Exercice proposé dans le cadre de l'ED3

Il existe trois politiques d'ordonnancement dans le système Linux : la première est utilisée pour ordonnancer des processus ordinaires et les deux autres pour ordonnancer des processus temps réel. Afin de connaître la politique d'ordonnancement à utiliser pour un processus, chaque processus possède un type qui peut être égal à :

- `SCHED_FIFO` pour un processus temps réel non préemptible,
- `SCHED_RR` pour un processus temps réel préemptible,
- `SCHED_OTHER` pour un processus ordinaire (non temps réel).

Trois files différentes accueilleront les processus prêts appartenant aux trois types. Les processus de la file `SCHED_FIFO` sont plus prioritaires que ceux de la file `SCHED_RR` qui eux-mêmes sont plus prioritaires que ceux de la file `SCHED_OTHER`.
Quel que soit son type, un processus Linux possède une priorité et est inséré dans la file associée à son type dans l'ordre décroissant de sa priorité. A tout moment, le processus de type *x* le plus prioritaire se trouve en tête de la file du même type.

L'ordonnanceur gère la file `SCHED_FIFO` en utilisant la stratégie non préemptive. Il choisit d'élire le processus de type `SCHED_FIFO` le plus prioritaire pour s'exécuter. N'étant pas préemptible, ce processus s'exécute jusqu'à la fin sans libération du processeur, excepté dans les cas suivants :

1. un autre processus de type `SCHED_FIFO` plus prioritaire vient d'être inséré dans la file ; il est alors exécuté à la place du processus courant qui est inséré en fin de file.

2. le processus demande à faire une entrée-sortie;

3. le processus abandonne le processeur en appelant la primitive *`sched_yield()`*.

La file `SCHED_RR` est gérée par la technique du tourniquet avec une seule file d'attente. En effet, tout processus de type `SCHED_RR` est exécuté pour la durée d'un quantum de temps (égale à 60 ms). A l'expiration du quantum de temps, le processus le plus prioritaire parmi les processus de type `SCHED_FIFO` est choisi. Si la file d'attente associée est vide, le processus `SCHED_RR` le plus prioritaire est élu.
La priorité pour les processus des files `SCHED_FIFO` et `SCHED_RR` varie entre 1 et 99. Le plus prioritaire étant celui qui a la plus grande valeur.

La file SCHED_OTHER accueille les processus ordinaires c'est-à-dire non temps réel. Les processus temps réel étant plus prioritaires, les processus de type SCHED_OTHER ne pourront s'exécuter que si les files de type SCHED_FIFO et SCHED_RR sont vides.
La file SCHED_OTHER est elle aussi gérée avec la technique du tourniquet à une seule file d'attente. L'insertion des processus se fait en fonction de leurs priorités mais lorsque la priorité est identique (en général égale à 0), l'insertion se fait en FIFO.

La priorité d'un processus (quel que soit son type) se calcule à partir :

- d'une partie modifiable par l'utilisateur (à l'aide des primitives `nice` ou `setpriority`). Un processus utilisateur ne pourra que diminuer sa priorité; il ne pourra l'augmenter que s'il est un processus privilégié.

- et d'une partie fixée par le système et qui baisse lorsque le processus consomme un certain nombre de cycles d'horloge ; ceci pour permettre à ceux de moindre priorité de s'exécuter rapidement.

Soit un programme C lancé sous Linux par le super-utilisateur, et qui crée presque en même temps 4 processus p1, p2, p3 et p4.

En supposant que ces processus ne font pas d'entrée-sortie, que leurs priorités ne changent pas durant l'exécution et que leurs durées d'exécution sont celles décrites ci-dessous, déterminer les temps de réponses de chaque processus en appliquant les politiques d'ordonnancement adéquates.

Nom du processus	durée d'exécution	actions particulières réalisées
P0	60ms	à la fin de son calcul, il crée les 4 processus : p1, p2, p3 et p4 - met p1 dans la file SCHED_RR avec la priorité 2 - met p2 dans la file SCHED_FIFO avec la priorité 3 - met p3 dans la file SCHED_RR avec la priorité 4 - met p4 dans la file SCHED_FIFO avec la priorité 4
	30ms	calcul + attente de la fin des processus
P1	100ms	calcul
P2	90ms	calcul

	30ms	exécute `usleep`(30)
	10ms	calcul
P3	110ms	calcul + exécute `sched_setscheduler()` à la fin du calcul pour se mettre dans la file SCHED_OTHER avec la priorité 1
	20ms	calcul
P4	70ms	calcul + exécute `sched_yield()` en fin de calcul pour libérer le processeur
	50ms	

