marquée comme réservée et le thread y accède directement. Si la ressource est occupée, le thread est bloqué par le système d'exploitation jusqu'à ce qu'elle ne devienne libre. A ce moment, le thread pourra poursuivre son exécution et utilisera la ressource avec la certitude qu'aucun autre thread ne pourra faire de même. Lorsque le thread a terminé d'utiliser la ressource associée au mutex, il appelle la fonction *unlock*. Cette fonction vérifie d'abord si un ou plusieurs autres threads sont en attente pour cette ressource (c'est-à-dire qu'ils ont appelé la fonction *lock* mais celle-ci n'a pas encore réussi). Si c'est le cas, un (et un seul) thread est choisi parmi les threads en attente et celui-ci accède à la ressource. Il est important de noter qu'un programme ne peut faire aucune hypothèse sur l'ordre dans lequel les threads qui sont en attente sur un **mutex** pourront accéder à la ressource partagée. Le programme doit être conçu en faisant l'hypothèse que si plusieurs threads sont bloqués sur un appel à *lock* pour un mutex, le thread qui sera libéré est choisi aléatoirement.

interne

Sans entrer dans des détails qui relèvent du fonctionnement internes des systèmes d'exploitation, on peut schématiquement représenter un **mutex** comme étant une structure de données qui contient deux informations :

- la valeur actuelle du **mutex** (*locked* ou *unlocked*)
- une queue contenant l'ensemble des threads qui sont bloqués en attente du mutex

Schématiquement, l'implémentation des fonctions `lock` et `unlock` peut être représentée par le code ci-dessous.

```
lock(mutex m) {
 if(m.val==unlocked)
 {
   m.val=locked;
 }
 else
 {
   // Place this thread in m.queue;
   // This thread is blocked;
 }
}
```

La

Le fonction `lock` vérifie si le **mutex** est libre. Dans ce cas, le **mutex** est marqué comme réservé et la fonction `lock` réussi. Sinon, le thread qui a appelé la fonction `lock` est placé dans la queue associée au **mutex** et passe dans l'état *Blocked* jusqu'à ce qu'un autre thread ne libère le mutex.

```
unlock(mutex m) {
 if(m.queue is empty)
 {
   m.val=unlocked;
 }
 else
 {
  // Remove one thread(T) from m.queue;
  // Mark Thread(T) as ready to run;
 }
}
```

La fonction `unlock` vérifie d'abord l'état de la queue associée au **mutex**. Si la queue est vide, cela indique qu'aucun thread n'est en attente. Dans ce cas, la valeur du **mutex** est mise à *unlocked* et la fonction se termine. Sinon, un des threads en attente dans la queue associée au **mutex** est choisi et marqué comme prêt à s'exécuter. Cela indique implicitement que l'appel à `lock` fait par ce thread réussi et qu'il peut accéder à la ressource.

Le code présenté ci-dessous n'est qu'une illustration du fonctionnement des opérations `lock` et `unlock`. Pour que ces opérations fonctionnent correctement, il faut bien entendu que les modifications aux valeurs du **mutex** et à la queue qui y est associée se fassent en garantissant qu'un seul thread exécute l'une de ces opérations sur un **mutex** à un instant donné. En pratique, les implémentations de `lock` et `unlock` utilisent des instructions atomiques telles que celles qui ont été présentées dans la section précédente pour garantir cette propriété.

Les **mutex** sont fréquemment utilisés pour protéger l'accès à une zone de mémoire partagée. Ainsi, si la variable globale *g* est utilisée en écriture et en lecture par deux threads, celle-ci devra être protégée par un **mutex**. Toute modification de cette variable devra être entourée par des appels à *lock* et *unlock*.

