E.N.S.E.I.R.B. Année 2000-2001
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)

Le principe d'une machine virtuelle est d'émuler de façon logicielle le fonctionnement d'une première machine (réelle ou abstraite) sur une deuxième. Les machines virtuelles ont deux avantages principaux :

– la transparence : l'existence d'une machine virtuelle permet de dissocier l'exécution des programmes du support physique. Ainsi, sur l'IBM 370, le système VM/370 permettait-il, en offrant de façon transparente plusieurs copies conformes de la machine physique, de supporter simultanément plusieurs systèmes d'exploitation différents alors que ceux-ci avaient été écrits pour la machine physique sous-jacente. Par ce biais, il est également possible d'offrir une compatibilité ascendente.
– l'abstraction : la définition d'une machine virtuelle peut être très éloignée de celle d'une machine réelle. Dans le cas de Java, la machine virtuelle JVM est une machine générique de haut niveau pouvant être implémentée facilement sur tout type de processeur actuel. La définition de cette machine abstraite permet, lors du chargement et de l'interprétation des byte-codes des programmes Java, de réaliser de nombreuses vérifications de sécurité.

Question 2 (3 points)

Dans les systèmes d'exploitation préemptifs, l'ordonnanceur a la possibilité d'interrompre de lui-même le procesus en cours d'exécution (sur déclenchement d'une interruption horloge) pour attribuer le processeur à un autre processus, alors que dans les systèmes d'exploitation non-préemptifs, c'est au processus courant de rendre la main à l'ordonnanceur.

Les systèmes d'exploitation préemptifs sont plus complexes à écrire, car il faut protéger les structures de données du système contre les interruptions, et penser aux interblocages entre processus. À l'opposé, les systèmes d'exploitation non préemptifs sont plus simples, car l'ordonnanceur n'est appelé explicitement qu'à des endroits parfaitement définis des appels système : généralement, on trouve au début du code de chaque appel système un appel à une routine qui vérifie si le quantum de temps du processus courant est épuisé, et appelle l'ordonnanceur si c'est le cas). Les systèmes non préemptifs sont donc moins lourds, et donc plus efficaces, que les systèmes préemptifs, mais sont plus fragiles vis-à-vis du multi-processus, comme on le verra ci-dessous.

Les programmes s'exécutant sur des systèmes non préemptifs sont plus délicats à écrire, car le programmeur doit penser à rendre la main aux autres processus : pour chaque fragment de code de taille suffisante, le programmeur doit regarder si le fragment réalise des appels système (dans ce cas, l'appel à l'ordonnanceur se fera dans l'appel système), ou bien alors introduire explicitement des appels à la routine de vérification du quantum, faute de quoi le processus actif ne rendra jamais la ressource processeur. De même, lorsque le processus actif est bogué et ne rend pas la main, l'ensemble du système est paralysé.

Question 3 (5 points)

(3.1) (1,5 points)

Le mécanisme de partage de pages entre processus avec copie en écriture est un mécanisme d'optimisation de l'occupation mémoire. Il permet à plusieurs processus qui exécutent le même programme de partager leurs pages physiques identiques (principalement les pages de code, car celles-ci sont en lecture seule), en faisant en sorte que les entrées des tables des pages des processs pointent sur ces pages physiques, qui sont marquées comme partagées.

Lorsqu'un processus partageant une de ces pages souhaite y faire une modification, celle-ci ne doit pas être visible des autres processus (puisque les espaces d'adressage des processus sont censés être privés). Pour ce faire, lorsqu'une écriture est demandée sur une page partagée, le système réalise une copie de la page en question, réservée au processus ayant demandé la modification, les autres processus continuant à partager l'ancienne version de la page.

