Operační systémy

Implementace procesů a vláken. Plánování vláken.

Jan Trdlička

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

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

Obsah přednášky

Implementace procesů

- **Tabulka procesů**
  - Jádro OS si udržuje informace o procesech nejčastěji pomocí nějaké varianty zřetězeného seznamu struktur, který můžeme nazývat "tabulkou procesů".
- **Process control block (PCB)**
  - Jedna položka tabulky (struktura) reprezentuje všechny nezbytné informace, které si jádro OS musí pamatovat pro jeden proces, a historicky se nazývá process control block.
- **Kolik procesů/vláken může být maximálně vytvořeno?**
  - V rámci celého systému
    - Většina OS odvozují maximální počet procesů/vláken např. od velikosti fyzické paměti (např. Linux, Solaris,....) a může být definován např. pomocí parametrů jádra.
  - Jedním uživatelem
    - Z důvodu bezpečnosti je většinou maximální počet procesů/vláken pro běžné uživatele nastaven (např. ochrana před fork-bombou).
    - V unixových systémech např. pomocí příkazů `ulimit`, `sysctl`, `rctladm`,...

Příklad: Maximální počet procesů/vláken v Linuxu

- Kolik vláken lze maximálně vytvořit v systému?

```
u1@linux:~> cat /proc/sys/kernel/threads-max
15523
```

- Kolik vláken může běžný uživatel `u1` vytvořit?

```
u1@linux:~> ulimit -u
7761
```

- Kolik vláken má běžný uživatel právě vytvořeno?

```
u1@linux:~> ps -Lu u1 | tail -n+2 | wc -l
13
```

- Nastavíme nový limit.

```
u1@linux:~> ulimit -u 100
u1@linux:~> ulimit -u
100
```

  - Ověříme jeho funkčnost pomocí bash fork-bomby.

```
u1@linux:~> f(){ f | f & } ; f
-bash: fork: retry: Resource temporarily unavailable
...
```

  - Ověříme, že limit zafungoval, a ukončíme fork-bombu.

```
root@linux:~> ps -Lu u1 | tail -n+2 | wc -l
100
root@linux:~> pkill -u u1
```

Process control block

- Obsahuje všechny nezbytné informace, které si OS musí pamatovat o procesu.
- **Informace pro identifikaci procesu**
  - číslo procesu (PID), číslo rodičovského proces (PPID),
  - číslo seance (SID), číslo úlohy, číslo projektu,
  - jméno procesu,...
- **Informace související se identitou procesu/bezpečností**
  - vlastník procesu (EUID, RUID,...),
  - příslušnost ke skupinám (GID),
  - privilegia přiřazená procesu, access token,. . .
- **Informace o alokovaných prostředcích**
  - paměť
    - ★ informace o přidělené fyzické paměti,
    - ★ informace nutné pro překlad logických adres na fyzické,...
  - otevřené soubory (tabulka deskriptorů souborů)
  - prostředky pro mezi procesovou komunikaci (IPC)
    - ★ synchronizační nástroje (semafory, roury, signály,...),...

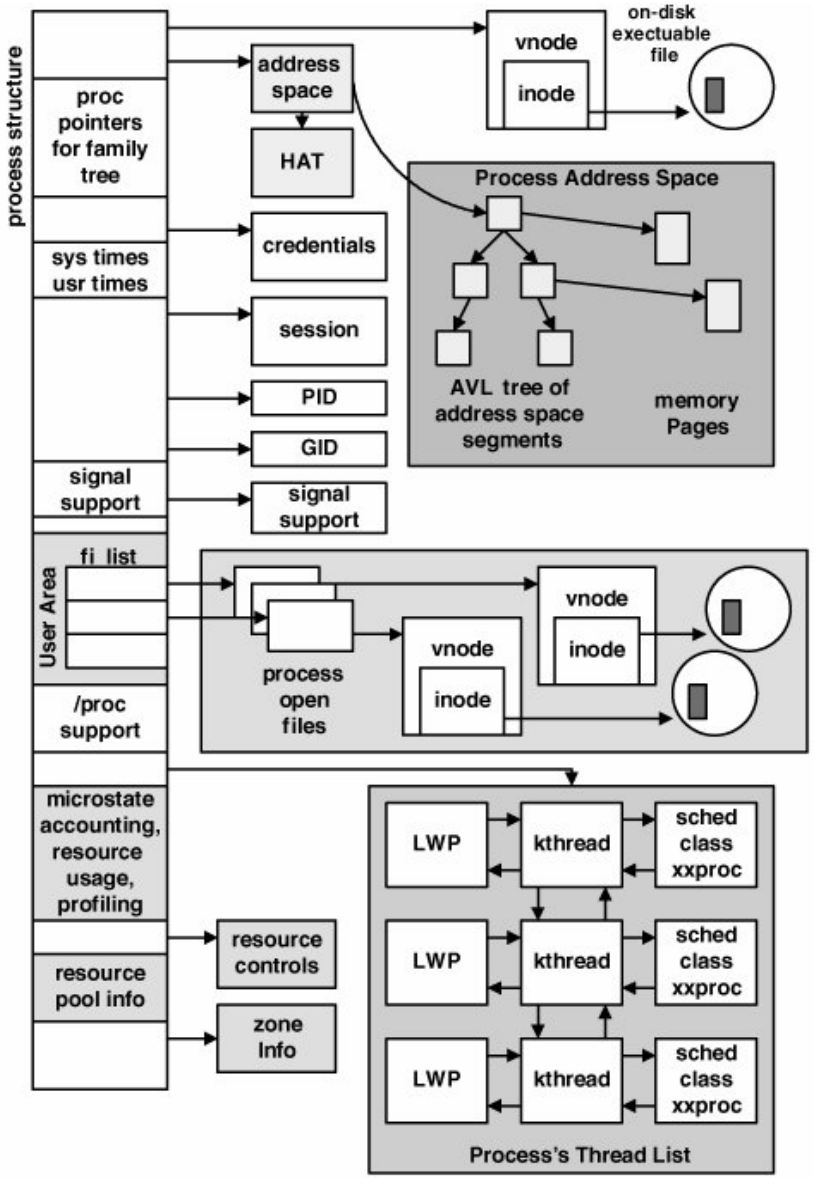
Thread control block

- Pro každé vlákno daného procesu si OS musí pamatovat řadu informací, které jsou uloženy v různých datových strukturách. Tyto struktury budeme označovat jako thread control block a obsahují následující informace.
  - **informace pro identifikaci vlákna**
    - číslo vlákna (TID).
  - **informace pro přepínání kontextu**
    - hodnoty viditelných registrů CPU,
    - hodnoty řídících a stavových registrů CPU (čítač instrukcí,. . .),
    - ukazatel na zásobník,. . .
  - **informace pro plánování vláken**
    - typ plánovacího algoritmu,
    - priorita,
    - stav vlákna,
    - informace o událostech, na které vlákno čeká,
    - využitý čas CPU,
    - důvod posledního přepnutí kontextu,...

Process control block

- **Jak zjistit struktru PCB na konkrétním OS?**
  - dokumentace k danému OS (MS Windows, Linux, Solaris),...
  - publikace typu OS internals
    - ★ Windows Internals Part 1, Part 2 (7th Edition),
    - ★ Linux Kernel Development (3rd Edition),
    - ★ MacOS and iOS Internals,
    - ★ Solaris Internals: Solaris 10 ... (2nd Edition),...
  - zdrojový kód, pokud je k dispozici (např. Linux kernel),
  - hlavičkové soubory
    - ★ MS Windows: `winternl.h`,...
    - ★ Linux: `sched.h`,...
    - ★ Solaris: `proc.h`, `thread.h`,...
- **Jak zjistit informace z PCB pro určitý process?**
  - applikace: `Task Manager`, `Process Explorer` (MS Windows),
  - příkazy: `ps`, `top`, `lsof`, `gstack`,...
  - OS API: `getpid`, `gettid`, `getuid`,...
  - debugger: `WinDbg` (MS Windows), `gdb` (Linux), `mdb` (Solaris),...
  - adresář **`/proc`**: v OS unixového typu,...

