Laborator 3 - Kernel API

Obiectivele laboratorului

- familiarizarea cu API-ul de bază pentru nucleul Linux şi nucleul Windows
- descrierea mecanismelor de alocare a memoriei
- descrierea mecanismelor de locking

Cuvinte cheie

- contexte de executie
- printk
- kmalloc / kfree
- list_head
- spinlock_t
- struct semaphore
- atomic_t
- IoWriteErrorLogEntry
- ExAllocatePool / ExFreePool
- ExAllocatePoolWithTag / ExFreePoolWithTag
- LIST_ENTRY
- KSPIN_LOCK
- KSEMAPHORE
- Interlocked*

Materiale ajutătoare

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

Noţiuni generale

În cadrul laboratorului curent se prezintă un set de concepte şi funcţii de bază necesare programării kernel. Este important de reţinut faptul că programarea kernel diferă extrem de mult faţă de programarea în user-space. Kernel-ul este o entitate de sine stătătoare, care nu poate folosi bibliotecile din user-space (nici chiar `libc` în Linux sau `kernel32.dll` în Windows). Drept urmare, funcţiile uzuale utilizate în user-space (`printf`, `malloc`, `free`, `open`, `read`, `write`, `memcpy`, `strcpy` etc.) nu mai pot fi folosite. În concluzie, programarea kernel se bazează pe un API total nou şi independent, ce nu are legătură cu API-ul din user-space, fie că ne referim la POSIX, Win32 sau ANSI C (funcţiile standard de bibliotecă pentru limbajul C).

Accesarea memoriei

O diferenţă importantă în programarea kernel este modul de accesare şi alocare a memoriei. Datorită faptului că programarea kernel se face la un nivel foarte aproape de maşina fizică, există reguli importante în ceea ce priveşte gestiunea memoriei. În primul rând, se lucrează cu mai multe tipuri de memorie:

- memorie fizică
- memorie virtuală din spaţiul de adresare kernel
- memorie virtuală din spaţiul de adresare al unui proces
- memorie rezidentă ? ştim sigur că paginile accesate sunt prezente în memoria fizică

Memoria virtuală din spaţiul de adresare al unui proces nu poate fi considerată rezidentă datorită mecanismelor de memorie virtuală implementate de sistemul de operare: paginile pot fi în swap, sau pur şi simplu pot să nu fie prezente în memoria fizică datorită mecanismului de demand paging. Memoria din spaţiul de adresare kernel poate fi rezidentă sau nu. Atât segmentele de date şi cod ale unui modul cât şi stiva kernel a unui proces sunt rezidente (în Windows, dacă se doreşte,

şi acestea se pot swapa). Memoria dinamică poate fi sau nu rezidentă, în funcţie de modul în care se alocă.

Atunci când se lucrează cu memorie rezidentă lucrurile sunt simple: memoria se poate accesa oricând. Dacă se lucrează însă cu memorie nerezidentă, atunci aceasta se poate accesa doar din anumite contexte. Memoria nerezidentă se poate accesa doar din context proces. Accesarea memoriei nerezidente din context întrerupere are rezultate impredictibile, şi din această cauză, atunci când sistemul de operare detectează un astfel de access, va lua măsuri drastice: blocarea sau resetarea sistemului, pentru a preveni coruperi grave.

Memoria virtuală a unui proces nu se poate accesa direct din kernel. În general este descurajată total accesarea spaţiului de adresă al unui process, dar există situaţii în care un device driver trebuie să o facă. Cazul tipic este cel în care device driver-ul trebuie să acceseze un buffer din user-space. În acest caz, device driverul va trebuie sa folosească funcţii speciale, şi nu să acceseze direct bufferul. Acest lucru este necesar pentru a preveni accesarea unor zone invalide de memorie.

O altă diferenţa fată de programarea din userspace, relativ la lucrul cu memoria este datorată stivei; stiva a cărei dimensiune este fixă şi limitată. În kernel-ul Linux 2.6.x se foloseşte implicit o stivă de `4K`, iar în Windows se foloseşte o stivă de `12K`. Din această cauză vor trebui evitate alocarea unor structuri de mari dimensiuni pe stivă sau folosirea apelurilor recursive.

Contexte de execuţie

Relativ la modul de execuţie în kernel, distingem două contexte: context **proces** şi context **întrerupere**. Ne aflăm în context proces atunci când rulăm cod ca urmare a unui apel de sistem sau când rulăm în contextul unui kernel thread. Atunci când rulăm în rutina de tratare a unei întreruperi sau a unei acţiuni amânabile, rulăm în context întrerupere.

Unele dintre apelurile din API-ul kernel pot duce la blocarea procesului curent. Exemple comune sunt folosirea unui semafor sau aşteptarea unei condiţii. În acest caz, procesul este trecut în starea `WAITING` şi alt proces este rulat. O situaţie interesantă apare în momentul în care o funcţie ce poate duce la suspendarea procesului curent este chemată din context întrerupere. În acest caz, nu există un proces curent, şi din această cauză rezultatele sunt impredictibile. De câte ori sistemul de operare detectează această condiţie va genera o condiţie de eroare care va duce la oprirea sistemului de operare.

