INFO-F-201 – Systèmes d'exploitation

Précisions diverses

Joël Goossens (N8.107) Olivier Markowitch (N8.106)
Arnaud Leponce (N8.213) Yannick Molinghen (N8.210)
Alexis Reynouard (N8.215)

Exercice 1. *Dans le système Unix, est-ce que tout processus a un père ?*

Réponse : Tous les processus ont un père, sauf le processus `init`, qui se trouve à la racine de l'arbre.

Exercice 2. *Que se passe-t-il lorsqu'un processus devient orphelin (mort de son père) ?*

Réponse : Ce qui se passe dépend de l'implémentation. Historiquement, les systèmes POSIX font adopter le processus orphelin par le processus `init`. C'est également le cas pour les systèmes de la famille Linux dont traite ce cours.

Pour être précis, il est possible que ça ne soit pas `init` qui adopte le processus car Linux permet la création de processus "subreaper" qui deviennent les parents de leurs descendants devenus orphelins. Ceci dépasse cependant le cadre de ce cours et est expliqué uniquement à titre informatif.

Exercice 3. *Est-ce qu'un processus peut exécuter plusieurs programmes, et est-ce qu'un programme peut être exécuté par plusieurs processus à la fois ? Expliquez.*

Réponse : Un programme peut effectivement être exécuté par plusieurs processus : il suffit de compiler un fichier et de l'exécuter deux fois. Un processus exécute habituellement un seul programme : lorsque vous lancez votre programme dans le terminal, vous créez un processus qui exécute ce programme. Cependant, il est possible que le programme en cours soit remplacé par un autre, par exemple à l'aide de l'appel système `exec`, comme vu au cours des TPs. On peut donc argumenter qu'un processus peut exécuter plusieurs programmes.

Exercice 4. *Vous souhaitez accélérer le traitement d'une application en parallélisant son contenu. Vaut-il mieux utiliser des threads ou des processus ? Pourquoi ?*

Réponse : Pour ce genre de traitement, l'utilisation de threads est à privilégier. En effet, les threads sont beaucoup plus légers pour l'OS que les processus (pas de duplication du heap, par exemple). De plus, il y aura certainement des données partagées et c'est plus facile à gérer avec des threads. Les processus sont intéressants lorsque vous souhaitez exécuter certaines tâches sans qu'une erreur critique ne termine l'entièreté du programme (une erreur critique dans thread termine le processus mais une erreur critique d'un processus fils ne termine pas le processus

parent ni les autres enfants par exemple). Utiliser plusieurs processus est aussi essentiel pour les applications communiquant par internet (pour du calcul distribué par exemple).

Exercice 5. *Quelle est la différence entre un ordonnanceur préemptif et un ordonnanceur non préemptif ?*

Réponse : Dans les ordonnanceurs préemptifs, un processus peut se faire interrompre avant la fin de son exécution et être remplacé par un autre processus, ce qui n'est pas le cas dans le cadre des ordonnanceurs non-préemptifs.

Exercice 6. *Quels sont les événements qui peuvent provoquer l'interruption de l'exécution d'un processus ?*

Réponse :

- Le processus se termine
- Le processus est en attente d'une entrée/sortie
- Le quantum de temps du processus est écoulé
- L'arrivée d'un processus plus prioritaire
- La réception d'une interruption matérielle

Exercice 7. *Expliquez la notion de pseudo parallélisme qu'implique un ordonnanceur préemptif dans un système monoprocesseur.*

Réponse : Dans le cadre d'un ordonnanceur préemptif (càd qui peut interrompre un processus en cours d'exécution pour en exécuter un autre), une multitude de processus se succèdent rapidement pour donner une illusion de parallélisme à l'utilisateur alors qu'en réalité, un seul processus est exécuté à la fois sur le processeur.

