Introduction aux processus et leur gestion

Orestis Malaspinas, Steven Liatti

1 Avant-propos

Ce chapitre est fortement inspiré du cours en ligne *Introduction to Operating Systems* se trouvant sur https://classroom.udacity.com/courses/ud923.

2 Les processus

Dans le chapitre précédent, nous avons discuté des concepts clés des systèmes d'exploitation et comment ils gèrent le matériel pour les applications à l'aide d'abstractions et représentent les opérations effectuées sur le matériel à l'aide de différents mécanismes. Dans ce chapitre, nous allons discuter une abstraction particulière: le processus.

2.1 Qu'est-ce qu'un processus?

Une application est un programme qui se trouve sur un support physique (le disque dur, une mémoire flash, ...): elle est "statique" ou en d'autres termes elle n'est pas en train de s'exécuter. Un processus (ou tâche) est une instance d'un programme en **cours d'exécution**: le programme est chargé en mémoire et commence à s'exécuter (c'est une entité active).

Exemple 1

Une instance d'un éditeur de texte dans lequel vous êtes en train d'éditer un code pour le cours de concurrence est un processus. Un autre processus est le lecteur de musique sur lequel vous écoutez le dernier morceau de Ludwig von 88. Cela peut également être votre browser que vous avez oublié de fermer et qui tourne arrière plan. Vous pouvez voir tous les processus sur votre machine en effectuant la commande `top`.

2.2 De quoi est composé un processus?

Un processus englobe tout ce dont a besoin notre programme pour s'exécuter:

- Son code;
- Les données statiques;
- Les variables que l'application a besoin d'allouer;

— ...

Chaque élément doit être repéré par une adresse unique en mémoire. En résumé, une abstraction pour représenter l'état d'un processus est un espace mémoire qui est défini par une séquence d'adresses.

Chaque état d'un processus apparaît dans une région différente en mémoire. Ils sont séparés en trois parties principales:

1. Une partie statique contenant toutes les données disponibles pour démarrer le processus (le code, les chaînes de caractères statiques, ...).
2. *Le tas*: durant l'exécution, un processus peut allouer dynamiquement de la mémoire, cela est fait sur le tas. Le tas est un espace mémoire non contigu, qui peut contenir des "trous" (des régions que le processus ne peut pas accéder).
3. *La pile*: Contrairement au tas qui n'a aucune structure particulière, la pile est une structure LIFO qui va changer de taille dynamiquement. Typiquement les appels de fonctions (ainsi que leur données) sont mises sur la pile.

Les adresses mémoire qui décrivent l'état d'un processus sont dites virtuelles: elles ne doivent pas forcément correspondre aux adresses physiques de la mémoire. Le "mapping" entre les adresses virtuelles et la mémoire physique est géré par le système d'exploitation. Cette façon de faire a l'avantage de découpler la gestion de la mémoire physique (et de la simplifier) de l'utilisation qu'aimerait en faire les différents processus. En fait, à chaque fois qu'un processus alloue une variable, le système d'exploitation lui alloue une adresse virtuelle et un mapping sera fait avec une adresse physique. Ce mapping sera inséré dans une table et pourra être utilisé à chaque fois que l'adresse virtuelle sera accédée par le processus qui l'aura alloué.

Comme nous l'avons dit précédemment, toutes les adresses mémoires ne sont pas forcément allouées par un processus: l'espace utilisé peut contenir des trous. Par ailleurs, on peut même se retrouver dans un état où il n'existe pas assez d'espace physique pour allouer toute la mémoire nécessaire à un processus. Par exemple, si on a un espace d'adressage virtuel qui a une longueur de 32 bits, et qu'on veut stocker en mémoire un fichier de plus de 4 Gb, on va dépasser la mémoire disponible. De même, on peut assez aisément dépasser la mémoire disponible sur un serveur (même très puissant) si on fait tourner plusieurs processus très gourmands en mémoire. La virtualisation de la mémoire permet au système d'exploitation de gérer ces cas en décidant quelle partie de la mémoire virtuelle est mappée sur la mémoire physique et laquelle ne l'est pas. Lorsque la mémoire physique est épuisée, une partie de l'état d'un processus peut être copiée sur le disque dur pour faire de la place: on dit qu'elle est "swapée". La partie de la mémoire mise sur le disque peut ainsi être récupérée à n'importe quel moment. Sans entrer dans des considérations sur la façon exacte dont la gestion est faite, il faut se souvenir à présent que le système d'exploitation doit garder une trace de ce qui se passe à tout moment avec la mémoire de chaque processus, non seulement pour pouvoir retrouver la mémoire liée à un processus mais également pour pouvoir décider si un accès est illégal ou non.