Locking

Una dintre cele mai importante caracteristici ale programării în kernel este paralelismul. Atât Linux cât şi Windows suportă sisteme SMP, cu mai multe procesoare, dar şi preemptivitate în kernel. Acest lucru face programarea kernel mai dificilă, deoarece accesul la variabilele globale trebuie sincronizat, fie cu primitive de spinlock, fie cu primitive blocante. Deşi este recomandat să se folosească primitive blocante, acestea nu pot fi folosite în context întrerupere, aşa că singura soluţie de locking în context întrerupere sunt spinlock-urile.

Spinlock-urile sunt folosite pentru realizarea excluderii mutuale. Atunci când nu pot obţine accesul la regiunea critică nu suspendă procesul curent, ci folosesc mecanismul de busy-waiting (aşteaptă într-un ciclu while eliberarea lock-ului). Codul care se execută în regiunea critică protejată de un spinlock nu are voie să suspende procesul curent (trebuie să respecte condiţiile execuţiei în context întrerupere). Mai mult, nu se va ceda procesorul decât pentru servirea întreruperilor. Datorită mecanismului folosit, este important ca un spinlock să fie deţinut cât mai puţin timp posibil.

Preemptivitate

Atât Linux cât şi Windows folosesc kernele preemptive. Nu trebuie confundată noţiunea de multitasking preemptiv cu noţiunea de kernel preemptiv. Noţiunea de multitasking preemptiv se referă la faptul că sistemul de operare întrerupe rularea unui proces în mod forţat, atunci când acestuia i-a expirat cuanta de timp şi rulează în user-space, pentru a rula alt proces. Un kernel este preemptiv dacă un proces ce rulează în kernel-mode (ca urmare a unui apel de sistem) poate fi întrerupt pentru a rula un alt proces.

Datorită preemptivităţii, atunci când partajăm resurse între două porţiuni de cod ce pot rula din contexte proces diferite, trebuie să ne protejăm cu primitive de sincronizare, chiar şi în cazul uni-procesor.

Linux Kernel API

Convenţie indicare erori

În Linux kernel convenţia folosită la apelul funcţiilor pentru a indica succes este identică cu cea din programarea UNIX: 0

pentru success, sau o valoare diferită de 0 pentru insucces. Pentru insucces se returnează valori negative, aşa cum este prezentat în exemplul de mai jos:

```
if (alloc_memory() != 0)
        return -ENOMEM;

if (user_parameter_valid() != 0)
        return -EINVAL;
```

Lista exhaustivă a erorilor şi o sumară explicaţie găsiţi în include/asm-generic/errno-base.h şi include/asm-generic/ernno.h.

şiruri de caractere

În Linux, programatorului de kernel i se pun la dispoziţie funcţiile uzuale de lucru pe şiruri: `strcpy`, `strncpy`, `strlcpy`, `strcat`, `strncat`, `strlcat`, `strcmp`, `strncmp`, `strnicmp`, `strchr`, `strnchr`, `strrchr`, `strstr`, `strlen`, `memset`, `memcpy`, `memmove`, `memscan`, `memcmp`, `memchr`. Aceste funcţii sunt declarate în headerul include/linux/string.h şi sunt implementate în kernel.

printk

Echivalentul `printf` în kernel este `printk`, definit în include/linux/kernel.h. Sintaxa printk seamănă foarte mult cu cea a printf. Primul parametru al printk decide categoria de mesaje în care se încadrează mesajul curent:

```
#define KERN_EMERG   "<0>"  /* system is unusable */
#define KERN_ALERT   "<1>"  /* action must be taken immediately */
#define KERN_CRIT    "<2>"  /* critical conditions */
#define KERN_ERR     "<3>"  /* error conditions */
#define KERN_WARNING "<4>"  /* warning conditions */
#define KERN_NOTICE  "<5>"  /* normal but significant condition */
#define KERN_INFO    "<6>"  /* informational */
#define KERN_DEBUG   "<7>"  /* debug-level messages */
```

Astfel, un mesaj în kernel de tip warning ar fi trimis cu:

```
printk(KERN_WARNING "my_module input string %s\n", buff);
```

În cazul în care nivelul de logging lipseşte din apelul printk, se realizeaza logging cu nivelul implicit de la momentul apelului. Un lucru ce trebuie reţinut este că mesajele trimise cu printk sunt vizibile doar pe consolă [1] şi doar dacă nivelul lor depaseşte nivelul implicit setat pe consolă [2].

Alocare memorie

În Linux se poate aloca doar memorie **rezidentă**, cu ajutorul apelului kmalloc. Un apel tipic kmalloc este prezentat în continuare:

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

if (!(string = kmalloc (string_len+1, GFP_KERNEL))) {
        //report error: -ENOMEM;
}
```

După cum se observă, primul parametru indică dimensiunea în octeţi a zonei de alocat. Funcţia întoarce un pointer către o zonă de memorie ce poate fi folosită direct în kernel, sau `NULL` dacă nu s-a putut aloca memorie. Cel de-al doilea parametru specifică modul în care se doreşte să se facă alocarea, iar cele mai folosite valori sunt:

- `GFP_KERNEL` - folosirea acestei valori poate duce la suspendarea procesului curent; nu poate fi deci folosit în context de întrerupere;
- `GFP_ATOMIC` - atunci când se foloseşte această valoare se garantează ca funcţia kmalloc nu suspendă procesul curent; poate fi folosită oricând.

Complementara funcţiei `kmalloc` este kfree, funcţie ce primeşte ca argument o zonă alocată de `kmalloc`. Această funcţie nu suspendă procesul curent şi, în consecinţă, poate fi apelată din orice context.

Liste

Pentru că listele înlănţuite sunt deseori folosite, Linux kernel API pune la dispoziţie o modalitate unitară de definire şi folosire a listelor. Aceasta implică folosirea unui element de tipul `struct list_head` în cadrul structurii pe care vrem să o folosim ca element al unei liste. Structura `list_head` este definită în /include/linux/list.h alături de toate celelalte

funcţii ce lucrează pe liste. Codul următor arată definiţia structurii `list_head` şi folosirea unui element din acest tip într-o altă structură bine cunoscută din kernelul de Linux:

```
struct list_head {
    struct list_head *next, *prev;
};

struct task_struct {
    ...
    struct list_head children;
    ...
};
```

Rutinele uzuale pentru lucrul cu liste sunt următoarele:

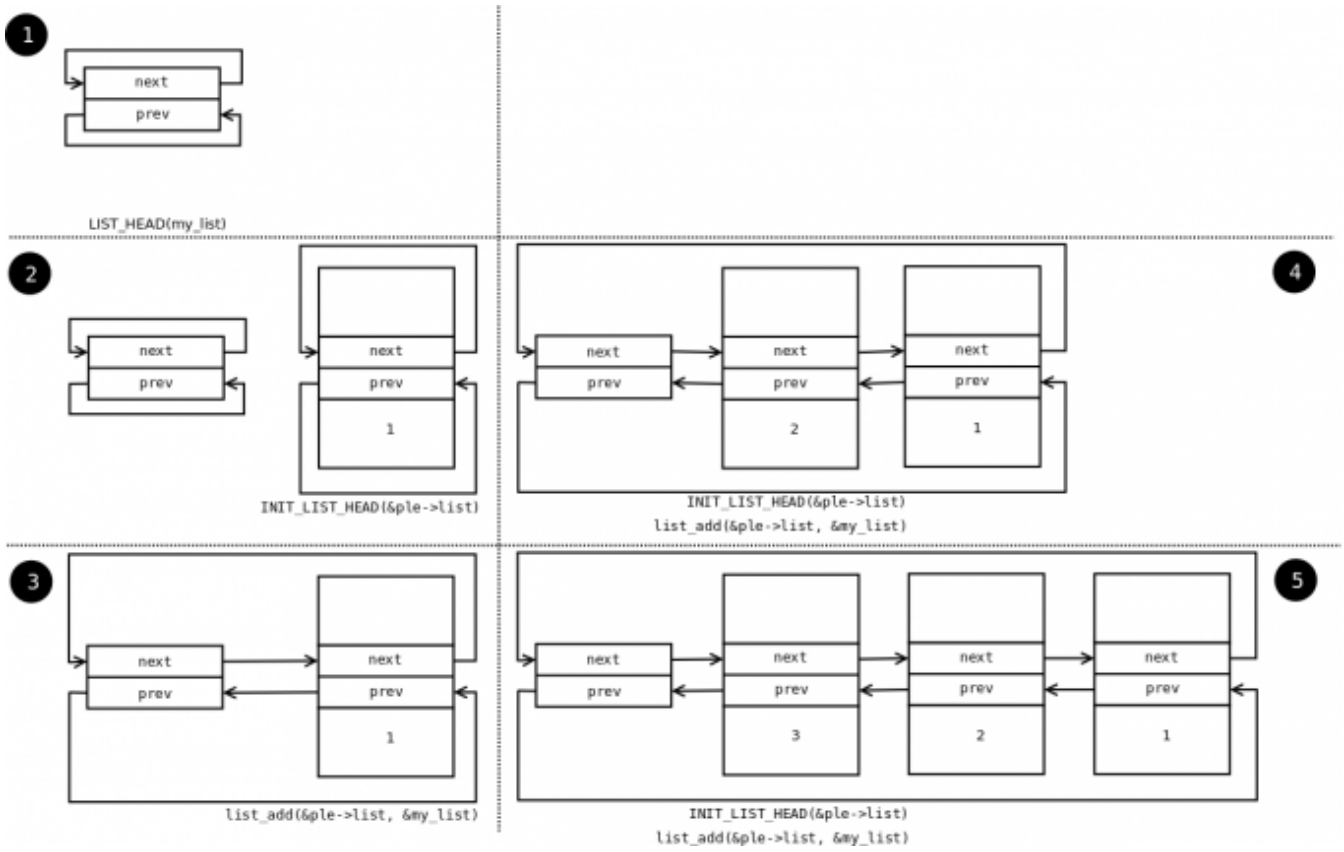
- `LIST_HEAD(name)` este folosit pentru a declara santinela unei liste
- `INIT_LIST_HEAD(struct list_head *list)` se foloseşte pentru a iniţializa un element de tipul `struct list_head` prin setarea valorii câmpurilor `next` şi `prev` la `list`.
- `list_add(struct list_head *new, struct list_head *head)` adaugă elementul `new` după elementul `head`.
- `list_del(struct list_head *entry)` şterge elementul ce conţine `entry` din lista din care face parte.
- `list_entry(ptr, type, member)` întoarce stuctura de tip `type` care conţine elementul `ptr` din listă cu numele `member` în cadrul structurii.
- `list_for_each(pos, head)` iterează o listă, folosind `pos` drept cursor.
- `list_for_each_safe(pos, n, head)` iterează o listă, folosind `pos` drept cursor şi `n` cursor temporar. Acest macro este folosit în cazul în care se doreşte ştergerea unui element din listă.

Următorul cod arată modul de folosire al acestor rutine:

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

struct pid_list {
    pid_t pid;
    struct list_head list;
};

LIST_HEAD(my_list);

static int add_pid(pid_t pid)
{
        struct pid_list *ple = kmalloc(sizeof *ple, GFP_KERNEL);

        if (!ple)
                return -ENOMEM;

        INIT_LIST_HEAD(&ple->list);
        ple->pid = pid;
        list_add(&ple->list, &my_list);

        return 0;
}

static int del_pid(pid_t pid)
{
        struct list_head *i, *tmp;
        struct pid_list *ple;

        list_for_each_safe(i, tmp, &my_list) {
                ple = list_entry(i, struct pid_list, list);
                if (ple->pid == pid) {
                        list_del(i);
                        kfree(ple);
                        return 0;
                }
        }

        return -EINVAL;
}

static void destroy_list(void)
{
        struct list_head *i, *n;
        struct pid_list *ple;

        list_for_each_safe(i, n, &my_list) {
                ple = list_entry(i, struct pid_list, list);
                list_del(i);
                kfree(ple);
```

```
        }
}
```

