

## Laborator 6 - Acțiuni amânabile

---

### Obiectivele laboratorului

---

- familiarizarea cu primitivele ce oferă suport pentru execuția codului la momente ulterioare de timp
- implementare unor task-uri uzuale ce au nevoie de ele
- înțelegerea particularităților legate de sincronizarea cu aceste primitive

### Cuvinte cheie

---

- softirq
- tasklet
- `struct tasklet_struct`
- bottom-half handlers
- jiffies, HZ
- timer
- `struct timer_list`
- `spin_lock_bh`, `spin_unlock_bh`
- workqueue
- `struct work_struct`
- kernel thread
- events/x

### Materiale ajutătoare

---

- Slide-uri de suport pentru laborator [<http://elf.cs.pub.ro/so2/res/laboratoare/lab06-slides.pdf>]
- SO2 Reference Card [[http://elf.cs.pub.ro/so2/res/extra/so2\\_reference.pdf](http://elf.cs.pub.ro/so2/res/extra/so2_reference.pdf)]

### Noțiuni generale

---

A acțiunile amânabile sunt primitive kernel care oferă posibilitatea de a planifica execuția de cod pentru un moment ulterior de timp. Acțiunile astfel planificate pot rula fie în context proces, fie în context întrerupere, în funcție de tipul de acțiune amânabilă. Acțiunile amânabile sunt folosite pentru a rezolva câteva probleme fundamentale ce apar în kernel:

- trebuie să putem planifica execuția unor acțiuni în viitor (timere);
- timpul de execuție a rutinei de tratare a întreruperii trebuie să fie cât mai mic;
- în context întrerupere nu putem folosi apeluri blocante.

Kernel thread-urile nu sunt în sine o acțiune amânabilă, dar pot fi folosite pentru a complementa mecanismul de acțiuni amânabile. În general, kernel thread-urile se folosesc ca "worker-i" ce prelucrează evenimente a căror execuție conține apeluri blocante.

Asupra tuturor tipurilor de acțiuni amânabile se pot aplica trei tipuri de operații:

- inițializarea: fiecare tip este descris de o structură ale cărei câmpuri vor trebui inițializate; la inițializare se stabilește și funcția de tratare a acțiunii;
- planificarea: planifică execuția rutinei de tratare a acțiunii imediat ce acest lucru este posibil (sau după expirarea unui timeout);
- mascarea: atunci când o acțiune este dezactivată, rutina de tratare nu va rula, chiar dacă acțiunea a fost planificată; la activare, dacă acțiunea a fost planificată, va fi rulată rutina de tratare.

Acțiunile amânabile sunt folosite de obicei pentru a complementa funcționalitatea întreruperilor. Rutina de tratare a unei întreruperi trebuie să se execute rapid, dar de cele mai multe ori operațiile care trebuie executate nu respectă această cerință. Pentru a rezolva această problemă, din rutina de tratare a întreruperii se planifică o acțiune amânabilă, pentru a rula la un moment ulterior și a executa restul operațiilor necesare.

## Locking

Pentru sincronizarea între cod ce rulează în context proces (A) și cod ce rulează în context întrerupere (B) cu handleri ale unor acțiuni amânabile avem nevoie de un locking mai special. Se vor folosi primitive de tip spinlock augmentate cu dezactivarea acțiunilor amânabile pe procesorul curent în (A), iar în (B) doar primitive de tip spinlock. Pe Linux, de exemplu, se folosesc apelurile `spin_lock_bh` și `spin_unlock_bh`. Pentru un rezumat al situațiilor în care trebuie folosită sincronizarea, consultați acest link [<http://www.kernel.org/pub/linux/kernel/people/rusty/kernel-locking/c214.html#MINIMUM-LOCK-REQUIREMENTS>].