Question 1

Si on a deux processus, P_1 et P_2, tournant en même temps sur un système, lequel de ces trois espaces d'adressage sera utilisé en général ?

— $P_1 : 0 - -32'000$, $P_2 : 32'001 - -64'000$.
— $P_2 : 0 - -32'000$, $P_1 : 32'001 - -64'000$.
— $P_1 : 0 - -64'000$, $P_2 : 0 - -64'000$.

2.3 Quelles informations le système d'exploitation possède sur les processus?

Pour pouvoir gérer un processus, un système d'exploitation doit savoir ce qu'il est en train de faire. En fait, il doit être capable d'interrompre un processus et de le faire recommencer depuis l'état exact où il se trouvait au moment où il a été arrêté.

Comment le système d'exploitation connaît l'état exact d'un processus ?

Lorsqu'on écrit un programme, on écrit du texte. Puis pour l'exécuter, il faut le compiler et produire un fichier binaire. Un fichier binaire est une suite d'instructions qui ne sont pas forcément exécutées de façon séquentielle (il peut y avoir des boucles, des branchements, ...). A n'importe quel moment, le processeur doit savoir où le programme se trouve dans la séquence d'instructions. Cette information est stockée dans un registre sur le processeur et est appelée le compteur du programme (Program Counter, PC). Il existe un grand nombre de registres actifs lors de l'exécution d'un processus (ils peuvent contenir des adresses mémoire, ou des informations qui vont influencer l'ordre des instructions par exemple). Une autre partie importante dans un processus est sa pile et en particulier ce qu'il y a à son sommet. Cette information est contenue dans le pointeur de pile (stack pointer) qui lui aussi doit être connu à tout moment. Il existe encore d'autres structures nécessaires. Tout cela est contenu dans le bloc de contrôle du processus (Process Control Block, PCB).

Le PCB est une structure du système d'exploitation qui est maintenue pour chaque processus qu'il gère. Il contient :

— un compteur de programme (représentant l'endroit où on se trouve dans la suite d'instructions du processus);
— le pointeur de pile (représentant le sommet de la pile du processus);
— des registres et ce qu'ils contiennent;
— on peut y trouver d'autres choses, comme la liste des fichiers ouverts ou des informations relatives à l'utilisation du CPU pour l'ordonnancement;

Le PCB est créé et initialisé lors de la création du processus (par exemple, le compteur de programme sera mis sur la première instruction du programme). Il est mis à jour lorsque l'état du processus change. Par exemple, lorsque le processus demande plus de mémoire, le système va allouer cette mémoire, créer de nouvelles adresses virtuelles valides pour le processus et mettre à jour les informations concernant les adresses du processus (leur limite et leur validité).

Exemple 2

Faisons un exemple de ce que cela veut dire. Imaginons que deux processus P_1 et P_2 sont gérés par le système d'exploitation. Ils sont déjà créés et donc leur PCB sont déjà quelque part en mémoire.

Supposons que seul P_1 est en cours d'exécution et que P_2 est en attente. A un moment donné, P_1 est mis en attente (pour une raison ou une autre) le système d'exploitation doit donc sauvegarder toutes les informations relatives à P_1 dans le PCB de P_1. Ensuite, il va exécuter P_2, donc il doit charger toutes les informations contenues dans le PCB de P_2 et mettre à jour les registres du processeur avec. Ensuite, si P_2 demande plus de mémoire, de nouvelles adresses virtuelles seront créées, et le PCB sera mis à jour avec ces nouvelles informations. Ainsi, si P_2 se termine, ou est mis en attente, les informations relatives à P_2 seront mises dans son PCB et le PCB de P_1 sera chargé et les registres mis à jour pour continuer l'exécution de P_1.

