Uváznutí

PB 152 ◇ Operační systémy

Jan Staudek
http://www.fi.muni.cz/usr/staudek/vyuka/

Verze: jaro 2017

Osnova přednášky

- Příklady ,,ze života"
- Model
- Charakteristika uváznutí
- Metody zvládání uváznutí
- Prevence uváznutí
- Obcházení uváznutí
- Detekce uváznutí
- Obnova po uváznutí
- Kombinovaný přístup ke zvládání uváznutí

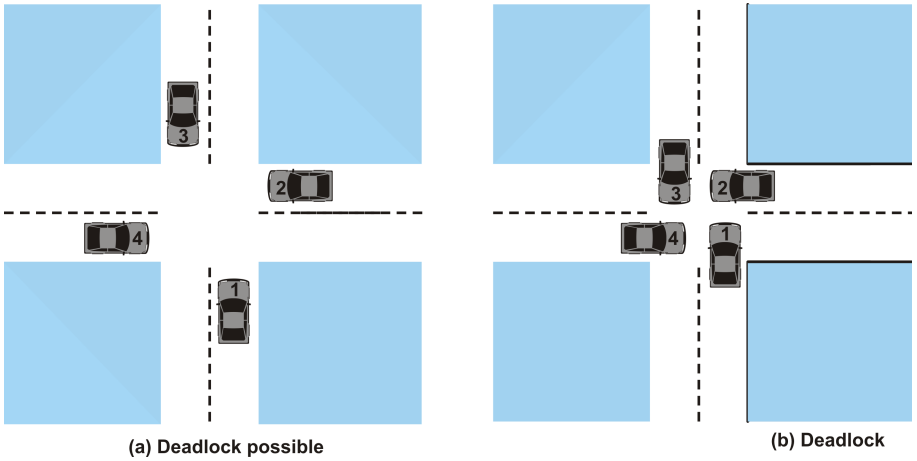
Problém uváznutí

When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.
Statute passed by the Kansas State Legislature,
early in the 20th century.

- Existuje množina blokovaných procesů:
  - každý vlastní nějaký prostředek (zdroj) a
  - každý čeká na zdroj držený jiným procesem z této množiny
- příklad
  - ✓ v systému se provozují dva mechanismy přidělovaných disků
  - ✓ v systému existují dva procesy
  - ✓ každý z procesů vlastní 1 mechanismus a požaduje alokaci dalšího
- Univerzální efektivní řešení problému uváznutí neexistuje

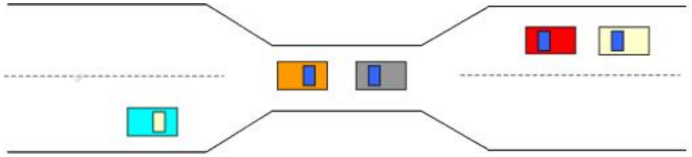
Problém uváznutí – soupeření o zdroj

Auta (procesy) soupeří o výhradní získání prostoru pro přejezd křižovatky (zdroj)



Problém uváznutí – soupeření o zdroj

Příklad řešení uváznutí odebráním zdroje

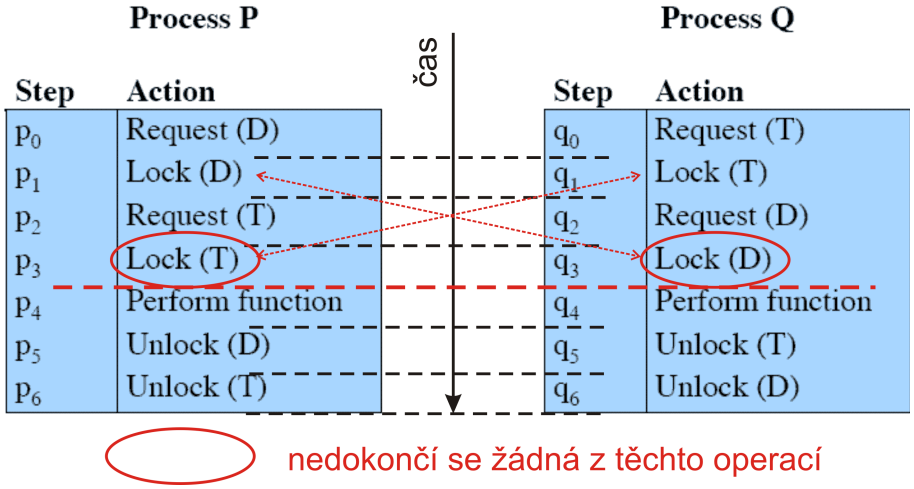


- každý vjezd na most lze chápat jako získání zdroje
- Dojde-li k uváznutí, lze ho řešit tím, že se jedno auto vrátí
- uplatnila se preempce zdroje – (*preemption* a *rollback*)
  (přivlastnění si zdroje, vlastněného někým jiný)
  a vrácení soupeře před žádost o přidělení zdroje
- Při řešení uváznutí
  - ✓ může se vracet i více vozů, zdroj lze odebírat více procesům
  - ✓ může docházet ke stárnutí konkrétního auta,
    pokud se toto bude opakovaně vracet

Kategorie zdrojů

- Zdroje použitelné pouze výlučně, opakovaně, *reusable*
  - ✓ v jednom čase použitelné jediným procesem
  - ✓ použitím zdroje se zdroj nevyčerpá, existuje dál
  - ✓ Procesor, kanál, IO zařízení, soubor, záznam, . . .
  - ✓ předchozí příklady používají opakovaně použitelné zdroje
- Zdroje spotřebovávané, *consumable*
  - ✓ vznikají generováním a zanikají zpřístupněním
  - ✓ přerušení, signály, zprávy, . . .

Dva procesy soupeří o opakovaně přístupný zdroj



Dva procesy soupeří o opakovaně přístupný zdroj

- prostor v paměti lze používat jako opakovaně použitelný zdroj
- příklad uváznutí při přidělování paměti
  - ✓ dostupná kapacita 200 MB

```
P0:              P1:
rqst(80KB);      rqst(70KB);
...              ...
rqst(60KB);      rqst(80KB);
```


Příklad uváznutí při použití spotřebovávaného zdroje

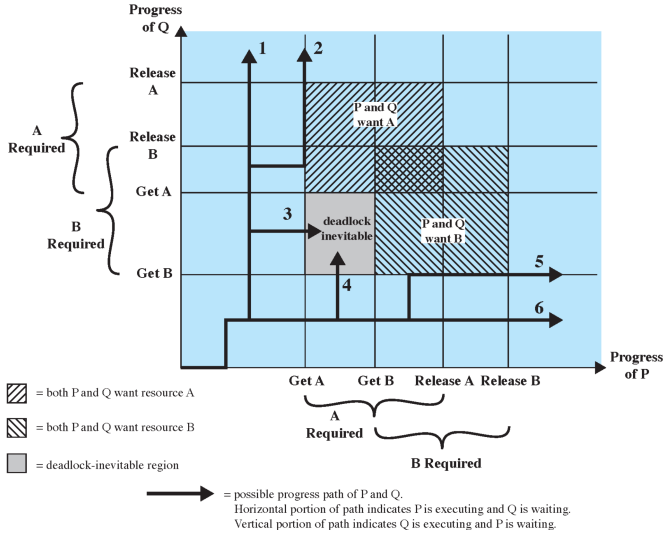
- příklad uváznutí při výměně zpráv, čekáním na událost
  - ✓ operace receive je synchronní, končí přijetím zprávy

```
P0:              P1:
...              receive(P0, q);
...              ...
receive(P1, m);  ...
```

