Mercredi 16 novembre 2005
Durée : 2 heures
Documents autorisés

Partiel : systèmes d'exploitation
L3, parcours informatique, 2006-2007

Examen partiel

Ordonnancement

Exercice 1. 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'exécution :

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

Rappelons que le *temps de traitement* TT_i du processus P_i est calculé comme le temps de terminaison de ses calculs moins le temps d'entrée dans la file d'attente. En utilisant cette table et en supposant que le *quantum* soit égal à 2 unités de temps, pour chacun des algorithmes d'ordonnancement Tourniquet et PCTER :

1. illustrez le déroulement de l'algorithme,
2. calculez les temps de traitement de chaque processus, le temps de traitement total, le temps de traitement moyen.

Solution. Dans la suite, nous allons décrire l'état de la file d'attente par une liste de couple. Chaque couple contiendra l'identificateur du processus ainsi que son temps d'exécution restant.

Algorithme Tourniquet :

temps	Proc. choisi	Etat de la file
0	P_1	$(P_1, 3)$
2	P_2	$(P_2, 2)(P_3, 1)(P_1, 1)$
4	P_3	$(P_3, 1)(P_1, 1)(P_4, 3)(P_5, 1)$
5	P_1	$(P_1, 1)(P_4, 3)(P_5, 1)$
6	P_4	$(P_4, 3)(P_5, 1)$
8	P_5	$(P_5, 1)(P_4, 1)$
9	P_4	$(P_4, 1)$
10	terminaison	

On obtient :

$$\begin{aligned} TT_1 &= 6 - 0 = 6 \\ TT_2 &= 4 - 1 = 3 \\ TT_3 &= 5 - 2 = 3 \\ TT_4 &= 10 - 4 = 6 \\ TT_5 &= 9 - 4 = 5 \\ TT_T &= (6 + 3 + 3 + 6 + 5) = 23 \\ TT_M &= 23/5 = 4,6 \end{aligned}$$

Algorithme PCTER :

temps	Proc. choisi	Etat de la file
0	P_1	$(P_1, 3)$
2	P_3	$(P_2, 2)(P_3, 1)(P_1, 1)$
3	P_1	$(P_2, 2)(P_1, 1)$
4	P_5	$(P_2, 2)(P_4, 3)(P_5, 1)$
5	P_2	$(P_2, 2)(P_4, 3)$
7	P_4	$(P_4, 3)$
9	P_4	$(P_4, 1)$
10	terminaison	

On obtient :

$$\begin{aligned} TT_1 &= 4 - 0 = 4 \\ TT_2 &= 7 - 1 = 6 \\ TT_3 &= 3 - 2 = 1 \\ TT_4 &= 10 - 4 = 6 \\ TT_5 &= 5 - 4 = 1 \\ TT_T &= 4 + 6 + 1 + 6 + 1 = 18 \\ TT_M &= 18/5 = 3,6 \end{aligned}$$

□

Les processus

Exercice 2. Considérez le programme suivant :

```
 6 static int n=3;
 7
 8 int main(int argc,char *argv[]) {
 9   int i;
10   for (i=0; i<n; i++)
11     if (fork()==0) { n--; main(argc,argv); }
12   while (wait(NULL)!= -1);
13   exit(EXIT_SUCCESS);
14 }
```

1. Combien de processus sont-ils créés lors d'une exécution ?

   Solution. Le père créera 3 fils. Chaque fils créera aura 2 grand-fils. Chaque grand fils créera 1 grand-grand-fils. En comptant le père in aura

   $$1 + 3 + 3 * (2 + 2 * 1) = 16 \text{ processus, en comptant aussi le père.}$$

   □

2. Dessinez l'arbre généalogique des processus créés.

   Solution. On peut modifier le programme ci-dessus comme il suit

   ```
    6 static int n=3;
    7
    8 int main(int argc,char *argv[]) {
    9   int i;
   10   for (i=0; i<n; i++)
   11     if (fork()==0) { n--; main(argc,argv); }
   12   while (wait(NULL)!= -1);
   13
   14   for(i=3-n;i>0;i--) printf("\t");
   15   printf("[%d] Mon père est : <%d>, je termine.\n",
   16          getpid(),getppid());
   17
   18   exit(EXIT_SUCCESS);
   19 }
   ```

   de façon qu'il affiche lui même l'arbre généalogique :

```
$ ./a.out
                        [15428] Mon père est : <15427>, je termine.
                [15427] Mon père est : <15426>, je termine.
                        [15430] Mon père est : <15429>, je termine.
                [15429] Mon père est : <15426>, je termine.
        [15426] Mon père est : <15419>, je termine.
                        [15434] Mon père est : <15433>, je termine.
                [15433] Mon père est : <15424>, je termine.
                        [15422] Mon père est : <15421>, je termine.
                [15421] Mon père est : <15420>, je termine.
                        [15431] Mon père est : <15423>, je termine.
                [15423] Mon père est : <15420>, je termine.
                        [15432] Mon père est : <15425>, je termine.
                [15425] Mon père est : <15424>, je termine.
        [15420] Mon père est : <15419>, je termine.
        [15424] Mon père est : <15419>, je termine.
[15419] Mon père est : <5392>, je termine.
$
```

