Question 1 : Généralités (9 pts)

Répondez aux questions suivantes. Pour les questions à choix multiples, répondez en sélectionnant une ou plusieurs réponses. Pour les autres, justifiez vos réponses.
Les questions font référence aux systèmes d'exploitation de la famille UNIX et considèrent les options par défaut des appels système POSIX.

1.1 [1 pt] Dans un système monoprocesseur de type traitement par lots, quel serait l'impact d'une attente active sur l'exécution des processus (en attente d'exécution) ?

Dans un système de type traitement par lots, un processus en cours d'exécution libère le processeur lorsqu'il se termine ou se bloque (par exemple, en attente d'une fin d'E/S). Si le processus en cours rentre dans une attente active infinie, il ne libèrera pas le processeur. Les autres processus en attente d'exécution vont donc attendre indéfiniment.

1.2 [1 pt] Dans un système monoprocesseur de type temps partagé, quel serait l'impact d'une attente active sur l'exécution des processus (en attente d'exécution) ?

Dans ce cas, le temps d'allocation du processeur à chaque processus est limité. Le système va suspendre l'exécution du processus lorsqu'il aura consommé ce temps limite (appelé quantum), au profit d'un autre processus en attente. Les processus en attente d'exécution vont tous pouvoir avoir à tour de rôle du temps CPU, même si un processus rentre dans une attente active infinie.

1.3 [1 pt] Un processus *P* ouvre un fichier « *int fd = open("data", O_RDWR);*» puis crée un fils *F* en appelant « *fork() »*. Dans ce cas,

a) *P* et *F* peuvent accéder au fichier data, via *fd*.
b) seul *P* peut accéder au fichier data, via *fd*.
c) *P* et *F* partagent le pointeur de fichier de *fd*.
d) la fermeture du fichier (*close(fd)*) par *P* supprime à *F* l'accès, via *fd*, au fichier.
e) aucune de ces réponses.

[La fonction fork va, entre autres, créer une table de descripteurs de fichiers (TDF) pour F identique à celle de son père P. Ce qui signifie que l'entrée fd dans chacune des deux TDFs (celle du père et celle du fils) pointe vers la même entrée dans la table des fichiers ouverts (TFO). Le pointeur de fichier de fd figure dans cette entrée de TFO. Si P ferme son descripteur fd, F aura toujours le sien et pourra donc continuer à accéder au fichier data via son descripteur fd.]

1.4 [1 pt] Un thread d'un processus ouvre un fichier « *fd = open("data", O_RDWR);*» où *fd* est une variable globale du processus. Dans ce cas,

a) tous les threads du processus peuvent accéder au fichier *data,* via *fd*.
b) seul le thread créateur du fichier peut accéder au fichier *data,* via *fd*.
c) tous les threads partagent le pointeur de fichier de *fd*.
d) la fermeture du fichier (*close(fd)*) par un thread du processus ne ferme pas l'accès au fichier, via *fd*, à tous les autres threads du processus.
e) aucune de ces réponses.

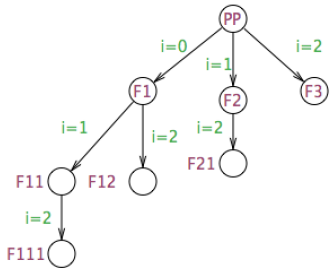
[Tous les threads d'un processus partagent la table de descripteurs de fichiers (TDF) du processus. Ils pourront donc accéder au fichier data via fd et partagent aussi le même pointeur de fichier. Par contre, la fermeture de fd par un thread du processus va supprimer l'accès au fichier à tous les threads du processus.]

1.5 [1 pt] Un processus réalise dans l'ordre ce qui suit :
« *int fd[2]; pipe(fd);*
if(fork()) write(fd[1],"X",1); else {char c[2]; while(read(fd[0],&c,2)>0);} exit(0); ».
Le processus fils :
a) va lire le caractère *X* du pipe puis se terminer.
b) va bloquer pour toujours.
c) pourrait être adopté par le processus *init*.
d) va bloquer*, jusqu'à ce que le père se termine.*
e) aucune de ces réponses.

[Le processus fils va bloquer en attente de fin de fichier qui n'arrivera jamais. Lorsque son père se termine, il sera adopté par le processus init.]

1.6 [1 pt] Peut-on utiliser les fonctions *sem_wait* et *sem_post* de la librairie <*semaphore.h*>, pour synchroniser 4 threads utilisateur d'un même processus ?
Non, car sem_wait est un appel système bloquant par défaut. Si un thread utilisateur est bloqué suite à un appel à sem_wait, les autres threads utilisateur ne pourront plus s'exécuter, car le système va basculer l'état du processus à l'état bloqué (en attente du sémaphore). Les threads détenteurs du sémaphore ne pourront pas s'exécuter et libérer le sémaphore. Le processus va donc rester bloqué indéfiniment. Pour synchroniser des threads utilisateur créés en utilisant une librairie, il est recommander d'utiliser les fonctions de synchronisation de cette même librairie.

