Nom, Prénom :

Examen de INF104, 18/04/2015

1h30 – Sans documents ni calculatrice

Note : Pour les questions B.1, B.2, C.2, D.1 et D.2, vous pouvez répondre sur le sujet d'examen pour gagner du temps. Indiquez le sur votre copie et n'oubliez pas de mettre votre nom et prénom sur les pages du sujet que vous nous rendez.

A - Questions de cours (6 points)

Question A.1 (1 point) : Définissez, en une phrase, ce qu'est un processus ?

Un processus est un programme chargé en mémoire pour être exécuté.

Question A.2 (2 points) : Nommez, et définissez en une phrase chacun, les trois états associés à un processus. Quelles sont les transitions possibles entre ces états ?

Etats possible : prêt, actif, bloqué

Etat prêt : le processus peut être exécuté mais il n'a pas été sélectionné par l'ordonnanceur pour être exécuté. Il est en attente, et sera exécuté lorsque l'ordonnanceur l'aura sélectionné.

Etat actif : le processus est en cours d'exécution sur le processeur (ou un des cœurs du processeur).

Etat bloqué : le processus est bloqué et ne peut pas être exécuté. Par exemple, il peut être bloqué en attente qu'une ressource se libère, via un sémaphore ou une entrée/sortie.

Transitions possibles : prêt ← → actif ; actif → bloqué ; bloqué → prêt.

Question A.3 (2 points) : décrivez brièvement le fonctionnement de la fonction *fork* : paramètre(s) d'entrée, de retour, et effet de la fonction. Décrivez en quelques lignes l'état de la mémoire après exécution de la fonction *fork*.

La fonction fork permet de créer un nouveau processus, appelé fils, à partir d'un processus donné appelé père. fork ne prend pas de paramètre en entré et renvoie : 0 dans la zone mémoire associée au fils ; l'identifiant du fils dans la zone mémoire associée au père ; -1 en cas d'erreur à l'exécution.

Après exécution de fork, une zone de donnée a été crée pour le fils, il s'agit d'une copie de la zone de donnée de son père (sauf pour la valeur de retour du père). La mémoire associée au code est partagée entre le père et le fils. Chaque processus a son propre contexte, et le compteur ordinal du père et du fils pointe vers la même instruction : celle qui suit la fin de l'exécution de la fonction fork.

Question A.4 (1 point) : décrivez brièvement l'effet d'un appel à la fonction *exec*.

La fonction exec remplace les zones mémoires associées à un processus par les données, et code associé à un programme exécutable sur le disque (le chemin d'accès à cet exécutable est passé en paramètre de la fonction exec).

Nom, Prénom :

B - Exercice sur les signaux (4 points)

On souhaite faire un long calcul représenté par un appel à la fonction ***compute()*** dans le programme ci-dessous. Le code de la fonction ***compute*** n'étant pas fourni, on supposera qu'il existe et que l'on ne sait pas bien estimer le temps de calcul associé à cette fonction. On suppose également que le code fournit ci-dessous compile sans warning (bien que l'intégralité du code n'est pas fourni).

```
1   int result = 0;
2   int mode = 1;
3   void printResult (int sig){
4     printf ("result = %d\n", result);
5     // Question B1: to be completed
6     alarm (PERIOD);
7     signal (sig, printResult);
8   }
9
10  void changeMode (int sig) {
11    mode = 1 - mode;
12    printf ("mode = %d\n", mode);
13    // Question B2: to be completed
14    if (mode) {signal (SIGALRM, printResult); alarm(PERIOD);}
15    else signal (SIGALRM, SIG_IGN);
16    signal (sig, changeMode);
17  }
18
19  int main (int argc, char *argv[]) {
20    // Question B1 and B2: to be completed
21    signal (SIGALRM, printResult); // B.1
22    alarm (PERIOD);                // B.1
23    signal (SIGINT, changeMode);   // B.2
24    compute() ;
25  }
```

Nom, Prénom :

Question B.1 (2 points) : On veut afficher périodiquement le résultat intermédiaire de ce calcul stocké dans la variable ***result***. La période en secondes sera notée PERIOD. On souhaite une solution utilisant les signaux. Vous ne ferez pas d'hypothèse sur le fait qu'un traitement associé au signal s'applique à toutes ou une seule occurrence d'un signal : il faut que votre solution fonctionne dans les deux cas.

Complétez le code ci-dessus pour répondre à cette question.

