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*) – spuštěný program
- Proces je identifikovatelný – jednoznačné číslo
  - PID – Process Identification Digit
- Stav procesu lze v každém okamžiku jeho existence jednoznačně určit
  - přidělené zdroje; události, na něž proces čeká; prioritu; …
- Co tvoří proces:
  - Obsahy registrů procesoru (čítač instrukcí, ukazatel zásobníku, PSW, uživatelské registry, FPU registry)
  - Otevřené soubory
  - Použitá paměť:
    - Zásobník - .stack
    - Data - .data
    - Program - .text
- V systémech podporujících vlákna ➔ bývá proces chápán jako obal či hostitel svých vláken

Požadavky na OS při práci s procesy

- Umožňovat procesům vytváření a spouštění dalších procesů
- Prokládat - „paralelizovat“ vykonávání jednotlivých procesů s cílem maximálního využití procesoru/ů
- Minimalizovat dobu odpovědi procesu prokládáním běhů procesů
- Přidělovat procesům požadované systémové prostředky
  - Soubory, V/V zařízení, synchronizační prostředky
- Podporovat vzájemnou komunikaci mezi procesy
- Poskytovat aplikačním procesům funkčně bohaté, bezpečné a konzistentní rozhraní k systémovým službám
  - Systémová volání – minulá přednáška

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

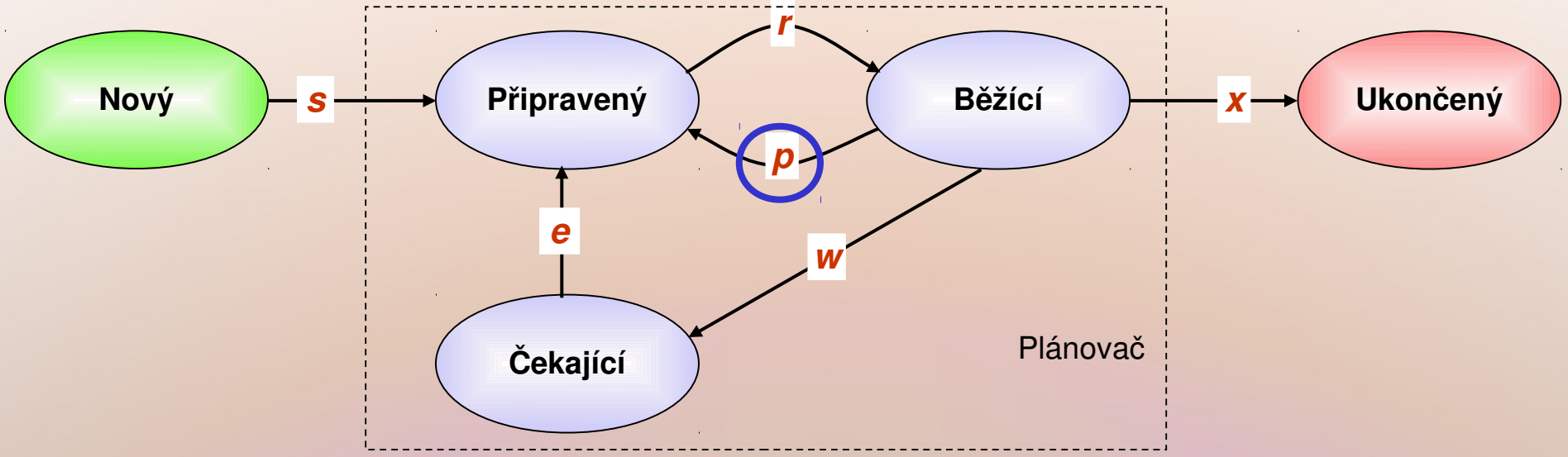
- Proces provede poslední příkaz programu a žádá OS o ukončení voláním služby **`exit(status)`**
  - Stavová data procesu-potomka (status) se mohou předat procesu-rodiči, který čeká v provádění služby **`wait()`**
  - Zdroje končícího procesu jádro uvolní
- Proces může skončit také:
  - přílišným nárokem 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

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, ještě není připraven na běh, ale již jsou připraveny některé části
  - Připravený (*ready*) – proces čeká na přidělení procesoru
  - Běžící (*running*) – instrukce procesu je právě vykonávány procesorem, tj. interpretovány některým procesorem
  - Čekající (*waiting*, *blocked*) – proces čeká na událost
  - Ukončený (*terminated*) – proces ukončil svoji činnost, avšak stále ještě vlastní některé systémové prostředky

