# Systémové programovanie / ÚINF/SPR1b

SPR1a

Róbert Novotný

robert.novotny@upjs.sk

8. 3. 2011

založené na Advanced Linux Programming
http://www.advancedlinuxprogramming.com/

# Atribúty vlákna



- definujú nuansy týkajúce sa behu vlákna
  - detach state: správanie sa po dobehnutí
  - veľkosť a adresa stacku pre vlákno
  - vlastnosti pre plánovač vlákien
- pre väčšinu aplikácií je najdôležitejší **detach state**

# Vlákno a jeho stav po dobehnutí



- **joinable** („napojiteľné") vlákno:
  - po dobehnutí nie je upratané
    - v tomto stave je analogické zombie procesu
  - pomocou `pthread_join()` možno získať návratovú hodnotu...
  - ...a upratať
- **detached** („odpojené") vlákno
  - po dobehnutí je automaticky upratané
  - nemožno sa naň `join()`núť,
  - ani získať návratovú hodnotu

# Nastavenie atribútov vlákna



```
void * thread_function(void * thread_arg) {
  /* Pracujeme asynchronne... */
}
int main() {
  pthread_attr_t attr;
  pthread_t thread;
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr,
                              PTHREAD_CREATE_DETACHED);
  pthread_create(&thread, &attr, &thread_function, NULL);
  pthread_attr_destroy(&attr);
  /* Pracujme... */
  return 0;
}
```

štruktúra pre atribúty vlákna

použitie vo vytváraní

inicializácia štruktúry a nastavenie konkrétneho atribútu

# Kedy vlákno dobehne?



- normálne ukončenie (zvnútra):
  - vlákno vráti hodnotu z funkcie vlákna
  - vlákno zavolá pthread_exit()
- zrušenie zvonku (**cancel**)

pthread_cancel

  - parametrom je ID vlákna
  - joineable vlákno následne treba upratať join()om
  - návratovou hodnotou je PTHREAD_CANCELED

# Problém so zastrelenými vláknami



- **zrušené vlákno** môže po sebe nechať otvorené prostriedky
  - otvorené súbory
  - nedealokované premenné
- vlákna by sa nemali strieľať!
- v Java VM: vlákno nemôže byť zastrelené
  - metóda Thread#stop() je zastaralá

memory leak!

# Tri stavy vlákna



- **asynchrónne** zastaviteľné (async. cancelable)
  - môže byť zastrelené kedykoľvek
- **synchrónne** zastaviteľné (async. cancelable)
  - žiadosti o zastrelenie sú radené do frontu
  - ak vlákno dosiahne konkrétny bod v programe, je prerušené

implicitné správanie

- **nezastaviteľné** (uncancelable)
  - pokusy o zastavenie sú ignorované

# Nastavenie atribútov vlákna



```
void * thread_function(void * thread_arg) {
  /* Pracujeme asynchronne... */
  pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,
                        NULL);
}
```

pthread_setcanceltype

pointer na premennú, do ktorej sa vloží predošlý stav zastaviteľnosti

zavolanie ovplyvní vlákno, v rámci ktorého sme funkciu zavolali

PTHREAD_CANCEL_DEFERED: synchrónne zastaviteľné

# Body zastaviteľnosti v sync. cancellable



- synchrónne zastaviteľné vlákno sa môže odstreliť len v bodoch zastavenia
  - cancellation points
- klasickým bodom je volanie pthread_testcancel()
  - spracuje požiadavky na zastrelenie, ktoré stoja vo fronte
- **rada**: pravidelne volať pri tiahlych výpočtoch vo chvíľach, keď sa vlákno môže korektne ukončiť s korektným uzavretím zdrojov

# Body zastaviteľnosti v sync. cancellable



- niektoré systémové volania sú **bodmi zastavenia**
  - http://www.kernel.org/doc/man-pages/online/pages/man7/pthreads.7.html
- pozor na to, že vlákno môže byť v nich prerušené
  - a systémové zdroje ostatnú neuvoľnené!

