Question 1 (4 pts) : Généralités

Répondez aux questions suivantes (les réponses doivent être justifiées, concises et claires) :

1.1) [1.5 pt] Donnez le nombre de processus créés par le bout de code suivant (supposez que les appels à « *fork et execlp* » ne retournent pas d'erreur) :

while (fork() !=0) { if(fork()) execlp("/bin/ps", "ps", NULL); break ;} printf("Fin") ;

(ne pas compter le processus créé par le shell).

Parmi les processus créés, indiquez ceux qui vont pouvoir atteindre l'instruction « printf ».

Le nombre de processus créés est 2. Le processus principal PP crée un fils F1. F1 ne rentre dans la boucle while. PP rentre dans la boucle, crée un autre fils F2 puis se transforme en ps. F2 exécute l'instruction break qui le fait sortir de la boucle. Seuls les processus F1 et F2 vont atteindre l'instruction printf.

1.2) [1.5 pt] L'espace d'adressage virtuel d'un processus Linux est composé de plusieurs régions. Indiquez pour chacune des variables suivantes dans quelle région elle sera présentée :

a) Une variable globale de type « *int* » initialisée à *10*,
b) Une variable globale non initialisée,
c) Une variable locale à la fonction « *main* » de type « *int* » initialisée à *10*.
d) Un pointeur global initialisé à l'adresse d'une zone allouée dynamiquement en utilisant la fonction « *malloc* ».
e) La zone allouée dynamiquement en d).

a) Dans la 1ère partie de la région de données (celle qui suit la région de code).
b) et d) Dans la région BSS (la 2ième partie de la région des données).
c) Dans le 1er frame (1ère section) de la région de pile (celle de l'appel à la fonction main).
e) Dans le tas.

1.3) [1 pt] Expliquez comment implémenter sous Windows, en utilisant l'API *win32*, le traitement réalisé par la commande Linux « *p.exe > fich* » où *p.exe* est un exécutable et *fich* est un nouveau fichier (qui n'existe pas dans le répertoire courant). Ne donnez pas de code. Indiquez, par contre, toutes les étapes à suivre sous forme de commentaires clairs et précis.

Il suffit de :

1- **Créer/ouvrir le fichier Fich (CreateFile), qui retourne le handle du fichier créé/ouvert.**
2- **Rendre héritable le fichier ouvert (handle du fichier).**
3- **Déclarer deux variables pi de type PROCESS_INFORMATION et si de type STARTUPINFO; Initialiser les attributs hStdInput à STDIN du processus, hStdError à STDERR du processus et hStdOutput au handle du fichier Fich créé.**
4- **Créer un processus fils : CreateProcess("p.exe", //name or path of executable**
NULL, // no command line.
NULL, // Process handle not inheritable.
NULL, // Thread handle not inheritable.
true, // Set handle inheritance to true.
0, // No creation flags.
NULL, // Use parent's environment block.
NULL, // Use parent's starting directory.
&si, // Pointer to STARTUPINFO structure.
&pi).
5- **Fermer le fichier ouvert (handle du fichier).**
6- **Attendre la fin du fils créé.**

Question 2 (4 pts) : Moniteurs et variables de condition

Considérez le moniteur ProducteurConsommateur suivant (cas d'un producteur et d'un consommateur) :

```
Moniteur ProducteurConsommateur
{     const N=100 ;
      int tampon[N], compteur =0, ic=0, ip=0 ;
      boolc nplein, nvide ; // variables de condition
     void produire (int pid, int objet)    // pid numéro du producteur
      {       if (compteur==N) wait(nplein) ;
              tampon[ip] = objet ;
              ip = (ip+1)%N ; compteur++ ;
              if (compteur==1) signal(nvide) ;
      }
      int consommer (int cid )  // cid numéro du consommateur
      {       int objet ;
              if (compteur ==0) wait(nvide) ;
              objet = tampon[ic] ;
              ic = (ic+1)%N ; compteur -- ;
              if(compteur==N-1) signal(nplein) ;
              return objet ;
      }
 }
```

Supposez qu'il y a exactement 3 producteurs (numérotés de 0 à 2) et 3 consommateurs (numérotés de 0 à 2). On veut que les producteurs produisent tour à tour (producteur 0, producteur 1, producteur 2, producteur 0, etc.) et que les consommateurs consomment aussi tour à tour (consommateur 0, consommateur 1, consommateur 2, consommateur 0, etc.).

2.1) [2.5 pts] Modifiez le pseudo-code précédent de manière à satisfaire les requis précédents (productions tour à tour et consommations tour à tour). Pour la synchronisation, vous devez vous limiter à l'utilisation de variables de condition dans le moniteur.

2.2) [1.5 pt] On veut maintenant que chaque producteur produise deux items à chaque tour. Modifiez le pseudo-code obtenu en 1 de manière à satisfaire ce requis supplémentaire. Pour la synchronisation, vous devez vous limiter à l'utilisation de variables de condition dans le moniteur.

