Mardi 9 janvier 2007
Durée : 3 heures
Documents autorisés

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

Examen

Les valeurs de retour des appels système ne sont pas systématiquement testées dans les programmes de l'énoncé : on suppose qu'ils ne renvoient jamais un code d'erreur. Vous pouvez faire de même dans vos solutions.

Gestion de la mémoire, pagination, pagination à la demande

Exercice 1. (3 points)

1. Expliquez ce qu'est la fragmentation de la mémoire.

   Solution. On dit que la mémoire est fragmentée quand, à la suite d'une séquence d'allocations/libérations de portions contiguës de mémoire, les portions de mémoire libres s'entrelacent aux portions de mémoire utilisés, de façon que, même si la taille totale des portions libres est importante, la taille individuelle des portions libres est insuffisante pour satisfaire des nouvelles demandes d'allocation de mémoire. □

2. Dans un cadre d'allocation de la mémoire contiguë, de quelle façon peut on apporter remède à la fragmentation ?

   Solution. On peut utiliser le compactage. Cette technique consiste à déplacer les portions utilisés de mémoire de façon à ce qu'ils soient contiguës et ne s'entrelacent pas avec les portions libres. Les portions libres seront alors contiguës, et on pourra les fusionner ensemble pour obtenir une seule portion de mémoire dont la taille est somme des tailles des portions libres existantes avant la compactage. □

3. Expliquez pourquoi la pagination est une solution à la fragmentation.

   Solution. La pagination est une solution à la fragmentation exactement par ce qu'elle permet de sortir d'un schéma d'allocation contiguë.
   Nous savons que la mémoire physique est divisé en cadres de page ayant taille fixe T. Même si les cadres de page occupés s'entrelacent aux cadres de page libres, et la taille de chaque cadre de page est insuffisante pour satisfaire une demande d'allocation de mémoire de taille X, il suffit de trouver $X/T + 1$ cadres de page libres pour pouvoir satisfaire cette demande. Ces cadres ne seront pas nécessairement contigus.
   D'ailleurs, on peut mapper la mémoire logique vers la mémoire physique de façon à ce que la mémoire allouée apparaisse comme une portion contiguë si l'on considère les adresses logiques. □

Exercice 2. (2 points) Dans une architecture où les adresses sont sur 32 bits, le 22 bits plus à gauche d'un adresse dénotent la page, le 10 bits plus à droite dénotent le déplacement par rapport au début de la page. En sachant que les premières entrées de la table des pages sont :

3
6
7
1
⋮

transformez les deux adresses logiques suivantes (exprimées en notation binaire) :

1. `101010101010`,
2. `1`,

en adresses physiques.

Solution. `10 1010101010` : la page demandé est `10` (i.e. 2), le deplacement etant `1010101010`.
Le cadre correspondant à 2 est 7, c.à.d. `111` en binaire.
L'adresse physyque correspondant est `111 1010101010`.

`1`, c.-à-d. `0 0000000001` : la page est `0`. Le cadre correspondant etant `3`, `11` en binaire, l'adresse correspondant est `11 0000000001`. □

Exercice 3. (3 points) La mémoire vive d'un ordinateur contient 4 cadres de page et, au début, tous les cadres sont vides. Combien de défauts de page produit la suite de références de page

3,4,4,1,5,2,3,1,4

en utilisant, respectivement, les algorithmes de remplacement FIFO, OPTIMAL, et LRU ? Justifiez vos réponses en élucidant les déroulements des algorithmes : on montrera le contenu des cadres après chaque référence (et son traitement éventuel).

Solution. Algorithme FIFO : 7 défauts de pages.

Référence	Défaut de page	Cadres après le traitement
3	*	3
4	*	34
4		34
1	*	341
5	*	3415
2	*	2415
3	*	2315
1		2315
4	*	2345

Algorithme OPTIMAL : 5 défauts de pages.

Référence	Défaut de page	Cadres après le traitement
3	*	3
4	*	34
4		34
1	*	341
5	*	3415
2	*	3412
3		3412
1		3412
4		3412

Algorithme LRU : 7 défauts de pages.

