```
KIV/ZOS 2003/2004
Přednáška 8

Obecně existují 4 strategie zacházení s uvíznutím:

1. Problém uvíznutí je zcela ignorován
2. Detekce a zotavení
3. Dynamické zabránění pomocí pečlivé alokace zdrojů
4. Prevence, pomocí strukturální negace jedné z výše uvedených
   4 nutných podmínek pro vznik uvíznutí

1. Ignorování problému
----------------------

* "pštrosí algoritmus: strčíme hlavy do písku a předstíráme, že problém
  neexistuje" :-)
* cena za eliminaci uvíznutí je vysoká (např. činnost
  uživatelských procesů je nepříjemně omezena apod.) => neexistuje
  všeobecně přijímané řešení
* žádný ze známých OS se nezabývá uvíznutím uživatelských procesů
  (většina OS ale snaha o eliminaci uvíznutí pro činnosti jádra)

2. Detekce a zotavení
---------------------

* systém se nesnaží zabránit vzniku, ale detekuje, po detekci provede
  akci pro zotavení

Detekce uvíznutí pro jeden zdroj každého typu
.............................................

* při žádostech o zdroj/přiřazení zdroje OS konstruuje graf alokace
  zdrojů
* detekce cyklu => nastalo uvíznutí
* pro detekci cyklů různé algoritmy (viz teorie grafů)
 - jednoduchý algoritmus: prohledávání do hloubky z každého uzlu (pokud
   dojdeme do uzlu, který jsme již jednou prošli, je cyklus)

Zotavení z uvíznutí
...................

* zotavení pomocí preempce - vlastníkovi zdroj dočasně odejmout
 - závisí na typu zdroje, často obtížné až nemožné
 - například tiskárna - po dotištění stránky proces pozastavit, odejmout
   mu tiskárnu a přiřadit jinému (vyžaduje ruční intervenci - vyjmout
   již vytištěné stránky)
* zotavení pomocí zrušení změn (rollback)
 - pokud jsou uvíznutí častá, může systém v pravidelných intervalech provádět
   checkpointing procesů = zápis stavu procesů do souboru, aby proces mohl
   být v případě potřeby "vrácen" do uloženého stavu
 - po detekci uvíznutí je proces vlastnící zdroje vrácen zpět -
   nastavení na některý jeho dřívější checkpoint, kdy ještě zdroj
   nevlastnil (práce procesu od checkpointu je tím ztracena)
 - zdroj přiřadíme uvízlému procesu => zrušíme deadlock
 - proces, kterému jsme zdroj odebrali, se ho pokusí alokovat => usne
* zotavení pomocí zrušení procesu
 - nejhroznější způsob - zrušíme jeden nebo více procesů
 - zrušit proces v cyklu; pokud nepomůže, zrušíme další atd.
 - není příliš dobré řešení; často alespoň snaha zrušit procesy, které je
   možné spustit znovu od začátku

3. Dynamické zabránění
----------------------

* ve většině systémů procesy žádají o zdroje po jednom
* systém rozhodne, zda je přiřazení zdroje bezpečné nebo hrozí uvíznutí
* pokud bezpečné, zdroj přiřadí, jinak proces pozastaví žádající proces
* stav je bezpečný, pokud existuje alespoň jedna posloupnost, ve které mohou
```

procesy doběhnout aniž by nastalo uvíznutí

I když stav není bezpečný, uvíznutí nemusí nastat nutně - uvidíme dále.

Bankéřův algoritmus pro jeden typ zdroje
....

* zde předpokládáme více zdrojů stejného typu (!)
* například N magnetopáskových jednotek apod.

* algoritmus plánování, který se dokáže "vyhnout" uvíznutí (Dijkstra 1965)
* bankéř na malém městě, má 4 zákazníky A, B, C a D, každému garantuje půjčku dané velikosti (6, 5, 4, 7), dohromady 22 (např. K$)
* bankéř ví, že všichni zákazníci nebudou chtít půjčku současně, pro obsluhu zákazníků si ponechává pouze 10

* stav: zák. má max.; 2 volné jednotky

zák.	má	max.
A	1	6
B	1	5
C	2	4
D	4	7

 - stav bezpečný, protože bankéř může pozastavit všechny požadavky kromě C
 - dá C 2 jednotky, C skončí a uvolní 4, můžeme použít pro D nebo B atd.

