Examen du cours Système de Majeure 2.

Durée 2 heures

Les appendices B et C du cours, éventuellement annotées, sont la seule documentation autorisée.

Dans les solutions, on pourra utiliser les fonctions du module `Misc` *décrites dans l'appendice B.7 en les appelant explicitement avec le préfixe* `Misc`.

On s'efforcera de donner des explications à la fois précises et concises.

Les quatre parties sont indépendantes. Les questions soulignées peuvent être plus difficiles ou plus longues que les autres de la même partie.

Partie 1: Fork - Execv

1) Que fait l'appel système `execv` *file argv* ?

Réponse: *Il remplace le programme en cours d'exécution par le programme file appelé avec les arguments argv.*

2) Peut-il échouer en raison d'une mauvaise valeur de `argv` (justifier très brièvement) ?

Réponse: *Non, c'est le programme appelé qui analyse les arguments et non le programme appelant.*

3) Est-ce que le *comportement* des signaux est hérité lors de cet appel système (justifier très brièvement) ?

Réponse: *Non, il ne peut pas être hérité car le code décrivant le comportement des signaux est du code utilisateur, écrasé par* `execv`.

4) Donner les principales étapes du point de vue du système lors de l'appel système `execv` *file argv* ?

Réponse: *Le système cherche le fichier désigné par le chemin (peut échouer). Ce fichier doit être un fichier exécutable (peut échouer). Le système charge le code et les données, en fait, met à jour la table des pages pour qu'elle pointe sur le code et les données (peut échouer s'il n'y a plus assez de mémoire pour allouer la table des pages).*

5) Que deviennent, par défaut, les descripteurs ouverts lorsqu'un processus fait `execv` ? Quel est l'intérêt de ce choix ?

Réponse: *Les descripteurs restent ouverts et à la même position. cela permet au nouveau programme de communiquer de la même façon que pouvait communiquer l'ancien programme.*

6) Quel est, à votre avis, la réponse à la question 3) pour l'appel système `fork()` (justifier brièvement) ?

Réponse: *Par contraste,* `fork` *partage toute la mémoire entre les deux processus, sauf le numéro de processus, la valeur de retour, et quelques variables système, donc il n'y a pas de raison de changer le* comportement *des signaux (bien que techniquement on pourrait les remettre à leur comportement par défaut).*

7) Même question que 5) pour l'appel système `fork`.

Réponse: *Les descripteurs restent également ouverts. Cela permet au fils de communiquer avec son père au travers d'un tuyau créé avant le* `fork`*. De plus, pour chaque descripteur, celui du fils et celui du père pointent vers la même entrée dans la table des fichiers : ce que l'un lit, l'autre ne le lira pas : si le fils* puis *le père écrivent dans un fichier de log ouvert avant le fork, leurs écritures ne s'écrasent pas (au pire, elles seront entrelacées).*

□

Partie 2: Mkpath

On rappelle qu'en général les appels systèmes qui accèdent au système de fichiers provoquent une erreur `ENOENT` lorsqu'on leur demande d'accéder à un chemin qui n'existe pas.

1) Écrire une fonction `mkpath` de type `string -> int -> unit` telle que l'appel `mkpath` *path perm* crée le répertoire *path* ainsi que tous les répertoires sur le chemin *path* qui n'existent pas. Tous les répertoires créés le seront avec les permissions *perm*. Le chemin *path* est absolu ou relatif. Si la commande `mkpath` échoue, on ne cherchera pas à retirer les répertoires qui auraient déjà pu être créés. Si un chemin intermédiaire de p existe mais n'est pas un répertoire, on déclenchera une erreur `ENOTDIR` comme le ferait la commande `mkdir` (et la plupart des commandes qui prennent des chemins en arguments).

Réponse:

```
open Unix;;

let rec mkpath path perm =
  try
    let st = stat path in
    if st.st_kind <> S_DIR then raise (Unix_error (ENOTDIR, "mkpath", path));
  with Unix_error (ENOENT, _, _) ->
    let parent = Filename.dirname path in
    if parent <> path then mkpath parent perm;
    mkdir path perm;;
```

2) On suppose que le répertoire `./A` existe avec les droits `0740` (*i.e.* `rwxr-----` en notation symbolique). Décrire une valeur de *perm* (en notation octale ou symbolique, ou en français) avec laquelle le répertoire `A/B` a été créé mais le répertoire `A/B/C` n'a pas pu l'être.

Réponse: 0o000

3) Donner une autre raison pour laquelle la création du répertoire `A/B/C` peut échouer alors que celle du répertoire `A/B` a réussi.