2.3.1 Changement de contexte

Chaque fois qu'un échange est effectué entre deux processus tel que vu dans l'exemple ci-dessus, on appelle cela un **changement de contexte**. Plus formellement, un changement de contexte est le mécanisme utilisé par le système d'exploitation pour changer de contexte d'exécution d'un processus à un autre. Dans l'exemple ci-dessus, cela se passe quand on passe de l'exécution de P_1 à P_2 et de P_2 à P_1.

Un changement de contexte a un coût de calcul certain. En premier lieu, il y a des coûts directs :

1. Le nombre de cycles CPU nécessaires à la sauvegarde de l'état de P_1 dans son PCB;
2. Le nombre de cycles nécessaires au chargement du PCB de P_2 en mémoire.

En second lieu, il y a également des coûts indirects : lorsqu'on exécute un processus, une partie de ses données vont se trouver dans la mémoire cache du CPU (qui est très rapide à accéder par rapport à l'accès à la mémoire de l'ordinateur, on parle de un à deux ordres de grandeur plus rapide). Lors d'un changement de contexte entre P_1 et P_2, les données de P_1 ne seront plus d'aucune utilité dans le cache du CPU et les données de P_2 devront y être chargées depuis la mémoire pour remplacer les données de P_1.

2.4 Les états d'un processus

On a vu qu'un processus peut être dans un état **en attente** ou **en exécution**. Un processus en exécution peut être interrompu et un changement de contexte peut se produire. A ce moment là, le processus sera mis en attente. Il existe deux autres états pour un processus : **éligible** ou **stoppé**. Étudions le cycle de la vie d'un processus pour voir à quoi correspondent ces deux autres cas. Lorsqu'un processus est exécuté, le système fait un certain nombre de vérifications puis crée et initialise son PCB et lui alloue des ressources initiales. Puis, le processus

est mis dans l'état *éligible* (il n'a pas encore démarré sur le CPU). Il va attendre que l'ordonnanceur l'autorise à démarrer et à ce moment, il rentre dans l'état en exécution. Depuis cet état, il peut être interrompu et mis en attente (par exemple s'il doit effectuer une opération I/O importante), puis retournera dans l'état éligible. Une autre solution est qu'il soit interrompu, par l'utilisateur par exemple (ctrl+z) ou à cause d'une erreur. Il est possible que cet arrêt soit définitif ou non (ctrl+z, bg fera en sorte que le processus reprenne en arrière plan). Dans ce cas il est dans un état *stoppé*. Si cela est définitif le processus est terminé.

2.5 Interactions entre processus

Par défaut, il n'existe pas de communications d'un processus à un autre. En particulier, la mémoire entre deux processus est isolée pour éviter des problèmes tragiques. Néanmoins, le système d'exploitation donne la possibilité aux processus de communiquer entre eux. Par exemple, on peut imaginer un serveur web consistant de deux processus. Le premier est un front-end qui accepte les requêtes des clients et le second est une base de données stockant les profils des utilisateurs et des informations diverses. Pour faire communiquer les processus, il faut des mécanismes supplémentaires : **les mécanismes de communication inter-processus**. Ils aident à transférer des informations d'un espace d'adressage à un autre, tout en maintenant l'isolation entre les processus. En très résumé, le système d'exploitation met à disposition une mémoire tampon dans laquelle les processus peuvent lire et écrire ce qui permet la mise en place d'un canal de communication potentiellement bi-directionnel. L'inconvénient principal de cette technique est que le coût de calcul est relativement élevé, car cela demande de copier des quantités parfois très grandes de données et de les relire. Une autre façon de faire communiquer les processus est d'avoir des régions de la mémoire qui sont mappées par les deux processus et ainsi ils peuvent **partager** une partie de leur mémoire respective et n'ont pas besoin de faire des copies. Le problème principal de cette approche est qu'il est très simple de faire des tas d'erreurs et de corrompre la mémoire d'un autre processus.