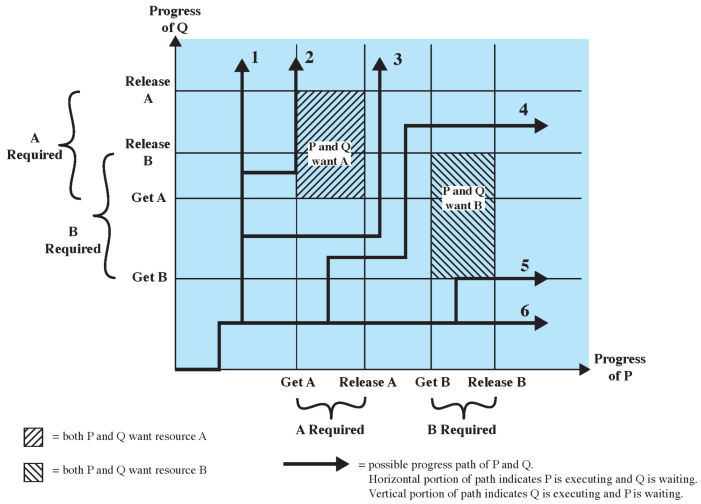

Definice uváznutí a stárnutí

- uváznutí
  - ✓ množina procesů P uvázla,
    jestliže každý proces Pi z P čeká na událost
    (uvolnění prostředku, zaslání zprávy),
    kterou vyvolá pouze některý procesů z P
  - ✓ prostředek – paměťový prostor, IO zařízení, soubor,...
- stárnutí
  - ✓ Požadavky 1 nebo více procesů z P nejsou dlouhodobě plněny
  - ✓ např. z důvodů priorit, prevence uváznutí apod.
  - ✓ dlouhodobě =
    déle než je akceptovatelné pro řešenou aplikační úlohu,
    bez záruky za splnění v konečném čase, ...

Postupový prostor s možností uváznutí



Postupový prostor bez možnosti uváznutí



Charakteristika uváznutí

K uváznutí dojde, když současně platí
4 následující podmínky (3x nutná, poslední postačující):

- vzájemné vyloučení, *Mutual Exclusion*
  - ✓ sdílený zdroj může v jednom okamžiku používat pouze jeden proces
- požadavky se uplatňují postupně, *Hold and Wait*
  - ✓ proces vlastnící alespoň jeden zdroj čeká na získání dalšího zdroje, dosud vlastněného jiným procesem
- nepřipouští se předbíhání, *No preemption*
  - ✓ zdroj lze uvolnit pouze procesem, který ho v dané době vlastní
- došlo k zacyklení požadavků, *Circular wait*:
  - ✓ existuje posloupnost n čekajících procesů $\{P_0, P_1, \ldots, P_{n-1}\}$ takových, že P_0 čeká na (uvolnění zdroje drženého) P_1, P_1 čeká (na uvolnění zdroje drženého) P_2, ... a P_{n-1} čeká (na uvolnění zdroje drženého) P_0.

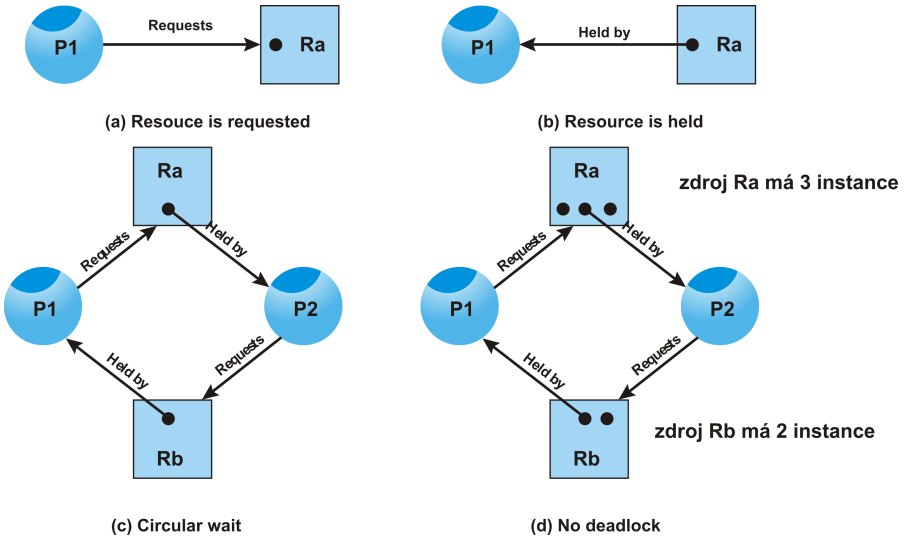
Použitý model

- Typy zdrojů $R_1, R_2, \ldots, R_m$
  - ✓ časové díly CPU, prostor paměti, I/O zařízení, ...
- Každý zdroj R_i má W_i instancí
- Každý proces používá zdroj podle následujícího schématu
  - ✓ žádost o přidělení zdroje, *Request* (*Get*)
  - ✓ použití zdroje (po konečnou dobu)
  - ✓ uvolnění zdroje, *Release*

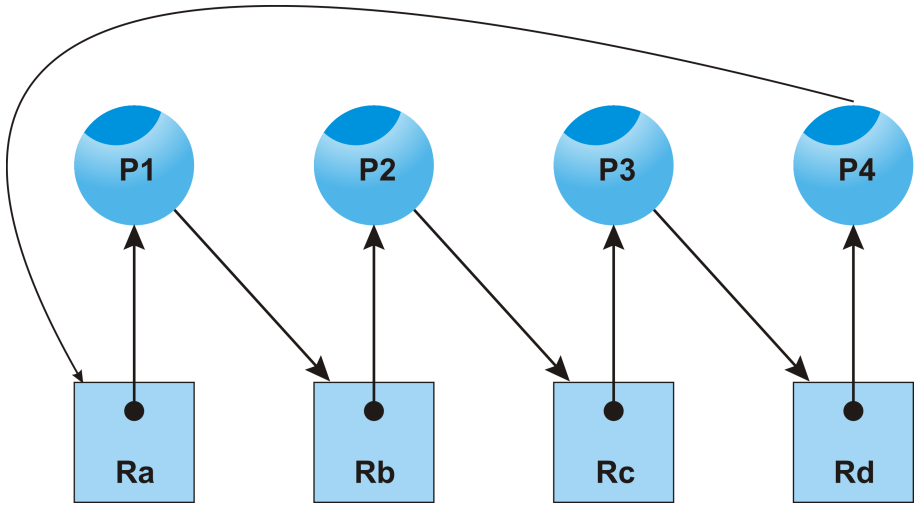
Graf přidělení zdrojů

- Resource-Allocation Graph, RAG
- množina uzlů V a množina hran E
- V se dělí do dvou typů:
  - ✓ $P = \{P_1, P_2, \ldots, P_n\}$, množina procesů existujících v systému
  - ✓ $R = \{R_1, R_2, \ldots, R_m\}$, množina zdrojů existujících v systému
- hrana požadavku – orientovaná hrana $P_i \rightarrow R_j$
- hrana přidělení – orientovaná hrana $R_i \rightarrow P_j$

Graf přidělení zdrojů, RAG

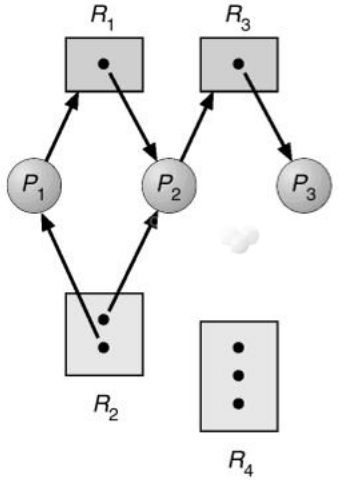


Příklad RAG – cyklické řetězení

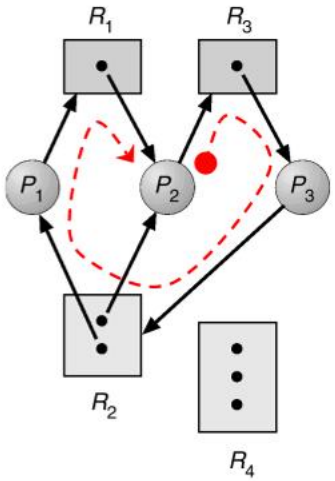


Příklady RAG

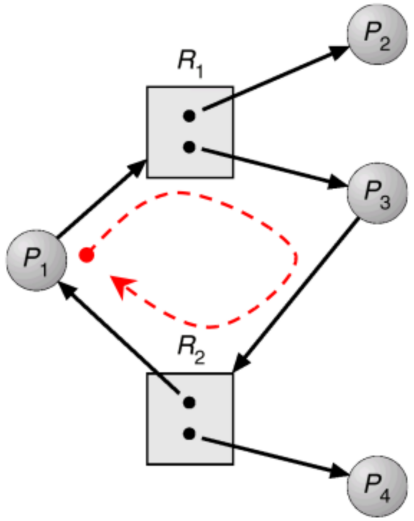
bez uváznutí



s uváznutím



Příklad RAG – s cyklem a bez uváznutí



Cyklus v RAG a uváznutí

- Jestliže se v RAG nevyskytuje cyklus – k uváznutí nedošlo
- Jestliže se v RAG vyskytuje cyklus,
  - a existuje pouze jedna instance zdroje daného typu – k uváznutí došlo
  - a existuje více instancí zdroje daného typu – k uváznutí může dojít