# Zastaviteľnosť vlákna a kritické sekcie



- **kritická sekcia:** časť programu, ktorá narába so zdieľanými prostriedkami a má prebehnúť buď úplne alebo vôbec
- banková transakcia:
  - odčítame peniaze z jedného účtu
  - pripočítame na druhý účet
- odstrelenie vlákna po prvej operácii spôsobí smútok u oboch klientov banky

kritická sekcia!

# Riešenie kritickej sekcie



```
float * stavy_na_uctoch;
int prevod(int ucet, int ucet_adresata, float suma) {
  int predosla_zastavitelnost;
  if(stavy_na_uctoch[ucet] < suma) return 1;
  pthread_setcancelstate(PTHREAD_CANCEL_DISABLE,
                         &predosla_zastavitelnost);
  stavy_na_uctoch[ucet_adresata] += suma;
  stavy_na_uctoch[ucet] -= suma;
  pthread_setcancelstate(predosla_zast vitelnost, NULL);
  return 0;
}
```

kritická sekcia!

# Riešenie kritickej sekcie



```
float * stavy_na_uctoch;
int prevod(int ucet, int ucet_adresata, float suma) {
  int predosla_zastavitelnost;
  if(stavy_na_uctoch[ucet] < suma) return 1;
  pthread_setcancelstate(PTHREAD_CANCEL_DISABLE,
                         &predosla_zastavitelnost);
  stavy_na_uctoch[ucet_adresata] += suma;
  stavy_na_uctoch[ucet] -= suma;
  pthread_setcancelstate(predosla_zastavitelnost, NULL);
  return 0;
}
```

na začiatku kritickej sekcie vyhlásime vlákno na nezastreliteľné
na konci KS obnovíme stav zastaviteľnosti

# Nastavenie zastaviteľnosti



pthread_setcancelstate

- na konci kritickej sekcie je nutné obnoviť pôvodný stav zastaviteľnosti
- zaistíme tým, že kritickú sekciu možno volať z inej kritickej sekcie
  - ak stav neobnovíme, funkcia, ktorá volá našu kritickú sekciu, bude tiež nezastaviteľná
  - v extréme: spôsobíme, že celé vlákno bude nezastaviteľné

# Synchronizácia a kritické sekcie



- plánovanie vlákien je **nedeterministické**
  - nedá sa spoľahnúť na to, že vlákno X pobeží skôr než vlákno Y
- veľmi častý zdroj chýb: modifikácia zdieľaných dát
  - vlákna totiž zdieľajú spoločný adresný priestor
- **race condition**: chyba, keď vlákna „súťažia" o modifikáciu zdieľaných dát
  - chyby sú ťažko reprodukovateľné a laditeľné

# Riešenie kritickej sekcie



```
void * thread_function(void * arg) {
  while(front_uloh != NULL) {
    /* Vytiahneme z frontu dalsiu ulohu. */
    struct job * uloha = front_uloh;
    /* Odstranime ju z frontu. */
    front_uloh = front_uloh->dalsi;
    /* spracujeme ju a upraceme */
    spracuj_ulohu(uloha);
    free(uloha);
  }
  return NULL;
}
```

kód funguje pri jednom vlákne bez chýba

```
struct job {
  struct job * dalsi;
  /* dalsie atributy pre ulohu... */
};
struct job * front_uloh;
```

globálna premenná

| Vlákno 1 | Vlákno 2 |
| --- | --- |
| `while(front_uloh != NULL)` | |
| `struct job * uloha = front_uloh;` | |
| | `while(front_uloh != NULL)` |
| | `struct job * uloha = front_uloh;` |
| | `front_uloh = front_uloh->dalsi;` |
| `front_uloh = front_uloh->dalsi;` | |
| `spracuj_ulohu(uloha);` | |
| `free(uloha);` | |
| | `spracuj_ulohu(uloha);` |
| | `free(uloha);` |