1.7 [3 pts] Donnez l'arborescence des processus créés par le bout de code suivant : « *int i; for (i=0; i<3; i++) fork(); printf(" i=%d\n",i);* ». Quelle est la valeur de *i* affichée par chaque processus (y compris le processus principal) ?
PP crée 3 processus F1, F2 et F3. F1 crée 2 processus F11 et F12. F2 crée 1 processus F21. F11 crée un processus F111. La valeur de i est 3.



Question 2 (4 pts) : Processus & signaux

Considérez le programme suivant :

```
#include  "includes.h"
#define MaxJoueurs 4
void jouer(int i)
{
  sleep(3);  // simule une partie de jeu
```

```
   exit(i);
}
/*0*/
int main( )
{       // créer MaxJoueurs processus fils du processus principal
        // chaque joueur i, pour i=0 à 3, exécute la fonction jouer(i).
        /*1*/
        while(1)
        {   // attendre la fin d'un joueur
            // lorsqu'un joueur i, pour i=0 à 3, se termine, un autre joueur i est créé.
            /*2*/
        }
   return 0;
}
```

2.1 [2 pts] Complétez le programme ci-dessus en y ajoutant le code qui réalise exactement les traitements spécifiés sous forme de commentaires.

```
#include  "includes.h"
#define MaxJoueurs 4
void jouer( int i)
{  sleep(3);  // simule une partie de jeu
   exit(i);
}
/*0*/
int main( )
{
        /*1*/ int i;
       for(i=0; i< MaxJoueurs; i++)
                if (fork()==0)  jouer(i);
        while(1)
        {
          /*2*/  if (wait(&i) > 0)  { if (fork()==0) jouer(WEXITSTATUS(i)); }
        }
        return 0;
}
```

2.2 [2 pts] On veut maintenant utiliser la fonction *alarm* pour limiter la durée totale du jeu à *60 secondes* (*alarm(60);*). Cette fonction est appelée juste avant l'instruction *while(1)*. Au bout de ces *60* secondes, le processus principal doit stopper le remplacement de processus, forcer la terminaison des joueurs puis se terminer. La terminaison de chaque joueur est forcée par l'envoi d'un signal *SIGKILL* au joueur.

Modifiez/complétez le code pour y inclure cette nouvelle directive.

La fonction *int alarm(int nb_sec)* programme une temporisation pour qu'elle envoie un signal *SIGALRM* au processus appelant dans *nb_sec* secondes. La valeur de retour de la fonction *alarm* ne nous intéresse pas dans le cadre de cette question. Le traitement par défaut du signal *SIGLARM* est : terminer le processus destinataire du signal.

```
#include  "includes.h"
#define MaxJoueurs 4
void jouer( int i)
{   sleep(3);  // simule une partie de jeu
  exit(i);
}
/*0*/  int pid[MaxJoueurs]; // pour sauvegarder les pids des joueurs
void action (int sig)   // gestionnaire à associer au signal SIGALRM
{ int i; for(i=0; i<MaxJoueurs; i++)
        kill (pid[i], SIGKILL);  // forcer la terminaison des joueurs
   exit(0);
}
int main( )
{       signal(SIGALRM,action); // associer le traitement action au signal SIGALRM

        int i; for(i=0; i< MaxJoueurs; i++)
             if ((pid[i]=fork())==0) {
                   jouer(i);
              }
        alarm(60); // armer un temporisateur (une alarme) pour qu'il envoie un signal SIGALRM au processus
        while(1)
        {       /*2*/ if (wait(&i) > 0)
                      {  if ((pid[WEXITSTATUS(i)]=fork())==0) { jouer(WEXITSTATUS(i));}
                      }
      }
        return 0;
}
```

Question 3 (3 pts) : Processus & redirection des E/S standards

On veut implémenter la fonction *Popen* définie comme suit :
int Popen(char * com[ ], char sens) ;
où :

- ***com*** est une commande composée d'un nom de fichier exécutable (supposé dans le répertoire courant et le *PATH*), suivi éventuellement de paramètres, et
- ***sens*** est soit égal à *'r'* ou *'w'*.

