E.N.S.E.I.R.B. Année 2001-2002
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 (5 points)

(1.1) (1 point)

Sur les systèmes à mémoire paginée, la MMU est un circuit matériel s'intercalant entre le processeur et le bus mémoire (il est d'ailleurs le plus souvent intégré sur la puce du processeur elle-même), et chargé de convertir les adresses virtuelles manipulées par le processeur en adresses physiques. Afin de limiter la taille de la table de correspondance entre numéros de pages virtuelles et numéros de pages physiques, cette table est organisée de façon hiérarchique (c'est-à-dire arborescente), afin que seules les portions utilisées de la table des pages soient effectivement présentes en mémoire centrale.

Du fait de la structure hiérarchique de la table des pages, chaque demande d'opération mémoire se traduit par plusieurs opérations de lecture sur les différents niveaux de la table des pages, ce qui peut être prohibitif en pratique. Le rôle du TLB (" *Translation Lookaside Buffer* ") est de limiter ces acces coûteux aux tables des pages, en conservant dans une petite mémoire cache auxiliaire les correspondances entre numéros de pages virtuelles et physiques les plus récemment demandées. De par les principes de localité, la probabilité de réutiliser les correspondances déjà mémorisées est très grande, ce qui rend le TLB très efficace.

(1.2) (2 points)

Afin de gérer la MMU et le TLB, le système d'exploitation doit pouvoir :
- traiter les interruptions générées par la MMU lors d'accès provoquant des défauts de page ou bien des accès illégaux ;
- mettre à jour des entrées individuelles à la MMU, par exemple lorsque le nombre de pages physiques allouées à un processus augmente (augmentation de la taille de la zone de données ou de pile du processus, ou bien chargement d'une page utile à partir du va-et-vient après un défaut de page) ou diminue (déplacement d'une page ancienne vers le va-et-vient). Dans le cas où une page ancienne est migrée sur disque, le TLB doit aussi être purgé de la référence à cette page, s'il la contenait ;
- mettre à jour l'adresse de la page racine de la hiérarchie de la table des pages lors d'un changement de contexte entre processus. Dans ce cas, il faut également pouvoir purger le TLB, afin de supprimer toutes les correspondances désormains invalides (on rappelle que tous les processus ont le même espace virtuel) ;
- accéder aux drapeaux associés aux pages (ancienneté, modification) et gérés de façon matérielle par la MMU, pour permettre à l'algorithme de remplacement de pages d'être le plus efficace possible.

(1.3) (2 points)

Comme il est dit ci-dessus, lors de chaque changement de contexte, le système doit purger le TLB afin que les correspondances mémorisées par l'ancien processus, et seulement valables pour lui, ne soient pas utilisées par le nouveau processus. Qui plus est, les emplacements physiques des pages d'un processus peuvent changer lors de son rechargement à partir du va-et-vient, ce qui rend inopérant la sauvegarde du contenu de la TLB à chaque changement de contexte.

Ceci pose un problème de performance, puisqu'après chaque changement de contexte un processus ne possède plus d'entrées valides dans le TLB, et que la MMU devra effectuer de coûteux accès à la table des pages du processus lors des premiers accés mémoire, jusqu'à ce que le TLB soit de nouveau efficace.

Une amélioration possible consiste à maintenir dans un registre de la MMU le numéro du processus courant, et à préfixer chaque entrée du TLB par le numéro du processus qui l'a écrite. Ainsi, une entrée du TLB n'est valide que si son numéro de page virtuelle correspond au numéro de page virtuelle recherché, et que son numéro de processus propriétaire correspond au numéro de processus courant mémorisé dans le registre de la MMU. Cette fonctionalité évite d'avoir à purger le TLB lors de chaque changement de contexte, le système n'ayant qu'à changer le numéro de processus courant dans le registre de la MMU. En revanche, lors de la terminaison d'un processus, le TLB doit toujours être purgé des entrées indicées par le numéro du processus venant de se terminer, afin de ne pas produire d'incohérence si un nouveau processus de même numéro vient à être recréé.

Ce système est très efficace pour de petits processus. Si les processus sont plus gros, le système fonctionne toujours, mais la probabilité qu'une ancienne entrée de TLB ne soit pas remplacée par une nouvelle avant qu'elle ne soit réutilisée diminue, et l'effet de cache entre deux élections successives du même processus est donc moins efficace.

