Examen LU3IN010

L3 – Licence d'Informatique -Mai 2023

2h - Aucun document autorisé - Barème donné à titre indicatif

1. QUESTIONS DE COURS (1,5 POINTS)

Les réponses à ces questions doivent être courtes (1-2 lignes).

1.1. (0,5 point)

La TLB est-elle située en RAM ? Justifiez.

Non, elle est située dans la mémoire interne du CPU, pour éviter des accès à la RAM lors des traductions d'adresses.

1.2. (0,5 point)

Lors d'une opération v++ sur une variable partagée en mémoire, pourquoi faut-il protéger l'accès à v avant de faire cette opération ?

v++ n'est pas une opération atomique sur le CPU, il y a un risque de commutation pendant l'opération pouvant entraîner une incohérence

1.3. (0,5 point)

Qui alloue les cases (frames) associées aux pages (le système ou le matériel) et à quel moment ?

Le système alloue les cases lors du premier défaut de page

2. MEMOIRE (4 POINTS)

On dispose d'une machine simplement paginée avec des pages de 1024 octets. On considère 2 processus P1 et P2.

Pour les 2 processus, leur **code** commence à l'adresse virtuelle 0 et se termine à l'adresse virtuelle 2047 octets, leur **pile** est située entre les adresses virtuelles 10240 et 20479 et leur zone contenant les **données** est située entre les adresses virtuelles 4096 et 8191.

La page 1 de P1 et P2 est stockée dans la case 1000 en RAM. La page 5 de P1 est stockée dans la case 2000 et la page 11 de P2 dans la case 4000.

2.1. (0,5 point)

Quels sont les numéros des pages associés au code, à la pile et aux données ?

Code : pages 0 et 1 (0 div 1024 = 0, 2047 div 1024 = 1)
pile : pages entre 10 et 19
données: pages entre 4 et 7

2.2. (0,5 point)

Les variables *a* et *b* de P1 situées respectivement aux adresses 20000 et 5000 sont-elles des variables locales ou globales ? Justifiez.

a est locale, elle fait partie de la pile.
b est globale, elle fait partie des données.

2.3. (0,5 point)

Pourquoi P1 et P2 ont leur page 1 stockée dans la même case mémoire ?
Serait-il possible que P2 stocke aussi sa page 5 dans la case 2000 ?

P1 et P2 partagent le même code en lecture
Ce n'est pas possible pour P2 de partager la case 2000 (il n'y a de partage des données)

2.4. (1 point)

Faites un schéma des tables des pages des processus P1 et P2 en précisant les cases, les droits et le bit de présence.

TP P1
<page,case,P,droits>

<0,-,0, LX>
<1,1000,1, LX>
…
<4,-,0,LE>
<5,2000,1,LE>
<6,-,0,LE>
<7,-,0,LE>

<10,-,0,LE>
…
<19,-,0,LE>

TP P2
<0,-,0, LX>
<1,1000,1, LX>
…
<4,-,0,LE>
…

<7,-,0,LE>

<10,-,0,LE>
<11,4000,1,LE>
<12,-,0,LE>
…
<19,-,0,LE>

2.5. (1, 5 points)

Que se passe-t-il lorsque :
a) P1 fait un accès en écriture à l'adresse 5200 ?
b) P1 fait un accès en lecture à l'adresse 10260 ?
c) P2 fait un accès en écriture à l'adresse 1200 ?

Vous préciserez les actions faites par le matériel et celles faites par le système. S'il n'y a pas d'erreur ou de défaut de page pendant l'accès, vous donnerez l'adresse physique correspondant à l'accès.

a) MMU : accès <5,80>, droit Ok, Présence Ok, ad physique 2000*1024 + 80 = 2048080
a) MMU : accès page <10,20>, IT défaut de page , Système : traite défaut de la page 10
c) MMU : IT faute sur droit d'accès page 1, Système : traite l'erreur et tue P2

3. REMPLACEMENT DE PAGES (4 POINTS)

L'algorithme NRU favorise le maintien en mémoire des pages récemment utilisées. La version de NRU proposée ici fonctionne de la manière suivante.

Deux bits sont associés à chaque page : un bit R de référencement, et un bit M de modification. Lorsqu'une page est utilisée, son bit R est positionné à 1 ; s'il s'agit d'un accès en écriture, alors son bit M est également positionné à 1. Une interruption horloge vient régulièrement remettre à 0 tous les bits R des pages chargées en mémoire. On obtient ainsi une hiérarchie des pages présentes en mémoire :

- Priorité 3 : référencée, modifiée (R=1, M=1)
- Priorité 2 : référencée, non modifiée (R=1, M=0)
- Priorité 1 : non référencée, modifiée (R=0, M=1)
- Priorité 0 : non référencée, non modifiée (R=0, M=0)

Lors d'un remplacement de page, la page victime est celle de plus faible priorité (priorité 0 correspond à la plus faible priorité). Si plusieurs pages tombent dans cette catégorie on applique l'ordre FIFO.

