OS - proces

16. prosince 2008
11:53

Proces
- Běžící instance programu (posloupnost instrukcí)
- Zapouzdřuje prostředí, ve kterém program běží - pro ty instrukce poskytuje schránku, protože s procesem je svázána hromada věcí - např. otevřené soubory, ty jsou k dispozici jen té otevřené části
- Na počítači jich může být mnoho
- Běžící program, který je natažen do paměti, má svá data, a nad nimi něco provádí, od programu se to liší tak, že se dá spustit vícekrát stejný program, a každý má vlastní data
- Instance je definována daty
- Tomuto prostředí se říká kontext

Proces kontext výpočtu
- Nejzákladnější kontext je kontext procesoru - stav procesoru, hodnoty registrů, zahrnuje instrukční registr, program counter [u jaké instrukce zrovna jsem], stack pointer [zásobník], a základní registry (R0 až R16 na 64bit intelu např.), paměť k registrům (stav zásobníku...)
- Kontext paměti (adresový prostor, obsah paměti)
  - Spojuje konkrétní virtuální paměť s obsahem paměti
- Kontext prostředí
  - Otevřené soubory, uživatelský terminál, komunikační kanály

- Pro OS je to datová struktura, která umožňuje asociovat různé kontexty s tou běžící instancí programu

Stavy procesu - ke změnám dochází v důsledku událostí
- Tři nejzákladnější
  - **Připraven** - připraven ke spuštění
  - **Běžící** - zde dělá co má, třeba právě něco vykresluje, když běžel moc dlouho, tak proces přeruší a přesune ho zpět do stavu Připraven
  - **Blokován** - čeká na nějaký prostředek, který momentálně není k dispozici, OS tedy ten proces uspí, jakmile bude prostředek (např. čtení z disku) k dispozici tak ho probudí do stavu Připraven

Události v systému
- 2 typy
  - Synchronní - vznikají v důsledku běhu procesu
    - Traps, speciální instrukce např. pro volání OS)
    - exceptions
  - Asynchronní - vznikají vně systému
    - Žádost zařízení o přerušení

- Přerušení z vnějšku: Časovač (integrován na procesoru) naprogramován, aby procesoru periodicky posílal interrupty - to přeruší běžící program, tím se k běhu dostane OS, uloží kontext procesoru, aby nezrušil výpočet) ten zkontroluje kdo ho vyvolal, když to byl časovač, tak se podívá na proces, podívá se, jak dlouho běží, pokud dlouho, tak se přepne na jiný kontext nové aplikace a tu vrátí k běhu
  - Přerušení se dá maskovat, to je privilegovaná operace, kterou může dělat jen OS
  - Aby se OS vždy jednou za čas dostal na procesor a podíval se, jestli není třeba přepnout proces
- Běžící <> Připraven .. Je synchronní akce

Vlákno
- Abstrakce toku výpočtů - aktivita
  - Aktivita Odpovídá toku instrukcí, které procesor vykonává
  - Více procesorů by znamenalo více aktivit
- Vlákno tedy odpovídá simulaci několika aktivit běžících zároveň .. Sdílí kontext paměti a

prostředkům
- Ale nesdílí kontext procesoru . Každé vlákno má vlastní proces; ani zásobník nesdílí - uchovává historii běhu výpočtu (volání funkcí)

PID -> nějaká tabulka seznamu procesů -> jeden proces [kontext paměti, kontext prostředí, a několik vláken aktivity, které má každé své vlastní kontext procesoru] každý proces má tedy alespoň jedno vlákno]
Pro iluzi víceprocesorového, přepínáme mezi sebou různá vlákna . Ale jim nechávám stejný kontext paměti a prostředí, to měním jen když přejdu na jiný proces

Přepínání kontextů mezi vlákny = **kontext switch**

Použití vláken
- Strukturování program
  - Samostatná vlákna pro zpracování požadavků na serveru
  - Potřeba větších abstrakcí - oddělení výpočtů, snazší kód, ač obětujeme trošku efektivity
- Implementace asynchronních I/O operací
  - Víc vláken, překrytí výpočetních operací s IO operacemi

