TP 11 : Fils d'exécutions et Mutex : des exemples simples

Reportez-vous à l'annexe en fin de TP pour toute question relative au lancement des machines virtuelles, à l'utilisation du terminal, à la compilation, ou aux fonctions de modules standard de C ou d'OCaml.

Les sources fournies pour ce TP sont à récupérer :

- Soit sur le réseau du lycée : `Ce PC > S: (Classes)/MPI/Sujets/Informatique/TP10`
- Soit sur la page Cahier-de-Prépa de notre classe : `https://cahier-de-prepa.fr/info-kleber`

I Entrelacements

Cette partie relativement facile a pour but d'observer les entrelacements d'exécution lorsqu'on exécute différent fils d'exécution en parallèle.

On travaille pour l'instant en OCaml. On ne fournit aucune source, on travaillera dans un fichier vierge.

Question 1 Déclarer un entier `borne` en variable globale. La valeur est laissée au choix, on pourra la changer au besoin lors des tests.

Question 2 Ecrire une fonction `f`, prenant en argument une chaîne de caracètres `s`, et qui, pour i de 0 jusqu'à `borne-1`, affichera la chaîne suivie de i. Par exemple, si `s = "A"`, la fonction doit afficher `A0 A1 A2...`.
Cette fonction a vocation à être utilisée dans un thread, et doit donc terminer par un appel mettant fin au thread après l'affichage.

Question 3 Ecrire un programme qui crée deux threads séparés appelant la fonction `f` sur `"A"` et sur `"B"`. Ce programme devra attendre que les deux threads aient fini de s'exécuter, puis afficher `"Fini"`.

Question 4 Que s'attendrait-on à observer ? Qu'observe-t-on ?

On travaille maintenant en C. On ne fournit aucune source, on travaillera dans un fichier vierge.

Question 5 Répéter les questions précédentes en C.

II Compteur

Cette partie est à réaliser en C. On ne fournit aucune source, on travaillera dans un fichier vierge.

Le but de cette partie est de réaliser un compteur sur une variable globale, réparti sur plusieurs fils d'exécution. On étudie notamment le gain en performance et les éventuelles limites du parallélisme. On n'oubliera pas d'inclure `stdio.h` dans le code à utiliser.

II.1 Compteur mono-fil

On commence par écrire un compteur qui ne crée pas de fil d'exécution secondaire. Le programme effectuera donc son travail uniquement avec le fil principal.

Question 6 Déclarer une variable globale `compteur` de type `long unsigned int` initialisée à 0 et une variable globale `objectif`.

Question 7 Ecrire une fonction `long unsigned int puissance(long unsigned int a, long unsigned int b)` permettant de calculer a^b. Utiliser cette fonction pour placer la valeur de `objectif` à 2^{30}.

Question 8 Écrire une fonction `void *f(void* arg)` qui incrémente la variable `compteur` autant de fois que la valeur donnée dans `objectif`.

Question 9 Appeler `f(NULL)` quatre fois dans le `main` et afficher la valeur de `compteur` après cet appel.

Question 10 Compiler le programme. Utiliser la commande `time ./compteur_simple` (si on appelle l'exécutable précédent `compteur_simple`) pour mesurer le temps de calcul. Le temps d'exécution devrait être autour de 20s en ordre de grandeur, mais peut varier selon votre machine et selon l'environnement. Passer à un objectif de 2^{31} s'il est significativement plus petit.

II.2 Compteur sur quatre fils

On va maintenant répartir le travail sur quatre fils d'exécution. L'exécution du programme comportera automatiquement un fil principal (qui exécute la fonction `main`, et qui est créé à l'exécution), qui utilisera 4 fils d'exécution que nous prendrons soin de créer nous-mêmes.

Question 11 Dans la fonction `main`, déclarer un tableau `travailleurs` de taille 4 dont les cases contiendront des fils d'exécution.[1]

Question 12 Dans la fonction `main`, dans chaque case, créer un fil d'exécution qui exécute `f`.

Question 13 Ecrire le code nécessaire pour que le programme principal attende la fin de l'exécution de chacun de ces fils avant de terminer.

Question 14 Qu'observe-t-on ?

Pour éviter le problème soulevé dans la question précédente, on va utiliser un `mutex` pour protéger l'accès au compteur.

