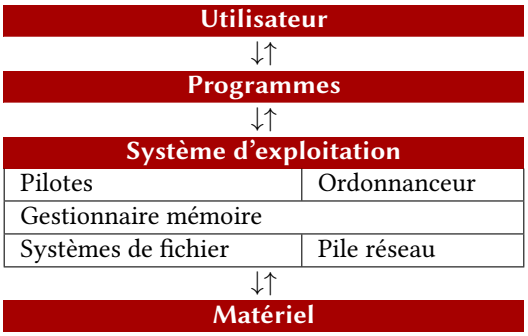
Concurrence et synchronisation

Les ordinateurs donnent l'impression de réaliser plusieurs tâches à la fois. En réalité, à un instant donné, un seul programme est en cours d'exécution. On parle de processus pour désigner ce programme en cours d'exécution. Dès lors, comment l'ordinateur, en fait le système d'exploitation plus précisément, gère-t-il l'apparente simultanéité de plusieurs processus ? Pour répondre à cette question, un système informatique dit *concurrent*, c'est-à-dire pour lequel plusieurs tâches paraissent s'exécuter *en même temps*, doit être conçu. Certaines tâches de ces systèmes peuvent *coopérer* pour résoudre un problème tandis que d'autres sont plus indépendantes mais *partagent* ou se *disputent* les ressources du systèmes. La *programmation concurrente* met en place des mécanismes pour gérer [1] les tâches, synchroniser les activités des tâches et faciliter leur communication, et enfin partager et protéger l'accès aux ressources du système. Ce chapitre présente les concepts de la *programmation concurrente* en utilisant des bibliothèques de gestion des processus légers, appelés *threads* en anglais, disponibles dans les langages C et OCaml.

Système d'exploitation

Un système d'exploitation [2] est un *programme* ou un *ensemble de programmes* qui *gère* les *ressources* matérielles et logicielles d'un ordinateur. Il fournit en particulier aux programmes utilisateurs un *accès unifié* à ces ressources. Le schéma ci-dessous indique la place du système d'exploitation et ses diverses interactions. L'utilisateur interagit avec des programmes (jeu, navigateur Web, traitement de texte). Ces derniers ont besoin d'utiliser les ressources de la machine pour effectuer leur tâches (lire ou sauvegarder des fichiers, afficher des images à l'écran, récupérer les caractères saisis au clavier ou la position du pointeur de la souris). Le système d'exploitation offre un ensemble de fonctions primitives permettant d'interagir avec le matériel.



Parmi les différents composants logiciels présents dans les systèmes d'exploitation modernes, on trouve :

- l'*ordonnanceur* qui décide quel programme s'exécute à un instant donné sur le processeur ;
- le *gestionnaire de mémoire*, qui répartit la mémoire vive entre les différents programmes en cours d'exécution ;
- les différents *systèmes de fichiers*, qui définissent la manière de stocker les fichiers sur les supports physiques (disques, clés USB, disques optiques, etc.) ;
- la *pile réseau* qui implémente entre autres des protocoles tels que TCP/IP ;
- les *pilotes de périphériques* [3], dont le but est de gérer les périphériques matériels (carte graphique, disques durs, clavier, etc).

Malgré la grande diversité des systèmes d'exploitation, il existe un ensemble de standards, regroupés sous le nom de POSIX [4]. Ces standards définissent aussi bien les fonctions de bibliothèques que doit offrir le système d'exploitation [5] que les programmes de base permettant d'utiliser le système. La plupart des systèmes d'exploitation modernes sont compatibles avec le standard POSIX.

Processus

Processus et organisation de la mémoire

Dans un système d'exploitation, une tâche, appelée *processus*, représente un *programme en cours d'exécution*. Un processus est donc le phénomène dynamique qui correspond à l'exécution d'un programme particulier. Le système d'exploitation identifie généralement les processus par un *numéro unique*. Un processus est décrit par un *contexte* qui rassemble :

1. Créer, exécuter et arrêter.
2. *Operating System* ou *OS* en anglais.
3. Ou *drivers* en anglais.
4. Pour *Portable Operating System Interface*.
5. Pour lire et écrire dans des fichiers, accéder au réseau, etc.

- l'ensemble de la mémoire allouée par le système pour l'exécution de ce programme (ce qui inclut le code exécutable copié en mémoire et toutes les données manipulées par le programme, sur la pile ou dans le tas) ;
- l'ensemble des ressources utilisées par le programme (fichiers ouverts, connexions réseaux, etc.) ;
- les valeurs stockées dans tous les registres du processeur.

Les espaces d'adressage des processus sont toujours disjoints. Il n'est donc pas possible pour un processus d'accéder à la mémoire d'un autre processus.

