Vlákna v uživatelském prostoru 12

Při studiu operačních systémů jsme velice často limitováni jejich složitostí, která z velké míry pramení ze složitosti soudobého hardwaru. Máme proto omezené možnosti, jak se prakticky podívat na to, jak jsou jednotlivé části OS implementovány. V tomto cvičení si ukážeme, jak je možné implementovat vlákna v uživatelském prostoru. Díky tomu můžeme nahlédnout, jak je řešena správa procesoru a multiprogramování, aniž bychom potřebovali podrobné znalosti implementace OS. Implementace vláken v uživatelském prostoru totiž používá stejné principy jako jádro OS, avšak využívá mnohem jednodušší prostředky.

1 Veřejné rozhraní

Začneme popisem rozhraní, které pro práci s vlákny budeme používat. Naše implementace vláken v uživatelském prostředí bude s menšími odlišnostmi kopírovat rozhraní knihovny *pthreads*.

1.1 Funkce a datové typy

Pro práci s vlákny budeme mít datový typ `uthread_t` reprezentující vlákno a sadu čtyř funkcí.

```
/** vytvori nove vlakno, rozhrani kopiruje pthreads */
uthread_t uthread_create(void * (*thr_proc)(void *), int attributes, void *arg);
/** prepne aktualni vlakno a zacne provadet jine */
void uthread_yield();
/** pocka na dobehnuti vlakna thread a pres argument result vrati navratovou hodnotu */
void uthread_join(uthread_t thread, void **result);
/** spusti planovac vlaken */
void uthread_start_scheduler();
```

Funkce `uthread_create` vytvoří nové vlákno, které je specifikované ukazatelem na funkci typu `void * (*f)(void *)`, stejně jako je tomu v případě knihovny pthreads. Při spuštění vlákna je této funkci předán argument `arg`. U vlákna je možné nastavit jeden ze dvou atributů:

- `UTHREAD_JOINABLE`, který značí, že jiné vlákno bude čekat na jeho dokončení, viz funkce `uthread_join`. Prostředky, které jsou pro toto vlákno alokovány, budou uvolněny společně s ukončením funkce `uthread_join`.
- `UTHREAD_DETACH`, který značí, že prostředky vlákna jsou uvolněny ihned po jeho skončení a žádné jiné vlákno nebude čekat na jeho dokončení.

Funkce `uthread_yield` se postará o přepnutí aktuálního vlákna na jiné. Narozdíl od knihovny pthreads, tato funkce je naprosto zásadní, protože naše implementace vláken používá kooperativní multitasking a bez ní by nedocházelo k přepínání vláken.

Funkce `uthread_join` zajistí, že aktuální vlákno čeká na dokončení zadaného vlákna, a pokud je jí předán ukazatel na místo v paměti, převezme návratovou hodnotu vlákna a uloží ji na místo dané tímto ukazatelem.

Poslední funkcí, kterou budeme potřebovat pro práci s vlákny, je funkce `uthread_start_scheduler`, která aktivuje plánovač a s ním přepínání námi vytvořených vláken. Před zavoláním této funkce je nutné vytvořit alespoň jedno vlákno, které by mohlo běžet. Funkce `uthread_start_scheduler` skončí (a pokračuje v provádění kódu) v momentě, kdy všechna námi vytvořená vlákna doběhnou do konce.

1.2 Příklad použití

Následující kód demonstruje vytvoření vláken a spuštění plánovače voláním funkce `uthread_start_scheduler()`.

```
uthread_t thr1 = uthread_create(&foo_thread, UTHREAD_DETACHED, INT_TO_PTR(1));
uthread_t thr2 = uthread_create(&foo_thread, UTHREAD_DETACHED, INT_TO_PTR(42));

printf("spustim vlakna ...\n");
uthread_start_scheduler();
printf("pokracuje se jiz bez vlaken\n");
```

Kód funkce s vláknem se přilíš neliší od kódu, který bychom použili s knihovnou pthreads.

```
void *foo_thread(void *arg) {
        printf("Spusteno vlakno A\n");
        int step = (long) arg;
        for (int i = 0; i < 10; i++) {
                printf("A:%i\n", i * step);
                uthread_yield();
        }
        return 0;
}
```

Avšak je nutné na vhodných místech umožnit běh jiných vláken pomocí volání funkce `uthread_yield()`[1].