Příklad: Struktura procesu `proc_t` v Solarisu




```
user@solaris:~> cat /usr/include/sys/proc.h
/*
 * One structure allocated per active process.  It contains all
 * data needed about the process while the process may be swapped
 * out.  Other per-process data (user.h) is also inside the proc structure.
 * Lightweight-process data (lwp.h) and the kernel stack may be swapped out.
 */
typedef struct  proc {
        /*
         * p_exec is protected by p_lock
         */
        struct  vnode *p_exec;          /* pointer to a.out vnode */
        /*
         * Fields requiring no explicit locking
         */
        struct  as *p_as;               /* process address space pointer */
        struct  plock *p_lockp;         /* ptr to proc struct's mutex lock */
        kmutex_t p_crlock;              /* lock for p_cred */
        struct  cred    *p_cred;        /* process credentials */
         /*
         * Fields protected by pidlock
         */
        int     p_swapcnt;              /* number of swapped out lwps */
        char    p_stat;                 /* status of process */
        char    p_wcode;                /* current wait code */
        ushort_t p_pidflag;             /* flags protected only by pidlock */
        int     p_wdata;                /* current wait return value */
        pid_t   p_ppid;                 /* process id of parent */
        struct  proc    *p_link;        /* forward link */
        struct  proc    *p_parent;      /* ptr to parent process */
        struct  proc    *p_child;       /* ptr to first child process */
        . . .
        /*
         * Per process signal stuff.
         */
        k_sigset_t p_sig;               /* signals pending to this process */
        k_sigset_t p_extsig;            /* signals sent from another ... */
        k_sigset_t p_ignore;            /* ignore when generated */
        k_sigset_t p_siginfo;           /* gets signal info with signal */
        struct sigqueue *p_sigqueue;    /* queued siginfo structures */
        struct sigqhdr *p_sigqhdr;      /* hdr to sigqueue structure pool */
        struct sigqhdr *p_signhdr;      /* hdr to signotify structure pool */
        uchar_t p_stopsig;              /* jobcontrol stop signal */
        . . .
} proc_t;
```

Příklad: Debuggování jádra Solarisu

- Zjistění informace o procesech pomocí debagování jádra.

```
root@solaris:~> echo "::ps" | mdb -k
S    PID   PPID   PGID    SID    UID      FLAGS             ADDR NAME
R      0      0      0      0      0 0x00000001 ffffffffc063270 sched
R      1      0      0      0      0 0x4a004000 ffffa1c002c571e8 init
. . .
R   1436      1   1302   1302    100 0x4a004000 ffffa1c00d2c0238 gnome-terminal
R   1439   1436   1439   1439    100 0x4a014000 ffffa1c00c4dc2b0 bash
R  26200   1439  26200   1439      0 0x5a006000 ffffa1000def87c8 su
R  26201  26200  26201   1439      0 0x4a014000 ffffa1c0089ca088 bash
. . .
```

- Zobrazení obsahu PCB konkrétního procesu s PID=26201.

```
root@solaris:~>  echo "ffffa1c0089ca088 ::print -ta proc_t" | mdb -k
ffffa1c0089ca088 proc_t {
    ffffa1c0089ca088 struct vnode *p_exec = 0xffffa1c00c586200
    ffffa1c0089ca090 struct as *p_as = 0xffffa1000df1a8d8
    ffffa1c0089ca098 struct plock *p_lockp = 0xffffa1c001d59680
    ffffa1c0089ca0a0 kmutex_t p_crlock = {
        ffffa1c0089ca0a0 void *[1] _opaque = [ 0 ]
    }
    ffffa1c0089ca0a8 struct cred *p_cred = 0xffffa1c0049c5eb0
    ffffa1c0089ca0b0 int p_swapcnt = 0
    ffffa1c0089ca0b4 char p_stat = '\002'
    ffffa1c0089ca0b5 char p_wcode = '\0'
    . . .
    ffffa1c0089ca0c8 struct proc *p_parent = 0xffffa1000def87c8
    ffffa1c0089ca0d0 struct proc *p_child = 0xffffa1000df50a18
    . . .
```

Příklad: Obsah adresáře `/proc` v Linuxu

- Do adresář `/proc` je připojen pseudo systém souborů (jeho obsah existuje pouze v paměti).

```
u1@linux:~> mount | grep /proc
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
```

- Prostřednictvím tohoto systému souborů jádro OS umožňuje získat/nastavit hodnoty některých parametrů jádra.
  - Parametr `threads-max` definuje maximální počet vláken, která lze v systému vytvořit.

```
u1@linux:~> cat /proc/sys/kernel/threads-max
31256
```

- V podadresářích, které mají jméno stejné jako je PID procesu, jsou uložené informace o konkrétních procesech.
  - Jak zjistit limity nastavené pro naš aktuální shell?

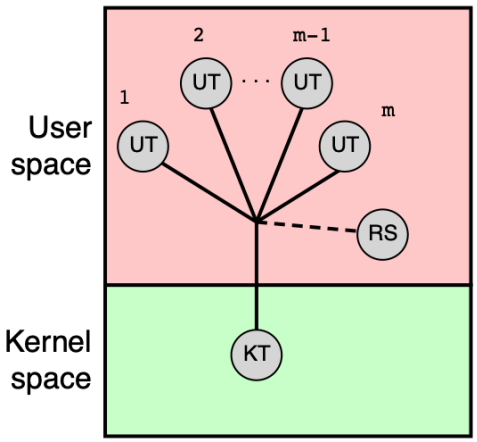
```
u1@linux:~> ps
   PID TTY          TIME CMD
 36282 pts/0    00:00:00 bash
u1@linux:~> cat /proc/36282/limits
Limit                     Soft Limit           Hard Limit           Units
. . .
Max stack size            8388608              unlimited            bytes
Max processes             100                  100                  processes
. . .
```

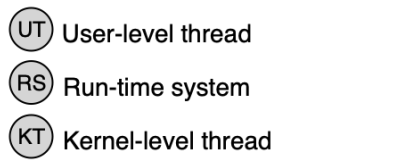
Implementace vláken

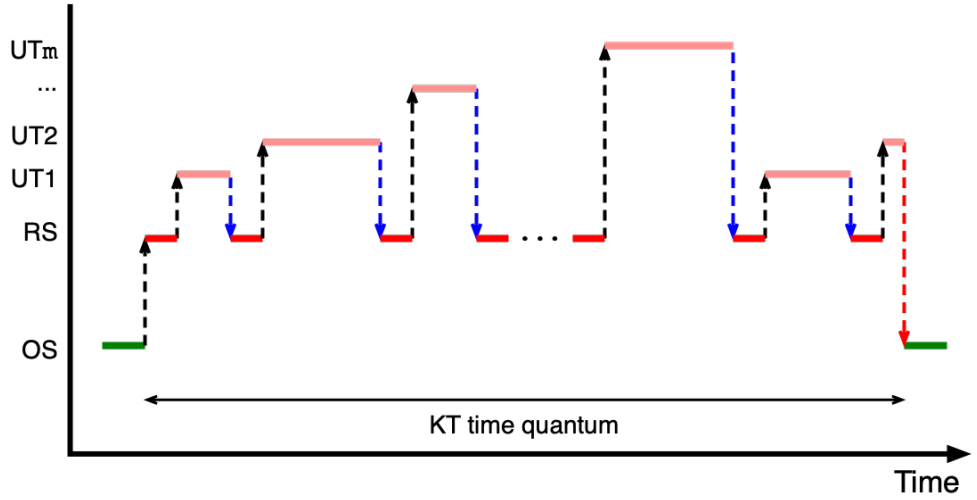
- **Historicky rozlišujeme dva způsoby implementace vláken**
  - v uživatelském prostoru (user-level threads),
  - v jádru OS (kernel-level threads).
- **Vlákna implementovaná v uživatelském prostoru**
  - Historicky používaná v OS, které podporovaly pouze proces s jedním "vláknem".
  - Vlákna jsou kompletně podporována/spravována v uživatelském prostoru pomocí "run-time" systému (user-level knihovna).
  - Uživatelská vlákna používají kooperativní plánování (po určitém čase vlákno předá řízení zpět run-time systému pomocí příslušné funkce).
  - Jádro OS nemá "žádnou" informaci o uživatelských vláknech v jednotlivých procesech a spravuje je jako proces s jedním vláknem.
  - **Vlastnosti**
    - ★ Jednoduchá/rychlá správa: vytvoření, přepínání, synchronizace vláken je v uživatelském prostoru bez zásahu jádra OS.
    - ★ Vlákna jednoho procesu jsou mapována na jedno jádro CPU.
    - ★ Blokující volání v jednom vláknu zablokuje i ostatní vlákna procesu.
  - V praxi je nutná aspoň minimální podpora jádra OS tak, aby blokující volání nezablokovala všechna vlákna v procesu.