Question B.2 (2 points) : On veut désormais qu'en tapant ctrl-C lors de l'exécution du processus, on puisse alternativement masquer et démasquer cet affichage. Par contre, le mécanisme périodique reste actif et en fonction du mode choisi, l'affichage sera effectif ou non. La variable *mode* vaudra 1 lorsque l'affichage est effectif.

On rappelle que ctrl-C produit le signal SIGINT envoyé au processus. Vous ne ferez pas d'hypothèse sur le fait qu'un traitement associé au signal s'applique à une ou toutes les occurrences d'un signal : il faut que votre solution fonctionne dans les deux cas.

Complétez le code ci-dessus pour répondre à cette question.

C – Exercice sur la pagination de la mémoire (4 points)

Nous considérons un système d'exploitation allouant la mémoire par pages, et un processeur supportant l'adressage virtuel. Nous supposons de plus que :

- le nombre de pages requis pour un processus est fixé au chargement du processus (i.e. l'allocation dynamique se fait dans un nombre de page prévu à l'avance).
- les adresses physiques sont codées sur 32 bits et permettent d'adresser chaque octet. Nous considérons la convention suivante pour identifier les bits définissant l'adresse : les 32 bits sont numérotés de droite à gauche $b_{31} \dots b_0$ de telle sorte que l'adresse désigne la position $\sum b_i * 2^i$

Question C.1 (1 point) : En considérant des pages de 4ko, indiquez comment déduire la valeur du numéro de page physique à partir de l'adresse physique complète (une formule, un diagramme, ou une méthode bien expliquée sont acceptés).

Le numéro de page physique est déduit de la formule suivante : $\sum b_i * 2^{i-12}$

Réponse alternative 1: le numéro de page physique est déterminé par les 20 bits de poids fort (ceux qui se trouvent à gauche dans la représentation binaire proposée)

Réponse alternative 2 : un dessin où les 20 bit de poids fort (ceux qui se trouvent à gauche dans la représentation binaire proposée) sont identifiés comme engendrant le n° de page physique.

Question C.2 (2 points) : Nous supposons désormais que la mémoire physique est limitée à 8 pages (e.g. 32 Ko de mémoire). Les pages sont numérotées de 0 à 7. Nous supposons que le système d'exploitation utilise les 5 premières pages et que ces pages sont verrouillées en mémoire (elle ne peuvent être réallouées). Il reste donc 3 pages physiques pour faire tourner les processus.

Le processus A est le seul à être chargé et exécuté. Il utilise 5 pages virtuelles - pv. On suppose qu'au moment du chargement de A, seule la page 0 est en mémoire physique.

Nom, Prénom :

L'évolution de sa table des pages est décrite dans le tableau ci dessous.

- La première ligne représente la séquence de pages virtuelles accédées.
- Puis, ligne par ligne l'état de la table des pages de A est indiqué pour la page virtuelle n° i. La valeur nv indique une page sur le disque, alors qu'une valeur numérique correspond à un numéro de page physique.

N° Page accédée	0	1	0	2	4	0	1
pv 0	5	5	5	5	5	5	5
pv 1	nv	6	6	6	nv	nv	7
pv 2	nv	nv	nv	7	7	7	nv
pv 3	nv	nv	nv	nv	nv	nv	nv
pv 4	nv	nv	nv	nv	6	6	6

Complétez ce tableau en supposant une politique LRU de remplacement de pages. Indiquez le nombre de défauts de page observés

4 défauts de page

Question C.3 (1 point)

La pagination vous a été présentée en cours comme un moyen de limiter le gaspillage de la mémoire physique en limitant la fragmentation de la mémoire. La pagination peut malgré tout engendrer un gaspillage de mémoire de moindre amplitude. Expliquez pourquoi.

La pagination peut entrainer un gaspillage de la mémoire à cause de la taille associée à une page qui, si elle est grande devant l'espace mémoire requis par les processus, peut entrainer des gaspillages au moment où les pages virtuelles (partiellement remplies) occupent des pages physiques.

Nom, Prénom :

D – Problème sur la synchronisation (6 points)

Nous considérons un problème de synchronisation dans un centre de tri postal automatisé. Des *robots* déposent des paquets de lettres dans des paniers. Une *machine* de tri sort les lettres du panier afin de les trier. Plusieurs robots (variable **numRobot**) circulent dans le centre et plusieurs machines (variable **numMachines**) sont disponibles. Chaque machine est reliée à un panier (paramètre **myBasket**). Un robot choisit le panier dans lequel il déposera son paquet de lettres (fonction **getBasket**).

