Etats transistions et Threads

SYE

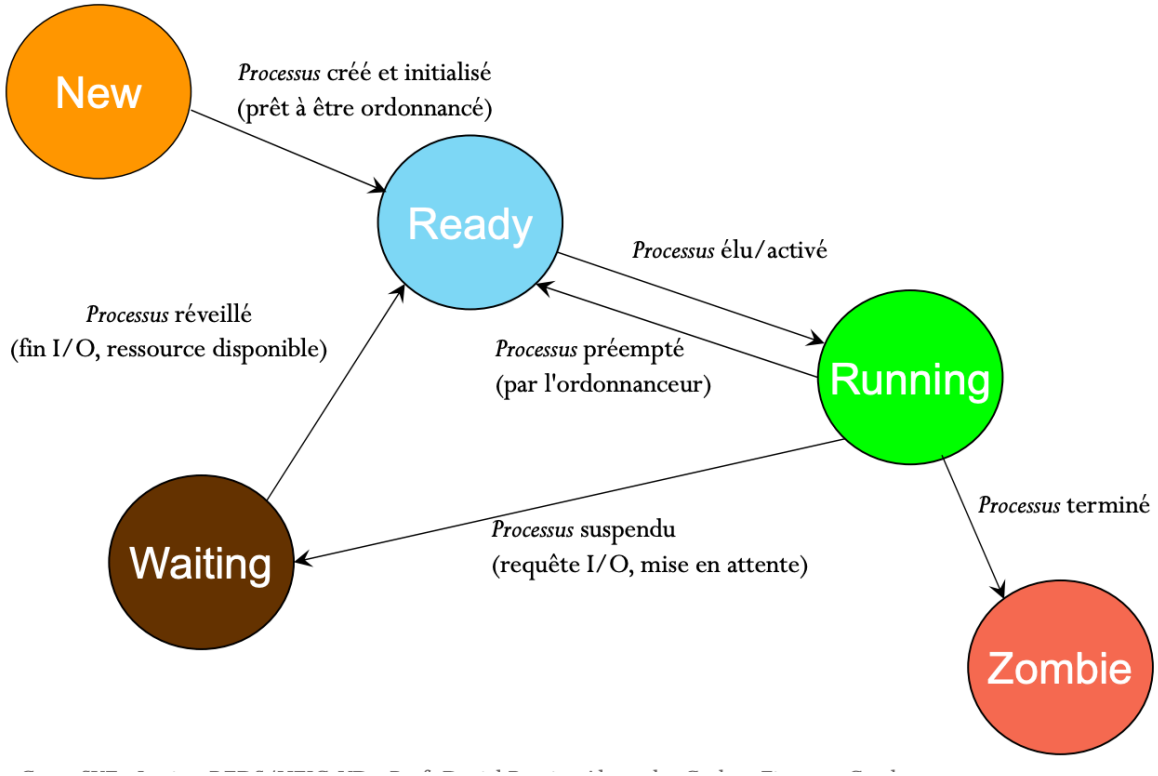
5 - Etats transition et Threads

Résumé du document

Definition

Table des matières

1. Etats et transitions



Voici une explication de chacun des états d'un processus mentionnés dans le texte :

1.1. État New

- Description : Un processus est dans l'état **new** lorsqu'il est en phase d'initialisation. Cela signifie qu'il a été créé, mais pas encore totalement configuré pour être exécuté.
- Caractéristiques : Dans cet état, le PCB (Process Control Block) du processus existe, mais toutes les structures de données nécessaires ne sont pas encore initialisées. Le processus n'est pas prêt à être ordonnancé (c'est-à-dire, il ne peut pas encore être exécuté par le processeur).

1.2. État Ready

- Description : Le processus est dans l'état **ready** lorsqu'il est prêt à être exécuté mais qu'il ne peut pas le faire en raison de la disponibilité limitée du processeur.
- Caractéristiques : À ce stade, toutes les structures de données nécessaires sont initialisées, et le processus est en attente de l'ordonnanceur, qui décidera quand lui accorder du temps CPU. Un processus peut passer de l'état **new** à **ready** après son initialisation.