3 Les threads

Nous venons de discuter des processus et en partie de comment ils sont gérés. Les processus tels qu'ils sont décrits ici ne peuvent être exécutés que sur un seul CPU. Si nous voulons qu'un processus soit exécutable sur plus d'un CPU afin de tirer profit des processeurs modernes, ce processus doit pouvoir posséder plusieurs contextes d'exécution.

Ces contextes d'exécution dans un seul processus sont appelés **fils d'exécution** ou **threads** en anglais. Dans ce chapitre, nous allons décrire ce que sont les threads, en quoi ils sont différents des processus et quels sont leurs avantages et inconvénients.

3.1 Les différences entre un fil d'exécution et un processus

Voyons à présent les différences entre un fil d'exécution et un processus. Un processus est caractérisé par son espace d'adressage mémoire qui contient les adresses virtuelles et leur cartographie vers leurs contreparties physiques, son

code, ses données, etc. Il est également caractérisé par son contexte d'exécution (ses registres, son pointeur de pile, ...) qui est représenté par son PCB.

Les fils eux, représentent des contextes d'exécution indépendants. Ils font partie du même espace d'adressage mémoire et vont partager les mappings des adresses virtuelles vers les adresses physiques. Ils vont également partager le code, les fichiers et les données. Néanmoins, ils vont exécuter des instructions **différentes**, accéder peut-être à des adresses différentes en mémoire, ... Cela signifie que chaque thread aura un compteur de programme différent, un pointeur de pile différent, une pile différente, et des registres différents. Ces informations seront donc stockées dans des structures propres à chaque thread. La représentation du PCB pour le système d'exploitation sera bien plus complexe que ce que nous venons de discuter pour les processus n'ayant qu'un seul fil d'exécution. Il contiendra toutes les informations partagées par chacun des threads, mais également toutes les informations qui seront propres à chaque fil d'exécution du processus.

3.2 Les avantages du multi-threading

Discutons d'abord pourquoi le multi-threading est avantageux.

Imaginons le cas de l'équation de la chaleur que nous avons vue en exercices. Nous pouvons assez aisément imaginer découper la matrice de températures en plusieurs blocs et répartir le travail sur plusieurs threads, chacun exécutant le même code mais s'occupant d'une partie différente des données. Cela ne veut pas dire qu'ils exécutent exactement la même opération à un point donné dans le temps, ils devront donc chacun avoir leurs registres, leur pile, ... Mais on voit assez aisément que si plus d'un processeur (ou plus d'un cœur) est disponible, le calcul sera grandement accéléré grâce à la parallélisation du calcul.

Une autre façon d'optimiser l'exécution d'un programme sur un processeur multi-coeurs est de diviser ce programme en différentes sous-tâches. Par exemple, on peut avoir un thread gérant l'I/O, un autre l'affichage, un autre les entrées au clavier, et encore un autre qui gère la souris.

On peut également imaginer avoir une gestion plus efficace des threads dans une application en différenciant leurs priorités. On peut imaginer mettre plus de fils d'exécution au service de parties plus importantes de l'application.

Il est également possible de spécialiser les threads comme dans une chaîne de montage. Chaque thread est responsable d'une partie bien déterminée de l'application, et effectuera toujours les mêmes opérations sur les mêmes données. De cette façon, sur des systèmes multi-processeurs, on pourra avoir (éventuellement) les mêmes données présentes dans le cache de chaque coeur et ainsi gagner en performance.

Mais vous allez me demander : "Pourquoi ne pas simplement faire tout cela avec un système multi-processus?" Et vous aurez raison. En fait l'avantage des fils d'exécution est leur relative "légèreté". Si on utilise un système multi-processus, il faut exécuter chacun des processus sur un processeur différent. Si cela est le cas, cela signifie que les processus ne partageront pas le même espace d'adressage et auront un contexte d'exécution totalement différent. Cela signifie par conséquent qu'il faudra allouer plusieurs espace d'adressage, rendant le cas multi-processus

beaucoup plus gourmand en ressources. De plus, il est nécessaire d'utiliser les mécanismes de communication inter-processus qui sont beaucoup plus coûteux.

