Question 1 (8 pts) : Processus & signaux

Considérez le programme suivant :

```
const int n=2;
int p=0;
int main()
{     int i;
      for(i=0; i<n; i++) {
            if(fork()>0)
            {    if(fork()>0) break; }
             p=p+1;
      }
      printf(" p=%d\n",p);
      return 0;
}
```

Supposez que tous les appels système ne retournent pas d'erreur.

1.1. [3 pts] Donnez l'arborescence des processus créés par ce programme. Indiquez la séquence d'instructions exécutées par chacun des processus, y compris le processus principal.

1.2. [1 pt] Donnez la valeur de *p* affichée par chacun des processus, y compris le processus principal. À quoi correspond-elle ?

1.3. [1 pt] Complétez le code précédent pour que les processus sans enfants se transforment pour exécuter le code « *foo.exe* », juste avant l'instruction « *return 0;* ». Cette transformation est réalisée par la fonction « *execlp ». Le fichier exécutable* « *foo.exe* » n'a aucun paramètre. Il est supposé existant dans le répertoire courant et le *PATH*.

1.4. [1 pt] Complétez le code (obtenu en 1.3) pour que les processus parents attendent la fin de leurs fils avant de se terminer.

1.5. [2 pts] Complétez le code (obtenu en 1.4) pour que :

- **a.** tous les processus sans enfants ignorent (action *SIG_IGN*) le signal *SIGINT* avant de commencer « *foo.exe* », et
- **b.** tous les autres processus (y compris le processus principal) captent le signal *SIGINT*. Le traitement consiste à afficher le message « signal *SIGINT* reçu » puis se terminer en appelant « *exit(0);* ».

Attention : Donnez un seul code, pour répondre aux questions 1.3, 1.4 et 1.5.

Question 2 (5+1 pts) : Processus & tubes de communication

Considérez la fonction ***getResult*** qui permet de récupérer, via un tube de communication nommé, les résultats d'un exécutable. Son prototype est :

int getResult(char * prog, char* infile, char* npipe) ;

Cette fonction a 3 paramètres listés dans l'ordre :

- un code exécutable qui n'a aucun paramètre,
- un nom de fichier de données supposé existant dans le répertoire courant *et*
- un nom de tube nommé supposé déjà créé par la commande *mkfifo dans le répertoire courant*.

La fonction ***getResult*** commence par créer un processus fils.

Pour le processus fils, la fonction ***getResult*** ouvre le tube ***npipe*** *en écriture et associe son descripteur à la* sortie standard. Elle ouvre ensuite le fichier ***infile*** et associe son descripteur à l'entrée standard. Enfin, elle appelle la fonction ***execlp*** pour exécuter ***prog***. Ce code est supposé existant dans le répertoire courant et dans le *PATH*.

Pour le processus appelant, *après la création du fils, la fonction* ***getResult*** ouvre le tube ***npipe*** en *lecture puis* retourne son descripteur au processus appelant.

2.1. [3 pts] Donnez le code de la fonction ***getResult***. Ne traitez pas les cas d'erreur. Par contre, il est important de fermer les descripteurs de fichiers non utilisés.

2.2. [2 pts] Dans la question 2.1, pour la création du processus fils, est-il plus approprié de remplacer la fonction ***fork*** par **v*fork*** ? Justifiez votre réponse.

2.3. [1 pt bonus] Dans la question 2.1, pour la création du processus fils, est-il plus approprié de remplacer la fonction ***fork*** par **pthread_create** (c-à-d créer un thread au lieu d'un processus et transférer le traitement réalisé par le processus fils dans la fonction exécutée par le thread créé) ? Justifiez votre réponse.

Question 3 (7 pts) : Synchronisation

3.1. [2 pts] Considérez les processus *P1, P2, P3* et *P4*, et les sémaphores *S1* et *S2*. Les processus sont lancés en concurrence. Donnez tous les ordres d'exécution des opérations atomiques *a, b, c, d, e, f, g* et *h*.

Semaphore S1=1, S2=0 ;			
P1	*P2*	*P3*	*P4*
P(S1) ; *a;* *b;* *V(S2) ;*	*P(S1) ;* *c;* *d;* *V(S2) ;*	*P(S2) ;* *e;* *f;* *V(S1) ;*	*P(S2) ;* *g ;* *h;* *V(S1) ;*

3.2 [2.5 pts] Pour assurer l'exclusion mutuelle entre deux processus *P0* et *P1*, je vous propose la solution suivante :

bool flag[2] = {false, false};	
P0	*P1*
while(1) { *flag[0]=true;* *while(flag[1]==true);* *section_critique(0);* *flag[0]=false;* *}*	*while(1) {* *flag[1]=true;* *while(flag[0]==true);* *section_critique(1);* *flag[1]=false;* *}*

