I Définitions

Définition : Programme

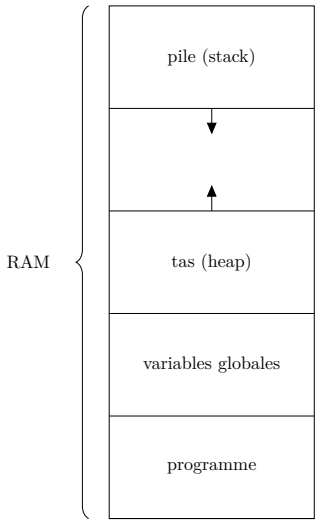
Un programme est une suite d'instructions dans un langage de programmation et stocké dans un fichier appelé code source.

Définition : Processus

Un processus est une instance d'un programme en cours d'exécution. Il est composé d'un espace mémoire, d'un identifiant de processus (PID)...

Définition : Thread

Un thread (ou fil d'exécution) est une unité d'exécution plus petite qu'un processus. Un processus peut contenir plusieurs threads, qui partagent un même espace mémoire.



Espace mémoire d'un processus.

Les threads d'un même processus partagent le même programme, le même tas et les mêmes variables globales.
Par contre, ils ont chacun leur propre pile.

Rappels :

- La pile contient les variables locales, qui sont automatiquement libérées en sortant de leurs portées. La pile est de taille fixe donc limitée (d'où l'erreur `stack overflow`).
- Le tas contient les variables allouées dynamiquement avec `malloc`. L'accès au tas est plus lent que la pile.

Définition : Programmation parallèle

Programmation parallèle (*multithreading*) : un programme exécute des threads sur plusieurs processeurs (ou cœurs) en même temps.

Intérêt : Accélérer l'exécution d'un programme.

Définition : Asynchronisme

Asynchronisme : un programme exécute des threads sur un processeur à tour de rôle.

Exemple : Éviter que l'interface graphique d'une application ne se fige pendant un calcul long.

Définition : Programmation concurrente

Programmation concurrente : un programme exécute des threads en parallèle ou de façon asynchrone.

Remarques :

- Dans les deux cas, les threads peuvent s'entrelacer.
- Une même instruction ne prend pas forcément le même temps à s'exécuter par chaque thread.

Autres notions (HP) :

- *Multiprocessing* : plusieurs processus s'exécutent en parallèle. Ils n'ont pas de variables partagées.
- Calcul distribué : plusieurs ordinateurs exécutent des processus en parallèle.

II En pratique

- En C, le parallélisme est possible avec la bibliothèque `pthread`.
- Jusqu'à la version 4 de OCaml, seule l'asynchronisme était possible avec le module `Thread`. En effet, le parallélisme est compliqué à cause du garbage collector d'OCaml (le système qui permet de libérer automatiquement la mémoire allouée sur le tas et empêche les fuites de mémoire).
  Depuis la version 5, le parallélisme est possible avec le module `Domain`.
- En Python, le module `threading` ne permet que l'asyncronisme et pas le parallélisme, pour la même raison que OCaml version 4.

exemple.c

```c
#include <pthread.h> // threads POSIX (standard Linux)
#include <stdio.h>

void* f(void* x) {
    int* n = (int*)x; // Conversion du type
    int* y = {0};
    if(*n == 1) y[1]=1;
    for(int i = 0; i < 100000; i++) {
        if(i % 20000 == 0)
            printf("%d %d\n", *n, i);
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;
    int n1 = 1, n2 = 2;
    pthread_create(&t1, NULL, f, (void*)&n1); // t1 exécute f(&n1)
    pthread_create(&t2, NULL, f, (void*)&n2); // t2 exécute f(&n2)
    pthread_join(t1, NULL); // Attendre la fin de t1
    pthread_join(t2, NULL); // Attendre la fin de t2
}
```

Compilation :

`void*` est un pointeur (adresse) vers un type quelconque (inconnu). Il permet du polymorphisme en C, comme en OCaml où une fonction peut avoir un type générique `'a` en argument.

