STU FIIT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES

Dátové štruktúry a algoritmy

Aula magna
streda 10:00

letný semester
2020/2021

Správa pamäte pri vykonávaní programu

24. 2. 2021

letný semester
2020/2021

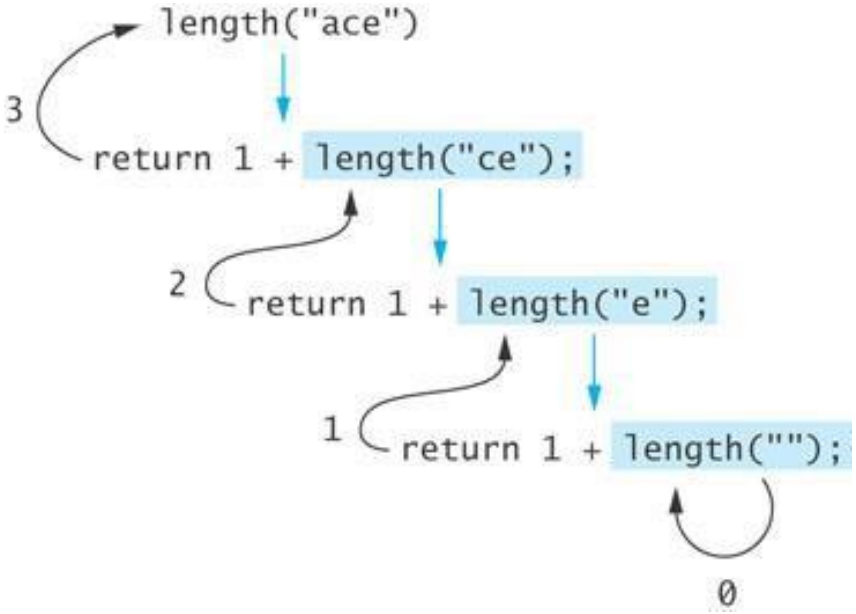
Príklad

- Rekurzívny algoritmus na určenie dĺžky reťazca
  - Ak je reťazec prázdny, výsledok je 0, inak výsledok je (1 + dĺžka reťazca bez prvého znaku)

  - Zdrojový kód:

```
int length(char *s)
{
  if (!s || *s == 0)
    return 0;
  return 1 + length(s+1);
}
```

Krokovanie **length**(“ace”):

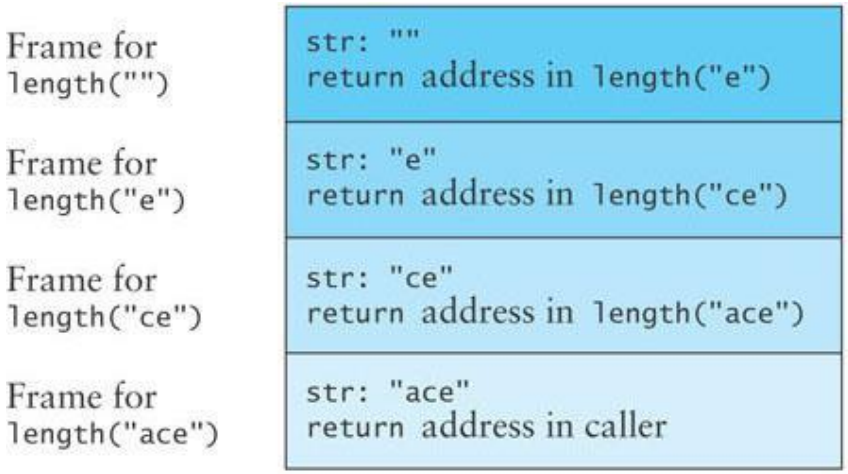


Aktivačný rámec

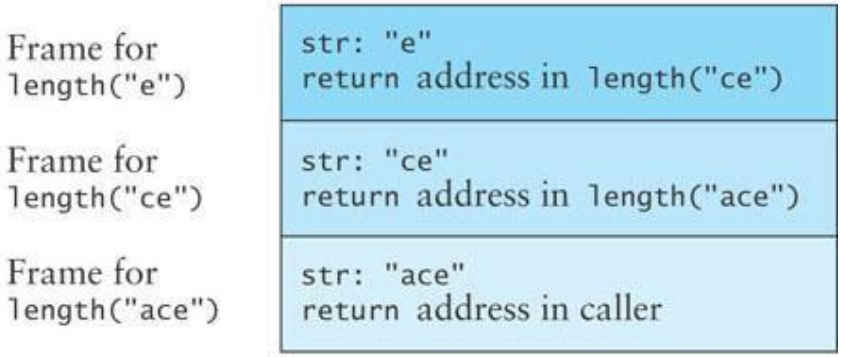
- Stavová informácia pre volanie funkcií
- Pre vykonanie volania funkcie je potrebné uchovať nasledovné informácie:
  - argumenty funkcie (hodnoty parametrov)
  - adresa, kam sa má vrátiť vykonávanie programu po ukončení volania funkcie (návratová adresa pre return)
  - lokálne premenné (hodnoty)
- Pre každé volanie funkcie sa vytvorí aktivačný rámec (stack frame) a vloží sa do zásobníka volaní (call stack)
- (Úmyselné) pretečenie zásobníka volaní predstavuje bezpečnostné riziko: **stack buffer overflow**

Zásobník volaní – ukážka

- Volanie length(“ace”):



obsah zásobníka po zavolaní length(“”), vrch je hore



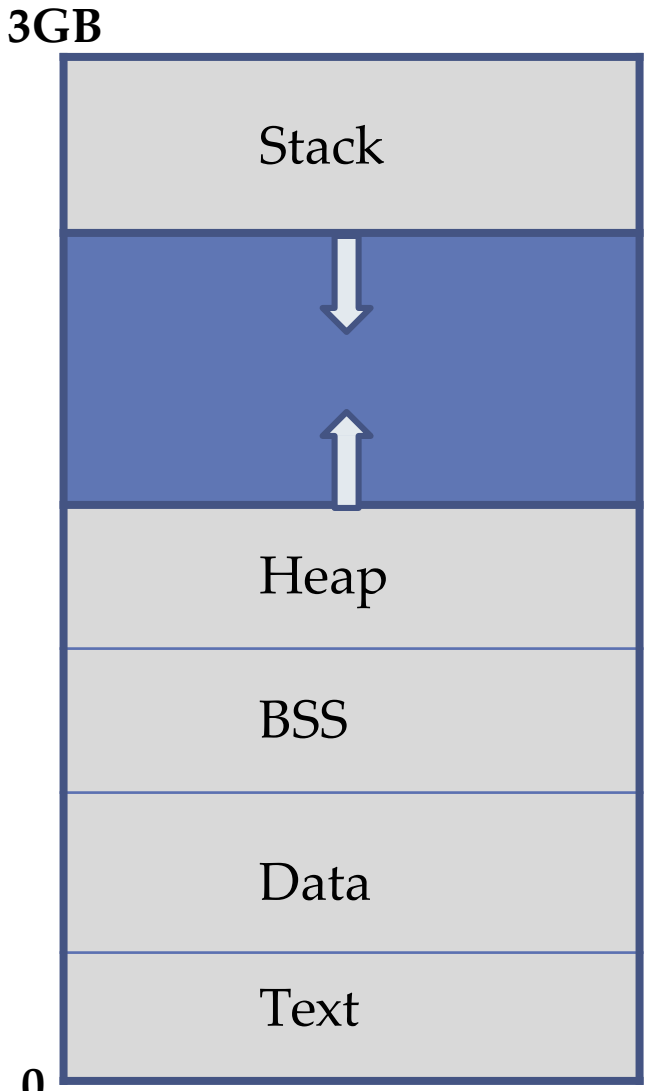
obsah zásobníka po návrate z vykonania length(“”)

Program vs. proces

- Program po preložení (**compile**), spojení s externými podprogramami (**link**) a načítaní do pamäti počítača (**load**) sa vykonáva (**execute**) – proces
- Riadiaci blok procesu (process control block)
  - Stav procesu – new, ready, running, waiting, ...
  - Registre – %eip, %eax, ...
  - Pamäť – všetko čo proces môže adresovať: kód, dáta, zásobník (stack), heap (halda)
  - I/O – stav otvorenia-čítania súborov
  - ...
- Program je statický kód a statické dáta
- Proces je dynamická inštancia kódu, dát a ďalšieho
- Bežiacemu procesu sa musí prideliť v počítači pamäť, aby mal kam zapisovať údaje (medzivýsledky atď.)

