Aplicaţii

– Implementarea alocării dinamice
– Tabele de dispersie
– Aplicaţie: tabele de simboluri

5 ianuarie 2004

Implementarea alocării dinamice

Funcțiile `malloc/calloc/realloc` și `free` gestionează memoria pentru cererile făcute de programul utilizator la rulare, pornind de la totalul de memorie pus la dispoziție de sistemul de operare.

Probleme de rezolvat:
- găsirea unui bloc de memorie de dimensiune potrivită (`malloc`)
- returnarea unui bloc în mulțimea celor disponibile (`free`)
- *fragmentarea* cât mai redusă în urma cererilor repetate
- *compactarea* în blocuri mai mari a fragmentelor adiacente eliberate
- structuri de date și algoritmi pentru implementare eficientă

Gestionarea și structura blocurilor de memorie

– inițial, un singur bloc cu întreaga memoria disponibilă pt. alocare
(eventual poate fi crescută prin apeluri la sistemul de operare)
– din acest bloc se separă cantitățile alocate la cerere
– ulterior, acestea pot fi eliberate și returnate
⇒ fragmentarea memoriei; trebuie o *listă* de blocuri disponibile

Informația necesară pentru gestionare:
– fiecare bloc conține un *antet*, pe lângă porțiunea utilă, cu:
lungimea blocului, și un fanion de *utilizare* (bit: alocat/liber)
– în blocurile libere (în plus): un *pointer* la următorul liber din listă
(eventual doi pointeri, pentru listă dublu înlănțuită)
⇒ e necesară o lungime minimă a blocurilor pentru a cuprinde antetul
– informația din antet poate fi codificată pentru a ocupa spațiu minim

Sisteme cu blocuri de dimensiuni fixe. Sisteme *buddy*

– dacă permitem alocarea blocurilor de orice dimensiuni, structurile de date și algoritmii devin mai complicați și ineficienți
– soluția: se selectează un șir s_1, s_2, ..., s_n de dimensiuni permise
orice solicitare e rotunjită în sus la cea mai mică lungime cuprinzătoare
$\Rightarrow$ e suficient să se țină minte o listă de blocuri libere pentru fiecare dimensiune permisă s_i (aceasta include informația de gestiune!)

Problemă: dacă nu există un bloc disponibil de dimensiune s_k, trebuie fragmentat unul mai mare. Pentru a nu crea blocuri de alte dimensiuni:
sistemul buddy [Knuth, 1973]: $s_{i+1} = s_i + s_{i-k}$ (de ordinul k)
– pentru $k = 0$: sistemul exponențial: 1, 2, 4, ... (puterile lui 2)
– pentru $k = 1$: sistemul Fibonacci: 1, 2, 3, 5, 8, ...
În practică, se pornește de la o dimensiune minimă a blocurilor (multipli de cuvânt de memorie, ex. 4, 8, 16).

Exemplu de structură de date

Conceptual:

```
struct block {
  unsigned char used;
  size_t size;
  struct block *prev;
  struct block *next;
}  b;
/* ocupă 16 octeţi */
```

Optimizat prin codificare pe biţi:

```
struct block {
  unsigned used: 1;
  unsigned szhi: 2;
  unsigned prev: 29;
  unsigned szlo: 3;
  unsigned next: 29;
} b;  /* începe pe 8 octeţi */
```

Valoarea `size = b.szhi << 3 + b.szlo` reprezintă un indice în tabela cu dimensiunile permise $\Rightarrow$ pe 5 biţi se pot codifica 32 de dimensiuni
E natural ca blocurile să fie aliniate la multipli de 8 octeţi
$\Rightarrow$ pointerii (pe 32 de biţi) se obţin cu: `(struct block *)(b.prev << 3)`
Transformarea întregilor în pointeri e frecventă în rutine de nivel scăzut dar nu e portabilă şi nu se recomandă în aplicaţii obişnuite

Compactarea fragmentelor eliberate

Blocurile libere se memorează în câte o listă pentru fiecare dimensiune:
`struct block *free[NUMSIZE]; /* tablou după nr. de dimensiuni */`
– când un bloc e eliberat, testăm dacă poate fi recombinat cu blocul din care a fost desprins inițial (dacă și fragmentul adiacent e liber)

– în sisteme buddy exponențiale, un bloc de dimensiune 2^k se află la deplasamentul $d = n \cdot 2^k$ în memoria disponibilă. Perechea sa (tot cu dimensiunea 2^k) are adresa: $d - 2^k$, pt. n impar; $d + 2^k$ pt. n par

– pt. sisteme buddy de ordin $k > 0$, găsirea perechii e mai complicată: la despărțirea întregului spațiu s_n cf. relației $s_{i+1} = s_i + s_{i-k}$, în fiecare bloc se contorizează de câte ori consecutiv e în stânga ultimei separări: dacă $B_{i+1} = B_i + B_{i-k}$, atunci $B_i.cnt = B_{i+1}.cnt + 1$ și $B_{i-k}.cnt = 0$
– la eliberare, dacă $B_i.cnt = 0$, perechea e blocul B_{i+k} din stânga;
– dacă $B_i.cnt \neq 0$, perechea de testat e blocul B_{i-k} din dreapta lui.