Evoluţia listei poate fi văzută în următoarea figură:



Se observă comportamentul de tip stivă introdus de macro-ul `list_add` precum şi folosirea unei santinele.

Din exemplul de mai sus se observă că modalitatea de definire şi folosire a unei liste (dublu înlănţuite) este generică şi în acelaşi timp nu introduce un overhead suplimentar. Structura `list_head` este folosită pentru a menţine legăturile între elementele listei. Se observă, de asemenea, că iterarea prin listă se face tot cu ajutorul acestei structuri, iar obţinerea elementelor din listă se face cu ajutorul `list_entry`. Această idee de implementare şi folosire a unei liste nu este nouă, ea fiind descrisă în `The Art of Computer Programming` de Donald Knuth în anii '80.

Mai multe funcţii şi macrodefiniţii de lucru cu liste kernel sunt prezentate şi explicate în headerul include/linux/list.h.

Locking

Spinlock-uri

`spinlock_t` (definit in linux/spinlock.h) este tipul de bază ce implementează conceptul de spinlock în Linux. El descrie un spinlock, iar operaţiile asociate cu un spinlock sunt `spin_lock_init`, `spin_lock`, `spin_unlock`. Un exemplu de utilizare este prezentat mai jos:

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

DEFINE_SPINLOCK(lock1);
spinlock_t lock2;

spin_lock_init(&lock2);

spin_lock(&lock1);
/* critical region */
spin_unlock(&lock1);

spin_lock(&lock2);
/* critical region */
spin_unlock(&lock2);
```

În Linux se pot folosi spinlock-uri de tip read/write, utile în probleme de genul cititori-scriitor. Aceste tipuri de lockuri sunt

identificate de `rwlock_t`, iar funcţiile cu care se poate opera asupra unui spinlock de tip read/write sunt `rwlock_init`, `read_lock`, `write_lock`. Un exemplu de utilizare:

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

DEFINE_RWLOCK(lock);

struct pid_list {
    pid_t pid;
    struct list_head list;
};

int have_pid(struct list_head *lh, int pid)
{
    struct list_head *i;
    void *elem;

    read_lock(&lock);
    list_for_each(i, lh) {
        struct pid_list *pl = list_entry(i, struct pid_list, list);
        if (pl->pid == pid) {
            read_unlock(&lock);
            return 1;
        }
    }
    read_unlock(&lock);

    return 0;
}

void add_pid(struct list_head *lh, struct pid_list *pl)
{
    write_lock(&lock);
    list_add(&pl->list, lh);
    write_unlock(&lock);
}
```

Semafoare

Un semafor este reprezentat de o variabila de tipul `struct semaphore` (definit in linux/semaphore.h). Functiile si macro-urile pentru lucrul cu semafoare sunt prezentate in continuare:

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

/* functii pentru initializarea semaforului */
void sema_init(struct semaphore *sem, int val);
DECLARE_MUTEX(name);
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);

/* functii pentru achizitionarea semaforului */
void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
int down_trylock(struct semaphore *sem);

/* functie pentru eliberarea semaforului */
void up(struct semaphore *sem);
```

Functia `down` decrementeaza valoarea semaforului si se blocheaza pana cand aceasta devine iar nenegativa. Functia `down_interruptible` face acelasi lucru numai ca operatia poate fi intrerupta. Se recomanda testarea de fiecare data a valorii intoarse de aceasta functie deoarece o valoare diferita de 0 inseamna ca operatia a fost intrerupta si apelantul nu a obtinut semaforul. Functia `down_trylock` este varianta neblocanta pentru achizitionarea unui semafor, daca nu se poate lua semaforul se intoarce o valoare diferita de 0.

Trebuie reţinut faptul că nu este permisa o operatie down blocanta in context de intrerupere sau intr-o regiune in care se detine un spinlock (context atomic).

Dupa apelul funcţiei `up` apelantul nu mai deţine semaforul.

Variabile atomice

De multe ori este nevoie doar de sincronizarea accesului la o variabila simpla, de exemplu un contor. Pentru aceasta se poate folosi o variabila de tip `atomic_t` (definit in include/asm-generic/atomic.h) care tine o valoare intreaga. Mai jos sunt prezentate unele operatii care pot fi efectuate asupra unei variabile `atomic_t`:

```
#include <asm/atomic.h>

void atomic_set(atomic_t *v, int i);
int atomic_read(atomic_t *v);
void atomic_add(int i, atomic_t *v);
void atomic_sub(int i, atomic_t *v);
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
```

Operatii atomice pe biti

Kernelul pune la dispozitie un set de functii (in include/linux/bitops.h) care modifica sau testeaza biti in mod atomic.

```
#include <asm/bitops.h>

void set_bit(int nr, void *addr);
void clear_bit(int nr, void *addr);
void change_bit(int nr, void *addr);
int test_and_set_bit(int nr, void *addr);
int test_and_clear_bit(int nr, void *addr);
int test_and_change_bit(int nr, void *addr);
```

`addr` reprezinta adresa zonei de memorie ai carei biti se modifica sau testeaza, iar `nr` reprezinta asupra carui bit din aceasta zona se efectueaza operatia.

Windows Kernel API

Convenţie indicare erori

În Windows kernel convenţia folosită la apelul funcţiilor pentru a indica succes este următoarea: se întoarce `STATUS_SUCCESS` pentru succes şi o valoare diferită de `STATUS_SUCCESS` pentru insuccess. Aceste coduri pentru succes sau diferite erori sunt definite în ntstatus.h. Semnificaţia codurilor de eroare este explicată în NTSTATUS values din MSDN.

şiruri de caractere (ASCII, Unicode)

În Windows, atunci când se lucrează cu funcţii uzuale de lucru pe şiruri programatorul are două posibilităţi: foloseşte apelurile clasice (strcpy, strncpy, strcat, etc.) sau foloseşte apeluri specifice programării kernel (RtlStringCbCat, RtlStringCbCopy, RtlStringCbPrintf, etc.). Folosirea primei variante are avantajul simplităţii dar introduce un overhead, pentru că funcţiile folosite vor fi legate în modul. Atunci când se foloseşte cea de a doua variantă, în afară de faptul că dispare overhead-ul prezentat anterior, mai avem un avantaj: se pot folosi atât şiruri ASCII cât şi Unicode.

În Windows, pentru a obţine efecte similare cu `printf`, trebuie folosită funcţa IoWriteErrorLogEntry, al cărei mod de folosire a fost prezentat sumar într-un exemplu în laboratorul trecut. Rolul acestei funcţii este să trimită un mesaj în user-space, unde acest mesaj va fi logat de serviciul `Event Viewer`.

Din păcate lucrul cu această funcţie (şi cu alte funcţii de altfel) este îngreunat de faptul că şirurile de caractere aşteptate sunt Unicode şi nu ASCII. Un sir unicode poate fi generat de compilator folosind prefixul `L`, astfel încât `L"string"` generează un sir Unicode, iar `"string"` generează un sir ASCII. Tipul de date al unui caracter unicode este `WCHAR` sau `wchar_t`, iar un sir Unicode este de tip `wchar_t *` sau `PWSTR`. Constanta de terminare a şirului este `UNICODE_NULL` (16 biţi de zero).

Există o structură predefinită în scopul lucrului mai uşor cu sirurile Unicode: `UNICODE_STRING`. La fel, kernel-ul pune la dispoziţie nişte funcţii de lucru cu aceste siruri care să înlocuiască funcţiile din biblioteca standard C. Funcţiile de manipulare de siruri Unicode includ: RTLInitUnicodeString, RtlAnsiStringToUnicodeSize şi RtlAnsiStringToUnicodeString, RtlIntegerToUnicodeString, RtlAppendUnicodeStringToString şi RtlCopyUnicodeString, RtlUpcaseUnicodeString, RtlCompareUnicodeString şi RtlEqualUnicodeString.

Spre exemplu, următoarea funcţie transformă un şir de caractere într-un şir unicode:

```
UNICODE_STRING* TO_UNICODE(const char *str, UNICODE_STRING *unicodeStr) {
    ANSI_STRING ansiStr;

    RtlInitAnsiString(&ansiStr, str);
    if (RtlAnsiStringToUnicodeString(unicodeStr, &ansiStr, TRUE)
                                    != STATUS_SUCCESS)
```

```
        return NULL;
    return unicodeStr;
}
```

Alocare memorie

Alocarea şi dezalocarea memoriei în Windows kernel se face cu ExAllocatePool, respectiv ExFreePool. Aceste funcţii alocă/dezalocă atât memorie **rezidentă** cât şi **nerezidentă**, la cerere. Un exemplu de folosire a acestor funcţii este prezentat mai jos:

```
#include <ntddk.h>

struct my_struct *ptr;

ptr = ExAllocatePool(NonPagedPool, sizeof(struct my_struct));
if (!ptr) {
        DbgPrint("Out of memory.\n");
        return STATUS_NO_MEMORY;
}
//...
ExFreePool(ptr);
```

După cum se observă, primul parametru specifică tipul de memorie de alocat: `NonPagedPool` pentru memorie rezidentă, `PagedPool` pentru memorie swapabilă. Pentru a uşura depanarea problemelor cauzate de leak-uri de memorie se pot folosi funcţiile ExAllocatePoolWithTag şi ExFreePoolWithTag care primesc un parametru suplimentar: un întreg pe 32 de biţi. În general acest întreg este scris sub forma `1gat` pentru a uşura procesul de debug (pentru exemplul dat, la debug vom observa că memoria care nu a fost dezalocată are un tag de tipul `tag1` [3]:

```
#include <ntddk.h>

struct my_struct *ptr;

ptr = ExAllocatePoolWithTag(NonPagedPool, sizeof(struct my_struct), '1gat');
if (!ptr) {
        DbgPrint("Out of memory.\n");
        return STATUS_NO_MEMORY;
}
//...
ExFreePoolWithTag(ptr, '1gat');
```

Funcţiile `ExAllocatePool` şi `ExFreePool` au fost declarate deprecated astfel încât se recomandă folosirea variantelor cu Tag.

Liste

şi în Windows kernel există construcţii şi funcţii ajutătoare pentru lucrul cu listele, urmând acelaşi model de implementare ca pe Linux. Un exemplu de folosire a acestora este prezentat in continuare:

```
#include <ntddk.h>

struct pid_list {
        SINGLE_LIST_ENTRY lh;
        HANDLE pid;
};

SINGLE_LIST_ENTRY my_list = { NULL };

static NTSTATUS add_pid(HANDLE pid)
{
        struct pid_list *ple;

        if (!(ple = ExAllocatePoolWithTag(NonPagedPool, sizeof(*ple), 'lp1t')))
                return STATUS_NO_MEMORY;
        ple->pid = pid;
        ple->lh.Next = NULL;
        PushEntryList(&my_list, &ple->lh);
        return STATUS_SUCCESS;
}

static NTSTATUS del_pid(HANDLE pid)
{
        SINGLE_LIST_ENTRY *i, *j;
        struct pid_list *ple = NULL;
        NTSTATUS ret = STATUS_INVALID_PARAMETER;

        for(j = &my_list, i = my_list.Next; i; j = i, i = i->Next) {
```

```
                /* e OK sa fie i - primul element e doar head */
                ple = CONTAINING_RECORD(i, struct pid_list, lh);
                if (ple->pid == pid) {
                        PopEntryList(j);
                        ret = STATUS_SUCCESS;
                        break;
                }
        }

        if (ret == STATUS_SUCCESS)
                ExFreePoolWithTag(ple, 'lp1t');
        return ret;
}


static void destroy_list(void)
{
        SINGLE_LIST_ENTRY *i, *j;
        struct pid_list *ple = NULL;
        j = NULL;

        for(i = my_list.Next; i; i = j)
        {
                ple = CONTAINING_RECORD(i, struct pid_list, lh);
                j = i->Next; /* salveaza pointerul catre elementul urmator pentru parcurgere */

                PopEntryList(&my_list);
                ExFreePoolWithTag(ple, 'lp1t');
        }
}
```

După cum se observă din exemplu, lista înlănţuită este una simplu înlănţuită, şi nu dublu înlănţuită. Daca se doreste folosirea unei liste dublu inlantuite se foloseste `LIST_ENTRY`. În rest modalitatea de folosire şi implementare a listei generice este similară, urmând acealaşi model al lui Donald Knuth.

Locking

Spinlock

Un spinlock este definit de `KSPIN_LOCK`, iar funcţiile de lock/unlock sunt KeInitializeSpinLock, KeAcquireSpinLock, KeReleaseSpinlock. Funcţiile `KeAcquireInStackQueuedSpinLock`, `KeReleaseInStackQueuedSpinLock` sunt variantele mai performante pentru sistemele multiprocesor. Un exemplu de folosire:

```
#include <ntddk.h>

KSPIN_LOCK lock;
KIRQL      irql;

KeInitializeSpinLock(&lock);

KeAcquireSpinLock(&lock, &irql);
/* critical region */
KeReleaseSpinLock(&lock, irql);
```

Semafoare

In Windows kernel un semafor este reprezentat de o variabila de tipul `KSEMAPHORE` care se initializeaza cu functia KeInitializeSemaphore. Pentru achizitionarea semaforului se folosesc functiile KeWaitForSingleObject si KeWaitForMultipleObjects, iar pentru eliberarea semaforului se foloseste functia KeReleaseSemaphore. De asemenea se poate afla valoarea unui semafor folosind functia KeReadStateSemaphore.

Un exemplu de utilizare:

```
#include <ntddk.h>

NTSTATUS status;
KSEMAPHORE sem;

KeInitializeSemaphore(&sem, 1, 1);

status = KeWaitForSingleObject(&sem, UserRequest, KernelMode, TRUE, NULL);
if (NT_SUCCESS(status)) {
        /* critical region */
        KeReleaseSemaphore(&sem, 0, 1, FALSE);
}
```