Adresný priestor procesu (Process address space)

- **Text**: obsahuje program v strojovom jazyku, ktorý sa vykonáva, reťazce, konštanty, a ďalšie údaje na čítanie
- **Data**: inicializované globálne a statické premenné
- **BSS**: (Block Started by Symbol) neinicializované globálne a statické premenné
- **Stack (zásobník)**: lokálne premenné bežiaceho procesu
- **Heap (halda voľnej pamäti)**:
  - dynamická pamäť procesu (môže sa zväčšovať aj zmenšovať)
  - toto je pamäť, ktorú prideľuje malloc()



Adresný priestor procesu - príklad

Data: globálna premenná

Text (read-only data)

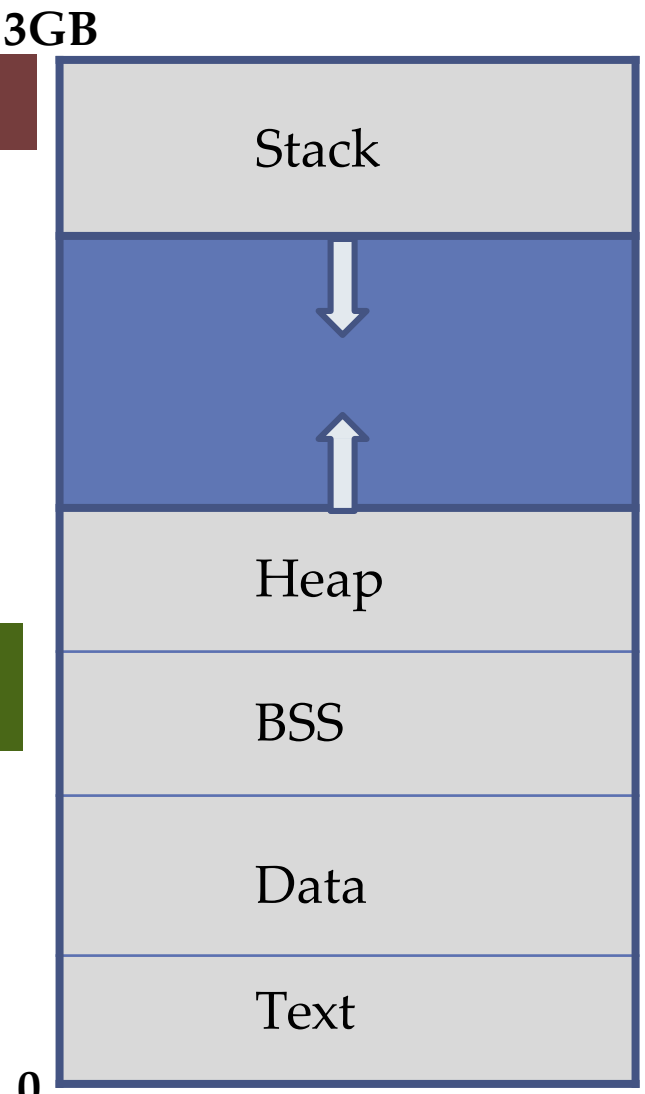
```
char str = “hello”;
int iSize;
char *f(void)
{
    char *p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```

BSS

Stack (lokálna premenná)

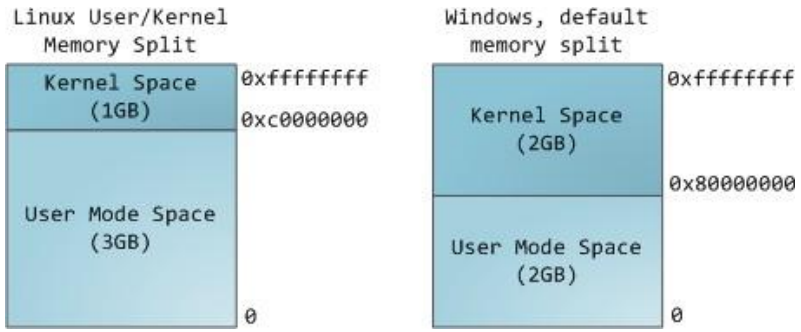
Heap (dynamická pamäť)

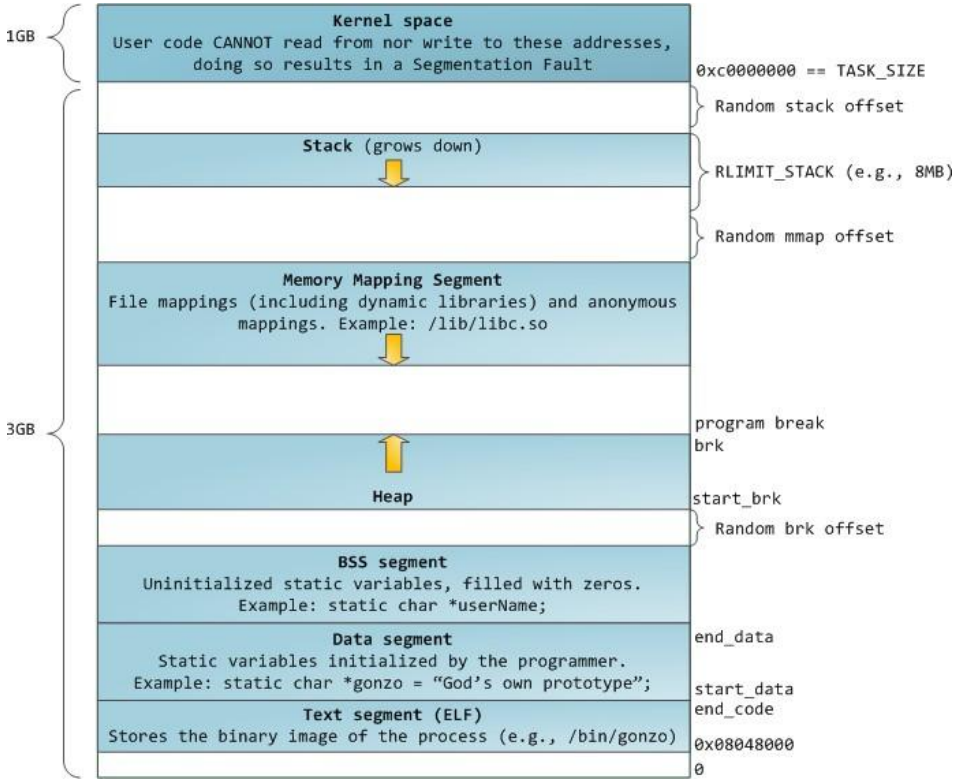
Preložený program jeText



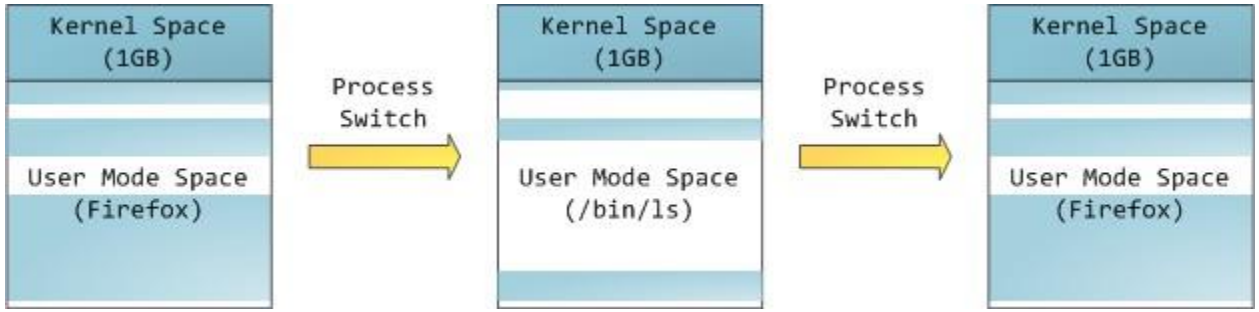
Detailnejší pohľad na pamäť v OS (32bit)

- Kernel vs. User mode





- Zmena vykonávaného procesu (context switch)



Typy prideľovania pamäte

- Statická veľkosť, statické prideľovanie
  - globálne premenné
  - spojovač (linker) pridelí definitívne virtuálne adresy
  - vykonateľný strojový program odkazuje na tieto pridelené adresy
- Statická veľkosť, dynamické prideľovanie
  - lokálne premenné
  - prekladač predpíše prideľovanie v zásobníku
  - posunutia voči ukazovateľu na vrch zásobníka (čo sú vlastne adresy premenných) sú priamo vo vykonateľnom strojovom programe
- Dynamická veľkosť, dynamické prideľovanie
  - ovláda programátor
  - prideľuje sa v dynamickej voľnej pamäti (heap – halda)

Prideľovanie dynamickej pamäti

- Dynamická pamäť sa prideľuje v čase výpočtu, nie v čase prekladu
- Veľkosť pridelenej pamäti nemusí byť známa až do okamihu pridelenia; napr. závisí od vstupného údaja zadaného používateľom
- Pretože veľkosť potrebnej pamäti môže byť rôzna, vyžiadanie jej pridelenia od procedúry **malloc** (apod) zahŕňa parameter veľkosť (**size**)

Funkcie pre prideľovanie pamäti v jazyku C

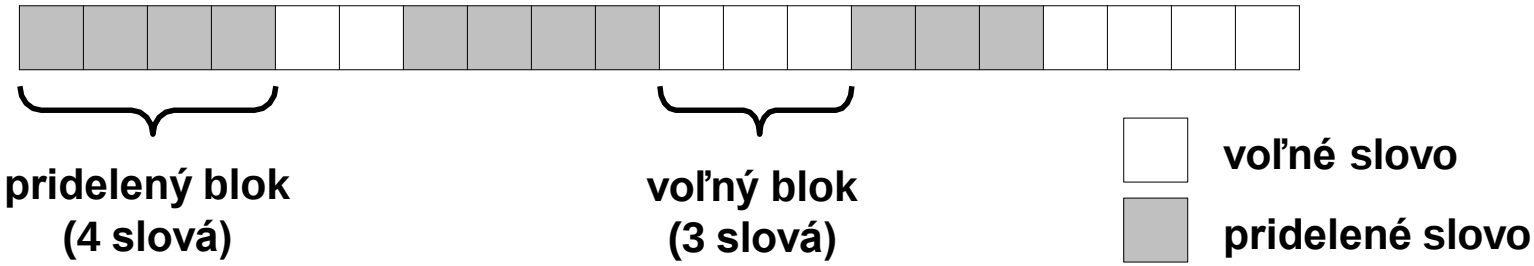
- Volanie **ptr ← malloc(size)** spôsobí, že sa pridelí pamäť veľkosťou čo najbližšia požadovanej
  - Pridelená pamäť nie je inicializovaná
- Volanie **free(ptr)** spôsobí, že pridelená pamäť (ptr) sa uvoľní – vráti späť do voľnej pamäti
- Veľkosť pridelenej pamäti možno zmeniť pomocou **newptr ← realloc(oldptr, size)**
- Volanie **ptr ← calloc(n, size)** spôsobí, že sa pridelí pamäť pre n prvkové pole s prvkami veľkosti size
  - Pridelená pamäť je inicializovaná na 0

Čo ak použitú pamäť nevrátime?

- Ak program nevráti (neuvoľní) pridelenú pamäť po tom, čo ju už netreba pre ďalší výpočet
  - stratí sa jediný odkaz na ňu
  - nebude sa dať jej obsah sprístupniť
  - je to trhlina v pamäti (**memory leak**)

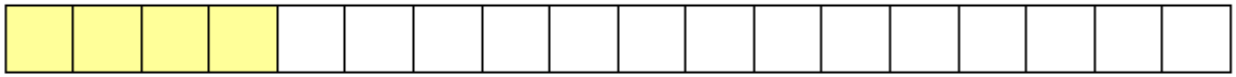
Ukážka prideľovania pamäte

- Pamäť sa adresuje po slovách
  - 4 byte (pre 32 bit architektúru)
- Na obrázkoch zobrazíme “štvorčeky” – slová
- Každé slovo môže obsahovať celé číslo (int) alebo smerník / ukazovateľ



Ukážka prideľovania pamäte (2)

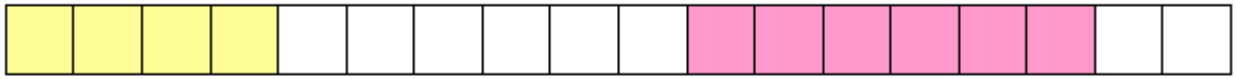
```
p1 = malloc(4*sizeof(int))
```



