Programmation système II

Correction du « partiel blanch » - Timothée Ravier

1e année cycle ingénieur STI, juin 2014

1h20 – Documents non autorisés

1 Threads et synchronisation (7 points)

- **Réponse 1.1 (2 points) :** Les threads sont des processus légers. Par opposition aux processus qui ne partagent aucun élément entre eux, les threads d'un même processus partagent certaines informations : le tas, les descripteurs de fichiers ouverts... En revanche, comme pour les processus, certains éléments sont spécifiques à chaque threads : la pile, les identifiants (PID).
- **Réponse 1.2 (1 point) :** Dans un programme multi-threadé, une section critique est une portion de code qui ne doit pas être exécutée par plus d'un thread à la fois ;
- **Réponse 1.3 (1 point) :** Tous deux permettent de protéger une section critique. Le mutex n'autorise qu'un seul thread à prendre le mutex simultanément alors que le sémaphore dispose d'un compteur permettant d'autoriser l'accès à plusieurs threads. Le mutex peut être vu comme un sémaphore avec pour compteur 1.
- **Réponse 1.4 (1 points) :** Ce programme crée deux threads qui essaient de prendre des parts de gâteau à intervalles réguliers, et ce uniquement si il en reste. Lors de l'exécution, on constate qu'un thread a pris une part de gâteau alors qu'il n'en restait en fait plus (d'où le -1). Cette situation s'est produite parce que les deux threads essaient d'accéder et de modifier une variable partagée sans s'assurer qu'ils sont les seuls en train de la modifier.
- **Réponse 1.5 (2 points) :** Pour corriger ce programme, il faut créer une section critique qui englobera tous les accès à la variable `parts_de_gateau`. Deux exemples sont donnés, le premier sans modification du code existant, le deuxième avec une modification plus importante. Seules les parties modifiées du code sont montrées ici.

Solution 1 :

```
1  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
2
3  void *prendre_part(void * num)
4  {
5      int numero = *((int *)num);
6
7      pthread_mutex_lock(&mutex);
8      while (parts_de_gateau > 0) {
9          printf("Thread %d : il y a %d parts de gateau.\n", numero, parts_de_gateau);
10         printf("Thread %d : mange une part de gateau.\n", numero);
11         --parts_de_gateau;
12         printf("Thread %d : il reste %d parts de gateau.\n", numero, parts_de_gateau);
13         pthread_mutex_unlock(&mutex);
14         usleep(1);
15         pthread_mutex_lock(&mutex);
16     }
17     pthread_mutex_unlock(&mutex);
18
```

```
19      pthread_exit(NULL);
20  }
```

Solution 2 :

```
1   pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
2
3   void *prendre_part(void * num)
4   {
5       int numero = *((int *)num);
6
7       while (1) {
8           pthread_mutex_lock(&mutex);
9           if(parts_de_gateau > 0) {
10              printf("Thread %d : il y a %d parts de gateau.\n", numero, parts_de_gateau);
11              printf("Thread %d : mange une part de gateau.\n", numero);
12              --parts_de_gateau;
13              printf("Thread %d : il reste %d parts de gateau.\n", numero, parts_de_gateau);
14              pthread_mutex_unlock(&mutex);
15              usleep(1);
16          } else {
17              pthread_mutex_unlock(&mutex);
18              break;
19          }
20      }
21
22      pthread_exit(NULL);
23  }
```

2 IPC (3 points)

- **Réponse 2.1 (1 points) :** Les quatres mécanismes de communication inter processus les plus utilisés sont : les tubes (pipes), les sockets (réseau et UNIX), la mémoire partagée et les signaux.
- **Réponse 2.2 (2 points) :** Ce programme est la correction de l'exercice 2 du TD sur les IPCs. Le programme 2 crée une file de message posix (POSIX MQ) et envoie les données qu'il lit depuis son entrée standard dans cette file, avec une priorité aléatoire (entre 0 et 10). Si il reçoit le caractère EOF, alors il envoie un message de priorité la plus basse (0) et vide. Le programme 1 ouvre la file de message créée par le programme 2 et lit les messages dans l'ordre de priorité d'arrivée et affiche la priorité et le contenu du message sur sa sortie standard. Il se termine lorsqu'il reçoit un message vide.