Solution

Instants	processus actif (durée d'exécution)	contenus des files d'ordonnancement - processus(priorité)	actions particulières
initialement	P0	SCHED_FIFO= {} SCHED_RR={} SCHED_OTHER={P0(0)}	Initialement, le programme correspond à un seul processus P0 de priorité 0 inséré dans la file la moins prioritaire
t0	P1(60ms)	SCHED_FIFO= {} SCHED_RR={} SCHED_OTHER={}	création des 4 processus à la fin de cette tranche de calcul
t1	P4(70ms)	SCHED_FIFO= {P2(3)} SCHED_RR={P3(4), P1(2)} SCHED_OTHER={P0(0)}	P4 libère le processeur après 70ms de calcul
t2	P2(90ms)	SCHED_FIFO= {P4(4)} SCHED_RR={ P3(4), P1(2)} SCHED_OTHER={ P0(0)}	
t3	P4(30ms)	SCHED_FIFO= {} SCHED_RR={ P3(4), P1(2)} SCHED_OTHER={ P0(0)}	P2 est à l'état bloqué pour une durée de 30ms.
t4	P4(20ms)	SCHED_FIFO= { P2(3)} SCHED_RR={ P3(4), P1(2)} SCHED_OTHER={ P0(0)}	P2 a débloqué, il passe dans la file SCHED_FIFO
t5	P2(10ms)	SCHED_FIFO= {} SCHED_RR={ P3(4), P1(2)} SCHED_OTHER={ P0(0)}	fin d'éxécution de P4, P4 passe à l'état zombi.
t6	P3(60ms)	SCHED_FIFO= {} SCHED_RR={P1(2)} SCHED_OTHER={ P0(0)}	fin d'éxécution de P2, P2 passe à l'état zombi.
t7	P3(50ms)	SCHED_FIFO= {} SCHED_RR={P1(2)} SCHED_OTHER={ P0(0)}	En fin d'exécution de cette tranche de calcul, P3 change de politique d'ordonnancement et se met dans la file SCHED_OTHER
t8	P1(60ms)	SCHED_FIFO= {}	à l'expiration du quantum de

		SCHED_RR={} SCHED_OTHER = { P0 (0), P3 (1)}	temps, on vérifie s'il y a un processus plus prioritaire que P1 (dans ce cas, il n'y en a pas).
t9	P1(40ms)	SCHED_FIFO= {} SCHED_RR={} SCHED_OTHER = { P0 (0), P3 (1)}	
t10	P3(20ms)	SCHED_FIFO= {} SCHED_RR={} SCHED_OTHER = { P0 (0)}	P1 termine son exécution, il passe à l'état zombi.
t11	P0(30ms)	SCHED_FIFO= {} SCHED_RR={} SCHED_OTHER = {}	P3 termine son exécution, il passe à l'état zombi.
t12		SCHED_FIFO= {} SCHED_RR={} SCHED_OTHER = {}	fin d'exécution de P0 avec prise en compte de la terminaison des processus fils

Le temps de réponse est l'intervalle de temps qui sépare la fin d'exécution d'un processus de l'instant de sa soumission dans le système.
P0 a été soumis à t0 et s'est terminé à t12; d'où un temps de réponse égal à :
60+70+90+30+20+10+60+50+60+40+20+30 =

Les autres processus sont créés à t1 par P0, d'où :
temps de réponse de P1 = 70+90+30+20+10+60+50+60+40=
temps de réponse de P2 = 70+90+30+20+10=
temps de réponse de P3 = 70+90+30+20+10+60+50+60+40+20=
temps de réponse de P4 = 70+90+30+20=

ANNEXE

A titre d'information, voici la syntaxe des primitives système auxquelles on a fait référence dans cet exercice.

- ```
  #include <sched.h>
  int sched_setscheduler(pid_t pid, int politique,
                         const struct sched_param *param);
  ```

`pid`: est l'identifiant du processus pour lequel on veut changer de politique d'ordonnancement

`politique` = {`SCHED_FIFO, SCHED_RR, SCHED_OTHER`}

`param` : paramètres d'ordonnancement, contient la priorité du processus.

La structure `sched_param` définie dans `<sched.h>` contient un seul champ : `int sched_priority`.

- ```
  #include <sched.h>
  int  sched_yield(void);
  ```

permet au processus courant de libérer volontairement le processeur et d'être placé en fin de file. Le processeur exécute un autre processus s'il y en a.

- ```
  #include <sys/resource.h>
  int setpriority(int p, int q, int prio);
  ```

modifie la priorité d'un processus, d'un groupe de processus ou de tous les processus de l'utilisateur.

P	Q
PRIO_PROCESS	getpid()
PRIO_PGRP	getgpid()
PRIO_USER	getuid()

`prio` est la nouvelle priorité.