Implementace vláken
- Přepínání kontextu procesoru
  - Na úrovni OS
  - Na úrovni uživatelského procesu - proces o našich vláken neví
  - Když to poskytuje OS, tak u více procesorových věcí nám je může rozdělit na více procesorů

Terminologie
- Multitasking
  - Více procesů sdílející jeden procesor, bez ohledu na implementaci
- Multiprogramming - systém spravuje více procesů najednou, spouští je na jednom nebo více provesorech
- Multiprocessing - použití více procesorů pro běh jednoho procesů

Motivace pro více procesů
- Zlepšení odezvy systému
  - Dlouhé odezvy při dávkovém zpracování úloh
  - Při sdílení procesoru by kratší úlohy byly hotovy dříve
- Zlepšení využití systému
  - Aplikace typicky něco počítá, nebo čeká na data
    - Doba čekání na data z disku v řádu ms
  - Během čekání na data mohou jiné aplikace "něco počítat"
  - Zlepšuje odezvu - jiný proces může postupovat vpřed
- Současný běh více procesů
  - Přepínání mezi více aplikacemi, spojování procesu pro práci na stejném problému, …

Plánování
- Rozhodování, jak procesy naplánovat k běhu, který proces poběží a jak dlouho
- Rozhodnutí optimalizuje nějakou metriku

Typické metriky
- Doba odezvy (response time)
  - Minimalizuje se doba do konce procesu; když zmáčkneme klávesu, aby se zobrazila co nejdříve - pro interaktivní aplikace
- Propustnost (throughput)
  - Co nejrychleji spočítal výsledek, tedy co nejvíc dokončených úloh za jednotku času)
- Využití procesoru (ulilization)
- Spravedlnost - aby se každému procesu dostalo nějakého přídělu

Offline plánování

- Předpoklady: víme jaké procesy poběží, a žádné už nepřibudou, a o všech procesech víme, jak dlouho poběží
- Výsledky: dávková zpracování s ohledem cílové metrik; běh procesu není nutné přerušovat, plán je z hlediska metriky optimální
- Problém:
  - Předpoklady jsou málo realistické [křišťálová koule]
  - Poskytují teoretické meze, pokud bychom měli požadované informace - jak nejlépe to jde udělat, kdyby… k té teoretické mezi můžeme srovnávat realističtější algoritmy

Základní offline algoritmy

FCFS - First come, first served

- Základní algoritmus dávkového zpracování
  - Procesy plánované v pořadí v jakém přicházejí; procesy běží dokud neskončí
  - Když víme, do kdy chceme výsledek [abychom stihli deadline]

SJF

Online plánování

- Odstraňuje základní nedostatky offline plánování
- Předpoklady
  - Procesy se objevují libovolné a neočekávaně
  - Doba běhu procesů je neznámá
- Kritéria plánování:
  - Vázanost na CPU nebo I/O, interaktivní/dávkový proces - jestli u toho je uživatel a čeká na výstup, nebo něco samostatného
  - Chování procesu v minulosti, výpadky stránek, priorita procesu…

- Nemůžeme zvolit optimální plán dopředu, a pokud OS zjistí, že je potřeba přehodnotit plán, tak
- Potřeba:
  - Potřebujeme podporu HW (časovač)
  - Možnost měnit plán na základě nových informací
  - **Context switch** - přepnutí na jiný proces / vlákno

Plánovací strategie:

FCFS - First Come First Served

- V pořadí jaké se objevují
- Neepremptivní
- Příklad klasického MS DOSu - dokud první proces neskončí, další nemá šanci

