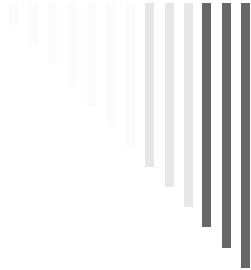


---




08.  
Deadlock  
Přidělování paměti

ZOS 2013, L. Pešička

---

---



Obsah

- Deadlock
  - Jak předcházet, detekovat, reagovat
- Metody přidělování paměti

---



## Jak se vypořádat 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 dříve uvedených nutných podmínek pro vznik uvíznutí



## 1. Ignorování problému

- Předstíráme, že problém neexistuje ☺
  - „pštroší algoritmus“
- Vysoká cena za eliminaci uvíznutí
  - Např. činnost uživatelských procesů je omezena
  - Neexistuje žádné univerzální řešení
- Žádný ze známých OS se nezabývá uvíznutím uživatelských procesů
  - Snaha o eliminaci uvíznutí pro činnosti jádra

U uživatelských procesů uvíznutí neřešíme, snažíme se, aby k uvíznutí nedošlo v jádře OS

## 2. Detekce a zotavení

- Systém se nesnaží zabránit vzniku uvíznutí
- Detekuje uvíznutí
- Pokud nastane, provede akci pro zotavení
  
- Detekce pro 1 zdroj každého typu
  - Při žádostech o zdroj OS konstruuje graf alokace zdrojů
  - Detekce cyklu – pozná, zda nastalo uvíznutí
  - Různé algoritmy detekce cyklu (teorie grafů)
    - Např. prohledávání do hloubky z každého uzlu, dojdeme-li do uzlu, který jsme již prošli - cyklus

Samotná detekce uvíznutí nemusí být snadná

## Zotavení z uvíznutí (pokračování 2.)


### Zotavení pomocí preempce

- Vlastníkovi zdroj dočasně odejmout
- Závisí na typu zdroje – často obtížné či nemožné
  - Tiskárna – po dotištění stránky proces zastavit, ručně vyjmout již vytištěné stránky, odejmout procesu a přiřadit jinému



## Zotavení z uvíznutí – zrušení změn

- Zotavení pomocí zrušení změn (rollback)
  - **Častá uvíznutí** – checkpointing procesů  
= zápis stavu procesů do souboru, aby proces mohl být v případě potřeby vrácen do uloženého stavu
  - **Detekce uvíznutí** – nastavení na dřívější checkpoint, kdy proces ještě zdroje nevlastnil (následná práce ztracena)
  - Zdroj přiřadíme uvízlému procesu – zrušíme deadlock
  - Proces, kterému jsme zdroj odebrali – pokusí se ho alokovat - usne




## Zotavení z uvíznutí – zrušení procesu

- Zotavení pomocí zrušení procesu
  - Nejhorší způsob – zrušíme jeden nebo více procesů
  - Zrušit proces v cyklu
    - Pokud nepomůže zrušit jeden, zrušíme i další
- Často alespoň snaha zrušit procesy, které je možné spustit 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 pozastaví žádající proces
- Stav je bezpečný, pokud existuje alespoň jedna posloupnost, ve které mohou procesy doběhnout bez uvíznutí
- I když stav není bezpečný, uvíznutí nemusí nutně nastat



### Bankéřův algoritmus pro jeden typ zdroje

- Předpokládáme více zdrojů stejného typu
  - Např. N magnetopáskových jednotek
- Algoritmus plánování, který se dokáže vyhnout uvíznutí (Dijkstra 1965)
- Bankéř na malém městě, 4 zákazníci – A, B, C, D
- Každému garantuje půjčku (6, 5, 4, 7) = 22 dohromady
- Bankéř ví, že všichni zákazníci nebudou chtít půjčku současně, pro obsluhu zákazníků si ponechává pouze 10

## Bankéřův algoritmus

Zákazník	Má půjčeno	Max. půjčka
A	1	6
B	1	5
C	2	4
D	4	7

Bankéř má volných prostředků:  $10 - (1+1+2+4) = 2$

Stav je bezpečný, bankéř může pozastavit všechny požadavky kromě C

Dá C 2 jednotky, C skončí a uvolní 4, může použít pro D nebo B atd.

## Bankéřův algoritmus (B o 1 více)

Zákazník	Má půjčeno	Max. půjčka
A	1	6
B	2	5
C	2	4
D	4	7

Dáme B o jednotku více; zůstane nám volných prostředků: 1

Stav není bezpečný – pokud všichni budou chtít maximální půjčku, bankéř nemůže uspokojit žádného – nastalo by uvíznutí

Uvíznutí nemusí nutně nastat, ale s tím bankéř nemůže počítat ...