## API Linux

---

Primitivele ce stau la baza acțiunilor amânabile în Linux sunt kernel thread-urile și softirq-urile. Pe baza kernel thread-urilor sunt implementate

cozile de sarcini (workqueues), iar pe baza softirq-urilor tasklets. Bottom-half handlers a fost prima implementare de acțiuni amânabile în Linux, dar între timp a fost înlocuită de softirq-uri. De aceea unele din funcțiile prezentate conțin **bh** în nume.

## Softirq-uri

Softirq-urile nu pot fi folosite de către device drivere, ele sunt rezervate pentru diverse subsisteme ale kernelului. Din această cauză există un număr fix de softirq-uri definite la momentul compilării. Pentru kernelul 3.13 [<http://lxr.linux.no/linux+v3.13/include/linux/interrupt.h#L342>] avem următoarele softirq-uri:

```
enum
{
    HI_SOFTIRQ=0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    BLOCK_SOFTIRQ,
    BLOCK_IOPOLL_SOFTIRQ,
    TASKLET_SOFTIRQ,
    SCHED_SOFTIRQ,
    HRTIMER_SOFTIRQ,
    RCU_SOFTIRQ,

    NR_SOFTIRQS
};
```

Scopul fiecăruia este:

- **HI\_SOFTIRQ** și **TASKLET\_SOFTIRQ** - rularea tasklets;
- **TIMER\_SOFTIRQ** - rularea timerelor;
- **NET\_TX\_SOFTIRQ** și **NET\_RX\_SOFTIRQ** - folosite de către subsistemul de networking;
- **BLOCK\_SOFTIRQ** - folosit de către subsistemul de IO;
- **BLOCK\_IOPOLL\_SOFTIRQ** - folosit de către subsistemul de IO pentru a crește performanța în momentul în care handler-ul iopoll este invocat;
- **SCHED\_SOFTIRQ** - load balancing;
- **HRTIMER\_SOFTIRQ** - implementarea timerelor de mare precizie <sup>1)</sup>;
- **RCU\_SOFTIRQ** - implementarea mecanismelor de tip RCU <sup>2)</sup>.

Prioritare sunt softirq-urile de tip **HI\_SOFTIRQ**, urmate în ordine de celelalte softirq-uri definite; cele de tip **RCU\_SOFTIRQ** au cea mai mică prioritate.

Softirq-urile rulează în context întrerupere, astfel încât din cadrul lor nu se pot apela funcții blocante. Dacă tratarea evenimentelor semnalizate din softirq-uri necesită apeluri către astfel de funcții, se pot planifica workqueue-uri care să execute aceste apeluri blocante.

## Tasklets

Un tasklet este caracterizat prin structura `struct tasklet_struct` [<http://lxr.linux.no/linux+v3.13/include/linux/interrupt.h#L420>]. Un tasklet preinițializat se definește astfel:

```
void handler(unsigned long data);
DECLARE_TASKLET(tasklet, handler, data);
DECLARE_TASKLET_DISABLED(tasklet, handler, data);
```

sau dacă dorim să inițializăm manual tasklet-ul:

```
void handler(unsigned long data);
struct tasklet_struct tasklet;

tasklet_init(&tasklet, handler, data);
```

Parametrul **data** va fi trimis handler-ului în momentul în care acesta se va executa.

Programarea de tasklets pentru rulare se numește planificare. Tasklets se planifica peste softirq-uri. Planificarea de tasklets se face cu:

```
void tasklet_schedule(struct tasklet_struct *tasklet);
void tasklet_hi_schedule(struct tasklet_struct *tasklet);
```

Pentru `tasklet_schedule` se planifică un softirq de tip `TASKLET_SOFTIRQ`, iar pentru `tasklet_hi_schedule` se planifică un softirq de tip `HI_SOFTIRQ`.

