Operační systémy

Procesy a vlákna. Časově závislé chyby. Kritické sekce.

Jan Trdlička

České vysoké učení technické v Praze, Fakulta informačních technologií
Katedra počítačových systémů

https://courses.fit.cvut.cz/BI-OSY

Obsah přednášky

Program(aplikace)

- Program je v systému reprezentován spustitelným binárním programem, který je uložený v sekundární paměti (např. disk).
- **Spustitelný binární program**
  - Formát je závislý na OS, pro který je zkompilovaný.
    - ★ Executable and Linkable Format (ELF) pro OS unixového typu,...
    - ★ Portable Executable format (PE/PE32+) pro MS Windows,...
  - Obsahuje
    - ★ TEXT = spustitelný binární kód,
    - ★ DATA = proměnné a jejich hodnoty,...
    - ★ další informace (např.o sdílených knihovnách,...).

Příklad: Určení formátu a informace o knihovnách

```
linux:~> file /usr/bin/date
/usr/bin/date: ELF 64-bit LSB shared object, x86-64, ...,
dynamically linked, ..., stripped
```

```
linux:~> ldd /usr/bin/date
        linux-vdso.so.1 (0x00007ffd66fc3000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f6e3c5b1000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f6e3cb84000)
```

Proces

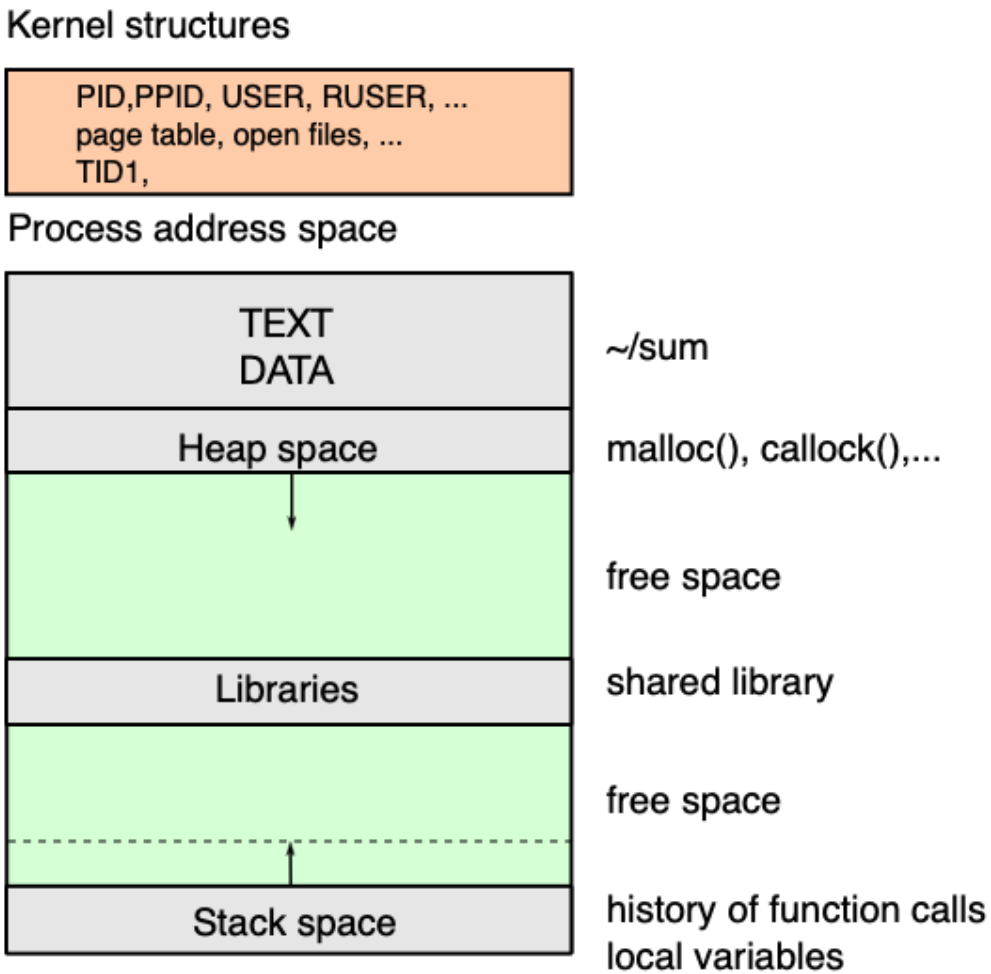
- Instance spuštěného programu/aplikace.
- Entita, v rámci které jsou alokovány prostředky (paměť, vlákna, otevřené soubory, zámky, semafory, sokety,...).
- Implicitně každý proces má jedno výpočetní "main" vlákno.
- Jádro OS si pro každý proces udržuje celou řadu datových struktur nezbytných pro
  - identifikaci: číslo procesu(PID), číslo rodič. procesu(PPID),...
  - bezpečnost: identita procesu (USER, RUSER),...
  - správu paměti: informace pro překlad virt. adres(page table),...
  - správu FS: tabulka deskriptorů souborů,...

  ...
- Při vzniku nového procesu, část datových struktur je zděděna od rodičovského procesu (tabulka deskriptorů souborů,...) a část je nastavena na nové hodnoty, které jsou specifické pro nový proces (číslo procesu,...).

