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

SYSTÈMES D'EXPLOITATION

EXAMEN DE RATTRAPAGE

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 (11 points)

(1.1) (1 point)
Si on lance simultanément plusieurs processus, on court le risque que ceux-ci demandent collectivement plus de ressources que n'en possède le système, et qu'ils se bloquent mutuellement.

(1.2) (1 point)
Si le système de soumission des processus oblige à définir la quantité maximum de ressource nécessaire à l'exécution de chaque processus, on peut garantir de façon certaine que l'ensemble des processus lancés terminera bien si et seulement si la somme des ressources déclarées est inférieure ou égale à la quantité totale de ressource disponible dans le système.
Le système de soumission peut alors choisir quels processus exécuter de façon à ce que cette règle soit vérifiée à tout moment. Si un processus demande plus de ressource que son maximum déclaré, le système peut refuser l'allocation et tuer le processus, afin d'empêcher tout interblocage.
Il faut cependant remarquer que cet algorithme est extrêmement peu efficace, car les processus n'utilisent habituellement que peu de temps la quantité maximale de ressource dont ils ont besoin pour terminer.

(1.3) (2 points)
À partir de cette configuration, on ne peut garantir qu'il existe un séquencement des processus garantissant leur terminaison certaine. En effet, chacun des processus peut encore légalement demander plus de ressource que n'en possède le système et bloquer.

(1.4) (2 points)
À partir de cette configuration, il existe un séquencement des processus garantissant leur terminaison certaine :
- exécution de C, qui peut allouer (de justesse) les 2 unités dont il peut avoir besoin, puis libérer toutes celles qu'il possède (il y en a alors 4 de disponibles dans le système) ;
- exécution de B ou D. On pourra choisir D, parce qu'il nécessite le moins de ressource pour terminer ;
- exécution de B, pour les mêmes raisons ;
- exécution de A.

(1.5) (2 points)
On a pu se sortir de la deuxième configuration, à la différence de la première, parce qu'il restait suffisament de ressource pour que le processus ayant besoin du moins de ressource puisse se les voir attribuer, et que la libération de ce processus permette au suivant de terminer, et ainsi de suite...
Une règle permettant d'affirmer qu'une configuration donnée autorise au moins un séquencement garantissant une terminaison certaine est qu'il reste assez de ressource pour que le processus qui en nécessite le moins pour terminer puisse se la voir attribuer, et que la ressource libérée par ce processus permette à celui restant qui en nécessite le moins de terminer également, et ainsi de suite jusqu'à ce que tous les processus aient terminé.

(1.6) (3 points)

L'algorithme d'allocation doit garantir que l'ensemble des processus termine dans tous les cas. Pour cela, il faut interdire l'allocation à un processus de ressource dont l'accaparation pourrait conduire à un interblocage. Si un processus qui demande des ressources peut être satisfait, on lui allouera immédiatement ces ressources ; sinon, on l'endormira jusqu'à ce qu'un autre processus libère de ses ressources (et fasse un `wakeup()` sur les processus endormis), ce qui permettra de réexaminer sa demande.

Pour déterminer l'algorithme, on part d'une configuration existante valide (c'est-à-dire permettant la terminaison de l'ensemble des processus). Un processus qui demande une certaine quantité de ressource peut être satisfait si, en supposant cette ressource allouée, la règle de la question précédente est vérifiée. Sinon, on endort le processus. On a donc les algorithmes suivants :

```
alloc (quantité) {
loop:                           // Boucle de réévaluation
  q = res_courante - quantité;  // Calcule quantité restante après
  P = ensemble_des_processus;
  tantque (non_vide (P)) {
    p = plus_pres_max (P);      // Processus le plus près de son maximum
    si (demandable (p) > q) {   // Si ce processus ne peut être satisfait
      sleep (adresse);          // On s'endort sur une adresse prédéfinie
      goto loop;                // On réévaluera au réveil
    }
    supprime (P, p);            // On retire le processus p de l'ensemble
  }
  res_courante -= quantité;     // On peut allouer : on alloue!
  return (do_alloc (quantité)); // Renvoie handler sur la ressource
}

free (ressource) {
  res_courante += quantité (ressource); // Compte la quantité rendue
  do_free (ressource);          // Libère effectivement la ressource
  wakeup (adresse);             // Réveille les processus en attente
}
```

Cet algorithme, dit ¡¡ du banquier ¿¿, a été proposé par Dijkstra en 1965. Plusieurs améliorations permettent de l'appliquer simultanément à plusieurs ensembles de ressources différentes.