Commandes Unix de gestion des processus

Dans les systèmes POSIX, la commande `ps` (pour l'anglais *process status* ou état des processus) permet d'obtenir des informations sur les processus en cours d'exécution.

```
$ ps aux
```

Les options `a`, `u` et `x` permettent respectivement d'afficher tous les processus (et pas seulement ceux de l'utilisateur qui lance la commande), d'afficher le nom des utilisateurs (plutôt que leur identifiant numérique) et de compter aussi les processus n'ayant pas été lancés depuis un terminal (comme les *daemon* ou les processus lancés depuis une interface graphique). La commande affiche sur la sortie standard des informations sur les processus, comme par exemple :

```
USER   PID %CPU %MEM    VSZ   RSS  TTY STAT START TIME COMMAND
root     1  0.1 0.0 170088 11796 ?    Ss   Apr22 2:40 /lib/systemd/sy
...
alice 1438  0.0 0.0  11548  4952 tty2 Ss   15:45 0:00 bash
alice 3537  5.4 3.6 998564 60764 ?    Sl   15:12 9:11 /usr/bin/firefox
root  6524  0.0 0.0  29260  7780 ?    Ss   00:00 0:00 /usr/sbin/cupsd
alice 6966  9.8 2.0 140692 24240 ?    SLl 15:41 2:56 /usr/bin/emacs
alice 7490  0.0 0.0  11668  2704 tty2 R+   15:47 0:00 ps -a -u -x
```

La colonne `USER` indique le nom de l'utilisateur qui a lancé le processus. La colonne `PID` donne l'identifiant numérique du processus. Les colonnes `%CPU` et `%MEM` indiquent respectivement le taux d'occupation du processeur et de la mémoire par le processus. Par exemple, dans l'affichage ci-dessus, on peut voir que le processus `6966` occupe 9,8% du temps de calcul du processeur et 2% de la mémoire. En simplifiant un peu, on peut dire que sur les dernières 100 secondes d'utilisation du système, 9,8 secondes ont été passées à exécuter des instructions du processus 6966. La colonne `TTY` indique l'identifiant du terminal où le processus a été lancé. Un caractère `?` indique que le processus n'a pas été lancé depuis un terminal. La colonne `STAT` indique l'état du processus, la première lettre étant en majuscule. Sur la plupart des systèmes Unix, les états sont :

`R` : *running* ou *runnable* : le processus est dans l'état prêt ou en exécution (`ps` ne différencie pas ces deux états) ;

`S` : *sleeping* : le processus est *en attente*.

Les colonnes `START` et `TIME` indiquent respectivement l'heure ou la date à laquelle le programme a été lancé et le temps cumulé d'exécution du processus correspondant (c'est-à-dire le temps total pendant lequel le processus était dans l'état *en exécution*). Enfin, la colonne `COMMAND` indique la ligne de commande utilisée pour lancer le programme [6].

Ordonnancement et entrelacement

Comment un ordinateur peut-il exécuter *en même temps* plusieurs processus ? Même si les machines disposent aujourd'hui de plusieurs processeurs et de plusieurs cœurs sur chaque processeur, le nombre de processus est très grand, avec souvent plusieurs centaines voire plusieurs milliers de processus actifs en même temps. C'est en fait l'*ordonnanceur de processus* du système d'exploitation qui gère l'exécution *concurrente* des processus. À des intervalles de temps réguliers [7], il *interrompt* le processus en cours d'exécution, *sauvegarde* son *contexte*, *choisit* un autre processus [8] en *restaurant* son contexte et lui « donne la main ». Ce passage d'un processus à un autre s'appelle une *commutation de contexte*.

L'exécution *en parallèle* n'est en réalité rien d'autre qu'un *entrelacement non déterministe* d'instructions des processus. Si on exécute plusieurs fois de suite les mêmes programmes, l'ordonnanceur peut décider de changer l'ordre de leur processus en fonction de divers paramètres [9]. Cela peut bien évidemment changer le comportement du système [10]. Ainsi, pour l'utilisateur, et pour les programmes exécutés, tout se passe comme si ils évoluaient dans un système concurrent.

6. Elle est tronquée dans l'exemple pour des raisons de place.
7. Fixés par une horloge.
8. Le choix du prochain processus à exécuter peut être guidé par un mécanisme de priorités.
9. Comme le nombre de processus total en cours d'exécution, valeurs des horloges, etc.
10. Par exemple, les réservations faites dans une billetterie peuvent être différentes, un système temps-réel peut réagir différemment, etc.

Problèmes de synchronisation

Cette façon d'entrelacer l'exécution des processus possède de nombreux avantages. Elle permet l'exécution d'un très grand nombre de programmes sur une machine monoprocesseur [11]. Elle permet aussi d'optimiser les ressources de la machine. Par exemple, si un processus est en attente d'entrées-sorties, il est simplement mis en pause et le système peut utiliser le processeur pour effectuer un calcul utile. Et même si tous les processus sont en attente d'un événement, le système peut alors décider dans ce cas de réduire la fréquence du processeur ou de le mettre partiellement en veille, ce qui permet d'économiser de l'énergie. Cet aspect est particulièrement important pour les systèmes mobiles ou embarqués.

Cependant, l'utilisation de systèmes multitâches n'est pas sans poser de problèmes dont, notamment, deux problèmes classiques liés à la *synchronisation* des activités des processus : l'*exclusion mutuelle* [12] et les *producteurs-consommateurs*.