Proces s jedním vláknem

```c
#include <stdio.h>

int sum(int a, int b)
{
  int s;
  s = a + b;
  return (s);
}

int main ( int argc, char * argv[ ] )
{
  int a, b;
  sscanf ( argv[1], "%d", &a);
  sscanf ( argv[2], "%d", &b);

  printf("Sum = %d\n", sum(a,b));

  return 0;
}
```

Program: sum.c



Process: ~/sum

Proces s jedním vláknem

Příklad: Zobrazení adresového prostoru procesu s číslem 5652 v OS unixového typu.

```
solaris:~> pmap -x 5652
5652:   -bash
         Address      Kbytes        RSS       Anon     Locked Mode   Mapped File
0000000100000000        1000       1000          -          - r-x----  bash
00000001001FA000          48         48         16          - rwx----  bash
0000000100300000          16         16         16          - rw-----  bash
000000CC71630000         256        256        192          - rw-----    [ heap ]
00007CBD99000000         232        208          -          - r-x----  libcurses.so.1
00007CBD9913A000          32         32          -          - rwx----  libcurses.so.1
00007CBD99142000          16         16          -          - rwx----  libcurses.so.1
00007CBD99200000          32         24          -          - r-x----  libgen.so.1
00007CBD99308000           8          8          -          - rwx----  libgen.so.1
00007CBD99400000          64         64          -          - rwx----    [ anon ]
00007CBD99500000          64         64          -          - rwx----    [ anon ]
00007CBD9952A000           8          8          -          - rwxs---    [ anon ]
00007CBD99600000          24         16          8          - rwx----    [ anon ]
00007CBD99700000        1560       1288          -          - r-x----  libc.so.1
00007CBD99986000          64         64         16          - rwx----  libc.so.1
00007CBD99996000           8          8          -          - rwx----  libc.so.1
00007CBD99A00000          64         64         64          - rw-----    [ anon ]
00007CBD99A30000           8          8          -          - r--s---    [ anon ]
00007CBD99B00000         272        272          -          - r-x----  ld.so.1
00007CBD99C44000          24         24         16          - rwx----  ld.so.1
FFFFFD6DA3C10000          56         56          8          - rw-----    [ stack ]
---------------- ---------- ---------- ---------- ----------
        total Kb        3856       3544        336          -
```

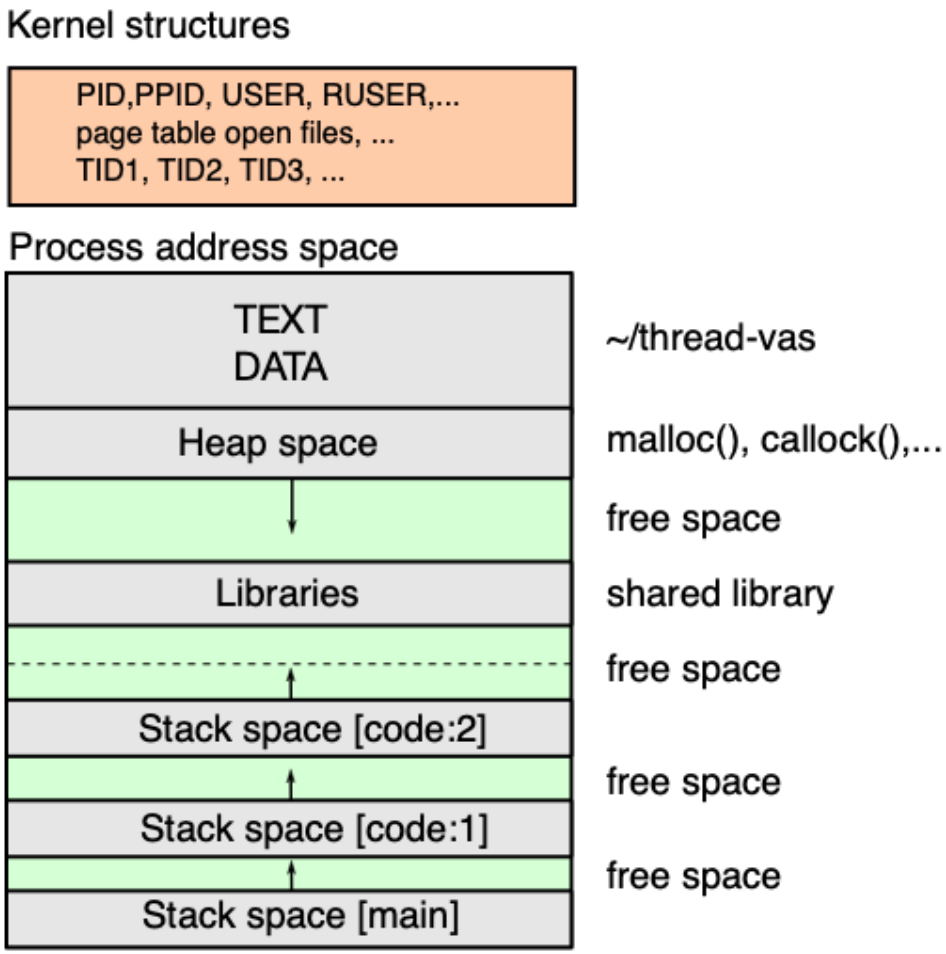
Vlákno

- Výpočetní entita (proud instrukcí), které je přidělováno jádro CPU.
- Historicky se vlákno nazývalo "light-weight process" (lwp).
- Vlákna vytvořená v rámci procesu sdílí většinu prostředků alokovaných v tomto procesu.
- Jádro OS si pro každé vlákno udržuje celou řadu datových struktur nezbytných pro
  - identifikaci: číslo vlákna(TID),...
  - zásobník: lokální proměnné, historie volání,...
  - informace nutné pro přepínání kontextu: čítač instrukcí, aktuální hodnoty registrů,...
  - informace pro plánování vláken: priorita, čas strávený na CPU,...
  - ...

