Programmation système

Steve Alabi - Taha Bendjeddou - Younes Benyamna - Malek Zemni

Correction des exercices

22/11/2018

Exercice 1

1/ Un appel-système est une fonction permettant d'accéder aux informations du système et de communiquer avec son noyau. Ce type de fonctions, depuis l'espace utilisateur, demande des services ou des ressources au système d'exploitation. Chaque architecture matérielle ne supporte que sa propre liste d'appels-systèmes, c'est pourquoi les appels-systèmes diffèrent d'une machine à l'autre. Mais la plupart d'entre eux sont implémentés sur toutes les machines. Cependant, les programmes système restent très peu portables.

Exécution : pour des raisons de sécurité évidentes, les applications de l'espace utilisateur ne peuvent pas directement exécuter le code du noyau ou manipuler ses données. Par conséquent, un mécanisme de signaux a été mis en place. Quand une application exécute un appel-système, elle peut alors effectuer un ***trap*** (permutation de l'exécution de l'application en mode noyau), et peut exécuter le code, du moment que le noyau le lui autorise.

2/ Le flux d'erreurs standard `stderr` se modélise par un fichier dans lequel on peut écrire nos erreurs. Grâce à cette écriture séparée, l'utilisateur pourra rediriger le flux standard afin d'isoler les erreurs et ainsi mieux les traiter, en nombre moins importants, devant la masse d'informations contenues en sortie.

3/

```
int myGetChar (int fd)
{
   int retour;
   unsigned char c;

   if ( read(fd, &c, 1) == 1 )
      retour = c;
   else
      retour = EOF;

   return retour;
}
```

4.1/ Un buffer, mémoire temporaire ou mémoire tampon, est une zone de mémoire virtuelle utilisée pour stocker temporairement les données, notamment entre 2 processus.

4.2/ Dans cette version bufferisée, on n'effectue plus l'appel-système `read` à chaque fois qu'on veut lire un caractère. On va lire un nombre MAX de caractères à la fois avec un seul `read` puis on stocke ces caractères lus dans un buffer. Ensuite, c'est à partir de ce buffer que l'on va lire le caractère. Finalement, si le buffer se vide, on refait une lecture.

4.3/

```
int myGetCharBuffer (int fd)
{
   static unsigned char buf[MAX];
   static int nbCar = 0;
   static char* p;

   int retour;

   if ( nbCarac == 0 ) {
      nbCarac = read(fd, buf, MAX);
      p = buf;
   }

   if ( nbCarac > 0)
      retour = *p;
   else
      retour = EOF;

   nbCarac--;
   p++;

   return retour;
}
```

Exercice 2

1/ Sous UNIX, les processus jouent un rôle très important. Le concept de processus a été mis au point dès les débuts de ce système : il a ainsi participé à sa gloire et à sa célébrité. Une des particularités de la gestion des processus sous UNIX consiste à séparer la création d'un processus et l'exécution d'une image binaire. Bien que la plupart du temps ces deux tâches sont exécutées ensemble, cette division a permis de nouvelles libertés quant à la gestion des tâches. Par exemple, cela permet d'avoir plusieurs processus pour un même programme.
Autrement dit, sous les autres systèmes d'exploitation (mis à part quelques exceptions), un processus est l'équivalent d'un nouveau programme, alors que sous UNIX ce n'est pas forcément le cas.
Ce principe, peu utilisé dans les autres systèmes, a survécu de nos jours. Alors que la plupart des

systèmes d'exploitation offrent un seul appel-système pour exécuter un nouveau programme, UNIX en possède deux : `fork` et `exec`.

2/

- 1 : élection, l'ordonnanceur choisit ce processus
- 2 : blocage, processus bloqué en attente d'une donnée
- 3 : déblocage, donnée devient disponible
- 4 : l'ordonnanceur choisit un autre processus

3/ Un processus **zombie** est un processus qui s'est achevé, mais qui dispose toujours d'un identifiant de processus (PID) et reste donc encore visible dans la table des processus. Lorsque le processus fils se termine avant le processus père (c'est à dire que le père n'a pas encore lu le code de retour du fils), le processus fils devient un zombie.

Pour permettre à un processus fils zombie de disparaître complètement, on utilise la fonction `wait()`. Lorsque l'on appelle cette fonction, celle-ci bloque le processus à partir duquel elle a été appelée (père) jusqu'à ce qu'un de ses fils se termine. Elle renvoie alors le PID de ce dernier. En cas d'erreur, la fonction renvoie la valeur -1.

4/ Les variables sont dupliquées. Les variables d'un processus père et ceux d'un processus fils sont complètement distinctes.
Les descripteurs de fichiers sont partagés entre les processus parents et les processus fils.

