Mercredi 16 novembre 2005
Durée : 2 heures
Documents autorisés

Partiel : systèmes d'exploitation
L3, parcours informatique, 2005-2006

Corrigé partiel.

Examen partiel

Ordonnancement

Exercice 1. Rappelons que, dans la table d'ordonnancement qui suit, t_i est le temps d'entrée dans la file d'attente du processus P_i et τ_i est son temps d'execution.

	t_i	τ_i
P_1	0	3
P_2	1	2
P_3	2	1
P_4	4	2
P_5	4	1

– Rappeler la définition du temps de traitement d'un processus et du temps de traitement moyen d'un ensemble de processus.

Solution. On a

$$temps\ traitement = (temps\ fin\ du\ traitement) - (temps\ entree\ dans\ la\ file\ d\ attente)$$

Si on dénote TT_i le temps de traitement du processus i, on aura

$$temps\ traitement\ moyen = TTM = \frac{\sum_{i=1,\ldots,n} TT_i}{n}$$

où n est le nombre des processus. □

– Calculer les temps de traitement moyen pour ces processus en utilisant les algorithmes d'ordonnancement FIFO et PCTE.

Solution. FIFO : on a :

$$TT_1 = 3 - 0 = 3$$
$$TT_2 = 5 - 1 = 4$$
$$TT_3 = 6 - 2 = 4$$
$$TT_4 = 8 - 4 = 4$$
$$TT_5 = 9 - 4 = 5$$

et donc

$$TTM = 20/5 = 4\,.$$

PCTE :

$$TT_1 = 3 - 0 = 3$$
$$TT_3 = 4 - 2 = 2$$
$$TT_5 = 5 - 4 = 1$$
$$TT_2 = 7 - 1 = 6$$
$$TT_4 = 9 - 4 = 5$$

et donc

$$TTM = 17/5 = 3,4\,.$$

□

Les processus

Exercice 2. Considérer le programme suivant :

```
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <stdio.h>
4  #include <wait.h>
5
6  char n;
7
8  int main(void)
9  {
10   pid_t pid;
11   int status;
12
13   if( (pid = fork()) == -1 ) exit(EXIT_FAILURE);
14   {
15
16     if(pid != 0){
17       n++;
18       wait(&status);
19       if(WIFEXITED(status)) n+=WEXITSTATUS(status);
20     }
21     else
22       n--;
23   }
24   printf("[%d] : valeur de n est %d\n",getpid(),-n);
25   exit(n);
26 }
```

1. Détailler (ligne par ligne) le comportement de ce programme.
2. On suppose que : le PID du père est 2000, le processus fils a PID 2001 et se termine de façon normale. Dire ce qui est affiché à l'écran.

   Solution.

   ```
   ex2
   [2001] : valeur de n est 1
   [2000] : valeur de n est 0
   ```

   □

3. Sous les mêmes hypothèses précédentes, dire ce qui est affiché à l'écran si on déplace la ligne 6 respectivement à la ligne 12 et à la ligne 15. Justifiez vos réponses.

   Solution. Si on déplace la ligne 6 à la ligne 12 : la variable n n'est plus initialisé à 0 car elle appartient à la pile, mais au début elle a le même valeur pour le père et le fils. Par exemple, si elle vaut 5, on aura :

   ```
   [2001] : valeur de n est -4
   [2000] : valeur de n est -10
   ```

   Si elle vaut x, alors on aura

   ```
   [2001] : valeur de n est -(x-1)
   [2000] : valeur de n est -(2x)
   ```

   Si on déplace la ligne 6 à la ligne 15, ... on obtiendra un erreur à la compilation (L.S. s'excuse pour cette question ratée).

   □

Exercice 3. Considérer le programme suivante :

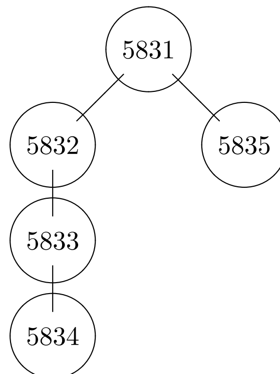
```
1  #include <unistd.h>
2
3  int main(void)
4  {
5    int i;
6
7    for(i=0;i<3;i++)
8      if(fork()) i++;
9    sleep(5);
10   return i;
11 }
```

On suppose que tout appel à la primitive `fork` ne renvoie pas un code d'erreur.