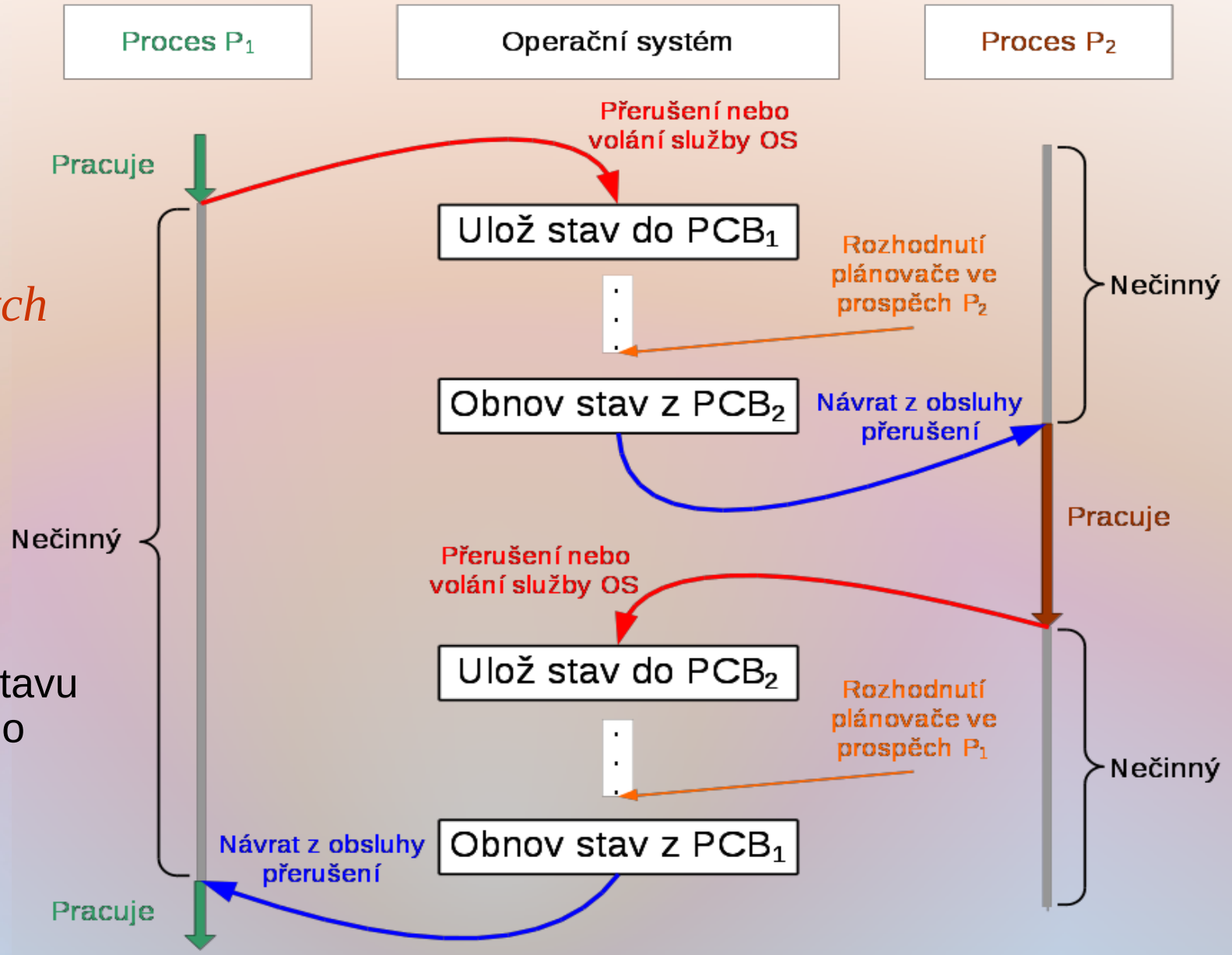
Základní (pětistavový) diagram procesů



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

Context switch

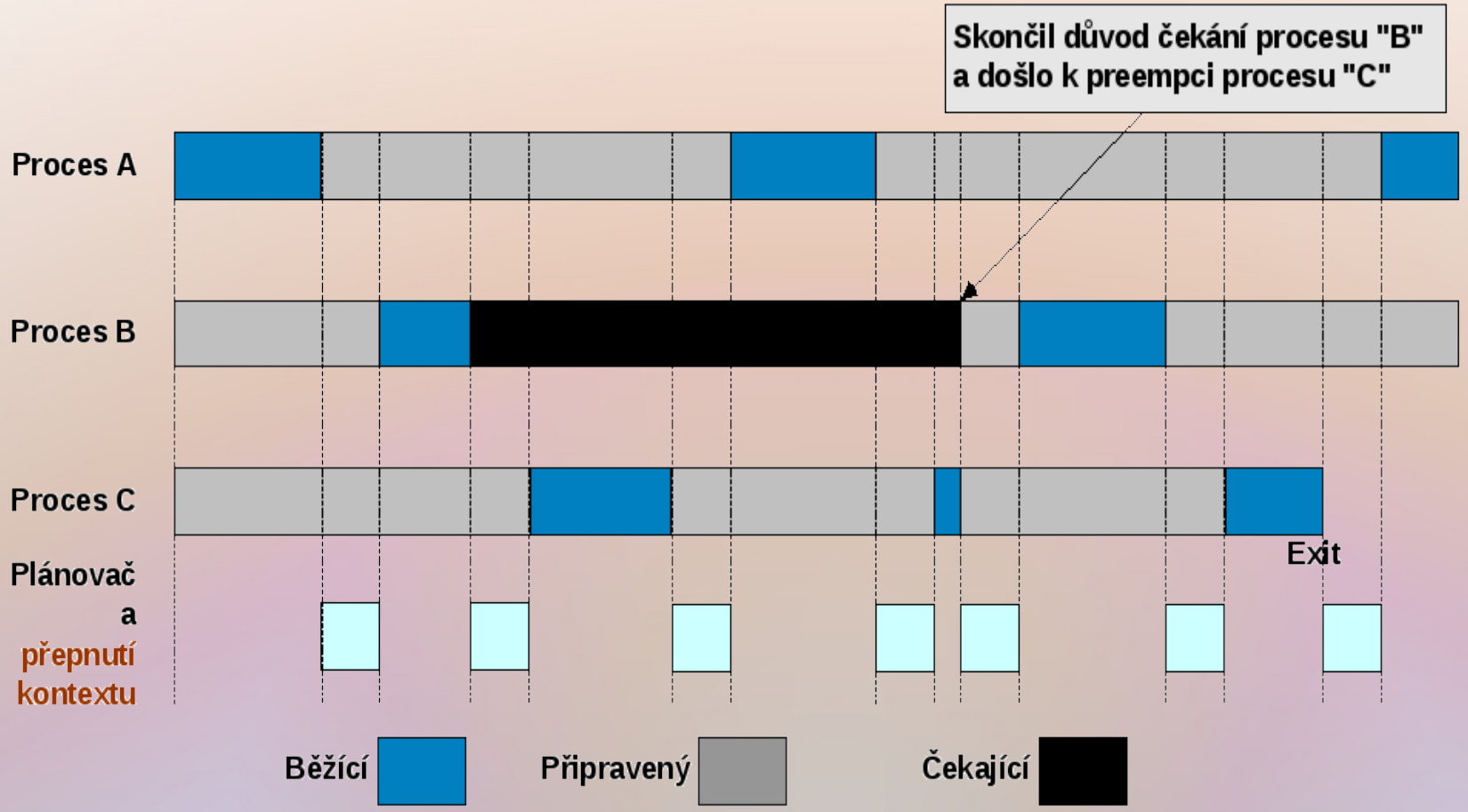


Nečinný =
proces je ve stavu
„**čekající**“ nebo
„**připravený**“

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

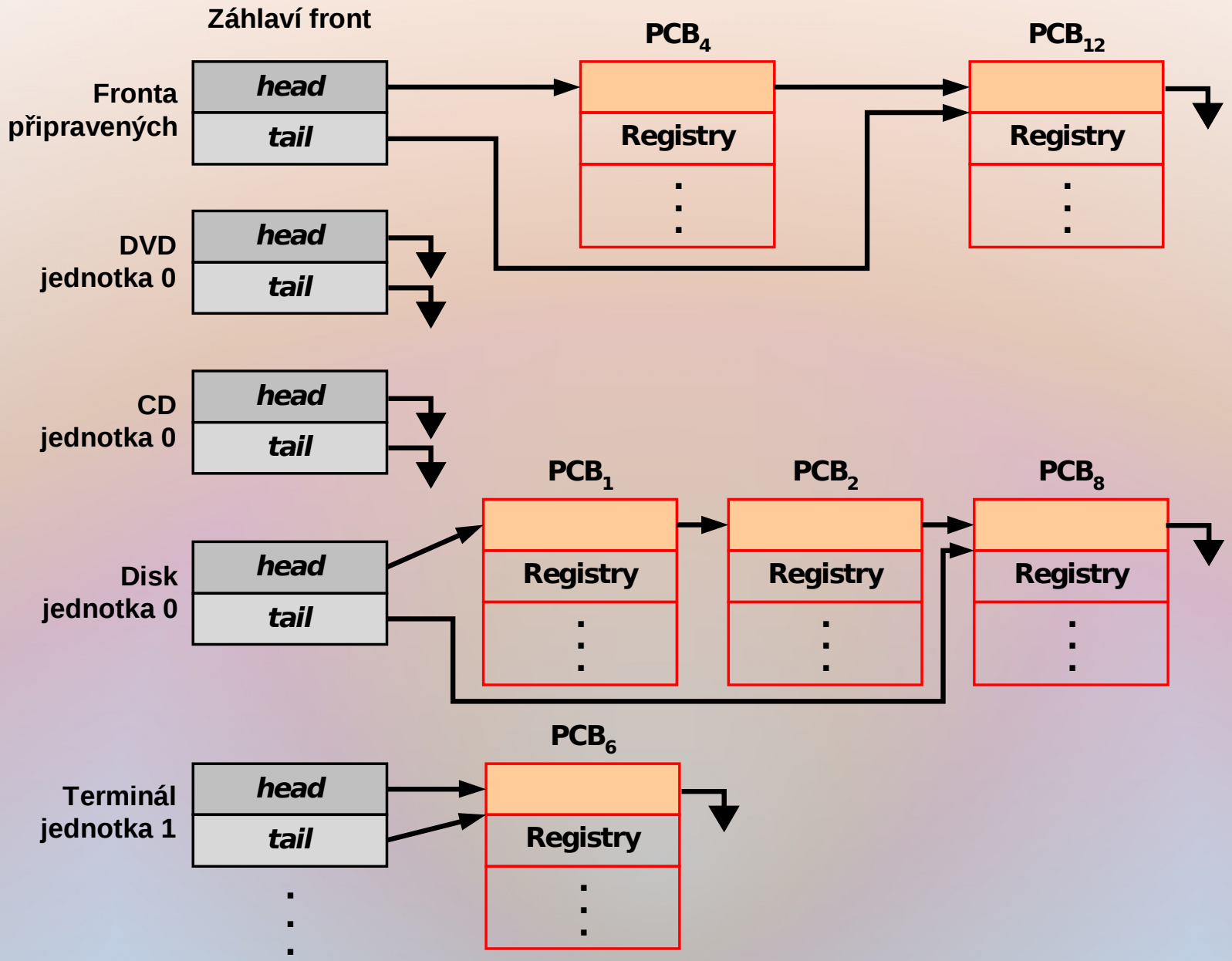
Popis procesů

- Deskriptor procesu – *Process Control Block* (PCB)
  - Datová struktura obsahující
    - Identifikátor procesu (pid) a rodičovského procesu (ppid)
    - Globální stav (*process state*)
    - Místo pro uložení všech registrů procesoru
    - Informace potřebné pro plánování procesoru/ů
      - Priorita, historie 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*)
    - …
    - Ukazatelé 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

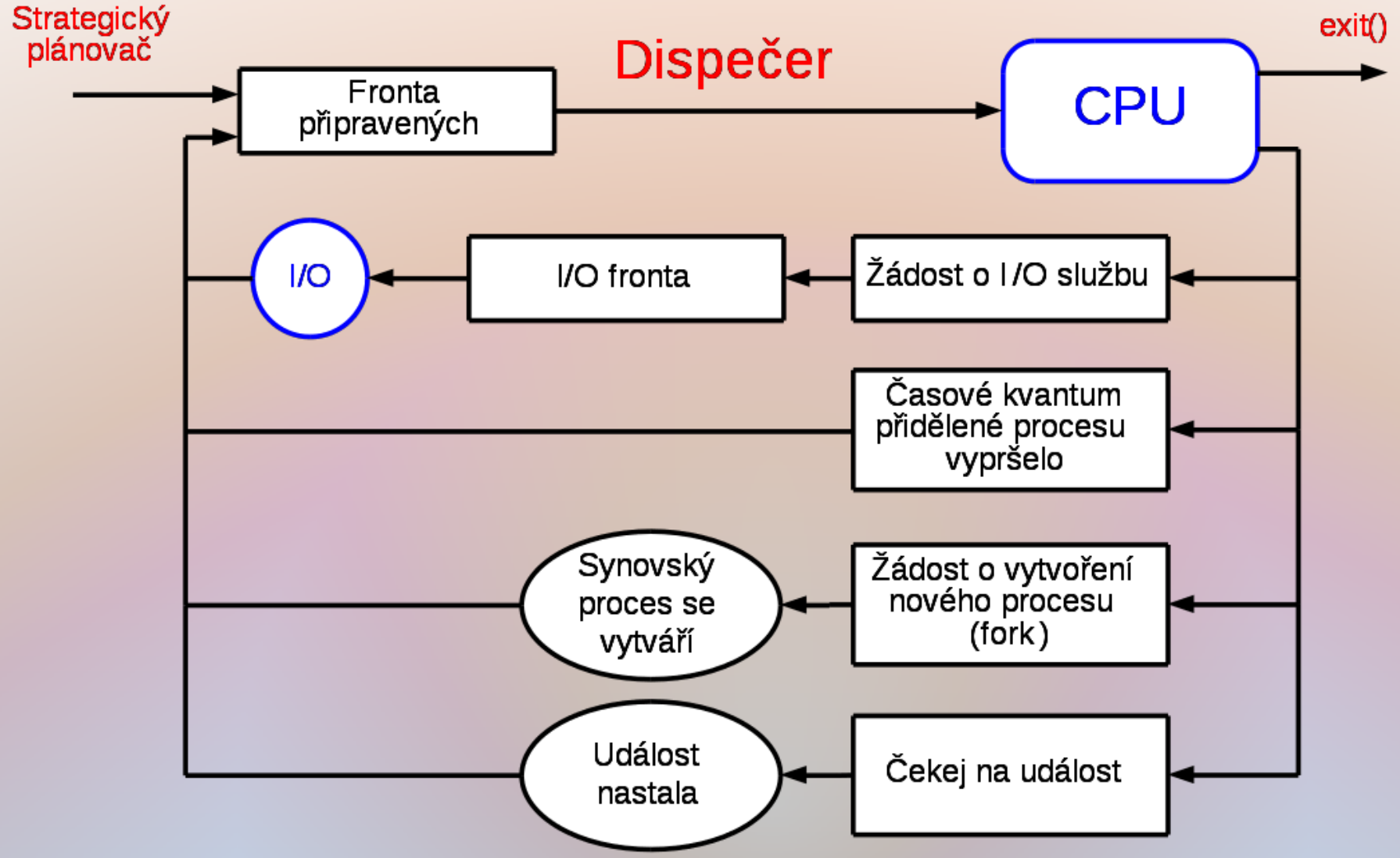
Fronty a seznamy procesů – příklad



Plánovače v OS

- Krátkodobý plánovač (operační plánovač, dispečer):
  - 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
- 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

