

Operační systémy

Přednáška 2: Procesy a vlákna

Proces

- Všechny běžící software v systému je organizován jako množina běžících procesů (kromě jádra).
- **Program** – posloupnost instrukcí definujících chování procesu

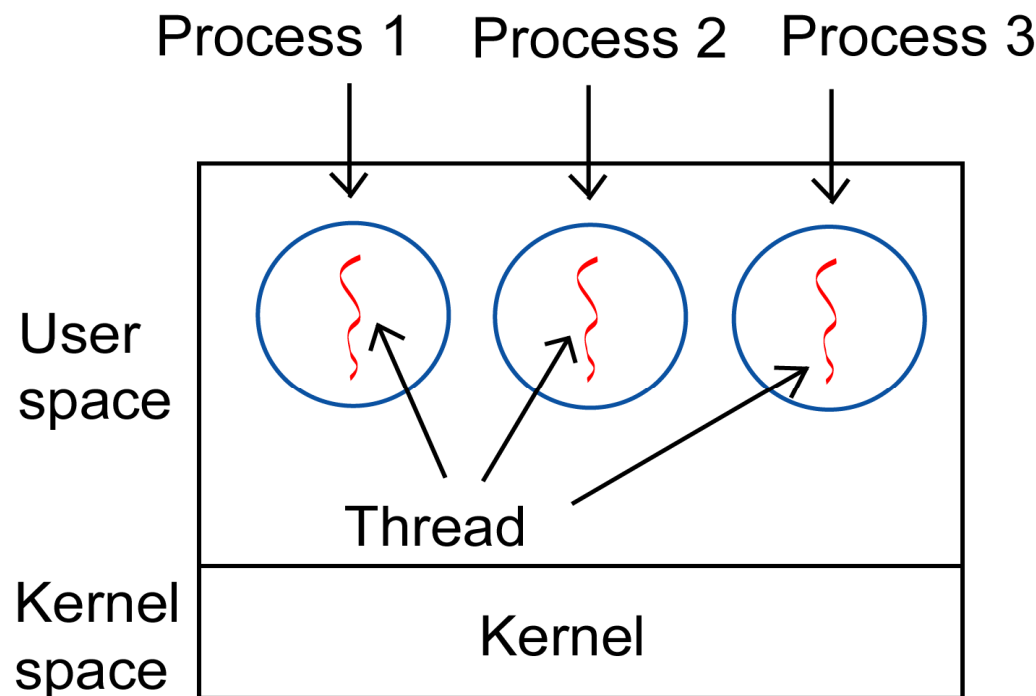
```
#include <stdio.h>
#include <stdlib.h>
...

int main (int argc, char *argv[])
{
    ...
};
```

- **Proces** – instance spuštěného programu

Procesový model

- Každý proces **alokuje příslušné prostředky** (adresový prostor obsahující kód a data, zásobník) má **atributy** (ID, identita, informace o rodiči a potomcích, informace pro plánování,...)
- Implicitně obsahuje **jedno vlákno výpočtu**.



Vytvoření procesu

- Nový proces se vytvoří když existující proces zavolá příslušné **systémové volání** (např. `fork()` a `exec()` v Unixu, nebo `CreateProcess()` v MS Windows).
- **Vytvoření**
 - OS inicializuje v jádře datové struktury spojené s novým procesem.
 - OS nahraje kód a data programu z disku do paměti a vytvoří prázdný systémový zásobník pro daný proces.
- **Klonování**
 - OS zastaví aktuální proces a uloží jeho stav.
 - OS inicializuje v jádře datové struktury spojené s novým procesem.
 - OS udělá kopii aktuálního kódu, dat, zásobníku, stavu procesu,...

Příklad: vytvoření procesu v Unixu

```
int main ()
{
    ...
    pid = fork();
    switch (pid) {

        case -1: /* doslo k chybe */
            perror ("chyba ve funkci fork()");
            exit(1);

        case 0: /* program provadeny v potomkovi */
            printf ("PID procesu potomka: %d\n", (int) getpid ());
            execlp("sleep", "sleep", "30", (char *) NULL);
            perror ("chyba ve funkci execlp()");
            exit (1);

        default: /* program provadeny v rodici */
            printf ("PID procesu rodice : %d\n", (int) getpid ());
            wait(&status);
    };
    ...
}
```

Příklad: vytvoření procesu v Unixu (2)

