Partiel de INF104, 15/04/2015

1h30 – Sans documents

Questions sur la création de processus (2 points)

Question 1 (1 point): Après un appel à la fonction *fork*, l'emplacement mémoire dans lequel le code du processus fils (instructions binaires) est stocké est :
a) un emplacement mémoire différent de celui utilisé pour le père mais contenant une copie du code du père.
b) le même emplacement mémoire que celui utilisé pour le processus père.
c) un emplacement mémoire vierge dédié au chargement du code du processus fils
Répondez en choisissant l'une des possibilités (a, b ou c).

Réponse : b)

Question 2 (1 point) : On souhaite synchroniser un père sur la terminaison de fils **sans utiliser de sémaphore**. Donner la fonction qui permet de faire cela en expliquant son fonctionnement et en décrivant le rôle des paramètres qu'elle prend et retourne.

Réponse : On peut utiliser int wait(int * state) : dans les conditions normaes d'utilisation (sans erreurs) le paramètre de retour et le PID du fils qui vient de se terminer, alors que Etat est un entier qui contient deux octets : 1 pour indiquer la valeur de retour du fils qui vient de se terminer (valeur passée en paramètre de l'appel à exit dans le code du fils) ; l'autre en cas d'arrêt brutal (sur réception d'un événement) du fils indique le numéro de signal qui a provoqué l'arrêt du fils.

Questions sur l'ordonnancement (4 points)

On considère l'exécution de trois processus P1, P2 et P3 ayant les caractéristiques suivantes :

- P1 démarre à la date T=0 seconde, P2 démarre à la date T=1 seconde, et P3 démarre à l'instante T=2 secondes.
- P1 dure 3 secondes, P2 dure 5 secondes et P3 dure 10 secondes.

Question 3 (2 points) : Nous avons représenté sous la forme d'un chronogramme l'exécution de ces processus. Sur ce chronogramme, chaque case représente un intervalle de temps de une seconde. Une case marquée d'un X signifie que le processus est dans l'état « prêt » alors qu'une case noire signifie que le processus est dans l'état « actif ». Donner le nom **des deux** politiques d'ordonnancement (vues en cours) permettant d'obtenir cette exécution quelles que soient les priorités de P1, P2 et P3.

P1	■	■	■														
P2		X	X	■	■	■	■	■									
P3			X	X	X	X	X	X	■	■	■	■	■	■	■	■	■

Réponse : Les politiques d'ordonnancement, vues en cours, qui permettent d'aboutir à ce scénario d'ordonnancement indépendamment des priorités des processus sont : Premier Arrivé Premier Servi (ou First Come First Served) et Plus Court d'Abord (ou Shortest Job First)

Question 4 (2 points) : En suivant la même convention de notation que pour le chronogramme de la question précédente, compléter le chronogramme **fourni en annexe**, pour représenter l'exécution de ces processus : on suppose ici que la politique d'ordonnancement est « **tourniquet par niveau de priorité** » (ou *« Round Robin within priority »*) ; que le quantum de temps associé à cette politique est de 1 seconde ; et que la priorité de P1 est égale à la priorité de P2 et supérieure à la priorité de P3.

Réponse : cf. annexe

Questions sur les signaux (5 points)

Dans cet exercice, nous nous intéressons au code du réveil matin qui vous permet d'arriver à l'heure en cours. Ce réveil possède trois fonctionnalités : 1 – sonner à l'heure de réveil de votre choix, 2 – arrêter de sonner pendant une courte durée, 3 – arrêter de sonner jusqu'à la prochaine heure de réveil. Le code fourni en annexe est un début de code, **à compléter,** pour représenter le fonctionnement du réveil. Pour le compléter, nous utiliserons les signaux suivants :

- SIGALRM, pour représenter l'arrivée de l'heure de réveil (déclencher la sonnerie).
- SIGUSR1, pour stopper temporairement la sonnerie, pendant un court laps de temps. Cette fonctionnalité est appelée « snooze » dans la description du code fourni.
- SIGUSR2, pour stopper la sonnerie jusqu'à la prochaine date de réveil. Cette fonctionnalité est appelée « stop » dans la description du code fourni.

Les indications sur le code fourni, ainsi que les lignes à compléter, sont décrites en commentaire dans le code. Notez que l'espace laissé pour compléter n'est pas indicatif du nombre de lignes à écrire.

Question 5 : Les questions qui suivent vont vous guider pour compléter le code à trous fourni en annexe. Il est attendu que vous n'utilisiez que des fonctions déclarées dans le code ou dans les APIs système que vous connaissez pour gérer les signaux. Vous pouvez répondre directement sur l'annexe fournie.