## Rozhodování bankéře

Zkusí „jako by“ přidělit zdroj a zkoumá, zda je nový stav bezpečný

- U každého požadavku – zkoumá, zda vede k bezpečnému stavu:
- Bankéř předpokládá, že požadovaný zdroj byl procesu přiřazen a že všechny procesy požádaly o všechny bankéřem garantované zdroje
- Bankéř zjistí, zda je dostatek zdrojů pro uspokojení některého zákazníka; pokud ano – předpokládá, že zákazníkovi byla suma vyplacena, skončil a uvolnil (vrátil) všechny zdroje
- Bankéř opakuje předchozí krok, pokud mohou všichni zákazníci skončit, je stav bezpečný

## Vykonání požadavku

- Proces požaduje nějaký zdroj
- Zdroje jsou poskytnuty pouze tehdy, pokud požadavek vede k bezpečnému stavu
- Jinak je požadavek odložen na později – proces je pozastaven

## Bankéřův algoritmus pro více typů zdrojů

- zobecněn pro více typů zdrojů
- používá dvě matice (sloupce – třídy zdrojů, řádky – zákazníci)
  - 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

	Zdroj R	Zdroj S	Zdroj T
Zák. A	3	0	1
Zák. B	0	1	0
Zák. C	1	1	1
Zák. D	1	1	0

Matice přiřazených zdrojů

	Zdroj R	Zdroj S	Zdroj T
Zák. A	1	1	0
Zák. B	0	1	1
Zák. C	3	1	0
Zák. D	0	0	1

Matice ještě požadovaných zdrojů

zavedeme vektor A volných zdrojů (available resources)


např.  $A = (1, 0, 1)$  znamená jeden volný zdroj typu R, 0 typu S, 1 typu T





## Určení, zda je daný stav bezpečný

1. V matici ještě požadovaných zdrojů hledáme řádek, který je menší nebo roven A.  
Pokud neexistuje, nastalo by uvíznutí.
2. Předpokládáme, 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)



## Bankéřův algoritmus & použití v praxi

- publikován 1965, uváděn ve všech učebnicích OS
- v praxi v podstatě nepoužitelný
  - procesy obvykle nevědí dopředu, jaké budou jejich maximální požadavky na zdroje
  - počet procesů není konstantní (uživatelé se přihlašují, odhlašují, spouštějí procesy, ...)
  - zdroje mohou zmizet (tiskárně dojde papír ...)
- nepoužívá se v praxi pro zabránění uvíznutí
- odvozené algoritmy lze použít pro detekci uvíznutí při více zdrojích stejného typu




## 4. Prevence uvíznutí

jak skutečné systémy zabraňují uvíznutí?  
viz 4 Coffmanovy podmínky vzniku uvíznutí

1. vzájemné vyloučení – výhradní přiřazení zdrojů
2. hold and wait – proces držící zdroje může požadovat další
3. nemožnost zdroje odejmout
4. cyklické čekání

pokud některá podmínka nebude splněna – uvíznutí strukturálně nemožné



## P1 – Vzájemné vyloučení

- prevence – zdroj nikdy nepřidat výhradně
- problém pro některé zdroje (tiskárna)
- spooling
  - pouze daemon přistupuje k tiskárně
  - nikdy nepožaduje další zdroje – není uvíznutí
- spooling není možný pro všechny zdroje (záznamy v databázi)
- převádí soutěžení o tiskárnu na soutěžení o diskový prostor – 2 procesy zaplní disk, žádný nemůže skončit



## P2- Hold and wait

- proces držící výhradně přiřazené zdroje může požadovat další zdroje
- požadovat, aby procesy alokovaly všechny zdroje před svým spuštěním
  - většinou nevědí, které zdroje budou chtít
  - příliš restriktivní
  - některé dávkové systémy i přes nevýhody používají, zabraňuje deadlocku
- pokud proces požaduje nové zdroje, musí uvolnit zdroje které drží a o všechny požádat v jediném požadavku



## P3 – Nemožnost zdroje odejmout

- odejímat zdroje je velmi obtížné



## P4 – Cyklické čekání

- Proces může mít jediný zdroj, pokud chce jiný, musí předchozí uvolnit – restriktivní, není řešení ☹
- Všechny zdroje očíslovány, požadavky musejí být prováděny v číselném pořadí
  - Alokační zdroj nemůže mít cykly
  - Problém – je těžké nalézt vhodné očíslování pro všechny zdroje
  - Není použitelné obecně, ale ve speciálních případech výhodné (jádro OS, databázový systém, ...)



