

Examen 3IN010

L3 – Licence d'Informatique -Mai 2021

1h30 - Aucun document autorisé - Barème donné à titre indicatif

1. MEMOIRE (7,5 POINTS)

On dispose d'une machine simplement paginée avec des pages de 1024 octets.

1.1. (0,5 point)

Le bit Référence (R) d'une page est-il mis à 1 par le matériel ou le système d'exploitation ? Justifiez votre réponse.

Mis à 1 par la MMU pour des raisons d'efficacité

1.2. (0,5 point)

A quoi sert la TLB ? A quel moment la TLB est lue et par qui (matériel ou système)

La TLB sert à accélérer la traduction d'adresse en évitant les accès à la RAM

Elle est lue à chaque accès mémoire par la MMU.

1.3. (0,5 point)

- Quelle est la différence entre une mémoire paginée et une mémoire segmentée?
- La taille d'une page est-elle égale à la taille d'une case ?

Paginé : découpage de la mémoire virtuelle en page de taille fixe

Segmenté : découpage de la mémoire virtuelle en segment de taille variable

Taille page = taille case

1.4. (0,5 point)

Pourquoi la taille des pages est toujours une puissance de 2 ?

Cela permet à la MMU de retrouver plus rapidement le numéro de page au sein d'une adresse : la division par une puissance 2 correspond à un simple décalage.

Soit un processus P, dont la table des pages est la suivante :

	case	Présence	LEX	Référence
0		0	101	0
1	100	1	101	1
2		0	101	0
3		0	110	0
4		0	110	0
6	200	1	110	0
7		0	110	0
8	300	1	110	0

(L : lecteur, E : écriture, X : eXécution)

Figure 1

1.5. (1,25 point)

Que se passe-t-il lorsque P fait un accès en lecture à l'adresse 1200 ?

Vous préciserez les actions faites par le matériel. Vous donnerez l'adresse physique correspondant à l'accès.

MMU : accès page <1,176>, droit OK, Présence OK,
adresse physique = $100 \cdot 1024 + 176 = 102\,576$

On considère la stratégie de remplacement de page de type FINUFO vue en TD. Une liste des pages triées selon leur date de chargement est gérée : la page en tête de liste est la plus récemment chargée, celle en fin de liste est la plus anciennement chargée. Lors d'un défaut de page, la page la plus anciennement chargée est examinée. Si son bit R vaut 1, elle est remise en tête de liste et son bit R passe à 0. La page déchargée est donc celle qui est la plus anciennement chargée et dont le bit R vaut 0.

La table des pages du processus P est celle de la Figure 1. Les pages 8, 6 et 1 ont été chargées dans cet ordre. On dispose uniquement des cases 100, 200 et 300. P fait alors la séquence d'accès suivant aux pages.

2 1 7 1 8 1 2 8

1.6. (3 points)

a) Quels sont les défauts de page engendrés par cette séquence ?

Faites un tableau pour indiquer à chaque accès l'état de la liste des pages en mémoire avec le bit R associé.

b) Quel est le nombre total de défauts de pages ?

	Init	2	1	7	1	8	1	2	8
File	1,1	2,1	2,1	7,1	7,1	8,1	1,1	2,1	2,1
	6,0	1,1	1,1	2,1	2,1	7,0	8,1	1,1	1,1
	8,0	6,0	6,0	1,1	1,1	2,0	7,0	8,1	8,1
100	1	1	1	1	1	8	8	8	8
200	6	6	6	7	7	7	7	2	2
300	8	2	2	2	2	2	1	1	1
Def.		X		X		X	X	X	

5 défauts

1.7. (0,75 point)

Donnez l'état des tables des pages à la fin de cette séquence. Pour chaque page, vous indiquerez : la case éventuelle, les bits de présence et référence.

	case	Présence	LEX	Référence
0		0	101	0
1	300	1	101	1
2	200	1	101	1
3		0		0
4		0		0
6		0	110	0
7		0	110	
8	100	1	110	1

2. SYNCHRONISATIONS PAR SEMAPHORES (7,5 POINTS)

Nous considérons deux types des processus. Les processus PA exécutent la fonction *calculA()* et les processus PB la fonction *calculB()*. PA et PB bouclent pour toujours. Le nombre de processus PA et PB est quelconque.