□

3. Expliquez en détailles à quoi sert la ligne 12.

   Solution. Un processus sortira de cette boucle `while` quand l'appel système `wait` échoue, ce qui s'avère quand un processus ne possède plus de fils (vivants ou zombies). Il s'agir donc d'une ligne de code permettant à une père de se mettre en attente de tous ses fils. □

4. Que ce passe-t'il si l'on remplace l'appel à `main(argc,argv[])` par un appel à `execvp(argv[0],argv)` ?

   Solution. Avec l'appel `execvp` l'espace d'adressage du processus `a.out` sera substitué par celui d'un autre processus `a.out`. En particulier, la variable entière `n` sera réinitialisé à 3, chaque fois que l'on crée un fils.

   Chaque processus essaiera par conséquence de créer 3 fils. Cette chaîne – qui en théorie est infinie – se terminera à cause du fait que toutes les ressources nécessaires pour créer des nouveaux processus (mémoire, entrées libres dans la table des processus, etc.) seront utilisées. □

Exercice 3. Après avoir expliqué ce que veut dire

1. qu'un processus est dans l'état zombie,
2. qu'un processus est orphelin,

écrivez un programme qui crée un processus qui rentre dans l'état zombie avant devenir orphelin.

Solution. Un processus se trouve dans l'état *zombie* après un appel système à la primitive `_exit` et avant que son père n'ait récupéré les statistiques de sortie de ce processus par la primitive `wait`. Dans l'état zombie l'espace d'adressage du processus a été libéré, mais l'entrée dans la table des processus est encore utilisée.
Un processus est *orphelin* de que son père fait un appel système `_exit` sans avoir préalablement fait un appel système à la primitive `wait` afin de récupérer les statistiques de ce fils.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>


void terminaison(int sig)
{
  if (sig == SIGCHLD)
    {
      printf("Le fils s'est terminé !!\n");
      printf("Le père se termine après le fils sans faire wait.\n");

```

```
14        exit(EXIT_SUCCESS);
15      }
16  }
17
18
19  int main(void)
20  {
21    signal(SIGCHLD,terminaison);
22
23    if(fork()== 0)
24      exit(EXIT_SUCCESS);
25
26    pause();
27
28    exit(EXIT_SUCCESS);
29
30  }
```

□

Exercice 4. Écrivez un programme permettant d'exécuter en parallèle un ensemble de programmes donnés en paramètres et qui s'arrête à la terminaison du dernier, le code de retour de la commande signalera un échec si et seulement si l'une des commandes termine en échec. Le prototype de la commande est : `parallele c1 c2 ... cn`.

Solution.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <wait.h>
5
6  int main(int argc, char *argv[])
7  {
8    int i,status,code_sortie=EXIT_SUCCESS;
9
10   for (i =1;i< argc; i++)
11     {
12       switch(fork())
13         {
14         case -1:
15           perror("fork");
16           break;
17         case 0:
18           execlp(argv[i],argv[i],NULL);
19           perror("execlp");
20           exit(EXIT_FAILURE); /* Ambiguité dans l'énoncé ? */
21         default:
22           break;
23         }
24     }
25
26   while(wait(&status) != -1)
27     {
28       if(WIFEXITED(status))
29         {
30           if(WEXITSTATUS(status) != EXIT_SUCCESS)
31             code_sortie = EXIT_FAILURE;
32         }
33       else
```

```
34          code_sortie = EXIT_FAILURE;
35      }
36
37    return code_sortie;
38  }
```

□

La communication par signaux

Exercice 5. Considérez le code suivant :

```
 6  struct tache{
 7    int signal;
 8    void (*handler)(int);
 9    int priorite;
10  };
```

```
 4  int comparer(const void *s, const void *t){
 5    const struct tache *s1 = s, *t1=t;
 6
 7    if (s1->priorite < t1->priorite)
 8      return -1 ;
 9    else if  (s1->priorite == t1->priorite)
10      return 0;
11    return 1;
12  }
13
14  int preparer(struct tache t[],int taille){
15    struct sigaction action;
16    int i;
17
18    qsort(t,taille,sizeof(struct tache),comparer);
19
20    action.sa_flags=0;
21    sigemptyset(&action.sa_mask);
22    for(i=0; i<taille;
23        sigaddset(&action.sa_mask,t[i++].signal))
24      {
25        action.sa_handler = t[i].handler;
26        sigaction(t[i].signal,&action,NULL);
27      }
28
29    return 0;
30  }
```

1. Expliquez ce que fait la fonction `preparer`.

   Solution. La fonction `preparer` prend en paramètre un tableau de structures `tache` ainsi que la taille de ce tableau. Ce tableau est trié par ordre croissante du champs `priorite`.
   La table des signaux du processus est initialisé par un cycle `for`, de façon que le traitement d'une tache de priorité plus petite soit bloquée pendant le traitement d'une tache de priorité plus haute. □

2. Proposez un exemple d'utilisation de cette fonction.

   Solution. Dans un système d'exploitation, à chaque interruption matérielle est associée une priorité. Par exemple, dans un système UNIX, l'interruption associée à l'horloge est la plus prioritaire. Pendant la prise en compte d'une interruption plus prioritaire on ne prend pas en compte les interruptions moins prioritaires.