5/

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>

pid_t create_process(void)
{
   pid_t pid;

   do {
      pid = fork();
   } while ((pid == -1) && (errno == EAGAIN));

   return pid;
}

void child_process(void)
{
   printf(" Nous sommes dans le fils !\n"
```

```
22        " Le PID du fils est %d.\n"
23        " Le PPID du fils est %d.\n", (int) getpid(), (int) getppid());
24  }
25
26  void father_process(int child_pid)
27  {
28     printf(" Nous sommes dans le père !\n"
29        " Le PID du fils est %d.\n"
30        " Le PID du père est %d.\n", (int) child_pid, (int) getpid());
31
32     if (wait(NULL) == -1) {
33        perror("wait :");
34        exit(EXIT_FAILURE);
35     }
36  }
37
38  int main(void)
39  {
40     pid_t pid = create_process();
41
42     switch (pid) {
43        case -1:
44           perror("fork");
45           return EXIT_FAILURE;
46           break;
47        case 0:
48           child_process();
49           break;
50        default:
51        father_process(pid);
52        break;
53     }
54
55     return EXIT_SUCCESS;
56  }
```

Exercice 3

1/ Un **thread** ou processus léger représente, comme un processus, l'exécution d'un ensemble d'instructions du langage machine d'un processeur. Toutefois, là où chaque processus possède sa propre mémoire virtuelle, les threads d'un même processus se partagent sa mémoire virtuelle. Par contre, tous les threads possèdent leur propre pile d'exécution.

Le principal avantage des threads par rapport aux processus, c'est la facilité et la rapidité de leur création. En effet, tous les threads d'un même processus partagent le même espace d'adressage, et donc toutes les variables. Cela évite donc l'allocation de tous ces espaces lors de la création, et il est à noter que, sur de nombreux systèmes, la création d'un thread est environ cent fois plus rapide que celle d'un processus.

Au-delà de la création, la superposition de l'exécution des activités dans une même application permet une importante accélération quant au fonctionnement de cette dernière.

2/

```c
void* helloWorld (void* arg) {
   printf("Hello world ! pid=%d\n", getpid());
   return NULL;
}

int main(int argc, char* argv)
{
   int i;
   int nb;
   pthread_t* threads;

   nb = atoi(argv[1]);

   threads = malloc(nb * sizeof(pthread_t));

   for (i=0 ; i<nb ; i++)
      pthread_create(&threads[i], NULL, helloWorld, NULL);

   printf("Attente \n");
   for (i=0 ; i<nb ; i++)
      pthread_join(threads[i], NULL);

   return 0;
}
```

3/ Un mutex est mécanisme de synchronisation, permettant l'exclusion mutuelle des thread. Avec les threads, toutes les variables sont partagées (mémoire partagée). Cela pose des problèmes quand deux threads cherchent à modifier deux variables en même temps. Dans ce cas, on utilise un mutex.
Le mutex va servir de verrou, pour permettre de protéger des données. Ce verrou peut donc prendre deux états : disponible et verrouillé.
Quand un thread a accès à une variable protégée par un mutex, on dit qu'il tient le mutex. Il ne peut y avoir qu'un seul thread qui tient le mutex en même temps.

4/

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_cond_t condition = PTHREAD_COND_INITIALIZER; /* Création de la
    condition */
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; /* Création du
    mutex */
```

```c

void* threadAlarme (void* arg);
void* threadCompteur (void* arg);

int main (void)
{
   pthread_t monThreadCompteur;
   pthread_t monThreadAlarme;

   pthread_create (&monThreadCompteur, NULL, threadCompteur,
    (void*)NULL);
   pthread_create (&monThreadAlarme, NULL, threadAlarme, (void*)NULL);
    /* Création des threads */

   pthread_join (monThreadCompteur, NULL);
   pthread_join (monThreadAlarme, NULL); /* Attente de la fin des
    threads */

   return 0;
}

void* threadCompteur (void* arg)
{
   int compteur = 0, nombre = 0;

   srand(time(NULL));

   while(1) /* Boucle infinie */
   {
      nombre = rand()%10; /* On tire un nombre entre 0 et 10 */
      compteur += nombre; /* On ajoute ce nombre à la variable
    compteur */

      printf("\n%d", compteur);

      if(compteur >= 20) /* Si compteur est plus grand ou égal à 20 */
      {
         pthread_mutex_lock (&mutex); /* On verrouille le mutex */
         pthread_cond_signal (&condition); /* On délivre le signal :
    condition remplie */
         pthread_mutex_unlock (&mutex); /* On déverrouille le mutex */

         compteur = 0; /* On remet la variable compteur à 0 */
      }

      sleep (1); /* On laisse 1 seconde de repos */
   }

   pthread_exit(NULL); /* Fin du thread */
}
```

```
52
53 void* threadAlarme (void* arg)
54 {
55    while(1) /* Boucle infinie */
56    {
57       pthread_mutex_lock(&mutex); /* On verrouille le mutex */
58       pthread_cond_wait (&condition, &mutex); /* On attend que la
    condition soit remplie */
59       printf("\nLE COMPTEUR A DÉPASSÉ 20.");
60       pthread_mutex_unlock(&mutex); /* On déverrouille le mutex */
61    }
62
63    pthread_exit(NULL); /* Fin du thread */
64 }
```