RR - Round Robin

- Každému procesu, který spustíme přidělíme časové kvantum,
- Každý proces, který přijde, necháme chviličku ohřát na procesoru, pak ho dáme na konec fronty
- Procesy se točí dokola
- Preemtipvní plánování
- "víc dětí se střídá o kolující hračku ;-) "
- Pásma např. v řádu 20ms, potom jde další proces
- Preemptivní = potřebujeme zajistit, že žádný proces na procesoru nezůstane dlouho a nesebere celý procesor
  - Zajištěno tak, že procesoru chodí periodicky přerušení od zařízení, časovače, který se dá naprogramovat ,aby procesoru periodicky posílal přerušení
  - OS v obslužné rutině přerušení se vždy podívá, jak dlouho daný proces na procesoru běží, a pokud se proces nevzdá procesoru, tak běží dokud mu kvantum nevyprší, pak se v obsluze provede context switch
  - Datová struktura si udržuje, jak dlouho smí procesor běžet, a kontroluje se, jestli ještě proces může být
  - Na intelských procesorech se dá např. zakázat přijímání přerušení na procesoru, ale kdyby toto proces provedl, tak by byl odstřelen, protože CLI (zrušeni přerušení) je privilegovaná operace, protože to vyvolá exception
- S jednou frontou není efektivní, pro aplikace čekající na vstup uživatele, není user-

friendly

Prioritní s více frontami a zpětnou vazbou

- můžeme mít několik front, hirerarchicky seřazené
  - První dvě fronty např. FIFO,
    - Dávám sem uspané procesy, např. čekající na vstup z procesoru a jakmile přijde vstup, tak probereme
  - poslední RR
- Plánovač se tedy dívá na druhy procesů
- Grafický program: chvíli nastavujeme co se má dělat, a pak spustíme nějakej render a tím přestává být program interaktivní a proces se začne posunovat do nižších priorit, až může skončit v RR.. Jakmile dopočítá a začne čekat na vstup, tak se zase může dostat do vyšších pozic priorit
- Pokud proces hodně spí, typicky to je proces vázaný na vstup a výstup => bude to asi interaktivní
- Pokud hodně počítač, a plánovač ho násilím přepne na jiný proces, => asi stále běží a bude třeba ho odpojovat až "násilím"

Plánování pro více procesorů:

- Je třeba dobře plánovat, abychom rovnoměrně zatěžovali jádra
- Afinita k procesoru - po procesu zůstala nějaká data v cache, takže příště, pokud poběží, tak by mohl běžet rychleji, protože už má data v cache
  - TO už není tak pravda u vícejádrových procesorů se sdílenou L2 cache (a L1 je tak mala, ze nemá cenu ji zohledňovat)

Pro real-time systémy:

- Aplikace řízené událostmi (např. brzdový systém v autě)
- Např. je tam časovač vyvolávající čtení ze senzorů
- Podstatné je, že běh úloh je omezen reálným časem, do kdy se musí něco stát; zde se časová nesnaží o nejlepší službu, ale o splnění do svého času -
- Omezen časem dokončení - deadline
- Hard real-time - netoleruje se žádné zpoždění; po zpoždění nemá cenu se snažit
  - Specializované systémy
  - Typicky offline plánování - víme jaké úlohy a jak dlouho trvají
  - Jednoúčelové systémy, často těžce programovatelné
- Soft real-time - drobné zdržení se toleruje, na úkor ohodnocení, má cenu odevzdávat
  - Tohoto se dá docílit na úrovni OS, (Linux, Windows)

Interakce mezi procesy:

- Přístup ke sdíleným datům
  - Datové struktury OS nebo vícevláknového programu
  - Aby se dalo zavolat ze dvou různých procesů
  - Potřeba, aby OS podporoval tento souběh procesů, aby nedošlo ke zničení vnitřních struktur OS
- Zablokování na sdílených prostředcích

- Komunikace mezi prosety
  - Spolupráce procesů v rámci paralelní/distribuované sítě
  - Bud společná paměť na PC, nebo po síti

