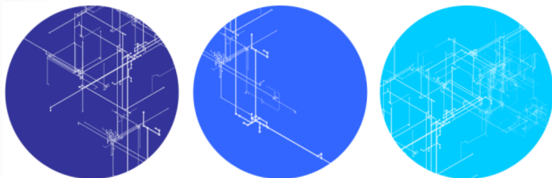
PB153
Operační systémy a jejich rozhraní



Uváznutí

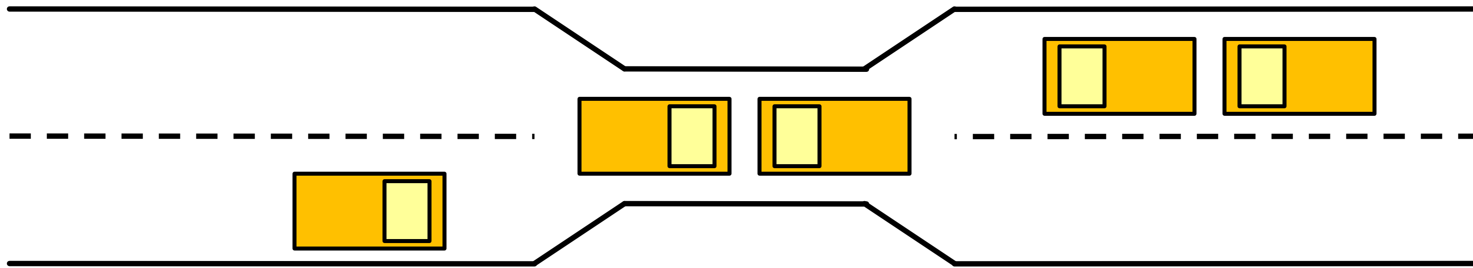
09

PROBLÉM UVÁZNUTÍ

- Existuje množina blokovaných procesů, každý proces vlastní nějaký prostředek (zdroj) a čeká na zdroj držený jiným procesem z této množiny
- Příklad 1
  - v systému existují 2 páskové mechaniky
  - procesy P_1 a P_2 chtějí kopírovat data z pásky na pásku, každý z procesů „vlastní“ jednu mechaniku a požaduje alokací druhé
- Příklad 2
  - Semafory A a B, inicializované na 1

P_0	P_1
wait (A);	*wait(B)*
wait (B);	*wait(A)*

PŘÍKLAD: ÚZKÝ MOST



- Most s jednosměrným provozem
- Každý vjezd mostu lze chápat jako zdroj
- Dojde-li k uváznutí, lze ho řešit tím, že se jedno auto vrátí
  - Preempce zdroje (přivlastnění si zdroje, který vlastnil někdo jiný) a vrácení soupeře do situace před žádostí o přidělení zdroje (preemption a rollback)
- Při řešení uváznutí se může vracet i více vozů
- Může docházet ke stárnutí

ANIMACE ÚZKÉHO MOSTU

DEFINICE UVÁZNUTÍ A STÁRNUTÍ

- Uváznutí
  - množina procesů P uvázla, jestliže každý proces P_i z P čeká na událost (uvolnění prostředku, zaslání zprávy), kterou vyvolá pouze některý z procesů P
- Stárnutí
  - požadavky 1 nebo více procesů z P nebudou splněny v konečném čase
    - z důvodů vyšších priorit jiného procesu
    - z důvodů prevence uváznutí apod.

MODEL

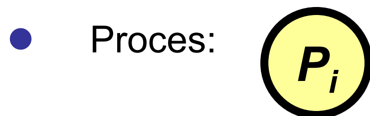
- Typy zdrojů R_1, R_2, …, R_m
  - tiskárna, paměť, I/O zařízení, …
- Každý zdroj R_i má W_i instancí
- Každý proces používá zdroj následujícím způsobem
  1. žádost
  2. použití
  3. uvolnění (v konečném čase)

