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

SYSTÈMES D'EXPLOITATION

EXAMEN

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)

(1.1) (1,5 points)

Comme on ne dispose pas d'informations sur l'emplacement sur le disque où se trouve le bloc courant du fichier journal, on suppose une position moyenne au milieu de la zone réservée aux données ; on fait de même pour le bloc de structure correspondant au fichier. En régime continu, si le disque ne sert qu'à l'enregistrement du fichier journal, le bras effectue un va-et-vient entre les positions du bloc de contrôle et du bloc de données. Le temps de déplacement du bras d'un point à l'autre est donc de $\left(\frac{s}{2}+\frac{d}{2}\right)\frac{1}{\beta}$, et le temps moyen d'attente du passage du bon bloc sous la tête de lecture est de la moitié de la circonférence de la piste, soit : $\frac{1}{2}\cdot\frac{1}{\theta}$.

$$\begin{aligned} t_1 &= 2\left(\left(\tfrac{s}{2}+\tfrac{d}{2}\right)\tfrac{1}{\beta}+2\tfrac{1}{2\theta}\right) \\ &= \tfrac{p}{\beta}+\tfrac{2}{\theta}\ . \end{aligned}$$

Avec les valeurs données, remises à l'échelle pour obtenir un temps en milli-secondes, on obtient : $t_1 = 40,475$ ms.

(1.2) (1,5 points)

$$\begin{aligned} t_2 &= 2\left(\left(\tfrac{s}{2}+\tfrac{d}{4}\right)\tfrac{1}{\beta}+2\tfrac{1}{2\theta}\right) \\ &= \tfrac{p+s}{2\beta}+\tfrac{2}{\theta}\ . \end{aligned}$$

Avec les valeurs données, on obtient : $t_2 = 31,025$ ms.

(1.3) (2 points)

Puisque d est divisible par s, on peut réaliser des blocs de $\frac{d}{s}$ pistes de données accolées chacun à une piste de structure. Si l'on positionne la piste de données à une extrémité de chaque bloc, on obtient le calcul suivant (ressemblant beaucoup à celui de t_1 :

$$\begin{aligned} t_3 &= 2\left(\tfrac{d}{2s\beta}+2\tfrac{1}{2\theta}\right) \\ &= \tfrac{d}{s\beta}+\tfrac{2}{\theta}\ . \end{aligned}$$

Avec les valeurs données, on obtient : $t_3 = 20,300$ ms.

Cependant, on a vu dans la question précédente l'intérêt de placer les pistes de structre au milieu des pistes de données qu'elles contrôlent. Dans ce cas, on a :

$$\begin{aligned} t'_3 &= 2\left(\tfrac{d}{4s\beta}+2\tfrac{1}{2\theta}\right) \\ &= \tfrac{d}{2s\beta}+\tfrac{2}{\theta}\ . \end{aligned}$$

Avec les valeurs données, on obtient : $t'_3 = 20,150$ ms. À ce niveau d'organisation, c'est la latence de rotation qui devient le facteur pénalisant, et c'est sur elle qu'il faut intervenir.

(1.4) (2 points)

Si l'on dispose de deux bras, et que le disque ne sert qu'aux opérations auxquelles nous nous intéressons, chacun des bras se stabilisera au dessus d'une des zones concernées (structure et données). La piste de structure sera toujours la même, et la piste de données presque toujours également (la taille des données à écrire étant petite par rapport à la taille d'un bloc). Avec les bras déjà positionnés sur les deux pistes, on a :

$$\begin{aligned} t_4 &= \tfrac{1}{2\theta} + \tfrac{1}{2\theta} \\ &= \tfrac{1}{\theta} \ . \end{aligned}$$

Avec les valeurs données, on obtient : $t_4 = 10,000$ ms.

Cependant, si le contrôleur disque travaille de façon asynchrone par rapport au processeur, les deux ordres d'écriture seront envoyés à la suite par le système, et traités en parallèle par le contrôleur. On aura donc :

$$\begin{aligned} t'_4 &= \max\left(\tfrac{1}{2\theta}, \tfrac{1}{2\theta}\right) \\ &= \tfrac{1}{2\theta} \ . \end{aligned}$$

Avec les valeurs données, on obtient : $t'_4 = 5,000$ ms.

Question 2 (5,5 points)

(2.1) (1,5 points)

Le problème pouvant résulter de cette implémentation est que, si un deuxième signal arrive au moment où le processus vient juste d'entrer dans la routine de traitement fournie par l'utilisateur (et donc avant que l'utilisateur ne puisse réarmer sa routine de traitement), le processus entrera dans sa routine de traitement par défaut, et donc terminera (pour les signaux les plus couramment émis). Voici un rapide exemple :

```
#include <stdio.h>
#include <signal.h>

void
signalTraite (
int                 signum)             /* Numéro de signal, non utilisé ici */
{
  /* ... Le problème peut se poser ici, avant le signal()     ... */
  signal (SIGINT, signalTraite);        /* Ré-arme le signal      */
  fprintf (stderr, "Signal capturé"\n); /* Matérialise la capture */
}

int
main ()
{
  signal (SIGINT, signalTraite);        /* Arme le signal pour la première fois */
  while (1)                             /* Boucle infinie                       */
    kill (getpid (), SIGINT);           /* Pilonne le gestionnaire de signaux   */
}
```

(2.2) (2,5 points)

Pour éviter qu'un deuxième signal ne puisse s'intercaler entre le début de la routine de traitement et le réarmement du gestionnaire de signal réalisé par celle-ci, la seule solution est que le système ne remette pas à sa valeur par défaut l'adresse de la routine au moment de prendre en compte le signal. On retombe alors sur le problème initial, qui est que l'arrivée de nombreux signaux conduit à l'empilement des contextes des routines de traitement, jusqu'à ce que la pile déborde.

