Question 1 (6 points) : Moniteurs, variables de condition et interblocage

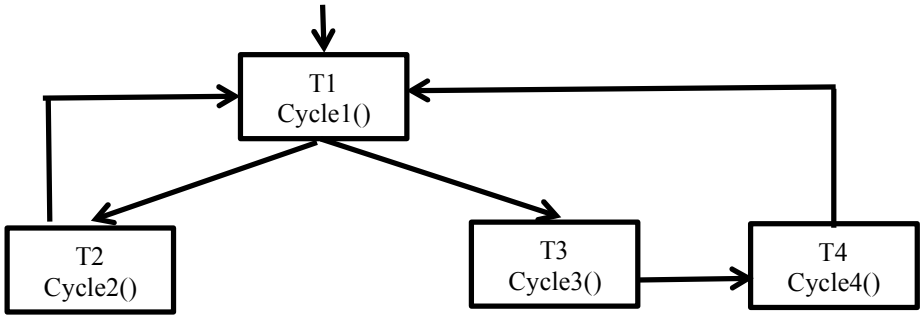

Figure 1 : Graphe de précédence des threads T1, T2, T3 et T4

Un processus crée 4 threads T1, T2, T3 et T4. Chaque thread Ti exécute à répétition la fonction Cyclei(). On veut synchroniser, en utilisant un moniteur et éventuellement des variables de condition, les cycles de ces threads conformément au graphe de précédence de la figure 1. Afin de réaliser cette synchronisation, chaque cycle Cyclei() est encadré par les appels aux fonctions PreCycle(i) et PostCycle(i). Ces fonctions sont implémentées dans le moniteur Synch_t suivant :

```
Moniteur  Synch_t ()
{     ….  // variables et constantes locales
   void PreCycle( int i)  { ….};
   void PostCycle( int i)  { ….};
}

Synch_t   s;
T1 (){
 while (1)
 {
   s.PreCycle(1);
   Cycle1();
  s.PostCycle(1);
 }

}
```

```
T2 (){
 while (1)
 {
   s.PreCycle(2);
    Cycle2();
   s.PostCycle(2);
 }

}
```

```
T3 (){
 while (1)
 {
   s.PreCycle(3);
   Cycle3();
  s.PostCycle(3);
 }

}
```

```
T4 (){
 while (1)
 {
   s.PreCycle(4);
    Cycle4();
   s.PostCycle(4);
 }

}
```

a) **[2 pts]** Complétez le moniteur Synch_t (les déclarations et les fonctions PreCycle et PostCycle) de manière à respecter le graphe de précédence de la figure 1. Expliquez le rôle de chaque variable utilisée.

```
Moniteur  Synch_t ()
{  int nb=0; tour =1;  // variables et constantes locales
   boolc  wT1, wT23, wT4;// pour attendre leurs tours respectifs.
   void PreCycle( int i)  {
         if (i==1 && tour!=1) wait(wT1);
         else       if ((i==2 || i==3) && tour !=2) wait(wT23);
                    else       if  (i==4 && tour !=3) wait(wT4);
   }
   void PostCycle( int i)  {
         if (i==1) { tour=2; signal(wT23); signal(wT23); }
```

```
        else    if (i==2 || i==4) { if (tour==0) {tour=1; signal(wT1);} else tour=0;}
                else    if (i==3) {tour =4;  signal(wT4);}
    }
}
```

b) **[1 pt]** Dans le pseudo-code précédent, les fonctions Cyclei(), i=1 à 4, ne font pas partie du moniteur. Doit-on déplacer ces fonctions dans le moniteur ? Justifiez votre réponse.

Non, car cela va forcer l'exclusion mutuelle des cycles de T2 et T3 (à tout moment, un seul thread est actif dans le moniteur). On perdra ainsi la concurrence de ces deux cycles.

c) **[3 pts]** Supposez maintenant que les threads T1, T2, T3 et T4 partagent 4 ressources de même type R. Chaque thread a besoin de 3 ressources de type R pour accomplir son cycle. Il demande au début de son cycle, une à une, les 3 ressources nécessaires. Il est mis en attente, s'il ne parvient pas à obtenir une ressource (il n'y a plus de ressources R). Il restera en attente jusqu'à l'acquisition d'une ressource R. À la fin de son cycle, il libère toutes les ressources R acquises.

