Annales

IN3ST01

2010/2011

Au programme :

- Sujet 2010 p.3 à 19

Partiel :

- 29/01/03 Sujet

Final:

- Février 2008 Sujet
- 24/02/06 Sujet + Copie
- 29/01/05 Sujet
- 04/02/04 Sujet + Copie
- 06/02/04 Sujet + Copie
- 2003 Sujet

N'oubliez pas ceci n'est qu'un support de révision !

Merci à tous ceux qui ont aidé et/ou participé à ces annales !

Michaël et Jérôme pour ExtaZ

Examen IN3ST01 - Systèmes d'Exploitation

Responsable : Laurent Najman
Février 2010
Durée: 3 heures - cinq pages

Instructions

- Ne vous contentez pas d'écrire un résultat. Justifiez et commentez vos réponses. Aucun point ne sera donné pour une réponse non justifiée ou une justification erronée.
- Pour les réponses qui demandent des programmes ou des morceaux de programmes, vous pouvez utilisez du C ou n'importe quel style de pseudocode suffisamment raisonnable (*i.e.*, la syntaxe et la sémantique du programme doivent être claire).
- Des réponses longues ne sont pas nécessaires. La plupart des questions sont conçues pour une réponse brève, de l'ordre d'un paragraphe.
- Document autorisé: polycopié du cours.

Les barèmes sont donnés à titre indicatif.

1 Échauffement (2 points)

Dans le projet, vous avez utilisé les appels système fork() et execve() (ou l'une des variantes de la famille execvp().

1. (1 point) Est ce qu'il est possible qu'un appel à fork() ne réussisse pas ? Si un tel appel ne peut échouer, expliquer pourquoi et si un tel appel peut échouer, donnez un exemple d'un cas d'échec.
2. (1 point) Est ce qu'il est possible qu'un appel à execve() (execvp()) ne réussisse pas ? Si un tel appel ne peut échouer, expliquer pourquoi et si un tel appel peut échouer, donnez un exemple d'un cas d'échec.

2 Gestion de la mémoire (3 points)

A. Une décision clef dans la conception d'un système d'exploitation est le choix entre la pagination et la segmentation. Comparez ces deux approches, en identifiant les forces et les faiblesses de chacune.

B. On considère une politique de remplacement de pages dans un système purement de page-à-la-demande. La taille de la mémoire est de 3 pages. Un processus qui vient d'être créé demande des pages. Ces pages désignées par A, B, C, D, E sont initialement en mémoire secondaire. La séquence de pages demandés par le processus est

A B C B A D A B C D A B A C B D

B1. Combien de fautes de pages sont produites par un algorithme de remplacement FIFO ? Est-ce qu'un tel algorithme est facile à implémenter ?

B2. Mêmes questions dans le cas d'un algorithme de remplacement LRU.

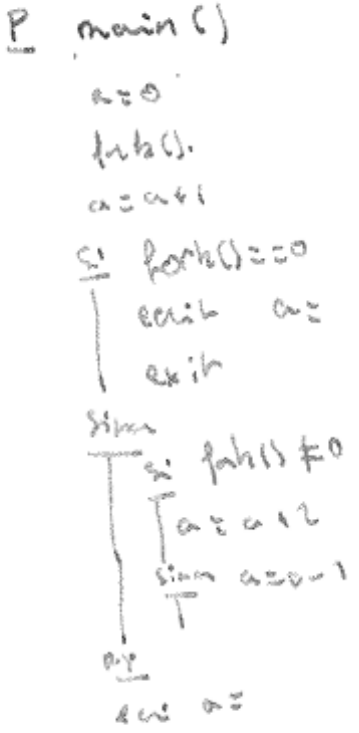
B3. Mêmes questions dans le cas d'un algorithme de remplacement optimal (Belady).

3 Fork() (3 points)

Soit le programme:

```
 1: int main(void)
 2: {
 3:   int a=0;

 4:   fork();
 5:   a = a+1;
 6:   if (fork() == 0) {
 7:     printf(" a = %d\n", a);
 8:     exit(0);
 9:   } else {
10:     if (fork() != 0)
11:       a = a+2;
12:     else
13:       a = a-1;
14:   }
15:   printf("a= %d\n", a);
16:
17:   while (wait(NULL) >=0);
18:   return 0;
19: }
```



1. Combien de processus sont-ils créés à l'exécution du programme (on supposera qu'aucun appel n'échoue) ?
2. Dessinez l'arbre de création des processus.
3. Décrivez le fonctionnement de ce programme. En particulier, à quoi sert la ligne 17?
4. L'affichage produit par ce programme est-il déterministe ? Que la réponse soit oui ou non, décrivez l'affichage produit par ce programme.
5. Quelle est (sont) la (les) valeur(s) de "a" affichée(s) par chacun des processus ?

4 La communication par tubes (4 point)

Vous voulez écrire un programme qui génère une série de nombres aléatoires, et affiche la sortie en classant ces nombres. Au lieu de classer ces nombres vous mêmes, vous souhaitez utiliser la commande "/usr/bin/sort -n" pour faire le tri (le programme sort fait un tri, le "-n" dit au sort de faire le tri numériquement au lieu de le faire de manière alphabétique. Pour vous faciliter la tâche, voici un début de code.

```
 1: int main(int argc, char **argv) {
 2:   int n;
 3:   int i;
 4:   if (argc==1) n=100000;
 5:   else n = atoi(argv[1]);
10:   /* Votre code doit être placé ici.
20:      Faites que le printf de la ligne 51 envoie sa
30:      sortie à  ``sort -n''
40:   */
50:   for (i=0;i<n;i++) {
51:     printf("%lu\n",random());
52:   }
53:   return(0);
54: }
```

1. Expliquez votre démarche, en vous référant au projet. En particulier, quelle partie du projet était similaire à ce que vous voulez faire maintenant ?

2. Est-il possible de faire ce que vous souhaitez avec un seul processus et un tube pour que le "printf" (ligne 51) envoie sa sortie au "sort -n" ? avec deux processus ? avec trois processus ou plus ? Vous expliquerez les avantages et les inconvénients de chacune des approches.

3. Écrivez le code qui doit aller entre les lignes 10 à 40 de telle sorte à renvoyer l'affichage de printf de la ligne 51 sur "/usr/bin/sort -n". Ne changez rien à ce qui est déjà écrit !

5 Synchronisation (4 points)

On considère 3 processus P1, P2, P3 exécutant les codes suivants :

```
mutex  A,B,C ;
```

process P1	process P2	process P3
{ ... P(A) ; P(B) ; ...; V(A) ; V(B) ; }	{ ... P(B) ; P(C) ; ...; V(B) ;V(C) ; }	{ ... P(C) ; P(A) ; ...; V(C) ; V(A) ; }

Les sémaphores A,B,C sont des sémaphores d'exclusion mutuelle.

- On souhaite que chacun des processus P1, P2 et P3 exécute son code jusqu'à la fin. Est-ce toujours le cas ? Si oui démontrez le. Sinon, décrivez (en vous appuyant sur un schéma) une situation problématique puis, en supposant que les processus ont toujours besoin d'acquérir les mêmes ressources critiques protégées par les sémaphores A,B,C, modifiez le code ci-dessus afin d'obtenir la garantie souhaitée.

6 Processus, tubes et signaux (4 points)

Considérez le code suivant, dont on suppose que tous les appels systèmes fonctionnent sans erreur :

```
01: int d;
02: void nain1(int sig) {
03:   printf("he hi!\n");
```

```
04:    kill(d,SIGUSR1);
05: }
06: void nain2(int sig) {
07:    printf("he ho!\n");
08:    kill(getppid(),SIGUSR1);
09: }
10: main() {
11:    int i; int fd;
12:    int pid, premier_proc;
13:    mkfifo("tube1", 0666);
14:    premier_proc = getpid();
15:    signal(SIGUSR1,nain1);
16:    for (i=0; i <= 2; i++) {
17:      pid = fork();
18:      if (pid > 0){
19:        break; // sortir de la boucle
20:      }
21:      else
22:        signal(SIGUSR1, nain2);
23:      if (i == 2) {
24:        fd = open("tube1", O_WRONLY);
25:        pid = getpid();
26:        write(fd,&pid,sizeof(int));
27:      }
28:    }
29:    if (getpid() == premier_proc) {
30:      fd = open("tube1", O_RDONLY);
31:      read(fd, &d,sizeof(int));
32:      kill(d,SIGUSR1);
33:    }
34:    for (;;); // boucle infinie
35:    exit(0);
36: }
```

Expliquez de manière claire et concise ce que fait ce programme. Donnez un exemple de ce qui est affiché lorsqu'on exécute ce programme.

ESIEE ENGINEERING

une école de la Chambre de commerce et d'industrie de Paris

13,5/20

Nom François LAHARRAGUE

Classe I3r N° table P6 T2

Le 4 février 2010

Composition de IN3S T01 page 1

I. ECHAUFFEMENT

① Oui un appel fork() peut échouer.
Il retourne -1 en cas d'échec dû au nombre de processus usager ou système excédé.
• causes : - pas suffisament d'espace mémoire
- nombre maximal de créations autorisé atteint

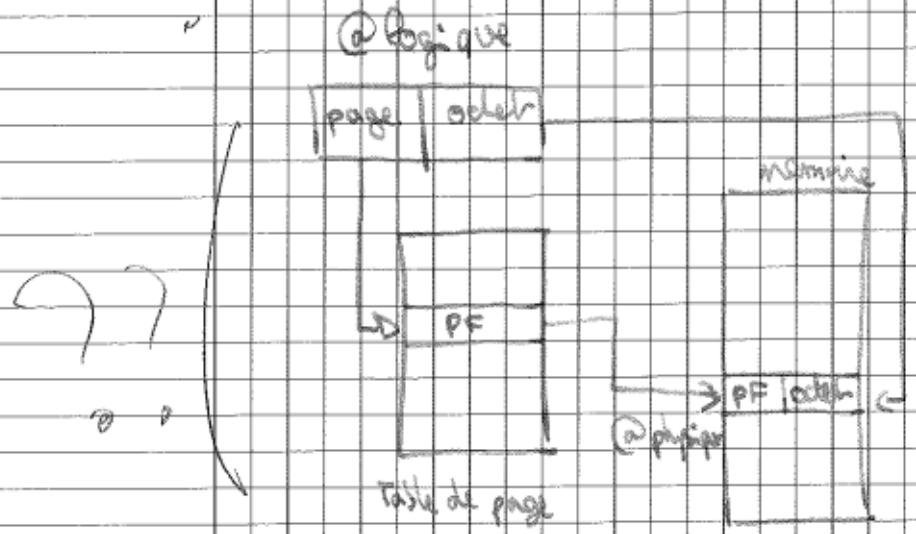
② Oui un appel à execve() peut échouer
car après une exec() primaire même du processus est écrasé par la nouvelle image mémoire du prog. exécutable, mais le pid ne change pas. Le contenu du contexte utilisateur qui existait avant l'appel exec() n'est plus accessible.

II GESTION DE LA MEMOIRE

Ⓐ ≠ entre pagination et segmentation :

*Pagination : la mem. virtuelle et la mem. physique st structurées en unités d'allocations appellés pages (pour mV) et cadres (pr m physique). De taille entre 2 Ko et 16 Ko (souvent 4 Ko). Il n'y a pas de fragmentation interne

car toutes les pages sont de même taille. Par contre il peut y avoir 1 fragmentation interne si la dernière page de l'espace d'adressage logique n'est pas pleine.