Référence	Défaut de page	Cadres après le traitement
3	*	3
4	*	34
4		34
1	*	341
5	*	3415
2	*	2415
3	*	2315
1		2315
4	*	2314

□

La concurrence

Exercice 4. (4 points)

1. Expliquez ce qu'est une région critique. Apportez un exemple.

   Solution. Une région critique d'un processus est une suite d'instructions (i.e. un morceaux du code) du processus dont l'exécution ne devrait pas s'entrelacer avec l'exécution de certain morceaux de code d'autres processus. D'habitude, une région est critique de qu'elle fait accès à des ressources partagés avec d'autres processus : l'accès devrait se faire de façon exclusive, c.-à-d. aucun autre processus devrait accéder à la ressource pendant que ce processus est en train d'exécuter sa région critique. Par exemple, un modification partielle des ressources partagés pourrai entraîner une inconsistance des ressources (et en général un crash du système d'exploitation).

Pour pouvoir assurer l'accès exclusif à la ressource, et donc la protection de la region crique, on met en place nombre d'algorithmes et structures de données (algorithme de Peterson, sémaphores, mutex, etc.)

Dans le code d'un système d'exploitation, la plupart des données sont partagés entre plusieurs processus. Si par exemple un processus démarre la modification d'une liste et une interruption déclenche un changement de contexte sans que cette modification soit complété, un autre processus pourrait essayer de lire/modifier la même liste : ce processus trouvera la liste dans un état inconsistant (par exemple, les pointeurs vers le suivant/précédent pourraient ne pas être à jour). □

2. Rappelons que l'appel système

```
int mkdir(const char *path, mode_t mode);
```

retourne un code d'erreur si le répertoire à créer existe déjà. Expliquez, par conséquence, comment par l'implantation suivante des fonctions `entrer_region` et `sortir_region` peut protéger la section critique d'un processus et assurer l'exclusion mutuelle :

```
1  void entrer_region(char *verrou)
2  {  while(mkdir(verrou,0777) < 0) sleep(1); }
3
4  void sortir_region(char *verrou)
5  {  unlink(verrou); }
```

Solution. Bien sur, `entrer_region` sera exécuté avant la section critique, et `sortir_region` sera exécuté à la fin de a region critique.

On se servira de l'existence du répertoire référencé par la chaîne de caractères `verrou`, comme un drapeau signalant que quelques autre processus est déjà en train d'exécuter sa region critique. Étant donné que l'appel système POSIX retourne un code d'erreur si le répertoire existe déjà, on peut paraphraser le code de `entrer_region` comme il suit : tant que le répertoire référencé par `verrou` existe, se mettre dans l'état endormi et ré-essayer après un seconde. Si le répertoire n'existe pas le créer et entrer dans la region critique.

Le code de `sortir_region` effacera (appel système `unlink`) le répertoire et donc signalera aux autres processus que l'on a terminé d'exécuter la region critique.

Observons que, dans le code de `entrer_region`, le test d'existence du répertoire `verrou` et la création éventuelle de ce répertoire se fait en un seul coup par l'appel système `mkdir`. Car les processus ne sont pas interruptibles pendant l'exécution d'un appel système, le test d'existence et la création seront atomiques. □

3. Dans le code suivant `verrou` est l'adresse d'une variable entière que l'on suppose être partagée entre plusieurs processus. On peut alors protéger la section critique d'un processus par

```
1  #define LIBRE 1
2  #define OCCUPE 0
3
4  /* verrou : adresse d'une variable partagée entre plusieurs processus */
5  void entrer_region(int *verrou) {
6    while(*verrou == OCCUPE) sleep(1);
7    *verrou = OCCUPE;
8  }
9
10 void sortir_region(int *verrou)
11 {  *verrou = LIBRE; }
```

Expliquez en quoi cette implantation est pire que la précédente.