2 Implementace

Při implementaci vláken v uživatelském prostoru si musíme uvědomit jednu důležitou věc. Kdykoliv dojde k zavolání funkce `uthread_yield()`, musíme si pro aktuálně běžící vlákno uložit adresu, kde se právě

[1] V případě OS je přepnutí obvykle vyvoláno na základě přerušení časovače.

v kódu nachází (registr `rip`) a obsah registrů, které kód vlákna používá. To je nutné proto, abychom se mohli přepnout do jiného vlákna, a po čase se vrátit do vlákna a pokračovat kódem za voláním funkce `uthread_yield()`. Tím, že implementujeme vlákna v uživatelském prostoru, se nám celý problém výrazným způsobem zjednodušuje. Jednak při volání funkce `uthread_yield` dojde k provedení instrukce `call uthread_yield`, která uloží obsah registru `rip` na zásobník, a tím se nám uložení aktuálního místa v programu vyřeší z části zcela přirozeně. A dále nemusíme ukládat obsah všech registrů, protože konvence vyžaduje, aby po návratu z funkce `uthread_yield` zůstal zachován jen obsah callee-saved registrů, tj. `rsp`, `rbp`, `rbx`, `r12`, ..., `r15`.[2]

Aby přepínání mohlo fungovat, je nutné ještě zajistit, že každé vlákno má svůj vlastní zásobník. Nastavení zásobníku a uložení, resp. načtení, obsahu registrů, nelze vyřešit přímo z jazyka C, ale budeme si muset pomoci krátkým kódem v assembleru.

2.1 Datové struktury

Nyní si představíme hlavní datové struktury, se kterými budeme pracovat.[3]

2.1.1 Thread control block (TCB)

Nejdůležitější datovou strukturou, se kterou budeme pracovat je `struct uthread_tcb`,[4] která obsahuje všechny informace o vláknu, tzv. *thread control block (TCB)*:

```
struct uthread_tcb {
        // kontext vlakna
        uint64_t rip;
        uint64_t rsp;

        // callee-saved registry
        uint64_t rbp;
        uint64_t rbx;
        uint64_t r12;
        uint64_t r13;
        uint64_t r14;
        uint64_t r15;

        // servisni informace
        void *(*thread_proc)(void *arg); // funkce vykonavana vlaknem
        void *stack;                     // zacatek zasobniku
        void *arg;                       // predany argument
        void *result;                    // vysledna hodnota
```

[2] Pokud by se jednalo o preemptivní přepínání, adresa aktuálního místa v programu by byla uložena při přerušení a bylo by nutné uložit obsah všech registrů.

[3] Vedle nich budou v kódu použity i další a jednodušší datové typy, jejichž význam by měl být zřejmý přímo z kódu, a proto se nebudeme věnovat jejich popisu.

[4] Datový typ `uthread_t` je aliasem této struktury.

```
        uint32_t id;                    // identifikator vlakna
        uint8_t attrs;                  // vlastnosti vlakna
        enum uthread_status status;     // stav vlakna

        // slouzi k umisteni vlakna do oboustranneho spojoveho seznamu (napr. fronty)
        struct uthread_tcb *prev;
        struct uthread_tcb *next;

        // ukazatel na vlakno, ktere ceka na dokonceni tohoto vlakna
        struct uthread_tcb *blocked_thread;

        // informace pro planovac
        uint64_t runtime;               // cas, po ktery vlakno bezelo
};
```

Informace v této struktuře se dají rozdělit do několika skupin:

1. obsah registrů (kontext vlákna),
2. servisní informace obsahující informace o vlastnostech vlákna (id, status, atributy, funkce vlákna, argument, návratová hodnota),
3. vlákno čekající ve funkci `uthread_join`,
4. informace pro plánovač,
5. pomocné atributy (`prev` a `next`) sloužící k uložení vlákna do fronty, která je realizována jako oboustranný spojový seznam.