Dacă un tasklet a fost planificat de mai multe ori și nu a rulat între planificări, el va rula o singură dată:

```
tasklet_schedule(&tasklet);

/* presupunem ca tasklet-ul nu a rulat inca */
tasklet_schedule(&tasklet);

/* in aceste conditii tasklet-ul va rula o singura data */
```

O dată ce tasklet-ul a rulat, el poate fi replanificat, și va rula după replanificare:

```
tasklet_schedule(&tasklet);  
/* presupunem ca tasklet-ul planificat a rulat */  
tasklet_schedule(&tasklet);  
/* in aceste condiții tasklet-ul va rula la un moment ulterior de timp */
```

Tasklets se pot replanifica din interiorul handler-ului și vor rula la un moment de timp ulterior după ieșirea din handler.

Tasklets pot fi mascați de la rulare. Mascarea tasklets se face cu:

```
void tasklet_enable(struct tasklet_struct *tasklet);  
void tasklet_disable(struct tasklet_struct *tasklet);
```

Intrucât tasklets sunt planificați peste softirq-uri, în codul asociat nu pot fi folosite apeluri blocante.

## Timere

Un caz particular de acțiuni amânabile, dar foarte des folosite, sunt timer-ele: struct timer\_list [<http://lxr.linux.no/linux+v3.13/include/linux/timer.h#L12>]. În Linux, timer-ele rulează din context întrerupere, fiind implementate cu ajutorul softirq-urilor.

Pentru a putea fi folosit, un timer trebuie mai întâi inițializat apelând setup\_timer [<http://lxr.linux.no/linux+v3.13/include/linux/timer.h#L152>]:

```
#include <linux/sched.h>  
  
void setup_timer(struct timer_list *timer,  
                void (*function)(unsigned long),  
                unsigned long data);
```

Funcția de mai sus inițializează câmpurile interne ale structurii și asociază `function` ca rutina de tratare a timerului; parametrul `data` va fi transmis funcției de tratare. Intrucât timer-ele sunt planificate peste softirq-uri, în codul asociat funcției de tratare nu pot fi folosite apeluri blocante.

Planificarea unui timer se face cu `mod_timer` [<http://lxr.linux.no/linux+v3.13/kernel/timer.c#L832>]:

```
int mod_timer(struct timer_list *timer, unsigned long expires);
```

unde `expires` este timpul (din viitor) la care să se ruleze funcția de tratare. Funcția poate fi folosită pentru a planifica sau pentru a replanifica un timer.

Unitatea în care se măsoară timpul pentru aceste tipuri de timere este `jiffie`<sup>3)</sup>. Valoarea în timp absolut a unui jiffie este dependentă de platformă, și se poate afla cu ajutorul macroului `HZ` care definește numărul de jiffies pentru 1 secundă. Pentru a transforma între un interval de timp în jiffies (`jiffies_value`) și un interval în secunde (`seconds_value`) se folosesc următoarele formule:

```
jiffies_value = seconds_value * HZ;  
seconds_value = jiffies_value / HZ;
```

În kernel există un contor care conține numărul de jiffies de la ultimul boot, ce poate fi accesat prin variabila `jiffies`. Astfel, atunci când este nevoie să se calculeze un timp în viitor se poate folosi această variabilă:

```
#include <linux/jiffies.h>  
  
unsigned long current_jiffies, next_jiffies;  
unsigned long seconds = 1;  
  
current_jiffies = jiffies;  
next_jiffies = jiffies + seconds * HZ; /* 'seconds' seconds in the future */
```

Pentru a opri un timer se folosesc funcțiile `del_timer` [<http://lxr.linux.no/linux+v3.13/kernel/timer.c#L948>] și `del_timer_sync` [<http://lxr.linux.no/linux+v3.13/kernel/timer.c#L1007>]:

```
int del_timer(struct timer_list *timer);  
int del_timer_sync(struct timer_list *timer);
```

Funcțiile se pot apela atât pentru un timer planificat cât și pentru un timer neplanificat. Funcția `del_timer_sync` [<http://lxr.linux.no/linux+v3.13/kernel/timer.c#L1007>] este folosită pentru a elimina race-urile ce pot apărea pe sisteme multiprocesor, întrucât la terminarea apelului se garantează că funcția de tratare a timer-ului nu rulează pe niciun procesor.