Proces s více vlákny

```c
#include <pthread.h>
. . .
void *code(void * dummy)
{
  printf("Thread: Hello.\n");
  sleep(60);
  return(NULL);
}

int main ( int argc, char * argv[] )
{
  void  *dummy;
  pthread_t  tid1;
  pthread_t  tid2;

  pthread_create(&tid1, NULL, &code, NULL);
  pthread_create(&tid2, NULL, &code, NULL);

  pthread_join(tid1, &dummy);
  pthread_join(tid2, &dummy);

  return 0;
}
```

Program: thread-vas.c



Process: ~/thread-vas

Vytvoření nového procesu

- Nový proces se vytvoří, když existující proces zavolá příslušné systémové volání
  - v OS unixového typu: `fork(2)`, `execve(2)`,...
  - v MS Windows: `CreateProcessA()`,...
- Nově vzniklý proces může představovat
  - kopii/klon původního procesu (např. po zavolání `fork(2)`)
    - ★ Jádro alokuje nové dat. struktury pro nový proces (část z nich bude nově zinicializována, část bude kopií od rodiče).
    - ★ Adresový prostor procesu bude zkopírován od rodiče (TEXT, DATA, zásobník, halda, ...).
    - ★ Čítač instrukcí bude ukazovat na následující instrikci za `fork(2)`.
  - úplně nový proces (např. po zavolání `fork(2)` a `execve(2)`)
    - ★ Jádro alokuje nové dat. struktury pro nový proces (část z nich bude nově zinicializována, část bude kopií od rodiče).
    - ★ Adresový prostor procesu bude nově zinicializován (prázdný zásobník, halda, ...) a TEXT, DATA, knihovny budou načteny ze souboru.
    - ★ Čítač instrukcí bude ukazovat na první instrukci programu.

Příklad: Vytvoření nového procesu v Unixu

```
1  ...
2  int main()
3  {
4     pid_t pid;
5     int   status;
6
7     pid = fork();
8
9     switch (pid) {
10
11       case -1: /* Error */
12          exit(1);
13
14       case  0: /* Child process */
15          char* argv[] = { "sleep", "30", NULL };      /* array of argument strings    */
16          char* envp[] = { NULL };                     /* array of environment strings */
17          execve("/bin/sleep", argv, envp);
18          exit(0);
19
20       default: /* Parent process */
21          wait(&status);
22    }
23
24    return 0;
25 }
```

Příklad: Vytvoření nového procesu v Unixu

`fork()`

- Vytvoří nový proces, který je kopií procesu, z kterého byla tato funkce zavolána.
- Funkce vrací
  - v rodičovském procesu číslo potomka (v případě chyby -1),
  - v potomkovi vždy hodnotu 0.
- Po návratu z funkce, rodič i potomek pokračuje na následující instrukci a běží na sobě "nezávisle".

`execve(const char *filename,...)`

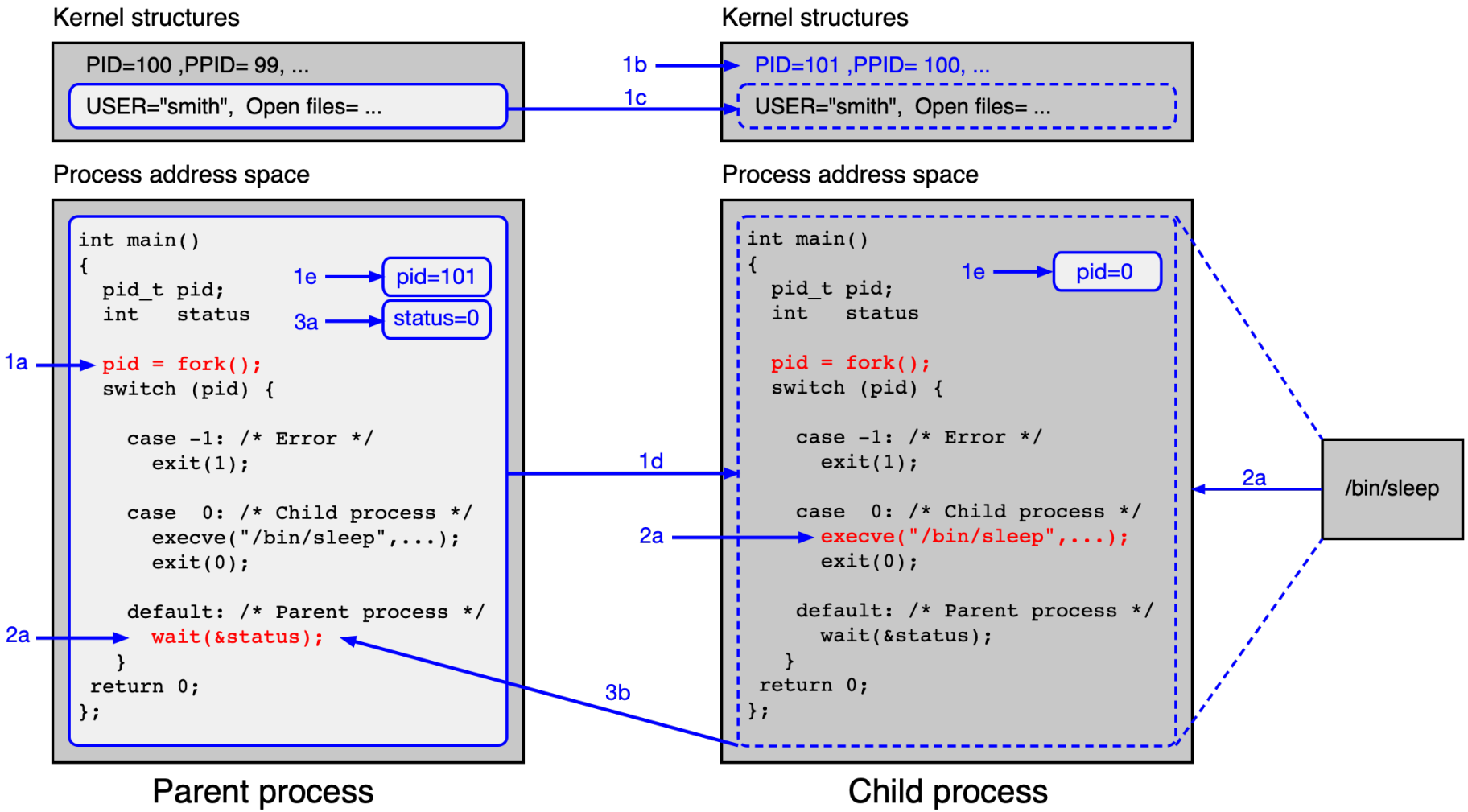
- Adresový prostor procesu, ze kterého je funkce volána, je přepsán obsahem souboru, který se začne vykonávat od začátku.