1. Dire combien de processus cette instruction engendre (on ne compte pas le père).
2. Faire un dessin de l'arbre généalogique du père et des processus engendrés.
3. Suggérer une ligne de commande shell qui permet de vérifier vos réponses.

Solution.

```
ex3 & (ps -ao pid,ppid,cmd | grep ex3)
 5830  5808 /bin/sh -c ex3 & (ps -ao pid,ppid,cmd | grep ex3)
 5831  5830 ex3
 5832  5831 ex3
 5833  5832 ex3
 5834  5833 ex3
 5835  5831 ex3
   ...
```



□

Exercice 4. Écrire un programme qui engendre trois fils. Un fils rentrera dans l'état zombie et puis deviendra orphelin. Un deuxième fils deviendra orphelin avant rentrer dans l'état zombie. Un troisième fils deviendra zombie et il ne sera jamais orphelin.

Solution.

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <wait.h>
5
6  #define MONPID printf("[%d] ",getpid());
7
8  void fils_orphelin()
9  {
10   sleep(5);
11
12   MONPID;
13   printf("Mon père est maintenant le processus %d.\n",getppid());
```

```
14
15   exit(EXIT_SUCCESS);
16 }
17
18 void fils_zombie()
19 {
20   MONPID;
21   printf("Fin.\n");
22   exit(EXIT_SUCCESS);
23 }
24
25 int main(void)
26 {
27   pid_t pid;
28   int i;
29
30   for(i=0;i<3;i++)
31     {
32       switch(pid = fork())
33         {
34         case -1: exit(EXIT_FAILURE);
35         case 0:
36           switch(i)
37             {
38             case 0:
39               fils_orphelin();
40             default:
41               fils_zombie();
42             }
43         default:
44           MONPID;
45           printf("Fils %d engendré\n",pid);
46         }
47     }
48   sleep(2);
49   pid = wait(NULL); /*
50                         On récupère un seul zombie.
51                         L'autre sera récupéré par le init (proc. no 1),
52                         et sera donc le fils zombie et orphelin.
53                      */
54   MONPID;
55   printf("Fils %d recuperé\n",pid);
56
57   exit(EXIT_SUCCESS);
58 }
```

□

Le système des fichiers

Exercice 5. Écrire un programme (en C) qui compte la taille totale de tous les fichier réguliers qui sont des liens durs contenus dans un répertoire. Ce répertoire sera passé en paramètre. Si la ligne de commande ne contient pas de paramètre, alors le répertoire courant de travail sera considéré.

Solution.

```
1 #include <stdio.h>
2 #include <stdlib.h>
```

```
3  #include <sys/stat.h>
4  #include <dirent.h>
5  #include <string.h>
6
7  #define TAILLE 1024
8
9  int main(int argc, char *argv[])
10 {
11   DIR * repertoire;
12   struct dirent *entree;
13   struct stat fic_info;
14   char chemin[TAILLE];
15   int longueur;
16   size_t taille_tot=0;
17
18   /*
19     Remarque :
20     strncat ajoute toujours '\0' a la fin de la chaîne
21     Pas la même chose avec strncpy
22   */
23   chemin[0] = 0;
24   if(argc > 1)
25     strncat(chemin,argv[1],TAILLE-1);
26   else
27     strncat(chemin,"./",TAILLE-1);
28
29   longueur=strlen(chemin);
30   if(chemin[longueur-1] != '/')
31     strncat(chemin,"/",TAILLE-longueur-1);
32   longueur=strlen(chemin);
33
34   if((repertoire = opendir(chemin)) == NULL)
35     {
36       fprintf(stderr,"%s: ",chemin);
37       perror("opendir");
38       exit(EXIT_FAILURE);
39     }
40
41   while((entree = readdir(repertoire)) != NULL)
42     {
43       chemin[longueur]=0;
44       strncat(chemin,entree->d_name,TAILLE-longueur-1);
45
46       if(lstat(chemin,&fic_info) == -1)
47         {
48           fprintf(stderr,"%s: ",chemin);
49           perror("lstat");
50           continue;
51         }
52       if(S_ISREG(fic_info.st_mode))
53         taille_tot+=fic_info.st_size;
54     }
55   closedir(repertoire);
56   chemin[longueur]=0;
57
58   printf("Répertoire %s :\n",chemin);
59   printf("%u octets utilisés par les fichiers réguliers.\n",taille_tot);
```