Question 2 (3 points)

(2.1) (1 point)

La taille d'une entrée de la table des pages est de 4 octets (32 bits), donc la taille d'une page de la table des pages est de $2^{10} \times 4 = 2^{12}$ octets, soit 4 ko. La hiérarche des pages correspond à une page racine et à 2^{10} pages de premier niveau indexées par la page racine, donc sa taille totale est de $2^{12} \times \left(1 + 2^{10}\right) = 2^{22} + 2^{12}$ octets, soit un peu plus de 4 Mo.

Il faut cependant remarquer que les adresses contenues dans les entrées des pages de la table des pages sont toutes des adresses physiques (en mémoire ou dans le va-et-vient). Ceci veut dire que, bien que la table des pages occupe des pages physiques dans la mémoire centrale au détriment du processus, ces pages n'apparaissent pas dans l'espace d'adressage virtuel du processus. L'espace d'adressage virtuel utile de chaque processus (et du système) est donc de 2^{32} octets, quel que soit le nombre de pages physiques occupées par la table des pages.

(2.2) (2 points)

La fraction de l'espace mémoire physique d'un processus (c'est-à-dire le nombre total de pages physiques occupées par le processus, qui est équivalent à l'espace disque occupé lorsque le processus est swappé sur disque), varie selon la taille du processus, mais également selon la disposition des pages du processus allouées, qui conditionne le nombre de pages de premier niveau devant être allouées.

Le ratio de valeur maximale est obtenue pour un processus constitué d'une unique page, pour lequel deux pages doivent être allouées dans la table des pages (une pour la page racine, et une pour la page de premier niveau indexant la page du processus). On obtient alors un ratio de $\frac{2}{1} = 2$.

Une page de premier niveau sert complètement lorsque toutes ses entrées servent à indexer des pages de l'espace d'adressage du processus. De même, la page racine sert complètement lorsque toutes ses entrées sont utilisées. Le ratio de valeur minimale est donc obtenu lorsque l'ensemble des pages du processus est alloué. Il est alors de $\frac{1+1024}{1024\times1024} = \frac{1025}{1048576} \geq \frac{1}{1024}$.

Question 3 (6 points)

(3.1) (2 points)

Pour que tous les caractères émis par `f()` sur la sortie standard puisse être placés dans `str`, il faut

rediriger les caractères émis par `f()` afin de pouvoir les récupérer à partir d'un descripteur ouvert en lecture, alors que la sortie standard est un descripteur ouvert seulement en écriture. Pour cela, on peut utiliser un pipe, dans l'entrée duquel on redirigera la sortie standard, et à partir duquel on pourra lire les caractères produits par `f()`. Ceci peut être réalisé au moyen du code suivant (les traitements d'erreur ont été omis pour simplifier l'écriture).

```
{
  char            str[TAILLE_ÉNORME_ET_TOUJOURS_SUFFISANTE];
  int             fdpipe[2];  /* Descripteurs du pipe           */
  int             fdsave;     /* Descripteur de sauvegarde de 1 */

  ...
  pipe (fdpipe);              /* Création du pipe                       */
  fdsave = dup (1);           /* Sauvegarde de la sortie standard       */
  dup2 (fdpipe[1], 1);        /* Redirection de la sortie vers le pipe  */
  close (fdpipe[1]);          /* Attention: un seul écrivain potentiel! */
  f ();
  dup2 (fdsave, 1);           /* Fermeture du pipe et remise en place de la sortie */
  close (fdsave);             /* Attention: une seule sortie potentielle!          */

  read (fdpipe[0], str, TAILLE...ANTE); /* Lecture en une seule fois */
  close (fdpipe[0]);          /* Libère les ressources du pipe       */
  ...
  utiliser_résultat (str);
}
```

Dans le code ci-dessus, on a bien pris soin de fermer tous les descripteurs d'écriture du pipe avant d'entamer la lecture de ses données, faute de quoi on aurait un interblocage, puisqu'alors le `read()` bloquerait en attente d'un écrivain potentiel (nous!) sans jamais renvoyer 0.

On effectue la lecture des données en une seule fois car on est sûr que l'ensemble des données à lire est présent dans le pipe avant qu'on entame la lecture.

