Laborator 7 - Acţiuni amânabile

Obiectivele laboratorului

- familiarizarea cu primitivele ce oferă suport pentru execuţia codului la momente ulterioare de timp
- implementare unor task-uri uzuale ce au nevoie de ele
- înţelegerea particularităţilor 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
- DPC
- KDPC
- KTIMER
- DISPATCH_LEVEL, DIRQ
- KeWaitForSingleObject
- IO_WORKITEM
- system worker thread
- system thread

Materiale ajutătoare

- Slide-uri de suport pentru laborator
- SO2 Reference Card

Noţiuni generale

Acţiunile amânabile sunt primitive kernel care oferă posibilitatea de a planifica execuţia de cod pentru un moment ulterior de timp. Acţiunile astfel planificate pot rula fie în context proces, fie în context întrerupere, în funcţie de tipul de acţiune amânabilă. Acţiunile amânabile sunt folosite pentru a rezolva câteva probleme fundamentale ce apar în kernel:

- trebuie să putem planifica execuţia unor acţiuni în viitor (timere);
- timpul de execuţie 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 acţiune amânabilă, dar pot fi folosite pentru a complementa mecanismul de acţiuni amânabile. În general, kernel thread-urile se folosesc ca "worker-i" ce prelucrează evenimente a căror execuţie conţine apeluri blocante.

Asupra tuturor tipurilor de acţiuni amânabile se pot aplica trei tipuri de operaţii:

- iniţializarea: fiecare tip este descris de o structură ale cărei câmpuri vor trebui iniţializate; la iniţializare se stabileşte şi funcţia de tratare a acţiunii;
- planificarea: planifică execuţia rutinei de tratare a acţiunii imediat ce acest lucru este posibil (sau după expirarea unui timeout);

- mascarea: atunci când o acţiune este dezactivată, rutina de tratare nu va rula, chiar dacă acţiunea a fost planificată; la activare, dacă acţiunea a fost planificată, va fi rulată rutina de tratare.

Acţiunile amânabile sunt folosite de obicei pentru a complementa funcţionalitatea întreruperilor. Rutina de tratare a unei întreruperi trebuie să se execute rapid, dar de cele mai multe ori operaţiile care trebuiesc executate nu respectă această cerinţă. Pentru a rezolva această problemă, din rutina de tratare a întreruperii se planifică o acţiune amânabilă, pentru a rula la un moment ulterior şi a executa restul operaţiilor necesare.

Locking

Pentru sincronizarea între cod ce rulează în context proces (A) şi cod ce rulează în context întrerupere (B) cu handlere ale unor acţiuni amânabile avem nevoie de un locking mai special. Se vor folosi primitive de tip spinlock augmentate cu dezactivarea acţiunilor amânabile pe procesorul curent în (A), iar în (B) doar primitive de tip spinlock. Pe Linux, de exemplu, se folosesc apelurile `spin_lock_bh` şi `spin_unlock_bh`. Pentru un rezumat al situaţiilor în care trebuie folosită sincronizarea, consultaţi acest link.

API Linux

Primitivele ce stau la baza acţiunilor amânabile în Linux sunt kernel thread-urile şi 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 acţiuni amânabile în Linux, dar între timp a fost înlocuită de softirq-uri. De aceea unele din funcţiile prezentate conţin `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 2.6.31 avem următoarele softirq-uri:

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

        NR_SOFTIRQS
};
```

Scopul fiecăruia este:

- `HI_SOFTIRQ` şi `TASKLET_SOFTIRQ` - rularea tasklets;
- `TIMER_SOFTIRQ` - rularea timerelor;
- `NET_TX_SOFIRQ` şi `NET_RX_SOFTIRQ` - folosite de către subsistemul de networking;
- `BLOCK_SOFTIRQ` - folosit de către subsistemul de IO;
- `SCHED_SOFTIRQ` - load balancing;
- `HRTIMER_SOFTIRQ` - implementarea timerelor de mare precizie [1];
- `RCU_SOFTIRQ` - implementarea mecanismelor de tip RCU [2].

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 funcţii blocante. Dacă tratarea evenimentelor semnalizate din softirq-uri necesită apeluri către astfel de funcţii, se pot planifica workqueue-uri care să execute aceste apeluri blocante.

Tasklets

Un tasklet este caracterizat prin structura struct tasklet_struct.

Un tasklet preiniţializat se defineşte astfel:

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

sau dacă dorim să iniţializăm 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 numeste 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 şi nu a rulat între planificări, el va rula o singură data:

```
tasklet_schedule(&tasklet);

/* presupunem ca tasklet-ul nu a rulat inca */

tasklet_schedule(&tasklet);

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

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

```
tasklet_schedule(&tasklet);

/* presupunem ca tasklet-ul planificat a rulat */

tasklet_schedule(&tasklet);

/* in aceste condiţii tasklet-ul va rula la un moment ulterior de timp */
```

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

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

```
void tasklet_enable(struct task_struct *tasklet);

void tasklet_disable(struct task_struct *tasklet);
```

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

Timere

Un caz particular de acţiuni amânabile, dar foarte des folosite, sunt timer-ele: struct timer_list.
În Linux, timer-ele rulează din context întrerupere, fiind implementate cu ajutorul softirq-urilor.