En C, cela se fait en utilisant les fonctions **pthread_mutex_lock(3posix)** et **pthread_mutex_unlock(3posix)**. Un **mutex** POSIX est représenté par une structure de données de type *pthread_mutex_t* qui est définie dans le fichier **pthread.h**. Avant d'être utilisé, un **mutex** doit être initialisé via la fonction **pthread_mutex_init(3posix)** et lorsqu'il n'est plus nécessaire, les ressources qui lui sont associées doivent être libérées avec la fonction **pthread_mutex_destroy(3posix)**.

L'exemple ci-dessous reprend le programme dans lequel une variable globale est incrémentée par plusieurs threads.

```
#include <pthread.h>
#define NTHREADS 4

long global=0;
pthread_mutex_t mutex_global;

int increment(int i) {
  return i+1;
}

void *func(void * param) {
  int err;
  for(int j=0;j<1000000;j++) {
    err=pthread_mutex_lock(&mutex_global);
    if(err!=0)
      error(err,"pthread_mutex_lock");
    global=increment(global);
    err=pthread_mutex_unlock(&mutex_global);
    if(err!=0)
      error(err,"pthread_mutex_unlock");
  }
  return(NULL);
}

int main (int argc, char *argv[])  {
  pthread_t thread[NTHREADS];
  int err;

  err=pthread_mutex_init( &mutex_global, NULL);
  if(err!=0)
      error(err,"pthread_mutex_init");

  for(int i=0;i<NTHREADS;i++) {
    err=pthread_create(&(thread[i]),NULL,&func,NULL);
    if(err!=0)
      error(err,"pthread_create");
  }
  for(int i=0; i<1000000000;i++) { /*...*/ }

  for(int i=NTHREADS-1;i>=0;i--) {
    err=pthread_join(thread[i],NULL);
    if(err!=0)
      error(err,"pthread_join");
  }

  err=pthread_mutex_destroy(&mutex_global);
  if(err!=0)
```

```
    error(err,"pthread_mutex_destroy");

  printf("global: %ld\n",global);

  return(EXIT_SUCCESS);
}
```

Il est utile de regarder un peu plus en détails les différentes fonctions utilisées par ce programme. Tout d'abord, la ressource partagée est ici la variable `global`. Dans l'ensemble du programme, l'accès à cette variable est protégé par le **mutex** `mutex_global`. Celui-ci est représenté par une structure de données de type `pthread_mutex_t`.

Avant de pouvoir utiliser un **mutex**, il est nécessaire de l'initialiser. Cette initialisation est effectuée par la fonction **pthread_mutex_init(3posix)** qui prend deux arguments **[6]**. Le premier est un pointeur vers une structure `pthread_mutex_t` et le second un pointeur vers une structure `pthread_mutexattr_t` contenant les attributs de ce **mutex**. Tout comme lors de la création d'un thread, ces attributs permettent de spécifier des paramètres à la création du **mutex**. Ces attributs peuvent être manipulés en utilisant les fonctions **pthread_mutexattr_gettype(3posix)** et **pthread_mutexattr_settype(3posix)**. Dans le cadre de ces notes, nous utiliserons exclusivement les attributs par défaut et créerons toujours un **mutex** en passant `NULL` comme second argument à la fonction **pthread_mutex_init(3posix)**.

Lorsqu'un **mutex** POSIX est initialisé, la ressource qui lui est associée est considérée comme libre. L'accès à la ressource doit se faire en précédant tout accès à la ressource par un appel à la fonction **pthread_mutex_lock(3posix)**. En fonction des attributs spécifiés à la création du **mutex**, il peut y avoir de très rares cas où la fonction retourne une valeur non nulle. Dans ce cas, le type d'erreur est indiqué via **errno**. Lorsque le thread n'a plus besoin de la ressource protégée par le mutex, il doit appeler la fonction **pthread_mutex_unlock(3posix)** pour libérer la ressource protégée.

pthread_mutex_lock(3posix) et **pthread_mutex_unlock(3posix)** sont toujours utilisés en couple. **pthread_mutex_lock(3posix)** doit toujours précéder l'accès à la ressource partagée et **pthread_mutex_unlock(3posix)** doit être appelé dès que l'accès exclusif à la ressource partagée n'est plus nécessaire.

