Examen partiel 2014 (1h)

RS : Réseaux et Systèmes

La notation tiendra compte de la validité des réponses, mais aussi de la présentation et de la clarté de la rédaction. Lisez entièrement le sujet avant de commencer calmement.

Documents interdits, à l'exception d'une feuille A4 recto-verso manuscrite.

★ Exercice 1: Questions de cours (6pts)

▷ **Question 1:** Quel problème fondamental rend les signaux peu robustes pour échanger des informations entre processus ? Expliquez quand ce problème peut survenir.

Réponse

Un premier problème est que c'est un moyen de communication très pauvre : on ne sait pas qui nous parle, et on ne reçoit que le type de signal, sans argument pour affiner la sémantique. Mais ceci est un bonus par rapport à la réponse attendue :

On peut très facilement perdre des signaux. Compter les messages que l'on reçoit n'est pas robuste du tout. Cela arrive s'ils sont envoyés avec des délais inter-messages trop courts. Lorsqu'un signal est en traitement (gestionnaire démarré) et qu'un autre signal est reçu, on stock ce fait dans une variable sur un seul bit (le signal est «marqué pending»). Si le signal arrive une troisième fois, l'information est perdue.

Fin réponse

▷ **Question 2:** Citez quatre mécanismes permettant d'échanger des informations entre processus.

Réponse

Signaux, tube, fichier sur disque, code de retour du binaire, socket réseau, sémaphores, shared memory, argument de la ligne de commande, ...

Fin réponse

▷ **Question 3:** Citez deux avantages et deux inconvénients des threads par rapport aux processus.

Réponse

Les threads vont plus vite à démarrer et consomment moins de mémoire. Les communications entre threads sont beaucoup plus rapides que les autres IPC. On peut les utiliser pour faire du recouvrement calcul/communications ou calcul/interface utilisateur.

Les threads sont dangereux car cela revient à retirer toutes les protections du systèmes qui nous simplifient la vie, dans le seul objectif de gagner en vitesse. On a alors des conditions de compétition, des interblocages et des famines. Il faut que toutes les bibliothèques utilisées soient thread-safe (ce qui est difficile à assurer), et les applications deviennent très difficiles à débugger (les bugs ne sont plus reproductibles).

Fin réponse

▷ **Question 4:** Citez un avantage et un inconvénient des bibliothèques dynamiques par rapport aux bibliothèques statiques.

Réponse

Slide 126 du cours.

Fin réponse

▷ **Question 5:** Citez un avantage et un inconvénient de l'interprétation par rapport à la compilation.

Réponse

Slide 106 du cours.

Fin réponse

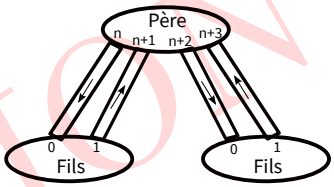
▷ **Question 6:** Définissez succinctement trois techniques classiques de protection des ressources.

Réponse

Slide 24 du cours. Préemption, interposition et mode privilégié.

Fin réponse

★ Exercice 2: Utiliser `fork`, `exec`, `wait`, `waitpid`, `pipe`, `dup`, `dup2` (6pts)



On souhaite écrire un programme réalisant un tri fusion d'entiers grâce à un arbre binaire de processus échangeant des informations au travers de tubes. À chaque étage de l'arbre, les processus seront interconnectés de la façon représentée sur le schéma ci-contre.

Pour la lisibilité du code, on supposera qu'aucun appel système n'échoue, et vous êtes donc dispensés de tester les codes de retour. Bien entendu, votre code ne doit pas déclarer de tableau destinés à stocker la liste à trier. On conserve au maximum deux entiers lus à la fois dans un processus donné. Vous pouvez supposer que l'on trie 2^n entiers afin de simplifier la logique de votre code et vous concentrer sur les aspects système.

▷ **Question 1:** Écrivez tout d'abord le programme des feuilles de l'arbre, qui lit deux entiers sur son entrée standard, et les écrit dans l'ordre croissant sur sa sortie standard.

On veut maintenant écrire le programme d'un nœud de l'arbre. Il prend le nombre d'entiers à trier en argument de la ligne de commande, et les entiers sont écrits sur son entrée standard.

▷ **Question 2:** Écrivez une fonction `creefils`, chargée de créer les tubes nécessaires (selon le schéma ci-dessus) et de démarrer deux processus enfants pouvant être eux-mêmes des nœuds ou des feuilles, en fonction du nombre d'entiers qu'ils devront gérer.

▷ **Question 3:** Écrivez une fonction `distribue`, qui lit tous les entiers disponibles sur l'entrée standard et les écrit alternativement à chacun de ses enfants.

▷ **Question 4:** Écrivez une fonction `fusion`, qui lit les sorties des enfants et les écrit sur la sortie standard dans l'ordre croissant (en ne stockant au plus que deux entiers à la fois).

▷ **Question 5:** Concluez en écrivant la fonction principale de ce programme.