Wait-for-Graph, WFG

- Závislost pouze mezi procesy
- Uzel – proces,
- Orientovaná hrana $u \to v$,
  u čeká na uvolnění zdroje drženého v
- Systém procesů uvázl, pokud je ve WFG cyklus

Metody zvládnutí uváznutí

- Ochrana před uváznutím prevencí
  - ✓ systém se nikdy nedostane do stavu uváznutí, ruší se platnost některé nutné podmínky
  - ✓ apriorní eliminace některé nutné podmínky návrhovým rozhodnutím
- Obcházení uváznutí
  - ✓ detekce potenciální možnosti vzniku postačující podmínky uváznutí a nepřipuštění takového stavu
  - ✓ prostředek se nepřidělí, pokud by hrozilo uváznutí $\Longrightarrow$ hrozí stárnutí
- Detekce uváznutí a obnova po uváznutí
  - ✓ detekce uváznutí, např. analýzou RAG, a obnova stavu bez uváznutí
- Ignorování hrozby uváznutí
  - ✓ řešení uplatněné v univerzálních OS – Unix, Windows, ..

Metody zvládnutí uváznutí

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests

Metody zvládnutí uváznutí

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates online handling	•Inherent preemption losses

Ochrana před uváznutím prevencí

konzervativní politika – omezuje se způsob provedení požadavku

- možnosti
  - ✓ nepřímé metody – zneplatnění některé nutné podmínky
  - ✓ přímá metoda – nepřipuštění platnosti postačující podmínky (cyklus v grafu) uspořádáním pořadí přidělování prostředků
- nepřímé metody
  - ✓ nepoužívání sdílených zdrojů, virtualizace prostředků (spooling), . . . ne vzájemná výlučnost
  - ✓ kdykoliv proces něco požaduje, nesmí vlastnit žádný jiný prostředek, např. požaduje všechny prostředky najednou při svém vytvoření nebo uvolňování dosud držených prostředků před žádostí dalšího prostředku – ne postupné uplatňování požadavků
  - ✓ odebírání prostředků procesu správcem, zařazení procesů do čekací fronty, aktivace procesu při dostupnosti potřebných prostředků – připouští se předbíhání, preemption

Prevence uváznutí, nepřímé metody

- ne vzájemné vyloučení
  - ✓ aplikovatelné pro procesy užívající zdroje, které lze sdílet, např. používání sdílených zámků při čtení objektů v databázi
  - ✓ neaplikovatelné při používání opakovaně dostupných zdrojů, modifikace sdíleného objektu v databázi vyžaduje aplikaci exkluzivitního zámku
  - ✓ někdy lze řešit virtualizací prostředků, pokud to jde (tiskárny, . . .)
  - ✓ obecně neuplatnitelné řešení

Prevence uváznutí, nepřímé metody

- ne postupná alokace
  - ✓ proces žádající nějaký zdroj nesmí nevlastnit žádný jiný zdroj
  - ✓ proces musí požádat o všechny požadované zdroje a obdržet je dříve než se spustí jeho běh nebo o ně smí žádat pouze tehdy, když žádné zdroje nevlastní
  - ✓ důsledek
    - nízká efektivita využití zdrojů
    - možnost stárnutí

Prevence, nepřímé metody

- ne předbíhání
  - ✓ nechť proces držící nějaké zdroje požaduje přidělení dalšího zdroje, který mu nelze přidělit okamžitě (zdroj není volný)
  - ✓ správce prostředků např. odebere procesu všechny jím držené zdroje v tomto okamžiku, „odebrané“ zdroje zapíše do seznamu zdrojů, na které proces čeká a proces bude obnoven pouze když může získat jak jím původně držené zdroje, tak jím nově požadovaný zdroj
  - ✓ správce prostředků může některému z uvázlých procesů z důvodu čekání na jistý zdroj tento zdroj odebrat a uvolnit, proces uspí a obnoví později, až bude požadovaný zdroj uvolněný

Prevence, přímá metoda

- zabránění zacyklení pořadí
  - ✓ zavede se úplné uspořádání typů zdrojů a každý proces požaduje prostředky v pořadí daném vzrůstajícím pořadím výčtu

Problém uváznutí na příkladu transakcí

- Uvažme dvě transakce:

T_1:	T_2:
write (B);	*write* (A);
write (A);	*write* (B);

- a plán s uváznutím:

T_1:	T_2:
lock-X(B);	
write (B);	
	lock-X(A);
	write (A);
čekání na dokončení *lock-X*(A);	čekání na dokončení *lock-X*(B);

Prevence uváznutí souběžných transakcí

- každá transakce si zamkne všechna data dříve než se spustí
  - ✓ eliminace nutné podmínky uváznutí – *hold-and-wait*
  - ✓ velmi neefektivní řešení, zvláště pro distribuované prostředí nepoužitelné
- nebo se definuje se pořadí zamykání dat
  - ✓ eliminace postačující podmínky uváznutí ve *wait-for* grafu
  - ✓ aplikuje se grafový protokol řízení souběžnosti
  - ✓ nebo úplné uspořádání dat + 2PLP aplikovaný na toto pořadí
- nebo se připustí preempce transakce, která drží potřebné zdroje
  - ✓ eliminace nutné podmínky uváznutí *no preemption*

Prevence uváznutí souběžných transakcí

- preempce a vracení (roll-back) transakcí
  - ✓ jestliže T_2 požaduje zámek, který drží T_1, může TPM např.
    - zámek odbrat T_1,
    - T_1 vrátit (roll-back) a
    - zámek dát do držby T_2
  - ✓ pro rozhodování o preempci slouží rozhodovací časová razítka transakcí
    - tato slouží pouze pro rozhodování o čekání/vracení transakcí
    - data si transakce musí zamykat explicitně pomocí instrukce *lock*
  - ✓ vrácené transakci se rozhodovací časové razítko nemění, restartuje se s „původním" rozhodovacím časovým razítkem, takže postupně z hlediska rozhodování stárne a zvyšuje si tím prioritu z hlediska práva odebrat zámek
    - iniciální hodnotou rozhodovacího časového razítka je čas vzniku T

Prevence uváznutí na bázi TS

- předpokládá se možnost předbíhání procesů při použití prostředků (prostředky lze procesům odebírat)
- TS se přiděluje procesu při prvním vydání žádosti o zdroj a při případném návratu procesu zpět na začátek se nemění
- Každý proces P_i odvozuje z TS jedinečné prioritní číslo
- Prioritní čísla se používají při rozhodování, zda proces P_i bude čekat na proces P_j, nebo bude žádat o zdroj znovu, např.
  - ✓ P_i čeká na P_j má-li P_i větší prioritu (např. tím, že je starší) než P_j
  - ✓ pokud P_i čekat nebude (má nižší prioritu, např. tím, že je mladší), bude pak ve svém běhu vrácen před žádost a žádost bude opakovat
- Schéma chrání před uváznutím
  - ✓ Pro každou hranu $P_i \rightarrow P_j$ v grafu čekání (*wait-for graph*) platí, že P_i má vyšší prioritu než P_j. Cyklus vzniknout nemůže.
  - ✓ návraty ale představují hrozbu stárnutí