- Le problème de l'*exclusion mutuelle* est lié à l'accès à une ressource partagée qui ne peut être utilisée que par un processus à la fois. Lorsqu'un processus est interrompu, il ne s'en « rend pas compte ». Dit autrement, lorsqu'un processus est interrompu, il reprendra son exécution exactement dans l'état où il s'était arrêté. Tant que ce processus manipule des objets visibles de lui seul [13], tout va bien. Mais si le processus accède à une *ressource partagée*, comme un fichier ou un périphérique matériel, alors de nombreux problèmes peuvent se produire.
- Le problème des *producteurs-consommateurs*, décrit par Edsger W. Dijkstra en 1965, met en jeu deux types de processus : les producteurs et les consommateurs. Les premiers passent leur temps à produire des données qu'ils stockent dans une zone mémoire de *taille finie* partagée avec les consommateurs. De manière concurrente, ces derniers cherchent à consommer des données en les retirant de cette zone. Ce schéma pose le problème suivant : les producteurs doivent arrêter de produire des données quand la zone de mémoire est pleine ; les consommateurs ne doivent pas retirer des données quand cette zone est vide.

Un mécanisme de synchronisation, appelé *sémaphore*, constitue une solution à ces problèmes.

Threads POSIX

La gestion de la concurrence à l'aide de processus a au moins deux défauts. Tout d'abord, elle ne permet pas facilement la communication entre processus, ni la gestion coopérative des ressources partagées. Ensuite, le coût du changement de contexte est important car il nécessite la sauvegarde et la restauration de nombreuses informations. Pour pallier ces deux problèmes, on programme avec des *processus légers*, appelés également *fils d'exécution* ou *threads* en anglais.

Le mécanisme de *threads* permet d'exécuter *en même temps* plusieurs fils d'exécution au sein d'un même processus. Un *thread* correspond à un bout de code, généralement une fonction, du programme en cours. Les *threads* partagent tous le même contexte d'évaluation du processus, ce qui leur permet d'avoir accès au même environnement global, aux mêmes ressources, etc. Les *commutations de contexte* entre *threads* sont aussi plus rapides.

Pour programmer avec des *threads* en C ou OCaml, on utilise des *threads POSIX*, également appelés *pthreads*. Il s'agit d'une bibliothèque dont le contenu est standardisé (IEEE Std 1003.c) et qui se trouve sur de nombreux systèmes d'exploitation (Linux, MacOS) [14] et pour de nombreux langages de programmation.

Le module Thread en OCaml

Dans le langage OCaml, les *threads POSIX* sont disponibles en utilisant le module `Thread`. On y trouve notamment la définition du type `Thread.t`, ainsi que des fonctions pour créer, terminer et attendre la fin de l'exécution d'un *thread*. Les types de ces fonctions et leur description sont donnés dans la table ci-dessous.

Fonction	Description
`Thread.create`	`('a -> 'b) -> 'a -> Thread.t` `Thread.create f v` crée un nouveau *thread* pour exécuter l'appel `f v`. La fonction s'exécute en même temps que le *thread* appelant. La valeur retour de `f` n'est pas utilisée.
`Thread.exit`	`unit -> unit` Termine le *thread* qui exécute cet appel.
`Thread.join`	`Thread.t -> unit` `Thread.join t` suspend l'exécution du *thread* appelant jusqu'à ce que `t` ait terminé son exécution.
`Thread.yield`	`unit -> unit` En appelant cette fonction, le *thread* appelant indique à l'ordonnanceur qu'il peut être interrompu (mais rien ne l'oblige à le faire). Cette fonction est utile pour forcer l'entrelacement des instructions (et faciliter le débogage).

11. Ou ayant un petit nombre de cœurs.
12. Connu également sous le nom de la *section critique*.
13. Par exemple, des variables allouées sur la pile ou dans le tas.
14. Sous Windows, cette bibliothèque est disponible sous le nom `pthread-win32`.

Le programme suivant utilise la bibliothèque `Thread` pour créer deux fils d'exécution de la fonction `f`.

```
let n_iter = 10
let f n =
  for i = 1 to n_iter do
    Printf.printf "%s%d " n i;
    flush stdout;
    Thread.yield();
  done;
  Thread.exit ()
```

```
let t1 = Thread.create f "A"
let t2 = Thread.create f "B"
let () =
  Thread.join t1;
  Thread.join t2;
  Printf.printf "\nEnd\n"
```

Lorsque ce programme est exécuté, les deux appels à `Thread.create` créent deux fils d'exécution `t1` et `t2` pour, respectivement, les appels `f "A"` et `f "B"` à la fonction `f`. Ces appels vont s'exécuter en même temps que le programme principal. Ce dernier doit alors attendre que chaque *thread* ait terminé de s'exécuter en appelant la fonction `Thread.join`, sans quoi les *threads* seront immédiatement terminés à la fin du processus du programme principal (en fait, il est probable qu'ils n'auront même pas eu le temps de démarrer). La fonction `f` prend en argument une chaîne de caractères `n` et exécute une boucle `for` pour afficher les messages `n1`, `n2`, ... Le nombre de messages à afficher est fixé par la variable `n_iter` à laquelle les *threads* ont accès puisqu'elle est dans l'environnement partagé. L'instruction `flush stdout` force l'affichage immédiat à l'écran (en vidant le *buffer* d'affichage). Enfin, comme l'exécution de cette boucle est très rapide, le système d'exploitation n'a pas le temps d'entrelacer l'exécution des deux *threads*. Chaque *thread* donne donc une chance à l'ordonnanceur de l'interrompre grâce à l'appel `Thread.yield ()`. Chaque *thread* se termine « proprement » en appelant la fonction `Thread.exit`. Cet appel n'est pas obligatoire car il est fait implicitement lorsque la fonction se termine. Enfin, lorsque les deux fils d'exécution sont terminés, le programme principal affiche le message `End`.

