

Procesy a thready

Proces

Vznik procesu

1. pri štarte OS
2. iným procesom
3. ako odozva na akciu užívateľa
4. vykonanie dávky

Stavy procesu

Pripravený
Čakajúci
Bežiaci

nepreemptívny (kooperatívny) OS

- nie je schopný násilne previesť proces zo stavu „bežiaci“ na „pripravený“ - proces oznámi plánovaču, že má vhodnú chvíľku aby bol prerušený
- napr. Win 3.0/Win 3.1

preemptívny OS

- je schopné previesť bežiaci proces na „pripravený“ (aby mohol začať vykonávať aj iný proces)
- potrebuje nejaký časovač, ktorý bude generovať prerušenia
- z hľadiska procesu: procesu môže byť hocikedy odobratý strojový čas (hocikedy – v čase prerušenia, ale proces nevie kedy bude prerušenie)

Spôsoby plánovania procesov

1. Dávkové spracovanie

- používa sa pri rutinných procesoch obvykle nad veľkým množstvom dát
- nie je to interaktívne – zaujíma nás len výsledok
- procesy nie je nutné vymieňať tak často (výmena procesov je časovo pomerne náročná a počas tohto času sa vlastne „nič“ nedeje)

2. Interaktívne systémy

- v súčasnosti najrozšírenejšie – užívateľ očakáva nejakú odozvu
- jeden proces nesmie dlho zdržovať ostatné

3. Real-time systémy

- systémy, ktoré riadia niečo, čo je mimo nich
- procesy musia stihnúť deadlines – inak nastane prúser (výrobná linka sa zastaví moc neskoro, ...)
- narozdiel od interaktívnych systémov beží len dopredu známa a odladená množina procesov – dá sa použiť aj nepreemptívny OS, lebo procesy sú naprogramované tak, aby sa dohodli

Ideálny plánovač (všeobecne)

1. férový – porovnateľné procesy by mali dostať porovnateľný čas
2. vynucujúci ciele a pravidlá – ak sú dané nejaké pravidlá (napr. jeden proces môže bežať max 200ms) tak plánovač ich musí byť schopný vynútiť
3. vyvážený – napr. zaťažovať procesor aj I/O zariadenia rovnako (a nie procesor maká, I/O čaká; a potom I/O maká a procesor čaká...) - pri systémoch kde plánovač nepozná podrobne procesy sa to dá monitorovať a odhadnúť
4. efektívny – plánovač má bežať čo najkratšie

Pre dávkové spracovanie - ciele

1. maximalizovať priepustnosť
– počet vykonaných úloh za jednotku času
2. minimalizovať čas obrátky
– priemerný čas od zadania požiadavky po dostanie výsledky
3. maximalizovať využitie procesora

Pre dávkové spracovanie – algoritmy

1. first-come first-served
 - nepreemptívne – kto prv príde ten prv melie – procesy sú zoradené podľa času, kedy prišli do „pripravený“
 - proces beží až kým neskončí alebo neodíde do stavu čakajúci
 - znevýhodňuje vstupno-výstupné procesy oproti výpočtovým
 - tento algoritmus je vhodný pre systémy, kde sú procesy približne podobné (s podobnou dĺžkou behu)
 - pokiaľ mám mix (procesy s rôznou dĺžkou behu) môžem nechať kratšie bežiacie procesy sa predbiehať, napr. majme 2 procesy: 1. počíta často a krátko, 2. málo často ale dlho – v prípade že 1. sa bude predbiehať, potom veľmi zvýhodníme krátku úlohu a málo znevýhodníme tú dlhú => pomerne vhodný algoritmus
2. shortest job first
 - minimalizuje čas obrátky
 - je potrebné vedieť koľko ktorý proces bude bežať