Souběžné vs. Paralelní zpracovávání

- Ač mám jedno aktivní vlákno, tak můžu mít i tak víc kontextů
- V msdosu, když jsme zavolali, že chceme přečíst soubor, a když jsme ho zavolali z jiného programu, a pokud jsme chtěli z něj také číst z disku, tak se mohla poškodit tabulka alokací souborů - nutná kontrola, jestli to jde

Souběžné provádění

- Více činností na různých místech prokládaně
- V daném okamžiku pouze jedno aktivní vlákno
- Může nastat i na jednoprocesorovém stroji

Paralelní zpracování

--- ze slidu ---

Implicitní sdílení dat:

- Synchronní události
  - ◇ Služby poskytované procesům
- Asynchronní - přerušení, paket ze sítě…

Explicitní sdílení

- V rámci procesu může být několik vláken, která sdílí společnou paměť
- OS by to měl podporovat, aby mohla běžet na více jádrech
- A systém musí hlídat, aby nedošlo k problémům s pamětí - všechna vlákna jdou na stejnou paměť
- Problém je stejný jako u souběžného nebo paralelního, ale menší škody

Problém v přístupu ke sdíleným datům:

- Operace nad daty sestávají z více kroků
- Problém, během provádění několika kroků může být datová struktura v nekonzistentním stavu a přeplánování je třeba protože vyžaduje procesor
- Např. Jedno vlákno začne přidávat, procesor přeruší a mezitím jiné vlákno z toho začne číst -> práce s nekonzistentními daty!

-- příklad na spojáky ze slidů --

- Kdyby první vlákno běželo o chviličku déle - o jednu instrukci, tak by bylo vše ok, ale teď nám vznikl paskvil spojený dvou spojáků
- Problému se říká **race condition**

Řešení:

- Chceme nějakou možnost, jak mít atomickou změnu, aby ji nikdo nemohl přerušit
- DOKUD ta operace není dokončena, nesmí začít jiná
- Musíme identifikovat kritické sekce programu - kusy programu, kde nesmí být najednou více než jeden proces; operace trošku spojeny s daty
- Když máme dva seznamy, dokud přidáváme do každého z nich tak je to ok, ale přesun je problém
  - Tyhle kusy programu musíme nějak označit. Znamená to, že tady se smí nacházet jen jeden proces, tím zajistíme datovou konzistenci dat
- Systém nám zajistí vzájemné vyloučení (mutual exclusion) = pouze jedna aktivita smí provádět takovou operaci

Realizace vyloučení:

Přidání sdílené proměnné pro řízení přístupu

```
While ( locked ); dokud je zamčeno, nic nedělím
Locked = true; zamknu
Critická sekce // provádím co chce
Locked = false; //zase odemknu
```

Přidali jsme do programu novou race condition - přidali jsme si de facto další operaci lock, která sama není atomická - čekáme dokud není odemčeno a pak zamkneme

- Jakmile zjistíme, že je odemknuto tak se posuneme na řádek locked = true, ale mezitím nás procesor přepojí, jiný proces také čekal a zamkl, pak nás OS vrátí zpět, a my nezjistíme teď druhý proces, a zase máme dva procesy v kritické sekci, naopak si to ještě zkomplikujeme dalším race

Řešení DVA:

- Pro každý proces uděláme locked proměnné

- A porovnáváme zámky jiných procesů - chci zamknout svůj zámek, ale dívám se na zámky sousedů, jestli oni nemají zamknuto
- Když vstupují postupně, tak ok, vyloučí se a je to v pohodě, ale pokud oba ve stejnou chvíli vstoupí do kritické fáze, tak tu máme zase race condition
- Když se oba zamkneme, a pak nás přeruší, a když se vrátíme zpět, tak jsou oba v cyklu a čekají - deadlock

Opět nejde

Řešení TŘI:

- Už funguje
- K proměnné locked přidávám i turn, kdo je na řadě
- Pak mám čekací cyklus pro Proces 1 while (p2_locked && turn == P2);
- Tím rozbijem situaci, že oba sledujeme úmysly, a vznikla by nekonečná smyčka, protože ještě zapisujem kdo je na řadě
- A nenastane deadlock
- Petersonův algoritmus, funguje pro n procesů
- Dnes se používá jiný postup, s podporou HW

Řešení s podporou HW

Zámek s podporou HW
While(test_and_set(locked)); // zeptá se na předchozí hodnotu a pak ji přečte, zapíšeme novou hodnotu a zjistíme původní - cyklíme zde, dokud je stará hodnota je 1 -- pokud zamkneme a původní hodnota byla 0, tak ok, předtím to nebylo zamknuté
Whilem čekáme, dokud nemůžeme zamknout, po provedení kritické sekce zase jednoduše odemkneme

- Prečte proměnnou, nastaví ji na true a vrátí původní hodnotu
- Ršení je, že tomuhle odpovídá jedna instrukce (popč. Na mipsu víc) ale je to atomická operace
- Spinlock : proměnná + operace lock/unlock

Zámky musím skládat na různá místa,aby nestál celý program

```
Spinlock_+ list_lock;
Spin_lock(&list_lock);
Kritická sekce
Spin_unlock(&list_lock);
```

Problém spin-locků:

- Aktivní čekání (busy wating) při zamčení zámku
  - Cyklus while čeká dokud se neodemkne a nikdo nám ho nemůže odemknout, dokud se nepřepne a jiný to odemkne, trošku to řeší víceprocesorové řešení
- Procesor nedělá nic užitečného

Jiné řešení?

- Pasivní čekání
  - Pokud je zamknuto, vlákno se uspí (ready -> blocked)
  - Procesor může dělat něco jiného (užitečného)
  - Ten kdo odemyká zámek vzbudí uspané vlákno
  - Uspání probuzení procesu vyžaduje podporu OS
- Realizace pasivního čekání
  - Zámek + sleep/wakeup
  - While(test_and_set(locked)) sleep(queue);

- Máme sice atomický test, ale uspání už atomické není
- Když se to přeruší nevhodně, tak se může proces sám uspat, ale už ho nemá kdo probudit
- Je potřeba zámek k frontě, který se při sleep uvolní
- V okamžiku, kdy se jdu uspat, tak to musí zajistit odemknutí zámku, když se uspávám, musí být podporou OS zajištěno, že se uspat mám
- Klasický zámek je proměnná + fronta čekajících
- OS poskytuje synchronizační primitiva - datová struktura + synchronizační operace
- Pokud by to bylo v OS, tak řešením by bylo chvilkové zakázání přerušení

-- mutex --

Časté synchronizační primitivum - semafor

- Datová struktura - zámek a fronta
- Má dvě operace
  - Down - zavede semafor, odkud je volný
  - Up - uvolní semafor

```
Semaphore {
    Int count;
    Queue_+ Queue;
}
```

Funguje tak, že dokud je count větší jak 0, tak operace down neblokuje, v okamžiku, kdy count je 0, tak operace down zablokuje vlákno

Semafor a tři kola.. Na začátku dáme counter na 3, přijde první, dostane kolo, přijde druhý, dostane, třetí dostane, čtvrtý, nedostane už není kolo k dispozici, páč counter je 0.. Musí čekat

Povolí prosředky přidělit více účastníkům

- Semafor je zabraný při hodnotě nula
- Zabrání snižuje parametr o 1; uvolnění zvyšuje

Např. u procesorů který něco produkuje a jiný konzumuje
Konzument odebírá.. Komunikace přes mezisklad
Producent produkuje, dokud je ve skladu místo a naopak konzument musí zase brát dokud něco ve skladu je

-- příklad ze slidu --

- Nesmí nastat deadlock - např. přehozením down empty a down mutex

Realizace

- Operace down

```
If ( --value  < 0) block_thic_process();
```

- Nutno vyřešit realizace vnitřní klasické sekce
- Spinlock na strukturu + odemknutí při uspání

