Les sémaphores

Le problème de la coordination entre threads est un problème majeur. Outre les **mutex** que nous avons présenté, d'autres solutions à ce problème ont été développées. Historiquement, une des premières propositions de coordination sont les sémaphores **[Dijkstra1965b]**. Un **sémaphore** est une structure de données qui est maintenue par le système d'exploitation et contient :

- un entier qui stocke la valeur, positive ou nulle, du sémaphore.
- une queue qui contient les pointeurs vers les threads qui sont bloqués en attente sur ce sémaphore.

Tout comme pour les **mutex**, la queue associée à un sémaphore permet de bloquer les threads qui sont en attente d'une modification de la valeur du sémaphore.

Une implémentation des sémaphores se compose en général de quatre fonctions :

- une fonction d'initialisation qui permet de créer le sémaphore et de lui attribuer une valeur initiale nulle ou positive.
- une fonction permettant de détruire un sémaphore et de libérer les ressources qui lui sont associées.
- une fonction `post` qui est utilisée par les threads pour modifier la valeur du sémaphore. S'il n'y a pas de thread en attente dans la queue associée au sémaphore, sa valeur est incrémentée d'une unité. Sinon, un des threads en attente est libéré et passe à l'état *Ready*.
- une fonction `wait` qui est utilisée par les threads pour tester la valeur d'un sémaphore. Si la valeur du sémaphore est positive, elle est décrémentée d'une unité et la fonction réussit. Si le sémaphore a une valeur nulle, le thread est bloqué jusqu'à ce qu'un autre thread le débloque en appelant la fonction `post`.

Les sémaphores sont utilisés pour résoudre de nombreux problèmes de coordination **[Downey2008]**. Comme ils permettent de stocker une valeur entière, ils sont plus flexibles que les **mutex** qui sont utiles surtout pour les problèmes d'exclusion mutuelle.

Sémaphores POSIX

La librairie POSIX comprend une implémentation des sémaphores **[1]** qui expose plusieurs fonctions aux utilisateurs. La page de manuel **sem_overview(7)** présente de façon sommaire les fonctions de la librairie relatives aux sémaphores. Les quatre principales sont les suivantes :

```
#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_post(sem_t *sem);
```

Le fichier **semaphore.h** contient les différentes définitions de structures qui sont nécessaires au bon fonctionnement des sémaphores ainsi que les signatures des fonctions de cette API. Un sémaphore est représenté par une structure de données de type `sem_t`. Toutes les fonctions de manipulation des sémaphores prennent comme argument un pointeur vers le sémaphore concerné.

Pour pouvoir utiliser un sémaphore, il faut d'abord l'initialiser. Cela se fait en utilisant la fonction **sem_init(3)** qui prend comme argument un pointeur vers le sémaphore à initialiser. Nous n'utiliserons pas le second argument dans ce chapitre. Le troisième argument est la valeur initiale, positive ou nulle, du sémaphore.

La fonction **sem_destroy(3)** permet de libérer un sémaphore qui a été initialisé avec **sem_init(3)**. Les sémaphores consomment des ressources qui peuvent être limitées dans certains environnements. Il est important de détruire proprement les sémaphores dès qu'ils ne sont plus nécessaires.

Les deux principales fonctions de manipulation des sémaphores sont **sem_wait(3)** et **sem_post(3)**. Certains auteurs utilisent `down` ou `P` à la place de **sem_wait(3)** et `up` ou `V` à la place de **sem_post(3)** **[Downey2008]**. Schématiquement, l'opération `sem_wait` peut s'implémenter en utilisant le pseudo-code suivant :

```
int sem_wait(semaphore *s)
{
 s->val=s->val-1;
 if(s->val<0)
 {
  // Place this thread in s.queue;
  // This thread is blocked;
 }
}
```

La fonction **sem_post(3)** quant à elle peut schématiquement s'implémenter comme suit :

```
int sem_post(semaphore *s)
{
  s->val=s->val+1;
  if(s.val<=0)
  {
    // Remove one thread(T) from s.queue;
    // Mark Thread(T) as ready to run;
  }
}
```

Ces deux opérations sont bien entendu des opérations qui ne peuvent s'exécuter simultanément. Leur implémentation réelle comprend des sections critiques qui doivent être construites avec soin. Le pseudo-code ci-dessus ignore ces sections critiques. Des détails complémentaires sur l'implémentation des sémaphores peuvent être obtenus dans livre sur les systèmes d'exploitation **[Stallings2011]** **[Tanenbaum+2009]** .