* co kdybychom B dali 1 jednotku oproti výše uvedenému?
 - stav: zák. má max.; 1 volná jednotka

zák.	má	max.
A	1	6
B	2	5
C	2	4
D	4	7

 - stav není bezpečný, protože kdyby všichni zákazníci požádali o maximální půjčku, bankéř by nemohl uspokojit žádného => nastalo by uvíznutí (uvíznutí nemusí nastat nutně, ale s tím bankéř nemůže počítat)

* bankéř uvažuje pro každý požadavek, zda vede k bezpečnému stavu následujícím způsobem:

1. bankéř předpokládá, že požadovaný zdroj byl procesu přiřazen a že všechny procesy požádaly o všechny bankéřem garantované zdroje
2. bankéř zjistí, zda je dostatek zdrojů pro uspokojení některého zákazníka; pokud je dostatek zdrojů, předpokládá, že zákazníkovi byla suma vyplacena, skončil a uvolnil (vrátil) všechny zdroje
3. bankéř opakuje krok (2); pokud mohou všichni zákazníci skončit, je stav bezpečný

* požadavek je vykonán (zdroje poskytnuty) pouze pokud vede k bezpečnému stavu, jinak je odložen na později (proces je pozastaven)

Bankéřův algoritmus pro více typů zdrojů
....

* bankéřův algoritmus může být zobecněn pro více typů zdrojů
* uvádím pouze pro zajímavost

* používáme dvě matice (sloupce představují třídy zdrojů, řádky zákazníky)
 - matice přiřazených zdrojů (current allocation matrix) - který zákazník má které zdroje
 - matice ještě požadovaných zdrojů (request matrix) - kolik zdrojů kterého typu budou procesy ještě chtít

Například:

matice přiřazených zdrojů	zdroje:	R	S	T
	zák.A	3	0	1
	zák.B	0	1	0
	zák.C	1	1	1
	zák.D	1	1	0

matice ještě požadovaných zdrojů	zdroje:	R	S	T
	zák.A	1	1	0
	zák.B	0	1	1
	zák.C	3	1	0
	zák.D	0	0	1

* zavedeme vektor A volných zdrojů (available resources)
  např. A = (1, 0, 1) znamená jeden volný zdroj typu R, žádný typu S atd.

Algoritmus pro určení, zda je stav bezpečný:

1. V matici ještě požadovaných zdrojů (request matrix) hledáme řádek, který je menší nebo roven A. Pokud takový řádek neexistuje, nastalo by uvíznutí.
2. Předpokládejme, že proces obdržel všechny požadované zdroje a skončil. Označíme proces jako ukončený a přičteme všechny jeho zdroje k vektoru A.
3. Opakujeme kroky 1 a 2, dokud všechny procesy neskončí (tj. původní stav byl bezpečný) nebo dokud nenastalo uvíznutí (původní stav nebyl bezpečný)

* Od 1965, kdy byl bankéřův algoritmus publikován, se objevuje ve všech učebnicích OS (+ mnoho odborných článků apod.)
* v praxi v podstatě nepoužitelný, protože:
  - procesy zřídka vědí dopředu, jaké budou jejich maximální požadavky na zdroje
  - počet procesů není konstantní, ale mění se v průběhu času (uživatelé se přihlašují, spouštějí procesy, odhlašují se...)
  - zdroje mohou "zmizet" - tiskárně dojde papír...
* nevím o žádném praktickém použití bankéřova algoritmu pro zabránění uvíznutí
* algoritmy odvozené od bankéřova algoritmu lze použít pro detekci uvíznutí při více zdrojích stejného typu

4. Prevence uvíznutí
----

* jak skutečné systémy zabraňují uvíznutí?
* podíváme se zpět na Coffmanovy 4 podmínky vzniku uvíznutí:

1. vzájemné vyloučení - výhradní přiřazování zdrojů
2. "hold and wait": proces držící zdroje může požadovat další
3. nemožnost zdroje odejmout
4. cyklické čekání

* pokud některá podmínka nebude nikdy splněna, je uvíznutí strukturálně nemožné

1. Vzájemné vyloučení
....

* zdroj nikdy nepřiřadit výhradně
* byl by problém pro některé zdroje (tiskárna)

=> spooling - pouze daemon přistupuje k tiskárně, nikdy nepožaduje další zdroje => není uvíznutí pro tiskárnu

Problémy:
* spooling není možný pro všechny zdroje (záznamy v databázi)
* spooling převádí soutěžení o tiskárnu na soutěžení o diskový prostor:
  - např. 2 procesy zaplní každý polovinu disku, žádný nemůže skončit

2. Podmínka "hold and wait"
....

* problém, že proces držící výhradně přiřazené zdroje může požadovat další zdroje
* řešení:
  - požadovat, aby procesy alokovaly všechny zdroje před svým spouštěním
    . procesy většinou nevědí, které zdroje budou chtít, a i kdyby věděly, tak je požadavek příliš restriktivní
    . některé dávkové systémy; i když popsané nevýhody, deadlocku zabraňuje
  - variace - pokud proces požaduje nové zdroje, musí uvolnit zdroje

které drží a o všechny zdroje požádat v jediném požadavku

3. Nemožnost zdroje odejmout
....

* odejímat zdroje poměrně obtížné

4. Cyklické čekání
....

* proces může mít jediný zdroj, a pokud chce jiný, musí předchozí uvolnit - poněkud restriktivní
* všechny zdroje očíslovány, požadavky musejí být prováděny v číselném pořadí
 - alokační graf nemůže mít cykly
 - problém: je těžké nalézt vhodné očíslování pro všechny zdroje
 - není obecně použitelné, ale ve speciálních případech (např. uvnitř OS, databázového systému) je výhodné použít

Příklad: dvoufázové zamykání (používá se v databázových systémech)

* první fáze - zamknutí všech potřebných záznamů v číselném pořadí
 - pokud je některý zamknut jiným procesem, uvolní všechny zámky a zkusí znovu
* druhá fáze - čtení/zápis & uvolňování zámků
* protože se zamyká vždy v číselném pořadí, uváznutí nemůže nastat

Shrnutí přístupů k uváznutí
....

1. Ignorování problému - většina OS ignoruje uváznutí uživatelských procesů
2. Detekce a zotavení - pokud uváznutí nastane, detekujeme a něco s tím uděláme (vrátíme čas zpátky - rollback, nebo zrušíme proces)
3. Dynamické zabránění - zdroj přiřadíme pouze pokud bude stav bezpečný (bankéřův algoritmus - jediné se týkalo více zdrojů stejného typu)
4. Prevence - strukturálně negujeme jednu z Coffmanových podmínek:
 - vzájemné vyloučení - spooling všeho
 - "hold and wait" - procesy požadují všechny zdroje na začátku
 - nemožnost odejmutí - odejmi (nefunguje)
 - cyklické čekání - zdroje očíslujeme a žádáme v číselném pořadí.

Vyhladovění

* procesy požadují zdroje, musí existovat pravidlo pro jejich přiřazení
* může se stát, že některý proces zdroj nikdy neobdrží, i když nenastalo uváznutí

* například večeřící filosofové v již uvedeném příkladu
 - každý zvedne levou vidličku, pokud je pravá obsazená tak levou položí
 - vyhladovění, pokud všichni zvedají a pokládají současně

* nebo pokud přiřazujeme zdroje podle strategie SJF
 - například tiskárnu dostane proces, který chce vytisknout nejkratší soubor
 - pokud některý proces chce vytisknout velký soubor a je mnoho malých požadavků - může dojít k vyhladovění
 - řešení - FIFO, nebo označíme požadavek časem příchodu a při překročení povolené doby setrvání v systému bude obsloužen

