Question 1 (5 pts + 1 pt bonus) : Généralités

1.1 [2 pts] Donnez l'arbre des processus ainsi que le nombre de processus créés par ce bout de code :

```
fork();
while (fork())  { execlp("ls", "ls", "-l", NULL);}
return 0;
```

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

Le processus principal crée deux fils F1 (1er fork) et F2 (fork de la condition du while) puis se transforme en ls.
F1 crée un fils F11 puis se transforme en ls.
F2 se termine avec return 0;
F11 se termine avec return 0;
Ce code crée donc 3 processus F1, F2 et F11.

1.2 [2 pts] Considérez le problème de synchronisation des *lecteurs et rédacteurs*. Expliquez, au moyen d'un pseudo-code, comment simplifier la solution vue en classe (reproduite ci-dessous), dans le cas d'un seul lecteur et plusieurs rédacteurs.

```
int NbL=0;
Semaphore mutex=1, redact =1;
```

```
Redacteur () {
  while(1) {
     P(redact);
    Ecriture();
    V(redact);
   }
}
```

```
Lecteur () {
  while(1) {
    P(mutex) ;
    if (NbL==0) P(redact);
    NbL++ ;
    V(mutex);
    Lecture();
    P(mutex);
    NbL--;
    if(NbL==0) V(redact);
    V(mutex);
  }
}
```

Dans ce cas, il s'agit d'assurer un accés exclusif au lecteur et à chaque rédacteur. Un seul sémaphore binaire suffit.
Semaphore redact=1;
Lecteur() { P(redact); lire(); V(redact); }
Redacteur() { P(redact); Ecrire(); V(redact); }

1.3 [2 pts] Dans les premiers systèmes d'exploitation *UNIX*, l'espace d'adressage d'un processus est créé et stocké en mémoire de réserve, une partie du disque dur appelée

zone de swap, avant l'exécution du processus. Cet espace d'adressage est chargé au complet en mémoire physique dans une partition avant de commencer l'exécution du processus. L'espace qu'il occupait en mémoire de réserve est alors libéré. La taille de la partition qu'il occupe en mémoire physique est la même que celle qu'il occupe en mémoire de réserve.

Plusieurs processus peuvent être présents en mémoire physique en même temps. Un processus en mémoire physique bloqué en attente d'un événement ou ayant atteint un temps de séjour de *2* secondes peut être copié en mémoire de réserve, s'il y a des processus prêts qui attendent en mémoire de réserve depuis plus de *3* secondes.

Sachant que les mémoires physique et de réserve ont des tailles limitées, **peut-on avoir des interblocages liés à cette gestion d'espaces** ? Si oui, est-il possible de les prévenir ? Justifiez vos réponses.

Oui, si, par exemple, :

- **il n'y a pas assez d'espace libre en mémoire physique pour charger un processus prêt (depuis plus de 3 secondes) de la zone de swap, et**
- **il n'y a pas assez d'espace libre dans la zone de swap pour transférer un processus de la mémoire physique vers la zone de swap.**
- **Tous les processus en mémoire physique sont bloqués et ne peuvent pas s'exécuter.**

Les processus en mémoire sont en attente de transfert vers la zone de swap. Les processus prêts de la zone de swap sont en attente de transfert vers la mémoire physique.

Il suffit de ne pas libérer l'espace alloué dans la zone de swap à un processus lorsqu'il est chargé en mémoire. Ainsi, il sera toujours possible de retirer le processus de la mémoire physique.

Question 2 (5 pts) Moniteurs et variables de condition

Des travaux de rénovations doivent être effectués dans la cour de récréation des garçons d'un internat. Durant ces travaux, les garçons devront utiliser la cour de récréation des filles. On veut installer un contrôleur programmable qui commande l'ouverture et la fermeture des portes d'accès à la cour des filles. Il y a deux portes d'accès : une pour les filles et l'autre pour les garçons. Chaque porte est munie de deux lecteurs de cartes d'internat, une pour entrer et l'autre pour sortir de la cour. Une fille ou un garçon qui désire accéder ou quitter la cour doit présenter, à un des lecteurs d'entrée ou de sortie, sa carte d'internat. Le contrôleur selon le cas exécute l'une des fonctions suivantes : *girl_access_request(), boy_access_request(), girl_exit_request()* et *boy_exit_request()*.

On décide d'implémenter ces fonctions dans un moniteur et d'utiliser des variables de condition pour contrôler les accès à la cour. En plus des fonctions *wait* et *signal* d'une

variable de condition, supposez que vous disposez de la fonction *signal_all* qui permet de débloquer tous les processus en attente d'une variable de condition.