Solution. Le problème est ici que le test de la variable `verrou` et son affectation éventuelle à la valeur `OCCUPE` pourraient s'entralacer avec l'execution d'autres processus (i.e. ne sont pas atomiques). Supposons qu'on a deux processus $P1$, $P2$, qui utilisent cette implantation des fonctions `entrer_region`, `sortir_region`. Considérons le déroulement parallèle suivant :

```
1          P1                        | P2
2          if(*verrou == LIBRE)      |
3                                    | if(*verrou == LIBRE)
4          *verrou=OCCUPE            |
```

Les processus $P1$ et $P2$ exécuteront leurs regions critiques de façon non exclusive. □

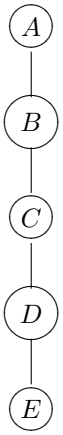
Les processus

Exercice 5. (4 points) Considérez le code suivant :

```
 5  int main(void){
 6    int i,naptime;
 7
 8    for(i=0;i<4;i++) if(fork()) break;
 9
10    srand(getpid());
11    sleep(naptime = rand()%4);
12
13    printf("Mon nom est <%c>, "
14           "j'ai dormi %d secondes\n",'A'+i,naptime);
15
16    exit(EXIT_SUCCESS);
17  }
```

1. Dessinez l'arbre généalogique des processus engendrés par ce programme.

   Solution.

   □

2. Décrivez en quelques mots quels sont tous les possibles affichages à l'écran de ce programme.

   Solution. La chaine de caracteres

   ```
   Mon nom est <c> j'ai dormi n secondes
   ```

   s'affichera 5 fois, avec `c` = A,B,C,D,E et `n` un nombre arbitraire entre 0 et 3 (inclus). L'ordre de la affichage dependera seulement du temps choisi pour dormir (et l'ordonnancement), et donc il sera arbitraire. □

3. Que se passe t-il si l'on déplace la ligne 10 à la ligne 7 ? Justifiez votre réponse.

   Solution. Chaque processus s'endormira le même nombre de seconds.
   En effet, le générateur de nombres aléatoires est initialisé une fois seule, par le processus ancêtre (appel à la fonction `srand` avec en paramètre le valeur de retour de l'appel système `getpid`, i.e. le pid du processus ancêtre).
   Par conséquence tous les processus choisiront (avec `rand` à la ligne 11) le premier nombre de la même suite aléatoire infinie. □

4. Sans modifier les lignes de 8 à 11, modifiez le programme de façon à ce que les processus fassent leur affichage par ordre alphabétique inversé du nom.

   Solution. Nous pouvons forcer chaque père à attendre la morts de son enfant avant faire son affichage. Pour cela nous pouvons ajouter

   ```
   12          wait(NULL);
   ```

   à la ligne 12. □

La communication par tubes

Exercice 6. (4 points) Écrivez un programme `exectube` dont le prototype est

```
exectube premier_commande [p_arg1 ...] tube deuxieme_commande [d_arg1 ...]
```

Ce programme :
– créera un tube et un fils, et rédirigera la sortie standard du fils vers l'entrée standard du père,
– le fils exécutera la commande (avec paramètres éventuels) contenue de `argv[1]` jusqu'au mot clés `tube` (exclus),
– le père exécutera la commande (avec paramètres éventuels) suivante le mot clés `tube`.
Par exemple, la commande `exectube ls -l tube wc -l` sera équivalente à la commande shell `ls -l | wc -l`.