Prevence uváznutí na bázi TS, 2

- Hrozí stárnutí, procesy nízké priority se mohou stále vracet.
- Stárnutí lze řešit pomocí rozhodovacích časových razítek
  - ✓ Wait-Die Scheme, čekej nebo zemři (zemři = dělej se znovu)
    - starší T čeká na mladší T dokud tato neuvolní datovou položku
    - mladší T nečeká na starší T, vždy je vrácená (zemře) a a spouští se znovu, ale se zachováním věku (možná i vícekrát, ale postupně stárne a jednou bude nejstarší . . .)
  - ✓ Wound-Wait Scheme, poraň (ať se dělá někdo znovu) nebo čekej
    - starší T nečeká na mladší T dokud tato neuvolní datovou položku, raní ji (tj. prosadí její vrácení, ale se zachováním věku)
    - mladší T čeká na starší T dokud tato neuvolní datovou položku
- schéma Wait-Die Scheme může způsobovat více návratů než schéma Wound-Wait Scheme

Prevence uváznutí transakcí na bázi TS, Wait-Die Scheme

- Založeno na nepreemptivní technice
- Když se T_i iniciálně startuje, získá rozhodovací TS_i
- Nechť při uplatnění žádosti T_i s TS_i o jistý zámek drží tento zámek T_j s TS_j
  - ✓ je-li žádající T_i starší než okamžitý vlastník prostředku, – transakce T_j, $TS_i < TS_j$, T_i počká na uvolnění zámku a získá ho po té, jakmile jej držitel zámku, tj. transakce T_j, uvolní
  - ✓ je-li žádající T_i mladší než okamžitý držitel zámku – T_j, $TS_i > TS_j$, T_i je vrácená, žádost o přidělení opakuje, ale se stejným TS_i
- Takže platí – jestliže T_i požaduje zámek držený T_j, smí T_i čekat (*wait*) jen když $TS_i < TS_j$ (T_i je starší než T_j). Jinak je T_i vrácená v historii zpět na vydání žádosti (*dies*, zemře).

Prevence uváznutí transakcí na bázi TS, Wait-Die Scheme

- Wait-Die Scheme zajišťuje, že transakce může zemřít vícekrát, ale ne nekonečně krát, jednou přijde její čas, kdy si na uvolnění prostředku určitě počká, protože bude ze všech transakcí nejstarší
- Příklad:

  Transakce T_1, T_2 a T_3 mají TS s hodnotami postupně 5, 10, resp. 15.

  Jestliže T_1 ($TS_1 = 5$) požaduje zámek držený T_2 ($TS_2 = 10$), T_1 bude čekat na uvolnění zámku transakcí T_2

  Jestliže T_3 ($TS_3 = 15$) požaduje zámek držený transakcí T_2 ($TS_2 = 10$), pak T_3 zemře, tj. bude vrácená na zopakování žádosti o přidělení zámku, časové razítko se jí zachová a tak časem se stává starší a starší a má stále větší šanci si na uvolnění zámku počkat.

Prevence uváznutí transakcí na bázi TS, Wound-Wait Scheme

- Wound-Wait Scheme zajišťuje, že starší transakce nikdy nečeká na mladší transakci
- Schéma je založené na preemptivní technice
  - ✓ je-li žádající T_i starší než okamžitý vlastník prostředku – T_j , T_i odebere T_j zámek a T_j se v historii vrátí, bude zopakovat žádost o jeho přidělení, se stejným časovým razítkem (*roll-back*, *undo*)
  - ✓ je-li žádající T_i mladší než okamžitý vlastník prostředku – T_j, T_i čeká na uvolnění zámku a získá ho po té jakmile jej dosavadní držitel zámku T_j uvolní
- návratů bývá méně, čekání více
- obě schémata wait-die i wound-wait restartují transakce s původním rozhodovacím TS, což zabraňuje stárnutí

Prevence uváznutí transakcí na bázi TS, Wound-Wait Scheme

Příklad:

- Transakce T_1, T_2 a T_3 mají TS s hodnotami 5, 10, resp. 15
- Jestliže T_1 ($TS_1 = 5$) požaduje zámek držený transakcí T_2 ($TS_2 = 10$), přebere T_1 zámek od T_2, T_2 je ,,zraněná"transakcí T_1, je vrácená, ...
- Jestliže T_3 ($TS_3 = 15$) požaduje zámek držený transakcí T_2 ($TS_2 = 10$), transakce T_3 bude čekat na jeho uvolnění

Obcházení uváznutí

- Procesy mohou žádat o zdroje dynamicky
- Povoluje se existence všech tří nutných podmínek, zabraňuje se ale vzniku postačující podmínky
- Dynamicky se testuje, zda splnění požadavku na přidělení zdroje by v budoucnosti mohlo vést k pozdějšímu uváznutí
- Požaduje se tudíž znalost budoucího chování procesů z hlediska jimi požadovaných zdrojů
- Připouští se ale více souběžnosti než při prevenci
- Dva možné přístupy k obcházení
  - ✓ proces, jehož požadavky by mohly způsobit uváznutí, nespouštět
  - ✓ nesplnit dynamický požadavek procesu na zdroj, jestliže by splnění požadavku v budoucnosti mohlo vést k uváznutí

Obcházení uváznutí – řešení iniciálním rozhodnutím

- Musí řešit middleware (správce alokací / uváznutí) a platí:
  - ✓ Správce zná počty všech přidělovaných zdrojů,
    Resource vector, $\boldsymbol{R = (R_1, R_2, \ldots, R_m)}$
  - ✓ Správce si vede přehled o počtech dostupných,
    dosud nepřidělených instancích zdrojů,
    Available vector, $\boldsymbol{V = (V_1, V_2, \ldots, V_m)}$
  - ✓ Každý z n procesů deklaruje maxima svých požadavků
    na jednotlivé zdroje v matici $\boldsymbol{Max} = \begin{matrix} M_{11} & \ldots & M_{1m} \\ \ldots & \ldots & \ldots \\ M_{n1} & \ldots & M_{nm} \end{matrix}$
  - ✓ Správce uváznutí si vede přehled o počtech přidělených instancí zdrojů
    každému z n procesů v matici $\boldsymbol{Allocation} = \begin{matrix} A_{11} & \ldots & A_{1m} \\ \ldots & \ldots & \ldots \\ A_{n1} & \ldots & A_{nm} \end{matrix}$

Obcházení uváznutí – řešení iniciálním rozhodnutím

- ✓ Všechny zdroje musí být dostupné nebo přidělené,
  $\boldsymbol{R_j = V_j + \sum_{i=1}^{n} A_{ij}}$ pro všechna j
- ✓ Žádný proces nemůže požadovat více zdrojů, než v systému existuje
  $\boldsymbol{Max_{ij} \leq R_j}$, pro všechna i, j
- ✓ Žádný proces nemůže získat více zdrojů než deklaroval jako maximum
  $\boldsymbol{A_{ij} \leq Max_{ij}}$, pro všechna i, j

- Nový proces P_{n+1} se spustí pouze když lze uspokojit do výše maxim všechny dosud existující procesy i P_{n+1}
  $\boldsymbol{R_j \geq \sum_{i=1}^{n} Max_{ij} + Max_{(n+1)j}}$ pro všechna j
  - ✓ předpokládá se nejhorší případ –
    všechny procesy uplatní maxima požadavku současně

Obcházení uváznutí – řešení dynamickým rozhodováním

- tzv. Bankéřův algoritmus (řešený v middleware)
- Middleware musí mít nějaké dodatečné apriorní informace o požadavcích procesů na zdroje
- Model požaduje, aby každý proces udal maxima počtů prostředků každého typu, které může požadovat.
- Algoritmus řešící obcházení uváznutí dynamicky zkouší, zda stav systému přidělování zdrojů zaručuje, že se procesy v žádném případě nedostanou do cyklického čekání na sebe
- Stav systému přidělování zdrojů se definuje
  – počtem dostupných a přidělených zdrojů a
  – maximy žádostí procesů

