Operační systémy a sítě

Petr Štěpán, K13133
KN-E-229
stepan@labe.felk.cvut.cz

Téma 3. Procesy a vlákna

Pojem „Výpočetní proces“

- Výpočetní proces (*job*, *task*) – aktivita provádění programu
- Proces je identifikovatelný a podléhá plánování
  - Proces je vlastníkem dynamicky přidělovaných zdrojů potřebných pro běh procesu
    - čas procesoru, úseky paměti ve FAP, soubory na disku, periferie, ...
- Stav procesu lze v každém okamžiku jeho existence jednoznačně určit
  - přidělené zdroje; události, na něž proces čeká; prioritu; …
- Komponenty vytvářející proces
  - obsahy registrů procesoru (čítač instrukcí, ...)
  - zásobník
  - datová sekce
  - program, který proces řídí
- V systémech podporujících vlákna ➔ bývá proces chápán jako kontejner či hostitel svých vláken

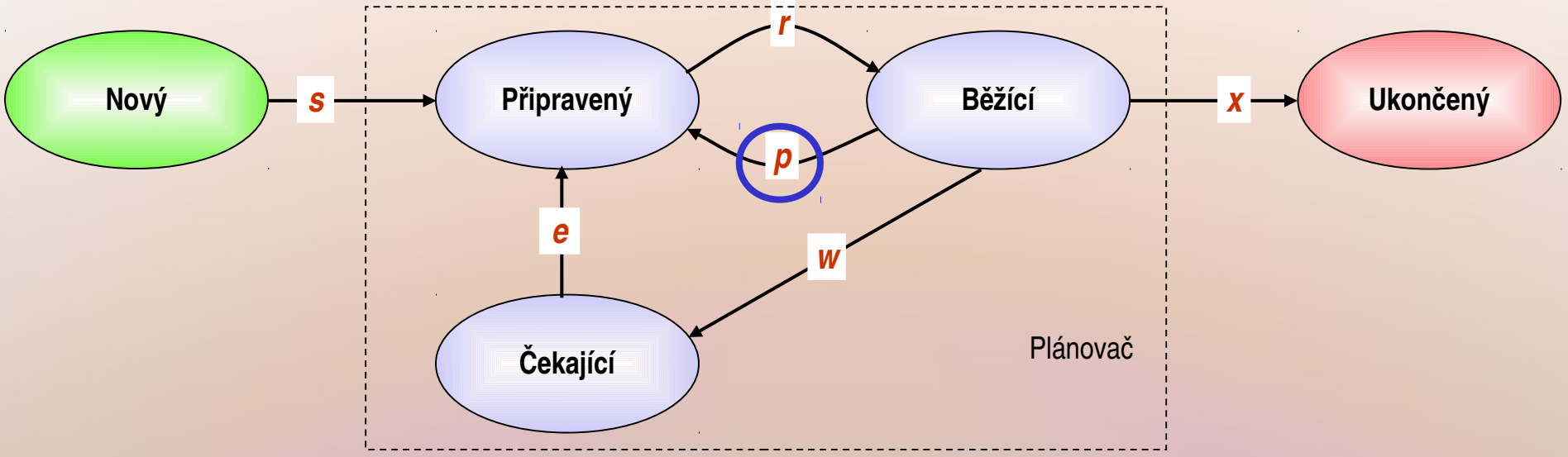
Požadavky na OS při práci s procesy

- Prokládat vykonávání jednotlivých procesů s cílem maximálního využití procesoru
- Minimalizovat dobu odpovědi procesu prokládáním běhů procesů
- Přidělovat procesům požadované systémové prostředky na základě vhodné politiky
  - priority, vzájemné vyloučení za současné zábrany uváznutí, …
- Umožňovat procesům vytváření a spouštění dalších procesů
- Podporovat vzájemnou komunikaci mezi procesy
- Poskytovat aplikačním procesům funkčně bohaté, bezpečné a konzistentní rozhraní k systémovým službám
  - včetně uniformní prezentace systémových prostředků (např. souborů)

Stavy procesů

- Proces se za dobu své existence prochází více stavy a nachází se vždy v jednom z následujících stavů:
  - Nový (*new*) – proces je právě vytvářen
  - Připravený (*ready*) – proces čeká na přidělení procesoru
  - Běžící (*running*) – program řídící tento proces je právě vykonáván, tj. interpretován některým procesorem
  - Čekající (*waiting*, *blocked*) – proces čeká na jistou událost
  - Ukončený (*terminated*) – proces ukončil svoji činnost, avšak stále ještě vlastní některé systémové prostředky

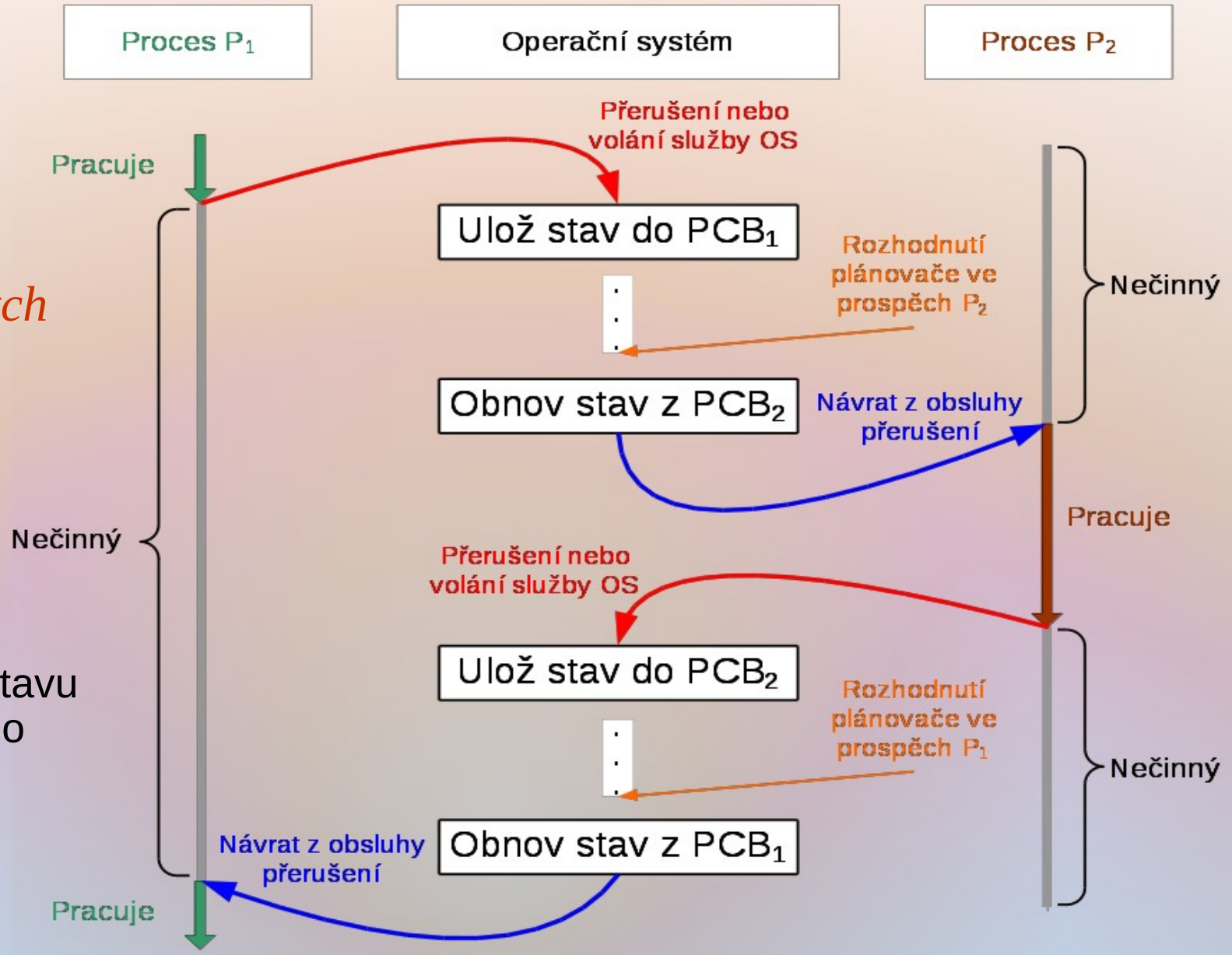
Základní (pětistavový) diagram procesů



Přechod	Význam
s	Proces vzniká – ***s****tart*
r	Procesu je přidělen procesor (může pracovat) – ***r****un*
w	Proces žádá o službu, na jejíž dokončení musí čekat – ***w****ait*
e	Vznikla událost, která způsobila, že se proces „dočkal“ – ***e****vent*
x	Skončila existence procesu (na žádost procesu nebo „násilně“) – *e****x****it*
p	Procesu byl odňat procesor, přestože je proces dále schopen běhu, tzv. **preempce** (např. vyčerpání časového kvanta) – ***p****reemption.*