Pour le résoudre, il faut interdire l'empilement des contextes, et donc indiquer au système si l'on se trouve déjà dans une routine de traitement de signal. Si c'est le cas, le système n'empilera pas un nouveau contexte, mais mémorisera le nouveau signal pour traitement ultérieur. La question qui en découle est donc de déterminer la capacité de mémorisation des signaux en attente, et donc de ce que représente l'arrivée d'un signal.

Si l'on utilise un compteur pour compter le nombre de signaux en attente, afin de ne vouloir perdre aucun signal reçu, on se trouve devant deux problèmes conceptuels. Le premier est que cette solution n'est pas fiable, car tout compteur est physiquement limité et peut déborder. Le deuxième est que plusieurs occurences du même signal ne nécessitent pas nécessairement des traitements distincts. Ainsi, si vous sonnez à la porte d'un ami et qu'il ne répond pas, vous re-sonnerez régulièrement jusqu'à ce qu'il ouvre, et d'autres de ses amis qui seront arrivées entretemps sur le pas de la porte également, mais il n'ouvrira pas la porte autant de fois qu'on a sonné, et ne fera pas nécessairement entrer ses amis un par un à chaque fois.

La solution techniquement la plus raisonnable dans ce cas est donc de ne mémoriser que le fait qu'un signal au moins d'un type donné est en attente, au moyen d'un vecteur de bits. Un entier est également nécessaire pour mémoriser si le processus est actuellement en train de traiter un signal, et si oui lequel.

L'exemple ci-dessus illustre magnifiquement les choix technologiques différents pouvant découler de conceptions philosophiques sur l'utilisation du système : dans le monde System V, plutôt orienté temps-réel, le fait de ne pouvoir traiter à temps un signal est considéré comme un événement grave, entraînant par défaut la terminaison du processus. En revanche, dans le monde BSD, plutôt orienté distribué, l'important n'est pas de savoir combien de signaux d'un type donné ont été reçus, ni de les traiter chacun individuellement, mais de savoir qu'un au moins a été reçu, indiquant un événement important que l'on traitera quand on le pourra.

(2.3) (1,5 points)

Le problème de conserver la table des signaux dans la structure `u` des processus est que celle-ci est swappée sur disque. Lorsqu'un signal est généré, et que le processus auquel il est destiné est swappé sur disque, le système doit recharger les pages de la structure `u` à partir du disque (en supprimant de la mémoire centrale des pages utilisées par d'autres processus), uniquement pour se rendre compte que le processus a choisi d'ignorer ce type de signaux.

Une première solution consiste à remonter l'intégralité de la table des signaux de la structure `u` vers la structure `proc`, qui elle n'est jamais swappée. Cependant, pour limiter la taille de la table des structures `proc` maintenue par le système, on peut adopter une solution intermédiaire : conserver la table des signaux dans la structure `u` de chaque processus, mais ajouter à leur structure `proc` un vecteur de bits indiquant si le signal doit être ignoré ou non. On évite ainsi les swaps inutiles, tout en n'augmentant que marginalement la taille de la structure `proc` et en ne dupliquant que peu d'information par rapport à la table complète.

Question 3 (7,5 points)

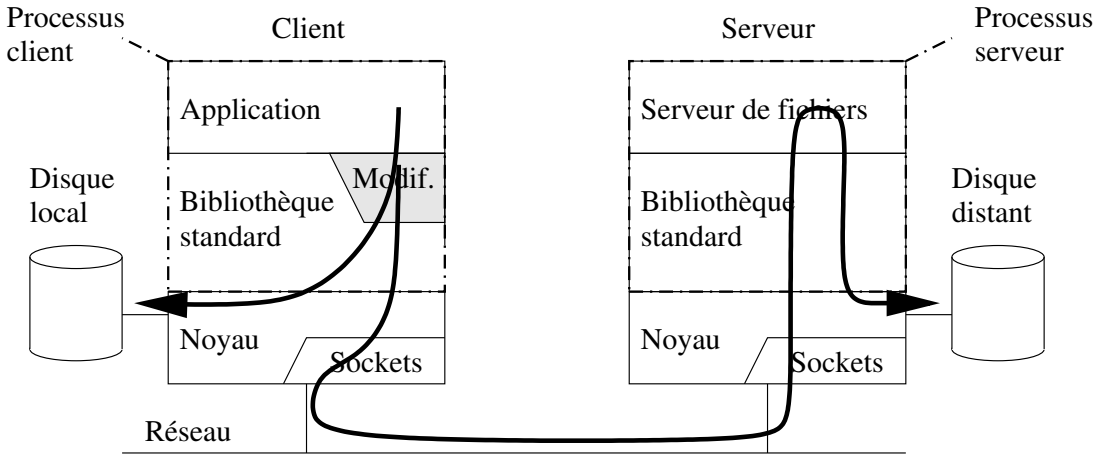
(3.1) (0,5 points)