Solution.

```
 1  #include <stdlib.h>
 2  #include <string.h>
 3  #include <stdio.h>
 4  #include <unistd.h>
 5
 6  #define APPEL_SYS(appel) \
 7  { if((appel) < 0) \
 8    {fprintf(stderr,"%s , ligne %d, probleme avec \n", argv[0],__LINE__);\
 9   perror(#appel);exit(EXIT_FAILURE);}}
10
11  int usage(char *nom_commande)
12  {
13    fprintf(stderr,"Usage : %s "
14            "premier_commande [args1] tube deuxieme_commande [args2]\n",
15            nom_commande);
16    exit(EXIT_FAILURE);
17  }
18
19  int chercher_tube(int argc,char *argv[])
20  {
21    int i;
22
23    for(i=1;i<argc;i++)
24      if(strcmp("tube",argv[i]) == 0) break;
25
26    return i;
27  }
28
29  int main(int argc, char *argv[])
30  {
31    int index_of_tube = chercher_tube(argc,argv);
32    int tube[2], pid;
33
34    if(index_of_tube == 1 || argc - (index_of_tube +1) <= 0)
35      usage(argv[0]);
36
37    APPEL_SYS(pipe(tube));
38    APPEL_SYS(pid = fork());
39    if(pid == 0)
40      {
41        APPEL_SYS(close(tube[0]));
42        APPEL_SYS(close(STDOUT_FILENO));
43        APPEL_SYS(dup(tube[1]));
44        APPEL_SYS(close(tube[1]));
45        argv[index_of_tube] = NULL;
```

```
46        APPEL_SYS(execvp(argv[1],&argv[1]));
47      }
48    else
49      {
50        APPEL_SYS(close(tube[1]));
51        APPEL_SYS(close(STDIN_FILENO));
52        APPEL_SYS(dup(tube[0]));
53        APPEL_SYS(close(tube[0]));
54        APPEL_SYS(execvp(argv[index_of_tube + 1],&argv[index_of_tube + 1]));
55      }
56
57    exit(EXIT_FAILURE);
58  }
```

□

Exercice 7. (4 points) La compilation du code suivant donne lieu à deux exécutables, un programme `serveur` et un programme `client` :

```
# Fichier Makefile

CFLAGS=-Wall -pedantic

all: client serveur

serveur : serveur.o
client : client.o

serveur.o : serveur.c common.h
client.o : client.c common.h
```

```
/* Fichier common.h */

#ifndef COMMON_H
#define COMMON_H

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

#define NOM_TUBE_CS "/tmp/tube_client_serveur"
#define NOM_TUBE_SC "/tmp/tube_serveur_client"
#define FORMATER_NOM_TUBE_SC(ou,pid) \
     sprintf((ou),"%s_%d",NOM_TUBE_SC,(int)(pid))

#endif /* COMMON_H */
```

```
/* Fichier serveur.c */

#include "common.h"

int main(void)
{
  int c = 0,df;
  pid_t pid;
  char nom[255];

  mkfifo(NOM_TUBE_CS,0777);
  chmod(NOM_TUBE_CS,0777);
  for(;;c++){
    df = open(NOM_TUBE_CS,O_RDONLY);
    read(df,&pid,sizeof(pid_t));
    close(df);

    FORMATER_NOM_TUBE_SC(nom,pid);
    df = open(nom,O_WRONLY);
    write(df,&c,sizeof(int));
    close(df);
  }

  exit(EXIT_FAILURE);
}
```

```
/* Fichier client.c */

#include "common.h"

int main(void)
{
  int c,df;
  pid_t pid = getpid();
  char nom[255];

  FORMATER_NOM_TUBE_SC(nom,pid);
  mkfifo(nom,0777);chmod(nom,0777);

  df = open(NOM_TUBE_CS,O_WRONLY);
  write(df,&pid,sizeof(pid_t));
  close(df);

  df = open(nom,O_RDONLY);
  read(df,&c,sizeof(int));
  close(df);

  printf("J'ai réçu le numero %d.\n",c);

  unlink(nom);
  exit(EXIT_SUCCESS);
}
```

1. Décrivez, en bref, ce qu'il se passe si le serveur est en train de s'exécuter, et plusieurs clients sont exécutés.

   Solution. Le client crée d'abord un tube nommé qui sera utilisé en suite pour recevoir la réponse du serveur. Ce tube est privé entre ce client et le serveur, car le nom du tube est post-fixé par les chiffres du `pid` du client.
   Ensuite, le client contactera le serveur (par l'intermédiaire d'un tube partagé entre plusieurs clients) et lui communiquant son `pid`.
   Le serveur sera alors en mesure de répondre au client en lui attribuant un numéro unique. Ce numéro est communiqué au client par le tube privée entre le client et le serveur.
   De cette façon, si le serveur est en train de s'exécuter et plusieurs clients s'exécutent, chaque client verra lui être attribué un numéro unique (le numéro correspondra au numéro d'arrivée), qui ensuite affichera à l'écran. □

2. Donnez une explication technique à votre première réponse en analysant le code. Votre discussion focalisera sur les aspects de programmation système (au lieu des aspects de programmation en C).