O greșeală frecventă a folosirii timer-elor este aceea că se uită oprirea timerelor pornite. De exemplu, **înainte de descărcarea unui modul trebuie să opriți timerele**, pentru că dacă un timer expiră după descărcarea modului, funcția de tratare nu va mai fi încărcată în kernel și se va genera un kernel oops.

Secvența uzuală folosită pentru inițializarea și planificarea unui timeout de `seconds` secunde este:

```
#include <linux/sched.h>

void timer_function(unsigned long arg);

struct timer_list timer;
unsigned long seconds = 1;

setup_timer(&timer, timer_function, 0);
mod_timer(&timer, jiffies + seconds * HZ);
```

iar pentru oprirea acestuia:

```
del_timer_sync(&timer);
```

## Locking softirq-uri

Pentru a masca softirq-urile (inclusiv timerele sau taskleții) puteți folosi funcțiile `local_bh_disable` [<http://lxr.linux.no/linux+v3.13/kernel/softirq.c#L125>] / `local_bh_enable` [<http://lxr.linux.no/linux+v3.13/kernel/softirq.c#L186>]:

```
void local_bh_disable(void);
void local_bh_enable(void);
```

**Atenție!** Aceste primitive vor dezactiva softirq-urile doar pe procesorul local. Sunt permise construcții imbricate, reactivarea efectivă a softirq-urilor făcându-se doar atunci când toate apelurile `local_bh_disable()` au fost complementate de apeluri `local_bh_enable()`:

```
/* presupunem că avem softirq-urile nemascate */

local_bh_disable(); /* softirq-urile sunt acum mascate */

local_bh_disable(); /* softirq-urile rămân mascate */

local_bh_enable(); /* softirq-urile rămân mascate */

local_bh_enable(); /* softirq-urile sunt acum nemascate */
```

Pentru situațiile în care trebuie să mascați softirq-urile dar și să folosiți lock-uri puteți folosi funcțiile de mai jos, definite în `linux/spinlock.h` [<http://lxr.linux.no/linux+v3.13/include/linux/spinlock.h#L296>]<sup>4)</sup>. Funcțiile `*_lock_bh()` vor dezactiva softirq-urile, iar apoi vor efectua operația de *lock*. Funcțiile `*_unlock_bh()` vor efectua operația de *unlock* și apoi vor reactiva softirq-urile.

```
void spin_lock_bh(spinlock_t *lock);
```

```
void spin_unlock_bh(spinlock_t *lock);  
void read_lock_bh(rwlock_t *lock);  
void read_unlock_bh(rwlock_t *lock);  
void write_lock_bh(rwlock_t *lock);  
void write_unlock_bh(rwlock_t *lock);
```

## Workqueues

Puteți folosi workqueue-uri pentru a planifica acțiuni care să ruleze în context proces. Unitatea de bază cu care se lucrează poartă denumirea de **work**. Există două structuri care definesc o sarcină: `struct work_struct` [<http://lxr.linux.no/linux+v3.13/include/linux/workqueue.h#L100>] (pentru a planifica o sarcină să ruleze la un moment ulterior de timp) și `struct delayed_work` [<http://lxr.linux.no/linux+v3.13/include/linux/workqueue.h#L113>] (pentru a putea planifica o sarcină să ruleze după cel puțin un interval de timp dat). O sarcină de tipul `struct delayed_work` folosește un **timer** pentru a rula după intervalul de timp specificat; funcțiile pentru lucrul cu acest tip de sarcini sunt similare cu cele pentru `struct work_struct`, dar conțin `delayed` în numele funcției. O sarcină se poate inițializa cu ajutorul următoarelor macrodefiniții:

```
#include <linux/workqueue.h>  
  
DECLARE_WORK(name, void (*function)( struct work_struct *));  
DECLARE_DELAYED_WORK(name, void (*function)( struct work_struct *));  
INIT_WORK(struct work_struct *work, void (*function)( struct work_struct *));  
INIT_DELAYED_WORK(struct delayed_work *work, void (*function)( struct work_struct *));
```

`DECLARE_WORK` și `DECLARE_DELAYED_WORK` declară și inițializează sarcina, iar `INIT_WORK` și `INIT_DELAYED_WORK` inițializează o sarcină deja declarată.

