KATEDRA
INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

Vlákna v uživatelském prostoru

KMI/OS1 Operační systémy I

Mgr. Markéta Trnečková, Ph.D.
www.marketa-trneckova.cz

Vlákna v uživatelském prostoru

Učební text + zdrojové kódy ke cvičení:
`https://phoenix.inf.upol.cz/~krajcap/courses/2025LS/OS1/tutorial12.pdf`

Vlákna v uživatelském prostoru

- Operační systémy jsou složité, což z velké míry pramení ze složitosti soudobého hardware
- Máme omezené možnosti, jak se podívat, jak jsou jednotlivé části OS implementovány
- Dnes se podíváme, jak je možné implementovat vlákna v uživatelském prostoru
- Díky tomu se podíváme na to, jak je řešená správa procesoru a multiprogramování
- Nepotřebujeme znát podrobnosti o implementaci OS
- V uživatelském prostoru používáme stejné principy (jako jádro OS), ale využíváme jednodušší prostředky

Veřejné rozhraní

- Popíšeme si prostředí, jaké budeme používat
- Naše implementace bude víceméně kopírovat rozhraní knihovny `pthreads`

Veřejné rozhraní

Funkce a datové typy

- **Datový typ**: `uthread_t` (reprezentuje vlákno)

Funkce:

- **uthread_create** – vytvoří nové vlákno (rozhraní kopíruje `pthreads`)

```
uthread_t uthread_create(void * (*thr_proc)(void *), int attributes,
                         void *arg);
```

- vytvoří nové vlákno, které je specifikované ukazatelem na funkci typu `void* (*f)(void*)` (stejně jako v `pthreads`)
- při spuštění vlákna je funkci předán argument `arg`
- u vlákna je možné nastavit jeden ze dvou atributů:
  - `UTHREAD_JOINABLE` – jiné vlákno bude čekat na jeho dokončení (viz fce `uthread_join`). Prostředky, které jsou pro toto vlákno alokovány, budou uvolněny spolu s ukončením fce `uthread_join`.
  - `UTHREAD_DETACH` – prostředky vlákna jsou uvolněny ihned po jeho skončení a žádné jiné vlákno nebude čekat na jeho dokončení

Veřejné rozhraní

Funkce a datové typy

Funkce:

- **uthread_yield** – přepne aktuální vlákno a začne vykonávat jiné

```
void uthread_yield();
```

- stará se přepínání aktuálního vlákna na jiné
- je zásadní pro přepínání vláken
- naše implementace používá kooperativní multitasking
- **uthread_join** – počká na doběhnuti vlákna `thread` a přes argument `result` vrátí návratovou hodnotu

```
void uthread_join(uthread_t thread, void **result);
```

- **uthread_start_scheduler** – spustí plánovač vláken a s ním přepínání námi vytvořených vláken

```
void uthread_start_scheduler();
```

- Před zavoláním této funkce je nutné vytvořit alespoň jedno
- funkce skončí v momentě, kdy všechna námi vytvořená vlákna doběhnou do konce

Veřejné rozhraní
Příklad

```
/* vytvoreni vlaken */
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");

/* spusteni planovace vlaken */
uthread_start_scheduler();

/* po dokonceni behu vsech vlaken*/
printf("pokracuje se jiz bez vlaken\n");
```

Veřejné rozhraní
Příklad

```c
/* funkce foo_thread */
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;
}
```

Příklad

Vysvětlete, co se ve funkci `foo_thread` děje.

Implementace

- kdykoliv dojde k volání funkce `uthread_yield()`, musíme si pro aktuálně běžící vlákno uložit adresu, kde se právě nachází (registr `rip`) a obsah registrů, které vlákno používá
- to potřebujeme, abychom se po čase mohli do tohoto stavu vrátit a pokračovat
- při volání fce `uthread_yield()` dojde k provedení instrukce `call uthread_yield`, která uloží obsah registru `rip` na zásobník (problém s návratem do tohot místa se vyřešil přirozeně)
- nemusíme ukládat ani obsah všech registrů – konvence vyžaduje, že po návratu z funkce zůstal zachován obsah callee-saved registrů
- **Jaké registry jsou callee-saved?**