(3.2) (1 point)

Dans le cas de systèmes d'exploitation multi-utilisateurs multi-processus, ce mécanisme permet de ne conserver en mémoire qu'une seule copie des pages possédées par un programme utilisé par plusieurs utilisateurs (éditeur de texte, navigateur, ...). Il permet ainsi de limiter le va-et-vient (¡¡ *swapping* ¿¿) lors de l'ordonnancement des processus. Il est également très utile lors des `fork()`, puisque le père et le fils partageront l'ensemble de leurs pages (code et même données) jusqu'à ce qu'ils y fassent l'un ou l'autre des modifications.

(3.3) (1 point)

Dans le cas de systèmes d'exploitation mono-utilisateurs, le gain est moindre, car moins de processus identiques sont lancés par un unique utilisateur (on lance rarement plusieurs navigateurs, éditeurs, etc.). Néanmoins, cette fonctionnalité est toujours extrêmement utile lors des `fork()`, toujours très nombreux.

(3.4) (1,5 points)

Dans la majorité des cas, les appels système `fork()` sont presque immédiatement suivis d'un appel `exec()` au niveau du fils, alors que le père continue son traitement ; c'est par exemple le cas pour tous les lancements de processus par les interpréteurs de commandes, et les démons réseau.

Si le processus père est ordonnancé en premier, alors chaque fois que le père effectue une écriture en mémoire dans une de ses pages de données, le système de gestion de la mémoire copie la page correspondante afin que le fils possède une copie conforme à l'état de la mémoire au moment du `fork()`. Lorsque le processus fils prend la main par la suite, et réalise son `exec()`, il détruit ses copies des pages de données, qui n'ont donc pas servi.

Si le fils s'exécute en premier, il détruit presque immédiatement ses références aux pages de données partagées avec son père en réalisant son `exec()`, et le processus père, ne nouveau seul propriétaire de ses pages de données, n'a pas à supporter le surcoût des recopies de pages.

Question 4 (3 points)

(4.1) (0,5 points)

La règle fondamentale qu'Unix cherche à respecter lorsqu'il réalise des entrées-sorties est de les rendre atomiques vis-à-vis des utilisateurs : si plusieurs entrées-sorties sont simultanément lancées sur la même destination, elles seront sérialisées, et chaque opération entamée devra être terminée avant qu'une autre puisse avoir lieu, même si l'opération en cours provoque l'endormissement du processus.

(4.2) (1 point)

S'il ne reste plus assez de place dans le pipe pour contenir les données que le processus souhaite

écrire (mais que la capacité totale du pipe est quand-même supérieure à la taille totale des données à écrire), le système doit nécessairement endormir le processus. On peut imaginer deux méthodes pour ce faire :

- le système endort immédiatement le processus, en attendant que le ou les lecteurs consomment suffisament de données pour que l'écrivain puisse réaliser son écriture de façon atomique. Cette technique garantit l'atomicité, mais pas l'équité lors de la sérialisation : si un autre écrivain, arrivé plus tard, a moins de données à écrire, il peut passer devant le premier ;
- le système écrit autant de données que possible dans le pipe, puis endort le processus au milieu de son appel système, en attente de place libre pour terminer son opération d'écriture. Dans ce cas, pour conserver l'atomicité des entrées-sorties du point de vue de l'utilisateur, il faut garantir que seul le processus ayant entamé l'entrée-sortie pourra écrire sur le pipe lorsque celui-ci commencera à se vider. Le meilleur moyen pour cela est d'attribuer à ce processus une priorité de réveil supérieure à celle des autres.

La deuxième solution est la plus raisonnable, car elle permet une sérialisation équitable sans surcoût important.

(4.3) (1,5 points)

Si la taille des données à écrire dépasse la capacité totale du pipe, le processus écrivain doit nécessairement s'endormir au moins une fois au milieu de son entrée-sortie pour réaliser celle-ci. Dans ce cas, on est obligé, pour respecter le principe que l'on s'est donné, de mettre en œuvre la deuxième solution énoncée ci-dessus. Ainsi, on ne sera pas en contradiction avec notre règle fondamentale.