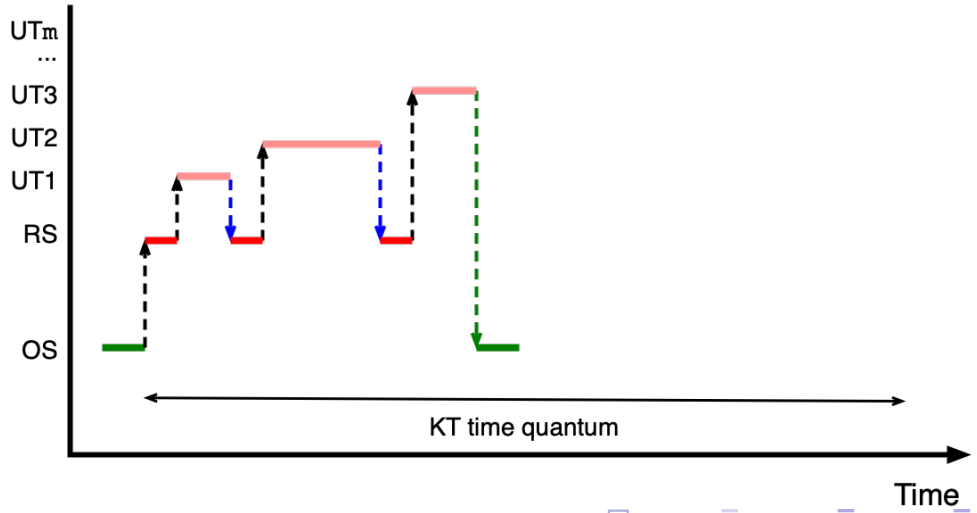
Implementace vláken

- **Vlákna implementovaná v uživatelském prostoru**
  - Tato implementace je někdy označovaná jako model many-to-one, ve kterém je více uživatelských vláken namapováno na jedno kernel vlákno.







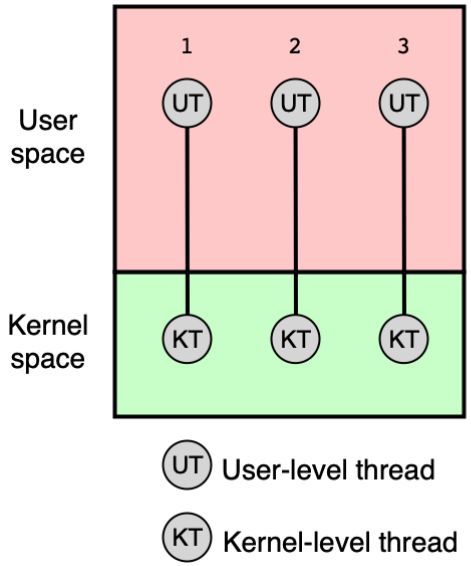


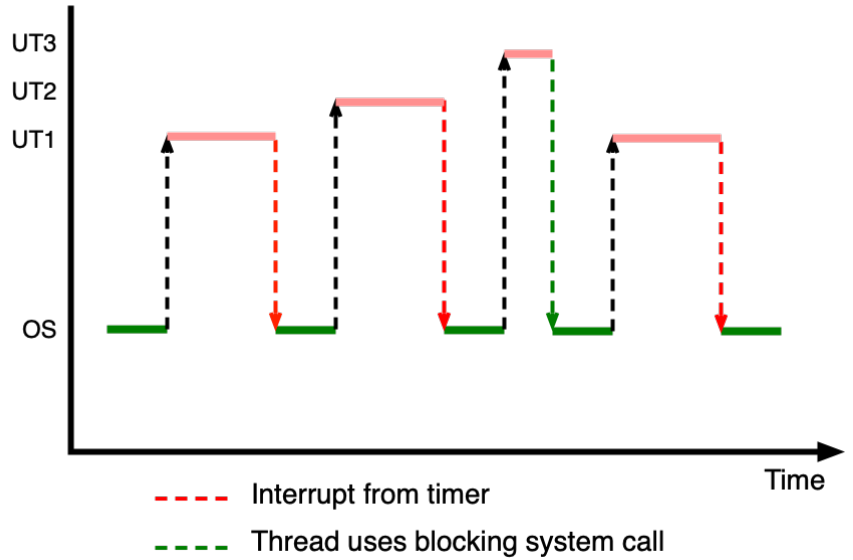
Implementace vláken

- **Vlákna implementovaná v jádře OS**
  - Typická implementace v současných OS s podporou vláken (MS Windows, Linux, Solaris, MacOS,...).
  - Vlákna jsou podporována/spravována přímo jádrem OS.
  - Jádro OS
    - ⋆ spravuje jeden PCB pro každý proces,
    - ⋆ spravuje jeden TCB pro každé vlákno,
    - ⋆ poskytuje systémová volání pro vytváření/správu vláken z uživatelského prostoru.
  - **Vlastnosti**
    - ⋆ Jádro má všechny informace o všech vláknech
      $\Rightarrow$ jádro OS přiděluje jednotlivá jádra CPU jednotlivým vláknům na určitou dobu.
    - ⋆ Pokud zavolá vlákno blokující systémové volání $\Rightarrow$ zablokuje se pouze toto vlákno.
    - ⋆ Vytvoření/ukončení vlákna, systémové volání, přepnutí kontextu,...
      $\Rightarrow$ představuje přepnutí z user modu do kernel modu.

Implementace vláken

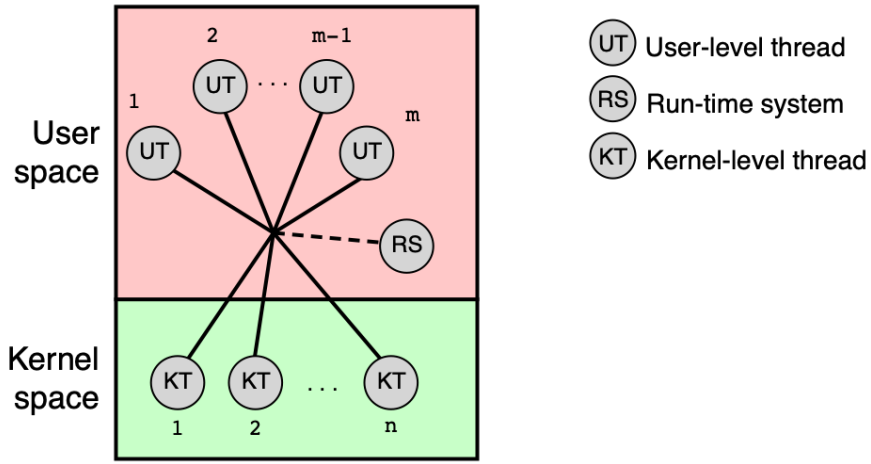
- **Vlákna implementovaná v jádře OS**
  - Tato implementace je někdy označovaná jako model one-to-one, ve kterém je jedno uživatelské vlákno namapováno na jedno kernel vlákno.