<p>Shortest job first:</p> <table> <tr><td>úloha</td><td>čas behu</td></tr> <tr><td>D</td><td>4</td></tr> <tr><td>C</td><td>4</td></tr> <tr><td>B</td><td>4</td></tr> <tr><td>A</td><td>8</td></tr> </table> <p>časová os</p> <p>-----></p> <table> <tr><td>D</td><td>C</td><td>B</td><td>A</td><td> </td></tr> <tr><td>0</td><td>4</td><td>8</td><td>12</td><td>20</td></tr> </table> <p>ak všetky procesy prišli naraz (v čase 0 tak:)</p> <table> <tr><td>úloha</td><td>dĺžka obrátky (kedy dostaneme výsledok):</td></tr> <tr><td>D</td><td>4</td></tr> <tr><td>C</td><td>8</td></tr> <tr><td>B</td><td>12</td></tr> <tr><td>A</td><td>20</td></tr> </table> <p>priem. čas obrátky: 11</p>	úloha	čas behu	D	4	C	4	B	4	A	8	D	C	B	A		0	4	8	12	20	úloha	dĺžka obrátky (kedy dostaneme výsledok):	D	4	C	8	B	12	A	20	<p>Nie shortest job first:</p> <table> <tr><td>úloha</td><td>čas behu</td></tr> <tr><td>D</td><td>4</td></tr> <tr><td>C</td><td>4</td></tr> <tr><td>B</td><td>4</td></tr> <tr><td>A</td><td>8</td></tr> </table> <p>časová os</p> <p>-----></p> <table> <tr><td>A</td><td>B</td><td>C</td><td>D</td><td> </td></tr> <tr><td>0</td><td>8</td><td>12</td><td>16</td><td>20</td></tr> </table> <p>ak všetky procesy prišli naraz (v čase 0 tak:)</p> <table> <tr><td>úloha</td><td>dĺžka obrátky</td></tr> <tr><td>A</td><td>8</td></tr> <tr><td>B</td><td>12</td></tr> <tr><td>C</td><td>16</td></tr> <tr><td>D</td><td>20</td></tr> </table> <p>priem. čas obrátky: 14</p>	úloha	čas behu	D	4	C	4	B	4	A	8	A	B	C	D		0	8	12	16	20	úloha	dĺžka obrátky	A	8	B	12	C	16	D	20
úloha	čas behu																																																												
D	4																																																												
C	4																																																												
B	4																																																												
A	8																																																												
D	C	B	A																																																										
0	4	8	12	20																																																									
úloha	dĺžka obrátky (kedy dostaneme výsledok):																																																												
D	4																																																												
C	8																																																												
B	12																																																												
A	20																																																												
úloha	čas behu																																																												
D	4																																																												
C	4																																																												
B	4																																																												
A	8																																																												
A	B	C	D																																																										
0	8	12	16	20																																																									
úloha	dĺžka obrátky																																																												
A	8																																																												
B	12																																																												
C	16																																																												
D	20																																																												

- táto metóda minimalizuje čas obrátky ak všetky procesy prídu v čase 0 - ak to budem robiť dynamicky („krátke procesy sa predbiehajú“) tak tým zoptimalizujem priepustnosť na úkor dlhých procesov
- 3. shortest remaining time next
 - preemtívne
 - predbiehať sa budú tí, ktorým do dokončenia chýba najmenej
- 4. trojúrovňové plánovanie
 - plánovanie prijatia – rozhodnúť, ktoré úlohy (nie procesy) prijmem do spracovania, napr. idem najprv počítať finančné výkazy alebo tlačiť faktúry?
 - plánovanie procesora
 - plánovanie výmen pamäť-disk – správa virtuálnej pamäti je časovo náročná – možno sa oplatí radšej spraviť nejaký proces, ktorý je v pamäti, aj keď „lepší“ kandidát je stále na disku, lebo kým by sa presunul do pamäte tak prejde veľa času

Interaktívne systémy – ciele

1. minimalizovať čas odozvy
 - predbiehajú sa procesy, ktoré prišli zo stavu čakajúci (to sú procesy, čo reagujú na užívateľa)
2. proporcionalita
 - to, čo užívateľ čaká že je náročné, môže trvať dlho; to, čo čaká že je nenáročné musí trvať krátko => psychológia