3 Signaux (5 points)

- **Réponse 3.1 (1 point) :** Un signal est une notification asynchrone envoyée à un processus pour lui signaler l'apparition d'un événement.
- **Réponse 3.2 (2 points) :** SIGKILL, SIGSTOP, SIGCHLD, SIGINT, SIGTERM... Voir les définitions dans le cours (une version fixée du cours incluant la définition de SIGINT est en ligne).
- **Réponse 3.3 (2 points) :** Ce programme met en place un gestionnaire de signaux pour les signaux SIGUSR1 et SIGUSR2. Il ignore aussi les signaux SIGINT (Control-C). Lorsqu'il reçoit le signal SIGUSR1, il décrémente la variable temperature. Lorsqu'il reçoit le signal SIGUSR2, il l'incrémente. Il affiche toutes les secondes la valeur actuelle de la variable temperature.

Sortie du programme :

```
Mon PID est : 7153
Temprature courante : 20 (Pas encore reçu de singaux)
Temprature courante : 19 (Premier signal reçu)
Temprature courante : 20
Temprature courante : 20 (SIGINT ignoré)
Temprature courante : 21
Temprature courante : 20
Temprature courante : 20 (SIGINT ignoré)
Temprature courante : 21
Temprature courante : 20 (SIGINT ignoré)
(Fin du programme, le reste des signaux est ignoré)
```

4 Socket UNIX (3 points)

- **Réponse 4.1 (1 point) :** Les sockets réseau ont une adresse sur le réseau (souvent une adresse IP), alors que les socket UNIX sont adressées à l'aide d'un chemin dans le système de fichier (/tmp/ma_socket). Les communications sur une socket UNIX sont internes au système alors que les communications sur une socket réseau sont généralement effectuées avec d'autres systèmes (l'interface lo, ou loopback, est un cas particulier).
- **Réponse 4.2 (2 point) :** Pseudo code ci dessous :

Pseudo code du client :

```
1  char *nom_socket = "/tmp/super_socket";
2  struct sockaddr_un adresse_local;
3
4  socket_unix = socket(AF_UNIX, SOCK_STREAM);
5  adresse_locale.sun_family = AF_UNIX;
6  strcpy(adresse_locale.sun_path, nom_socket);
7  connect(socket_unix, &adresse_local);
8  ...
9  close(socket_unix);
```

Pseudo code du serveur :

```
1  char *nom_socket = "/tmp/super_socket";
2  struct sockaddr_un adresse_locale;
3
4  socket_unix = socket(AF_UNIX, SOCK_STREAM);
5  adresse_locale.sun_family = AF_UNIX;
6  strcpy(adresse_locale.sun_path, nom_socket);
7  bind(socket_unix, &adresse_locale);
8  listen(socket_unix);
9  connection = accept(socket_unix, &adresse_locale);
10 ...
11 close(connection);
12 close(socket_unix);
13 unlink(socket_name);
```

5 Gestion des entrées sorties (2 points)

- **Réponse 5.1 (1 point) :** Cette attribut indique au noyau que les appels systèmes interagissant avec ce descripteur de fichier ne doivent pas bloquer l'exécution du programme. Par exemple, l'appel système read ne doit pas bloquer si il n'y a pas de données à lire dans un tube.
- **Réponse 5.2 (1 point) :** Ces fonctions servent à gérer de façon asynchrone les entrées / sorties. Cela signifie qu'un seul processus ou un seul thread peut gérer les évènement se produisant sur un ensemble de descripteurs de fichiers (fichier, socket, IPC). Les fonctions select et poll sont des standards POSIX, la fonction epoll est spécifique à Linux.