CHARAKTERISTIKA UVÁZNUTÍ

- K uváznutí dojde, když začnou současně platit 4 následující podmínky
  - vzájemné vyloučení (mutual exclusion)
    - sdílený zdroj může v jednom okamžiku používat pouze jeden proces
  - ponechání si zdroje a čekání na další (hold and wait)
    - proces vlastnící alespoň zdroj čeká na získání dalšího zdroje, dosud vlastněného jiným procesem
  - bez předbíhání (no preemption)
    - zdroj lze uvolnit pouze procesem, který ho vlastní, dobrovolně po té, co daný proces zdroj dále nepotřebuje
  - kruhové čekání (circular wait)
    - existuje takový seznam čekajících procesů (P_0, P_1, …, P_n), že P_0 čeká na uvolnění zdroje drženého P_1, P_1 čeká na uvolnění zdroje drženého P_2, …, P_{n-1}čeká na uvolnění zdroje drženého P_n, a P_n čeká na uvolnění zdroje drženého P_0

GRAF PŘIDĚLENÍ ZDROJŮ

- Resource-Allocation Graph, RAG
- Množina uzlů V a množina hran E
- uzly jsou 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 $R_j \rightarrow P_i$
- Proces:

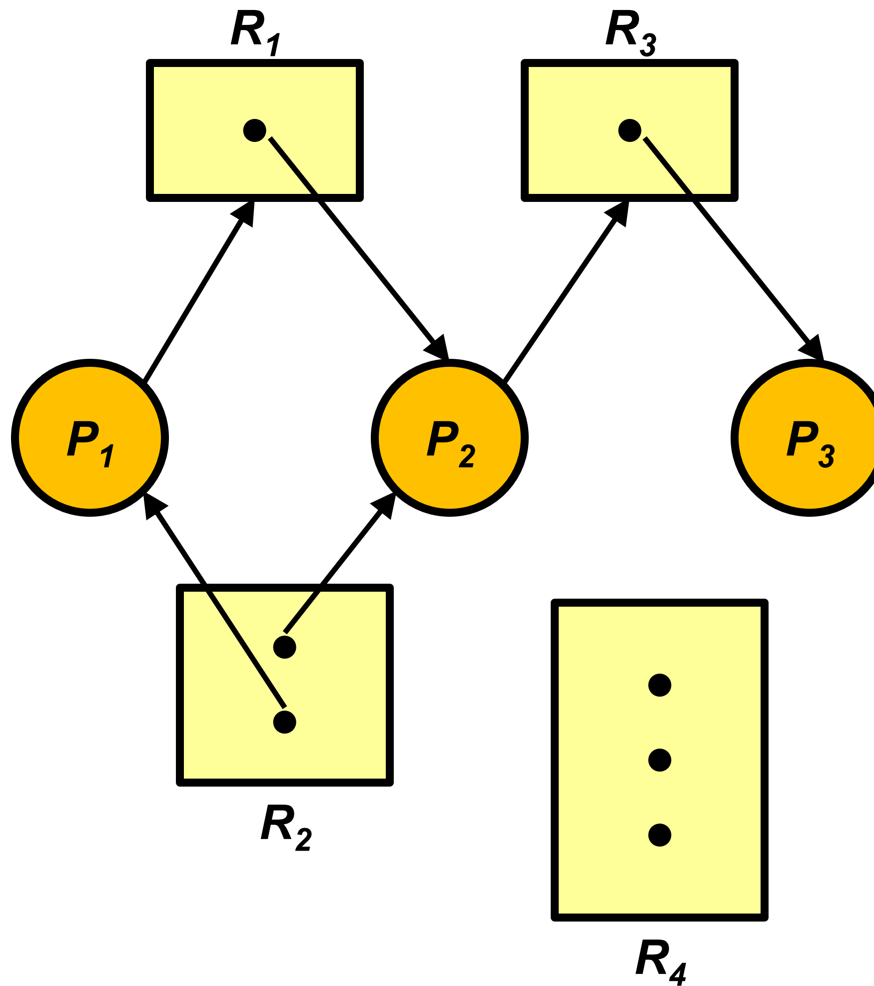


- Zdroj se 4 instancemi:

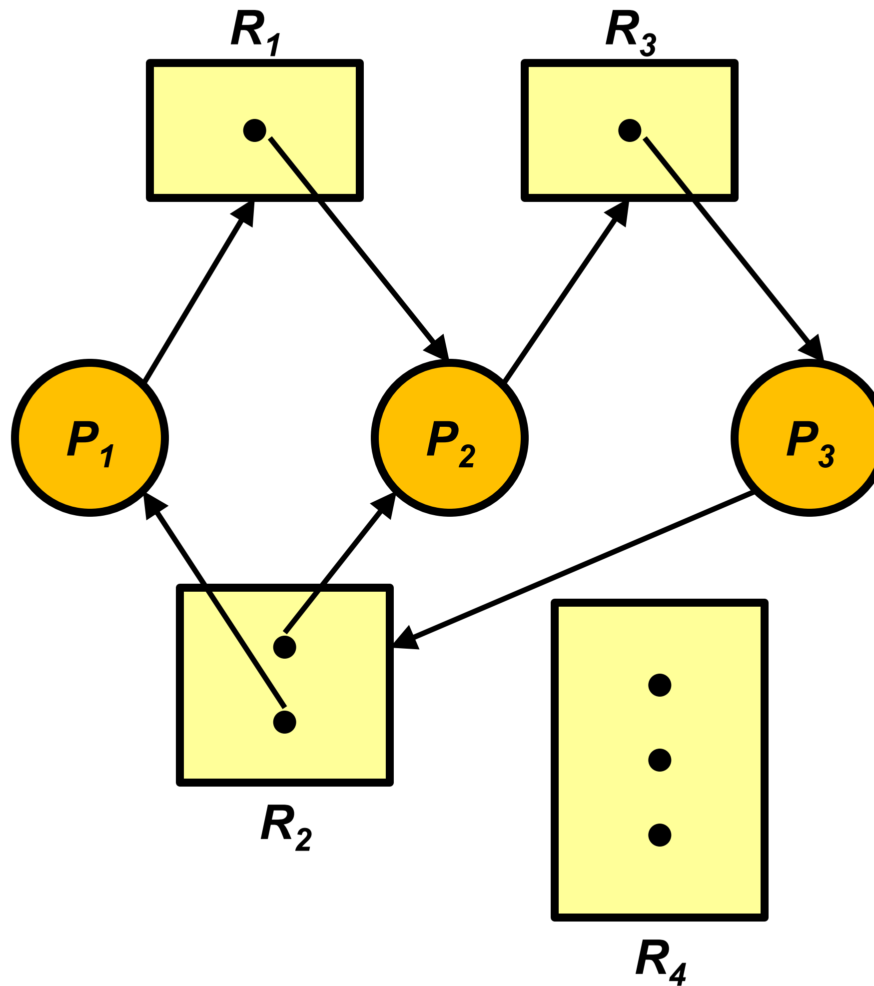
- Proces P_i požadující prostředek R_j:

- Proces P_i vlastnící prostředek R_j :

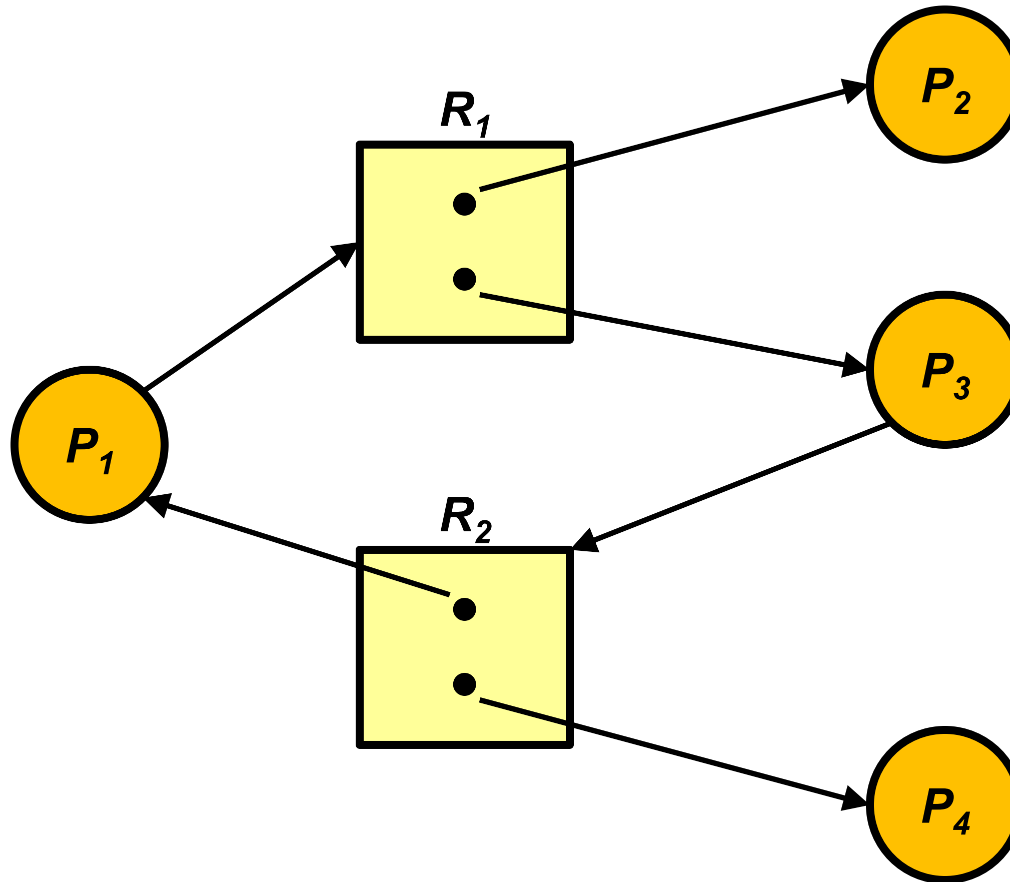
PŘÍKLAD RAG (BEZ CYKLU)