Přepínání mezi procesy

Context switch



Nečinný =
proces je ve stavu
„**čekající**“ nebo
„**připravený**“

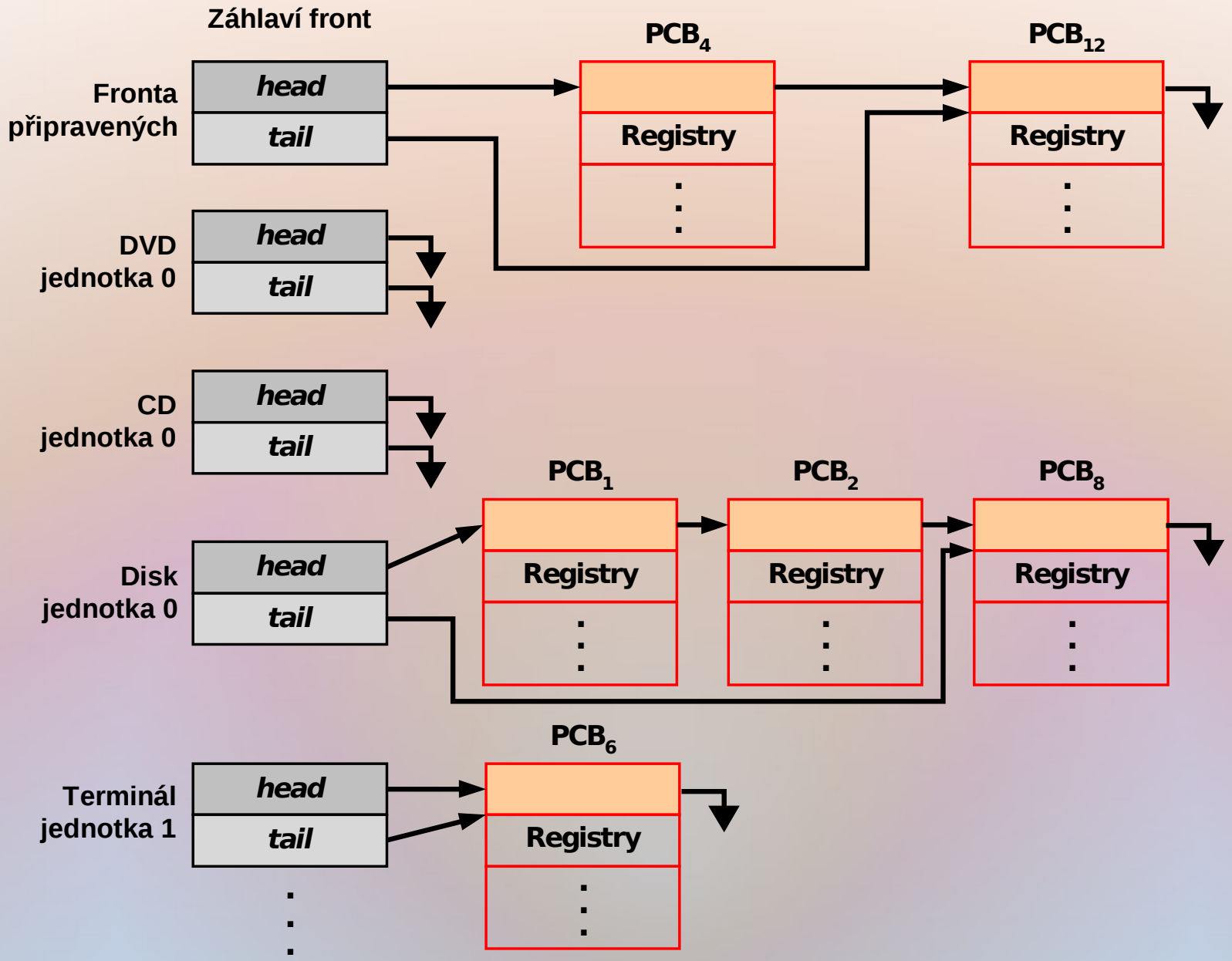
Popis procesů

- Deskriptor procesu – *Process Control Block* (PCB)
  - Datová struktura obsahující
    - Identifikátor procesu (pid)
    - Globální stav (*process state*)
    - Místo pro uložení čítače instrukcí (PC) a stavového slova (PSW), popř. odkaz na místo, kde jsou tyto údaje uloženy
    - Místo pro uložení registrů procesoru
    - Informace potřebné pro plánování procesoru/ů
      - Priorita, využití CPU, … ➔
    - Informace potřebné pro správu paměti
      - Odkazy do paměti (*memory pointers*), popř. registry MMU
    - Účtovací informace (*accounting*)
    - Stavové informace o V/V (*I/O status*)
    - Kontextová data (*context data*)
      - Otevřené soubory
      - Proměnné prostředí (*environment variables*)
    - …
    - Spojka pro řazení PCB do front a seznamů

Fronty a seznamy procesů pro plánování

- Fronta připravených procesů
  - množina procesů připravených k běhu čekajících pouze na přidělení procesoru
- Fronta na dokončení I/O operace
  - samostatná fronta pro každé zařízení
- Seznam odložených procesů
  - množina procesů čekajících na přidělení místa v hlavní paměti, FAP
- Fronty související se synchronizací procesů ➔
  - množiny procesů čekajících synchronizační události
- Fronta na přidělení prostoru v paměti
  - množina procesů potřebujících zvětšit svůj adresní prostor
- …
- Procesy mezi různými frontami migrují

Fronty a seznamy procesů – příklad



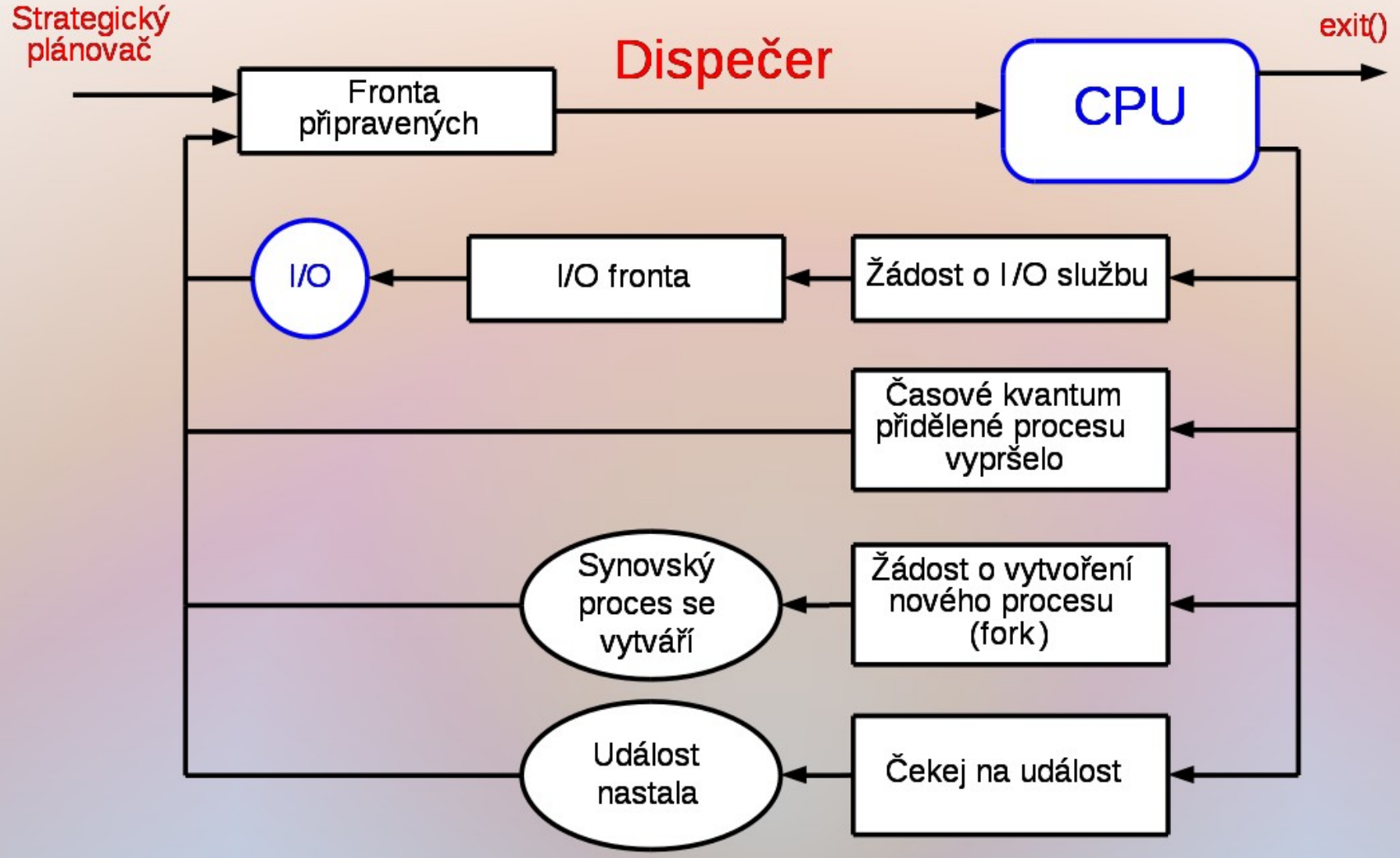
Plánovače v OS

- Dlouhodobý plánovač (strategický plánovač, *job scheduler*)
  - Vybírá, který požadavek na výpočet lze zařadit mezi procesy, a definuje tak stupeň multiprogramování
  - Je volán zřídka (jednotky až desítky sekund), nemusí být rychlý
    - V interaktivních systémech (typu Windows) se prakticky nepoužívá a degeneruje na přímé předání práce krátkodobému plánovači
- Krátkodobý plánovač (operační plánovač, dispečer, *dispatcher*):
  - Základní správa procesoru/ů
  - Vybírá proces, který poběží na uvolněném procesoru přiděluje procesu procesor (CPU)
  - vyvoláván velmi často, musí být extrémně rychlý
- Střednědobý plánovač (taktický plánovač)
  - Úzce spolupracuje se správou hlavní paměti
  - Taktika využívání omezené kapacity FAP při multitaskingu
  - Vybírá, který proces je možno zařadit mezi odložené procesy ➔
    - uvolní tím prostor zabíraný procesem ve FAP
  - Vybírá, kterému odloženému procesu lze znovu přidělit prostor ve FAP

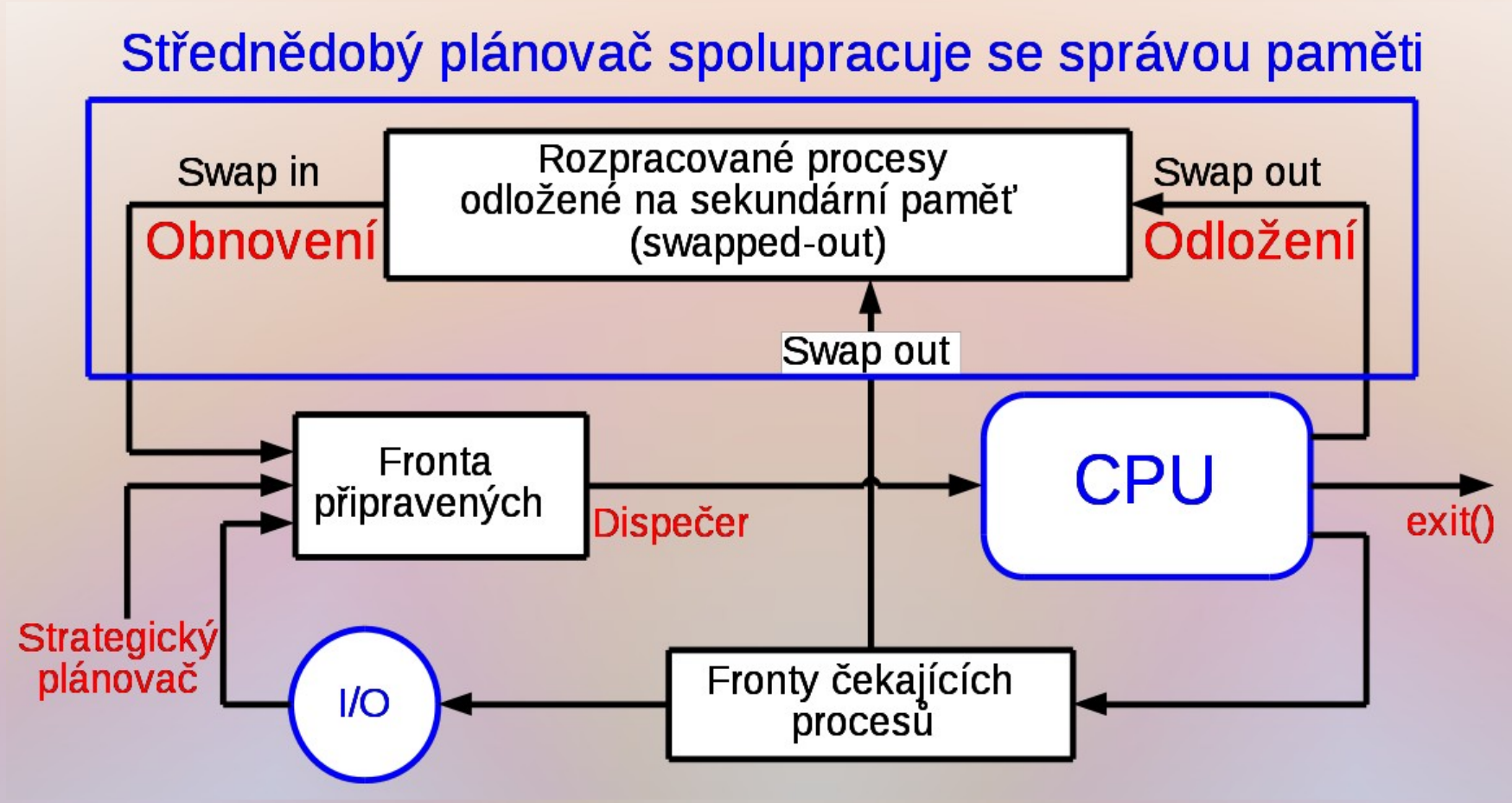
Cíle plánování a kriteria kvality plánů

- Využití CPU
  - *maximalizace* kontinuální užitečné činnosti CPU
- Propustnost
  - *maximalizace* počtu procesů, které dokončí svůj běh za jednotku času
- Doba obrátky
  - *minimalizace* doby potřebné pro provedení konkrétního procesu
- Doba čekání
  - *minimalizace* doby, po kterou proces čekal ve frontě připravených
- Doba odpovědi
  - minimalizace doby, která uplyne od okamžiku zadání požadavku na spuštění procesu do jeho první reakce, např. prvního výpisu na terminál,
    - Nikoli doba do poskytnutí úplného výstupu jakožto výsledku běhu celého procesu
    - Užívají se různé "triky" pro odvrácení pozornosti operátora v interaktivních OS (bannery Windows)