exemple.ml

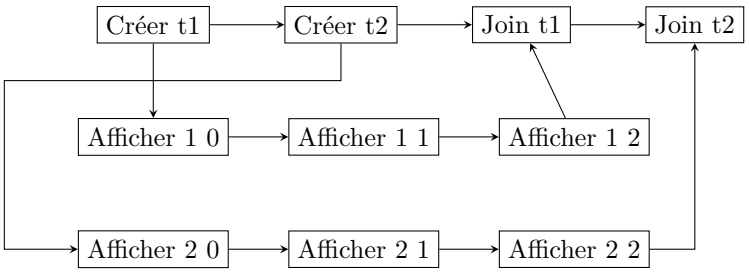
```ocaml
let f x =
  Printf.printf "Thread %d\n" x;
  for i = 0 to 2 do
      Printf.printf "%d %d\n" x i
  done

let () =
  let t1 = Thread.create f 1 in
  let t2 = Thread.create f 2 in
  Thread.join t1;
  Thread.join t2
```

Compilation :

III Graphe des exécutions

On peut représenter les exécutions possibles du programme précédent par un graphe, où un arc $u \to v$ signifie que v est exécuté après u.
Exemple de graphe pour `exemple.ml` :



Dans un programme séquentiel, on a un ordre total des instructions.

Dans un programme concurrent, on a seulement un ordre partiel, défini par la relation d'accessibilité du graphe précédent : s'il y a un chemin de u à v, alors u doit être exécuté avant v.

IV Trace

Définition : Trace

Une trace d'un programme est l'ordre dans lequel les instructions sont exécutées (qui respecte le graphe précédent).

Exemple de trace pour `exemple.ml` :

Créer t0	Afficher 0 0	Créer t1	Afficher 1 0	Afficher 1 1	Afficher 0 1	···

Remarques :

- Un même programme peut avoir plusieurs traces possibles.
- On supposera qu'il n'y a jamais deux instructions exécutés exactement en même temps, ce qui est simplification théorique.

V Cas simples de parallélisation

Rappel :

- Une structure de donnée est persistante (immutable) si elle ne peut pas être modifiée après sa création.
  Exemples en OCaml : liste, arbre binaire...
- Une structure de donnée est mutable si elle peut être modifiée après sa création.
  Exemples en OCaml : tableau, référence...

Il est facile de paralléliser un programme si les structures de données sont persistantes : il n'y a pas de conflit d'accès.

Exemple : tri fusion en parallèle, où chaque thread trie une partie du tableau.

De manière générale, il est facile de paralléliser un programme si les calculs peuvent être faits de manière indépendante : on parle de programmes trivialement parallèles (*embarassingly parallel*).

Exemple : Multiplication de matrices de taille $n \times n$. Si on utilise N threads calculant chacun $\frac{n^3}{N}$ cases du produit matriciel, on divise le temps de calcul par N.

VI Opération atomique

Définition : Opération atomique

Une opération est atomique si elle est exécutée en une seule fois, sans être interrompue.

Exemple : Une opération élémentaire (lecture ou écriture d'une variable de type `int` par exemple) est atomique.

Exercice 1.
On considère une variable n initialisée à 0 et trois fils d'exécutions qui effectuent les opérations suivantes :

- T_1 : $a \leftarrow n$, $n \leftarrow 1$, $b \leftarrow n$;
- T_2 : $c \leftarrow n$, $n \leftarrow 2$, $d \leftarrow n$;
- T_3 : $e \leftarrow n$, $f \leftarrow n$.

On suppose que l'instruction $x \leftarrow y$ consiste à écrire le contenu de y dans x et est une instruction atomique. Après l'exécution des trois fils :

1. que peut valoir c ? _____
2. que peut valoir b ? _____
3. que peut valoir e ? _____
4. si c vaut 1, que peut valoir d ? _____
5. si f vaut 0, que peut valoir e ? _____

VII Incrémentation d'un compteur

Considérons le programme suivant :

```c
int counter = 0;
void *increment(void *arg){
    for (int i = 1; i <= 1000000; i++)
        counter++;
}
int main(){
    pthread_t t0, t1;
    pthread_create(&t0, NULL, increment, NULL);
    pthread_create(&t1, NULL, increment, NULL);
    pthread_join(t0, NULL);
    pthread_join(t1, NULL);
}
```