```
p2 = malloc(5*sizeof(int))
```

```
p3 = malloc(6*sizeof(int))
```

```
free(p2)
```



```
p4 = malloc(2*sizeof(int))
```

Ohraničenia prideľovania

- Programy, ktoré sa vykonávajú:
  - môže mať ľubovoľnú postupnosť požiadaviek malloc a free
  - požiadavky na free sa musia vzťahovať na pridelenú pamäť

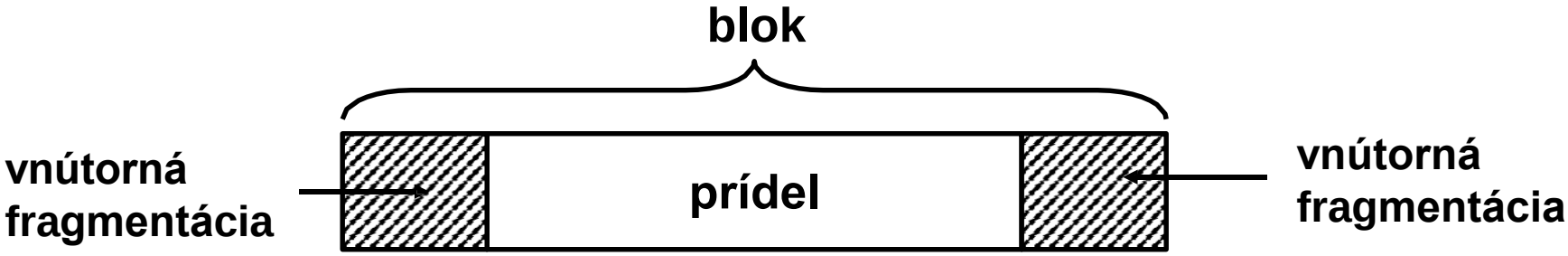
- Správca dynamickej pamäti
  - neovláda počet ani veľkosť prideľovaných blokov pamäti
  - musí vyhovovať všetkým požiadavkám okamžite
    (nemôže ich preusporiadať alebo odložiť na neskôr)
  - musí prideľovať pamäť z voľnej pamäti
  - musí zarovnať veľkosť bloku tak, aby splnila všetky požiadavky na zarovnávanie (zvyčajne na 8 byte-ov)
  - môže manipulovať a meniť iba voľnú pamäť
  - nemôže presúvať už pridelený blok pamäti
    (nebudeme predpokladať možnosť skompaktňovania)

Ciele dobrej implementácie prideľovania pamäte

- Dobrá časová efektívnosť **malloc** aj **free**
  - ideálne, v konštantnom čase (nie vždy možné)
  - určite by nemali potrebovať lineárny čas v závislosti od počtu blokov
- Dobré využívanie pamäti
  - pridelené bloky pamäti by mali využívať pamäť čo najlepšie
  - minimalizovať "fragmentáciu"
- Vlastnosti dobrej lokálnosti
  - štruktúry pridelené blízko v čase by mali byť blízko seba v pamäti
  - "podobné" objekty by mali byť umiestnené blízko seba
- Robustnosť
  - vie overiť, že **`free(p1)`** sa týka platného prideleného objektu **`p1`**
  - vie overiť, či ukazovatele odkazujú do prideleného úseku pamäti

Vnútorná fragmentácia

- Pamäť nie je efektívne využitá celá – fragmentácia
  - vnútorná a vonkajšia
- **Vnútorná fragmentácia**
  - pre daný blok je vnútorná fragmentácia rozdiel medzi veľkosťou bloku a veľkosťou prídelu

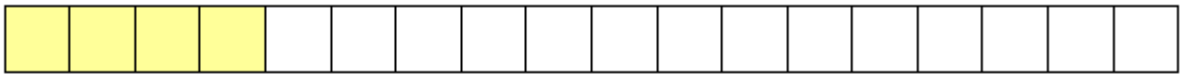


  - spôsobuje ju réžia (overhead) udržiavania dynamickej pamäti, zarovnávanie, prípadne rozhodnutia správy pamäti (napr. nerozbiť blok)
  - je určená tým, aké požiadavky boli doteraz, dá sa ľahko vyhodnotiť

Vonkajšia fragmentácia

- nastáva, keď je síce dosť voľnej pamäti spolu (agregátne), ale žiadny voľný blok nie je dostatočne veľký

```
p1 = malloc(4*sizeof(int))
```



```
p2 = malloc(5*sizeof(int))
```

```
p3 = malloc(6*sizeof(int))
```

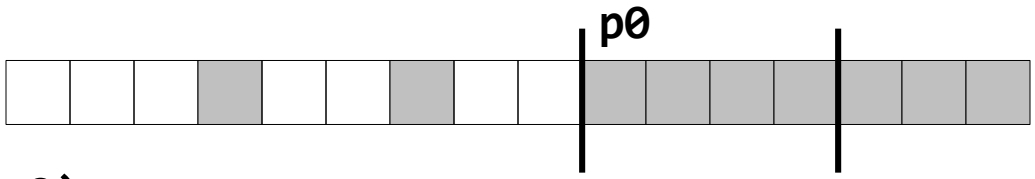
```
free(p2)
```