PŘÍKLAD RAG (S UVÁZNUTÍM)



PŘÍKLAD RAG (BEZ UVÁZNUTÍ)



RAG: ZÁVĚRY

- Jestliže se v RAG nevyskytuje cyklus – k uváznutí nedošlo
- Jestliže se v RAG vyskytuje cyklus
  - existuje pouze jedna instance zdroje daného typu → k uváznutí došlo
  - existuje více instancí zdroje daného typu → k uváznutí může (ale nemusí) dojít

PROBLÉM UVÁZNUTÍ

- Ochrana před uváznutím prevencí
  - zajistíme, že se systém nikdy nedostane do stavu uváznutí
  - zrušíme platnost některé nutné podmínky
- Obcházení uváznutí
  - detekce potenciální možnosti vzniku uváznutí a nepřipuštění takového stavu
  - zamezujeme současné platnosti všech nutných podmínek
  - prostředek se nepřidělí, pokud by hrozilo uváznutí (hrozí stárnutí)
- Obnova po uváznutí
  - uváznutí povolíme, ale jeho vznik detekujeme a řešíme
- Ignorování hrozby uváznutí
  - uváznutí je věc aplikace ne systému
  - způsob řešení zvolený většinou OS

OCHRANA PREVENCÍ

- Nepřímé metody
  - zneplatnění některé nutné podmínky
    - Virtualizací prostředků, ruším nutnost vzájemné výlučnosti při přístupu
    - požadováním všech prostředků najednou
    - odebíráním prostředků
- Přímé metody
  - nepřipuštění platnosti postačující podmínky (cyklus v grafu)
    - uspořádání pořadí vyžadování prostředků

PREVENCE UVÁZNUTÍ (1)

- Vzájemné vyloučení
  - podmínka není nutná pro sdílené zdroje
  - u nesdílených zdrojů musí podmínka platit
  - řeší se např. virtualizací prostředků (např. tiskárny)
- Ponechání zdrojů a čekání na další
  - při žádosti o zdroje proces žádné zdroje „vlastnit“ nesmí
  - proces musí požádat o zdroje a obdržet je dříve než je spuštěn běh procesu
  - důsledkem je nízká efektivita využití zdrojů a možnost stárnutí

PREVENCE UVÁZNUTÍ (2)

- Zakázané předbíhání
  - jestliže proces držící nějaké zdroje a požadující přidělení dalšího zdroje, nemůže zdroje získat okamžitě, pak se uvolní všechny tímto procesem držené zdroje
  - „odebrané“ zdroje se zapíší do seznamu zdrojů, na které proces čeká
  - proces bude obnoven, pouze jakmile může získat jak jím původně držené zdroje, tak jím nově požadované zdroje
- Zabránění kruhovému pořadí
  - zavedeme úplné uspořádání typů zdrojů a každý proces bude žádat o prostředky v pořadí daném vzrůstajícím pořadí výčtu

