Dátové štruktúry a algoritmy

2013

Môžete získať až 100 bodov:

- priebežne riešené úlohy (testy a zadania, max. 45 bodov: na cvičeniach 25 a doma 20):
- na cvičení sa budú riešiť testy a zadania:
  - test (každé druhé cvičenie) na základné znalosti o téme cvičenia, najviac za 1 bod (spolu najviac 5).
  - zadania (môže ich byť viac, každé najviac za 2 body), do konečného hodnotenia sa započíta len jeden (lepší) výsledok v rámci jedného cvičenia (spolu najviac 2 body za cvičenie)
- doma sa budú riešiť 3 zadania (homework assignments):
  - prvé zadanie má vytvoriť vlastnú implementáciu dynamickej pamäti (do 8 bodov).
  - druhé a tretie zadanie (do 6 bodov každé). ak treba dynamickú pamäť, má sa použiť vlastná implementácia.

- priebežný test midterm exam (max. 15 bodov):
- záverecná skúška (max. 40 bodov) final exam

Podmienky absolvovania predmetu:

- Zápočet:
- získať minimálne 18 bodov z priebežne riešených úloh
  - (v tom minimalne 3 body z prveho zadania A
  - minimalne 2 body z druheho zadania A
  - minimalne 2 body z tretieho zadania)
- A
- získať minimálne 5 bodov z priebežného testu

- Predmet:
- získať zápočet A
- získať minimálne 18 bodov zo skúšky A
- získať minimálne 56 bodov spolu.

- Všetko, čo sa predkladá na hodnotenie, musí byť vlastná samostatná práca študenta alebo musí byť označené ako prevzaté. Samozrejme, body možno získať len za vlastnú prácu.
  Opisovanie sa netoleruje. Pokiaľ sa študent pokúša absolvovať tento predmet nie vlastnou prácou, kvalifikuje sa na FX.

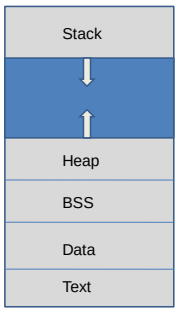
Prideľovanie pamäti

O čo ide?

- program po preložení (compile), spojení s externými podprogramami apod. (link) a uložení do pamäti počítača (load) sa vykonáva (execute) – proces
- bežiacemu procesu sa musí prideliť v počítači pamäť, aby mal kam zapisovať údaje (medzivýsledky atď.)
- čo a ako?

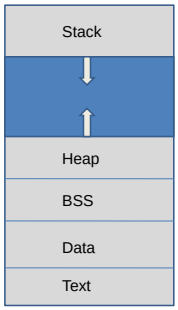
adresový priestor pridelený procesu

- Text: obsahuje program v strojovom jazyku, ktorý sa vykonáva, údaje typu reťazec, konštanty, ďalšie údaje, ktoré sa len čítajú
- Data: inicializované globálne a statické premenné
- BSS: (Block Started by Symbol); neinicializované globálne a statické premenné



adresový priestor pridelený procesu

- Stack (zásobník): lokálne premenné bežiaceho procesu
- Heap (halda voľnej pamäti):
  - dynamická pamäť procesu
  - môže sa zväčšovať aj zmenšovať (prečo?)
    - vieme, koľko objektov bude treba pri výpočte (podľa daného programu) predtým, ako sa program začne vykonávať?
  - toto je pamäť, ktorú vracia (prideľuje) malloc()

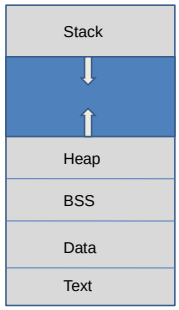


príklad 1

- uvažujme tento program

```
Data: globálna premenná   Text (read-only data)
char string = "hello"
int iSize;                BSS
char *f(void)
{
    char *p;              Stack; lokálna premenná
    iSize = 8;
    p = malloc(iSize);    Heap; dynamická pamäť
    return p;
}
```

preložený program je Text



prideľovanie pamäti

- 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)