## Př. Dvoufázové zamykání

- V DB 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
- Zamyká se vždy v číselném pořadí, uvíznutí nemůže nastat



## Shrnutí přístupu k uvíznutí (!)

- Ignorování problému – většina OS ignoruje uvíznutí uživatelských procesů
- Detekce a zotavení – pokud uvíznutí nastane, detekujeme a něco s tím uděláme (vrátíme čas – rollback, zrušíme proces ...)
- Dynamické zabránění – zdroj přiřadíme, pouze pokud bude stav bezpečný (bankéřův algoritmus)
- Prevence – strukturálně negujeme jednu z Coffman. podmínek
  - Vzájemné vyloučení – spooling všeho
  - Hold and wait – procesy požadují zdroje na začátku
  - Nemožnost odejmání – odejmi (nefunguje)
  - Cyklické čekání – zdroje očíslováme a žádáme v číselném pořadí



## Vyhladovění

- Procesy požadují zdroje – pravidlo pro jejich přiřazení
- Může se stát, že některý proces zdroj nikdy neobdrží
  - I když nenastalo uvíznutí !
- Př. Večeřící filozofové
  - Každý zvedne levou vidličku, pokud je pravá obsazena, levou položí
  - Vyhladovění, pokud všichni zvedají a pokládají současně



## Vyhladovění 2

- Př. Přiřazování zdroje strategií SJF
  - Tiskárnu dostane proces, který chce vytisknout nejkratší soubor
  - 1 proces chce velký soubor, hodně malých požadavků – může dojít k vyhladovění, neustále předbíhán
- Řešení – FIFO
- Řešení – označíme požadavek časem příchodu a při překročení povolené doby setrvání v systému bude obslužen



## Terminologie

- Blokový (blocked, waiting), někdy: čekající
  - Základní stav procesu
- Uváznutí, uváznutí, deadlock, někdy: zablokování
  - Neomezené čekání na událost
- Vyhladovění, starvation někdy: umoření
  - Procesy běží, ale nemohou vykonávat žádnou činnost
- Aktivní čekání (busy wait), s předbíháním (preemptive)



## Bernsteinovy podmínky



## Windows – ukázky funkcí

### Správa vláken

CreateThread()

SuspendThread(), ResumeThread()


ExitThread() // ukončení vlákna

TerminateThread() // ukončí jiné vlákno

WaitForSingleObject() // čeká na jeden

WaitForMultipleObjects() // čeká na 1 nebo všechny

CloseHandle()



## Windows - synchronizace

Kritické sekce


InitializeCriticalSection()  
DeleteCriticalSection()

EnterCriticalSection()  
LeaveCriticalSection()

viz dokumentace:  
kritickou sekci mohou  
využít pouze vlákna  
stejného procesu

optimalizovanější

---



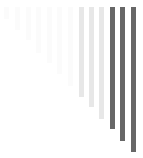
## Windows - synchronizace

Mutexy

CreateMutex()  
OpenMutex()  
WaitForSingleObject() // čekáme na mutex  
WaitForMultipleObjects()  
ReleaseMutex() // uvolníme mutex  
CloseHandle()

---





## Windows - semaforey

### Semaforey

```
CreateSemaphore(), // inic. hodnota, max. hodnota  
WaitForSingleObject(), // operace P()  
WaitForMultipleObjects()  
ReleaseSemaphore(), // operace V()  
CloseHandle()
```


<http://msdn.microsoft.com/en-us/library/windows/desktop/ms686946%28v=vs.85%29.aspx>



## Windows - synchronizace

### Eventy


```
CreateEvent()  
SetEvent()  
ResetEvent()  
WaitForSingleObject()  
WaitForMultipleObjects()  
CloseHandle()
```



poslání signálu  
vlákn

indikuje, že  
nějaká událost  
nastala

---




## Windows

Atomické operace

InterlockedIncrement() // inkrementuje o 1  
InterlockedDecrement()  
InterlockedExchange() // nastaví novou hodnotu  
// a vrátí původní

---

---



## Windows

priorita vláken

SetThreadPriority()  
GetThreadPriority()