```
p4 = malloc(7*sizeof(int))
Hopla!
```

- vonkajšia fragmentácia závisí od toho, aké budú budúce požiadavky a preto sa nedá ľahko vyhodnotiť

Čo treba riešiť pri implementácii

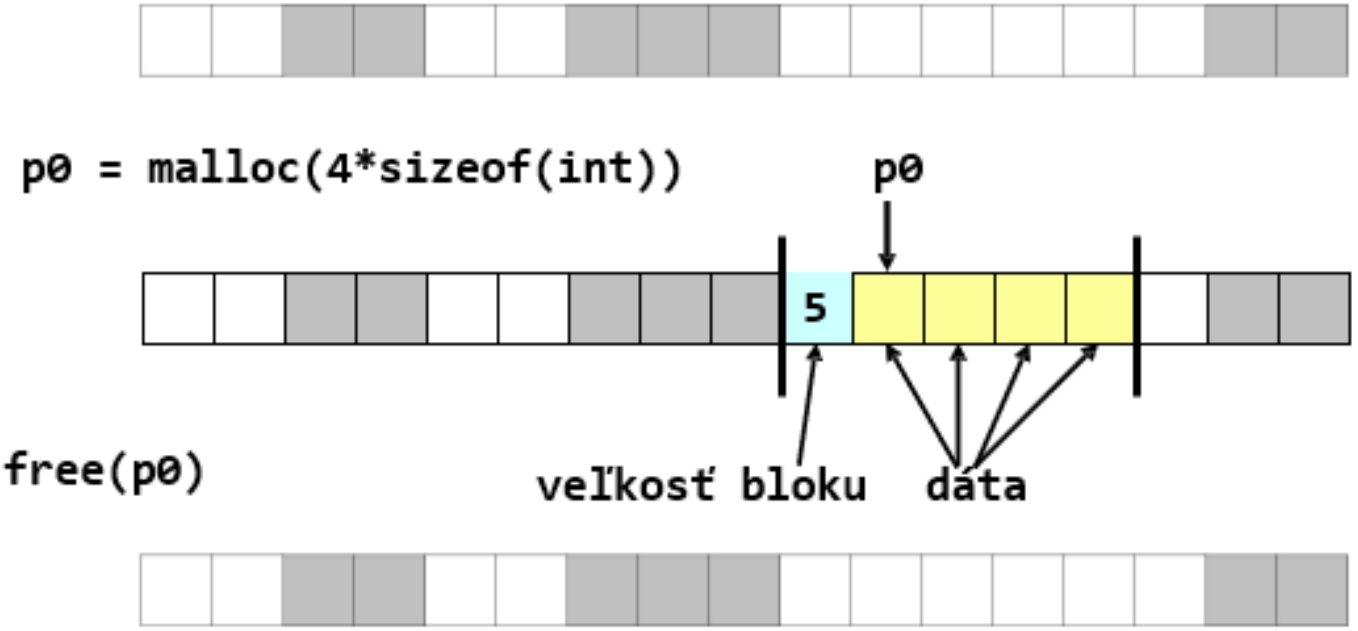
- Ako vieme, koľko pamäti sa má uvoľniť, keď **free** dostane len ukazovateľ?
- Ako si udržiavame záznam o tom, ktoré bloky sú voľné?
- Čo spravíme s nadbytočným kúskom pamäti keď prideľujeme pamäť štruktúre, ktorá je menšia než voľný blok, do ktorého ju umiestňujeme?
- Ako vyberieme blok, ktorý sa použije na pridelenie – môže ich byť viac vhodných?
- Ako vrátime uvoľnený blok do voľnej pamäti?




```
free(p0)
p1 = malloc(1)
```

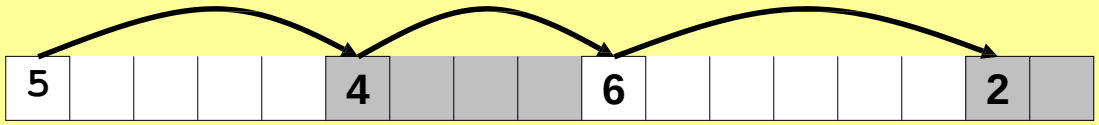
Čo (koľko) sa má vrátiť?

- zapísať dĺžku bloku do slova, predchádzajúceho bloku
  - toto slovo sa často nazýva hlavička
- vyžaduje jedno slovo navyše pre každý pridelený blok

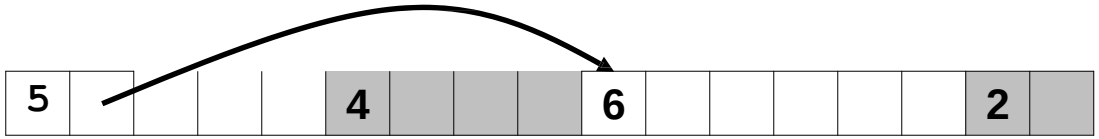


Udržiavanie voľnej pamäte

- **Metóda 1: implicitný zoznam s použitím dĺžok – spája všetky bloky**



- Metóda 2: explicitný zoznam blokov voľnej pamäti pomocou ukazovateľov zapísaných priamo vo voľných blokoch

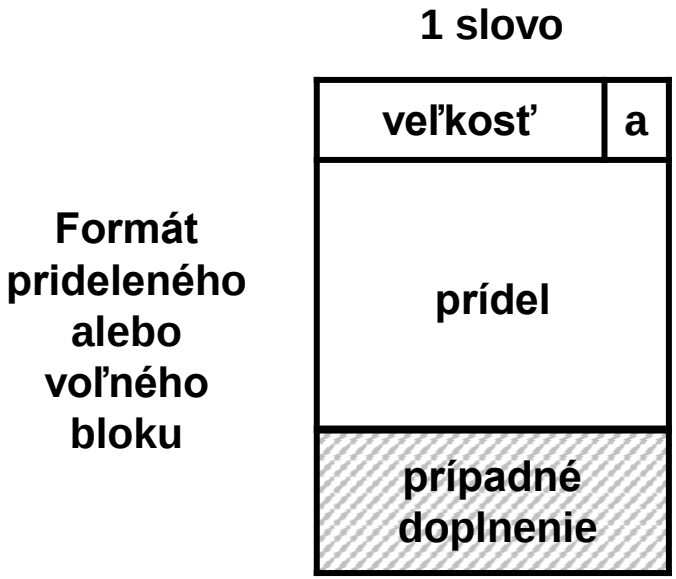


- Metóda 3: oddelené zoznamy blokov voľnej pamäti
  - rôzne zoznamy pre triedy blokov voľnej pamäti podľa dĺžky

- Metóda 4: bloky usporiadané podľa veľkosti
  - možno použiť napr. vyvážený strom s ukazovateľmi zapísanými v každom voľnom bloku, dĺžka bloku je kľúč

Implicitný zoznam blokov pamäti

- Treba rozpoznať (u každého bloku), či je voľný alebo pridelený
  - možno použiť 1 bit (navyše, niekde ho treba vziať)
  - bit možno vyhradiť v rovnakom slove, v ktorom je zapísaná veľkosť bloku ak sú veľkosti blokov vždy zarovnané aspoň na 2 (pri čítaní veľkosti sa maskuje najnižší bit)



a = 1: pridelený blok
a = 0: voľný blok

veľkosť: veľkosť bloku

prídel: údaje vykonávaného programu (len v prípade prideleného bloku)

Nájdenie voľného bloku

- **Prvý vhodný (first fit)**
  - prehľadáva sa zoznam od začiatku, vyberie sa prvý voľný blok, ktorý vyhovuje

```
p = start;
while ((p < end) &&     // nie sme na konci
       ((*p & 1) ||     // už pridelený
       ((*p&~0x1) <= len)))   // príliš malý
  p = NEXT_BLKP(p);
```

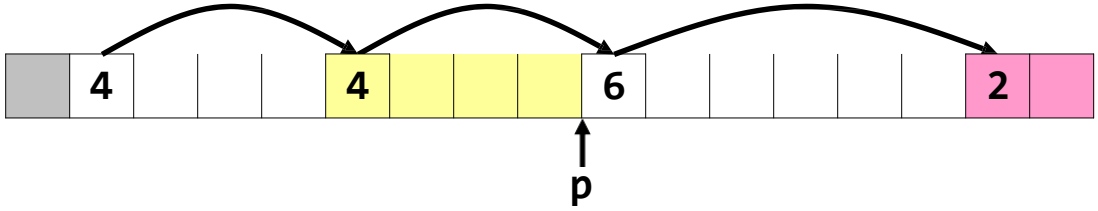
  - môže vyžadovať čas lineárne úmerný celkovému počtu blokov
  - môže spôsobiť postupné vznikanie malých voľných blokov na začiatku zoznamu
- **Nasledujúci vhodný (next fit)**
  - ako metóda prvý vhodný, len sa prehľadávanie začne od miesta, kde skončilo predchádzajúce
  - skúsenosť hovorí, že fragmentácia je horšia
- **Najlepší vhodný (best fit)**
  - vyberie voľný blok s veľkosťou najbližšou k požadovanej (vyžaduje úplné prezretie celého zoznamu)
  - udržiava fragmenty malé
  - pomalší spôsob než prvý vhodný

Nájdenie voľného bloku (2)

- **Najhorší vhodný (worst fit)**
  - vyberie voľný blok s najväčšou veľkosťou (vyžaduje úplné prezretie celého zoznamu)
  - vyhľadanie bloku je pomerne rýchle
  - externá fragmentácia býva horšia

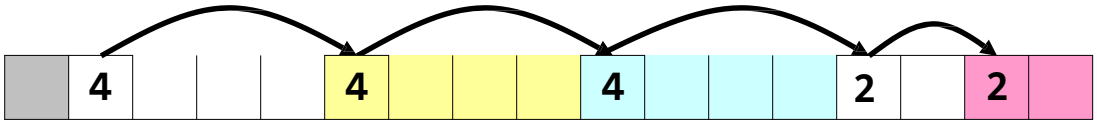
Pridelenie do voľného bloku

- Rozdelenie pôvodného voľného bloku
  - ak sa má prideliť menej pamäti než je veľkosť vybraného voľného bloku, môžeme ho rozdeliť