- Nový proces vzniká použitím systémových volání:
- **fork()**
 - vytvoří nový proces, který je kopií procesu, z kterého byla tato funkce zavolána
 - v rodičovském procesu vrátí funkce PID potomka (v případě chyby -1)
 - v potomkovi vrátí funkce 0
 - nový proces má jiné PID a PPID, ostatní vlastnosti dědí (např. EUID, EGID, prostředí, ...) nebo sdílí s rodičem (např. soubory, ...)
 - kódový segment sdílí potomek s rodičem
 - datový a zásobníkový segment vznikají kopií dat a zásobníku rodiče

Příklad: vytvoření procesu v Unixu (3)

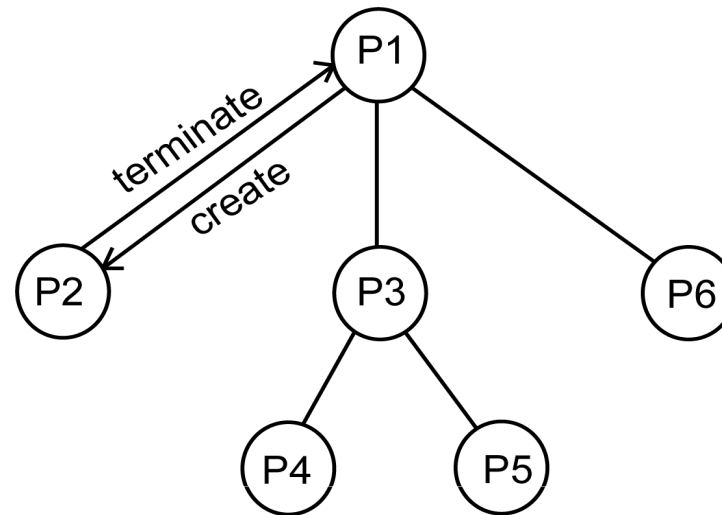
- **exec ()**

- v procesu, ze kterého je funkce volána, spustí nový program (obsah původního procesu je přepsán novým programem)
- atributy procesu se nemění (PID, PPID, ...)

Ukončení procesu

- **Normal exit** (dobrovolné)
 - Když proces dokončí svou práci, použije systémové volání, aby řekl OS, že končí (např. `exit()` v Unixu nebo `ExitProcess()` v MS Windows).
- **Error exit** (dobrovolné)
 - Například když proces zjistí fatální chybu (např. žádný vstupní soubor,...).
- **Fatal error** (nedobrovolné)
 - Chyba způsobená procesem, často např. díky chybě v programu. OS proces násilně ukončí.
- **Ukončení jiným procesem** (nedobrovolné)
 - Proces použije systémové volání, aby řekl OS o ukončení nějakého jiného procesu (např. `kill()` v Unixu nebo `TerminateProcess()` v MS Windows).

Hierarchie procesů



- Vztah mezi rodičovským procesem a potomkem.
- Potomek může **zdědit** některé rysy od svého rodiče (např. kód procesu, globální data,...).
- Na druhé straně, každý nový proces má **své vlastní struktury** (např. ID procesu, reakce na signály,...).

Hierarchie v Unixu

- Jádro OS → proces init
spouštěcí skripty:
 - Solaris: /sbin/rc[0123456S]
 - Fedora: /etc/rc.d/rc
 - ukončení služeb
 - Solaris & Fedora: /etc/rc[0123456S].d/K##...
 - spuštění služeb
 - Solaris & Fedora: /etc/rc[0123456S].d/S##...
- **Informace o procesech**
 - ps
 - pgrep
 - Pstree (Linux)/ptree (Solaris)

Hierarchie ve Windows XP

- Jádro OS → Smss (Session Manager)
Csrss (Windows subsystem process)
Winlogon
services.exe
explorer.exe (user shell)
- **Informace o procesech**
 - Správce úloh (součást OS)
 - Process Explorer (www.sysinternals.com)
 - tasklist

Identita procesu

- **Unix:**

- Uživatel – má jméno a UID, patří aspoň do jedné skupiny
- Skupina – má jméno a GID
- **Efektivní identita**: uživatel (EUID) + skupiny (EGID's)
- **Reálná identita**: uživatel (RUID) + skupiny (RGID's)
- **Privilegia**: definují co proces může (např. v Solarisu 10)

- **Windows XP**

- Uživatel – SID (Security identifier)
- **Access token** – přidělen každému procesu
seznam SIDů, seznam privilegií, restriktce