---



## Osnova

### Základní moduly OS

- Modul pro správu procesů - probráno
- Modul pro správu paměti - nyní začínáme
- Modul pro správu periférií
- Modul pro správu souborů



## Správa hlavní paměti

- Ideál programátora
  - Paměť nekonečně velká, rychlá, levná
  - Zároveň persistentní (uchovává obsah po vypnutí napájení)
  - Bohužel neexistuje
- Reálný počítač – hierarchie pamětí („pyramida“)
  - Registry CPU
  - Malé množství rychlé cache paměti
  - Stovky MB až gigabajty RAM paměti
  - GB na pomalých, levných, persistentních discích



## Správce paměti

- Část OS, která spravuje paměť
- Udržuje informaci, které části paměti se používají a které jsou volné
- Alokuje paměť procesům podle potřeby
  - funkce **malloc** v jazyce C, (new v Pascalu)
- Zařazuje paměť do volné paměti po uvolnění procesem
  - funkce **free** v jazyce C, (release v Pascalu)



## Jak to reálně funguje? (!!)

- proces požádá o alokaci n bajtů paměti funkcí ukazatel = malloc (n)
- malloc je knihovní fce alokátoru paměti (součást glibc)
- paměť je alokována z haldy (heapu) !
- alokátor se podívá, zda má volnou paměť k dispozici, když ne, požádá OS o přidělení dalších stránek paměti (systémové volání sbrk)
- proces uvolní paměť, když už ji nepotřebuje voláním free(ukazatel)

## Příklad alokace

zkuste: man malloc

Příklad:

1. proces bude chtít alokovat 500B, zavolá malloc
2. alokátor koukne, nemá volnou paměť, požádá OS o přidělení stránky paměti (4KB) – sbrk
3. proces je obslužen, dostane paměť
4. proces bude chtít dalších 200B, zavolá malloc
5. alokátor už má paměť v zásobě, rovnou ji přidělí procesu
6. když už proces paměť nepotřebuje, zavolá free

## man malloc

Windows:  
také malloc() nebo HeapAlloc()

```
MALLOC(3)                               Linux Programmer's Manual                               MALLOC(3)
NAME
  calloc, malloc, free, realloc - Allocate and free dynamic memory
SYNOPSIS
  #include <stdlib.h>

  void *calloc(size_t nmemb, size_t size);
  void *malloc(size_t size);
  void free(void *ptr);
  void *realloc(void *ptr, size_t size);
DESCRIPTION
  calloc() allocates memory for an array of nmemb elements of size bytes
  each and returns a pointer to the allocated memory. The memory is set
  to zero. If nmemb or size is 0, then calloc() returns either NULL, or
  a unique pointer value that can later be successfully passed to free().

  malloc() allocates size bytes and returns a pointer to the allocated
  memory. The memory is not cleared. If size is 0, then malloc()
  returns either NULL, or a unique pointer value that can later be suc-
  cessfully passed to free().

  free() frees the memory space pointed to by ptr, which must have been
  returned by a previous call to malloc(), calloc() or realloc(). Other-
  wise, or if free(ptr) has already been called before, undefined behav-
  ior occurs. If ptr is NULL, no operation is performed.
```



## poznámka k pointerům

ukazatel = malloc (size)

takto získaný ukazatel obsahuje virtuální adresu, tj. není to přímo adresa do fyzické paměti (RAM) !!

virtuální adresa se uvnitř procesoru převede na fyzickou adresu (s využitím tabulky stránek atd.)




## Mechanismy správy paměti

Od nejjednodušších (program má veškerou paměť) po propracovaná schémata (stránkování se segmentací)

Dvě kategorie:

- Základní mechanismy
  - Program je v paměti po celou dobu svého běhu
- Mechanismy s odkládáním
  - Programy přesouvány mezi hlavní pamětí a diskem



## Základní mechanismy pro správu paměti

Nejprve probereme základní mechanismy  
Bez odkládání a stránkování

1. Jednoprogramové systémy
2. Multiprogramování s pevným přidělením paměti
3. Multiprogramování s proměnnou velikostí oblasti