```
60     exit(EXIT_SUCCESS);
61  }
```

□

La communication par signaux

Exercice 6. Considérer le programme suivant :

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <unistd.h>
 4  #include <signal.h>
 5  #include <time.h>
 6
 7  void interruption(int signum)
 8  {
 9    printf("Deux\n");
10  }
11
12  int main(void)
13  {
14    signal(SIGINT,interruption);
15    srand(time(NULL));
16
17    switch(fork()){
18    case -1:
19      exit(EXIT_FAILURE);
20    case 0:
21      sleep(rand()% 2);
22      printf("Trois\n");
23      kill(getppid(),SIGINT);
24    default:
25      sleep(rand()% 2);
26      printf("Un\n");
27    }
28
29    exit(EXIT_SUCCESS);
30  }
```

– Quels sont les possibles affichages à l'exécution de ce programme ?

  Solution. Le fils affichera
  `Trois`
  `Un`
  et le père
  `Un Deux`
  où l'ordre d'affichage dépendra de l'istant auquel le fils envoie au père le signal `SIGINT`. On a donc les contraintes suivantes : `Trois` est suivis par un `Un` (il n'y pas de `break` après la ligne 23), `Trois` est suivis par `Deux`. On aura donc un des affichages suivants :

`Trois`	`Trois`
`Un`	`Deux`
`Deux`	`Un`

  avec un `Un` supplémentaire dans une position quelconque. □

– À l'exécution sur une machine, on note que *Trois* est toujours affiché avant que *Un*. Pour quelle raison peut cela se passer ?

  Solution. On note que le nombre des seconds choisis par le père et le fils est le même (initialisation du générateur

de nombre aléatoire avant l'appel à `fork`). Si l'ordonnancer donne préférence au fils au retour de l'appel système `fork`, le père affichera son `Un` toujours après le `Trois` du fils.
On peut alors déplacer l'initialisation du générateur de nombre aléatoire à la ligne 21 et a la ligne 25, de façon que cette initialisation ne soit pas le même pour le père et le fils et qu'ils s'endort pendant un nombre différents de seconds. □

Exercice 7. Considérer le programme suivant :

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <wait.h>
4
5  int main(void)
6  {
7    if(fork()) wait(NULL);
8
9    printf("[%d] fini \n",getpid());
10   return 0;
11 }
```

En sachant qu'à la terminaison d'un fils le signal `SIGCHLD` est envoyé au père, réécrire ce programme sans utiliser l'appel système `wait`. Par exemple, en utilisant l'appel système `pause`, on pourra écrire une fonction

`void wait_simple(void)`

qui met le processus dans l'état endormis jusqu'à ce qu'un signal `SIGCHLD` est reçu.

Solution.

```
1  #ifndef WAIT_SIMPLE_H
2  #define WAIT_SIMPLE_H
3
4  #include <signal.h>
5
6  void init_wait_simple(void);
7  void end_wait_simple(void);
8
9  void wait_simple(void);
10
11 #endif /* WAIT_SIMPLE_H */
```

```
1  #include <unistd.h>
2  #include "wait_simple.h"
3
4  static int drapeau = 0 ;
5
6  static void
7  interruption(int signum)
8  {
9    drapeau = 1;
10   return;
11 }
12
13 static void (*old)(int signum);
14 void init_wait_simple(void)
15 {
16   old = signal(SIGCHLD,interruption);
17 }
18
19 void end_wait_simple(void)
20 {
```

```
21    signal(SIGCHLD,old);
22  }
23
24  void wait_simple(void)
25  {
26    while(drapeau == 0)
27      pause();
28    drapeau = 0;
29  }
```

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <unistd.h>
 4
 5  #include "wait_simple.h"
 6
 7  int main(void)
 8  {
 9    init_wait_simple();
10
11    if(fork())
12      wait_simple();
13
14    printf("[%d] fini\n",getpid());
15
16    end_wait_simple();
17
18    exit(EXIT_SUCCESS);
19  }
```

□