`wait(int *status)`

- Funkce zablokuje rodičovský proces, ve kterém je zavolána, dokud se jeden jeho potomek neukončí.
- Varianta `waitpid(2)` umožňuje čekat na konkrétního potomka.

Více informací viz. `man 2 fork`, `man 2 execve`, `man 2 wait`.

Příklad: Vytvoření nového procesu v Unixu



Příklad: Vytvoření nového procesu v Unixu

- Komentář k předchozímu obrázku
  - 1a Volání `fork()` v rodiči způsobí alokování dat. struktur v jádře a paměti pro nový proces.
  - 1b Do některých datových struktur nového procesu jsou nastaveny nové hodnoty.
  - 1c Do dalších datových struktur nového procesu jsou zkopírovány hodnoty z rodiče.
  - 1d Obsah adresového prostoru rodiče se "zkopíruje" do adresového prostoru potomka.
  - 1e Při návratu z `fork()` se do proměnné `pid`
    - ⋆ v rodiči uloží číslo potomka a pokračuje se následující instrukci,
    - ⋆ v potomkovi uloží hodnota 0 a pokračuje se následující instrukci.
  - 2a
    - ⋆ Rodič bude čekat na dokončení potomka pomocí volání `wait()`.
    - ⋆ Potomek načte nový program do svého adresového prostoru pomocí volání `execve()`.
  - 3a Po ukončení potomka (např. pomocí funkce `exit()` nebo `return`) bude návratový kód (parametr těchto funkcí) uložen v rodiči do proměnné `status`.
  - 3b Rodič se vráti z funkce `wait()` a pokračuje následující instrukcí.

Ukončení procesu

- **Jádro OS při ukončení procesu**
  - se pokusí předat "návratový kod" rodiči,
  - ukončí všechna vlákna, která existují v rámci daného procesu,
  - uvolní adresový prostor procesu a příslušné dat. struktury v jádře související s daným procesem.
- **Varianty ukončení procesu**
  - Proces se ukončí sám
    - pokud dojde na konec programu (např. `return` v "main" vláknu),
    - zavolá příslušnou knihovní funkci (např. `exit(3)`, `TerminateProcess()`, ...).
  - Proces je ukončen jádrem
    - pokud dojde k fatální chybě (např. dělní nulou, špatná manupilace s pamětí,...),
    - na základě např. přijatého signálu,...

Příklad: Vytváření nových procesů

- Kolik vznikne celkem procesů, pokud spustíme následující program?
- Kdy se ukončí poslední z nich?

```
1  ...
2  int main()
3  {
4     pid_t pid;
5     int   i;
6
7     for ( i = 0; i < 3; i++ )
8     {
9        pid = fork();
10
11       if  ( pid == 0 )
12       {
13          sleep(10);
14       }
15    }
16
17    return 0;
18 }
```

Vytvoření a ukončení vlákna

- Procesy se implicitně vytváří s jedním "main" vláknem.
- Pokud chceme vytvořit v rámci procesu další vlákna, pak můžeme použít
  - OS API/knihovny
    - ★ proprietární knihovny jednotlivých OS (Microsoft Windows API, Solaris thread library,...),
    - ★ POSIX thread library (MS Windows, GNU/Linux, FreeBSD, ...),
    - ★ OpenMP (MS Windows, GNU/Linux, FreeBSD, Solaris, ...),...
  - Programovací jazyky s vestavěnou podporou vláken
    - ★ C++ (od C++11), Java,...

Příklad: POSIX vlákna

```
1  #include <pthread.h>
2  . . .
3  void *start_routine(void * dummy) {
4     printf("Thread: Hello.\n");
5     sleep(60);
6     return(NULL);
7  }
8
9  int main ( int argc, char * argv[] ) {
10    void  *dummy;
11    pthread_t  tid1, tid2;
12    . . .
13    /* Creating threads */
14    pthread_create(&tid1, NULL, &start_routine, NULL);
15    pthread_create(&tid2, NULL, &start_routine, NULL);
16
17    /* Waiting for threads */
18    pthread_join(tid1, &dummy);
19    pthread_join(tid2, &dummy);
20    . . .
21    return 0;
22 }
```

Příklad: POSIX vlákna

```
pthread_create(pthread_t *tid, ...,
        void *(*start_routine) (void *),...)
```