Terminologické rozdíly

V češtině není pro některé pojmy OS ustálená terminologie. Následující tabulka se bude snažit upozornit na terminologické rozdíly mezi mnou a texty, se kterými se můžete setkat.

blokovaný (základní stav procesu, z angl. blocked, někdy waiting)
        Mrázek a někteří další: čekající

uvíznutí, uváznutí, v české literatuře se užívá i angl. deadlock
        (neomezené čekání na událost)
        Mrázek: zablokování

vyhladovění (programy běží ale nemohou vykonat žádanou činnost; z angl.
        starvation)
        Mrázek: umoření

aktivní čekání (busy wait)

s předbíháním (preemptive)

Správa hlavní paměti
====================

* ideální přání programátora - mít nekonečně velkou a rychlou paměť a
  levnou, která je zároveň perzistentní, tj. uchová obsah po vypnutí
* taková paměť ale neexistuje
* většina počítačů má hierarchii pamětí
  - malé množství rychlé cache
  - desítky až stovky MB středně rychlé a středně drahé RAM
  - desítky až stovky GB na pomalých, ale levných a perzistentních discích

* část OS, která spravuje paměť se nazývá správce paměti
  - udržuje informaci, které části paměti se používají a které jsou
    volné
  - alokuje paměť procesům podle potřeby (např. jako důsledek
    Pascalského "new" nebo "malloc" v jazyce C)
  - zařazuje paměť do volné paměti po uvolnění procesem (např. po
    "release" v Pascalu nebo "free" v jazyce C)

* v této části probereme různé mechanismy správy paměti
* postupně od nejjednodušších ("program má veškerou paměť") po
  propracovaná schémata (např. "stránkování se segmentací")

* mechanismy pro správu paměti lze rozdělit do dvou kategorií
  - program je v paměti po celou dobu běhu - "základní mechanismy"
  - programy jsou přesouvány mezi hlavní pamětí a diskem - "mechanismy s
    odkládáním"
* nejdříve probereme základní mechanismy - jsou jednodušší

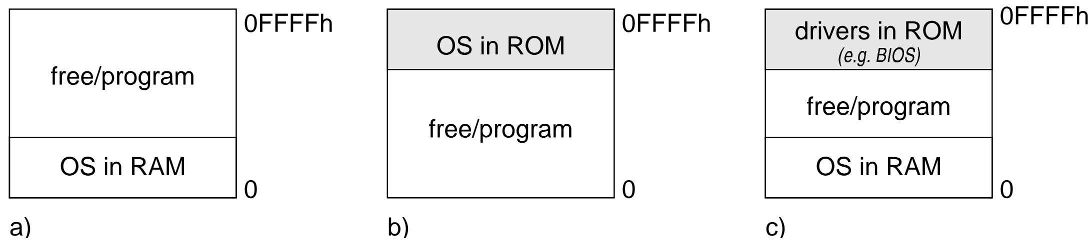
Základní mechanismy pro správu paměti
=====================================

* základní mechanismy = správa paměti bez odkládání a stránkování

Jednoprogramové systémy bez odkládání a stránkování
----

* nejjednodušší - spouštíme pouze jeden program v jednom čase
* uživatel zadá příkaz, OS zavede odpovídající program do paměti
* dovoluje procesu použít veškerou paměť, kterou nepotřebuje OS
* po skončení procesu je možné spustit další proces

* jako příklad tři varianty, jak může vypadat rozdělení paměti:
  - OS je ve spodní části adresního prostoru v RAM, dříve na
    minipočítačích apod.
  - OS je v horní části adresního prostoru v ROM - např. zapouzdřené
    systémy
  - OS je v RAM, ovladače v ROM, dříve osobní počítače (MS DOS v RAM,
    BIOS v ROM)