3.3 Le multi-threading sur un seul CPU

La plupart des exemples que nous avons donnés jusque là concernent des applications qui tournent sur plusieurs CPUs. Le multi-threading existe depuis bien avant l'apparition des processeurs possédant plusieurs cœurs [1]. Il doit bien y avoir une raison à cela. En fait, on peut imaginer le scénario suivant. Imaginons qu'un thread, T_1, effectue une requête vers le disque dur. A ce moment-là, le disque a besoin d'un certain temps pour répondre à la requête. Pendant ce temps, T_1 ne fait qu'attendre. Le CPU devrait donc simplement attendre sans rien faire. Si ce temps est plus long que le temps qu'il faudrait pour faire deux changements de contexte, il peut être avantageux d'utiliser un autre thread, T_2, pour effectuer une autre opération en attendant que le disque ait répondu à la requête de T_1. Néanmoins, cela est également vrai pour les processus. Souvenez-vous que le changement de contexte d'un processus est beaucoup plus long (il faut créer un nouvel espace d'adressage, passer par l'ordonnanceur du système d'exploitation, etc). Le temps de changement de contexte pour les fils d'exécution est lui beaucoup plus court et il sera beaucoup plus rapide de faire ce changement de contexte et donc il sera bien plus aisé d'obtenir des gains de performance.

Question 2

Est-ce que les assertions suivantes s'appliquent aux fils d'exécutions, aux processus ou aux deux ?

— Peuvent partager le même espace d'adressage.
— Prennent plus de temps pour effectuer un changement de contexte.
— Ont un contexte d'exécution.
— Doit posséder des mécanismes de communication.

3.4 Les différents modèles de multi-threading

Il existe un grand nombre de modèles pour décomposer le travail à effectuer par un processus par plusieurs threads.

Nous allons brièvement en discuter trois ici :

— Le modèle maître-esclave.
— Le modèle pipeline.
— Le modèle pair.

3.4.1 Le modèle maître-esclave

Le modèle **maître-esclave** (boss/worker model) se caractérise par la présence d'un thread **maître** et d'un certain nombre de threads **esclaves**. Le maître

1. On peut se poser la question de façon plus générale. Peut-on quand même tirer partie d'un code multi-threadé si on a plus de threads que de cœurs?

est chargé de donner du travail aux esclaves et les esclaves sont responsables d'effectuer une tâche entière qui leur est affectée. Les esclaves se synchronisent avec le maître lorsqu'ils ont fini leur tâche. Ce modèle est limité par la performance du maître. Si celui-ci est trop lent à répartir le travail, les esclaves passeront leur temps à l'attendre, son travail doit donc être le plus simple possible et limiter un maximum les interactions entre le maître et les esclaves. Par ailleurs, on essaie de limiter au maximum les dépendances entre esclaves (ce qui ruinerait la performance).

Un exemple de la vie de tous les jours pourrait être le suivant. Considérons une entreprise fabriquant des marteaux. Ces marteaux sont fabriqués à la demande et sur mesure. Une fois la commande passée il faut réaliser les tâches suivantes :

1. Accepter la commande;
2. Lire la commande;
3. Découper le manche dans un manche plus gros;
4. Fondre la tête;
5. Peindre le manche et le vernir;
6. Assembler le manche et la tête;
7. Envoyer le marteau.