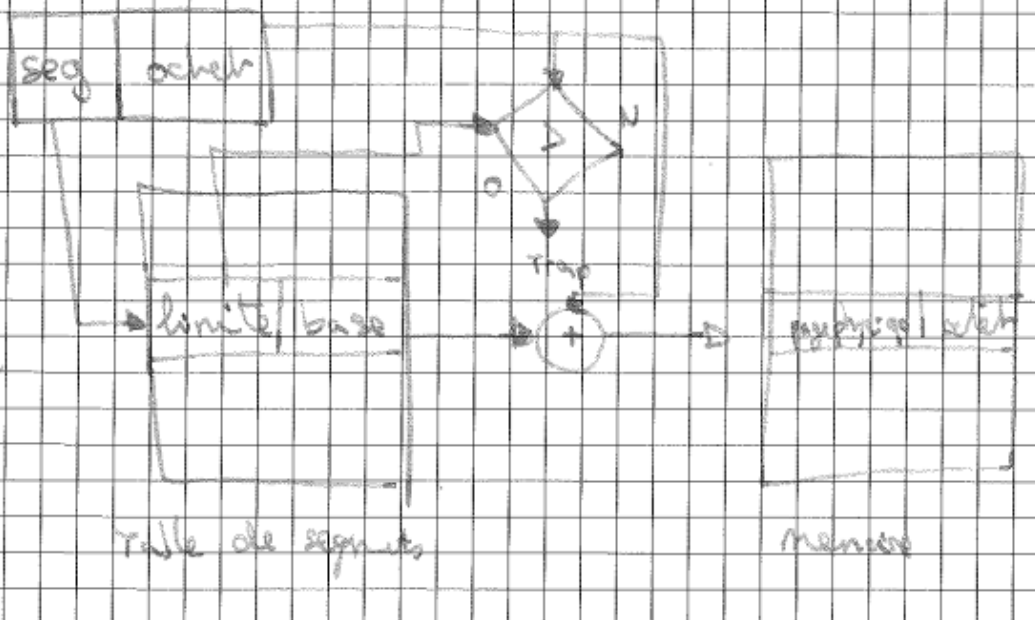
• Segmentation: Dans un système paginé l'espace d'adressage virtuel d'un processus est à 1 dimension. Or en général, 1 processus est composé de plusieurs unités logiques: - 1 code
- données initialisées
- données non init
- pile d'exécution.

Donc ici l'espace d'adressage est à 1 dim.

De taille différente pour ce segments (= fragmentation externe).

La segmentation facilite l'édition de liens, ainsi que le partage entre processus de segments de données ou de code.

@logique



?? ~~

B

B1 : FIFO

t / w	A	B	C	B	A	D	A	B	C	D	A	B	A	C	B	D
0	(A)	A	A	A	A	(B)	D	D	(C)	C	C	(B)	B	B	B	B
1		(B)	B	B	B	B	(A)	A	A	(D)	D	D	D	(C)	C	C
2			(C)	C	C	C	C	(B)	B	B	(A)	A	A	A	A	(D)

OK ✓

Les pages sont mémorisées en mémoire (FIFO : on sort ce qui est rentré le plus anciennement) lorsqu'il y a un défaut de page : la plus ancienne est remplacée, elle se retrouve en tête de liste

12 défauts de page. Facile à implanter (dite FIFO), mais nul (+) Trop de défauts de page

OK

B2 : LRU

t / w	A	B	C	B	A	D	A	B	C	D	A	B	A	C	B	D
0	(A)	A	A	A	A	A	A	A	A	(D)	D	D	D	(C)	C	C
1		(B)	B	B	B	B	B	B	B	(A)	A	A	A	A	(D)
2			(C)	C	C	(D)	D	D	(C)	C	C	(B)	B	B	B	B

OK

10 défauts de pages.

Les pages en mémoire st mémorisé ds 1 liste chaînée. La page la moins récemment utilisée est en queue de liste et sera retirée.

⇒ Dure à implanter (besoin d'un support hardware). Il faut 1 manière de mémo le tps à chaque fois qu'1 page est référencée. On peut aussi utiliser 1 technique de vieillissement où 1 registre de n bits est associé à chaque page. Le bit le plus significatif est mis à 1 chaque fois que la page est référencée. Régulièrement, on décale vers la droite les bits de ce registre. Lorsqu'on doit expulser une page, on choisit celle dont la valeur est la plus petite. On pourrait aussi mettre une page au dessus d'1 pile chaque fois qu'elle est réf. On expulsera la page au fond de la pile.

(B3): BELADY.

A	B	C	B	A	D	A	B	C	D	A	B	A	C	B	D
(A)	A	A	A	A	A	A	A	A	A	A	A	A	A	A	
	(B)	B	B	B	B	B	B	(C)	C	C	C	C	C	C	
		(C)	C	C	(D)	D	D	D	D	D	(B)	B	B	B	

on sait pas (dépend du futur)

On retire la page qui sera réf le plus tard possible

⇒ Impossible à mettre en œuvre : difficile de prévoir les références d'un prog. Sert de réf comme

ESIEE ENGINEERING

une école de la Chambre de commerce et d'industrie de Paris

Nom François Lahaneque

Classe I3T

Le 4/02 20 10

Composition d IN... (2) Page 2

défauts de page mini.

7 défauts de page.

2,5/3

③ FOR i<1

① NB: fork() renvoie:

- -1 si erreur
- =0 si fils

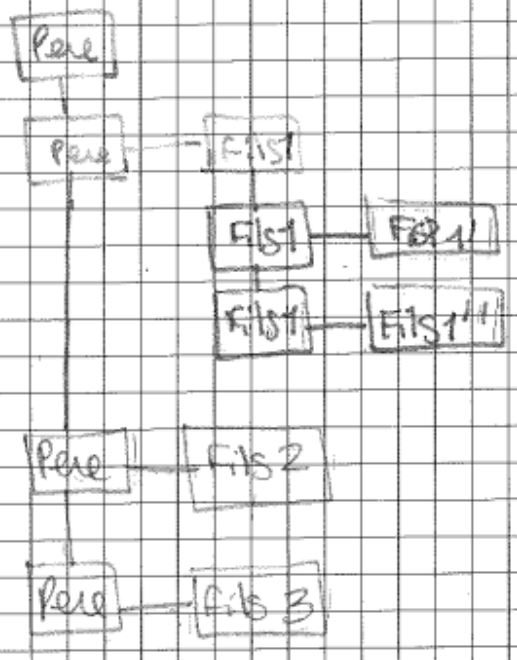
Déroulement:

- le père crée 1 fils avec fork().
- le père crée 1 deuxième fils avec if (fork()==0)
- le père crée 1 3ème fils avec if (fork()!=0)
- le premier fils crée 2 fils.

OK

On a donc 6 processus en total.

2)



OK

On initialise a à 0

3) * le père crée 1 fils. (= Fils 1).
a = a+1 (a = 1)

• Fils 1 créé 1 fils (= Fils 1')
fils 1' affiche a et s'arrête affiche 1.

• Fils 1 crée 1 autre fils (= Fils 1'')
a = a+2 si processus = Fils 1
a = a-1 si processus = Fils 1''

• Fils 1'' affiche a

• Fils 1 affiche a

* le père crée fils 2
• fils 2 affiche a et s'arrête

* le père crée fils 3
• a = a+2 si processus = père

• a = a-1 si processus = fils 3.

• père affiche a
fils 3 affiche a-

3) <u>La ligne 17:</u>

Wait(1) permet la synchro père avec ses fils.
pid_t wait (int *status);
status contient le code de sortie de terminaison du processus fils.
valeur retournée: pid du premier fils terminé
-1 si pas de fils.

Donc while (wait (NULL) >= 0) veut dire qu'on continue tant qu'on a pas d'erreur

Ntn ??

④ • Le programme n'affichera pas toujours la même chose car cela dépend du séquenceur : et à qui il donne la priorité. De plus on a vu auparavant le piège que l'affichage sur le shell est asynchrone du comportement réel du programme donc c'est non déterministe.

OK

• Déjà décrit dans ③

⑤ • fils 1 affiche a = 1
• fils 1 part a = a-1 = 1-1 = 0
• fils 1 affiche a+a = 1+2 = 3

- fils 2 affiche a et soit a = 1
- fils 3 affiche a = a - 1 = 1 - 1 = 0
- père affiche a = a + 2 = 1 + 2 = 3

4 Communication par tubes.

① Ça ressemble à quand on prenait 1 n° de clé et on un montant aléatoirement dans 1 fichier rsk (depuis terminal) et qu'on l'envoyait à acquisition

- on va tirer 1 nombre avec random
- créer 1 tube
- écrire dedans
- et soit va lire dans le tube

NB: Faire attention à l'ordre d'écriture et lecture

② Le problème de faire avec plusieurs processus c'est qu'on peut avoir un interblocage.

Comment? Sinon oui c'est possible de le faire avec 1, 2, 3 voire plus de processus.

A garder qu'on voit on gagne en mémoire et en facilité de programmation et en complexité.

ESIEE ENGINEERING

une école de la Chambre de commerce et d'industrie de Paris

Nom François LAHARRAGUE

Classe E3T

Le 4/02 20 10

Composition de INF3510 : Page 3

(3)

```
if(fork() == -1) perror("erreur creation du f(s");
else
{
  sprintf( read, %d, pipe[R])
  sprintf( write, %d, pipe[W])
  erreur_exe = execlp("/usr/bin/sshd-n", "sshd -n",
  read, write, NULL);
  if( erreur_exe == -1) {printf("erreur exe");
  exit (EXIT SUCCESS); }
}
```

N°2 write(pipe[W], random() , strlen(random()));

NB : Il faut déclarer int pipe[2];

```
#define R 0
#define W 1
char *read= malloc[5* sizeof(char)];
char *write=malloc[5*sizeof(char)];
```

2/4

EXERCICE 6 : PROCESSUS, TUBES et SIGNAUX

déf :

- SIGUSR1 : numéro 10

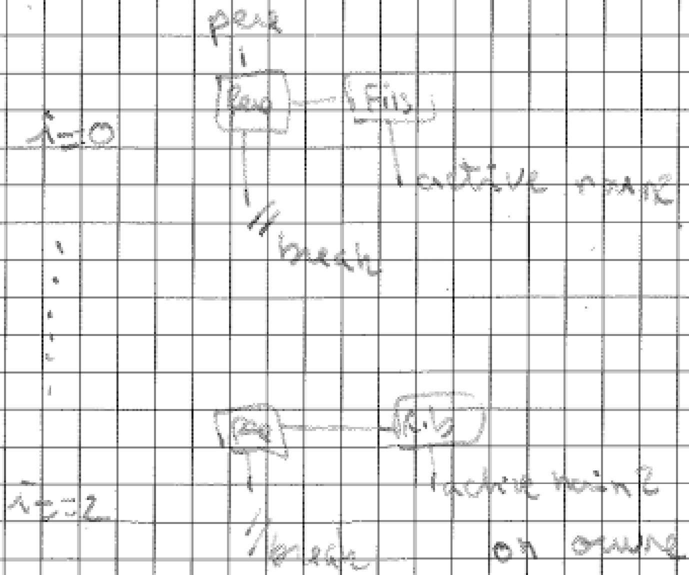
  type A : terminaison du processus

Au début le "nain 1" écrit : "he hi!"
Puis il tue le [illegible] [illegible] kill.

le "nain 2" [illegible] "[illegible]!"
et tue le process.

main()

