E.N.S.E.R.B. Année 1999-2000
Filière Informatique $2^{ème}$ année

SYSTÈMES D'EXPLOITATION

PARTIEL

CORRIGÉ

N.B. : - Ceci doit être considéré comme un corrigé-type : les réponses qu'il contient sont justes, mais leur rédaction n'était pas la seule possible.
- Le barème est donné à titre définitif. Outre l'exactitude des réponses aux questions posées, il a été tenu compte de leur concision et, dans une moindre mesure, de la présentation.

Question 1 (3 points)

L'appel système `wait ()` permet à un processus père d'attendre la terminaison de l'un de ses fils et de connaitre la valeur de retour de celui-ci (définie par l'appel à `exit()` ou dans un `return ()` de la fontion `main()` du fils) ainsi que son PID. Si l'un des fils est déjà mort, l'appel système `wait ()` retourne immédiatement.

Conserver les processus zombies dans la table des processus permet alors d'y conserver leurs PID et d'y stocker leurs valeurs de retour en attendant qu'ils soient pris en compte par leur père. Ce stockage est peu coûteux, car toutes les autres ressources système associées à un processus ont été libérées au moment de sa terminaison. C'est aussi le moyen le plus simple pour éviter au père de recréer un processus de même PID que celui de son fils zombie, qui génèrerait des ambiguïtés pour savoir lors du `wait ()` de quel fils les informations de retour proviennent. De plus, le fait que les zombies soient visibles, au moyen de la commande `ps` par exemple, peut permettre de savoir si leurs processus pères fonctionnent correctement : un père qui ne s'occupe pas de ses fils zombies alors qu'il le devrait est vraisemblablement bloqué ou du moins extrêmement ralenti...

La valeur de retour est intéressante par exemple dans le cas d'un serveur qui, chaque fois qu'un client se connecte à lui, forque un processus fils pour traiter le nouveau client : le code de retour du fils sert alors à savoir si la requête effectuée par le client a bien été traitée par le fils du serveur ; le PID permet de savoir quel fils a terminé.

On peut aussi imaginer un programme `gzip` pour machine multi-processeurs : si de nombreux fichiers doivent être compressés, le programme initial lancera autant de compressions de fichiers que de processeurs, et fera ensuite un `wait ()` ; dès qu'un fils terminera, le processus père se réveillera et lancera la compression d'un nouveau fichier. On aura ainsi au plus autant de compressions que de processeurs (moins un, si l'on veut laisser un minimum de ressources aux autres utilisateurs), et donc les compressions ne seront pas en compétition entre elles pour la ressource CPU.

Question 2 (11 points)

(2.1) (2,5 points)

Pour qu'aucune communication ne soit nécessaire, il faut que chaque processeur soit totalement autonome, et gère entièrement l'allocation de ses processus ; en particulier, il ne faut pas qu'il ait à se demander si un numéro de processus donné a déjà été attribué par un autre processeur.

La solution qui en découle est donc de partitionner le domaine des numéros de processus en P sous-domaines indépendants : par exemple, les numéros de 0 à $\lfloor \frac{N}{P} \rfloor$ sont attribués au processeur 0, de $\lfloor \frac{N}{P} \rfloor + 1$ à $\lfloor \frac{2N}{P} \rfloor$ au processeur 1, etc.

La principale limitation de cette solution est qu'un processeur ne peut lui-même créer plus de $\lfloor \frac{N}{P} \rfloor$ processus coexistant simultanément. Si la plupart des processus est lancée à partir d'un petit nombre de processeurs, le système sera sous-utilisé.

(2.2) (2,5 points)

Si le système permet la migration dynamique de processus, un processeur ne peut savoir si les processus qu'il a lancés et qui ont migré ont terminé ou non.

Pour que le mécanisme d'attribution continue à fonctionner, il faut donc garder, dans la structure de contrôle de chaque processus, le numéro du processeur créateur (on peut remarquer que ce numéro peut facilement être retrouvé en effectuant la division entière par P du numéro de processus), pour pouvoir informer celui-ci de la terminaison du processus, afin que son numéro puisse être réattribué. Ceci s'effectue au prix d'un message par processus. Ici encore, un même processeur ne peut créer plus de $\lfloor \frac{N}{P} \rfloor$ processus.

(2.3) (2,5 points)

Une solution possible consiste à utiliser un des processeurs comme serveur de numéros de processus : chaque fois qu'un processeur a besoin de créer un processus, il envoie un message au serveur de numéros, qui lui renvoie en retour un numéro libre. Chaque fois qu'un processus meurt, un message de libération est envoyé au serveur afin de rendre le numéro libre à nouveau. Le coût de cet algorithme est donc de trois messages par processus.