Attention : **Donnez un seul code qui inclut toutes les modifications demandées. Vous devez par contre bien indiquer les modifications associées à chaque question.**

```
Moniteur ProducteurConsommateur
{     const N=100 ;
      int tourp =0, tourc=0 ;
      int tampon[N], compteur =0, ic=0, ip=0 ;
      boolc nplein, nvide ; // variables de condition
      boolc wtourp[3], wtourc[3] ;
     void produire (int pid, int objet)
      {       if (tourp !=pid) wait(wtourp[pid]) ;
              if (compteur==N) wait(nplein) ;
              tampon[ip] = objet ;
              ip = (ip+1)%N ; compteur++ ;
              if (compteur==1) signal(nvide) ;
```

```
            // pour 2 - début
              if (compteur==N) wait(nplein) ;
               tampon[ip] = objet ;
               ip = (ip+1)%N ; compteur++ ;
               if (compteur==1) signal(nvide) ;
            // pour 2- fin
               tourp = (tourp+1) % 3 ; signal(wtourp[tourp]) ;
      }
      int consommer (int cid )
      {       int objet ;
              if (tourp !=cid) wait(wtourc[cid]) ;
               if (compteur ==0) wait(nvide) ;
               objet = tampon[ic] ;
               ic = (ic+1)%N ; compteur -- ;
               if(compteur==N-1) signal(nplein) ;
               tourc = (tourc+1) % 3 ; signal(wtourp[tourc]) ;
               return objet ;
      }
  }
```

Question 3 (4 pts) : Interblocage

Considérez un système d'exploitation monoprocesseur doté de 6 types de ressources (R1, R2 ... R6), une seule de chaque type, partagées en exclusion mutuelle par 3 processus P1, P2 et P3. Ces processus exécutent respectivement les codes suivants :

```
P1()
{
    while(1){
        prendre (R4);
        prendre (R5);
        prendre (R3);
    // Utiliser R4, R5, R3
        liberer(R4);
        liberer(R5);
        liberer(R3);
    }
}
```

```
P2()
{
    while(1){
        prendre (R3);
        prendre (R2);
        prendre (R6);
     //Utiliser R3, R2, R6
        liberer(R6);
        liberer(R2);
        liberer(R3);
    }
}
```

```
P3()
{
    while(1){
        prendre (R1);
        prendre (R2);
        prendre (R5);
      // Utiliser R1, R2, R5
        liberer(R5);
        liberer(R2);
        liberer(R1);
    }
}
```

La fonction « *prendre* » permet d'allouer une ressource au processus appelant, si cette dernière est libre. Dans le cas contraire, elle bloque le processus appelant jusqu'à l'obtention de la ressource demandée. Une ressource allouée est libérée par le processus détenteur lorsqu'il invoquera la fonction « *liberer* ».

3.1) [1 pt] Donnez une séquence d'entrelacements des instructions de *P1, P2* et *P3* qui mène vers un interblocage.

P1 : prendre(R4) ; P1 : prendre(R5) ; P2 : prendre(R3) ; P3 : prendre(R1) ; P3 : prendre(R2) ;
P3 : prendre(R5) ; -> va bloquer P3
P2 : prendre(R2) ; -> va bloquer P2
P1 : prendre(R3) ; -> va bloquer P1

3.2) [1 pt] Peut-on prévenir les interblocages en modifiant l'ordre des demandes de ressources ? Justifiez votre réponse.

Oui, il suffit d'associer un ordre aux ressources (par exemple R1 < R2 < …< R6) et si un processus a besoin de deux ressources Ri et Rj avec Ri < Rj, il doit d'abord demander Ri avant Rj. Ce qui donne le code suivant :

```
P1()
{
    while(1){
        prendre (R3);
        prendre (R4);
        prendre (R5);
    // Utiliser R4, R5, R3
        liberer(R4);
        liberer(R5);
        liberer(R3);
    }
}
```

```
P2()
{
    while(1){
        prendre (R2);
        prendre (R3);
        prendre (R6);
     //Utiliser R3, R2, R6
        liberer(R6);
        liberer(R2);
        liberer(R3);
    }
}
```

```
P3()
{
    while(1){
        prendre (R1);
        prendre (R2);
        prendre (R5);
      // Utiliser R1, R2, R5
        liberer(R5);
        liberer(R2);
        liberer(R1);
    }
}
```

3.3) [2 pts] On veut éviter les interblocages en utilisant l'algorithme du banquier. Doit-on modifier la fonction « prendre » ? Si oui, expliquez comment. Supposez que le système est dans l'état où *P1* détient les ressources *R4* et *R5*, et *P3* détient les ressources *R1* et *R2*. Le processus P2 demande la ressource R3. Le système va-t-il satisfaire cette demande ? Justifiez votre réponse.