Změna identity - v Unixu

- **Program:** práva (rwxr-xr-x), vlastník (uživatel - UID, skupina - GID)
- **Shell:** efektivní identita (uživatel - EUID, množina skupin – EGID's)
 - **Proces:** většinou dědí od rodiče (EUID, EGID's)
- **Jak spustit potomka pod jinou identitou**
 - Speciální práva (SUID,SGID)
 - Příkazem sudo
 - Mechanismus Role Base Acces Control (RBAC)

Implementace procesu

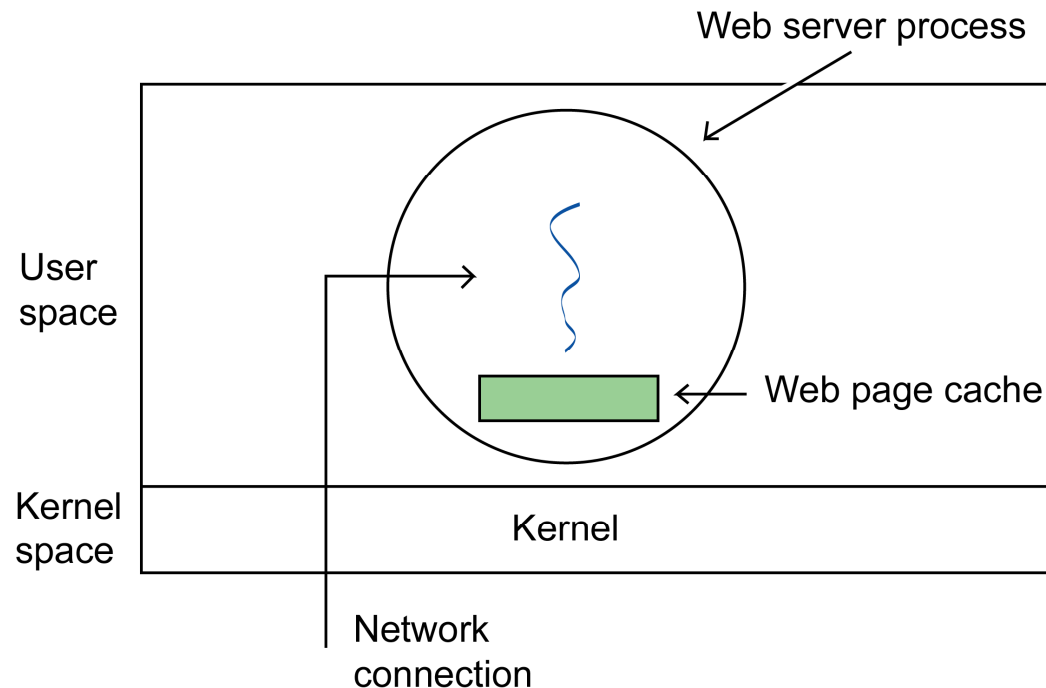
- OS spravuje tabulku (pole struktur), která se nazývá **tabulka procesů**, s jednou položkou pro jeden proces, nazývanou **process control block (PCB)**.
- **PCB** obsahuje **informace o procesu**, které jsou nutné pro správu procesů.
- Například:
 - V Unixu, maximální velikost tabulky procesů je definována parametrem jádra **nproc**.
 - PCB má v Unix přibližně 35 položek.

