Sommaire

Séquence 1

1) Tester le programme **scheduler.1.c** fourni qui permet d'obtenir des informations sur la politique d'ordonnancement du processus qui s'exécute.

```
int main(void)
{
   printf("debut\n");

   printf("\t-> informations sur les priorités des ordonnancements
connus\n");
   affiche_infopriorite();

   printf("\t-> informations l'ordonnancement de ce processus\n");
   affiche_ordonnancement();

   printf("fin\n");
   exit(0);
}
```

Vous pouvez consulter ces pages **man** :

```
$ man sched_getscheduler
..
$ man getpriority
..
```

2) Indiquer alors les critères (Faible ou Haute) pour chacune de ces priorités :

Priorités statiques :

- valeur 1 →
- valeur 99 →

Priorités dynamiques :

- valeur 19 →
- valeur -20 →

3) Donner la commande qui permet de modifier la priorité dynamique d'un processus.

Séquence 2

4) Tester le programme **scheduler.2.c** fourni qui permet de modifier la politique d'ordonnancement du processus qui s'exécute.

```
int main(void)
{
   printf("debut\n");

   printf("\t-> informations sur les priorités des ordonnancements
connus\n");
   affiche_infopriorite();

   printf("\t-> informations l'ordonnancement de ce processus\n");
   affiche_ordonnancement();

   printf("\t-> modification de l'ordonnancement et des priorités de
ce processus\n");

   //modifie_ordonnancement(SCHED_OTHER, DYN_PRIORITE_FAIBLE, 0);
   //modifie_ordonnancement(SCHED_OTHER, DYN_PRIORITE_HAUTE, 0);
   //modifie_ordonnancement(SCHED_RR, 0, STAT_PRIORITE_FAIBLE);
   //modifie_ordonnancement(SCHED_RR, 0, STAT_PRIORITE_HAUTE);
   //modifie_ordonnancement(SCHED_FIFO, 0, STAT_PRIORITE_FAIBLE);
   //modifie_ordonnancement(SCHED_FIFO, 0, STAT_PRIORITE_HAUTE);

   affiche_ordonnancement();

   printf("fin\n");
   exit(0);
}
```

5) Quel est le problème qui apparaît lorsqu'on veut modifier la politique d'ordonnancement du processus sous sa session d'utilisateur ? Que faut-il faire alors ?

6) Tester maintenant différentes politiques et priorités.

7) Que se passe-t-il lorsqu'on essaye d'appliquer la modification suivante ? Pourquoi ?

```
modifie_ordonnancement(SCHED_OTHER, 10, 10);
```

Séquence 3

On va reprendre un exemple simple à base de deux processus : un affichera des étoiles ('*') et l'autre des dièses ('#').

Le premier programme à utiliser se nomme **setscheduler.1.c** et s'utilise de la manière suivante :

On passe en paramètre la priorité dynamique (ici 10) qui sera fixée pour le processus père

```
$ ./setscheduler.1 10
```

8) Tester le programme **setscheduler.1.c** fourni et indiquer les politiques et priorités des processus fils. Réessayer en modifiant celle du père dans la fonction main() et en observant celles des fils. Conclure.

9) Si vous possédez un processeur multi-core, indiquer pour chacun des processus du programme précédent le numéro de CPU utilisé pour s'exécuter.

- Processus père : CPU =
- Processus fils (étoile) : CPU =
- Processus fils (dièse) : CPU =

Par exemple :

```
$ ./setscheduler.1 10
PID = 32433 - PPID = 21626 - UID = 500 - EUID = 500 - CPU = 3
    Ordonnancement SCHED_OTHER : [32433]
          Priorité statique  = 0 (sched_priority) [32433]
          priority dynamique = 10 (getpriority) [32433]

Je suis le fils <etoile> : PID = 32434
PID = 32434 - PPID = 32433 - UID = 500 - EUID = 500 - CPU = 0
    Ordonnancement SCHED_OTHER : [32434]
          Priorité statique  = 0 (sched_priority) [32434]
Je suis le fils <diese> : PID = 32435
          priority dynamique = 10 (getpriority) [32434]

PID = 32435 - PPID = 32433 - UID = 500 - EUID = 500 - CPU = 1
    Ordonnancement SCHED_OTHER : [32435]
          Priorité statique  = 0 (sched_priority) [32435]
          priority dynamique = 10 (getpriority) [32435]

*****************#*#*# . . .
. . .
```

On désire maintenant observer l'exécution concurrente des deux processus.

Pour cela, il faut tout d'abord restreindre notre processeur multi-core à un seul coeur. On utilisera la fonction fournie :

```
// rend le système mono-cpu
modifie_cpu(0);
```

10) Tester le programme **setscheduler.2.c** fourni qui rend le processeur mono-cpu. Que se passe-t-il pour l'affichage des * et des # ? Conclure.

```
$ ./setscheduler.2 19
```

Un processus peut volontairement libérer le processeur sans se bloquer en appelant **sched_yield()**. Le processus sera alors déplacé à la fin de la *file* des processus prêts de sa priorité, et un autre processus sera exécuté.

Remarque : si le processus appelant est le seul processus prêt de sa priorité, il continuera son exécution après un appel à sched_ yield().

11) Tester le programme **setscheduler.3.c** fourni qui libère le processeur après l'affichage d'un caractère. Que se passe-t-il pour l'affichage des * et des # ? Conclure.

```
$ ./setscheduler.3 19
```

On désire maintenant appliquer des priorités différentes suivant les politiques d'ordonnancement choisies sur plusieurs processus pour observer leur exécution concurrente. Pour cela, vous utiliserez le programme **setscheduler.4.c**.

12) Tester le programme **setscheduler.4.c** fourni et observer l'exécution des processus fils en leur indiquant des priorités différentes pour une même politique. Conclure.

Bilan : pour notre « application temps-réel », le processus étoile ('*') représente une tâche qui doit toujours s'exécuter avant celle représentée par le processus dièse ('#'). On ne désire pas appliquer de quantum limité à nos deux processus.

13) Proposer les priorités à fixer à nos deux processus fils ainsi que la politique d'ordonnancement choisie.

```
void etoile()
{
. . .
   modifie_ordonnancement(           ,    ,   );
. . .
}

void diese()
{
. . .
   modifie_ordonnancement(           ,    ,   );
. . .
}
```