

Temps-réel natif de Linux

Christophe Blaess

christophe.blaess@logilin.fr

<https://www.blaess.fr/christophe/>
twitter: @chrisblaess



Ingénierie et formations sur Linux et les logiciels libres
<https://www.logilin.fr>

Ordonnancements Fifo et Round-Robin.....	3
Introduction.....	3
Priorités temps-réel.....	4
Passage en temps-réel.....	5
Travaux pratiques : ordonnancement FIFO de priorité maximale.....	7
Limitations du temps réel Vanilla.....	9
Première limite : interruptions monolithiques.....	9
Traitements dus à un « ping ».....	10
Amélioration avec les <i>Threaded Interrupts</i>	11
Deuxième limite : réveil d'une tâche temps réel.....	12
Travaux pratiques : mesure de latence d'interruptions.....	14
Amélioration avec la préemptibilité du noyau.....	15
Problèmes temps-réel classiques.....	16
Lancements parallèles en Round-Robin.....	16
Inversion de priorités.....	18
Travaux pratiques : inversions de priorités.....	19
Travaux pratiques : héritage de priorité des mutex.....	21
Reprise de mutex.....	22

Ce support de formation est distribué sous licence **Creative Commons 4.0**



(Attribution - Partage dans les mêmes conditions).

Vous êtes libres de copier et partager ce document, en mentionnant son origine. Si vous l'intégrez dans un contenu plus vaste, ce dernier devra être distribué avec les mêmes droits.

Ce cours a été rédigé en utilisant des logiciels libres sur système d'exploitation Linux :

- *LibreOffice Writer* pour le support et la mise en page
- *LibreOffice Draw* pour les dessins vectoriels
- *Gimp* pour les images bitmap

Temps réel Linux et Xenomai

v. 5.4

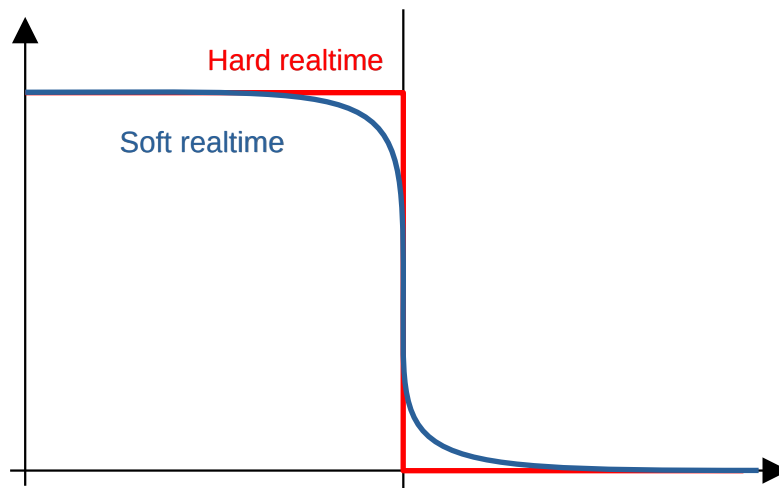
<https://www.blaess.fr/christophe/>

<https://www.logilin.fr>

Ordonnancements Fifo et Round-Robin

Introduction

Linux permet d'obtenir de configurer des tâches en leur donnant des priorités temps-réel. Il s'agit de temps-réel souple (*soft realtime*) à ne pas confondre avec le temps-réel strict (*hard realtime*).



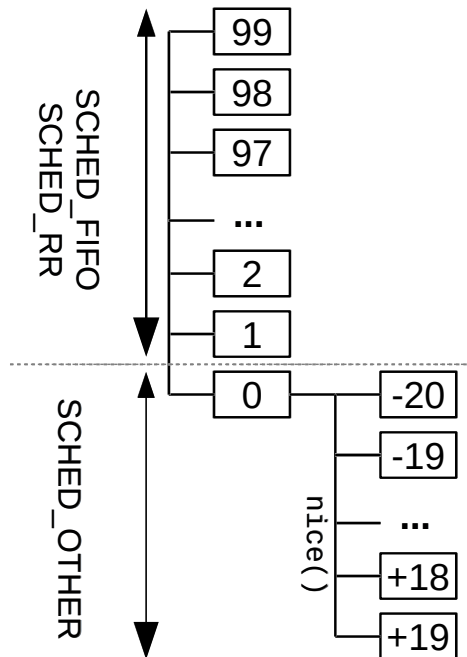
Dans un système temps-réel souple, on impose des contraintes d'exécutions **moyennes**. Avec le temps-réel strict on impose des contraintes dans le **pire** des cas.