Pentru a putea fi folosit, un timer trebuie mai întâi iniţializat apelând setup_timer:

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

void setup_timer(struct timer_list *timer,
                 void (*function)(unsigned long),
                 unsigned long data);
```

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

Planificarea unui timer se face cu mod_timer:

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

unde `expires` este timpul (din viitor) la care să se ruleze funcţia de tratare. Funcţia 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` [3]. Valoarea în timp absolut a unui jiffie este dependentă de platformă, şi se poate afla cu ajutorul macroului `HZ` care defineşte numărul de jiffies pentru 1 secundă. Pentru a transforma între un interval de timp în jiffies (`jiffies_value`) şi 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 conţine numarul de jiffies de la ultimul boot, ce poate fi accesat prin variabila `jiffies`. Astfel, atunci când este nevoie să se calculeze un timp in 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 funcţiile del_timer şi del_timer_sync:

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

Funcţiile se pot apela atât pentru un timer planificat cât şi pentru un timer neplanificat. Funcţia del_timer_sync este folosită pentru a elimina race-urile ce pot apărea pe sisteme multiprocesor, întrucât la terminarea apelului se garantează că funcţia de tratare a timer-ului nu rulează pe niciun procesor.

O greşeală frecventă a folosirii timer-elor este aceea că se uită oprirea timerelor pornite. De exemplu, înainte de descărcarea unui modul trebuie să oprim timerele, pentru că dacă un timer expiră după descărcarea modulului, funcţia de tratare nu va mai fi încărcată în kernel şi se va genera un kernel oops.

Secvenţa uzuală folosită pentru iniţializarea şi 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 taskleţii) puteţi folosi funcţiile local_bh_disable / local_bh_enable:

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

Atenţie! Aceste primitive vor dezactiva softirq-urile doar pe procesorul local. Sunt permise construcţii 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 situaţiile în care trebuie să mascaţi softirq-urile dar şi să folosiţi lock-uri puteţi folosi funcţiile de mai jos, definite în linux/spinlock.h [4]. Funcţiile `*_lock_bh()` vor dezactiva softirq-urile, iar apoi vor efectua operaţia de *lock*. Funcţiile `*_unlock_bh()` vor efectua operaţia de *unlock* şi 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

Puteţi folosi workqueue-uri pentru a planifica acţiuni 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 (pentru a planifica o sarcină să ruleze la un moment ulterior de timp) şi struct delayed_work (pentru a putea planifica o sarcină să ruleze după cel puţin un interval de timp dat). O sarcină de tipul `struct delayed_work` foloseşte un `timer` pentru a rula după intervalul de timp specificat; funcţiile pentru lucrul cu acest tip de sarcini sunt similare cu cele pentru `struct work_struct`, dar conţin `delayed` în numele funcţiei. O sarcină se poate iniţializa cu ajutorul următoarelor macrodefiniţii:

```
#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` şi `DECLARE_DELAYED_WORK` declară şi iniţializează sarcina, iar `INIT_WORK` şi `INIT_DELAYED_WORK` iniţializează o sarcină deja declarată.

Secvenţa următoare declară şi iniţializează 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 iniţializăm manual sarcina:

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

struct work_struct my_work;

INIT_WORK(&my_work, my_work_handler);
```

Odată declarată şi iniţializată, putem planifica sarcina folosind funcţiile schedule_work şi schedule_delayed_work:

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

Funcţia schedule_delayed_work poate fi folosită pentru a planifica un work pentru execuţie 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:

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

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

Putem să aşteptăm terminarea rulării sarcinilor din coada folosind flush_scheduled_work:

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

Această funcţie este blocantă şi, din această cauză, nu poate fi folosită din context întrerupere. La execuţia acestei funcţii se va aştepta terminarea tuturor sarcinilor din coadă existente la momentul apelului; pentru sarcinile planificate cu întârziere trebuie apelată funcţia `cancel_delayed_work` înainte de apelul `flush_scheduled_work`.

În fine, următoarele funcţii pot fi folosite pentru a planifica sarcini pe un anumit procesor (schedule_delayed_work_on), respectiv pe toate procesoarele (schedule_on_each_cpu):

```
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 secvenţa uzuală de iniţializare şi 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 aşteptarea terminării sarcinii:

```
flush_scheduled_work();
```

După cum se poate observa, funcţia `my_work_handler` primeşte drept parametru sarcina care se execută. Pentru a putea accesa date private ale modulului, se poate folosi macrodefiniţia container_of [5]:

```
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 funcţiile 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 iniţializare 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
...
```

Funcţiile declarate mai sus folosesc o coadă de sarcini predefinită (numită **events**), iar acestea rulează in contextul thread-ului **events/x**, după cum s-a precizat mai sus. Deşi aceasta este suficientă in majoritatea cazurilor, este o resursă partajată şi întârzieri mari în funcţiile asociate sarcinilor pot cauza întârzieri celorlalţi utilizatori ai cozii. Din acest motiv, există funcţii pentru crearea de cozi de sarcini suplimentare.

O coada de sarcini este reprezentată de struct workqueue_struct. Poate fi creată cu ajutorul funcţiilor:

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

Funcţia create_workqueue foloseşte câte un thread pentru fiecare procesor din sistem, iar create_singlethread_workqueue foloseşte un singur thread.

