3I010- Système - Durée : 1 h45min

Mars 2018 - Les documents ne sont pas autorisés

Partiel - Barème indicatif

1. ORDONNANCEMENT (7 PTS)

Soit un système avec deux types de tâches :

Les tâches « system » fonctionnent en mode batch FIFO (sans quantum) et sont les plus prioritaires. Les autres tâches « user » suivent l'algorithme d'ordonnancement en temps partagé de type **tourniquet** avec **un quantum de 100 ms**.

Pour les tâches « user », il y a une file des tâches en mémoire. A la création, une nouvelle tâche « user » est insérée à l'état Prêt en queue de file. Si aucune tâche « system » n'est prête l'ordonnanceur **élit la tâche à l'état Prêt la plus en tête dans la file** des tâches « user ».

En cas d'entrées/sorties (E/S) la tâche élue passe à l'état « Bloqué » mais **reste en tête de sa file**. C'est uniquement à la fin de son quantum qu'une 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 ne l'avait complètement utilisée**.

Soit le scénario suivant :

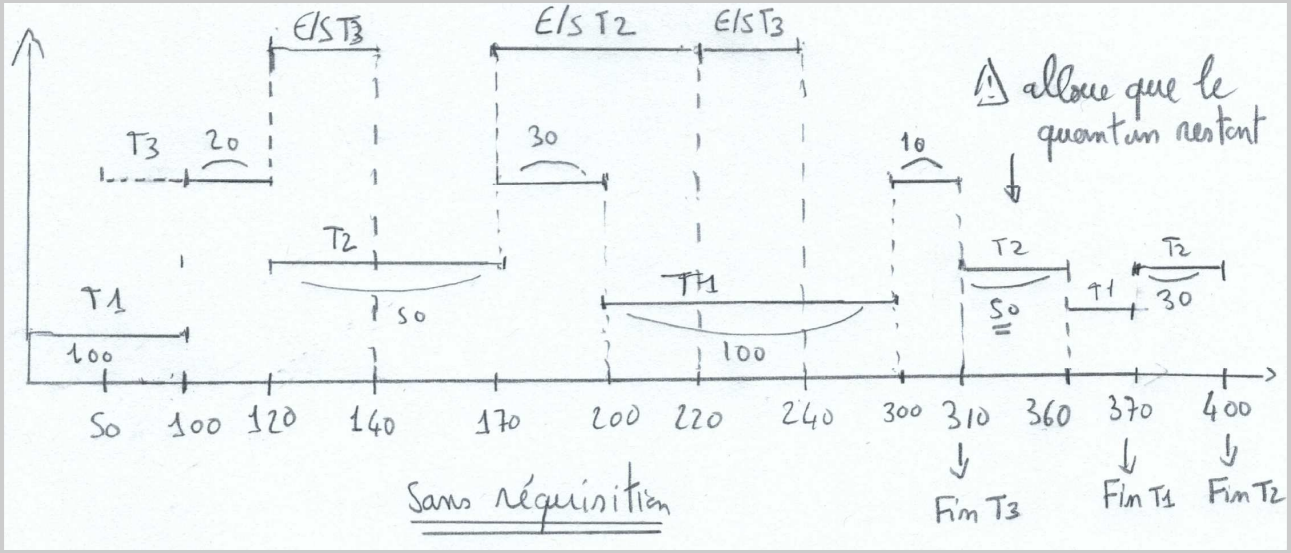
Tâches	Instant création	Types	Durée d'exécution sur le processeur	E/S
T1	0 ms	« user »	210ms	aucune E/S
T2	0 ms	« user »	130ms	une seule E/S de 50 ms après 50 ms d'exécution
T3	50 ms	« system »	60ms	une E/S de 20 ms après 20ms d'exécution, puis une seconde E/S après 50ms d'exécution depuis le début

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

1.1 (3,5 points)

On considère une stratégie **sans réquisition**.