Question 5 (6 points)

(5.1) (1 point)

Comme les droits d'accès à un même fichier peuvent différer entre deux ouvertures différentes, ces droits ne peuvent se trouver dans la structure i-node mémoire, et doivent donc être localisés dans les structures descripteurs des fichiers. Il en est de même pour la position courante de lecture et d'écriture dans le fichier.

(5.2) (1,5 points)

Lorsqu'un appel système `open()` est réalisé, l'i-node disque du fichier à ouvrir est recopié dans une case du tableau global des i-nodes mémoire si le fichier pré-existait sur disque, ou bien est initialisé si le fichier doit être créé ; dans tous les cas, le compteur de références de l'i-node mémoire est mis à 1. Une case du tableau local des structures descripteurs du processus est affectée au fichier, ses droits sont positionnés comme indiqué dans les paramètres de l'appel système, et

Lorsqu'un deuxième processus (ou bien le même) souhaite ouvrir le même fichier, le système détecte qu'il existe déjà un i-node mémoire correspondant à ce fichier. Au lieu d'allouer un nouvel i-node mémoire, il incrémente donc le compteur de références de un, mais alloue cependant une nouvelle structure descripteur de fichier dans la `u_aera` du processus, qui pointera sur l'i-node mémoire existant, recevra les nouveaux droits, et indexera le début du fichier.

(5.3) (1 point)

Il n'est pas possible, avec cette implémentation, de mettre en œuvre l'appel système `dup()`. En effet, `dup()` sert à dupliquer les index de descripteurs de fichier (en particulier pour les redirections d'entrées-sorties), et donc à faire que deux index de descripteurs différents fassent référence au même fichier ouvert. Comme, avec cette implémentation, la position courante en lecture/écriture est stockée dans la structure descripteur de fichier, dupliquer la structure de descripteur de fichier revient à copier la position courante dans la nouvelle structure descripteur de fichier. On aura donc deux positions courantes qui pourront évoluer indépendament, ce qui ne respecte pas la sémantique de `dup()`.

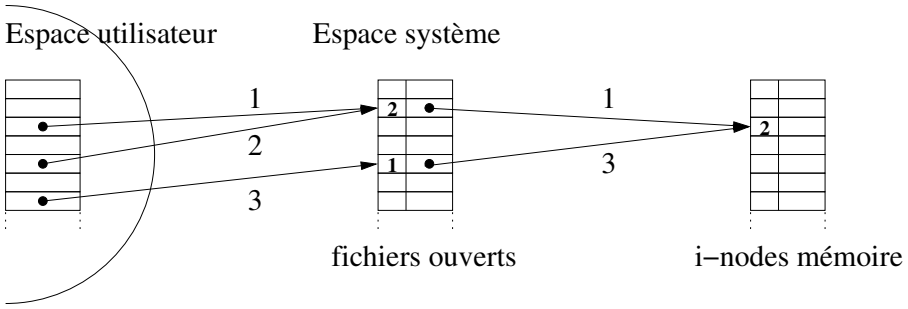
(5.4) (1 point)

Cette structure intermédiaire permet de résoudre les problèmes qui empêchaient la mise en œuvre de

l'appel système `dup()`. Pour cela, deux champs doivent s'y trouver. On dispose ainsi d'une structure définissant au niveau du système les fichiers ouverts, qui peuvent être référencés simultanément à partir de plusieurs descripteurs de fichiers du même processus (cas d'un `dup()`), voire à partir de plusieurs processus (cas d'un `fork()`, non gérable auparavant, pour les mêmes raisons).

(5.5) (1 point)

Lorsqu'un processus ouvre un fichier avec `open()` (étape 1), le duplique avec `dup()` (étape 2), et re-ouvre le même fichier (étape 3), on obtient l'état suivant.



(5.6) (0.5 points)

Comme plusieurs descripteurs de fichiers peuvent faire référence à la même structure dans le tableau système des fichiers ouverts, chacune d'elles doit posséder un compteur de références, dont la décrémentation à zéro indiquera que le fichier doit être fermé, ce qui décrémentera à son tour le compteur de l'i-node mémoire.