- on crée une pile (first in first out) avec mkfifo. OK
- premier proc reçoit l'utilisation du pid du processus.
- avec le signal on active nain 1
- on fait 3 fils. OK



on ouvre le tube 1
avec le descripteur
de fichier et on écrit le pid
du fils dedans

if (getpid() == pid_pere) ⇒ si je suis le père

on ouvre le tube et on lit

on lit le nombre

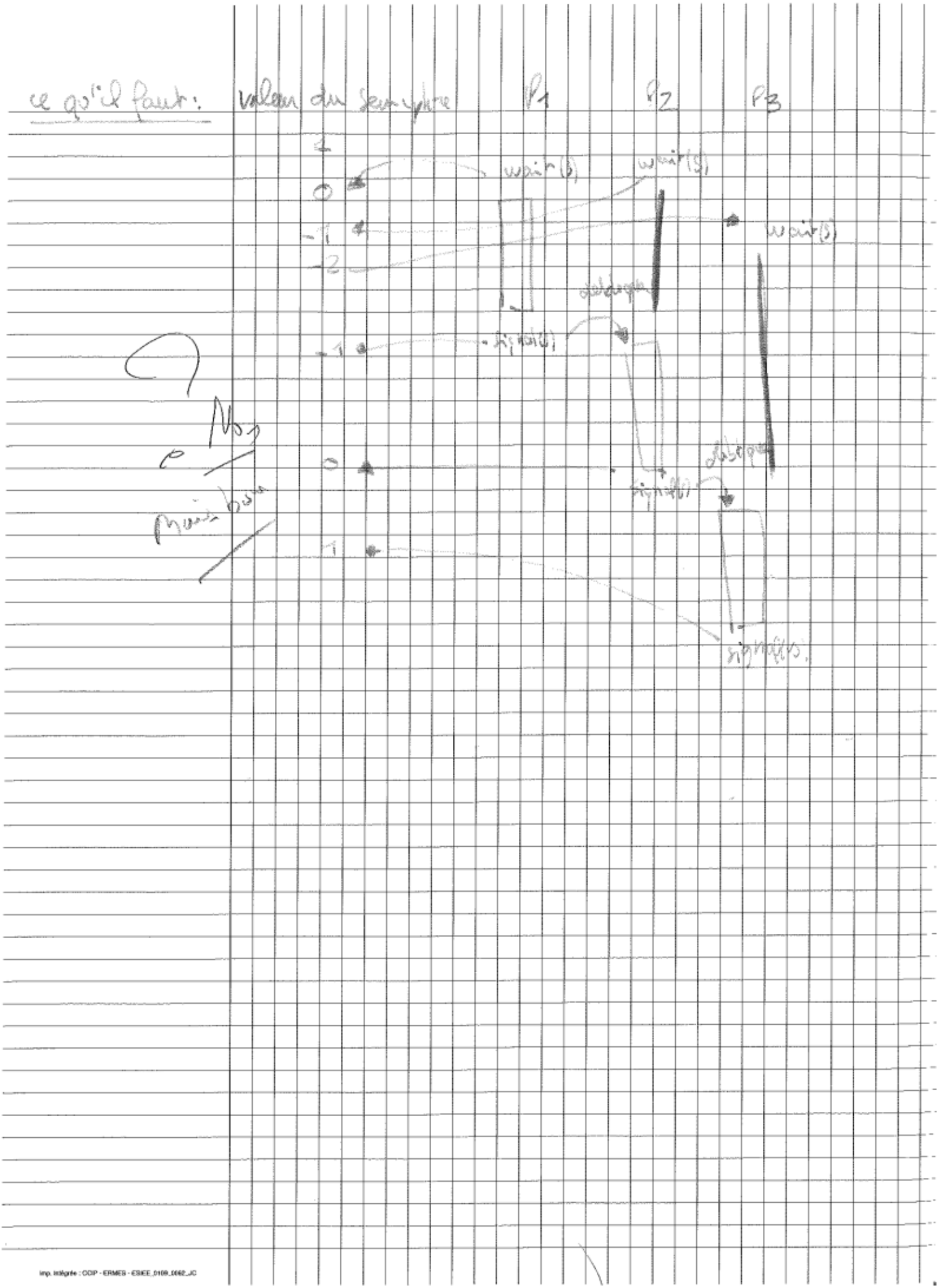
le met en boucle de

AFFICHAGE:

He hi!
He ho!
He ho!
He ho!
He hi!
He ho!
He ho!
He ho!
He hi!

5 SYNCHRONISATION.

Ce n'est pas toujours le cas : risque d'interblocage ⇒ problème des philosophes avec seulement 3 philosophes, chacun a besoin de 2 fourchettes pour manger ⇒ alternance penser/manger.



Final de Février 2008 :

Enoncé

Instructions

- Ne vous contentez pas d'écrire un résultat. Justifiez et commentez vos réponses. Aucun point ne sera donné pour une réponse non justifiée ou une justification erronée.
- Pour les réponses qui demandent des programmes ou des morceaux de programmes, vous pouvez utiliser du C ou n'importe quel style de pseudo-code suffisement raisonnable (i.e., la syntaxe et la sémantique du programme doivent être claire).
- Des réponses longues ne sont pas nécessaires. La plupart des questions sont conçues pour une réponse brève, de l'ordre d'un paragraphe.
- Document autorisé: polycopié du cours.

Les barèmes sont donnés à titre indicatif.

1 Echauffement (2 points)

1. Question bonus (0.5 points): A qui donc est ce la faute ?
2. (1 point) Deux processus ouvrent un même fichier physique en utilisant deux chemins d'accès différents (fichier avec deux liens physiques). Est-ce que les deux processus partagent le même pointeur de fichier ? Justifiez.
3. (1 point) Quelle est la différence entre un fichier ouvert avant le fork et un fichier ouvert après le fork par un processus père et un processus fils ?

2 Gestion de la mémoire (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 pages :

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).

3 Fork() (2 points)

Soit le programme:

```
fork(); if (fork()) fork();
```

1. Combien de processus sont-ils créés à l'exécution du programme (on supposera qu'aucun appel n'échoue) ?
2. Dessinez l'arbre de création des processus

4 La communication par tubes (4 point)

Soit le programme suivant :

```
 1: #include <unistd.h>
 2: #include <stdlib.h>
 4: int main ( void )
 5: {
 6:   int p [2];
 8:   pipe (p);
10:   if( fork ())
11:   {
12:     close (STDIN_FILENO );
13:     dup2(p[0], STDIN_FILENO);
14:     close (p[0]);
16:     execlp ("wc", "wc", "-l", NULL );
17:   }
18:   else
19:   {
20:     close ( STDOUT_FILENO );
21:     dup2(p[1], STDOUT_FILENO);
22:     close (p[1]);
24:     execlp ("ls", "ls", "-l", NULL );

25:   }
26:   exit ( EXIT_FAILURE );
27: }
```

Ce programme contient un erreur de traitement des tubes.

1. (1 point) Que devrait faire le programme ? On rappelle que "ls -l" affiche sur la sortie standard la liste des fichiers du répertoire courant, et que "wc -l" compte le nombre de lignes de l'entrée standard.
2. (1 point) Que se passe-t'il à l'exécution ?
3. (1 point) Expliquez la raison du comportement inattendu du programme.
4. (1 point) Corrigez le programme par conséquent.

5 Fork, exec et signal (4 points)

Soient les deux programmes suivant p1 et p2. Le programme p1 est le suivant :

```
1: /* Programme p1.c */
2: void ma_fonction(int signal_recu) {
3:   printf("J'ai bien recu le signal %d\n",
           signal_recu);
4:   return;
5: }
6: int main(int argc,char *argv[]) {
7:   pid_t pid; int retour;
8:
9:   signal(SIGUSR1,ma_fonction);
10:  switch (pid=fork()) {
11:    case -1: fprintf(stderr,"Rate\n"); exit(EXIT_FAILURE);
12:    case 0:  execlp("./p2","p2",NULL);
                fprintf(stderr,"Pile rate\n"); exit(EXIT_FAILURE);
13:    default: kill(getpid(),SIGUSR1); kill(pid,SIGUSR1);
                kill(pid,SIGTERM); wait(&retour);
14:  }
15:  if (WIFEXITED(retour)) {
16:    printf("Le fils a pour code de retour %d\n",
               WEXITSTATUS(retour));
17:  } else if (WIFSIGNALED(retour)) {
18:    printf("Le fils est mort apres avoir recu %d\n",
               WTERMSIG(retour));
19:  } else {
20:    printf("Le fils a ete suspendu apres avoir recu %d\n",
               WSTOPSIG(retour));
21:  }
22:  exit(EXIT_SUCCESS);
23: }
```

et le programme p2:

```
1: /* Programme p2.c */
2: #include <stdlib.h>
3: #include <stdio.h>
4:
5: void ma_fonction(int signal_recu) {
6:   printf("J'ai bien recu le signal %d\n",
           signal_recu);
7:   return;
8: }
9:
10: int main(int argc,char *argv[]) {
11:   signal(SIGUSR1,ma_fonction);
12:   while (1) pause();
13:   exit(EXIT_SUCCESS);
14: }
```

- (1 point) Expliquez le fonctionnement de ces programmes si tout se passe bien.
- (0.5 point) Si p2 n'est pas accessible quel est le résultat produit ?
- (0.5 point) Dans le programme p2, les lignes 5 à 8 sont les mêmes que les lignes 2 à 5 du programme p1. Ces lignes de p2 sont elles nécessaires ?
- (0.5 point) Dans le programme p2, la ligne 11 est-elle nécessaire ?
- (1 point) Ces programmes donnent-ils toujours le résultat escompté ? Si vous pensez que oui, vous expliquerez pourquoi. Si vous pensez que non, vous corrigerez le problème (en expliquant).
- (0.5 point) Quelle(s) est/sont la/les sortie(s) produite(s) ?

6 Synchronisation (5 points)

Dans cet exercice, il s'agit d'utiliser des sémaphores pour synchroniser l'implémentation des tubes au sein d'un système d'exploitation.

On dispose d'un type pipe_t qui peut contenir MAX_PIPE octets au maximum, et des opérations __write_char et __read_char qui permettent respectivement d'insérer ou d'extraire un caractère dans/depuis un tube :

```
typedef struct {
  ???
} pipe_t;
void __write_char(pipe_t *p, char c);
void __read_char(pipe_t *p, char *c);
```

Les primitives __write_char et __read_char ne contiennent pas de code de synchronisation ; l'exécution de __write_char provoque une erreur si le tube est plein, l'exécution de __read_char provoque une erreur si le tube est vide, et les appels concurrents à __write_char (resp. __read_char) ne sont pas supportés. Par contre, __write_char et __read_char peuvent s'exécuter en parallèle si le tube n'est ni vide ni plein. La structure pipe_t contient les champs nécessaires à la gestion interne du tube ; vous n'avez pas besoin de connaître ces champs mais vous pouvez bien sûr en ajouter de nouveaux...

1. En utilisant des sémaphores (n'oubliez pas d'indiquer leur valeur initiale), donnez le code des fonctions write_char et read_char, versions synchronisées respectives des fonctions __write_char et __read_char.
2. On souhaite maintenant disposer d'une primitive write permettant d'écrire plusieurs caractères dans le tube. Voici une première ébauche de cette fonction :

```
int write(pipe_t *p, char *buf, unsigned len)
{
  unsigned i;
  for(i=0; i<len; i++)
    write_char(p, buf[i]);
}
```

On voudrait que l'exécution de la fonction write soit atomique, c'est-à-dire que les caractères injectés dans le tube ne se mélangent pas avec d'autres exécutions de write (on suppose que la fonction write_char n'est plus utilisée directement par les processus). Donnez le nouveau code de write.

3. En pratique, les systèmes ne garantissent l'atomicité des opérations d'écritures dans les tubes que jusqu'à une certaine taille (appelons cette constante MAX_ATOMIC). Lorsqu'un appel à write porte sur une quantité de données supérieure à MAX_ATOMIC, l'opération peut-être vue comme une séquence d'écritures atomiques de MAX_ATOMIC octets (sauf pour la dernière séquence qui peut bien sûr porter sur un nombre d'octets inférieur). Entre ces séquences, il est donc possible que s'insèrent des écritures effectuées par d'autres processus... Donnez une nouvelle version de la fonction write fonctionnant selon ce principe.

4. Il est parfois nécessaire de pouvoir privilégier l'exécution de certains processus, comme par exemple dans les systèmes "critiques". Ainsi, on voudrait disposer d'une fonction write_char_urgent, de même profil que write_char, qui permettrait d'accéder au tube en écriture de manière ultra privilégiée par rapport à write_ char[1]. Pour simplifier, on suppose qu'il n'y aura jamais deux appels à write_char_urgent en parallèle. Par exemple, lorsqu'un tube est plein et que plusieurs processus sont bloqués en attendant de pouvoir y insérer des caractères, si l'un d'entre eux est bloqué dans write_char_urgent, alors le prochain caractère écrit dans le tube doit être le sien... Donnez le code de write_char_urgent ainsi que la nouvelle version de write_char.

IN3ST01	Systèmes d'exploitation - UNIX	Date MAJ : 20/07/2006

Statut : Obligatoire ESIEE 3e année S 1er semestre ESIEE 3e année T 1er semestre

Horaires : Cours : 16 h Travaux dirigés : 4 h Travaux pratiques : 15 h Projet : 15 h

Langue(s) de l'unité enseignée : FRANCAISE **Crédits ECTS :** 2.5

Responsable(s) : NAJMAN Laurent (najmanl@esiee.fr)

Objectifs :

Le but du cours est de présenter les principes de fonctionnement des systèmes d'exploitation.
L'objectif du cours est que l'étudiant comprenne les concepts fondamentaux : gestion des fichiers, gestion de la mémoire, gestion des processus, gestion des entrées-sorties. Ces concepts seront illustrés en particulier sur les exemples de Unix, Linux et Windows NT/2000/XP.