Položky PCB

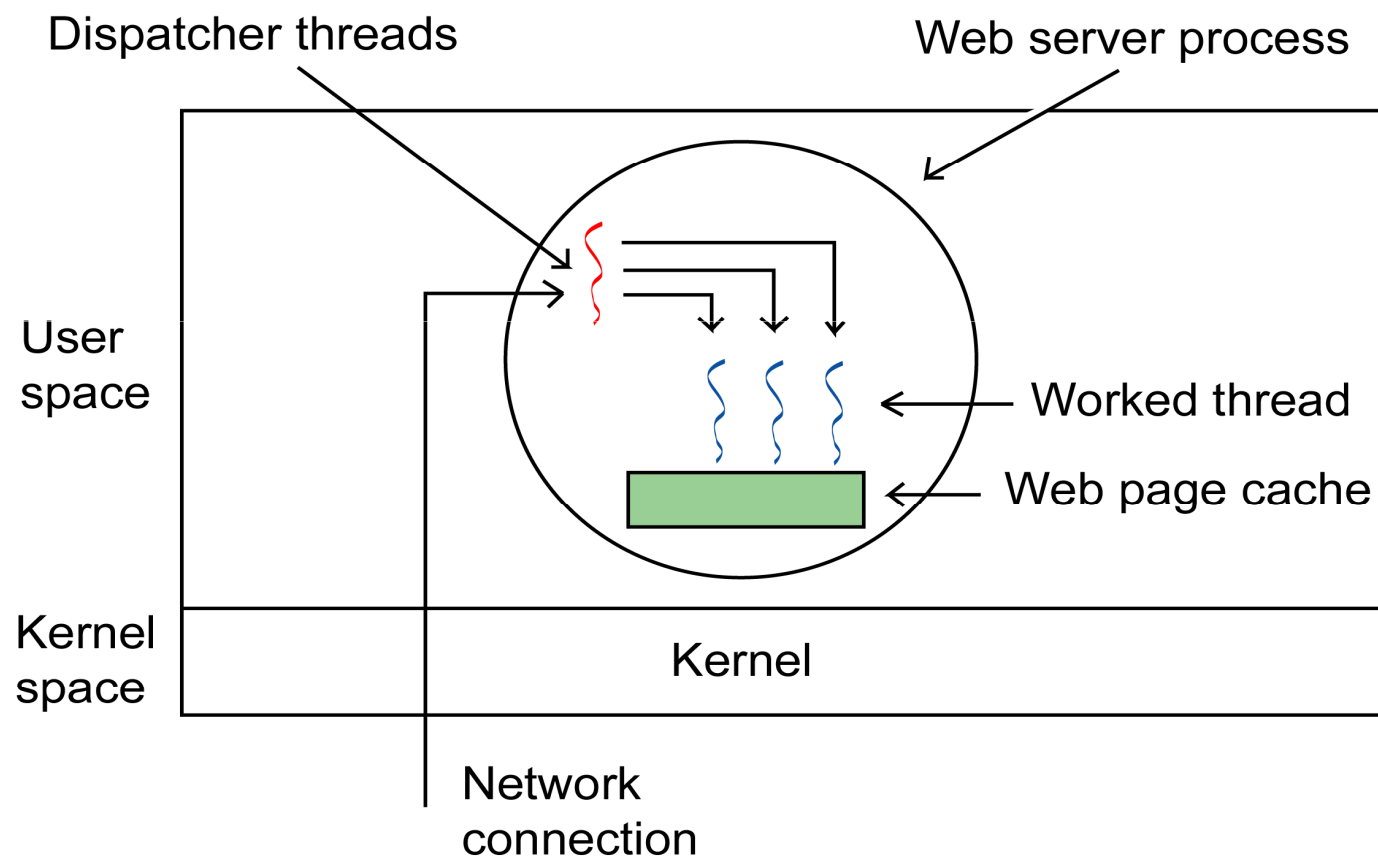
- **Informace pro identifikaci procesu** (process identification)
 - jméno procesu, číslo procesu (PID), rodičovský proces (PPID), vlastník procesu (UID, EUID), seznam vláken, ...
- **Stavové informace procesoru** (processor state information) – pro každé kernel vlákno
 - hodnoty viditelných registrů CPU,
 - hodnoty řídicích a stavových registrů CPU (program counter, program status word (PSW), status information, ...)
 - ukazatelé na zásobníky, ...
- **Informace pro správu procesu** (process control information)
 - Vlákna:
 - stav vlákna, priorita, informace nutné pro plánování
 - informace o událostech, na které proces čeká,
 - informace pro meziprocesovou komunikaci, ...
 - Proces:
 - informace pro správu paměti (ukazatel na tabulku stránek,...)
 - alokované a používané prostředky,...

Příklad: jednovláknový Web Server

- **Klient**
 - pošle požadavek na konkrétní web. stránku
- **Server**
 - ověří zda klient může přistupovat k dané stránce
 - načte stránku a pošle obsah stránky klientovi
- **Často používané stránky** zůstávají uloženy v **hlavní paměti**, abychom minimalizovali čtení z disku.



Příklad: vícevláknový Web Server



Příklad: vícevláknový Web Server (2)

- **Dispatcher thread:** čte příchozí požadavky, zkoumá požadavek, vybere nevyužité pracovní vlákno a předá mu tento požadavek.
- **Worked thread:** načte požadovanou stránku z hlavní paměti nebo disku a pošle ji klientovi.