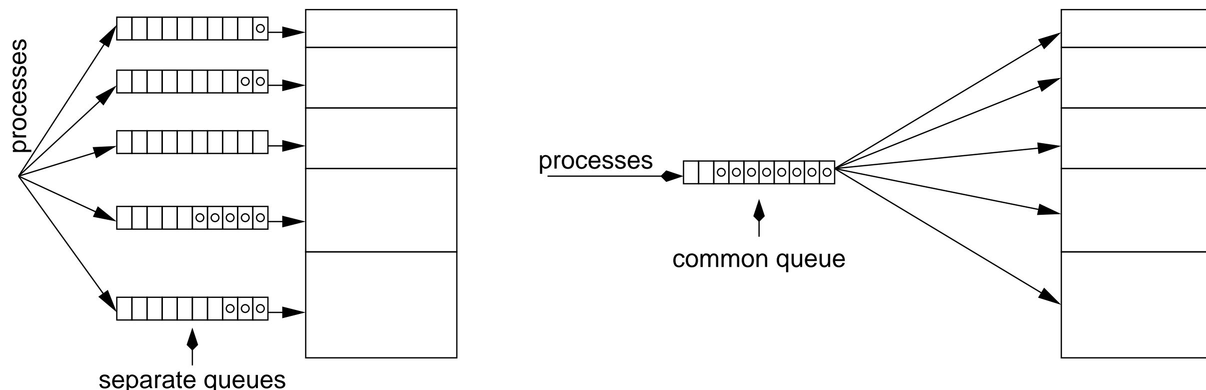
Multiprogramování s pevným přidělením paměti
----

* většina současných systémů dovoluje paralelní nebo pseudoparalelní běh více procesů = multiprogramování
* např. kvůli současné práci více uživatelů, maximálnímu využití CPU apod.

* nejjednodušší schéma - rozdělit paměť na n oblastí (mohou být i různé velikosti)
  - v historických systémech se provádělo např. ručně při startu stroje
  - po načtení úlohy do oblasti je obvykle část oblasti nevyužitá
  - proto snaha umístit úlohu do nejmenší oblasti, do které se vejde

* několik strategií:

1. více front, každá úloha do fronty nejmenší oblasti, kam se vejde
2. jedna fronta - po uvolnění oblasti z fronty vybrat největší úlohu, která se vejde



* v případě (1) se může stát, že existuje prázdná oblast, která se nevyužije, protože úlohy čekají na jiné oblasti
* algoritmus (2) diskriminuje malé úlohy, ale přitom je obvykle dobré poskytnout malým úlohám nejlepší službu
  - jedno řešení je vždy mít malou oblast, kde mohou běžet malé úlohy
  - druhé řešení je s každou úlohou ve frontě sdružit "čítač přeskočení", který bude zvětšen při každém přeskočení úlohy; po dosažení mezní hodnoty už nesmí být úloha přeskočena

Systém s pevným rozdělením paměti (Multiprogramming with Fixed Number of Tasks) se používal např. v systému OS/360.

Poznámka (multiprogramování a velikost paměti)
....

* multiprogramování také zvyšuje využití CPU
  - předpokládejme, že proces stráví část p svého času čekáním na dokončení I/O
  - při n procesech bude pravděpodobnost, že všech n procesů čeká na dokončení I/O p^n
  - tj. využití CPU je u = 1 - p^n
  - např. pokud proces tráví 80% času čekáním, pak p=0.8:

```
n=1  =>  u=0.2     (20% času CPU využito)
n=2  =>  u=0.36    (36%)
```

n=3 => u=0.488 (49%)
n=4 => u=0.5904 (59%)
...
n je tzv. stupeň multiprogramování
(Pozor, je to zjednodušení, protože předpokládáme, že procesy běží navzájem nezávisle, což při jednom CPU není pravda.)