Question 15 Déclarer en variable globale un mutex `m`. On l'intialisera dans la fonction `main`, puis on l'utilisera pour protéger l'accès au compteur, en autorisant `f` à verrouiller et déverrouiller `m` quand le compteur doit être incrémenté.

Question 16 Comparer le temps d'exécution de ce nouveau programme au précédent.

Avec cette version multi-threadé, on ne réalise en réalité aucun gain en efficacité. En effet, même si les threads peuvent être exécutés en parallèle par plusieurs coeurs, les incréments sur `compteur` ne se font pas en réalité pas en parallèle, à cause du mutex, et l'exécution doit en plus payer un léger changement de contexte à chaque changement de fil d'exécution (il n'est pas nécessaire de changer tout le contexte, mais au moins les appels à réaliser par le fil courant). Sur un grand nombre d'itérations, comme c'est le cas ici, on perd alors beaucoup en performance.

Pour améliorer les performances on va utiliser des compteurs locaux pour chacun des fils. On cumulera ensuite ce résultat dans `compteur`.

Question 17 Ecrire une nouvelle fonction `g`, qui initialise une variable locale `long unsigned int local` à 0. Dans une boucle, `g` doit incrémenter la variable `local` autant de fois que la valeur de `objectif`. Après la boucle, `g` doit ajouter le résultat `local` à `compteur`.

Question 18 Quelle section critique peut-t-on trouver dans `g` ? Protéger cette fonction à l'aide d'un mutex. (On peut réutiliser le mutex `m`, comme on ne fait plus appel à `f`.)

Question 19 Compiler cette nouvelle version et observer le nouveau temps d'exécution.

Question 20 Reproduire toute cette section en OCaml. On sera particulièrement attentif à la manière de gérer un compteur global en OCaml.

III Listes chaînées

Nous allons chercher dans cette partie à observer sur un exemple (de programme déjà écrit) l'importance de l'utilisation des mutex lorsqu'un programme exécute plusieurs threads en parallèle, et réfléchir à la manière de déterminer une section critique de manière judicieuse dan sun programme.

[1] On pourrait aussi déclarer individuellement quatre fils dans quatre variables séparées, mais il faut s'habituer à travailler avec des structures de données comportant des fils d'exécution, pour varier facilement le nombre de fils qu'on utilisera.

On fournit pour cette partie deux fichiers sources déjà complets : `texte_aleatoire.c` et `lecture.c` ; ces deux fichiers peuvent être récupérés sur la page Cahier-de-Prépa du cours, ou bien sur le réseau du lycée.

Le fichier `texte_aleatoire.c` contient un code qui crée un fichier `noms` s'il n'existe pas déjà, puis le remplit de lignes contenant au plus 20 caractères. Le nombre de lignes à remplir est donné en argument lors de l'exécution.

Question 21 Compiler le fichier `texte_aleatoire.c`. Nous appellerons `text` l'exécutable dans la suite.

Question 22 Exécuter ce dernier en donnant un nombre de lignes en argument. Par exemple, `./text 10` et `./text 25`. Dans les deux cas, observer le contenu du fichier `noms` obtenu.

Le fichier `lecture.c` contient :

- un type `liste`, représentant un pointeur sur un type structuré `struct cellule` ; on s'en sert pour représenter une liste chaînée dont chaque élément est une chaîne de caractère, de type `char*`
- une variable globale `lg`, représentant le pointeur de tête d'une liste chaînée.
- une fonction `f` utilisée divers fils d'exécution, qui va lire une ligne du fichier `noms` tant que c'est encore possible (tant qu'il reste des lignes à lire), puis ajouter à la liste chaînée `lg` un élément contenant la chaîne de caractères qui vient d'être lue.

Le corps de ce fichier consiste en l'exécution de `f` par 8 threads en simultané, qui vont tenter de lire les lignes du fichier `noms` généré par le programme précédent, et les ajouter dans la liste chaînée `lg`. A la fin, le programme compte le nombre d'éléments dans la liste.

Question 23 Compiler le fichier `lecture.c`. Nous appelerons `lect` l'exécutable dans la suite.

