+1/1/60+

INF104
Examen du 04/04/2016

Nom, prénom, n° de table :

.....

Durée : 1h30.

Aucun document n'est autorisé. L'usage de la calculatrice est interdit.

Les premières questions sont des questions à choix multiples. Chaque question n'a qu'une seule bonne réponse parmi celles proposées. Chaque question vaut 0.5 ou 1 point. Il n'est pas demandé de justifier les réponses. Les questions sont notées sur 0.5 point ou 1 point. Si vous ne répondez correctement, vous obtenez 0.5 point ou 1 point. Si vous ne répondez pas, vous obtenez 0 point. Si vous répondez faux, vous obtenez -0.5 ou -1 point selon la question.

---

1 Questions à choix multiple (4 points)

Listing 1: Exemple de code avec fork

```
#include <stdio.h>

int main(int argc, char * argv[]){
  int f = fork();
  printf("Executing process %d, fork returned %d\n", getpid(), f);
}
```

Question 1 (0.5 point) On considère le code donné au listing 149. Parmi les affichages suivants, lequel est impossible ?

- ☐ Executing process 2016, fork returned 2017
  Executing process 2017, fork returned 0
- ☐ Executing process 2017, fork returned 0
  Executing process 2016, fork returned 2017
- ☑ Executing process 2016, fork returned 0
  Executing process 2016, fork returned 2017
- ☐ Tous les affichages proposés sont possibles

Question 2 (0.5 point) L'exécution de la fonction exec a pour effet:

- ☐ De créer un nouveau processus et d'exécuter une commande dans ce processus
- ☐ De charger une bibliothèque dynamique (*dynamic library*) en mémoire pour pouvoir l'exécuter
- ☑ De ré-initialiser les zones mémoires (code, données, sauvegardes de registres) associées à un processus
- ☐ Toutes les réponses proposées sont fausses

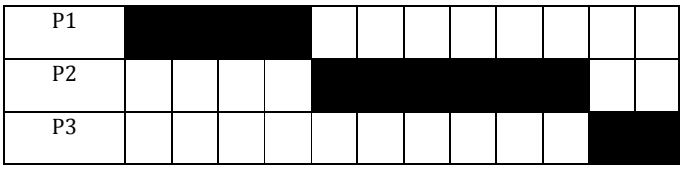
Question 3 (0.5 point) Après émission d'un signal, celui-ci est-il traité par le récepteur:

- ☐ Au moment ou le processeur remarque un variation sur le bus d'interruption
- ☐ Immédiatement, c'est à dire au moment même où l'émetteur appelle la fonction kill
- ☐ Après un délai fixe, déterminé dans le code source (en langage C) du système d'exploitation
- ☑ Toutes les réponses proposées sont fausses

Question 4 **(0.5 point)** La commande “kill -2 2016” a pour effet :

- ☐ d’arrêter tous les processus utilisateurs (-2) lancés en 2016 sur la machine (date 2016)
- ☐ de tuer jusqu’à deux processus fils du processus dont l’identifiant est 2016
- ☑ d’envoyer le signal 2 au processus dont l’identifiant est 2016
- ☐ Toutes les réponses proposées sont fausses

On considère 3 processus, P1, P2, et P3, dont les dates de démarrage sont respectivement 0, 2, et 4 quantums de temps. On suppose que P1 a une durée de 4 quantum de temps, P2 a une durée de 6 unités de temps, et P3 a une durée de 2 unités de temps. Le chronogramme ci-dessous, qui illustre l’ordonnancement des processus P1, P2, et P3.



Question 5 **(0.5 point)** Quelle politique d’ordonnancement permet d’aboutir à ce chronogramme

- ☑ Premier arrivé premier servi (First Come First Served)
- ☐ Le tourniquet (Round Robin)
- ☐ Le plus court d’abord (Shortest Job First)
- ☐ Aucune des réponses proposées ne convient

Question 6 **(1 point)** En considérant le même scénario d’ordonnancement que précédemment, quel est le temps moyen de service?