9

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() zahŕňa parameter veľkosť (size)

```
#include <stdlib.h>
void * malloc (size_t size); //size požadovaná veľkosť
```

príklad 2

```
struct polozka_uctu *polozka;
/* * prideľ dostatok pamäti na zapísanie údaja typu
   struct polozka_uctu *  a nastav do polozka
   ukazovateľ na ňu */
polozka = malloc (sizeof (struct polozka_uctu));
// ak bolo pridelenie úspešné, vracia ukazovateľ
   na začiatok práve pridelenej oblasti pamäti
if (!polozka)
   perror ("malloc");
```

prideľovanie polí

- čo ak je dynamická aj samotná veľkosť
- napr. koľko prvkov má obsahovať pole
- v takom prípade možno použiť calloc()

```
#include <stdlib.h>
void * calloc (size_t nr, size_t size);
```

calloc(m, n) je to isté ako
p = malloc(m * n); if(p) memset(p, 0, m * n);

príklad na vytvorenie poľa

- vytvoriť pole celých čísel

```
int *x, *y;
x = malloc (50 * sizeof (int));
if (!x) {
    perror ("malloc");
    return -1;
    }
y = calloc (50, sizeof (int));
if (!y) {
   perror ("calloc");
   return -1;
}
```

ale…
malloc () nezaručuje, čo bude obsah pridelenej pamäti (t.j. neinicializuje ju)

calloc() zaručuje obsah, t.j. inicializuje (samé nuly)

takže…
prvky poľa y majú všetky hodnotu 0, ale prvky poľa x nemajú definovanú hodnotu

zmena veľkosti pridelenej pamäti a uvoľnenie

- veľkosť pridelenej pamäti možno zmeniť pomocou realloc()

```
#include <stdlib.h>
void * realloc (void *ptr, size_t size);
```

- uvoľniť pridelenú pamäť, t.j. vrátiť ju do voľnej pamäti možno pomocou free()
  - void free (void *ptr);

príklad

1. Vyhradiť pamäť (alokovať) n polí znakov. Každé pole obsahuje od 2 do n+1 prvkov
2. Pre každé pole, cyklus vpíše znak c do každého prvku
3. Vytlačiť pole ako reťazec
4. Vrátiť polia do voľnej pamäti

```
void print_chars (int n, char c) {
    int i;
    for (i = 0; i < n; i++) { //vytvor n polí
            char *s;
             int j; /* * prideľ a vynuluj  i+2-prvkové pole * znakov. Všimnime si, že
    'sizeof (char)' * je vždy 1. */

            s = calloc (i + 2, 1); //každé pole má veľkosť 2 alebo viac, ako pole znakov
            for (j = 0; j < i + 1; j++)  //inicializuje sa pole znakmi c
                    s[j] = c;
            printf ("%s\n", s); /* vytlačiť pole a už len vrátiť pamäť. */
            free (s);
            } }
```

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*
  - ak sa v programe urobí odkaz na pamäť, ktorá bola medzitým uvoľnená, je to odkaz visiaci vo vzduchu dangling reference

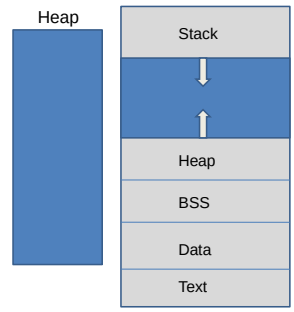
* leak – diera, otvor, puklina, trhlina, únik, priesak, netesnosť, prezradenie

implementácia malloc()

- zavolanie malloc() spôsobí, že sa pridelí pamäť veľkosťou čo najbližšia (closest fit) požadovanej
- zavolanie free() spôsobí, že pridelená pamäť sa uvoľní, t.j. vráti späť do voľnej pamäti