La meilleure façon de comprendre leur utilisation est d'analyser des problèmes classiques de coordination qui peuvent être résolus en utilisant des sémaphores.

Exclusion mutuelle

Les sémaphores permettent de résoudre de nombreux problèmes classiques. Le premier est celui de l'exclusion mutuelle. Lorsqu'il est initialisé à `1`, un sémaphore peut être utilisé de la même façon qu'un **mutex**. En utilisant des sémaphores, une exclusion mutuelle peut être protégée comme suit :

```
#include <semaphore.h>

//...

sem_t semaphore;

sem_init(&semaphore, 0, 1);

sem_wait(&semaphore);
// section critique
sem_post(&semaphore);

sem_destroy(&semaphore);
```

Les sémaphores peuvent être utilisés pour d'autres types de synchronisation. Par exemple, considérons une application découpée en threads dans laquelle la fonction `after` ne peut jamais être exécutée avant la fin de l'exécution de la fonction `before`. Ce problème de coordination peut facilement être résolu en utilisant un sémaphore qui est initialisé à la valeur `0`. La fonction `after` doit démarrer par un appel à **sem_wait(3)** sur ce sémaphore tandis que la fonction `before` doit se terminer par un appel à la fonction **sem_post(3)** sur ce sémaphore. De cette façon, si le thread qui exécute la fonction `after` est trop rapide, il sera bloqué sur l'appel à **sem_wait(3)**. S'il arrive à cette fonction après la fin de la fonction `before` dans l'autre thread, il pourra passer sans être bloqué. Le programme ci-dessous illustre cette utilisation des sémaphores POSIX.

```
#define NTHREADS 2
sem_t semaphore;

void *before(void * param) {
  // do something
  for(int j=0;j<1000000;j++) {
  }
  sem_post(&semaphore);
  return(NULL);
}

void *after(void * param) {
  sem_wait(&semaphore);
  // do something
  for(int j=0;j<1000000;j++) {
  }
  return(NULL);
}

int main (int argc, char *argv[])  {
  pthread_t thread[NTHREADS];
  void * (* func[])(void *)={before, after};
  int err;

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

  for(int i=0;i<NTHREADS;i++) {
    err=pthread_join(thread[i],NULL);
    if(err!=0) {
      error(err,"pthread_join");
    }
  }
  sem_destroy(&semaphore);
  if(err!=0) {
      error(err,"sem_destroy");
  }
```

```
  return(EXIT_SUCCESS);
}
```

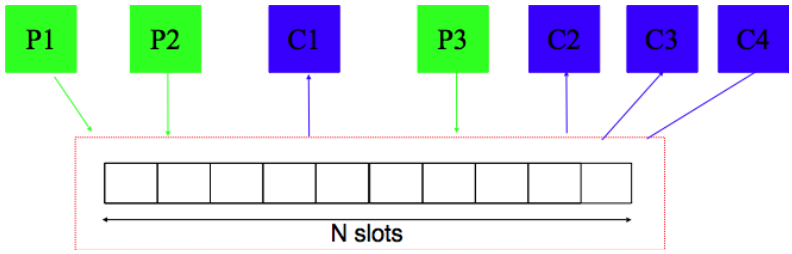
Si un sémaphore initialisé à la valeur `1` est généralement utilisé comme un **mutex**, il y a une différence importante entre les implémentations des sémaphores et des **mutex**. Un sémaphore est conçu pour être manipulé par différents threads et il est fort possible qu'un thread exécute **sem_wait(3)** et qu'un autre exécute **sem_post(3)**. Pour les mutex, certaines implémentations supposent que le même thread exécute **pthread_mutex_lock(3posix)** et **pthread_mutex_unlock(3posix)**. Lorsque ces opérations doivent être effectuées dans des threads différents, il est préférable d'utiliser des sémaphores à la place de mutex.

Problème des producteurs-consommateurs

Le problème des producteurs-consommateurs est un problème extrêmement fréquent et important dans les applications découpées en plusieurs threads. Il est courant de structurer une telle application, notamment si elle réalise de longs calculs, en deux types de threads :

- les *producteurs* : Ce sont des threads qui produisent des données et placent le résultat de leurs calculs dans une zone mémoire accessible aux consommateurs.
- les *consommateurs* : Ce sont des threads qui utilisent les valeurs calculées par les producteurs.