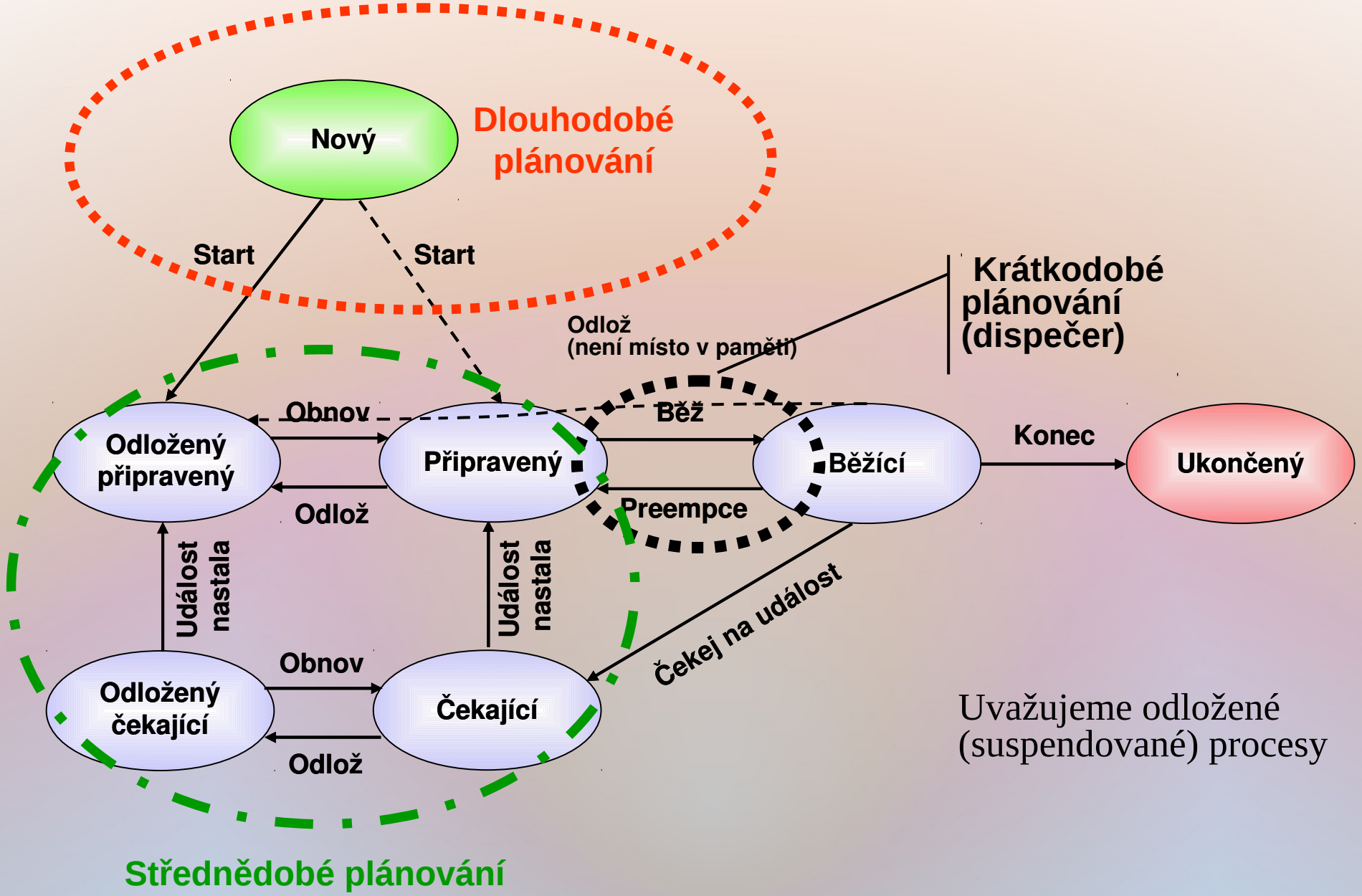
Strategický plánovač a dispečer



Středně dobý plánovač - *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

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

Plánovací algoritmy

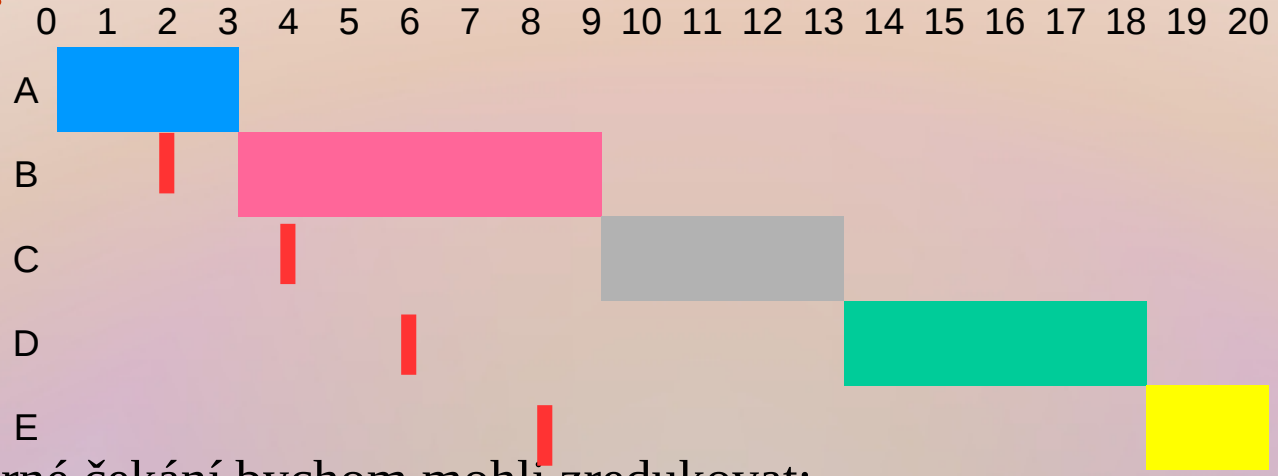
- Ukážeme plánování:
  - FCFS (*First-Come First-Served*)
  - SPN (SJF) (*Shortest Process Next*)
  - SRT (*Shortest Remaining Time*)
  - cyklické (*Round-Robin*)
  - zpětnovazební (*Feedback*)
- Příklad
  - používaný v tomto textu pro ilustraci algoritmů

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

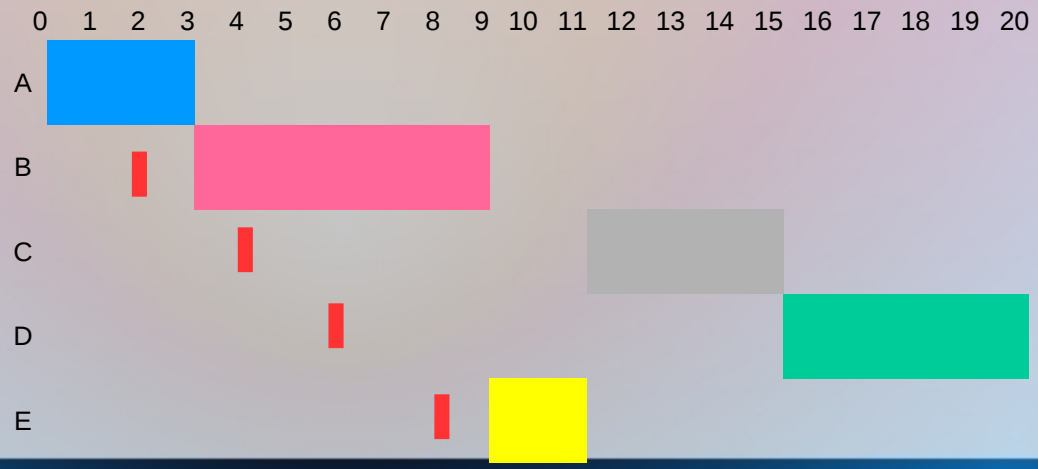
Plánování FCFS

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



- 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 T_{Avg}^{w} 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:

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

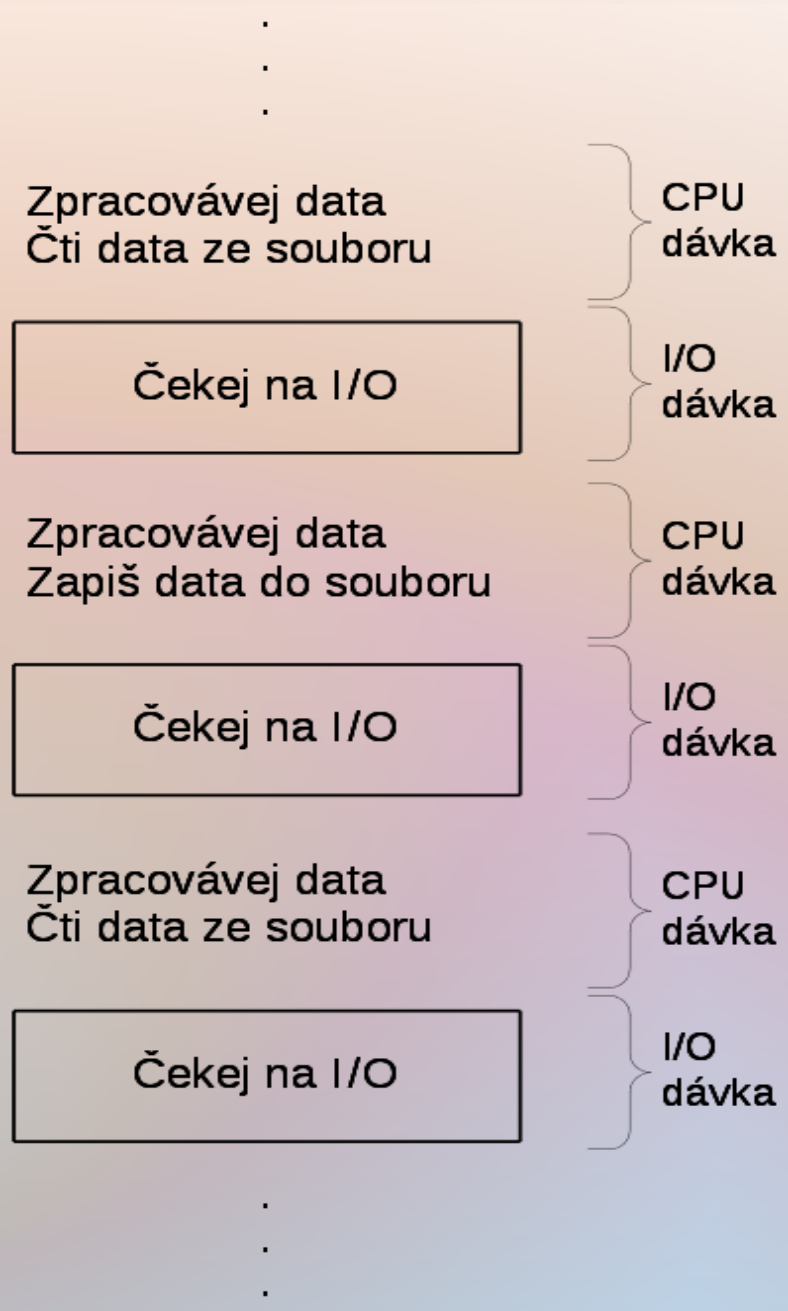
Plánování SRT

- 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:

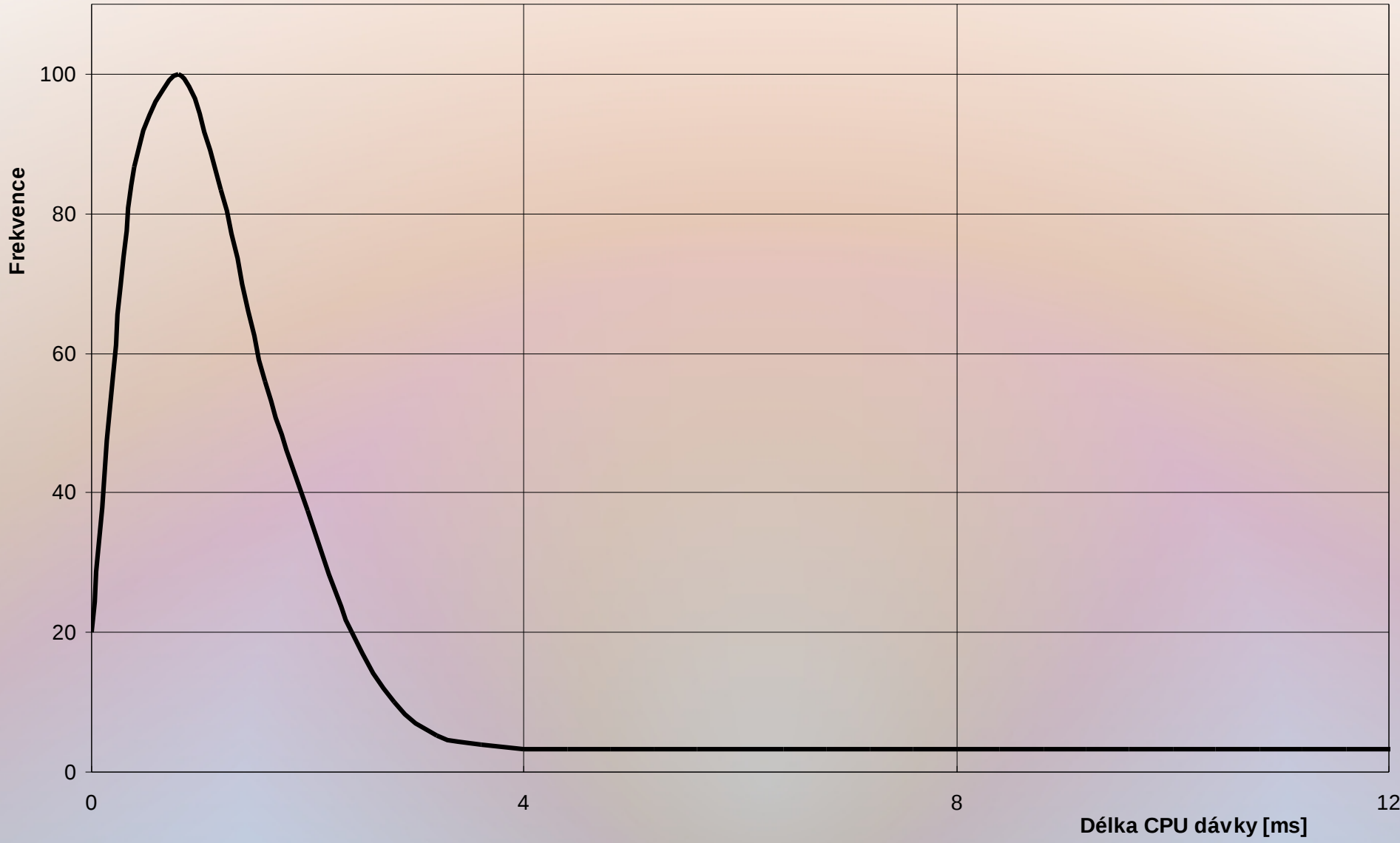
$$^{w}T_{\text{Avg}} = \frac{0+1+0+9+0}{5} = 2{,}0$$