  Nebo

- Zákad přerušení na procesoru
  - Tohle ale může zkomplikovat více procesorů, kdy někdo z jiného procesoru se nám v tom může pohrabat
  - To už řeší jen spinlocky

Pasivní čekání pomocí monitoru

- Datová struktura + operace pro čtení/změnu stavu
- Operace ve stejné instanci se vzájemně vylučují

- Tuhle realizaci poskytuje Java - zde jsou objekty, a metody, které ty datové struktury mohou měnit.. A můžeme označit,metodu, že je synchronized.. A to zajistí runtime, že jen jedna z nich bude v danou chvíli aktivní
- Speciální operace wait
  - Když ji zavoláme a jsme uvnitř monitoru, se nám volající proces zablokuje a uvolní se pro jiný proces.. Odemkne se monitor pro jiný proces, vědomě!
- Speciální operace signal
  - Vzbudíme všechny uspané procesy, ale neznamená, že monitor hned předáme, ten se uvolní až z něj vyjdeme a pak si procesy sami zajistí kdo získá monitor
- Realizace: zámek + fronta spících procesů

Další synchronizační primitiva
- Read/write zámky - více čtenářů může zamknout, jen jeden writer může zamknout, až všichni čtenáři odemknou

Ekivalence synchronizačních primitiv
- Když máme pasivní zámek, tak můžeme zbylé naimplementovat podle něj..to stejné pro semafor..

Obecně funguje pouze u sdílené paměti
-> komplikace u distribuovaných systémů

Prostředky
- Výpočetní - pro běh programu
- Synchronizační - ke koordinaci konfliktů

Přidělování prostředků
- OS jako centrální správce prostředků, přiděluje právo používat nějaký prostředek (nebo část, jeli dělitelná)
- K zablokování -deadlocku- může dojít v situaci, kdy procesy žádají současné přidělení více prostředků

Práce s procesy
- V několika fázích
  - Žádost - blokující
  - Použití prostředků
  - Dobrovolné odevzdání prostředků - poté odblokování

Zablokování
- Množina procesů je zablokována, jestliže každý proces z této množiny čeká na událost, kterou může způsobit pouze jiný proces z této množiny
- Např: Dva procesy, kdy každý čeká až ten druhý odemkne, ale oba čekají…
- Např 2: filozofové
  - 5 filozofů = 5 procesů
  - Každá vidlička je sdílena 2 filozofy - jedna operace je sdílena 2 procesy
  - Blokující: Když přijde 5 filozofů najednou, tak každý vezme levou vidličku, ale pravou už ne, protože ji vezme druhý a nikdo se už nenají = DEADLOCK
  - Opatrná funkce: Pokud nemůžu vzít druhou vidličku, tak položím tu předchozí - zde všichni pracují, ale už se nenají = LIVELOCK + vyhladovění - sice pracují, ale vyhladoví se
  - Možné řešení:
    - Jeden z filozof vezme vidličky v jiném pořadí
      - Bud první - sebere kolegovi vidličku; nebo poslední - hned čeká
    - Do jídelny pustíme maximálně 4 filozofy najednou
    - Randomizace časů - v případě opatrné funkce - je třeba zavést prodlevy do rozhodování (kdy zkusí vzít obě vidličky…)
- Formální model
  - Stav reprezentován orientovaným grafem
  - Za běhu systému konstruujeme model závislostí, a pokud v něm najdeme cyklus a jsou splněny nějaké další podmínky, tak to je silný indikátor potencionálního