* při multiprogramování je nutné mít všechny procesy (alespoň částečně) zavedeny v paměti, jinak je neefektivní
* výše uvedený předpoklad o výkonnosti stačí pro odhad potřebné velikosti paměti

* např. předpokládejme počítač s 32 MB RAM, OS zabírá 16 MB a každý uživatelský program 4 MB (tj. max. 4 programy v paměti)
* pokud čekání na I/O 80% času, je využití CPU u = 1 - 0.8^4 = 0.5904
* přidáním dalších 16 MB bychom zvýšili stupeň multiprogramování na 8, využití CPU u = 1 - 0.8^8 = 0.83222784
* přidáním dalších 16 MB na 12 procesů, u = 0.9313

* tj. prvním přidáním zvýšíme průchodnost cca 1.4x (tj. o 40%), druhým přidáním 1.12x (tj. o 12%) - druhé přidání paměti se asi nevyplatí

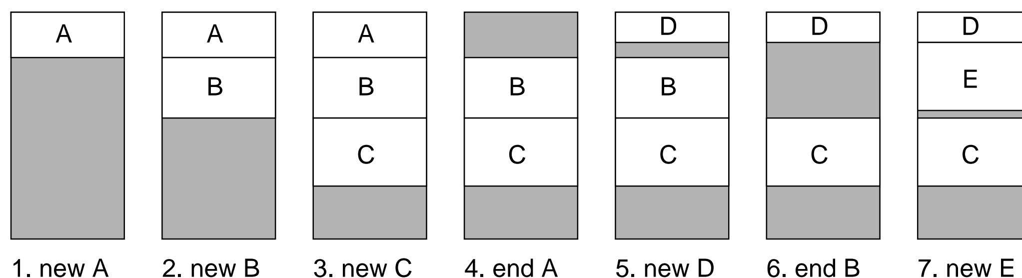
[]

Multiprogramování s proměnnou velikostí oblasti
----

* každé úloze je přidělena paměť podle požadavku
* obsazení paměti se postupně mění, jak úlohy přicházejí a končí

Příklad:

new = proces vytvořen
end = proces ukončen



[Obrázek je zároveň příklad algoritmu first fit, popsaného dále.]

* hlavní rozdíl od pevných oblastí - počet, velikost, umístění oblastí se mění s časem
* zlepšuje využití paměti - nejsme omezeni existujícími oblastmi
* komplikuje alokaci/dealokaci paměti

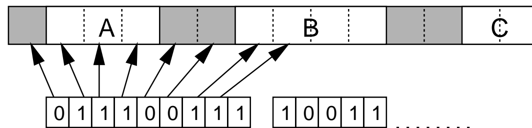
* postupem času může vznikat mnoho malých volných oblastí ("děr")
* teoreticky možné přesunout všechny procesy směrem dolů - kompakce paměti (memory compaction)
* kompakce je ale drahá (např. pokud přesun 1 byte 10 ns, 256 MB 2.7 sec)
* proto se obvykle neprovádí, pokud není k dispozici speciální HW

* OS musí vědět, která paměť je volná a která alokovaná
* nejpoužívanější způsoby zajištění správy paměti jsou:
 - pomocí bitových map
 - pomocí seznamů
 - "buddy systems".

Správa paměti pomocí bitových map
....

* paměť rozdělena do alokačních jednotek stejné délky (velikost od několika bytů do několika KB)

* s každou alokační jednotkou sdružen 1 bit (např. 0 = volno, 1 = obsazeno)



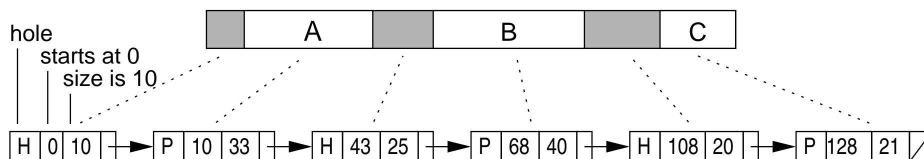
* menší alokační jednotky = větší bitmapa (a obráceně)
* větší alokační jednotky = více nevyužité paměti, protože velikost
  procesu nebude přesně násobek alokační jednotky

Například při velikosti alokační jednotky 4 byty (32 bitů) bude bitová mapa zabírat 1/33 paměti, tj. ani při malé alokační jednotce není neúnosné.

* výhoda - konstantní velikost bitmapy
* nevýhoda - pokud požadujeme úsek paměti velikost N alokačních
  jednotek, musí se v bitmapě vyhledat N po sobě následujících nulových
  bitů - drahá operace - proto se v praxi pro správu hlavní paměti příliš
  často nepoužívá.