La répartition du travail entre le maître et les esclaves pourrait être la suivante. Le maître se contente d'accepter la commande et de la passer à un esclave libre. Il effectue ainsi un minimum de travail et est immédiatement disponible pour traiter une autre commande. Ensuite l'esclave s'occupe du reste: c'est-à-dire des parties 2 à 7. La tâche principale du maître est de trouver un esclave libre pour lui donner son travail (mais comme tout bon esclave, il n'aime pas trop travailler alors il se cache). Comment fait donc le maître ? Une façon de débusquer l'esclave est pour le maître de garder une liste des esclaves libres. Il doit également attendre que l'esclave ait accepté sa tâche. Cette façon de faire n'est donc pas idéale. En revanche, chaque esclave est totalement découplé des autres esclaves. Une autre façon de faire, serait d'avoir une file entre le maître et les esclaves qui viennent se servir en travail tout seuls. L'avantage de cette approche est que le maître n'a jamais besoin de savoir quel esclave est libre ou non : il ne fait que mettre un travail dans la file. Le désavantage est que les esclaves doivent se synchroniser entre eux pour éviter de faire le travail à double. Cette méthode est en général préférable, car la limitation principale de performance est dûe au maître plutôt qu'aux esclaves.

3.4.2 Le modèle pair

Dans le modèle **pair** (peer model), tous les threads effectuent leur part du travail en même temps et n'ont pas de chef (ni Dieu, ni Maître). Dans ce modèle, le thread principal crée tous les autres threads puis il devient l'équivalent des autres fils d'exécution. Chaque thread doit gérer à sa manière ses données (il n'y a pas d'autre thread responsable pour lui donner du travail, donc il doit les récupérer seul) et doit avoir son propre mécanisme pour se synchroniser si nécessaire. Le modèle pair est très indiqué pour les problèmes dont les entrées sont bien déterminées. Elles permettent de s'affranchir du maître dans le modèle maître/esclave, car il n'y a aucune gestion à faire pour le partage du travail. Nous avons vu un tel exemple (l'équation de la chaleur) et en verrons d'autres. D'autres

exemples typiques sont les multiplications de matrices, la recherche en parallèle dans une base de données, ... Dans le cas de la fabrication de marteau, si le marteau est suffisamment grand, on pourrait imaginer que plusieurs travailleurs libres tailleraient le manche, le peindraient et verniraient différentes parties en même temps afin d'accélérer le processus.

3.4.3 Le modèle pipeline

Le modèle **pipeline** est très similaire à la chaîne de montage. Le travail global est subdivisé en sous-tâches et chaque thread se spécialise dans une sous-tâche. Ainsi, on espère que les threads spécialisés seront plus efficaces que des threads généralistes. Un peu comme dans une chaîne de montage humaine où chaque travailleur fait de façon très efficace (mais très répétitive) qu'une seul partie de la chaîne.

Dans notre exemple du marteau (cela pourrait aussi être une faucille), cela voudrait dire qu'on a sept threads qui effectuent chacun une tâche : un qui accepte la commande, un qui la lit, et ainsi de suite. La performance de cette approche est clairement limitée par la tâche la plus lente à effectuer, il est donc primordial que chacune soit bien équilibrée. A première vue, cette façon de faire ne semble pas être très efficace, car chaque thread doit attendre sur le thread précédent pour effectuer sa tâche. En fait cela n'est pas un problème. Lorsque le premier thread reçoit la première commande, il l'accepte et la passe au thread suivant. Il est donc libre de recevoir une autre commande pendant que le deuxième thread lit la commande. Quand le deuxième thread a lu la commande et la passe au troisième thread, il peut recevoir la commande suivante du premier, et ainsi de suite. Dans ce modèle, il est nécessaire d'avoir un mécanisme pour que les threads se passent le travail. On peut imaginer une approche où chaque thread du niveau n passe le travail à un thread libre de niveau $n+1$ mais cela nécessiterait un surcoût dans la mesure où il faudrait que le thread n attende la réponse du thread $n+1$ pour savoir s'il a accepté le travail. A nouveau, la solution de la file d'attente est la plus adaptée. Les threads du niveau $n+1$ viennent récupérer dans la file leur travail et se synchronisent pour éviter que plusieurs ne prennent la même tâche. L'inconvénient principal de cette méthode est que chaque niveau doit avoir une charge de travail équivalente et qu'il est assez complexe de maintenir une charge équivalente au cours du temps : si une tâche commence à prendre plus de temps, il devient nécessaire de revoir l'équilibrage entier du pipeline. On peut, par exemple, imaginer utiliser le modèle pair pour améliorer les performances d'un étage de la chaîne en rajoutant des threads.