

**1. ORDONNANCEMENT (7 PTS)**

Soit un système à temps partagé avec des quantums de 100 ms. Pour simplifier, il n'y a pas de tick. La stratégie d'ordonnancement est la suivante :

Pour chaque tâche on maintient une variable qrestant (quantum restant)

- A la création d'une tâche qrestant est mis à 0
- Lorsqu'une tâche perd le processeur, si elle a utilisé tout son quantum, alors qrestant est mis à 0 sinon qrestant est égal au temps non consommé.
- L'élection consiste à choisir la tâche ayant le qrestant maximum, en cas d'égalité on choisit la tâche la plus vieille (celle créée il y a plus longtemps).
- Pour la nouvelle tâche élue, on positionne qrestant à la valeur du quantum (100 ms) et on réinitialise le registre horloge à la valeur du quantum.

Soit le scénario suivant :

Tâches	Instant création	Durée d'exécution sur le processeur	Entrées/sorties
T1	0 ms	110ms	1 E/S de 40 ms après 20 ms d'exécution
T2	0 ms	120ms	1 E/S de 20 ms après 30 ms
T3	0 ms	60ms	aucune entrée sortie

On suppose que T1, T2 et T3 sont créées dans cet ordre. Les entrées/sorties se font sur le même disque (i.e., il y a **une seule unité d'échange**)

**1.1 (0,5 point)**

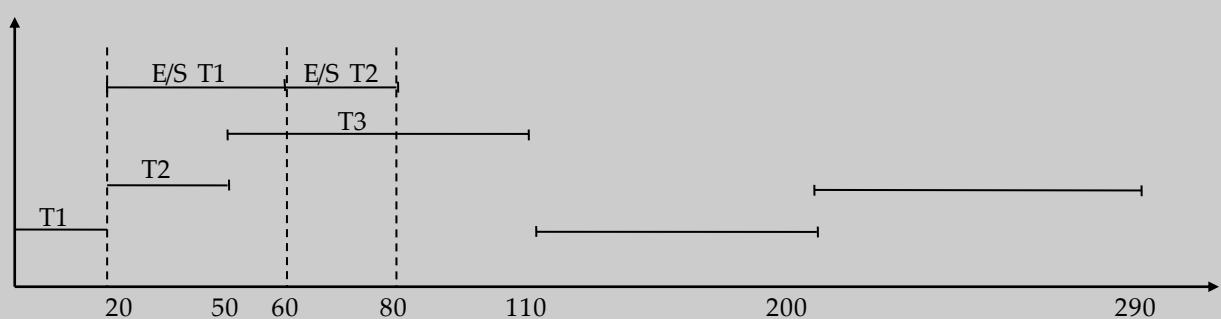
Lors d'une demande d'E/S comment le système connaît la valeur du quantum restant de la tâche élue ?

C'est la valeur contenue dans le registre horloge

### 1.2 (2,5 points)

On considère une stratégie **sans réquisition**.

- Faites un diagramme temporel (Gantt) de l'évolution des tâches
- Indiquez les temps de réponse des tâches.



$$T_R T1 = 200$$

$$T_R T2 = 290$$

$$T_R T3 = 110$$

### 1.3 (2,5 points)

On considère maintenant une stratégie **avec réquisition** possible en fin d'Entrée/Sortie.

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



$$T_R T1 = 170$$

T<sub>R</sub> T2 = 260  
T<sub>R</sub> T3 = 290  
1 seule réquisition à t=80

### 1.4 (1,5 points)

Montrez qu'avec cette stratégie avec réquisition une tâche peut être avantageée par rapport à une autre.

Proposez une modification de l'algorithme pour que le processeur soit équitablement réparti.

Comme on ré-alloue systématiquement un quantum entier, les processus faisant les E/S sont très avantageés.

Il suffit de modifier la stratégie en allouant uniquement le quantum restant

## 2. PROCESSUS (5 PTS)

Note : A la fin du sujet, vous trouverez en annexe des fonctions C qui pourront vous être utiles dans les exercices 2 et 3.

Soit le programme suivant :

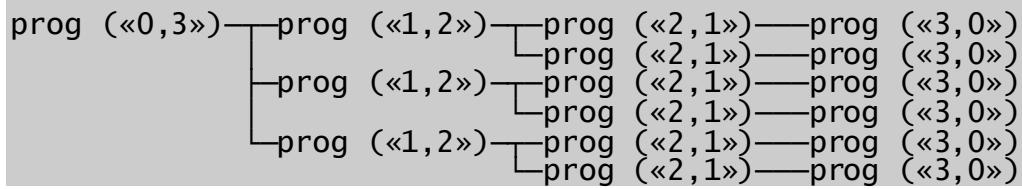
```
#define N 3

int main(int argc, char *argv[]) {
    int i,j,c;
    pid_t p;
    for (i=0; i< N; i++) {
        j=0;
        while (j<N-i && (p=fork()) !=0)
            j++;
        if (p>0)
            break;
    }
    printf("%d,%d\n",i,j);
    c = calculbinaire();
    return 0;
}
```

La fonction calculbinaire() renvoie soit 0 soit 1.