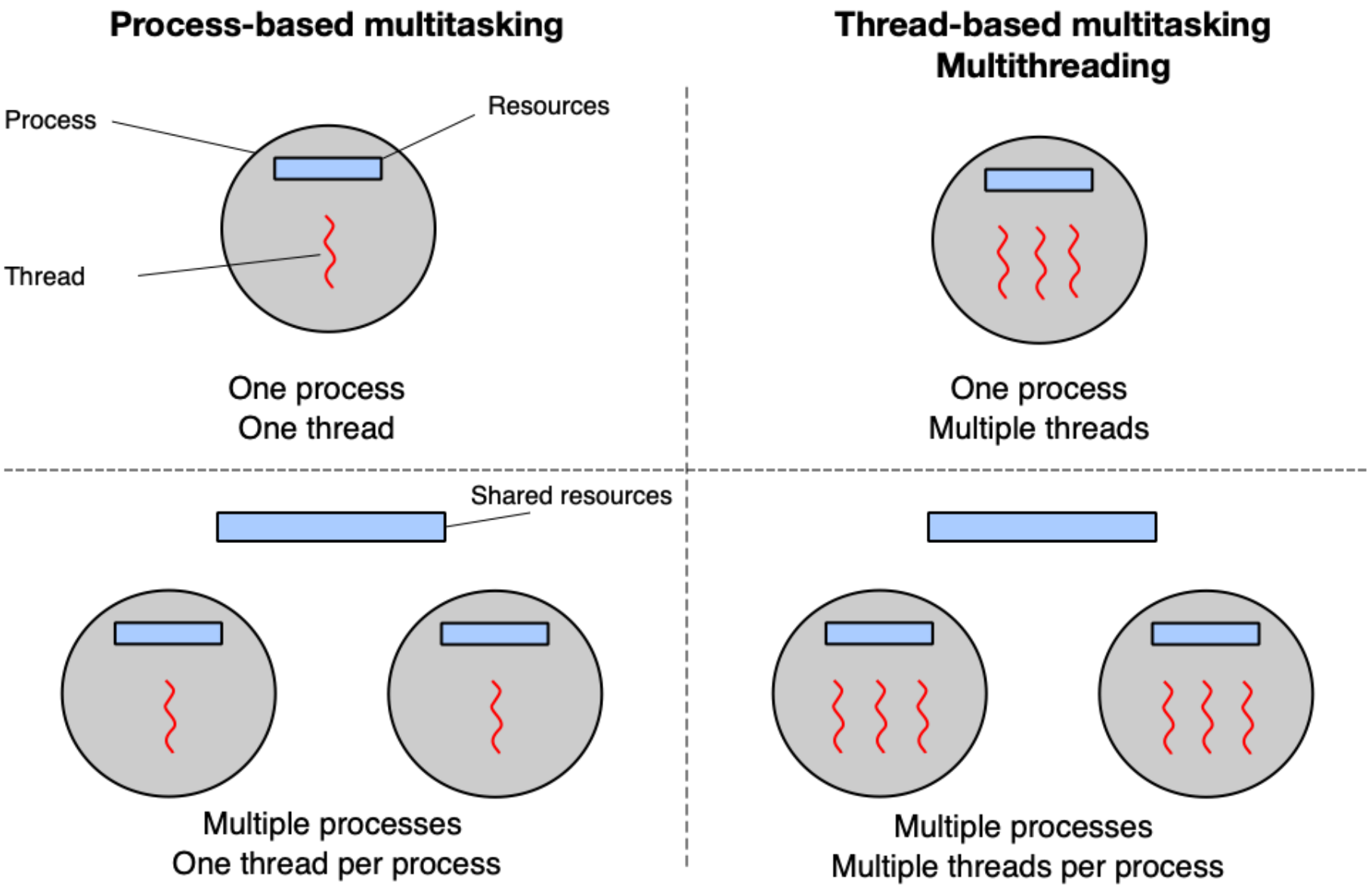
- Funkce v aktuálním procesu vytvoří nové vlákno, které bude reprezentováno funkcí `start_routine()` a na adresu definovanou prvním parametrem `tid` uloží číslo nového vlákna.

```
pthread_join(pthread_t tid,...)
```

- Vlákno, ze kterého tuto funkci zavoláme, bude čekat na dokončení vlákna, které je se specifikováno prvním parametrem `tid`.

Bližší informace viz. manuálové stránky: `man pthread_create` a `man pthread_join`.