On peut considérer qu'un système d'exploitation peut répondre aux exigences du temps-réel souple s'il peut garantir que le déroulement d'une tâche plus prioritaire ne sera jamais perturbé par la présence d'une tâche moins prioritaire.

Le noyau Linux n'offre pas de garantie de fonctionnement en temps-réel strict car certaines opérations complexes (filtrage réseau, routage, interprétation des systèmes de fichiers, gestion de la mémoire virtuelle, etc.) sont réalisées directement dans le noyau en ayant précedence sur l'activité de l'espace utilisateur.

Pour obtenir des performances de temps-réel strict avec Linux, il faut employer des extensions comme Xenomai ou RTAI.

Priorités temps-réel



```
int sched_get_priority_min(int scheduling);  
int sched_get_priority_max(int scheduling);
```

Il existe trois type d'ordonnancement que l'on fixe tâche par tâche :

- **SCHED_FIFO** : Une tâche ne peut être préemptée que par une autre tâche temps-réel de priorité strictement supérieure qui vient de devenir *Prête* ;
- **SCHED_RR** : Une tâche est toujours préemptée par une autre tâche temps-réel de priorité strictement supérieure. En outre au bout d'un certain temps d'exécution, elle peut être préemptée par une autre tâche temps-réel de même priorité ;
- **SCHED_OTHER** : non précisé par la norme Posix.1b, sous Linux il s'agit de l'ordonnancement temps-partagé vu précédemment.

Les ordonnancements **SCHED_FIFO** et **SCHED_RR** sont considérés comme temps-réel, et nécessitent les privilèges *root*. Les priorités temps-réel qu'ils utilisent sont différentes des priorités temps-partagé fixées par `setpriority()` ou `nice()`.

L'échelle des priorités temps-réel peut varier suivant les systèmes et suivant le type d'ordonnancement. Utiliser `sched_get_priority_min()` et `sched_get_priority_max()` pour déterminer l'intervalle utilisable de manière portable.

Passage en temps-réel

Ordonnancement d'un processus :

```
int sched_setscheduler(pid_t pid, int sched,
                      const struct sched_param *param);

int sched_getscheduler(pid_t pid);

int sched_setparam(pid_t pid, struct sched_param *param);
int sched_getparam(pid_t pid, struct sched_param *param);

struct sched_param {
    int sched_priority;
    ...
}
```

Ordonnancement d'un thread :

```
int pthread_setschedparam(pthread_t thr, int sched,
                          struct sched_param *param);
int pthread_getschedparam(pthread_t thr, int *sched,
                          struct sched_param *param);

int pthread_setschedprio(pthread_t thread, int prio);
```

À vous

Examinez le code-source de l'**exemple-III-01.c** puis compilez-le

Lancez-en un exemplaire en passant en argument un niveau de priorité temps réel. Le processus attend une seconde puis fait un comptage pendant trois secondes et affiche son résultat.

Le niveau de priorité influe-t-il sur la valeur obtenue ?

Lancez deux exemplaires avec des valeurs de priorité différentes (sur le même processeur). Quels sont les résultats ?

Lancez deux puis trois exemplaires au même niveau de priorité (l'ordonnancement est en *Round-Robin*). Quels sont les valeurs obtenues ?