Popen crée un tube anonyme et un processus fils. Le tube créé devient l'entrée standard du processus fils, si ***sens*** est *'r'*. Il devient la sortie standard du processus fils, dans le cas où **sens** est *'w'*. Le processus fils créé se charge d'exécuter la commande ***com*** en appelant la fonction ***execvp***.
*Pope*n retourne le descripteur du tube non utilisé par le fils, au processus appelant.
Ne traitez pas les cas d'erreur.

Exemple d'utilisation de la fonction *Popen* :

```
#include  "includes.h"
#define Taille 1024
int Popen(char * com[ ], char sens) ;
int main (int argc, char* argv[])
```

```
{
        char Data;
        int fd = Popen (&argv[1], 'w');
// Dans ce cas, Popen crée un tube anonyme et un processus fils.
// Le processus fils associe sa sortie standard au tube anonyme puis se transforme
// pour exécuter la commande passée en paramètre au programme.
// Popen retourne le descripteur de lecture du pipe.

        while (read(fd, Data, 1)>0  )
                write(1, Data, 1);
        close (fd);
        wait(NULL);
        return 0;
}
int Popen(char * com[ ], char sens)
{  /*0*/ }
```

3.1 [2 pts] Donnez le code de la fonction ***Popen***.

```
int Popen(char * com[ ], char sens)
{   /*0*/
     int fd[2] ;
     pipe(fd) ;
     if (fork() ==0)
     {     // il s'agit du fils
          if (sens =='w')   dup2(fd[1],1) ;  // STDOUT du fils devient le pipe
           else  dup2(fd[0],0) ; // STDIN du fils devient le pipe
           close(fd[0]) ;
           close(fd[1]) ;
           execvp(com[0], com) ;
      } else // il s'agit du père
          if (sens =='w')  {  close(fd[1]) ; return fd[0]; }
        else { close(fd[0]) ; return fd[1] ;}
}
```

3.2 [1 pt] Pour l'exemple précédent, indiquez l'entrée standard et les sorties (standard et erreur) du processus fils juste avant d'entamer l'exécution de la commande ***com***.

STDIN = clavier, STDOUT= pipe en écriture, STDERR = écran

Question 4 (4 pts) : Synchronisation

Considérez les 3 processus A, B et C concurrents suivants :

A	B	C
a1; a2;	b1; b2;	c1; c2;

Où ai, bi et ci, pour i=1,2, sont des actions atomiques.

4.1 [1 pt] Synchronisez, en utilisant les sémaphores, les processus A, B et C de manière à empêcher l'entrelacement des actions de A, B et C.

Semaphore S=1;		
A	**B**	**C**
P(S); **a1; a2;** **V(S);**	**P(S);** **b1; b2;** **V(S);**	**P(S);** **c1; c2;** **V(S);**

4.2 [1 pt] Utilisez les sémaphores pour que au plus 2 processus sur 3 s'exécutent en concurrence.

Semaphore S=2;		
A	**B**	**C**
P(S); a1; a2; **V(S);**	**P(S);** b1; b2; **V(S);**	**P(S);** c1; c2; **V(S);**

4.3 [1 pt] Utilisez les sémaphores pour forcer l'exécution des actions du processus A avant celles de B et C. Les actions de B et C s'exécutent en concurrence.

Semaphore S=0;		
A	**B**	**C**
a1; a2**;** **V(S);** **V(S);**	**P(S);** b1; b2;	**P(S);** c1; c2**;**

4.4 [1 pt] Pour empêcher l'entrelacement des actions de A, B et C, un collègue vous propose d'utiliser l'instruction atomique TSL comme suit :

int v=0;		
A	B	C

while(v==0 && TSL(&v)!=0);
a1; a2;
v=0;
	while(v==0 && TSL(&v)!=0);
	b1; b2;
	v=0;
		while(v==0 && TSL(&v)!=0);
		c1; c2;
		v=0;

Est-ce que cette solution présente un intérêt ou un problème par rapport à la solution qui se base sur une boucle sur TSL (while(TSL(&v)!=0);)?

Relativement à « TSL(&v)!=0 », la condition « v==0 && TSL(&v) » a pour intérêt de réduire le nombre d'appels à TSL(&v) (qui est plus couteuse qu'une simple lecture de la mémoire). Cette fonction est appelée uniquement si la valeur de v est égale à 0 (le verrou est libre). Si la valeur de v est 1 (le verrou est déjà pris), la fonction TSL n'est pas appelée. Par contre, la condition du while va être évaluée à faux pour tout autre processus qui tente d'entrer en section critique. Plusieurs processus peuvent ainsi se retrouver en même temps en section critique. Une solution correcte qui réduirait le nombre d'appels à TSL tout en assurant l'exclusion mutuelle pourrait être : while (v!=0 || TSL(&v)!=0);.