```
void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;    // zvýš o 1 a zarovnaj hore (na 2)
  int oldsize = *p & ~0x1;                // zamaskuj najnižší bit
  *p = newsize | 0x1;                     // nastav novú dĺžku
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;     // nastav dĺžky v zostavájúcej
}                                         // časti bloku
```

```
addblock(p, 4)
```

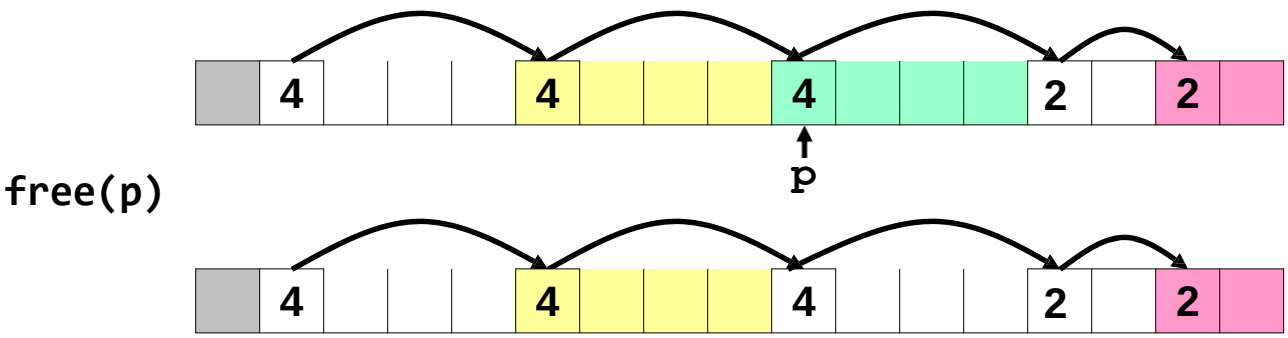


Uvoľnenie bloku

- Najjednoduchšia implementácia:
  - treba len nastaviť príznak voľnosti (najnižší bit na 0)

```
void free_block(ptr p) { *p = *p & ~0x1}
```

  - môže však viesť ku "falošnej fragmentácii"