OBCHÁZENÍ UVÁZNUTÍ

- Systém musí mít nějaké dodatečné apriorní informace
- Nejjednodušší a nejužitečnější model požaduje, aby každý proces udal maxima počtu 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 cyklické fronty čekání
- Stav systému přidělování zdrojů se definuje počtem dostupných a přidělených zdrojů a maximem žádostí procesů

DETEKCE UVÁZNUTÍ

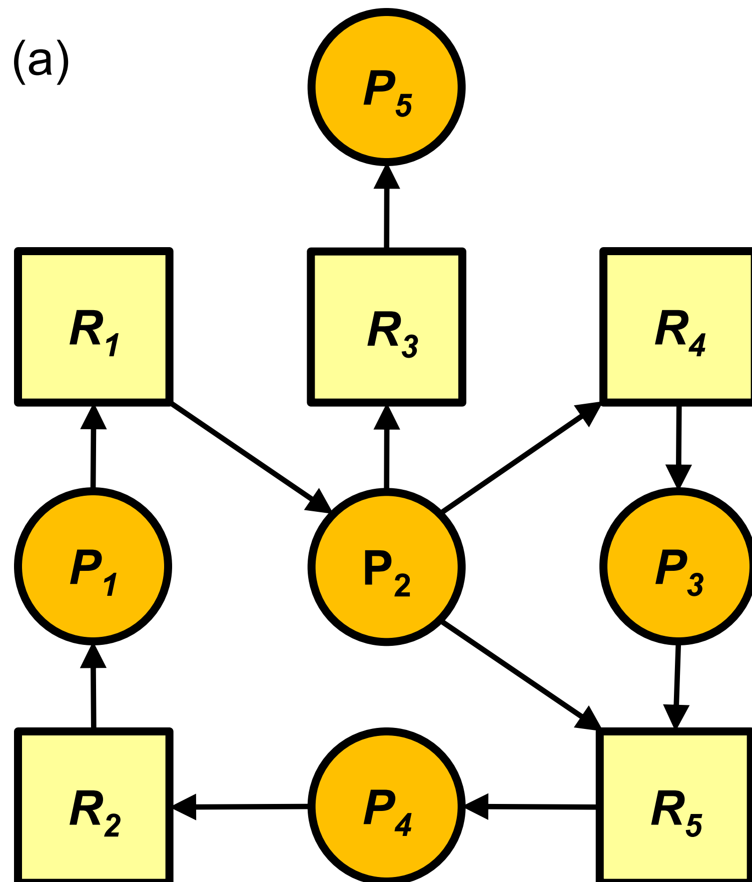
- Umožníme, aby došlo k uváznutí
- Ale toto uváznutí detekujeme
- Aplikujeme plán obnovy

1 INSTANCE PROSTŘEDKU KAŽDÉHO TYPU

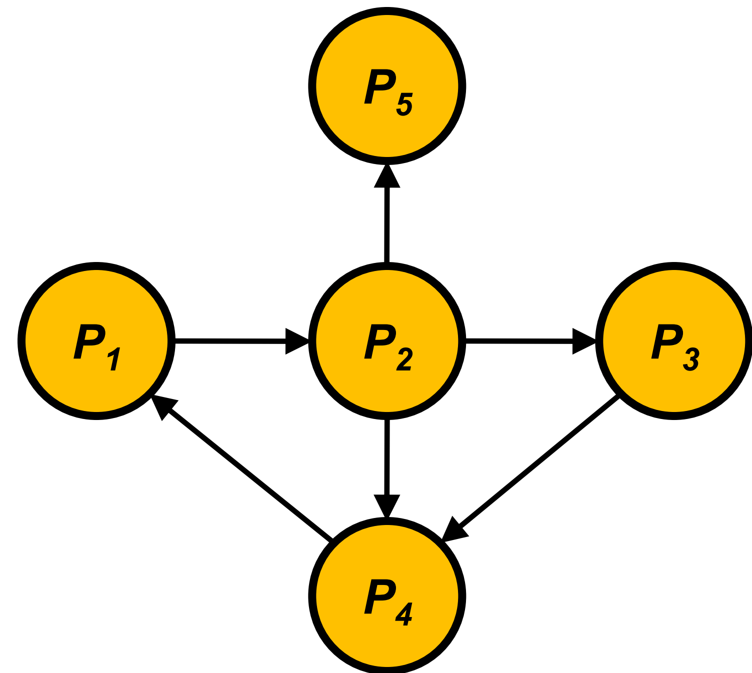
- Udržuje se graf čekání (wait-for graph)
  - uzly jsou procesy
  - $P_i \rightarrow P_j$ jestliže P_i čeká na P_j
- Periodicky se provádí algoritmus, který v grafu hledá cykly
- Algoritmus pro detekci cyklu v grafu požaduje provedení n^2 operací, kde n je počet uzlů v grafu