- Faites un diagramme temporel (Gantt) de l'évolution des tâches
- Indiquez les temps de réponse des tâches.
- Indiquez à quels moments les tâches T1 et T2 ont eu leur fin de quantum



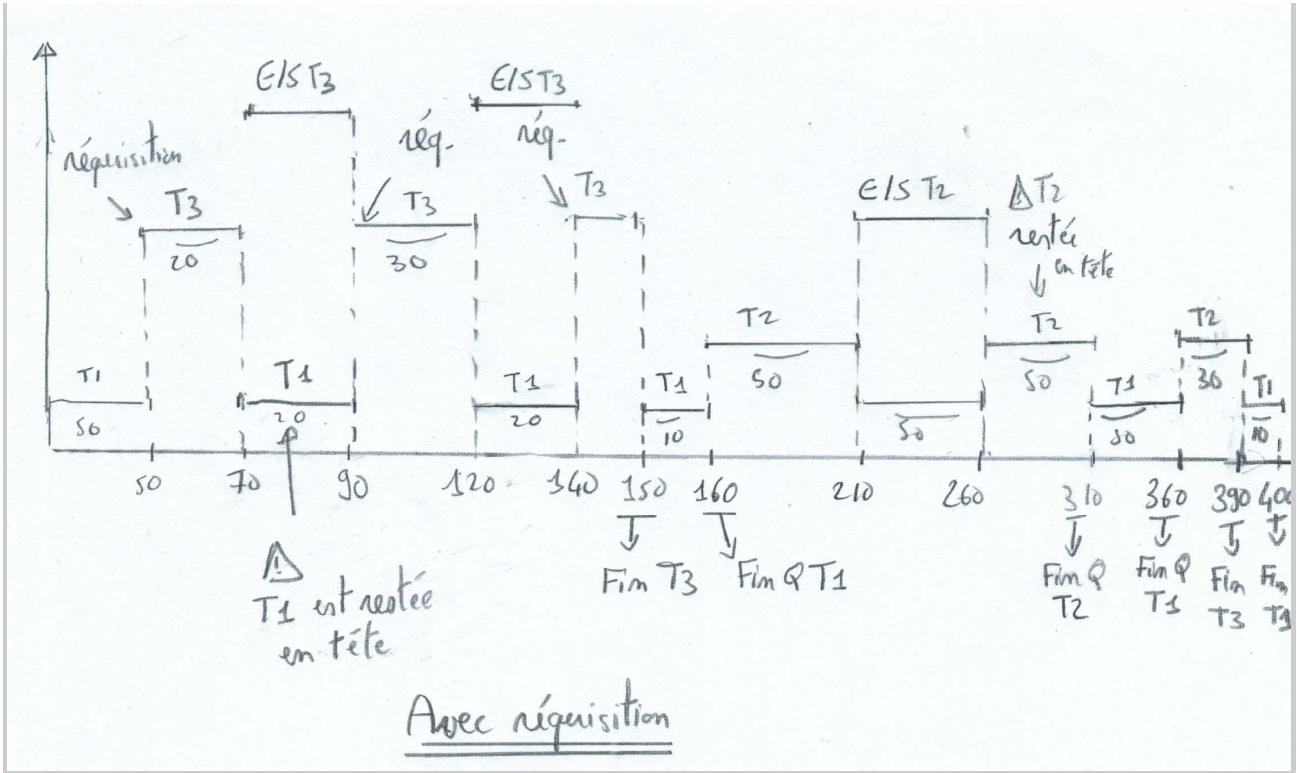
Temps réponse :
T1 = 370 ms
T2 = 400 ms
T3 = 310 -50 = 260ms

Fin quantum T1 : 100 et 300
Fin quantum T2 : 360

1.2 (3,5 points)

On considère maintenant une stratégie **avec réquisition** en cas de création de tâche plus prioritaire ou en fin d'Entrée/Sortie : la tâche créée ou qui finit une E/S peut être directement être élue. La tâche qui perd le processeur lors d'une réquisition garde sa place 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 ?
- Indiquez à quel moment les tâches T1 et T2 ont eu leur fin de quantum