- ☐ 6 quantum de temps
- ☐ 8,44 quantum de temps
- ☑ 6,67 quantum de temps
- ☐ 4 quantum de temps
- ☐ Aucune des réponses proposées ne convient

Question 7 **(0.5 point)** Un défaut de page (page fault) se produit

- ☐ Lorsque le système ne trouve plus suffisament d’espace disque pour stocker le contenu d’un fichier
- ☐ Lorsqu’un fichier utilise un trop grand nombre de blocs disque
- ☐ Lorsqu’un processus essaie d’accéder à un espace mémoire qui ne lui appartient pas
- ☑ Lorsqu’un processus essaie d’accéder à du code ou à des données qui ne sont pas stockés en mémoire, mais sur le disque
- ☐ Aucune des réponses proposées ne convient

2 Questions sur le système de gestion de fichiers (4 points)

Question 8 **(2 points)** Sur un système UNIX, un répertoire est un fichier qui contient des informations permettant de faire la correspondance entre une structuration logique des fichiers (de type arbre) et une structuration physique du disque (blocs de mémoire contigus). En complément, la `i-list` stocke l'ensemble des `i-nodes`, et un `i-node` est un descripteur de fichier. En décrivant brièvement le contenu d'un fichier de type répertoire, ainsi que le contenu d'un `i-node`, expliquer comment est implémentée la correspondance entre structurations logique et physique de l'espace disque.

☐0 ☐0.5 ☐1 ☐1.5 ☐2

L'i-node contient les adresses des blocs du disque sur lesquels sont stockés les différentes parties d'un fichier.

Un répertoire R est un fichier qui contient les i-node des fichiers (qui peuvent aussi être des répertoires) contenus dans R.

Le lien entre le contenu d'un répertoire et les i-node assure la structuration logique du système de fichiers; le lient entre i-node et blocs disque assure la structuration physique du système de fichier. L'i-node est donc l'élément qui assure le lient entre structuration logique et structuration physique.

Question 9 **(2 points)** On considère un système de fichier UNIX tel que celui vu en cours. On suppose qu'un bloc d'information sur le disque permet de stocker 128 octets, et qu'une adresse de bloc disque est encodée sur 4 octets. Quelle est la taille maximale d'un fichier dans cette configuration? Il n'est pas nécessaire de mener les calculs jusqu'au bout: vous devez développer tous les arguments et vous pouvez laisser les calculs sous une forme du type $(128*25)/2+128^2$....

☐0 ☐0.5 ☐1 ☐1.5 ☐2

Nombre de blocs accessibles directement: 10x128
Avec 1 niveau d'indirection: (128^2) / 4
Avec 2 niveaux d'indirection: 128x(32^2)
Avec 3 niveaux d'indirection: 128x(32^3)

la taille max m est: 10x128 + (128^2) / 4 + 128x(32^2) + 128x(32^3)

3 Questions sur la pagination (4 points)

On considère un processus nécessitant l'utilisation de 7 pages en mémoire. Le tableau ci-dessous donne la correspondance entre pages logiques et pages physiques.

Page logique	Page physique
0	4
1	5
2	1
3	9
4	7
5	6
6	2

Question 10 **(1 point)** En supposant que la taille d'une page est de 1 Ko (1024 octets; $1024 = 2^{10}$), quelle est l'adresse physique correspondant à l'adresse logique 0x05F2 (format hexadecimal).

Donnez votre réponse au format hexadécimal. ☐0 ☐0.5 ☐1

0x05F2 = 0000 0101 1111 0010

page logique = 0000 01, ce qui correspond à la page physique 5.

0001 0101 1111 0010, ce qui correspond en hexadecimal à l'adresse physique 0x15F2

Question 11 **(1 point)** En supposant que la taille d'une page est de 0.5 Ko (512 octets; $512 = 2^9$), quelle est l'adresse logique correspondant à l'adresse physique 0x072E (format hexadecimal).

Donnez votre réponse au format hexadécimal. ☐0 ☐0.5 ☐1

0x72E = 0000 0111 0010 1110

page physique = 0000 011. Cette page n'est pas occupée par le processus considéré: on ne peut pas savoir quelle page logique occupe la page physique.

Question 12 **(1 point)** En lien avec la pagination, donnez la définition de la politique LRU. Dans quelle cas est-elle utilisée? ☐0 ☐0.5 ☐1

LRU : Least Recently Used, signifie que lorsque le système d'exploitation doit remplacer la page d'un processus par la page d'un autre processus il choisira la page qui a été utilisée (lue ou écrite) pour la dernière fois le moins récemment (c'est à dire avec la date d'utilisation la plus ancienne). Cette politique est utilisée en cas de défaut de page nécessitant un remplacement de page (mémoire saturée).