Question 2 (3 points)

(2.1) (1 point)

La taille moyenne de la table des pages d'un processus est de $\left\lceil \frac{s}{p} \right\rceil e$ octets.

Faute d'informations sur la distribution des tailles des processus par rapport à la taille moyenne s, on peut supposer que les tailles sont uniformément réparties dans l'intervalle $[0; p[$ modulo p, et donc la quantité moyenne de mémoire perdue dans la dernière page est $\frac{p}{2}$.

La quantité totale de mémoire perdue lors de l'allocation d'un processus est donc, en moyenne, de :

$$\left\lceil \frac{s}{p} \right\rceil e + \frac{p}{2} \ .$$

(2.2) (2 points)

Pour déterminer la taille optimale p minimisant l'équation ci-dessus, on supprime la partie entière (on introduit donc une erreur d'au plus e sur la taille), et l'on cherche le minimum de cette expression T en annulant sa dérivée T' par rapport à p :

$$\begin{aligned} T &= \frac{s}{p} e + \frac{p}{2} \ , \\ T' &= -\frac{s}{p^2} e + \frac{1}{2} \ . \end{aligned}$$

On doit donc trouver les racines de l'équation :

$$\frac{1}{2}p^2 - se = 0 \ ,$$

dont l'unique racine réelle est :

$$p = \sqrt{2se} \ .$$

L'erreur obtenue en supprimant la partie entière dans T n'est pas significative, car les tailles des pages sont des puissances de 2, et il s'agit donc de prendre la taille réelle de page la plus proche possible de cette valeur théorique.

Question 3 (3 points)

Les routines d'entrées/sorties décrites dans la section 2 du `man` Unix sont des appels sytèmes, qui s'exécutent en mode noyau. Elles manipulent des descripteurs de fichiers, codés comme des entiers, sur des flots non bufférisés.

Celles de la section 3 sont des fonctions de la bibliothèque standard (`libc`), qui s'appuient sur les appels système de la section 2 pour s'exécuter. Elles sont de plus haut niveau, s'appliquent à des flux, codés comme des pointeurs sur des structures `FILE`, et mettent en place des mécanismes tels que la bufférisation ou le *push-back* (remise de caractères dans le flux).

Question 4 (3 points)

Les attentes actives sont celles qui consomment des ressources. Elles reviennent à évaluer une condition et à boucler tant que celle-ci n'est pas vérifiée, comme c'est le cas par exemple avec les boucles de type Test-and-Set-Lock (TSL) exécutées par le processeur.

Les attentes passives, en revanche, ne consomment pas de ressources. Elles sont basées sur l'endormissement du processus courant, qui ne pourra redevenir actif que lorsqu'une certaine condition, évaluée par un autre processus ou le système (par exemple à la suite d'une interruption), permettra le réveil et la continuation du processus endormi.

Les attentes passives sont de plus haut niveau que les attentes actives, car elles supposent l'existence de processus et d'un ordonnanceur, alors que les attentes actives peuvent être directement utilisées au plus bas niveau pour surveiller le changement d'état d'un matériel (par l'entremise de registres mappés en mémoire, par exemple).

On peut les combiner de façon efficace en démarrant une attente indéterminée par une attente active, en espérant que la condition sera vite réalisée, puis par une attente passive au bout d'un certain temps, pour libérer le processeur. La partie en attente active permet d'épargner le coût énorme d'un changement de contexte de processus si la condition est rapidement satisfaite, et la partie passive permet d'utiliser le processeur à des tâches plus utiles si l'attente dure longtemps.