Obcházení uváznutí, bezpečný stav

- Když proces požaduje přidělení dostupného prostředku, algoritmus musí rozhodnout, zda toto přidělení ponechá systém procesů v „bezpečném stavu“, ve kterém platí $\boldsymbol{Max_{ij} - A_{ij} \leq V_j}$, pro všechna j a i
- Systém procesů je v bezpečném stavu, jestliže existuje „bezpečná posloupnost procesů“
- Posloupnost procesů $\{P_1, P_2, \ldots, P_n\}$ je bezpečná, jestliže požadavky každého P_i lze uspokojit právě volnými zdroji a zdroji drženými všemi předchozími P_j , $j < i$.
  - ✓ Pokud nejsou zdroje požadované procesem P_i dostupné, pak P_i může vyčkat dokud se všechny předchozí P_j neukončí.
  - ✓ Poté P_i může získat všechny jím požadované zdroje a provést se a ukončit se a jemu přidělené zdroje se vrátí mezi dostupné zdroje.
  - ✓ Když skončí P_i, může své potřebné zdroje získat P_{i+1} atd.

Obcházení uváznutí, fakta

- Jestliže je systém procesů v bezpečném stavu (*safe state*), do stavu uváznutí (*deadlock*) nepřejde
- Jestliže se by se systém procesů dostal do stavu, který není bezpečný, (*unsafe state*), přechod do stavu uváznutí je hrozbou, stav uváznutí případně může nastat
- Obcházení
  - ✓ výkon správy přidělování zdrojů procesům zajišťující, že systém procesů se trvale nachází v bezpečném stavu, resp. nikdy nepřejde do stavu, který není bezpečný
  - ✓ nepovoluje se běh potenciálně nebezpečného procesu
  - ✓ neprovádí se potenciálně nebezpečné přidělení zdroje

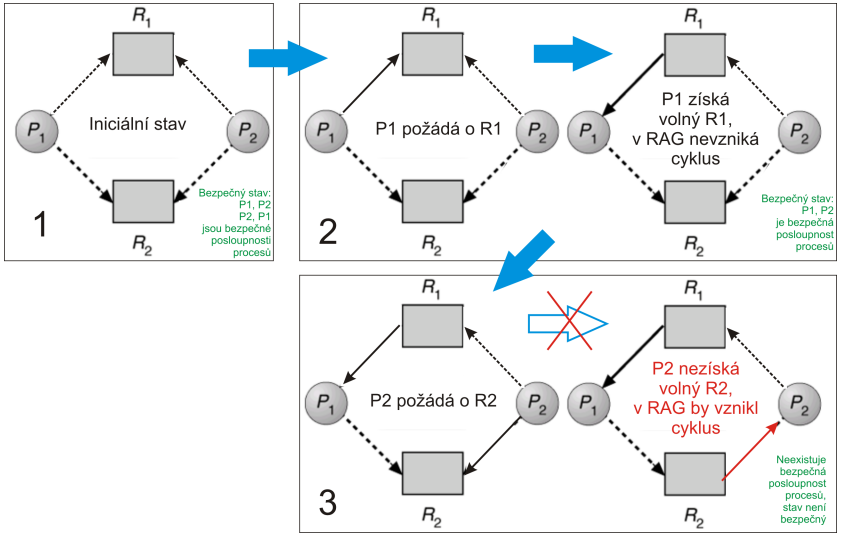
Ilustrace obcházení pomocí RAG

- Zdroje musí být nárokovány a priori – nárokovými hranami
- Nároková hrana $P_i \rightarrow R_j$ indikuje, že proces P_i může někdy v budoucnu požadovat zdroj R_j;
  - ✓ vede stejným směrem jako požadavková hrana
  - ✓ v RAG je zobrazována čárkovaně
- Nároková hrana $P_i \rightarrow R_j$ se konvertuje na požadavkovou hranu v okamžiku, kdy proces P_i požádá o zdroj R_j
  - ✓ vede stejným směrem jako nároková hrana
  - ✓ v RAG je zobrazována plnou čarou

Algoritmus RAG pro obcházení

- Když proces P_i zdroj získá, požadavková hrana se mění na hranu přidělení, $P_i \leftarrow R_j$
- Konverze požadavkové hrany na hranu přidělení nesmí v RAG vytvořit cyklus
- Když proces zdroj uvolní, hrana přidělení se mění na nárokovou hranu

Algoritmus RAG pro obcházení



Bankéřův algoritmus

- procesy
  - ✓ zákazníci, kteří si chtějí půjčovat peníze (prostředky) z banky, předem deklarují maximální výše úvěrů
  - ✓ úvěry v konečném čase zákazníci splácí
- bankéř nemůže úvěr poskytnout, pokud si není jistý, že může uspokojit všechny požadavky všech svých zákazníků alespoň v jednom pořadí uspokojení
- Algoritmus obcházení uváznutí aplikovatelný i při násobnosti instancí zdrojů
- Každý proces musí maxima používaných zdrojů deklarovat apriori, tj. deklarací ihned při svém vytvoření
- Proces požadující přidělení může být dán do stavu čekání

Bankéřův algoritmus, 2

- Proces získané zdroje v konečném čase uvolní
- Nikdy nedojde k uváznutí
  - ✓ proces vznikne jen tehdy, bude-li možno uspokojit všechny jeho požadavky (není důležité v jakém pořadí)
  - ✓ požadavek na přidělení se uspokojí jen tehdy, bude-li možno uspokojit všechny požadavky všech procesů (alespoň v jednom pořadí)
- Jde o sub-optimální strategii
  - ✓ předpokládá se nejhorší případ, tj. předpokládá se, že všechny procesy si vyžádají deklarovaná maxima

Datové struktury bankéřova algoritmu

- n – počet procesů,
- m – počet typů zdrojů
- $Available$ –
  - ✓ Vektor délky m
  - ✓ Jestliže platí $Available[j] = k$, pak je dostupných k instancí typu R_j
- Max –
  - ✓ matice $n \times m$
  - ✓ Jestliže platí $Max[i,j] = k$, (jinak zapsáno $Max_i[j] = k$, vektor maxim procesu i) pak proces P_i může požadovat nejvýše k instancí zdroje typu R_j
  - ✓ vektor maxim $Max_i[j] = k, j = 1, \ldots, m$ je povinnou deklarací v procesu P_i

Datové struktury bankéřova algoritmu, 2

- $Allocation$ –
  - ✓ matice $n \times m$
  - ✓ Jestliže platí $Allocation[i,j] = k$, (jinak zapsáno $Allocation_i[j] = k$, vektor přidělení procesu i) pak proces P_i má v tomto okamžiku přiděleno k instancí zdroje typu R_j
- $Need$ –
  - ✓ matice $n \times m$
  - ✓ Jestliže platí $Need[i,j] = k$, (jinak zapsáno $Need_i[j] = k$, vektor možných požadavků procesu i) pak proces P_i může pro ukončení svého úkolu potřebovat ještě k instancí zdroje typu R_j
  - ✓ Obsah matice $Need$ je definován jako $Max - Allocation$
- $Request[i,j]$ –
  - ✓ (jinak zapsáno $Request_i[j] = k$, vektor okamžitých požadavků procesu i)
  - ✓ Jestliže platí $Request_i[j] = k$, pak proces P_i požaduje k instancí zdroje typu R_j.