Solution. Voir par exemple les commentaires suivants :

```
1  /* Fichier serveur.c */
2
3  #include "common.h"
4
5  int main(void)
6  {
7    int c = 0,df;
8    pid_t pid;
9    char nom[255];
10   // Tampon à remplir ... est que 255 est assez grand ?
11
12   mkfifo(NOM_TUBE_CS,0777);
13   // Creation du tube Client->Seveur
14   chmod(NOM_TUBE_CS,0777);
15   // Si la masque de creation est 022 (comme d'habitude)
16   // Alors le tube cree a mainetant les droits 755
17   // Mettre les droits à 777
18
19   for(;;c++){
20     df = open(NOM_TUBE_CS,O_RDONLY);
21     // Ouvir le tube C->S en lecture
22     // Attention au mechanisme de synchronisation:
23     // attente dans l'etat endormi jusqu'à ce que
24     // un processus ouvre le tube en ecriture.
25     read(df,&pid,sizeof(pid_t));
26     // Lecture : attente jusque sizeof(pid) lu ou
27     // nomebre ecrivain du tube egal à 0
28     // On lit le pid d'un client
29     close(df);
30     // On ferme ce tube pour pouvoir
31     // se resynchroniser avec un autre processus
32
33     FORMATER_NOM_TUBE_SC(nom,pid);
34     // Le nom du tube qu'on va creer est (par exemple)
35     // /tmp/tube_serveur_client_5000 ou 5000 est le pid
36     // que le serveur vient de lire du tube C->S
37     df = open(nom,O_WRONLY);
38     // On ouvre (en ecriture) un tube S->C
39     // Ce tube connu seulement entre le serveur et le client 5000
40     // Ouverture est bloquante
41     write(df,&c,sizeof(int));
42     // Ecriture de la valeur de c dans le tube
43     close(df);
44     // Fermeture de ce tube
45
46     //Incrementer c à la fin de la boucle
47   }
48
49   exit(EXIT_FAILURE);
50   // Ligne jaimais executé.
51
52 }
```

□

3. Dans ce code on a utilisé des tubes nommés. Peut-on remplacer les tubes nommés par des tubes anonymes ? Justifiez votre réponse.

Solution. Le fichier `Makefile` montre que le client et le serveur n'auront aucun lien de parenté au moment de l'exécution. La réponse est donc négative, car l'utilisation de tubes anonymes est possibles seulement pour les processus ayant un ancêtre commun qui a créé le tube anonyme. □

4. Modifiez le code du serveur afin qu'on puisse l'arrêter par l'envoie d'un signal `SIGQUIT` : à sa réception le serveur effacera le tube client-serveur avant de s'arrêter.

Solution.

```
1  /* Fichier serveur.c */
2
3  #include "common.h"
4  #include <signal.h>
5
6
7  void terminaison(int sig)
8  {
9    if(sig == SIGQUIT)
10     {
11       unlink(NOM_TUBE_CS);
12       exit(EXIT_SUCCESS);
13     }
14 }
15
16
17 int main(void)
18 {
19   int c = 0,df;
20   pid_t pid;
21   char nom[255];
22
23   struct sigaction action;
24   action.sa_handler=terminaison;
25   action.sa_flags=0;
26   sigemptyset(&action.sa_mask);
27   sigaction(SIGQUIT,&action,NULL);
28
29   mkfifo(NOM_TUBE_CS,0777);
30   chmod(NOM_TUBE_CS,0777);
31   for(;;c++){
32     df = open(NOM_TUBE_CS,O_RDONLY);
33     read(df,&pid,sizeof(pid_t));
34     close(df);
35
36     FORMATER_NOM_TUBE_SC(nom,pid);
37     df = open(nom,O_WRONLY);
38     write(df,&c,sizeof(int));
39     close(df);
40   }
41
42   exit(EXIT_FAILURE);
43 }
```

□