Pour compiler un programme OCaml avec la bibliothèque `Thread`, il faut utiliser l'option `-I +threads` et ajouter explicitement les fichiers `unix.cmxa` et `threads.cmxa` (dans cet ordre) dans la commande de compilation. Ainsi, pour compiler le programme ci-dessous, on tape la commande suivante.

```
$ ocamlopt -I +threads unix.cmxa threads.cmxa -o test test.ml
```

En exécutant ce programme, on peut obtenir le résultat suivant (le non-déterminisme peut changer l'ordre d'affichage).

```
$ ./test
A1 B1 A2 B2 A3 B3 A4 B4 A5 B5 A6 A7 A8 A9 B6 B7 B8 B9
End
```

L'affichage des messages alterne entre les instructions des deux *threads* en raison de l'entrelacement des fils d'exécution par l'ordonnanceur.

La bibliothèque `pthread` en C

En C, un programme avec des *threads* utilise la bibliothèque `pthread`, chargée à l'aide de `#include <pthread.h>`. Ce fichier contient la définition du type `pthread_t` ainsi que les signatures des fonctions décrites ci-dessous.

Fonction	Description
`pthread_create`	`(pthread_t *thread, pthread_attr_t *attr, void *(* f)(void *), void *arg)` Un appel `pthread_create(&t, &attr, f, p)` crée un nouveau *thread* pour exécuter `f(p)`. L'argument `t` reçoit la valeur pour identifier le *thread*. Le pointeur `attr` contient les attributs de `t`. S'il vaut `NULL`, `t` a les attributs par défaut : (1) on peut attendre qu'il termine avec `pthread_join`, (2) il est ordonnancé « normalement » (*i.e.* pas prioritaire). La fonction renvoie `0` (succès) ou un code d'erreur.
`pthread_exit`	`(void *ret)` Termine le *thread* qui exécute cet appel. Le pointeur `ret` permet de passer une valeur de retour à un *thread* qui attend la terminaison avec `pthread_join`.
`pthread_join`	`(pthread_t t, void **ret)` Suspend l'exécution du *thread* appelant jusqu'à ce que le *thread* en paramètre ait terminé. L'argument `ret` récupère la valeur transmise avec `pthread_exit`.
`sched_yield`	`(void)` Le *thread* appelant indique à l'ordonnanceur qu'il peut être interrompu.

L'exemple donné plus haut en OCaml peut s'adapter comme suit.

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

int n_iter = 10;

void *f(void *arg) {
  for (int i = 1; i <= n_iter; i++) {
    printf("%s%d ", (char *)arg, i);
    fflush(stdout);
    sched_yield();
  };
  pthread_exit(NULL);
}
```

```c
int main(int argc, char **argv) {
  pthread_t t1;
  pthread_t t2;
  pthread_create(&t1, NULL, f, "A");
  pthread_create(&t2, NULL, f, "B");
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("\nEnd\n");
}
```

La structure de ce programme est très similaire à celle du programme en OCaml. Les seules différences notables sont liées au typage. Puisque le système de types de C n'a pas de polymorphisme, des expressions explicites de conversion de type [15] sont nécecessaires pour que la vérification de typage du compilateur C se fasse sans erreur. La compilation se fait avec l'option `-pthread`. À l'exécution, on observe le même comportement qu'avec le programme OCaml.

```
$ gcc -pthread -o test test.c
$ ./test
A1 B1 B2 B3 B4 A2 B5 A3 A4 A5 A6 A7 A8 B6 A9 B7 B8 B9
End
```

Arrêter un fil d'exécution

La bibliothèque POSIX contient une fonction pour arrêter prématurément un *thread*. Il s'agit de la fonction `Thread.kill` en OCaml, et `pthread_kill` en C. Il n'est pas conseillé d'utiliser ces fonctions. En effet, comme nous le verrons dans la section suivante, forcer la fin d'un *thread* de cette manière ne permet pas de l'arrêter « proprement ». Par exemple, si le *thread* « possède » une ressource critique, le tuer ainsi va rendre la ressource inutilisable pour le reste du système.

Atomicité

Dans cette partie, on suppose que tous les programmes écrits en C commencent par charger les bibliothèques `stdio.h` et `pthread.h`. On omet de présenter le code pour exécuter des *threads* quand celui-ci est identique ou très similaire aux programmes présentés dans la section précédente.

Comme nous l'avons déjà indiqué, un *thread* peut être interrompu à *n'importe quel moment* par le système d'exploitation pour « donner la main » à un autre *thread*. Cela peut se produire pendant une instruction qui peut paraître *atomique*, c'est-à-dire *non interruptible par le programmeur*. Prenons par exemple la fonction `f` ci-dessous qui incrémente 10000 fois de suite une variable `x` (initialisée à 0), et supposons qu'un programme C exécute de manière concurrente deux fils d'exécution pour cette fonction. Quelle sera la valeur affichée pour `x` quand les deux *threads* auront terminé ?