Interaktívne systémy – algoritmy

1. round robin (socialistický – každému rovnako)
 - preemtívne
 - v stave pripravený spravíme frontu a ideme porade – každý dostane nejaký čas (časové kvantum), a buď proces skončí, alebo mu procesor odoberieme a vrátime na koniec fronty
 - problémom je určenie správneho časového kvanta – príliš vysoké: znížim čas odozvy, príliš nízke: nízka efektívnosť (veľa času počítač len vymieňa procesy). Obvykle sa používa okolo 50ms.
 - obvykle sa round robin nepoužíva v čistej forme, ale sa modifikuje
2. prioritné plánovanie
 - vyberá sa proces s vyššou prioritou (priorita sa dá dopredu určiť len ak vieme, čo vlastne procesy robia) – znižuje férovosť (nízko prioritné procesy sa teoreticky vôbec nemusia dostať k slovu)
 - zjemnenie: každý proces má statickú prioritu (od vytvorenia) a dynamickú prioritu (na základe nejakých pravidelne kontrolovaných parametrov) – zlepšenie férovosti
3. viac frontov
 - mix round robina a prioritného plánovania
 - mám napr. 4 frony (1. vysoká priorita, 2. nižšia priorita, 3. ešte nižšia, 4. najnižšia priorita)... ale časové kvantum ide naopak (1. malé kvantum, 2. väčšie kvantum, ..., 4. najväčšie kvantum) – čiže procesy s vysokou prioritou počítajú často ale krátko, s najnižšou prioritou zriedkavo ale dlho
 - ak pridáme rozumné pravidlá pre zmenu priorit a zatriedovanie do front dostávame pomerne slušné plánovanie
 - obvyklé rozdelenie:
 1. terminal – interaktívna operácia

2. I/O
3. short quantum
4. long quantum
4. shortest process next
 - interaktívny variant shortest job first
 - najkratší proces: proces, ktorý v minulosti strávil najmenej času medzi dvoma čakaniami
 - s použitím zložitejších algoritmov sa dá ošetriť aj to, že procesy sa časom menia (začne viac počítať, začne menej počítať, ...)
5. garantované plánovanie
 - stanovím si nejakú podmienka a snažím sa ju dodržať... napr. ak mám N procesov, každý musí dostať 1/N času
6. lotéria
 - každý proces má los (alebo viac losov) a plánovač ich losuje
 - viac losov = ako keby vyššia priorita – ale narozdiel od prioritného plánovania viem štatisticky spočítať, kto bude mať koľko času (ak mám 2 procesy, jeden má 5 losov, druhý 10, viem že dostanú čas v pomere 1:2)
 - spolupracujúce procesy si môže losy odovzdávať
7. férový podiel
 - z pohľadu používateľov: ak už A má 5 procesov a B 4 procesy a každý proces dostane rovnako, užívateľ A dostane viac strojového času - „férovejšie“ je ak každý užívateľ má rovnaký strojový čas a jeho procesy sa o neho bijú

Real-Time systémy – ciele

1. stihnúť termíny
 - ak sa niečo začne prehrievať a nevypnem to do 2 sekúnd, továreň vybuchne
2. predvídateľnosť
 - pravidelnosť správania (ak to nestíha teraz, tak to nebude stíhať ani zajtra a treba to preprogramovať)

Komunikácia medzi procesmi

= zdieľanie pamäte medzi dvoma threadmi jedného procesu (napr. globálne premenné)

Nasledovný príklad je vysvetlený na poli spooler, ktoré zdieľajú dva procesy. Spooler je pole, kam sa ukladajú názvy súborov, ktoré chceme vytlačiť na tlačiarňi. Keďže o samotnú tlač sa nestaráme (o tú sa stará print daemon), stačí nám ak do poľa zapíšeme názov súboru a smerník pin (najbližší voľný prvok poľa) posunieme na ďalší prvok. (Smerník pout ukazuje na prvok, kde je názov súboru, ktorý sa bude tlačiť ako ďalší – o to sa nestaráme, o to sa stará print daemon)

```
var Spooler    : array[1..10000] of string;
    pin, pout  : integer;
...
procedure ZaradDoFrontu(subor: string);
begin
    Spooler[pin] := subor;           //riadok 1
    pin := pin+1;                   //riadok 2
end;
```

Nech naše pole spooler vyzerá takto:

SPOOLER:

```

...           pout = 4, pin = 7
4  abc
5  prog.c
6  prog.n
7
...