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



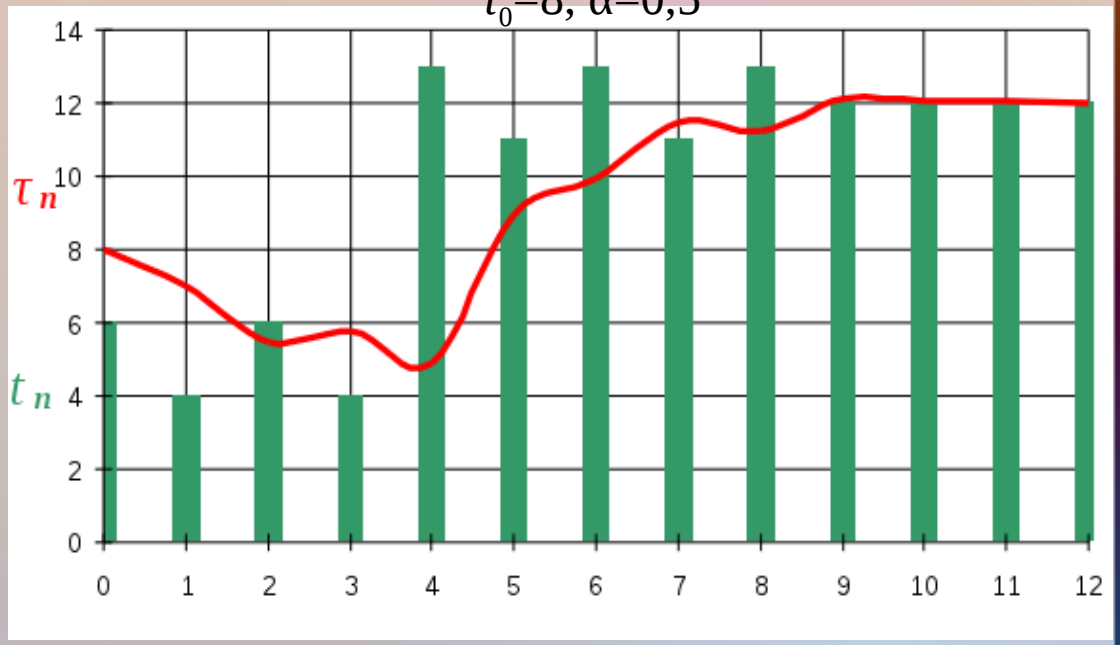
Typický histogram délek CPU dávek



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

τ_0=8, α=0,5



Prioritní plánování (1)

- Každému procesu je přiřazeno prioritní číslo
  - Prioritní číslo – preference procesu při výběru procesu, kterému má být přiřazena CPU
  - CPU se přiděluje procesu s nejvyšší prioritou
  - Nejvyšší prioritě obvykle odpovídá (obvykle) nejnižší prioritní číslo
    - Ve Windows je to obráceně
- Existují se opět dvě varianty:
  - Nepreemptivní
    - Jakmile se vybranému procesu procesor předá, procesor mu nebude odňat, dokud se jeho CPU dávka nedokončí
  - Preemptivní
    - Jakmile se ve frontě připravených objeví proces s prioritou vyšší, než je priorita právě běžícího procesu, nový proces předběhne právě běžící proces a odejme mu procesor
- SPN i SRT jsou vlastně případy prioritního plánování
  - Prioritou je predikovaná délka příští CPU dávky
  - SPN je nepreemptivní prioritní plánování
  - SRT je preemptivní prioritní plánování

Prioritní plánování (2)

- Problém stárnutí (*starvation*):
  - Procesy s nízkou prioritou nikdy nepoběží; nikdy na ně nepřijde řada
    - Údajně: Když po řadě let vypínali v roce 1973 na M.I.T. svůj IBM 7094 (jeden z největších strojů své doby), našli proces s nízkou prioritou, který čekal od roku 1967. 
- Řešení problému stárnutí: zrání procesů (*aging*)
  - Je nutno dovolit, aby se procesu zvyšovala priorita na základě jeho historie a doby setrvávání ve frontě připravených
    - Během čekání na procesor se priorita procesu zvyšuje

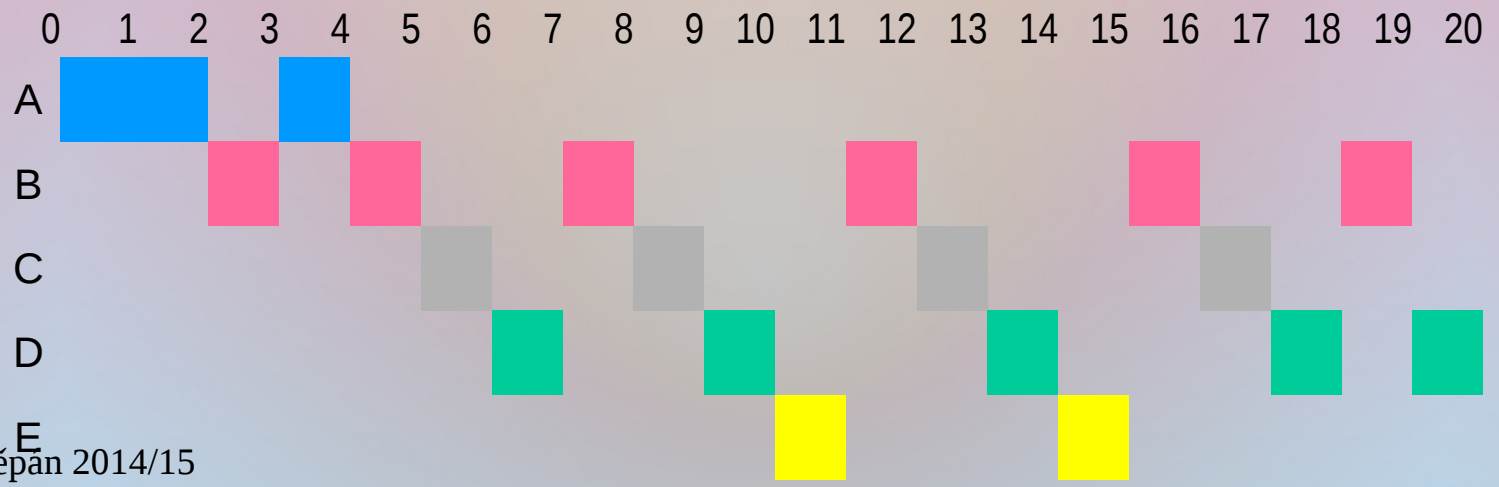
Cyklické plánování

- Cyklická obsluha (*Round-robin*) – RR
- Z principu preemptivní plánování
- Každý proces dostává CPU periodicky na malý časový úsek, tzv. časové kvantum, délky q (~ desítky ms)
  - V „čistém“ RR se uvažuje shodná priorita všech procesů
  - Po vyčerpání kvanta je běžícímu procesu odňato CPU ve prospěch nejstaršího procesu ve frontě připravených a dosud běžící proces se zařazuje na konec této fronty
  - Je-li ve frontě připravených procesů n procesů, pak každý proces získává $1/n$-tinu doby CPU
  - Žádný proces nedostane 2 kvanta za sebou (samozřejmě pokud není jediný připravený)
  - Žádný proces nečeká na přidělení CPU déle než $(n-1)q$ časových jednotek

Cyklické plánování (2)

- Efektivita silně závisí na velikosti kvanta
  - Veliké kvantum – blíží se chování FCFS
    - Procesy dokončí svoji CPU dávku dříve, než jim vyprší kvantum.
  - Malé kvantum => časté přepínání kontextu => značná režie
- Typicky
  - Dosahuje se průměrné doby obrátky delší oproti plánování SRT
  - Výrazně lepší je čas odezvy
  - Průměrná doba obrátky se může zlepšit, pokud většina procesů se době *q* ukončí
  - Empirické pravidlo pro stanovení *q*: cca 80% procesů by nemělo vyčerpat kvantum

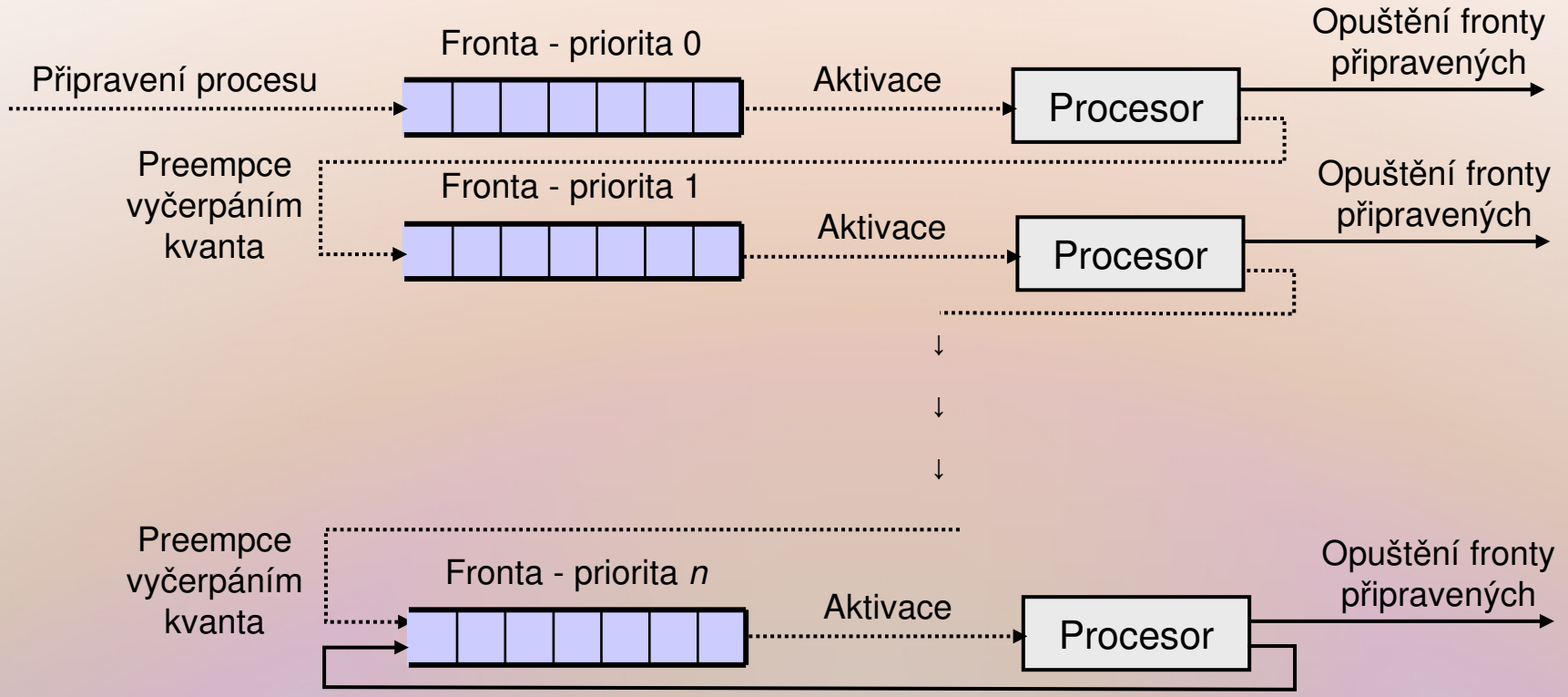
Příklad:



Zpětnovazební plánování

- Základní problém:
  - Neznáme předem časy, které budou procesy potřebovat
- Východisko:
  - Penalizace procesů, které běžely dlouho
- Řešení:
  - Dojde-li k preempci přečerpáním časového kvanta, procesu se snižuje priorita
  - Implementace pomocí víceúrovňových front
    - pro každou prioritu jedna
  - Nad každou frontou samostatně běží algoritmus určitého typu plánování
    - obvykle RR s různými kvanty a FCFS pro frontu s nejnižší prioritou

Víceúrovňové fronty - Windows



- Proces opouštějící procesor kvůli vyčerpání časového kvanta je přeřazen do fronty s nižší prioritou
- Fronty s nižší prioritou mohou mít delší kvanta
- Problém stárnutí ve frontě s nejnižší prioritou
  - Řeší se pomocí zrání (*aging*) – v jistých časových intervalech (~ 10^1 s) se zvyšuje procesům priorita, a tak se přemísťují do „vyšších“ front

O(1) plánovač – Linux 2.6-2.6.23

- O(1) - rychlost plánovače nezávisí na počtu běžících procesů – je rychlý a deterministický
- Dvě sady víceúrovňových front
  - Na začátku první sada obsahuje připravené procesy, druhá je prázdná
  - Při vyčerpání časového kvanta je proces přeřazen do druhé sady front do nové úrovně
  - Vzbuzené procesy jsou zařazovány podle toho, zda ještě nevyužily celé svoje časové kvantum do aktivní sady front, nebo do druhé sady front
  - Pokud je první sada prázdná, dojde k prohození první a druhé sady front procesů
- Heuristika pro odhad interaktivních procesů a jejich udržování na nejvyšších prioritách s odpovídajícími časovými kvanty

Zcela férový plánovač

- Linux od verze 2.6.23 (Completely Fair Scheduler)
- Nepoužívá fronty, ale jednu strukturu, která udržuje všechny procesy uspořádané podle délky již spotřebovaného času – čím méně proces spotřeboval strojového času, tím větší má nárok na přidělení procesoru
- Pro rychlou implementaci se používá vyvážený binární červeno-černý strom, zaručující složitost úměrnou *log(n)* počtu procesů
- Nepotřebuje složité heuristiky pro detekci interaktivních procesů
- Jediný parametr je časové kvantum:
  - pro uživatelské PC se volí menší pro větší
  - pro serverové počítače větší kvantu omezuje režii s přepínáním procesů a tím zvyšuje propustnost serveru
- Žádný proces nemůže zestárnout, všechny procesy mají stejné podmínky

Plánování v multiprocesorech

- Přiřazování procesů (vláken) procesorům:
  - Architektura „master/slave“
    - Klíčové funkce jádra běží vždy na jednom konkrétním procesoru
    - Master odpovídá za plánování
    - Slave žádá o služby mastera
    - Nevýhoda: dedikace
      - Přetížený master se stává úzkým místem systému
  - Symetrický multiprocesing (SMP)
    - Všechny procesory jsou si navzájem rovny
    - Funkce jádra mohou běžet na kterémkoliv procesoru
  - SMP vyžaduje podporu vláken v jádře
- Proces musí být dělen na vlákna, aby SMP byl účinný
  - Aplikace je sada vláken pracujících paralelně do společného adresního prostoru
  - Vlákno běží nezávisle na ostatních vláknech svého procesu
  - Vlákna běžící na různých procesorech dramaticky zvyšují účinnost systému

Symetrický multiprocesing (SMP)

- Dvě řešení:
  - Jedna společná fronta pro všechny procesory
  - Každý procesor svojí frontu a migrace procesů mezi procesory
- Jedna společná (globální) fronta pro všechna vlákna
  - Fronta „napájí“ společnou sadu procesorů
    - Fronta může být víceúrovňová dle priorit
  - Každý procesor si sám vyhledává příští vlákno
    - Přesněji: instance plánovače běžící na procesoru si je sama vyhledává ...
  - Problémy
  - Jedna centrální fronta připravených sledů vyžaduje používání vzájemného vylučování v jádře
    - Kritické místo v okamžiku, kdy si hledá práci více procesorů
    - Předběhnutá (přerušená) vlákna nebudou nutně pokračovat na stejném procesoru – nelze proto plně využívat cache paměti procesorů
- Každý procesor svoji frontu
  - Heuristická pravidla, kdy frontu změnit
- SMP používáno ve
  - Windows, Linux, Mac OS X, Solaris, BSD4.4

Poznámky k plánování v multiprocesorech

- Používají se různá (heuristická) pravidla (i při globální frontě):
  - Afinita vlákna k CPU – použij procesor, kde vlákno již běželo (možná, že v cache CPU budou ještě údaje z minulého běhu)
  - Použij nejméně využívaný procesor
- Mnohdy značně složité
  - při malém počtu procesorů (≤ 4) může přílišná snaha o optimalizaci plánování vést až k poklesu výkonu systému
    - Tedy aspoň v tom smyslu, že výkon systému neporoste lineárně s počtem procesorů
  - při velkém počtu procesorů dojde naopak k „nasycení“, neboť plánovač se musí věnovat rozhodování velmi často (končí CPU dávky na mnoha procesorech)
    - Režie tak neúměrně roste

Systémy reálného času (RT)

- Obvykle malé systémy se specializovaným použitím
  - Často vestavěné (*embedded*)
- Správná funkce systému závisí nejen na logickém (či numerickém) výsledku ale i na čase, kdy bude výsledek získán
  - Správně určený výsledek dodaný pozdě je k ničemu
- Úlohy a procesy reagují na události pocházející zvenčí systému a navenek dodávají své výsledky
  - Nastávají v „reálném čase“ a potřebná reakce musí být včasná
- Příklady
  - Řízení laboratorních či průmyslových systémů
  - Robotika
  - Řízení letového provozu
  - Telekomunikační aplikace (digitální ústředny)
  - Vojenské systémy velení a řízení
  - ...

Charakteristiky OS RT

- Determinismus
  - Operace jsou prováděny ve fixovaných, předem určených časech nebo časových intervalech
  - Reakce na přerušení musí proběhnout tak, aby systém byl schopen obsluhy všech požadavků v požadovaném čase (včetně vnořených přerušení)
- Uživatelské řízení
  - Uživatel (návrhář systému) specifikuje:
    - Priority
    - Práva procesů
    - Co musí vždy zůstat v paměti
- Spolehlivost
  - Degradace chování může mít katastrofální důsledky
- Zabezpečení
  - Schopnost systému zachovat v případě chyby aspoň částečnou funkcionalitu a udržet maximální množství dat

Požadavky na OS RT

- Extrémně rychlé přepínání mezi procesy či vlákny
- OS musí být malý
- Multiprogramování s meziprocesními komunikačními nástroji
  - semafory, signály, události ➔
- Speciální souborové systémy s velkou rychlostí
  - RAM disky, souvislé soubory
- Plánování založené na prioritách
  - Pozor: preempce je ale časově náročná
- Minimalizace časových úseků, kdy je vypnut přerušovací systém
- Zvláštní hardwarové vybavení
  - hlídací časovače (*watch-dog timers*) a alarmy

Plánování CPU v RT systémech

- Tabulkou řízené statické plánování
  - Určuje pevně, kdy bude který proces spuštěn tak, aby včas skončil
  - Nejčastější případ v uzavřených systémech s předem známými procesy a jejich vlastnostmi
- Preemptivní plánování se statickými prioritami
  - Používá klasický prioritní plánovač s fixními prioritami
    - Může být problematické kvůli velké režii spojené s preempcí
- Dynamické plánování
  - Za běhu určuje proveditelnost (splnitelnost požadavků)
  - V tzv. přísných RT systémech (*Hard real-time systems*) téměř nepoužitelné vlivem velké režie
    - *Hard real-time systems* musí přísně zaručovat dokončení časově kritických procesů do předepsaného termínu

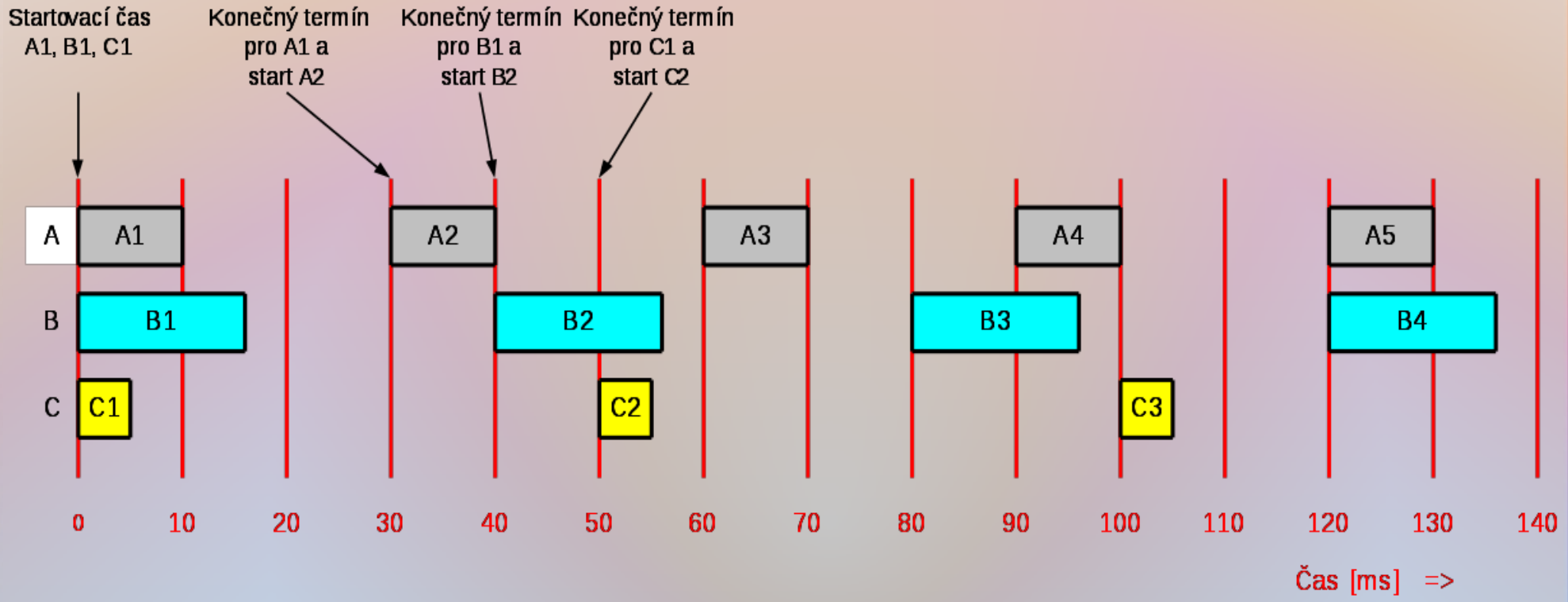
Periodické plánování dle konečného termínu

