Question 1 (3 pts + 2 pts bonus) : Généralités

1.1. [2.5 pts] Donnez l'arborescence des processus créés par le code suivant (supposez que les appels système ne retournent pas d'erreur). **Donnez également** tous les affichages possibles ainsi que la (les) valeur(s) affichée(s) par chaque processus.

```
int v=10;
int main() {
        if (fork() ) {    v=v+1;
                          if (fork() == 0) { printf("%d\n", v); exit(0); }
        } else {  v=v+2;
                  if (fork() == 0) { printf("%d\n", v);}
                  else { v=v+2;  wait(NULL); printf("%d\n", v); }
                  exit(0);
              }
    v=v+2;
    while(wait(NULL)>0);
    printf("%d\n", v);
    return 0;
}
```

1.2. [0.5 pt] Dans la séquence « *v=v+2; while(wait(NULL)>0); printf("v=%d\n", v);* » du code précédent, une attente de la fin des fils sépare la modification de l'affichage de la variable *v*. Cette séparation a-t-elle un impact sur la valeur affichée de *v* ?

1.3. [2 pts bonus] Complétez le code précédent pour que tous les affichages soient redirigés vers un fichier nommé « inf2610.txt » crée par le processus principal.

Question 2 (6 pts) : Moniteurs et interblocage

On décide d'utiliser les moniteurs et les variables de condition pour contrôler les accès à une base de données, selon le modèle de synchronisation « lecteurs-rédacteurs ». On veut donc permettre des accès multiples en lecture tout en assurant un accès exclusif en écriture.

On vous sollicite pour compléter et implémenter le moniteur *AccesBD* ci-dessous. Ce moniteur est composé de quatre fonctions getR, getW, endR, et *endW*. Les fonctions *getR et getW sont* respectivement appelées pour demander des accès en lecture et en écriture à une base de données. Elles bloquent tout appelant tant que celui-ci n'a pas obtenu l'accès demandé. Les appelants bloqués sont mis en attente dans une même file. Les fonctions *endR* et *endW* sont respectivement appelées pour libérer les accès en lecture et en écriture.

```
Moniteur AccesBD ( )
{       …
        void getR()  { … }
        void endR ( ) {  …  }
        void getW()  { … }
        void endW ( ) {  …  }
}
```

2.1 [4 pts] Complétez le moniteur *AccesBD* pour gérer les accès aux bases de données conformément aux directives précédentes.

2.2 [2 pts] Supposez maintenant que deux threads partagent deux bases de données BD1 et BD2. Ces threads réalisent régulièrement des transferts de données d'une base vers une autre. Avant de commencer le transfert, un thread doit d'abord obtenir les accès nécessaires aux bases de données. Il utilise à cet effet la fonction *getT* :

void getT (AccesBD& BD1, AccesBD& BD2) { BD1.getR(); BD2.getW(); }

À la fin du transfert, il libère les accès en appelant la fonction :

void endT(AccesBD& BD1, AccesBD& BD2) { BD1.endR(); BD2.endW(); }

Les deux threads peuvent-ils se retrouver interbloqués dans la fonction getT ? Si oui, peut-on prévenir de tels interblocages ? Justifiez vos réponses.

Question 3 (5 pts) : Gestion de la mémoire

3.1. [2 pts] Complétez la fonction suivante pour qu'elle affiche le numéro de page *nump*, l'offset *off* et l'adresse de la page *adp* de la variable pointée par le paramètre *x*. Le paramètre *psize* est la taille d'une page.

```
void affiche_adr (void* x, int psize)
{ // convertir en un long l'adresse pointée par x
   long adx = (long)x ;
  // calculer l'offset
   long off = …… ;
  // calculer le numéro de page
   long nump = ……;
  // calculer l'adresse de la page
   long adp = ……;
   printf("nump= %ld, off =%ld, adp=0x%lx\n", nump, off, adp);
}
```

3.2. [3 pts] Considérez un système de pagination pure à *3* niveaux dans lequel les adresses virtuelles sont codées sur *32* bits et la taille d'une page est *2KiO*. Toutes les tables de pages, peu importe leurs niveaux, sont de même taille et ont le même nombre d'entrées.

3.2.1 Donnez la taille maximale en nombre de pages de l'espace d'adressage virtuel d'un processus.

3.2.2 Donnez le format d'une adresse virtuelle.

3.2.3 Donnez le numéro de page qui correspond à l'adresse virtuelle *0x00110A10.*

Question 4 (6 pts) : Ordonnancement de processus

4.1. Considérez un système monoprocesseur et les *4* processus suivants :

Processus	*Priorité*	*Date d'arrivée*	*Temps d'exécution*
P1	*3*	*0*	*5*
P2	*1*	*3*	*4*
P3	*4*	*2*	*2*
P4	*2*	*4*	*4*

Supposez que le temps de commutation de contexte est égal à *0 et que* les priorités sont statiques (la priorité la plus basse est *0).*

4.1.1 **[1.5 pt] Donnez** le diagramme de Gantt montrant l'ordre d'exécution des processus, dans le cas d'un ordonnancement non-préemptif. **Calculez le temps d'attente moyen.**

4.1.2 **[1.5 pt] Donnez** le diagramme de Gantt montrant l'ordre d'exécution des processus, dans le cas d'un ordonnancement préemptif. **Calculez le temps d'attente moyen.**

