

**1. ORDONNANCEMENT (8 PTS)**

On considère un ordonnancement de type **batch** (sans quantum).

On applique une variante de l'algorithme SJF prenant en compte non seulement le temps CPU restant d'une tâche mais également son temps d'attente. La tâche élue est la tâche prête avec **le plus grand ratio R** défini de la manière suivante :

$$R_i = \frac{\text{Temps\_attente}_i + \text{Durée\_CPU}_i}{\text{Durée\_CPU}_i}$$

où  $\text{Temps\_attente}_i$  = le temps pendant lequel une tâche ne s'est pas exécutée depuis sa création (càd, le temps où elle n'a pas été élue)

$\text{Durée\_CPU}_i$  = le temps CPU restant pour  $T_i$

Soit le scénario suivant :

Tâches	Instant création	Durée d'exécution sur le processeur	Entrées/Sorties (E/S)
T1	0 ms	80 ms	aucune E/S
T2	40 ms	40 ms	une seule E/S de 20 ms après 30 ms d'exécution
T3	40 ms	80ms	aucune E/S
T4	90 ms	10 ms	aucune E/S

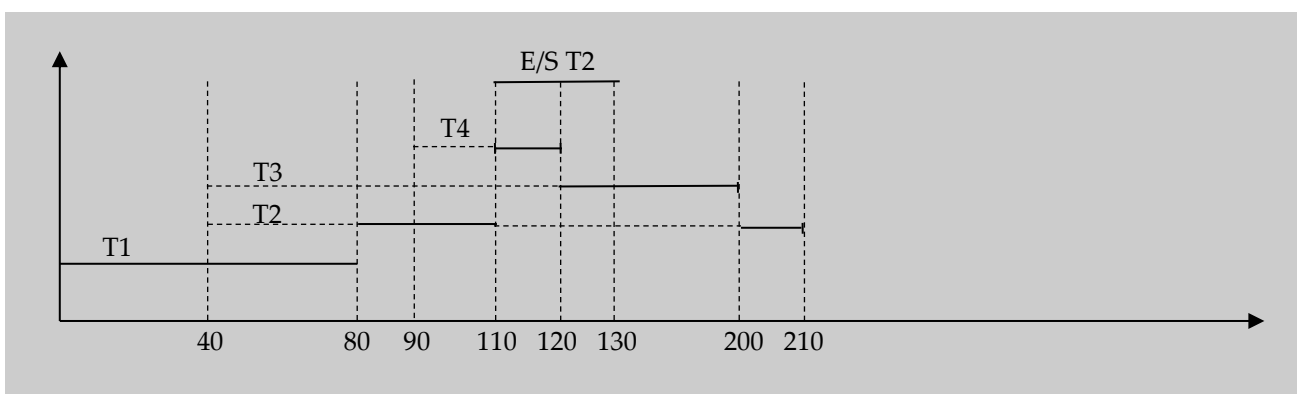
**1.1 (4 points)**

On considère une stratégie **sans réquisition**.

- Faites un diagramme temporel (Gantt) de l'évolution des tâches.

Vous indiquerez également les valeurs des ratios au moment où une nouvelle tâche est élue.

- Indiquez les temps de réponse des tâches.



$t=80\text{ ms}$

$$R_2 = (40+40)/40 = 2$$

$$R_3 = (40+80)/80 = 1,5$$

$t=110\text{ms}$

$$R_3 = (70+80)/80 = 15/8 = 1,875$$

$$R_4 = (20+10)/10 = 3$$

$t=120\text{ms}$

$$R_3 = (80+80)/80 = 2$$

$t=200\text{ms}$

$$R_2 = ((40+90) + 10)/10 = 14$$

$$TR_1 = 80 - 0 = 80\text{ms}$$

$$TR_2 = 210 - 40 = 170\text{ms}$$

$$TR_3 = 200 - 40 = 160\text{ms}$$

$$TR_4 = 120 - 90 = 30\text{ms}$$

### 1.2 (1 point)

Y-a-t-il un risque de famine avec cette stratégie ? Justifiez votre réponse.

Non car plus une tâche attend plus son ratio augmente et quelle que soit leur durée, à leur création les tâches ont un ratio égal à 1.