Strategický plánovač a dispečer



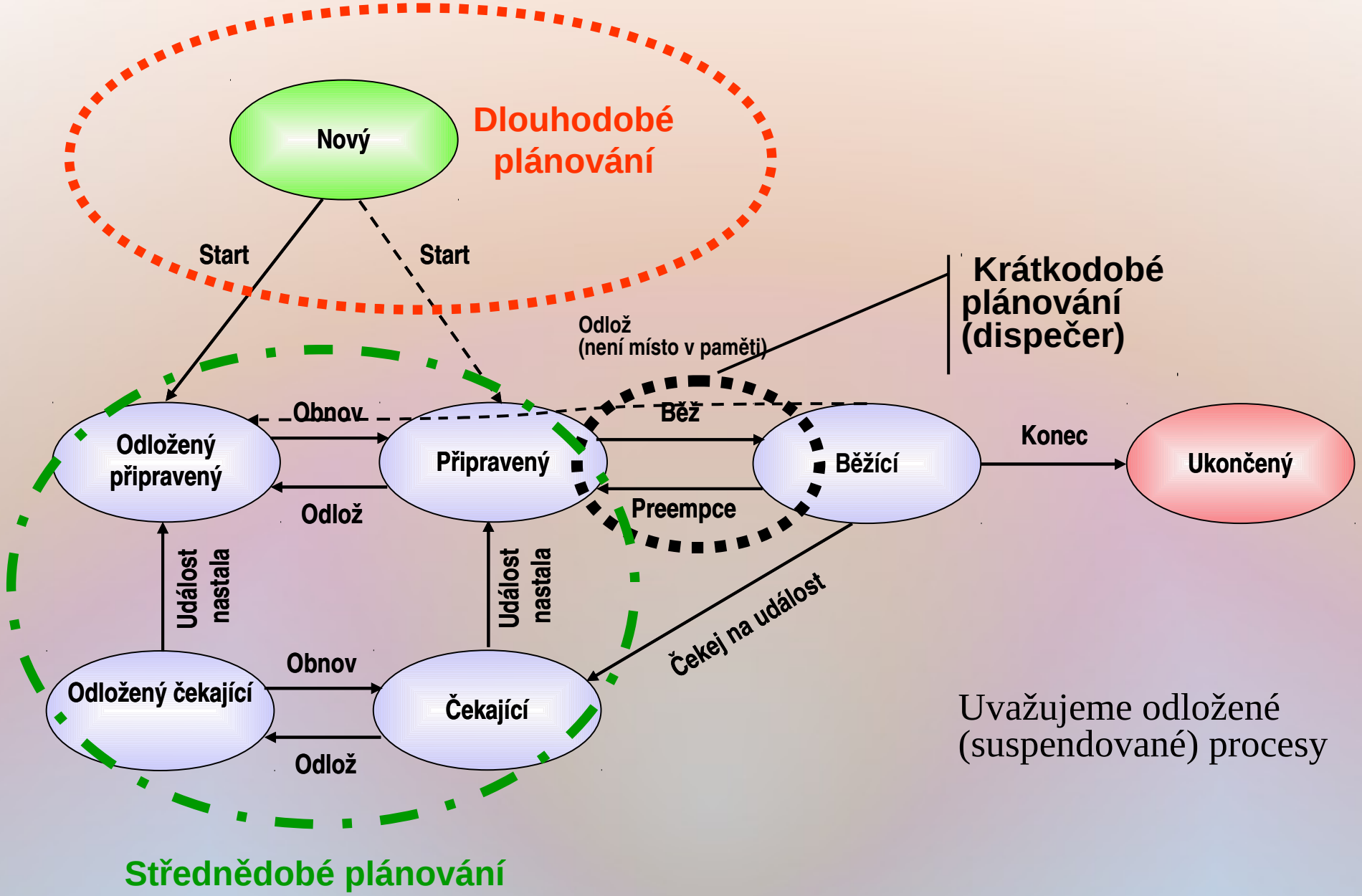
Odkládání a střednědobé plánování



Odkládání - *swapping*

- Běžící proces musí mít alespoň pro aktuální části svého LAP přidělen prostor ve FAP
  - jinak by nemohl pracovat
- I když se používá princip virtuální paměti
  - příliš mnoho procesů ve FAP (alespoň částečně) snižuje výkonnost systému
    - jednotlivé procesy obdrží malý prostor ve FAP a aktuální úsek LAP ve FAP se jim vyměňuje příliš často (problém „výprasku“ ➔)
- OS musí paměťový prostor některých procesů odložit
  - takové procesy nemohou běžet
  - odložení – *swap-out*, okopírování na disk
  - obnova – *swap-in*, zavedení do FAP
- Přibývají tak další dva stavy procesů
  - odložený čekající – čeká na nějakou událost a, i kdyby byl v paměti, stejně by nebyl schopen běhu
  - odložený připravený – nechybí mu nic kromě místa v paměti

Sedmistavový diagram procesů



Uvažujeme odložené (suspendované) procesy

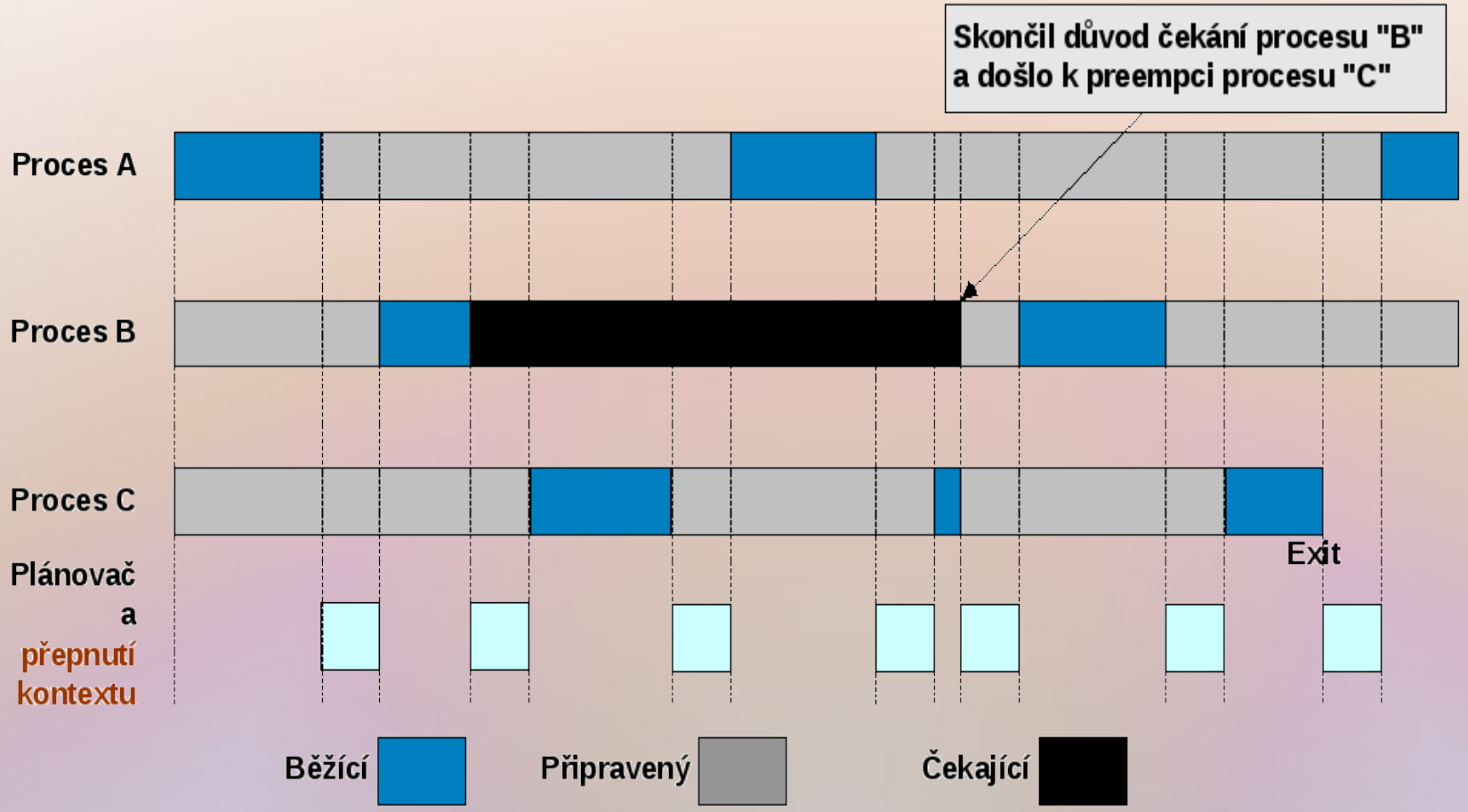
Plánovač CPU (dispečer) a typy plánování

- Dispečer pracuje s procesy, které jsou v hlavní paměti a jsou schopné běhu, tj. připravené (*ready*)
- Existují 2 typy plánování
  - nepreemptivní plánování (plánování bez předbíhání, někdy také kooperativní plánování), kdy procesu schopnému dalšího běhu procesor není „násilně“ odnímán
    - Používá se zpravidla jen v „uzavřených systémech“, kde jsou předem známy všechny procesy a jejich vlastnosti. Navíc musí být naprogramovány tak, aby samy uvolňovaly procesor ve prospěch procesů ostatních
  - preemptivní plánování (plánování s předbíháním), kdy procesu schopnému dalšího běhu může být procesor odňat i „bez jeho souhlasu“ (tedy kdykoliv)
- Plánovač rozhoduje (vstupuje do hry) v okamžiku, kdy některý proces:
  1. přechází ze stavu běžící do stavu čekající nebo končí
  2. přechází ze stavu čekající do stavu připravený
  3. přechází ze stavu běžící do stavu připravený
- První dva případy se vyskytují v obou typech plánování
- Poslední je charakteristický pro plánování preemptivní

Přepnutí kontextu procesu

- Přechod od procesu *A* k *B* zahrnuje tzv. přepnutí kontextu
  - Přepnutí od jednoho procesu k jinému nastává výhradně v důsledku nějakého přerušení (či výjimky)
  - Proces *A* → operační systém/přepnutí kontextu → proces *B*
  - Nejprve OS uchová (zapamatuje v PCB_A) stav původně běžícího procesu *A*
  - Provedou se potřebné akce v jádru OS a dojde k rozhodnutí ve prospěch procesu *B*
  - Obnoví se stav „nově rozbíhaného“ procesu *B* (z PCB_B)
- Přepnutí kontextu představuje režijní ztrátu (zátěž)
  - během přepínání systém nedělá nic efektivního
  - časově nejnáročnější je správa paměti dotčených procesů
- Doba přepnutí závisí na hardwarové podpoře v procesoru
  - minimální hardwarová podpora při přerušení:
    - uchování čítače instrukcí
    - naplnění čítače instrukcí hodnotou z vektoru přerušení
  - lepší podpora:
    - ukládání a obnova více registrů procesoru jedinou instrukcí