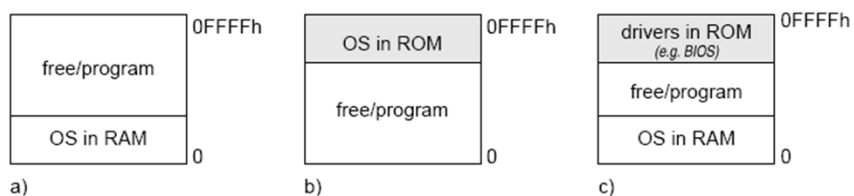
## Jednoprogramové systémy

- Spouštíme pouze jeden program v jednom čase
- Uživatel – zadá příkaz , OS zavede program do paměti
- Dovoluje použít veškerou paměť, kterou nepotřebuje OS
- Po skončení procesu lze spustit další proces

Tři varianty rozdělení paměti:

- a) OS ve spodní části adresního prostoru v RAM (minipočítače)
- b) OS v horní části adresního prostoru v ROM (zapouzdřené systémy)
- c) OS v RAM, ovladače v ROM  
(na PC – MS DOS v RAM, BIOS v ROM)

## Jednoprogramové systémy



## Multiprogramování s pevným přidělením paměti

- Většina současných systémů – paralelní nebo pseudoparalelní běh více programů = multiprogramování
- Práce více uživatelů, maximalizace využití CPU apod.
- Nejjednodušší schéma – rozdělit paměť na  $n$  oblastí (i různé velikosti)
  - V historických systémech – rozdělení ručně při startu stroje
  - Po načtení úlohy do oblasti je obvykle část oblasti nevyužitá
  - Snaha umístit úlohu do nejmenší oblasti, do které se vejde

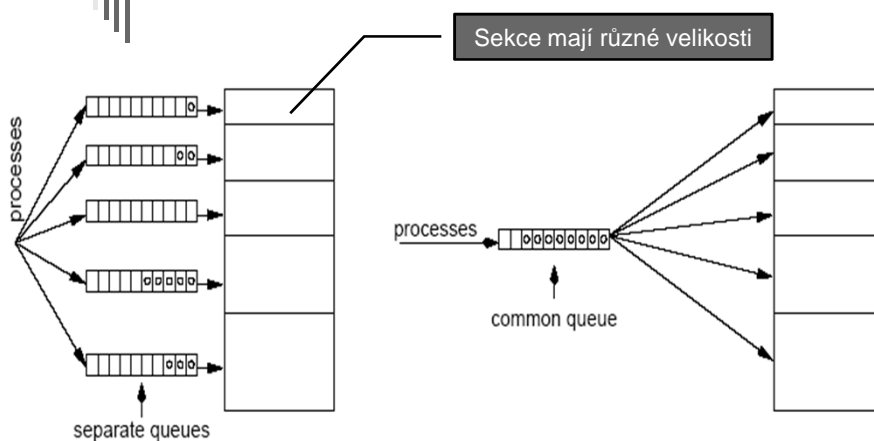


## Pevné rozdělení sekcí

Několik strategií:

1. Více front, každá úloha do nejmenší oblasti, kam se vejde
2. Jedna fronta – po uvolnění oblasti z fronty vybrat největší úlohu, která se vejde

## Pevné rozdělení sekcí





## Pevné rozdělení sekcí - vlastnosti

- Strategie 1.
  - Může se stát, že existuje neprázdná oblast, která se nevyužije, protože úlohy čekají na jiné oblasti
- Strategie 2.
  - Diskriminuje malé úlohy (vybíráme největší co se vejde) x malým bychom měli obvykle poskytnout nejlepší službu
  - Řešení – mít vždy malou oblast, kde poběží malé úlohy
  - Řešení – s každou úlohou ve frontě sdružit „čítač přeskočení“, bude zvětšen při každém přeskočení úlohy; po dosažení mezní hodnoty už nesmí být úloha přeskočena



## Pevné rozdělení sekcí - poznámky

- Používal např. systém OS/360 (Multiprogramming with Fixed Number of Tasks)
- Multiprogramování zvyšuje využití CPU
- Proces – část času p tráví čekáním na dokončení I/O
- N procesů – pst, že **všechny** čekají na I/O je:  $p^n$
- Využití CPU je  $u = 1 - p^n$



## Poznámky

- Využití CPU je  $u = 1 - p^n$
- Pokud proces tráví 80% času čekáním,  $p = 0.8$
- $n = 1$       ...       $u = 0.2$  (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
- Zjednodušení, předpokládá nezávislost procesů, což při jednom CPU není pravda



