

**1. ORDONNANCEMENT (8 PTS)**

Soit un système à temps partagé avec des **quantums de 100 ms**. Pour simplifier, il n'y a pas de tick. A la création d'une tâche on donne sa date limite d'échéance (son deadline) indiquant **la date maximum à laquelle elle doit se terminer**.

On applique une stratégie d'ordonnancement EDF pour **Earliest Deadline First** qui consiste à choisir la tâche prête dont l'échéance est la plus proche. Si une tâche n'a pas utilisé tout son quantum, la prochaine fois qu'elle est élue, on lui attribue **son quantum restant**.

Si la tâche élue atteint la date de son échéance, elle est **détruite** par le système. La destruction de la tâche entraîne alors une commutation.

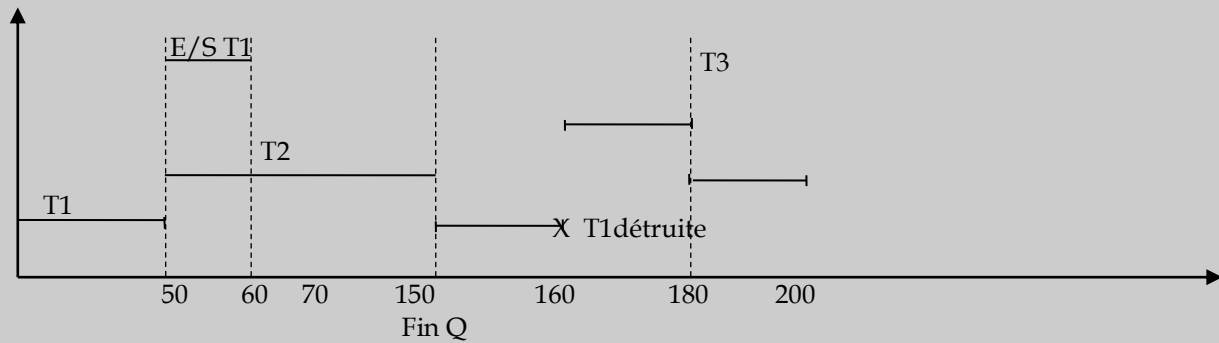
Soit le scénario suivant :

Tâches	Instant création	Durée d'exécution sur le processeur	Entrées/sorties (E/S)	Echéance (deadline)
T1	0 ms	110ms	1 E/S de 10 ms après 50 ms d'exécution	160 ms
T2	0 ms	120ms	Aucune	300 ms
T3	70 ms	20ms	Aucune	190 ms

**1.1 (3 points)**

On considère une stratégie **sans réquisition**.

- Faites un diagramme temporel (Gantt) de l'évolution des tâches en indiquant clairement les fins de quantum et les E/S.
- Y-a-t-il des tâches qui ont été détruites avant leur fin, si oui à quel moment et combien de temps se sont elles exécutées ?
- Indiquez les temps de réponse des tâches pour les tâches non détruites.



T1 est détruite à 160 ms, exécution pendant 60 ms.

$T_R T2 = 200 \text{ ms}$

$T_R T3 = 180 - 70 = 110 \text{ ms}$

On considère maintenant une stratégie **avec réquisition** possible en fin d'Entrée/Sortie ou à la création des tâches.

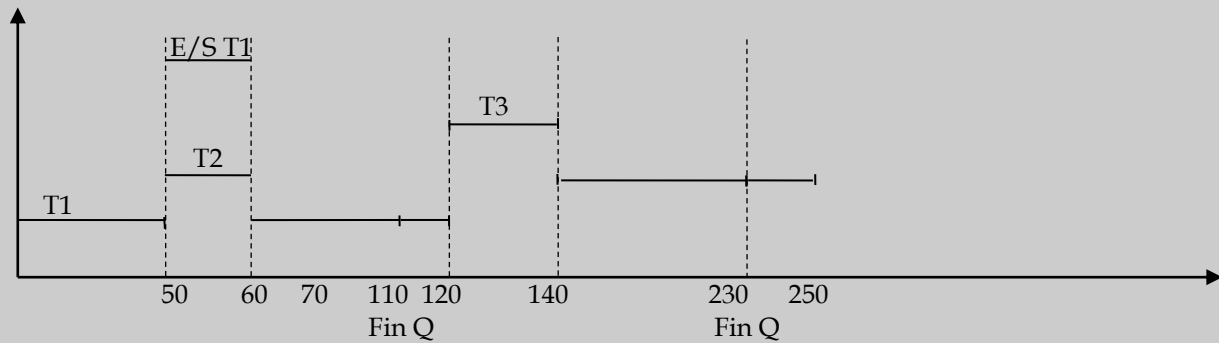
### 1.2 (1 point)