4.1.3 **[1 pt]** Supposez maintenant que P1 et P3 accèdent en lecture et écriture à une donnée partagée. **Indiquez** pour chacun des ordonnancements précédents la nécessité ou non d'utiliser des mécanismes de synchronisation, afin d'assurer un accès exclusif à cette donnée. **Justifiez vos réponses.**

4.2. **[2 pts]** Une application temps réel est composée de *3* tâches périodiques indépendantes : *T1(5,25), T2(4,10) et T3(3,20)*, où pour *i* de *1* à *3, Ti(ci,pi)* signifie que *ci* et *pi* sont respectivement le pire temps d'exécution et la période de la tâche *Ti*. Afin de vérifier la robustesse de l'application, on décide d'ajouter une tâche périodique indépendante *T4(3,p4) qui récupère et analyse les données en sortie de l'application. Le pire temps d'exécution de cette tâche est connu, par contre, on cherche à lui attribuer une période.*

Est-il possible d'ajouter *T4* à l'application temps réel sans compromettre son ordonnançabilté ? **Si oui, donnez** la plus petite valeur de *p4* possible. **Justifiez vos réponses.**

Le corrigé (FINAL INF2610 – Hiver 2018)

Question 1 : **Généralités**

1.1 PP crée deux fils F1 et F2. F1 crée F11.

PP affiche 13 F1 affiche 14 F2 affiche 11 et F11 affiche 12

Les ordres possibles (1 nombre par ligne)

11 12 14 13 12 11 14 13 12 14 11 13

1.2 non car chaque processus à sa propre variable v (elle est dupliquée dès le premier accès en écriture).

1.3

```
int v=10;
int main() {   int fd= open("INF2610.txt", O_WRONLY);
               dup2(fd,1); close(fd);
        if (fork() ) {    v=v+1;
        if (fork() == 0) { printf("%d\n", v); exit(0); }
        } else {  v=v+2;
if (fork() == 0) { printf("%d\n", v);}
               else { v=v+2;  wait(NULL); printf("%d\n", v); }
               exit(0);
    }
   v=v+2;
   while(wait(NULL)>0);
   printf("%d\n", v);
   return 0;
}
```

Question 2 : Moniteurs et interblocage

2.1.

```
Moniteur AccesBD ( )
{       int nbl=0 ;
        bool libre=true ;
        boolc wa ;
        void getR()
       { while (nbl==0 && libre==false) wait(wa) ;
         if(nbl==0)   libre= false ; nbl++ ; }
         void endR ( ) {  nbl--; if(nbl==0) {libre=true; signal(wa); }  }
```

```
        void getW()  { { while (libre==false) wait(wa) ;  libre=false ;  }
        void endW ( ) { libre=true; signal(wa); }
    }
```

2.2.

Oui, par exemple dans le cas où T1 appelle *getT (BD1, BD2)* et T2 appelle *getT (BD2, BD1),* T1 obtient un accès à BD1 et T2 obtient un accès à BD2. T1 et T2 se bloquent ensuite respectivement en attente d'accès à BD2 et BD1. Oui, il suffit d'ordonner les bases de données (ex. BD1 < BD2) et d'imposer un ordre aux demandes d'accès (ex. de la plus petite à la plus grande).

```
void getT (AccesBD& BD1, AccesBD& BD2)  {
if ( ordre(BD1) < ordre(BD2)) {BD1.getR();  BD2.getW();}
else { BD2.getR();  BD1.getW()}  }
```

Question 3 (6 pts) : Gestion de la mémoire

3.1

```
void affiche_adr (void* x, int psize)
{ // convertir en « long » l'adresse pointée par x
   long adx = (long)x ;
  // calculer l'offset
   long off = adx % psize ;
  // calculer le numéro de page
   long nump = adx / psize;
  // calculer l'adresse de la page
   long adp = adx - offset;
   printf("nump= %ld, off =%ld, adp=0x%lx\n", nump, off, adp);
}
```

3.2

- 1 page = 2KiO = 2^{11} octets ➔ Sur les 32 bits, 11 bits sont réservés au déplacement dans la page. Il reste donc 21 bits pour le numéro de page ➔ 2^{21} pages = 2Mi pages.

- Le format d'une adresse virtuelle : 3 champs de 7 bits chacun pour les niveaux de tables de pages et 11 bits pour l'offset.

- Le numéro de page de *0x00110A10 est donnée par les 21 bits de poids le plus fort 0x0221 c-à-d $2^9 + 2^5$ +1= 512 + 32 +1 = 545. La page 545. Les 11 bits restants (0x210) donnent le déplacement dans la page.*

Question 4 : Ordonnancement de processus

P1		P3		P4		P2	
0	5	5	7	7	11	11	15

TMA = (0 + 8 + 3 + 3)/ 4 = 3.5

P1		P3		P1		P4		P2	
0	2	2	4	4	7	7	11	11	15

TMA = (2 + 8 + 0 + 3)/ 4 = 3.25

1.3 Pour les tâches périodiques indépendantes, une condition nécessaire et suffisante d'ordonnançabilité est celle de EDF (moins stricte que celles de RMA et DMA) :

5/25 + 4/10 + 3/20 + 3/p4 <= 1 => (4+8+3)/20 + 3/p4 <=1 => 3/p4 <= 5/20

➔ oui pour p4 >= 12 . La plus petite de p4 est 12.