1) **[1.5 pts]** Ces threads peuvent-ils atteindre une situation d'interblocage ? Si, oui, donnez un scénario qui mène vers un interblocage (indiquez les threads interbloqués). Peut-on utiliser l'algorithme du banquier pour éviter les interblocages ? Justifiez votre réponse.
2) **[1.5 pts]** Supposez que les 4 ressources sont de deux types R1 et R2 avec la répartition 3 ressources R1 et 1 ressource R2. Chaque thread a besoin de 2 ressources R1 et 1 ressource R2 pour accomplir son cycle. Ces threads peuvent-ils atteindre une situation d'interblocage ? Si, oui, donnez un scénario qui mène vers un interblocage (indiquez les threads interbloqués). Peut-on les prévenir ? Justifiez votre réponse.

1) Seuls les threads T2 et T3 s'exécutent en concurrence. Oui, une situation d'interblocage est se produit, si T2 et T3 parviennent chacun à acquérir 2 ressources R et se mettent en attente de la 3 ième ressource R manquante.
Oui, on peut les éviter en utilisant l'algorithme du banquier car on connaît les besoins maximaux des threads. Dans ce cas, chaque demande de ressource R est analysée et est refusée, si elle mène vers un état non sûr.
2) Oui, il y a risque d'interblocage, si les ressources sont demandées dans des ordres différents. Exemple de situation d'interblocage : T2 détient 2 R1 et demande R2 ; T3 détient 1 R2 et 1 R1 puis demande R1.
Dans ce cas, T2 ou T3 va parvenir à obtenir les 2 ressources R1 nécessaires. Oui, on peut prévenir les interblocages, si on force les threads à demander les ressources dans le même ordre (par ex. R1 ; R1 ; R2). On éviterait ainsi une attente circulaire.

Question 2 (4 pts) : Gestion de la mémoire

a) **[1 pt]** Un système qui implémente la pagination à la demande dispose de 4 cadres (cases) de mémoire physique qui sont toutes occupées, à un instant donné. La tableau suivant indique, pour chaque case de mémoire physique, la date de chargement de la page qu'elle contient ($t_{chargement}$), la date du dernier accès à cette page ($t_{dernier\text{-}accès}$) et les bits de modification (M) et de présence (P). Les dates sont données en tops d'horloge.

Case	$t_{chargement}$	$t_{dernier\text{-}accès}$	M	P
0	126	270	0	1
1	230	255	0	1
2	110	260	1	1
3	180	275	1	1

Indiquez la page qui sera remplacée en cas de défaut de page, pour chacun des algorithmes de remplacement de pages suivants :

1) LRU
2) FIFO

1) Pour LRU, on retire la page la moins récemment utilisée. Il s'agit donc de choisir une page selon le critère de la colonne $t_{dernier\text{-}accès}$. La page à retirer est celle chargée dans la case 1, qui a été accédée pour la dernière fois à la date 255.
2) Pour FIFO, on retire la page qui a le plus grand temps de séjour. Il s'agit donc de suivre le critère de la colonne $t_{chargement}$. La page à retirer est celle chargée dans la case 2 qui est en mémoire depuis la date 110.

b) [1 pt] Dans le même système, avec pagination à la demande, le temps d'accès à une page non présente en mémoire physique est de 10ms, si le chargement en mémoire est réalisé dans une case libre ou non modifiée depuis le dernier chargement. Il est de 20ms, dans le cas contraire. Supposez que dans 80% des cas de défaut de page, la page à retirer a été modifiée. Calculez le temps moyen de traitement des défauts de page.

t_{moyen} = (0.8*20 + 0.2*10) = 18 ms

c) [2 pts] On considère un système avec une mémoire virtuelle segmentée paginée où la taille d'une page est de 1KiO et une mémoire physique de 64KiO. L'espace d'adressage d'un processus est composé de 3 à 4 segments. La taille maximale de chaque segment est de 16KiO. Les adresses virtuelles et physiques sont codées sur 16 bits. Supposez un processus P composé de 3 segments S0, S1 et S2 de taille, respectivement 16KiO, 8KiO et 4KiO. Les pages 1 et 2 du segment S0, la page 1 du segment S1 et la page 0 du segment S2 sont chargées en mémoire physique, respectivement dans les cadres 2, 0, 9, 12.