Správa paměti pomocí seznamů
....

* myšlenka udržovat seznam alokovaných a volných oblastí ("procesů" a "děr")
* každá položka seznamu obsahuje:
  - informaci, zda se jedná o "proces" (P) nebo "díru" (H)
  - počáteční adresu oblasti
  - délku oblasti



* práce se seznamem
  - pokud proces skončí, P se nahradí H
  - pokud jsou 2H vedle sebe - sloučí se
* seznam je dobré mít seřazený podle počáteční adresy oblasti
* je lepší mít jako dvojitě vázaný seznam - známe polohu nové H, podíváme se
  zda je předchozí nebo následující položka také H (pokud ano, sloučíme)

Jak alokovat paměť pro procesy? Pokud jsou procesy a díry udržovány v seznamu seřazeném podle adresy oblasti, můžeme paměť pro procesy alokovat některým z následujících způsobů:

* algoritmus first fit, česky "první vhodná"
  - nejjednodušší, prohledávání seznamu dokud se nenajde dostatečně velká díra
  - díra se rozdělí na část pro proces a nepoužitou oblast (kromě málo
    pravděpodobného případu, že sedne přesně)
  - algoritmus je rychlý, protože prohledává co nejméně.

* algoritmus next fit, česky "další vhodná"
  - malá změna - prohledávání začne tam, kde skončilo předchozí
  - simulace ukázaly, že next fit je jen o málo horší než first fit

* algoritmus best fit, česky "nejmenší vhodná" nebo "nejlepší volná"
  - prohledá celý seznam, vezme nejmenší díru, do které se proces vejde
  - algoritmus je pomalejší (prohledává celý seznam)
  - podle simulací také více ztracené paměti než first fit nebo next
    fit, protože zaplňuje paměť malými nepoužitelnými dírami.

Mohli bychom ještě uvažovat o variantě worst fit (vždy vybere největší díru), simulace však ukázaly, že to není dobrá myšlenka.

Urychlení algoritmů:

* výše uvedené algoritmy alokace mohou být urychleny oddělenými seznamy pro proces a díry, za cenu složitosti a rychlosti při dealokaci
  - při alokaci prohledáváme pouze seznam děr
  - při dealokaci musí být uvolněná oblast přesunuta ze seznamu procesů do seznamu děr
  - někdy se ale přesto vyplatí (např. v případě alokace paměti pro data přicházející od I/O zařízení - alokace musí být rychlá)

* další optimalizace algoritmu best fit
  - pokud oddělené seznamy, seznam děr může být podle velikosti (od nejmenší do největší)
  - při nalezení první vhodné víme, že je nejmenší vhodná
  - stejně rychlé jako first fit, ale vyšší režie na dealokaci (sousední díry nemusí být sousední v seznamu)

Pokud jsou známy další vlastnosti (resp. pravděpodobnost distribuce požadovaných vlastností) jsou možné další algoritmy.

Poznámka pro zajímavost (algoritmus quick fit)
....

Algoritmus quick fit udržuje samostatné seznamy pro díry některých častěji požadovaných délek. Jako centrální datovou strukturu můžeme mít tabulku, jejíž první položka ukazuje na hlavu seznamu děr velikosti 4 KB, další 8 KB, ... např. díry ostatních velikostí mohou být v samostatném seznamu. Alokace požadované velikosti je velmi rychlá, ale opět je obtížné sdružování sousedů.

[]

* jak ušetřit paměť
  - místo samostatného seznamu děr mohou být využity díry
  - první slovo díry může obsahovat velikost díry, druhé ukazatel na na další díru
  - takto funguje např. alokátor paměti pro proces v jazyce C pod UNIXem (používá algoritmus next fit, viz příklad 8.7 v [Kernighan & Ritchie])

Poznámka (asymetrie mezi procesy a dírami)

Dvě sousedící volné oblasti (H) se sloučí, zatímco dva procesy (P) se nesloučí - při normálním běhu je počet H poloviční oproti počtu P.