Réponse: *Si le système de fichier est plein, ou si un autre processus a coupé l'herbe sous le pied de la commande* `mkpath` *(en effaçant un répertoire intermédiaire déjà créé...)*

4) On souhaiterait que la commande réussisse quand même dans une majorité des cas décrits en 2) comme produisant des erreurs, tout en assurant au final que les répertoires créés

auront bien les droits demandés. Donner une variante `mkpath_plus` de `mkpath` réalisant cela.
Réponse:

```
let mkpath_plus path perm =
  let user = 0o700 in
  let finish dirs = List.iter (fun file -> chmod file perm) dirs in
  let rec mkpath path =
    try
      let st = stat path in
      if st.st_kind <> S_DIR then raise (Unix_error (ENOTDIR, "mkdirp", path));
      []
    with Unix_error (ENOENT, _, _) ->
      let parent = Filename.dirname path in
      let dirs = if parent <> path then mkpath parent else [] in
      let () = try mkdir path user with exn -> finish dirs; raise exn in
      path :: dirs in
  finish (mkpath path);;
```

□

Partie 3: Descripteurs en accès direct

On rappelle que l'appel système `lseek` lève l'exception `ESPIPE` si le descripteur auquel il est appliqué n'est pas en accès direct (*i.e.* ne supporte pas `lseek`).

1) Écrire une fonction `seekable` de type `Unix.file_descr -> bool` qui teste si le descripteur reçu en argument est en accès direct.
Réponse:

```
open Unix
let seekable fd =
  try let _ = lseek fd 0 SEEK_CUR in true
  with Unix_error (ESPIPE, _, _) -> false
```

2) Écrire une fonction `unlinkf` qui se comporte comme `unlink`, mais ne lève pas d'erreur s'il n'est pas possible d'effectuer l'opération.
Réponse:

```
let unlinkf filename = try unlink filename with _ -> ();;
```

3) On vous donne la fonction suivante :

```
0 let rec create_file_descr() =
1   let tmpdir = "/tmp" in
2   let rec open_tmp n =
3     try
4       let name = Filename.concat tmpdir (string_of_int n) in
5       let descr = openfile name [ O_CREAT; O_RDWR; O_EXCL ] 0 in
6       unlinkf name;
7       descr
8     with Unix_error (EEXIST, _, _) ->
9       open_tmp (n+1) in
```

```
10    open_tmp (getpid());;
```

Que fait cette fonction ?

Réponse: *Cette fonction crée et ouvre un nouveau fichier temporaire en lecture et écriture et retourne le descripteur ouvert après avoir retiré le nom temporaire du système de fichier. Le fichier continue à exister tant que le descripteur reste ouvert, mais ne peut plus être réouvert.*

4) Y a-t-il quelque chose à faire pour se prémunir contre un risque éventuel, si, après le retour de la fonction `create_file_descr`, un fichier est à nouveau créé avec le même nom que celui qui a servi à créer `descr` ?

Réponse: *Non, le second fichier sera créé avec un autre* `inode` *et sera donc complètement indépendant.*

5) Est-il pour autant garanti que le descripteur de fichier `create_file_descr` donne un accès privé dans le sens où ce qui est lu ou écrit dans `desc` n'a pu être écrit ou lu que par le processus courant ou un de ses descendants ?

Réponse: *Non, car entre les lignes 5 et 6 un autre processus a pu également ouvrir ce fichier (après en avoir changé le mode) ou a pu faire un lien dur dessus.*

6) On suppose que le programme Unix `prog` prend un seul argument qui est un nom de fichier ou bien zéro argument et dans ce cas utilise l'entrée standard en supposant qu'elle est en accès direct.

Écrire un programme `wrap_prog` de telle façon que `wrap_prog` se comporte comme `prog`, mais fonctionne également lorsque qu'il est appelé sans argument et que l'entrée standard n'est pas en accès direct. Pour cela, dans ce cas, il recopie l'entrée standard dans un descripteur auxiliaire en accès direct et fait ce qu'il faut pour que la commande `prog` lise dans ce descripteur.

On supposera que le programme `prog` se trouve dans le chemin d'exécution (variable d'environnement `PATH`). On supposera que le code des trois premières questions (énoncés ou réponses) a déjà été placé dans un fichier `seek.ml`.