- Dans quels cas une tâche peut ne pas utiliser tout son quantum ?
- Comment le système connaît la valeur du quantum restant d'une tâche ?

- 3 cas : demande d'E/S, réquisition (création, Fin d'E/S), fin de la tâche, destruction
- Le quantum restant correspond à la valeur dans le registre horloge

### 1.3 (3 points)

- Faites un diagramme temporel (Gantt) de l'évolution des tâches avec la réquisition en indiquant clairement les fin de quantum et les E/S.
- Combien y-a-t-il eu de réquisitions et à quels moments ?
- Y-a-t-il des tâches qui ont été détruites avant leur fin ?
- Indiquez les temps de réponse des tâches.



1 seule réquisition à  $t=60$   
Pas de tâche détruite

$T_R T1 = 120 \text{ ms}$

$T_R T2 = 250 \text{ ms}$

$T_R T3 = 140 - 70 = 70 \text{ ms}$

#### 1.4 (1 point)

Comment le système peut être prévenu que la tâche élue a atteint son deadline ? Proposez une solution en quelques lignes.

Le système peut être prévenu par l'IT horloge. Lors de l'élection d'une nouvelle tâche, il suffit de positionner le registre horloge à la valeur  $\text{Min}(\text{Quantum restant}, \text{Deadline} - \text{date courante})$

## 2. PROCESSUS (4 PTS)

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

```

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

```

```

18:         exit(3);
19:     }
20: }
21: else {
22:     wait(&e);
23:     printf("a : %d, e : %d\n", a, WEXITSTATUS(e));
24: }
25: return 0;
26: }

```

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

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

```
$ ./prog
```

### 2.1 (3 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 — prog3

Affichage dans l'ordre :

prog2 « a : 33 », prog3 « a : 11 » (pas d'ordre sur ces 2 affichages)

prog1 « a : 10, e : 1 »

### 2.2 (1 point)

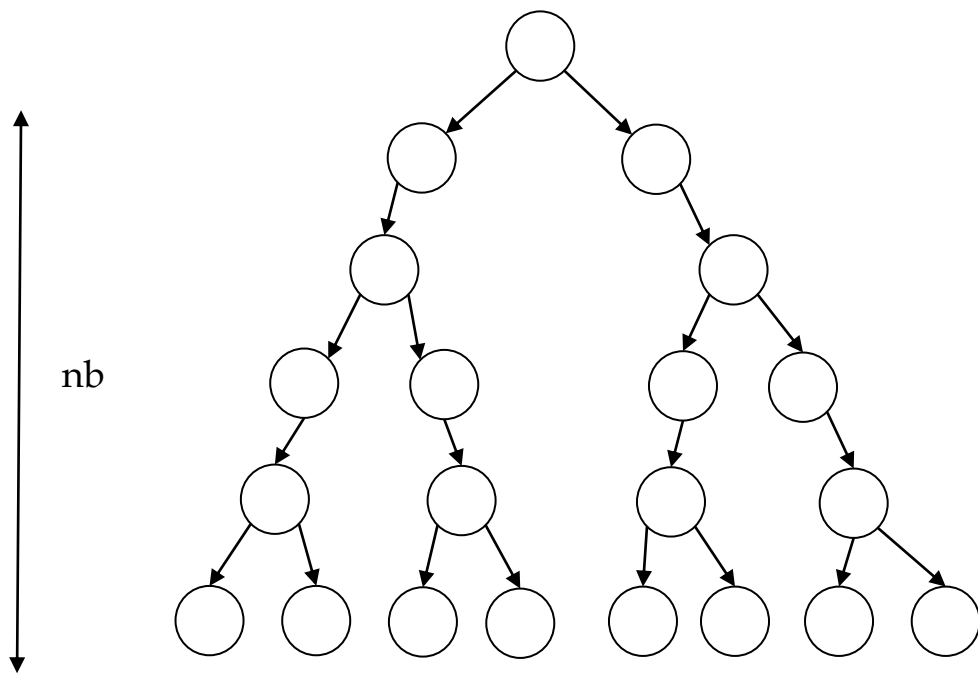
Est-ce que ce code crée un processus zombie ?