La commande shell **chrt** permet de lancer une commande en temps-réel ou de modifier l'ordonnancement d'un processus existant.

```
# chrt -f 50 ./command    lance la commande en Fifo priorité 50
# chrt -r 40 ./command    lance la commande en Round-Robin priorité 40
# chrt -pf 30 12345        passe le processus PID 12345 en Fifo prio. 30
# chrt -pr 20 6789         passe le processus 6789 en Round-Robin prio 20
# chrt -po 0 2468          passe le processus 2468 en temps partagé.
```

On peut fixer l'ordonnancement d'un thread avant sa création en remplissant l'objet `pthread_attr_t`:

```
int pthread_attr_setschedparam(pthread_attr_t *attr,
                               int sched,
                               struct sched_param *param);
```

Il faut également fixer l'héritage d'ordonnancement à `PTHREAD_EXPLICIT_SCHED` :

```
int pthread_attr_setinheritsched(pthread_attr_t *attr,
                                  int inherit);
```

Céder volontairement le processeur à une autre tâche de même priorité :

```
int sched_yield(void);
```

Travaux pratiques : ordonnancement FIFO de priorité maximale

Examinez le programme **exemple-III-02.c**. Compilez-le.

Le processus lance un thread temps-réel *Fifo* 99 sur chaque CPU actif. Chaque thread exécute une boucle active pendant quelques secondes.

Lancez le programme. Vérifiez si votre système est totalement gelé pendant ces quelques secondes.

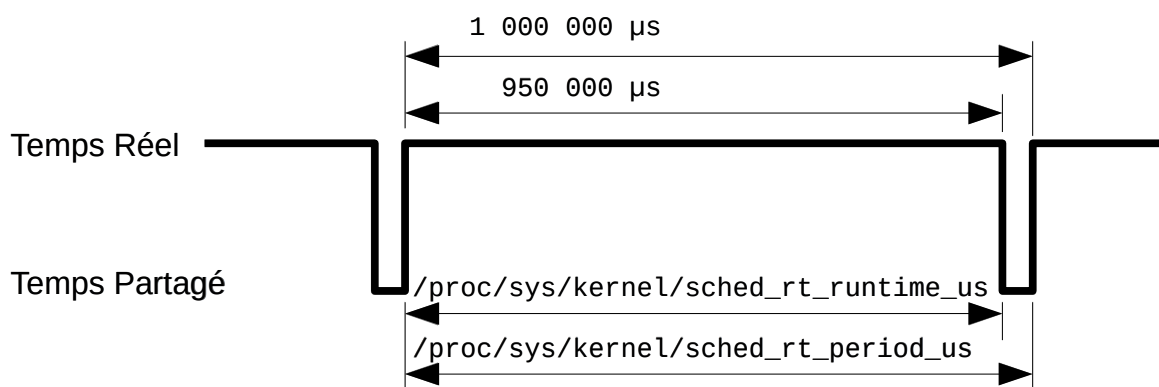
Si ce n'est pas le cas, relancez le programme après avoir exécuté :

```
$ sudo sysctl kernel.sched_rt_runtime_us=-1
```

Comment se comporte le système cette fois ?

Garde-fou temps réel

Si une ou plusieurs tâches temps-réel travaillent plus de 950 ms d'affilée, elles seront préemptées pendant 50 ms pour laisser passer les tâches temps partagé.



Cette mesure de précaution pour se protéger des processus bouclant involontairement nuit à la qualité du temps réel. On peut la désactiver dans un script d'initialisation ainsi :

```
echo -1 > /proc/sys/kernel/sched_rt_runtime_us
ou
sysctl kernel.sched_rt_runtime_us=-1
```


