

Concurrency

Quentin Fortier

February 1, 2026

Processus et thread

Définition

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

Processus et thread

Définition

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

Définition

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)...

Processus et thread

Définition

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

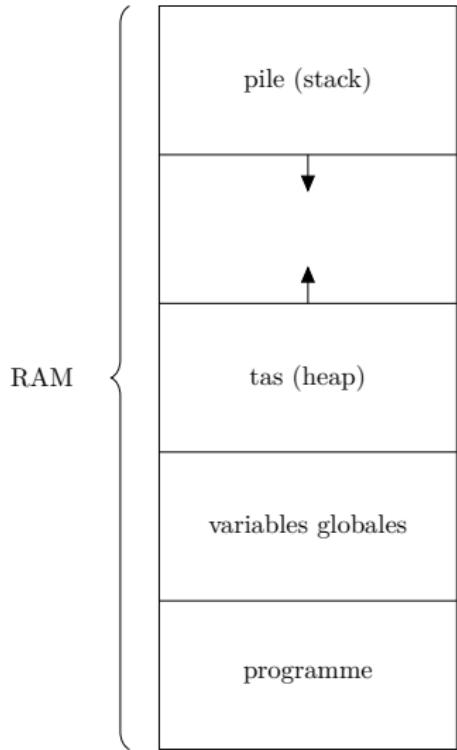
Définition

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

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.

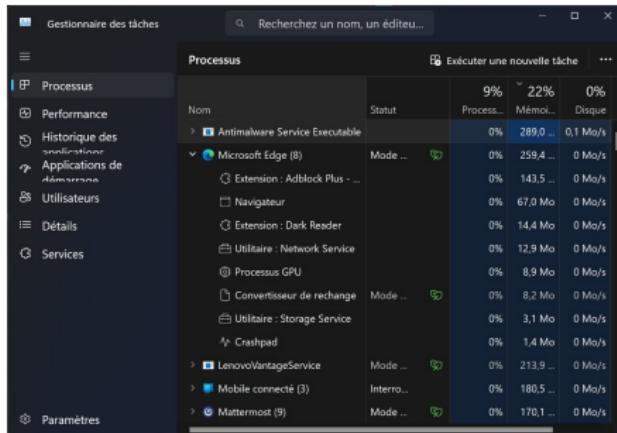
Par contre, ils ont chacun leur propre pile.

Espace mémoire d'un processus

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.

