

**Premier examen réparti 3I010**  
**L3 – Licence d’Informatique -Année 2016**  
**1h45 - Tout document papier autorisé**  
**Barème donné à titre indicatif**

---

## **I. ORDONNANCEMENT (7 POINTS)**

---

On considère un algorithme d'ordonnancement en temps partagé de type tourniquet avec **un quantum de 100 ms**.

Il y a une file des tâches en mémoire. A la création, une nouvelle tâche est insérée à l'état Prêt en queue de file. L'ordonnanceur élit la tâche à l'état Prêt la plus en tête. En cas d'entrées/sorties (E/S) la tâche élue passe à l'état Bloqué mais **reste en tête de la file**. En fin de quantum la tâche est réinsérée en queue de file.

Lorsqu'une tâche est allouée sur le processeur (i.e., elle devient élue), le système lui attribue soit un quantum entier ou le reste de son quantum si elle était précédemment bloquée (i.e., le restant du dernier quantum qu'elle n'a pas utilisé dû à son blocage).

---

### **I.1. (1 point)**

En cas d'élection d'une tâche précédemment bloquée pourquoi le système ne lui alloue pas un quantum entier ? Quel est le risque de lui allouer un quantum ?

Risque que des tâches faisant des petites E/S monopolisent le processus (famille possible)

Soit le scénario suivant :

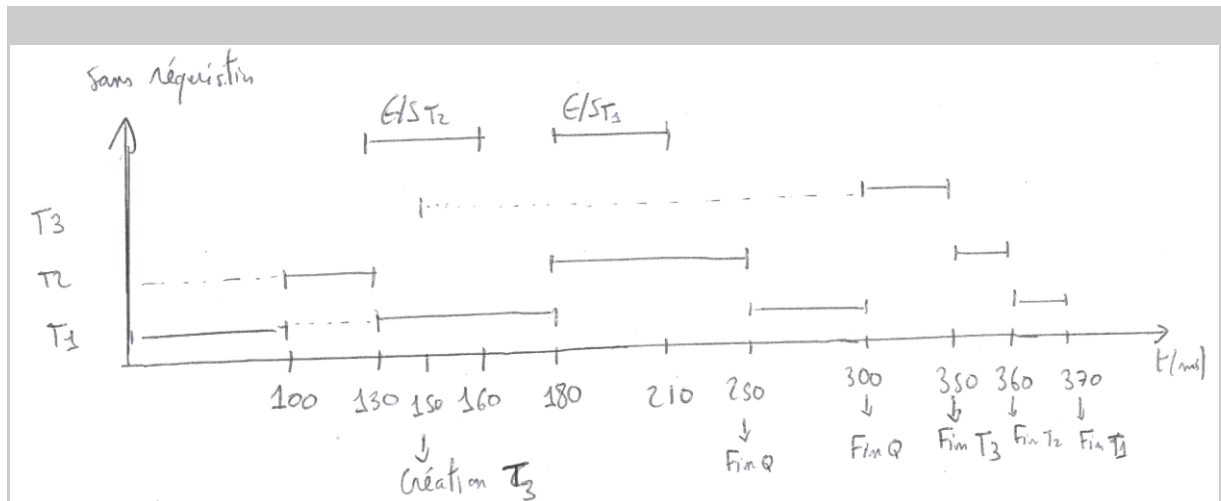
Tâches	Instant création	Durée d'exécution sur le processeur	E/S
T1	0 ms	210ms	Une seule après 150 ms
T2	0 ms	110ms	Une seule après 30 ms d'exécution
T3	150 ms	50ms	Aucune

On suppose que T1 est créée avant T2. Les entrées/sorties durent **30 ms** et se font sur le même disque (i.e., il y a une seule unité d'échange)

### I.2. (3 points)

On considère une stratégie **sans réquisition**.

- Faites un diagramme temporel (Gantt) de l'évolution des tâches
- Indiquez les temps de réponse des tâches.
- Indiquez l'état la file d'attente juste après la création de la tâche T2 (à l'instant 150 ms)



Temps réponse :