L'utilisation des mutex permet de résoudre correctement le problème de l'exclusion mutuelle. Pour s'en convaincre, considérons le programme ci-dessus et les threads qui exécutent la fonction `func`. Celle-ci peut être résumée par les trois lignes suivantes :

```
pthread_mutex_lock(&mutex_global);
global=increment(global);
pthread_mutex_unlock(&mutex_global);
```

Pour montrer que cette solution répond bien au problème de l'exclusion mutuelle, il faut montrer qu'elle respecte la propriété de sûreté et la propriété de vivacité. Pour la propriété de sûreté, c'est par construction des **mutex** et parce que chaque thread exécute **pthread_mutex_lock(3posix)** avant d'entrer en section critique et **pthread_mutex_unlock(3posix)** dès qu'il en sort. Considérons le cas de deux threads qui sont en concurrence pour accéder à cette section critique. Le premier exécute **pthread_mutex_lock(3posix)**. Il accède à sa section critique. A partir de cet instant, le second thread sera bloqué dès qu'il exécute l'appel à **pthread_mutex_lock(3posix)**. Il restera bloqué dans l'exécution de cette fonction jusqu'à ce que le premier thread sorte de sa section critique et exécute **pthread_mutex_unlock(3posix)**. A ce moment, le premier thread n'est plus dans sa section critique et le système peut laisser le second y entrer en terminant l'exécution de l'appel à **pthread_mutex_lock(3posix)**. Si un troisième thread essaye à ce moment d'entrer dans la section critique, il sera bloqué sur son appel à **pthread_mutex_lock(3posix)**.

Pour montrer que la propriété de vivacité est bien respectée, il faut montrer qu'un thread ne sera pas empêché éternellement d'entrer dans sa section critique. Un thread peut être empêché d'entrer dans sa section critique en étant bloqué sur l'appel à **pthread_mutex_lock(3posix)**. Comme chaque thread exécute **pthread_mutex_unlock(3posix)** dès qu'il sort de sa section critique, le thread en attente finira par être exécuté. Pour qu'un thread utilisant le code ci-dessus ne puisse jamais entrer en section critique, il faudrait qu'il y ait en permanence plusieurs threads en attente sur **pthread_mutex_unlock(3posix)** et que notre thread ne soit jamais sélectionné par le système lorsque le thread précédent termine sa section critique.

Footnotes

[1] Le programme complet est accessible via **`/Threads/S5-src/pthread-test.c`**

[2] Le programme complet est accessible via **`/Threads/S6-src/pthread.c`**

[3] De nombreux processeurs supportent plusieurs signaux d'interruption différents. Dans le cadre de ce cours, nous nous limiterons à l'utilisation d'un seul signal de ce type.

[4] Les appels systèmes sont décrits dans la section `2` des pages de manuel tandis que la section `3` décrit les fonctions de la librairie.

[5] Certains systèmes d'exploitation utilisent une désactivation parfois partielle des interruptions pour résoudre des problèmes d'exclusion mutuelle qui portent sur quelques instructions à l'intérieur du système d'exploitation lui-même. Il faut cependant noter qu'une désactivation des interruptions peut être particulièrement coûteuse en termes de performances dans un environnement multiprocesseurs.

[6] Linux supporte également la macro `PTHREAD_MUTEX_INITIALIZER` qui permet d'initialiser directement un `pthread_mutex_t` déclaré comme variable globale. Dans cet exemple, la déclaration aurait été : `pthread_mutex_t global_mutex=PTHREAD_MUTEX_INITIALIZER;` et l'appel à **pthread_mutex_init(3posix)** aurait été inutile. Comme il s'agit d'une extension spécifique à Linux, il est préférable de ne pas l'utiliser pour garantir la portabilité du code.

[7] Le programme complet est **`/Threads/S6-src/pthread-philo.c`**