Implementace vláken

- **Hybridní implementace**
  - Tato implementace je někdy označovaná jako model many-to-many, ve kterém je *m* uživatelských vláken namapováno na *n* kernel vláken .
  - Jedná se o kombinaci předchozích implementací.



- **Nejednoznačná terminologie**
  - V současných OS jsou vlákna implementována v jádru OS.
  - Občas jsou v různých dokumentacích zmiňována "uživatelská vlákna", která jsou implementována pomocí knihovny. Většinou je tím ale myšlena knihovna, která implementuje model vláken "one-to-one" (v jádru OS).
  - Samotné jádro OS je implementováno jako vícevláknové a někdy je pod pojmem "kernel vlákno" myšleno vlákno, které implementuje příslušnou funkci jádra.

Příklad: Implementace procesů/vláken v Solarisu

- **Vlákno**
  - User-level thread (ULT): implementovaný pomocí knihovny vláken v adresovém prostoru procesu (neviditelný pro OS).
  - Kernel threads: základní jednotky, které jsou plánovány a spuštěny na jednotlivých jádrech CPU.
  - Lightweight process (LWP): mapování mezi ULT a kernel vlákny. Od Solarisu 8 (rok 2000) už je podporován pouze model one-to-one.
- **Proces**
  - Normální Unixový proces.
- **Úloha (task)**
  - Množina procesů jednoho uživatele, pro které lze nastavit limity (maximální počet vláken, procesů,...).
- **Projekt (project)**
  - Množina procesů jednoho/více uživatelů, pro které lze nastavit limity.
- **Zóna (zone)**
  - Virtuální instance OS (množina systémových a uživatelských procesů, které využívají pouze zdroje přidělené dané zóně).
- Limity, které lze nastavit viz. příkaz `rctladm`.

Příklad: Implementace procesů/vláken v MS Windows

- **Vlákno**
  - Fiber: vlákno spravované celé v uživatelském prostoru.
  - Thread: jednotka plánována jádrem.
- **Proces**
  - Jednotka, která alokuje zdroje. Každý proces má aspoň jedno vlákno (thread).
- **Job**
  - Množina procesů, pro které lze nastavit limity v rámci OS (maximální počet procesů, čas CPU, paměť, . . .).

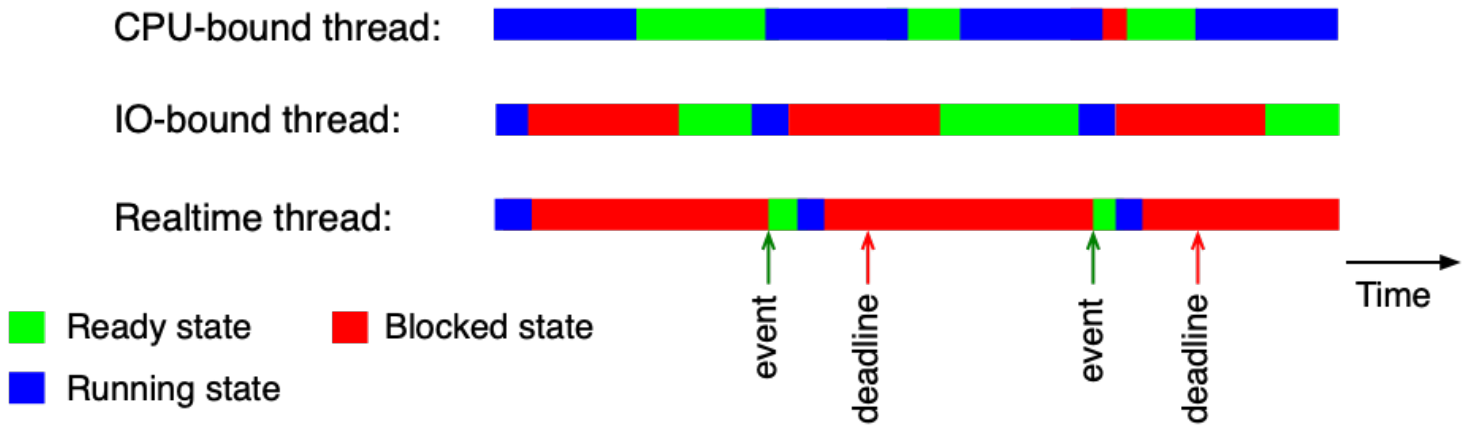
Plánování vláken

- V okamžiku, kdy se uvolní jádro CPU, musí OS vybrat "vhodné" vlákno ve stavu "Ready", a umožní mu používat toto jádro po určitou dobu.
- V OS je obvykle implementováno několik různých plánovacích strategií a uživatel/administrátor může pro různé aplikace nastavit vhodnou strategii, popřípadě určí, na kterých jádrech CPU poběží.
- **Typy aplikací z hlediska plánování vláken**
  - Dávkové úlohy (aplikace běžící na pozadí)
    - obvykle zpracovávají data, která načítají ze/zapisují do souborů, a nekomunikují přímo s uživatelem,
    - vyžadují relativně hodně CPU výkonu.
  - Interaktivní aplikace
    - reagují na události, které přicházejí např. z GUI, CLI, síťového rozhraní,...
  - Úlohy reálného času
    - většinou nevyžadují mnoho CPU výkonu, ale je nutné zajistit/garantovat co nejrychlejší reakci na událost (např. aplikace řídící technologický proces).

Cíle plánování

- **Z hlediska systému**
  - Spravedlivost: každý uživatel/proces/vlákno získá úměrnou část CPU výkonu.
  - Vyváženost: rozložit zátěž rovnoměrně na všechny části systému.
  - Prosazení strategie: možnost uplatnit zvolenou plánovací strategii.
- **Dávkové úlohy**
  - Doba zpracování (turnaround time): čas, který uplyne od spuštění do ukončení úlohy.
- **Interaktivní aplikace**
  - Doba odezvy (response time): čas, který uplyne od okamžiku zadání požadavku (události) do okamžiku první reakce.
  - Přiměřenost (proportionality): splnit očekávání uživatele.
- **Úlohy reálného času**
  - Garantování odezvy (meeting deadlines): zabránění ztrátě dat.
  - Předvídatelnost: zamezení degradaci výkonu.

Typy vláken



- Z hlediska plánování OS většinou rozlišuje tři typy vláken.
  1. **Vlákna orientovaná na CPU (CPU-bound threads)**
     - ★ CPU využíváno po dlouhou dobu, málo blokujících operací.
  2. **Vlákna orientovaná na V/V (I/O-bound threads)**
     - ★ CPU využito po krátkou dobu, hodně blokujících operací.
  3. **Vlákna reálného času (Realtime threads)**
     - ★ Vlákno musí zareagovat na událost během daného intervalu.
- První dva typy vyplývají z předchozího chování vlákna a během existence vlákna se může typ měnit.
- Třetí typ vyplývá ze způsobu použití vlákna, kdy je nutné zajistit rychlou reakci na příslušnou událost.

Kdy plánujeme?