Pré-requis :

- Connaissance de l'utilisation d'Unix est nécessaire. Les 5 premières heures (2h cours + 3h TP) seront consacrées à une initiation à l'utilisation 'utilisateur avancé' d'Unix.
- Connaissance d'un langage de programation de 'haut-niveau' (C, C++ ou Java) .

Contenu et planning des enseignements	C	T.D	T.P	P
Introduction à Unix	2.00		3.00	
Introduction aux Sytèmes d'exploitation	2.00			
Processus	2.00			
Threads	2.00			
Communication entre processus	2.00			
Synchronisation des processus	2.00			
Gestion de la mémoire	2.00			
Gestion des entrées sorties et des fichiers	2.00			
TD Synchronisation		2.00		
TD consacré au projet		2.00		
TP Processus			3.00	
TP Synchronisation			3.00	
TPs consacré au projet			6.00	
Heures de projet encadré				15.00

Nature de l'évaluation	Commentaire	Durée	Coef.
Rapports de TP			1.00
Examen écrit			2.00

Bibliographie :

[1] JM.Rifflet, *La programmation sous UNIX,* Ediscience International

[2] A. Tanenbaum, *Systèmes d'exploitation (systèmes centralisés, systèmes distribués),* Interedition

[3] *The Cathedral and the Bazaar,* Livre électronique
www.tuxedo.org/%7Eesr/writings/cathedral-bazaar/

[4] *L'histoire d'Unix et ses dérivées impressionnant graphe d'héritage*
perso.wanadoo.fr/levenez/unix/

[5] Bart Lamiroy, Mines de Nancy, *Polycopié du cours*
http://www.mines.inpl-nancy.fr/~lamiroy/ENSEIGNEMENT/SI132/poly.pdf

Moyens pédagogiques particuliers :

Afin de s'éviter de perdre trop de temps lors des TP, l'étudiant est invité à suivre les liens suivants :
<http://www.esiee.fr/~info/unix/> (Station HP sous Unix avec tcsh, en français)
<http://www-106.ibm.com/developerworks/library/l-bash.html> (programmation sous Bash).

IN 3504 —

Contrôle système UNIX
Durée 2 heures, Tous documents autorisés.

Le contrôle étant long, la notation se fera sur 24 points.

4 points **I.** Donnez les noms des trois mécanismes de communication interprocessus (IPC). Expliquez brièvement les fonctionnalités de chacun d'eux.

4 points **II.** Donnez les prototypes (déclarations) de la fonction `fwrite` (bibliothèque d'entrée-sortie standard) et de la fonction `write` (appel système). Décrivez les paramètres et la valeur de retour de chaque fonction. Explicitez les différences entre ces deux fonctions.

4 points **III.** Que fait ce programme ?

```
#include <stdio.h>
char line[80];

void prompt(void)     {
  fprintf(stderr,"$ ");
}
main() {
  int status, pid;
  prompt();
  while (fgets(line,sizeof(line)-1,stdin) != NULL) {
    if ((pid = fork()) < 0) {
      fprintf(stderr,"Echec de fork()\n");
      exit(1);
    }
    else
      if (pid == 0) {
        line[strlen(line)-1] = 0;
        execlp(line,line,0);
        fprintf(stderr,"Echec d'exec()\n");
      }
      else {
        wait(&status);
      }
    prompt();
  }
}
```

4 points **IV.** Que fait ce shell script?

Note : Le filtre `tr` (`tr ch1 ch2`) lit l'entrée standard et l'affiche sur la sortie standard remplaçant chaque caractère de la chaîne1 par le caractère de même rang dans la chaîne2.

```
DIR=$1
for nom in 'ls $DIR'
  do
    nouveau='echo $nom | tr A-Z a-z'
    mv $DIR/$nom $DIR/$nouveau
done;
```

4 points **V.** Soit le programme suivant :

```
#include <stdio.h>
#include <unistd.h>
int main() {
  int i;
  pid_t n;
  printf("Début : pid=%d\n",getpid());
  for (i=0;i<2;++i) {
    if ((n=fork())!=0)
      waitpid(n,NULL,0);
  }
  printf("Bonjour, je suis %d de père %d\n",getpid(), getppid());
  return 0;
}
```

Combien de fois la phrase `Début : pid=...` s'affichera ?
Combien de fois la phrase `Bonjour, je suis ...` s'affichera ?
En supposant que le processus de départ soit de *pid* 100 et de père de *pid* 50, et que les *pid* soient attribués de 1 en 1 (i.e. le premier fils sera de numéro 101), donner la sortie du programme.

4 points **VI.** Écrivez un programme C qui prend comme argument un nom de fichier (on suppose qu'il exis et affiche les informations sur ce fichier (n° d'inode, type, droits d'accès, propriétaire, taille, dates)

Examen IN3ST01 - Systèmes d'Exploitation

Responsable: Laurent Najman
24 février 2006
Durée: 3 heures

Instructions

- Ne vous contentez pas d'écrire un résultat. Justifiez et commentez vos réponses. Aucun point ne sera donné pour une réponse non justifiée ou une justification erronée.
- Pour les réponses qui demandent des programmes ou des morceaux de programmes, vous pouvez utilisez du C ou n'importe quel style de pseudocode suffisament raisonable (*i.e.*, la syntaxe et la sémantique du programme doivent être claire).
- Des réponses longues ne sont pas nécessaires. La plupart des questions sont conçues pour une réponse brêve, de l'ordre d'un paragraphe.
- Document autorisé: polycopié du cours.

Les barèmes sont donnés à titre indicatif.

1 Mémoire (3 points)

Supposons que nous ayons un programme utilisant 5 pages (de 0 à 4), et auquel le système d'exploitation a alloué 4 cadres de page. Le programme référence les pages dans l'ordre 0, 1, 4, 2, 1, 0, 3, 0, 1, 4. Donnez, pour les algorithmes FIFO, LRU et Belady, les pages présentes en mémoire physique aux différents instants de la séquence, et indiquez le nombre de défauts de page.

2 Sémaphores (4 points)

La laverie du coin vient d'entrer dans l'ère de l'informatique. Lorsqu'un client entre, il ou elle tape sur le clavier d'un ordinateur le nombre de machines à laver dont il ou elle a besoin. Plusieurs ordinateurs sont disponibles pour faire cette manipulation. Ces ordinateurs sont reliés à un ordinateur central qui assigne automatiquement les machines à laver disponibles. Le client dépose son linge à laver dans les machines indiquées. Lorsqu'une machine à laver finit son cycle, elle informe l'ordinateur central qu'elle est de nouveau disponible.

L'ordinateur central maintient un tableau de booléens `disponible[NMACHINES]` qui indique si la machine à laver correspondante est disponible (`NMACHINES` est une constante qui indique le nombre de machines à laver dans la laverie). Voici le code pour allouer et relacher une machine ; le tableau `available` est initialisé à `VRAI` et `nlibre` est un sémaphore initialisé à `NMACHINES`.

```
int alloue() /* Retourne l'indexe d'une machine disponible. */
{
  sem_wait(nlibre); /* Attend qu'une machine soit disponible */
  for (int i=0; i < NMACHINES; i++)
    if (disponible[i] == VRAI)
    {
      disponible[i] = FAUX;
      return i;
    }
}

void relache(int machine) /* Libère une machine */
{
  disponible[machine] = VRAI;
  sem_post(nlibre);
}
```

Est ce que ce code fonctionne correctement ? En particulier, vous détaillerez ce qui peut se passer lorsque deux clients demandent en même temps une machine à laver, et vous montrerez si un problème peut se produire.

Dans le cas où il n'y a pas de problème, vous donnerez des arguments convainquants pour le justifier, éventuellement en faisant référence au cours. Dans le cas où il y a un problème, vous modifierez le code en conséquence pour corriger le problème, et donnerez les arguments nécéssaires pour convaincre qu'il n'y a plus de problème (éventuellement en faisant référence au cours).

3 tubes et sémaphores (4 points)

Les anciens systèmes Unix ne fournissaient pas de primitives pour la synchronisation des processus utilisateurs, mais ils fournissaient les tubes (`pipe`). Un tube est un canal pour transmettre des données, qui a une taille maximale (souvent 4Ko). Lire un tube bloque jusqu'à ce qu'une donnée soit écrite. Ecrire dans un tube dont le buffer est plein bloque jusqu'à ce qu'une donnée soit lue.

- Décrivez comment implémenter un sémaphore en utilisant un tube. (Conseil : tirez partie de la sémantique bloquante du tube. Faites simple). Vous écrirez du code pour les fonctions `sem_wait` et `sem_post`.
- Un sémaphore implémenté par tube peut conduire à des interbloquage dans des situations ou un sémaphore conventionnel fonctionnerait sans problème. Décrivez une telle situation.

4 Fork (4 points)

Considérez le code suivant :

```
int main () {
```

```
  if (fork() == 0) {
    if (fork() == 0) {
      pid_t pid; int status;
      pid = wait(&status);
      printf("1");
    } else {
      pid_t pid; int status;
      if ((pid = wait(&status)) > 0) {
        printf("2");
      }
    }
  } else {
    printf("3");
    exit(0);
  }
  printf("4");
  return 0;
}
```

Pour des raisons de place, le code d'erreur des fonctions ne sont pas vérifiés, aussi ont supposera que toutes les fonctions marchent correctement.

1. Décrivez le déroulement de ce programme.
2. Ce programme produit-il toujours le même affichage ?
3. Existe-t-il des lignes de code que vous pourriez supprimez sans changer le comportement du programme ?
4. Décrivez la (ou les) sortie(s) possible(s) de ce programme.

5 Signaux (5 points)

Considérez le code suivant :

```
int compteur, pid1, pid2;

void gest_pere(int sig) {
  if (compteur == 3) exit(0);
  compteur++;
  if (compteur % 2 == 1)   kill(pid2,SIGUSR2);
  else  kill(pid1,SIGUSR1);
}
void gest1(int sig) {
  printf("Oui\n");
  kill(getppid(),SIGCONT);
}
```

```
void gest2(int sig) {
  printf("Non\n");
  kill(getppid(),SIGCONT);
}

void code_fils1() {
  printf("Je suis le fils 1. Je suis pret.\n");
  signal(SIGUSR1, gest1);
  while(1); // boucle infinie
}
void code_fils2() {
  printf("Je suis le fils 2. Je suis pret.\n");
  signal(SIGUSR2, gest2);
  while(1); // boucle infinie
}

main() {
 compteur = 0;
 signal(SIGCONT,gest_pere);
 pid1 = fork();
 if (pid1 == -1) perror("fork");
 else if (pid1 == 0) code_fils1();
 pid2 = fork();
 if (pid2 == -1)  perror("fork");
 else if (pid2 == 0) code_fils2();
 sleep(4);
 kill(pid1,SIGUSR1);
 while(1); // boucle infinie
 exit(0);
}
```

- Expliquez brièvement ce que fait ce programme et donnez un exemple d'exécution.
- Si on ne met pas l'instruction sleep(4) pour retarder l'envoi du premier signal par le processus parent, on peut se retrouver avec un sérieux problème, comme l'illustre l'exécution suivante:

  ```
  $ a.out
  Je suis le fils 2. Je suis pret.
  (boucle infinie)
  ```

  Expliquez le problème et proposez une solution.
- Ce programme fonctionne-t-il toujours comme on peut le souhaiter ? Expliquez pourquoi.

GROUPE ESIEE PARIS

CHAMBRE DE COMMERCE ET D'INDUSTRIE DE PARIS

19,5/20

Nom VINATIER Jean-Christophe

Classe I 2 T

Le 24 Février 20 06

Composition d'INSTO1

Feuille 1/2

2,5/3

1/ Mémoire :

FIFO :

E/3	0	1	4	2	1	0	3	0	1	4
0	0	0	0	0	0	0	3	3	3	3
1		1	1	1	1	1	1	0	0	0
2			4	4	4	4	4	4	1	1
3				2	2	2	2	2	2	4

4 fautes de pages.

Les pages sont mémorisées en mémoire le principe FIF (premier entré – premier sorti). Lorsqu'il y a un défaut de pages, la plus ancienne est retiré, elle se trouve en tête de liste.