Un robot ne doit pas déposer son paquet de lettres dans un panier alors que la machine sort une lettre du panier. De même, une machine ne peut sortir une lettre alors qu'un robot est en train de déposer un paquet de lettres.

Le code ci-dessous modélise les activités au sein du centre de tri sous forme de processus qui exécutent les fonctions **machine** et **robot**. Certaines fonctions sont à compléter en répondant aux questions ci-dessous. D'autres fonctions sont supposées implémentés, on donne la signature de ces fonctions et la description de leur fonctionnement en commentaire.

Question D.1 (3.5 points) : Déclarez et initialisez un tableau de sémaphores permettant de synchroniser la dépose des lettres par des robots et la récupération des lettres par des machines (complétez la fonction **createSemaphores**). Vous utiliserez pour cela la fonction **sem_open**. Notez que le nombre des machines/robots est donné par **numMachines** et **numRobot** respectivement. Ces nombres sont toujours inférieurs à **MAXN** (voir la directive #define).

Utilisez les sémaphores et les fonctions **sem_wait** et **sem_post** afin de garantir l'exclusion mutuelle entre un robot (qui dépose un paquet de lettres) et une machine (qui sort une lettre de son panier). De plus, le robot vérifie, via la fonction **getBasket**, que le panier ne déborde pas. Si le panier est trop plein le robot en choisit un autre.

Les parties du code qui sont potentiellement à compléter sont identifiées avec le commentaire **// Synchronization**.

Question D.2 (2.5 points) : Il s'avère que souvent durant une journée le panier de certaines machines se vide complètement, alors que d'autres machines disposent encore de lettres dans leurs paniers. Évidement une machine avec un panier vide ne peut pas trier. Les machines ne sont donc pas utilisées à leur capacité maximale.

Dans le code ci-dessous un troisième type de processus est donc prévu (fonction **balance**). Ce processus choisit deux paniers au hasard et vérifie le niveau de remplissage. Le processus enlève ensuite un certain nombre de lettres du panier le plus remplit et les met dans l'autre panier. La fonction **balance** utilise la fonction **getBasket** qui garantit (grâce à votre réponse à la question précédente) qu'un transfert ne peut jamais être en conflit avec une machine ou un robot – même si plusieurs processus de chaque type (**machine**, **robot**, **balance**) s'exécutent. Cependant, un interblocage (deadlock) peut se produire.

Déclarez et initialisez un ou plusieurs sémaphores en utilisant la fonction **sem_open**. Utilisez le(s) sémaphore(s) et les fonctions **sem_wait** ainsi que **sem_post** afin d'assurer qu'aucun interblocage ne puisse se produire.

Les parties du code qui sont potentiellement à compléter sont identifiées avec le commentaire **// Deadlock**.

Nom, Prénom :

```
1   /*************           Initialization      ************/
2
3   #define MAXN 50
4
5   // Number of machines, robot, and balance processes.
6   int numMachines;
7   int numRobot;
8   int numBalance;
9
10  // Process IDs of machine, robot, and balance processes.
11  int MPIDs[MAXN];
12  int DPIDs[MAXN];
13  int BPIDs[MAXN];
14
15            // Question D.1 Synchronization (declare semaphores using table)
16
17
18  sem_t *SemBaskets[MAXN];
19
20
21                              // Question D.2 Deadlock (declare semaphore(s))
22
23  sem_t *SemConflict;
24
25
26
27
28
29  void createSemaphores() {
30
31                            // Question D.2 Deadlock (initialize semaphore(s))
32
33    SemConflict = sem_open("/semc", O_CREAT | O_EXCL, 0644, 1);
34
35
36
37
38
39    for(int i = 0; i < numMachines; i++) {
40      char semName[8];
41      sprintf(semName, "/semP%02d", i);
42
43                                                 // Question D.1 Synchonization
44                                                // (initialize semaphores using
45                                                //  semName and store in table)
46
47
48      sem_t *tmp = sem_open(semname, O_CREAT | O_EXCL, 0644, 1);
49      SemBaskets[i] = tmp;
50
51
52
53
54    }
55  }
56
57
58
59
60
```

