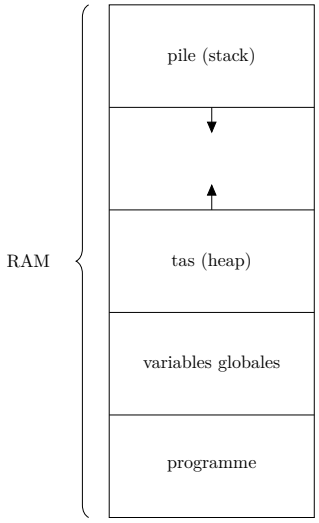
I Définitions

Définition : Programme

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

Définition : Processus

Définition : Thread (ou fil d'exécution)



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

Même avec un seul cœur, on peut entrelacer les instructions des différents processus ou threads.
Exemple : Gestion des applications sur un système d'exploitation. Les applications sont exécutées en alternance.

Définition : Programmation concurrente

II Exemples en C et OCaml

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

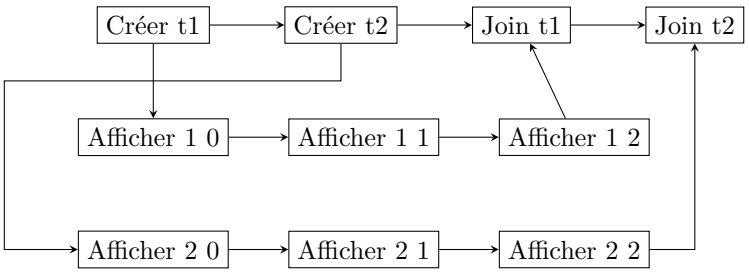
```ocaml
let f x =
  Printf.printf "Thread %d\n" x;
  for i = 0 to 2 do
      Printf.printf "%d %d\n" x i
  done

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 le programme précédent :



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

Exemple de trace pour le programme précédent :

Créer t1	Afficher 1 0	Créer t2	Afficher 2 0	Afficher 2 1	Afficher 1 1	···

V Programme trivialement parallèle (Embarassingly parallel)

Le parallèlisme est particulièrement simple quand les calculs peuvent être faits de manière indépendante.

Exercice 1.
Décrire un programme calculant le produit de deux matrices de manière parallèle. Quel est le gain de temps par rapport à une version séquentielle ?

VI Opération atomique

Définition : Opération atomique

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

Exercice 2.
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

Exercice 3.
Quelles sont les valeurs possibles de `counter` à la fin du programme ?

Exercice 4.
Quelle est la valeur minimum et maximum de `counter` si on utilise k threads ?