## 2.1 (3 points)

Donnez l'arbre des processus fait par ce programme ainsi que l'affichage fait par chacun des processus.



Nous voulons modifier le programme pour le père initiale (le main), affiche le nombre total de processus dont le résultat du calcul binaire a retourné 1.

## 2.2 (2 points)

Modifiez le programme en conséquence (sans utiliser de variables partagées).

Indiquez uniquement les modifications apportées au programme initial.

```
int main(int argc, char *argv[]) {  
    int i,j,c,cpt,k,cf;  
    pid_t p;  
    for (i=0; i< N; i++) {  
        j=0;  
        while (j<N-i && (p=fork()) !=0)  
            j++;  
        if (p>0)  
            break;  
    }  
    printf("%d,%d\n",i,j);  
    c = calculbinaire();  
    if (i==N)  
        return c;  
    cpt = c;  
    for (k=0; k< N-i; k++) {  
        wait(&cf);  
        cpt +=WEXITSTATUS(cf);  
    }  
    if (i==0) {  
        printf("%d\n",c);  
        return 0;  
    }  
    return cpt;  
}
```

### 3. EXECUTION (3 PTS)

A l'aide des fonctions fork et exec, on veut écrire un programme prog.c qui lance un exécutable avec des arguments et attend sa fin.

Depuis le shell le programme s'utilise de la manière suivante :

\$ ./prog ls f1 f2

Dans cet exemple, prog va créer un processus fils qui exécutera ls f1 f2

#### 3.1 (3 points)

Ecrivez le programme prog.c

```
int main(int argc, char *argv[]) {  
    if (fork()==0)  
        execvp(argv[1],&argv[1]);  
    wait(NULL);  
    return 0;  
}
```

### 4. SYNCHRONISATION (5 PTS)

On considère les quatre tâches suivantes :

A	B	C	D
P(S1);	P(S1);	P(S2);	P(S2);
v+=1 ; // a	v*=2 ; // b	P(S3)	P(S3)
V(S1);	V(S1);	P(S1);	P(S1);
V(S2);	V(S3);	v*=2 ; // c	v-=1 ; // d
V(S2);	V(S3);	V(S1);	V(S1);

La variable partagée  $v$  est initialisée à 1. Les sémaphores  $S1$ ,  $S2$  et  $S3$  sont initialisés respectivement à 1, 0 et 0. Les files d'attentes des sémaphores sont supposées être FIFO (premier arrivé, premier parti).

#### 4.1 (2 points)

Quels sont les ordres d'exécution possibles pour les lignes a, b, c et d? Quelles sont alors les valeurs possibles de la variable  $v$  ?

a,b,c,d : v=7

b,a,c,d : v=5

a,b,d,c : v=6

b,a,d,c v= 4

#### 4.2 (1 points)

Que se passe-t-il si C et D commencent à s'exécuter avant A ? Même question si C et D commencent à s'exécuter avant B.

- Avant A : C et D seront forcément bloqués sur le sémaphore S2 ( S2.cpt = -2) .
- Avant B: C et D seront bloqués soient sur le sémaphore S2 ( S2.cpt = -2) , s'il exécutent avant A aussi soient sur S3 ( S3.cpt = -2) .

#### 4.3 (2 points)

Proposez une modification afin de conserver l'accès exclusif à la variable *v* en garantissant que sa valeur finale après l'exécution des 4 tâches soit toujours supérieure ou égale à 6. Vous pouvez ajouter d'autres sémaphores. Si c'est le cas, n'oubliez pas d'indiquer leur initialisation.

B doit s'exécuter après A : On y ajoute S4 initialisé à 0

A	B	C	D
P(S1) ; v+=1 ; // a	P(S4); P(S1) ; v*=2 ; // b	P(S2) ; P(S3) P(S1) ; v*=2 ; // c	P(S2) ; P(S3) P(S1) ; v-=1 ; // d
V(S1) ; V(S4); V(S2) ; V(S2) ;	V(S1) ; V(S3); V(S3);	V(S1) ;	V(S1) ;

---

#### ANNEXE fonctions C qui peuvent être utiles

- pid\_t fork(void) ;
- int execlp(const char \*path, const char \*arg, ...);
- int execvp(const char \*path, char\*arg[]);
- pid\_t wait(int \*status);

la macro WEXITSTATUS(int status) retourne le code de sortie du fils