- Procesům jde zejména o včasné dokončení v rámci zadané periody běžících procesů
  - Např. v daných intervalech je třeba vzorkovat napětí z čidel
- O každém procesu je znám
  - Potřebný čas (horní hranice délky CPU dávky)
  - Termín začátku a nejzazšího konce každého běhu periodicky spouštěného procesu
- Předpoklady (zjednodušení)
  - Termín dokončení je identický s počátkem následující periody
  - Požadavky na systémové prostředky (či potřebu čekání na jejich přidělení) budeme ignorovat

Příklad 1

- 3 periodické procesy

Proces	Perioda p_i	Procesní čas T_i	T_i / p_i
A	30	10	0,333
B	40	15	0,375
C	50	5	0,100



Plánovatelnost v periodických úlohách

- Relativní využití strojového času
  - Proces *i* využije poměrnou část $r_i = \frac{T_i}{p_i}$ celkového

    strojového času, kde T_i je procesní čas a p_i je jeho perioda
  - Celkové využití je $r = \sum_{i=1}^{N} r_i = \sum_{i=1}^{N} \frac{T_i}{p_i}$
  - Aby vše mohlo pracovat musí platit (fyzikální podmínka plánovatelnosti) $r = \sum_{i=1}^{N} \frac{T_i}{p_i} \leq 1$
- Náš Příklad 1 $r = \sum_{i=1}^{3} \frac{T_i}{p_i} = \frac{10}{30} + \frac{15}{40} + \frac{5}{50} = 0{,}808 < 1$

Plánování RMS

- Algoritmus RMS = *Rate Monotonic Scheduling*
- Statické priority
  - $Prio_i \approx p_i$ (kratší perioda = menší číslo ~ vyšší priorita)
- Používá se pro procesy s následujícími vlastnostmi
  - Periodický proces musí skončit během své periody
  - Procesy jsou vzájemně nezávislé
  - Každý běh procesu (CPU dávka) spotřebuje konstantní čas
  - Předpokládá se, že preempce nastává okamžitě (bez režie)
- Poznatek
  - Plánování je úspěšné, pokud se všechny procesy stihnou v době odpovídající periodě procesu s nejdelší periodou

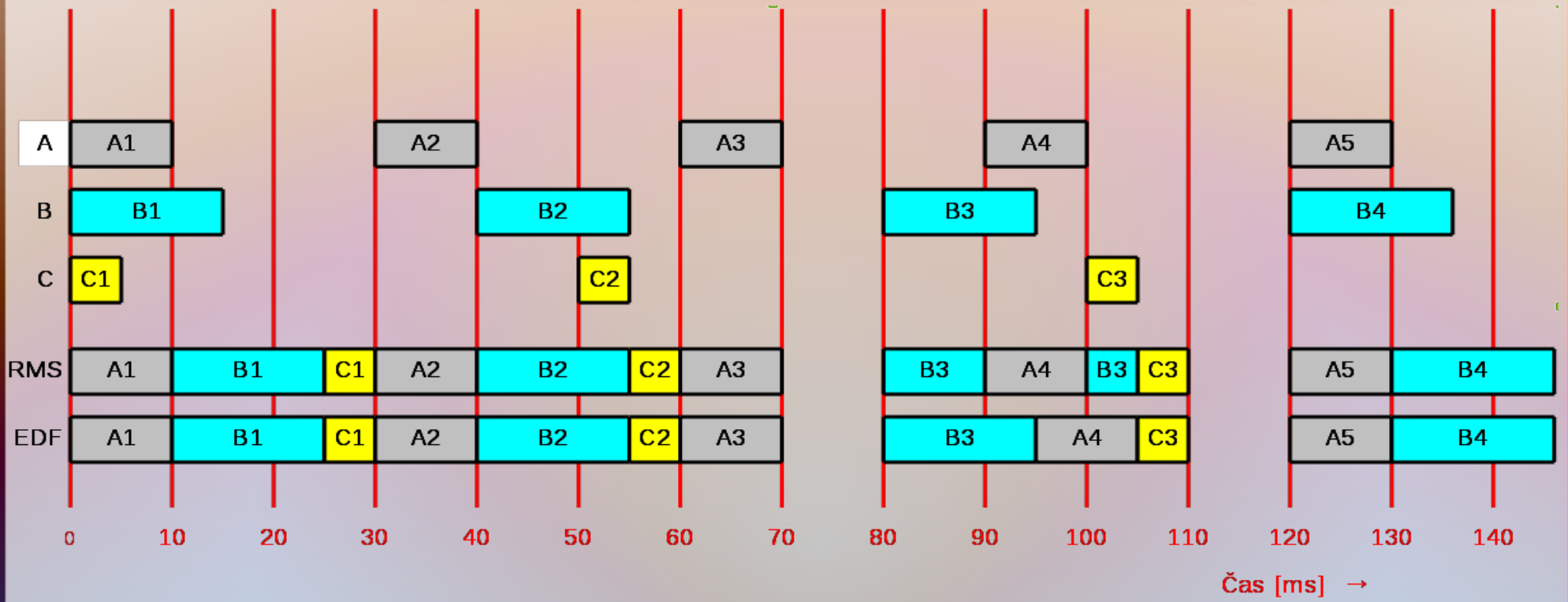
Plánování EDF

- EDF = *Earliest Deadline First*
  - Upřednostňuje proces s nejbližším termínem dokončení
- Dynamické priority
  - Plánovač vede seznam připravených procesů uspořádaný podle požadovaných časů dokončení a spustí vždy ten s nejbližším požadovaným termínem dokončení
- Použitelné i v následujících situacích
  - Procesy nemusí být přísně periodické ani nemusí mít konstantní dobu běhu
  - Pokud preempce nastává okamžitě, pak při plánování periodických procesů lze dodržet dokončovací termíny i při vytížení téměř 100%
- Vlastnosti
  - Algoritmus není analyticky plně prozkoumán
  - Následky přetížení nejsou známy a nejsou předvídatelné
  - Není známo chování v případech, kdy dokončovací termín a perioda jsou různé

Příklad 1 (pokračování)

Proces	Perioda p_i	Procesní čas T_i
A	30	10
B	40	15
C	50	5

$$r = 0{,}808$$



- Oba algoritmy fungují
  - Dokonce chvílemi zbývá volný čas k běhu nějakého procesu „na pozadí“

Příklad 2

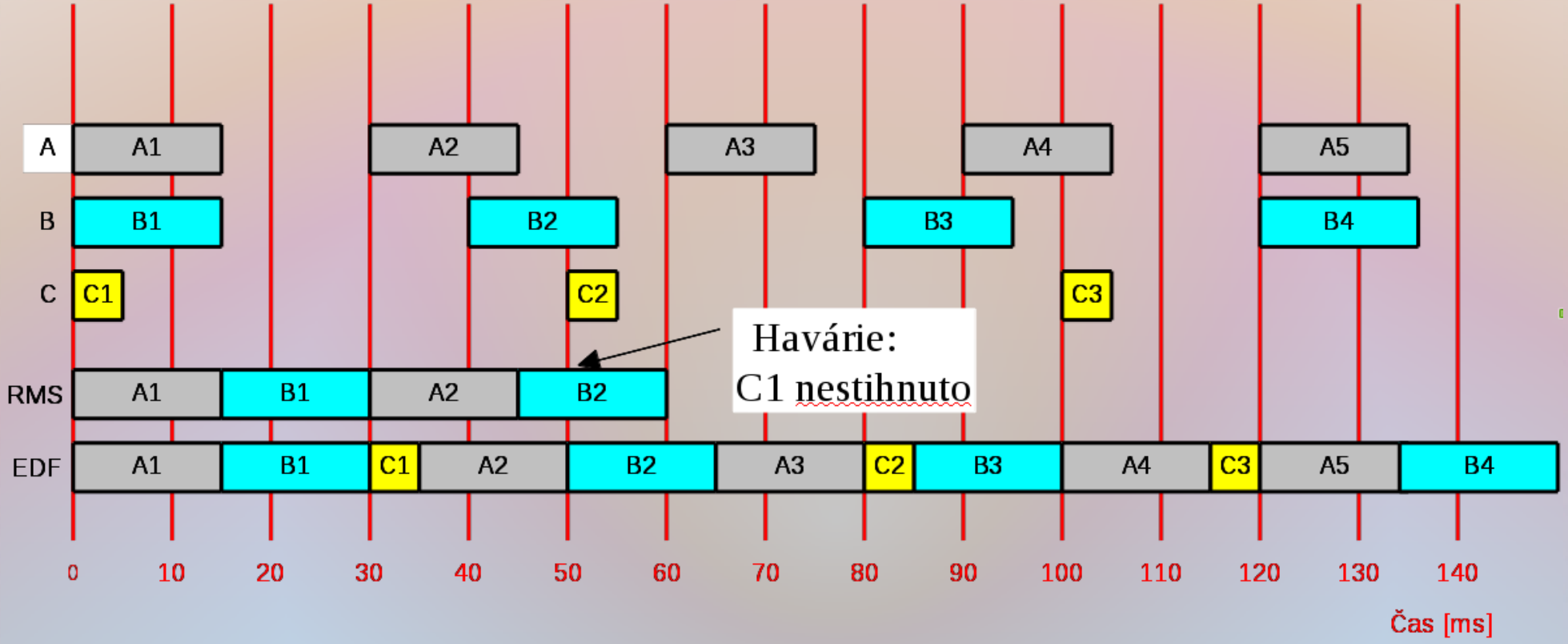
- Opět 3 periodické procesy

Proces	Perioda p_i	Procesní čas T_i	T_i / p_i
A	30	**15**	0,500
B	40	15	0,375
C	50	5	0,100

Suma

$R = 0{,}975 < 1$

Plánovatelné



Plánování RMS podrobněji

- Dobře analyticky zpracovaný algoritmus zaručující dodržení termínů dokončení, pokud při N procesech platí (***postačující podmínka***) [Liu & Layland 1973]:

$$r = \sum_{i=1}^{N} \frac{T_i}{p_i} \leq N\left(\sqrt[N]{2} - 1\right);$$

$$\lim_{N \to \infty} N\left(\sqrt[N]{2} - 1\right) = \ln 2 \approx 0.693147$$

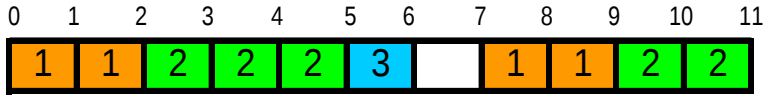
N	$N\left(\sqrt[N]{2} - 1\right)$
2	0,828427
3	0,779763
4	0,756828
5	0,743491
10	0,717734
20	0,705298

- Jsou vypracovány i způsoby spolupráce sdílených systémových prostředků
- Je známo i chování algoritmu při přechodném „přetížení“ systému
- Používáno v téměř všech komerčních RT OS

Plánování RMS podrobněji (2)

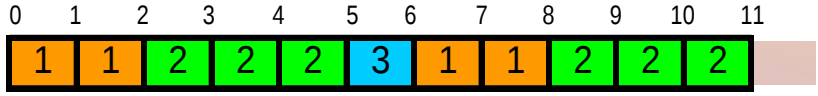
- Jak je to s použitelností RMS?