Pentru a adăuga o sarcina in coada suplimentară se vor folosi funcţiile queue_work şi queue_delayed_work:

```
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);
```

Funcţia queue_delayed_work poate fi folosită pentru a planifica un work pentru execuţie cu o întârziere de minim `delay`; întârzierea este dată în jiffies.

Pentru a aştepta terminarea sarcinilor din coadă se apelează flush_workqueue:

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

iar pentru a distruge coada destroy_workqueue:

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

Următoarea secvenţă declară şi iniţializează o coadă suplimentară de sarcini, declară şi iniţializează o sarcină şi 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 aşteptarea terminării sarcinilor din coadă se va apela:

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

Sarcinile planificate cu aceste funcţii vor rula în contextul unei nou kernel thread denumit **my_workqueue**, numele dat la crearea cozii de sarcini cu apelul create_singlethread_workqueue.

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 esenţă, un kernel thread este un thread ce rulează în contextul procesului init şi nu rulează decât în kernel-mode. Din acest motiv, un kernel thread nu are asociat informaţii precum un spaţiu de adresă user.

Pentru a crea un kernel thread, se apelează funcţia kthread_create:

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

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

unde:

- `threadfn` este o funcţia ce va fi rulată de kernel thread
- `data` este un parametru ce va fi trimis funcţiei
- `namefmt` reprezintă numele kernel thread-ului, aşa cum este el afişat în `ps`/`top`; poate conţine secvenţe `%d`, `%s` etc. care vor fi inlocuite conform semnificaţiei standard `printf`.

De exemplu, următorul apel:

```
kthread_create(f, NULL, "%skthread%d", "my", 0);
```

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

Kernel thread-ul creat cu această funcţie va fi oprit, în starea `TASK_INTERRUPTIBLE`. Pentru a porni kernel thread-ul se va apela funcţia wake_up_process:

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

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

Alternativ, se poate folosi macro-ul kthread_run:

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

pentru a crea şi porni un kernel thread.

Chiar dacă restricţiile de programare pentru funcţia ce rulează în cadrul kernel thread-ului sunt mai relaxate şi programarea se apropie mai mult de programarea în userspace, există, totuşi, câteva limitări de care trebuie să ţinut cont. Vom enumera mai jos acţiunile care se pot sau nu se pot face dintr-un kernel thread:

- nu se poate accesa spaţiul de adresă utilizator (nici măcar cu funcţii gen `copy_from_user`, `copy_to_user`) pentru că un kernel thread nu are un spaţiu utilizator şi este planificat pentru execuţie indiferent de procesul din user-space;
- nu se poate implementa cod busy waiting cu o durata mare de timp; dacă aveţi un kernel compilat fără opţiunea de preemptivitate, respectivul cod va rula fără a putea fi preemptat de alte procese/kernel thread-uri şi, în acest mod, veţi "bloca" sistemul;
- puteţi chema operaţii blocante din kernel thread;
- puteţi folosi spinlock-uri, dar, dacă durata de ţinere a lock-ului este mare, se recomandă folosirea de semafoare.

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

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

Datorită limitărilor existente, cea mai mare parte din implementările funcţiilor ce rulează în kernel thread-uri folosesc acelaşi 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);
}
```

API Windows

Primitivele ce stau la baza acţiunilor amânabile în Windows sunt DPC-urile şi system thread-urile. **DPC**-urile rulează în **context întrerupere** şi sunt echivalentul **softirq**-urilor din Linux, în vreme ce **system thread**-urile sunt echivalentul **kernel thread**-urilor din Linux şi rulează în **context proces**.

DPC-uri

Deferred Procedure Call (DPC) este echivalentul în Windows a softirq-urilor. La fel ca şi în Linux, DPC-urile rulează din context întrerupere (la nivelul dispatch) şi, din această cauză, în cadrul funcţiilor de tratare a DPC-urilor nu se pot realiza apeluri blocante (sleep). Totuşi, un DPC rulează la un nivel IRQ (`DISPATCH_LEVEL`) mai mic decât cel la care rulează rutina de tratare a unui întreruperi (`DIRQ`). Astfel, există funcţii care nu pot fi apelate din rutina de tratare a unei întreruperi, dar pot fi apelate dintr-un DPC (IoCompleteRequest).

Pentru mai multe detalii despre nivelurile IRQ din Windows consultaţi Interrupt Request Levels şi Managing Hardware Priorities.

Un DPC, ca şi celelalte abstractizări din kernel-ul Windows, este reprezentat de un obiect (KDPC). Iniţializarea obiectului ce abstractizează DPC-urile (`KDPC`) se face cu funcţia KeInitializeDpc:

```
VOID
  KeInitializeDpc(
    IN PRKDPC  Dpc,
    IN PKDEFERRED_ROUTINE  DeferredRoutine,
    IN PVOID  DeferredContext
    );
```

Funcţia de tratare a DPC-urilor (DeferredRoutine) are următoarea signatură:

```
VOID
  DeferredRoutine(
    IN KDPC  *Dpc,
    IN PVOID  DeferredContext,
    IN PVOID  SystemArgument1,
    IN PVOID  SystemArgument2
    );
```

unde:

- `Dpc` reprezintă obiectul DPC;
- `DeferredContext` reprezintă argumentul trimis funcţiei de iniţializare;
- `SystemArgument1` şi `SystemArgument2` reprezintă argumentele transmise funcţiei ce planifică un DPC pentru o execuţie ulterioară.

Funcţia KeInsertQueueDpc se foloseşte pentru a planifica rularea unui DPC la un moment ulterior de timp