Ces deux types de threads communiquent en utilisant un buffer qui a une capacité limitée à *N* places comme illustré dans la figure ci-dessous.



Problème des producteurs-consommateurs

La difficulté du problème est de trouver une solution qui permet aux producteurs et aux consommateurs d'avancer à leur rythme sans que les producteurs ne bloquent inutilement les consommateurs et inversement. Le nombre de producteurs et de consommateurs ne doit pas nécessairement être connu à l'avance et ne doit pas être fixe. Un producteur peut arrêter de produire à n'importe quel moment.

Le buffer étant partagé entre les producteurs et les consommateurs, il doit nécessairement être protégé par un **mutex**. Les producteurs doivent pouvoir ajouter de l'information dans le buffer partagé tant qu'il y a au moins une place de libre dans le buffer. Un producteur ne doit être bloqué que si tout le buffer est rempli. Inversement, les consommateurs doivent être bloqués uniquement si le buffer est entièrement vide. Dès qu'une donnée est ajoutée dans le buffer, un consommateur doit être réveillé pour traiter cette donnée.

Ce problème peut être résolu en utilisant deux sémaphores et un mutex. L'accès au buffer, que ce soit par les consommateurs ou les producteurs est une section critique. Cet accès doit donc être protégé par l'utilisation d'un mutex. Quant aux sémaphores, le premier, baptisé `empty` dans l'exemple ci-dessous, sert à compter le nombre de places qui sont vides dans le buffer partagé. Ce sémaphore doit être initialisé à la taille du buffer puisque celui-ci est initialement vide. Le second sémaphore est baptisé `full` dans le pseudo-code ci-dessous. Sa valeur représente le nombre de places du buffer qui sont occupées. Il doit être initialisé à la valeur `0`.

```
// Initialisation
#define N 10 // places dans le buffer
pthread_mutex_t mutex;
sem_t empty;
sem_t full;

pthread_mutex_init(&mutex, NULL);
sem_init(&empty, 0 , N);  // buffer vide
sem_init(&full, 0 , 0);   // buffer vide
```

Le fonctionnement général d'un producteur est le suivant. Tout d'abord, le producteur est mis en attente sur le sémaphore `empty`. Il ne pourra passer que si il y a au moins une place du buffer qui est vide. Lorsque la ligne `sem_wait(&empty);` réussit, le producteur s'approprie le `mutex` et modifie le buffer de façon à insérer l'élément produit (dans ce cas un entier). Il libère ensuite le `mutex` pour sortir de sa section critique.

```
// Producteur
void producer(void)
{
  int item;
  while(true)
  {
    item=produce(item);
    sem_wait(&empty); // attente d'une place libre
    pthread_mutex_lock(&mutex);
     // section critique
     insert_item();
    pthread_mutex_unlock(&mutex);
```

```
    sem_post(&full); // il y a une place remplie en plus
  }
}
```

Le consommateur quant à lui essaie d'abord de prendre le sémaphore `full`. Si celui-ci est positif, cela indique la présence d'au moins un élément dans le buffer partagé. Ensuite, il entre dans la section critique protégée par le `mutex` et récupère la donnée se trouvant dans le buffer. Puis, il incrémente la valeur du sémaphore `empty` de façon à indiquer à un producteur qu'une nouvelle place est disponible dans le buffer.

```
// Consommateur
void consumer(void)
{
 int item;
 while(true)
 {
   sem_wait(&full); // attente d'une place remplie
   pthread_mutex_lock(&mutex);
    // section critique
    item=remove(item);
   pthread_mutex_unlock(&mutex);
   sem_post(&empty); // il y a une place libre en plus
 }
}
```

De nombreux programmes découpés en threads fonctionnent avec un ensemble de producteurs et un ensemble de consommateurs.

Compléments sur les threads POSIX

Il existe différentes implémentations des threads POSIX. Les mécanismes de coordination utilisables varient parfois d'une implémentation à l'autre. Dans les sections précédentes, nous nous sommes focalisés sur les fonctions principales qui sont en général bien implémentées. Une discussion plus détaillée des fonctions implémentées sous Linux peut se trouver dans **[Kerrisk2010]**. **[Gove2011]** présente de façon détaillée les mécanismes de coordination utilisables sous Linux, Windows et Oracle Solaris. **[StevensRago2008]** comprend également une description des threads POSIX mais présente des exemples sur des versions plus anciennes de Linux, FreeBSD, Solaris et MacOS.