Algoritmus testu stavu systému procesů

1. Nechť $Work$ a $Finish$ jsou pomocné vektory délek m a n.
   Inicializace:
   $Work = Available$; $Finish[i] = false$ pro $i = 1, 2, \ldots, n$
2. Nalezni i takové, že platí obě následující podmínky:
   (a) $Finish[i] = false$ (b) $Need[i] \leq Work$
   Pokud neexistuje, pokračuj krokem 4
3. simulace ukončení:
   $Work = Work + Allocation[i]$; $Finish[i] = true$
   pokračuj krokem 2
4. Pokud platí
   $Finish[i] = true$ pro všechna i,
   potom je systém ve stavu bezpečný,
   v opačném případě není ve stavu bezpečný

Algoritmus vyžádání zdroje pro proces P_i

1. Pokud platí $Request[i, j] \leq Need[i, j]$ pokračuj krokem 2.
   V opačném případě oznam chybu – proces překročil deklarované maximum
2. Pokud platí $Request[i, j] \leq Available[j]$, pokračuj krokem 3.
   V opačném případě P_i musí čekat, není dostupná instance zdroje
3. Simuluje se přidělení požadovaného zdroje P_i změnou stavu:
   $Available[j] = Available[j] - Request[i, j]$;
   $Allocation[i, j] = Allocation[i, j] + Request[i, j]$;
   $Need[i, j] = Need[i, j] - Request[i, j]$;
   Provede se test stavu systému.
   Pokud je stav bezpečný, pak se požadované zdroje procesu P_i přidělí, pokud není stav bezpečný, pak P_i musí čekat a zachová se původní stav přidělení zdrojů

Příklad 1 bankéřova algoritmu

- pět procesů P0 až P4
- tři typy zdrojů –
  A (10 instancí), B (5 instancí) a C (7 instancí)
- Momentka stavu v čase t :

	Allocation	*Max*	*Need*	*Available*
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3	3 3 2
P1	2 0 0	3 2 2	1 2 2	
P2	3 0 2	9 0 2	6 0 0	
P3	2 1 1	2 2 2	0 1 1	
P4	0 0 2	4 3 3	4 3 1	

- Systém je v bezpečném stavu, poněvadž např. posloupnost
  P1, P3, P4, P2, P0 splňuje kritéria bezpečnosti

Příklad 1 bankéřova algoritmu

- ukončil se P1

	Allocation	*Max*	*Need*	*Available*
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3	5 3 2
P2	3 0 2	9 0 2	6 0 0	
P3	2 1 1	2 2 2	0 1 1	
P4	0 0 2	4 3 3	4 3 1	

- ukončil se P3

	Allocation	*Max*	*Need*	*Available*
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3	7 4 3
P2	3 0 2	9 0 2	6 0 0	
P4	0 0 2	4 3 3	4 3 1	

Příklad 1 bankéřova algoritmu

- ukončil se P4

	Allocation	*Max*	*Need*	*Available*
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3	7 4 5
P2	3 0 2	9 0 2	6 0 0	

- ukončil se P2

	Allocation	*Max*	*Need*	*Available*
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3	10 4 7

- ukončil se P0 – stav je bezpečný

Allocation	*Max*	*Need*	*Available*
A B C	A B C	A B C	A B C
			10 5 7

Příklad 1 bankéřova algoritmu

- Nechť P1 v čase t požaduje $(1, 0, 2)$
- kontrola $\boldsymbol{Request}_1 \leq \boldsymbol{Need}_1$, $(1, 0, 2) \leq (1, 2, 2)$ je $\boldsymbol{true}$.
- kontrola $\boldsymbol{Request}_1 \leq \boldsymbol{Available}$, $(1, 0, 2) \leq (3, 3, 2)$ je $\boldsymbol{true}$.
- provedení kontroly stavu ukáže, že podmínku bezpečnosti splňuje posloupnost procesů P1, P3, P4, P0, P2:

  Momentka stavu v čase t :

	Allocation	*Max*	*Need*	*Available*
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3	3 3 2
P1	2 0 0	3 2 2	1 2 2	
P2	3 0 2	9 0 2	6 0 0	
P3	2 1 1	2 2 2	0 1 1	
P4	0 0 2	4 3 3	4 3 1	

Příklad 1 bankéřova algoritmu

- Simulované přidělení procesu P1 v čase t :

	Allocation	*Max*	*Need*	*Available*
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3	2 3 0
P1	3 0 2	3 2 2	0 2 0	
P2	3 0 2	9 0 2	6 0 0	
P3	2 1 1	2 2 2	0 1 1	
P4	0 0 2	4 3 3	4 3 1	

- ukončil se P1 :

	Allocation	*Max*	*Need*	*Available*
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3	5 3 2
P2	3 0 2	9 0 2	6 0 0	
P3	2 1 1	2 2 2	0 1 1	
P4	0 0 2	4 3 3	4 3 1	

Příklad 1 bankéřova algoritmu

- ukončil se P3 :

	Allocation	*Max*	*Need*	*Available*
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3	7 4 3
P2	3 0 2	9 0 2	6 0 0	
P4	0 0 2	4 3 3	4 3 1	

- ukončil se P4 :

	Allocation	*Max*	*Need*	*Available*
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3	7 4 5
P2	3 0 2	9 0 2	6 0 0	

Příklad 1 bankéřova algoritmu

- ukončil se P0 :

Allocation	*Max*	*Need*	*Available*
A B C	A B C	A B C	A B C
P2 3 0 2	9 0 2	6 0 0	7 5 5

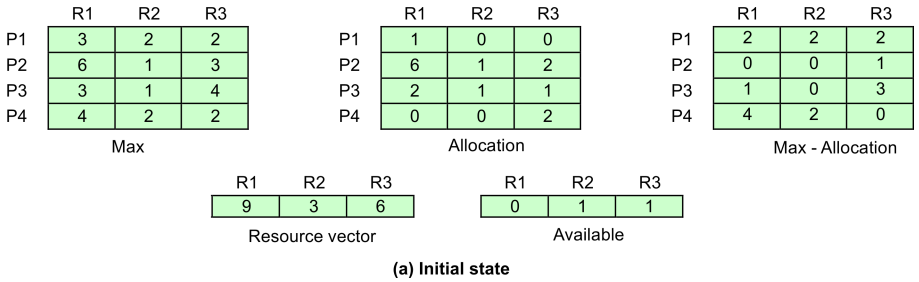
- ukončil se P2 – stav je bezpečný:

Allocation	*Max*	*Need*	*Available*
A B C	A B C	A B C	A B C
			10 5 7

- požadavek P1 na přidělení $(1, 0, 2)$ se uspokojí

Příklad 2 bankéřova algoritmu, bezpečný stav

- Iniciální stav prověřovaný zda je bezpečným stavem

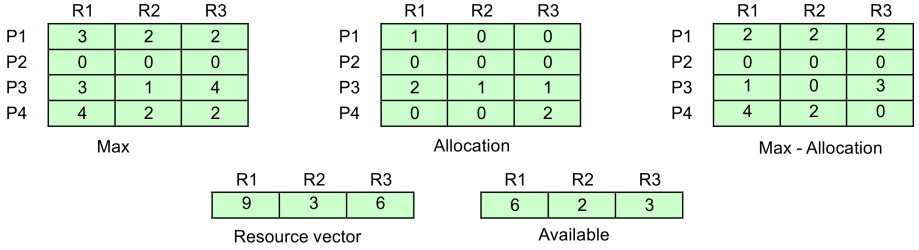


(a) Initial state

- Ukončit se může proces P2

Příklad 2 bankéřova algoritmu, bezpečný stav

- Ukončil se proces P2

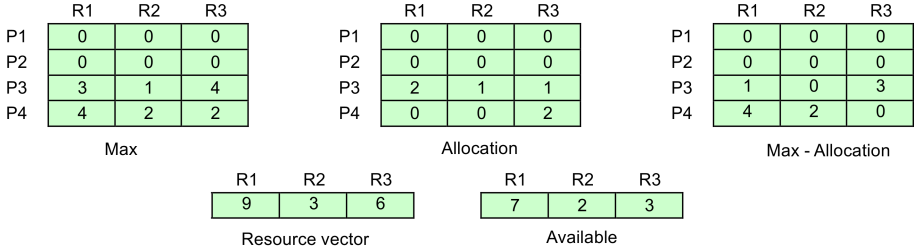


(b) P2 runs to completion

- Ukončit se může kterýkoliv proces z P1, P2 a P3
- Ukončí se např. proces P1

Příklad 2 bankéřova algoritmu, bezpečný stav

- Ukončil se proces P1

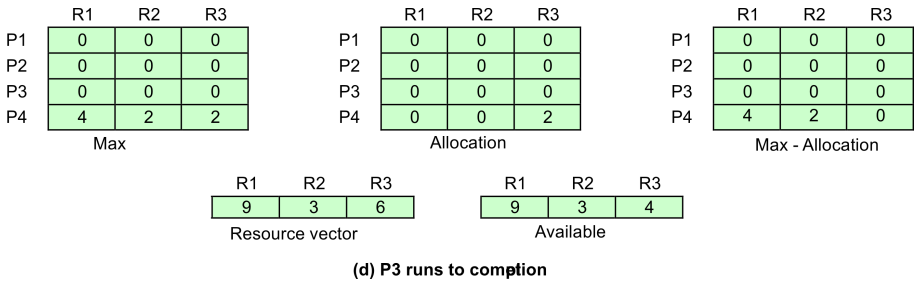


(c) P1 runs to completion

- Ukončit se může kterýkoliv proces z P2 a P3
- Ukončí se např. proces P3

Příklad 2 bankéřova algoritmu, bezpečný stav

- Ukončil se proces P3

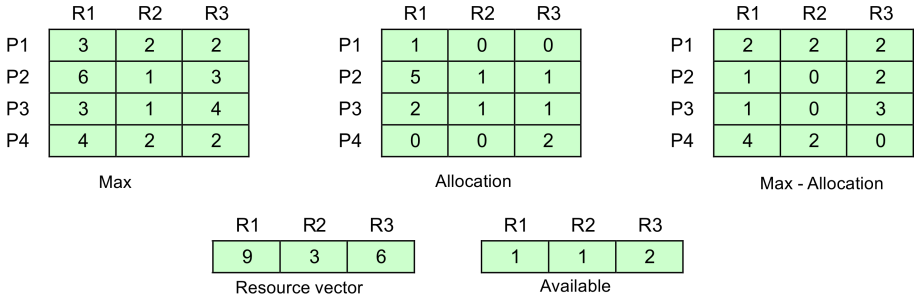


(d) P3 runs to compltion

- Ukončit se může proces P4
- Iniciální stav byl bezpečný, ukončit se mohly všechny procesy

Příklad 3 bankéřova algoritmu

- Iniciální stav

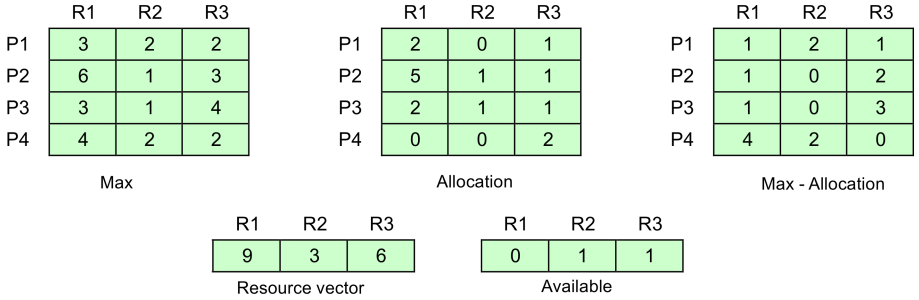


(a) Initial state

- ✓ Kdyby nyní proces P2 požádal o zdroj R1 a R3, výsledkem je inicialní stav příkladu 2 a ten je bezpečný
- ✓ Kdyby nyní proces P1 požádal o zdroj R1 a R3, pak výsledný stav je na následujícím obrázku

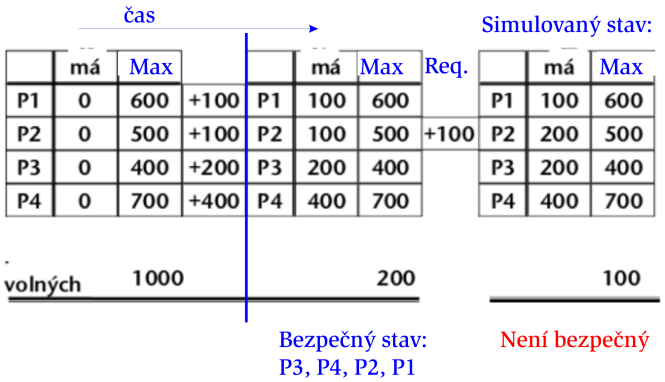
Příklad 3 bankéřova algoritmu

- Tento stav bezpečný není, každý proces může požádat o R1 a ten dostupný není
- Žádost procesu P1 z předchozího obrázku se musí zamítnout mohlo by nastat uváznutí



(b) P1 requests one unit each of R1 and R3

Příklad 4 bankéřova algoritmu



- Požadovaná alokace 100 pro P2 se odmítne

Bankéřův algoritmus – poznámky

- běhy procesů z bezpečného stav nezpůsobí uváznutí
- běhy procesů ze stavu, který není bezpečný, ještě nemusí způsobit uváznutí
- všechny algoritmy obcházení uváznutí předpokládají, že procesy jsou nezávislé, že se synchronizačně neomezují
- obcházení uváznutí zamezí uváznutí, nemusí ale zamezit stárnutí

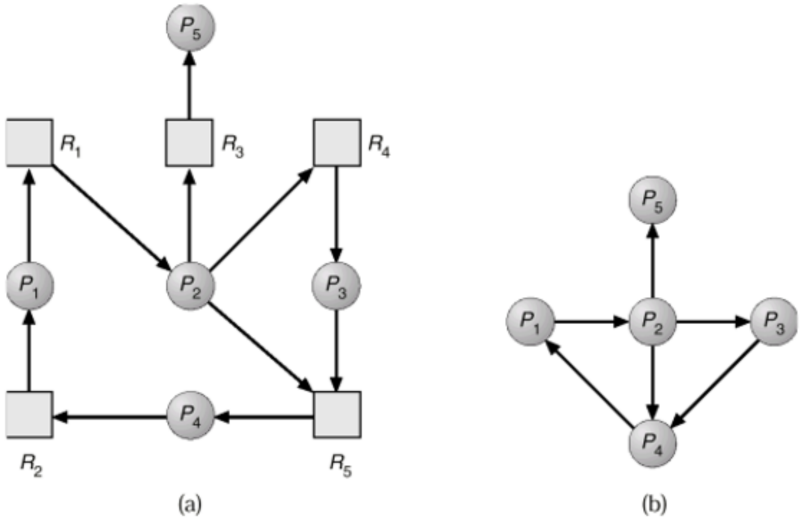
Detekce uváznutí

- umožňuje se, aby se systém uváznul
- provede se algoritmus detekce uváznutí – hledání cyklů
- aplikuje se plán obnovy
  - ✓ Násilně se ukončuje jednotlivě proces po procesu, dokud se neodstraní cyklus
- Čím je dáno pořadí násilného ukončení?
  - ✓ priorita procesu
  - ✓ doba běhu procesu, doba potřebná k ukončení procesu
  - ✓ prostředky, které proces použil
  - ✓ prostředky, které proces potřebuje k ukončení
  - ✓ počet procesů, které bude potřeba ukončit
  - ✓ Je proces interaktivní nebo dávkový?
  - ✓ . . .

Detekce uváznutí

- Algoritmus detekce pro případ 1 instance prostředku každého typu
  - ✓ Udržuje se graf čekání, uzly jsou procesy, $P_i \rightarrow P_j$ jestliže P_i čeká na P_j
  - ✓ Periodicky se provádí algoritmus, který v grafu čekání hledá cykly
  - ✓ Algoritmus pro detekci cyklu v grafu požaduje provedení n^2 operací, kde n je počet hran v grafu

Graf přidělení zdrojů, RAG a graf čekání



Případ m instancí prostředku každého typu

- $\boldsymbol{Available}$
  - ✓ Vektor délky $\boldsymbol{m}$
  - ✓ indikuje počet dostupných instancí prostředků každého typu
- $\boldsymbol{Allocation}$
  - ✓ Matice $\boldsymbol{n \times m}$
  - ✓ indikuje počet instancí prostředků každého typu přidělených každému procesu
- $\boldsymbol{Request}$
  - ✓ Matice $\boldsymbol{n \times m}$
  - ✓ indikuje okamžité požadavky každého procesu
  - ✓ Jestliže platí $\boldsymbol{Request[i,j] = k}$, pak proces $\boldsymbol{P_i}$ požaduje $\boldsymbol{k}$ dalších instancí typu $\boldsymbol{R_j}$.