Processus PA :	Processus PB :
<pre>while (1) { // à compléter calculA(); // à compléter }</pre>	<pre>while (1) { // à compléter calculB(); // à compléter }</pre>

Dans les questions qui suivent :

- N est définie comme une constante
- Pour créer et initialiser un sémaphore vous utiliserez la fonction *SEM*CS (int val)* et une variable partagée doit être déclarée en utilisant *shared* (ex. *shared int x*).
- Les solutions que vous proposerez doivent être le plus parallèle possible.

Note: 1) On rappelle qu'il est interdit d'initialiser les sémaphores de Dijkstra à des valeurs strictement négatives. 2) Il n'est pas possible de consulter la valeur du compteur d'un sémaphore.

Les trois questions suivantes sont indépendantes.

2.1. (2,5 points)

Nous voulons modifier le programme pour que les processus PB se terminent après qu'au moins N *calculA()* aient été réalisés globalement par les processus PA.

Programmez cette synchronisation en complétant le code des processus PA et PB. Vous préciserez les sémaphores et variables partagées nécessaires pour une telle synchronisation ainsi que leur valeur initiale.

```
Shared int nb_A=0;  
mutex= CS (1)  
  
Processus PA :  
while (1) {  
    calculA() ;  
    P(mutex)
```

```

    nb_A ++;
    V(mutex)

Processus B :
    while (1) {
        calculB() ;
        P(mutex)
        if (nb_A >= N) {
            V(mutex)
            return ;
        }
        V(mutex)
    }

```

2.2. (2,5 points)

Nous voulons maintenant modifier le programme original pour que globalement les processus PA exécutent N calculA puis ensuite les processus PB exécutent N calculB et ainsi de suite. Par conséquent, l'exécution des processus produira, dans l'ordre :

N fois calculA, N fois calculB, N fois calculA, N fois calculB ...

Programmez cette synchronisation en complétant le code des processus PA et PB. Vous préciserez les sémaphores et variables partagées nécessaires pour une telle synchronisation ainsi que leur valeur initiale.

```

sem_A= CS (N); sem_B= CS 0; /*PAs ont le droit à N CalculA ; PBs bloqués
*/
shared int nb=0;

Processus PA :
    while (1) {
        P(sem_A)
        calculA();
        P(mutex)
        nb++;
        if (nb == N) {
            nb=0;
            for (i=0; i < N; i++)
                V(sem_B);
        }
        V(mutex);
    }

Processus PB :
    while (1) {
        P(sem B)
        calculB();
        P(mutex)
        nb++;
        if (nb = N) {

```

```

    nb=0;
    for (i=0; i < N; i++)
        V(sem_A);
    }
    V(mutex);
}

```

2.3. (2,5 points)

Nous désirons modifier encore une fois le programme original de sorte que le premier calculB () ne soit exécuté que lorsqu'au moins N calculA () ont été réalisés globalement. Après, les processus du type PA et PB peuvent s'exécuter sans contrainte et bouclent pour toujours.

Programmez cette synchronisation en complétant le code des processus PA et PB. Vous préciserez les sémaphores et variables partagées nécessaires pour une telle synchronisation ainsi que leur valeur initiale.

Note: Aucun processus doit rester bloqué pour toujours après les N CalculA().

```

shared int nb_A=0; /* contrôler le nombre de calculA */
shared int nb_B=0; /* contrôler le nombre de PB bloqués */
mutexA= CS (1); mutexB = CS (1); /* pour nb_A et nb_B */
sem_B=CS (0); /* bloque B */

```

Processus PA :

```

while (1) {
    calculA() ;
    P(mutexA)
    nb_A++ ;
    if (nb_A == N) {
        P(mutexB)
        for (i=0; i < nb_B; i++)
            V(sem_B);
        V(mutexB)
    }
    V(mutexA);
}

```

Processus B :

```

while (1) {
    P(mutexA)
    if (nb_A < N) { /* se bloquer */
        P(mutexB);
        nb_B++;
        V(mutexB);
        V(mutexA);
        P(B);
    }
    else
        V(mutexA);
    calculB() ;
}

```

```
}
```

Autre solution possible avec 1 seul compteur

Processus PA :