```
BOOLEAN
  KeInsertQueueDpc(
    IN PRKDPC  Dpc,
    IN PVOID  SystemArgument1,
    IN PVOID  SystemArgument2
    );
```

Cazul tipic de folosire a DPC-urilor este acela în care dorim să planificăm execuţia unor acţiuni pentru un moment ulterior de timp din rutina de tratare a unei întreruperi.

Timere

Obiectul ce defineşte un timer în Windows este KTIMER.

Pentru iniţializare apelaţi funcţia KeInitializeTimer:

```
VOID
  KeInitializeTimer(
    IN PKTIMER  Timer
    );
```

O dată iniţializat, un timer poate fi planificat cu ajutorul funcţiei KeSetTimer:

```
BOOLEAN
  KeSetTimer(
    IN PKTIMER  Timer,
    IN LARGE_INTEGER  DueTime,
    IN PKDPC  Dpc  OPTIONAL
    );
```

După cum se observă, pentru a rula o funcţie de tratare la sfârşitul timeout-ului, trebuie să transmitem funcţiei de planificare un DPC. DPC-ul trebuie să fie în prealabil iniţializat. Valoarea `DueTime` reprezintă timpul la care timerul trebuie să expire. Dacă valoarea este pozitivă, reprezintă un timp absolut. Dacă valoarea este negativă, reprezintă un timp relativ la timpul curent. Unitatea de masură pentru această valoare este "system time units", e.g. 100 nanosecunde.

Un timer poate fi oprit cu ajutorul funcţiei KeCancelTimer:

```
BOOLEAN
  KeCancelTimer(
    IN PKTIMER  Timer
    );
```

Funcţia poate fi chemată atât pentru timerele planificate, cât şi pentru cele neplanificate.

O secvenţă uzuală de iniţializare şi planificare a unui timer este următoarea:

```
#define SECONDS           1
#define SEC_TO_NANOSEC(s)  ((s) * 1000 * 1000 * 10)

static KTIMER timer;
static KDPC myDpc;
static LARGE_INTEGER systemDelay;

static VOID myDpcRoutine(KDPC* dpc, PVOID context, PVOID arg1, PVOID arg2);

KeInitializeTimer(&timer);
KeInitializeDpc(&myDpc, myDpcRoutine, NULL);
systemDelay.QuadPart = -SEC_TO_NANOSEC(SECONDS);
KeSetTimer(&timer, systemDelay, &myDpc);
```

iar pentru oprire se va apela:

```
KeCancelTimer(&timer);
```

Un timer este un obiect de sincronizare. Pentru a aştepta expirarea timer-ului se poate folosi funcţia KeWaitForSingleObject (Laboratorul 5). Pentru exemplul anterior, se va apela:

```
KeWaitForSingleObject(&timer, Executive, KernelMode, FALSE, NULL);
```

Locking DPC-uri

Pentru a sincroniza accesul între cod ce rulează în context proces şi un DPC se folosesc spinlock-uri. Codul protejat de un spinlock rulează la nivelul IRQ `DISPATCH_LEVEL`, iar un DPC tot la nivelul `DISPATCH_LEVEL`. Deoarece codul care rulează la un anumit nivel IRQ poate fi întrerupt doar de cod care ruleaza la un IRQ strict mai mare, nu este nevoie de operaţii suplimentare (cum ar fi dezactivarea funcţiilor amânabile) pentru sincronizare în acest caz.

System worker threads

Pentru planificarea acţiunilor amânabile care să ruleze în context proces se folosesc thread-uri sistem de tip worker. Acestea sunt corespondentul workqueue-urilor din Linux. Acest tip de funcţii amânabile sunt planificate dintr-un DPC pentru a executa operaţii care nu sunt permise in contextul acestuia.

O sarcină este definită de obiectul IO_WORKITEM, care se alocă cu ajutorul funcţiei IoAllocateWorkItem şi se eliberează cu ajutorul funcţiei IoFreeWorkItem:

```
PIO_WORKITEM IoAllocateWorkItem( IN PDEVICE_OBJECT  DeviceObject );
VOID  IoFreeWorkItem( IN PIO_WORKITEM  IoWorkItem );
```

Pentru a planifica sarcina se foloseşte funcţia IoQueueWorkItem:

```
VOID
    IoQueueWorkItem(
        IN PIO_WORKITEM  IoWorkItem,
        IN PIO_WORKITEM_ROUTINE  WorkerRoutine,
        IN WORK_QUEUE_TYPE  QueueType,
        IN PVOID  Context
    );
```

unde:

- `IoWorkItem` reprezintă sarcina alocată anterior cu ajutorul funcţiei `IoAllocateWorkItem`;
- `WorkerRoutine` este un pointer către rutina ce va rula sarcina;
- `QueueType` specifică tipul de thread sistem în contextul căruia va rula sarcina; poate fi `CriticalWorkQueue` pentru thread cu prioritate real-time sau `DelayedWorkQueue` pentru thread cu prioritate variabila; se recomandă folosirea tipului `DelayedWorkQueue`;
- `Context` este un pointer către date private trimise rutinei ce va rula sarcina.

Rutina WorkItem ce va rula sarcina are următorul prototip:

```
VOID WorkItem(IN PDEVICE_OBJECT  DeviceObject, IN PVOID  Context );
```

La iniţializarea sistemului, sunt create câteva thread-uri în cadrul procesului `System`, thread-uri care există pentru a executa sarcini în numele altor thread-uri. Sarcina planificată în modul descris mai sus va rula în contextul unui astfel de thread al sistemului, la nivelul IRQ `PASSIVE_LEVEL`.

O secvenţă uzuală de folosire a unei sarcini este următoarea:

```
struct my_device_data {
        PDEVICE_OBJECT DeviceObject;
        PIO_WORKITEM my_work;
};

VOID myWorkHandler(PDEVICE_OBJECT  DeviceObject, PVOID  Context ) {
        struct my_device_data *my_data = (struct my_device_data *) Context;
        /* do work */
        IoFreeWorkItem(my_data->my_work);
}

//...
        struct my_device_data *my_data;
        my_data->my_work = IoAllocateWorkItem(my_data->DeviceObject);
        IoQueueWorkItem(my_data->my_work, myWorkHandler, DelayedWorkQueue, my_data);
```

```
//...
```

După cum se poate observa, de obicei sarcina alocată înainte de planificare este dealocată în rutina care rulează sarcina.

System threads

Echivalentul kernel thread-urilor din Linux sunt system thread-urile în Windows.

Un system thread se creează cu ajutorul funcţiei PsCreateSystemThread:

```
NTSTATUS
  PsCreateSystemThread(
    OUT PHANDLE  ThreadHandle,
    IN ULONG  DesiredAccess,
    IN POBJECT_ATTRIBUTES  ObjectAttributes  OPTIONAL,
    IN HANDLE  ProcessHandle  OPTIONAL,
    OUT PCLIENT_ID  ClientId  OPTIONAL,
    IN PKSTART_ROUTINE  StartRoutine,
    IN PVOID  StartContext
    );
```

unde:

- `ThreadHandle` reprezintă handle-ul thread-ului creat;
- `DesiredAccess` modul de acces asupra thread-ului; se setează la `THREAD_ALL_ACCESS`;
- `ObjectAttributes`, `ProcessHandle`, `ClientId` se setează pe `NULL`;
- `StartRoutine` funcţia ce va rula în contextul kernel thread-ului;
- `StartContext` parametrul trimis funcţiei.

În momentul în care handle-ul thread-ului creat nu mai este folosit, acesta trebuie eliberat printr-un apel al funcţiei ZwClose.

Signatura funcţiei StartRoutine ce rulează în contextul kernel thread-ului este:

```
void  StartRoutine (void *StartContext);
```

Funcţia rulează în context proces, la nivel IRQ `PASSIVE_LEVEL`.

Pentru a termina un system thread, trebuie chemată funcţia PsTerminateSystemThread, din contextul system thread-ului ce dorim să îl terminăm:

```
NTSTATUS
  PsTerminateSystemThread(
    IN NTSTATUS  ExitStatus
    );
```

Întrucât şi system thread-urile sunt obiecte dispecer, un alt thread poate aştepta terminarea acestuia. Pentru a obţine un pointer la obiectul `KTHREAD` asociat unui system thread, un driver trebuie să apeleze ObReferenceObjectByHandle şi să îi transmită handle-ul obţinut în urma apelului funcţiei `PsCreateSystemThread`. După ce a terminat cu system thread-ul, driverul trebuie să apeleze funcţia ObDereferenceObject.

O secvenţă uzuală de creare şi terminare a unui system thread este următoarea:

```
struct my_device_data {
        PKTHREAD thread;
        ...
};

void  myThreadRoutine(void *StartContext) {
        int done = 0;
        while(!done) {
                /* wait for condition */
                /* do work */
        }
        PsTerminateSystemThread(STATUS_SUCCESS);
}

//...

        /* start thread */
        NTSTATUS status;
        HANDLE hthread;
        struct my_device_data *my_data;

        status = PsCreateSystemThread(&hthread, THREAD_ALL_ACCESS,
```

```
                        NULL, NULL, NULL, (PKSTART_ROUTINE) myThreadRoutine, my_data);
        if (!NT_SUCCESS(status))
                return status;
        ObReferenceObjectByHandle(hthread, THREAD_ALL_ACCESS, NULL,
                        KernelMode, (PVOID*) &my_data->thread, NULL);
        ZwClose(hthread);

//...