Dispatcher thread

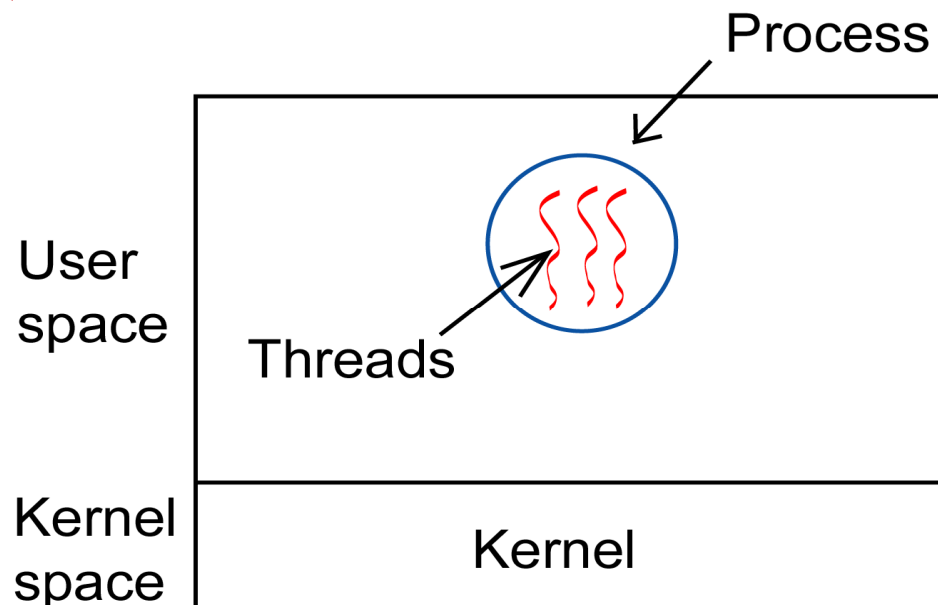
```
while (TRUE) {  
    get_next_request(&buf);  
    handoff_work(&buf);  
}
```

Worked thread

```
while(TRUE) {  
    wait_for_work(&buf);  
    look_for_page_in_cache(&buf,&page);  
    if (page_not_in_cache(&page))  
        read_page_from_disk(&buf,&page);  
    return_page(&page);  
}
```

Vláknový model

- Oddělení alokace prostředků a samotný výpočet.
- **Proces** slouží k **alokaci společných prostředků**.
- **Vlákná** jsou **jednotky plánované pro spuštění** na CPU.
- **Vlákn** má svůj vlastní **program counter** (pro uchování informace o výpočtu), **registry** (pro uchování aktuálních hodnot), **zásobník** (který obsahuje historii výpočtu), **lokální proměnné**, ale **ostatní prostředky** a **identita jsou sdílené**.



Vláknový model (2)

- Jednotlivá vlákna v daném procesu **nejsou nezávislá** tak jako jednotlivé procesy.
- Všechny vlákna v procesu **sdílí** stejný adresový prostor, stejné otevřené soubory, potomky, reakce na signály, ...

