Examen de SELC/INF104
1h30 – Sans documents

Note : En annexe du sujet, vous trouverez un canevas de code et de tableaux à remplir. L'espace laissé pour répondre n'est pas représentatif du nombre de lignes de code à écrire.

Processus et fichiers (5 points)

Question 1 (1,5 points)

Pour créer des processus, les systèmes UNIX implémentent la fonction *fork.* La valeur de retour d'un fork peut signifier trois choses différentes. Quelles sont les trois interprétations possibles pour la valeur de retour d'un appel à fork, et les ensembles de valeurs qui leur correspondent ?

Fork peut renvoyer
-1 : une erreur s'est produite dans l'exécution de fork.
0 : le processus courant est le processus fils ;
le pid du fils (une valeur strictement positive) : le processus courant est le processus père.

Question 2 (1,5 points)

Expliquer brièvement l'effet de la fonction *execv* dont on donne la signature ci-dessous. Commencez par expliquer l'effet de l'exécution de la fonction *execv,* puis expliquez à quoi correspondent les paramètres *path* et *argv*.

```
int execv(const char *path, char *const argv[]);
```

execv remplace le code du processus qui exécute cette fonction par le code du fichier binaire exécutable (i.e. programme) qui se trouve à l'emplacement référencé par l'argument *path. argv* est le tableau de caractères qui sera passé en paramètre de l'exécution du point d'entré (fonction *main*) de ce programme.

Question 3 (2 points)

On donne le code ci-dessous.

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
char* p[]={"ps", NULL};
int main(int argc, char *argv[]){
  printf("Début d'exécution du programme %s\n", argv[0]) ;
  sleep(2);/* Laisser le temps à la fonction printf d'être finalisée */
  int f = fork() ;
  if(f==0)
    execv("/bin/ps", p);
  else
  {
    int wret ;
    wait(&wret) ;
    printf("Fin de l'exécution de %s", argv[0]) ;
  }
 return 0 ;
}
```

On suppose que l'application compilée à partir de ce code s'appelle *prog-fichiers*, et que l'utilisateur invoque cette application grâce à la commande ci-dessous ('$' identifie le début d'une ligne de commande). On supposera que l'exécutable de ps se trouve dans /bin/:

$./ prog-fichiers > exec_trace.txt

On exécute ensuite la commande cat (affiche le contenu d'un fichier sous format textuel) :
$cat exec_trace.txt

Lequel des résultats ci-dessous est correct ? On rappelle que ps liste les processus qui s'exécutent sur la machine. Justifiez votre réponse, en décrivant l'état de la *table des fichiers ouverts* (table des descripteurs de fichiers) pour chaque processus produit par l'exécution du programme *prog-fichiers.*

Possibilité 1 :

```
$./ prog-fichiers > exec_trace.txt
Début d'exécution du programme prog-fichiers
  PID TTY          TIME CMD
 7431 pts/0    00:00:00 prog-fichiers
 7434 pts/0    00:00:00 bash
18585 pts/0    00:00:00 ps
Fin de l'exécution de prog-fichiers
$cat exec_trace.txt
```

Possibilité 2 :

```
$./ prog-fichiers > exec_trace.txt
$cat exec_trace.txt
Début d'exécution du programme prog-fichiers
  PID TTY          TIME CMD
 7431 pts/0    00:00:00 prog-fichiers
 7434 pts/0    00:00:00 bash
18585 pts/0    00:00:00 ps
Fin de l'exécution de prog-fichiers
```

Possibilité 3 :

```
$./ prog-fichiers > exec_trace.txt
  PID TTY          TIME CMD
 7431 pts/0    00:00:00 prog-fichiers
 7434 pts/0    00:00:00 bash
18585 pts/0    00:00:00 ps
$cat exec_trace.txt
Début d'exécution du programme prog-fichiers
Fin de l'exécution de prog-fichiers
```

La possibilité 2 est correcte : au lancement du processus, dans la table des descripteurs de fichiers du processus principal (celui créé lors de l'exécution de la commande *$./ prog-fichiers > exec_trace.txt*) associe à la sortie standard *stdout* le fichier *exec_trace.txt*. Lors de l'éxécution du fork, un nouveau processus est créé. Par duplication de la zone mémoire des données, le processus fils a une table des descripteurs de fichiers qui associe également la sortie standard du processus au fichier *exec_trace.txt*. L'exécution de cat affiche le contenu du fichier, qui contient toutes les chaines de caractères envoyés par les deux processus (père et fils) vers la sortie standard.

Synchronisation (5 points)

Sur un pont étroit, les voitures ne peuvent passer que dans un seul sens à la fois : le pont peut être traversé dans n'importe quel sens, mais les voitures ne peuvent pas se croiser sur le pont. De plus, ce pont est en mauvais état, et si plus de trois voitures s'y engagent, il risque de s'effondrer. Le scénario de passage pour les voitures est le suivant :

1. Toutes les voitures se trouvant sur le pont vont dans le même sens,
2. Si le pont est vide et si aucune voiture n'attend, la voiture qui veut entrer passe sur le pont,
3. Si le pont n'est pas vide et qu'une voiture attend dans le même sens que celles qui sont déjà sur le pont, elle peut s'y engager, en respectant la contrainte du maximum de trois voitures sur le pont.

Chaque sens de traversée du pont est noté 0 et 1.

On suppose trois fonctions associées aux sémaphores : void init(sem_t s, int val), void P(sem_t s) et void V(sem_t s) correspondant aux actions classiques d'un sémaphore : initialisation, P et V.

Sémaphores utilisés :

- *sem_t XU*, avec *Init(XU,1)*
- *sem_t VouloirPasser[2]*, avec *Init(VouloirPasser[0], 0)* et *Init(VouloirPasser[1], 0)*

Variables partagées :

- *int Voitures*, initialisée à 0, indique le nombre de voitures engagées sur le pont
- *int SensPassage*, initialisée à 1, indique le sens de passage courant sur le pont
- *int NbEnAttente[2]*, initialisé ainsi : *NbEnAttente[0] =0 , NbEnAttente[1]=0*, indique le nombre de voiture en attente pour passer dans chaque sens

On considère le code ci-dessous. Les processus « Voiture » exécutent chacun :