Question 13 **(1 point)** En lien avec la pagination, expliquez à quoi correspond une adresse logique, physique, virtuelle? ☐0 ☐0.5 ☐1

Adresse logique: adresse des données et instructions d'un programme qui n'est pas encore chargé en mémoire (dans le fichier binaire obtenu après édition de liens par gc). Ces adresses sont définies par section (.bss, .data, etc.) et commencent à 0.
Adresse physique: adresse des emplacements d'un composant matériel utilisable par le processeur (RAM).
Adresse virtuelle: adresse sur le disque contenant des données et/ou le code d'un processus.

4 La disparition (2 points)

Question 14 **(1 point)** En suivant la philosophie de l'oeuvre de Pérec, nous allons supprimer la lettre 'e' du texte contenu dans un fichier. En utilisant les fonctions `open`, `read`, et `write`, écrivez un petit programme qui supprime toutes les occurrences de la lettre 'e' d'un texte contenu dans le fichier nommé "mon_lipogramme.in". Le résultat doit être stocké dans un fichier nommé "mon_lipogramme.out".

☐0 ☐0.5 ☐1 ☐1.5 ☐2

```
int main(int arcgc, char * argv[])
{
  int infile  = open("mon_lipogramme.in", O_RDONLY);
  int outfile = open("mon_lipogramme.out", O_WRONLY);
  char c;
  while (read(infile, &c, 1) == 1)
    if (c != 'e') write(outfile, &c, 1);
}
```

5 Au dela de la barrière (6 points)

Nous nous intéressons ici à l'implémentation d'une barrière. Le principe de la barrière est simple: les processus s'attendent mutuellement jusqu'à ce que le dernier processus arrive. Ce dernier processus libère tous les processus en attente, qui reprennent leur exécution. L'objectif est de s'assurer que tous les processus ont atteint un certain point dans le code avant qu'ils continuent à s'exécuter. Le canevas de code à compléter est fourni ci-dessous. Les commentaires dans le code identifient les parties à compléter, comme indiqué dans les questions ci-dessous.

Dans ce code, nous supposons que le nombre de processus à synchroniser sur la barrière est donné par la macro `MAX_PROCESSES`. Le verrou `lock_for_shared_variables` sert à protéger l'accès aux variables partagées.

Le nombre de processus ayant atteint la barrière est stocké dans un fichier partagé. Les fonctions `incr_reached` et `decr_reached` ont pour effet d'incrémenter et de décrémenter le nombre de processus qui ont atteint la barrière, et d'enregistrer le résultat dans le fichier partagé. La fonction `get_reached` lit ce fichier et retourne le nombre de processus qui ont atteint la barrière. Le code de ces fonctions est ignoré pour simplifier le code à compléter, nous précisons simplement que leur

exécution n'est pas atomique.

Question 15 (1 point) ☐0 ☐0.5 ☐1

Utilisez les fonctions `sem_post` et `sem_wait` pour protéger l'accès au fichier partagé qui comptabilise le nombre de processus ayant atteint la barrière. Les parties du code à compléter sont identifiées avec le commentaire `Protect shared variables`.

Question 16 (2 points) ☐0 ☐0.5 ☐1 ☐1.5 ☐2

Déclarez et initialisez le (ou les) sémaphore(s) permettant d'implémenter le mécanisme de la barrière.

Utilisez la fonction `sem_wait` et le(s) sémaphore(s) de la barrière pour bloquer les processus qui arrivent sur la barrière.

Utilisez le signal `SIGUSR1` pour déclencher l'ouverture de la barrière: le dernier processus arrivé sur la barrière envoie le signal `SIGUSR1` à tous les autres processus. Les PID de ces processus peuvent être récupérés en utilisant la fonction `int get_child_pid(int i)` qui renvoie le PID du i-ème fils du processus principal (le processus principal est celui qui exécute la fonction `main`). L'utilisation de cette fonction vous est fournie ligne 92.

Implémentez la fonction `on_usr1` qui libère les processus en attente sur la barrière, et faites en sorte que cette fonction soit appelée lorque le processus reçoit un signal `SIGUSR1`.

Les parties du code à compléter sont identifiées avec le commentaire `Barrier implementation`.