1.3. État Running

- Description : Dans cet état, le processus est effectivement en cours d'exécution sur le processeur.
- Caractéristiques : Le processus a été sélectionné par l'ordonnanceur et utilise le CPU pour exécuter ses instructions. Il peut passer à un autre état en fonction d'événements (comme une interruption ou une demande de ressource).

1.4. État Waiting

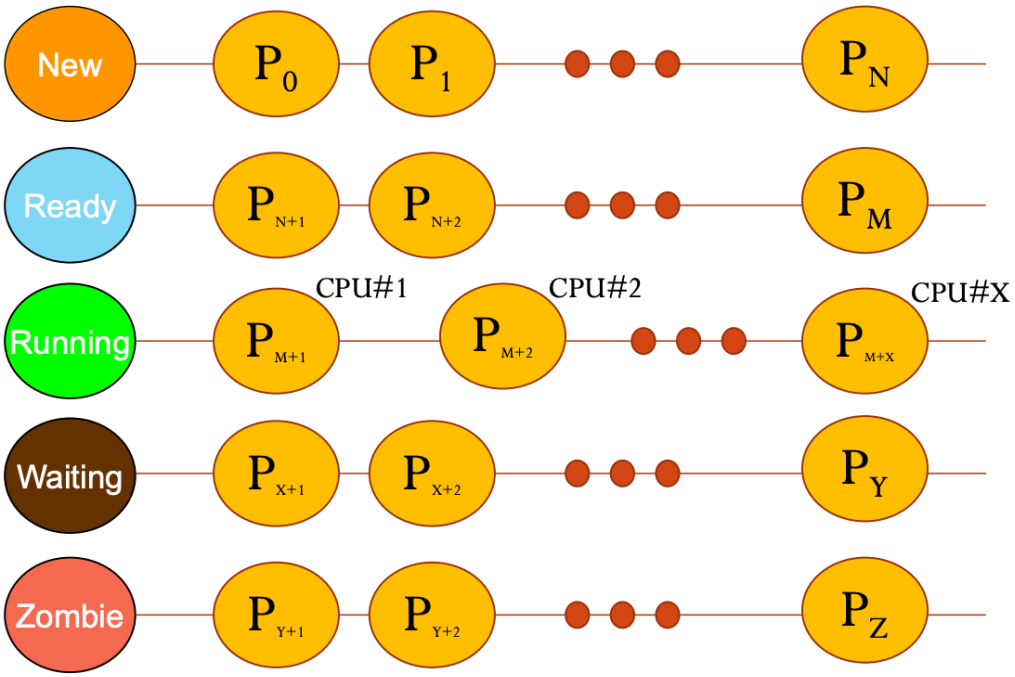
- Description : Un processus entre dans l'état **waiting** lorsqu'il attend une ressource nécessaire pour continuer son exécution, comme l'accès à un fichier, à une entrée/sortie, ou à une autre ressource logique ou physique.
- Caractéristiques : Dans cet état, le processus ne peut pas être exécuté. Lorsqu'il obtient la ressource qu'il attendait, il doit d'abord passer par l'état **ready** avant de pouvoir redevenir **running**.

1.5. État Zombie

- Description : Cet état se produit lorsqu'un processus a terminé son exécution, mais son PCB est encore présent dans le système.