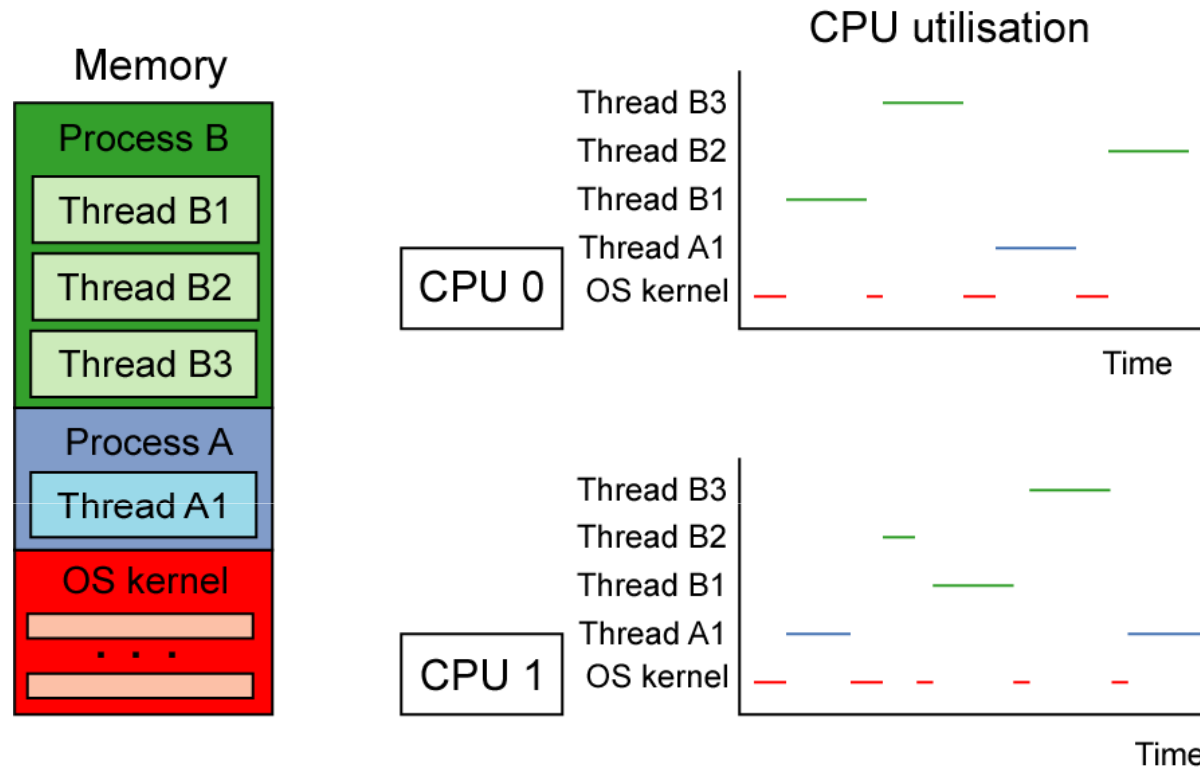
Vytvoření a ukončení vlákna

- Procesy se spouští s jedním vláknem.
- Toto vlákno může **vytvářet další vlákna** pomocí knihovní funkce (např. *pthread_create(name_of_function)*).
- Když chce vlákno skončit, může se opět **ukončit** pomocí knihovní funkce (např. *pthread_exit()*).

Příklad: POSIX vlákna

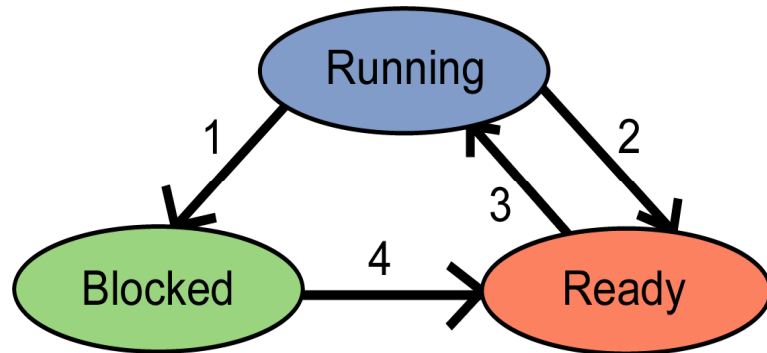
```
...  
void *kod_vlakna(void *threadid)  
{ printf("ID vlakna: %d\n", threadid);  
  sleep(60);  
  pthread_exit(NULL);  
}  
...  
int main()  
{ pthread_t threads[NUM_THREADS];  
  int rc, i;  
  for(i=0; i<NUM_THREADS; i++){  
    rc = pthread_create(&threads[i], NULL, kod_vlakna, (void *) i);  
    if (rc){ perror("Chyba ve funkci pthread_create()"); exit(1); }  
  }  
  ...  
}
```

Přepínání kontextu



- **Pseudo paralelismus:** vlákna běží (pseudo) paralelně na jednoprosesorovém systému díky **přepínání kontextu**, procesor střídavě provádí kód jednotlivých vláken (**multiprogramming, timesharing, multiprocessing**).
- **Skutečný hardwarový paralelismus:** každé vlákno běží na svém procesoru (multiprosesorový systém).