Secvența următoare declară și inițializează o sarcină:

```
#include <linux/workqueue.h>  
  
void my_work_handler(struct work_struct *work);  
  
DECLARE_WORK(my_work, my_work_handler);
```

sau, dacă dorim să inițializăm manual sarcina:

```
void my_work_handler(struct work_struct *work);  
  
struct work_struct my_work;
```



```
INIT_WORK(&my_work, my_work_handler);
```

Odată declarată și inițializată, putem planifica sarcina folosind funcțiile `schedule_work` [<http://lxr.linux.no/linux+v3.13/include/linux/workqueue.h#L555>] și `schedule_delayed_work` [<http://lxr.linux.no/linux+v3.13/include/linux/workqueue.h#L586>]:

```
schedule_work(struct work_struct *work);
schedule_delayed_work(struct delayed_work *work, unsigned long delay);
```

Funcția `schedule_delayed_work` [<http://lxr.linux.no/linux+v3.13/include/linux/workqueue.h#L586>] poate fi folosită pentru a planifica un work pentru execuție cu o întârziere de minim **delay**; întârzierea este dată în jiffies.

Sarcinile nu pot fi mascate.

O sarcină planificată cu întârziere, dar care nu a rulat încă, poate fi anulată apelând `cancel_delayed_work_sync` [<http://lxr.linux.no/linux+v3.13/kernel/workqueue.c#L2987>]:

```
int cancel_delayed_work_sync(struct delayed_work *work);
```

Apelul nu face decât să oprească execuția ulterioară a sarcinii; dacă funcția asociată sarcinii este deja în execuție la momentul apelului, aceasta va rula în continuare.

Putem să așteptăm terminarea rulării sarcinilor din coada folosind `flush_scheduled_work` [<http://lxr.linux.no/linux+v3.13/kernel/workqueue.c#L3039>]:

```
void flush_scheduled_work(void);
```

Această funcție este blocantă și, din această cauză, nu poate fi folosită din context întrerupere. La execuția acestei funcții se va aștepta terminarea tuturor sarcinilor din coadă existente la momentul apelului; pentru sarcinile planificate cu întârziere trebuie apelată funcția `cancel_delayed_work` înainte de apelul `flush_scheduled_work`.

În fine, următoarele funcții pot fi folosite pentru a planifica sarcini pe un anumit procesor (`schedule_delayed_work_on` [<http://lxr.linux.no/linux+v3.13/include/linux/workqueue.h#L571>]), respectiv pe toate procesoarele (`schedule_on_each_cpu` [<http://lxr.linux.no/linux+v3.13/kernel/workqueue.c#L3002>]):

```
int schedule_delayed_work_on(int cpu, struct delayed_work *work, unsigned long delay);
int schedule_on_each_cpu(void (*func)( struct work_struct *));
```

O secvență uzuală de inițializare și planificare a unei sarcini este următoarea:

```
void my_work_handler(struct work_struct *work);

struct work_struct my_work;

INIT_WORK(&my_work, my_work_handler);

schedule_work(&my_work);
```

iar pentru așteptarea terminării sarcinii:

```
flush_scheduled_work();
```

După cum se poate observa, funcția `my_work_handler` primește drept parametru sarcina care se execută. Pentru a putea accesa date private ale modulului, se poate folosi macrodefiniția `container_of` [<http://lxr.linux.no/linux+v3.13/include/linux/kernel.h#L784>]<sup>5)</sup>:

```
struct my_device_data {
    struct work_struct my_work;
    //...
};

void my_work_handler(struct work_struct *work) {
    struct my_device_data *my_data = container_of(work, struct my_device_data, my_work);
    //...
}
```