        /* stop thread */
        /* set end of thread condition / event */
        KeWaitForSingleObject(my_data->thread, Executive, KernelMode, FALSE, NULL);
        ObDereferenceObject(my_data->thread);

//...
```

Quiz

Pentru auto-evaluare, de preferat înainte de laborator, răspundeţi la întrebările din quiz.

Exerciţii

- Folosiţi arhiva de sarcini a laboratorului.
- Punctaj total: **11 puncte**
- Punctajul maxim se ia cu **10 puncte**. Bonusul poate recupera lipsa de activitate de la alte laboratoare.

Linux

- Folosiţi directorul `lin/` din arhiva de sarcini a laboratorului.
- Punctaj total: **5,5 puncte**
- Recomandăm folosirea unei console de lucru prin SSH (`ssh -l root spook.local`) şi o consolă cu netconsole (shortcut pe Desktop).

1. (**1 punct**) Creaţi un modul care să afişeze o dată la fiecare `TIMER_TIMEOUT` secunde un mesaj cu numărul de secunde care au trecut de la inserarea modulului. Modulul trebuie sa poată fi descărcat.
   - Porniţi de la fişierele din directorul `lin/1-timer/` din arhiva de sarcini a laboratorului.
   - **Hints:**
     - Urmăriţi secţiunile marcate cu `TODO 1` în scheletul de laborator.
     - Pentru afişare folosiţi `printk(LOG_LEVEL ... )`. Pentru a şterge mesajele afişate de comanda `dmesg` folosiţi `dmesg -c`.
     - Pentru a nu face mai multe apeluri ale comenzii `dmesg` folosiţi `watch -n1 'dmesg | tail'`
     - Nu uitaţi să faceţi un snapshot al maşinii virtuale înainte de inserarea modulului.
     - Parcurgeţi secţiunea Timere din laborator.
2. (**2 puncte**) Creaţi un modul care să afişeze informaţii despre procesul curent după **N** secunde de la primirea unei comenzi ioctl. N este dat ca parametru prin ioctl.
   - Porniţi de la fişierele din directorul `lin/2-3-deferred/kernel` din arhiva de sarcini a laboratorului.
   - **Hints:**
     - Urmăriţi secţiunile marcate cu `TODO 2` în scheletul de laborator.
   - **2.1.** Implementaţi următoarele operaţii ioctl:
     - `MY_IOCTL_TIMER_SET` pentru planificarea unui timer să ruleze dupa un număr de secunde primit ca argument de rutina `ioctl`.
     - `MY_IOCTL_TIMER_CANCEL` pentru dezactivarea timer-ului.
   - **Hints:**
     - Prima operaţie ioctl primeşte ca argument din user-space (din programul de test din `lin/deferred/user`) direct o valoare, nu un pointer. Pentru modalitatea de acces, revedeţi Laboratorul 4
     - Citiţi secţiunea Timere din laborator.
   - **2.2.** Afişaţi în rutina de tratare a timer-ului id-ul procesului curent (pid-ul) şi numele imaginii de executabil.
   - **Hints:**
     - Id-ul procesului curent îl puteţi afla din `current`pid, iar imaginea de executabil din `current`comm. Pentru detalii, revedeţi Laboratorul 2.
   - **2.3.** Activaţi şi dezactivaţi timer-ul prin apelul operaţiilor ioctl din user-space.
   - **Hints:**
     - Utilizaţi programul `lin/deferred/user/test` din arhiva de sarcini. Programul primeşte ca parametri în linie de comandă operaţia ioctl si parametrii acesteia (dacă e cazul). Rulaţi programul fără parametri pentru modul de utilizare.

    - Pentru a putea folosi device driver-ul din user-space, trebuie sa creaţi fişierul de tip caracter `/dev/deferred` folosind utilitarul `mknod`. Alternativ, puteţi rula script-ul `makenode` din `lin/deferred/kernel/`, care realizează aceste operaţii.
  - **2.4.** Ce proces este identificat drept proces curent la rularea rutinei de tratare a timer-ului? De ce?
3. (**2,5 puncte**) Modificaţi modulul de la punctul precedent pentru a aloca, în momentul afişării pid-ului procesului care se execută, o zonă de memorie folosind `GFP_KERNEL`.
  - Afişarea/alocarea se face după **N** secunde de la primirea unei comenzi ioctl. N este dat ca parametru prin `ioctl`.
  - Implementaţi operaţia `MY_IOCTL_TIMER_ALLOC` pentru a aloca o zonă de memorie de tip `GFP_KERNEL` după **N** secunde şi afişarea pid-ului şi comenzii.
  - Exerciţiul are valoare academică. Cerinţa de alocare de memorie de tip `GFP_KERNEL` are scopul de a "forţa" folosirea workqueues.
  - **Hints:**
    - Implementarea operaţiei ioctl este similară cu cea a operaţiei ioctl `MY_IOCTL_TIMER_SET` de la exerciţiul 2.
    - Folosiţi **workqueue** pentru alocarea memoriei şi afişarea detaliilor despre proces. Planificaţi sarcina din rutina de tratare a **timer**-ului.
    - În sarcina planificată:
      - alocaţi cu ajutorul funcţiei `kmalloc` o zonă de memorie folosind `GFP_KERNEL`,
      - afişaţi pid-ul procesului şi numele imaginii de executabil,
      - dealocaţi memoria alocată.
    - Folosiţi acelaşi **timer** ca şi cel de la modulul anterior, păstrând funcţionalitatea ambelor module.
    - Pentru a diferenţia funcţionalităţile în rutina de tratare a **timer**-ului, folosiţi un flag în structura device-ului.