Limitations du temps réel Vanilla

Première limite : interruptions monolithiques

Travaux pratiques : influences des interruptions sur le temps réel

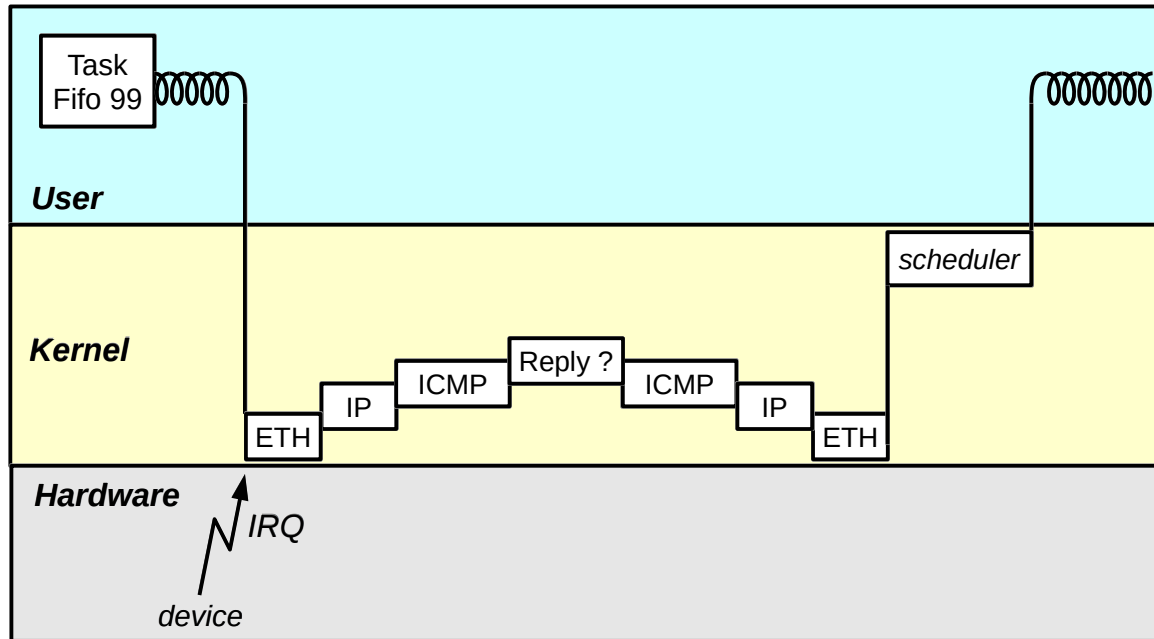
Exécutez le programme **exemple-III-02.c** comme nous l'avons fait précédemment.

Avant son démarrage, lancez (sur une autre machine) un « *ping* » vers votre poste.

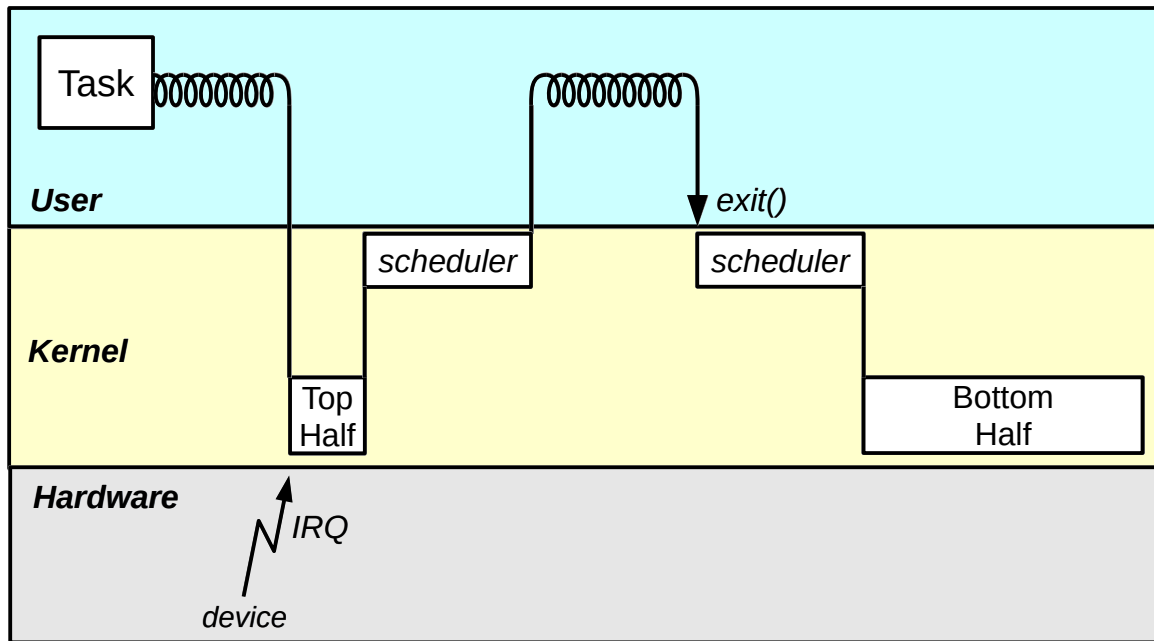
Le processus gèle tout votre système pendant quinze secondes.