1) Donnez les formats (les champs) des adresses virtuelles et physiques.
2) Pour une donnée située dans l'espace d'adressage du processus P à l'adresse décimale 2068, **indiquez** :
   - le segment,
   - le numéro de page dans le segment,
   - le déplacement dans la page,
   - le numéro de case,
   - le déplacement dans la case, et
   - l'adresse physique en hexadécimal (expliquez le calcul de l'adresse physique).

c)
1) On a au plus 4 segments dans l'espace d'adressage virtuel d'un processus et au plus 16 pages par segment (16KiO / 1KiO). L'adresse virtuelle codée sur 16 bits est donc composée de 2 bits pour le numéro segment, 4 bits pour le numéro de pages dans le segment et 10 bits pour le déplacement dans la page (1KiO = 2^{10}). L'adresse physique codée sur 16 bits (64KiO = 2^{16}) est donc composée de 6 bits pour le numéro de case et 10 bits pour le déplacement dans la case.
2) Le code binaire de 2068 est : 0000 1000 0001 0100.
Segment : Les 2 bits de poids fort 00 indiquent qu'il s'agit du segment S0.
Page : Les 4 bits suivants 0010 indiquent qu'il s'agit de la page 2.
Déplacement dans la page : Les derniers 10 bits donnent le déplacement dans la page c-à-d 20.
Case : La page 2 du segment S0 est chargée dans la case 0 de la mémoire physique.
Déplacement dans la case est 20
Adresse physique est obtenue à partir de l'adresse virtuelle en remplaçant les numéros de segment et de page par le numéro de case : 0000 0000 0001 0100 -> 0x00014.

Question 3 (3 points) : Windows

Vous devez implémenter une queue bloquante, de dimension maximale fixe, qui peut accepter des requêtes de plusieurs fils (threads) d'exécution, en utilisant l'API de Windows (par exemple pour les primitives de synchronisation comme les sémaphores et les mutex). Fournissez la déclaration des champs de données de la classe Queue et fournissez une implémentation pour son constructeur, son destructeur et ses méthodes enqueue et dequeue dont la signature est fournie. Pour simplifier le problème, vous n'avez pas besoin de vérifier les valeurs de retour pour les conditions d'erreur.

```
Queue::Queue(int capacity);
Queue::~Queue();
void Queue::enqueue(void *item);
void *Queue::dequeue();
```

Dans la classe queue, les champs de donnée suivants sont requis:

```
void **queue;
HANDLE sem_free;
HANDLE sem_busy;
HANDLE mutex;
int size, ip, ic;
```

Pour le constructeur, il faut initialiser les champs de données, créer les sémaphores et mutex et allouer l'espace.

```
Queue::Queue(int capacity) {
        size = capacity;
        queue = new void*(size);
        ip = ic = 0;
        sem_free = CreateSemaphore(NULL, size, size, NULL);
        sem_busy = CreateSemaphore(NULL, 0, size, NULL);
        mutex = CreateMutex(NULL, FALSE, NULL);
}
```

Pour ajouter un item, il faut attendre qu'il y ait de la place, ajouter l'item sous la protection du mutex, et signaler qu'un item a été ajouté et est disponible pour être retiré de la queue.

```
void Queue::enqueue(void *item) {
        WaitForSingleObject(sem_free, INFINITE);
        WaitForSingleObject(mutex, INFINITE);
        queue[ip] = item;
        ip = (ip + 1) % size;
        ReleaseMutex(mutex);
        ReleaseSemaphore(sem_busy, 1, NULL);
}
```

Pour retirer un item, il faut attendre qu'un item soit disponible, retirer l'item sous la protection du mutex, et signaler qu'un nouvel emplacement libre est disponible.

```
void *Queue::dequeue() {
         void *item;
         WaitForSingleObject(sem_busy, INFINITE);
         WaitForSingleObject(mutex, INFINITE);
        item = queue[ic];
        ic = (ic + 1) % size;
        ReleaseMutex(mutex);
        ReleaseSemaphore(sem_free, 1, NULL);
        return item;
}
```

Question 4 (4 points) : Ordonnancement

a) [**2 pts]** Un système d'exploitation, sur un ordinateur mono-processeur, utilise un ordonnanceur préemptif à files multiples, une par niveau de priorité (1 la moins élevée et 5 la plus élevée). Un ordonnancement circulaire avec quantum de 3 est utilisé entre les fils (threads) de même niveau de priorité; les fils sont toujours insérés en fin de file. Pour les 6 fils (threads) suivants, avec leurs durée, temps d'arrivée et priorité, donnez le diagramme de Gantt de leur ordonnancement et calculez le temps moyen de séjour. Le temps de commutation est considéré comme négligeable.

	T1	T2	T3	T4	T5	T6
durée:	8	6	4	2	4	6
temps d'arrivée:	0	1	2	3	4	5
priorité:	1	2	3	1	2	3

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
T1 T2 T3 T3 T3 T6 T6 T6 T3 T6 T6 T6 T2 T2 T5 T5 T5 T2 T2 T2 T5 T1 T1 T4 T4 T1 T1 T1 T1 T1
Le temps de séjour moyen est de 30 (T1) + 25 - 3 (T4) + 21 - 4 (T5) + 20 - 1 (T2) + 12 - 5 (T6) + 9 - 2 (T3) / 6

= 102 / 6 = 17.

b) **[1 pt]** Un système temps réel mono-processeur préemptif exécute plusieurs tâches périodiques indépendantes. La période et la durée de chacune des 4 tâches sont fournies. Si des priorités fixes de type RMA sont utilisées, que peut-on dire sur la possibilité de réaliser un ordonnancement selon la formule de la condition de Liu et Layland ? Démontrez la possibilité ou non de faire cet ordonnancement.