Réponse:

```
open Unix;;
let main () =
  let () =
    if Array.length Sys.argv < 2 && not (Seek.seekable stdin) then
      let descr = Seek.create_file_descr () in
      Misc.retransmit stdin descr;
      ignore (lseek descr 0 SEEK_SET);
      dup2 descr stdin;
      close descr in
  execvp "prog" Sys.argv in
handle_unix_error main();;
```

□

Partie 4: Le coût de la concurrence

On rappelle qu'un serveur séquentiel n'accepte une nouvelle requête que lorsque la précédente est complètement traitée alors qu'un serveur concurrent accepte plusieurs requêtes en pa-

rallèle et est capable de les traiter de façon entrelacée en utilisant, par exemple, (a) plusieurs processus, (b) des coprocessus, (c) `select` et un seul processus. Dans tout cet exercice, on suppose que la machine est mono-processeur.

1) Expliquer pourquoi, bien que la machine soit mono-processeur, un serveur concurrent est souvent plus efficace.

Réponse: *Si le client communique avec le serveur lentement, le temps entre la réponse du serveur au client et la prochaine question du client au serveur peut être bien supérieur au temps de calcul de la réponse. Le serveur passe donc une grande partie de son temps à attendre sans rien faire la prochaine question du client. Le serveur concurrent permet pendant cette attente de traiter une question d'un autre client.*

2) Expliquer la solution (c) : on n'expliquera pas la mise en place du service mais simplement la façon dont `select` permet d'augmenter l'efficacité du serveur par rapport au serveur séquentiel.

Réponse: *Le serveur peut accepter plusieurs connexions en parallèle. Chaque connexion est contrôlée par une prise sur laquelle le client envoie ses requêtes et attend les réponses du serveur.* `select` *permet d'examiner les connexions prêtes à recevoir ou à émettre et donc au serveur de ne jamais attendre après un client lorsque d'autres clients sont prêts à communiquer.*

3) Quel est, sur un ordinateur d'aujourd'hui, l'ordre de grandeur le plus proche du coût minimum d'un appel système : 0.001 μs, 1μs, 1ms, 1s ?

Réponse: 1μs.

4) Un changement de contexte est le remplacement du processus en cours d'exécution par un autre. Donner deux raisons pour effectuer un changement de contexte.

Réponse:

- *Le processus en cours d'exécution a épuisé son quota de CPU.*
- *Le processus en cours d'exécution est bloqué sur une ressource manquante. En fait un pourrait ici distinguer l'attente courte d'une ressource système (cache ou périphérique) d'une attente longue, non bornée (telle qu'un timeout, des entrées-sorties sur un tuyau ou une prise,* etc.*)*

5) Dans la suite, on notera S le coût minimal d'un changement de contexte. Ce coût est de l'ordre de 20 fois celui d'un appel système simple. Expliquer cette différence.

Réponse: *Un changement de contexte ne peut se faire qu'en mode système (soit suite à un appel système, soit sur une interruption). Le coût inclut donc celui d'un appel système.*

En plus, un changement de contexte doit sauvegarder l'état du processus, notamment la table des pages, élire un nouveau processus, réinstaller l'état du processus élu. Cela constitue le coût direct. Il existe également un coût indirect lié à la rupture de flux provoquant l'invalidation des caches (notamment le changement de la table des pages).

6) On veut comparer l'efficacité respective des modèles (b) et (c) du serveur concurrent. Pour cela, on modélise le client comme une boîte noire qui lit les réponses du serveur et lui envoie de nouvelles questions. On suppose que tous les clients se comportent de façon similaire, *i.e.* posent des questions de difficultés comparables qui demandent au serveur un temps de calcul q dans le modèle du serveur séquentiel et prennent un temps c entre une

réponse et la question suivante. Typiquement ce temps est dû au temps de transmission sur le réseau entre le serveur et le client. On fera les hypothèses que $q \ll c$ et $S \ll c$.

Pour simplifier la modélisation, on supposera que les questions et réponses sont suffisamment courtes pour être lues et écrites en un seul `read` ou `write` et qu'on écrit les réponses en mode non bloquant et que l'écriture n'échoue jamais.

Enfin, on se place dans un régime stable où un nombre fixe n de clients sont (déjà) connectés. Pour n petit, le serveur attend après les clients. Pour n grand, les clients attendent après le serveur un temps supérieur au temps minimal q et on dit alors que le serveur est en *régime saturé*. On dit que le serveur est en *situation critique* lorsque la moitié des clients sont en attente après le serveur.

Avant de s'intéresser respectivement aux modèles (b) et (c), on commence par ignorer le coût de la gestion de la concurrence. ***i)*** Quelle est la valeur N de n pour laquelle le serveur passe en régime saturé ? ***ii)*** Quelle est la valeur N' de n pour laquelle le serveur passe en situation critique ?

Réponse: ***i)*** $N = c/q$. ***ii)*** $N' = 2c/q$.