L'intérêt de la deuxième approche est qu'elle n'utilise pas de caractères illégaux, et permet donc de saisir des chemins réseau sans toucher aux programmes et interfaces existantes, qui rejettent les chemins illégaux et qu'il faudrait donc modifier. L'extension des fonctionnalités sans toucher à l'existant est capitale en termes de temps et de coût de développement, et doit être privilégiée autant que possible.

(3.2) (3 points)

Les opérations sur les fichiers locaux et distants, demandées par les applications, sont traitées par les routines (modifiées) de la bibliothèque standard, et donnent lieu à des appels système. Si les opérations portent sur des fichiers distants, des requêtes sont envoyées sur le réseau vers la machine

serveur de fichiers correspondante, et traitées par le serveur distant, qui réalise effectivement les opérations demandées sur ses disques locaux.



Les modifications à apporter à la bibliothèque standard concernent les primitives de bas niveau (celles du `man 2`), à savoir `open()`, `close()`, `read()`, `write()`, `lseek()`, etc.

Pour `open()`, il s'agit de vérifier la structure du nom de chemin passé en paramètre, par rapport au chemin courant (¡¡ *working directory* ¿¿, qui peut déjà être un chemin réseau), pour savoir si le fichier doit être ouvert localement ou à travers le réseau. Cette différence est mémorisée dans une table auxiliaire des fichiers ouverts.

Pour les primitives utilisant les descripteurs de fichiers renvoyés par `open()`, il s'agit de vérifier tout d'abord si le numéro de descripteur passé en paramètre correspond à un fichier local ou à un fichier réseau, et de réaliser l'opération demandée en conséquence.

Les primitives de haut niveau (celles du `man 3`), à savoir `fopen()`, `fclose()`, ... n'ont pas à être modifiées, car elles s'appuient sur les primitives de bas niveau modifiées, et n'ont donc pas à être réécrites. Ici encore, on voit que l'on peut étendre les fonctionnalités en touchant le moins possible à l'existant.

(3.3) (1,5 points)

Le premier problème est de s'assurer que l'utilisateur demandant l'accès à un fichier distant est bien déclaré sur la machine serveur. En effet, il ne faut pas que le numéro de l'utilisateur soit déjà attribué à un autre utilisateur sur la machine distante. Ceci suppose une administration commune des différentes machines, par exemple par la distribution des informations du fichier `/etc/passwd` au moyen des ¡¡ pages jaunes ¿¿ (`ypcat -k passwd`).

Le deuxième problème est un problème de sécurité : il ne faut pas qu'un pirate puisse simuler les requêtes d'un autre utilisateur. Pour lutter contre cela, il faut mettre en œuvre des mécanismes de cryptage pour que le démon distant, qui s'exécute au nom du super-utilisateur, ait toute confiance en les demandes de création de serveurs.

(3.4) (1,5 points)

Dans un système centralisé, lorsqu'un processus exécute un `fork()`, les structures de gestion des fichiers ouverts sont mises à jour pour prendre en compte l'augmentation du nombre de processus référençant les fichiers ; il faut donc faire de même avec les fichiers ouverts à travers le réseau. Il faut donc détourner la routine `fork()` afin que le processus gestionnaire de fichiers s'exécutant au nom de l'utilisateur sur le serveur puisse lui aussi forker pour créer un processus gestionnaire propre au nouveau processus. On préfèrera cette solution à celle d'un unique processus gestionnaire par utilisateur, afin qu'il ne soit pas limité en nombre de descripteurs de fichiers ouverts par processus, qui est borné.

(3.5) (1 point)

Si deux processus accèdent en modification au même fichier distant, les deux processus gestionnaires s'exécutant au nom de l'utilisateur sur le serveur et réalisant effectivement la mise à jour vont être en

conflit, et le dernier à écrire écrasera les modifications du premier. Le bit `O_EXCL` est efficace s'il est utilisé par les processus gestionnaires lors de l'ouverture des fichiers distants ; il faut donc transmettre les bits d'ouverture sur le réseau à l'intérieur du message de demande d'ouverture distante.

Si un fichier verrou doit être créé, il doit se trouver dans le même répertoire que le fichier qu'il protège, là où tous les processus concernés peuvent le voir, c'est-à-dire sur le serveur.

(3.6) (1 point)

Comme l'architecture envisagée est basée uniquement sur une modification de la bibliothèque standard, un programme binaire obtenu d'une personne tierce ne pourra bénéficier des fonctionnalités d'accès en réseau que s'il peut utiliser les fonctions modifiées, c'est-à-dire s'il a fait l'objet d'une édition de liens dynamique.