

Operační systémy

IOS 2024/2025

Adam Rogalewicz

rogalew@fit.vut.cz

Autor slidů: **Tomáš Vojnar**

vojnar@fit.vut.cz



Správa procesů

❖ **Správa procesů** (process management) zahrnuje:

- **přepínání kontextu** (dispatcher) – fyzické odebrání a přidělování procesoru na základě rozhodnutí plánovače,
- **plánovač** (scheduler) – rozhoduje, který proces (procesy) poběží a případně, jak dlouho,
- **správu paměti** (memory management) – přiděluje paměť,
- podporu **meziprocesové komunikace** (IPC) – signály, RPC, ...

Proces

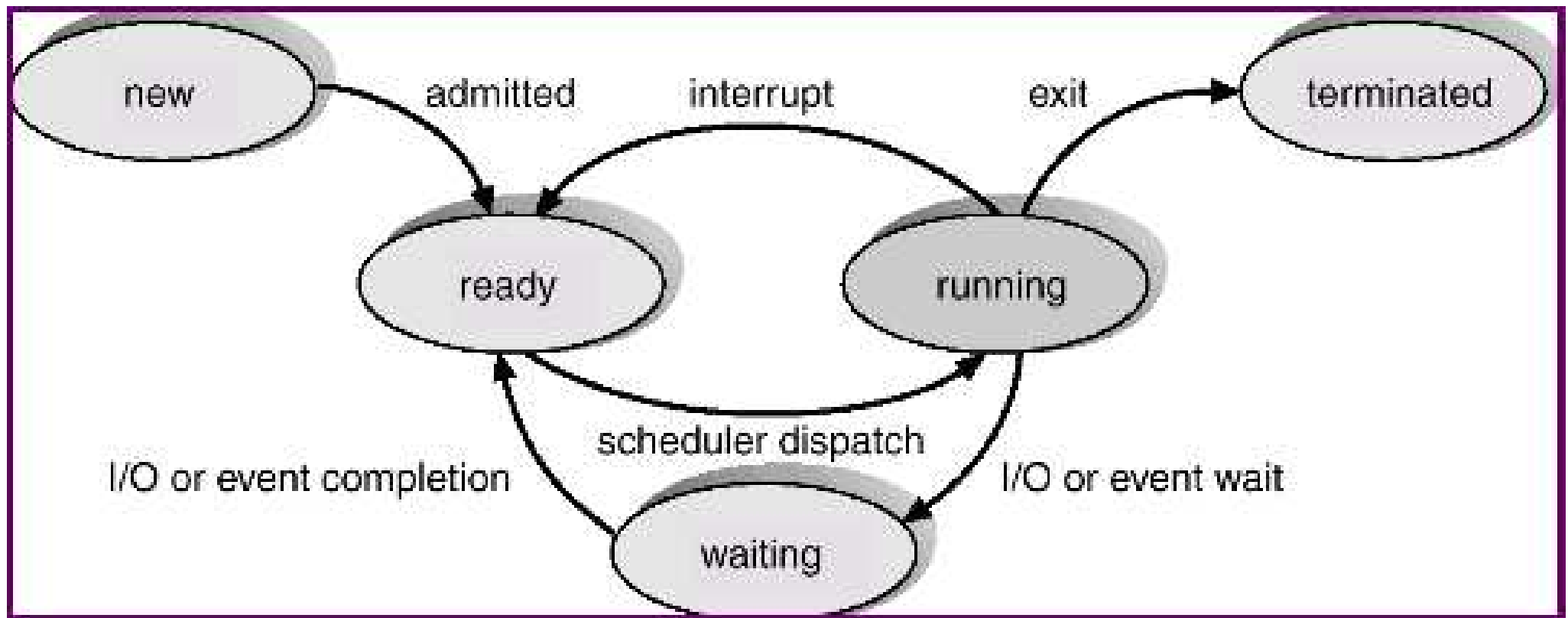
❖ Proces = běžící program.

❖ Proces je v OS definován:

- identifikátorem (PID),
- stavem jeho plánování,
- programem, kterým je řízen,
- obsahem registrů (včetně EIP a ESP apod.),
- zásobníkem – rozpracované funkce,
- daty: statická inicializovaná a neinicializovaná data, hromada, individuálně alokované úseky paměti,
- využitím dalších zdrojů OS a vazbami na další objekty OS: otevřené soubory, signály, PPID, UID, GID, semafore, sdílená paměť, sdílené knihovny, ...