Il reste cependant quelques concepts qu'il est utile de connaître lorsque l'on développe des programmes découpés en threads en langage C.

Variables spécifiques à un thread

Dans un programme C séquentiel, on doit souvent combiner les variables globales, les variables locales et les arguments de fonctions. Lorsque le programme est découpé en threads, les variables globales restent utilisables, mais il faut faire attention aux problèmes d'accès concurrent. En pratique, il est parfois utile de pouvoir disposer dans chaque thread de variables qui tout en étant accessibles depuis toutes les fonctions du thread ne sont pas accessibles aux autres threads. Il y a différentes solutions pour résoudre ce problème.

Une première solution serait d'utiliser une zone mémoire qui est spécifique au thread et d'y placer par exemple une structure contenant toutes les variables auxquelles on souhaite pouvoir accéder depuis toutes les fonctions du thread. Cette zone mémoire pourrait être créée avant l'appel à **pthread_create(3)** et un pointeur vers cette zone pourrait être passé comme argument à la fonction qui démarre le thread. Malheureusement l'argument qui est passé à cette fonction n'est pas équivalent à une variable globale et n'est pas accessible à toutes les fonctions du thread.

Une deuxième solution serait d'avoir un tableau global qui contiendrait des pointeurs vers des zones de mémoires qui ont été allouées pour chaque thread. Chaque thread pourrait alors accéder à ce tableau sur base de son identifiant. Cette solution pourrait fonctionner si le nombre de threads est fixe et que les identifiants de threads sont des entiers croissants. Malheureusement la librairie threads POSIX ne fournit pas de tels identifiants croissants. Officiellement, la fonction **pthread_self(3)** retourne un identifiant unique d'un thread qui a été créé. Malheureusement cet identifiant est de type `pthread_t` et ne peut pas être utilisé comme index dans un tableau. Sous Linux, l'appel système non-standard **gettid(2)** retourne l'identifiant du thread, mais il ne peut pas non plus être utilisé comme index dans un tableau.

Pour résoudre ce problème, deux solutions sont possibles. La première combine une extension au langage C qui est supportée par **gcc(1)** avec la librairie threads POSIX. Il s'agit du qualificatif `__thread` qui peut être utilisé avant une déclaration de variable. Lorsqu'il est utilisé dans la déclaration d'une variable globale, il indique au compilateur et à la libraire POSIX qu'une copie de cette variable doit être créée pour chaque thread. Cette variable est initialisée au démarrage du thread et est utilisable uniquement à l'intérieur de ce thread. Le programme ci-dessous illustre cette utilisation du qualificatif `__thread`.

```
#define LOOP 1000000
#define NTHREADS 4

__thread int count=0;
int global_count=0;

void *f( void* param) {
  for(int i=0;i<LOOP;i++) {
    count++;
    global_count=global_count-1;
  }
  printf("Valeurs : count=%d, global_count=%d\n",count, global_count);
  return(NULL);
```

```
}

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

  for(int i=0;i<NTHREADS;i++) {
    count=i;   // local au thread du programme principal
    err=pthread_create(&(threads[i]),NULL,&f,NULL);
    if(err!=0)
      error(err,"pthread_create");
  }

  for(int i=0;i<NTHREADS;i++) {
    err=pthread_join(threads[i],NULL);
    if(err!=0)
      error(err,"pthread_create");
  }

  return(EXIT_SUCCESS);
}
```

Lors de son exécution, ce programme affiche la sortie suivante sur **stdout**. Cette sortie illustre bien que les variables dont la déclaration est précédée du qualificatif `__thread` sont utilisables uniquement à l'intérieur d'un thread.

```
Valeurs : count=1000000, global_count=-870754
Valeurs : count=1000000, global_count=-880737
Valeurs : count=1000000, global_count=-916383
Valeurs : count=1000000, global_count=-923423
```

La seconde solution proposée par la librairie POSIX est plus complexe. Elle nécessite l'utilisation des fonctions **pthread_key_create(3posix)**, **pthread_setspecific(3posix)**, **pthread_getspecific(3posix)** et **pthread_key_delete(3posix)**. Cette API est malheureusement plus difficile à utiliser que le qualificatif `__thread`, mais elle illustre ce qu'il se passe en pratique lorsque ce qualificatif est utilisé.

