Université Paris Diderot
Master 1 Informatique

Programmation Système
Année 2011 – 2012

Examen

Mercredi 4 janvier 2012

Instructions

- Motivez vos réponses.
- On recommande de *bien lire* l'énoncé d'un exercice, en entier, avant de commencer à le résoudre.
- Tout document papier est autorisé.
- Les ordinateurs, les téléphones portables, et tout autre moyen de communication, sont interdits et doivent être éteints.
- La durée de l'examen est de 3 heures.

Processus

Exercice 1 a) Un processus est en train d'exécuter les lignes de code C suivantes :

```
fork();
fork();
fork();
fork();
```

Si tout les appels à `fork()` retournent une valeur différente de `-1`, combien y a-t-il de *nouveaux* processus après l'exécution du code ci-dessus ?

b) On suppose que la variable n contient un entier strictement positif et que tous les appels à `fork()` renvoient une valeur positive ou nulle :

```
for(i = 0; i < n; i++)
    if (n % 2 == 0)
        fork();
```

Combien y a-t-il de *nouveaux* processus après l'exécution de ce fragment ?

Exercice 2 a) Qu'est-ce qu'est un *processus zombie* ? Est-il problématique pour la vie d'un système d'exploitation UNIX ? Pourquoi ?

b) Écrire un programme `zombie_fun` qui crée un nombre $n \geq 0$ de processus zombies, où n est un paramètre passé sur la ligne de commande. Les zombies créés par `zombie_fun` doivent survivre le plus longtemps possible.

c) Les processus zombies créés par `zombie_fun` peuvent-ils rester zombies suite à la mort de `zombie_fun` lui-même ? Explicitez vos suppositions sur le comportement d'autres processus du système.

d) Écrire un programme `double_zombie_fun` qui crée le même nombre de processus que `zombie_fun`, mais qui utilise la technique du *double fork* pour éviter de créer des zombies.

Signaux

Exercice 3 a) En utilisant l'appel système `signal()`, écrire un programme `sigadder` qui initialise la valeur d'un compteur t (pour « total ») à 0 et attende l'arrivée des signaux suivants :
- à la réception de `SIGUSR1`, il incrémente t de 1,
- à la réception de `SIGUSR2`, il décrémente t de 1,
- quand l'utilisateur effectue un `CTRL-C` dans le terminal, il affiche la valeur courante de t à l'écran puis remet t à zéro.

Tout autre signal doit être ignoré.

b) Écrire un client `sigadd` qui prenne sur la ligne de commande le *process ID* de `sigadder` et un entier signé n et :
- si $n > 0$, augmente t de n,
- si $n < 0$, diminue t de $|n|$ (valeur absolue de n),
- si $n = 0$, affiche la valeur courante de t et le remette à 0.

c) Votre implémentation garantit-elle qu'un appel à `sigadd` *`pid n`* fasse changer la valeur de t exactement d'une quantité n ? Si oui, expliquez pourquoi. Si non, discutez les modifications à apporter à `sigadder` et `sigadd` pour garantir que cette propriété soit respectée.

Race condition

Exercice 4 a) Qu'est-ce qu'une *race condition* (ou « situation de compétition » ?)

b) Écrire un programme, le plus concis possible, dont le comportement exhibe une race condition. Expliquez de quel type de race condition il s'agit et sous quelles conditions il sera le plus probable de la voir apparaître.

Exercice 5 a) Écrire un programme `once` dont la page man serait la suivante :

```
ONCE(1)                          User Commands                          ONCE(1)

NAME
    once - ensure a single concurrent instance of a program is executed

SYNOPSIS
    once COMMAND [ ARG ... ]

DESCRIPTION
    wrapper that guarantees that COMMAND is executed concurrently only once
    for the current user.
```

Par exemple, l'exécution de `once backup.sh /home` doit garantir que d'autres exécutions de `backup.sh` préfixées par `once` ne sont pas permises avant la fin de la première invocation. Utilisez la technique que vous préférez pour garantir l'exclusion mutuelle.

b) Serait-il possible de modifier `once` pour permettre l'exécution parallèle du même programme quand il est appelé avec *des paramètres différents* ? Justifiez la réponse.

Architectures *client-server*

Exercice 6 [**Problème**] On se propose de réaliser un serveur de *streaming* de fichiers audio local à une machine UNIX. Le serveur est démarré en appelant `audio_server DIRECTORY`,

où DIRECTORY est la racine d'une partie du système de fichier contenant des sous-répertoires imbriqués, des fichiers audio de type WAV dont le nom est de la forme *.wav, et d'autres fichiers quelconques. Par exemple :

```
$ ls -R
.:
Talking Heads

./Talking Heads:
Once In a Lifetime

./Talking Heads/Once In a Lifetime:
01 - Psycho Killer.wav                          09 - Road to Nowhere.wav
02 - Take Me to the River.wav                   10 - Wild Wild Life.wav
03 - Once in a Lifetime.wav                     11 - Blind.wav
04 - Burning Down the House.wav                 12 - (Nothing But) Flowers.wav
05 - This Must Be the Place (Naive Melody).wav  13 - Sax and Violins.wav
06 - Slippery People [Live].wav                 14 - Lifetime Piling Up.wav
07 - Life During Wartime [Live].wav             talking_heads_-_the_best_of_back.jpg
08 - And She Was.wav                            talking_heads_-_the_best_of_front.jpg
```

Le serveur attend des requêtes sur un objet IPC de votre choix, dont le nom est public et connu par le clients. Un client est démarré en appelant `audio_client QUERY`, où `QUERY` est une chaîne de caractères qui apparaît dans le nom des fichiers audio que le client veux écouter (par exemple, « Life »). Une fois la requête reçue, le serveur doit trouver tous les fichiers *.wav qui contiennent la chaîne de caractères `QUERY` et les envoyer au client correspondant, l'un après l'autre.

a) Proposez une architecture et un protocole de communication pour l'implémentation de `audio_server` et `audio_client`. L'architecture proposée doit être parallèle, c'est-à-dire qu'elle doit permettre de traiter plusieurs requêtes des clients différents au même moment, et doit éviter toute *race condition* entre clients différents.

   Votre proposition doit être bien motivée et détaillée, notamment en ce qui concerne les mécanismes d'IPC choisis, les conventions entre client et serveur (par exemple le nom de l'objet IPC pour l'envoi des requêtes ; ou encore le ou les noms des objets IPC pour l'envoi des réponses), les formats de messages, et les limitations (par exexemple la taille maximale de requêtes).

b) Montrez les 10 lignes de code (maximum !) que vous considérez les plus intéressantes dans l'implémentation d'une version 1.0 de `audio_server` et `audio_client`. Motivez votre choix. Pour cette première version, les clients doivent simplement copier sur la sortie standard les fichiers reçus du serveur.

   Conseil : évitez le syndrome *not invented here* (pas inventé ici). Appuyez-vous autant que possible sur les programmes POSIX existants, dans l'esprit de la philosophie UNIX.

c) Pour la version 2.0, nous souhaitons ajouter la conversion de WAV à MP3 côté serveur. En sachant que l'exécution de « `/usr/bin/lame - -` » est un *filtre UNIX* qui convertit de WAV à MP3, décrivez les modifications à effectuer sur `audio_server` et `audio_client` entre la version 1.0 et la version 2.0.

d) Pour la version 2.1, nous souhaitons rendre *optionnelle* la conversion de WAV→MP3. Encore une fois, décrivez les modifications nécessaires. Est-il plus difficile de passer de 1.0 à 2.0 ou de passer de 2.0 à 2.1 ? Pourquoi ?