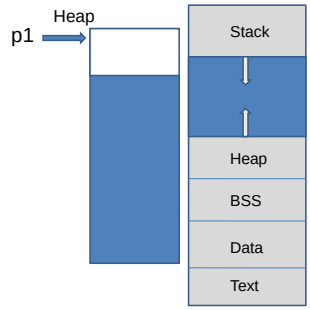
príklad

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



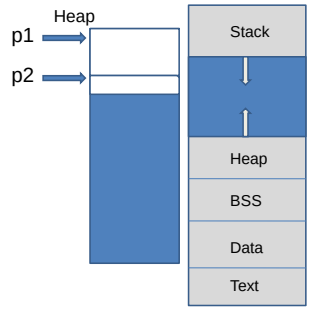
príklad

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



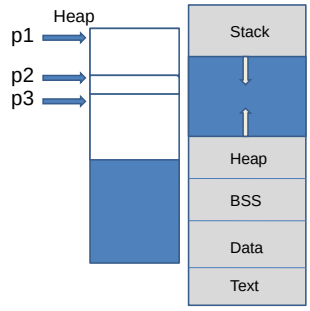
príklad

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



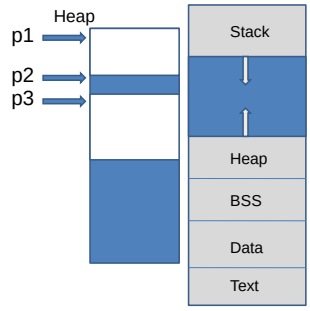
príklad

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



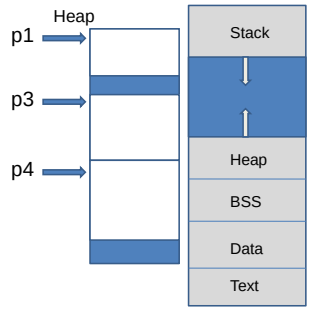
príklad

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



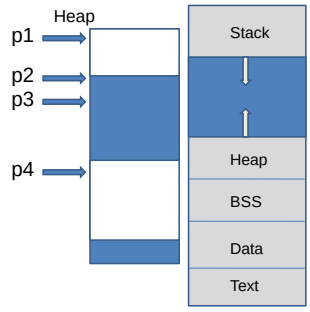
príklad

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



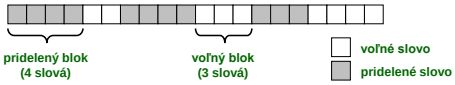
príklad

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



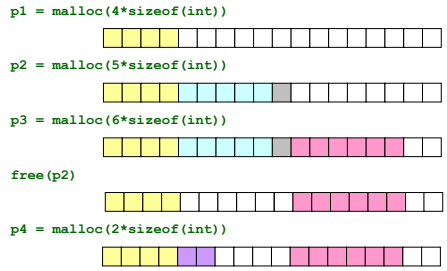
dohoda o predpokladoch pre ďalší výklad

– pamäť sa adresuje po slovách
– "štvorčeky" na obrázkoch predstavujú slová
– každé slovo môže obsahovať celé číslo alebo smerník/ukazovateľ



25

príklady pridelenia pamäti



26

ohraničenia

- programy, ktoré sa vykonávajú:
  - môže mať ľuboľnú postupnosť požiadaviek malloc a free
  - požiadavky na free musia sa vzťahovať na pridelenú pamäť
- správca dynamickej pamäti
  - neovláda počet ani veľkosť prideľovaných blokov pamäti
  - musí vyhovovať všetkým požiadavkám okamžite
    - t.j., nemôže ich preusporiadať alebo odložiť na neskôr
  - musí prideľovať pamäť z voľnej pamäti
  - musí zarovnať veľkosť bloku tak, aby splnila všetky požiadavky na zarovnávanie
    - často zarovnanie na 8 slabík (bajtov)
  - môže manipulovať a meniť iba voľnú pamäť
  - nemôže presúvať už pridelený blok pamäti
    - t.j., nebudeme predpokladať možnosť skompaktňovania

27

ciele dobrej implementácie `malloc`/`free`

- prvotné ciele
  - 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 čo najväčšiu časť haldy
    - minimalizovať "fragmentáciu"
- niektoré ďalšie ciele
  - vlastnosti dobrej lokálnosti
    - štruktúry pridelené blízko v čase by mali byť blízko seba v pamäti
    - "podobné" objekty by mali byť umiestnené blízo seba
  - robustnosť
    - vie overiť, že `free(p1)` sa týka platného prideleného objektu `p1`
    - vie overiť, ukazovatele odkazujú do prideleného úseku pamäti