Stavy plánování a jejich změny

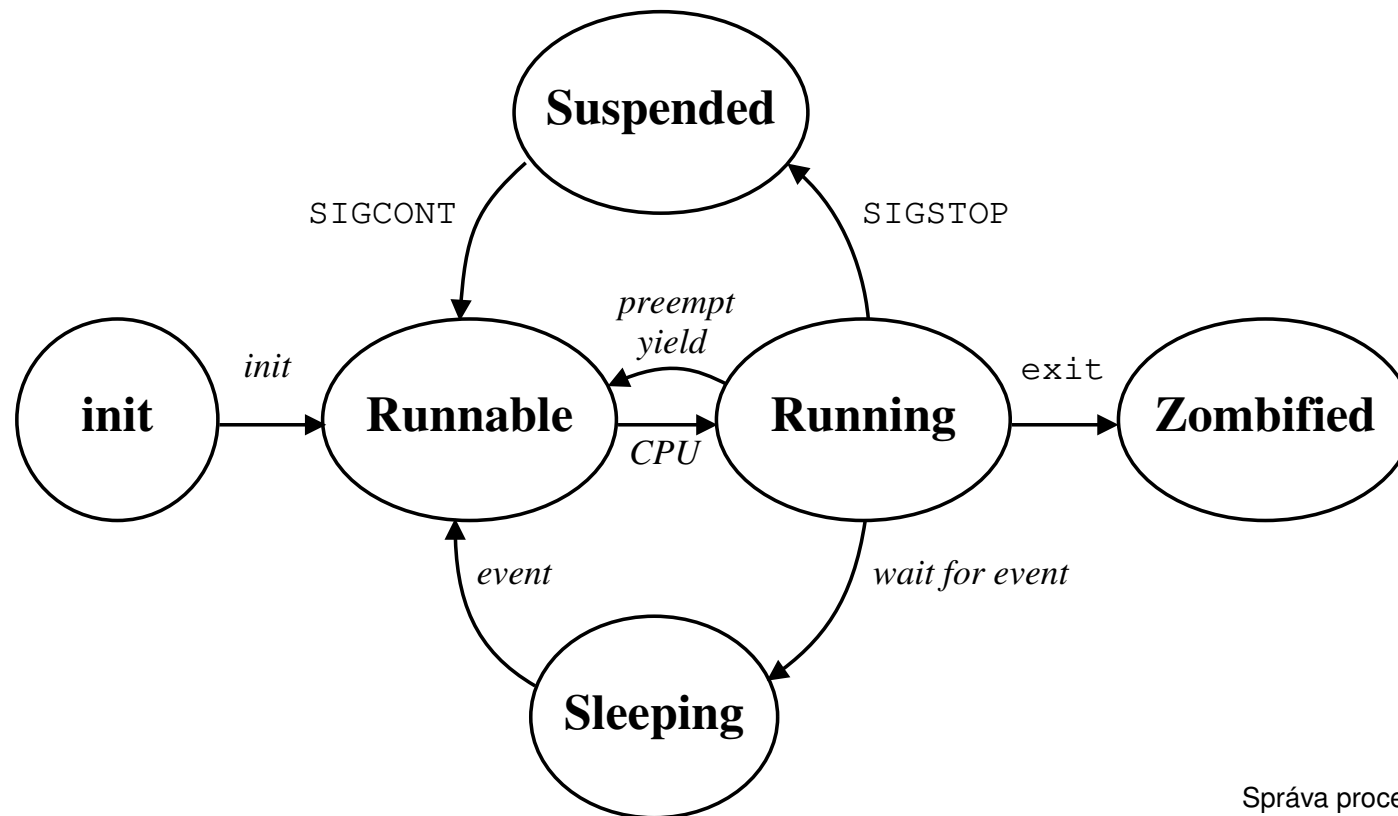
❖ Běžně se rozlišují (různě přejmenované a případně zjemněné) následující stavy procesů:



❖ Stavy plánování procesu v Unixu:

stav	význam
Vytvořený	ještě neinicializovaný
Připravený	mohl by běžet, ale nemá CPU
Běžící	používá CPU
Mátoha	po <code>exit</code> , rodič ještě nepřevzal exit-code
Čekající	čeká na událost (např. dokončení <code>read</code>)
Odložený	"zmrazený" signálem <code>SIGSTOP</code>

❖ Přejchodový diagram stavů plánování procesu v Unixu:



❖ V OS bývá proces **reprezentován** strukturou označovanou jako **PCB (Process Control Block)** nebo též *task control block* či *task struct* apod.

❖ **PCB zahrnuje** (buď přímo, nebo aspoň odkazuje na):

- identifikátory spojené s procesem,
- stav plánování procesu,
- obsah registrů (včetně EIP a ESP apod.),
- plánovací informace (priorita, ukazatele na plánovací fronty, ...),
- informace spojené se správou paměti (tabulky stránek, ...),
- informace spojené s účtováním (spotřeba procesoru, ...),
- využití I/O zdrojů (otevřené soubory, používaná zařízení, ...).

❖ PCB může být někdy rozdělen do několika dílčích struktur.

Části procesu v paměti v Unixu

❖ Uživatelský adresový prostor (user address space) přístupný procesu:

- kód (code area/text segment),
- data (inicializovaná/neinicializovaná data, hromada, individuálně alokovaná paměť),
- zásobník,
- soukromá data sdílených knihoven, sdílené knihovny, sdílená paměť.

❖ Uživatelská oblast (user area) – ne vždy použita:

- Uložena zvlášť pro každý proces spolu s daty, kódem a zásobníkem v user address space příslušného procesu (s nímž může být odložena na disk).
- Je ale přístupná pouze jádru.
- Obsahuje část PCB, která je používána zejména za běhu procesu:
 - PID, PPID, UID, EID, GID, EGID,
 - obsah registrů,
 - deskriptory souborů,
 - obslužné funkce signálů,
 - účtování (spotřebovaný čas CPU, ...),
 - pracovní a kořenový adresář, ...