4 réquisitons à : 50, 90, 140 et 260

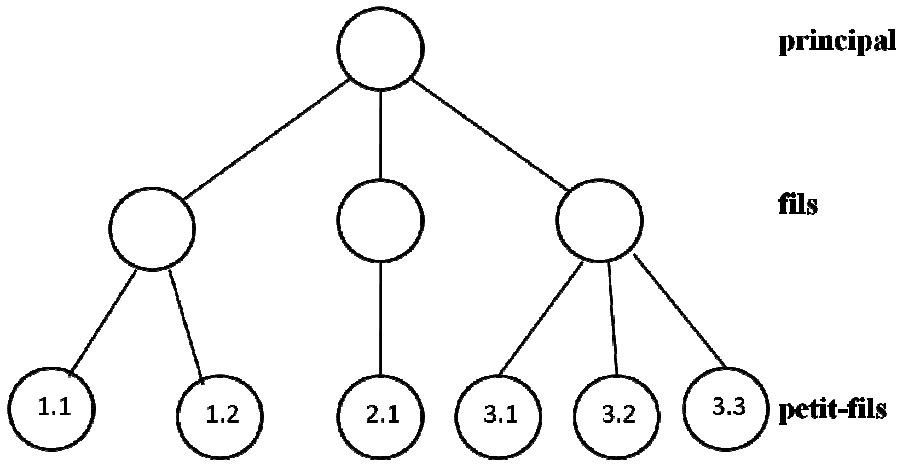
2. PROCESSUS (7 PTS)

Nous voulons créer le programme `arbre.c` qui reçoit comme arguments le nombre de processus fils que le processus principal doit créer ainsi que le nombre de fils que chacun des fils du processus principal doit créer. Ces derniers processus crées sont, par conséquent, petit-fils du processus principal.

En plus du pid, chaque processus fils et petit-fils est identifié par le numéro d'ordre de leur création par rapport à ses frères. Chaque processus **petit-fils** doit alors afficher l'identifiant de son père (i.e., le numéro d'ordre du père) et le sien. Par exemple, si l'usager exécute

> arbre 3 2 1 3

on aura l'arborescence et l'affichage suivants :



Pour simplifier nous considérons que l'usager appelle toujours le programme correctement et qu'il n'y a jamais d'erreur lors de l'appel à `fork ().`

2.1 (3 points)

Donner le code en C du programme arbre.c

```
int main (int argc, char* argv[ ]) {
  int i=1,j=1,N;
  pid_t p,q=-1;

  N=atoi (argv [1]);
  while (i<= N && (p=fork()) !=0)
    i++;

  if (p ==0){
    N=atoi (argv[i+1]);
    while (j<= N && (q=fork()) !=0)
      j++;
    if (q==0)
       printf ("\ %d.%d \n", i, j);
  }
  exit (0);
}
```

Nous voulons modifiez le programme `arbre.c` pour que le processus principal ne se termine qu'après la terminaison de son dernier fils créé (dans l'exemple le troisième fils). Le processus principal doit alors afficher le code de la commande `exit` de ce fils.

Note: dès que le processus principal prend connaissance de la terminaison de son dernier fils, il ne doit pas attendre la fin de ses autres fils s'ils ne sont pas encore terminés.

2.2 (2 points)

Modifiez le programme en conséquence.

```
int main (int argc, char* argv[ ]) {
  int i=1,j=1,N;
  pid_t p,q=-1;

  N=atoi (argv [1]);
  while (i<= N && (p=fork()) !=0)
    i++;

   if (p ==0){
    N=atoi (argv[i+1]);
    while (j<= N && (q=fork()) !=0)
      j++;
    if (q==0)
       printf ("\ %d.%d \n", i, j);
  }

  if (q == -1) {
    while (wait (&N) !=p);
  /* waitpid (p, &N, 0) */
    printf ("code exit %d \n", WEXITSTATUS (N) );
  }
  exit (0);
}
```

Nous voulons modifier encore une fois le programme `arbre.c` pour que les processus petit- fils utilisent pour l'affichage, au lieu de `printf`, la commande `echo` qui se trouve dans le répertoire */bin*.