LRU :

E/3	0	1	4	2	1	0	3	0	1	4
0	0	0	0	0	0	0	0	0	0	0
1		1	1	1	1	1	1	1	1	1
2			4	4	4	4	3	3	3	3
3				2	2	2	2	2	2	4

6 fautes de pages

LRU : Remplacement de la page la moins utilisée

Les pages en mémoire sont mémorisées dans une liste chaînée. La page la plus utilisée est en tête et la moins utilisée en queue de liste. Lorsqu'il y a un défaut de pages, c'est la page la moins utilisée qui est retirée.
Pour minimiser les temps, il faut réaliser la fonction par le matériel.

Belady : remplacement de page optimal.

Elle consiste à enlever de la mémoire la page qui sera référencée le plus tard dans le futur. On ne peut pas l'utiliser car on n'est pas capable de prédire les références futures d'un programme.

t / ω	0	1	4	2	1	0	3	0	1	4	
0	0	0	0	0	0	0	0	0	0	0	
1		1	1	1	1	1	1	1	1	1	1 faute de page
2			4	4	4	4	4	4	4	4	
3				2	2	2	3	3	3	3	

Il faut juste la page 0 reste en mémoire car elle est utilisée juste après

II) SEMAPHORES

Le code de la fonction allouer() ne fonctionne pas correctement car :

on suppose qu'il y a 2 machines de libres (dont la 0 pour simplifier (mais le principe reste même pour une autre machine)

le sémaphore mlibre vaut 2

* un premier client prend un sémaphore mlibre donc mlibre = 1 maintenant.

la fonction allouer va tester si la machine 0 est dispo ? → la réponse est OUI.

* à ce moment là le séquenceur donne la main à un autre client qui va prendre un sémaphore mlibre puis la fonction allouer va tester si la machine 0 est disponible, la réponse est vraie, la machine 0 est positionnée comme non disponible et revoie au client la machine 0.

Maintenant le premier client a de nouveau la main la machine 0 va aussi lui être attribuée.

Il y aura donc la possibilité que 2 clients se voient attribuer la même machine car il manque un système

de protection en lecture et en écriture sur le tabl
du booléen "disponible".

Pour corriger le problème, il faut rajouter un sémaphore
mutex pour avoir accès à la variable disponible

/* Déclarations des tableaux de mutex */

sem_t mutex;

/* initialisation du tableau de mutex */

sem_init (&mutex, 0, 1);

/* Fonction allouer */

```
int allouer ()
{
    sem_wait (&nlibre);
    sem_wait (&mutex);
    for (int i=0; i< NMACHINES; i++)
    {
        if ( disponible [i] == VRAI)
        {
            disponible [i] = FAUX;
            sem_post(&mutex);
            return i;
        }
    }
}

void relache (int machine)
{
    sem_wait (& mutex);
    disponible [machine] = VRAI;
    sem_post (& mutex);
    sem_post (& nlibre);
```

maintenant, il n'y a plus de problème car avant de lire et écrire les valeurs du tableau disponible, les fonctions doivent prendre un sémaphore initialisé à 1 au début du programme.

On doit pouvoir améliorer le système en mettant un tableau de sémaphores ainsi chaque machine aurait son propre mutex.

voir à la fin de la feuille 2

III) TUBES ET SEMAPHORES

* La fonction read() était bloquante si le tube est vide, elle permettra de réaliser la fonction sem_wait.

* La fonction write() permet de réaliser la fonction sem_post() car à partir du moment où il y aura quelquechose d'écrit dans le pipe, la fonction read() se débloquera.

```
/* Déclarations et initialisation du pipe */
int tube[2];
pipe(tube); // création du pipe, on ne fait pas le test du retour

/* fonction semwait */
void sem_wait(int fd[2])
{
    char c;
    read(fd[0], &c, 1);
}

void sem_post(int fd[2])
{
    write(fd[1], "1", 1);
}
```

GROUPE ESIEE PARIS

CHAMBRE DE COMMERCE ET D'INDUSTRIE DE PARIS

Nom VINATIER Jean-Christophe

Classe I3 I

Le 24 Février 20 05

Composition de INSTOL

Il peut y avoir un interblocage dans le cas où on se sert du pipe pour faire autre chose que le sémaphore.

Il y aura interblocage si une fonction utilise ce code

```
void fonction()
{
  sem_wait(tube);
  char buf[4096] = {'a'}; // initialise toutes les cases du buffer avec la lettre a
  write(tube[1], buf, 4096); // 4096 car un pipe fait en général 4Ko
  sem_post(tube);
}
```

Il y aura interblocage car l'écriture dans le pipe est bloquante lorsque celui-ci est plein donc on ne pourra jamais reposter le sémaphore.

IV) FORK :

1) Déroulement du programme :

- Un processus père crée un fils à l'aide de la commande fork() puis affiche 3 à sa sortie

- Le fils crée un fils2 puis attend la fin de fils2 pour afficher 2 et affiche 4 à sa sortie

- fils2 attend la fin de son fils qui n'existe pas donc la fonction va créer une erreur, puis affiche 1 et 4 à sa sortie

2) Le programme n'affichera pas toujours la même chose, cela dépendra du séquenceur et des processus auquel il donnera la main.

La chose dont on est sûr c'est qu'avant d'afficher 2, il y aura d'afficher 1 et 4 car le fils attend la fin du fils2 avant d'afficher 2.

ex d'affichage. 3 1 4 2 4 1 4 3 2 4

1 3 4 2 4 1 4 2 3 4

1 4 2 4 3

3) Les lignes que l'on peut supprimer sont:

pid_t pid; int status;

pid = wait(&status);

car le fils2 ne crée pas de fils donc la fonction wait renverra une erreur et ne modifiera donc pas le code.

4) Les sorties possibles sont:

— père — Fils — Fils2

3 1 4 2 4

1 3 4 2 4

1 4 3 2 4

1 4 2 3 4

1 4 2 4 3

V) SIGNAUX.

* Ce que fait le programme :

la commande signal (nom du signal, handle) permet d'associer au signal reçu le traitement effectué.

Le père crée un fils 1, puis un fils 2, dort 4 secondes, envoie le signal SIGUSR1 au fils 1 et entre dans une boucle infinie.

Le fils 1 affiche qu'il est le fils 1 puis attend dans une boucle infinie.

Le fils 2 affiche qu'il est le fils 2 puis attend dans une boucle infinie.

Lorsque le fils 1 reçoit le signal SIGUSR1, il affiche OUI puis envoie SIGCONT au père.

Lorsque le fils 2 reçoit SIGUSR2, il affiche NON et envoie SIGCONT à son père.

Lorsque le père reçoit le signal SIGCONT, il s'arrête si le compteur vaut 3 sinon il incrémente le compteur et envoie SIGUSR2 au fils 2 si le compteur est impair sinon SIGUSR1 au fils 1.

exemple d'exécution :

Je suis le fils 1. Je suis prêt.
Je suis le fils 2. Je suis prêt.
Oui
Non
Oui
Fin du programme

* Si on ne met pas de sleep(4), il y a un problème car le séquenceur n'aura pas forcément donné la main au fils 1. Celui-ci ne pourra donc pas avoir associé la réception du signal SIGUSR1 au traitement à effectuer c'est-à-dire geste1.

avant pid2 = fork()

Solution : mettre pause() après la création du fils 1 et pause() après la création du fils 2 à la place de sleep.

Cette fonction permet au processus d'attendre un signal avant de continuer son exécution.

Il faudrait dans ce cas modifier le code des fils 1 et 2, et celui de gest_pere. Les fils pourraient envoyer le signal SIGCONT à leur père lorsqu'ils ont terminés leurs initialisations et dans gest_pere gérer le traitement des 2 premières réceptions

```
void code_Fils1 ()
{ printf("Je suis le Fils 1. Je suis prêt.\n");
   signal(SIGUSR1, gest1);
   kill(getppid(), SIGCONT);
   while(1);
}
```

idem pour code_Fils2 en remplaçant les 1 par des 2

```
void gest_pere (int sig)
{ if (compteur == 5) exit(0);
   compteur ++;
   switch (compteur)
   { case 3 : kill(pid2, SIGUSR2);
              break;
     case 4 : kill(pid1, SIGUSR1);
              break;
   }
}
```

dans le main :

* Ce programme fonctionne toujours comme on le souhaite car il est indépendant de l'ordonnanceur, ce sont les signaux qui effectuent l'ordonnancement et la synchronisation.

Exercice 2 : Fin

```
int alloue()
{
 sem_wait(&mlibre);
 for(int i=0; i<NMACHINES; i++)
 { sem_wait(&mutex[i]);
   if( disponible[i] == VRAI)
   {
     disponible = FAUX;
     sem_post(&mutex[i]);
     return i;
   }
 sem_post(&mutex[i]);
 }
}

void relache (int machine)
{
 sem_wait(&mutex[machine]);
 disponible[machine] = VRAI;
 sem_post(&mutex[machine]);
 sem_post(&mlibre);
}
```

On aurait dispo = mutex et sem = 1 ?

Examen IN301 - Systèmes d'Exploitation

Responsable: Laurent Najman

29 janvier 2005

Durée: 3 heures

Instructions

- Ne vous contentez pas d'écrire un résultat. Justifiez et commentez vos réponses. Aucun point ne sera donné pour une réponse non justifiée ou une justification erronée.
- Document autorisé: polycopié du cours.

Les barèmes sont donnés à titre indicatif.

1 fork et signal (4 points)

Considérez le code suivant :

```
int counter = 0;
void handler(int sig)
{
    counter ++;
}
int  main()
{
    int i;

    signal(SIGCHLD, handler);

    for (i = 0; i < 5; i ++){
        if (fork() == 0){
            exit(0);
        }
    }
    /* Attendre les enfants */
    while (wait(NULL) != -1);

    printf("counter = %d\n", counter);
    return 0;
}
```

1. Décrivez le déroulement de ce programme.

2. Est-ce que le programme affiche la même chose à chaque fois qu'il est exécuté ? Est ce que la (ou les) valeur(s) de `counter` qui s'affiche(nt) à l'écran est(sont) la(les) même(s) à chaque exécution ?

3. Si la réponse à la question précédente est `Oui`, donnez la valeur de la variable `counter`. Si la réponse est `Non`, listez toutes les valeurs possibles de la variable `counter`.

2 sémaphore (6 points)

D'après E.N.S.E.I.R.B.

On dispose d'un mécanisme d'enregistrement à un ensemble de cours, tel que tout élève ne peut être enregistré qu'à au plus trois cours, et que chaque cours a un nombre limité de places offertes. Un élève qui a déjà choisi ses trois cours souhaite en abandonner un, pour en choisir un autre qu'il préfère. S'il commence par se désinscrire, pour s'inscrire à l'autre, et si le nouveau cours est plein, il se peut que l'élève ne puisse même plus retrouver le cours qu'il a quitté si un autre élève s'inscrit dans l'intervalle. Le service de la scolarité souhaite donc mettre en place un système de permutation de cours, permettant à un élève de changer de cours sans perdre le bénéfice des cours auxquels il est déjà inscrit. On propose l'implémentation suivante :

```
CoursT cours[N];
sem_t SEMcours[N];

void coursEchange (UserT utilisateur, int c1, int c2) {
  sem_wait(SEMcours[c1]);
  desinscrit (cours[c1], utilisateur);
  if (estPlein(cours[c2]) == false) {
    sem_wait(SEMcours[c2]);
    inscrit (cours[c2], utilisateur);
    sem_post(SEMcours[c2]);
  }
  sem_post(SEMcours[c1]);
}
```

Vérifiez si l'implémentation est correcte. Si elle est correcte, expliquez en détail pourquoi c'est le cas, en montrant comment est géré le cas où deux étudiants

(ou plus) veulent accéder en même temps au système. Si elle est incorrecte, listez et expliquez en détail l'ensemble des problèmes, et proposez une solution qui fonctionne.

3 tubes (4 points)

D'après E.N.S.E.I.R.B.

On s'intéresse à la manière dont le système Unix gère les tubes (`pipe`).