❖ Záznam v **tabulce procesů** (process table):

- Uložen trvale v jádru.
- Obsahuje zejména informace o procesu, které jsou **důležité, i když proces neběží**:
 - PID, PPID, UID, EID, GID, EGID,
 - stav plánování,
 - událost, na kterou se čeká,
 - plánovací informace (priorita, spotřeba času, ...),
 - čekající signály,
 - odkaz na tabulku paměťových regionů procesu,
 - ...

❖ Tabulka **paměťových regionů procesu** (per-process region table) – popis paměťových regionů procesu (spojitá oblast virtuální paměti použitá za určitým účelem: data, kód, zásobník, sdílenou paměť) + příslušné položky **tabulky regionů**, **tabulka stránek**.

❖ **Zásobník jádra** využívaný za běhu služeb jádra pro daný proces.

Kontext procesu

- ❖ Někdy se též používá pojem **kontext procesu** = stav procesu.
- ❖ Rozlišujeme:
 - **uživatelský kontext** (user-level context): kód, data, zásobník, sdílená data,
 - **registrový kontext**,
 - **systémový kontext** (system-level context): uživatelská oblast, položka tabulky procesů, tabulka paměťových regionů procesu, ...

Systemová volání nad procesy v Unixu

❖ Systemová volání spojená s procesy v Unixu:

- fork, exec, exit, wait, waitpid,
- kill, signal,
- getpid, getppid,
- ...

❖ Identifikátory spojené s procesy v UNIXu:

- identifikace procesu **PID**,
- identifikace předka **PPID**,
- reálný (skutečný) uživatel, skupina uživatelů **UID**, **GID**,
- efektivní uživatel, skupina uživatelů **EUID**, **EGID**,
- **uložená EUID**, **uložená EGID** – umožňuje dočasně snížit efektivní práva a pak se k nim vrátit (při zpětném nastavení se kontroluje, zda se proces vrací k reálnému ID nebo k uloženému EUID),
- v Linuxu navíc **FSUID** a **FSGID** (pro přístup k souborům se zvýšenými privilegii),
- skupina procesů a sezení, do kterých proces patří – **PGID**, **SID**.

Vytváření procesů

❖ Vznik procesů v UNIXu – služba `fork`: duplikuje proces na takřka identického potomka:

- dědí řídicí kód, data, zásobník, sdílenou paměť, otevřené soubory, obsluhu signálů, většinu synchronizačních prostředků, ...;
 - Pro efektivitu používá pro práci s pamětí `copy-on-write`.
- liší se v návratovém kódu `fork`, identifikátorech, údajích spojených s plánováním a účtováním (spotřeba času, ...), nedědí čekající signály, souborové zámky a některé další specilizované zdroje a nastavení.

```
pid=fork();
if (pid==0) {
    // kód pro proces potomka
    // exec(...), exit(exitcode)
} else if (pid==-1) {
    // kód pro rodiče, nastala chyba při fork()
    // errno obsahuje bližší informace
} else {
    // kód pro rodiče, pid = PID potomka
    // pid2 = wait(&stav);
}
```

❖ Vzniká vztah rodič–potomek (parent–child) a hierarchie procesů.

Hierarchie procesů v Unixu

- ❖ Předkem všech uživatelských procesů je proces s PID=1.
 - Starší systémy—proces `init`
 - Novější systémy (Linux od roku 2010)—proces `systemd`
- ❖ Pokud procesu skončí předek, jeho předkem se automaticky stane `init/systemd`, který později převezme jeho návratový kód (proces nemůže definitivně skončit a jako `zombie` čeká, dokud neodevzdá návratový kód).
- ❖ Výpis stromu procesů: např. `ps tree`.
- ❖ Existují `procesy jádra` (kernel processes/threads), jejichž předkem `init/systemd` není:
 - Jejich kód je součástí jádra, běží v režimu jádra.
 - Vyskytuje se i proces s PID=0 s různými rolemi: podíl na inicializaci jádra, následně `swapper` (FreeBSD) či `idle smyčka` (Linux, nevypisuje se).
 - Na Linuxu existuje proces jádra `kthreadd`, který spouští ostatní procesy jádra a je jejich předkem.
 - Vztahy mezi procesy jádra nejsou příliš významné a mohou se lišit.

Změna programu – exec

❖ Skupina funkcí:

- `execve` – základní volání,
- `execl`, `execlp`, `execle`, `execv`, `execvp`.
- `execl("/bin/ls", "ls", "-l", NULL);`
– spouštěný program a jeho argumenty odpovídající `$0/argv[0]`, `$1/argv[1]`, ...

❖ Nevrací se, pokud nedojde k chybě!

❖ Procesu zůstává řada jeho zdrojů a vazeb v OS (identifikátory, otevřené soubory, ...), zanikají vazby a zdroje vázané na původní řídicí kód (obslužné funkce signálů, sdílená paměť, paměťově mapované soubory, semaforey).

❖ Windows: `CreateProcess(...)` – zahrnuje funkčnost `fork` i `exec`.