2.3 (2 points)

Sans donner tout le code du programme, donner la déclaration des variables et les instructions afin que les petit-fils fassent l'affichage en utilisant la commande `echo`.

```
int main (int argc, char* argv[ ]) {
  int i=1,j=1,N;
  pid_t p,q=-1;
 char buff[20];

  N=atoi (argv [1]);
  while (i<= N && (p=fork()) !=0)
    i++;

   if (p ==0){
    N=atoi (argv[i+1]);
    while (j<= N && (q=fork()) !=0)
      j++;
     if (q==0) {
       sprintf (buff,"%d.%d \n", i,j);
        execl ("/bin/echo", "echo", buff, NULL);
```

```
    perror ("execl");
   }
  }
  exit (0);
}
```

3. SYNCHRONISATION (6 PTS)

Nous considérons les trois processus suivants exécutant dans un système à temps partagé:

A	B	C

(1) V(S2) ;
(2) P(S1) ;
(3) val=val+1 ; // a
(4) V(S1) ;
(5) V(S2);
	(5) P(S2) ;
	(6) P(S1)
	(7) val=val-2 ; // b
	(8) V(S1) ;
		(9) P(S2) ;
		(10) P(S1) ;
		(11) val=val*2 ; // c
		(12) V(S1) ;

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

3.1 (0,5 point)

Quel est le rôle du sémaphore S1?

S1 garantit l'exclusion mutuelle à val.

3.2 (1,5 point)

Supposons que C, B et A exécutent leur première instruction (9) (5) et (1) respectivement dans cet ordre. Donnez pour chacune de ces instructions la valeur du compteur et de la file du S2 ainsi que l'état des processus. Quelles sont les prochaines instructions qui peuvent être exécutées ? Justifier votre réponse.

(9) P(S2) : S2.cpt =-1 S2.file = {C} ; C élu ->bloqué ; A et B prêts
(5) P(S2) : S2.cpt =-2 S2.file = {C,B} ; B élu ->bloqué ; A prêt; C bloqué
(1) V(S2) S2.cpt =-1 S2.file = {B} ; A élu ->bloqué; C bloqué -> prêt
Les instructions (2) et (10) peuvent être exécutées mais non pas (6) étant donné que B est toujours bloqué.

3.3 (1,5 point)

Quels sont les ordres d'exécution possibles pour les lignes (3) (7) et (11) ? Quelles sont alors les valeurs possibles de `val` ?

- a,b,c : ((0+1)-2)*2 = -2
-a,c,b : ((0+1)*2)-2 = 0
-b,a,c : ((0-2)+1)*2 = -2

-c,a,b : ((0*2)+1)-2 = -1

3.4 (1,5 point)

Modifiez le programme pour garantir que la valeur de `val` ne soit jamais négative. Vous pouvez ajouter des nouveaux sémaphores et/ou déplacer les opérations sur S2, mais vous devez conserver le rôle du sémaphore `S1`.

Il faut avoir la suite -a,c,b = 0.
On ajoute le sémaphore S3, initialisé à 0.
- A: on enlève (1) V(S2)
- B: on remplace S2 par S3;
- C: on ajoute à la fin V(S3).

A	B	C

(2) P(S1) ;
(3) val=val+1 ; // a
(4) V(S1) ;
(5) V(S2);
	(5) P(S3) ;
	(6) P(S1)
	(7) val=val-2 ; // b
	(8) V(S1) ;
		(9) P(S2) ;
		(10) P(S1) ;
		(11) val=val*2 ; // c
		(12) V(S1) ;
		(12.1) V(S3)

3.5 (1 point)

Nous modifions le programme original en inversant les instructions (5) et (6) du processus A. Est-ce que cette modification pose un problème? Justifiez votre réponse

Oui. On aura un problème d'interblocage :
A : (1) V(S2) - S2.cpt =1
B : (6) P(S1) - S1.cpt =0
C: (9) P(S2) - S2.cpt =0
C: (10) P(S1) - S1.cpt=-1 ; C bloqué
B: (5) P(S2) - S2.cpt = -1 ; B bloqué
A: (2) P(S1) - S1.cpt = -2 : A bloqué