Stavy vláken – třístavový model

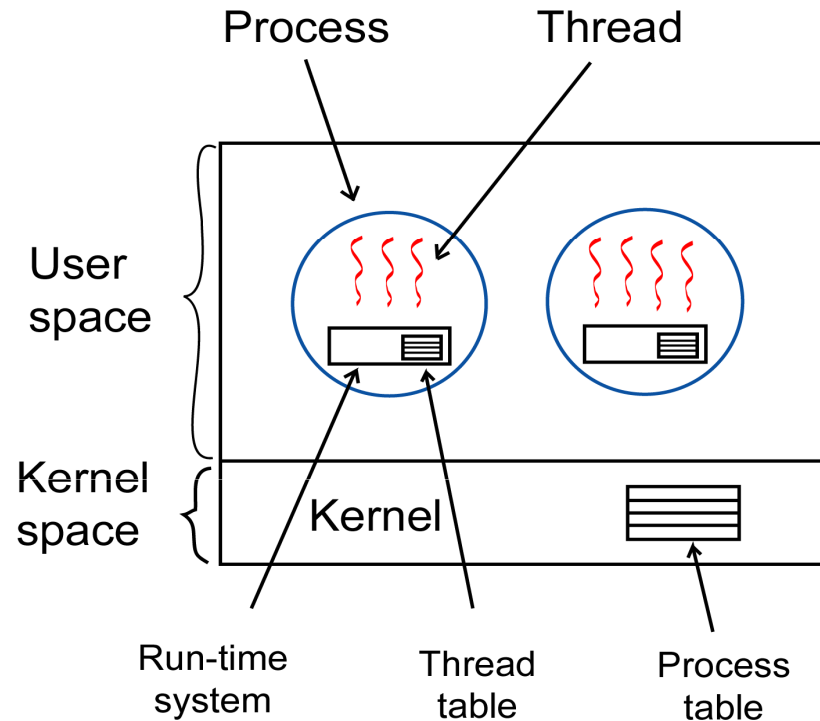


1. Thread blocks for input
2. Scheduler picks another thread
3. Scheduler picks this thread
4. Input becomes available

- **Základní stavy vláken**

- **Running**: v tomto okamžiku právě používá CPU.
- **Ready**: připraven použít CPU, dočasně je proces zastaven a čeká až mu bude přiřazeno CPU.
- **Blocked**: neschopný použít CPU v tomto okamžiku, čeká na nějakou externí událost (např. načtení dat z disku,...).

Implementace vláken v uživatelském prostoru



- **Run-time system:** množina funkcí, která spravuje vlákna.
- Vlákna jsou implementována pouze v uživatelském prostoru.
- Jádro o vláknech nemá žádné informace.

Implementace vláken v uživatelském prostoru (2)

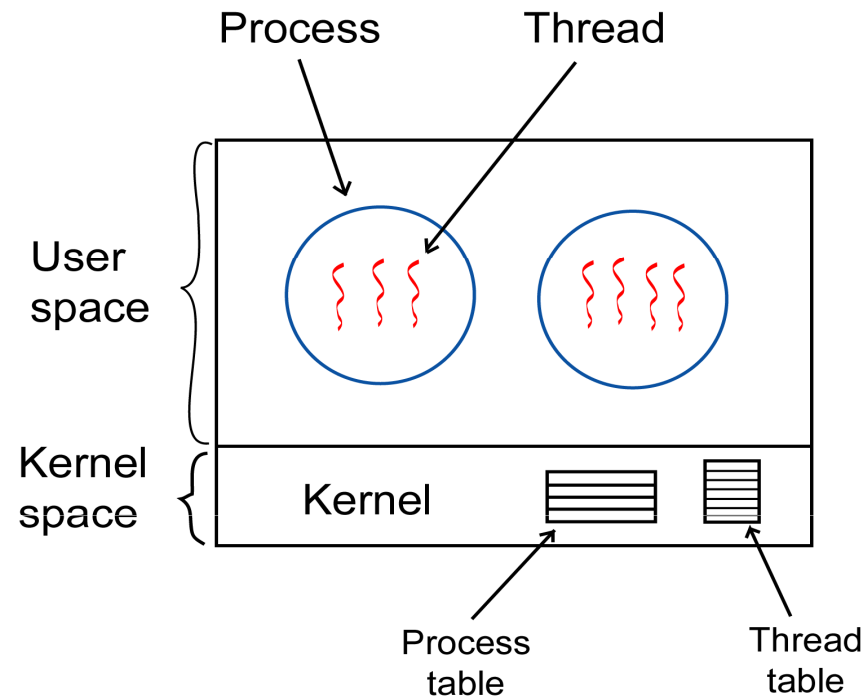
- **Výhody**

- Vlákna mohou být implementována v OS, které nepodporuje vlákna.
- Rychlé plánování vláken.
- Každý proces může mít svůj vlastní plánovací algoritmus.