Multitasking/Multithreading



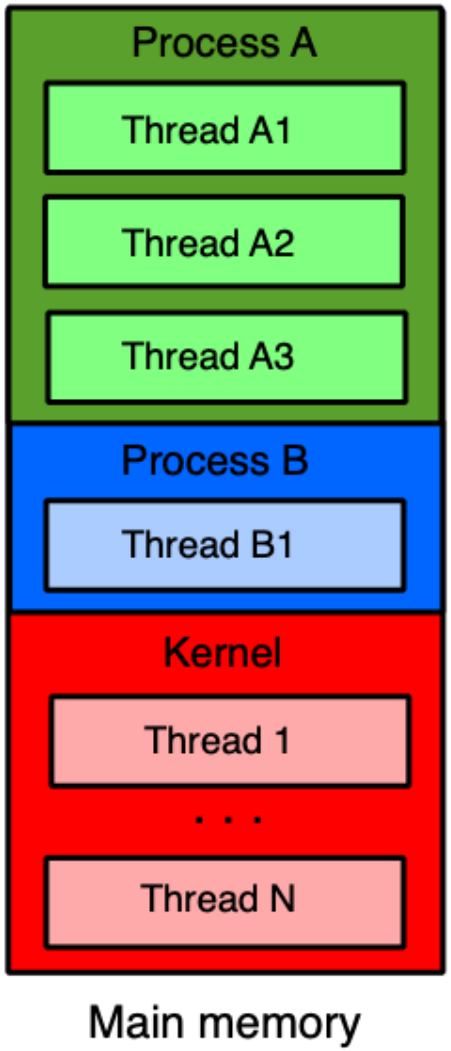
Plánování vláken

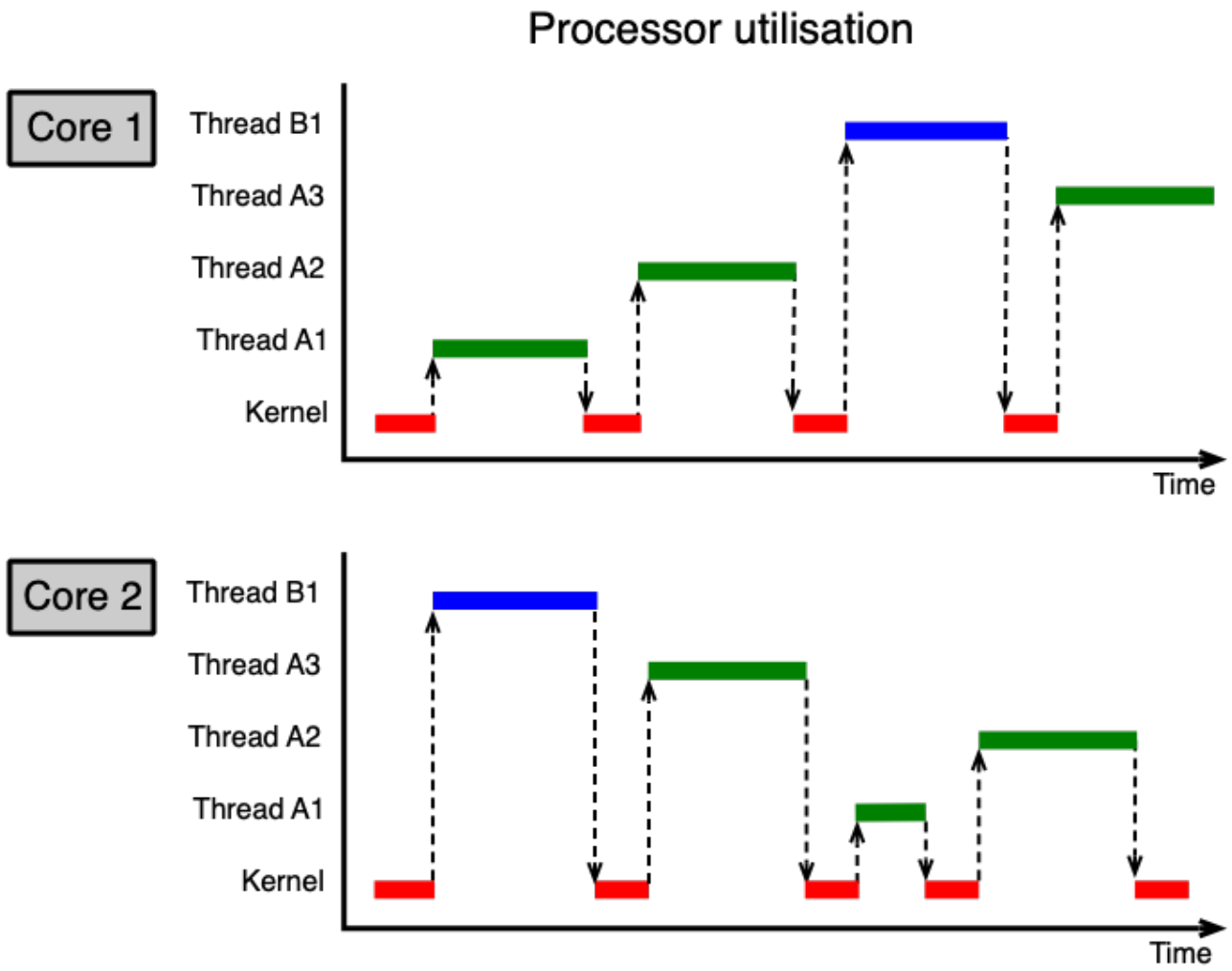
- Jádro OS a vytvořená vlákna sdílí omezený počet jader výpočetního systému.
- Jedno vlákno (instrukční proud) může být v jednom okamžiku zpracováváno jedním "logickým" jádrem CPU.
- Aby se vlákna "rozumným" způsobem podělila o omezený počet jader, tak se vlákna na jádrech střídají nejčastěji pomocí preemptivního plánování.
- **Preemptivní plánování vláken**
  - Vláknu je přiděleno volné jádro CPU, pokud ho jádro OS vybere na základě plánovacích kriterií (např. priority,...).
  - Jádro OS vláknu přidělí časové kvantum, během kterého vlákno bude zpracováváno jádrem CPU.
  - Vláknu je jádro CPU odebráno pokud
    - dojde k uplynutí časového kvanta (přerušení od časovače),
    - vlákno provede systémové volání (např. V/V operaci),
    - dojde k přerušení (např. je dokončena V/V operace,...).

Přepínání kontextu

- **Přepínání kontextu**
  - Je to mechanismus, při kterém se vlákna vystřídají v používání jádra CPU.
  - Kontextem se rozumí všechny nezbytné informace, které jsou nutné pro pozdější spuštění přerušeného vlákna od okamžiku přerušení (např. čítač instrukcí, obsahy registrů,...).
  - Samotná vlákna žijí v iluzi, že běží bez přerušení od začátku do konce.
  - Přepínání kontextu probíhá v několika krocích
    1. uloží se kontext původního vlákna,
    2. jádro OS naplánuje další vlákno,
    3. nastaví se kontext tohoto vlákna.