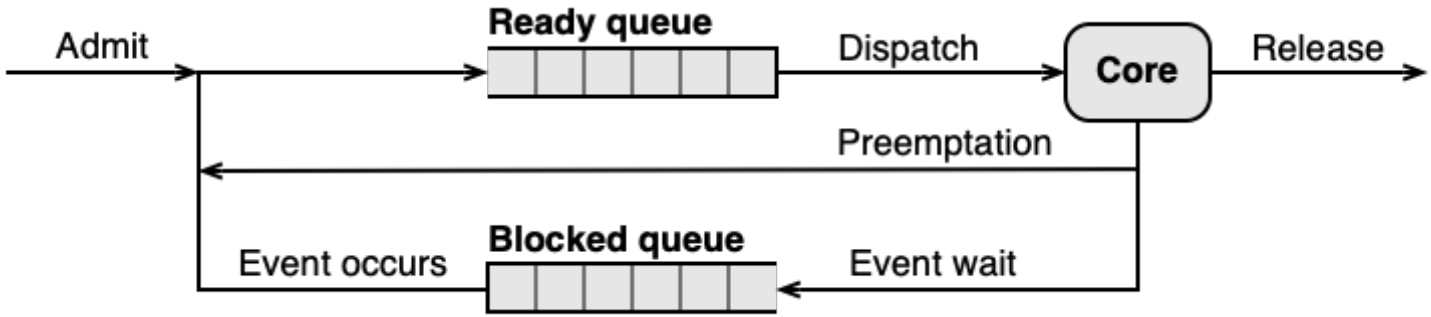
- **V okamžiku, kdy běžící vlákno "odevzdá" jádro CPU.**
  - Vlákno končí svůj výpočet
    - ★ Vlákno dokončilo svojí "práci" a zavolalo příslušné systémové volání (např. `exit()`, ...)
  - Vlákno dobrovolně odevzdává jádro CPU
    - ★ Vlákno se dobrovolně vzdalo jádra CPU pomocí příslušného systémového volání (např. `pthread_yield()`, ...)
  - Vlákno volá blokující systémové volání/knihovní funkci
    - ★ např. `read()`, `write()`, `pthread_mutex()`,...)
- **V okamžiku, kdy nastane nějaká událost**
  - Přerušení od
    - ★ V/V zařízení: dokončila se V/V operace a je nutné na to zareagovat,
    - ★ časovače: uplynulo časové kvantum přidělené danému vláknu,...

Strategie plánování

- **Plánování s odnímáním (preemptive scheduling)**
  - Jádro OS přidělí vláknu jádro CPU pouze na určitou dobu (časové kvantum) a po uplynutí této doby mu ho odebere.
  - Vlastnosti
    - ⋆ Strategie vhodná pro vlákna v interaktivních systémech.
    - ⋆ Umožňuje "rozumným" způsobem sdílet CPU výkon systému.
    - ⋆ Dochází zde k častějšímu přepínání kontextu
      $\Rightarrow$ horší využití CPU/lepší doba odezvy.
- **Kooperativní plánování (cooperative scheduling)**
  - Jádro OS přidělí vláknu jádro CPU a vlákno ho používá, dokud ho samo neuvolní (např. dokončení výpočtu, blokující funkce,...).
  - Vlastnosti
    - ⋆ Vlákno může blokovat systém a musí "spolupracovat" pouze když žádá službu jádra $\Rightarrow$ tato strategie je vhodná pouze pro "prověřená/kernel" vlákna a není vhodná pro běžná uživatelská vlákna.
    - ⋆ Minimalizuje se počet přepnutí kontextu
      $\Rightarrow$ lepší využití CPU/horší doba odezvy.

Plánování s odnímáním

- **Round-robin plánování (RR)**



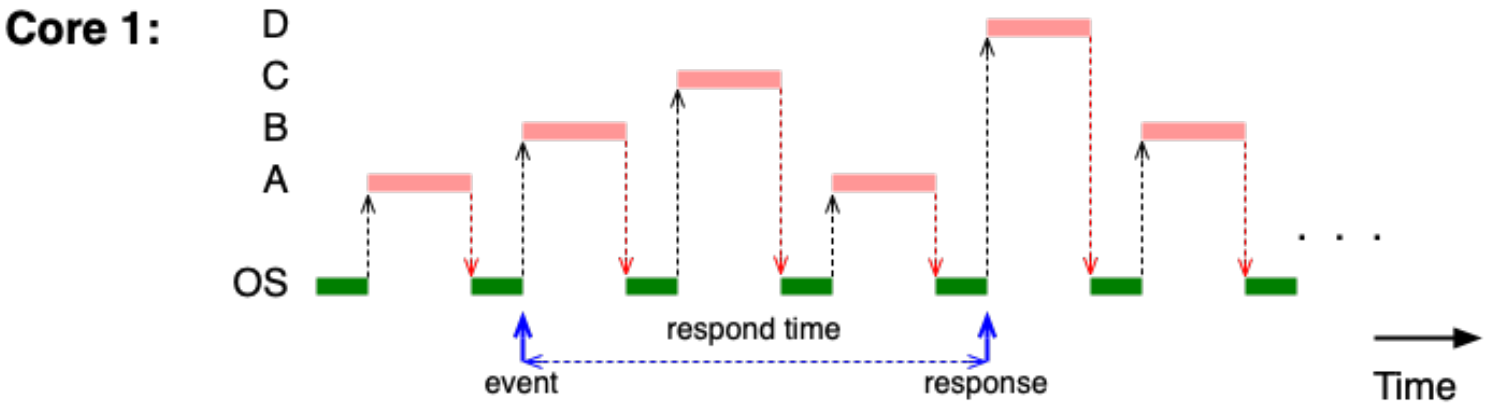
  - Plánování s odnímáním, ve kterém vlákna ve stavu "Ready" čekají ve frontě FIFO, až jim bude přiděleno jádro CPU.
  - Všem vláknům je jádro CPU propůjčováno na stejně velkou dobu (časové kvantum). Po uplynutí této doby je jádro CPU vláknům odebráno, změní se jejich stav na "Ready" a vlákna se zařadí na konec fronty (Ready queue).
  - Volitelným parametrem plánování je velikost časového kvanta:
    - ★ Krátké čas. kvantum ⇒ horší využití CPU/krátká doba odezvy.
    - ★ Dlouhé čas. kvantum ⇒ dlouhá doba odezvy/lepší využití CPU.
    - ★ Rozumný kompromis je 10-50ms.

Příklad: Plánování s odnímáním RR

- **Předpokládejme následující parametry systému.**
  - Jádro CPU sdílejí pouze čtyři vlákna A,...,D pomocí strategie RR .
    - ★ Vlákna A, B a C jsou pouze ve stavech Ready a Running.
    - ★ Vlákno D je zablokováno (např. na mutexu).
    - ★ Těsně po přepnutí kontextu je vlákno D odblokováno.
  - Doba přepnutí kontextu je t_{cs} a časové kvantun je t_q.
- **Jaká je efektivita využití CPU a doba odezvy vlákna D na probuzení?**
  1. Pokud $t_{cs} = t_q = 10$ms.
  2. Pokud $t_{cs} = 10$ms a $t_q = 90$ms.

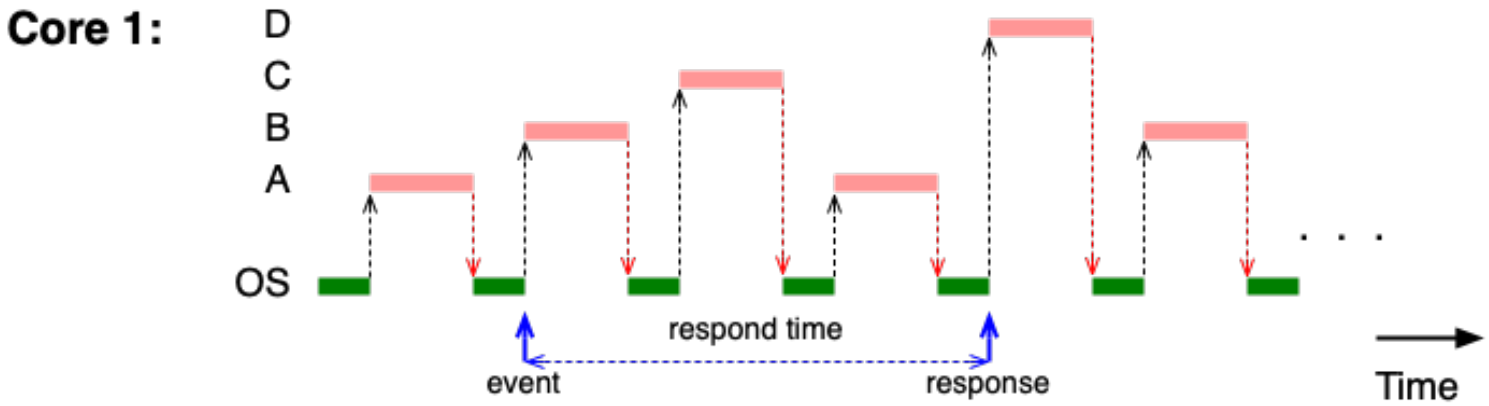
Příklad: Plánování s odnímáním RR

- **Předpokládejme následující parametry systému.**
  - Jádro CPU sdílejí pouze čtyři vlákna A,...,D pomocí strategie RR .
    - ★ Vlákna A, B a C jsou pouze ve stavech Ready a Running.
    - ★ Vlákno D je zablokováno (např. na mutexu).
    - ★ Těsně po přepnutí kontextu je vlákno D odblokováno.
  - Doba přepnutí kontextu je t_{cs} a časové kvantun je t_q.
- **Jaká je efektivita využití CPU a doba odezvy vlákna D na probuzení?**
  1. Pokud $t_{cs} = t_q = 10$ms.
  2. Pokud $t_{cs} = 10$ms a $t_q = 90$ms.