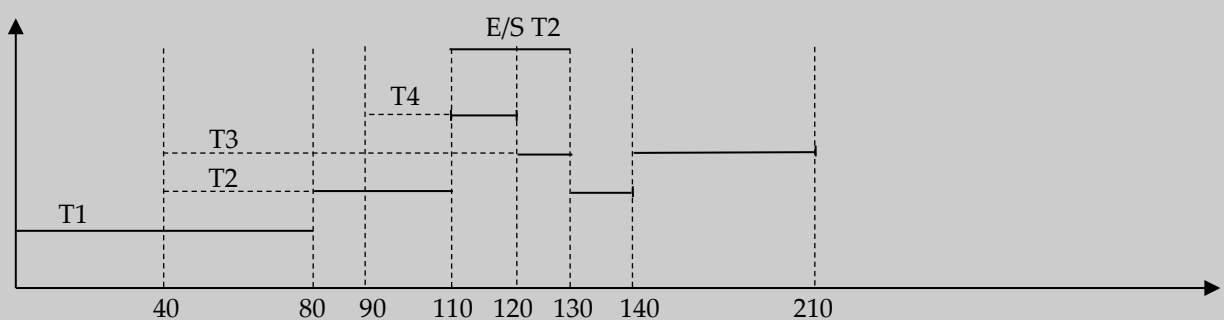
### 1.3 (3 points)

On considère maintenant qu'une **réquisition** de tâche est possible lors d'une **fin d'E/S**.

- Faites un diagramme temporel (Gantt) de l'évolution des tâches.

Vous indiquerez aussi les valeurs des ratios au moment où une nouvelle tâche est élue.

- Indiquez les temps de réponse des tâches.



Identique jusqu'à  $t=130\text{ ms}$ .

$t=130\text{ms}$

$R_2 = ((40+20)+10)/10 = 7$

$R_3 = (80+70)/70 = 15/7$

$t=140\text{ms}$

$R_3 = ((80+10) + 70)/70 = 16/7$

$TR_1 = 80 - 0 = 80\text{ms}$

$TR_2 = 140 - 40 = 100\text{ms}$

$TR_3 = 210 - 40 = 170\text{ms}$

$TR_4 = 120 - 90 = 30\text{ms}$

## 2. PROCESSUS (2,5 PTS)

Soit le programme suivant correspondant à l'exécutable «./prog »

```
1: int main(int argc, char *argv[]) {
2:   int e;
3:   int a = 1;
4:   if (argc == 2) {
5:     printf("1 : %s, %s, a=%d \n", argv[0], argv[1], a);
6:     return 0;
7:   }
8:   if (fork() == 0) {
9:     a *=2;
10:    printf("2 : a = %d \n", a);
11:    execl("./prog", "prog", "arg1", NULL);
12:    a++;
13:    printf("3 : a = %d \n", a);
14:    exit(2);
15:  }
16:  a*=3;
17:  wait(&e);
18:  printf("4 : a = %d, e = %d \n", a, WEXITSTATUS(e));
19:  return 0;
20: }
```

On suppose que les appels à fork n'échouent pas.

Le programme est appelé depuis le shell par la commande suivante :

`$ ./prog`

### 2.1 (2,5 points)

Donnez l'arbre des processus fait par ce programme ainsi que l'affichage fait par chacun des processus en précisant l'ordre dans lequel les affichages peuvent avoir lieu.

prog1 — prog2

Affichage dans l'ordre :

prog2

«2 : 1 a = 2 »

«1 : prog, arg1, a=1 »

prog1 «4: a = 3, e = 0 »

## 3. SYNCHRONISATION (3,5 PTS)

On considère les quatre tâches suivantes :

A	B	C	D
P(Mutex1);	P(Mutex1);	P(S1);	P(S2);
v=v*2 ; // a	v=v+3 ; // b	P(S1)	P(Mutex1);
V(Mutex1) ;	V(Mutex1) ;	P(Mutex1);	v = v *3; //d
V(S1) ;	V(S1);	v = v +1 ; //c	V(Mutex1) ;
		V(Mutex1);	
		V(S2);	

La variable partagée  $v$  est initialisée à 1 et protégée par le sémaphore Mutex1 initialisé à 1. Les sémaphores S1 et S2 sont initialisés à 0.

### 3.1 (2 points)

A la fin des 4 processus, quels sont les ordres d'exécution possibles pour les lignes a, b, c et d ? Quelles sont alors les valeurs possibles pour la variable  $v$  à la fin de ces exécutions ?