2.1 [4 pts] Complétez le pseudo-code suivant pour que le contrôleur empêche les filles d'accéder à la cour tant qu'il y a au moins un garçon dans la cour et les garçons d'y accéder tant qu'il y a au moins une fille dans la cour.

```
Moniteur  Cour_partagee
{               /* 1*/
                void girl_access_request ( ) { /*2*/ }
                void boy_access_request ( ) { /*3*/ }
                void girl_exit_request ( ) { /*4*/ }
                void boy_exit_request ( ) { /*5*/ }

}
```

2.2 [1 pt] Est-ce que votre solution précédente présente un risque de famine d'accès à la cour ? Justifiez votre réponse.

```
Moniteur  Cour_partagee
{               /* 1*/ int nbg=0, nbb=0 ; boolc gw, bw ;
                void girl_access_request ( ) { /*2*/ while(nbb>0) wait(gw) ; nbg++ ; }
                void boy_access_request ( ) { /*3*/ while(nbg>0) wait(bw) ; nbb++ ;}
                void girl_exit_request ( ) { /*4*/ nbg-- ; if(nbg==0) signal_all(bw) ;}
                void boy_exit_request ( ) { /*5*/ nbb-- ; if(nbb==0) signal_all(gw) ;}

}
```

Oui, les filles peuvent s'accaparer l'utilisation de la cour. Elles peuvent sortir et revenir à leur guise tant qu'il y a au moins une fille dans la cour. Idem pour les garçons.

Question 3 (3 pts) : Ordonnancement classique

Considérez un système monoprocesseur à ordonnancement circulaire avec un quantum de **2** unités de temps. Supposez que *2* lecteurs *L1, L2* et *1* rédacteur *R1* arrivent dans le système et sont insérés dans la file d'attente du processeur dans l'ordre *L1, R1* et *L2*. Les lecteurs et le rédacteur accèdent à une même partie d'une base de données BD mappée en mémoire, respectivement en lecture et en écriture.

Le traitement réalisé par chaque lecteur consiste en une boucle (sans fin) de lectures. Chaque lecture de *L1* dure *2* quanta. Celle de *L2* dure *1* quantum. Les lecteurs accèdent à la base de données, si le rédacteur *R1* n'est pas dans la base de données. Le rédacteur *R1* accède à la base de données, si aucun lecteur n'est dans la base de données. Son traitement est une boucle (sans fin) d'écritures dans la base de données. Chaque accès en écriture de *R1* dure *2* quanta. Initialement, il n'y a aucun lecteur et aucun rédacteur dans la base de données.

Supposez que si un lecteur ou un rédacteur ne peut pas accéder à la base de données, il rentre dans une attente active jusqu'à ce qu'il obtienne l'accès.

Donnez pour chaque processus le temps de séjour (en quanta) dans la base de données à la fin de 13 quanta. Pour répondre à cette question, vous devez donner le diagramme de Gantt.

L1	R1 (aa)	L2	L1	R1	L2 (aa)	L1 (aa)	R1	L2	L1	R1 (aa)	L2	L1
0 2	2 4	4 6	6 8	8 10	10 12	12 14	14 16	16 18	18 20	20 22	22 24	24 26

L1 : (8-0) + (26-18) unités de temps = 8 quanta ;
L2 : (6-4) + (18-16) + (24-22) unités = 3 quanta ;
R1 : (16-8) unités de temps = 4 quanta.

Question 4 (3 pts) : Ordonnancement temps réel

Considérez les *3* tâches périodiques suivantes :

Tâche	*Date d'arrivée*	*Temps d'exécution Ci*	*Échéance Di*	*Période Pi*
T1	*0*	*10 (ERRRRRRREE)*	*30*	*40*
T2	*2*	*5 (EEEEE)*	*20*	*20*

T3	*5*	*8 (ERRRRREE)*	*15*	*20*

Les tâches *T1* et *T3* partagent une ressource R. La tâche *T2* n'utilise pas de ressources partagées.

Donnez le diagramme de Gantt des *35* premières unités de temps, dans le cas où :
- les tâches sont ordonnancées en utilisant *DMA* et
- le protocole *PIP* est utilisé pour traiter les éventuelles inversions de priorités.

Ces tâches sont-elles ordonnançables ? Justifiez votre réponse.