P	i	p_i	T_i	T_i/p_i	$\Sigma(T_i/p_i)$
A	1	7	2	0,286	0,286
B	2	8	3	0,375	0,661
C	3	10	1	0,100	**0,761**

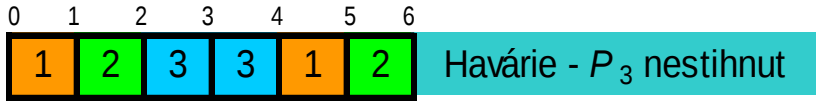


$$3(\sqrt[3]{2}-1) = 0,7797$$

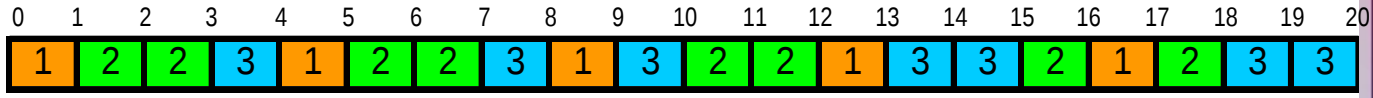
P	i	p_i	T_i	T_i/p_i	$\Sigma(T_i/p_i)$
A	1	6	2	0,333	0,333
B	2	8	3	0,375	0,708
C	3	10	1	0,100	**0,808**



P	i	p_i	T_i	T_i/p_i	$\Sigma(T_i/p_i)$
A	1	4	1	0,250	0,250
B	2	5	1	0,200	0,450
C	3	6	3	0,500	**0,950**



P	i	p_i	T_i	T_i/p_i	$\Sigma(T_i/p_i)$
A	1	4	1	0,250	0,250
B	2	5	2	0,400	0,650
C	3	20	7	0,350	**1,000**



  - Celkové vytížení není zřejmě základním předpokladem pro použitelnost RMS

Plánování RMS podrobněji (3)

- Lehoczky, Sha & Ding [1989] podrobili RMS analýze znovu. Výsledek:
  - Mějme procesy $\left\{ P_i, i = 1 \ldots N \,\middle|\, p_i \le p_{i+1}, i = 1 \ldots N-1 \right\}$
  - Soustřeďme se na procesy 1...*i*, (*i*=1...*N*) a určeme vždy

$$W_i(t) = \sum_{j=1}^{i} T_j \left\lceil t / p_j \right\rceil, \; L_i(t) = W_i(t)/t,$$

$$L_i = \min_{\{0<t\le T_i\}} L_i(t), \;\; L = \max_{\{1\le i\le N\}} L_i$$

$W_i(t)$ reprezentuje kumulativní potřeby procesů P_1 ... P_i v časovém úseku [0, t]

  - Nutnou a postačující podmínkou pro spolehlivé použití algoritmu RMS je $L \le 1$.
  - Pro určování $W_i(t)$ je nepříjemný „spojitý“ čas t.
  - Autoři ukázali, že stačí určovat $W_i(t)$ jen v časech t rovných násobku periody každého z procesů
  - Např. pro $\{p_1 = 4; p_2 = 5; p_3 = 13\}$ stačí počítat $W_i(t)$ a $L_i(t)$ pouze pro

$$t \in \{4, 5, 8, 10, 12, 13\}$$

Plánování RMS podrobněji (4)

- Příklady použití uvedené teorie
  - RMS zhavaruje

i	p_i	T_i	T_i/p_i	$\Sigma(T_i/p_i)$
1	4	1	0,250	0,250
2	5	1	0,200	0,450
3	6	3	0,500	**0,950**

$L_i(4)$	$L_i(5)$	$L_i(6)$	L_i	L
0,250	0,400	0,333	0,250	1,167
0,500	0,600	0,667	0,500	
1,250	1,200	1,167	1,167	

  - RMS bude funkční

i	p_i	T_i	T_i/p_i	$\Sigma(T_i/p_i)$
1	4	1	0,250	0,250
2	5	2	0,400	0,650
3	20	7	0,350	**1,000**

$L_i(4)$	$L_i(5)$	$L_i(8)$	$L_i(10)$	$L_i(12)$	$L_i(15)$	$L_i(16)$	$L_i(20)$	L_i	L
0,250	0,400	0,250	0,300	0,250	0,267	0,250	0,250	0,250	1,000
0,750	0,800	0,750	0,700	0,750	0,667	0,750	0,650	0,650	
2,500	2,200	1,625	1,400	1,333	1,133	1,188	1,000	1,000	