28

maximalizovanie priepustnosti

- nech je daná nejaká postupnosť volaní (požiadaviek) malloc a free:
  - $R_0, R_1, ..., R_k, ... , R_{n-1}$
- treba maximalizovať priepustnosť a špičkové využitie pamäti
  - tieto ciele sú často v protiklade
- priepustnosť:
  - počet vybavených požiadaviek za jednotku času
  - príklad:
    - 5,000 malloc volaní a 5,000 free volaní počas 10 sekúnd
    - priepustnosť je 1,000 operácií/s

29

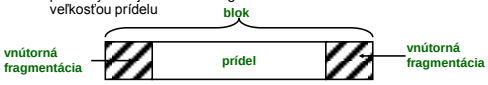
maximalizovanie využitia pamäti

- nech je daná nejaká postupnosť volaní (požiadaviek) malloc a free:
  - $R_0, R_1, ..., R_k, ... , R_{n-1}$
- Def: agregátny prídel (payload) P_k:
  - `malloc(p)` má ako výsledok, že sa pridelí blok s prídelom `p` slov
  - po dokončení požiadavky R_k je agregovaný prídel P_k súčet momentálne pridelených prídelov
- Def: momentálna veľkosť haldy sa označí H_k
  - predpokladáme, že H_k sa monotónne zväčšuje
- Def: špičkové využitie pamäti:
  - po k požiadavkách je:
    - $U_k = (\max_{i<k} P_i) \ / \ H_k$

30

vnútorná fragmentácia

- nízke využitie pamäti spôsobuje fragmentácia
  - vnútorná a vonkajšia
- vnútorná fragmentácia
  - pre daný blok je vnútorná fragmentácia rozdiel medzi veľkosťou bloku a veľkosťou prídelu

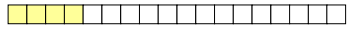


  - spôsobuje ju réžia (overhead) udržiavania dynamickej pamäti, zarovnávanie, prípadne rozhodnutia správy pamäti (napr. nerozbiť blok)
  - je určená tým, aké požiadavky boli doteraz, dá sa ľahko vyhodnotiť

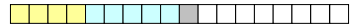
vonkajšia fragmentácia

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

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

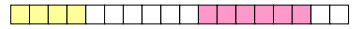


```
p2 = malloc(5*sizeof(int))
```



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

```
free(p2)
```



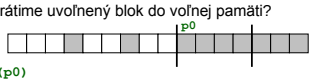
```
p4 = malloc(7*sizeof(int))
```

hopla!

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

čo treba riešiť pri implementácii

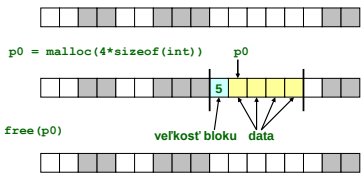
- Ako vieme, koľko pamäti sa má uvoľniť, keď free dostane len ukazovateľ?
- Ako si udržiavame záznam o tom, ktoré bloky sú voľné?
- Čo spravíme s nadbytočným kúskom pamäti keď prideľujeme pamäť štruktúre, ktorá je menšia než voľný blok, do ktorého ju umiestňujeme?
- Ako vyberieme blok, ktorý sa použije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




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


čo (koľko) sa má vrátiť?

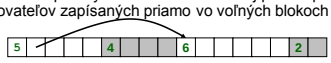
- bežný postup
  - zapísať dĺžku bloku do slova, predchádzajuceho bloku
    - toto slovo sa často nazýva hlavička
  - vyžaduje jedno slovo navyše pre každý pridelený blok



udržiavanie voľnej pamäti

- 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 (napr. červeno-čierny) s ukazovateľmi zapísanými v každom voľnom bloku, dĺžka je kľúč