Lokálne premenné **uloha** ukazujú na to isté miesto v pamäti

SEGFAULT!
Pamäť v premennej **uloha** bola uvoľnená prvým vláknom

| Vlákno 1 | Vlákno 2 |
| --- | --- |
| `while(front_uloh != NULL)` | |
| `struct job * uloha = front_uloh;` | |
| `front_uloh = front_uloh->dalsi;` | |
| | `while(front_uloh != NULL)` |
| | `struct job * uloha = front_uloh;` |
| | `front_uloh = front_uloh->dalsi;` |
| | |
| `spracuj_ulohu(uloha);` | |
| `free(uloha);` | |
| | `spracuj_ulohu(uloha);` |
| | `free(uloha);` |

nech je vo fronte už len jedna úloha

**front_uloh** bude null

SEGFAULT!
null->dalsi!

# Race conditions



- chyba sa prejaví v závislosti od poradia, v ktorom pobežia vlákna
  - podľa toho, ako thread scheduler strieda vlákna
  - niekedy sa neprejaví vôbec, inokedy pravidelne
- reprodukovateľnosť takmer nemožná

Návrh na riešenie?
Potrebujeme zaistiť atomicitu operácií!

- atomickú akciu nemožno zrušiť uprostred behu
- nemožno ju ani pozastaviť, aby zbehla iná akcia

# Podstata problému z príkladu



- obe vlákna pristupujú k **zdieľanému** prostriedku **súčasne**

Riešenie? Zamykanie!

- zdieľaný prostriedok: WC misa
- riešenie: **zamknem** sa na toalete, použijem WC misu, **odomknem** sa
- ak chce niekto použiť obsadenú misu, musí ísť do frontu a **čakať**

# Implementácia vo svete vlákien: mutexy



- **mutex** = mutual exclusion lock
  - zámok so vzájomným vylúčením
- zámok, ktorý môže byť uzamknutý v danom čase len jedným vláknom
- pokus o **druhé uzamknutie** zamknutého zámku vyústi v čakanie (**blokovanie**)
- blokovanie trvá až dovtedy, kým sa zámok neodomkne

# Práca s mutexami



- `pthread_mutex_t`: dátový typ pre mutex
- inicializácia:

pthread_mutex_init()

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
```

jednoduchšia inicializácia

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

# Front úloh s mutexom



```
pthread_mutex_t mutex_frontu = PTHREAD_MUTEX_INITIALIZER;

void * thread_function(void * arg) {
  while(1) {
    struct job * uloha;
    pthread_mutex_lock(&mutex_frontu);
    tu pracujeme so zdieľaným prostriedkom
    pthread_mutex_unlock(&mutex_frontu);
    spracuj_ulohu(uloha);
    free(uloha);
  }
  return NULL;
}
```

zamkneme mutex

odomkneme mutex

# Front úloh s mutexom



```
pthread_mutex_lock(&mutex_frontu);

if (front_uloh == NULL) {
  uloha = NULL;
} else {
  uloha = front_uloh;
  front_uloh = front_uloh->dalsi;
}
pthread_mutex_unlock(&mutex_frontu);

if (uloha == NULL) {
  break;
}
```

Front je prázdny, niet čo robiť.

vytiahneme ďalší front

ak niet čo robiť, končíme

# Mutex chráni zdieľaný prostriedok!



- zdieľaným prostriedkom v príklade je front úloh
- prístupy, ktoré **pristupujú** zdieľaný prostriedok, musia byť v kritickej sekcii
  - buď čítajú
  - alebo zapisujú
- kritická sekcia je uvedená pomocou `pthread_mutex_lock`
- ukončená `pthread_mutex_unlock`
  - ak zistíme, že front je prázdny a niet čo robiť, nezabudnime odomknúť mutex!

# Príklad: zaradenie úlohy do frontu



```
void front_zaraď(struct job * nova_uloha) {
  pthread_mutex_lock(&mutex_frontu);

  nova_uloha->dalsi = front_uloh;
  front_uloh = nova_uloha;

  pthread_mutex_unlock(&mutex_frontu);
}
```

kritická sekcia!

klasická práca so spájaným zoznamom