Oui, car on connaît les besoins de chaque processus. Oui, il faut modifier la fonction prendre pour, par exemple, boucler sur une attente active ou alternée par des pauses bornées, tant que la demande mène vers un état non sûr. Car dans ce cas, toute demande est analysée. La demande est acceptée que si la ressource est libre et maintient le système dans un état sûr. L'état correspondant à (*P1* détient les ressources *R4* et *R5*, et *P3* détient les ressources *R1* et *R2) est :*

Alloc	R1	R2	R3	R4	R5	R6
P1	0	0	0	1	1	0
P2	0	0	0	0	0	0
P3	1	1	0	0	0	0

Req	R1	R2	R3	R4	R5	R6
P1	0	0	1	0	0	0
P2	0	1	1	0	0	1
P3	0	0	0	0	1	0

A = (0 0 1 0 0 1)
Si on accorde R3 à P2, on atteindrait l'état :

Alloc	R1	R2	R3	R4	R5	R6
P1	0	0	0	1	1	0
P2	0	0	1	0	0	0
P3	1	1	0	0	0	0

Req	R1	R2	R3	R4	R5	R6
P1	0	0	1	0	0	0
P2	0	1	0	0	0	1
P3	0	0	0	0	1	0

A = (0 0 0 0 0 1)
Cet état n'est pas sûr car aucune des relations suivantes n'est vraie Req(P1) <= A, Req(P2) <= A et Req(P3) <= A.

La demande sera donc rejetée.

Question 4 (4.5 pts) : Gestion de la mémoire

On considère un système de pagination à *2* niveaux dans lequel les adresses (virtuelles et physiques) sont codées sur *32* bits. La taille maximale de chaque table de pages (peu importe son niveau) est de 64 *KiO*. Chaque entrée d'une table de pages est composée de *64* octets.

4.1) [1 pt] Donnez le format d'une adresse virtuelle. Quelle est la taille d'une page ?
Le nombre d'entrées maximal dans une table de pages est 64 KiO/ 64 octet = 1 Ki. L'adresse virtuelle est donc composé de 3 champs : 10 bits (1er niveau) 10 bits (2iéme niveau) (12 bits pour l'offset). La taille d'une page est 2^{12} octets = 4 KiO.

4.2) [1.5 pt] Donnez l'adresse physique de l'adresse virtuelle *0x 0010 1210*, si la page référencée par cette adresse est chargée dans le cadre mémoire *5* (la numérotation commence à *0*). Donnez le numéro de page référencée par cette adresse virtuelle.
0x 0010 1210 il suffit de remplacer 00101 par 00005 : 0000 5210.
Il s'agit de la page 0x 00101 c-à-d la 2^8 + 1 = 257 ième page.

4.3) [2 pts] Supposez un processus qui référence dans l'ordre les pages *244 257 150 256 257 244 256 120 257*. Le système réserve les *3* premiers cadres mémoire supposés vides au processus et utilise une allocation locale. Donnez le nombre de défauts de pages générés par le processus pour chacun des algorithmes de remplacement de pages suivants :
- LRU et
- Optimal.

LRU	244	257	150	256	257	244	256	120	257
0	244	244	244	256	256	256	256	256	256
1		257	257	257	257	257	257	120	120
2			150	150	150	244	244	244	257

7 défauts de page pour LRU.

Optimal	244	257	150	256	257	244	256	120	257
0	244	244	244	244	244	244	244	120	120
1		257	257	257	257	257	257	257	257
2			150	256	256	256	256	256	256

5 défauts de page

Question 5 (3.5 pts) : Ordonnancement

Considérons un système composé de *3* processus *P1, P2* et *P3.*

Processus	*Date d'arrivée*	*Temps d'exécution (unités de temps CPU)*
P1	*0*	*9*

P2	*1*	*2*
P3	*2*	*7*

5.1) [2 pts] Donnez le diagramme de Gant de l'exécution de ces processus dans le cas d'un ordonnancement circulaire de quantum de *4*. Donnez les temps de séjour et d'attente moyens (TMS et TMA). Les temps de commutation de contexte sont supposés nuls.

0 P1 1 P1 2 P1 3 P1 4 P2 5 P2 6 P3 7 P3 8 P3 9 P3 10 P1 11 P1 12 P1 13 P1 14 P3 15 P3 16 P3 17 P1 18

TMS = (18 + 5 + 15) / 3 = 38/3 = 12,66 TMA = ((0 + 6 +3) + 3 + (4+ 4)) / 3 = 20 /3 = 6,66

5.2) [1.5 pt] Supposez maintenant que les processus sont périodiques et indépendants. Les périodes de *P1, P2* et *P3* sont respectivement *30, 10* et *20*. Donnez le diagramme de Gant de l'exécution des processus des *30* premières unités de temps, dans le cas d'un ordonnancement *RMA*.

P3				**x**	**x**	**x**	**x**	**x**	**x**	**x**													**x**	**x**	**x**	**x**	**x**	**x**	**x**
P2		**x**	**x**									**x**	**x**									**x**	**x**							
P1	**x**										**x**			**x**	**x**	**x**	**x**	**x**	**x**	**x**										
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**				**20**	**21**	**22**							**30**