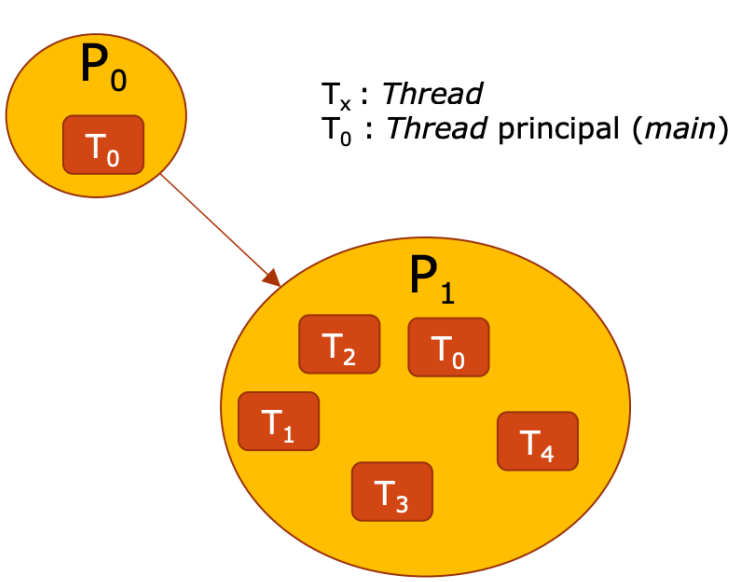
- Caractéristiques : Un processus zombie ne peut plus être ordonnancé et reste dans cet état pour permettre au processus parent de récupérer son code de sortie. Le processus parent peut alors lire les informations sur l'état du processus terminé (générées par l'appel système `exit()`) avant de libérer le PCB et de supprimer définitivement le processus du système.

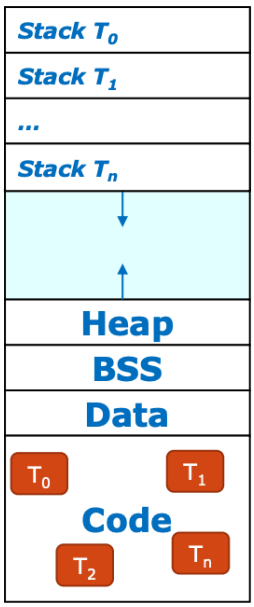
Ces états permettent au système d'exploitation de gérer efficacement les processus, en assurant une utilisation optimale des ressources et en maintenant un contrôle sur l'exécution des programmes.



En principe, il existe dans le noyau une file de processus par état (à l'exception de l'état running auquel peut être associé plusieurs processus que si la machine est multi-cœur). Les processus dans l'état ready sont des processus prêts à être ordonnancés. Les mouvements de cette file dépendra en particulier de l'ordonnanceur.

2. Définition d'un thread





2.1. Accès aux Ressources

- Contexte d'Exécution : Un thread représente un contexte d'exécution au sein d'un processus et doit avoir accès à toutes les ressources allouées à ce processus, comme la mémoire et d'autres ressources logiques.
- Espace d'Adressage : Tous les threads d'un même processus partagent le même espace d'adressage. Cela signifie qu'ils ont accès aux différentes sections du processus, comme le code, les données dans les sections data et bss, et le tas (heap).
- Pile : Chaque thread a sa propre pile, qui fait partie de son contexte d'exécution. Cela permet à chaque thread de gérer ses propres variables locales et ses appels de fonction.