Pour avoir une variable accessible depuis toutes les fonctions d'un thread, il faut tout d'abord créer une clé qui identifie cette variable. Cette clé est de type `pthread_key_t` et c'est l'adresse de cette structure en mémoire qui est utilisée comme identifiant pour la variable spécifique à chaque thread. Cette clé ne doit être créée qu'une seule fois. Cela peut se faire dans le programme qui lance les threads ou alors dans le premier thread lancé en utilisant la fonction **pthread_once(3posix)**. Une clé est créée grâce à la fonction **pthread_key_create(3posix)**. Cette fonction prend deux arguments. Le premier est un pointeur vers une structure de type `pthread_key_t`. Le second est la fonction optionnelle à appeler lorsque le thread utilisant la clé se termine.

Il faut noter que la fonction **pthread_key_create(3posix)** associe en pratique le pointeur `NULL` à la clé qui a été créée dans chaque thread. Le thread qui veut utiliser la variable correspondant à cette clé doit réserver la zone mémoire correspondante. Cela se fait en général en utilisant **malloc(3)** puis en appelant la fonction **pthread_setspecific(3posix)**. Celle-ci prend deux arguments. Le premier est une clé de type `pthread_key_t` qui a été préalablement créée. Le second est un pointeur (de type `void *`) vers la zone mémoire correspondant à la variable spécifique. Une fois que le lien entre la clé et le pointeur a été fait, la fonction **pthread_getspecific(3posix)** peut être utilisée pour récupérer le pointeur depuis n'importe quelle fonction du thread. L'implémentation des fonctions **pthread_setspecific(3posix)** et **pthread_getspecific(3posix)** garantit que chaque thread aura sa variable qui lui est propre.

L'exemple ci-dessous illustre l'utilisation de cette API. Elle est nettement plus lourde à utiliser que le qualificatif `__thread`. Dans ce code, chaque thread démarre par la fonction `f`. Celle-ci crée une variable spécifique de type `int` qui joue le même rôle que la variable `__thread int count;` dans l'exemple précédent. La fonction `g` qui est appelée sans argument peut accéder à la zone mémoire créée en appelant `pthread_getspecific(count)`. Elle peut ensuite exécuter ses calculs en utilisant le pointeur `count_ptr`. Avant de se terminer, la fonction `f` libère la zone mémoire qui avait été allouée par **malloc(3)**. Une alternative à l'appel explicite à **free(3)** aurait été de passer `free` comme second argument à **pthread_key_create(3posix)** lors de la création de la clé `count`. En effet, ce second argument est la fonction à appeler à la fin du thread pour libérer la mémoire correspondant à cette clé.

```
#define LOOP 1000000
#define NTHREADS 4

pthread_key_t count;
int global_count=0;

void g(void ) {
  void * data=pthread_getspecific(count);
  if(data==NULL)
    error(-1,"pthread_getspecific");
  int *count_ptr=(int *)data;
  for(int i=0;i<LOOP;i++) {
    *count_ptr=*(count_ptr)+1;
    global_count=global_count-1;
  }
}

void *f( void* param) {
  int err;
  int *int_ptr=malloc(sizeof(int));
  *int_ptr=0;
  err=pthread_setspecific(count, (void *)int_ptr);
```

```
  if(err!=0)
    error(err,"pthread_setspecific");
  g();
  printf("Valeurs : count=%d, global_count=%d\n",*int_ptr, global_count);
  free(int_ptr);
  return(NULL);
}

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

  err=pthread_key_create(&(count),NULL);
  if(err!=0)
    error(err,"pthread_key_create");

  for(int i=0;i<NTHREADS;i++) {
    err=pthread_create(&(threads[i]),NULL,&f,NULL);
    if(err!=0)
      error(err,"pthread_create");
  }

  for(int i=0;i<NTHREADS;i++) {
    err=pthread_join(threads[i],NULL);
    if(err!=0)
      error(err,"pthread_create");
  }
  err=pthread_key_delete(count);
  if(err!=0)
    error(err,"pthread_key_delete");

  return(EXIT_SUCCESS);
}
```

En pratique, on préférera évidemment d'utiliser le qualificatif `__thread` plutôt que d'utiliser une API explicite lorsque c'est possible. Cependant, il ne faut pas oublier que lorsque ce qualificatif est utilisé, le compilateur doit introduire dans le programme du code permettant de faire le même genre d'opérations que les fonctions explicites de la librairie.