```

Mám dva procesy (A a B), oba používajú rovnaký spooler. Proces A zavolá ZaradDoFrontu('xy.dat'), proces B zavolá ZaradDoFrontu('ab.pas'). A prúser je, ak procesu A plánovač zobere procesor pred vykonaním riadku 2. Vtedy sa stane toto:

1. proces A – do spooler[7] priradí 'xy.dat'
2. plánovač zobere procesoru A procesor a pridelí ho procesu B
3. proces B – do spooler[7]!!! priradí 'ab.pas'
4. proces B – zvýši pin o 1 (pin = 8)
5. proces B skončil, plánovač opäť spustí proces A. Ten pokračuje tam, kde bol predtým prerušený, t.j. na riadku 2
6. proces A – zvýši pin o 1 (pin = 9)

Výsledok:	Čo sme chceli dostať:
SPOOLER: <pre> ... pout = 4, pin = 9 4 abc 5 prog.c 6 prog.n 7 ab.pas 8 9 ... </pre>	SPOOLER: <pre> ... pout = 4, pin = 9 4 abc 5 prog.c 6 prog.n 7 xy.dat 8 ab.pas 9 ... </pre>

- tento problém sa nazýva **Časová závislosť** – nastáva, ak výsledok práce viacerých spolupracujúcich procesov závisí od poradia, v ktorom boli spustené plánovačom procesov
- v princípe časová závislosť hneď nemusí viesť k chybe – napr. rezervačný systém... Cestujúci A a B si „naraz“ zarezervujú letenku. Ak sa požiadavka A spracuje o chvíľku skorej ako B, výsledok bude, že cestujúci A dostane miesto 17 a cestujúci B miesto 18. Ak sa to stane naopak, cestujúci A dostane miesto 18 a cestujúci B miesto 17. => Je to časová závislosť, ale nevedie k problémom
- v príklade vyššie ale časová závislosť viedla k problému – jeden súbor sa nám vôbec nevytlačí, a ešte k tomu máme v poli prázdne miesto, ktoré tam nemá čo robiť
- najhoršie na časovej závislosti je, že pravdepodobnosť, že problém nastane je veľmi malá: môže to 5 rokov fungovať a nikto si nič nevšimne, ale zrazu sa to všetko začne srať – je takmer nemožné ich odhaliť testovaním
- **kritická sekcia** – úsek programu, ktorý pracuje so zdieľanými dátami a počas vykonávania tohto úseku môže byť údaje nekonzistentné*, v našej procedúre je to:


```

Spooler[pin] := subor; //riadok 1
pin := pin+1; //riadok 2

```
- časová závislosť nastane, ak dva procesy budú naraz v kritických sekciách
- * - v našom prípade... smerník pin po vykonaní riadku 1 neukazuje na prvé voľné miesto => nekonzistentnosť

Riešenie časovej závislosti

- **Vzájomné vylúčenie** – riešenie časovej závislosti - žiadne 2 procesy nie sú naraz vo svojich kritických sekciách
- ukázalo sa, že vzájomné vylúčenie nie je úplné riešenie časovej závislosti, pri úplnom riešení treba dodržať aj nasledovné pravidlá:
 1. nepredpokladať nič o vzájomnej rýchlosti a/alebo počte procesorov
 2. žiadny proces mimo svojej kritickej sekcie nesmie brániť ostatným procesom vstúpiť do kritickej sekcie
 3. žiadny proces nesmie donekonečna čakať na vstup do kritickej sekcie
- vo všeobecnosti: ak chcú dva procesy vstúpiť do kritickej sekcie, jednému to treba povoliť a druhý musí počkať
- kritické sekcie musia skončiť a neostať zacyklenené!
- rozlišujú sa dva spôsoby:
 1. busy waiting
 - logicky program čaká, ale technicky beží... napr.:
repeat until not PovolenyVstupDoKritickejSekcie
 2. čakanie v zmysle plánovača procesov
 - proces naozaj čaká
 - vyžaduje podporu v operačnom systéme