Question 5.1 (1 point) : complétez le code de la fonction `main` en suivant les indications données en commentaire. Au lancement du programme, il faut ignorer les signaux correspondants aux fonctionnalités snooze et stop : ces deux fonctions ne sont utiles que lorsque le réveil sonne. Enfin, il faut exécuter la fonction « on_alarm » lorsque le réveil sonne.

Réponse : cf. annexe

Question 5.2 (1,5 point) : complétez le code de la fonction « on_`alarm` ». Lorsque cette fonction est exécutée, cela signifie que le réveil sonne. On pourra donc l'arrêter ou demander à ce que le réveil sonne après un petit délai (fonctionnalité snooze). Tant que le réveil sonne, l'utilisateur peut utiliser cette fonctionnalité snooze. Complétez le code de la boucle while en conséquence.

Réponse : cf. annexe

Question 5.3 (1,5 point) : complétez le code de la fonction `on_stop` de sorte que le réveil sonne de nouveau après un délai obtenu en appelant la fonction `get_next_alarm_time`.

Réponse : cf. annexe

Question 6 (1 point) : quelle commande peut-on utiliser pour simuler le comportement d'un utilisateur qui active la fonctionnalité d'arrêt temporaire de la sonnerie (fonctionnalité snooze) ou d'arrêt de la sonnerie jusqu'à la prochaine date de réveil (fonctionnalité stop) ? On suppose que le signal SIGUSR1 et SIGUSR2 portent respectivement les numéro 30 et 31, et que la trace d'exécution commence par :

```
PID: 3065
RINGING !!!!!!
```

Réponse :
snooze : kill -30 3065
stop : kill -31 3065

Questions sur la pagination de la mémoire (5 points)

Question 7 (1 point) : Donner la définition d'un défaut de page.

Réponse : Un défaut de page se produit lorsqu'un processus a besoin de référencer une information qui ne se trouve pas en mémoire. La page logique sur laquelle se trouve cette information devra donc être chargé ou rechargé sur une page physique.

Question 8 (1 point) : en pagination, que signifie LRU ? Donnez sa définition et expliquez dans quel type de situation LRU est utilisé.

Réponse : LRU est un algorithme de remplacement de page. Le sigle LRU signifie *Least Recently Used :* avec cet algorithme, la page qui sera remplacée est celle dont la dernière date d'utilisation est la plus ancienne.

Question 9 : On considère, sur une machine 16 bits (une adresse est encodée sur 16 bits), un espace mémoire physique de 1 Kilo-octets divisé en 4 pages de 256 Octets (pour rappel, 256 = 2^8). On note ces pages P1, P2, P3, et P4. Le système d'exploitation implémente la pagination avec LRU. On souhaite exécuter un programme nécessitant 8 Ko, soit 32 pages logiques.

Question 9.a (1 point) : En supposant que les pages peuvent-être virtualisées sans limite d'espace de virtualisation, est-il possible d'exécuter un programme constitué de 32 pages logiques ? **Justifiez votre réponse en donnant le nombre maximum de pages logiques** que l'on peut utiliser pour un processus compte tenu de la configuration proposée.

Réponse : La réponse est oui : le nombre maximum de pages logiques que l'on peut utiliser est 2^8, soit 256. En effet, sur 16 bit d'adresse, 8 sont utilisés pour encoder les déplacements dans les pages et 8 sont alors utilisables pour encoder les numéros de page.

Question 9.b (2 points) : On suppose que le programme fait une séquence d'accès aux pages comme suit :

0,1,3,4,3,1,5,3,1,6,0,6,4

Remplir le tableau fourni en annexe pour représenter le contenu de la table des pages pendant l'exécution de ce programme. L'initialisation du tableau est fournie pour vous aider à démarrer.

Réponse : cf. annexe

Questions sur la synchronisation (4 points)

Un programmeur a conçu une application de simulation relativement gourmande en puissance de calcul. Après analyse de son code, il se rend compte qu'il peut décomposer les calculs en 5 sous fonctions : f1, f2, f3, f4, f5. Ces fonctions n'utilisent que des variables locales où leurs paramètres d'entrée. On supposera que toutes ces fonctions prennent un ou plusieurs entiers en paramètre, et retournent un entier comme résultat (pour simplifier).