Algoritmus detekce

1. Nechť $\boldsymbol{Work}$ a $\boldsymbol{Finish}$ jsou pracovní vektory délek $\boldsymbol{m}$ a $\boldsymbol{n}$.
   Inicializace:
   – $\boldsymbol{Work = Available}$
   – pro $\boldsymbol{i = 1, 2, \ldots, n}$: je-li $\boldsymbol{Allocation[i] \neq 0}$,
   pak $\boldsymbol{Finish[i] = false}$, jinak $\boldsymbol{Finish[i] = true}$
2. Nalezni takové $\boldsymbol{i}$,že :
   (a) $\boldsymbol{Finish[i] = false}$
   (b) $\boldsymbol{Request[i] \leq Work}$
   jestliže takové $\boldsymbol{i}$ neexistuje, krok 4.
3. $\boldsymbol{Work = Work + Allocation[i]; Finish[i] = true}$
   pokračuj krokem 2.
4. Jestliže pro nějaké $\boldsymbol{i}$, $\boldsymbol{1 \leq i \leq n}$ platí $\boldsymbol{Finish[i] = false}$,
   pak v sytému ex. uváznutí. Uvázly $\boldsymbol{P_i}$ s $\boldsymbol{Finish[i] = false}$

Algoritmus požaduje pro detekci uváznutí $\boldsymbol{O(mn^2)}$ operací.

Příklad 1 detekce uváznutí

- pět procesů P0 až P4,
- tři typy zdrojů – A (7 instancí), B (2 instancí) a C (6 instancí)
- Momentka stavu v T0 :

	Allocation	*Request*	*Available*
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

- V systému procesů neexistuje uváznutí, posloupnost P0, P2, P3, P1, P4 končí s $\boldsymbol{Finish[i] = true}$ pro všechna $\boldsymbol{i}$.

Příklad 1 detekce uváznutí, 2

- P2 požaduje další exemplář zdroje typu $\boldsymbol{C}$:

	Allocation	*Request*	*Available*
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 1	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

- Jaký je stav systému?
  - ✓ I když se uvolní zdroje držené procesem P0, požadavky procesů P1, P2, P3, a P4 se neuspokojí
  - ✓ Systém uvázl, uváznutí se týká procesů P1, P2, P3, a P4

Příklad 2 detekce uváznutí

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Allocation vector

- Jaký je stav systému?
  - ✓ Dostupný je pouze jeden zdroj R5
  - ✓ Všechny procesy požadují více zdrojů než jeden R5
  - ✓ Systém uvázl, uváznutí se týká všech procesů P1, P2, P3, a P4

Použitelnost algoritmu detekce

- Kdy, jak často algoritmus detekce uváznutí vyvolávat?
  - ✓ zdroje držené uváznutými procesy jsou nedostupné do zrušení uváznutí
- Vždy, když nelze uspokojit požadavek na přidělení
  - ✓ detekuje se proces způsobující uváznutí a uvázlé procesy
  - ✓ jeden proces může způsobit více cyklů v RAG
  - ✓ zvyšuje se systémová režie uváznutí
- Algoritmus detekce uváznutí se vyvolává náhodně/cyklicky
  - ✓ v grafu čekání může existovat více cyklů a nelze určit, který z mnoha procesů uváznutí ,,způsobil“

Obnova po detekci – ukončení procesu

a. Násilné ukončení všech uvázlých procesů

b. Násilně se ukončuje jednotlivě proces po procesu, dokud se neodstraní cyklus

- Čím je dáno pořadí násilného ukončování?
- heuristiky:
  - ✓ priorita procesu
  - ✓ doba běhu procesu, doba potřebná k ukončení procesu
  - ✓ prostředky, které proces použil
  - ✓ prostředky, které proces potřebuje k ukončení
  - ✓ počet procesů, které bude potřeba ukončit
  - ✓ Je proces interaktivní nebo dávkový?

Obnova po detekci – přerozdělení prostředků

Problémy:

- Výběr oběti – minimalizace ceny – co je cenou ???
- Návrat zpět (Rollback) – návrat procesu do některého bezpečného stavu, restart procesu z tohoto stavu (checkpoints)
- Stárnutí – některý proces může být vybírán jako oběť trvale, i když se bude do cenového kritéria počítat počet návratů

Kombinovaná ochrana před uváznutím

- Kombinují se tři základní přístupy
  - ✓ prevence
  - ✓ obcházení
  - ✓ detekce a obnova
- optimalizovaný přístup k ochraně pro každý typ zdroje
- dělení zdrojů do hierarchických tříd
  - ✓ použije se prevence cyklického čekání na sebe mezi třídami prostředků
- použití nejvhodnější techniky pro zvládání uváznutí v každé třídě

Kombinovaná ochrana před uváznutím, příklad

- Kategorie zdrojů z hlediska pořadí přidělování
  - ✓ oblast na disku pro odkládání procesu
  - ✓ zdroje procesů – periferie typu disky, pasky, soubory
  - ✓ oblast hlavní / operační paměť
  - ✓ I/O kanály
- oblast na disku pro odkládání procesu
  - ✓ prevence přidělením prostoru při zahájení
  - ✓ lze uplatnit i obcházení
- zdroje procesů – periferie typu disky, pasky, soubory
  - ✓ lze uplatnit obcházení
  - ✓ lze použít prevenci stanovením pořadí požadavků
- oblast hlavní / operační paměť
  - ✓ prevence odebíráním zdroje (preempce)
- I/O kanály
  - ✓ prevence stanovením pořadí požadavků

Problém uváznutí transakcí, ochrana časovými limity

- časový limit, time-out
- transakce čekající na zámek se po uplynutí časového limitu vrací a restartuje
- pokud byly některé transakce uvázlé, návraty se uváznutí ruší
- snadná implementace
- obtížný odhad délky čekání
- možnost stárnutí
- vhodné pro prostředí s krátkými transakcemi a s pravděpodobným dlouhým čekáním při uváznutí
- velmi omezená použitelnost

Problém uváznutí transakcí, detekce uváznutí a obnova

- analogické řešení jako v případě procesů
  - ✓ udržuje se *wait-for-graph*
    - uzly – transakce
    - orientované hrany – která transakce čeká na kterou transakci
  - ✓ ve vhodných intervalech se řeší detekce cyklu ve WfG
  - ✓ co jsou to ,,vhodné intervaly''?
    - volbu ovlivňuje jak často dochází k uváznutí
    - volbu ovlivňuje kolik transakcí může být ovlivněno uváznutím ?
  - ✓ je nutné řešit problém výběru oběti pro vrácení
    - všechny ?
    - jen některé až do zrušení cyklu ve WfG ? (Problém ocenění.)
    - problém stárnutí

Ignorování uváznutí

- ,,pštrosí" politika
  - ✓ Matematici – najdou si totálně neuskutečnitelné uváznutí a říkají, že se mu musí za každou cenu zabránit prevencí
  - ✓ Inženýři – otáží se na závažnost a odmítnou zbytečně plýtvat energií na řešení nepodstatného problému:
    - Řešení v univerzálních operačních systémech (Unixový přístup):
      - uživatelé preferují možnost řídkého výskytu uváznutí před omezením, že by jejich procesy dynamicky žádající zdroje byly řešeny totální serializací
    - Pokud řešení v univerzálních operačních systémech nevyhovuje, musí systém aplikačních procesů řešit uváznutí v middleware