## Poznámky

- Při multiprogramování – všechny procesy je nutné mít alespoň částečně zavedeny v paměti, jinak neefektivní
- Odhad velikosti paměti
- Fiktivní PC 32MB RAM, OS 16MB, uživ. programy po 4MB
  - Max. 4 programy v paměti
- Čekání na I/O 80% času, využití CPU  $u = 1 - 0.8^4 = 0.5904$
- Přidáme 16MB RAM, stupeň multiprogramování  $n$  bude 8
  - Využití CPU  $u = 1 - 0.8^8 = 0.83222784$
- Přidání dalších 16MB – 12 procesů,  $u = 0.9313$
- První přidání zvýší průchodnost 1.4x (o 40%)  
další přidání 1.12x (o 12%) – druhé přidání se tolik nevyplatí

## Multiprogramování s proměnnou velikostí oblastí

- Úloze je přidělena paměť dle požadavku
- V čase se mění
  - Počet oblastí
  - Velikost oblastí
  - Umístění oblastí
- Zlepšuje využití paměti
- Komplikovanější alokace / dealokace

## Př.: IBM OS/360




batch processing  
operating system

1964


zdroj obrázku:

[http://www.maximumpc.com/article/features/ibm\\_os360\\_windows\\_31\\_s\\_ofirmware\\_changed\\_computing\\_forever](http://www.maximumpc.com/article/features/ibm_os360_windows_31_s_ofirmware_changed_computing_forever)



## IBM OS/360


- **Single Sequential Scheduler (SSS)**
  - Option 1
  - Primary Control Program (PCP)
- **Multiple Sequential Schedulers (MSS)**
  - Option 2
  - Multiprogramming with a Fixed number of Tasks (MFT)
  - MFT 2
- **Multiple Priority Schedulers (MPS)**
  - Option 4
  - VMS<sup>[NB 1]</sup>
  - Multiprogramming with a Variable number of Tasks (MVT)
  - Model 65 Multiprocessing (M65MP)



zdroj:  
<http://www.escapistmagazine.com/forums/read/18.85690-Esoteric-Operating-Systems-The-History-of-OS-360-and-its-successors>

---

zdroj: [http://en.wikipedia.org/wiki/OS/360\\_and\\_successors](http://en.wikipedia.org/wiki/OS/360_and_successors)



## Problém mnoha volných oblastí

- **Může vzniknout mnoho volných oblastí (děr)**
  - Paměť se „rozdrobí“
  
- **Kompaktace paměti (compaction)**
  - Přesunout procesy směrem dolů
  - Drahá operace (1B .. 10ns, 256MB .. 2.7s)
  - Neprovádí se bez speciálního HW

---

## Volná x alokovaná paměť

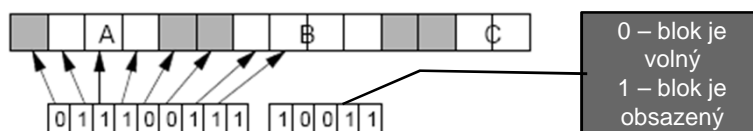
Pro zajištění správy paměti se používají:

1. bitové mapy
2. seznamy
  - first fit, best fit, next fit, ...
3. buddy systems

U každého bloku paměti potřebujeme rozhodnout, zda je volný nebo někomu přidělený

## Správa pomocí bitových map

- Paměť rozdělíme na alokační jednotky stejné délky (B až KB)
- S každou jednotkou 1bit (volno x obsazeno)



Menší alokační jednotky – větší bitmapa

Větší jednotky – více nevyužitá paměť

Alokační jednotka 4 byty (32bitů):

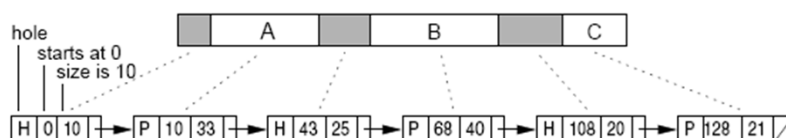
na každých 32bitů paměti potřebujeme 1bit signalační  
tedy .. 1/33 paměti

## Bitové mapy

- + konstantní velikost bitové mapy
- najít požadovaný úsek  $N$  volných jednotek
  - Náročné, příliš často se nepoužívá pro tento účel

## Správa pomocí seznamů

- Seznam alokovaných a volných oblastí (procesů, děr)
- Položka seznamu:
  - Info o typu – proces nebo díra (P vs. H)
  - Počáteční adresa oblasti
  - Délka oblasti





## Práce se seznamem

- Proces skončí – P se nahradí H (dírou)
- Dvě H vedle sebe – sloučí se

Seznam seřazený podle počáteční adresy oblasti

Může být obousměrně vázaný seznam

– snadno k předchozí položce

Jak prohledávat seznam, když proces potřebuje alokovat paměť?