Stavy procesů v čase – preemptivní případ



Doby běhu plánovače by měly být co nejkratší

- režijní ztráty systému

Vznik procesu

- Rodičovský proces vytváří procesy-potomky
  - pomocí služby OS. Potomci mohou vystupovat v roli rodičů a vytvářet další potomky, …
  - Vzniká tak strom procesů
- Sdílení zdrojů mezi rodiči a potomky:
  - rodič a potomek mohou sdílet všechny zdroje původně vlastněné rodičem (obvyklá situace v POSIXu)
  - potomek může sdílet s rodičem podmnožinu zdrojů rodičem k tomu účelu vyčleněnou
  - potomek a rodič jsou plně samostatné procesy, nesdílí žádný zdroj
- Souběh mezi rodiči a potomky:
  - Možnost 1: rodič čeká na dokončení potomka
  - Možnost 2: rodič a potomek mohou běžet souběžně
- V POSIXu je každý proces potomkem jiného procesu
  - Výjimka: proces *init* vytvořen při spuštění systému
    - Spustí řadu *sh* skriptů (rc), ty inicializují celý systém a vytvoří démony (procesy běžící na pozadí bez úplného kontextu) ~ *service* ve Win32
    - *init* spustí pro terminály proces *getty*, který čeká na uživatele => *login* => uživatelův *shell*

Příklad vytvoření procesu (POSIX)

- Rodič vytváří nový proces – potomka voláním služby **`fork()`**
- Vznikne identická kopie rodičovského procesu
  - potomek je úplným duplikátem rodiče
  - každý z obou procesů se při vytváření procesu dozvídá, zda je rodičem nebo potomkem
  - do adresního prostoru potomka se automaticky zavádí program shodný rodičem
- Potomek použije volání služby **`exec`** pro náhradu programu ve svém adresním prostoru jiným programem
  - Pozn.: Program řídí vykonávání procesu ...

Ukončení procesu

- Proces provede poslední příkaz programu a žádá OS o ukončení voláním služby **`exit(status)`**
  - Stavová data procesu-potomka (status) se mohou předat procesu-rodiči, který čeká v provádění služby **`wait()`**
  - Zdroje končícího procesu jádro uvolní
- Proces může skončit také:
  - přílišným nárokem na paměť (tolik paměti není a nebude nikdy k dispozici)
  - narušením ochrany paměti („zběhnutí“ programu)
  - pokusem o provedení nedovolené (privilegované) operace (zakázaný přístup k systémovému prostředku, r/o soubor)
  - aritmetickou chybou (dělení nulou, arcsin(2), ...) či neopravitelnou chybou V/V
  - žádostí rodičovského procesu (v POSIXu signál)
  - zánikem rodiče
    - Může tak docházet ke kaskádnímu ukončování procesů
    - V POSIXu lze proces „odpojit“ od rodiče – démon

  a v mnoha dalších chybových situacích

Program, proces a vlákno

- Program (z pohledu jádra OS):
  - soubor přesně definovaného formátu obsahující
    - instrukce,
    - data
    - údaje potřebné k zavedení do paměti a inicializaci procesu
- Proces:
  - systémový objekt – entita realizující výpočet podle programu charakterizovaná svým paměťovým prostorem a kontextem
  - prostor ve FAP se přiděluje procesům (nikoli programům!)
  - patří mu i případný obraz jeho adresního prostoru (nebo jeho části) na vnější paměti
  - může vlastnit soubory, I/O zařízení a komunikační kanály vedené k jiným procesům
- Vlákno:
  - objekt vytvářený programem v rámci procesu

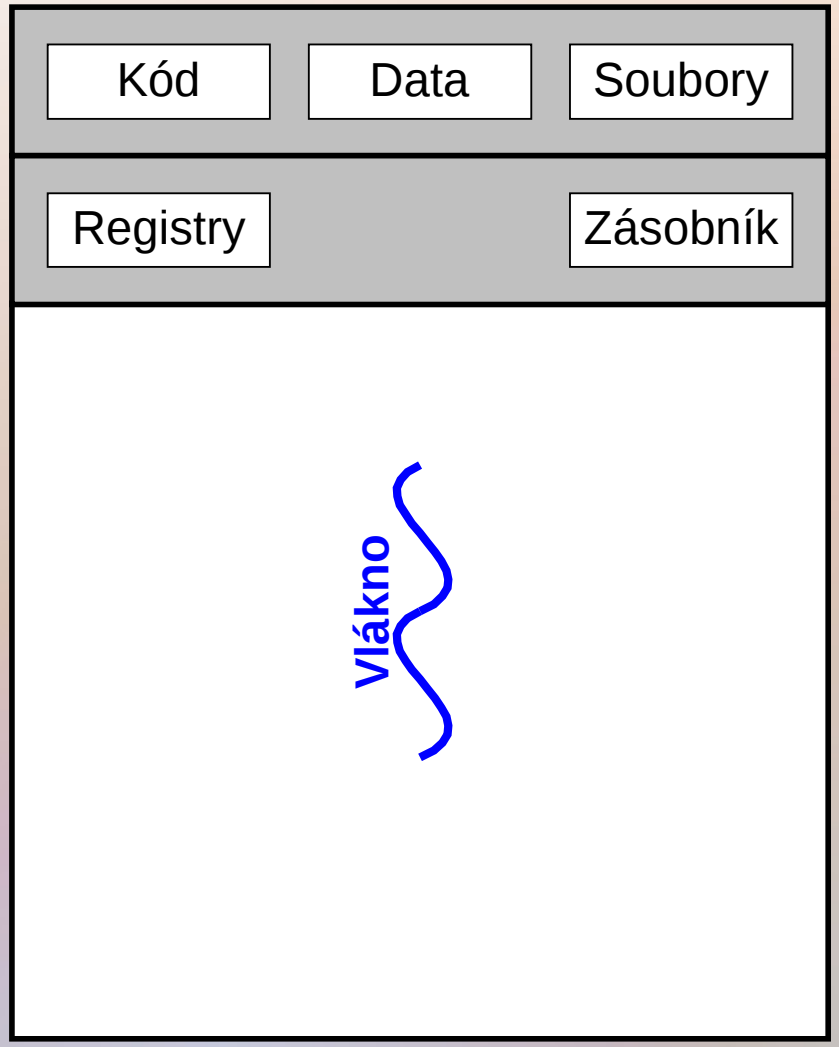
Vztah procesu a vlákna

- Vlákno (*thread*)
  - Objekt vytvářený v rámci procesu a viditelný uvnitř procesu
  - Tradiční proces je proces tvořený jediným vláknem
  - Vlákna podléhají plánování a přiděluje se jim strojový čas i procesory
  - Vlákno se nachází ve stavech: běží, připravené, čekající, …
    - Podobně jako při přidělování času procesům
  - Když vlákno neběží, je kontext vlákna uložený v TCB (*Thread Control Block*):
    - analogie PCB
    - prováděcí zásobník vlákna, obraz PC, obraz registrů, ...
  - Vlákno může přistupovat k LAP a k ostatním zdrojům svého procesu a ty jsou sdíleny všemi vlákny tohoto procesu
    - Změnu obsahu některé buňky LAP procesu vidí všechna ostatní vlákna téhož procesu
    - Soubor otevřený jedním vláknem je viditelný pro všechna ostatní vlákna téhož procesu
    - Vlákna patřící k jednomu procesu sdílí proměnné a systémové zdroje přidělené tomuto procesu

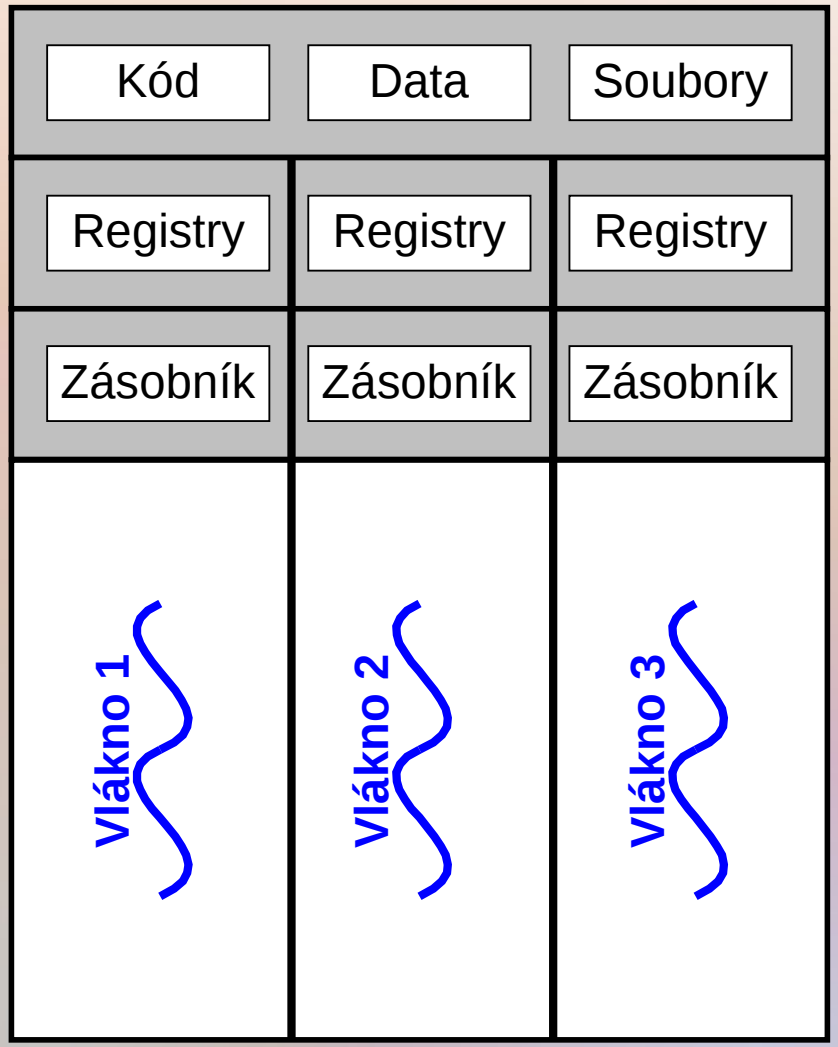
Proces a jeho vlákna

- Jednovláknové (tradiční) procesy
  - proces: jednotka plánování činnosti a jednotka vlastnící přidělené prostředky
  - každé vlákno je současně procesem s vlastním adresovým prostorem a s vlastními prostředky
  - tradiční UNIXy
    - moderní implementace UNIXů jsou již většinou vláknově orientované
- Procesy a vlákna (Windows, Solaris, …)
  - proces: jednotka vlastnící prostředky
  - vlákno: jednotka plánování činnosti systému
  - v rámci jednoho procesu lze vytvořit více vláken
  - proces definuje adresový prostor a dynamicky vlastní prostředky