Příklad: Plánování s odnímáním RR

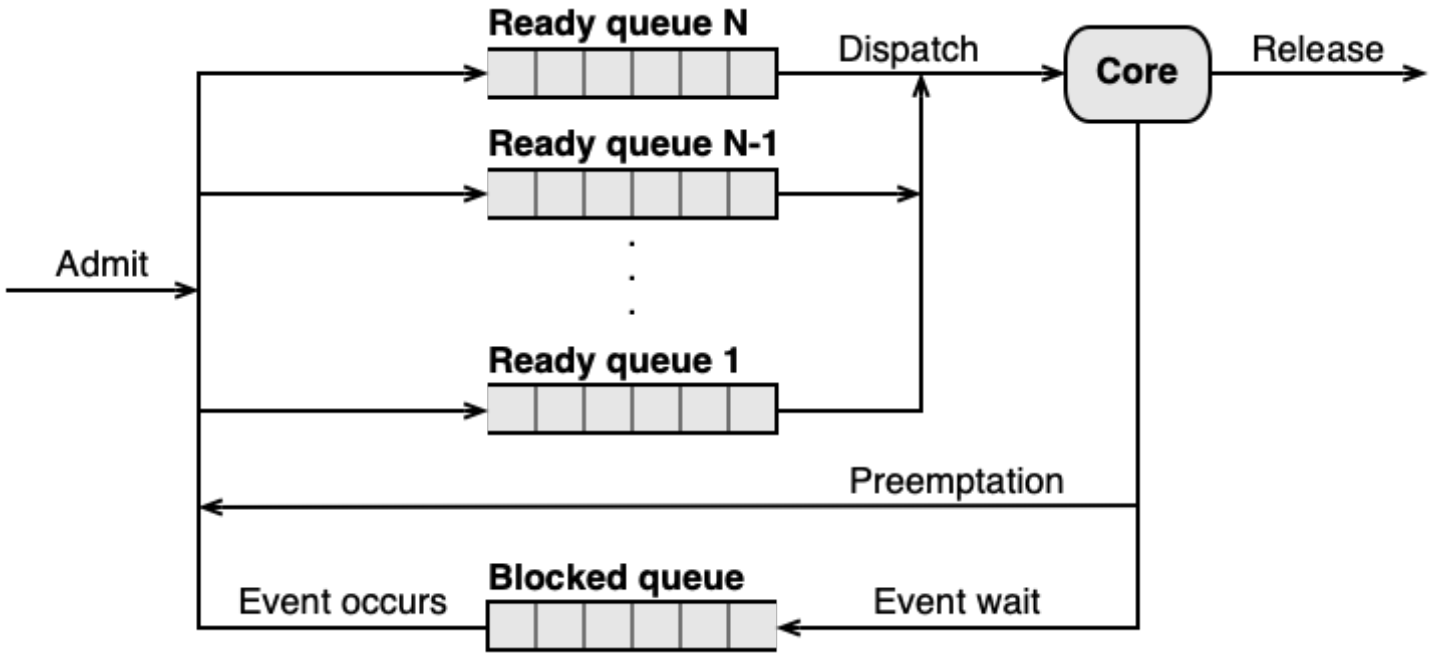
- **Předpokládejme následující parametry systému.**
  - Jádro CPU sdílejí pouze čtyři vlákna A,...,D pomocí strategie RR .
    - ★ Vlákna A, B a C jsou pouze ve stavech Ready a Running.
    - ★ Vlákno D je zablokováno (např. na mutexu).
    - ★ Těsně po přepnutí kontextu je vlákno D odblokováno.
  - Doba přepnutí kontextu je t_{cs} a časové kvantun je t_q.
- **Jaká je efektivita využití CPU a doba odezvy vlákna D na probuzení?**
  1. Pokud $t_{cs} = t_q = 10$ms.
  2. Pokud $t_{cs} = 10$ms a $t_q = 90$ms.



1. Efektivita: $t_q/(t_{cs} + t_q) \times 100 = 50\%$, odezva: $3 \times (t_{cs} + t_q) = 60$ms.
2. Efektivita: $t_q/(t_{cs} + t_q) \times 100 = 90\%$, odezva: $3 \times (t_{cs} + t_q) = 300$ms.

Plánování s odnímáním

- **Prioritní RR plánování se statickou prioritou**



  - Plánování s odnímáním, ve kterém je každému vláknu přiřazena statická priorita (číslo $1, \ldots, N$), která se během existence vlákna nemění (obvykle vyšší číslo reprezentuje vyšší prioritu).
  - Volné jádro CPU je vždy přiřazeno vláknu s nejvyšší prioritou, které je na začátku příslušné fronty (Ready queue).
  - "Ready" fronty jsou implementovány jako RR $\Rightarrow$ FIFO.
  - Časové kvantum může být pro různé priority různě velké.

Příklad: Prioritní RR plánování se statickou prioritou

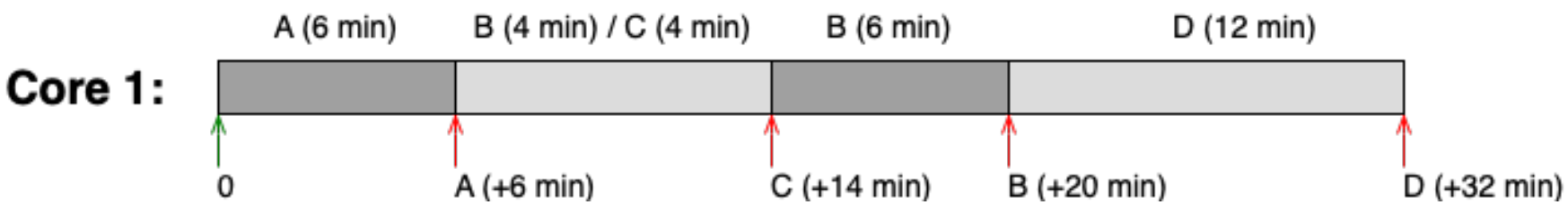
- **Předpokládejme následující parametry systému.**
  - Vlákna mají přidělenu statickou prioritu $1, \ldots, 99$.
  - Pro všechny priority je velikost časového kvanta 100ms.
  - Režie na přepínání kontextu je zanedbatelná vzhledem k velikosti časového kvanta.
  - Na jednom jádře CPU běží pouze vlákna A, B, C a D, která jsou spuštěna ve stejném okamžiku a jsou orientovaná na CPU.