Exercice 8. *Dans un système d'ordonnancement circulaire (round robin = tourniquet), quel est l'effet d'un quantum de temps trop long ? Quel est l'effet d'un quantum de temps trop court ?*

Réponse : Un quantum de temps trop long a pour conséquence un manque de réactivité du système car les alternances entre processus s'effectuent lentement. Un quantum de temps trop court est une surcharge pour le processeur qui doit effectuer beaucoup de changements de contextes. On y perd de la puissance de calcul.

Exercice 9. *Comment expliquez-vous qu'un processus qui attend une entrée sur stdin par l'utilisateur n'occupe pas 100% d'un processeur (ce qui serait de l'attente active) ? Décrivez le mécanisme qui permet cela.*

Réponse : Lorsqu'un processus demande une entrée/sortie comme une ligne sur stdin, celui-ci se voit changer d'état et passe de l'état "actif" à l'état "bloqué". Ensuite, lorsque l'utilisateur appuie sur "enter", le processus passe en état "prêt". Enfin, lorsque l'ordonnanceur le décidera, le processus pourra repasser en état "actif" pour être exécuté.
Comme seuls les processus en état "actif" occupent le CPU, le processus n'utilise pas de CPU lorsqu'il est en attente de l'entrée/sortie.

Exercice 10. *Définissez un interblocage (deadlock) en quelques mots.*

Réponse : Un interblocage survient par exemple lorsque deux tâches ont verrouillé une ressource partagée et tentent de verrouiller la ressource détenue par l'autre tâche. Comme nous l'avons vu précédemment, chaque tâche va être mise en attente (état bloqué) jusqu'à ce que la mutex soit libérée, ce qui n'arrivera jamais vu que l'autre tâche est aussi bloquée.

Exercice 11. *Quelle est la différence entre un thread et un processus au niveau de leur création et de leur terminaison.*

Réponse : La création d'un thread n'implique pas la duplication de son heap, ce qui le rend plus léger. De plus, chaque thread dépend du processus qui l'a créé. Par conséquent, quand le processus se termine, tous les threads en cours du processus en question sont tués.

Exercice 12. *Considérons un système mono-processeur comprenant les 5 ressources informatiques additionnelles $R_1, R_2, \ldots, R_5$ à accès exclusif (non partageables). Dans les pseudo-codes ci-dessous la fonction `lock(R)` permet d'allouer la ressource au processus appelant si celle-ci est libre. Sinon, le processus appelant est bloqué. La fonction `unlock(R)`, appelée par le détenteur de la ressource, libère cette ressource.*

Considérons les 4 processus P_1, P_2, P_3 et P_4 qui exécutent respectivement les codes suivants :

```
// P1
while (1){
    lock(R1);
    lock(R2);
    lock(R3);
    lock(R4);
    // Utiliser R1, R2, R3, R4
    unlock(R1);
    unlock(R2);
    unlock(R3);
    unlock(R4);
}
```

```
// P2
while (1){
    lock(R5);
    lock(R2);
    lock(R3);
    lock(R4);
    // Utiliser R2, R3, R4, R5
    unlock(R5);
    unlock(R2);
    unlock(R3);
    unlock(R4);
}
```

```
// P3
while (1){
    lock(R4);
    lock(R5);
    // Utiliser R4, R5
    unlock(R4);
    unlock(R5);
}
```

```
// P4
while (1){
    lock(R3);
    lock(R4);
    lock(R5);
    // Utiliser R3, R4, R5
    unlock(R3);
    unlock(R4);
    unlock(R5);
}
```

1. *Donnez une séquence d'entrelacements (par exemple P_1 lock(R4), P_2 lock(R2), …) des instructions des 4 processus qui mènent à un interblocage.*
2. *Pourrions-nous éviter les interblocages en modifiant l'ordre des demandes (les locks) des ressources ? Justifiez votre réponse.*

Réponse :

1. $P_1(R_1), P_1(R_2), P_2(R_5), P_4(R_3), P_3(R_4), P_3(R_5)$;
2. Oui en se basant sur l'ordre total des ressources (les numéros de ressource) avec la contrainte de faire des demandes (locks) par ordre croissant (ou décroissant) par rapport au numéro de la ressource.