S datovým typem `struct uthread_tcb` se setkáme hned na několika místech. Jednak slouží k uložení informací o jednotlivých vláknech. Dále v globální proměnné `uthread_active_tcb` si budeme držet ukazatel na TCB aktuálně běžícího vlákna a vedle toho nám tato struktura budou sloužit k vytvoření front čekajících vláken, ať už připravených k běhu, tak čekajících na synchronizaci s jinými vlákny.

2.1.2 Fronty vláken

Pro realizaci front čekajících vláken budeme používat oboustranný spojový seznam reprezentovaný typem:

```
struct uthread_queue {
        struct uthread_tcb *head;
        struct uthread_tcb *tail;
};
```

Všimněme si, že tato struktura obsahuje jen ukazatel na začátek a konec fronty. Seřazení vláken se děje přímo v typu `struct uthread_tcb` pomocí atributů (`prev` a `next`). Toto řešení není omezující, protože

předpokládáme, že každé vlákno se může nacházat nanejvýš v jedné frontě. Funkce, pro práci s frontou jsou definovány v souborech `uthreads-util.[ch]`, a protože jejich význam by měl zřejmý, nebudeme je podrobně popisovat.

2.2 Plánovač

Nad frontou vláken si postavíme jednoduchý plánovač typu round-robin.[5] Tento plánovač bude používat globální proměnnou `struct uthread_queue queue` a budou v něm uložena vlákna připravená k běhu. S plánovačem se pracuje pomocí tří obligátních funkcí: (i) pro inicializaci, (ii) pro zařazení vlákna do fronty a (iii) pro vybrání vlákna z fronty.

```
void uthread_scheduler_init();
void uthread_scheduler_enqueue(struct uthread_tcb *thread);
struct uthread_tcb *uthread_scheduler_dequeue();
```

2.3 Uložení a načtení kontextu

Při přepínání vláken je klíčové uložit a obnovit kontext prováděného vlákna, tj. obsah registrů. V tomto místě si pomůžeme dvěma funkcemi v assembleru. Mírně jednodušší je obnovit kontext z TCB a začít provádět program od zadaného místa, k tomu slouží funkce `void uthread_run()`:

```
;
; void uthread_run();
;
thread_run:
    mov rdx, [uthread_active_tcb] ; ziska adresu TCB
    mov rsp, [rdx + 8]            ; obnovi obsah registru
    mov rbp, [rdx + 16]
    mov rbx, [rdx + 24]
    mov r12, [rdx + 32]
    mov r13, [rdx + 40]
    mov r14, [rdx + 48]
    mov r15, [rdx + 56]
    jmp [rdx]                     ; skoci na adresu uthread_active_tcb->rip
```

Uložení kontextu provedeme ve velice podobném duchu funkcí `uthread_internal_yield`.

```
;
; void uthread_internal_yield(int block_current_thread);
;
uthread_internal_yield:
    pop rax ; nacte navratovou adresu z/do vlakna
```

[5] V principu je možné vytvořit si libovolný plánovač, který bude mít stejné rozhraní jako náš ukázkový plánovač.

```
    mov rdx, [uthread_active_tcb]
    mov [rdx], rax
    mov [rdx + 8], rsp
    mov [rdx + 16], rbp
    mov [rdx + 24], rbx
    mov [rdx + 32], r12
    mov [rdx + 40], r13
    mov [rdx + 48], r14
    mov [rdx + 56], r15
    jmp uthread_switch
```

Nejdříve odeberem hodnotu ze zásobníku (ta obsahuje návratovou adresu z funkce `uthread_yield`) a uložíme ji do `uthread_active_tcb->rip`. Dále uložíme obsah všech registrů a provedeme skok do funkce, která se postará o přepnutí do jiného vlákna. Tou je v našem případě funkce `uthread_switch`.

Funkce `uthread_internal_yield` a `uthread_switch` mají jeden argument, který indikuje, zda se má vlákno po přepnutí zařadit mezi ostatní vlákna připravená k běhu, nebo jestli je blokováno a čeká na nějakou událost. Protože tento argument je předán funkci `uthread_internal_yield` v registru `rdi`, je automaticky předán i funkci `uthread_switch` při provedení instrukce `jmp`.