```
else {
        /* failed to acquire the semaphore */
}
```

Pentru sincronizare mai pot fi folosite si alte obiecte cum ar fi procese, threaduri, mutexuri, fastmutexuri, timere, evenimente. Detalii despre acestea se gasesc in documentatia de DDK.

Operatii atomice

In Windows kernel pentru a realiza operatii atomice se folosesc functiile `Interlocked`.
In continuare sunt prezentate cateva dintre acestea:

- InterlockedCompareExchange compara doua valori de 32 biti si in functie de rezultatul comparatiei schimba cu o a treia valoare de 32 biti
- InterlockedDecrement decrementeaza cu o unitate o variabila intreaga
- InterlockedExchange schimba doua valori
- InterlockedExchangeAdd aduna doua valori si intoarce suma
- InterlockedIncrement incrementeaza cu o unitate o variabila intreaga

Quiz

Pentru auto-evaluare (de preferat înainte de laborator) răspundeţi la întrebările de aici.

Exerciţii

- Folosiţi arhiva de sarcini a laboratorului.
- Pe maşina virtuală de Linux recomandăm folosirea `wget` pentru descărcarea arhivei.
- Maşinile virtuale pot fi accesate, respectiv, prin Multicast DNS, folosind numele `spook.local` (Linux) şi `chooch.local` Windows.
  - Pentru accesarea maşinilor virtuale puteţi folosi SSH (are mai mult sens pentru Linux). Autentificarea se realizează folosind chei (fără parolă):$ ssh -l root spook.local
    $ ssh -l Administrator chooch.local
  - Conturile maşinilor virtuale sunt:
    - Linux: root/student, student/student
    - Windows: Administrator/student, student/student
    - fiind vorba de kernel programming/driver development veţi folosi preponderent conturile privilegiate (`root` respectiv `Administrator`)
- Fiecare maşină virtuală are un director partajat prin SMB/CIFS. Un set de intrări în `/etc/fstab` pe sistemul gazdă permit montarea facilă a acestora. Share-urile sunt:
  - (**Linux**) directorul `/home/student` de pe maşina virtuală Linux se poate monta în sistemul gazdă folosind comanda:$ mount /home/student/spook-share/
    - directorul `/home/student/spook-share` sau icon-ul proaspăt creat pe Desktop permite accesarea share-ului
  - (**Windows**) directorul `C:\Cygwin\home\Administrator\share` se poate monta în sistemul gazdă folosind comanda:$ mount /home/student/chooch-share/
    - directorul `/home/student/chooch-share` sau icon-ul proaspăt creat pe Desktop permite accesarea share-ului
- Daca nu sunt recunoscute numele spook.local sau chooch.local este posibil ca daemon-ul Avahi sa fie oprit. Reporniti folosind comanda `/etc/init.d/avahi-daemon start`.
- Pe Windows, pentru a folosi Vim in prompt-ul de Windows DDK sau Visual Studio, folositi comanda vim-nox.

Linux

- Folosiţi directorul `lin/` din arhiva de sarcini a laboratorului.
- În cazul apariţiei unui oops, reporniţi maşina virtuală (sau faceţi revert la un snapshot realizat anterior de voi)
- Punctaj total: **7 puncte**

1. (**1 punct**) Creaţi un modul de kernel în care să afişaţi conţinutul unei zone de memorie alocate cu `kmalloc`
   - **Hints**:
     - Nu uitaţi de `kfree`
     - Puteţi porni de la fişierele din directorul `lin/1-2-mem/` din arhiva de sarcini a laboratorului

    - Afişaţi caracterele printabile cu `%c`, folosiţi funcţia (macro-ul) `isprint(char c)`
    - Citiţi secţiunea Alocare memorie din laborator.
    - Alocaţi memorie de ordinul KB
2. (**1 punct**) Creaţi un modul de kernel în care să protejaţi cu spinlock-uri o instrucţiune de alocare de memorie de tip `GFP_KERNEL`.
    - Ce se întâmplă la încărcarea modulului?
    - Ce se întâmplă dacă alocaţi memorie de tip `GFP_ATOMIC`?
    - Înlocuiţi alocarea de memorie cu un apel al funcţiei `schedule_timeout()`
    - Ce face funcţia `schedule_timeout`?
        - Funcţia `schedule_timeout()` primeste ca parametru valoarea pentru timeout in jiffies[4]; puteti seta timeout-ul la valoarea `0`.
        - Citiţi secţiunile Alocare memorie şi Spinlock-uri din laborator.
3. (**2 puncte**) Creati trei module kernel, dupa cum urmeaza:
    - În primul definiţi, exportaţi şi iniţializaţi un semafor
    - **Hints**:
        - Pentru a exporta semaforul folosiţi macro-ul `EXPORT_SYMBOL`
        - Semaforul trebuie initializat cu valoarea 0
    - În cel de-al doilea importaţi semaforul şi aşteptaţi la semafor.
    - **Hints**:
        - Pentru a importa semaforul folosiţi `extern` la declarare
    - În cel de-al treilea importaţi semaforul şi eliberaţi semaforul.
    - Operaţiile asupra semaforului (iniţializare, aşteptare, eliberare) se execută în funcţiile `init` ale modulelor.
    - Afişaţi mesaje sugestive înainte şi după fiecare operaţie asupra semaforului.
    - Folosiţi `lsmod` pentru a observa dependenţele între module după încărcare.