    - Sincronizaţi accesul la acest flag cu ajutorul unui **spinlock**.
    - Pentru valorile pe care le poate lua flag-ul folosiţi constantele `TIMER_TYPE_ALLOC` şi `TIMER_TYPE_SET` definite în scheletul de cod. Pentru iniţializare folosiţi `TIMER_TYPE_NONE`.
    - Folosiţi **workqueue**-ul predefinit.
    - Citiţi secţiunea Workqueues şi secţiunea Timere din laborator.
    - Ce proces este identificat drept proces curent la rularea sarcinii? De ce?

Pentru acasă

1. Creaţi un modul care să folosească un kernel thread pentru afişarea id-ului procesului curent.
  - **1.1.** Creaţi şi porniţi kernel thread-ul la încărcarea modulului.
  - **Hints:**
    - Citiţi secţiunea Kernel threads din laborator.
  - **1.2.** Afişaţi id-ul procesului curent şi numele imaginii de executabil în rutina de execuţie a thread-ului.
  - **Hints:**
    - Id-ul procesului curent îl puteţi afla din `current->pid`, iar imaginea de executabil din `current->comm`. Pentru detalii, revedeţi Laboratorul 2
  - **1.3.** Aşteptaţi descărcarea modulului în rutina de execuţie a thread-ului.
  - **Hints:**
    - Se recomandă utilizarea unei cozi de aşteptare. Pentru modalitatea de utilizare a cozilor de aşteptare, revedeţi Laboratorul 4.
    - Pentru sincronizarea accesului la flag-ul asociat cozii de aşteptare folosiţi o variabila atomica. Pentru modul de utilizare al variabilelor atomice revedeţi Laboratorul 3.
  - **1.4.** La descărcarea modulului deblocaţi kernel thread-ul şi aşteptaţi terminarea lui.
  - **Hints:**
    - Puteţi folosi încă o coadă de aşteptare pentru a detecta terminarea kernel thread-ului.
  - **1.5.** Ce proces este identificat drept proces curent la rularea kernel thread-ului? De ce?

Windows

- Folosiţi directorul `win/` din arhiva de sarcini a laboratorului.
- Punctaj total: **5,5 puncte**

1. (**1 punct**) Creaţi un modul care să afişeze o dată la `TIMER_TIMEOUT` secunde un mesaj cu numărul de secunde care au trecut de la inserarea modulului. Modulul trebuie să poată fi descărcat.
  - Porniţi de la fişierele din directorul `win/1-timer/` din arhiva de sarcini a laboratorului.
  - **Hints:**
    - Urmăriţi secţiunile marcate cu `TODO 1` în scheletul de laborator.
    - Folosiţi un **timer** pe care îl rearmaţi din interiorul rutinei **DPC**.
    - Pentru afişare folosiţi `DbgPrint`.
    - Nu uitaţi să faceţi un snapshot al maşinii virtuale înainte de inserarea modulului.
    - Citiţi secţiunile DPC şi Timere din laborator.
    - Folosiţi funcţia KeGetCurrentIrql pentru a afişa IRQL curent.

    - Ce IRQL este în handler-ul de timerului?
2. (**2 puncte**) Creaţi un modul care să afişeze informaţii despre procesul curent după **N** secunde. Numărul de secunde **N** va fi trasmis ca argument al unei comenzi `ioctl`.
    - Porniţi de la fişierele din directorul `win/2-3-deferred/kernel/` din arhiva de sarcini a laboratorului.
    - **Hints:**
        - Urmăriţi secţiunile marcate cu `TODO 2` în scheletul de laborator.
    - **2.1.** Implementaţi următoarele operaţii ioctl:
        - `MY_IOCTL_TIMER_SET` pentru planificarea unui timer care să ruleze dupa un număr de secunde primit ca argument de rutina `ioctl`.
        - `MY_IOCTL_TIMER_CANCEL` pentru dezactivarea timer-ului.
    - **Hints:**
        - Prima operaţie ioctl primeşte ca argument din user-space (din programul de test din `win/1-2-deferred/user`) un pointer la o valoare de tip `long`.
        - Citiţi secţiunea Timere din laborator.
    - **2.2.** Afişaţi în rutina de tratare a timer-ului id-ul procesului curent.
    - **Hints:**
        - Id-ul procesului curent îl puteţi afla cu ajutorul funcţiei PsGetCurrentProcessId. Pentru detalii, revedeţi Laboratorul 2
    - **2.3.** Activaţi şi dezactivaţi timer-ul prin apelul operaţiilor ioctl din user-space.
    - **Hints:**
        - Utilizaţi programul `win/deferred/user/test` din arhiva de sarcini. Programul primeşte ca parametri în linie de comandă operaţia ioctl şi parametrii acesteia (dacă e cazul). Rulaţi programul fără parametri pentru a vedea modul de utilizare.
        - Folosiţi consola `Visual Studio 2008 Command Prompt` şi comanda `nmake` pentru a compila programul.
    - **2.4.** Ce proces este identificat drept proces curent la rularea rutinei de tratare a timer-ului? De ce?
    - **Hints:**
        - Verificaţi în `Task Manager` ce proces corespunde id-ului afişat.
3. (**2,5 puncte**) Modificaţi modulul de la punctul precedent pentru a aloca, în momentul afişării pid-ului procesului care se execută, o zonă de memorie (un întreg).
    - Afişarea/alocarea se face după **N** secunde de la primirea unei comenzi ioctl. N este dat ca parametru prin `ioctl`.