```c
int x = 0;

void *f(void *arg){
  for (int i = 1; i <= 10000; i++){
    x++;
  };
  pthread_exit(NULL);
}
```

Lorsqu'on exécute ce programme, on remarque qu'on trouve des valeurs différentes pour `x` (16498, 15312, 17640, etc.) et jamais la valeur attendue 20000. Pourquoi ? Tout simplement parce que l'instruction `x++` n'est pas *atomique*. Elle correspond à un code assembleur constitué des trois instructions suivantes :

```
movl   x,    %eax
addl   $1,   %eax
movl   %eax, x
```

La première charge le contenu de `x` dans le registre `eax`, puis ce registre est incrémenté par la deuxième instruction, et enfin le contenu du registre est sauvegardé dans `x`. Les fils d'exécution `t1` et `t2` pour ces instructions pouvant être interrompus avant ou après chaque opération, on peut avoir l'entrelacement suivant pour deux opérations `x++` par `t1` et `t2`. Pour simplifier, on suppose que `t1` et `t2` utilisent respectivement les registres `eax` et `ebx` pour réaliser ces opérations [16].

15. De `void *` vers `char *` ici.
16. Cela permet d'ignorer les sauvegardes liées aux changements de contexte.

```
movl x,    %eax   # t1 a la main :
                  # eax = 0
addl $1,   %eax   # eax = 1
movl x,    %ebx   # t1 est interrompu,
                  # t2 prend la main :
                  # ebx = 0
movl %eax, x      # t2 est interrompu,
                  # t1 prend la main :
                  # x = 1
addl $1,   %ebx   # t1 est terminé,
                  # t2 prend la main :
                  # ebx = 1
movl %ebx, x      # x = 1
```

Si on suppose que `x` contient `0` au début de cette séquence d'instructions, alors les registres `eax` et `ebx` sont chargés avec `0` à la première et troisième ligne. Après incrémentation des deux registres, c'est la valeur `1` qui est sauvegardée dans `x` par les deux *threads*. Au final, `x` vaut `1` et non `2` après cet entrelacement. On comprend donc pourquoi la valeur finale a peu de chance d'être 20000 après avoir exécuté ce programme.

Pour que ce programme soit correct, il faudrait rendre *atomique* l'instruction `x++`, c'est-à-dire soit empêcher le *thread* qui exécute cette instruction d'être interrompu, soit faire en sorte que le *thread* ait l'exclusivité sur la mémoire le temps qu'il fasse cette opération. Ccette deuxième solution est présentée dans la section suivante.

Mutex

Une *section critique* est une portion de programme qui, pour garantir la sûreté du système, ne peut être exécutée que par un nombre maximal de *threads* en même temps, très souvent, *un thread* à la fois. Elle peut être nécessaire pour avoir l'exclusivité sur une ressource. Toutefois, l est préférable que les sections critiques soient les plus petites possibles pour permettre au plus grand nombre de *threads* de s'excécuter en même temps. C'est pour cela qu'on programme un système concurrent ! Aussi, il est importe de minimiser les endroits du code qui nécessitent *absolument* d'être en section critique. Plus une section critique contiendra de code, plus on aura de chances que le programme soit correct mais moins il sera efficace. Dans le programme précédent, seule l'instruction `x++` doit être mise en section critque.

Mutex

Pour réaliser une section critique, on utilise une primitive de synchronisation, appelée *verrou*[17], qui possède deux opérations : `lock(m)`, pour prendre le verrou `m`, et `unlock(m)`, pour libérer le verrou. Pour délimiter une section critique, on utilise un verrou `m` avec le motif suivant.

```
lock(m);
<section critique>
unlock(m);
```

Il existe plusieurs solutions pour implémenter un verrou mais toutes doivent garantir les propriétés suivantes.

- (P1) Il ne peut y avoir qu'un seul *thread* à la fois dans une section critique.
- (P2) Un *thread* bloqué en dehors d'une section critique ne peut bloquer les autres *threads*.
- (P3) Un *thread* ne doit pas attendre infiniment pour entrer en section critique.

Solution OCaml

Le système de *threads* POSIX d'OCaml fournit un module `Mutex` pour manipuler des verrous. On y trouve la définition du type `Mutex.t`, ainsi que des fonctions pour *créer*, *verrouiller* ou *déverrouiller* des verrous.

Fonction	Description
`Mutex.create`	`unit -> Mutex.t` `Mutex.create ()` crée un nouveau *mutex*.
`Mutex.lock`	`Mutex.t -> unit` Un appel `Mutex.lock m` verrouille le *mutex* `m`. Un seul *thread* à la fois peut verrouiller ce *mutex*. Si un *thread* `t2` essaie de verrouiller `m` alors qu'il l'est déjà par `t1`, le *thread* `t2` est mis en attente jusqu'à ce que `t1` déverrouille `m`.
`Mutex.unlock`	`Mutex.t -> unit` Un appel `Mutex.unlock m` déverrouille le *mutex* `m`. Tous les *threads* suspendus parce qu'ils ont essayé de verrouiller `m` sont réveillés (pour tenter à nouveau de verrouiller `m`).

17. Ou *mutex* pour *mutual exclusion* en anglais.

La fonction `f` ci-dessous utilise un *mutex* pour délimiter une section critique autour de l'instruction `x := !x + 1` qui incrémente, de manière non atomique, la variable `x` qui peut être partagée par plusieurs *threads*.