Čekání na potomka – wait, waitpid

- ❖ Systémové volání `wait` umožňuje pasivní čekání na potomka.
 - Vrací PID ukončeného procesu (nebo -1: příchod signálu, neexistence potomků).
 - Přes argument zpřístupňuje návratový kód potomka.
 - Pokud nějaký potomek je již ukončen a čeká na předání návratového kódu, končí okamžitě.

- ❖ Obecnější je systémové volání `waitpid`:
 - Umožňuje čekat na určitého potomka či potomka z určité skupiny.
 - Umožňuje čekat i na pozastavení či probuzení z pozastavení příjmem signálu.

Start systému

❖ Typická posloupnost akcí při startu systému:

1. **Firmware** (BIOS/UEFI).
2. Načtení a spuštění **zavaděče OS**, někdy v několika fázích (např. BIOS využívá kód v MBR a následně v dalších částech disku).
3. Načtení **jádra**, spuštění jeho inicializačních funkcí.
4. Inicializační funkce jádra mj. vytvoří **proces jádra 0**, ten vytvoří případné další procesy jádra a proces *init*.
5. **init** (systemd) načítá inicializační konfigurace a spouští další procesy.
 - V určitém okamžiku spustí **gdm/sddm/lightdm/...** pro přihlášení v grafickém režimu: z něj či z něj notifikovaných spolupracujících procesů se pak spouští další procesy pro práci pod X Window.
 - Na konzolích spustí **getty**, který umožní zadat přihlašovací jméno a změnit se na **login**. Ten načte heslo a změnit se na shell, ze kterého se spouští další procesy. Po ukončení se na terminálu spustí opět **getty**.
 - Proces **init** i **nadále běží**, přebírá návratové kódy procesů, jejichž rodič již skončil a řeší případnou reinicializaci systému či jeho částí při výskytu různých nakonfigurovaných událostí nebo na přání uživatele.

Úrovně běhu

- ❖ V UNIXu System V byl zaveden **system úrovní běhu** – SYSV init run-levels: 0-6, s/S (0=halt, 1=single user, s/S=alternativní přechod do single user, 6=reboot).
- ❖ Změna úrovně běhu: `telinit N`.
- ❖ **Konfigurace:**
 - Adresáře */etc/rcX.d* obsahují odkazy na skripty spouštěné při vstupu do určité úrovně. Spouští se v pořadí daném jejich jmény, skripty se jménem začínajícím *K* s argumentem *stop*, skripty se jménem začínajícím *S* s parametrem *start*.
 - Vlastní implementace v adresáři */etc/init.d*, lze spouštět i ručně (např. také s argumentem *reload* či *restart* – reinicializace různých služeb, např. sítě).
 - Soubor */etc/inittab* obsahuje hlavní konfigurační úroveň: např. implicitní úroveň běhu, odkaz na skript implementující výše uvedené chování, popis akcí při mimořádných situacích, inicializace konzolí apod.
- ❖ Existují různé nové implementace procesu *init* – např. **systemd**:
 - Úrovně běhu nahrazují **jednotky** (units) různých typů (služby, sockets, přípojně body, ...): `/lib/systemd/`, `/usr/lib/systemd/`, ...
 - Spouští inicializační jednotky **paralelně** na základě jejich závislostí.
 - **Emuluje úrovně běhu.**

Plánování procesů

- ❖ Plánovač rozhoduje, **který proces** (procesy) poběží a případně, **jak dlouho**.
- ❖ **Nepreemptivní plánování**: ke změně běžícího procesu může dojít pouze tehdy, pokud to běžící proces umožní **předáním řízení jádru** tím, že **požádá o službu**:
 - **typicky** I/O operace, konec – volání `exit`, vzdání se procesoru – volání `yield`.
- ❖ **Preemptivní plánování**: mimo výše uvedené může navíc ke změně běžícího procesu dojít, aniž by tento jakkoliv přepnutí kontextu napomohl, a to na základě **přerušení**:
 - **typicky** od časovače, ale může se jednat i o jiné přerušení (např. disk, ...).
- ❖ Vlastní přepnutí kontextu řeší na základě rozhodnutí plánovače tzv. **dispečer**.
- ❖ Plánování může být též ovlivněno systémem **swapování** rozhodujícím o tom, kterým procesům je přidělena paměť, aby mohly běžet, případně systémem **spouštění nových procesů**, který může spuštění procesů odkládat na vhodný okamžik.
 - V těchto případech někdy hovoříme o **střednědobém a dlouhodobém plánování**.

Přepnutí procesu (kontextu)

- ❖ Dispečer odebere procesor procesu A a přidělí ho procesu B, což typicky zahrnuje:
 - úschovu stavu (některých) registrů (včetně různých řídicích registrů) v rámci procesu A do PCB,
 - úpravu některých řídicích struktur v jádře (např. v rámci plánovače, správy paměti, účtování apod.),
 - obnovu stavu (některých) registrů v rámci procesu B z PCB,
 - předání řízení na adresu, kde bylo dříve přerušeno provádění procesu B.

- ❖ Neukládá se/neobnovuje se celý stav procesů: např. se uloží jen ukazatel na tabulku stránek, tabulka stránek a vlastní obsah paměti procesu může zůstat.