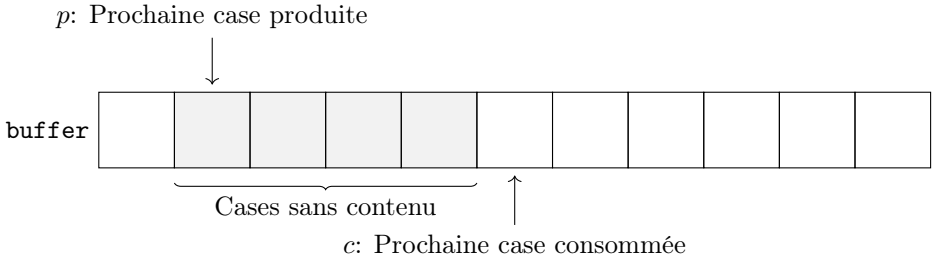
Question 24 Générer un fichier `noms` contenant un grand nombre de lignes (par exemple, `10000` ou `20000`) avec l'exécutable `text`. Puis exécuter `lect`. Qu'observe-t-on ?

Question 25 Quelle est la section critique qui doit être protégée par un mutex dans la fonction `f` ? Modifier le code pour protéger cette section par un mutex, puis recompiler et réexécuter `lect`. Qu'observe-t-on ?

IV Sémaphores : problème des producteurs et consommateurs

Le but de cette partie est d'apprendre à manier les sémaphores, en réimplémentant la solution proposée pour le problème des producteurs et consommateurs vus en cours. On n'oubliera pas d'inclure `stdio.h` dans le code à utiliser. On fournit le fichier `prod-conso.c`, à récupérer et à compléter.

On rappelle le problème :



- Les *producteurs* sont des fils qui produisent des données qui sont stockées dans le buffer, i.e. remplissent une case. Si ce dernier est plein, le producteur doit attendre qu'une case soit libérée (par un consommateur).
- Les *consommateurs* sont des fils consomment les données du buffer, i.e. libèrent une case. Si celui-ci est vide, le consommateur doit attendre qu'une case soit remplie (par un producteur).
- Un seul fil peut accéder au buffer à la fois (exclusion mutuelle).
- Le buffer est initialement vide.

Et voici la solution proposée dans le cours, où n est la taille maximale du buffer circulaire stocké dans un tableau. Cette solution utilise un mutex et deux sémaphores.

Algorithme 1 : Fil Producteur	**Algorithme 2 :** Fil Consommateur
Génère une donnée x	P(remplies_s)
P(libres_s)	lock(buffer_m)
lock(buffer_m)	$x \leftarrow$ `buffer`$[c]$
`buffer`$[p] \leftarrow x$	$c \leftarrow (c+1) \bmod n$
$p \leftarrow (p+1) \bmod n$	unlock(buffer_m)
unlock(buffer_m)	V(libres_s)
V(remplies_s)	Utilise la donnée x

Nous allons implémenter des fils producteurs et consommateurs basés sur ce modèle.

Question 26 Déclarer deux sémaphores `libres_s` et `remplies_s`, ainsi qu'un mutex `buffer_m`. On les déclarera en variable globale, puis on les initialiser dans la fonction `main`.

Le tableau représentant le buffer est déjà défini en variable globale, et est alloué dans la fonction `main`. On fournit également trois variables globales représentant la taille maximale du buffer, l'indice de la prochaine case où se trouve une donnée à consommer, et l'indice de la prochaine case où se trouve une donnée à écrire.

Question 27 Ecrire une fonction `void *prod(void *arg)` correspondant à une exécution de l'algorithme de producteur donné ci-dessus. On supposera que la donnée `x` est un entier entre `0` et `99` généré aléatoirement.

Question 28 Ecrire une fonction `void *cons(void *arg)` correspondant à une exécution de l'algorithme de consommateur donné ci-dessus. On supposera que la donnée `x` est récupérée par le consommateur, mais n'est pas utilisée.

Question 29 Dans la fonction `main`, déclarer un tableau `producteurs` et un tableau `consommateurs` contenant `100` fils d'exécution chacun.

Question 30 Initialiser les fils d'exécutions des tableaux précédents à l'aide de `pthread_create`, en leur faisant exécuter la fonctoin `prod` ou la fonction `cons` selon le cas.

Question 31 Compiler et exécuter le programme. On pourra visualiser l'évolution du buffer en appelant la fonction `afficher` (fournie dans le fichier) à des endroits adaptés.