7) On s'intéresse maintenant au modèle à coprocessus (b) en comptant le coût de la gestion de la concurrence. ***i)*** Quelle est la valeur N_b pour laquelle le serveur passe en régume saturé ? ***ii)*** Quelle est la valeur N'_b pour laquelle le serveur passe en situation critique ? ***iii)*** Sous quelles conditions le coût de la gestion de la concurrence est-il significatif ? ***iv)*** Ces conditions vous paraissent-elles raisonnables ?

Réponse:

Il faut changer de contexte après chaque client, donc le temps de traitement d'un client est $q + S$. D'où ***i)*** $N_b = c/(q + S)$ *et* ***ii)*** $N'_b = 2c/(q + S)$. ***iii)*** *Le coût de la concurrence est prédominant tant que $q \ll S$ et reste significatif jusqu'à $q \approx S$.* ***iv)*** *C'est une situation tout à fait raisonnable, par exemple si le traitement consiste à envoyer des informations contenues dans la mémoire du processus ou même transférer un petit fichier.*

8) Le choix de la solution (a) plutôt que (b) améliorerait-il l'efficacité du serveur ?

Réponse: *Non : le changement de contexte pour des processus est au moins aussi cher que le changement de contexte pour des coprocessus.*

9) On considère maintenant le modèle (c) avec `select`. L'appel système `select` est un appel système cher et son coût est nettement supérieur au coût minimal d'un appel système mais reste aussi inférieur à celui d'un changement de contexte. Pour simplifier on fera l'approximation par excès que sont coût est celui S d'un changement de contexte, S.

i) Dans le modèle avec `select`, quelle est la valeur N_c de n pour laquelle le serveur passe en régime saturé ? ***ii)*** En régime saturé, on note k le nombre de clients qui sont en moyenne en attente après le serveur. Quel est le temps nécessaire pour traiter ces k clients ? ***iii)*** En déduire la valeur N'_c de n lorsque le serveur passe en situation critique.

Réponse: ***i)*** *Avant le régime saturé, le serveur attend les clients, donc à chaque* `select` *il ne retourne qu'un seul client sur lequel il peut lire une question. Il faut donc compter un appel* `select` *par question par client dont le coût est (par hypothèse) celui d'un changement de contexte. D'où $N_c = N_b = c/(q + S)$.*

ii) *En régime critique,* `select` *retourne immédiatement avec k clients. Le temps pour*

traiter les k clients est donc $kq + S$.

iii) *Lorsque $k = N'_c/2$, la moitié des clients sont traités à chaque cycle. Il faut donc deux cycles pour traiter à nouveau les mêmes clients. Le temps de deux cycles est donc le temps c pris par chaque client, plus le temps c que chaque client attend. Soit $(N'_c/2)q + S = c$. D'où $N'_c = 2(c - s)/q$.*

10) ***i)*** Quel est le modèle le plus efficace, sous les hypothèses décrites ? (on regardera les valeurs de passage en régime saturé et en particulier en régime critique.) ***ii)*** Sous quelles conditions le gain est-il significatif ?

Réponse: ***i)*** *L'entrée en régime de saturation se fait pour la même valeur $N_b = N_c$ dans les deux modèles. Une fois saturé, le modèle avec* `select` *se comporte mieux, puisque*

$$N'_c/N'_b = \frac{2(c-S)}{q} \Big/ \frac{2c}{q+S} = 1 + \frac{S(c - S(q+S))}{qc} = 1 + \frac{S}{q}\left(1 - \frac{q+S}{c}\right) \approx 1 + \frac{S}{q}$$

car $q + S \ll c$ par hypothèse. ***ii)*** *Le gain est donc significatif si $q \ll S$, modéré si $q \approx S$ et mineure si $q \gg S$. En fait, dans ce dernier cas, le modèle n'est probablement plus valide.*

Donc bien que les deux modèles entre en même temps en régime saturé, le modèle (c) tient mieux la charge puisqu'il passera plus tard en régime critique.

11) Donner un exemple, pas forcément lié au modèle client serveur, où une solution similaire à (c) n'apporterait aucune amélioration par rapport à une solution séquentielle, alors qu'une solution avec coprocessus en apporterait.

Réponse: *Un examples est la sauvegarde d'un système de fichiers. Ici, la ressource critique est l'accès au(x) disque(s). Comme les entrées-sorties sur les fichiers ne sont pas bloquantes,* `select` *ne fera pas la différence entre les lectures/écritures qui sont prêtes (les caches sont en mémoire) et celles qui demandent un accès disque (avec un temps de latence). La solution avec select se comporterait alors comme une solution séquentielle. Par contre une solution avec coprocessus pourra demander en avance plusieurs accès disques en parallèle donc solliciter le disque en permanence.*

□