Un processus P fait la suite d'accès suivant aux pages.

0L 1L 5E 4L 1L 3E 5L 4L 5E 6E

Le programme qui effectue ces accès dispose de 3 cases mémoire, les cases 10, 20 et 30. La référence 1E représente un accès en écriture à la page 1, tandis que la référence 0L représente un accès en lecture à la page 0. Les bits R sont remis à 0 tous les 4 référencements.

3.1. (3 points)

Remplissez le tableau suivant en indiquant pour chaque accès : s'il y a eu un défaut de page ou non et la liste des pages présentes en mémoire, ordonnées de la plus récemment chargée à la plus anciennement chargée. Pour chaque élément de cette liste, vous devrez utiliser la notation X(C,P) : page X dans case C à la priorité P.

Page	Défaut	Pages en mémoire
0L		
1L		
5E		
4L		
Horloge		
1L		
3E		
5L		
4L		
Horloge		
5^E		
6E		

Page	Défaut	Pages en mémoire
0L	D	0(10,2)
1L	D	1(20,2) 0(10,2)
5E	D	5(30,3) 1(20,2) 0(10,2)
4R	D	4(10, 2) 5(30,3) 1(20,2)
Horloge		4(10,0) 5 (30,1) 1 (20,0)
1L		4(10,0) 5 (30,1) 1(20,2)
3^E	D	3(10,3) 5 (30,1) 1(20,2)
5L		3(10,3) 5(30,3) 1(20,2)
4L	D	4(20,2), 3(10,3) 5(30,3)
Horloge		4(20,0) 3(10,1) 5(30,1)
5^E		4(20,0) 3(10,1) 5(30,3)
6E	D	6(20,3) 3(10,1) 5(30,3)

3.2. (1 point)

Combien de défauts de pages sont causés par cette suite de références ? Donnez la table des pages du processus à la fin de cette séquence en indiquant notamment les bits de présence, R et M.

7 défauts de pages

Table des pages

page	P	case	R	M
3	1	10	0	1
5	1	30	1	1
6	1	20	1	1

4. SYNCHRONISATIONS PAR SEMAPHORES (6,5 POINTS)

On considère un nombre quelconque de processus de calcul, ProcCal et **K jetons disponibles**. Tant qu'il y a des jetons disponibles, les processus peuvent appeler la fonction calcul(). Cependant, s'il n'y a plus de jeton, les processus ProcCal doivent se bloquer.

Un processus libérateur, ProcLib, attend qu'il n'y ait plus de jetons, alors il débloque les processus de calcul éventuellement en attente et met à nouveau à disposition K jetons.

Les processus sont cycliques :

```
1.    void ProcCal() {
2.    while (1) {
3.      /* Demander 1 jeton*/
4.      …
5.      calcul();
6.    }
```

```
1.    void ProcLib() {
2.    while (1) {
3.      /* Attendre 0 jeton disponible
          */
4.       …
5.    }
```

Dans les questions qui suivent :

- *K* 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.

4.1. (1 points)

Quels sont les sémaphores et variables partagées nécessaires à cette synchronisation. Vous préciserez les valeurs initiales.

shared int count =0 ; nombre de ressource + processus bloqué ;

Sémaphores :
SEM * Mutex = CS(1) ; /* protéger l'accès à count */
SEM* Ressource= CS(K) ; /* contrôler le nombre de ressources disponibles ; initialisé à K */
SEM *Lib=CS(0); /* bloquer le libérateur */

4.2. (3 points)

Complétez les codes des processus *ProdCons* et *ProcLib* pour réaliser cette synchronisation.

```
void ProcCal() {
  while (1) {
   P(Mutex);
   count++;
   if (count == K)
      V(Lib);
   V(Mutex);
   P(Ressource);
   /* acces ressource */
  }
}

void ProcLib() {
  while (1) {
   P(Lib);
   P(Mutex);
   for (int i; i < K; i++)
      V(Ressouce)
   count = 0;
   V(Mutex)
  }
}
```

4.3. (2,5 points)