1. Quelle garantie offre le sytème Unix vis-à-vis des entrées-sorties sur un tube ? (0,5 point)
2. Que se passe-t-il si un processus désire écrire des données sur le `pipe` et qu'il ne reste plus assez de place dans la zone de stockage du tube pour contenir toutes les données ? (0,5 point)
3. Que se passe-t-il si la taille des données à écrire dépasse la capacité totale du tube ? Est-ce en contradiction avec la règle fondamentale énoncée précédemment ? (1 point)
4. Proposez (au moins) une manière pour le système d'exploitation de gérer plusieurs processus écrivains de telle sorte à offrir la garantie Unix de la question (1), sans avoir les problèmes potentiels des questions (2) et (3). Discutez et illustrez votre proposition (2 points).
   Note: Votre proposition peut être faite en français, il n'y a pas besoin d'écrire du code.

4 Gestion des fichiers ouverts (6 points)

D'après E.N.S.E.I.R.B.

On s'intéresse à la gestion interne par Unix des fichiers ouverts. Dans un premier temps, on supposera que seulement deux structures de données sont mises en oeuvre pour implémenter la gestion des fichiers ouverts dans l'ensemble du système :

- un tableau local de structures descripteurs de fichiers, situé dans la zone `u_area` de chaque processus (donc seulement manipulable au travers d'appels système), que l'utilisateur ne peut référencer qu'indirectement au moyen de valeurs entières appelées aussi, par raccourci, `descripteurs de fichiers`, alors qu'il faudrait plutôt dire `index de descripteur de fichiers` (`file handles` en anglais);

- un tableau d'i–nodes mémoire, situé dans l'espace noyau, dont chacun contient un compteur de références pour mémoriser le nombre de fichiers actuellement ouverts et utilisant cet i-node.

1. Sachant qu'il ne peut y avoir qu'un seul i-node mémoire alloué par fichier disque ouvert, justifiez dans lesquelles des deux structures doivent se trouver :
   - les droits d'accès d'ouverture du fichier (positionnés par `open()`) (0,5 point);
   - la position courante de lecture et/ou d'écriture dans le fichier (0,5 point).
2. Décrivez l'impact de l'appel système `open()` sur ces structures. Illustrez l'état de ces structures lorsqu'un deuxième processus ouvre le même fichier que le premier. (1,5 points)
3. Est-il possible, avec cette implémentation, de mettre en oeuvre l'appel système `dup()` ? Pourquoi ? (1 point)

   En fait, les concepteurs d'Unix ont mis en place une structure de données intermédiaire, la table des fichiers ouverts dans le système, qui s'intercale entre la table des descripteurs de fichiers et la table des i-nodes.
4. Où doivent maintenant se trouver chacun des deux champs de la question (1) ? (1 point)
5. Illustrez l'état de ces structures lorsqu'un processus ouvre un fichier avec `open()`, le duplique avec `dup()`, et ré-ouvre le même fichier. (1 point)
6. Quels champs doivent posséder les entrées de la table des fichiers ouverts pour permettre une gestion correcte de cette ressource ? (0,5 point)

IN301 – Examen

Durée 3h
Responsable : L. Najman
Aucune documentation permise - Calculatrice non autorisée

Consigne générale : Ne vous contentez pas d'écrire un résultat. Justifiez et commentez vos réponses. Aucun point ne sera donné pour une réponse non justifiée ou une justification erronée.
Les barèmes sont donnés à titre indicatif.

I. Questions de cours (4 pts)

A. Est-ce qu'un SE multitâche est nécessairement multiusager ? (0.5pts) Et pour l'inverse (0.5pts). Expliquez brièvement.

B. Le programme suivant effectue une opération mathématique très compliquée. Une fonction diviser_travail s'occupe de diviser le travail en plusieurs tâches.

```
int diviser_travail(void) {
     int i, pid, result = 0 ;
     int r[10] ;
     for (i=0 ; i<10 ; i++) r[i] = 0 ;
     for (i=1 ; i< 5 ; i++) {
          pid = fork() ;
          if (pid == 0) {
               int x ;
               if (x=(i%2)*10) r[i]=x ;
               else r[i] = 0 ;
          }
     }
     for (i=0 ; i<10 ; i++) {
          sleep(1) ;
          result += r[i] ;
     }
     return result ;
}
int main(void) {
     int x= diviser_travail() ;
     printf("x final=%d\n", x);
}
```

a/ Combien d'enfant(s) ce code génère-t-il ? (0.5 pts)

b/ Quels problèmes cette application engendre-t-elle ? (1 pt)

c/ Corriger le code et modifier le moyen de communication pour que les enfants et le père utilisent l'appel système exit() pour l'échange d'information. (1.5 pts)

Aide : le décalage à droite `(x>>8)` prend la valeur réelle du statut de sortie de l'appel système `wait(x)`.

II. Un peu d'unix (2 pts) [Ligne de commande]

Supposons que nous avons un fichier de texte dans lequel chaque page est numérotée avec la notation

```
<page n="[un nombre]" id n="[un nom de chapitre]">.
```

Le nombre et le nom de chapitre change de page en page, mais le reste de la chaîne reste identique. Remplacer dans tout le fichier la notation précédente par

```
Page [un nombre], Chapitre [un nom de chapitre]
```

III. Compréhension de fork() (5 points)

Considérez le programme C ci-dessous (pour des raisons de place, nous ne vérifions pas le code de retour des appels, aussi supposez que tous les appels retournent normalement)

```
main() {
     if (fork() == 0) {
          if(fork() == 0) {
               printf("3") ;
          } else {
               pid_t pid ; int status ;
               if ((pid = wait(&status)) > 0) {
                    printf("4") ;
               }
          }
     } else {
          printf("2") ;
          exit(0) ;
     }
     printf("0") ;
     return(0) ;
}
```

Des 5 sorties possibles listées ci-dessous, dites (et justifiez) lesquelles sont des sorties possibles pour ce programme. On suppose que le programme va jusqu'à sa fin normale.

A. 32040
B. 34002
C. 30402
D. 23040
E. 40302

IV. Synchronisation (4 points)

Supposons l'existence d'un appel système write_char qui insère une donnée dans un tampon circulaire d'affichage, ainsi qu'un thread écran qui lit ce même tampon pour afficher les données qui s'y trouvent. Le thread écran affiche à l'écran seulement quand le tampon est plein. Pendant qu'il vide le tampon pour afficher les données qu'il contient, les applications ne doivent pas insérer de nouvelles données dans le tampon circulaire. Considérez les deux implémentations de write_char ci-dessous.

Sachant qu'il peut y avoir plusieurs applications (plusieurs threads) en même temps qui veulent écrire dans le tampon, analysez chacune de ces implémentations et dites si elles satisfont les spécifications. Si elles sont erronées, indiquez les problèmes et corrigez le code en minimisant les modifications apportées.

Remarque : vous ne devez pas modifiez la fonction thread_ecran. Ce thread fonctionne correctement, et est donné ici à titre indicatif.

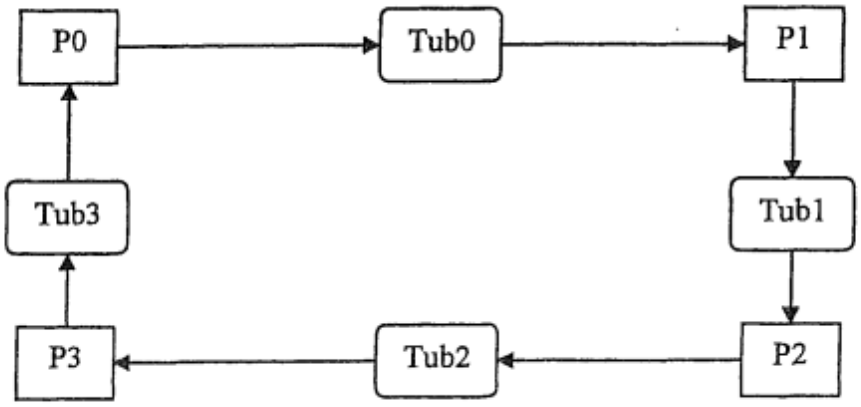
```
void write_char1(char c) {
     // sem1 initialise a 1
     // sem2 initialise a 0
     static int i=0:
     if (i%TAILLE == 0)
          sem_wait(&sem1);
     data[i%TAILLE] = c;
     i++;
     if (i%TAILLE == 0)
          sem_post(&sem2);
}
```

```
void write_char2(char c) {
     // sem1 initialise a 1
     // sem2 initialise a 0
     static int i = 0;
     sem_wait(&sem1);
     if (i%TAILLE == 0)
          sem_post(&sem2);
     else {
          data[i%TAILLE]=c;
          i++;
          sem_post(&sem1);
     }
}
```

```
void *thread_ecran(void *arg) {
     int i=0;
     while(1){
          sem_wait(&sem2);
          for (i=0; i<TAILLE; i++)
               affiche(data[i%TAILLE]);
          sem_post(&sem1);
     }
}
```

IV. Communication interprocessus (5 pts)

Considérez N processus qui communiquent au moyen de tubes de communication non nommés (unnamed pipe). Chaque processus partage deux tubes (un avec le processus de droite, et un autre avec le processus de gauche). Par exemple pour N=4, les processus communiquent selon le schéma suivant :



Complétez le code ci-après de manière à implémenter cette architecture de communication des N processus créés. L'entrée standard et la sortie standard de chaque processus Pi sont redirigés vers les tubes appropriés. Par exemple, pour le processus P0, l'entrée et la sortie standards deviennent respectivement les tubes tub3 et tub0.

```
#include "EntetesNecessaires.h"
#define N 4
void proc(int)

int main()
{
     int i;
     for (i=0; i<N; i++)
          if (fork() == 0) {
               proc(i);
               exit(0);
          }
     exit(0);
}
#include "codeproc"
```

Attention: vous ne devez pas écrire le code de la fonction proc.

NOM

dup, dup2 - Dupliquer un descripteur de fichier.

SYNOPSIS

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

DESCRIPTION

dup et dup2 créent une copie du descripteur de fichier oldfd. […]

VALEUR RENVOYÉE

dup et dup2 renvoient le nouveau descripteur, ou -1 s'ils échouent, auquel cas errno contient le code d'erreur.

NOM

pipe - Créer un tube.

SYNOPSIS

```
#include <unistd.h>

int pipe(int filedes[2]);
```

DESCRIPTION

pipe crée une paire de descripteurs de fichiers, pointant sur un i-noeud de tube, et les place dans un tableau filedes. filedes[0] est utilisé pour la lecture, et filedes[1] pour l'écriture.

En général deux processus (créés par fork) vont se partager le tube, et utiliser les fonctions read et write pour se transmettre des données.

VALEUR RENVOYÉE

pipe renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas errno contient le code d'erreur.

NOM

open, creat - Ouvrir ou créer éventuellement un fichier.

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

NOM

close - Fermer un descripteur de fichier.

SYNOPSIS

```
#include <unistd.h>

int close(int fd);
```

Nom : de Grancey Florence

Classe : I3 Groupe 3

Le 4 février 2004

CHAMBRE DE COMMERCE ET D'INDUSTRIE DE PARIS

12,5 / 20

Composition d IN301

I questions de cours

1,5 / 4

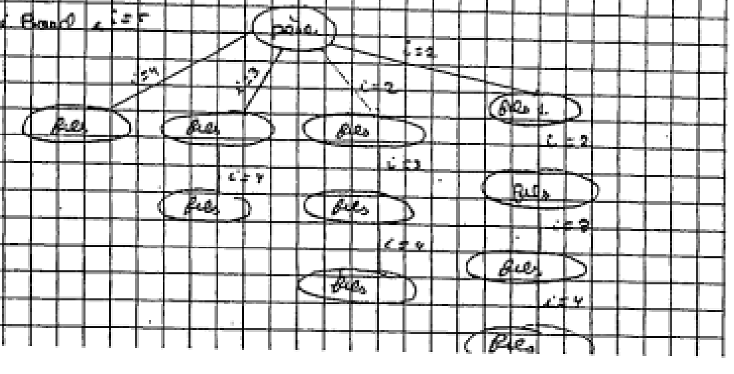
1) Un SE multitâche n'est pas forcément multiusager car un seul utilisateur peut lancer/créer plusieurs processus et donc avoir besoin d'un SE multitâche (ex : traitement texte + impression + programme de [illegible] IN301...)

1/

Inversement, un SE multiusager est forcément multitâche car il existe au moins un processus par utilisateur et un autre processus pour gérer l'ensemble des processus.

