Examen du 11 octobre 2017 (1h)

RS : Réseaux et Systèmes

Deuxième année

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 à rendre avec votre copie.

★ Exercice 1: Question de cours (3 pts)

▷ **Question 1:** Qu'est ce que l'attente active ? (Quelle est l'alternative ?)

▷ **Question 2:** Qu'est ce qu'un i-node (ou i-nœud) ?

▷ **Question 3:** Quel est l'intérêt du mécanisme des appels systèmes dans un système d'exploitation ? Pourquoi ne pas utiliser de simples appels de fonctions normaux à la place ?

★ Exercice 2: Clonage de processus (4 pts)

▷ **Question 1:** Dessiner (comme lors du TD 1) l'exécution du Programme 1 ci-dessous. Combien de "hello !" affiche-t-il ?

▷ **Question 2:** Dessiner (comme lors du TD 1) l'exécution du Programme 2 ci-dessous (y compris ses affichages – pour les PID, vous pouvez supposer que le premier processus a le PID 1000, par exemple).

★ Exercice 3: Savoir utiliser les threads POSIX (3 pts)

Lors du TP3, un étudiant a écrit le Programme 3 ci-dessous.

▷ **Question 1:** Que pouvez-vous dire des différents affichages de `getpid()` ? Pourquoi ?

▷ **Question 2:** L'étudiant a fait deux erreurs grossières. Quelles sont-elles ? (Il n'est pas demandé d'écrire un programme correct, mais uniquement d'expliquer précisément quelles sont les deux erreurs)

Programme 1

```
1  void doit() {
2    fork();
3    fork();
4    printf("hello!\n");
5  }
6  int main() {
7    doit();
8    printf("hello!\n");
9    exit(0);
10 }
```

Programme 2

```
1  int main() {
2    int n = 3;
3    int i;
4    printf("proc %d fils de %d\n",
5           getpid(),getppid());
6    for (i=0; i<n; i++) {
7      if (fork() == 0) {
8        printf("proc %d fils de %d\n",
9               getpid(),getppid());
10     } else {
11       wait(NULL);
12       exit(0);
13     }
14   }
15 }
```

Programme 3

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <sys/types.h>
5  #include <unistd.h>
6
7  #define NBTH 10
8
9  int somme = 0;
10
11 void * ajouteur(void * arg) {
12   int numthread = * (int*) arg;
13   int i;
14   printf("ajouteur: getpid()=%d\n",
15          getpid());
16   for (i = 0; i < 1000000; i++) {
17     somme += numthread;
18   }
19   return NULL;
20 }
21
22 int main(int argc, char** argv) {
23   int i;
24   pthread_t threads[NBTH];
25   printf("main: getpid()=%d\n",
26          getpid());
27   for (i = 0; i < NBTH; i++) {
28     pthread_create(
29        &threads[i], NULL,
30              ajouteur, &i);
31   }
32   for (i = 0; i < NBTH; i++) {
33     pthread_join(threads[i], NULL);
34   }
35   printf("somme=%d\n", somme);
36   return 0;
37 }
```

★ Exercice 4: Utiliser `fork`, `exec`, `wait`, `waitpid`, `pipe`, `dup`, `dup2`, etc. (4 pts)

Donner le pseudo-code C (sans utiliser la fonction `system`) correspondant à ce que fait un *shell* lorsqu'on tape les lignes de commandes suivantes. Afin d'obtenir un code lisible et court, vous êtes dispensés des tests d'erreur des appels systèmes.

▷ **Question 1:** `prog1 < fichier1.txt`
(Il faut donc : lancer le programme *prog1* après avoir redirigé son entrée standard pour qu'il reçoive le contenu du fichier au lieu de ce qui est écrit au clavier)

▷ **Question 2:** `prog1 | prog2`
(Il faut donc : lancer deux programmes *prog1* et *prog2*, en liant la sortie standard de *prog1* à l'entrée de *prog2*)

Quelques fonctions et macros peut-être utiles

```
pid_t fork(void);
int execlp(const char *file, const char *arg, ...);
   exemple: execlp ("prog1", "prog1", "arg1", "arg2, NULL);
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);
int open(const char *pathname, int flags); // flags = O_RDONLY, O_WRONLY, etc.
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int close(int fd);
```

★ Exercice 5: Savoir reconnaître les schémas de synchronisation classiques et utiliser les sémaphores (6 pts) (inspiré d'un exercice de Cédric Bastoul)

Un étudiant qui se spécialise en anthropologie et accessoirement en informatique s'est embarqué dans un projet de recherche pour voir s'il était possible d'enseigner les interblocages aux babouins d'Afrique. Il repère un profond canyon et y jette une corde au travers, de sorte qu'un babouin puisse le traverser à bouts de bras.

Chaque babouin répète l'algorithme suivant :

- RÉPÉTER à l'infini
  - Se présenter devant la corde
  - Attraper la corde
  - Traverser le canyon
  - Continuer à pieds
- FIN RÉPÉTER

L'installation ayant du succès, les babouins se rendent vite compte que si deux d'entre eux commencent à traverser dans les deux sens opposés, ils se retrouvent bloqués au milieu du canyon.

▷ **Question 1:** Proposez une solution (modifiez l'algorithme ci-dessus) à base de sémaphore garantissant que seul un babouin à la fois pourra accéder à la corde et traverser.

▷ **Question 2:** Votre solution se rapproche d'un schéma de synchronisation classique. Lequel ?

Cette solution est loin d'être idéale. Les babouins se rendent rapidement compte que plusieurs babouins pourraient traverser simultanément, à condition qu'ils traversent tous dans le même sens, ce qui serait bien plus efficace.

▷ **Question 3:** Modifiez l'algorithme ci-dessus. Il est conseillé d'écrire deux algorithmes : celui pour les babouins qui traversent du Nord vers le Sud, et celui pour ceux qui font l'inverse.

▷ **Question 4:** Votre solution se rapproche d'un schéma de synchronisation classique. Lequel ?

▷ **Question 5:** Que pouvez-vous dire d'un problème de famine éventuel avec votre solution ? (Il n'est pas demandé de proposer obligatoirement une solution sans famine)