T1 = 370 ms

T2 = 360 ms

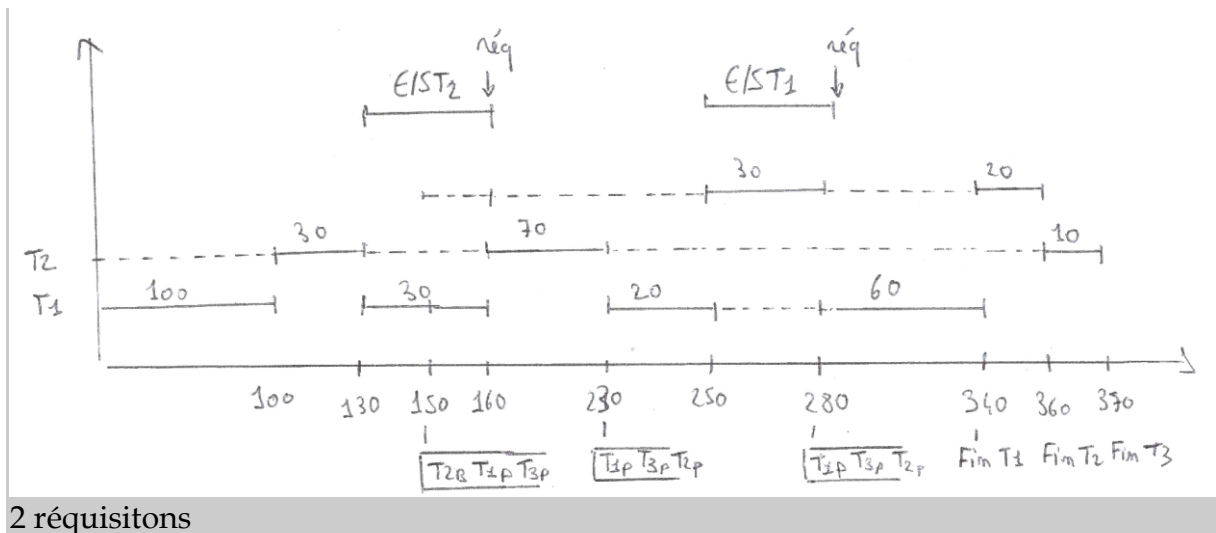
T3 = 350 - 150 = 200ms

File : (T2,b), (T1,p), (T3,p)

### I.3. (3 points)

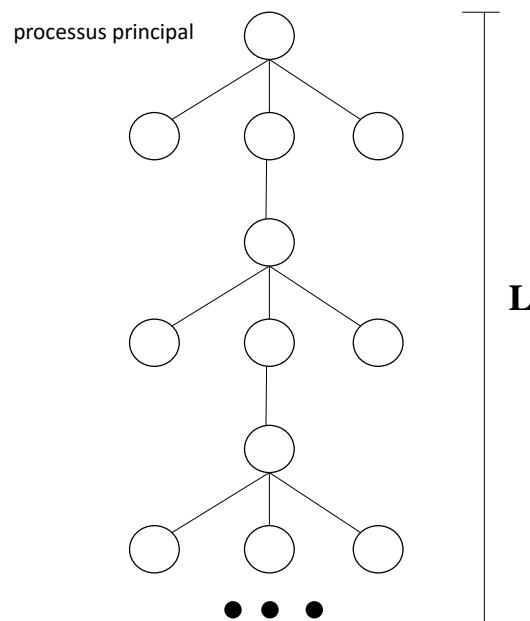
On considère maintenant une stratégie **avec réquisition** en cas de fin d'Entrée/Sortie : la tâche qui finit une d'E/S peut être directement être élue si elle précède l'ancienne tâche élue dans la file.

- Faites un diagramme temporel (Gantt) de l'évolution des tâches
- Combien y-a-t-il eu de réquisitions et à quels moments ?



## II. PROCESSUS UNIX (4 POINTS)

Considérez l'arborescence de processus ci-dessous où le programme principal crée 3 fils ; à son tour, le deuxième fils crée lui aussi 3 fils ainsi de suite jusqu'à ce que l'arborescence de processus ait une hauteur  $L$ .



### II.1. (2,5 points)

En utilisant l'appel système `fork()`, donnez le code du programme qui crée une telle arborescence. Avant de se terminer un processus affiche son `pid`. La valeur de  $L$  est définie comme une constante.

```
#define L 3
#define NB_FILS 3
```

```

pid_t p=0; int i=0; int j=1;

int main (int argc, char* argv[]){

    while ((i < L) && (p==0) && (j==1)) {

        /* creer les 3 fils */
        j=0;
        while (j<NB_FILS ) {
            if ((p=fork ()) ==0)
                break;
            else
                j++;
        }

        i++;
    }

    printf ("pid:%d \n", getpid ());
    return 0;
}

```

## II.2. (2,5 points)

Complétez le programme pour que le processus principal ne se termine qu'après tous les autres processus.

Observation : dans votre solution l'appel à la fonction *pid\_t wait (int\* val)* ne doit jamais renvoyer la valeur -1.

```

define L 3
#define NB_FILS 3

pid_t p=0; int i=0; int j=1;

int main (int argc, char* argv[]){

    while ((i < L) && (p==0) && (j==1)) {

        /* creer les 3 fils */
        j=0;
        while (j<NB_FILS ) {
            if ((p=fork ()) ==0)
                break;
            else
                j++;
        }

        i++;
    }

    printf ("pid:%d \n", getpid ());

    if (p!= 0) /* ou j==NB_FILS */
        for (i=0; i< NB_FILS; i++)

```

```

    wait (NULL);

    return 0;
}

```

### III. SEMAPHORE - INTERBLOCAGE (3 POINTS)

Soient les 4 processus suivants manipulant trois variables partagées A,B et C. Les variables sont protégées par les 3 sémaphores MutexA, MutexB et MutexC initialisés à 1.