```
let m = Mutex.create()
let x = ref 0
let f() =
  for i = 1 to 10000 do
    Mutex.lock m;
    x := !x + 1;
    Mutex.unlock m
  done;
  Thread.exit()
```

Si un *thread* exécute la fonction `f`, le *mutex* garantit qu'il est le seul à incrémenter `x` à chaque tour de boucle. Si on démarre deux fils d'exécution de `f`, on obtient bien `20000` dans `x` à la fin du programme.

Solution C

Les *mutex* en C sont disponibles avec la bibliothèque `pthread`. Le type des *mutex* est `pthread_mutex_t`. On retrouve les mêmes fonctions que pour le langage OCaml.

Fonction	Description
`pthread_mutex_init`	`(pthread_mutex_t *m, const pthread_mutexattr_t *attr)` `pthread_mutex_init(&m, &attr)` initialise un *mutex* pointé par `m` en utilisant l'attribut `attr`. Cet attribut spécifie ce qui se passe si un *thread* essaye de verrouiller `m` alors qu'il l'est déjà. Par défaut (`NULL`), le *thread* est bloqué. Il est également possible d'initialiser statiquement un *mutex* avec la valeur `PTHREAD_MUTEX_INITIALIZER`.
`pthread_mutex_lock`	`(pthread_mutex_t *m)` Un appel `pthread_mutex_lock(&m)` verrouille le *mutex* pointé par `m`. Le *thread* appelant est mis en attente si `m` est déjà verrouillé.
`pthread_mutex_unlock`	`(pthread_mutex_t *m)` Un appel `pthread_mutex_unlock(&m)` déverrouille le *mutex* pointé par `m` et libère les *threads* en attente, qui vont être en compétition pour acquérir le verrou.

On peut ainsi reprendre le programme C de la section prédédente pour définir une section critique autour de l'instruction `x++` à l'aide d'un *mutex* `m`. On obtient le code ci-dessous.

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
int x = 0;

void *f(void *arg){
  for (int i = 1; i <= 10000; i++){
    pthread_mutex_lock(&m);
    x++;
    pthread_mutex_unlock(&m);
  };
  pthread_exit(NULL);
}
```

Comme pour OCaml, si on démarre deux fils d'exécution de `f`, on obtient bien `20000` dans `x` à la fin du programme.

Algorithme de Peterson

Il existe de nombreuses solutions pour implémenter des *mutex*. L'*algorithme de Peterson*, inventé par Gary L. Peterson en 1981, est une solution purement logicielle. Bien que respectant les trois propriétés imposées plus haut, il n'est pas pratiquement pas utilisé. D'abord car il ne fonctionne que pour deux *threads*, ensuite, car le *thread* qui est bloqué pour entrer en section critique doit continuer à exécuter une boucle pour tester une condition. C'est le principe d'une *attente active*, peu efficace et consommatrice d'énergie. Néanmoins, cet algorithme est un exercice qui illustre les difficultés de conception d'une solution correcte au problème de l'exclusion mutuelle.

Une *première solution* utilise un tableau de booléens `flag` de sorte que chaque case `flag[i]` indique si le *thread* `i` est en section critique ou non. Cette solution ne respecte pas la propriété (P1). En effet, les deux *threads* peuvent se retrouver en même temps en section critique. Pour cela, il suffit qu'ils atteignent la condition d'entrée de la boucle `while` en même temps. À ce moment-là, la valeur de `flag[other]` vaut `false` pour chaque *thread*, et ils peuvent entrer tous les deux en section critque.

```
bool flag[2] = { false, false };
void lock(int i) {
  int other = 1 - i;
  // attente active
  while (flag[other]) ;
  flag[i] = true;
}
void unlock(int i) {
  flag[i] = false;
}
```

```
bool want[2] = { false, false };
void lock(int i) {
  int other = 1 - i;
  want[i] = true;
  // attente active
  while (want[other]) ;
}
void unlock(int i) {
  want[i] = false;
}
```

Une *deuxième solution* utilise un tableau de booléens `want` de sorte que chaque case `want[i]` indique si le *thread* `i` veut entrer en section critique ou non. Cette solution ne respecte pas la propriété (P2). En effet, les deux *threads* peuvent se bloquer mutuellement. Pour cela, il suffit qu'ils atteignent la condition d'entrée de la boucle `while` en même temps. Dès lors, la valeur de `want[other]` vaut `true` pour chaque *thread* et ils ne peuvent entrer tous les deux en section critque.