Contrairement à ce qu'on pourrait penser, `counter` ne vaut pas forcément 2000000 à la fin du programme. En effet, l'opération `counter++` n'est pas atomique, car elle est équivalente à :

```c
int tmp = counter;
tmp = tmp + 1;
counter = tmp;
```

```
movl    counter(%rip), %eax
addl    $1, %eax
movl    %eax, counter(%rip)
```

On peut ainsi avoir le début de trace suivante où `counter` vaut 1 au lieu de 2 :

`tmp = counter;`	`tmp = counter;`	`tmp = tmp + 1;`	`counter = tmp;`	`tmp = tmp + 1;`	`counter = tmp;`
1	2	2	2	1	1

Remarque : Seule la variable globale `counter` est partagé par les deux threads, les variables locales `i` et `tmp` sont propres à chaque thread.

Exercice 2.
Quelles sont les valeurs possibles de `counter` à la fin du programme ?

_____

_____

_____

Exercice 3.
Quelle est la valeur minimum et maximum de `counter` si on utilise k threads ?

Exercice 4 *Activation de processus (type A)*

Soit un système temps réel à n processus asynchrones $i \in [\![1, n]\!]$ et m ressources r_j. Quand un processus i est actif, il bloque un certain nombre de ressources listées dans un ensemble P_i et une ressource ne peut être utilisée que par un seul processus. On cherche à activer simultanément le plus de processus possible.

Le problème de décision `ACTIVATION` correspondant ajoute un entier k aux entrées et cherche à répondre à la question : "Est-il possible d'activer au moins k processus en même temps ?"

1. Soit $n = 4$ et $m = 5$. On suppose que $P_1 = \{r_1, r_2\}$, $P_2 = \{r_1, r_3\}$, $P_3 = \{r_2, r_4, r_5\}$ et $P_4 = \{r_1, r_2, r_4\}$. Est-il possible d'activer 2 processus en même temps ? Même question avec 3 processus.
2. Dans le cas où chaque processus n'utilise qu'une seule ressource, proposer un algorithme résolvant le problème `ACTIVATION`. Évaluer la complexité de votre algorithme.

On souhaite montrer que `ACTIVATION` est NP-complet.

3. Donner un certificat pour ce problème.
4. Écrire en pseudo code un algorithme de vérification polynomial. On supposera disposer de trois primitives, toutes trois de complexité polynomiale :
   (a) `appartient(c,i)` qui renvoie `Vrai` si le processus `i` est dans l'ensemble d'entiers `c`.
   (b) `intersecte(P`$_i$`,R)` qui renvoie `Vrai` si le processus `i` utilise une ressource incluse dans un ensemble de ressources `R`.
   (c) `ajoute(P`$_i$`,R)` qui ajoute les ressources `P`$_i$ dans l'ensemble `R` et renvoie ce nouvel ensemble.

En théorie des graphes, le problème `STABLE` se pose la question de l'existence dans un graphe non orienté $G = (S, A)$ d'un ensemble d'au moins k sommets ne contenant aucune paire de sommets voisins. En d'autres termes, existe-t-il $S' \subset S$, $|S'| \geq k$ tel que $s, p \in S' \Rightarrow (s, p) \notin A$?

5. En utilisant le fait que `STABLE` est NP-complet, montrer par réduction que le problème `ACTIVATION` est également NP-complet.

Commentaires du jury

1. On attend du candidat une réponse en oui / non avec une précision de quels processus activer dans le cas où la réponse est oui et une très rapide justification dans le cas où la réponse est non. Une réponse orale suffit.
2. De multiples réponses sont possibles sur cette question et sont acceptées du moment qu'elles sont clairement décrites et correctement analysées. On n'attend pas une complexité optimale.
3. Une courte phrase décrivant la nature d'un tel certificat et indiquant sa polynomialité en la taille de l'entrée suffit à obtenir tous les points.
4. On attend un algorithme utilisant à bon escient les primitives proposées par le sujet.
5. Le jury est attentif au fait que tous les arguments soient bien présents : description de la réduction, preuve que c'en est une et justification de son caractère polynomial pour le caractère NP-difficile ; caractère NP pour pouvoir conclure quant à la NP-complétude.