Implementace

- kdykoliv dojde k volání funkce `uthread_yield()`, musíme si pro aktuálně běžící vlákno uložit adresu, kde se právě nachází (registr `rip`) a obsah registrů, které vlákno používá
- to potřebujeme, abychom se po čase mohli do tohoto stavu vrátit a pokračovat
- při volání fce `uthread_yield()` dojde k provedení instrukce `call uthread_yield`, která uloží obsah registru `rip` na zásobník (problém s návratem do tohot místa se vyřešil přirozeně)
- nemusíme ukládat ani obsah všech registrů – konvence vyžaduje, že po návratu z funkce zůstal zachován obsah callee-saved registrů
- **Jaké registry jsou callee-saved?**
- `rsp`, `rbp`, `rbx`, `r12`, ..., `r15`
- aby přepínání mohlo fungovat, je nutné zajistit, aby každé vlákno mělo svůj zásobník
- nelze to vyřešit přímo z jazyka C, musíme si pomoci krátkým kódem v assembleru

Datové struktury
Thread control block (TCB)

- `uthread_tcb` – obsahuje všechny informace o vláknu

```
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
    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;
};
```

Datové struktury
Thread control block (TCB)

- **Informace ve struktuře**:
  - obsah registrů – kontext vlákna
  - servisní informace – informace o vlastnostech vlákna (id, status, atributy, funkce vlákna, argument, návratová hodnota)
  - vlákno čekající ve funkce `uthread_join`
  - informace pro plánovač
  - pomocné atributy (`prev` a `next`) sloužící k uložení vlákna do fronty (oboustranný spojový seznam)
- s touto strukturou se setkáme:
  - slouží k uložení informací o jednotlivých vláknech
  - v globální proměnné `uthread_active_tcb` si držíme ukazatel na TCB aktuálně běžícího vlákna
  - ve frontách čekajících vláken (připravených k běhu, nebo čekajících na synchronizaci s jinými vlákny)

Datové struktury

Fronty vláken

- spojový seznam

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

- máme pouze ukazatel na začátek a konec fronty (seřazení je totiž přímo ve struktuře `uthread_tcb` – atributy `prev` a `next`
- předpokládáme, že každé vlákno se může nacházet nanejvýš v jedné frontě
- funkce pro práci s frontou jsou definovány v souborech `uthreads-util.c` a `uthreads-util.h`

Příklad

Sami si projděte soubory `uthreads-util.c` a `uthreads-util.h`. V případě nejasností se zeptejte.

Plánovač

- nástavba nad frontu vláken
- plánovač typu round-robin
- využívá globální proměnnou `struct uthread_queue queue`
- jsou v něm uložena vlákna připravená k běhu
- **funkce**:
  - inicializace

    ```
    void uthread_scheduler_init();
    ```

  - zařazení vlákna do fronty

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

  - vybrání vlákna z fronty

    ```
    struct uthread_tcb *uthread_scheduler_dequeue();
    ```

Uložení a načtení kontextu

- při přepínání vláken je klíčové uložit a pak obnovit kontext prováděného vlákna (obsah registrů)
- pomůžeme si dvěma funkcemi v assembleru
- obnovení – `uthread_run`
- uložení – `uthread_internal_yield`

Uložení a načtení kontextu

Obnovení

```
;
; 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 a načtení kontextu

Uložení

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

Uložení a načtení kontextu

Uložení

- odebereme hodnotu za zásobníku – ta obsahuje návratovou adresu z funkce `uthread_yield`
- uložíme ji do `uthread_active_tcb -> rip`
- uložíme obsah všech registrů a provedeme skok do funkce, která se stará o přepnutí vlákna (`uthread_switch`)
- `uthread_yield` a `uthread_switch` mají jeden argument, který uchovává informaci, zda se má vlákno po přepnutí zařadit mezi ostatní vlákna připravená k běhu, nebo jestli je blokováno a čeká na nějakou událost
- tento argument je předán v registru `rdi`

Přepnutí vláken

- je řešeno funkcí
  `void uthread_switch(int block_current_thread)`
- pokud existuje vlákno, které je právě přepínáno, na základě `block_current_thread` je zařazeno plánovačem mezi vlákna čekající na provedení, nebo je zablokováno
- následně je plánovačem vybráno další vlákno, to je nastaveno do `uthread_active_tcb`
- jeho provedení je aktivováno zavoláním `uthread_run()`

Přepnutí vláken

Zjednodušená varianta