Fonctions `thread-safe`

Dans un programme séquentiel, il n'y a qu'un thread d'exécution et de nombreux programmeurs, y compris ceux qui ont développé la librairie standard, utilisent cette hypothèse lors de l'écriture de fonctions. Lorsqu'un programme est découpé en threads, chaque fonction peut être appelée par plusieurs threads simultanément. Cette exécution simultanée d'une fonction peut poser des difficultés notamment lorsque la fonction utilise des variables globales ou des variables statiques.

Pour comprendre le problème, il est intéressant de comparer plusieurs implémentations d'une fonction simple. Considérons le problème de déterminer l'élément maximum d'une structure de données contenant des entiers. Si la structure de données est un tableau, une solution simple est de le parcourir entièrement pour déterminer l'élément maximum. C'est ce que fait la fonction `max_vector` dans le programme ci-dessous. Dans un programme purement séquentiel dans lequel le tableau peut être modifié de temps en temps, parcourir tout le tableau pour déterminer son maximum n'est pas nécessairement la solution la plus efficace. Une alternative est de mettre à jour la valeur du maximum chaque fois qu'un élément du tableau est modifié. Les fonctions `max_global` et `max_static` sont deux solutions possibles. Chacune de ces fonctions doit être appelée chaque fois qu'un élément du tableau est modifié. `max_global` stocke dans une variable globale la valeur actuelle du maximum du tableau et met à jour cette valeur à chaque appel. La fonction `max_static` fait de même en utilisant une variable statique. Ces deux solutions sont équivalentes et elles pourraient très bien être intégrées à une librairie utilisée par de nombreux programmes.

```
#include <stdint.h>
#define SIZE 10000

int g_max=INT32_MIN;
int v[SIZE];

int max_vector(int n, int *v) {
  int max=INT32_MIN;
  for(int i=0;i<n;i++) {
    if(v[i]>max)
      max=v[i];
  }
  return max;
}

int max_global(int *v) {
    if (*v>g_max) {
      g_max=*v;
    }
    return(g_max);
}

int max_static(int *v){
```

```
  static int s_max=INT32_MIN;
  if (*v>s_max) {
    s_max=*v;
  }
  return(s_max);
}
```

Considérons maintenant un programme découpé en plusieurs threads qui chacun maintient un tableau d'entiers dont il faut connaître le maximum. Ces tableaux d'entiers sont distincts et ne sont pas partagés entre les threads. La fonction `max_vector` peut être utilisée par chaque thread pour déterminer le maximum du tableau. Par contre, les fonctions `max_global` et `max_static` ne peuvent pas être utilisées. En effet, chacune de ces fonctions maintient *un* état (dans ce cas le maximum calculé) alors qu'elle peut être appelée par différents threads qui auraient chacun besoin d'un état qui leur est propre. Pour que ces fonctions soient utilisables, il faudrait que les variables `s_max` et `g_max` soient spécifiques à chaque thread.

En pratique, ce problème de l'accès concurrent à des fonctions se pose pour de nombreuses fonctions et notamment celles de la librairie standard. Lorsque l'on développe une fonction qui peut être réutilisée, il est important de s'assurer que cette fonction peut être exécutée par plusieurs threads simultanément sans que cela ne pose de problèmes à l'exécution.