Une *troisième solution* utilise une variable entière `turn` qui indique quel *thread* est autorisé à entrer en section critique. Cette solution ne respecte pas la propriété (P3). En effet, si l'un des deux *threads* termine, l'autre est bloqué.

```
int turn = 0;
void lock(int i) {
  int other = 1 - i;
  while (turn == other) ; // attente active
}
void unlock(int i) {
  int other = 1 - i; turn = other;
}
```

La *quatrième version*, proposée par Peterson, combine les deuxième et troisième solutions. Deux variables contrôlent l'entrée dans la section critique : un tableau `want` est utilisé par chaque `thread` pour indiquer qu'il veut entrer en section critique et une variable entière `turn` indique quel *thread* peut entrer en section critique.

```
int turn = 0;
bool want[2] = { false, false };
void lock(int i) {
  int other = 1 - i;
  want[i] = true;
  turn = other;
  // attente active
  while (want[other] && turn == other) ;
}
void unlock(int i) { want[i] = false; }
```

Supposons que le *thread* `t1` souhaite entrer en section critique. Il commence par indiquer qu'il veut y entrer (`want[0] = true`), puis il donne la possibilité à l'autre *thread* d'acquérir le verrou en positionnant la variable `turn` avec son numéro `1`. Si `t2` ne souhaite pas le verrou (ou simplement s'il est terminé), la condition de la boucle `while` est fausse et `t1` peut entrer en section critique. Cela respecte bien la propriété (P3). Si `t2` souhaite également entrer en section critique, alors la variable `turn` décide lequel des deux *threads* peut entrer. Cela permet de respecter la propriété (P1). En aucun cas les deux *threads* ne peuvent rester bloquer devant la condition de la boucle `while`, ce qui permet de respecter la propriété (P2).

Instructions atomiques

De nombreuses implémentations de *mutex* reposent sur des instructions atomiques des processeurs. On utilise par exemple l'instruction `test-and-set` qui positionne une variable entière à `1` et renvoie sa valeur précédente. La sémantique de cette instruction est équivalente au code ci-dessous, en supposant que la fonction `test_and_set` puisse être exécutée de manière atomique.

```
int test_and_set(int *m) {
  int old = *m;
  *m = 1;
  return old;
}
```

On trouve également d'autres instructions atomiques comme `compare-and-swap`, `compare-and-exchange`, etc.

Interblocages

Il convient d'être très prudent lorsqu'on manipule des verrous. Par exemple, dans le programme ci-dessous, les deux fils d'exécution `t1` et `t2` (qui exécutent respectivement les fonctions `f1` et `f2`) peuvent se retrouver en situation

d'*interblocage*, c'est-à-dire qu'il leur est impossible de progresser à cause de l'état de l'autre *thread*. Supposons que le *thread* `t1` soit exécuté en premier et verrouille `m1`. Si `t1` est interrompu et que `t2` prend la main, alors `t2` peut à son tour verrouiller `m2`. À ce moment-là, les deux *threads* sont *mutuellement* bloqués l'un par l'autre.

```
let m1 = Mutex.create ()
let m2 = Mutex.create ()

let f1 () =
  Mutex.lock m1;
  Thread.yield();
  Mutex.lock m2;
  Format.printf "section critique@.";
  Mutex.unlock m2;
  Mutex.unlock m1

let f2 () =
  Mutex.lock m2;
  Thread.yield();
  Mutex.lock m1;
  Format.printf "section critique@.";
  Mutex.unlock m1;
  Mutex.unlock m2

let t1 = Thread.create f1 ()
let t2 = Thread.create f2 ()
let () = Thread.join t1; Thread.join t2
```

Sémaphores

Des problèmes de synchronisation plus complexes que celui de l'exclusion mutuelle sont résolus à l'aide des *sémaphores*, primitives inventées par Dijkstra en 1965, qui généralisent la primitive du *mutex*. Il existe deux types de sémaphores : les sémaphores *binaires* et les sémaphores *à compteur*. Les sémaphores binaires sont équivalents aux *mutex*. Cette section présente uniquement les *sémaphores à compteur*, structures de données constituées de deux parties.

- Une variable entière `cnt`, appelée *compteur*, initialisée avec une valeur positive ou nulle. Pendant toute l'utilisation d'un sémaphore, son compteur ne peut contenir que des valeurs positives (ou nulles).
- Une *file d'attente* `queue`, initialement vide, utilisée pour mémoriser des *threads* en attente.

Dans la suite, si une variable `s` contient un sémaphore, on écrira `s.cnt` le compteur associé à `s`, et `s.queue` la file de `s`.

Une fois créé et initialisé, un sémaphore `s` s'utilise à l'aide de deux opérations. Traditionnellement, les noms de ces opérations [18] sont `P(s)` et `V(s)`.

- `P(s)` teste `s.cnt > 0`. Si le test réussit alors le compteur `s.cnt` est décrémenté. Sinon, le *thread* ayant appelé `P(s)` est suspendu et mis en attente dans la file `s.queue`.
- `V(s)` réveille un *thread* en attente dans `s.queue`, s'il en existe, et incrémente `s.cnt` sinon.

On retient que l'opération `P(s)` peut être bloquante pour le *thread* qui l'exécute tandis que l'opération `V(s)` n'est jamais bloquante. Cependant elle peut réveiller un *thread*. Par ailleurs, la définition d'un sémaphore ne précise pas le mode de gestion de la file d'attente ; on ne peut donc pas savoir à l'avance quel *thread* est réveillé par une opération `V(s)`.

Solution OCaml

Le langage OCaml fournit un module `Semaphore` qui contient deux sous-modules `Binary` et `Counting`. Le premier implémente des sémaphores binaires et le second des sémaphores à compteur. Les fonctions et le type `t` donnés dans la table ci-dessous sont disponibles dans le module `Semaphore.Counting`. Il faut donc préfixer tous les identificateurs par `Semaphore.Counting` pour y avoir accès, par exemple en tapant `Semaphore.Counting.make n` pour la fonction `make`. Les opérations `P` et `V` sont respectivement implémentées par les fonctions `acquire` et `release`.