- ❖ Přepnutí trvá přesto typicky **stovky až tisíce instrukcí**: interval mezi přepínáním musí být tedy volen tak, aby režie přepnutí nepřevážila běžný běh procesů.

Klasické plánovací algoritmy

❖ Klasické plánovací algoritmy uvedené níže se dají užít přímo, ale často se užívají spíše v různých modifikacích a/nebo kombinacích.

❖ FCFS (First Come, First Served):

- Procesy čekají na přidělení procesoru ve **FIFO frontě**.
- Při **vzniku** procesu, jeho **uvolnění z čekání** (na I/O, synchronizaci apod.), nebo **vzdá-li** se proces procesoru, je tento proces zařazen na konec fronty.
- Procesor se přiděluje procesu na začátku fronty.
- Algoritmus je **nepreemptivní** a k přepnutí kontextu dojde pouze tehdy, pokud se běžící proces vzdá procesoru (voláním služeb např. pro I/O, konec, dobrovolné vzdání se procesoru – volání `yield`, Linux: `sched_yield`).

❖ Round-robin – **preemptivní** obdoba FCFS:

- Pracuje podobně jako FCFS, navíc má ale každý proces přiděleno **časové kvantum**, po jehož vypršení je mu odebrán procesor a proces je zařazen na konec fronty připravených procesů.

Klasické plánovací algoritmy

❖ SJF (Shortest Job First):

- Přiděluje procesor procesu, který požaduje **nejkratší dobu** pro své další **provádění na procesoru** bez I/O operací – tzv. CPU burst.
- **Nepreemptivní** algoritmus, který nepřerušuje proces před dokončením jeho aktuální výpočetní fáze.
- **Minimalizuje průměrnou dobu čekání, zvyšuje propustnost systému.**
- Nutno znát dopředu dobu běhu procesů na procesoru nebo mít možnost tuto rozumně odhadnout na základě předchozího chování.
- Používá se pro **opakovaně prováděné podobné úlohy**, zejména ve specializovaných (např. dávkových) systémech.
- Hrozí **stárnutí** (někdy též hladovění – starvation):
 - Stárnutí při přidělování zdrojů (procesor, zámek, ...) je obecně situace, kdy některý proces, který o zdroj žádá, na něj čeká bez záruky, že jej někdy získá.
 - V případě SJF hrozí hladovění procesů čekajících na procesor a majících dlouhé výpočetní fáze, které mohou být neustále předbíhány kratšími výpočty.

❖ SRT (Shortest Remaining Time): obdoba SJF s **preempcí** při vzniku či uvolnění procesu.

Klasické plánovací algoritmy

❖ Víceúrovňové plánování:

- Procesy jsou rozděleny do různých skupin:
 - typicky podle priority, ale lze i jinak, např. dle typu procesu.
- V rámci každé skupiny může být použit jiný dílčí plánovací algoritmus (např. FCFS či round-robin).
- Je použit také algoritmus, který určuje, ze které skupiny bude vybrán proces, který má aktuálně běžet – často jednoduše na základě priority skupin.
- Může hrozit hladovění některých (obvykle nízko prioritních) procesů.

❖ Víceúrovňové plánování se zpětnou vazbou:

- Víceúrovňové plánování se skupinami procesů rozdělenými dle priorit.
- Proces nově připravený běžet je zařazen do fronty s nejvyšší prioritou, postupně klesá do nižších prioritních front, nakonec plánován round-robin na nejnižší úrovni.
- Používají se varianty, kdy je proces zařazen do počáteční fronty na základě své statické priority. Následně se může jeho dynamická priorita snižovat, spotřebovává-li mnoho procesorového času, nebo naopak zvyšovat, pokud hodně čeká na vstup/výstupních operacích.
 - Cílem je zajistit rychlou reakci interaktivních procesů.

Plánovač v Linuxu od verze 6.6

❖ Víceúrovňové prioritní plánování se 100 základními statickými prioritními úrovněmi, doplněné o rozlišování **několika typů procesů**:

- Priority 1–99 pro procesy **reálného času** plánované
 - FCFS (doplněným o preempci na základě priorit),
 - round-robin, nebo
 - plánováním pro **sporadické periodické úlohy** s očekávanou dobou výpočetní fáze a časovým limitem, do kdy se má provést.
- Priorita 0 pro **běžné procesy** plánované tzv. **EEVDF** plánovačem.
 - Poznámka: V roce 2023 nahradil CFS plánovač a ten dříve O(1) plánovač.
- V rámci úrovně 0:
 - jsou používány podúrovně v rozmezí -20 (nejvyšší) až 19 (nejnižší), nastavené uživatelem příkazy **nice** či **renice**;
 - rozlišuje se plánování pro **běžné**, **dávkové** (delší kvantum, jistá penalizace) a „**idle**“ procesy (zvláště nízká priorita).
- Základní prioritní úroveň a typ plánování mohou ovlivnit procesy s patřičnými právy: viz služba `sched_setscheduler`.