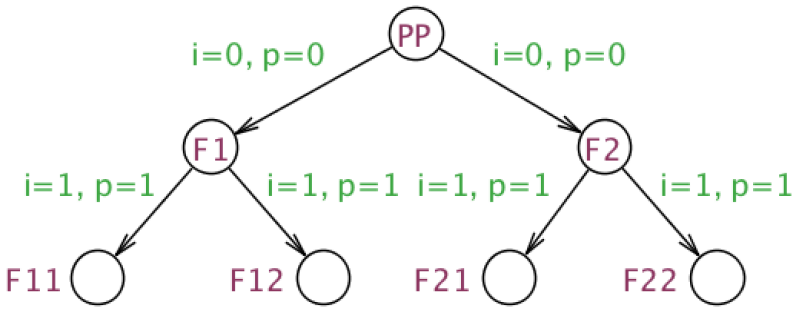
Cette solution est-elle correcte ? Est-ce que les deux processus peuvent se retrouver tous les deux en même temps dans leurs sections critiques ? Justifiez vos réponses.

3.3 [2.5 pts] Supposez qu'une librairie de threads utilisateur dispose des fonctions ***void uceder()*** et ***int TSL(int&lock)***. La fonction ***uceder()*** suspend l'exécution du thread appelant. Le thread suspendu est inséré à la fin de la file d'attente de l'ordonnanceur des threads utilisateur du processus.
On veut implémenter, en utilisant les fonctions ***TSL(int&lock)*** et ***uceder()***, les *mutex* au niveau utilisateur tout en évitant les attentes actives.

Donnez les codes des fonctions ***void uP(m_t&m)*** et ***void uV(m_t&m)*** qui permettent respectivement de verrouiller et déverrouiller un *mutex* ***m*** au niveau utilisateur. Définissez clairement la structure ***m_t*** utilisée.

Le corrigé
Question 1

1.1. et 1.2.



PP

i=0						
fork()				F1		
fork()	F2			p=p+1		
printf(…)	p=p+1			i=1		
return 0	i=1			fork()		F11
	fork()		F21	fork()	F12	p=p+1
	fork()	F22	p=p+1	printf(…)	p=p+1	i=2
	printf(…)	p=p+1	i=2	return 0;	i=2	printf(…)
	return 0	i=2	printf(…)		printf(…)	return 0
		printf(…)	return 0		return 0	
		return 0				
p=0	p=1	p=2	p=2	p=1	p=2	p=2

p indique le niveau du processus dans l'arbre binaire créé par ce programme.

1.3., 1.4. et 1.5.

```
const int n=2;
int p=0;
// 1.5
void gestionnaire(int sig) { printf("signal SIGINT reçu \n"); exit(0);}
int main()
{     int i;
      signal(SIGINT,gestionnaire); // 1.5
      for(i=0;i<n;i++) {
            if(fork()>0)
            {    if(fork()>0) break; }
             p=p+1;
      }
    printf(" p=%d\n",p);
   // 1.3 et 1.5
  if (i==n)  { signal(SIGINT, SIG_IGN); execlp("foo.exe","foo.exe",NULL); }
// 1.4
 else while(wait(NULL)>0);
      return 0;
}
```

Question 2

2.1.

```
int getResult (char * prog, char* infile, char*npipe)
{  /*0*/
     int fd;

     if (fork() ==0)
     {    // il s'agit du fils
          fd=open(npipe,O_WRONLY);
          dup2(fd,1); // STDOUT ←→ pipe en écriture
          close(fd) ;
         fd=open(infile, O_RDONLY);
         dup2(fd,0); // STDTIN ←→ infile en lecture
         close(fd);
         execlp(prog, prog, NULL) ;
     } else { // il s'agit du père c-à-d le processus appelant
                fd=open(npipe,O_RDONLY); return fd; }
}
```

2.2.

Non, car après vfork,

- le processus père bloque jusqu'à ce que le processus se termine ou se transforme, et
- le processus fils va bloquer dans la fonction open *(open(npipe,O_WRONLY)*) en attente qu'un autre processus ouvre le pipe en lecture.

Les processus père et fils vont s'interbloquer pour toujours.

2.3. Non, car lorsque le thread va appeler une fonction exec, l'espace d'adressage du processus va être remplacé et par conséquent le thread appelant la fonction getResult n'existera plus (ainsi que les autres threads du processus).

Question 3

3.1.
a ; b ; e ; f; c ; d ; g ; h;
c ; d ; e ; f; a ; b ; g ; h;
a ; b ; g ; h ; c ; d ; e; f;
c ; d; g ; h; a ; b ; e ; f ;

3.2.
Non car les processus peuvent se retrouver tous les deux dans des attentes actives infinies :
P0 flag[0]=true; P1 flag[1] =true; P0 et P1 rentrent dans des boucles infinies.
Par contre, les deux processus ne peuvent pas se retrouver en même temps en section critique car si Pi est en section critique alors flag[i] = true est exécutée avant flag[1-i] = true. Quand P(1-i) atteindra la boucle vide (while(flag[i]==true);), il bouclera tant que Pi est en section critique.

3.3.
struct m_t { int lock=0; }
void uP(m_t&m) { while (TSL(m.lock)!=0) uceder(); }
void uV(m_t&m) {m.lock=0; }