(3.2) (1 point)

Si la taille des données redirigées est trop importante, on risque d'être bloqué lors des écritures réalisées dans la fonction `f()` parce que le pipe sera plein, et sans jamais pouvoir être réveillé puisque les lectures sur le pipe ne pourront avoir lieu qu'à la sortie de `f()`. On se trouvera donc en situation d'interblocage... tout seul.

(3.3) (2 points)

On peut résoudre ce problème en déportant l'exécution de `f()` dans un processus indépendant. Par cette augmentation du degré de parallélisme du programme, on casse l'auto-dépendence en lecture-écriture potentiellement dangereuse, en la remplaçant par une relation simple de type producteur-consommateur. Le code correspondant est le suivant (ici également sans traitements d'erreur).

```
{
  char            str[TAILLE_ÉNORME_ET_TOUJOURS_SUFFISANTE];
  int             fdpipe[2];  /* Descripteurs du pipe           */
  int             fdsave;     /* Descripteur de sauvegarde de 1 */
  char *          p;          /* Index de parcours de la chaîne */

  ...
  pipe (fdpipe);              /* Création du pipe                       */
  if (fork () == 0) {         /* Si on est le fils                      */
    close (fdpipe[0]);        /* Libère les ressources inutilisées      */
    dup2 (fdpipe[1], 1);      /* Redirection de la sortie vers le pipe  */
    close (fdpipe[1]);        /* Attention: un seul écrivain potentiel! */
    f ();
    close (1);                /* Fermeture de l'entrée du pipe */
    exit (0);                 /* Fin du fils producteur        */
  }
  else {
    close (fdpipe[1]);        /* Attention: un seul écrivain!           */
    for (p = str; read (fdpipe[0], p, 1) > 0; p ++) ; /* Lit le pipe    */
    *p = '\0';                /* Marque la fin de la chaîne             */
    close (fdpipe[0]);        /* Libère les ressources du pipe          */
    wait ();                  /* Attente de la fin du fils (dézombifié) */
  }
  ...
  utiliser_résultat (str);
```

```
}
```

Dans le code ci-dessus, on effectue la lecture des données dans une boucle, car on n'a aucune garantie que l'ensemble des données soit présent dans le pipe lors de la première lecture.

(3.4) (1 point)

En dehors du lancement du processus supplémentaire, cette deuxième méthode n'est pas beaucoup plus coûteuse en ressources que la première. En effet, les gestionnaires de MMU modernes implémentent un mécanisme qui ne duplique que les pages modifiées par l'un des processus fils ou père, les autres pages étant partagées en lecture. De fait, ces pages étant déjà présentes dans la mémoire physique du système lors du `fork()`, le surcoût sera marginal eu regard à l'avantage de supprimer la possibilité d'auto-interblocage qui rend le premier programme faux.

Question 4 (6 points)

(4.1) (3 points)

Les fonctionnalités que l'on attend d'une telle API sont de :

- permettre à un processus de déclarer et d'attacher dans son espace d'adressage (virtuel) une zone de mémoire partagée ;
- permettre à d'autres processus d'attacher à leur espace d'adressage une zone de mémoire partagée déjà existante. Comme les processus souhaitant attacher une zone de mémoire partagée peuvent ne pas être cousins (c'est-à-dire avoir un ancêtre commun), il faut mettre en place un mécanisme de nommage/adressage (à l'image par exemple des numéros de ports pour les sockets) permettant d'identifier la zone de mémoire partagée à laquelle on souhaite s'attacher. De plus, un mécanisme de droits doit être mis en place afin d'empêcher des processus non désirés d'espionner une zone de mémoire partagée ;
- permettre à un processus de libérer son attachement à une zone de mémoire partagée. Lorsqu'il ne reste plus de processus attachés à une zone, celle-ci doit être supprimée du système.