Vlákno	Priorita	Požadovaný čas na jednom jádře [min]
A	99	6
B	75	10
C	75	4
D	50	12

1. **Za jak dlouho od spuštění jednotlivá vlákna skončí, pokud pouze ona poběží na jednom jádře CPU?**
2. **Za jak dlouho od spuštění jednotlivá vlákna skončí, pokud pouze ona poběží na dvou jádrech CPU?**

Příklad: Prioritní RR plánování se statickou prioritou

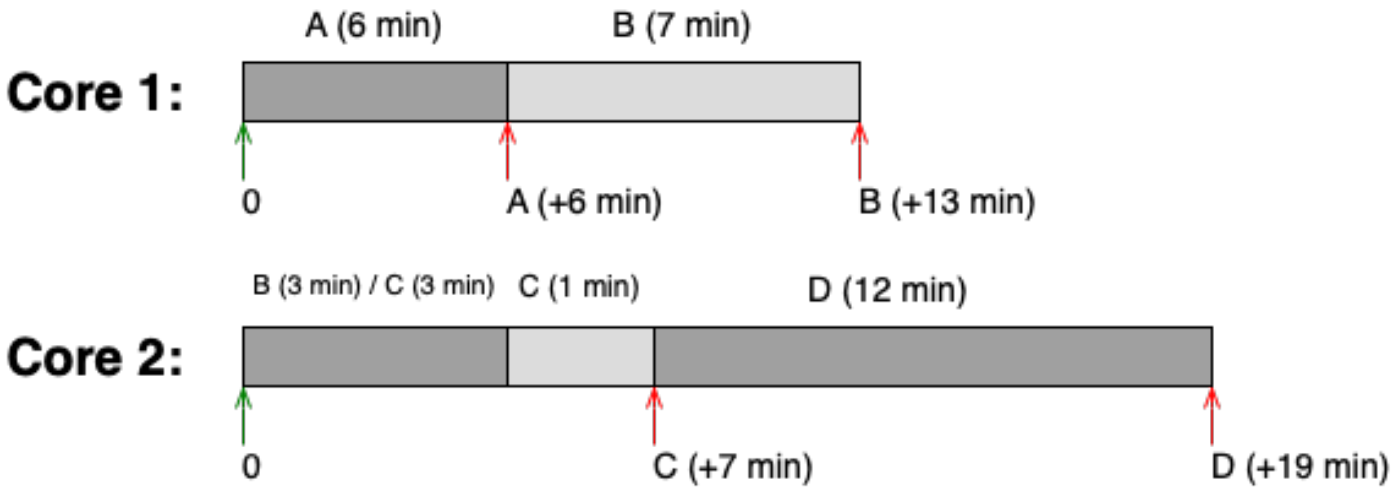
1. **Za jak dlouho od spuštění jednotlivá vlákna skončí, pokud pouze ona poběží na jednom jádře CPU?**



- **Statická priorita**
  - Vláknům s nízkou prioritou může být přiděleno jádro CPU až za "dlouhou dobu" $\Rightarrow$ problémy s hladověním a pomalou odezvou.

Příklad: Prioritní RR plánování se statickou prioritou

2. **Za jak dlouho od spuštění jednotlivá vlákna skončí, pokud pouze ona poběží na dvou jádrech CPU?**

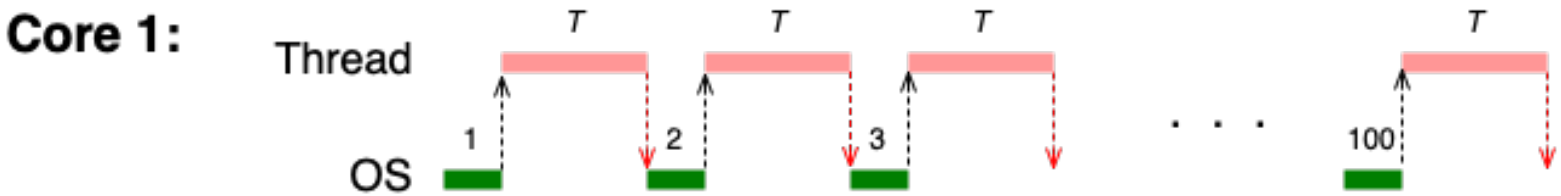


Příklad: Prioritní RR plánování se statickou prioritou

- **Předpokládejme následující parametry systému.**
  - Pro všechny priority je velikost časového kvanta T ms.
  - Na jednom jádru CPU běží pouze jedno vlákno orientované na CPU, které vyžaduje $100xT$ času jednoho jádra pro dokončení výpočtu.
- **Ke kolika přepnutím kontextu dojde než se vlákno ukončí?**

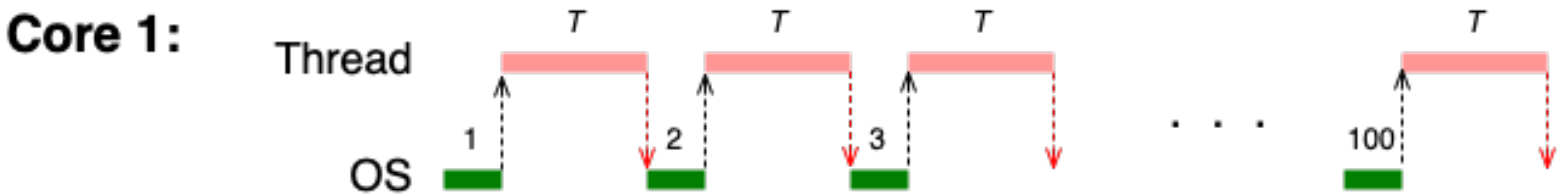
Příklad: Prioritní RR plánování se statickou prioritou

- **Předpokládejme následující parametry systému.**
  - Pro všechny priority je velikost časového kvanta T ms.
  - Na jednom jádru CPU běží pouze jedno vlákno orientované na CPU, které vyžaduje $100xT$ času jednoho jádra pro dokončení výpočtu.
- **Ke kolika přepnutím kontextu dojde než se vlákno ukončí?**



Příklad: Prioritní RR plánování se statickou prioritou

- **Předpokládejme následující parametry systému.**
  - Pro všechny priority je velikost časového kvanta T ms.
  - Na jednom jádru CPU běží pouze jedno vlákno orientované na CPU, které vyžaduje $100xT$ času jednoho jádra pro dokončení výpočtu.
- **Ke kolika přepnutím kontextu dojde než se vlákno ukončí?**



- Kontext se přepne v důsledku zpracování vlákna 100x.
- **Fixní časové kvantum**
  - Při plánování s fixním časovým kvantem se nezohledňuje chování vláken $\Rightarrow$ vlákna orientovaná na CPU mohou vygenerovat velký počet přepnutí kontextu během své existence.

Prioritní RR s dynamickou prioritou

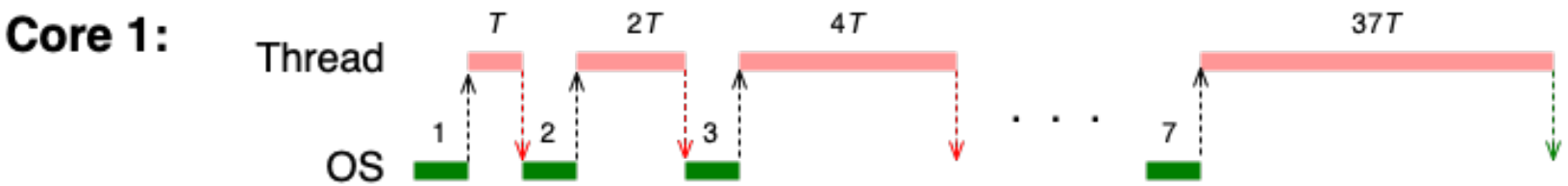
- Plánování s odnímáním, ve kterém se priorita i velikost časového kvanta během existence vlákna může měnit, tak aby bylo dosaženou určitého cíle.
- Nejčastějším cílem je minimalizovat problém hladovění vláken, zlepšit odezvu a snížit počet přepnutí kontextu.
- Strategie pro dosažení tohoto cíle může být následující
  - Priorita se zvýší a časové kvantum sníží
    - pokud vlákno v posledním běhu nevyužilo celé své časové kvantum (vlákno orientované na V/V),
    - pokud vlákno dlouho čeká na CPU (hrozí problém hladovění vlákna).
  - Priorita se sníží a časové kvantum se zvýší
    - pokud vlákno v posledním běhu využilo celé své časové kvantum (vlákno orientované na CPU).
- Tato strategie je výchozí plánovací strategií v běžných OS.
- Její implementace je závislá na konkrétním OS.