Busy waiting

1. zakázanie prerušení
 - operačnému systému som zobral možnosť ma prerušiť, ale zároveň som odstaviť aj prerušenia z I/O zariadení -> počítač prestane úplne reagovať na užívateľa
 - problémy: OS mi to pravdepodobne nedovolí (napr. Win XP to už nedovolí, ale Win 98 to zožral)
 - táto metóda nie je reálne použiteľná v normálnych procesoch, ale používa sa na úrovni OS a driverov
2. zamykacie premenné
 - okrem spoolera, pin, pout budeme mať aj zamknute : boolean

```

var Spooler    : array[1..10000] of string;
    pin, pout  : integer;
    zamknute   : boolean;

...
procedure ZaradDoFrontu(subor: string);
begin
    repeat until not zamknute; // 1
    zamknute := true;         // 2
    Spooler[pin] := subor;
    pin := pin+1;
    zamknute := false;
end;

```

 - v tomto prístupe sa opäť môžu dostať 2 procesy do kritickej sekcie... Problém sa len presunul medzi riadky 1 a 2. Ak je proces A prerušený medzi riadkami 1 a 2, obidva procesy vojdú do kritickej sekcie => **toto nie je riešenie**
3. striktné striedanie

```

var Spooler    : array[1..10000] of string;
    pin, pout  : integer;
    turn       : integer = 1;

```

```

...
proces A:
procedure ZaradDoFrontu(subor: string);
begin
  repeat until turn = 2;
  Spooler[pin] := subor;
  pin := pin+1;
  turn := 2;
end;

proces B:
procedure ZaradDoFrontu(subor: string);
begin
  repeat until turn = 1;
  Spooler[pin] := subor;
  pin := pin+1;
  turn := 1;
end;

```

- proces A = 1, proces B = 2... a navzajom si nastavujú, kto ďalší môže ísť do kritickej sekcie
- problém: proces A zavolá ZaradDoFrontu, to prebehne ok, turn sa nastaví na 2; A teraz chce znova tlačíť proces A, ale podmienka repeat until turn = 2 ho nepustí
- toto riešenie porušuje podmienku: „žiadny proces mimo svojej kritickej sekcie nesmie brániť ostatným procesom vstúpiť do kritickej sekcie“
- problém: je to fixné riešenie pre 2 procesy, pre viac procesov by sme to museli meniť
- ak som si ale istý, že procesy A a B budú vždy chodiť striedavo, je to dobrý spôsob

4. Petersonovo riešenie

(pozri v slajde)

- málo použiteľné, lebo musíme vedieť, koľko je procesov s kritickou sekciou
- pre počet procesov > 2 by bolo zložité testovanie, ale je to riešenie použiteľné

5. Neprearušiteľné „testuj a nastav“

- vyžaduje podporu v strojovom kóde, na IBM 360 – TSL (skopírovala pamäťové miesto do registra a potom pamäťové miesto vynulovala)
 - na Inteli sa na to dá zneužiť *SHR odomknute, 1*
- ```

var Spooler : array[1..10000] of string;
 pin, pout : integer;
 odomknute : integer; // 0-zamknuté, 1-odomknuté

```

```

...
procedure ZaradDoFrontu(subor: string);
begin
 asm
 @1: shr odomknute, 1
 jnc @1
 end;
 Spooler[pin] := subor;
 pin := pin+1;
 odomknute := 1;
end;

```

- čo sa deje:
  - proces A spraví *shr odomknute 1*, čím zmení odomknute na 0, ale zároveň si predchádzajúci stav skopíruje do CF. CF je v registroch, preto ak je proces A teraz prerušený, hodnota CF sa uloží do virtuálnej pamäte – čiže sa už nestratí.
  - Ak sa teraz pustí proces B, ten tiež spraví *shr odomknute 1* ale v odomknuté už je 0 – preto nevstúpi do kritickej sekcie

## Procesy a thready vo Windows 32-bit

- proces = obal, vlastní virtuálnu pamäť, má nejaký stav, ale nebeží
- thread = vlastní zásobník a to je to, čo beží
- používa sa prioritné plánovanie, plánujú sa thready (nie procesy)
- **fiber** – thread, ktorý sa neplánuje – musia sa plánovať ručne (vrámci aplikácie); nie v Win95/98/Me