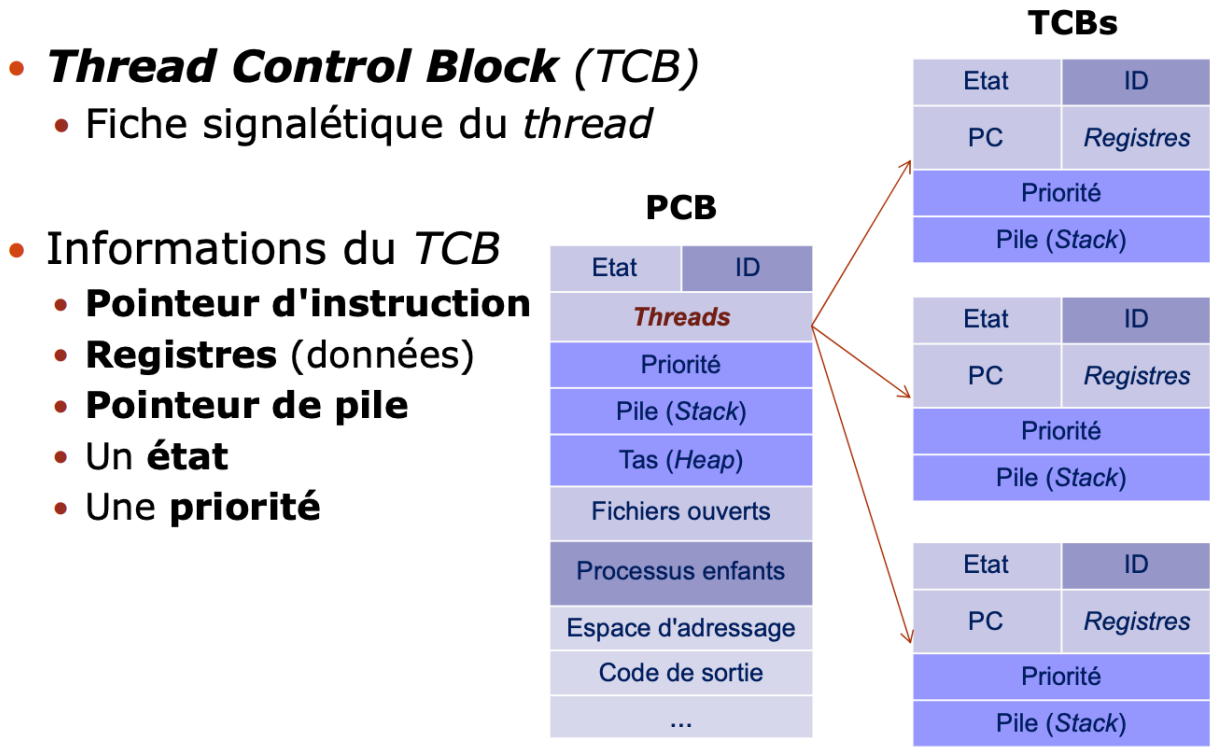
2.2. Exemple de Threads

Un exemple illustratif est le traitement de texte. Lorsqu'un utilisateur saisit du texte, le traitement de texte peut lancer un correcteur orthographique en arrière-plan pour souligner les fautes. Les deux tâches — la saisie et le correcteur — sont indépendantes, mais elles doivent accéder au même contenu.

2.3. Gestion des Accès Concurrents

Les deux threads associés à ces tâches doivent gérer les accès concurrents au contenu partagé. Pour ce faire, ils peuvent utiliser des objets de synchronisation, tels que des mutex ou des sémaphores, afin de s'assurer que les données sont correctement gérées sans conflit.

2.4. Thread Control Break (TCB)



Le TCB est une structure de données contenant les informations propres au thread, comme son état, sa priorité, le pointeur de pile, etc. Il est **fortement** sollicité lors d'un changement de contexte.

Par ailleurs, on remarque que le TCB est beaucoup plus petit que le PCB ; il ne contient aucune information particulière concernant les ressources qui pourraient être utilisées par le thread, comme des fichiers ouverts ou d'autres objets de synchronisation ou de communication.

3. Librairie de threads POSIX

- Portable Operating System Interface (X pour uniX/linuX/macosX)
- API standardisée (IEEE 1003.1c) largement répandue pour les threads
- Création/terminaison
- Synchronisation
- Accès concurrents
- Supporté par Windows (sous-système POSIX)

3.1. Creation d'un thread

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *, void *(*start_routine) (void *),
void *arg);

voicd start_routine(void *);
```

3.2. Sychronisation d'un thread

```
int pthread_join(pthread_t thread, void **retval)
```

3.3. Terminaison d'un thread

Retour de fonction (pas de valeur particulière à transmettre)

```
void pthread_exit(void *retval);
```

4. Exécution de threads

Pour exécuter un thread il faut dans un premier temps créer un identifiant pour le thread avec la ligne de code suivante:

```
pthread_t hello_thread;
```

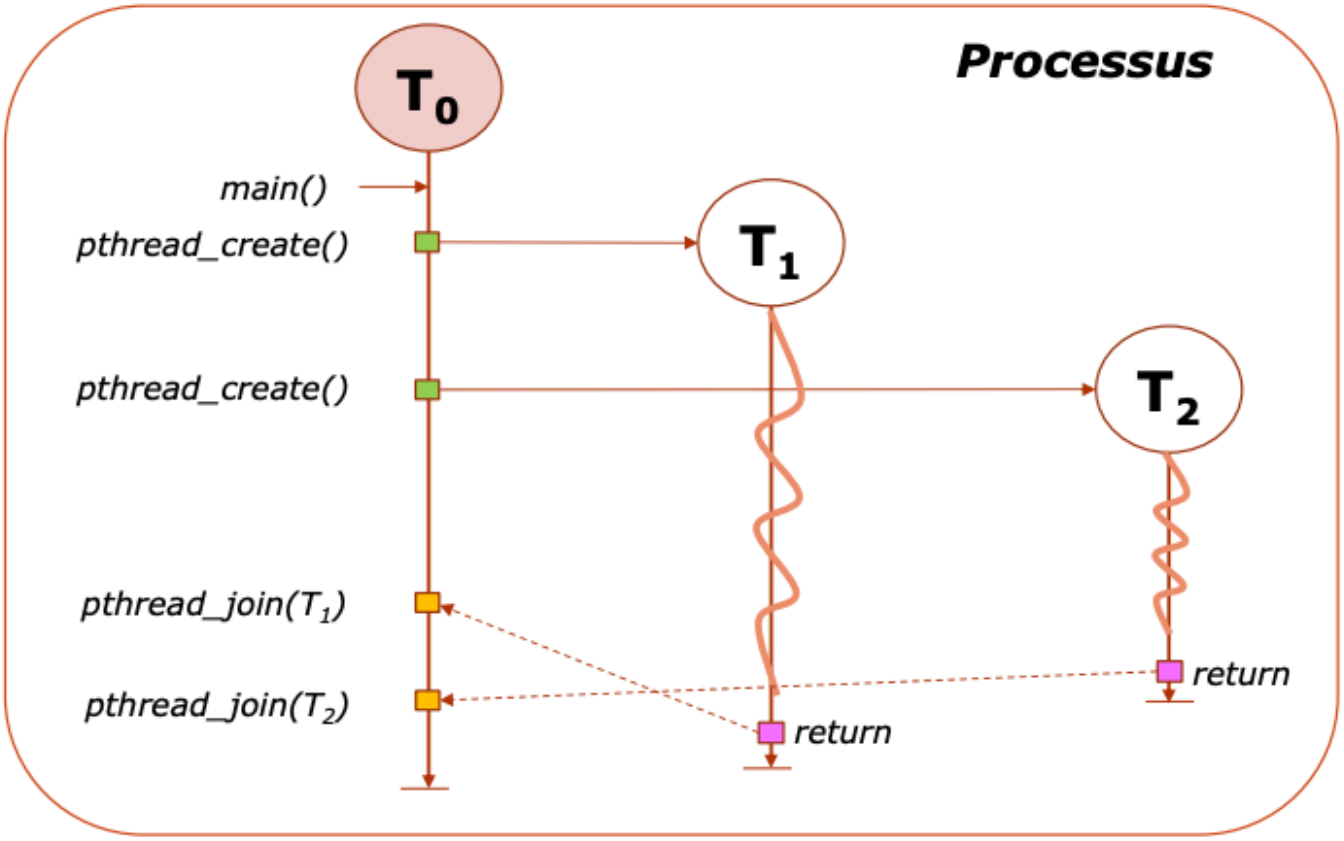
Pour ensuite démarrer le thread il faut écrire:

```
retour = pthread_create(&hello_thread, NULL, fonction, (void *) args)
```

La commande suivante permet au programme principal d'attendre la fin du thread crée:

```
pthread_join(hello_thread, NULL)
```

4.1. Séquençage des threads



Dans cet exemple, on remarque que le thread T2 se termine avant le thread T1. Cependant, le premier appel à `pthread_join()` concerne bien le premier thread. Par conséquent, lorsque le thread T2 se termine, le thread principal reste bloqué jusqu'à ce que le thread T1 se termine. Le second appel à `pthread_join()` se fera sans suspension du thread principal, car le thread T2 aura déjà terminé son exécution, et la fonction retournera immédiatement.