Sarcinile planificate cu funcțiile discutate mai sus vor rula în contextul unui kernel thread denumit **events/x**, unde **x** este numărul procesorului pe care rulează kernel thread-ul. Kernelul va crea la inițializare câte un kernel thread pentru fiecare procesor prezent în sistem:

```
$ ps -e
  PID TTY          TIME CMD
   1 ?            00:00:00 init
   2 ?            00:00:00 ksoftirqd/0
   3 ?            00:00:00 events/0    <--- kernel thread-ul peste care rulează workqueue-urile
   4 ?            00:00:00 khelper
   5 ?            00:00:00 kthread
   7 ?            00:00:00 kblockd/0
   8 ?            00:00:00 kacpid
...
```

Funcțiile declarate mai sus folosesc o coadă de sarcini predefinită (numită **events**), iar acestea rulează în contextul thread-ului **events/x**, după cum s-a precizat mai sus. Deși aceasta este suficientă în majoritatea cazurilor, este o resursă partajată și întârzieri mari în funcțiile asociate

sarcinilor pot cauza întârzieri celorlalți utilizatori ai cozii. Din acest motiv, există funcții pentru crearea de cozi de sarcini suplimentare.

O coada de sarcini este reprezentată de struct `workqueue_struct` [<http://lxr.linux.no/linux+v3.13/kernel/workqueue.c#L228>]. Poate fi creată cu ajutorul funcțiilor:

```
struct workqueue_struct *create_workqueue(const char *name);
struct workqueue_struct *create_singlethread_workqueue(const char *name);
```

Funcția `create_workqueue` [<http://lxr.linux.no/linux+v3.13/include/linux/workqueue.h#L452>] folosește câte un thread pentru fiecare procesor din sistem, iar `create_singlethread_workqueue` [<http://lxr.linux.no/linux+v3.13/include/linux/workqueue.h#L457>] folosește un singur thread.

Pentru a adăuga o sarcina in coada suplimentară se vor folosi funcțiile `queue_work` [<http://lxr.linux.no/linux+v3.13/include/linux/workqueue.h#L497>] și `queue_delayed_work` [<http://lxr.linux.no/linux+v3.13/include/linux/workqueue.h#L513>]:

```
int queue_work(struct workqueue_struct *queue, struct work_struct *work);
int queue_delayed_work(struct workqueue_struct *queue,
                      struct delayed_work *work, unsigned long delay);
```

Funcția `queue_delayed_work` [<http://lxr.linux.no/linux+v3.13/include/linux/workqueue.h#L513>] poate fi folosită pentru a planifica un work pentru execuție cu o întârziere de minim **delay**; întârzierea este dată în jiffies.

Pentru a aștepta terminarea sarcinilor din coadă se apelează `flush_workqueue` [<http://lxr.linux.no/linux+v3.13/kernel/workqueue.c#L2597>]:

```
void flush_workqueue(struct workqueue_struct *queue);
```

iar pentru a distruge coada `destroy_workqueue` [<http://lxr.linux.no/linux+v3.13/kernel/workqueue.c#L4266>]:

```
void destroy_workqueue(struct workqueue_struct *queue);
```

Următoarea secvență declară și inițializează o coadă suplimentară de sarcini, declară și inițializează o sarcină și o adaugă în coadă:

```
void my_work_handler(struct work_struct *work);

struct work_struct my_work;
struct workqueue_struct *my_workqueue;

my_workqueue = create_singlethread_workqueue("my_workqueue");
INIT_WORK(&my_work, my_work_handler);
queue_work(my_workqueue, &my_work);
```

Pentru așteptarea terminării sarcinilor din coadă se va apela:

```
flush_workqueue(my_workqueue);  
destroy_workqueue(my_workqueue);
```

Sarcinile planificate cu aceste funcții vor rula în contextul unei nou kernel thread denumit **my\_workqueue**, numele dat la crearea cozii de sarcini cu apelul `create_singlethread_workqueue` [<http://lxr.linux.no/linux+v3.13/include/linux/workqueue.h#L457>].