18. Qui viennent du néerlandais *Proberen* (tester) et *Verhogen* (incrémenter)

Fonction	Description
`make`	`int -> t` Un appel `make n` crée un nouveau sémaphore avec un compteur initialisé à `n` (un entier positif ou nul).
`acquire`	`t -> unit` `acquire s` bloque le *thread* appelant tant que le compteur de `s` est égal à 0, puis il décrémente le compteur de manière atomique.
`release`	`t -> unit` Un appel `release s` incrémente le compteur de `s`. Si des *threads* sont en attente sur `s`, un d'eux est réveillé.

Solution C

La bibliothèqe `pthread` du langage C fournit également une implémentation des sémaphores. Le type des sémaphores est `sem_t`. Les opérations `P` et `V` sont implémentées respectivement par les fonctions `sem_wait` et `sem_post`.

Fonction	Description
`sem_init`	`(sem_t *s, int sh, unsigned int v)` `sem_init(&s, sh, v)` initialise un sémaphore pointé par `s`. L'entier `v` spécifie la valeur initiale du compteur. Si `sh` vaut `0`, alors `s` est partagé entre tous les *threads* d'un même processus.
`sem_wait`	`(sem_t *s)` `sem_wait(&s)` décrémente le compteur du sémaphore pointé par `s`. Si le compteur est toujours positif, l'appel se termine immédiatement. Sinon, le *thread* appelant est bloqué.
`sem_post`	`(sem_t *s)` `sem_post(&s)` incrémente le compteur du sémaphore pointé par `s`. Réveille un *thread* bloqué sur `s` si le compteur devient supérieur à `0`

Producteurs et consommateurs

Dans le problème des *producteurs* et des *consommateurs*, il existe deux types de *threads* qui souhaitent s'échanger de l'information. Pour ce faire, ils utilisent un tampon mémoire (*buffer*) partagé. Les producteurs passent leur temps à écrire des données dans le *buffer* et les consommateurs ne font que supprimer ces données du *buffer*. Le *buffer* est de taille finie et il est initialement vide. Les autres contraintes liées à ce problème sont les suivantes.

1. Un seul *thread* à la fois (producteur ou consommateur) accède au *buffer*, pour y écrire ou supprimer des données.
2. Un consommateur qui souhaite supprimer une donnée du *buffer*, alors qu'il est *vide*, est mis en attente.
3. Un producteur qui tente d'écrire une donnée dans le *buffer*, alors qu'il est *plein*, est mis en attente.

Ce problème se résout en utilisant des *sémaphores* et des *mutex*. Tout d'abord, la contrainte **1** implique que l'accès au *buffer* par les producteurs ou les consommateurs est une section critique. Il faut donc le protéger par un *mutex*. Ensuite, on va utiliser deux sémaphores à compteur, `empty` et `full`, pour prendre en compte les contraintes **2** et **3**.

- Le sémaphore `empty` est utilisé pour compter le nombre de places *libres* dans le *buffer*. Ce sémaphore est donc initialisé avec la taille du *buffer*. Un producteur qui souhaite ajouter une valeur dans le *buffer* devra donc tout d'abord décrémenter le compteur de `empty` pour s'assurer qu'une place est libre. Dans le cas contraire, il sera bloqué en attendant qu'une place se libère. C'est au consommateur d'incrémenter le compteur de `empty` chaque fois qu'il supprime une donnée du *buffer* afin de débloquer un producteur.
- De manière symétrique, le sémaphore `full` est utilisé pour compter le nombre de places *occupées* dans le *buffer*. Ce sémaphore est initialisé à 0 au début du programme. Un consommateur qui souhaite supprimer une valeur dans le *buffer* devra donc tout d'abord décrémenter le compteur de `full` pour s'assurer qu'une donnée est disponible. Dans le cas contraire, il sera bloqué en attendant qu'une donnée soit déposée par un producteur. C'est au producteur d'incrémenter le compteur de `full` chaque fois qu'il ajoute une donnée au *buffer* afin de débloquer un consommateur.

Le programme suivant implémente une solution au problème des producteurs-consommateurs en utilisant les *mutex* et les sémaphores du langage OCaml.

```
let buffer = Queue.create()
let size = 5
let nb_p = 2
let nb_c = 5

let empty = Semaphore.Counting.make size
```

```
let full = Semaphore.Counting.make 0
let m = Mutex.create ()

let producer i =
  while true do
    Semaphore.Counting.acquire empty;
    Mutex.lock m;
    let v = Random.int 100 in
    Printf.printf "Producer %d : %d\n" i v;
    Queue.push v buffer;
    Mutex.unlock m;
    Semaphore.Counting.release full;
    Thread.yield()
  done

let consumer i =
  while true do
    Semaphore.Counting.acquire full;
    Mutex.lock m;
    let v = Queue.take buffer in
    Printf.printf "Consumer %d : %d\n" i v;
    Mutex.unlock m;
    Semaphore.Counting.release empty;
    Thread.yield()
  done

let tc = Array.init nb_c (fun i -> Thread.create consumer i)
let tp = Array.init nb_p (fun i -> Thread.create producer i)
let () =
  Array.iter Thread.join tc;
  Array.iter Thread.join tp
```