```
struct job {
  struct job * dalsi;
  /* dalsie atributy pre ulohu... */
};
struct job * front_uloh;
```

# Mutexy v Jave



- mutexy v Jave majú podporu v syntaxi jazyka
- metódu možno vyhlásiť za kritickú sekciu

```
public class FrontUloh {
  private Queue<Uloha> ulohy = new LinkedList<Uloha>();

  public void synchronized zarad(Uloha novaUloha) {
    ulohy.offer(novaUloha);
  }
}
```

celá metóda je kritickou sekciou

**zdieľaný prostriedok:** celá inštancia triedy!

# Mutexy v Jave



- mutexy v Jave majú podporu v syntaxi jazyka
- **synchronized** blok ohraničuje kritickú sekciu

```
public class FrontUloh {
  private Queue<Uloha> ulohy = new LinkedList<Uloha>();

  public void zarad(Uloha novaUloha) {
    synchronized(ulohy) {
      ulohy.offer(novaUloha);
    }
  }
}
```

kritická sekcia

**zdieľaný prostriedok:** front úloh

# Deadlocky v mutexoch



- **deadlock**: vlákna čakajú na situáciu, ktorá nikdy nenastane

> *„Ak sa na križovatke stretnú dve súpravy, obe sú povinné zastaviť. Súprava môže pokračovať v jazde až po tom, čo druhá súprava opustí križovatku.“*
>
> ***-- nelogický železničný zákon v Kansase***

- čo sa stane, ak sa jedno vlákno pokúsi dvakrát zamknúť ten istý zámok?

# Tri druhy mutexov v Linuxe



| | |
|---|---|
| fast | • efektívny, ale duplicitné zamykanie vedie k deadlockom |
| recursive | • zamknutia a odomknutia musia byť v páre |
| error-checking | • duplicitné zamykanie vráti chybu |

# Tri druhy mutexov v Linuxe



- **fast mutex**
  - dvojnásobné zamknutie tým istým vláknom vedie k deadlocku
  - pokus o zamknutie už zamknutého mutexu spôsobí čakanie
  - vlákno čaké samé na seba => deadlock!
- implicitný druh mutexu v Linuxe
  - konštanta PTHREAD_MUTEX_NORMAL

# Nastavenie typu mutexu



```
pthread_mutexattr_t atributy_mutexu;
pthread_mutex_t mutex;

pthread_mutexattr_init(&atributy_mutexu);

pthread_mutexattr_setkind_np(&atributy_mutexu,
                             PTHREAD_MUTEX_NORMAL);

pthread_mutex_init(&mutex, &atributy_mutexu);