La logique d'exécution du programme qu'il a conçu est la suivante : au début de l'exécution 3 variables X Y et Z sont définies et servent à stocker les données d'entrée de la simulation. La simulation doit fournir les valeurs (R1 et R2) associées résultats de deux calculs : f4(f1(X), f2(Y)) et f5(f2(Y), f3(Z)). Une première implémentation de ce simulateur, **séquentielle**, repose sur le code suivant :

```
int main (){
  printf ("R1 vaut %d", f4(f1(X),f2(Y))) ;
  printf("R2 vaut %d", f5(f2(Y),f3(Z))) ;
  return 0 ;
}
```

Le programmeur pense qu'il serait utile de paralléliser les traitements pour accélérer ses simulations. Il propose de répartir les traitements dans 3 processus **partageant les variables Aux1 et Aux2**. Le code de ces trois processus est donné ci-après. Dans ce code, on fera l'hypothèse que **les variables partagées Aux1 et Aux2 sont initialisée à 0** (avant leur utilisation dans le code ci-après). Le programmeur utilise aussi 3 variables locales (non partagées) : R1, R2 et RESLOC. On supposera que **les variables X, Y, et Z sont correctement initialisées au démarrage de l'exécution des processus**.

Le programmeur propose le code suivant :

Processus 1	Processus 2	Processus 3
1. void main(){ 2. RESLOC=f1(X) 3. R1=f4(RESLOC,Aux1) ; 4. printf("R1 vaut :"); 5. printf("%d\n",R1); 6. }	7. void main(){ 8. Aux1= f2(Y) ; 9. R2=f5(Aux1,Aux2) ; 10. printf("R2 vaut :") ; 11. printf("%d\n",R2) ; 12. }	13. void main(){ 14. Aux2=f3(Z) ; 15. }

Le développeur lance les trois processus simultanément et constate que les calculs ne correspondent pas du tout à ce qui était obtenu dans la version séquentielle.

Question 10 (1 point) : Expliquez pourquoi cette mise en parallèle des calculs **peut aboutir à des valeurs fausses pour R1 et R2**. Illustrez en proposant un scénario d'exécution amenant à des calculs faux.

Réponse : Les variables aux1 et aux2 sont partagées entre les processus. Elles sont écrites dans l'un et lues dans un autre. Il faut donc s'assurer que lors de la lecture, l'écriture a bien déjà eu lieu. L'exécution entrelacée des lignes suivantes amène à une valeur incorrecte de R1 et R2 : 1 à 5 puis 7 à 11 puis 13 à 15 cela donne R1= f4(f1(X), 0) et R2= f5(f2(Y), 0).

Question 11 (1 point) : Un de ses collègues lui dit qu'un premier problème vient du code utilisé pour l'affichage : ce collègue pense que même si par chance les valeurs de R1 et R2 sont correctes, **l'affichage peut aboutir à un résultat impossible à interpréter**. **Ce collègue a raison ; expliquez pourquoi.**

Réponse : Supposons que R1 vaut 12 et R2 vaut 14. L'exécution suivante est possible : ligne 4 puis lignes 10 puis ligne 5 puis ligne 11. Cela donne l'affichage

R1 vaut : R2 vaut : 12
14

Il en est de même de l'exécution ligne 4 puis 10 puis 11 puis 5 qui se traduit par
R1 vaut : R2 vaut : 14
12

Il devient donc impossible de savoir si 12 correspond à R1 ou R2.

Il est possible de synchroniser ces processus **en utilisant des sémaphores** pour garantir que les calculs et l'affichage soient systématiquement corrects.

Question 12 (2 points) : Proposez des modifications du code des trois processus permettant d'assurer leur synchronisation, **en utilisant des sémaphores**. Cette synchronisation doit avoir pour objectif d'autoriser le maximum de parallélisme entre ces processus tant que cela ne remet pas en cause la **correction des calculs et de l'affichage**. Pour répondre à cette question, vous devez indiquer, hors du code des processus, le nom des sémaphores que vous utilisez, ainsi que la valeur initiale de leurs compteurs. Par exemple, *Init(s, 54)* permet de déclarer un sémaphore s et d'initialiser sont compteur à 54. Vous pouvez ensuite utiliser, pour compléter le code ci-dessus, les fonctions *P(s)* et *V(s)* qui réalisent les opérations de prise et restitution de « jeton » d'un sémaphore s.

Réponse :

On utilise 3 sémaphores :

- Ex permettant de réaliser une exclusion mutuelle,
- C1 pour assurer que Aux1 est bien mise à jour lors de la lecture,
- C2 qui joue le même rôle pour Aux2.