Les deux premières fonctionnalités pouvant être intégrées dans la même routine, on aura une routine de création/attachement, et une routine de libération. Les prototypes proposés pour ces deux fonctions sont donc les suivants :

```
void *    shm_attach (char * key, size_t size, int flags);
int       shm_detach (void * memptr);
```

Le sens des différents paramètres est le suivant :

`key` : Clé texte permettant de nommer la zone de façon non ambiguë. On évite ainsi le plus possible que deux programmes différents utilisent le même identificateur de zone, ce qui est plus probable avec des numéros de ports pour lesquels les programmeurs ont tendence à prendre des valeurs " symboliques " (comme 42, 357, ou 666, ... selon leurs références culturelles et humeurs) ;

`size` : Nombre d'octets demandés lors de la création de la zone. Lorsqu'on souhaite juste s'attacher à une zone que l'on suppose déjà existante, on peut spécifier la taille minimale que doit faire cette zone pour que l'appel n'échoue pas (c'est-à-dire la taille minimale en dessous de laquelle on ne peut travailler dans cette zone). La valeur 0 peut d'ailleurs signifier que l'on prendra cette zone quelle que soit sa taille, à charge pour le créateur de signifier la taille aux autres processus (valeur entière placée en début de zone ou autre) ;

`flags` : Ensemble de drapeaux définissant les droits et modes de création/attachement à la zone. On peut s'appuyer sur les droits d'`open()` pour imaginer le fonctionnement de cette API. En ce qui concerne les droits d'accès, on peut s'appuyer sur les droits Unix : un processus qui crée une zone la crée avec des droits d'accès pour les processus lancés avec le même identificateur d'utilisateur, pour ceux lancés avec le même identificateur de groupe, et pour tous les autres processus. De même, il faut un drapeau similaire à `O_CREAT` pour signifier qu'on accepte de créer la zone si personne ne l'a créé avant nous, et un drapeau similaire à `O_EXCL` pour signifier que l'on renverra une erreur si la zone existe déjà. On suppose ici que tout processus accédant à une zone partagée peut à la

fois y lire et y écrire (mais ceci peut se discuter, et il n'est pas difficile de positionner des droits en lecture seule au moyen des bits de contrôle de la MMU).

`memptr` : Pointeur sur une zone de mémoire partagée à libérer.

(4.2) (3 points)

Dans une architecture mémoire moderne, le partage de pages entre processus est réalisé par la MMU ; c'est ce mécanisme qui permet de rendre l'appel système `fork()` efficace en permettant le partage des pages de données entre un processus père et ses fils, tant qu'elles ne sont pas modifiées (mécanisme de " *copy on demand* "). On peut s'appuyer sur ce mécanisme pour mettre en place la mémoire partagée : lorsqu'une certaine quantité de mémoire partagée est demandée par un processus, on alloue le nombre de pages correspondant (arrondi au nombre de pages supérieur), et l'on renvoie au processus demandeur l'adresse virtuelle de la première page demandée. Lorsqu'un autre processus demande à accéder à la zone de mémoire partagée, le système recherche dans la table des pages du nouveau processus une plage contiguë de pages non encore allouées de taille suffisante, insère les pages partagées à cet emplacement, et renvoie au processus l'adresse virtuelle de la première page dans son espace virtuel (deux processus accédant à la même zone de mémoire partagée peuvent donc la voir chacun à deux adresses différentes, ce qui n'est pas gênant).

Afin que le système garde la trace des zones de mémoire partagée et puisse les libérer quand elles ne sont plus accédées, chaque zone de mémoire partagée doit être enregistrée auprès du système dans une table des zones actives, dont chaque entrée est une structure comprenant :

- la valeur de la clé (chaîne de caractères) identifiant cette zone mémoire ;
- le nombre de processus référençant actuellement la zone ;
- la taille déclarée de la zone ;
- les valeurs de droits de la zone (UID et GID du processus créateur, et les droits d'accès qu'il a positionné à la création) ;
- le nombre de pages mémoire utilisées, ainsi que leur emplacement en mémoire physique/swap, toutes informations nécessaires pour pouvoir insérer ces pages dans les tables des pages des processus qui en feraient la demande.

Chaque processus doit également posséder, dans sa structure `u`, la liste des zones de mémoire partagée qu'il référence, qui peut être implémentée comme un tableau contenant les indices de ces zones dans la table globale du système. Cette liste sert à incrémenter les compteurs de référence de toutes les zones qu'il possède lors d'un `fork()` (puisque le fils hérite de l'espace de données du père), et les décrémenter dans le cas d'un `exec()` ou d'un `_exit ()`. Toute zone dont le compteur de référence tombe à zéro est supprimée de la table du système.

Pour voir comment la mémoire partagée est réellement implémentée au niveau utilisateur, vous pouvez consulter les pages de manuel des appels système `shmat()` et `shmdt()` .