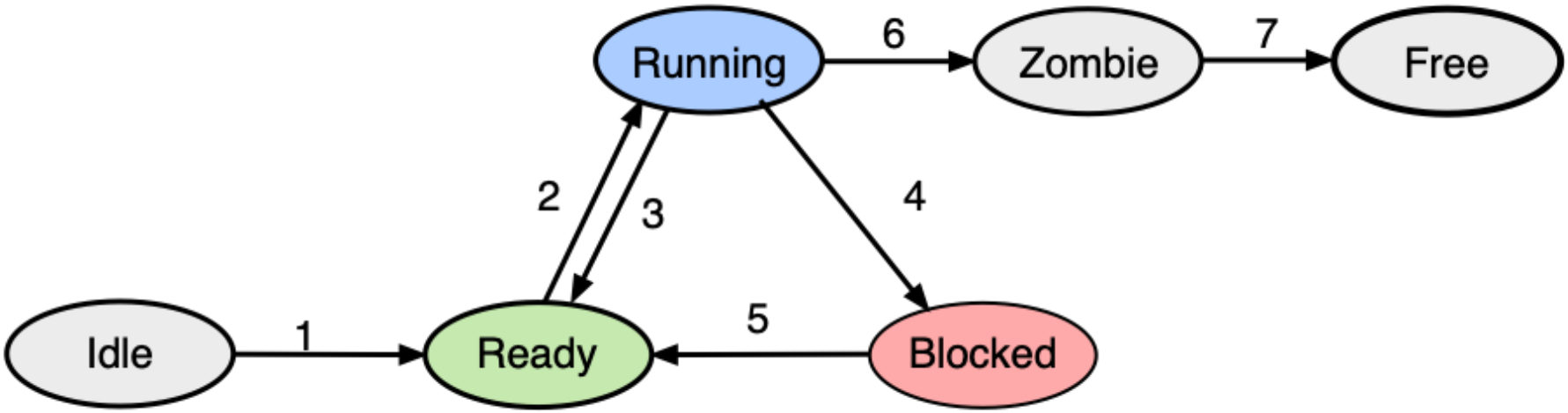
Přepínání kontextu





Stavy vláken

- Stav vlákna popisuje, co se s daným vláknem právě děje.
- Mezi základní stavy patří
  - Idle: vznik nového vlákna,
  - Ready: vlákno čeká až mu bude přiděleno jádro CPU,
  - Running: vlákno je zpracováváno jádrem CPU,
  - Blocked: vlákno čeká na událost (dokončení V/V operace, příchod signálu,...),
  - Zombie: vlákno je ukončováno, ale zatím ještě nebylo vše dokončeno,
  - Free: vlákno bylo kompletně zrušeno (pouze teoretický stav).

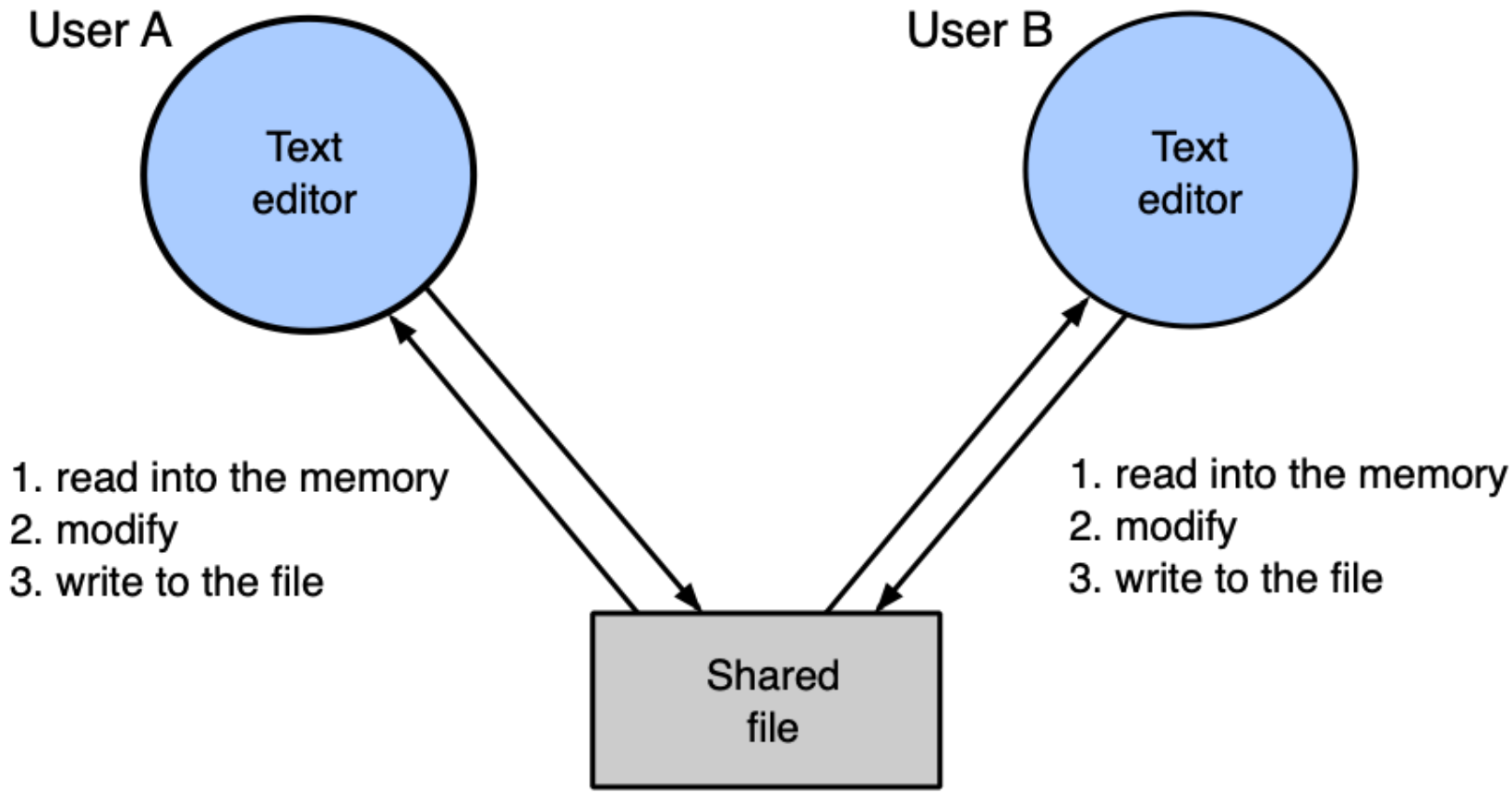


Časově závislé chyby (race conditions)

- Deterministický algoritmus
  - vždy ze stejných výchozích (vstupních) podmínek svým během vytvoří stejné výsledky.
- Časově závislé chyby
  - Situace, kdy dvě nebo několik vláken používá (čte/zapisuje) společné sdílené prostředky (např. sdílení proměnné, sdílené soubory,...) a výsledek deterministického algoritmu je závislý na rychlosti jednotlivých vláken, které používají tyto prostředky.
- Časově závislé chyby vykazují náhodný výskyt
  $\Rightarrow$ špatně se detekují!
- Předcházet časově závislým chybám bychom měli správným návrhem paralelního algoritmu. Při hledání chyb nám mohou pomoc některé ladící nástroje (např. Valgrind,...), ale neměli bychom na ně stoprocentně spoléhat.