Question 32 Que se passe-t-il si on crée `100` fils producteurs et `200` fils consommateurs ? Expliquer pourquoi, en se basant sur les opérations `sem_post` et `sem_wait`. Même question si on utilise `200` producteurs et `100` consommateurs.

V Problème du rendez-vous généralisé (facultatif)

On veut généraliser le problème du rendez-vous à N fils d'exécution tous identiques. On souhaite que les fils exécutent un certain code a puis se donnent rendez-vous avant d'exécuter un code b. On travaillera pour cela dans un nouveau fichier `rdv.c` qu'on prendra soin de créer.

Question 33 Proposer un algorithme basé tel que, si les N fils exécutent ce même algorithme, le problème du rendez-vous est correctement résolu.

Indice 1 : chaque fil signale qu'il a terminé son exécution de a, mais seul le dernier à avoir fini va libérer les autres.

Indice 2 : on se servira d'un compteur, d'un mutex et d'un sémaphore.

Question 34 Implémenter une fonction `void *f(void *arg)` implémentant l'algorithme proposé.
On pourra récupérer le nombre N à travers l'argument `arg`, en interprétant celui-ci comme un pointeur sur `int` : le case `* (int *)arg` permet de récupérer le contenu du pointeur en l'interpréetant comme un `int`.

Question 35 A l'instar des sections précédentes, créer un tableau de fils d'exécution, par exemple de taille `10`. Puis faire exécuter à tous ces fils la fonction `f` précédente.

Question 36 Visualiser l'exécution. On pourra choisir, dans `f`, de représenter la section correspondant à a (resp. b) par un affichage du caractère `a` (resp. `b`).

Annexes

Utilisation des machines virtuelles

Voici la marche à suivre pour utiliser les machines virtuelles présentes sur le réseau du lycée :

- Ouvrez `VirtualBox` (présent sur le bureau, ou cherchez le dans les applications).
- Allez dans `Ajouter`, puis cherchez la machine virtuelle nommée Proanos 1.1 dans le répertoire `D:/` de la machine physique.
- Chargez la dans VirtualBox, puis lancez la. Le mot de passe demandé est `concours`.
- Dans le répertoire `D:/` de la machine physique se trouve un dossier qui est partagé avec la machine virtuelle. Vous pouvez utiliser ce dossier pour transférer des fichiers de/vers la machine virtuelle, en les récupérant dans le dossier `sf_sortie` de la machine virtuelle.

Compilation et exécution

Terminal

- L'ouverture du terminal dans la machine virtuelle se fait avec `Ctrl+Alt+T`.
- Une fois dans le terminal, vous pouvez vous déplacer de dossier en dossier avec la commande `cd`. Le dossier parent du dossier courant est désigné par `..`, et on y remonte donc avec `cd ..`
- La commande `ls` affiche les dossiers et fichiers existants dans le dossier courant.
- Pour lancer un fichier exécutable `file`, résultant d'une compilation, il suffit de taper `./file`

Compilation C

Ce TP nécessite l'utilisation des bibliothèque `<pthread.h>` et `<semaphore.h>`. On compilera les programmes avec

```
gcc -pthread -Wall -Wextra -o nom_executable source.c
```

On pourra rajouter les options `-Wall -Wextra` pour obtenir des avertissement en plus si nécessaire.

Compilation OCaml

On compilera les programmes avec

```
ocamlc -I +unix -I +threads unix.cma threads.cma -o nom_executable source.ml
```

Mutex et sémaphores en C.

Les fonctions suivantes permettent de manipuler les fils d'exécution en C.

Fonction	Description

`int pthread_create(`
`pthread_t*,`
`pthread_attr_t*,`
`void *(* f)(void *),`
`void *)`
	Un appel `pthread_create(&t, &attr, f, v)` crée un fil d'exécution, qui est stocké dans `t` et qui exécute l'appel à `f(v)`, et renvoie 0 si la création du fil est un succès, un code d'erreur sinon.

`pthread_exit(void *ret)`	Termine le fil d'exécution effectuant cet appel en renvoyant la valeur passée en argument.
`pthread_join(pthread_t, void**)`	L'appel `pthread_join(t, &ret)` suspend l'exécution du fil d'exécution jusqu'à ce que le fil d'exécution `t` est terminé (par un `pthread_exit`). Le pointeur sur la valeur de retour remplie par `pthread_exit` est affecté à `ret`.