Procesy a vlákna

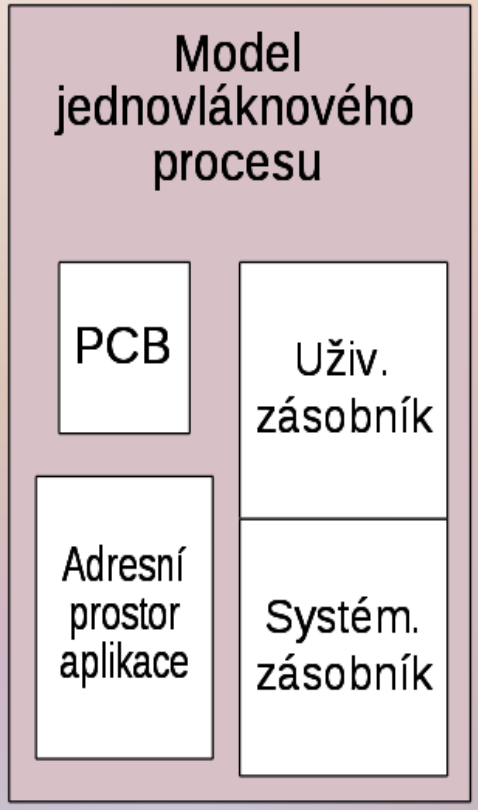


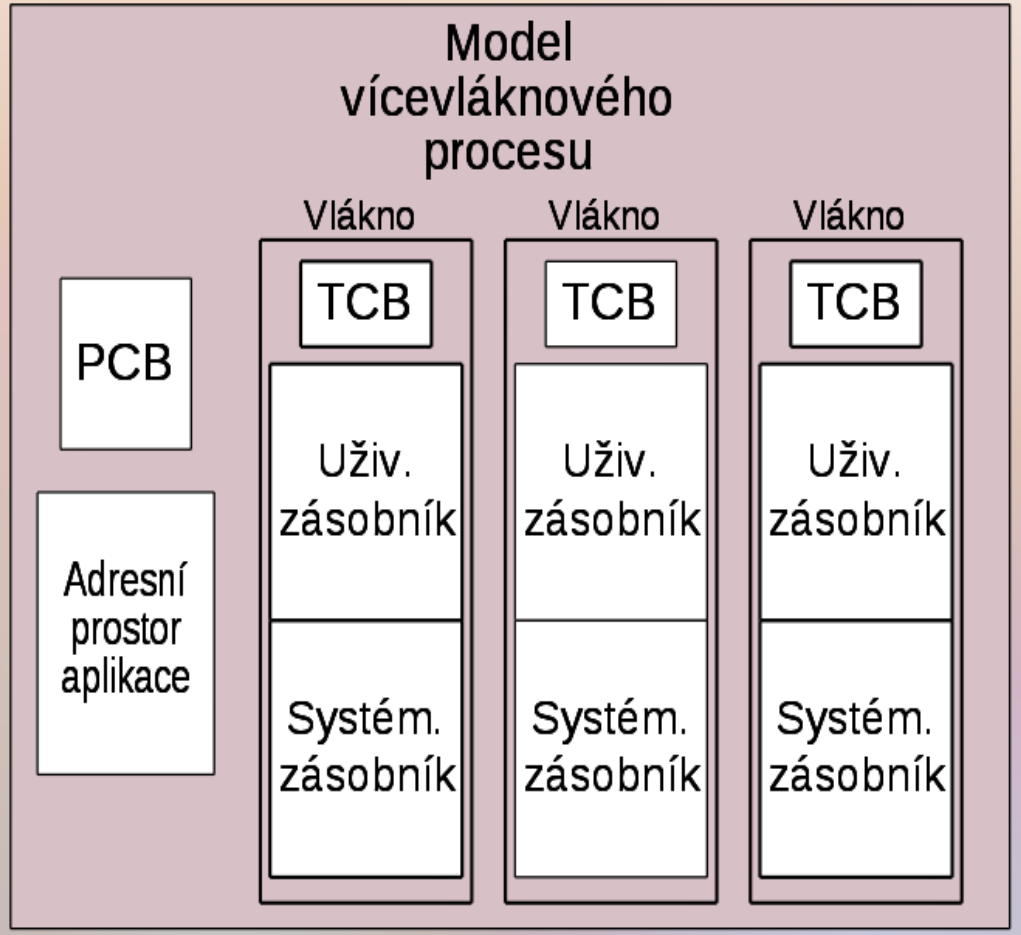
Jednovláknový proces



Vícevláknový proces

Procesy a vlákna – řídicí struktury





Procesy, vlákna a jejich komponenty

Co patří procesu a co vláknu?

kód programu:	počítač
lokální a pracovní data:	vlákno
globální data:	proces
alokované systémové zdroje:	proces
zásobník:	vlákno
data pro správu paměti:	proces
čítač instrukcí:	vlákno
registry procesoru:	vlákno
plánovací stav:	vlákno
uživatelská práva a identifikace:	proces

Účel vláken

- Přednosti
  - Vlákno se vytvoří i ukončí rychleji než proces
  - Přepínání mezi vlákny je rychlejší než mezi procesy
  - Dosáhne se lepší strukturalizace programu
- Příklady
  - Souborový server v LAN
    - Musí vyřizovat během krátké doby několik požadavků na soubory
    - Pro vyřízení každého požadavku se zřídí samostatné vlákno
  - Symetrický multiprocesor
    - na různých procesorech mohou běžet vlákna souběžně
  - Menu vypisované souběžně se zpracováním prováděným jiným vláknem
  - Překreslování obrazovky souběžně se zpracováním dat
  - Paralelizace algoritmu v multiprocesoru
- Lepší a přehlednější strukturalizace programu

Problém konzistence sdílených dat

- Mějme aplikaci, která sestává z více nezávislých částí, z nichž každá je implementována jako samostatné vlákno
- Vlákna nemusí běžet v sekvenci
  - Když vlákno čeká na nějakou událost, může běžet jiné vlákno téhož procesu, aniž by se přepínalo mezi procesy
- Vlastnosti takové implementace
  - Vlákna jednoho procesu sdílí paměť, a tudíž mohou mezi sebou komunikovat přímo, aniž by k tomu potřebovaly služby jádra
  - Vlákna jedné aplikace se proto musí mezi sebou synchronizovat, aby se zachovala konzistence zpracovávaných dat
    - Aby si vzájemně nepřepisovala data

Problém konzistence – příklad

- Scénář:
  - Proces vytvořil vlákna T_1 a T_2
  - T_1 počítá $C = A + B$,
  - T_2 používá hodnotu X: $A = A - X$; $B = B + X$;
  - T_1 a T_2 pracují souběžně a jejich běhy se tak mohou prokládat
- Úmysl programátora
  - Nechť $A = 2$, $B = 3$, $X = 10$
  - T_2 udělá $A = A - X$ a $B = B + X$ [$A = -8$, $B = 13$]
  - T_1 spočítá $C = A + B$, hodnota C nezávisí na X [$C = 5$]
- Možná realita
  - T_2 udělá $A = A - X$ a pak je mu odňat procesor [$A = -8$]
  - T_1 spočítá $C = A + B = A - X + B$ [$C = -5$]
  - T_2 udělá $B = B + X$ a to už hodnotu C neovlivní [$B = 13$]
  - Máme dva různé výsledky v proměnné C
- Poznámka
  - Kdyby nedošlo k preempci vlákna T_2, žádný problém by nenastal! Preempce tak může být nebezpečná!

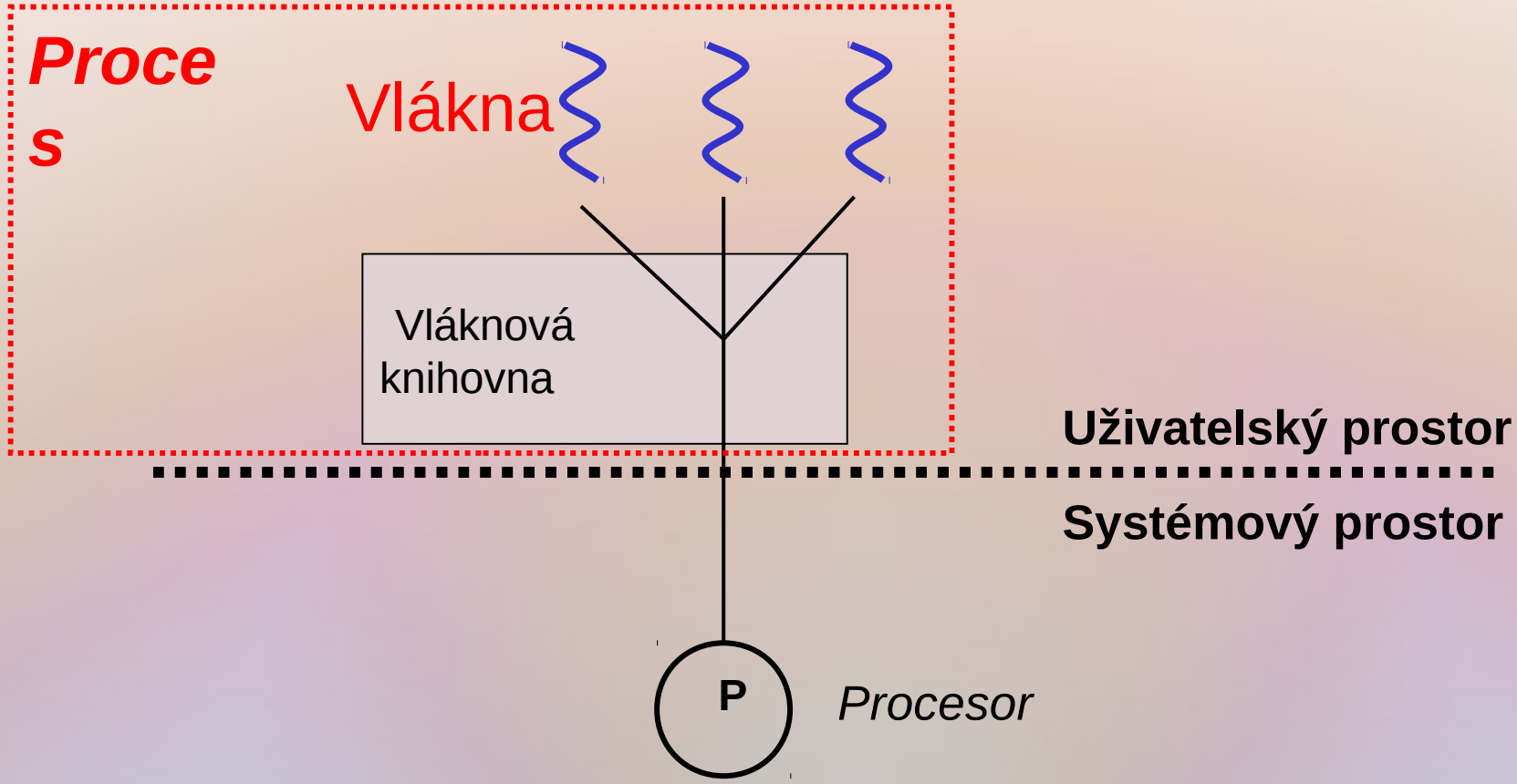
Stavy a odkládání vláken

- Vlákna podléhají plánování a mají své stavy podobně jako procesy
- Základní stavy
  - běžící
  - připravené
  - čekající
- Všechna vlákna jednoho procesu sdílejí společný adresní prostor
  - => vlákna se samostatně neodkládají, odkládá je jen proces
- Ukončení (havárie) procesu ukončuje všechna vlákna existující v tomto procesu

Vlákna na uživatelské úrovni, ULT (1)