3) a) On a 4 tours de boucle, la création des processus s'effectue suivant le chemin suivant

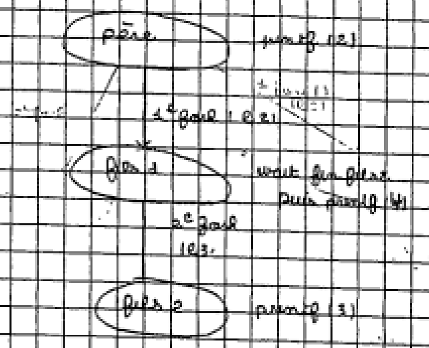
Sortie Boucle, i=5



5/5

1) compréhension de fork()

Afin de comprendre l'algorithme du programme on trace un diagramme processus/action



Le chiffre 3 est affiché quand on est dans processus fils 2

" " 4 " " " " " " " " " 1 (celui créé par le 1er fork).

Le chiffre 2 est affiché si on se trouve dans le père

" " a est affiché à la fin du programme

Le chiffre 4 est obligatoirement affiché après la fin du processus fils 2 et après le chiffre 3 (la commande wait() permet de forcer le fils 1 à attendre que le fils 2 se termine avant d'afficher 4

On a donc obligatoirement ... 3 ... 4 ...

La proposition E n'est pas valide

pas tout à fait

On remarquera que parce que le fils 1 [illegible] le statut de fin du fils 2 il faut que le fils 2 soit [illegible] terminé et donc pit [illegible] (comme les fils ont tous accès grâce à [illegible]) effectue [illegible] printf (3) [illegible] obligatoirement avant

wait() il y a duplication processus et du code père, d'où l'exécution printf(4)

6 [illegible]

Quand i = 1, grâce à fork () le processus père créé un fils

i = 2, père et fils ayant le même code, ceux-ci se dédoublent.

Et ainsi de suite jusqu'à ce que i = 5 et que l'on sorte de la boucle

Non On a donc 1+2+3+4 processus fils créés = 10

b) cette application engendre des problèmes de synchronisation des processus père et fils. En effet comme il n'y a pas de fonction

Non wait(), le père peut se terminer avant son fils et son fils peut devenir orphelin

En outre, il va y avoir des problèmes d'accès aux variables

Non x et n. Il est probable qu'au moins 2 processus accèdent en même temps aux variables créant ainsi des résultats imprévisibles

0,5 c) unit diviser travail

```
{ int i , pid, result = 0
     for(i=0 ; i<n, i++) n[i]=0

     for (i=1, i<5, i++)
  {      pid = fork()

        if (pid ==0)
          { int x ;
             if (i%2)!=0) n[i]=x
             else n[i]=0
             exit(x)
          }
      else { //
  for(i=0, i++ i<10)
     Y = wait(x) ;
         result += (Y%8) ;
     }, };
   return result ;
}
```

2 ?

0,5/2 II un peu d'unix

Pour remplacer les ... notations, il faut utiliser la commande sed qui permet d'effectuer des traitements

```
(a)
sed -f1 "<page"   "Page"
sed -f2 "n g[0-9]"   '[0-9]'
sed -f3 "tit "   " chapitre"
sed f4 "n o[0-9]"   '[0-9]'

(b) cut  '<'  '>'
```

?

car le test de fin de tampon étant effectué avant l'écriture et l'incrément de i, il est fort probable que l'on appelle write-char 4 fois pour rien et perde un caractère

Il faudrait en fait inverser les instructions de la boucle de test

Modifications

```
void write_char 2
 { static int i =0
   sem_wait (& sem 1)
   if (i % TAILLE != 0)
     { data [ i % TAILLE ] = 6 ;
       i++ ; }
       sem_post (& sem 1)
   else { sem_post (& sem 2)
 }
```

perte de 4/

2,5/5

IV Communication inter processus

On doit ici établi des tubes de communication entre processus non nommés. L'intérêt de ces tubes est qu'ils permettent de synchroniser facilement les processus car

- quand le tube est plein, le processus écrivain ne peut le remplir
- " " " est vide " " " lecteur ne peut le lire

ESIEE PARIS
CHAMBRE DE COMMERCE ET D'INDUSTRIE DE PARIS

Nom DEGRANCEY Florent

Classe I3

Le 4 Février 2D

Composition d 1 x 301

2/3

Les sorties retenues sont donc la A et la C et la D

La sortie B signifierait que le fils 2 se termine et le fils ensuite le père des processus se termine et ensuite le fils 1 se termine

La sortie C signifie que le fils 1 se termine, puis le fils 2 puis le père

La sortie D, elle, signifie que le père se termine puis le fils 2, puis le 1

Les 3 sorties respectent la synchronisation entre les 2 fils

En ce qui concerne le processus père, celui-ci affichant un des 2 ou qu'il a affiché 2 et n'attendant pas ses fils, il peut se terminer n'importe quand

3/4

IV Synchronisation de processus

On veut que :

- le thread écran n'affiche que si le tampon est plein
- les applications n'insèrent pas de données pendant que le thread écran vide le tampon

On effectue donc une synchronisation par exclusion mutuelle à l'aide de sémaphores

Étude de ecrit_char

Cette procédure ~~prend~~ attend et sem1 lorsqu'elle est en début de tampon, ensuite écrit le caractère, incrémente son compteur de place dans le tampon puis relâche sem2 si elle est en début de tampon

Le thread écran

- attend que le sem2 soit libre
- écrit à l'écran
- relâche le sem1

On comprend donc que sem1 est équivalent à "droit d'écrire" et sem2 à "droit d'afficher (ou vider le tampon)"

La procédure ecrit_char1 n'est en fait pas conforme aux spécifications car elle n'attend sem1 que si elle est en début de tampon. Si elle est en milieu, elle n'attend pas le sem1 et peut (même si thread_ecran peut être en train d'écrire).

Il y a donc un problème au niveau de la synchronisation de droit d'écriture

En revanche, cette procédure respecte bien la condition du thread qui n'affiche tampon (de plus, l'incrément avant le test permet d'éviter des séquences d'affichage où le tampon n'a pas été rempli)

Modification

```
void ecrit_char (char c)
{
    static int i=0
    sem_wait (& sem1) ;        // attente de sem1
    tab[i%TAILLE] = c ;
    i++ ;
    sem_post (& sem1) ;
    if (i%taille == 0)
        { sem_post (& sem2) }
    else { sem_post (& sem1) } ;   // si on n'est pas en fin de tampon
                                   // relâcher sem1 pour pouvoir
                                   // rappeler la procédure
}
```

O.K.

Procédure ecrit_char2

Cette procédure
- attend que sem1 soit relâché
- relâche sem2 si elle est en fin de tampon
- sinon elle écrit dans le tampon, incrémente i et relâche sem1

Cette procédure respecte bien l'attente que thread_écran ait terminé pour écrire dans le tampon (par l'attente de sem1 qui n'est pas relâché si elle est en fin de tampon).

Elle permet aussi de bien synchroniser le lancement de l'affichage en ne relâchant sem2 (i.e. droit de vider le tampon) que lorsqu'elle est

N 24/

ESIEE PARIS

CHAMBRE DE COMMERCE ET D'INDUSTRIE DE PARIS

Nom de Gyan.cay Florent

Classe I 2

Le 4 Février 20

Composition d INFO 1

3/3

```
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>

#define N 4

int tube[N][2]
void proc(int i)

int main ()
{ int i; int entree[N], sortie[N]

for (i=0; i<N; i++)

pid_t pid = pipe(tube[i][2])

if (fork() == 0)   // on est dans le fils

{   entree[i] = open(0, O_RDONLY);

    sortie[i] = open(1, O_WRONLY);

    tube[i,d] = entree;

// entrée standard mise sur la zone d'ecriture du tube

if (i != 3):        tube[1-i, 3-0/3][0]   = sortie

// sortie standard du processus sur la zone de lecture du tube

    proc(i);

    close(1); close(0)

    exit(0);

} }

exit(0);
}
```

On définit le tube comme un tableau à 2 dimensions où le numéro de ligne (0 à 3) correspond au numéro du tube et le numéro de colonne au champ lecture ou écriture du tube

Pour chaque processus, on effectue des redirections d'entrée / sortie standard en

- ouvrant la sortie et l'entrée standard de fd respectifs 1 et 0
- allouant les descripteurs de fichiers du tube à celui des entrées sorties
- puis en fermant les entrées sortie standard

IN3ST01 – Examen

Durée 3h
Responsable : L. Najman
Aucune documentation permise - Calculatrice non autorisée

Consigne générale : Ne vous contentez pas d'écrire un résultat. Justifiez et commentez vos réponses. Aucun point ne sera donné pour une réponse non justifiée ou une justification erronée.
Les barèmes sont donnés à titre indicatif.

I. Questions de cours (4 points):

Pour chacune des affirmations suivantes, dites si elle est vraie ou fausse et justifiez votre réponse.

A/ Un programme, lorsqu'il est exécuté, n'a jamais les privilèges du noyau. Seuls les processus du système d'exploitation ont ces privilèges.

B/ En général, un système d'exploitation multi-threads requiert l'utilisation de mécanismes de synchronisation.

C/ Le programme suivant est déterministe (toute exécution affichera toujours les mêmes résultats)

```
int main() {
    pid_t pid ;
    if ((pid=fork()) == 0)
        printf("Je suis le premier message\n") ;
    else {
        printf("Je suis le second message\n") ;
        wait(NULL) ;
    }
}
```

D/ Dans le programme suivant, le processus sera éliminé par l'envoi du signal SIGSTOP

```
int Continuer = 1 ;

void action(int SIG) {
    switch(SIG) {
        case SIGSTOP :
            printf("SIGNAL : SIGSTOP attrapé\n") ;
            Continuer = 0;
            break;
        case SIGINT :
            printf("SIGNAL : SIGINT attrapé\n") ;
            break;
        case SIGUSR1 :
            printf("SIGNAL : SIGUSR1 attrapé\n") ;
            break;
        default :
            printf("SIGNAL : Signal inconnu\n") ;
            break;
    }
}
```

```
int main(int argc, char *argv[]) {
     signal(SIGSTOP, action);
     signal(SIGINT, action);

     while (Continuer) {
          kill(getpid(), SIGINT);
          sleep(1);
          kill(getpid(), SIGUSR1);
          sleep(1);
          kill(getpid(), SIGSTOP);
     }
     exit(0) ;
}
```

II. Un peu d'unix (2 points) [Ligne de commande]

1/ Transformez la phrase : "J'ai deux chiens et trois chats" en "J'ai trois chiens et deux chats".
2/ Cherchez dans le fichier texte MonFichier les expressions finissant ".html", commençant par une lettre majuscule, et ne contenant aux autres endroits que des lettres minuscules. Affichez le résultat en remplaçant le .html par .sgml. (Attention, cela peut se faire en une seule commande).

III. Compréhension de fork (5 points)

Considérez le programme C ci-dessous (pour des raisons de place, nous ne vérifions pas le code de retour des appels, aussi supposez que tous les appels retournent normalement)

```
main() {
     if (fork() == 0) {
          if(fork() == 0)
               printf("3") ;
          else {
               pid_t pid ; int status ;
               if ((pid = wait(&status)) > 0)
                    printf("4") ;
          }
     } else {
          if (fork() == 0) {
               printf("1") ;
               exit(0) ;
          }
          printf("2") ;
     }
     printf("0") ;
     return(0) ;
}
```

Des 5 sorties possibles listées ci-dessous, dites (et justifiez) lesquelles sont des sorties possibles pour ce programme. On suppose que le programme va jusqu'à sa fin normale.

- A. 2030401
- B. 1234000
- C. 2300140
- D. 2034012
- E. 3200410

IV. Gestion mémoire (3 points)

Soit un système de mémoire paginée, dans lequel il y a un processus avec la chaîne des références de pages : S = {244, 1A1, 244, 363, 244, 268, 244, 1A1, 1A2, 363}. Montrer l'état de la mémoire à chaque instant, si l'on suppose que le système possède 3 cadres de pages et qu'initialement ils sont libres. Utilisez :

a/ algorithme optimal

b/ LRU

Le même système de gestion de mémoire paginée à **deux niveaux** utilise une TLB. On a mesuré les temps suivants :

Temps moyen d'accès au TLB = 4nsg

Temps moyen d'accès à la mémoire principale = 33nsg

c/ Si le TLB a un taux de réussite de 98%, quel est le temps moyen d'accès à la mémoire ?