Question 17 (3 points) ☐0 ☐0.5 ☐1 ☐1.5 ☐2 ☐2.5 ☐3

Pour aller plus loin, nous voulons borner le temps d'attente des processus à la barrière: un processus ne doit pas rester bloqué plus de 5 secondes. Utilisez le signal `SIGALRM` pour vous assurer qu'un processus est réveillé au bout de 5 secondes d'attente. La fonction `on_timeout` sera appelée sur échéance du délai de 5 secondes. Lorsque cette fonction est exécutée, seul le processus qui a reçu le signal SIGALRM doit pouvoir quitter la barrière. Or l'ordre dans lequel les processus sont débloqués ou sortis de la file d'attente d'un sémaphore n'est pas connu à priori. Sur réception du signal SIGALRM, nous devrons donc réveiller tous les processus en attente et remettre en attente ceux qui ne sont pas concernés par cet évènement. Ceci explique la présence de la boucle `while` ligne 117.

Les parties de code à compléter sont identifiées avec le commentaire `Timeout implementation`.

```
1  #define MAX_PROCESSES 100
2  int processes_pid[MAX_PROCESSES];
3
4  #define DOWN -1;
5  #define TIMEOUT 0
6  #define UP 1
7  char barrier_state = DOWN;
8
9  sem_t * lock_for_shared_variables;
10
11
12 sem_t * sem_for_barrier;
13                                                  // Barrier implementation
14
15
16 int get_reached(); // returns the number of process at the barrier
17 void incr_reached(); // increment (+1) the number of process at the barrier
18 void decr_reached(); // decrement (-1) the number of process at the barrier
19
20 void child_function(); // function executed by processes
21 void on_timeout(int sig_nb); // signal handler (SIGALRM)
22 void on_usr1(int sig_nb); // signal handler (SIGUSR1)
23
24 int get_child_pid(int i); // returns the PID of the i-th child of the main process
25 void store_pid_in_file(int i, int pid); // store in file the map i-th child --> PID
26
27 int main(int argc, char *argv[]) {
28   /* Create and initialize semaphores */
29   lock_for_shared_variables = sem_open("/sem1", O_CREAT, 0644, 1);
30
31   sem_for_barrier = sem_open("/sem2", O_CREAT, 0644, 0);
32
33                                                  // Barrier implementation
34
35
36
37   /*** Creation of child processes ***/
38   for (i = 0; i < MAX_PROCESSES ; i++)  {
39     int f = fork();
40     if(f!=0) {
41       store_pid_in_file(i, f);
42     }
43     else {
44       child_function();
45     }
46   }
47
48   /* Wait for the end of all the processes */
49   while (wait(&etat) != -1)
50     printf("main - end of child process with state: %04x (hexa)\n", etat);
51   // Deletion of semaphores is ignored
52 }
53
54 void on_timeout(int sig_nb)
55 {
56   int i;
57   barrier_state = TIMEOUT;
58
59   sem_wait(lock_for_shared_variables);          // Protect shared variables
60
61   int reached_nb = get_reached();
62
63   sem_post(lock_for_shared_variables);          // Protect shared variables
64
65   for(i=0; i < reached_nb; i++)
66     sem_post(sem_for_barrier);                  // Timeout implementation
67
68
69
70 }
```

```
71
72  void on_usr1(int sig_nb)
73  {
74    barrier_state = UP;
75
76
77    sem_post(sem_for_barrier);                    // Barrier implementation
78
79
80  }
81
82
83  void child_function(){
84
85    sem_wait(lock_for_shared_variables);          // Protect shared variables
86    int reached_nb = get_reached();
87
88    if  (reached_nb == MAX_PROCESSES - 1) {
89      int i;
90      for(i=0; i<reached_nb; i++){
91        /* Release an process present at the barrier */
92        int pid = get_process_pid(i);
93
94        kill(pid, SIGUSR1);                       // Barrier implementation
95
96
97        /* Decrement the number of processes at the barrier */
98        decr_reached();
99
100     }
101
102     sem_post(lock_for_shared_variables);         // Protect shared variables
103
104   }
105   else {
106     /* Increment the number of processes that reached the barrier */
107     incr_reached();
108
109     /* process should be wating at the barrier */
110
111     signal(SIGALRM, on_timeout);                 // Timeout implementation
112     alarm(5);
113     sem_post(lock_for_shared_variables);
114                                                  // Protect shared variables
115
116     while(barrier_state==DOWN) {
117
118       signal(SIGUSR1, on_usr1);                  // Barrier implementation
119       sem_wait(sem_for_barrier);
120     }
121   }
122 }
123
```