pthread_mutexattr_destroy(&atributy_mutexu);
```

uvoľníme pamäť pre atribúty mutexu

# Tri druhy mutexov v Linuxe



- **recursive mutex**
  - pamätá si, koľkokrát na ňom vlákno držiace zámok zavolalo uzamknutie
  - každé uzamknutie musí byť spárované s odomknutím
  - inak sa zámok neodomnkne
  - nevedie k deadlocku
- konštanta PTHREAD_MUTEX_RECURSIVE_NP

# Tri druhy mutexov v Linuxe



- **error-checking mutex**
  - druhé uzamknutie tým istým vláknom vráti z metódy pthread_mutex_lock() chybu EDEADLK
  - nevedie k deadlocku
- konštanta PTHREAD_MUTEX_ERRORCHECK_NP

užitočné pri ladení

**Pozor!**
Konštanty a pthread_mutexattr_setkind_np() sú neportovateľné!

# Ako zistiť, či je zamknuté?



**pthread_mutex_trylock**

- na otvorenom mutexe je ekvivalentný `pthread_mutex_lock`()u
  - vráti nulu
- na uzamknutom mutexe vráti **EBUSY** a nebude blokovať

**Knock, knock!**

# Semafory a producent-konzument



- ešte raz príklad s frontom úloh!
- po vybratí poslednej úlohy z frontu vlákno dobehne
- problém **producent-konzument**
  - jedno vlákno plní front
  - druhé z neho vyberá
- front musíme plniť dostatočne rýchlo na to, aby vyberajúce vlákno nedobehlo...
  - ...a ak všetky vyberajúce vlákna dobehnú, front sa začne neobmedzene plniť

# Producent-konzument: metafora skladu



- **metafora skladu**: sklad neobmedzenej veľkosti
- **producent**: vkladá do skladu tovar
- **konzument**: vyberá zo skladu tovar a spracováva ho
- ak sa sklad vyprázdni, konzument musí **čakať**

**Riešenie:**
semafor!

# Semafory ako synchronizačný mechanizmus



- **počítadlo s nenulovou hodnotou**, pomocou ktorého synchronizujeme beh vlákien
- sada povoleniek prístupu k prostriedku
  - nulová hodnota: vlákna stoja
  - kladná hodnota **N**: k prostriedku možno ihneď pristúpiť **N** krát (bez čakania)
- **analógia**: semafor = počet voľných okienok v banke

Toto nie sú semafory z IPC / System V!
Tie majú úplne iný význam!

# Semafor


| Operácia POST | Operácia WAIT |
|---|---|
| Inkrementuje počítadlo | Dekrementuje počítadlo |
| Ak inkrementujeme z nuly na jednotku, jedno z **čakajúcich** vlákien sa odblokuje a vykoná svoj WAIT | Ak chceme dekrementovať nulu, musíme počkať, kým počítadlo nebude kladné. |

- Linux garantuje, že zmena počítadla semaforu je vláknovo bezpečná i bez mutexu
  - semafor je zdieľaný prostriedok
    - paralelný prístup by za normálnych okolností viedol k race conditions!

# Použitie semafora vo fronte úloh



- použitie 2: front úloh spracovávaný vláknami
  - ukážeme o chvíľu
- špeciálny prípad: **binárny** semafor
  - počítadlo, ktoré je buď 0 alebo 1
  - funguje ako klasický mutex
  - môže ho však odomknúť aj iné vlákno než to, ktoré ho zamklo

# Použitie semafora v C



| | |
|---|---|
| `sem_t` | dátový typ pre semafor |
| `sem_init(semafor, typ, iniciálna_hodnota)` | inicializácia semafora |
| `sem_wait` | operácia WAIT |
| `semn_post` | operácia POST |
| `sem_trywait` | analogické k pthread_mutex_trylock |
| `sem_getvalue` | aktuálna hodnota semafora |

**<semaphore.h>**

# Použitie semafora vo fronte úloh



- semafor je globálna premenná a obsahuje počet úloh vo front

```
void zarad_ulohu(struct job * nova_uloha) {
  pthread_mutex_lock(&mutex_frontu);

  nova_uloha->next = front_uloh;
  front_uloh = nova_uloha;

  sem_post(&semafor_pocet_uloh);

  pthread_mutex_unlock(&mutex_frontu);
}
```

kritická sekcia

Zvýšime počítadlo v semafore. Jedno z čakajúcich vlákien sa odblokuje.

# Použitie semafora vo fronte úloh



```
void * thread_function(void * arg) {
  while(1) {
    struct job * uloha;
    sem_wait(&semafor_pocet_uloh);
    pthread_mutex_lock(&mutex_frontu);
    uloha = front_uloh;
    front_uloh = front_uloh->dalsi;
    pthread_mutex_unlock(&mutex_frontu);
    spracuj_ulohu(uloha);
    free(uloha);
  }
  return NULL;
}
```

kritická sekcia

Vďaka semaforu vieme, že front sa nikdy nevyprázdni.
WAIT čaká, kým sa vo fronte nezjaví aspoň 1 úloha.

# Rekapitulácia



## sekcia „prejde všetko alebo nič"

- vlákno nastavím ako nezastaviteľné, vykonám čo treba, a vlákno vyhlásim za zastaviteľné

## k prostriedku môže pristúpiť **jediné vlákno**

- zamknem mutex, vykonám, čo treba, odomknem mutex

## k prostriedku môže pristúpiť **N** vlákien

- nastavím semafor na **N**, po každom prístupe dekrementujem **N**, po skončení prístupu zvýšim **N**

# Condition variables: premenné podmienky



- úloha: vlákno chce čakať na konkrétnu hodnotu v zdieľaných dátach
- alebo: vlákno má bežať len vtedy, ak zdieľané dáta spĺňajú nejakú **podmienku**
- **condition variable:** premenná s podmienkou

# Condition variables: drevorubačské riešenie

# Premenné podmienky (príznaky)



- **condition variable:** premenná obsahujúca podmienku, pri ktorej vlákno beží
- úloha: vlákno v cykle, ktoré beží len vtedy, keď je splnená podmienka
- riešenie č. 1:
  - cyklíme sa
  - v každej iterácii otestujeme príznak
    - príznak je chránený mutexom

# Použitie semafora vo fronte úloh