Plánovač EEVDF

❖ EEVDF (Earliest Eligible Virtual Deadline First):

- V základním případě se snaží explicitně každému procesu poskytnout **stejné procento strojového času**: 5 procesů, plánovací perioda 1 s – každý proces 200 ms.
- U procesů s **vyšší prioritou** (nižším "nice") cílí na **větší procento**.
- V reálném světě proces obvykle neběží přesně po tu dobu, na jakou má "právo".
 - Počítá se **zpoždění** ("lag"): kladné – dostal méně, záporné – dostal více.
- Procesor mohou dostat pouze procesy s aktuálně **kladným zpožděním**.
 - U ostatních se zpoždění v čase postupně stane kladným.
 - Procesy s **větším zpožděním** mají **přednost**.
- Procesy vyžadující **rychlou reakci**: **kratší časové kvantum** ("latency nice").
 - **Virtuální deadline** – do kdy má/mělo být dokončeno další kvantum procesu: aktuální zpoždění minus kvantum.
 - První běží proces s největší hodnotou virtuálního deadline.
 - Největší zpoždění konce dalšího kvanta.
 - Kratší kvantum: proces poběží v kratších úsecích, ale rychleji reaguje.
- Implementováno s využitím efektivní vyhledávací stromové struktury.

Plánování ve Windows NT a novějších

❖ Víceúrovňové prioritní plánování se zpětnou vazbou na základě interaktivity:

- 32 prioritních úrovní: 0 – nulování volných stránek, 1 – 15 běžné procesy, 16 – 31 procesy reálného času.
- Základní priorita je dána staticky.
- Systém může prioritu běžných procesů dynamicky zvyšovat či snižovat:
 - Zvyšuje prioritu procesů spojených s oknem, které se dostane na popředí.
 - Zvyšuje prioritu procesů spojených s oknem, do kterého přichází vstupní zpráva (myš, časovač, klávesnice, ...).
 - Zvyšuje prioritu procesů, které jsou uvolněny z čekání (např. na I/O operaci).
 - Zvýšená priorita se snižuje po každém vyčerpání kvanta o jednu úroveň až do dosažení základní priority.

Inverze priorit

❖ Problém inverze priorit:

- Nízko prioritní proces si naalokuje nějaký zdroj, více prioritní procesy ho předbíhají a nemůže dokončit práci s tímto zdrojem.
- Časem tento zdroj mohou potřebovat více prioritní procesy, jsou nutně zablokovány a musí čekat na nízko prioritní proces.
- Pokud v systému jsou v tomto okamžiku středně prioritní procesy, které nepotřebují daný zdroj, pak poběží a budou dále předbíhat nízko prioritní proces.
- Tímto způsobem uvedené středně a nízko prioritní procesy získávají efektivně vyšší prioritu.

❖ Inverze priorit **nemusí, ale může, vadit**: může **zvyšovat odezvu systému** a způsobit i vážnější problémy, zejména pokud jsou blokovány nějaké kritické procesy reálného času (ovládání hardware apod.).

Inverze priorit a další komplikace plánování

❖ Možnosti řešení inverze priorit:

- **Priority ceiling**: procesy v kritické sekci získávají nejvyšší prioritu.
- **Priority inheritance**: proces v kritické sekci, který blokuje výše prioritní procesy, dědí (po dobu běhu v kritické sekci) prioritu čekajícího procesu s největší prioritou.
- **Zákaz přerušení po dobu běhu v kritické sekci** (na jednoprocessorovém systému): proces v podstatě získává vyšší prioritu než všichni ostatní.

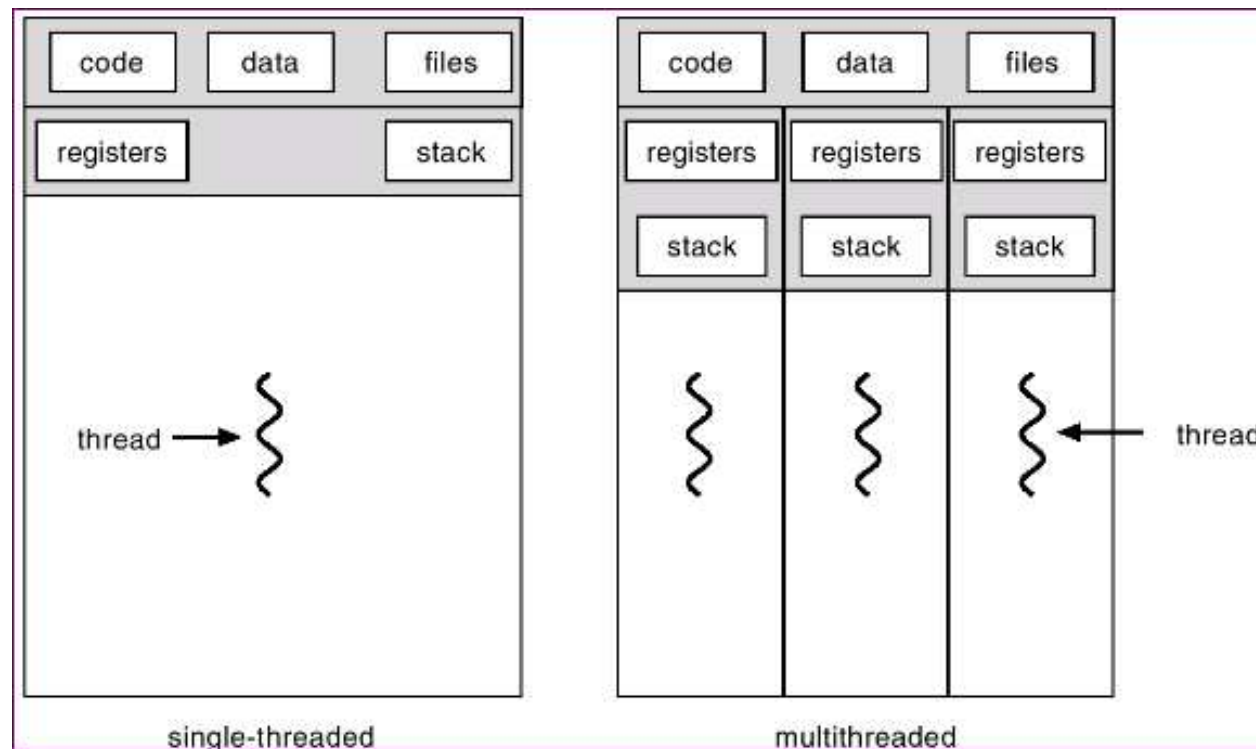
❖ Další výraznou komplikaci plánování představují:

- **víceprocesorové systémy** – nutnost vyvažovat výkon, respektovat obsah cache procesorů, příp. lokalitu pamětí (při neuniformním přístupu do paměti),
- **hard real-time systémy** – nutnost zajistit garantovanou odezvu některých akcí.

Vlákna, úlohy, skupiny procesů

❖ Vlákno (thread):

- „odlehčený proces“ (LWP – lightweight process), „podproces“,
- jednotka výpočtu, kterých může v rámci procesu běžet více paralelně,
- **vlastní** obsah registrů (včetně EIP a ESP) a zásobník,
- **sdílí** kód, data a další zdroje (otevřené soubory, signály),
- **výhody** – nižší režie: rychlejší spouštění, přepínání apod.



Úlohy, skupiny procesů, sezení

❖ **Úloha** (job) v bashi (a podobných shellech): skupina paralelně běžících procesů spuštěných jedním příkazem a propojených do kolony (pipeline).

❖ **Skupina procesů** (process group) v UNIXu:

- Množina procesů, které je možno poslat signál jako jedné jednotce. Předek také může čekat na dokončení potomka z určité skupiny (`waitpid`).
- Každý proces je v jedné skupině procesů, po vytvoření je to skupina jeho předka.
- Skupina může mít tzv. vedoucího – její první proces, pokud tento neskončí (skupina je identifikována číslem procesu svého vedoucího).
- **Zjišťování a změna skupiny**: `getpgid`, `setpgid`.

❖ **Sezení** (session) v UNIXu:

- Každá skupina procesů je v jednom sezení. Sezení může mít vedoucího.
- Vytvoření nového sezení: `setsid`.
- Sezení může mít řídicí terminál (`/dev/tty`).
- Jedna skupina sezení je tzv. na popředí (čte z terminálu), ostatní jsou na pozadí.
- O ukončení terminálu je signálem `SIGHUP` informován vedoucí sezení (typicky shell), který ho dále řeší – všem, na které nebyl užit `nohup/disown`, `SIGHUP`, pozastaveným navíc `SIGCONT`.

Komunikace procesů

❖ IPC = Inter-Process Communication:

- signály (kill, signal, ...)
- roury (pipe, mknod p, ...)
- zprávy (msgget, msgsnd, msgrcv, msgctl, ...)
- sdílená paměť (shmget, shmat, ...)
- sockety (socket, ...)
- RPC = Remote Procedure Call
- ...

Signály

❖ **Signál** (v základní verzi) je číslo (`int`) zaslané procesu prostřednictvím pro to zvláště definovaného rozhraní. Signály jsou generovány

- **při chybách** (např. aritmetická chyba, chyba práce s pamětí, ...),
- **externích událostech** (vypršení časovače, dostupnost I/O, ...),
- **na žádost procesu** – IPC (`kill`, ...).

❖ Signály často vznikají **asynchronně** k činnosti programu – **není tedy možné jednoznačně předpovědět, kdy daný signál bude doručen.**

❖ **Nutno pečlivě zvažovat obsluhu**, jinak mohou vzniknout „záhadné“, zřídka se objevující a velice špatně laditelné chyby – mj. také motivace pro **pokročilé testování** (vkládání šumu, systematická enumerace prokládání do určité meze) a **verifikaci s formálními základy** (statická analýza, model checking).

❖ Mezi běžně používané signály patří:

- SIGHUP – odpojení, ukončení terminálu
- SIGINT – přerušení z klávesnice (Ctrl-C)
- SIGKILL – tvrdé ukončení
- SIGSEGV, SIGBUS – chybný odkaz do paměti
- SIGPIPE – zápis do roury bez čtenáře
- SIGALRM – signál od časovače (alarm)
- SIGTERM – měkké ukončení
- SIGUSR1, SIGUSR2 – uživatelské signály
- SIGCHLD – pozastaven nebo ukončen potomek
- SIGCONT – pokračuj, jsi-li pozastaven
- SIGSTOP, SIGTSTP – tvrdé/měkké pozastavení

❖ Další signály – viz [man 7 signal](#).