## Kernel threads

Kernel thread-urile au apărut din necesitatea de a rula cod în context proces din kernel. Kernel thread-urile stau la baza mecanismului de workqueue. În esență, un kernel thread este un thread ce rulează în contextul procesului init și nu rulează decât în kernel-mode. Din acest motiv, un kernel thread nu are asociat informații precum un spațiu de adresă user.

Pentru a crea un kernel thread, se apelează funcția `kthread_create_on_node` [<http://lxr.linux.no/linux+v3.13/kernel/kthread.c#L245>] sau `kthread_create_on_cpu` [<http://lxr.linux.no/linux+v3.13/kernel/kthread.c#L355>]:

```
#include <linux/kthread.h>  
  
struct task_struct *kthread_create_on_node(int (*threadfn)(void *data),  
                                           void *data, int node,  
                                           const char namefmt[],  
                                           ...);  
  
struct task_struct *kthread_create_on_cpu(int (*threadfn)(void *data),  
                                          void *data, unsigned int cpu,  
                                          const char *namefmt);
```

unde:

- `threadfn` este o funcția ce va fi rulată de kernel thread
- `data` este un parametru ce va fi trimis funcției
- `namefmt` reprezintă numele kernel thread-ului, așa cum este el afișat în `ps/top`; poate conține secvențe `%d`, `%s` etc. care vor fi înlocuite conform semnificației standard `printf`.

De exemplu, următorul apel:

```
kthread_create_on_node(f, NULL, -1, "%skthread%d", "my", 0);
```

va crea un kernel thread cu numele **mykthread0**.

Kernel thread-ul creat cu această funcție va fi oprit, în starea **TASK\_INTERRUPTIBLE**. Pentru a porni kernel thread-ul se va apela funcția `wake_up_process` [<http://lxr.linux.no/linux+v3.13/kernel/sched/core.c#L1674>]:

```
#include <linux/sched.h>

int wake_up_process(struct task_struct *p);
```

Alternativ, se poate folosi macro-ul `kthread_run` [<http://lxr.linux.no/linux+v3.13/include/linux/kthread.h#L22>]:

```
struct task_struct *kthread_run(int (*threadfn)(void *data),
                                void *data, const char namefmt[], ...);
```

pentru a crea și porni un kernel thread.

Chiar dacă restricțiile de programare pentru funcția ce rulează în cadrul kernel thread-ului sunt mai relaxate și programarea se apropie mai mult de programarea în userspace, există, totuși, câteva limitări de care trebuie să ținut cont. Vom enumera mai jos acțiunile care se pot sau nu se pot face dintr-un kernel thread:

- nu se poate accesa spațiul de adresă utilizator (nici măcar cu funcții gen `copy_from_user`, `copy_to_user`) pentru că un kernel thread nu are un spațiu utilizator și este planificat pentru execuție indiferent de procesul din user-space;
- nu se poate implementa cod busy waiting cu o durată mare de timp; dacă aveți un kernel compilat fără opțiunea de preemptivitate, respectivul cod va rula fără a putea fi preemptat de alte procese/kernel thread-uri și, în acest mod, veți “bloca” sistemul;
- puteți chema operații blocante din kernel thread;
- puteți folosi spinlock-uri, dar, dacă durata de ținere a lock-ului este mare, se recomandă folosirea de semafoare.