- *User-Level Threads* (**ULT**)
- Vlastnosti
  - Správu vláken provádí tzv. vláknová knihovna (*thread library*) na úrovni aplikačního procesu, JOS o jejich existenci neví
  - Přepojování mezi vlákny nepožaduje provádění funkcí jádra
  - Nepřepíná se ani kontext procesu ani režim procesoru
  - Aplikace má možnost zvolit si nejvhodnější strategii a algoritmus pro plánování vláken
  - Lze používat i v OS, který neobsahuje žádnou podporu vláken v jádře, stačí speciální knihovna (model 1 : *M*)
- Vláknová knihovna obsahuje funkce pro
  - vytváření a rušení vláken
  - předávání zpráv a dat mezi vlákny
  - plánování běhů vláken
  - uchovávání a obnova kontextů vláken

Vlákna na uživatelské úrovni, ULT (2)



Vlákna na uživatelské úrovni, ULT (3)

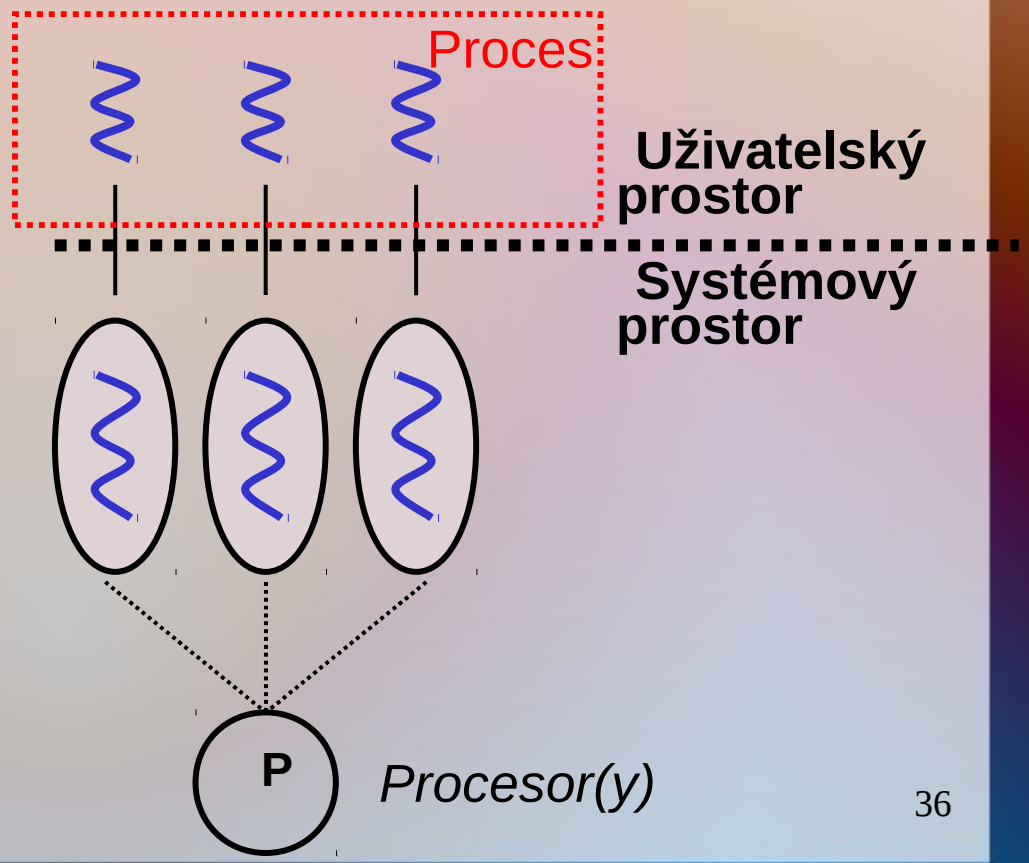
- Problém stavu vláken: Co když se proces nebo vlákno zablokuje?
  - Nechť proces *A* má dvě vlákna T_1 a T_2, přičemž T_1 právě běží
  - Mohou nastat následující situace:
    - T_1 požádá JOS o I/O operaci nebo jinou službu:
      - Jádro zablokuje proces *A* jako celek.
      - Celý proces čeká, přestože by mohlo běžet vlákno T_2.
    - Proces *A* vyčerpá časové kvantum:
      - JOS přeřadí proces *A* mezi připravené
      - TCB_1 však indikuje, že T_1 je stále ve stavu „běžící“ (ve skutečnosti *neběží*!)
    - T_1 potřebuje akci realizovanou vláknem T_2:
      - Vlákno T_1 se zablokuje. Vlákno T_2 se rozběhne
      - Proces *A* zůstane ve stavu „běžící“ (což je správně)
- Závěr
  - V ULT nelze stav vláken věrohodně sledovat

Výhody a nevýhody uživatelských vláken

- Výhody:
  - Rychlé přepínání mezi vlákny (bez účasti JOS)
  - Rychlá tvorba a zánik vláken
  - Uživatelský proces má plnou kontrolu nad vlákny
    - např. může za běhu zadávat priority či volit plánovací algoritmus
- Nevýhody:
  - Volání systémové služby jedním vláknem zablokuje všechna vlákna procesu
  - Dodatečná práce programátora pro řízení vláken
    - Lze však ponechat knihovnou definovaný implicitní algoritmus plánování vláken
  - Jádro o vláknech „nic neví“, a tudíž přiděluje procesor pouze procesům, Dvě vlákna téhož procesu nemohou běžet současně, i když je k dispozici více procesorů

Vlákna na úrovni jádra, KLT

- *Kernel-Level Threads* (**KLT**)
- Veškerá správa vláken je realizována OS
- Každé vlákno v uživatelském prostoru je zobrazeno na vlákno v jádře (model 1:1)
- JOS vytváří, plánuje a ruší vlákna
- Jádro může plánovat vlákna na různé CPU
  - Skutečný multiprocessing
- Příklady
  - Windows NT/2000/XP
  - Linuxy
  - 4.4BSD UNIXy
  - Tru64 UNIX
  - ... všechny moderní OS



Výhody a nevýhody KLT

- Výhody:
  - Volání systému neblokuje ostatní vlákna téhož procesu
  - **Jeden proces může využít více procesorů**
    - skutečný paralelismus uvnitř jednoho procesu – každé vlákno běží na jiném procesoru
  - Tvorba, rušení a přepínání mezi vlákny je levnější než přepínání mezi procesy
    - netřeba dělat cokoliv s přidělenou pamětí
  - I moduly jádra mohou mít vícevláknový charakter
- Nevýhody:
  - Systémová správa je režijně nákladnější než u čistě uživatelských vláken
  - Klasické plánování není "spravedlivé": Dostává-li vlákno své časové kvantum(➔), pak procesy s více vlákny dostávají více času

Knihovna Pthreads

- Pthreads je POSIX-ový standard definující API pro vytváření a synchronizaci vláken a specifikace chování těchto vláken
- Knihovna Pthreads poskytuje unifikované API:
  - Nepodporuje-li JOS vlákna, knihovna Pthreads bude pracovat čistě s ULT
  - Implementuje-li příslušné jádro KLT, pak toho knihovna Pthreads bude využívat
  - Pthreads je tedy systémově závislá knihovna
- **Příklad:** Samostatné vlákno, které počítá součet prvních *n* celých čísel

Příklad volání API Pthreads

Příklad: Samostatné vlákno, které počítá součet prvních *n* celých čísel; *n* se zadává jako parametr programu na příkazové řádce

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

int sum;                                                    /* sdílená data */
void *runner(void *param);                                  /* rutina realizující vlákno */

main(int argc, char *argv[]) {
      pthread_t tid;                                        /* identifikátor vlákna*/
      pthread_attr_t attr;                                  /* atributy vlákna */
      pthread_attr_init(&attr);                             /* inicializuj implicitní atributy */
      pthread_create(&tid, &attr, runner, argv[1]);         /* vytvoř vlákno */
      pthread_join(tid,NULL);                               /* čekej až vlákno skončí */
      printf("sum = %d\n",sum);
}

void *runner(void *param) {
    int upper = atoi(param);  int i;  sum = 0;
    if (upper > 0) {
        for (i = 1; i <= upper; i++)
            sum += i;
     }
    pthread_exit(0);
}
```

Vlákna ve Windows 2000/XP/7

- Aplikace ve Windows běží jako proces tvořený jedním nebo více vlákny
- Windows implementují mapování 1:1
- Někteří autoři dokonce tvrdí, že *Proces se nemůže vykonávat, neboť je jen kontejnerem pro vlákna a jen ta jsou schopná běhu*
- Každé vlákno má
  - svůj identifikátor vlákna
  - sadu registrů (obsah je ukládán v TCM)
  - samostatný uživatelský a systémový zásobník
  - privátní datovou oblast

Vlákna v Linuxu a Javě

- Vlákna Linux:
  - Linux nazývá vlákna *tasks*
  - Lze použít knihovnu pthreads
  - Vytváření vláken je realizováno službou OS **`clone()`**
  - **`clone()`** umožňuje vláknu (task) sdílet adresní prostor s rodičem
    - **`fork()`** vytvoří zcela samostatný proces s kopií prostoru rodičovského procesu
    - **`clone()`** vytvoří vlákno, které dostane odkaz (pointer) na adresní prostor rodiče
- Vlákna v Javě:
  - Java má třídu „Thread“ a instancí je vlákno
    - Samozřejmě lze z třídy Thread odvodit podtřídu a některé metody přepsat
  - Vlákna jsou spravována přímo JVM
    - JVM spolu se základními Java třídami vlastně vytváří virtuální stroj obsahující jak „hardware“ (vlastní JVM) tak i na něm běžící OS podporující vlákna
    - Většinou jsou vlákna JVM mapována 1:1 na vlákna OS

Vytvoření vlákna v JavaAPI

```
class CounterThread extends Thread
{
    public void run() {
        for(int i = 0; i < 10; i++) {
            System.out.println(i);
        }
    }
}

Thread counterThread = new
CounterThread();