```
malloc(5*sizeof(int))
```

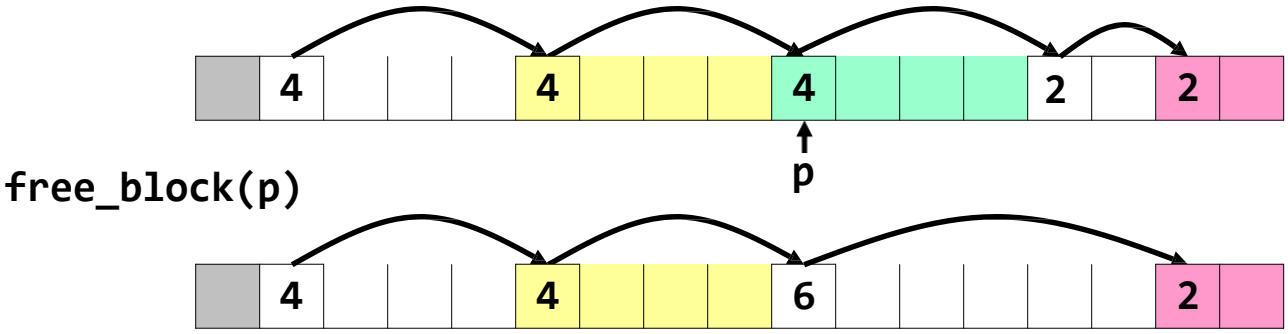
Hopla!

- **Síce je dosť voľnej pamäti na pridelenie bloku veľkosti 5, ale správca ju nevie nájsť!**

Spájanie

- Spojiť s nasledujúcim a/alebo predchádzajúcim blokom ak sú voľné
  - spojenie s nasledujúcim blokom

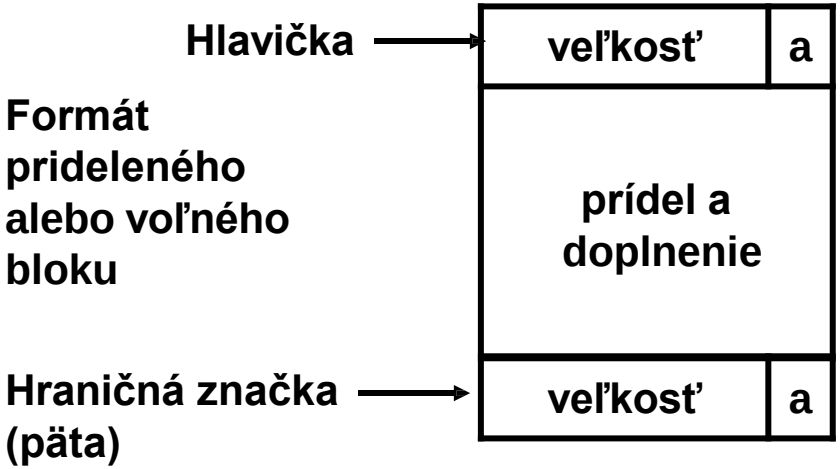
```
void free_block(ptr p) {
    *p = *p & ~0x1;           // vyčisti značku pridelenia
    next = p + *p;            // nájdi nasledujúci blok
    if ((*next & 0x1) == 0) // ak nie je pridelený
      *p = *p + *next;        // pridaj (dĺžku) k tomuto bloku
}
```



- Ale ako spojiť s predchádzajúcim blokom?

Obojsmerné spájanie

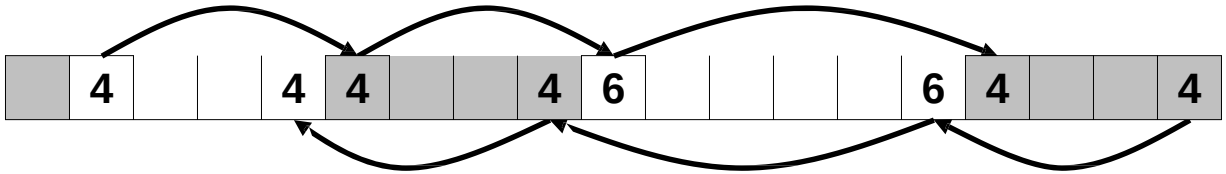
- Hraničné značky (boundary tags) [Knuth73]
  - skopírovať hlavičku aj na konci bloku
  - umožňuje prechádzať zoznam aj pospiatky, vyžaduje však pamäť navyše
  - dôležitá a všeobecná technika!



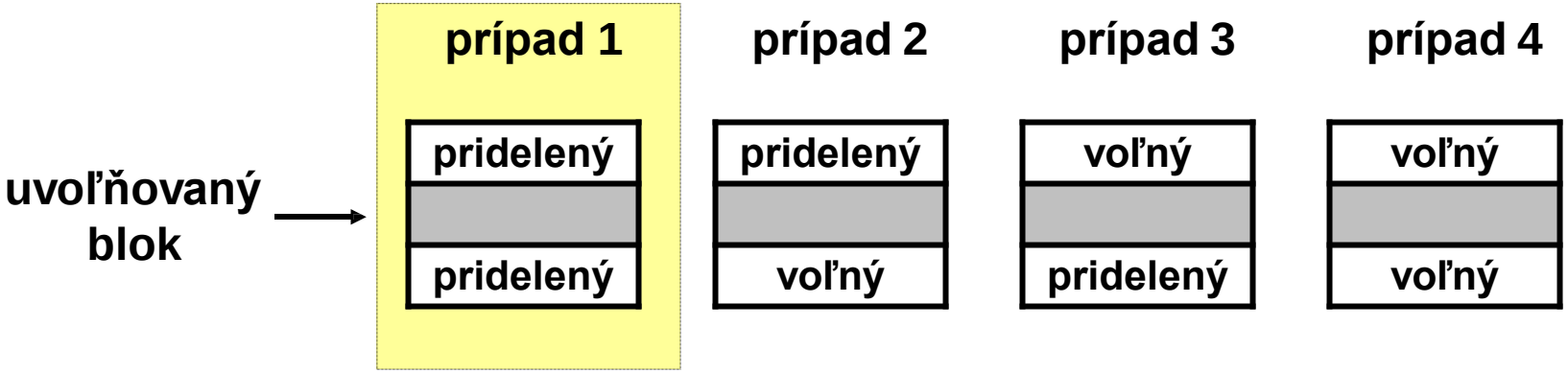
a = 1: pridelený blok
a = 0: voľný blok

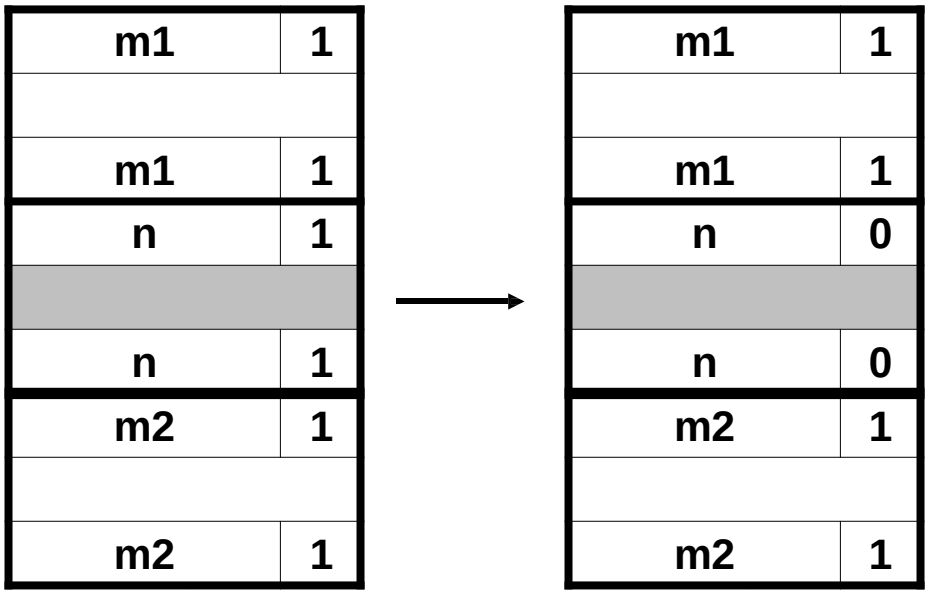
veľkosť: celková veľkosť bloku

prídel: údaje vykonávaného programu (len v prípade prideleného bloku)

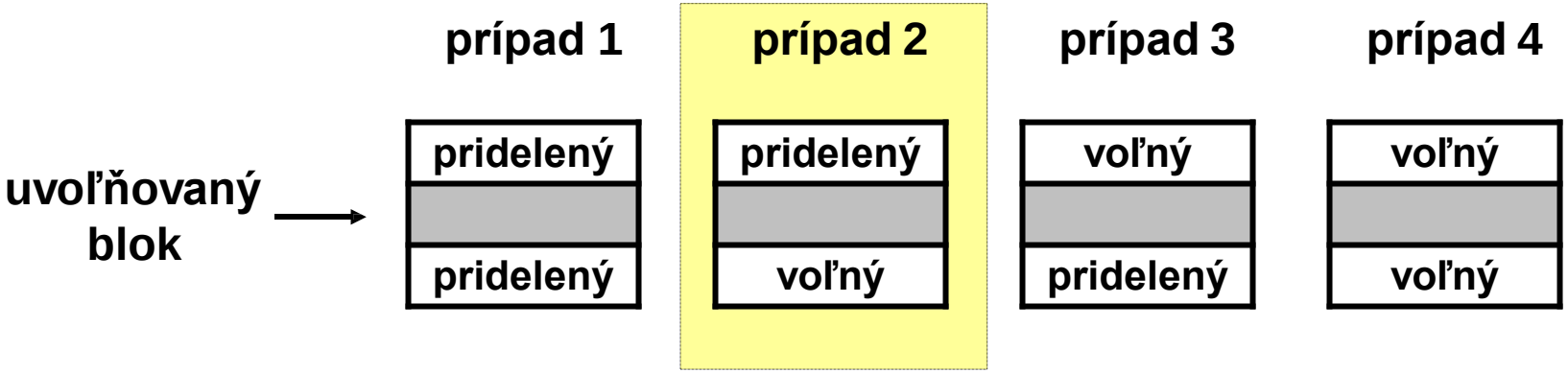


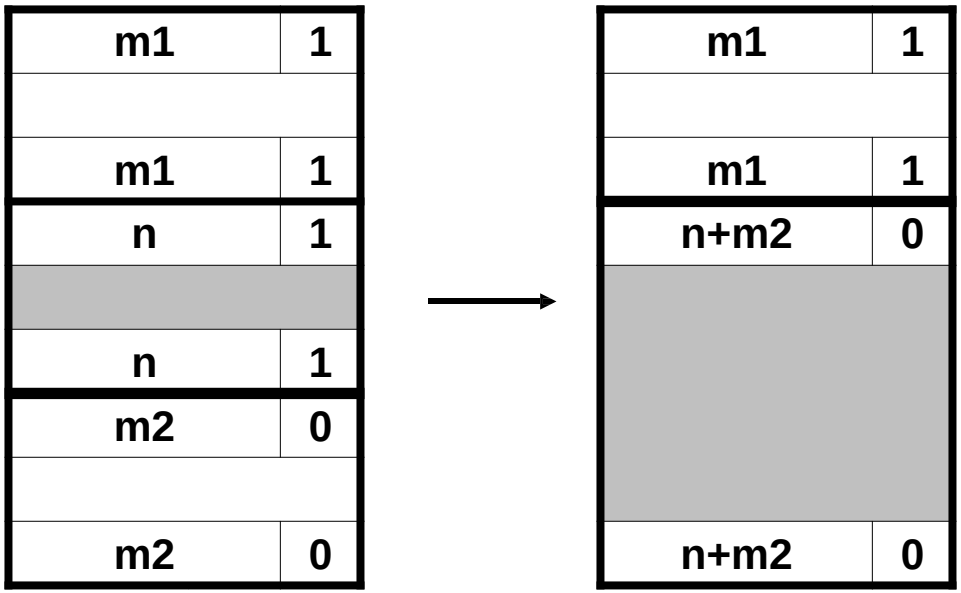
Spájanie v konštantnom čase



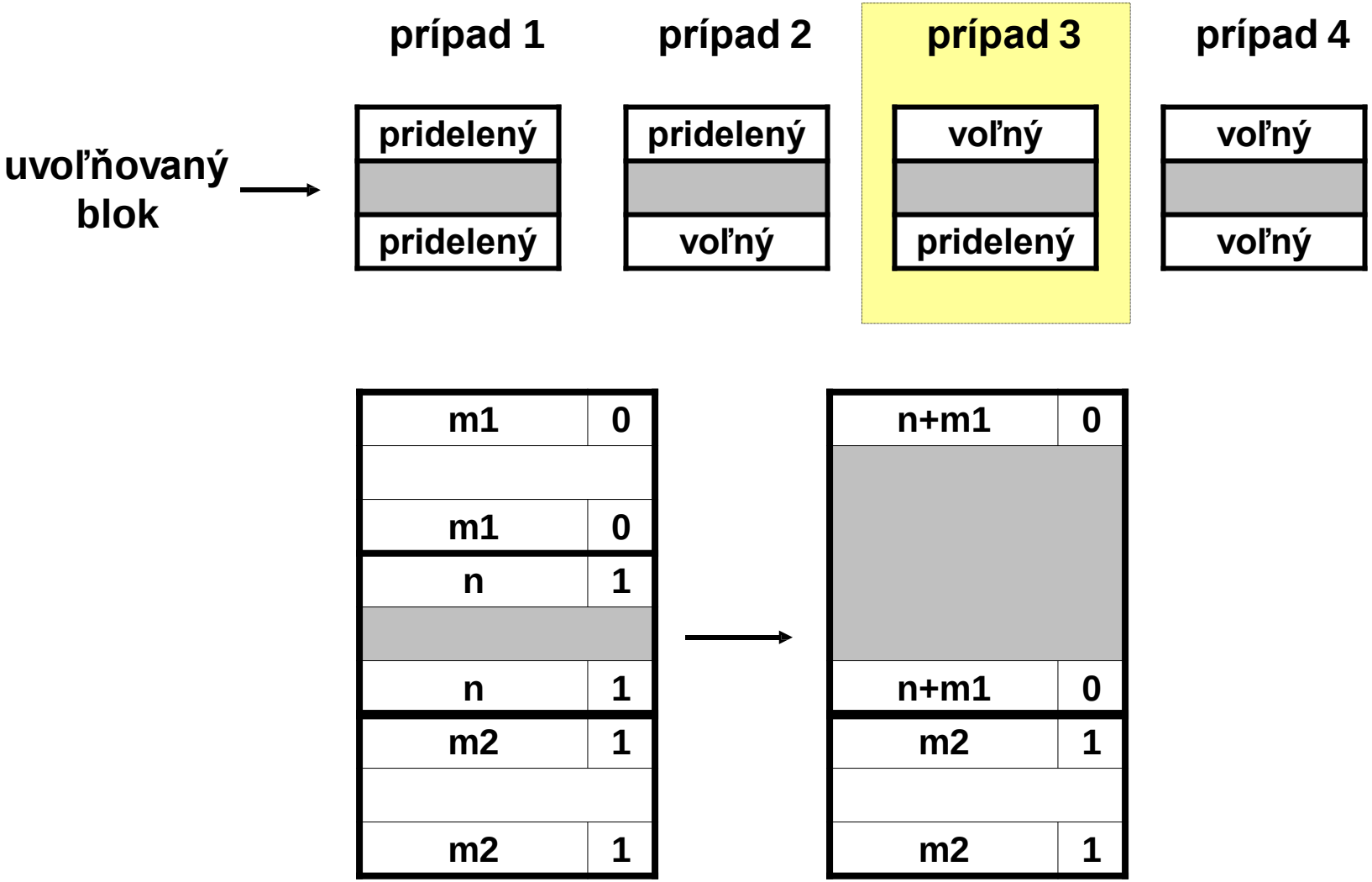


Spájanie v konštantnom čase

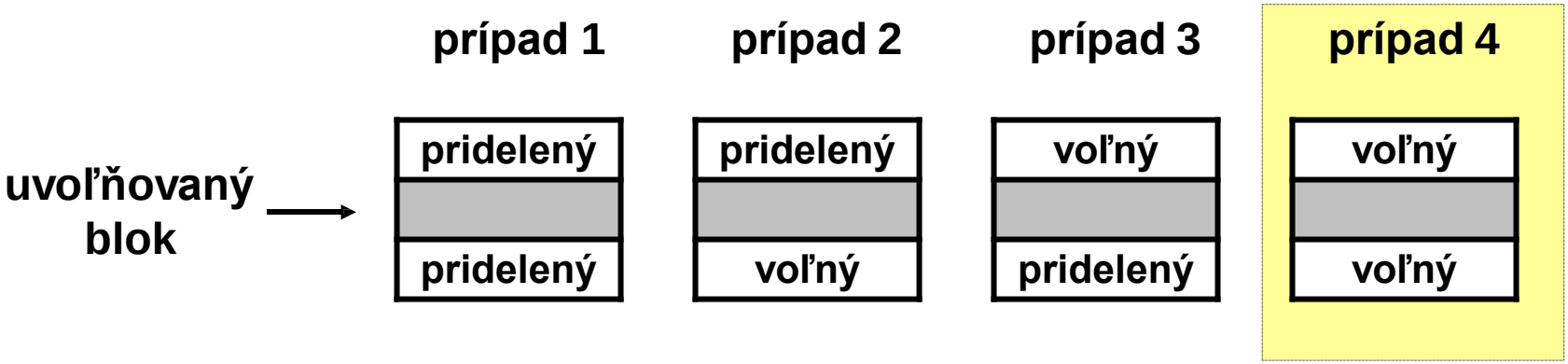


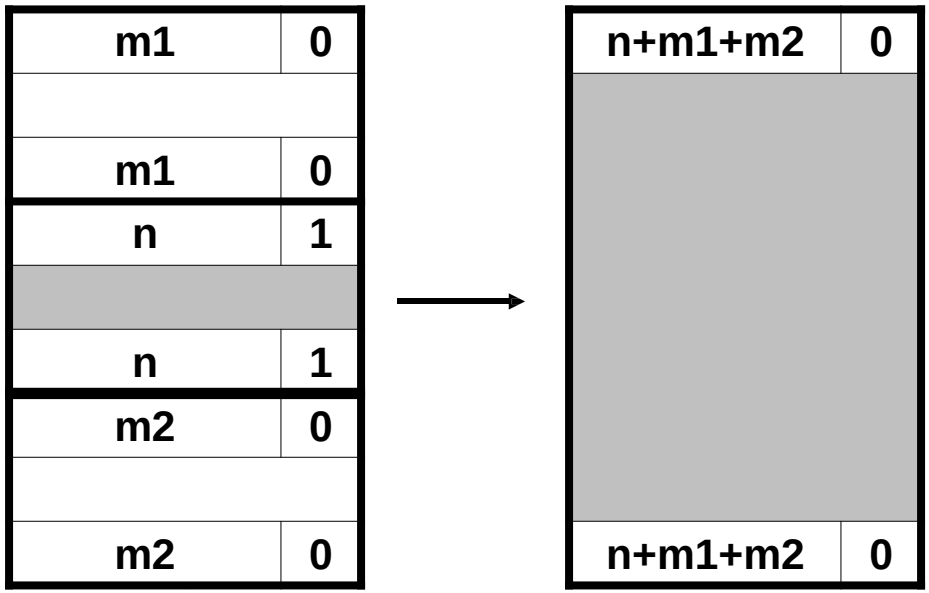


Spájanie v konštantnom čase



Spájanie v konštantnom čase





Rozhodovacie postupy správcu pamäti

- Umiestnenie
  - Prvý vhodný, nasledujúci vhodný, najlepší vhodný, najhorší vhodný…
  - nižšia priepustnosť za nižšiu fragmentáciu
- Rozdelenie
  - Kedy rozdeliť voľný blok?
  - Koľko vnútornej fragmentácie ešte pripustíme?
- Spájanie
  - Okamžité spájanie: spojiť susediace bloky vždy, keď sa zavolá free
  - Odložené spájanie: skúsiť zrýchliť free odložením spájania dovtedy, kým to bude treba, napr.

    spojiť, až keď sa prezerá zoznam voľných blokov pre malloc

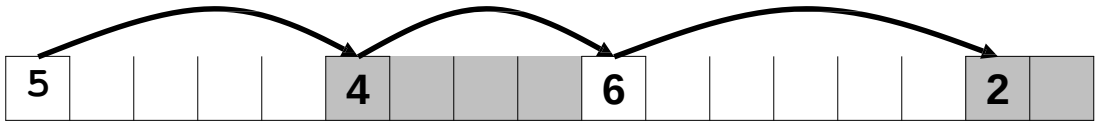
    spojiť, keď rozsah vonkajšej fragmentácie dosiahne nejaký určený prah

Implicitný zoznam blokov pamäti - zhrnutie

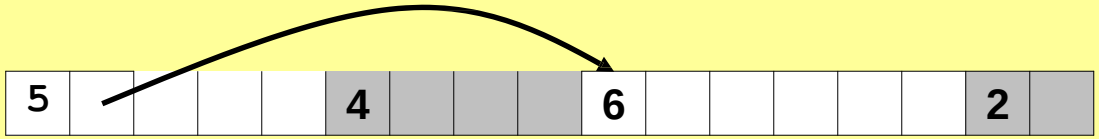
- Jednoduchá implementácia