DEADLOCKu

- Podmínky - Coffmanovy podmínky:
  - Při splnění všech těchto podmínek dojde k zablokování
  - Dány prostředím - programovacím modelem
    - Výlučný přístup - prostředek přidělujeme výhradně jednomu uživateli (máme tři kola, vždy půjčíme jednomu člověku)
    - Neodnímatelnost - když ho přidělíme, nesmíme ho násilím sebrat, ale čekáme až odevzdá - nesmíme ukrást zámek
    - Drž a čekej - proces může zároveň držet jeden prostředek a čeká na další prostředek - až někdo uvolní druhý zámek
    - Kruhová závislost - dva zámky a dva procesy, čekají na sebe
  - Když tyhle platí, tak nastává situace deadlock
  - Kdyby nějaká neplatila, tak to nemusí být deadlock
- Možnosti řešení deadlocku
  - Deadlock prevention: Prevence - napadení jedné z předchozím podmínek
    - Explusive use - dá se zařídit jakouli virtualizací
      - Např u tisku spooling - iluze výlučného přístupu - tiskový server má jediný přístup k tiskárně a všichni své tiskové úlohy dávají jemu
    - Neodnímatelnost - odnímatelné prostředky lze odejmout bez následků - jde je odejmout nějak transparentně, aniž by o tom aplikace tušila - u procesoru přeplánováním; u paměti swapping - paměť dáme na disk, použijeme pro jiný proces a pak zase dáme paměť zpět
      - Neodnímatelné prostředky nelze odejmout bez nebezpečí selhání výpočtu
      - Nemáme způsob, jak říct programátorovi, že jsme přišli o paměť
    - Drž a čekej
      - OS vrátí chybu místo zablokování procesu
      - Nutno žádat a všechny prostředky najednou
      - Před žádostí nutno všechny prostředky uvolnit
    - Kruhová závislost
      - Rozbít možnost vzniku cyklu, např očíslování prostředků a povolit žádání o další prostředky pouze s vyšším číslem
      - Pořadí nemusí být globální pro celý OS, ale stačí v rámci množiny prostředků sdílených současně v nějakém kontextu
  - Deadlock Avoidance - nezrušíme jednu podmínku, ale průběžně kličkujeme a snažíme
    - Výchozí stav - známe počet dostupných a přidělených prostředků
    - Následující stav - sem se v následku nějaké události přesuneme - stále nutno zajistit, že procesy nejsou zablokovány - žádosti o přesun do tohoto stavu bude vyhověno, pokud bude následující stav bezpečný
    - Bezpečný stav: existuje pořadí, v jakém uspokojit všechny procesy; je možné plně uspokojit alespon 1 proces - takový proces je zase vrátí

      Bankéřův algoritmus [viz slidy - obr - v druhém obr můžeme schválit jen procesu C, ten vezme další dva - tím splní počet a pak vše = 4 vrátí]
      - Máme např. 100 tisíc a třem firmám slíbíme 100 tisíc, tedy celkem 300.. Ale nepůjčujeme vše najednou, ale po částech a mezitím se nějaké vrací zpět
      - U systému se předpokládá, že požadavky budou přicházet postupně a že se prostředky budou vracet
      - O rozdělení prostředků se složitě rozhoduje - složitost O(n^2)
      - Požadované informace jsou typicky nedostupné
  - se jednu z nich vyvrátit
  - Řešení problému až když nastane

    - *[přesněji zařadit- nevím jestli primo sem]*
    - *Deadlock detection*
        - *Model závislostí mezi procesy ve formě grafu*
        - *Test n přítomnost kruhových závislostí - hledání cyklu v orientovaném grafu*
    - *Deadlock recovery*
        - *Odebrání prostředku - na přehodnou dobu, pod dohledem operátora*
        - *Odstranění nepohodlných prostředků*
        - *Checkpointiong/rollback - OS ukládá stav procesů; restart procesu v předchozím stavu*
        - *Transakční zpracování - u databází - změny, které se prováděli se nezohlední, dokud není transakce dokončena*
- Pštrosí algoritmus
    - Předstíráme, že problém neexistuje - systém dělá že neví a problém se vyřeší sám
    - Např. na linuxu -- problém typicky vyřeší uživatel sám (kill -9)