Metóda 1: implicitný zoznam

- 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)

implicitný zoznam: nájdenie voľného bloku

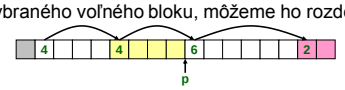
- prvý vhodný (first fit)
  - prehľadáva sa zoznam od začiatku, vyberie sa prvý voľný blok, ktorý vyhovuje

```
p = start;
while ((p < end) &&     \\ not past end
       ((*p & 1) ||     \\ already allocated
        (*p <= len)))   \\ too small
  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 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

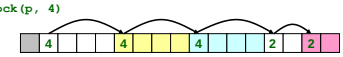
implicitný zoznam: pridelenie do voľného bloku

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



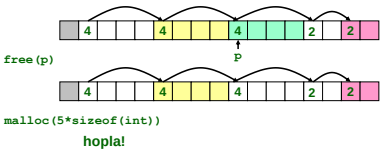

```
void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;  // add 1 and round up
  int oldsize = *p & ~0x1;              // mask out low bit
  *p = newsize | 0x1;                   // set new length
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;   // set length in remaining
}                                       //   part of block
```

addblock(p, 4)



implicitný zoznam: 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"

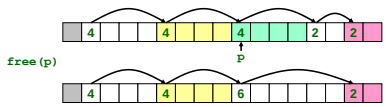


  - síce je dosť voľnej pamäti na pridelenie bloku veľkosti 5, ale správca ju nevie nájsť!

implicitný zoznam: spájanie

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

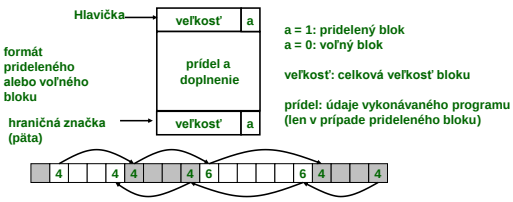
```
void free_block(ptr p) {
    *p = *p & ~0x1;          // clear allocated flag
    next = p + *p;           // find next block
    if ((*next & 0x1) == 0)
      *p = *p + *next;       // add to this block if
}                            //    not allocated
```



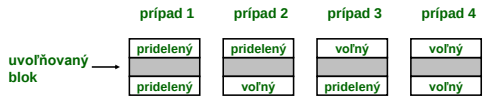
  - ale ako spojiť s predchádzajúcim blokom?

implicitný zoznam: obojsmerné spájanie

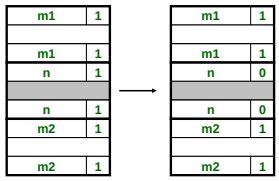
- hraničné značky(boundary tags) [Knuth73]
  - skopírovať hlavičku aj na konci bloku
  - umožňuje prechádzať zoznam aj pospiatky, vyžaduje však pamäť navyše
  - dôležitá a všeobecná technika!



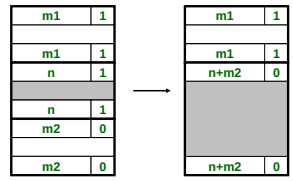
spájanie v konštantnom čase



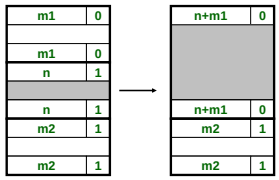
spájanie v konštantnom čase (prípad 1)



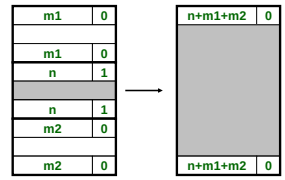
spájanie v konštantnom čase (prípad 2)



spájanie v konštantnom čase (prípad 3)



spájanie v konštantnom čase (prípad 4)



súhrn najdôležitejších rozhodovacích postupov správcu pamäti

- umiestnenie:
  - prvý vhodný, nasledujúci vhodný, najlepší vhodný atď.
  - nižšia priepustnosť za nižšiu fragmentáciu
- rozdelenie:
  - Kedy rozdeliť voľný blok?
  - Koľko vnútornej fragmentácie ešte pripustíme?
- spájanie:
  - okamžité spájanie: spojiť susediace bloky vždy keď sa volá 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

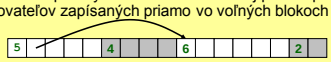
implicitné zoznamy: súhrn

- implementácia: veľmi jednoduchá
- pridelenie: v lineárnom čase v najhoršom prípade