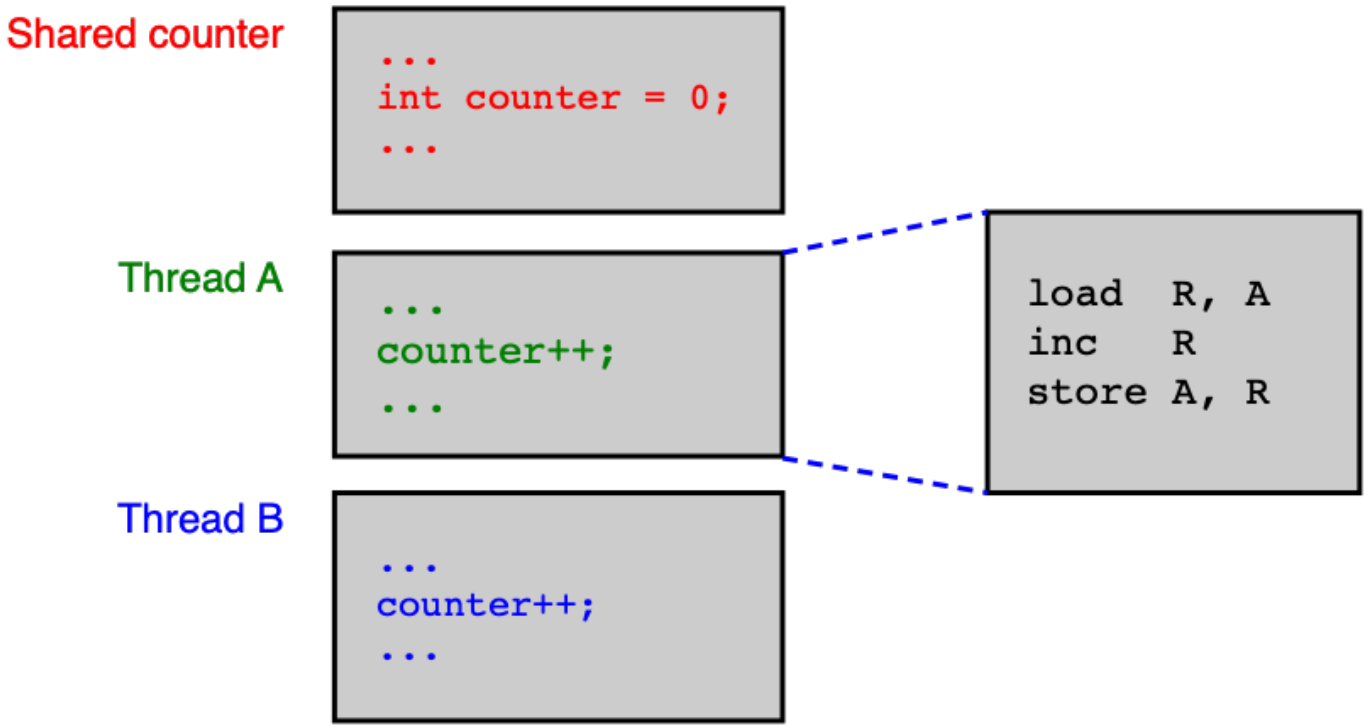
Příklad: Časově závislé chyby

- Dva uživatelé editují stejný soubor.



Příklad: Časově závislé chyby

- Dvě vlákna inkrementují sdílený čítač.



  - Hodnota čítače je uložená v paměti na adrese `A` a procesor obsahuje registr `R`.
  - Instrukce `load` načte hodnotu z paměti do registru.
  - Instrukce `inc` inkrementuje hodnotu registru.
  - Instrukce `strore` uloží hodnotu z registru do paměti.

Kritické sekce (critical regions)

- Kritická sekce
  - Část programu, kde vlákna používají sdílené prostředky (např. sdílená proměnná, sdílený soubor, . . .).
- Sdružené kritické sekce
  - Kritické sekce dvou (nebo více) vláken, které se týkají stejného sdíleného prostředku.
- Vzájemné vyloučení
  - Vláknům není dovoleno sdílet stejný prostředek ve stejném čase. $\Rightarrow$ Vlákna se nesmí nacházet ve stejné sdružené kritické sekci současně.

Korektní paralelní program

- Při psaní paralelních programů bychom měli dodržovat některá pravidla
  1. Žádné předpoklady nesmí být kladeny na rychlost vláken a počet jader, která sdílí (různá vlákna mohou běžet různě rychle v závislosti na plánování vláken a zátěži systému).
  2. Zajistit výlučný přístup ke sdíleným prostředkům
     - ★ Abychom toho dosáhli mohou být vlákna před kritickou sekcí pozastavena.
     - ★ Žádné vlákno ale nesmí čekat do nekonečna nebo "neúměrně dlouho" na vstup do kritické sekce.
  3. Mimo kritickou sekci by vlákno nemělo být blokováno (zpomalováno) ostatními vlákny.

Použité zdroje

1. A. S. Tanenbaum, H. Bos: *Modern Operating Systems (4th edition)*, Pearson, 2014.
2. W. Stallings: *Operating Systems: Internals and Design Principles (9th edition)*, Pearson, 2017.
3. A. Silberschatz, P. B. Galvin, G. Gagne: *Operating System Concepts (9th edition)*, Wiley, 2012.