GRAFY

Graf přidělení zdrojů



Odpovídající graf čekání



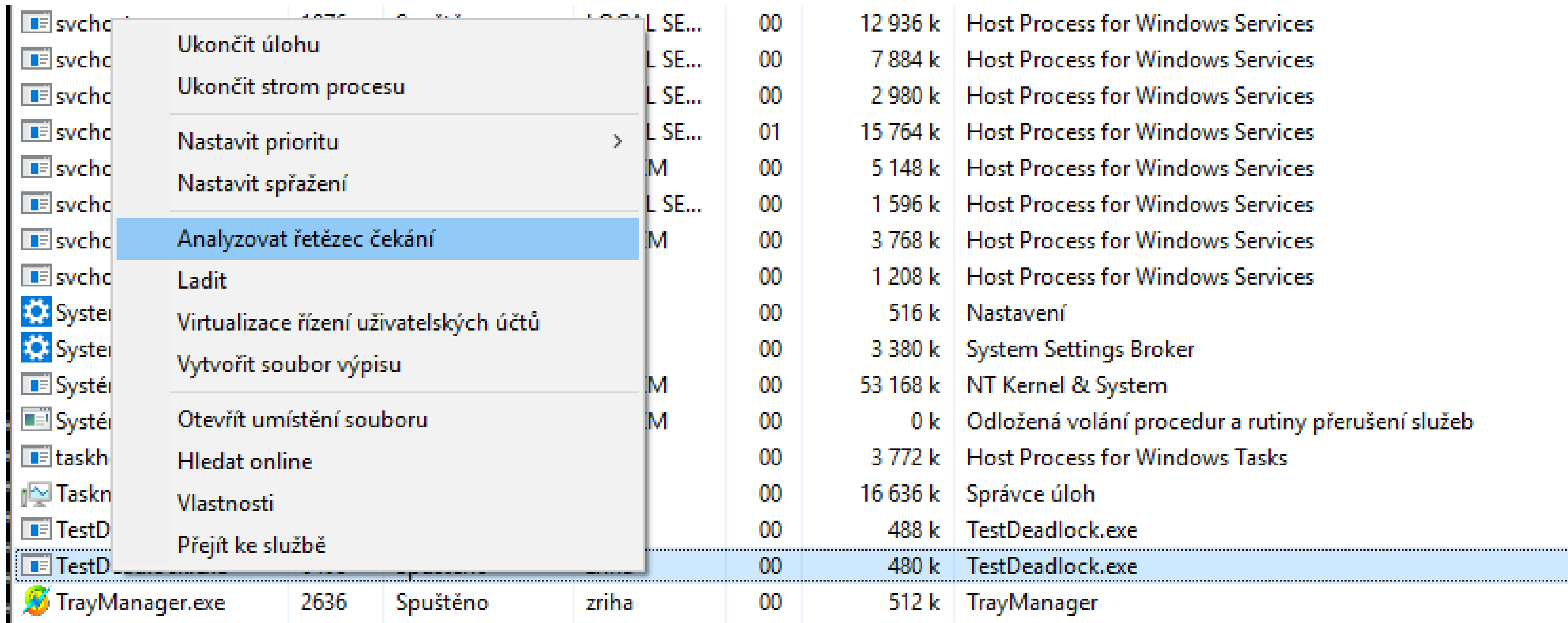
OBNOVA: UKONČENÍ PROCESU

- Násilné ukončení uváznutých procesů
- 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
  - preference interaktivních nebo dávkových procesů