- uvoľnenie: v konštantom čase v najhoršom prípade – dokonca aj so spájaním
- využitie pamäti: závisí od postupu prideľovania
  - prvý vhodný, nasledujúci vhodný alebo najlepší vhodný
- v praxi sa nepoužíva pre `malloc`/`free` kvôli lineárnemu času pre prideľovanie
  - používa sa v mnohých zvláštnych prípadoch aplikácií
- pojmy spájania a hraničnej značky sú všeobecné pre všetky metódy správy pamäti

udržiavanie voľnej pamäti

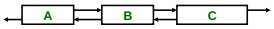
- Metóda 1: implicitný zoznam s použitím dĺžok – spája všetky bloky

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

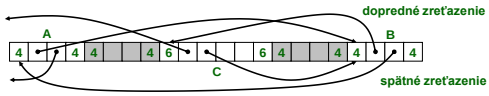


- Metóda 3: oddelené zoznamy blokov voľnej pamäti
  - rôzne zoznamy pre triedy blokov voľnej pamäti podľa dĺžky
- Metóda 4: bloky usporiadané podľa veľkosti
  - možno použiť vyvážený strom (napr. červeno-čierny) s ukazovateľmi zapísanými v každom voľnom bloku, dĺžka je kľúč

explicitný zoznam blokov voľnej pamäti

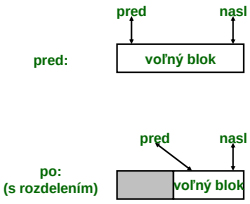


- používa sa pamäť pre údaje na ukazovatele
  - typicky sú obojsmerne zreťazené
  - aj tak treba hraničné značky na spájanie



  - poradie v zreťazení nemusí byť rovnaké ako poradie v pamäti

pridelenie z explicitného zoznamu voľných blokov

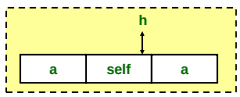


uvoľnenie do explicitného zoznamu voľných blokov

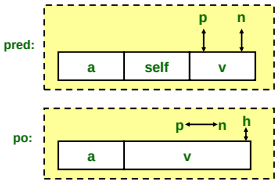
- postup pre vloženie: Kam do zoznamu voľných blokov vložiť uvoľnený blok?
  - postup LIFO (last-in-first-out)
    - vložiť uvoľnený blok na začiatok zoznamu voľných blokov
    - za: jednoduchá implementácia, vykoná sa v konštatnom čase
    - proti: horšia fragmentácia ako pri postupe zachovávajúcom 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
      - t.j. addr(pred) < addr(súč) < addr(nasl)
    - proti: vyžaduje hľadanie
    - za: fragmentácia je lepšia ako pri LIFO

uvoľnenie s LIFO

•prípad 1: a-a-a
  – vložiť self na začiatok voľného zoznamu



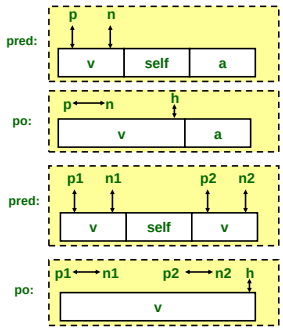
•prípad 2: a-a-v
  – preskočiť nasledujúci vo voľnom zozname, pripojiť nasledujúci a pridať na začiatok voľného zoznamu



uvoľnenie s LIFO

•prípad 3: v-a-a
  – preskočiť predchádzajúci vo voľnom zozname, pripojiť predchádzajúci a pridať na začiatok voľného zoznamu

•prípad 4: v-a-v
  – preskočiť predchádzajúci a nasledujúci vo voľnom zozname, pripojiť predchádzajúci a nasledujúci a pridať na začiatok voľného zoznamu



explicitný zoznam: súhrn

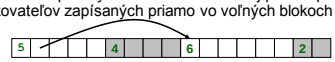
- porovnanie s implicitným zoznamom:
  - pridelenie je v lineárnom čase závislé od počtu voľných blokov namiesto počtu všetkých blokov – je omnoho rýchlejšie keď je väčšina pamäti plná
  - trochu zložitejšie pridelenie aj uvoľnenie lebo treba zabezpečiť preskočenie bloku
  - o niečo viac pamäti treba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

55

udržiavanie voľnej pamäti

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