## Alokace – first fit, next fit

- First Fit (první vhodná)
  - Prohledávání, dokud se nenajde dostatečně velká díra
  - Díra se rozdělí na část pro proces a nepoužitou oblast (většinou „nesedne“ přesně)
  - Rychlý, prohledává co nejméně
- Next Fit (další vhodná)
  - Prohledávání začne tam, kde skončilo předchozí
  - O málo horší než *first fit*





## Alokace best fit

- Best fit (nejmenší/nejlepší vhodná)
  - Prohlédne celý seznam, vezme nejmenší díru, do které se proces vejde
  - Pomalejší – prochází celý seznam
  - Více ztracené paměti než FF,NF – zaplňuje paměť malými nepoužitelnými dírami
  
- Worst fit (největší díra) – není vhodné
  - nepoužívá se



## Urychlení

- Oddělené seznamy pro proces a díry
  - Složitější a pomalejší dealokace
  - Vyplatí se při rychlé alokaci paměti pro data z I/O zařízení
  - *Alokace* – jen seznam děr
  - *Dealokace* – složitější – přesun mezi seznamy, z děr do procesů
- Oddělené seznamy, seznam děr dle velikosti
  - Optimalizace best fitu
  - První vhodná – je i nejmenší vhodná, rychlost First fitu
  - Režie na dealokaci – sousední fyzické díry nemusí být sousední v seznamu



## Další varianty – Quick Fit

### Quick Fit

- Samostatné seznamy děr nejčastěji požadovaných délek
- Díry velikosti 4KB, 8KB,...
- Ostatní velikosti v samostatném seznamu
- Alokace – rychlá
- Dealokace – obtížné sdružování sousedů



## Šetření paměti

- Místo samostatného seznamu děr lze využít díry
- Obsah díry
  - 1. slovo – velikost díry
  - 2. slovo – ukazatel na další díru

Např. alokátor paměti pro proces v jazyce C pod Unixem používá strategii next fit (viz ukázka malloc dříve)



## KVIZ

Jaký je vzájemný poměr počtu  
děr a procesů?

Předpokládejme, že pro daný proces  
alokujeme paměť jednorázově (v celku)