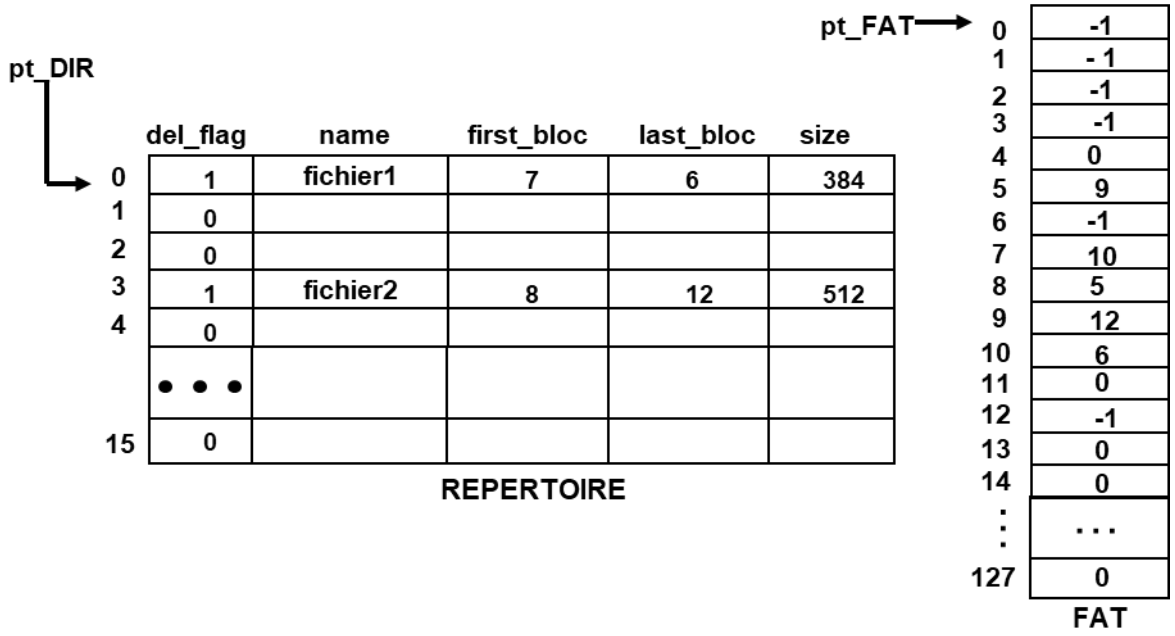
On suppose maintenant qu'il y a N processus ProcLib numérotés de 0 à N-1. La fonction ProcLib prend maintenant en paramètre le numéro du ProcLib : `void ProcLib(int i)`. Lorsque tous les jetons ont été pris, les N ProcLib doivent maintenant être réveillés. Chacun appelle la fonction calcul_lib() puis uniquement le dernier des ProcLib élu débloque les processus de calcul en attente et met à nouveau à disposition K jetons.

Modifiez le programme en conséquence en indiquant les éventuelles variables partagées et sémaphores supplémentaires avec leur initialisation.

```
shared int countlib = 0; /* compteur des proc lib */

MutexLib = CS(1);    /* protection countlib */
SEM* Lib[N];
for (int i = 0; i< N; i++)
  Lib[i] = CS(0);

void ProcCal() {
  while (1) {
   P(Mutex);
   count++;
   if (count == K)
      for (int i=0; i<N; i++)
         V(Lib[i]);
   V(Mutex);
   P(Ressource);
   /* acces ressource */
  }
}

void ProcLib(int i) {
  while (1) {
   P(Lib[i]);
   P(MutexLib);
   countlib++;
   if (countlib == N){
     P(Mutex);
     for (int i; i < count; i++)
        V(Ressouce)
      count = 0;
     V(Mutex)
     countlib = 0;
   }
   V(MutexLib);
}
```

5. FICHIERS (4 POINTS)

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



Le pointeur short* pt_FAT pointe vers le début de la FAT et le pointeur struct ent_dir* 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 13 à 127 de la table pt_FAT contiennent 0.

La fonction **int write_DIR_FAT_sectors ( )** permet d'écrire sur le disque les secteurs contenant la FAT et le répertoire.

5.1. (1 point)

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

fichier1 : 7 10 6
fichier2 : 8 5 9 12

Libres : 4, 11, 13 à 127

On veut écrire une fonction **`int reduceFile(char *fich, int p)`** qui supprime les p premiers blocs du fichier fich. Si p est supérieur ou égal au nombre de blocs du fichier, celui-ci est supprimé.

La fonction doit retourner -1 en cas d'erreur : si fich n'existe pas. Dans les autres cas, elle retourne 0.

5.2. (0,5 point)

Donnez la nouvelle configuration des entrées du répertoire et de la FAT qui ont été modifiées à la suite de l'appel à reduceFile("fichier2", 2).

```
pt_DIR[3].first_bloc = 9;
pt_DIR[3].size = 256;
pt_FAT[8] = 0;
pt_FAT[5] = 0;
```

5.3. (2,5 points)

Donnez le programme C de la fonction reduceFile.

```
int reduceFile(char *fich, int p){
  int i,nb;
  struct ent_dir * pt = pt_DIR;

/* Rechercher fich */
  for (i=0; i< NB_DIR; i++) {
    if (pt->del_flag != 0 &&  (strcmp (pt->name, file)))
      break;
    pt++;
  }
   if (i== NB_DIR)
      return -1;

/* Mettre à jour FAT */

  m = pt->first_bloc.
 nb = p;
  while (m != -1 && nb!=0) {
      aux = pt_FAT[m];
     pt_FAT[m] = 0;
     m = aux;
     nb--;
  }
 if (m==-1) ({ /* Supprimer entrée DIR */
    pt->del_flag = 0;
  }
 else {
   pt->first_bloc = m;
   pt->size -= p*128;
 }
```

```
  write_DIR_FAT_sectors ( );
  return 0 ;
}
```