Si oui lequel et modifier le code pour éviter la création d'un zombie. Vous pouvez ajouter des lignes mais pas en supprimer (donnez que la ou les lignes à ajouter en précisant à quel endroit dans le code).

prog3 est zombie pendant 10 car son père (prog2) fait un sleep avant de se terminer  
4bis : wait(NULL) ; (avant la ligne 5)

## 3. PROGRAMMATION PROCESSUS (4 PTS)

On veut écrire une fonction void creer-arbre(int nb) qui l'arbre de processus suivant :



Cette fonction crée un arbre de hauteur *nb* en alternant la création d'un et deux fils.

### 3.1 (2,5 points)

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

```
void creer-arbre(int nb) {
    int i;
    for (i=0; i< nb; i++) {
        if (i%2) {
            if (fork()!=0)
                break;
        }
        else
            if (fork() != 0 && fork() != 0)
                break;
    }
}
```

### 3.2 (1,5 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-arbre(int nb) {
    int i;
    for (i=0; i< nb; i++) {
        if (i%2) {
            if (fork()!=0)
                break;
        }
    }
}
```

```

else
    if (fork() != 0 && fork() != 0)
        break;
}
if (i!=nb) {
    if (i%2)
        wait(NULL);
    else {
        wait(NULL);
        wait(NULL);
    }
}
}
}

```

#### 4. SYNCHRONISATION (5 PTS)

On considère les quatre tâches suivantes :

A	B	C	D
P(Mutex1);	P(Mutex1);	P(S1);	P(S1);
v1+=2 ; // a	v1*=2 ; // b	P(Mutex1);	P(Mutex2);
V(Mutex1) ;	V(Mutex1) ;	if (v1>4) {	if (v2 > 5) {
V(S1) ;	V(S1);	P(Mutex2)	P(Mutex1);
		v2 += v1; //c	v1 += v2; //d
		V(Mutex2);	V(Mutex1) ;
		}	}
		V(Mutex1);	V(Mutex2);

Les variables partagées  $v1$  et  $v2$  sont initialisées à 1 et protégées par les sémaphores Mutex1 et Mutex2 initialisés à 1. Le sémaphore  $S1$  est initialisé 0.

##### 4.1 (2,5 points)

A la fin des 4 processus, quels sont les ordres d'exécution possibles pour les lignes a, b, c et d ? (il est possible que certaines des ces lignes ne s'exécutent pas)

Quelles sont alors les couples de valeurs possibles pour les variables  $v1$  et  $v2$  à la fin de ces exécutions ?

```

abcd : v1 = 13, v2 = 7
abc  : v1 = 6,  v2 = 7
ba   : v1 = 4,  v2 = 1
ab   : v1=6 et v2=1.

```

#### 4.2 (1 point)

Montrez, via un exemple, que cette exécution peut entraîner un interblocage si initialement  $v2 = 6$ .

Exécution de A et B puis C s'exécute jusqu'au Mutex1 inclus et D s'exécute P(Mutex2) puis se bloque sur P(Mutex2) car  $v2 > 5$ , C alors va se bloquer sur P(Mutex1).

=> Interblocage entre C et D

#### 4.3 (1,5 point)

Proposez une modification du code pour qu'il n'y ait pas l'interblocage quelle que soit la valeur initiale de  $v2$ .

```
D
P(S1);
P(Mutex1);
P(Mutex2);
if (v2 > 5) {
    v1 += v2; //d
    V(Mutex1);
}
V(Mutex2);
```