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

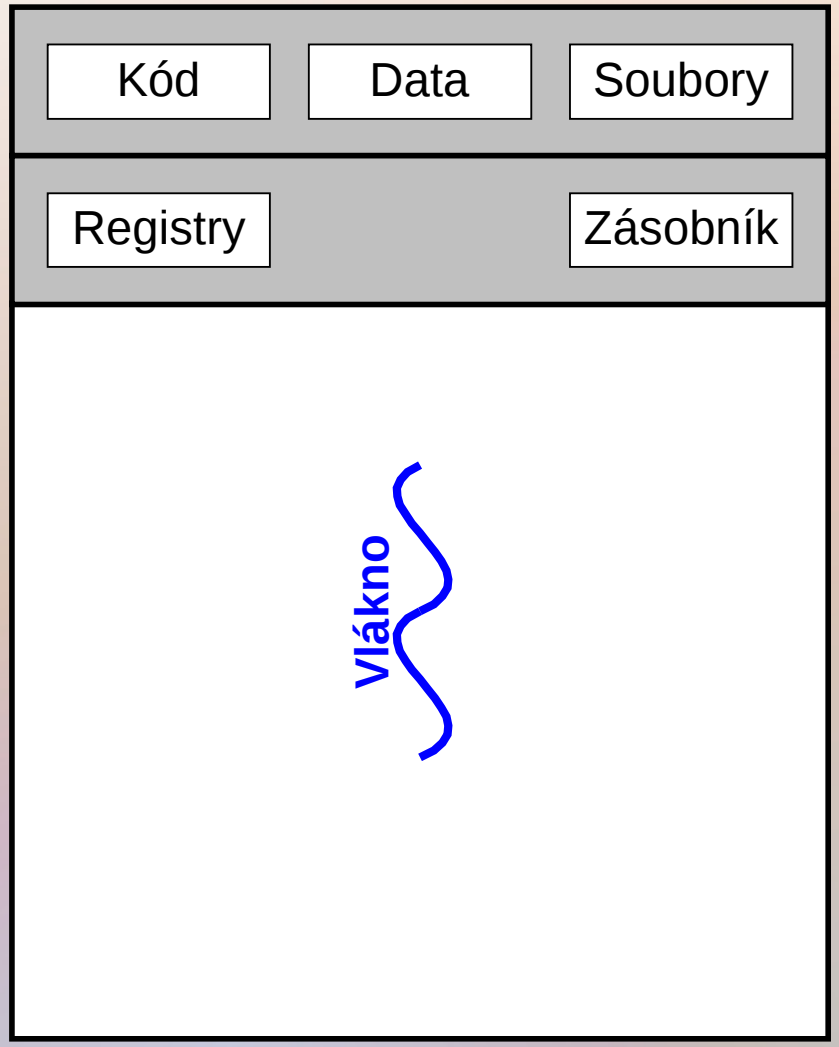
Vztah procesu a vlákna

- 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

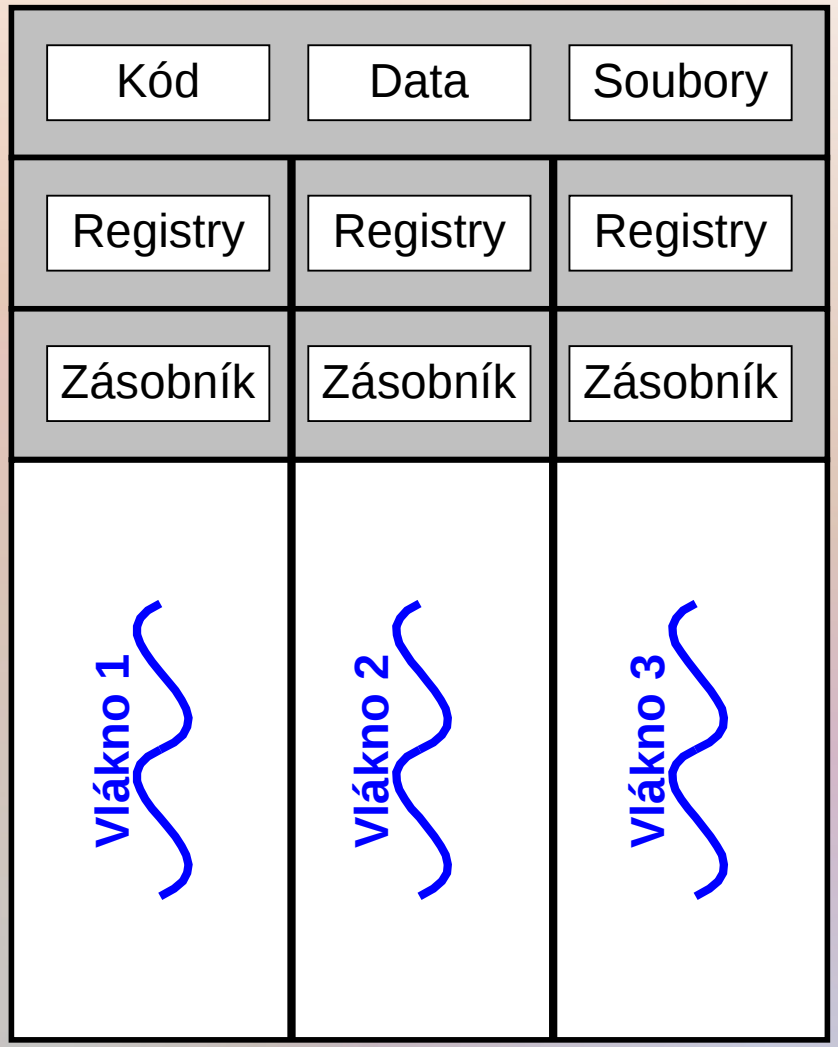
Proces a jeho vlákna

- 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

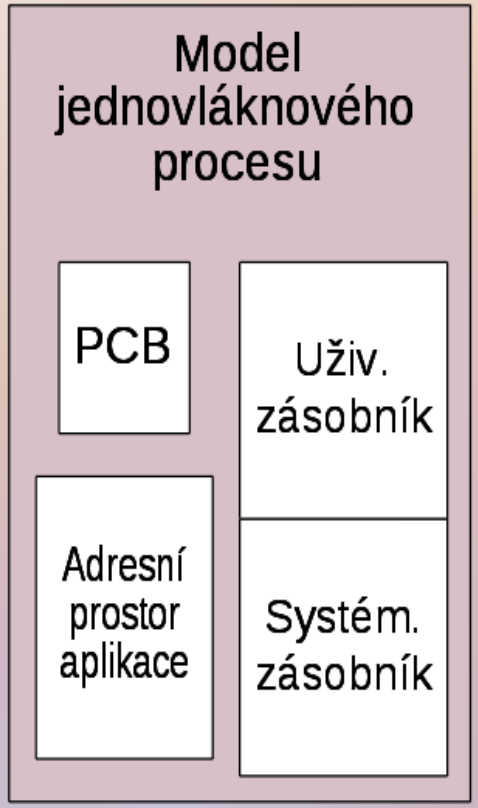


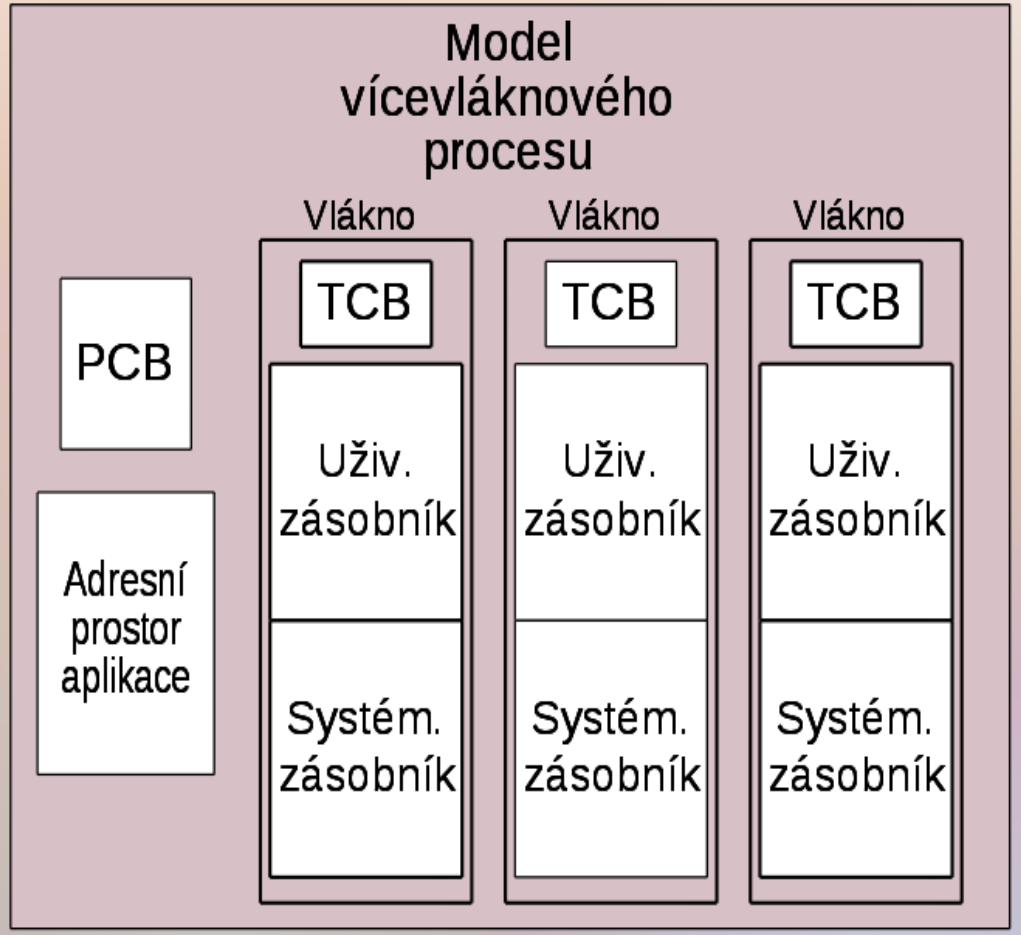
Jednovláknový proces



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

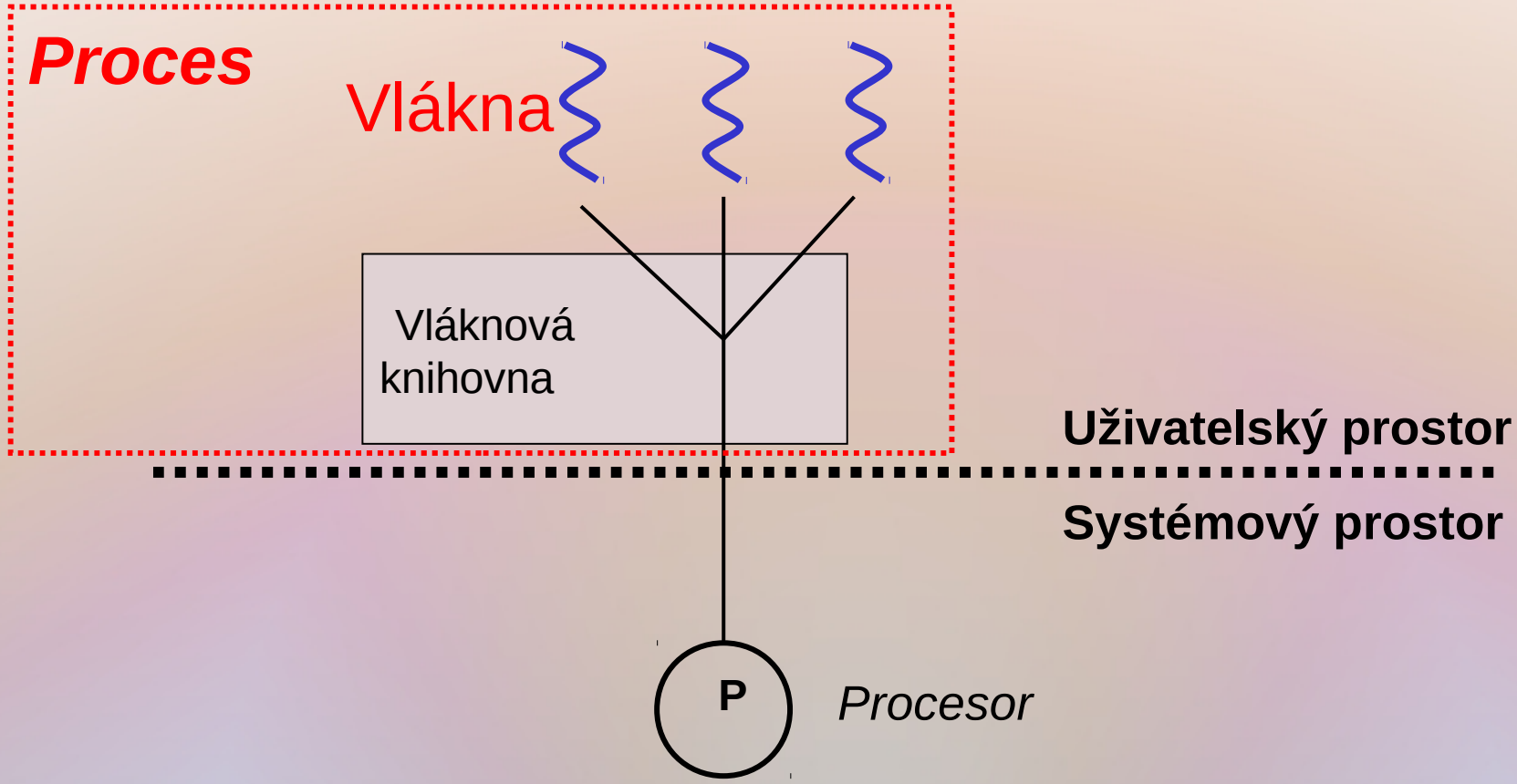
Stavy a odkládání vláken

- 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)

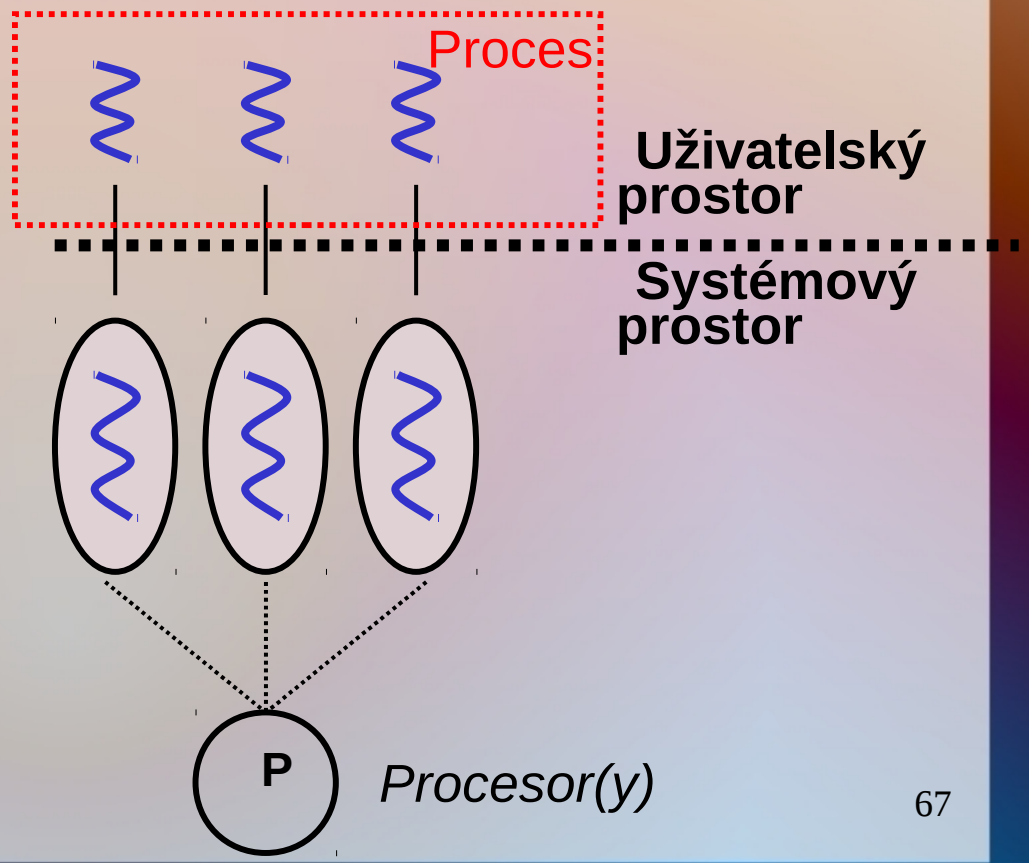
- Problém stavu vláken: Co když se proces nebo vlákno zablokuje?
  - Nechť proces *A* má dvě vlákna T_1 a T_2, přičemž T_1 právě běží
  - Mohou nastat následující situace:
    - T_1 požádá JOS o I/O operaci nebo jinou službu:
      - Jádro zablokuje proces *A* jako celek.
      - Celý proces čeká, přestože 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();
```

To je dnes vše.

Otázky?