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

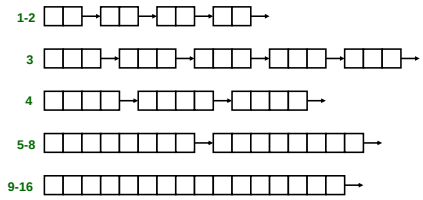


- Metóda 3: oddelené zoznamy blokov voľnej pamäti
  - rôzne zoznamy pre triedy blokov voľnej pamäti podľa dĺžky
- Metóda 4: bloky usporiadané podľa veľkosti
  - možno použiť vyvážený strom (napr. červeno-čierny) s ukazovateľmi zapísanými v každom voľnom bloku, dĺžka je kľúč

56

oddelená (segregovaná) pamäť

- každá trieda veľkostí blokov má svoj zoznam



- často oddelené triedy pre každú malú veľkosť (2,3,4,…)
- väčšie veľkosti sa často zgrupujú podľa mocniny 2

57

pridelenie a uvoľnenie v oddelenej pamäti

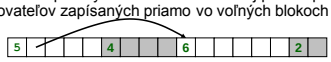
- prideliť blok veľkosti n:
  - prehľadať vhodný zoznam voľných blokov hľadajúc blok veľkosti m >= n
  - ak sa nájde vhodný blok:
    - rozdeliť blok a umiestniť zvyšok do vhodného zoznamu (ak prichádza do úvahy)
  - ak sa nenájde vhodný blok v tomto zozname, skúsiť zoznam s triedou najbližších väčších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

58

udržiavanie voľnej pamäti

- 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 (napr. červeno-čierny) s ukazovateľmi zapísanými v každom voľnom bloku, dĺžka je kľúč

59


acm International Collegiate Programming Contest

Súťaž podporuje informatika a

Nadácia pre rozvoj GRATEX INTERNATIONAL

ACM súťaž je opäť tu!

Lokálne kolo programátorskej súťaže na STU prebehne v rámci **CTU Open Contest** už **18.-19.10.2013**

Info: www.fiit.stuba.sk/acm

Max. 3-členné družstvá pošlite mail na adresu **acm.icpc@fiit.stuba.sk do stredy 16.10.2013 do 18:00 hod.** Ako odpoveď na Váš mail dostanete potvrdenie účasti a ďalšie pokyny k dokončeniu registrácie. Počet tímov je obmedzený!

PROGRAMÁTORSKÁ SÚŤAŽ !

- Lokálne kolo ACM súťaže na FIIT STU prebehne v rámci CTU Open Contest **18.- 19.10.2013.** Súťaž bude súčasne prebiehať na univerzitách v Čechách aj na Slovensku. Organizátorom je ČVUT v Prahe.
- Max. 3-členné družstvá, ktoré sa chcú zúčastniť, nech pošlú mail na adresu **acm.icpc@fiit.stuba.sk** do stredy 16.10.2013 do 18:00 hod. Ako odpoveď na Váš mail dostanete potvrdenie účasti a ďalšie pokyny.
- **REGISTRÁCIA** Zaregistrujte sa elektronicky na tejto stránke . Tu je návod na registráciu. Pri registrácii používajte vo svojich menách diakritiku.
- **PROGRAM** K dispozícii je predbežný program súťaže lokálneho kola v Prahe CTU Open Contest. Počas súťaže je možné používať literatúru (programátorskú, anglicko-slovenský slovník a pod.) výlučne v tlačenej forme. Zadania príkladov sú v anglickom jazyku.

- **PIATOK 18.10.2013**
  14:30 - 15:00 Registrácia (stačí jeden člen tímu)
  15:20 - 15:40 Otvorenie, privítanie
  15:40 - 17:00 Informácie o sútažnom prostredí a spôsobe vyhodnocovania.(Videoprenos z Prahy.)
  17:00 - 18:30 Skúšobné kolo
  SOBOTA 19.10.2013
  9:30 - 9:45 Súťažné pokyny (Videoprenos z Prahy.)
  10:00 - 15:00 Súťaž
  15:00 - 17:00 Prezentácia riešení, vyhlásenie výsledkov (Videoprenos z Prahy.)