- Pridelenie v lineárnom čase v najhoršom prípade
- Uvoľnenie v konštantnom čase v najhoršom prípade – dokonca aj so spájaním
- Využitie pamäte závisí od postupu prideľovania
  - Prvý vhodný
  - Nasledujúci vhodný
  - Najlepší vhodný
  - Najhorší vhodný
- V praxi sa nepoužíva pre **`malloc/free`** kvôli lineárnemu času pre prideľovanie
- Pojmy spájania a hraničnej značky sú všeobecné pre všetky metódy správy pamäti

Udržiavanie voľnej pamäte

- Metóda 1: implicitný zoznam s použitím dĺžok – spája všetky bloky



- **Metóda 2: explicitný zoznam blokov voľnej pamäti pomocou ukazovateľov zapísaných priamo vo voľných blokoch**

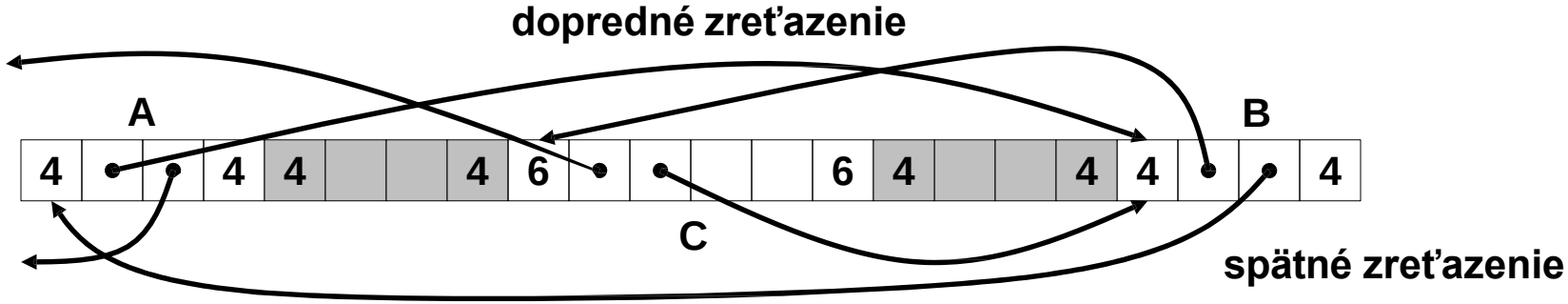


- Metóda 3: oddelené zoznamy blokov voľnej pamäti
  - rôzne zoznamy pre triedy blokov voľnej pamäti podľa dĺžky

- Metóda 4: bloky usporiadané podľa veľkosti
  - možno použiť vyvážený strom s ukazovateľmi zapísanými v každom voľnom bloku, dĺžka je kľúč

Explicitný zoznam blokov voľnej pamäti

- používa sa pamäť pre údaje na ukazovatele
  - typicky sú obojsmerne zreťazené
  - aj tak treba hraničné značky na spájanie



  - poradie v zreťazení nemusí byť rovnaké ako poradie v pamäti

Uvoľnenie do explicitného zoznamu voľných blokov

- Postup pre vloženie:
  Kam do zoznamu voľných blokov vložiť uvoľnený blok?
- Postup LIFO (last-in-first-out)
  - vložiť uvoľnený blok na začiatok zoznamu voľných blokov
  - za: jednoduchá implementácia, vykoná sa v konštantnom čase
  - proti: horšia fragmentácia ako pri postupe zachovávajúcom poradie v pamäti
- Postup zachovávajúci poradie v pamäti (usporiadanie podľa adries)
  - vkladať uvoľnené bloky tak, aby stále boli voľné bloky v zozname v takom poradí, v akom sú adresy, na ktorých sú zapísané v pamäti

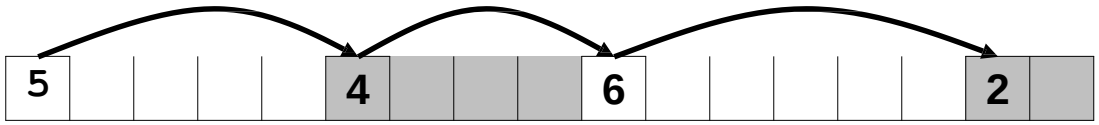
    addr(predchádzajúci) < addr(aktuálny) < addr(nasledujúci)
  - proti: vyžaduje hľadanie
  - za: fragmentácia je lepšia ako pri LIFO

Explicitný zoznam blokov voľnej pamäti – zhrnutie

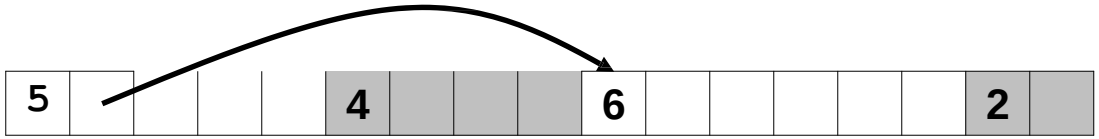
- Porovnanie s implicitným zoznamom:
  - pridelenie je v lineárnom čase závislé od počtu voľných blokov namiesto počtu všetkých blokov – je omnoho rýchlejšie keď je väčšina pamäte plná
  - trochu zložitejšie pridelenie aj uvoľnenie lebo treba zabezpečiť preskočenie bloku