Ce problème affecte certaines fonctions de la librairie standard et plusieurs d'entre elles ont dû être modifiées pour pouvoir supporter les threads. A titre d'exemple, considérons la fonction **strerror(3)**. Cette fonction prend comme argument le numéro de l'erreur et retourne une chaîne de caractères décrivant cette erreur. Cette fonction ne peut pas être utilisée telle quelle par des threads qui pourraient l'appeler simultanément. Pour s'en convaincre, regardons une version simplifiée d'une implémentation de cette fonction **[3]**. Cette fonction utilise le tableau **sys_errlist(3)** qui contient les messages d'erreur associés aux principaux codes numériques d'erreur. Lorsque l'erreur est une erreur standard, tout se passe bien et la fonction retourne simplement un pointeur vers l'entrée du tableau `sys_errlist` correspondante. Par contre, si le code d'erreur n'est pas connu, un message est généré dans le tableau `buf[32]` qui est déclaré de façon statique. Si plusieurs threads exécutent `strerror`, ce sera le même tableau qui sera utilisé dans les différents threads. On pourrait remplacer le tableau statique par une allocation de zone mémoire faite via **malloc(3)**, mais alors la zone mémoire créée risque de ne jamais être libérée par **free(3)** car l'utilisateur de **strerror(3)** ne doit pas libérer le pointeur qu'il a reçu, ce qui pose d'autres problèmes en pratique.

```
char * strerror (int errnoval)
{
 char * msg;
 static char buf[32];
 if ((errnoval < 0) || (errnoval >= sys_nerr))
   { // Out of range, just return NULL
     msg = NULL;
   }
 else if ((sys_errlist == NULL) || (sys_errlist[errnoval] == NULL))
   { // In range, but no sys_errlist or no entry at this index.
     sprintf (buf, "Error %d", errnoval);
     msg = buf;
   }
 else
   { // In range, and a valid message.  Just return the message.
     msg = (char *) sys_errlist[errnoval];
   }
 return (msg);
}
```

La fonction **strerror_r(3)** évite ce problème de tableau statique en utilisant trois arguments : le code d'erreur, un pointeur `char *` vers la zone devant stocker le message d'erreur et la taille de cette zone. Cela permet à **strerror_r(3)** d'utiliser une zone mémoire qui lui est passée par le thread qu'il appelle et garantit que chaque thread disposera de son message d'erreur. Voici une implémentation possible de **strerror_r(3)** **[4]**.

```
strerror_r(int num, char *buf, size_t buflen)
{
  #define UPREFIX "Unknown error: %u"
  unsigned int errnum = num;
  int retval = 0;
  size_t slen;
  if (errnum < (unsigned int) sys_nerr) {
     slen = strlcpy(buf, sys_errlist[errnum], buflen);
  } else {
     slen = snprintf(buf, buflen, UPREFIX, errnum);
     retval = EINVAL;
  }
  if (slen >= buflen)
       retval = ERANGE;
  return retval;
}
```

Lorsque l'on intègre des fonctions provenant de la librairie standard ou d'une autre librairie dans un programme découpé en threads, il est important de vérifier que les fonctions utilisées sont bien **thread-safe**. La page de manuel **pthreads(7)** liste les fonctions qui ne sont pas **thread-safe** dans la librairie standard.

Footnotes

[1] Les systèmes Unix supportent également des sémaphores dits *System V* du nom de la version de Unix dans laquelle ils ont été introduits. Dans ces notes, nous nous focalisons sur les sémaphores POSIX qui ont une API un peu plus simple que les ~~es~~ sémaphores *System V*. Les principales fonctions pour les sémaphores *System V* sont **semget(3posix)**, **semctl(3posix)** et **semop(3posix)**.

[2] Les premiers compilateurs C permettaient au programmeur de donner des indications au compilateur en faisant précéder les déclarations de certaines variables avec le qualificatif `register` **[KernighanRitchie1998]**. Ce qualificatif indiquait que la variable était utilisée fréquemment et que le compilateur devrait en placer le contenu dans un registre. Les compilateurs actuels sont nettement plus performants et ils sont capables de détecter quelles sont les variables qu'il faut placer dans un registre. Il est inutile de chercher à influencer le compilateur en utilisant le qualificatif `register`. Les compilateurs actuels, dont **gcc(1)** supportent de nombreuses **options** permettant d'optimiser les performances des programmes compilés. Certaines ont comme objectif d'accélérer l'exécution du programme, d'autres visent à réduire sa taille. Pour les programmes qui consomment beaucoup de temps CPU, il est utile d'activer l'optimisation du compilateur.

[3] Cette implémentation est adaptée de **http://opensource.apple.com/source/gcc/gcc-926/libiberty/strerror.c** et est dans le domaine public.

[4] Cette implémentation est adaptée de **https://www-asim.lip6.fr/trac/netbsdtsar/browser/vendor/netbsd/5/src/lib/libc/string/strerror_r.c?rev=2** et est *Copyright (c) 1988 Regents of the University of California*.