R. Namyst
LaBRI, bureau 322

le 28 avril 2004

Licence informatique — Programmation Système
Corrigé du devoir surveillé

Remarque préliminaire : *Certaines réponses proposées aux questions de cours sont volontairement "verbeuses", et ce dans un but pédagogique. Bien sûr, il n'était pas nécessaire de fournir autant de détails pour répondre correctement aux questions (et donc obtenir le maximum de points) ...*

1 Mode utilisateur, mode noyau

Durant l'exécution d'un processus en mode utilisateur, le processeur fonctionne dans un mode "protégé" (encore appelé mode *utilisateur*, c'est-à-dire que certaines instructions sont inaccessibles, que certaines zones mémoire sont inaccessibles et d'une manière générale que l'accès direct au matériel n'est pas possible. Pour effectuer ces opérations "privilégiées", le processeur doit fonctionner en mode "réel" (encore appelé mode *noyau*). Lors du démarrage d'une machine, le processeur fonctionne en mode noyau. Il basculera en mode utilisateur lorsqu'il exécutera la première instruction du premier processus du système...

La transition du mode utilisateur vers le mode noyau s'effectue à chaque fois qu'une interruption est levée ou qu'un appel système est effectué (qui peut être assimilé au déclenchement explicite d'une interruption). L'exécution est alors redirigée automatiquement vers une routine du noyau (que le processus ne peut pas modifier).

La transition dans l'autre sens s'effectue lors du retour d'une interruption ou d'un appel système.

Si le processeur n'offrait pas un moyen d'exécuter du code binaire en empêchant l'exécution de certaines instructions, alors les processus pourraient facilement accéder directement au matériel en court-circuitant l'arbitrage du système d'exploitation. Il serait alors impossible de garantir le bon fonctionnement du clavier ou de l'écran et, pire, de garantir le respect des protections instaurées par les droits Unix sur les fichiers par exemple...

2 Gestion des fichiers

Question 1 Le programme `lire` affichera :

```
bon
jou
r
```

Question 2 L'exécution simultanée de deux processus exécutant le programme `ecrire` aura le même effet qu'une seule exécution : chaque processus disposera d'une position courante dans le fichier indépendante, donc quelle que soit sa vitesse de progression, il écrira de toute façon la même lettre à la même position que l'autre processus. Le contenu du fichier sera donc la chaîne `bonjour`.

Question 3 Pour obtenir le code de `ecrire_impair` à partir de `ecrire`, il suffit de modifier le corps de la boucle `for` de cette façon :

```
for(i=0; i<strlen(message); i++)
  if(i % 2 == 0) {
    lseek(fd, i, SEEK_SET);
    write(fd, &message[i], 1);
  }
```

Question 4 Le résultat de l'exécution de `ecrire_impair` suivi de celle de `ecrire_pair` sera le même que celui de l'exécution de `ecrire_pair` suivi de `ecrire_impair`. Les deux programmes écrivent à des positions différentes, donc n'écrasent les caractères écrits par leur homologue.

3 Communication par tubes

Question 1 Le programme crée `nb-1` processus en chaîne. Á chaque itération, le père sort de la boucle. Le premier processus (lancé depuis le shell) sort donc de la boucle au premier tour et sa variable `i` vaut 0. Le second (son premier fils) est le prochain à sortir de la boucle et sa variable `i` vaut 1. Etc. En tout, `nb-1` sont ainsi créés par le `fork`, ce qui donne un total de `nb` en comptant le processus de départ.

L'affichage produira donc les lignes `Processus 0`, `Processus 1` et `Processus 2` dans un ordre non déterministe.

Question 2 Voici la nouvelle version du programme `lanceur` :

```
int main(int argc, char *argv[])
{
  int i = 0;
  int tube[2];
  int in = STDIN_FILENO;
  int out = STDOUT_FILENO;
  int nb = argc-1;

  do {
    pipe(tube);
    if(fork() != 0) {
      close(tube[1]); in = tube[0];
    } else {
      close(tube[0]); out = tube[1];
      break;
    }
    i++;
  } while(i < nb-1);

  if(in != STDIN_FILENO) {
    dup2(in, STDIN_FILENO); close(in);
  }
  if(out != STDOUT_FILENO) {
    dup2(out, STDOUT_FILENO); close(out);
  }
  execlp(argv[i+1], argv[i+1], NULL);
}
```

Il fallait donc simplement modifier l'initialisation de la variable `nb` et ajouter les quelques lignes juste après la boucle.

Notez (mais je rappelle que cela n'était pas demandé) que pour que les processus se terminent dans le bon ordre, il fallait inverser le test sur le résultat du `fork` comme cela a été fait dans le programme ci-dessus (`fork() != 0` au lieu de `fork() == 0`). Ainsi le "père" termine son exécution en dernier, son premier fils en avant-dernier, etc.

4 Tableaux dynamiques

```
typedef int ***tab3D;

tab3D
tab3D_creer(size_t m, size_t l, size_t c)
{
  tab3D t;
  size_t i, j;

  t = (tab3D) malloc (m * sizeof (int **));
  t[0] = (int **) malloc (m * l * sizeof (int *));
  t[0][0] = (int *) malloc (m * l * c * sizeof (int));

  for (i = 0; i < m; i++)
    {
      t[i] = t[0] + i * l;
      for (j = 0; j < l; j++)
        t[i][j] = t[0][0] + i * l * c + j * c;
    }
  return t;
}

void
tab3D_detruire(tab3D t)
{
  free(t[0][0]);
  free(t[0]);
  free(t);
}
```