```
Voiture ()
{
  int MonSens=((int)getpid() %2);// indique le sens de
                                 // traversé de la
                                 // voiture (1 ou 0)
  EntrerPont(MonSens);
  Rouler_sur_le_pont();
  SortirPont();
}
```

Les deux fonctions EntrerPont(int direction) et SortirPont() vont gérer l'accès au pont. Le code de EntrerPont() est fourni ci-dessous. Le code de SortirPont() est fourni en annexe A (à rendre). Il faut le compléter en écrivant du code en dessous des commentaires */* A compléter */*. Les commentaires /* Texte descriptif */ sont des indications sur ce que fait le code qui suit le commentaire.

```
EntrerPont(int MonSens)
{
  P(XU);
  /*** Attendre de pouvoir entrer sur le pont ***/
  while ((Voitures == 3) || (Voitures > 0 &&
                    SensPassage != MonSens) )
  {
    NbEnAttente[MonSens]++;
    V(XU);
    P(VouloirPasser[MonSens]);
    P(XU);
    NbEnAttente[MonSens]--;

  }
  /*** Entrer sur le pont ***/
  Voitures++;
  SensPassage = MonSens;
  V(XU);
}
```

Question 4 (3 points)

Compléter de façon simple les deux branches du « if » de la fonction *SortirPont()* en justifiant les instructions ajoutées. Voir annexe A pour la correction.

Remarque : « 1-SensPassage » donne le sens de passage contraire à SensPassage.

Question 5 (2 points)

A priori, la solution mise en œuvre à la question 4 pose un problème de famine. Si votre solution résout déjà ce problème, vous avez répondu à la question 5 et vous pouvez passer à la suite. Sinon, complétez la solution proposée à la question 4 pour résoudre le problème de famine. Voir annexe A pour la correction.

Signaux (5 points)

On considère le code en annexe C (à rendre) qu'il faut compléter en écrivant du code en dessous des commentaires */* A compléter */*.

Question 6 (2 points)

Expliquer brièvement l'effet de l'appel à la fonction *signal*, ligne 20 du code en annexe C.

L'exécution de cette fonction permet de remplacer le comportement par défaut associé à la réception d'un signal. Dans le code fourni, on appellera la fonction *on_timeout* sur réception d'un événement *SIG_ALRM.*

Expliquer brièvement l'effet de l'appel à la fonction *alarm*, ligne 22 du code en annexe C.

Cette fonction demande au système d'exploitation d'envoyer un signal *SIG_ALRM* au processus courant (celui qui exécute la fonction) dans au moins X secondes ; X étant une valeur entière positive passé en paramètre de l'appel à *alarm.*

Question 7 (3 points)

Le code proposé fait appel à la fonction *process_picture_precise.* L'exécution de cette fonction peut prendre beaucoup de temps, et on dispose d'une fonction de traitement d'image moins précise mais plus rapide. On souhaite donc essayer la fonction *process_picture_precise* pendant un certain temps (précisé par la variable timeout), mais on veut exécuter une solution de repli via la fonction *process_picture _fast* si *process_picture_precise* ne fournit pas son résultat suffisamment rapidement (c'est-à-dire avant échéance du timeout).

Pour ce faire, compléter le code fourni en annexe C en utilisant les fonctions suivantes (on considérera qu'on passe la valeur 0 pour le paramètre *savemask* de la fonction *sigsetjmp*):

```
int sigsetjmp(sigjmp_buf env, int savemask);
void siglongjmp(sigjmp_buf env, int val);
```

Voir annexe C pour la correction.

Indication donnée par *man* pour la fonction *sigsetjmp*:

```
RETURN VALUE

    If the return is from a successful direct invocation, sigsetjmp()
returns 0. If the return is from a call to siglongjmp(), sigsetjmp()
returns a non-zero value.
```

Indication donnée par *man* pour la fonction *siglongjmp*:

```
RETURN VALUE

    After siglongjmp() is completed, program execution continues as if the
corresponding invocation of sigsetjmp() had just returned the value
specified by val.
```

Mémoire (5 points)

On considère un système de pagination dans lequel une page fait 256 octets (on rappelle que 256 = 2^8). On considère un processus P dont l'espace d'adressage (logique) nécessite 6 pages (P0, P1, P2, P3, P4, P5). P s'exécute sur une machine dont la mémoire propose 3 blocs de 256 octets représentés ci-dessous :

B0	B1	B2

B0 a des adresses physiques allant de 0 à 255, B1 a des adresses physiques allant de 256 à 511, B2 a des adresses physiques allant de 512 à 767. Enfin, on considère qu'une adresse est codée sur 2 octets (16 bits, 2^{16}=65536).

Question 8 (2 points)

On considère que la table des pages du processus P est comme suit :

Entrée dans la table des pages	Bloc utilisé
0	Bloc disque
1	Bloc disque
2	B0
3	B1
4	B2
5	Bloc disque

On considère qu'une instruction du processus P nécessite de récupérer une donnée à l'adresse logique 1215.

Dans quel bloc se trouve cette information ? Justifiez votre réponse. Calculez la valeur du déplacement à effectuer, depuis le début de ce bloc, pour accéder aux données dont l'adresse logique est 1215.

1215 = 4 x 265 + 191. L'information recherchée se trouve donc dans la quatrième page du processus : P3. P3 se trouve, d'après la table des pages du processus, dans le bloc B1. L'information se trouve donc dans le bloc B1, et il faudra se décaler de 191 adresses à partir du début de ce bloc pour trouver les données qui se trouvent à l'adresse logique 1215.

Question 9 (3 points)

On suppose que le processus P utilise les pages dans l'ordre qui suit :

P0,P1,P0,P2,P3,P5,P4,P3,P4,P5,P2

Au départ, tous les blocs mémoires sont libres et on allouera dans l'ordre B0, puis B1, puis B2. Représentez dans le tableau en annexe B (à rendre) l'état de la table des pages après chaque accès de la liste précédente. On considère la politique de remplacement LRU. Combien de défauts de pages comptez-vous ? Voir annexe B pour la correction.

Nom, prénom, groupe :
Numéro de table :

Annexe : Support de réponse

A. Code pour la question sur la synchronisation

Solution question 4 en rouge, question 5 en bleu.

```
SortirPont()
{
  P(XU);
  /*** Sortir du pont ***/
  Voitures--;
  /*** Debloquer les voitures qui attendent dans
  le meme sens, s'il y en a.
  Si plus aucune voiture n'attend dans le même
  sens, faire passer celles qui attendent dans
  l'autre sens (s'il y en a) ***/
  if (NbEnAttente[SensPassage] > 0)
  {
    /* A completer */
    if(NbEnAttente[1-SensPassage]==0)
      V(VouloirPasser[SensPassage]);
    else if(Voiture==0)
      V(VouloirPasser[1-SensPassage]);


  }
  else if (Voitures == 0)
  {
    /* A completer */

    if(NbAttente[1-SensPassage]>0)
      V(VouloirPasser[1-SensPassage]);

  }
  V(XU);
}
```

B. Tableau à remplir pour la question sur la mémoire

Entrée de la table de pages	P0	P1	P0	P2	P3	P5	P4	P3	P4	P5	P2
0	B0	B0	B0	B0	B0						
1		B1	B1	B1							
2				B2	B2	B2					B1
3					B1	B1	B1	B1	B1	B1	
4							B2	B2	B2	B2	
5						B0	B0	B0	B0	B0	B0

Défaut de page ?	Oui	Oui	Non	Oui	Oui	Oui	Oui	Non	Non	Non	Oui

Nom, prénom, groupe :
Numéro de table :

C. Code pour la question sur les signaux

```
1  #include <stdio.h>
2  #include <signal.h>
3  #include <unistd.h>
4
5  void on_timeout(int num_sig);
6  void process_picture_precise(char * inputpath, char *outputpath);
7  void process_picture_fast(char * inputpath, char *outputpath);
8  char completed = 0;
9  /* A completer */
10
11 sigjmp_buf context;
12
13 int main(int argc, char *argv[]){
14   int timeout=0;
15   if(argc!=4)
16   {
17     printf("ERROR, expected usage:\n");
18     printf("\t%s input_pic_path output_pic_path timeout_value (sec.)",argv[0]);
19   }
20   signal(SIG_ALRM, on_timeout);
21   timeout=atoi(argv[3]);
22   alarm(timeout);
23   /* A completer */
24   int ret = sigsetjmp(context, 0);
25   if(ret == 0)
26   {
27
28   process_picture_precise(argv[1], argv[2]) ;
29   completed=1 ;
30   /* A compléter */
31   } else {
32     process_picture_fast(argv[1], argv[2]) ;
33   }
34
35   return 0;
36 }
37
38 void on_timeout(int num_sig)
39 {
40   if(completed==1)
41   {
42     Completed=0;
43     return;
44   }
45   else
46   {
47     /* A completer */
48     siglongjmp(context, num_sig);
49
50
51   }
52 }
```