Comment se comporte le *ping* ?

Traitements dus à un « ping »



Amélioration avec les *Threaded Interrupts*



Dans un noyau Linux Vanilla, rares sont les traitements en *threaded interrupts* (priorité à la performance)

Avec un traitement d'interruption classique, dès l'arrivée d'une IRQ, le traitement en cours est interrompu quelle que soit sa priorité pour laisser place à la routine de service associée.

Avec un traitement sous forme de *threaded interrupt*, seule une courte portion de la routine de service (sa « *top half* ») est exécutée immédiatement. Le reste du traitement est réalisé dans un thread du noyau dont la priorité est configurable. Son exécution peut être sensiblement différée si une tâche plus prioritaire est active.

Le remplacement de la plupart des traitements classiques par des *threaded interrupts* est l'un des points clés de l'action du patch **PREEMPT_RT** dont nous parlerons plus loin.

Deuxième limite : réveil d'une tâche temps réel

Timers temps-réel

Lire l'heure :

```
int gettimeofday(struct timeval *tv, struct timezone *tz);

struct timeval {
    time_t tv_sec; // seconds since 1970/01/01
    long tv_usec; // microseconds
}

int clock_gettime(CLOCK_REALTIME, struct timespec *ts);

struct timespec {
    time_t tv_sec; // seconds since 1970/01/01
    long tv_nsec; // nanoseconds
}
```

Programmer un timer (méthode Unix) :

```
int setitimer (int type, struct itimerval *config,
               struct itimerval *previous);

struct itimerval {
    struct timeval it_value;
    struct timeval it_interval;
}
```

Si le *type* précisé dans `setitimer()` est **ITIMER_REAL**, le signal reçu à expiration du timer est **SIGALRM**.

À vous

Exécutez à quelques reprises le programme **exemple-III-03.c** qui boucle cinquante fois autour de l'appel-système `clock_gettime()` puis affiche en sortie ses résultats de mesures.

Quelle précision pouvez-vous attendre de `clock_gettime()` sur votre système ?

Programmer un timer (méthode Posix) :

```
int timer_create(CLOCK_REALTIME, struct sigevent *event,
                 timer_t *tmr);

    struct sigevent {
        int sigev_notify; // SIGEV_SIGNAL
        int sigev_signo;  // SIGALRM
        ...
    }

int timer_settime(timer_t tmr, int absolute
                  const struct itimerspec *spec,
                  struct itimerspec *previous);

    struct itimerspec {
        struct timespec it_value;
        struct timespec it_interval;
    }

int timer_getoverrun (timer_t tmr);
```

Travaux pratiques : mesure de latence d'interruptions

Le programme **exemple-III-04.c** programme un timer dont on indique la période en microsecondes sur la ligne de commande.