Par exemple, le système ne traitera pas l'interruption associée au clavier s'il est en train de mettre à jour des données à cause de l'interruption horloge.

On peut se servir du code ci-dessus pour simuler le traitement des interruption matérielles (qui se déroulent dans l'espace noyau) par le traitement des signaux (aussi appelés interruptions logicielles, dont le traitement se déroule dans l'espace utilisateur). □

3. Critiquez cette implémentation et suggérez comment l'améliorer.

Solution. On remarquera qu'avec ce code deux taches pourraient avoir la même priorité, mais une sera bloquée pendant l'exécution de l'autre.

Pour prendre en considération ce cas on peut modifier le code comme il suit :

```
 1 #include <stdlib.h>
 2 #include <assert.h>
 3 #include <limits.h>
 4 #include "sigp.h"
 5
 6 int comparer(const void *s, const void *t){
 7   const struct tache *s1 = s, *t1=t;
 8
 9   if (s1->priorite < t1->priorite)
10     return -1 ;
11   else if  (s1->priorite == t1->priorite)
12     return 0;
13   return 1;
14 }
15
16 int ajouter(sigset_t *dest, sigset_t *source)
17 {
18   int sig;
19
20   for(sig=0; sig<NSIG;sig++)
21       if(sigismember(source,sig))
22         sigaddset(dest,sig);
23
24   return 0;
25 }
26
27 int preparer(struct tache t[],int taille){
28   struct sigaction action;
29   sigset_t ens;
30   int priorite_courante;
31   int i;
32
33   assert(taille >= 0);
34   qsort(t,taille,sizeof(struct tache),comparer);
35
36   action.sa_flags=0;
37   sigemptyset(&action.sa_mask);
38   sigemptyset(&ens);
39   priorite_courante = INT_MIN;
40
41   for(i=0; i< taille ; i++)
42     {
43       if(t[i].priorite > priorite_courante)
44         {
45           t[i].priorite = priorite_courante;
46           ajouter(&action.sa_mask,&ens);
47           sigemptyset(&ens);
```

```
48          }
49        action.sa_handler = t[i].handler;
50        sigaction(t[i].signal,&action,NULL);
51
52        sigaddset(ens,t.signal);
53      }
54
55    return 0;
56  }
```

□

Le système des fichiers

Exercice 6. Considérez la séquence de commandes suivantes (rien d'autre ne se passe dans le système pendant cette séquence) :

```
$ ls rep/
fic1  fic2
$ ls -l rep/
total 0
?--------- ? ? ? ?                ? rep/fic1
?--------- ? ? ? ?                ? rep/fic2
```

Comment une telle situation est-elle possible ?

Solution. Il suffit que le répertoire `rep` n'ait pas le droit en exécution pour l'utilisateur à la console. Il sera alors possible lire et d'afficher son contenu (une suite de liens). L'accès (par exemple à l'aide d'un appel système `stat`) aux données relatifs à cette liste de liens sera interdit.

```
$ ls -ld rep/
drw-r-xr-x 2 lsantoca lsantoca 4096 2006-11-17 09:48 rep/
```

□

Exercice 7. Considérez les deux programmes suivants :

```
1  /* cp1.c */
2  int main(int argc,char *argv[]) {
3    int d1, d2;
4    char c;
5
6    d1 = open(argv[1],O_RDONLY);
7    d2 = open(argv[2],O_WRONLY|O_CREAT|O_TRUNC,S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
8    while ( read(d1,&c,1)==1 ) write(d2,&c,1);
9    close(d1);
10   close(d2);
11   exit(EXIT_SUCCESS);
12 }
13
14 /* cp2.c */
15 int main(int argc,char *argv[]) {
16   FILE *d1, *d2;
17   char c;
18
19   d1 = fopen(argv[1],"r");
20   d2 = fopen(argv[2],"w+");
21   while ( fread(&c,1,1,d1)==1 ) fwrite(&c,1,1,d2);
22   fclose(d1);
23   fclose(d2);
```

```
24     exit(EXIT_SUCCESS);
25  }
```

Voici les temps d'exécution de chacun d'eux alors que le fichier f est de taille 1752064 octets :

commande	temps utilisateur	temps système
`cp1 f g`	8,91s	53,86s
`cp2 f g`	1,01s	0,06s

Pouvez-vous justifier ces mesures ?

Solution. Les fonctions `fread` et `fwrite` appartiennent à la bibliothèque standard C, qu'implémente pour chaque flot (fichier ouvert où structure de type `FILE`) un tampon pour optimiser les opérations d'entrées-sorties. Par conséquence, à une opération d'écriture/lecture à l'aide de `fwrite` ou `fread` ne corresponde pas nécessairement un appel système `write`/`read`, de façon que les changements de contextes – en nombre de au moins 2*(longueur du fichier) pour `cp1` – soient en nombre assez inférieur pour `cp2`. □