P1	P2	P3	P4
(1.1) P(MutexA);	(2.1) P(MutexC);	(3.1) P(MutexB);	(4.1) P(MutexC);
(1.2) P(MutexB);	(2.2) P(MutexA);	(3.2) P(MutexC);	(4.2) C=C+1;
(1.3) A = A +B;	(2.3) C=C+A;	(3.3) B=B+C;	(4.3) V(MutexC);
(1.4) V(MutexA);	(2.4) V(MutexA);	(3.4) V(MutexC);	
(1.5) V(MutexB);	(2.5) V(MutexC);	(3.4) V(MutexB);	

#### III.1. (1,5 points)

Indiquez un scénario d'exécution aboutissant à un interblocage.

```

(1.1)
(3.1)
(2.1)
(4.1)

```

#### III.2. (1,5 points)

Modifiez le programme pour éviter l'interblocage. Attention, on souhaite préserver le même niveau de parallélisme tout en assurant la protection des trois variables partagées.

Inverser (2.1) et (2.2)

### IV. SYNCHRONISATION – ECRIVAIN / LECTEURS (6 POINTS)

Nous considérons **un** processus écrivain et **NL** processus lecteur.

Les processus *écrivain* et *lecteurs* sont cycliques. Autrement dit, chaque processus exécute une boucle infinie :

```

ecrivain ( ) {
    while (1) {
        ...
    }
}

lecteur (int i) {
    while (1) {
        ...
    }
}

```

}

}

L'argument  $i$  ( $0 \leq i < NL$ ) de *lecture* (*int i*) correspond à l'identifiant d'un processus lecteur.

L'écrivain écrit des valeurs sur une variable  $x$  partagée, du type entier, initialement vide. Il ne peut pas y avoir simultanément un accès en lecture et un accès en écriture sur  $x$ .

Plusieurs lectures de  $x$  peuvent avoir lieu simultanément. Cependant, la 1<sup>ère</sup> valeur écrite sur  $x$  par le processus écrivain ne doit être lue **une fois** que par  $M = NL/2$  différents lecteurs avant de pouvoir être effacée par l'écriture d'une 2<sup>ème</sup> valeur. Notez que les  $M$  lecteurs ne sont pas choisis. Autrement dit, parmi les  $NL$  lecteurs,  $M$  lecteurs (pas forcément ceux dont l'identifiant est compris entre 0 et  $M-1$ ) vont lire et afficher la valeur de  $x$  mais pas l'autre moitié ( $NL-M$ ) qui devra après lire la 2<sup>ème</sup> valeur écrite. La procédure de lecture se répète alors : la 3<sup>ème</sup> valeur écrite sur  $x$  par l'écrivain doit être lue par  $M = NL/2$  lecteurs parmi  $NL$  et la 4<sup>ème</sup> par l'autre moitié de lecteurs qui n'ont pas lu la 3<sup>ème</sup> valeur, ainsi de suite, pour toujours.

Pour simplifier, nous considérons que  $NL$  (nombre de lecteurs) est une valeur paire.

#### IV.1. (6 points)

Donnez le pseudo code des fonctions *écrivain* ( ) et *lecteur* (*int i*).

Spécifiez aussi les sémaphores et variables partagées utilisées ainsi que leur initialisation.

Les variables partagées doivent être déclarées en utilisant la notation *shared* (exemple : *shared int x*). Pour les sémaphores, vous devez utiliser les fonctions définies en TD : *SEM \*CS(cpt)*, *DS(sem)*, *P(sem)*, *V(sem)* respectivement pour la création d'un sémaphore, destruction d'un sémaphore, demande d'acquisition d'une ressource et libération d'une ressource.

**Suggestion** : inspirez-vous sur la solution du premier exercice de synchronisation du TME 5-6 (solution à une case)

```
# define NL 4
# define M NL/2

SEM EMET = CS (1);
SEM MUTEX = CS(1);

SEM VRECEP [NL];
SEM FILE = CS (0) ; /* bloquer les M lecteurs */

for (i=0 ; i< NL ; i++)
    VRECEP [NL] = CS (0);
```

```

shared int x, nb_lec ;

ecrivain( ) {

    int k ;
    int cpt =0 /* compteur écriture */;
    while (1) {
        P(EMET)
        x = valeur ;
        cpt ++ ;
        if (cpt %2 == 1)
            for k=0 ; k< NL, k++
                V(RECEP[k]) ;

        For (k=0; k<M; k++)
            /* réveiller M lecteurs */
            V(FILE)

    }
}

lecteur (int i){
    while (1) {
        P(RECEP[i]);
        P(FILE);

        P(mutex)
        nb_lec++;
        if (nb_lec == M) {
            nb_lec =0;
            V(EMET);
        }
        V(mutex);
    }
}

```