Le principal problème de cette solution est que le serveur de numéros constitue un goulot d'étranglement pour le système, et qui devient de plus en plus étroit avec l'augmentation du nombre de processeurs.

Une autre solution consiste à considérer que les numéros de processus libres sont des jetons librement échangeables entre processeurs, correspondant à un droit de création d'un processus de ce numéro. Lorsqu'un processus migre, le jeton correspondant est donné au processus destinataire qui, pour ne pas limiter un processus en création, peut lui renvoyer en échange l'un de ses jetons s'il lui en reste. Lorsqu'un processeur ne dispose plus de jetons pour créer un processus, il est obligé d'en demander auprès de ses voisins, ce qui est plus coûteux en termes de messages que la solution du serveur centralisé, mais ne sollicite pas qu'une unique machine.

(2.4) (3,5 points)

Pour améliorer l'approche précédente, il convient à la fois de diminuer le nombre moyen de messages nécessaires par processus, ainsi que de faire disparaitre le goulot d'étranglement provenant du fait qu'un unique serveur de numéros est le destinataire de l'ensemble des messages.

Pour réduire le nombre de messages nécessaires, on peut faire en sorte que le serveur de numéros alloue ces derniers aux processeurs par blocs de k numéros consécutifs. Dans ce cas, un processeur n'envoie de message au serveur de numéros que pour redemander l'attribution d'un nouveau bloc de numéros, lorsqu'il a épuisé tous les numéros du dernier bloc qu'il avait précédemment demandé.

Qui plus est, les messages de terminaison ne reviennent qu'au processeur qui a créé le processus qui a terminé, ce qui réduit considérablement la charge du serveur de numéros.

Avec ce procédé, on a deux possibilités : soit un processeur peut réutiliser immédiatement un numéro qui vient d'être libéré dans un bloc qu'il possède (mais ceci peut perturber l'utilisateur car il ne voit pas augmenter les numéros des processus qu'il lance), soit on attend que tous les numéros d'un bloc soient redevenus libres pour rendre le bloc au serveur de numéros, afin d'utiliser plutôt les numéros d'un autre bloc.

Comme chaque processeur a au moins besoin d'un bloc pour charger ses processus système (de type `init`, `swapper`, ...), le nombre maximum de processus allouables par un processeur est de $N - (P-1)k$.

Avec le procédé décrit ci-dessus, un processeur n'envoie de messages au serveur de numéros qu'une fois sur k, pour demander un nouveau bloc, et un autre pour l'informer de la libération du bloc. En comptant les messages envoyés aux processeurs lors de la terminaison des processus, le nombre moyen de messages envoyés par processus retombe à $1 + \frac{2}{k}$, sans goulot d'étranglement.

Plus k est grand, moins le nombre de messages est important, mais aussi moins un unique processeur peut créer de processus.

Question 3 (6 points)

La sémantique des moniteurs est qu'il ne peut y avoir deux processus simultanément actifs à l'intérieur d'un même moniteur. Il s'agit donc de faire respecter cette règle au moyen des sémaphores.

Pour cela, le compilateur doit créer un sémaphore `verrou` pour chaque moniteur, qui sera initialisé à 1 afin qu'un seul processus puisse entrer dans la section critique que constitue le moniteur. Le compilateur doit donc aussi insérer un appel `down(verrou)` comme première instruction de chacune des routines du moniteur, et un appel `up(verrou)` avant chaque instruction de terminaison de procédure (instruction `return` explicite ou implicite).

À chaque variable de condition est associée un sémaphore initialisé à 0. Ainsi, l'appel à `signal(condition)` est remplacé par `up(condition)`, et l'appel à `wait(condition)` est traduit en la séquence `up(verrou)`/`down(condition)`/`down(verrou)`. Cette séquence garantit qu'un processus qui effectue un `wait()` puisse permette à un autre de rentrer dans le moniteur pour effectuer l'opération `signal()` correspondante, et que le processus réveillé par l'appel à `signal()` ne puisse revenir au sein du moniteur tant que le processus qui a fait le `signal()` s'y trouve encore.

Notez bien que, pour la primitive `signal(condition)`, on n'encadre pas le `up(condition)` par une paire `up(verrou)`...`down(verrou)`, car sinon un autre processus pourrait en profiter pour rentrer dans le sémaphore et nous forcer à nous endormir sur le `down(verrou)`, alors qu'on est actuellement actif dans le moniteur.