## Asymetrie mezi procesy a dírami

- Dvě sousední díry (H) se sloučí
- Dva procesy (P) se nesloučí


Při normálním běhu je počet děr poloviční  
oproti počtu procesů



## Opakování

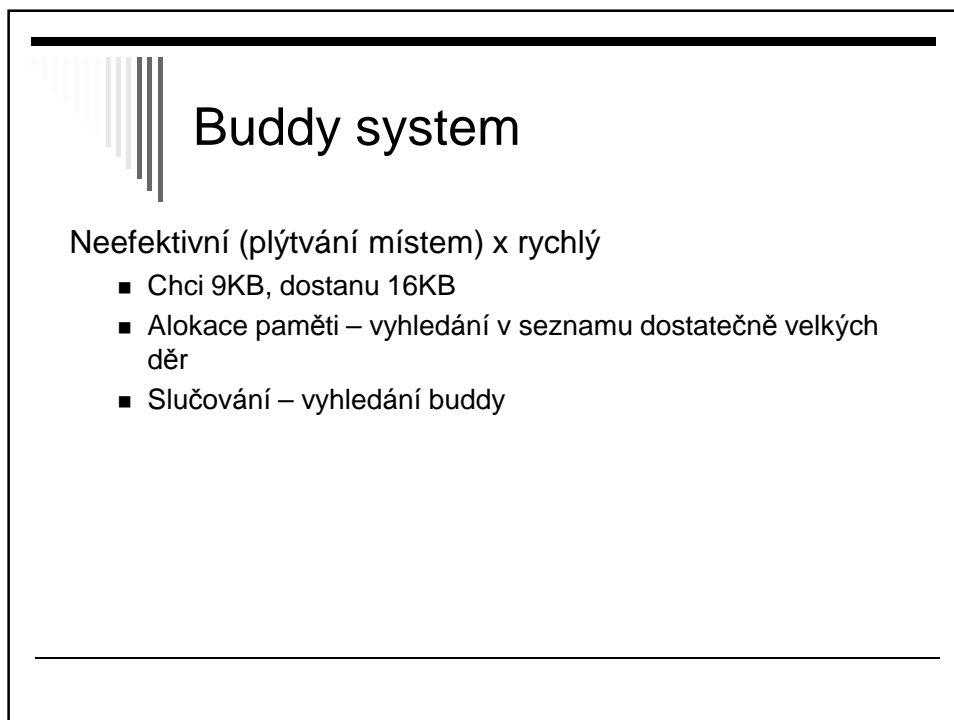
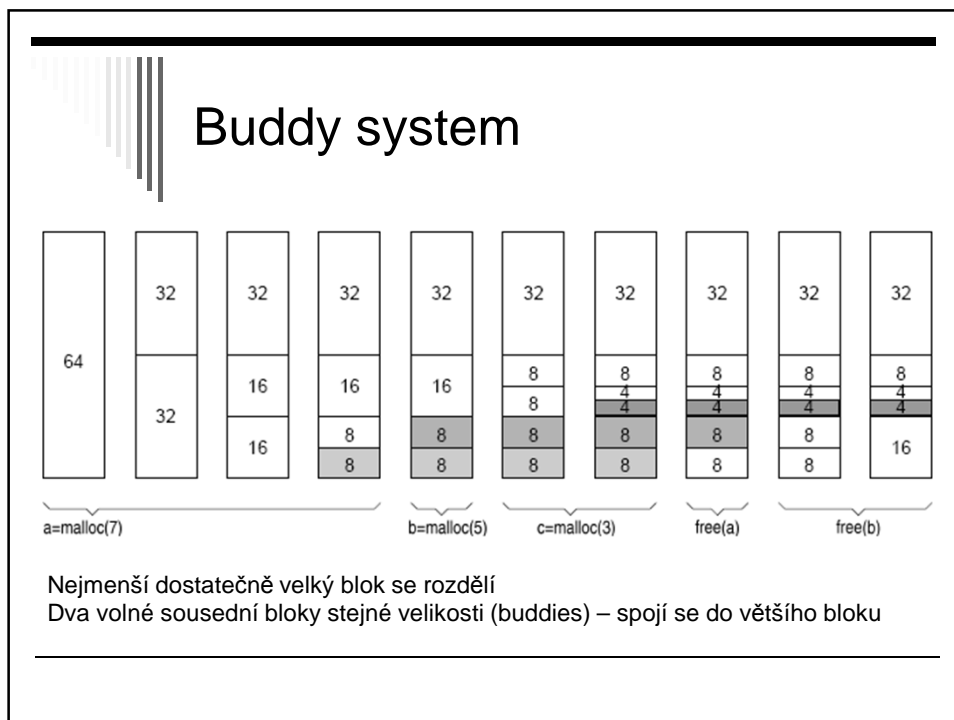
Správa paměti:

1. Bitové mapy
2. Seznamy ( first fit, ...)
3. Buddy systems



## Buddy systems

- Seznamy volných bloků 1, 2, 4, 8, 16 ... alokačních jednotek až po velikost celé paměti
- Nejprve seznamy prázdné vyjma 1 položky v seznamu o velikosti paměti
- Příklad: Alokační jednotka 1KB, paměť velikosti 64KB
- Seznamy 1, 2, 4, 8, 16, 32, 64 (7 seznamů)
- Požadavek se zaokrouhlí na mocninu dvou nahoru
  - například požadavek 7KB na 8KB
- Blok 64KB se rozdělí na 2 bloky 32KB (buddies) a dělíme dále...



## Použití algoritmů

U řady algoritmů můžeme pozorovat, že se nepoužívají ke svému „původnímu účelu“, tj. ke správě hlavní paměti, ale používají se pro řešení dílčích úkolů.

Např. runtime knihovna požádá OS o přidělení stránky paměti, a získanou oblast dále přiděluje procesu, když si o ní zažádá funkcí `malloc()` – a zde se uplatní další strategie správy paměti

## Použití algoritmů

- Přidělení paměti procesům
  - dnes mechanismy virtuální paměti
- Další oblasti použití
  - přidělování paměti uvnitř jádra nebo uvnitř procesu

### Buddy system

Jádro Linuxu běží ve fyzické paměti, pro správu paměti jádra používá buddy system

viz: `cat /proc/buddyinfo`



## Použití algoritmů

Použití first fit, next fit:

Fce malloc v jazyce C

žádá OS o větší blok paměti a získanou paměť pak aplikaci přiděluje algoritmem first fit či next fit

Správa odkládacího prostoru -

Linux spravuje odkládací prostor pomocí bitové mapy algoritmem next fit