Předefinování obsluhy signálů

- ❖ Mezi implicitní (přednastavené) reakce na signál patří: ukončení procesu (příp. s generováním core dump), ignorování signálu, zmrazení/rozmrazení procesu.
- ❖ Lze předefinovat obsluhu všech signálů mimo SIGKILL a SIGSTOP. SIGCONT lze předefinovat, ale vždy dojde k odblokování procesu.
- ❖ K předefinování obsluhy signálů slouží funkce:
 - `sighandler_t signal(int signum, sighandler_t handler);` kde `typedef void (*sighandler_t)(int);`
 - handler je ukazatel na **obslužnou funkci**, příp. SIG_DFL či SIG_IGN.
 - `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`
 - komplexnější a přenosné nastavení obsluhy,
 - nastavení blokování signálů během obsluhy (jinak stávající nastavení + obsluhovaný signál),
 - nastavení režimu obsluhy (automatické obnovení implicitní obsluhy, ...),
 - nastavení obsluhy s příjmem dodatečných informací spolu se signálem.
- ❖ Z obsluhy signálu lze volat pouze vybrané **bezpečné knihovní funkce**: u ostatních hrozí nekonzistence při interferenci s rozpracovanými knihovními voláními.

Blokování signálů

❖ K nastavení masky blokováných signálů lze užít:

- `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);`
- `how` je `SIG_BLOCK`, `SIG_UNBLOCK` či `SIG_SETMASK`.

❖ K sestavení masky signálů slouží `sigemptyset`, `sigfillset`, `sigaddset`, `sigdelset`, `sigismember`.

❖ Nelze blokovat: `SIGKILL`, `SIGSTOP`, `SIGCONT`.

❖ Nastavení blokování se dědí do potomků. Dědí se také obslužné funkce signálů, při použití `exec` se ovšem nastaví implicitní obslužné funkce.

❖ Zjištění čekajících signálů: `int sigpending(sigset_t *set);`

❖ Upozornění: Pokud je nějaký zablokovaný signál přijat vícekrát, zapamatuje se jen jedna instance! (Neplatí pro tzv. real-time signály.)

Zasílání signálů

- ❖ `int kill(pid_t pid, int sig);` umožňuje zasílat signály
 - určitému procesu,
 - skupině procesů – odesílatelovy či explicitně zadané,
 - všem procesům, kterým daný proces může signál poslat (mimo `init`, pokud nemá pro příslušný signál definovanou obsluhu).

- ❖ Aby proces mohl zaslat signál jinému procesu musí odpovídat jeho UID nebo EUID UID nebo saved set-user-ID cílového procesu (`SIGCONT` lze zasílat všem procesům v sezení – session), případně se musí jednat o privilegovaného odesílatele (např. `EUID=0`, nebo kapabilita `CAP_KILL`).

- ❖ Poznámka: `sigqueue` – zasílání signálu spolu s dalšími daty.

Čekání na signál

❖ **Jednoduché čekání:** `int pause(void);`

- Nelze spolehlivě přepínat mezi signály, které mají být blokovány po dobu, kdy se čeká, a mimo dobu, kdy se čeká.

❖ **Zabezpečené čekání:** `int sigsuspend(const sigset_t *mask);`

- Lze spolehlivě přepínat mezi signály blokovány mimo a po dobu čekání.
- `mask` jsou blokovány po dobu čekání, po ukončení se nastaví původní blokování.

❖ **Příklad** – proces spustí potomka a počká na signál od něj:

```
#include <signal.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

/* When a SIGUSR1 signal arrives, set this variable. */
volatile sig_atomic_t usr_interrupt = 0;

void synch_signal (int sig) {
    usr_interrupt = 1;
}

/* The child process executes this function. */
void child_function (void) {

    /* Let the parent know you're here. */
    kill (getppid (), SIGUSR1);

    exit(0);

}
```

❖ Pokračování příkladu – pozor na zamezení ztráty signálu mezi testem a čekáním:

```
int main (void) {
    struct sigaction usr_action;
    sigset_t block_mask, old_mask;
    pid_t pid;

    /* Establish the signal handler. */
    sigfillset (&block_mask);
    usr_action.sa_handler = synch_signal;
    usr_action.sa_mask = block_mask;
    usr_action.sa_flags = 0;
    sigaction (SIGUSR1, &usr_action, NULL);

    /* Create the child process. */
    if ((pid = fork()) == 0) child_function (); /* Does not return. */
    else if (pid == -1) { /* A problem with fork - exit. */ }

    /* Wait for the child to send a signal */
    sigemptyset (&block_mask);
    sigaddset (&block_mask, SIGUSR1);
    sigprocmask (SIG_BLOCK, &block_mask, &old_mask);
    while (!usr_interrupt)
        sigsuspend (&old_mask);
    sigprocmask (SIG_UNBLOCK, &block_mask, NULL);
    ...
}
```