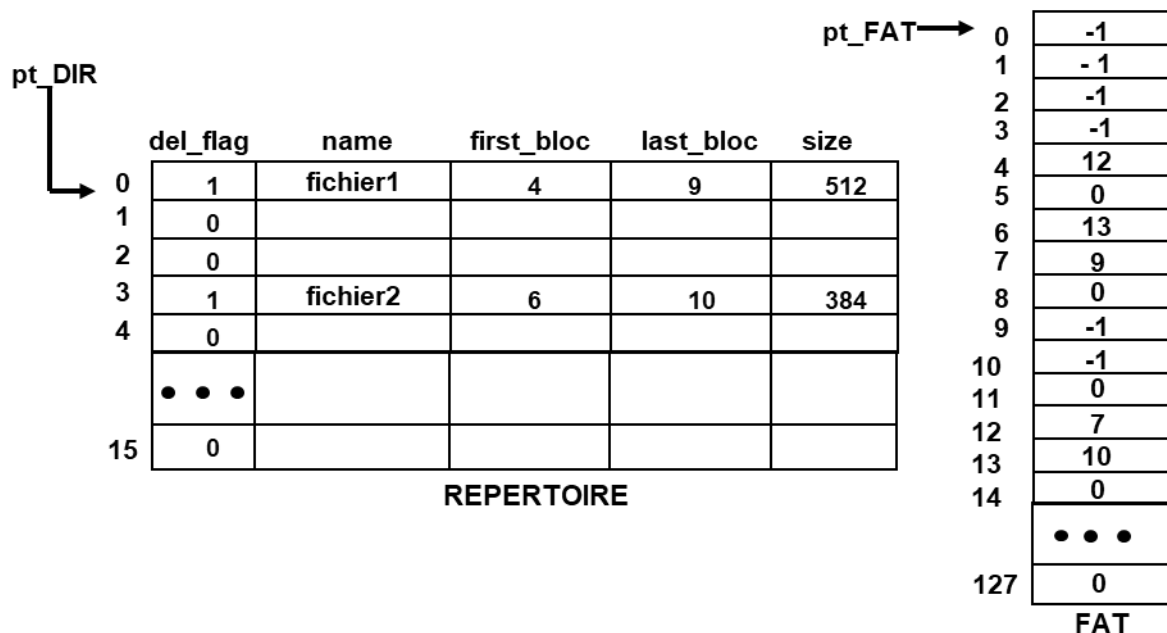
```
while (1) {  
    calculA() ;  
    P(Mutex)  
    nb++ ;  
    if (nb == N)  
        V(sem_B) ;  
    V(Mutex)  
}
```

Processus B :

```
while (1) {  
    P(sem_B) ;  
    V(sem_B) ;  
    calculB() ;  
}
```

3. FICHIERS (5 POINTS)

On considère la gestion de la FAT vue en TME dont le format est le suivant :



Le pointeur *pt_FAT* pointe vers le début de la FAT et le pointeur *pt_DIR* pointe vers le début du répertoire. La taille des blocs (secteurs) est de 128 octets.

Chaque entrée du répertoire possède les champs suivants :

- **del_flag** : 0 indique que l'entrée est libre ; 1 indique que l'entrée est occupée.
- **name** : nom du fichier
- **first_bloc, last_bloc**: premier et dernier bloc du fichier.
- **size** : taille du fichier.

Les entrées 14 à 127 de la table pt_FAT contiennent 0.

3.1. (1 point)

- Quels sont les blocs composant les fichiers « fichier1 » et « fichier2 » ?
- Quels sont les blocs libres ?

fichier1 : 4 12 7 9

fichier2 : 6 13 10

Libres : 5, 8, 11, 14 à 127

On veut écrire une fonction `int addfirst(char *fich, int b)` qui alloue le bloc libre `b` **au début du fichier** `fich`. La fonction doit retourner -1 en cas d'erreur : si le fichier n'existe pas ou si `b` n'est pas un bloc libre. Dans les autres cas, elle retourne 0.

3.2. (1 point)

Donnez la nouvelle configuration des entrées du répertoire et de la FAT qui ont été modifiées à la suite de l'appel à `addfirst("fichier1", 5)`.

```
pt_DIR[0].first_bloc = 5;
pt_DIR[0].size = 640;
pt_FAT[5] = 4
```

3.3. (3 points)

Donnez le programme C de la fonction `addfirst`.

```
int addfirst(char *fich, int b){
    int i = 0;
    struct ent_dir * pt = pt_DIR;

    if (pt_FAT[b] != 0)
        return -1;

    for (i=0; i< NB_DIR; i++) {
        if (!pt->del_flag || strcmp (pt->name, file)))
```

```
    pt++;  
else  
    break;  
}  
if (i == NB__DIR)  
    return -1;  
pt_FAT[b] = pt->first;  
pt->first_bloc = b;  
pt->size += 128;  
return 0 ;  
}
```