    - Vizualizaţi fişierul `modules.order`, observaţi dependenţele
    - **Hints**:
        - Semafoarele le definiti ca "non-pointer", si apoi le apelati cu &
        - La inserarea celui de-al doilea modul, acesta va ramane blocat in executie; pentru inserarea celui de-al treilea modul, deschideti o noua consola in masina virtuala cu `Alt+F2`; puteti reveni la consola anterioara cu `Alt+F1`
        - Puteţi porni de la fişierele din directorul `lin/3-sem/` din arhiva de sarcini a laboratorului; pentru compilarea celor trei submodule simultan, rulati comanda `make` in directorul `3-sem/`.
        - Citiţi secţiunea Semafoare din laborator.
4. (**2 puncte**) Creaţi un modul de kernel pentru lucrul cu liste.
    - În funcţia `init` creaţi o listă kernel de întregi şi afişaţi lista.
    - În funcţia `exit` distrugeţi lista şi afişaţi elementele pe măsură ce sunt şterse.
    - **Hints**:
        - Puteţi porni de la fişierele din directorul `lin/4-list/` din arhiva de sarcini a laboratorului.
        - Citiţi secţiunea Liste din laborator.
5. (**1 punct**) Creaţi două module kernel care comunică folosind o variabilă atomică.
    - Primul modul iniţializează variabila şi o exportă.
    - La încărcare, cel de-al doilea modul incrementează variabila.
    - La descărcare, primul modul raportează dacă al doilea modul a fost prezent.
    - **Hints**:
        - Puteţi porni de la fişierele din directorul `lin/5-atom/` din arhiva de sarcini a laboratorului; pentru compilarea celor doua submodule simultan, rulati comanda `make` in directorul `5-atom/`.
        - Citiţi secţiunea Variabile atomice din laborator.

Windows

- Folosiţi directorul `win/` din arhiva de sarcini a laboratorului.
- Punctaj total: **4 puncte**

1. (**1 punct**) Creaţi un modul de kernel în care să afişaţi conţinutul unei zone de memorie alocate cu `ExAllocatePool`.
    - **Hints**:
        - Folositi DebugView pentru captura mesajelor afisate din kernel.
        - Nu uitaţi de `ExFreePool`.
        - Puteţi porni de la fişierele din directorul `win/1-mem/` din arhiva de sarcini a laboratorului
        - Citiţi secţiunea Alocare memorie din laborator.
2. (**1 punct**) Creaţi un modul de kernel în care să utilizaţi un semafor.
    - Iniţializaţi semaforul la valoarea 5.
    - Într-un ciclu `for` cu 5 paşi aşteptaţi la semafor şi afişaţi simultan valoarea acestuia.
    - **Hints**:
        - Pentru a lua valoarea semaforului folosiţi funcţia `KeReadStateSemaphore`.
        - Puteţi porni de la fişierele din directorul `win/2-sem/` din arhiva de sarcini a laboratorului

    - Citiţi secţiunea Locking din laborator.
3. (**2 puncte**) Creaţi un modul de kernel pentru lucrul cu liste.
    - În funcţia `DriverEntry` creaţi o listă kernel de întregi şi afişaţi lista.
    - În funcţia `DriverUnload` distrugeţi lista şi afişaţi elementele pe masură ce sunt şterse.
        - Nu folositi variabile auxiliare pentru parcurgerea listei. Folositi **doar** capatul listei.
    - **Hints**:
        - Puteţi porni de la fişierele din directorul `win/3-list/` din arhiva de sarcini a laboratorului
        - Citiţi secţiunea Liste din laborator.

Soluţii

- Soluţii exerciţii laborator 3

Resurse utile

Linux

1. Linux Device Drivers, 3rd edition
    - Chapter 5. Concurrency and Race Conditions
    - Chapter 8. Allocating Memory
    - Chapter 11. Data Types in the Kernel
2. The Linux Kernel API
3. Linux Kernel Linked-List Explained
4. Unreliable Guide To Locking
5. Linux Kernel Walkthrough Screencast

Windows

1. The Windows 2000 Device Driver Book, Second Edition ? Chapter 5. General Development Issues
2. Windows Driver Programming Techniques
3. Windows Synchronization Techniques
4. Windows Memory Management
5. Summary of Kernel-Mode Support Routines
6. Preventing Errors and Deadlocks While Using Spin Locks
7. Locks, Deadlocks, and Synchronization
8. Managing Hardware Priorities
9. Scheduling, Thread Context, and IRQL
10. Interlocked Variable Access

[1] în Linux consola este terminalul virtual curent; din această cauză atunci când folosiţi X Windows, aceste mesaje nu o sa apară în emultorul de terminal `xterm`, însă le puteţi afişa folosind comanda `dmesg` sau accesând log-ul asociat `/var/log/syslog`

[2] Pentru mai multe detalii despre configurari pentru logging consultati Laboratorul 2

[3] pentru testarea memoriei marcată cu tag, se pot folosi o serie de utilitare (Testing for Errors in Accessing and Allocating Memory), printre care Driver Verifier (How to Use Driver Verifier to Troubleshoot Windows Drivers)

[4] Pentru mai multe detalii despre jiffies consultati Laboratorul 7