counterThread.start();
```

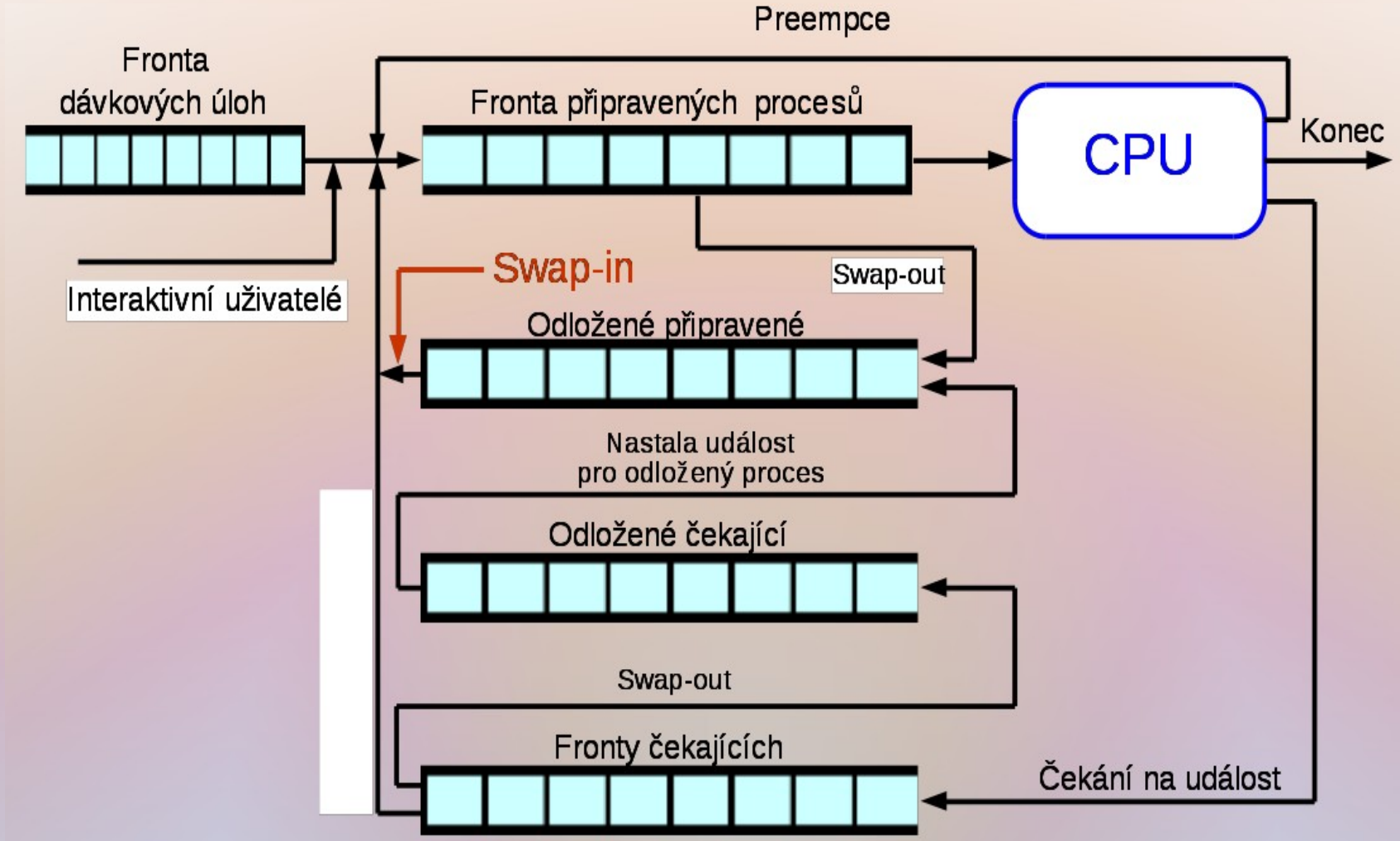
```
class Counter implements Runnable {
    public void run() {
        for(int i = 0; i < 10; i++) {
            System.out.println(i);
        }
    }
}

Runnable counter = new Counter();

Thread counterThread = new
Thread(counter);