Nom, Prénom :

```
61  /*************           Robot functions       ************/
62
63  // function getNewPack : attach a pack of letters to the robot and
64  // returns its size (number of letters)
65  int getNewPack();
66
67  // procedure doDelivery: delivers "packSize" letters to a "basket".
68  void doDelivery(int basket, int packSize);
69
70  // Function executed by robot processes.
71  void robot(int ID) {
72    while(1) {
73      int packSize = getNewPack();
74
75      // randomly choose a machine's basket.
76      int basket = getBasket(-1, packSize);
77
78      // deliver letters to a machine's basket.
79      doDelivery(basket, packSize);
80
81                              // Question D.1 Synchonization (mutual exclusion)
82
83      sem_post(SemBaskets[basket]) ;
84
85    }
86  }
87
88  // function fitBasket : returns true if "packSize" letters can be
89  // added, false otherwise.
90  char fitBasket(int basket, int packSize);
91
92  // function getBasket : randomly select a basket. The basket has
93  // to be different from the basket indicated by "exclude". There needs
94  // to be enough space in the basket to add "packSize" letters.
95  int getBasket(int exclude, int packSize) {
96    while (1) {
97      int basket = rand() % numMachines; // randomly choose a basket
98
99      if (basket == exclude) // try another basket
100       continue;
101                             // Question D.1 Synchonization (mutual exclusion)
102
103
104     sem_wait(SemBaskets[basket]) ;
105
106
107     // check if letters fit into basket
108     if (!fitBasket(basket, packSize)) {
109                             // Question D.1 Synchonization (mutual exclusion)
110
111       sem_post(SemBaskets[basket]) ;
112
113
114     }
115     else {
116       // letters fit into basket: choose it and return.
117       return basket;
118     }
119   }
120 }
```

Nom, Prénom :

```
121  /*************           Machine functions        ************/
122
123  // procedure takeLetter: take a letter from basket
124  void takeLetter(int basket);
125
126  // Function executed by machine processes.
127  void machine(int myBasket) {
128    while(1) {
129      printf("sort a basket\n");
130                              // Question D.1 Synchonization (mutual exclusion)
131      sem_wait(SemBaskets[mybasket]) ;
132
133
134
135      takeLetter(myBasket);
136                              // Question D.1 Synchonization (mutual exclusion)
137      sem_post(SemBaskets[mybasket]) ;
138
139
140
141      sleep(5) ; // wait for new letters
142    }
143  }
144
145  // procedure checkBalanceAndTransfer : Balance the level between
146  // "basketA" and "basketB". Takes some letters from the basket that is
147  // more full and transfer it into the other basket. Synchronization is NOT
148  // performed by this function.
149  void checkBalanceAndTransfer(int basketA, int basketB);
150
151  /*************           Balance functions        ************/
152
153  // Function executed by balance processes (Question D.2).
154  void balance(int ID) {
155    while(1) {
156                                      // Question D.2 Deadlock (avoid deadlock)
157
158      sem_wait(SemConflict)) ;
159
160
161      // randomly choose two baskets
162      int basketA = getBasket(-1, 0);
163      int basketB = getBasket(basketA, 0);
164
165                                      // Question D.2 Deadlock (avoid deadlock)
166      sem_post(SemConflict) ;
167
168
169
170      checkBalanceAndTransfer(basketA, basketB);
171
172                                      // Question D.2 (end of mutual exclusion)
173
174      sem_post(SemBaskets[basketa]) ;
175      sem_post(SemBaskets[basketb]) ;
176
177
178    }
179  }
180
```

Nom, Prénom :

```
181  /*************           Init and Main functions        ************/
182
183  // procedure createProcesses : creates n child processes (used to
184  // create processes machine, robot, and balance) using fork. Store
185  // process identifiers in table PIDs. f is the identifier of the function
186  // executed by the created process.
187  void createProcesses(int n, int PIDs[MAXN], void (*f)(int ID));
188
189  // procedure waitForChildren : waits for numChildren child processes
190  // to terminate.
191  void waitForChildren(int numChildren);
192
193
194  int main(int argc, char **argv) {
195    // read arguments
196    numMachines = atoi(argv[1]);
197    numRobot = atoi(argv[2]);
198    numBalance = atoi(argv[3]);
199
200    // create and initialize semaphores.
201    createSemaphores();
202
203    // create child processes.
204    createProcesses(numMachines, MPIDs, machine);
205    createProcesses(numRobot, DPIDs, robot);
206    createProcesses(numBalance, BPIDs, balance);
207
208    // wait for processes to terminate.
209    waitForChildren(numMachines + numBalance + numRobot);
210
211    return 0;
212  }
213
```

Nom, Prénom :