abcd ;  $v = 18$

bacd ;  $v = 27$

### 3.2 (1,5 point)

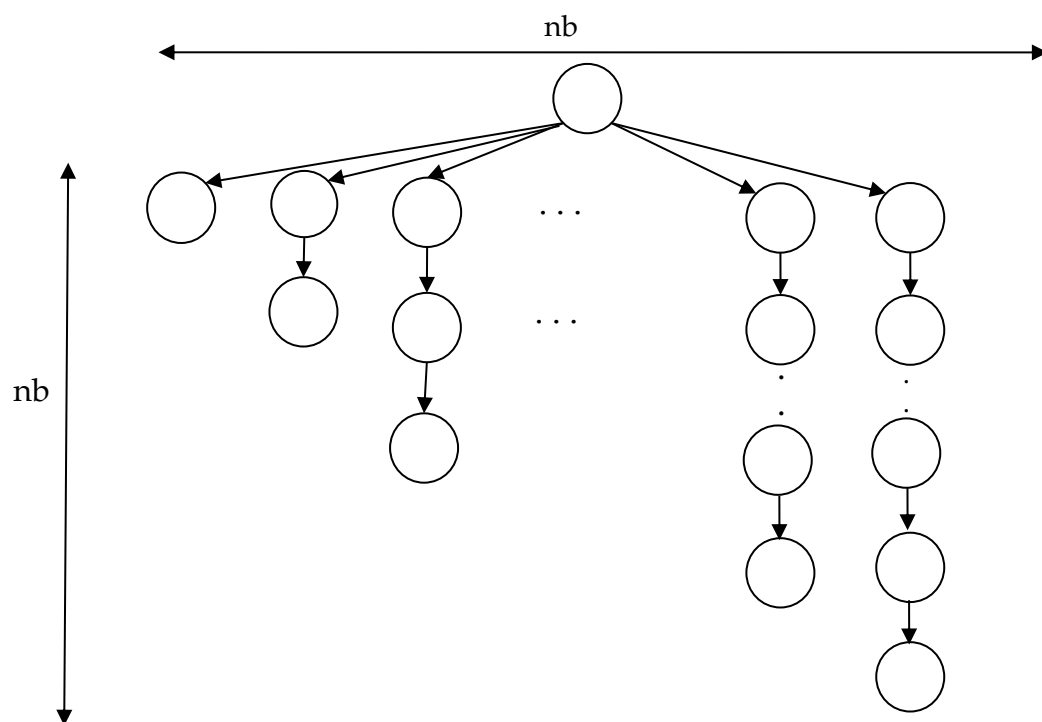
Sans modifier les lignes a, b, c et d, modifiez ce programme pour systématiquement avoir  $v=18$  à la fin de l'exécution (vous pouvez si nécessaire définir de nouveaux sémaphores, dans ce cas n'oubliez pas d'indiquer leurs valeurs initiales)

Forcer l'ordre abcd  
S3=CS(0)

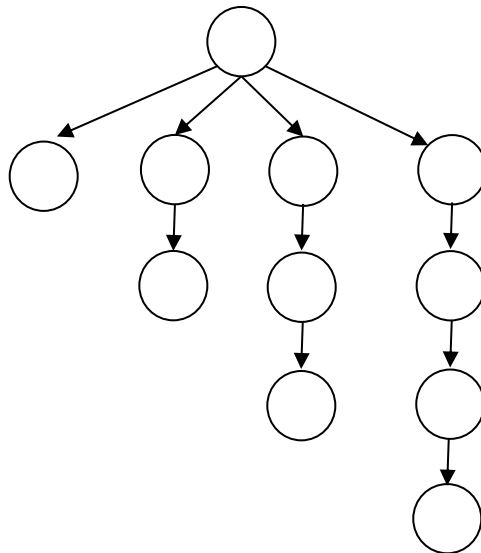
A            B  
...        P(S3)  
V(S3);    ....

#### 4. PROGRAMMATION PROCESSUS (6 PTS)

On veut écrire une fonction void créer\_arbre(int nb) qui crée l'arbre de processus suivant :



Cette fonction crée un arbre de hauteur  $nb$  ayant la structure suivante. Le processus initial commence par créer  $nb$  fils. Chacun de ces processus crée ensuite une chaîne de processus de longueur égale à son rang de création. Ainsi, le premier fils du processus initial n'aura pas de descendance, le second fils du processus initial aura uniquement un fils, le second fils du processus initial aura un fils et un petit-fils, et ainsi de suite. Par exemple, l'appel à `créer_arbre(4)` entraînera l'arbre suivant :



#### 4.1 (4 points)

Donnez le code de la fonction void creer-arbre(int nb).

```

void creer_chaine(int n){
    int i;
    for(i=0;i<n;i++){
        if(fork()>0){
            break;
        }
    }
}

void creer_arbre(int nb) {
    int nb;
    for(i=0;i<nb;i++){
        if(fork()==0) {
            creer_chaine(i);
            break;
        }
    }
}

```

#### 4.2 (2 points)

Modifiez la fonction pour qu'un processus quitte la fonction uniquement lorsque ses fils directs sont terminés. Un processus ne doit pas attendre plus de fils qu'il n'en a.

```

void creer_chaine(int n){
    int i;
    for(i=0;i<n;i++){
        if(fork()>0){
            wait(NULL);
        }
    }
}

```

```
void creer_arbre(int nb) {  
    int nb;  
    for(i=0;i<nb;i++)  
        if(fork()==0) {  
            creer_chaine(i);  
            break;  
        }  
    for(i=0;i<nb;i++)  
        wait(NULL);  
}
```