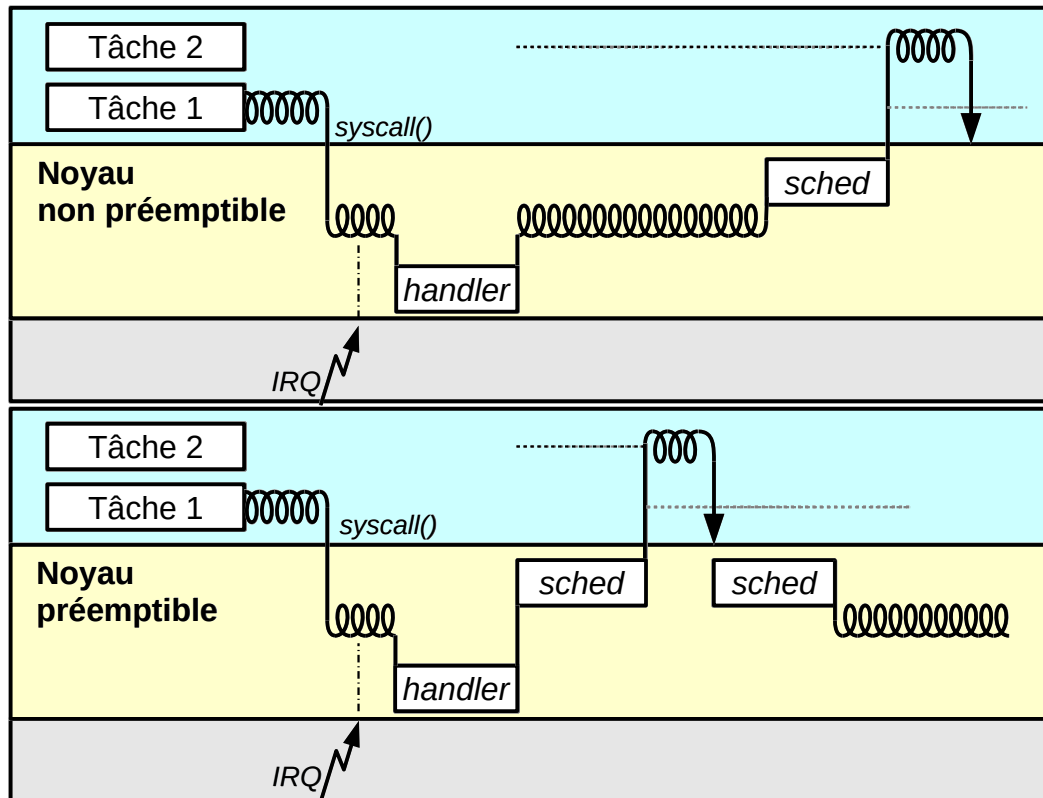
Ensuite il affiche régulièrement l'intervalle minimum, maximum et moyen mesuré entre deux déclenchements.

Lancez le processus en un ordonnancement temps réel (avec `chrt`). Observez les fluctuations en fonction de la charge système.

L'utilitaire **cyclictest** (présent dans le package **rt-tests**) permet de réaliser le même travail avec une mesure plus fine : « `cyclictest -p 99` ».

Pour perturber l'exécution de `cyclictest`, on peut lancer dans une autre console l'outil **hackbench** qui fait également partie de `rt-tests` : « `hackbench -p -g 20 -l 1000` ».

Amélioration avec la préemptibilité du noyau



Dans le noyau Linux standard, l'option « PREEMPTIBLE KERNEL » améliore le temps de réponse aux événements externes en rendant le temps de réponse prédictible.

Avec PREEMPT_RT, la prédictibilité est améliorée en évitant la latence entre l'arrivée de l'interruption et le déclenchement du *handler*.

En savoir plus...

[BLAESS 2011] Christophe Blaess – *Expérimentations sur la préemptibilité du noyau Linux*

<http://www.blaess.fr/christophe/articles/> (4 novembre 2011)

À vous...

Le noyau sur lequel votre système fonctionne est-il préemptible ?

Problèmes temps-réel classiques

Lancements parallèles en Round-Robin

Nous souhaitons lancer 4 threads de calcul en *Round-Robin* (sur le même processeur).

[...]

```
#define NB 4
```

```
int main(void)
{
```

```
    pthread_t thr[NB];
    pthread_attr_t attr;
    struct sched_param param;
    int i;
```

```
    pthread_attr_init(& attr);
    pthread_attr_setschedpolicy(&attr, SCHED_RR);
    param.sched_priority = 10;
    pthread_attr_setschedparam(&attr, &param);
    pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
```

```
    for (i = 0; i < NB; i++) {
        if (pthread_create(&(thr[i]), &attr, thread_function, NULL) != 0) {
            [...]
        }
    }
    [...]
```

À vous...