DMA Prio(T3)>Prio(T2)>Prio(T1)
T1 dates d'activation : 0, 40, … Échéances, 0+30, 40+30, ….
T2 dates d'activation : 2, 22, … Échéances, 2+20, 22+20, ….
T3 dates d'activation : 5, 25, … Échéances, 5+15, 25+15, ….

T1	T2	T3	T1	T3	T2	T1	T2	T3	T2
ER	EEE	E	RRRRRR	RRRRREE	EE	E	EEE	ERRRRRREE	EE
0 2	2 5	5 6	6 12	12 19	19 21	21 22	22 25	25 33	33 35

Non, elles ne sont pas ordonnançables car à la 30^{ième} unité de temps, la tâche T1 va atteindre son échéance sans avoir terminé son traitement périodique.

Question 5 (4 pts) : Gestion de la mémoire

L'espace d'adressage d'un processus *P1* est composé de *8* pages dont *4* pages de code, *3* pages de données et *1* page pour la pile des appels. L'état de la table des pages au moment où *P1* crée un processus fils (désigné dans ce suit par *P2*) est :

	Bit de présence	*Cadre*	*Type*
0	*0*	--	*Code*
1	*1*	*Cadre 0*	*Code*
2	*0*	--	*Code*
3	*0*	--	*Code*
4	*0*	--	*Données*
5	*1*	*Cadre 2*	*Données*
6	*0*	--	*Données*
7	*1*	*Cadre 3*	*Pile*

5.1 [1 pt] Donnez l'adresse physique en hexadécimal de l'adresse virtuelle *0x1CEF* dans le cas où la taille d'une page est *1Kio*. Les adresses physiques et virtuelles sont codées sur *16 bits*.

5.2 [3 pts] Supposez ce qui suit :

- Les pages *1, 5* et *7* ont été chargées successivement en mémoire dans les cadres *0, 2* et *3*.
- La partie de la mémoire physique réservée aux processus *P1* et *P2* est composée des *4* premiers cadres dont *3* sont déjà alloués à *P1* et *1* est libre.
- La création du processus fils *P2* est réalisée par duplication selon le principe « copy-on-write ». À la création de *P2*, sa table des pages est une copie de celle de *P1*.
- Après la création du processus *P2*, le processeur (système monoprocesseur) référence, dans l'ordre suivant, les pages des processus *P1* et *P2* :
  (7, P2, E); (1, P2, L); (7, P1, E); (5, P2, E); (6, P1, L); (6, P2, L); (6, P1, L); (7, P1, L).
  Le triplet *(X, Py, mode)* signifie que le processeur référence la page *X* du processus *Py*.
  Le troisième champ indique le mode d'accès en lecture (L) ou en écriture (E) à la page.
- Lorsqu'un défaut de page se produit et qu'un retrait de page est nécessaire, le système choisit une page parmi celles qui sont réservées aux processus *P1* et *P2*. L'algorithme de remplacement des pages utilisé est *FIFO* (*First In First Out*).

Donnez l'évolution de l'état de la mémoire (la partie réservée aux processus *P1* et *P2*), suite aux références précédentes. **Donnez le nombre de défauts de page** provoqués par chacun des processus *P1* et *P2*.

5.1 L'adresse physique de *0x1CEF est 0x0CEF. L'offset est sur 10bits et le numéro de page sur 6 bits. 0001 1100 1110 1111 et la page 000111 (la page 7) est dans le cadre 3*
➔ *0000 1100 1110 1111 ➔ 0x0CEF*

5.2

		7 P2 E	1 P2 L	7 P1 E	5 P2 E	6 P1 L	6 P2 L	6 P1 L	7 P1 L
Cadre 0	1 (0) P1 P2	1 (0) P1 P2	1 (0) P1 P2	1 (0) P1 P2	5 (4) P2	5 (4) P2	5 (4) P2	5 (4) P2	5 (4) P2
Cadre 1		7 (3) P2	7 (3) P2	7 (3) P2	7 (3) P2	7 (3) P2	7 (3) P2	7 (3) P2	7 (3) P2
Cadre 2	5 (1) P1 P2	5 (1) P1 P2	5 (1) P1 P2	5 (1) P1 P2	5 (1) P1	6 (5) P1	6 (5) P1	6 (5) P1	6 (5) P1
Cadre 3	7 (2) P1 P2	7 (2) P1	7 (2) P1	7 (2) P1	7 (2) P1	7 (2) P1	7 (2) P1	7 (2) P1	7 (2) P1

Il y a au total 3 défauts de page dont 2 sont provoqués par P2 et 1 par P1.