  - o niečo viac pamäti treba na 2 ukazovatele (2 slová navyše treba pre každý blok)
- Hlavné použitie zreťazených zoznamov voľnej pamäti je v súvislosti s oddelenými zoznamami (Metóda 3)
  - udržiavať viacero reťazených zoznamov voľnej pamäti podľa veľkosti blokov alebo typu objektov

Udržiavanie voľnej pamäte

- Metóda 1: implicitný zoznam s použitím dĺžok – spája všetky bloky



- Metóda 2: explicitný zoznam blokov voľnej pamäti pomocou ukazovateľov zapísaných priamo vo voľných blokoch

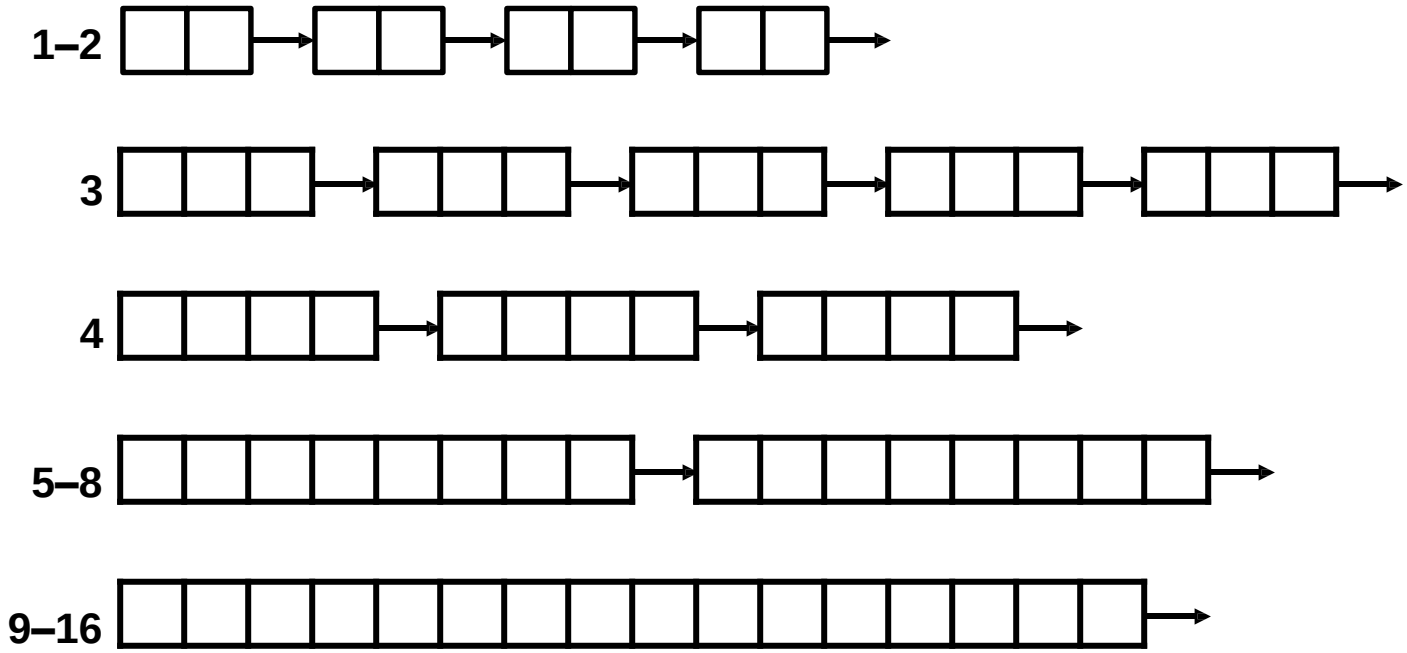


- **Metóda 3: oddelené zoznamy blokovvoľnej pamäti**
  - rôzne zoznamy pre triedy blokov voľnej pamäti podľa dĺžky

- Metóda 4: bloky usporiadané podľa veľkosti
  - možno použiť vyvážený strom s ukazovateľmi zapísanými v každom voľnom bloku, dĺžka je kľúč

Oddelená (segregovaná) pamäť

- Každá trieda veľkostí blokov má svoj zoznam



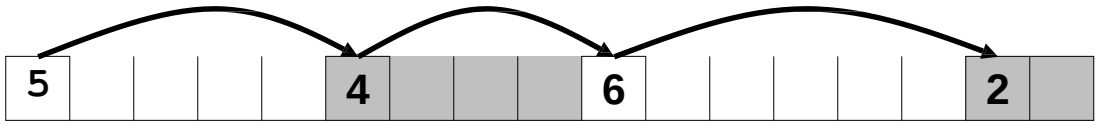
- Zvyčajne sú oddelené triedy pre každú malú veľkosť (2,3,...)
- Väčšie veľkosti sa zoskupia podľa mocniny 2

Pridelenie a uvoľnenie v oddelenej pamäti

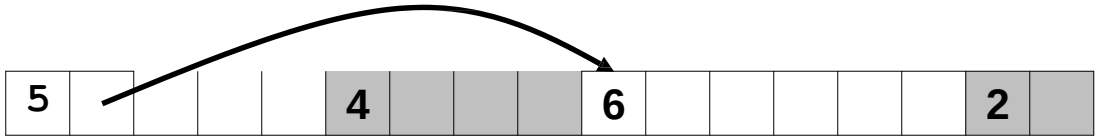
- Prideliť blok veľkosti N:
  - prehľadať vhodný zoznam voľných blokov hľadajúc blok veľkosti m >= n
  - ak sa nájde vhodný blok:
    rozdeliť blok a umiestniť zvyšok do vhodného zoznamu (ak prichádza do úvahy)
  - ak sa nenájde vhodný blok v tomto zozname, skúsiť zoznam s triedou najbližších väčších blokov
  - opakuj, dokiaľ sa nájde blok
- Uvoľniť blok:
  - spojiť a umiestniť do vhodného zoznamu
- Vlastnosti
  - hľadanie je rýchlejšie než pri sekvenčnej organizácii (logaritmický čas pre triedy veľkostí podľa mocniny 2)
  - spájanie môže predĺžiť hľadanie
    odloženie spájania to môže zlepšiť

Udržiavanie voľnej pamäte

- Metóda 1: implicitný zoznam s použitím dĺžok – spája všetky bloky



- Metóda 2: explicitný zoznam blokov voľnej pamäti pomocou ukazovateľov zapísaných priamo vo voľných blokoch



- Metóda 3: oddelené zoznamy blokov voľnej pamäti
  - rôzne zoznamy pre triedy blokov voľnej pamäti podľa dĺžky

- **Metóda 4: bloky usporiadané podľa veľkosti**
  - možno použiť vyvážený strom s ukazovateľmi zapísanými v každom voľnom bloku, dĺžka je kľúč