Compilez le programme **exemple-III-07.c** présenté ci-dessus et exécutez-le (en utilisant `taskset` pour fixer les threads sur le même CPU).

En fonction de votre processeur, il peut être nécessaire d'ajuster les limites des boucles dans la fonction des threads afin que l'exécution dure quelques dizaines de secondes.

L'exécution se fait-elle en *Round-Robin* ?

Une solution parmi d'autres : utiliser des barrières Posix :

[...]

```
pthread_barrier_t barrier;
```

```
void *thread_function (void *unused)
```

```
{  
    pthread_barrier_wait(&barrier);  
    [...]  
}
```

```
#define NB 4
```

```
int main(void)
```

```
{  
    pthread_t thr[NB];  
    pthread_attr_t attr;  
    struct sched_param param;  
    int i;  
  
    pthread_barrier_init(&barrier, NULL, NB);  
    [...]  
}
```

À vous...

Lancez le programme **exemple-III-08** et vérifiez que cette fois les threads s'exécutent bien en parallèle.

Inversion de priorités			
Trois threads temps-réel T1, T2 et T3 de priorités croissantes.			
	Thread 1	Thread 2	Thread 3
	Running	Sleeping	Sleeping
Le Thread 1 prend le mutex	Running	Sleeping	Sleeping
Le Thread 3 se réveille	Runnable	Sleeping	Running
Le Thread 3 demande le mutex	Running	Sleeping	Sleeping
Le Thread 2 se réveille	Runnable	Running	Sleeping
Le thread T3 est soumis à l'exécution de T2 de plus faible priorité.			

Travaux pratiques : inversions de priorités

Examinez le programme **exemple-III-09.c** qui implémente le scénario ci-dessus.

Exécutez-le (sur un seul CPU).

Y a-t-il une inversion de priorité ?

Une solution à certains cas d'inversion de priorité réside dans l'héritage de priorité (<i>priority inheritance</i>) des mutex.				
	Thread 1 (priorité)	Thread 2 (priorité)	Thread 3 (priorité)	Priorité mutex
	Running (10)	Sleeping (20)	Sleeping (30)	0
Le Thread 1 prend le mutex	Running (10)	Sleeping (20)	Sleeping (30)	10
Le Thread 3 se réveille	Runnable (10)	Sleeping (20)	Running (30)	10
Le Thread 3 demande le mutex	Running (30)	Sleeping (20)	Sleeping (30)	30
Le Thread 2 se réveille	Running (30)	Runnable (20)	Sleeping (30)	30
Le Thread 1 libère le mutex	Runnable (10)	Runnable (20)	Running (30)	30
Le Thread 3 libère le mutex	Runnable (10)	Runnable (20)	Running (30)	0
Le Thread 3 se termine	Runnable (10)	Running (20)		

- Un mutex hérite de la priorité la plus élevée parmi celles des threads qui le réclament.
- Un thread hérite de la priorité la plus élevée parmi celles de tous les mutex qu'il possède.

Travaux pratiques : héritage de priorité des mutex

Regardez la différence d'initialisation entre le programme **exemple-III-10.c** (qui utilise *PIP*) et le précédent.

Lancez **exemple-III-10**

Subsiste-t-il une inversion de priorité ?

Reprise de mutex

À vous...

Dans l'**exemple-III-11.c**, quatre threads de priorité Fifo différentes essayent d'accéder à un mutex initialement verrouillé par le thread *main* et le relâchent aussitôt obtenu.

Essayez le programme. Dans quel ordre les threads obtiennent-ils le mutex lorsqu'il est libéré par *main* ?

Inversez l'ordre de création des threads (donc de demande du mutex) pour s'assurer qu'il n'influe pas sur l'ordre d'obtention du mutex.

Dans l'**exemple-III-12.c** deux threads de priorité Round-Robin égale se disputent le même mutex.

Exécutez le programme. La répartition correspond-elle à ce que vous attendiez ?

Comment améliorer l'équité de la reprise de mutex.