Ex est initialisé à 1, C1 et C2 à 0

Init (Ex,1) ; Init (C2,0) ; Init (C1,0) ;

Le code mis à jour est donné ci-dessous :

Processus 1	Processus 2	Processus 3
1. void main(){ 2. RESLOC=f1(X) ; *3. P(C1) ;* 4. R1=f4(RESLOC,Aux1) ; *5. P(Ex) ;* 6. printf("R1 vaut :"); 7. printf("%d\n",R1); *8. V(Ex) ;* 9. }	10. void main(){ 11. Aux1= f2(Y) ; *12. V(C1) ;P(C2) ;* 13. R2=f5(Aux1,Aux2) ; *14. P(Ex) ;* 15. printf("R2 vaut :") ; 16. printf("%d\n",R2) ; *17. V(Ex) ;* 18. }	19. void main(){ 20. Aux2=f3(Z) ; *21. V(C2) ;* 22. }

Annexe à rendre
Nom, Prénom :

Extrait de code pour répondre à la question 5

```
// the alarm clock starts ringing when this function is called void
start_ringing();
// the alarm clock stops ringing when this function is called
void stop_ringing();
// returns the amount of time to reach the next alarm time
int get_next_alarm_time();

// function to execute when an alarm time is reached
void on_alarm(int num_sig);
// function to execute when a user temporarily stops the alarm
void on_snooze(int num_sig);
// function to execute when a user stops the alarm until the next alarm time
void on_stop(int num_sig);

// Boolean values to represent the state of the alarm clock (snoozing, stopped,
ringing)
char snooze_pushed=0;
char stop_pushed=0;
char alarm_ringing=0;

int main()
{
  printf("PID: %d\n", getpid());
  // To be completed
  // 1 - Ignore the signal snoozing the alarm, should not trigger
  // anything when the alarm is not ringing

  signal(SIGUSR1, SIG_IGN);


  // 2 - Ignore the signal for stopping the alarm,
  // should not trigger anything when the alarm is not ringing

  signal(SIGUSR2, SIG_IGN);


  // 3 - When alarm is triggered, execute the on_alarm function

  signal(SIGALRM, on_alarm);


  int duration = get_next_alarm_time();
  alarm(duration);
  while(1); // infinite loop: the alarm always work to be on time at school!
}

void on_snooze(int num_sig)
{
  snooze_pushed = 1;
}
```

Annexe à rendre
Nom, Prénom :

```c
void on_stop(int num_sig)
{
  alarm_ringing = 0;
  // To be completed: make sure the alarm will ring for the next alarm time!

  int next_alarm_time = get_next_alarm_time();
  alarm(next_alarm_time);

}

void on_alarm(int num_sig)
{
  start_ringing();
  alarm_ringing = 1;
  // To be completed: now that the alarm is ringing, stop and snooze
  // functions can be triggered

  signal(SIGUSR1, on_snooze);
  signal(SIGUSR2, on_stop);


  while(alarm_ringing)
  {
    if(snooze_pushed)
    {
      // To be completed: ignore signals for snooze and stop while snooze
      // is being processed

      signal(SIGUSR1, SIG_IGN);
      signal(SIGUSR2, SIG_IGN);

      snooze_pushed=0;
      stop_ringing();
      sleep(2); // snooze time is predefined to 2 (seconds) for simulation;
      // sleep(N) blocks the process during N seconds.
      start_ringing();
      // To be completed: alarm is ringing again: snooze and stop function can be
      // reactivated

      signal(SIGUSR1, on_snooze);
      signal(SIGUSR2, on_stop);

    }
  }
  stop_ringing();
}

int get_next_alarm_time()
{
  return 4; // next alarm time is predefined to 4 (seconds) for simulation;
}

void start_ringing()
{
  printf("RINGING !!!!!!\n");
}
```

Annexe à rendre
Nom, Prénom :

Tableau fournit pour répondre à la question 4: pour rappel, P1, P2, et P3 sont ici des processus.

P1		X		X														
P2			X		X													
P3			X	X	X	X	X	X										

Tableau fournit pour répondre à la question 9: pour rappel, P1, P2, P3 et P4 sont ici des pages de la mémoire physique.

Page logique référencée → 0 1 3 4 3 1 5 3 1 6 0 6 4 -- Entrée dans la table des pages du processus ↓													
0	P1	P1	P1	P1	P1	P1	X				P1	P1	P1
1		P2	P2	P2	P2	P2	P2	P2	P2	P2	P2	P2	P2
2													
3			P3	P3	P3	P3	P3	P3	P3	P3	P3	P3	X
4				P4	P4	P4	P4	P4	P4	X			P3
5							P1	P1	P1	P1	X		
6										P4	P4	P4	P4