## Plánovanie

- každý **proces** patrí do jednej z **prioritných tried** (1):
  - idle
  - normal
  - high
  - realtime
- každý **thread** má **úroveň priority** (2):
  - idle
  - lowest
  - below normal
  - normal
  - above normal
  - highest
  - time critical
- defaultne normal class, normal priority
- okrem toho sa rozlišuje aj **F/B stav** (3)
  - **foreground** – proces, ktorý má aktívne okno (focus)
  - **background** – proces, ktorý nemá aktívne okno (focus)
- na základe údajov (1), (2), (3) sa určí **base priority threadu** (statická priorita). Je to číslo 1 až 31.
- trieda a úroveň sa dá meniť programovo (thread sebe alebo inému thread), F/B stav záleží len od užívateľa
- v stave Pripravený je 31 frontov – podľa dynamickej priority. Čiže vrámci jednej priority funguje round-robin, medzi triedami platí striktné prioritné plánovanie
- priorita (dynamická) sa dá dočasne zvyšovať:
  - a. priority boost** – snaha dosiahnuť lepší čas odozvy
    - dočasné zvýšenie priority threadu nad jeho base priority (len pre procesy s base prioritou 1..15)
    - nastáva v 3 prípadoch:
      - keď sa proces stane Foreground - priorita sa zvýši, aby bola väčšia alebo rovná ako prioritná trieda iných procesov
      - keď okno dostane vstup z myši, klávesnice, časovača – priorita sa zvýši threadu, ktorý okno vytvoril
      - keď sa thread presunie do stavu pripravený zo stavu čakajúci
    - po každom prechode zo stavu bežiaci do pripravený sa zníži priorita, až opäť nedosiahne base priority
  - b. priority inversion**
    - v prípade prioritného plánovania a komunikácie medzi threadmi (=kritické sekcie) môže nastať situácia, že proces s nižšou prioritou blokuje proces s vyššou prioritou...
    - vo Win NT/2K/XP/...: plánovač náhodne zvyšuje prioritu procesom v stave pripravený

- vo Win 95/98/Me: snaha detekovať stav, že sa procesy blbo blokujú

## Synchronizácia

- synchronizačné objekty
- sú 4 typov:
  - event
  - mutex – dvojestavový semafor
  - semaphore – všeobecný semafor
  - timer
- okrem toho sa dá čakať aj na iné objekty (napr. čakaj kým skončí thread XY, čakaj kým skončí process XY)

### mutex

CreateMutex(security, true/false, názov)

- táto fcia vráti handle, tento handle ale platí len v rámci jedného procesu. Meno funguje, ak chcú tento mutex používať viaceré procesy

ReleaseMutex

- spraví up

WaitForSingleObject(object, ms)

- spraví down

WAAAAAAAA????

### semaphore

CreateMutext(security, počiaočný stav, maximálny stav, názov)

ReleaseSemaphore

- pripočíta 1

WaitForSingleObject(semafór, ...)

- odpočíta 1

WAAAAAAAA????

## Správa pamäti

- úloha: aby programátor mal dojem, že pamäť je veľká a vcelku
- pridelovať pamäť, chrániť a zdieľať

## Fixed Partition

- IBM OS/MFT
- na začiatku dňa sa nastavila pamäť na oddiely (fixne) a úlohy sa radili do front pre konkrétny oddiel pamäte
- upgrade: s použitím relokácie sa dá spraviť len jedna fronta a aj menšie úlohy môžu počítať vo veľkých oddieloch

## Relokácia

= schopnosť kódu bežať nie vždy na tých istých adresách - čiže nepoužívať absolútne adresy premenných a skokov

- riešenie relokácie
  1. vždy znova skompilovať
  2. použiť relatívne adresovanie (od nejakého počiatku, ktorý je niekde v registri)
  3. uložiť informáciu pre relokovanie do súboru spolu s kódom (na začiatku súboru je vyslovene povedané, ktoré bajty označujú adresy, a OS prebehne relokačnú tabuľku a upraví adresy) – takto fungovali EXE v MS-DOSe