- **Nevýhody**

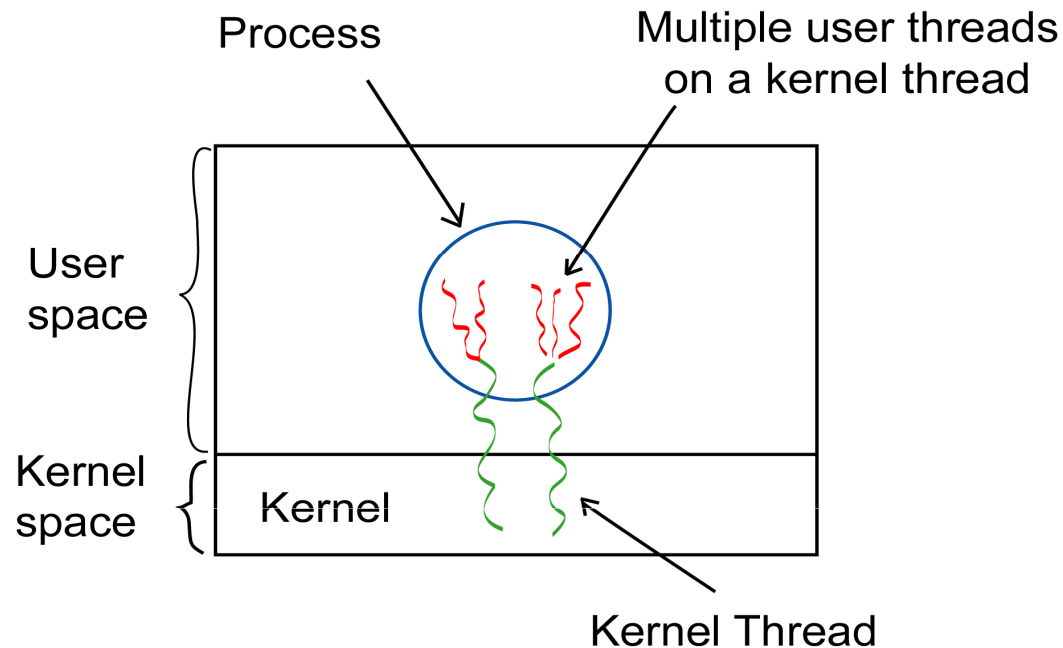
- Jak budou implementována blokující systémová volání? (změna systémových volání na neblokující nebo požití systémového volání `select`)
- Co se stane když dojde k výpadku stránky?
- Žádný clock interrupt uvnitř procesu.
(jedno vlákno může okupovat CPU během celého časového kvanta procesu)

Implementace vláken v prostoru jádra



- Jádro má tabulku vláken, která obsahuje informace o všech vláknech v systému.
- **Výhody**
 - Žádný problém s blokujícími systémovými voláními.
- **Nevýhody**
 - Vytváření, ukončování a plánování vláken je pomalejší.

Hybridní implementace vláken



- Jádro se stará pouze o **kernel-level threads** a plánuje je.
- Některé kernel-level threads mohou mít user-level threads.
- **User-level threads** jsou vytvářena, ukončovaná a plánovaná uvnitř procesu.
- Např. Solaris, Linux, MS Windows

Solaris

- **Project:** množina procesů, které sdílejí kvóty a limity (maximální počet procesů, čas CPU, paměť, ...).
- **Task:** množina procesů, které sdílejí některé kvóty
- **Process:** normální Unixový proces.
- **User-level threads** (ULT): implementovaný pomocí knihovny vláken v adresovém prostoru procesu (neviditelný pro OS).
- **Lightweight processes** (LWP): mapování mezi ULT a kernel vlákny. Každý LWP podporuje jedno nebo více ULT a je mapováno na jedno kernel vlákno.
- **Kernel threads:** základní jednotky, které mohou být plánovány a spuštěny na jednom nebo více CPU.

Vlákná ve Windows XP

- **Job**: množina procesů, které sdílejí kvóty a limity (maximální počet procesů, čas CPU, paměť, ...).
- **Process**: jednotka, která alokuje zdroje. Každý proces má aspoň jedno vlákno (thread).
- **Fiber**: vlákno spravované celé v uživatelském prostoru.
- **Thread**: jednotka plánována jádrem.