```
int príznak;
pthread_mutex_t príznak_mutex;

void * thread_function(void * thread_arg) {
  while(1) {
    int príznak_zapnutý;
    pthread_mutex_lock(&príznak_mutex);
    príznak_zapnutý = príznak;
    pthread_mutex_unlock(&príznak_mutex);
    if(príznak_zapnutý) {
      pracuj();
    }
  }
  return NULL;
}
```

čítanie zo zdieľaného zdroja

na mnohých architektúrach je čítanie z globálnej volatile primitívnej premennej atomické

# Použitie semafora vo fronte úloh



```
void inicializuj_príznak() {
  pthread_mutex_init(&príznak_mutex, NULL);
  príznak = 0;
}
```

```
void nastav_príznak(int hodnota) {
  pthread_mutex_lock(&príznak_mutex);
  príznak = hodnota;
  pthread_mutex_unlock(&príznak_mutex);
}
```

zápis do zdieľaného zdroja

# Príznak ako mechanizmus synchronizácie



- uvedený príklad je korektný, ale nie je efektívny
- aktívne čakanie žerie čas CPU
  - dokola testujeme zapnutie príznaku
  - plus otvárame a zatvárame mutex
- tzv. „thread spinning" alebo aktívne čakanie

# Príznak ako mechanizmus synchronizácie



- príznak môže plniť rolu binárneho semaforu
- zelené vlákno môže čakať na príznak
  - t. j. je blokovať dovtedy, kým modré vlákno príznak nezapne
  - čakanie realizujeme cyklickým testovaním premennej
- príznak nemá pamäť ani počítadlo
  - zelené vlákno musí začať čakať skôr než modré vlákno zapne príznak, inak sa signal „zapnutia" môže stratiť

# Čakanie na konkrétnu hodnotu premennej



- použijeme dve globálne premenné:
  - **príznak**: určený na signalizáciu behu / blokovania
  - **condition variable** obsahujúca hodnotu, na ktorú sa bude čakať
- v thread function môžeme v kritickej sekcii blokovať dovtedy, kým sa nesplní podmienka
  - v kritickej sekcii zeleného vlákna bude cyklus s testom na hodnotu premennej
  - v kritickej sekcii modrého vlákna nastavíme hodnotu premennej a zapneme príznak

Race condition!

# Ako sa to rieši reálne?



- potrebujeme zamknúť príznak i podmienkovú premennú jediným mutexom
- zabudovaný mechanizmus:
  1. zamkneme mutex a prečítame príznak
  2. ak je príznak nastavený, odomkneme mutex a spracujeme dáta
  3. v opačnom prípade odomkneme mutex a čakáme na splnenie podmienkovej premennej

tretí krok sa môže udiať atomicky

# Atomické odomknutie mutextu



- atomické odomknutie odomknutie mutexu a čakanie je dôležité!
- v opačnom prípade ukážka chyby:
  - vlákno 1 zistí, že príznak je true
  - vlákno 2 zmení príznak na false a nastaví inú hodnotu podmienkovej premennej
  - vlákno 1 bude čakať na hodnotu podmienkovej premennej, ktorá sa nikdy nemusí objaviť.