V. Synchronisation (6 points)

Dans les systèmes modernes, un appel système d'affichage met d'abord les données dans un tampon circulaire. Généralement, lorsque celui-ci est plein, il est vidé et affiché à l'écran.

Le modèle illustré à la figure 1 contient deux threads (un thread Application et un thread Ecran). Lorsque l'application veut afficher, elle utilise un appel système (tel write) qui appelle à son tour la méthode write_char. Celle-ci met un caractère à la fois dans un tampon circulaire. Un thread Ecran affiche au fur et à mesure les valeurs qui sont ajoutées dans le tampon. Les codes ci-dessous décrivent les fonctions write_char et le thread Ecran.

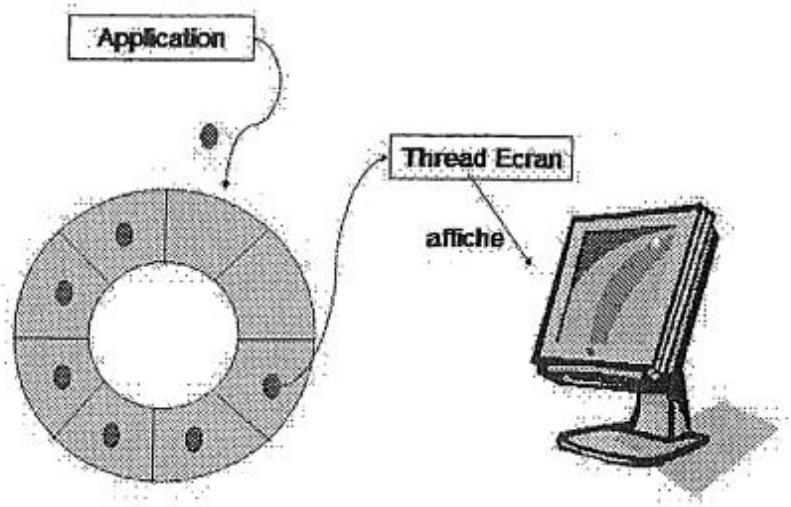


Figure 1 : Illustration

```
void write_char(char c) {
      static int i=0;
      sem_wait(&sem1);
      data[i%TAILLE] = c;
      i++;
      sem_post(&sem2);
}

void *thread_ecran(void *arg) {
      int i=0;
      while (1) {
            sem_wait(&sem2);
            affiche(data[i%TAILLE]);
            i++;
            sem_post(&sem1);
      }
}
```

Rappel: une variable dite `static` n'est initialisée que lors de la première exécution de la fonction dans laquelle elle se trouve, et la donnée dans la variable n'est pas effacée lorsqu'on sort de la fonction.

1/ Initialisez les sémaphores utilisés dans le code
2/ Afin d'éviter la corruption de données, ne devrait-il pas y avoir un mutex pour protéger le tableau data ? Justifiez
3/ Modifiez les codes ci-dessus,en utilisant les fonctions Posix de synchronisation et des structures de contrôle C/C++ (if, for, while) afin que le thread Ecran démarre l'affichage chaque fois que le tampon est plein et reste en attente tant que le tampon n'est pas plein. Pendant que l'écran affiche, les applications ne doivent pas remplir le tampon circulaire. Elles doivent attendre que l'écran ait terminé son travail. N'oubliez pas d'indiquer les valeurs d'initialisation de tous les sémaphores. Il est important que votre implémentation fonctionne correctement dans un environnement multi-threads où il pourrait y avoir plusieurs threads qui veulent exécuter en même temps l'appel write_char. Vous devrez aussi éviter les attentes actives.

NOM

sem_init, sem_wait, sem_trywait, sem_post, sem_getvalue, sem_destroy - operations sur les sémaphores

SYNOPSIS

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int valeur);
int sem_wait(sem_t * sem);
int sem_trywait(sem_t * sem);
int sem_post(sem_t * sem);
int sem_getvalue(sem_t * sem, int * sval);
int sem_destroy(sem_t * sem);
```

16,5/20

Nom Laurent Benoît

Classe I3 T

Le 6 02 2004

Composition d' IN 307

I question de cours

4/4

A)

Un programme lorsqu'il est stocké sur le disque possède des droits, en lecture en écriture, en exécution ainsi qu'un propriétaire (owner).

Ainsi à l'exécution un programme d'un utilisateur standard ne peut faire toutes sortes d'opération il n'est donc pas exécuté en mode noyau, il peut faire à des bibliothèque gérés par le noyau, mais il ne s'exécute pas en mode noyau

Manque un verbe

Cependant si l'utilisateur root compile un programme dans lequel qui fait des adresses mémoire illégales, voire même effacé des fichiers système rien ne l'en empêchera.

En ce qui concerne le processus du système
et exploitation, l'ordonnanceur, le
démon Kswap, par exemple
doivent pouvoir garder le dessus lorsque
nécessaire il faut donc qu'ils soient placés
en mode noyau.

11 c)

On peut très bien imaginer
un système d'exploitation multi-threads
qui lance des threads n'ayant
aucun besoin de communiquer entre eux.
La synchronisation n'est alors pas
nécessaire. On ne peut pas faire grand
chose, le partage de table, les //, la
gestion de + eurs clients en même temps
n'est pas possible.

Mouais!

d) NON

Puisque rien ne permet de dire
si le fils commence en premier,
certes le père attend la fin du fils mais
le père peut très bien s'exécuter
avant le fils, de toute manière cela
dépend de l'ordonnanceur.

D 9)

Dans ce programme on a utilisé en réalité une modification de signaux, avec

signal (SIGSTOP, action)

signal (SIGINT, action)

On a donc redéfini le comportement par défaut de SIGSTOP et SIGINT suivi par celui de SIGUSR1 donc lors que l'exécution

kill (getpid(), SIGUSR1)

le comportement de SIGUSR1 est de kill

donc le processus sera éliminé par l'envoi de SIGUSR1

1/1

II

2/2

1) sed 's/mois/ans/g' \2\1 "J'ai deux enfants et trois chats"

1/1

2)

1/0

sed 's/[A-Z].*\.html/\.html/\ syml/g' Monfichier

2/5

III

Pour la méthode de simplification, adoptons la convention suivante

```
main () {
    if (fork() == 0) {
        if (fork() == 0) {
            printf("3");
        } else {
            pid = pid . int fork
            if (pid = wait(&status)) > 0) {
                printf("4");
            }
        }
    } else {
        if (fork() == 0) {
            printf("2");
            sleep(2);
        } else {
            printf("1");
            sleep(2);
        }
        printf("0");
    }
```

Si l'on considère le résultat de l'exécution par le programme

Père 1 Père 2

1 0

Nom Laurent Benoît

Classe I4 T

Le 6 02 20 04

Composition d' IN 3[illegible]

Fils 1 Fils 3

3 0

Fils 1 Père 3

4 0

Père 1 Fils 2

Les affichages peuvent s'exécuter dans n'importe quel ordre.

On voit de suite qu'on peut 2

OU s'affiche. Donc D ne se produira pas

le bloc Fils 1 Père 3 ne commence à afficher que lors de la fin d'un processus soit après un zéro (c.f. exp. précédent) soit après un exit (Père 1 Fils 2)

l'affichage A

20 30 10 7

est la première séquence possible.

A correspond à l'ordre d'exécution.

Père 1 Père 2 Fils 1 Fils 3 Fils 1 Fils 3 Cas 1 Fi

↑

Fin du processus
Fils 3

Si l'ordonnanceur n'exécute pas les processus
dans le même ordre

On peut avoir B si Père 1 Fils 2
s'exécute en parallèle

$F_1 P_3$
1, 2, 3, 4 o o o
$P_1 P_2$ $F_2 F_3$
$P_1 F_2$

car il y a un en cours avec Fils 2

Les 3 derniers peuvent venir de n'importe
lesquels des 3 processus restants

E est possible.

2 processus se sont terminés,
on peut afficher
dans P_3

$F_1 F_3$ $P_1 P_2$ 4
Ainsi $F_1 P_3$
$P_1 F_2$ s'exécutent
Soit
$F_1 F_3$ en 1er
soit $P_1 P_2$

2,5/1,5

IV

a) Belady

244	244	244	244	244	244	244	244	1A2	363
	1A1	1A1	1A1	1A1	1A1	1A1	1A1	1A1	1A1
			363	363	268	268	268	268	268

1/

244 1A1 244 363 244 268 244 1A1 1A2 363

Oui! Il y a + (?) [illegible] [illegible] dans les [illegible] [illegible] puisque l'on ne sait pas les pages changées dans l'avenir.

b) LRU

244	244	244	244	244	244	244	244	244	363
	1A1	1A1	1A1	1A1	268	268	268	1A2	1A2
			363	363	363	363	1A1	1A1	1A1

1/

244 1A1 244 363 244 268 244 1A1 1A2 363

c)

dans 98% des cas le temps d'accès est de 4 ns

0,5 et dans 2% il est de 33 ns

⇒ temps accès moyen

Non

$$\frac{98 \times 4 + 2 \times 33}{100} = 4,58 \text{ ns}$$

V

1)

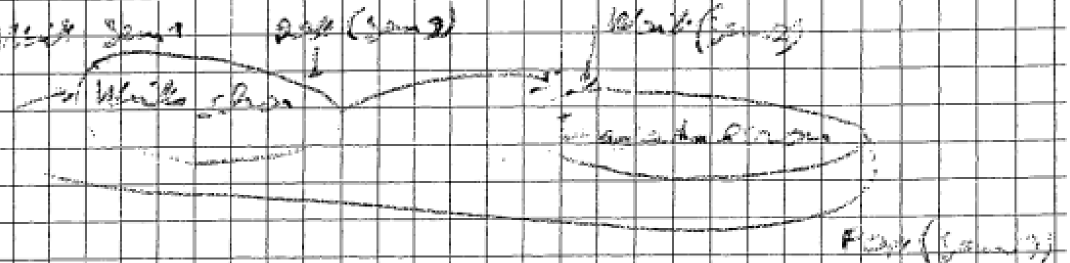
```
sem_t sem1, sem2 ;

sem_init ( & sem1, 0, 1 ) ;
sem_init ( & sem2, 0, 0 ) ;
```

-> On considère le buffer vide initialement il est donc possible d'écrire d'abord. le thread écrit est en attente d'une condition il est bloqué.

2)

Utiliser de la sorte les sémaphores sont les mêmes, puisque while_chan ne s'exécute que si thread écrit ... while ...



Voilà donc besoin de mutex.

![ESIEE Paris logo]
CHAMBRE DE COMMERCE ET D'INDUSTRIE DE PARIS

Nom Yamout Francis

Classe I5 T

Le 6 02 2004

Composition d IV 3 07

1°)

On considère 2 sémaphores.
Writer pour l'écriture dans le Buffer
Buffer pour l'accès au Buff en parallèle

```
sem_t Buffer;
sem_t Writer;

sem_init(&Buffer, 0, 1); // Buffer
   occupé par personne

sem_init(&Writer, 0, 1); // aucun
   n'écrivain n'y accède



void * thread_sem(void * arg) {
   int i;
   while(1)
      if (sem_trywait(&Buffer)==0)
      {
         for(i=0 ; i< Taille ; i++)
            affiche(tableau[i]);
```

```
        sem_post(&mutex);
    }
    else
    { faire autre chose
    }
}

int write_char(char c) {
    static int i = 0;
    if (sem_trywait(&mutex) == 0)
    {
        data[i] = c;
        i++;
        if (i == TAILLE)
        {   i = 0;
            sem_post(&buffer);
        }
        else
            sem_post(&mutex);
    }
    else
    {
        return OCCUPER;
    }
}
```

Alors si Write-then [illegible] OCCUPER

l'Appli peut faire autre chose.

Ok plus [illegible] exemple le problème de

d'overflow du i++ car

i n'était protégé [illegible] [illegible]

Write-then peut être appelé par

+ threads différents puisqu'on peut Write

peut être fait à la fois.

Le thread client peut si le buffer

n'est pas plein faire autre chose

puisqu'il ne bloque pas (Reprendre exemple).

J'ai ici utilisé deux buffers

que l'on échange lorsque l'on

en a rempli 1, laissant

l'autre disponible pour traitement par le thread

client.

4

Mais je ne suis pas tout à fait sûr que ça marche !