Les fonctions suivantes permettent de manipuler les mutex en C.

Fonction	Description
`int pthread_mutex_init( pthread_mutex_t*, const pthread_mutexattr_t*)`	Un appel `pthread_mutex_init(&m, &attr)` initialise le mutex `m` avec l'attribut `attr`. On se contentera d'attributs `NULL`, avec les effets suivants : si `m` est déjà verrouillé et qu'un autre fil tente de le verrouiller, il est bloqué.
`int pthread_mutex_lock( pthread_mutex_t* mutex)`	Un appel `pthread_mutex_lock(&m)` verrouille le mutex `m`. Si `m` est déjà verrouillé, le fil essayant de le verrouiller est mis en attente.
`int pthread_mutex_unlock( pthread_mutex_t* mutex)`	Un appel `pthread_mutex_unlock(&m)` déverrouille le mutex `m`. Si des fils sont en attente pour verrouiller `m`, ils sont désormais en concurrence pour le verrouiller.

Les fonctions suivantes permettent de manipuler les sémaphores en C.

Fonction	Description
`sem_init(sem_t *s, int pshared, unsigned int val)`	Initialise le sémaphore `s` à la valeur fournie. `pshared` sera toujours 0 dans nos applications.
`sem_wait(sem_t *s)`	Opération `P`: le fil d'exécution acquiert un jeton du sémaphore ou est mis en attente s'il n'y en a plus.
`sem_post(sem_t *s)`	Opération `V`: rend un jeton du sémaphore ou réveille un fil d'exécution qui était en attente.

Mutex et sémaphores en OCaml

Les fonctions suivantes permettent de manipuler les fils d'exécution en OCaml.

Fonction	Description
`Thread.create : ('a -> 'b) -> 'a -> Thread.t`	`Thread.create f v` crée un fil d'exécution pour exécuter l'appel `f v`. Le retour de l'appel de `f` ne sera pas utilisé. La fonction `create` renvoie un pointeur sur le fil d'exécution créé.
`Thread.exit : unit -> unit`	Termine le fil d'exécution effectuant cet appel.
`Thread.join : t -> unit`	`Thread.join t` suspend l'exécution du fil d'exécution jusqu'à ce que le fil d'exécution `t` est terminé (par un `exit`).
`Thread.yield : unit -> unit` (Hors-programme)	`Thread.yield` indique à l'ordonnanceur que le fil d'exécution qui exécute cette instruction peut être interrompu si nécessaire. Cela peut être utile pour forcer des instructions à s'entrelacer et permettre de se débuguer..

Les fonctions suivantes permettent de manipuler les mutex en OCaml.

Fonction	Description
`Mutex.create : unit -> Mutex.t`	`Mutex.create ()` crée un nouveau mutex (un verrou) qu'on peut ensuite appliquer dans les threads.
`Mutex.lock : Mutex.t -> unit`	Un appel `Mutex.lock m` verrouille le mutex `m`. Un mutex ne peut être verrouillé que par un thread à la fois : si t_1 verrouille m, alors t_2 doit attendre que t_1 déverrouille m avant de le verrouiller à son tour.
`Mutex.unlock : Mutex.t -> unit`	Un appel `Mutex.unlock m` déverrouille le mutex `m`. Si des processus sont en attente pour verrouiller `m`, ils sont désormais en concurrence pour le verrouiller.

Les fonctions suivantes permettent de manipuler les sémaphores en OCaml.

Fonction	Description
`Semaphore.Counting.make : int -> Semaphore.Counting.t`	`Semaphore.Counting.make v` renvoie un nouveau sémaphore dont le compteur est initialisé à `v`.
`Semaphore.Counting.acquire : Semaphore.Counting.t -> unit`	Opération `P`: avec un appel `Semaphore.Counting.acquire s`, le fil d'exécution décrémente le compteur de `s` s'il est non-nul, ou est mis en attente dans la file sinon.
`Semaphore.Counting.release : Semaphore.Counting.t -> unit`	Opération `V`: avec un appel `Semaphore.Counting.release s` incrémente le compteur de `s`. Ceci réveille automatiquement un fil d'exécution qui est en attente dans la file s'il y en a.