2.4 Přepnutí vláken

Mechanismus přepínání vláken je řešen funkcí `void uthread_switch(int block_current_thread)`. Pokud existuje vlákno, které je právě přepínáno,[6] tak na základě `block_current_thread` zařazeno plánovačem mezi vlákna čekající na provedení, nebo je zablokováno. Následně je plánovačem vybráno další vlákno, to je nastaveno do `uthread_active_tcb` a jeho provedení je aktivováno zavoláním `uthread_run()`. Zjednodušenou variantu této funkce ukazuje následující kód.[7]

```
void uthread_switch(int block_current_thread) {
        // pokud existuje aktivni vlakno, zaradime jej do fronty
        if (uthread_active_tcb) {
                // zmena stavu a zarazeni do fronty
                if (!block_current_thread) {
                        uthread_active_tcb->status = UT_READY;
                        uthread_scheduler_enqueue(uthread_active_tcb);
                } else {
                        uthread_active_tcb->status = UT_BLOCKED;
                        blocked++;
                }
        }
        // volba a aktivace noveho vlakna
        uthread_active_tcb = uthread_scheduler_dequeue();
```

[6] V případě, že nějaké vlákno dokončení svou činnost, tak je hodnota `uthread_active_tcb` nastavena na `NULL`.

[7] Reálný kód navíc řeší počítání procesorového času nebo ukončení plánovače.

```
        uthread_run();
}
```

Nad těmito funkcemi si vytvoříme jednoduché rozhraní, které se stará o přepínání vláken mezi stavy *ready* a *blocked*.

```
/** prepne aktualni vlakno (prepnuti RUNNING -> READY) */
void uthread_yield() {
        uthread_internal_yield(0);
}
/** zablokuje aktualni vlakno a da provadet dalsi (prepnuti RUNNING -> BLOCKED) */
static inline void uthread_block() {
        uthread_internal_yield(1);
}
/** prepne vlakno ze stavu BLOCKED do stavu READY */
static inline void uthread_wakeup(struct uthread_tcb *thread) {
        thread->status = UT_READY;
        blocked--;
        uthread_scheduler_enqueue(thread);
}
```

2.5 Vytvoření a zrušení vlákna

Při vytvoření vlákna je nutné primárně zajistit inicializaci struktury `uthread_tcb` a alokaci zásobníku pro vlákno. Protože je zásobník úsek paměti jako každý jiný, můžeme tak učinit funkcí `malloc`. Následující kód nastiňuje, jak je vlákno vytvořeno.

```
uthread_t uthread_create(void * (*thr_proc)(void *), int attributes, void *arg) {
    struct uthread_tcb *tcb   = malloc(sizeof(struct uthread_tcb));
    // vytvori zasobnik pro nove vytvorene vlakno
    unsigned char      *stack = malloc(UTHREAD_STACK_SIZE);
    // nevolame primo funkci, ale obalovou funkci, ktera resi uvolneni
    // prostredku a synchronizaci s ostatnimi vlakny
    tcb->rip            = (uint64_t) uthread_wrapper;
    tcb->rsp            = (uint64_t) (stack + UTHREAD_STACK_SIZE);
                                    // zasobnik roste od vyssich adres
    tcb->stack          = stack;
    tcb->id             = threads_total++;
    tcb->arg            = arg;
    tcb->thread_proc    = thr_proc;
    tcb->attrs          = attributes;
    tcb->blocked_thread = NULL;
    tcb->runtime        = 0;
    tcb->status         = UT_NEW;
```

```
    uthread_scheduler_enqueue(tcb);
    return tcb;
}
```

Nejdříve jsou nastaveny vlastnosti vlákna a následně je vlákno zařazeno mezi procesy připravené k běhu. Ale protože při skončení vlákna musíme buď uvolnit prostředky s ním spojené (např. zásobník) nebo zajistit synchronizaci s jiným vláknem pomocí funkce `uthread_join`, nemůžeme nastavit do registru `rip` přímo adresu funkce vlákna, ale musíme zavolat obalovou funkci, která tyto úkony obstará za nás.