	T1	T2	T3	T4
durée de travail:	1	2	3	5
période:	4	8	15	17

La priorité de chaque tâche sera fixe et proportionnelle à l'inverse de sa période. On a donc en ordre décroissant T1, T2, T3 et T4. Le critère de Liu et Layland dit que c'est nécessairement ordonnançable si la somme Ci/Pi, (1/4 + 2/8 + 3/15 + 5/17 = 0.9941), est inférieure à 75.7% pour 4 tâches, ce qui n'est pas le cas. Nous ne pouvons donc rien en conclure et il faut essayer de réaliser l'ordonnancement.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
T1 T2 T2 T3 T1 T3 T3 T4 T1 T2 T2 T4 T1 T4 T4 T3 T1

L'ordonnancement échoue, T4 n'est pas terminé après 17 unités, et n'est donc pas possible avec RMA.

c) [**1 pt]** Les mêmes 4 tâches de b) doivent être ordonnancées avec des priorités dynamiques, selon l'échéance la plus proche (EDF). Quel critère peut-on utiliser pour calculer rapidement si l'ordonnancement est réalisable ? Faites l'ordonnancement et donnez le diagramme de Gantt correspondant.

La somme 1/4 + 2/8 + 3/15 + 5/17 = 0.9941 est inférieure à 1 et peut donc être ordonnancée avec EDF. La priorité est toujours donnée à la tâche qui a l'échéance (fin de période) la plus proche. Ceci donne:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
T1 T2 T2 T3 T1 T3 T3 T4 T1 T2 T2 T4 T1 T4 T4 T4 T1

Question 5 (3 points) : Systèmes de fichiers et gestion de périphériques

a) **[2 pts]** Un système de fichiers semblable à FFS de Unix ou Ext2 de Linux utilise des blocs de 16KiO et des numéros de blocs de 32 bits. Chaque i-noeud contient 5 entrées pour des blocs directs, 1 entrée pour un bloc indirect et 1 entrée pour de bloc doublement indirect. Quelle est la taille maximale du système de fichiers ? Quelle est la taille maximale d'un fichier ? Si le système d'exploitation connaît déjà le numéro du i-noeud d'un fichier et veut accéder directement l'octet à la position 128Mi (134217728) de ce fichier, combien de blocs devra-t-il lire avant d'y accéder ? Expliquez pour chaque bloc lu, ce qu'il représente (e.g. bloc indirect de premier niveau) et pourquoi il est requis.

La taille maximale du système de fichiers est de 4Gi x 16KiO = 64TiO. Puisqu'un bloc de 16KiO peut contenir 4Ki entrées de 32 bits (4 octets), pour un fichier, on a 5 blocs directs, 4Ki blocs adressés par le bloc indirect et 4Ki x 4Ki blocs adressés par l'entrée du bloc doublement indirect, pour un total de 5 + 4Ki + 16Mi blocs = 16781317 blocs ou 274945097728 octets (256.06 GiO).

Pour lire l'octet 128Mi, il lui faudra lire le bloc contenant le i-noeud, (pour trouver l'adresse du bloc doublement indirect), lire le bloc doublement indirect, (pour trouver l'adresse du bon bloc à un niveau d'indirection), lire le bloc à un niveau (pour trouver l'adresse du bloc contenant l'octet 128Mi), et lire le bon bloc de donnée. Il faut donc lire 4 blocs.

b) **[1 pt]** Dites à quel niveau (application, librairie système, pilote d'interface, reste du système d'exploitation, ou matériel) s'effectue chacune des 3 tâches suivantes ? Expliquez.

- encryption d'une requête et de la réponse pour l'accès à un site web par https;
- boucle de transfert pour le contenu d'un bloc du disque par DMA;
- choix de la prochaine tâche à exécuter lorsqu'un programme se termine et libère un processeur.

L'encryption d'une requête https se fait dans l'application de furetage Web ou dans une librairie d'encryption (e.g. OpenSSL). Le transfert par DMA est effectué directement par le matériel, qui est un contrôleur DMA soit indépendant soit incorporé au contrôleur de disque. Le choix de la prochaine tâche à exécuter se fait dans l'ordonnanceur du système d'exploitation.