Extraits de quelques pages de manuel

```
pid_t fork(void);
int execlp(const char *file, const char *arg, ...);
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
WIFEXITED(status) -- returns true if the child terminated normally
WEXITSTATUS(status) -- returns the exit status of the child
int pipe(int pipefd[2]);
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

★ Exercice 3: Édition de liens et symboles multiples (4pts).

Pour chacun des cas ci-après, indiquez si la compilation et l'édition de lien des modules sont possibles. Si c'est impossible, indiquez pourquoi. Si les deux sont possibles, indiquez le résultat du code (affichage, valeurs de retour, etc). On ne demande pas la valeur numérique des pointeurs, mais ce vers quoi ils pointent. Dans tous les cas, argumentez vos affirmations : si vous étiez un compilateur ou un éditeur de liens, quel message d'erreur afficheriez-vous dans les différents cas ?

▷ Question 1:

fichier_1.c
```
1 int proc(void) {
2   return 42;
3 }
```

fichier_2.c
```
1 int proc(void) {
2   return 24;
3 }
```

fichier_3.c
```
1 int main(int argc, char**argv) {
2   printf ("%d", proc());
3 }
```

Réponse

proc est un symbole fort dans les deux premiers modules =¿ erreur à l'édition de lien.
Exemple de message d'erreur : "multiples définitions du symbole proc"

Fin réponse

▷ Question 2:

fichier_1.c
```
1 int proc(void) {
2   return 42;
3 }
```

fichier_2.c
```
1 int proc(void);
2 int main(void) {
3   printf ("%d", proc());
4 }
```

Réponse

Le symbole proc est fort à gauche (on lui donne une valeur, un contenu) et faible à droite (on y fait référence, on l'utilise, sans lui donner de valeur). Il n'y a donc pas de problème.

Le programme affiche 42 sans aucun message d'erreur.

Fin réponse

▷ Question 3:

fichier_1.c
```
1 int x;
2 int y;
3 int suite_carre(int i) {
4    x += i;
5    y += i * i;
6    return y;
7 }
```

fichier_2.c
```
1 double x ;
2 double inv(int i) {
3   x = 1.0 / (double) i;
4   return x;
5 }
```

fichier_3.c
```
1 int x = 1;
2 int y = 2;
3 int main(int argc, char**argv) {
4   printf ("%f", inv(2));
5   printf ("%d", suite_carre(2));
6 }
```

Réponse

Le symbole x est faible dans fichier1, faible dans fichier2 et fort dans fichier3. La compilation et l'édition de liens vont donc se passer sans message d'erreur (si on fait abstraction de l'absence de #include qui est une imprécision de l'énoncé).

À l'exécution en revanche, on va avoir des résultats très bizarres, puisque le x de fichier2 est plus large que ceux des autres fichiers. C'est x@fichier3 qui donne la taille utilisée dans le binaire, puisque c'est lui le symbole fort. Comme x@fichier2 est plus large, on change d'autres variables en mémoire quand on le modifie (en particulier y@fichier3 qui se trouve juste après).

Ca c'était la réponse attendue. En pratique les éditeurs de liens modernes nous préviennent du problème à venir. C'est heureux, car l'exécution n'est pas du tout ce qu'on attendait.

```
1 $ gcc fich1.c fich2.c fich3.c -o Q3
2 /usr/bin/ld: Attention: alignement 4 du symbole «x» dans /tmp/cck1trVq.o est plus petit que 8 dans /tmp/ccWa3X82.o
3 /usr/bin/ld: Attention: taille du symbole «x» a changé de 8 dans /tmp/ccWa3X82.o à 4 dans /tmp/cck1trVq.o
4 $ ./Q3
5 0.000000
6 1071644676
```

Fin réponse

▷ Question 4:

fichier_1.c
```
1 void p2(void);
2 int main() {
3   p2();
4   return 0;
5 }
```

fichier_2.c
```
1 #include <stdio.h>
2 char main;
3 void p2() {
4   printf("0x%x\n", main);
5 }
```

Réponse

Le symbole `main` est fort à gauche et faible à droite, tandis que `p2` est faible à gauche et fort à droite. Pas d'erreur à la compilation ou à l'édition de liens à craindre.

À l'exécution, le module de droite pense que `main` est un char (qui, en C, est un type numérique) et affiche le contenu de cette variable. Il va donc aller lire le premier octet à l'adresse de main. Mais la définition forte de main est celle du module de gauche, et il s'agit en fait d'une fonction, pas d'un `char`. Du coup, quand le module de gauche lit le premier octet à l'adresse de main, il lit le premier octet du code de la fonction main quand on l'écrit en hexadécimal.

Au final, cela affiche 0x55 chez moi (mais la valeur numérique me semble impossible à prédire).

Fin réponse

★ Exercice 4: Programmation concurrente, d'après Luigi Santocanale (4pts)

▷ **Question 1:** Rappelons que l'appel système `int mkdir(const char *path , mode_t mode);` retourne un code d'erreur si le répertoire à créer existe déjà. Expliquez, par conséquence, comment par l'implantation suivante des fonctions `entrer_region` et `sortir_region` peut protéger la section critique d'un processus et assurer l'exclusion mutuelle :

```
1 void entrer_region(char *verrou) {
2   while(mkdir(verrou ,0777) < 0) {
3     sleep(1);
4   }
5 }
```

```
1 void sortir_region(char *verrou) {
2   unlink(verrou);
3 }
```

▷ **Question 2:** Dans le code suivant `verrou` est l'adresse d'une variable entière que l'on suppose être partagée entre plusieurs processus. On peut alors protéger la section critique d'un processus par

```
1 void entrer_region(int *verrou) {
2   while (*verrou == 1) {
3     sleep(1);
4   }
5   *verrou = 1;
6 }
```

```
1 void sortir_region(int *verrou) {
2   *verrou = 0;
3 }
```

Expliquez en quoi cette implantation est pire que la précédente.

Réponse

Fin réponse