OBNOVA: NOVÉ ROZDĚLENÍ PROSTŘEDKŮ

- Výběr oběti: minimalizace ceny
- Návrat zpět (rollback) – návrat do některého bezpečného stavu, proces restartujeme z tohoto stavu
- Stárnutí – některý proces může být vybírán jako oběť trvale
  - řešení: do cenové funkce zahrneme počet restartů (rollbacků)

Windows: řetězec čekání (1)



Windows: řetězec čekání (2)

Analyzovat řetězec čekání

Proces TestDeadlock.exe je zablokován jedním či více procesy. Zablokování můžete vyřešit okamžitě ukončením procesů nebo můžete počkat, zda nedojde k automatickému vyřešení problému.

- TestDeadlock.exe (PID: 6408) Vlákno: 8568
  - TestDeadlock.exe (PID: 7712) Vlákno: 1680
    - TestDeadlock.exe (PID: 6408) Vlákno: 8568

Strom analýzy řetězce čekání zobrazuje, které procesy (kořenové uzly v zobrazení stromu) používají nebo čekají na použití prostředku, který je používán jiným procesem (podřízené uzly v zobrazení stromu) a je nezbytný k pokračování vybraného procesu.

Další informace o řetězcích čekání

Ukončit proces | Zrušit

Windows: nástroje pro vývojáře

- Driver verifier – Deadlock detection (viz MSDN)
  - When Deadlock Detection finds a violation, it will issue bug check 0xC4. The first parameter of this bug check will indicate the exact violation. Possible violations include:
    - Two or more threads involved in a lock hierarchy violation
    - A resource that is released out of sequence
    - A thread that tries to acquire the same resource twice (a self-deadlock)
    - A resource that is released without having been acquired first
    - A resource that is released by a different thread than the one that acquired it
    - A resource that is initialized more than once, or not initialized at all
    - A thread that is deleted while still owning resources

Windows Applications: Best Practices

Users like responsive applications. When they click a menu, they want the application to react instantly, even if it is currently printing their work. When they save a lengthy document in their favorite word processor, they want to continue typing while the disk is still spinning. Users get impatient rather quickly when the application does not react in a timely fashion to their input.

A programmer might recognize many legitimate reasons for an application not to instantly respond to user input. The application might be busy recalculating some data, or simply waiting for its disk I/O to complete. However, from user research, we know that users get annoyed and frustrated after just a couple of seconds of unresponsiveness. After 5 seconds, they will try to terminate a hung application. Next to crashes, application hangs are the most common source of user disruption when working with Win32 applications.

Zdroj: MSDN

Windows Applications: Best Practices

Do:

•Design a lock hierarchy and obey it. Add all the necessary locks. There are many more synchronization primitives than just Mutex and CriticalSections; they all need to be included. Include the loader lock in your hierarchy if you take any locks in DllMain()

•Agree on locking protocol with your dependencies. Any code your application calls or that might call your application needs to share the same lock hierarchy

•Lock data structures not functions. Move lock acquisitions away from function entry points and guard only data access with locks. If less code operates under a lock, there is less of a chance for deadlocks

•Analyze lock acquisitions and releases in your error handling code. Often the lock hierarchy if forgotten when trying to recover from an error condition

•Replace nested locks with reference counters - they cannot deadlock. Individually locked elements in lists and tables are good candidates

•Be careful when waiting on a thread handle from a DLL. Always assume that your code could be called under the loader lock. It's better to reference-count your resources and let the worker thread do its own cleanup (and then use FreeLibraryAndExitThread to terminate cleanly)

•Use the Wait Chain Traversal API if you want to diagnose your own deadlocks

Do not:

•Do anything other than very simple initialization work in your DllMain() function. See DllMain Callback Function for more details. Especially do not call LoadLibraryEx or CoCreateInstance

•Write your own locking primitives. Custom synchronization code can easily introduce subtle bugs into your code base. Use the rich selection of operating system synchronization objects instead

•Do any work in the constructors and destructors for global variables, they are executed under the loader lock

Zdroj: MSDN

Výukovou pomůcku zpracovalo

Servisní středisko pro e-learning na MU

CZ.1.07/2.2.00/28.0041

Centrum interaktivních a multimediálních studijních opor pro inovaci výuky a efektivní učení

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