## Ochrana

= zabrániť programu, aby zapisoval do pamäte inému programu, resp. operačnému systému

- možnosti
  1. vykašľať sa na to (MS-DOS)
  2. zámok/kľúč
  3. báza/limit – mám dva registre.. začiatok a veľkosť... a vždy sa kontrolovalo či program nechce zasahovať na adresu väčšiu ako začiatok+veľkosť

## Swapping

- OS/MVT
- nebudem mať fixne rozdelené oblasti, ale na začiatku budem mať jednu oblasť. Príde jedna úloha, useknem mu toľko pamäti koľko potrebuje. Príde ďalšia, jej useknem ďalší kus. Atd'.. Ak príde ďalšia úloha, ktorá sa už nezmestí do pamäte, tak nejaký proces hodím na disk a na uvoľnené miesto hodím ten nový.
- opäť treba nejak riešiť relokáciu – lebo sa môže stať, že proces po „návrate“ z disku príde na iné miesto, ako kde bol pôvodne
- okrem relokácie a ochrany nastáva problém fragmentácie – vznikajú diery – kompaktácia

## Implementácia jednoduchej správy pamäti

- 2 procedúry: GetMem(p:TPointer, n:integer) a FreeMem(p:TPointer, n:integer)
- ide o to pamätať si, ktoré časti pamäte sú voľné a ktoré obsadené
  - **bitová mapa**
    - pre každý sektor si pamätám 0 – voľné, 1 – obsadené
    - bitová mapa tiež musí byť niekde v pamäti
    - ak príde volanie napr. GetMem(p, 3) tak hľadám prvé 3 nuly za sebou a vrátim na ne smerník
    - ak by 1 sektor = 1 byte, tak na 1 byte použijem 1 bit = 1/9 pamäte by som použil len na bitovú mapu; preto jeden sektor je v praxi okolo 16 bajtov
    - trochu problémom je lineárne prehľadávanie bitovej mapy – po jednom bite prechádzam celú bitmapu; takisto ak idem obsadiť veľký úsek, potrebujem na veľa miest ukladať jednotky => je to pomalé
  - **spájaný zoznam**
    - kde každý prvok je:
      - TPrvok = record

```

typ: P,H; // P-process (obsadené), H-hole (diera)
adr: integer; // adresa odkiaľ
vel: integer; // koľko
end;

```

- treba pozor pri uvoľňovaní – treba pozrieť či pred a/alebo za uvoľňovaným nie je niečo voľné => potrebujem pozerať dopredu aj dozadu – čiže radšej obojsmerne spájaný zoznam
- potrebujem vedieť rozpájať a spájať prvky
- neviem dopredu koľko miesta budem potrebovať pre spájaný zoznam. A môže sa stať, že nakoniec samotný spájaný zoznam bude (v najhoršom prípade) aj väčší ako samotná pamäť.
- mám požiadavku na pridelenie nejakej pamäte. Existuje viacero algoritmov:
  - **first fit**
    - prvá diera, ktorá vyhovuje podmienke sa obsadí
    - v praxi sa používa viac ako best fit
    - nevýhoda: má tendenciu nechávať malé kúsky dier na začiatku a veľké až na konci – vylepšenie: Circular First Fit – nový kúsok hľadám až odtiaľ, kde som naposledy nejaký pridelil
  - **best fit**
    - pridelím tú dieru, kde ostane najmenší nevyužitý zvyšok
    - nevýhoda: ostane mi veľa malých kúskov, ktoré nikto nikdy nevyužije => zvýši fragmentáciu pamäte
  - **worst fit**
    - kde ostane najväčší nevyužitý zvyšok
  - **quick fit**
    - spájané zoznamy voľných miest podľa očakávaných veľkostí (keď viem, že budem potrebovať len veľkosti 512, 1024, 4098 a žiadne iné)



This work is licensed under a [Creative Commons Attribution-NonCommercial-Share Alike 3.0 License](https://creativecommons.org/licenses/by-nc-sa/3.0/).

Original source: <http://matfyz.adammuller.sk/opsys>