counterThread.start();
```

Frontový model plánování CPU



Kriteria krátkodobého plánování

- Uživatelsky orientovaná
  - čas odezvy
    - doba od vzniku požadavku do reakce na něj
  - doba obrátky
    - doba od vzniku procesu do jeho dokončení
  - konečná lhůta (*deadline*)
    - požadavek dodržení stanoveného času dokončení
  - předvídatelnost
    - Úloha by měla být dokončena za zhruba stejnou dobu bez ohledu na celkovou zátěž systému
    - Je-li systém vytížen, prodloužení odezvy by mělo být rovnoměrně rozděleno mezi procesy
- Systémově orientovaná
  - průchodnost
    - počet procesů dokončených za jednotku času
  - využití procesoru
    - relativní čas procesoru věnovaný aplikačním procesům
  - spravedlivost
    - každý proces by měl dostat svůj čas (ne „hladovění“ či „stárnutí“)
  - vyvažování zátěže systémových prostředků
    - systémové prostředky (periferie, hlavní paměť) by měly být zatěžovány v čase rovnoměrně

Plánovač procesů

- Aktivace plánovače (dispečeru)
  - Obslužná rutina přerušení na svém konci vyhlásí tzv. významnou událost v systému
    - např. dokončení přenosu dat, vyčerpání časového kvanta
  - Významná událost aktivuje plánovač, který rozhodne, co dále
  - Plánovač může přepnout kontext ⇒ přechod od jednoho procesu k jinému je VŽDY důsledkem nějaké VÝJIMKY (nebo přerušení)
- Fronta připravených procesů
  - Plánovač rozhoduje, který proces aktivovat
  - Proces v čele fronty dostává procesor a může tak způsobit preempci. Ta může nastat kdykoliv (i bez aktivity či „vědomí“ běžícího procesu)
  - Fronty nemusí být prosté (FIFO), může se v nich předbíhat dle priorit
  - Dynamické určení priority procesu
    - Klíč k dosažení cílů plánovače (spravedlivost, propustnost, ...)
    - Odhadují se měnící se charakteristiky procesu
    - Zpravidla založeno na měření chování procesu

Plánovač procesů

- Aktivace plánovače (dispečeru)
  - Obslužná rutina přerušení na svém konci vyhlásí tzv. významnou událost v systému
    - např. dokončení přenosu dat, vyčerpání časového kvanta
  - Významná událost aktivuje plánovač, který rozhodne, co dále
  - Plánovač může přepnout kontext ⇒ přechod od jednoho procesu k jinému je VŽDY důsledkem nějaké VÝJIMKY (nebo přerušení)
- Fronta připravených procesů
  - Plánovač rozhoduje, který proces aktivovat
  - Proces v čele fronty dostává procesor a může tak způsobit preempci. Ta může nastat kdykoliv (i bez aktivity či „vědomí“ běžícího procesu)
  - Fronty nemusí být prosté (FIFO), může se v nich předbíhat dle priorit
  - Dynamické určení priority procesu
    - Klíč k dosažení cílů plánovače (spravedlivost, propustnost, ...)
    - Odhadují se měnící se charakteristiky procesu
    - Zpravidla založeno na měření chování procesu

Plánovací algoritmy

- Ukážeme plánování:
  - FCFS (*First-Come First-Served*)
  - SPN (SJF) (*Shortest Process Next*)
  - SRT (*Shortest Remaining Time*)
  - cyklické (*Round-Robin*)
  - zpětnovazební (*Feedback*)
- Příklad
  - používaný v tomto textu pro ilustraci algoritmů

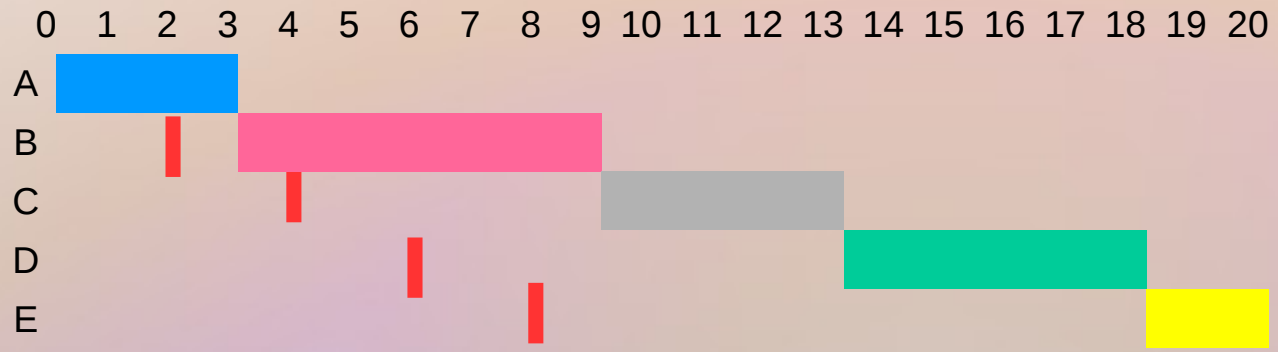
Proces	Čas příchodu	Potřebný čas (délka CPU dávky)
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

- Chování se ilustruje tzv. Ganttovými diagramy

Plánování FCFS

- FCFS = *First Come First Served* – prostá fronta FIFO
- Nejjednodušší nepreemptivní plánování
- Nově příchozí proces se zařadí na konec fronty
- Průměrné čekání může být velmi dlouhé
- Příklad:

$$^{w}T_{Avg} = \frac{0+1+5+7+10}{5} = 4{,}6$$



- Průměrné čekání bychom mohli zredukovat:
  - Např. v čase 9 je procesor volný a máme na výběr procesy C, D a E

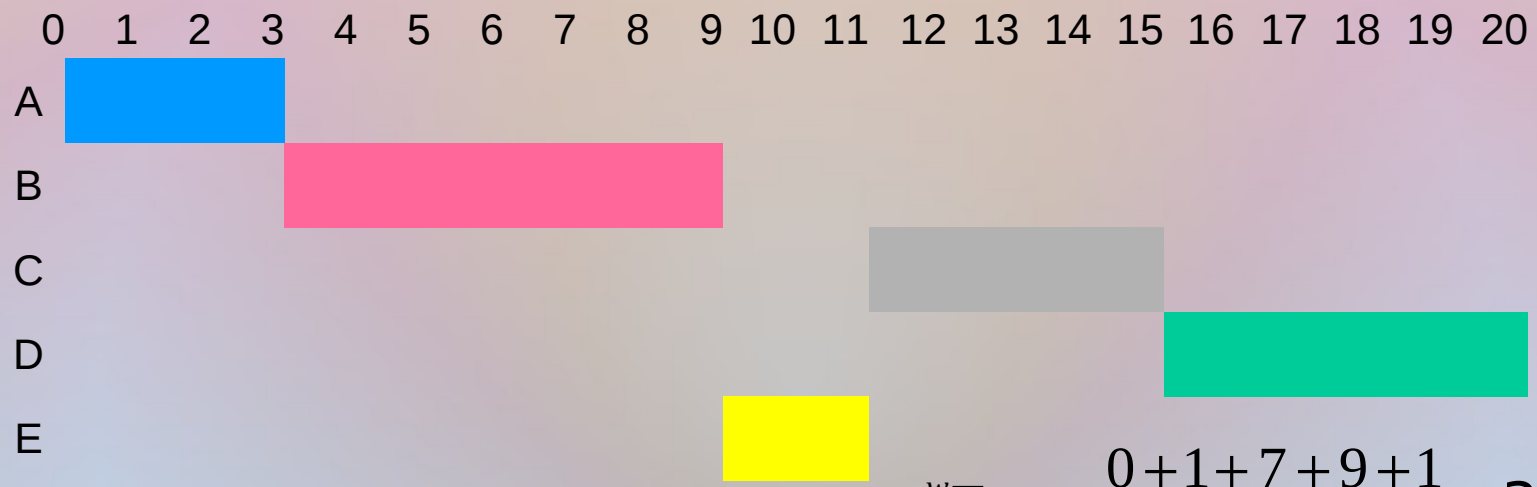
$$^{w}T_{Avg} = \frac{0+1+7+9+1}{5} = 3{,}6$$

Vlastnosti FCFS

- FCFS je primitivní nepreemptivní plánovací postup
- Průměrná doba čekání $\bar{w}T_{\mathrm{Avg}}$ silně závisí na pořadí přicházejících dávek
- Krátké procesy, které se připravily po dlouhém procesu, vytváří tzv. konvojový efekt
  - Všechny procesy čekají, až skončí dlouhý proces
- Pro krátkodobé plánování se FCFS samostatně fakticky nepoužívá.
  - Používá se pouze jako složka složitějších plánovacích postupů

Plánování SPN

- SPN = *Shortest Process Next* (nejkratší proces jako příští); též nazýváno SJF = *Shortest Job First*
  - Opět nepreemptivní
  - Vybírá se připravený proces s nejkratší příští dávkou CPU
  - Krátké procesy předbíhají delší, nebezpečí stárnutí dlouhých procesů
  - Je-li kritériem kvality plánování průměrná doba čekání, je SPN optimálním algoritmem, což se dá exaktně dokázat
- Příklad:



$$^{w}T_{\text{Avg}} = \frac{0+1+7+9+1}{5} = 3{,}6$$

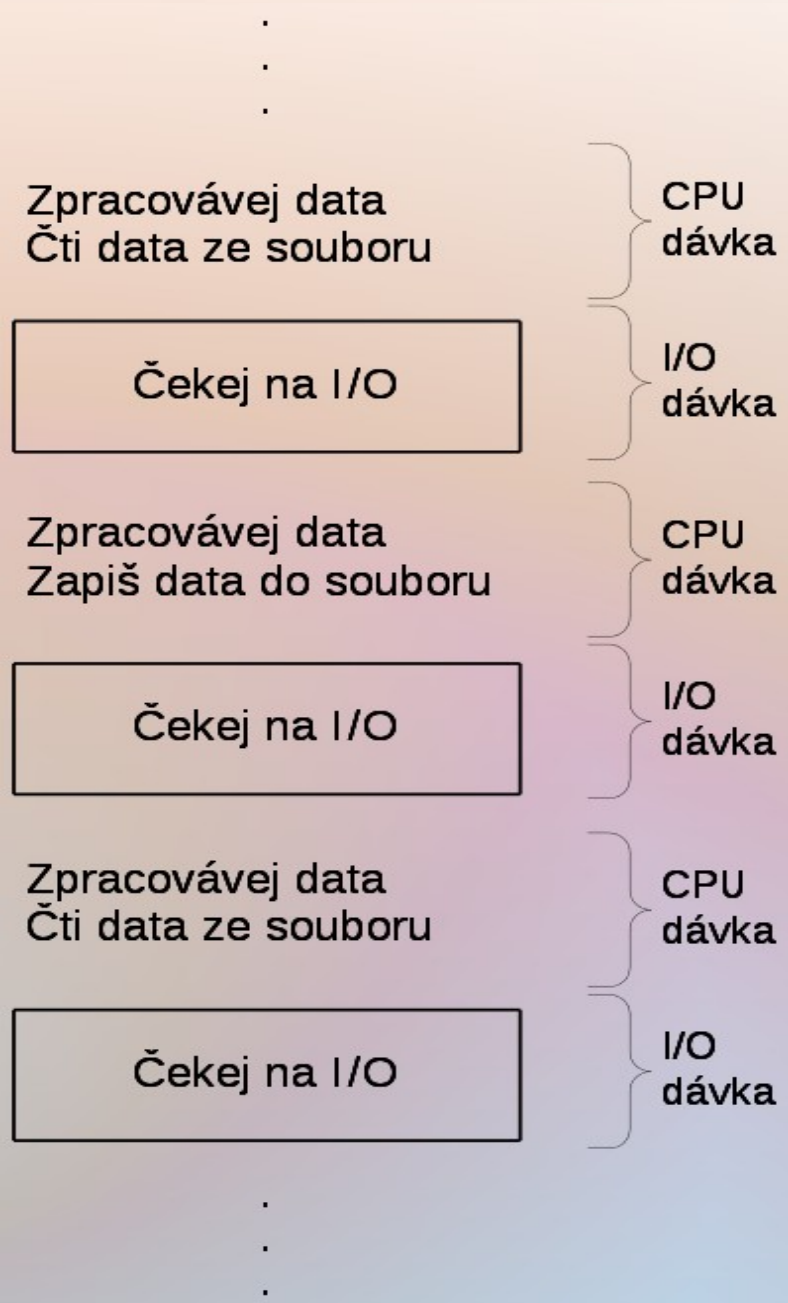
Plánování SRT

- SRT = *Shortest Remaining Time* (nejkratší zbývající čas)
  - Preemptivní varianta SPN
  - CPU dostane proces, který potřebuje nejméně času do svého ukončení
  - Jestliže existuje proces, kterému zbývá k jeho dokončení čas kratší, než je čas zbývající do skončení procesu běžícího, dojde k preempci
    - Může existovat procesů se stejným zbývajícím časem, a pak je nutno použít jakési „arbitrážní pravidlo“
- Příklad:

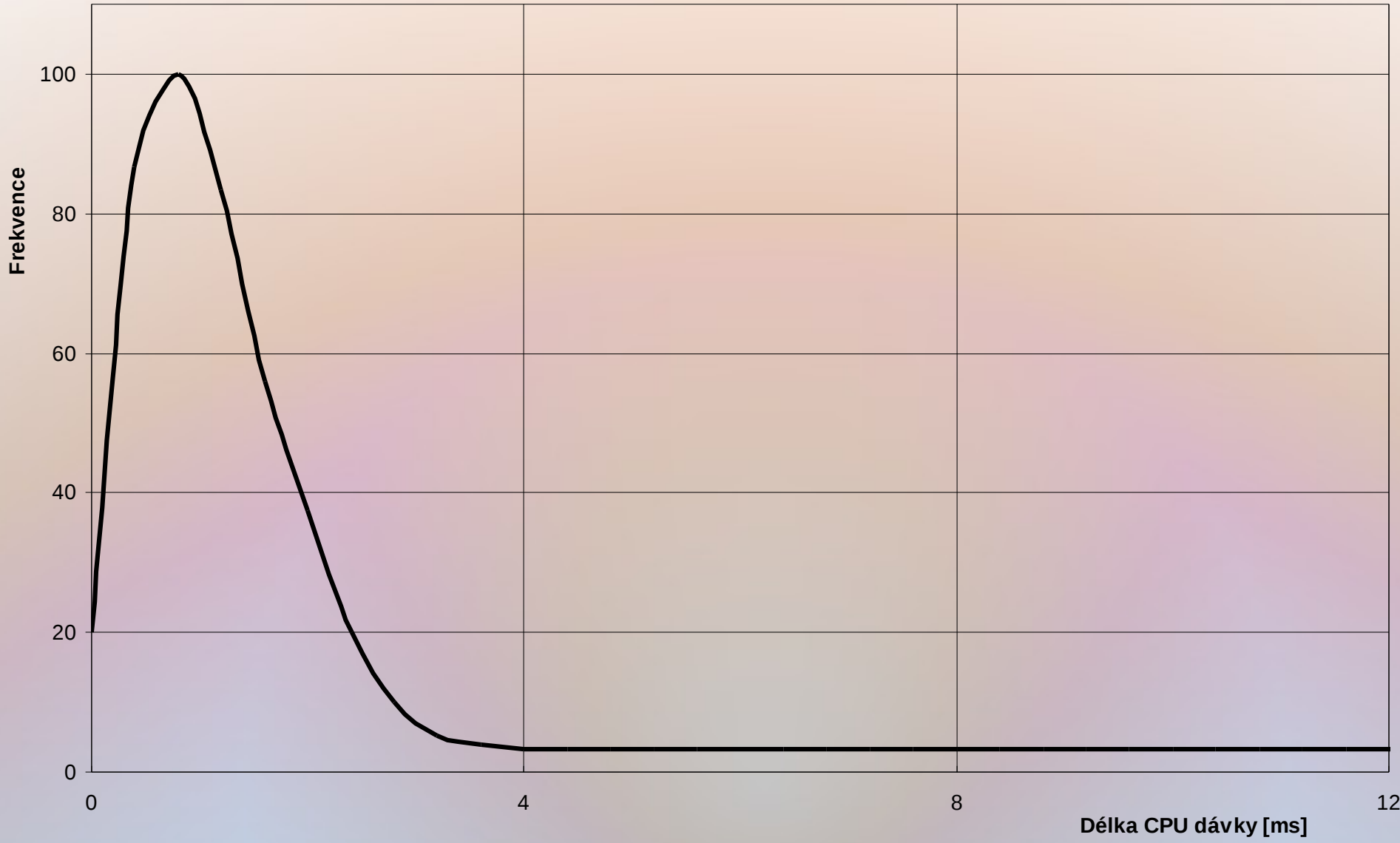
$$^{w}T_{\text{Avg}} = \frac{0+1+0+9+0}{5} = 2{,}0$$

Motivace plánování CPU

- Maximálního využití CPU se dosáhne uplatněním multiprogramování
- Jak ?
- Běh procesu = cykly alternujících dávek
  - [: CPU dávka, I/O dávka :]
- CPU dávka se může v čase překrývat s I/O dávkami dalších procesů

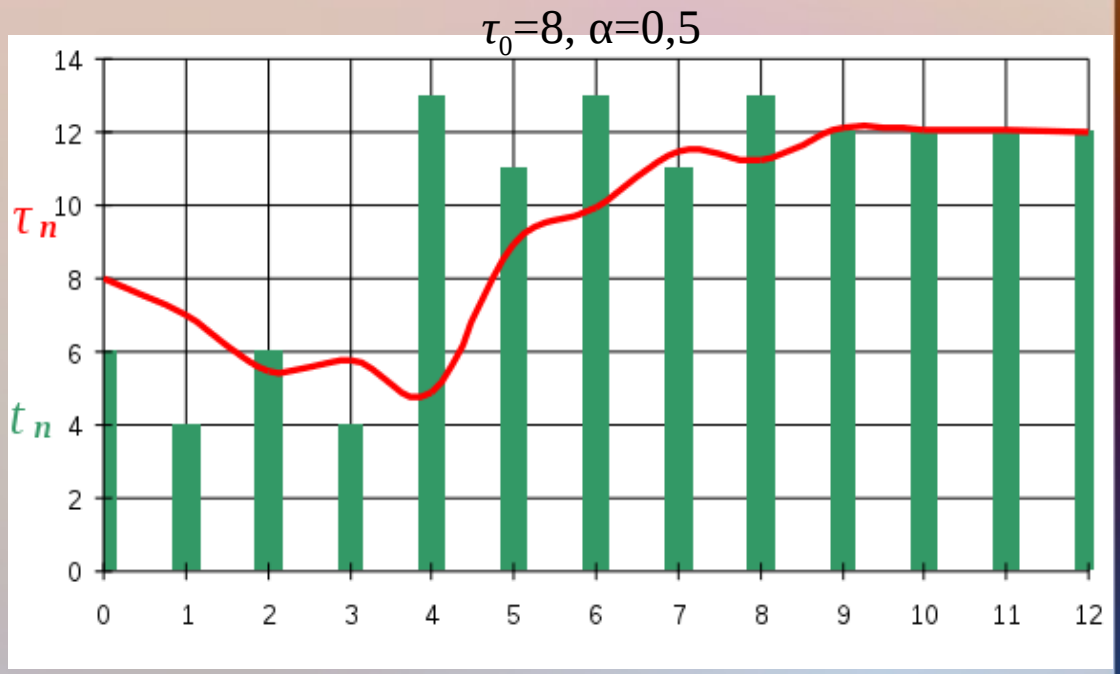


Typický histogram délek CPU dávek



Odhad délky příští dávky CPU procesu

- Délka příští dávky CPU skutečného procesu je známa jen ve velmi speciálních případech
  - Délka dávky se odhaduje na základě nedávné historie procesu
  - Nejčastěji se používá tzv. exponenciální průměrování
- Exponenciální průměrování
  - t_n … skutečná délka *n*-té dávky CPU
  - τ_{n+1} … odhad délky příští dávky CPU
  - α, $0 \leq \alpha \leq 1$ … parametr vlivu historie
  - $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$
  - Příklad:
    - $\alpha = 0{,}5$
    - $\tau_{n+1} = 0{,}5t_n + 0{,}5\tau_n = 0{,}5(t_n + \tau_n)$
    - τ_0 se volí jako průměrná délka CPU dávky v systému nebo se odvodí z typu nejčastějších programů



To je dnes vše.

Otázky?