Processeur et cœur

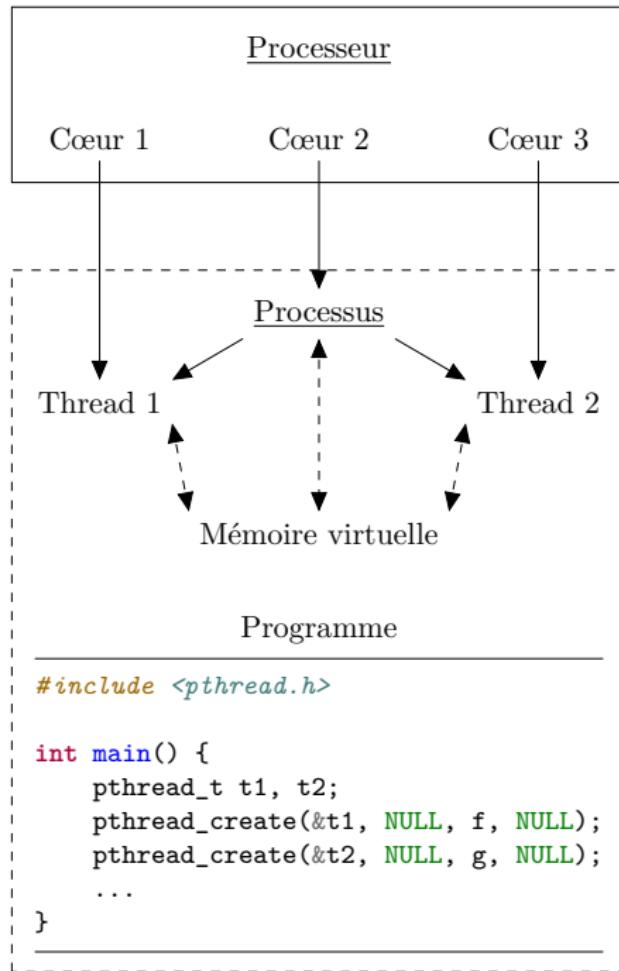


Gestionnaire des tâches de Windows

```
top - 14:14:20 up 1:06, 0 user, load average: 0.76, 0.86, 0.94
Tasks: 20 total, 1 running, 19 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.9 us, 0.3 sy, 0.8 ni, 98.6 id, 0.0 wa, 0.0 hi, 0.1 si, 0.0 st
MiB Mem : 7841.2 total, 5674.8 free, 1651.7 used, 688.6 buff/cache
MiB Swap: 1024.0 total, 1024.0 free, 0.0 used. 6189.5 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
354	user	20	0	13.7g	671860	55552	S	1.7	8.4	1:47.69	node
20717	user	20	0	287852	4700	2176	S	0.7	0.1	0:00.02	sleep
1	user	20	0	287540	4440	1920	S	0.3	0.1	0:02.39	sh
232	user	20	0	1630204	129588	40192	S	0.3	1.6	0:15.87	node
244	user	20	0	1296432	71888	34816	S	0.3	0.9	0:03.55	node
405	user	20	0	1411252	88720	38016	S	0.3	1.1	0:28.51	node
21	root	20	0	287528	4576	2048	S	0.0	0.1	0:00.01	sh
26	user	20	0	287544	4576	2048	S	0.0	0.1	0:00.05	sh
33	root	20	0	287500	4452	1920	S	0.0	0.1	0:00.00	sh
46	root	20	0	287540	4580	2048	S	0.0	0.1	0:00.01	sh
112	user	20	0	287548	4580	2048	S	0.0	0.1	0:00.01	sh

Commande top sous Linux/macOS



Concurrence et parallélisme

Définition

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

Intérêt : Accélérer l'exécution d'un programme.

Concurrence et parallélisme

Définition

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 : 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.

Concurrence et parallélisme

Définition

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

Concurrence et parallélisme

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.

Concurrence et parallélisme

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 variable partagée.
- Calcul distribué : plusieurs ordinateurs exécutent des processus en parallèle.

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'asynchronisme et pas le parallélisme, pour la même raison que OCaml version 4.

En pratique

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

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

int main() {
    pthread_t t0, t1;
    int n1 = 1, n2 = 2;
    pthread_create(&t0, NULL, f, (void*)&n1);
    pthread_create(&t1, NULL, f, (void*)&n2);
    pthread_join(t0, NULL); // Attendre la fin de t0
    pthread_join(t1, NULL); // Attendre la fin de t1
}
```

Compilation : gcc -pthread exemple.c

En pratique

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.

Threads en OCaml

```
let f x =...  
let t = Thread.create f x  
Thread.join t (* Attendre la fin de t* )
```

Threads en OCaml

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

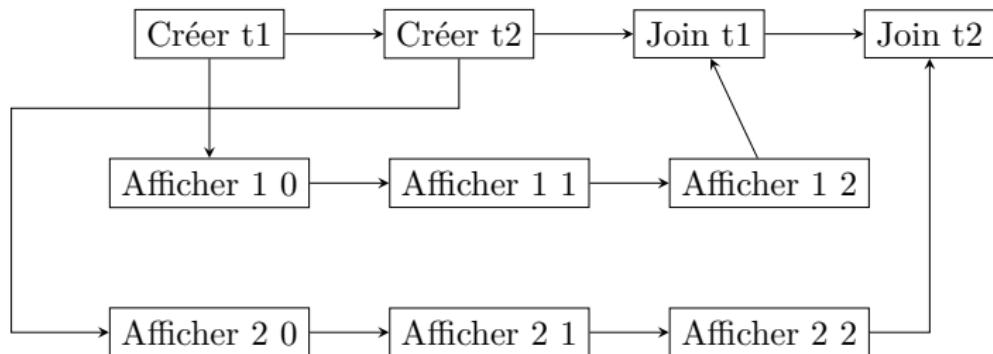
let () =
  let t0 = Thread.create f 0 in
  let t1 = Thread.create f 1 in
  Thread.join t0;
  Thread.join t1
```

Compilation :

```
ocamlopt -I +unix -I +threads unix.cmxo threads.cmxo
exemple.ml
```

Graphe des exécutions

On peut représenter les exécutions possibles du programme précédent par un graphe, où un arc $u \rightarrow v$ signifie que v est exécuté après u .



Graphe des exécutions

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

Graphe des exécutions

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 .

Trace

Définition

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

Une trace possible :

Créer t0	Afficher 0 0	Créer t1	Afficher 1 0	Afficher 1 1	Afficher 0 1	...
----------	--------------	----------	--------------	--------------	--------------	-----

Trace

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.

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.

Cas simples de parallélisation

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 (*embarrassingly 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 .

Opération atomique

Définition

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

Opération atomique

Définition

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

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

Opération atomique

Exercice

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 ?

Incrémentation d'un compteur

```
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);
}
```

Question

Quelle est la valeur de counter à la fin de l'exécution de main ?

Incrémentation d'un compteur

```
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);
}
```

Question

Quelle est la valeur de counter à la fin de l'exécution de main ?

Pas forcément 2000000...

Incrémantion d'un compteur

counter++ n'est pas atomique, car elle est traduite en assembleur par :

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

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

Incrémentation d'un compteur

```
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);
}
```

Début de trace 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

La valeur maximum de counter est

La valeur maximum de counter est 2000000.

La valeur minimum de counter est

La valeur maximum de counter est 2000000.

La valeur minimum de counter est 2 :

- le thread 1 lit 0 dans counter
- le thread 2 fait $n - 1$ itérations
- le thread 1 écrit 1 dans counter
- le thread 2 lit 1 dans counter
- le thread 1 fait $n - 1$ itérations
- le thread 2 écrit 2 dans counter

Répété $n - 1$ fois					
tmp = counter;	tmp = counter;	tmp = tmp + 1;	counter = tmp;	tmp = tmp + 1;	counter = tmp;
1	2	2	2	1	1

Répété $n - 1$ fois					
tmp = counter;	tmp = counter;	tmp = tmp + 1;	counter = tmp;	tmp = tmp + 1;	counter = tmp;
2	1	1	1	2	2

Incrémantion d'un compteur

Exercice

Quelle est la valeur minimum et maximum de counter si on utilise k threads ?

Incrémantion d'un compteur

Remarque : Il existe des types et structures de données qui garantissent des opérations atomiques, comme `std::atomic<int> counter;` en C++ et `Atomic` en OCaml.

Mais cela ne permet pas de définir des sections critiques.