Terminarea unui kernel thread se face voluntar, din interiorul funcției ce rulează în kernel thread, prin apelarea funcției:

```
fastcall NORET_TYPE void do_exit(long code);
```

Datorită limitărilor existente, cea mai mare parte din implementările funcțiilor ce rulează în kernel thread-uri folosesc același model. Mai jos prezentăm o posibilă implementare a acestui model:

```
#include <linux/kthread.h>

// semafor folosit pentru a semnaliza kernel thread-ului ca are evenimente de procesat
struct semaphore sem;
```

```
// lista de evenimente de procesat de kernel thread
struct list_head lista_evenimente;

// structura ce descrie evenimentul de procesat
struct eveniment {
    struct list_head lh;
    // ...
};

int my_thread_f(void *data)
{
    while (1) {

        down(&sem);

        spin_lock(&lock_lista);
        while (i = lista_evenimente.next) {
            struct eveniment *ev = list_entry(i, struct eveniment, lh);
            list_del(i);
            spin_unlock(&lock_lista);

            /* procesare eveniment */
            // ...

            /* daca se cere terminarea kernel thread-ului */
            if (ev->...)
                break;
            spin_lock(&lock_lista);
        }
        spin_unlock(&lock_lista);

    }

    do_exit(0);
}

// ...

// initializare si pornire kthread
kthread_run(my_thread_f, NULL, "%skthread%d", "my", 0);

// ...
```

Cu modelul prezentat mai sus, comunicarea de cereri către kernel thread se face astfel:

```
void trimite_cerere(struct eveniment *ev)
{
    spin_lock(&lock_lista);
```

```
list_add(&ev->lh, &lista_evenimente);  
spin_unlock(&lock_lista);  
up(&sem);  
}
```

## Resurse utile

---

### Linux

- Linux Device Drivers, 3rd ed., Ch. 7: Time, Delays, and Deferred Work [<http://lwn.net/images/pdf/LDD3/ch07.pdf>]
- Scheduling Tasks [<http://tldp.org/LDP/lkmpg/2.6/html/x1211.html>]
- Driver porting: the workqueue interface [<http://lwn.net/Articles/23634/>]
- Workqueues get a rework [<http://lwn.net/Articles/211279/>]
- Kernel threads made easy [<http://lwn.net/Articles/65178/>]
- Unreliable Guide to Locking [<http://www.kernel.org/pub/linux/kernel/people/rusty/kernel-locking/index.html>]

---

<sup>1)</sup> jiffie – începând cu 2.6.21, arhitectura tratării timerelor a fost complet rescrisă; se așteaptă ca în viitor și jiffie-ul să dispară. Pentru detalii consultați [Clockevents and dyntick](http://lwn.net/Articles/223185/) [<http://lwn.net/Articles/223185/>] și [Documentation/timers/hrtimers.txt](http://lxr.linux.no/linux+v3.13/Documentation/timers/hrtimers.txt) [<http://lxr.linux.no/linux+v3.13/Documentation/timers/hrtimers.txt>]

<sup>2)</sup> read-copy update – un mecanism prin care operațiile distructive (ex: ștergerea unui element dintr-o listă înlănțuită) se fac în două etape: eliminarea referințelor către datele de șters și ștergerea propriu-zisă, care se face numai după ce este sigur că nimeni nu le mai folosește. Avantajul este că accesarea datelor se poate face fără sincronizare. Pentru mai multe informații citiți documentația RCU [<http://lxr.linux.no/linux+v3.13/Documentation/RCU/rcu.txt>] din kernelul Linux

<sup>3)</sup> jiffie – începând cu 2.6.21, arhitectura tratării timerelor a fost complet rescrisă; se așteaptă ca în viitor și jiffie-ul să dispară. Pentru detalii consultați [Clockevents and dyntick](http://lwn.net/Articles/223185/) [<http://lwn.net/Articles/223185/>] și [Documentation/timers/hrtimers.txt](http://lxr.linux.no/linux+v3.13/Documentation/timers/hrtimers.txt) [<http://lxr.linux.no/linux+v3.13/Documentation/timers/hrtimers.txt>]

<sup>4)</sup> spinlock – pentru mai multe detalii despre folosirea spinlock-urilor, revedeți [Laboratorul 3](#)

<sup>5)</sup> container\_of – un exemplu de utilizare pentru macrodefiniția `container_of` este la parcurgerea [listelor](#) din kernel