Exemple de programme :

```
#include <sched.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/resource.h>

void proc_fils (int pid) {
int i;

      sleep(rand()%3); // retarder l'execution de chaque processus fils
                  // le temps de changer sa priorite
      printf("\nle processus %d commence a s'executer \n",pid);
```
```

```
      for (i=0 ; i<10000000 ; i++) {
            if (i==500000)  {
                  printf("\nle processus %d vient de liberer le
processeur\n ", pid) ;
                  sched_yield() ;
                  printf("\nle processus %d reprend son execution apres un
sched_yield()\n ", pid) ;
            }
}
exit(0);
}

void afficher_ordo_priorite(int pid) {
  int ordo;
  struct sched_param  p1;

  printf ("\n Processus %d  : (ordo =", pid);
  ordo= sched_getscheduler(pid); // retourne la file d'ordo du processus
  switch (ordo) {
      case SCHED_FIFO :  printf(" SCHED_FIFO, ") ; break ;
      case SCHED_RR :    printf(" SCHED_RR, ") ; break ;
      case SCHED_OTHER : printf("SCHED_OTHER, ") ;
  }
  sched_getparam(pid,&p1);
  printf ("priorite = %d )\n", p1.sched_priority);
}


void changer_ordo_priorite(int pid, int file, int prio) {
struct sched_param  p1;
int res;

      /* changer la file d'ordo et la priorite d'un processus */
      p1.sched_priority=prio;
      if ((res=sched_setscheduler(pid, file, &p1))!=-1) {
            afficher_ordo_priorite(pid);
      }
}

// M A I N
int main (void){
int pidf1, pidf2;

struct timespec intervalle;

printf("Les priorites minimale et maximale de la politique SCHED_FIFO sont
: %d et %d\n", sched_get_priority_min(SCHED_FIFO),
sched_get_priority_max(SCHED_FIFO));
printf("Les priorites minimale et maximale de la politique SCHED_RR sont :
%d et %d\n", sched_get_priority_min(SCHED_RR),
sched_get_priority_max(SCHED_RR));
printf("Les priorites minimale et maximale de la politique SCHED_OTHER sont
: %d  et %d\n",
sched_get_priority_min(SCHED_OTHER),sched_get_priority_max(SCHED_OTHER));

srand(getpid()); // initialisation du generateur de nombres aleatoires

if ((pidf1=fork())==0) { // processus fils 1
      proc_fils(getpid());
    }
```

```
if ((pidf2=fork())==0) { // processus fils 2
      proc_fils(getpid());
    }

if (pidf1==-1 || pidf2==-1) { printf("erreur\n"); exit(1); }

/* processus pere*/

printf("\n ==>Parametres initiaux du processus pere et ses fils: \n");
afficher_ordo_priorite(getpid()) ;
afficher_ordo_priorite(pidf1);
afficher_ordo_priorite(pidf2);

printf("\n\n ==>Nouveaux parametres des processus fils: \n");
changer_ordo_priorite(pidf2, SCHED_FIFO,3);
changer_ordo_priorite(pidf1, SCHED_RR,5);

sched_rr_get_interval(pidf1,&intervalle);
printf("\nquantum de temps de la file SCHED_RR :  %d milli-sec\n",
intervalle.tv_nsec/1000000);

while ( waitpid(-1,NULL, 0) > 0); /* le pere attend la fin de tous ses fils
*/
      printf("\nTerminaison de tous les processus\n");
return 0;
}
```

Exécution

```
$cc scheduling.c -o sched
$./sched
Les priorites minimale et maximale de la politique SCHED_FIFO sont : 1 et
99
Les priorites minimale et maximale de la politique SCHED_RR sont : 1 et 99
Les priorites minimale et maximale de la politique SCHED_OTHER sont : 0  et
0

 ==>Parametres initiaux du processus pere et ses fils:

 Processus 1711  : (ordo =SCHED_OTHER, priorite = 0 )

 Processus 1712  : (ordo =SCHED_OTHER, priorite = 0 )

 Processus 1713  : (ordo =SCHED_OTHER, priorite = 0 )


 ==>Nouveaux parametres des processus fils:

 Processus 1713  : (ordo = SCHED_FIFO, priorite = 3 )

 Processus 1712  : (ordo = SCHED_RR, priorite = 5 )

quantum de temps de la file SCHED_RR :  150 milli-sec

le processus 1713 commence a s'executer

le processus 1713 vient de liberer le processeur

le processus 1713 reprend son execution apres un sched_yield()

le processus 1712 commence a s'executer
```

```
le processus 1712 vient de liberer le processeur

le processus 1712 reprend son execution apres un sched_yield()

Terminaison de tous les processus
$
```