Příklad 3: Prioritní RR plánování

- **Předpokládejme následující parametry systému.**
  - Používá se dynamická priorita s proměnným časovým kvantem.
    - ★ Při spuštění mají vlákna nastavenou prioritu P_0 a časové kvantum T.
    - ★ Pokud vlákno s aktuální prioritou P_i využije celé časové kvantum $\Rightarrow$ v následujícím běhu poběží s prioritou $P_i - 1$ a dvojnásobným časovým kvantem.
    - ★ Pokud vlákno s aktuální prioritou P_i nevyužije celé časové kvantum $\Rightarrow$ v následujícím běhu poběží s prioritou $P_i + 1$ a poloviční časovým kvantem.
  - Na jednom jádru CPU běží pouze jedno vlákno orientované na CPU, které vyžaduje $100xT$ času jednoho jádra pro dokončení výpočtu.
- **Ke kolika přepnutím kontextu dojde než se vlákno ukončí?**

Příklad 3: Prioritní RR plánování

- **Předpokládejme následující parametry systému.**
  - Používá se dynamická priorita s proměnným časovým kvantem.
    - ★ Při spuštění mají vlákna nastavenou prioritu P_0 a časové kvantum T.
    - ★ Pokud vlákno s aktuální prioritou P_i využije celé časové kvantum $\Rightarrow$ v následujícím běhu poběží s prioritou $P_i - 1$ a dvojnásobným časovým kvantem.
    - ★ Pokud vlákno s aktuální prioritou P_i nevyužije celé časové kvantum $\Rightarrow$ v následujícím běhu poběží s prioritou $P_i + 1$ a poloviční časovým kvantem.
  - Na jednom jádru CPU běží pouze jedno vlákno orientované na CPU, které vyžaduje $100xT$ času jednoho jádra pro dokončení výpočtu.
- **Ke kolika přepnutím kontextu dojde než se vlákno ukončí?**



- Kontext se přepne 7x, protože 100=1+2+4+8+16+32+37.

Příklad: "Time Sharing" (TS) třída v Solarisu

- Třída TS představuje plánování s odnímáním s dynamickou prioritou a proměnným časovým kvantem.
- Celý algoritmus je definovaný tabulkou, kterou lze zobrazit/nastavit pomocí příkazu `dispadmin`.

```
user@solaris:~> dispadmin -c TS -g
. . .
# ts_quantum  ts_tqexp  ts_slpret  ts_maxwait ts_lwait  PRIORITY LEVEL
       200          0         50           0         50        #     0
       200          0         50           0         50        #     1
       200          0         50           0         50        #     2
                                    . . .
       160          9         51           0         51        #    19
       120         10         52           0         52        #    20
       120         11         52           0         52        #    21
                                    . . .
       120         19         52           0         52        #    29
        80         20         53           0         53        #    30
        80         21         53           0         53        #    31
                                    . . .
        40         42         58           0         59        #    52
        40         43         58           0         59        #    53
        40         44         58           0         59        #    54
                                    . . .
        40         47         58           0         59        #    57
        40         48         58           0         59        #    58
        20         49         59       32000         59        #    59
```

Next priority for CPU-bound thread

Current thread priority

Next priority for starving thread

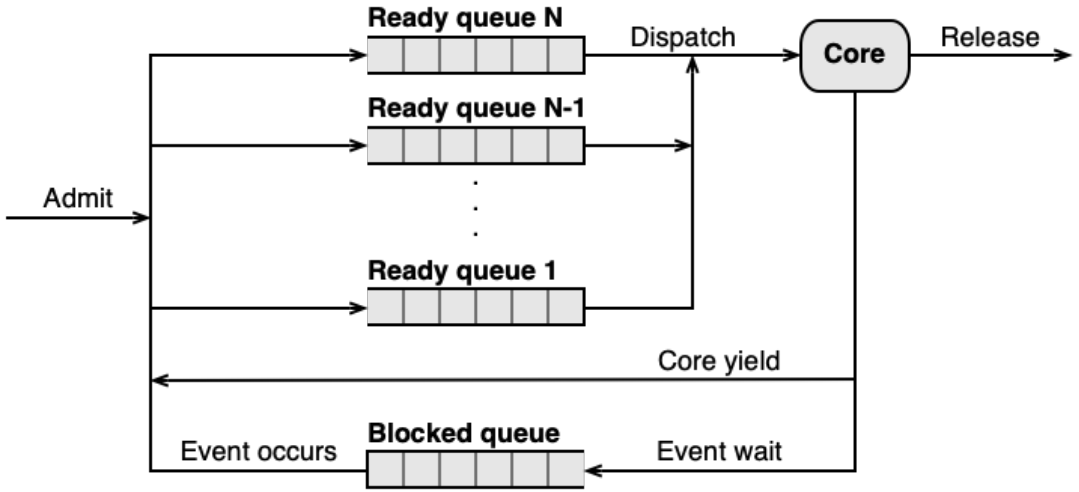
Next priority for IO-bound thread

Příklad: "Time Sharing" (TS) třída v Solarisu

- **Význam sloupečků ve výstupu příkazu `dispadmin`.**
  - Jedna řádka představuje parametry pro konkrétní hodnotu priority.
    - ★ PRIORITY LEVEL: hodnota priority.
    - ★ ts_quantum: velikost časového kvanta pro danou prioritu.
    - ★ ts_tqexp: nová hodnota priority pro vlákno, které využilo celé časové kvantum.
    - ★ ts_slpret: nová hodnota priority pro vlákno, které nevyužilo celé časové kvantum.
    - ★ ts_maxwait: počet sekund, které musí vlákno čekat v Ready frontě, než se mu nastaví nová priorita ts_lwait (hodnota 0 znamená, že po jedné sekundě se mu zvýší priorita).

Kooperativní plánování

- **First-come-first-served (FCFS)**



  - Kooperativní plánování, ve kterém vlákna ve stavu "Ready" čekají v jedné/několika FIFO frontách na přidělení jádra CPU.
  - Když běžící vlákno uvolní jádro CPU, první vlákno z Ready fronty s nejvyšším číslem bude pokračovat.
  - Vlákna ve stavu "Blocked" čekají ve frontě Blocked.
  - Vlastnosti
    - ★ Jednoduché na pochopení i implementování.
    - ★ Minimalizuje počet přepnutí kontextu.
    - ★ Zpomaluje vlákna orientovaná na V/V.

Příklad: "System" (SYS) třída v Solarisu

- Třída SYS představuje kooperativní plánování se statickou prioritou.
- Tato třída je vhodná pouze pro systémová vlákna, která krátce reagují na nějakou událost a je vhodné, aby reakci celou dokončila.
- Pouze kernel přiřazuje vlákna do této plánovací třídy.

```
user@solaris:~> ps -eLo class,pri,pid,lwp,comm | grep " *SYS "
 SYS  96     0      1 sched
 SYS  98     2      1 pageout
 SYS  97     2      2 pageout
 SYS  60     3      1 fsflush
 SYS  60     7      1 intrd
 SYS  60     8      1 vmtasks
 SYS  60   581      1 lockd_kproc
 SYS  60   581      2 lockd_kproc
 SYS  60 29436      1 nfs4cbd_kproc
 SYS  60 29436      2 nfs4cbd_kproc
15523
```

- V předchozím výpisu si můžeme např. všimnout procesu/vlákna `fsflush`, jehož úkolem je pravidelně ukládat změny ze skryté paměti (filesystem cache) do systému souborů (na disk).

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.
4. R. McDougall, J. Mauro: *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture (2nd edition)*, Prentice Hall, 2006.