Tato funkce vypadá následovně:

```
static void uthread_wrapper() {
        // spusti kod vlakna se zadanym argumentem
        void *result = uthread_active_tcb->thread_proc(uthread_active_tcb->arg);

        // resi uvolneni prostredku vlakna
        if (uthread_active_tcb->attrs & UTHREAD_DETACHED) {
                // uvolnime prostredky okamzite, jine vlakno neceka
                uthread_dispose(uthread_active_tcb);
        } else {
                // ulozime vysledek, a pokud existuje vlakno, ktere ceka na dokonceni
                // tohoto vlakna, probudime jej
                uthread_active_tcb->result = result;
                uthread_active_tcb->status = UT_TERMINATED;
                if (uthread_active_tcb->blocked_thread) {
                        uthread_wakeup(uthread_active_tcb->blocked_thread);
                }
        }
        uthread_active_tcb = NULL;
        uthread_switch(0);
}
```

Nejdříve spustíme kód vlákna zavoláním funkce s tím, že do funkce předáme argument, se kterým má být vlákno spuštěno. Adresa volané funkce i argument jsou uloženy v TCB.

Po skončení funkce, která reprezentuje vlákno, získáme návratovou hodnotu. A následně, podle nastaveného atributu, naložíme s vláknem.

Pokud vlákno bylo vytvořeno jako `UTHREAD_DETACHED`, uvolníme všechny jeho prostředky okamžitě. Funkce `uthread_dispose` uvolní veškerou paměť, která byla pro vlákno alokována, např. zásobník.

V případě, že vlákno bylo vytvořeno s atributem `UTHREAD_JOINABLE`, uložíme návratovou hodnotu do TCB a vláknu nastavíme stav *terminated*. Pokud nějaké vlákno zavolalo funkci `uthread_join` a čeká na právě skončené vlákno (víme to z atributu `uthread_active_tcb->blocked_thread`), tak jej probudíme, přesenume jej ze stavu blocked do stavu ready. V každém případě voláme funkci `uthread_switch`, která přepne na další vlákno.

U vláken, které jsou typu `UTHREAD_JOINABLE`, se mimo jiné o uvolnění prostředků stará funkce `uthread_join`. U této funkce mohou nastat dva stavy. Buď vlákno, na které má počkat ještě nedoběhlo. V takovém případě dojde k uspání aktuálního vlákna. Abychom po skončení vlákna věděli, které vlákno se má probudit, uložíme si tuto informace do TCB jako atribut `blocked_thread`. Pokud vlákno, na které se má počkat skončilo, uložíme návratovou hodnotu a uvolníme přidělené prostředky, jak ukazuje následující kód.

```
void uthread_join(uthread_t thread, void **result) {
        // pokud vlakno jeste nedobehlo, uspime aktualni vlakno
        if (thread->status != UT_TERMINATED) {
                thread->blocked_thread = uthread_active_tcb;
                uthread_block();
        }
        // vratime hodnotu a uvolnime prostredky
        if (result) {
                *result = thread->result;
        }
        uthread_dispose(thread);
}
```

2.6 Aktivace a deaktivace prostředí

Poslední záležitost, kterou musíme ošetřit, je spuštění a ukončení plánovače, přičemž chceme, aby po skončení běhu všech vláken program pokračoval standardním způsobem. Abychom toho mohli docílit, potřebujeme si uložit stav registrů před spuštěním plánovače. Ten pak obnovíme po skončení všech vláken. Zde si opět pomůžeme krátkým kódem v assembleru.

```
global uthread_start_scheduler
global uthread_mainthread_context
section .text
uthread_start_scheduler:
    ; ulozi kontext volajici funkce
    mov rdx, uthread_mainthread_context
    mov [rdx + 8], rsp
    mov [rdx + 16], rbp
    mov [rdx + 24], rbx
    mov [rdx + 32], r12
    mov [rdx + 40], r13
    mov [rdx + 48], r14
    mov [rdx + 56], r15

    ; do uthread_mainthread_context->rip  ulozi adresu kodu, ktery funkci ukonci
    mov qword [rdx], uthreads_complete
```

```
    mov qword [uthread_active_tcb], 0 ; na zacatku neni aktivni zadne vlakno
    mov rdi, 0                        ; neblokujeme vlakno
    jmp uthread_switch                ; spustime planovac

uthreads_complete:
    ret

section .bss
uthread_mainthread_context:
   resq 8 ; pamet nutna pro ulozeni kontextu, s ostatnimi hodnotami nepracujeme
```

Vedle samotné funkce pro spuštění plánovače si vytvoříme globální proměnnou `uthread_mainthread_context`, která je typu `struct uthread_tcb`. Do ní uložíme kontext vlákna, které zavolalo funkci `uthread_start_scheduler`. V momentě, kdy všechna námi vytvořená vlákna skončí, chceme, aby se provedl návrat z funkce `uthread_start_scheduler()`, proto do atributu `rip` uložíme adresu instrukce `ret`.