    - Exerciţiul are valoare academică. Cerinţa de alocare de memorie are scopul de a "forţa" folosirea workqueues.
    - **Hints:**
        - Urmăriţi secţiunile marcate cu `TODO 3` în scheletul de laborator.
    - **3.1.** Implementaţi următoarea operaţie `ioctl`:
        - `MY_IOCTL_TIMER_ALLOC` pentru a aloca o zonă de memorie de tip `PagedPool` după o secundă.
    - **Hints:**
        - Implementarea operaţiei ioctl este similară cu cea a operaţiei ioctl `MY_IOCTL_TIMER_SET` de la exerciţiul 2.
        - Folosiţi acelaşi timer ca şi cel de la modulul anterior, pastrând funcţionalitatea ambelor module. Pentru a diferenţia funcţionalităţile în rutina de tratare a timer-ului, folosiţi un flag. Sincronizaţi accesul la acest flag cu ajutorul funcţiilor `Interlocked*()`; acestea au fost prezentate în Laboratorul 3.
        - Pentru valorile pe care le poate lua flag-ul folosiţi constantele `TIMER_TYPE_ALLOC` şi `TIMER_TYPE_SET` definite în scheletul de cod. Pentru iniţializare folosiţi `TIMER_TYPE_NONE`.
        - Citiţi secţiunea Timere din laborator.
    - **3.2.** Folosiţi system worker threads pentru alocarea memoriei. Planificaţi sarcina din rutina de tratare a timer-ului.
    - **Hints:**
        - Citiţi secţiunea System worker threads din laborator.
    - **3.3.** În rutina de execuţie a sarcinii alocaţi cu ajutorul funcţiei ExAllocatePoolWithTag o zonă de memorie în care să păstraţi id-ul procesului curent. Afişaţi valoarea alocată. Dealocaţi memoria alocată.
    - **Hints:**
        - Id-ul procesului curent îl puteţi afla cu ajutorul funcţiei PsGetCurrentProcessId. Pentru detalii, revedeţi Laboratorul 2
    - **3.4.** Ce proces este identificat drept proces curent la rularea sarcinii? De ce?
    - **Hints:**
        - Verificaţi în `Task Manager` ce proces corespunde id-ului afişat.
    - **3.5.** Puteaţi face alocarea de memorie direct din rutina de tratare a timer-ului? De ce?

Pentru acasă

1. Creaţi un modul care să folosească un system thread pentru afişarea id-ului procesului curent.
    - **1.1.** În rutina `DriverEntry` creaţi system thread-ul.
    - **Hints:**
        - Citiţi secţiunea System threads din laborator.
    - **1.2.** Afişaţi id-ul procesului curent în rutina de execuţie a thread-ului.
    - **Hints:**
        - Id-ul procesului curent îl puteţi afla cu ajutorul funcţiei PsGetCurrentProcessId. Pentru detalii, revedeţi Laboratorul 2

- **1.3.** Aşteptaţi descărcarea modulului în rutina de execuţie a thread-ului.
- **Hints:**
  - Se recomandă utilizarea unui eveniment. Pentru modalitatea de utilizare a evenimentelor, revedeţi Laboratorul 5.
- **1.4.** În rutina `DriverUnload` deblocaţi system thread-ul şi aşteptaţi terminarea lui.
- **1.5.** Ce proces este identificat drept proces curent la rularea system thread-ului? De ce?

Soluţii

Soluţii exerciţii laborator 7

Resurse utile

Linux

1. Linux Device Drivers, 3rd ed., Ch. 7: Time, Delays, and Deferred Work
2. Scheduling Tasks
3. Driver porting: the workqueue interface
4. Workqueues get a rework
5. Kernel threads made easy
6. Unreliable Guide to Locking

Windows

1. The Windows 2000 Device Driver Book, Second Edition ? Chapter 14. System Threads
2. Programming the Microsoft Windows Driver Model, Second Edition ? Chapter 4. Synchronization - Kernel Dispatcher Objects, Chapter 14. Specialized Topics ? System Threads, Work Items
3. Microsoft Windows Internals, Fourth Edition - System Worker Threads
4. DPC Objects and DPCs
5. Timer Objects and DPCs
6. System Worker Threads
7. Device-Dedicated Threads
8. Managing Hardware Priorities
9. Interrupt Request Levels

[1)] jiffie ? începând cu 2.6.21, arhitectura tratării timerelor a fost complet rescrisă; se aşteaptă ca in viitor şi jiffie-ul să dispară. Pentru detalii consultaţi Clockevents and dyntick şi Documentation/timers/hrtimers.txt
[2)] read-copy update ? un mecanism prin care operaţile distructive (ex: ştergerea unui element dintr-o listă înlănţuită) se fac în două etape: eliminarea referinţelor către datele de şters şi ştergerea propriu-zisă, care se face numai după ce este sigur că nimeni nu le mai foloseşte. Avantajul este că accesarea datelor se poate face fără sincronizare. Pentru mai multe informaţii citiţi documentaţia RCU din kernelul Linux
[3)] jiffie ? începând cu 2.6.21, arhitectura tratării timerelor a fost complet rescrisă; se aşteaptă ca în viitor şi jiffie-ul să dispară. Pentru detalii consultaţi Clockevents and dyntick şi Documentation/timers/hrtimers.txt
[4)] spinlock ? pentru mai multe detalii despre folosirea spinlock-urilor, revedeţi Laboratorul 3
[5)] container_of ? un exemplu de utilizare pentru macrodefiniţia `container_of` este la parcurgerea listelor din kernel