```
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();
    uthread_run();
}
```

Přepnutí vláken

Rozhraní pro přepínání mezi stavy ready a blocked

```
/** prepne aktualni vlakno (prepnuti READY-> READY) */
void uthread_yield() {
    uthread_internal_yield(0);
}

/** zablokuje aktualni vlakno a da provadet dalsi (prepnuti READY->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);
}
```

Vytvoření a zrušení vlákna

- při vytvoření vlákna je nutné primárně zajistit inicializaci struktury `uthread_tcb` a alokaci zásobníku pro vlákno
- zásobník = úsek paměti, použijeme `malloc()`

Vytvoření a zrušení vlákna

Funkce pro vytvoření

```
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;
}
```

Vytvoření a zrušení vlákna

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

Vytvoření a zrušení vlákna

Obalová funkce

```
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);
}
```

Vytvoření a zrušení vlákna

Obalová funkce

- spustíme kód vlákna zavoláním funkce s tím, že funkci předáme argument, se kterým bylo vlákno spuštěno
- adresa volané funkce i argument jsou uloženy v TCB
- po skončení funkce, která reprezentuje vlákno, získáme návratovou hodnotu
- podle nastaveného atributu s ním dále naložíme
  - `UTHREAD_DETACHED` – uvolníme všechny prostředky okamžitě pomocí funkce `uthread_dispose()`
  - `UTHREAD_JOINABLE` – uložíme návratovou hodnotu do TCB a vláknu nastavíme stav `terminated`. Pokud nějaké vlákno zavolalo `uthread_join` a čeká na právě skončené vlákno (to víme z atributu `uthread_active_tcb->blocked_thread`, tak jej probudíme – přeneseme jej ze stavu `blocked` do stavu `ready`

Vytvoření a zrušení vlákna
Obalová funkce

- voláme funkci `uthread_switch`, která přepne na další vlákno
- u vláken, které jsou `UTHREAD_JOINABLE`, se mimo jiné o uvolnění prostředků stará funkce `uthread_join`
- mohou nastat dva stavy
  - vlákno, které má počkat, ještě nedoběhlo – dojde k uspání aktuálního vlákna.
    Abychom po skončení vlákna věděli, které vlákno se má probudit, uložíme si tuto informaci do TCB jako atribut `blocked_thread`
  - vlákno skončilo – uložíme návratovou hodnotu a uvolníme přidělené prostředky

Vytvoření a zrušení vlákna

```
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);
}
```

Aktivace a deaktivace prostředí

- poslední, co je potřeba udělat je spuštění a ukončení plánovače
- chceme, aby i po ukončení běhu všech vláken program pokračoval standardním způsobem
- potřebujeme si uložit stav registrů před spuštěním plánovače
- tento stav obnovíme po skončení všech vláken

Aktivace a deaktivace prostředí

```
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
```

Aktivace a deaktivace prostředí

- vytvoříme také globální proměnnou `uthread_mainthread_context` (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 `rip` uložíme adresu instrukce `ret`
- aktivujeme plánovač – nastavíme, že není aktivní žádné vlákno, provedeme skok do funkce `uthread_switch`, která vybere první vlákno určené k běhu
- jednotlivá vlákna se mezi sebou 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 pokračuje

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
- i přes to, synchronizační prostředky i zde dávají smysl
- ukážeme si, jak implementovat semafory s pasivním čekáním
- základem je struktura, která obsluhuje hodnotu semaforu a seznam vláken, které 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;
};
```

Synchronizace

Funkce

- Inicializace semaforu

```
void uthread_sem_init(uthread_sem_t *sem, int value) {
    sem->value = value;
    uthread_queue_init(&sem->blocked_threads);
}
```

- Snížení hodnoty semaforu

```
/** 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();
    }
}
```

Synchronizace
Funkce

- Zvýšení hodnoty semaforu

```
/** 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++;
}
```

Příklady

Příklad

Doplňte funkci `void uthread_set_priority(int)`, která aktuálnímu vláknu nastaví zadanou prioritu.

Příklad

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.

Příklad

(Volitelně) Implementujte lottery scheduler, který je postavený na tom, že každé vlákno bude mít přiděleno určité množství losů, ze kterých bude plánovač vybírat.

Příklad

Vyzkoušejte si prakticky, jak jsou vlákna přidělována různými plánovači.