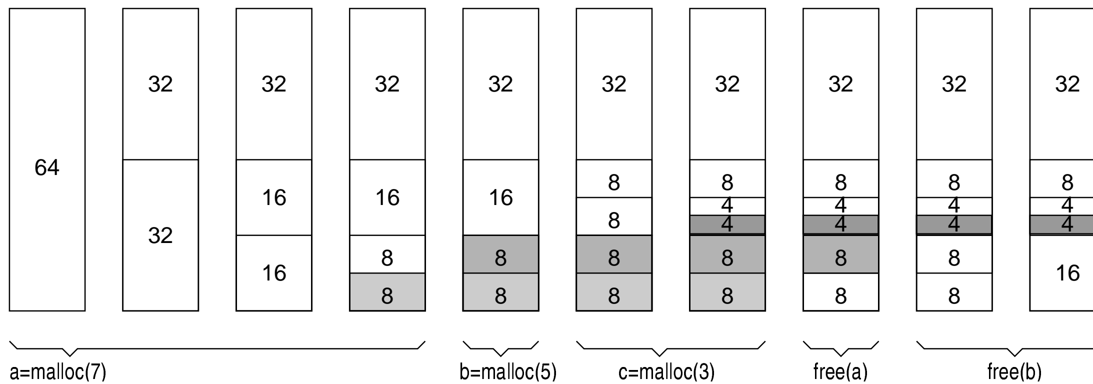
[]

Mechanismus "buddy system"
....

* mějme seznamy volných bloků velikosti 1, 2, 4, 8, 16 ... alokačních jednotek až do seznamu bloků velikosti celé paměti
* na začátku veškerá paměť volná, všechny seznamy jsou prázdné kromě seznamu obsahujícího 1 položku velikosti paměti

* např. alokační jednotka 1 KB, paměť velikosti 64 KB (7 seznamů)

* přijde-li požadavek, zaokrouhlí se nahoru na mocninu 2 (např. požadavek na 7 KB se zaokrouhlí na 8 KB)
* blok velikosti 64 KB se rozdělí na 2 bloky velikosti 32 KB, tzv. "buddies"
* protože ještě moc velké, jeden z nich (ten s nižší adresou) se rozdělí na 2 bloky 16 KB, jeden z nich na 2 bloky 8 KB, jeden z nich alokován



* požadavek na 5 KB (tj. 8 KB), je přidělen druhý 8 KB blok
* požadavek na 3 KB (tj. 4 KB), další 16 KB blok se rozdělí na 8+8 K,
  nižší 8 K blok se rozdělí na 4+4 K, nižší se přidělí
  (tj. nejmenší dostatečně velký blok se rozdělí)

* uvolnění paměti: pokud jsou volné oba sousední bloky stejné velikosti
  ("buddies"), spojí se do jednoho většího bloku

Neefektivní ve využití paměti (chci-li 9 K, dostanu 16 K), ale rychlý:
* alokace paměti - vyhledání v seznamu dostatečně velkých děr
* slučování - vyhledání buddy.

Poznámka:

Výše uvedená schémata (pevné přidělení, proměnné oblasti) se používala v historických dávkových systémech. V moderních univerzálních systémech se uvedená schémata pro alokaci paměti procesům nepoužívají, protože je zastupuje mechanismus virtuální paměti (o kterém budeme hovořit zanedlouho).

Uvedené algoritmy (first fit, ...) však mají obecnější platnost. V současných systémech se tyto algoritmy používají pro alokaci paměti uvnitř jádra OS nebo uvnitř procesu.

Příklad: Jádro systému Linux běží ve fyzické paměti, pro správu paměti jádra používá buddy system.

Příklad: Fce malloc(3) v jazyce C žádá OS o větší blok paměti a získanou paměť pak aplikaci přiděluje např. algoritmem first fit.

Algoritmy mají ovšem obecnější platnost a lze je používat i v jiných kontextech.

Příklad: Linux spravuje pomocí odkládací prostor pomocí bitové mapy algoritmem next fit.

[]

❇