Dále aktivujeme plánovač, a to tak, že nastavíme, že není aktivní žádné vlákno, a provedeme skok do funkce `uthread_switch`, která vybere první vlákno určené k běhu. Následně se mezi sebou jednotlivá vlákna střídají ve využívání procesoru.

Pokud není k dispozici vlákno, které by mohlo být vykonáváno, nastaví se jako aktivní vlákno ukazatel na `uthread_mainthread_context` a je zavolána funkce `uthread_run()`. Díky ní je obnoven obsah registrů funkce, která volala `uthread_start_scheduler`, skočí se na instrukci `ret`, a program může dál pokračovat.

3 Synchronizace

V případě kooperativního multitaskingu se můžeme chybám souběhu (race condition) snadno vyvarovat, protože můžeme vložit přepnutí na vhodná místa a nemusíme uvažovat, že může dojít k přepnutí vláken v libovolném bodě. Přesto synchronizační prostředky dávají i v tomto prostředí smysl. Ukážeme si proto, jakým způsobem se dají implementovat semafory s pasivním čekáním. Tato implementace je přímočará a vychází z funkcí, které jsme si představili v předchozí kapitole.

Základem je struktura, která obsahuje hodnotu semaforu a seznam vláken, které na tomto semaforu čekají:

```
struct uthread_sem {
        // hodnota semaforu, pokud je hodnota < 0, znamena to pocet cekajicich vlaken
        int value;
        // fronta vlaken cekajicich na semaforu
        struct uthread_queue blocked_threads;
};
```

V následujícím kódu můžeme vidět funkce pro inicializaci, snížení a zvýšení hodnoty semaforu. Naše řešení umožňuje, aby hodnoty šly do záporných čísel. To nám signalizuje, kolik vláken na daném semaforu čeká.

```
/** inicializuje semafor */
void uthread_sem_init(uthread_sem_t *sem, int value) {
        sem->value = value;
        uthread_queue_init(&sem->blocked_threads);
}
/** snizi hodnotu semaforu o jedna, a pokud je uz na nule, tak ceka */
void uthread_sem_wait(uthread_sem_t *sem) {
        sem->value--;
        if (sem->value < 0) {
                uthread_queue_put(&sem->blocked_threads, uthread_active_tcb);
                uthread_block();
        }
}
/** zvysi hodnotu semaforu o jedna, a pokud nejake vlakno na nej ceka, probudi jej  */
void uthread_sem_post(uthread_sem_t *sem) {
        if (sem->value < 0) {
                struct uthread_tcb *blocked = uthread_queue_poll(&sem->blocked_threads);
                uthread_wakeup(blocked);
        }
        sem->value++;
}
```

Aby semafory mohly fungovat jako synchronizační nástroj, musí být jeho operace prováděny atomicky. To je v případě kooperativního multitaskingu přirozeně splněno, pokud bychom měli preemptivní multitasking, museli bychom operace semaforu ještě obalit zámky, nejspíše spinlocky.

Úkoly

1. Doplňte funkci `void uthread_set_priority(int)`, která aktuálnímu vláknu nastaví zadanou prioritu.
2. Upravte plánovač tak, aby fungoval jako Completely Fair Scheduler z Linuxového jádra. Uspořádejte vlákna podle toho, kolik dostaly procesorového času a vždy vyberte to, co dostalo nejméně. Aby byla implementace jednodušší, nemusíte používat červeno-černý strom pro evidenci vláken. Postačí libovolná datová strukturu (např. pole), kde budou vlákna seřazena. Do řešení zahrňte prioritu vlákna.
3. (Volitelně) Implementujte *lottery scheduler*, který je postavený na tom, že každé vlákno bude mít přiděleno určité množství losů, ze kterých bude plánovač vybírat.
4. Vyzkoušejte si prakticky, jak jsou vlákna přidělována různými plánovači.