Analiza timpului de rulare al algoritmilor

Metode teoretice

– relații matematice pentru cazul cel mai defavorabil (uneori și mediu, mai rar: cel mai favorabil)
– de regulă, nu timpul fizic, ci numărul de operații de un anumit tip (ex. pt. sortare: numărul de comparații, numărul de interschimbări)

Evaluare practică

– prin rularea programelor pe diverse seturi de date (aleatoare sau cu anumite proprietăți), și măsurarea timpilor de execuție

Discutăm:
– funcții pentru generarea de numere (pseudo)aleatoare
– funcții legate de măsurarea timpului

Funcții pentru manipularea timpului (`time.h`)

Tipuri definite pentru reprezentarea timpului: `clock_t` și `time_t`
(sunt de fapt tipuri aritmetice, de ex. `unsigned` sau `unsigned long`)

`clock_t clock(void);`
returnează timpul scurs de la lansarea programului, în unități de ceas
date de constanta `CLOCKS_PER_SEC` (în standardul POSIX, 1 milion)
e o aproximație dependentă de granularitatea ceasului de timp real
poate interveni depășire (pe sistem de 32 de biți, după cca 72 min.)

`time_t time(time_t *timer);`
returnează o valoare aritmetică reprezentând data/ora curentă
(în UNIX, numărul de secunde trecute de la 1 ian. 1970 UTC)
dacă argumentul pointer e nenul, valoarea e stocată și la acea adresă

`double difftime(time_t time1, time_t time0);`
returnează diferența exprimată în secunde, ca `double`
Pentru reprezentări descompuse (zi/oră/min./etc.): tipul `struct tm`
(vezi detalii în standard)

Funcții pentru numere pseudoaleatoare (`stdlib.h`)

Numerele generate sunt *pseudo*aleatoare (de fapt deterministe, bazate pe un algoritm, dar cu distribuție cât mai uniformă)
(numere cu adevărat aleatoare ar trebui să fie bazate pe fenomene fizice, ex. aruncarea unei monede sau descompunerea unor particule)

`int rand(void);`
returnează un număr pseudoaleator între `0` și `RAND_MAX` (min. 32767)
pt. un număr aleator între `1` și `N` putem folosi `1 + rand() % N`

`void srand(unsigned int seed);`
reinițializează generatorul de numere pseudoaleatoare cu valoarea dată
următorul număr va fi generat de `rand()` pornind de la această valoare
fără apelarea ei, două rulări generează același șir de valori cu `rand()`
se poate folosi de ex. cu `srand(unsigned)time(NULL));`

Tabele de dispersie (hashtables)

– pentru *regăsirea eficientă* a unui obiect când acesta nu are o valoare numerică de identificare utilizabilă direct ca indice într-un tablou
– ideea: găsirea unei funcţii `h` cu o *valoare numerică unică* pentru fiecare obiect considerat, într-un domeniu restrâns (utilizabil ca indice)
$\Rightarrow$ fiecare obiect `x` e memorat într-un tablou la indicele `h(x)`
– matematic: o funcţie parţială $h : D \rightarrow V$, unde D e domeniul tuturor obiectelor posibile, iar domeniul de valori V e $0, 1, \ldots, N-1$.
– ex. la compilare: D e mulţimea tuturor identificatorilor
– practic, $|D| >> |V|$, deci h nu poate fi injectivă pe D, dar avem nevoie de valori distincte doar pt. submulţimea $D_u \subseteq D$ a obiectelor efectiv utilizate (ex. identificatorii dintr-un anumit program C)
– se caută funcţii de dispersie (hash functions) cu proprietăţi cât mai bune (distribuţie uniformă $\Rightarrow$ probabilitate mică de valori egale)

Exemple de funcţii de dispersie

– funcţii simple, calculate rapid, folosind (aproape) toate caracterele
– adesea cu deplasări pe biţi (în loc de înmulţiri, şi pt. uniformizare)
Exemple pentru şiruri (`char *s;` (se parcurge secvenţial):

```
for (h=len; len--;) h = ((h<<7) ^ (h<<27)) ^ *s++;     /* Knuth */
for (h=5381; c=*s++; ) h += (h << 5) + c;          /* Bernstein */
for (h=0; c=*s++; ) h = (h<<6) + (h<<16) - h + c;       /* SDBM */
```

Pentru alte obiecte: calcule cu întregii obţinuţi grupând octeţii câte 4

În toate cazurile: valoarea finală luată modulo dimensiunea tabloului

Şi funcţiile bune au *coliziuni* (valori egale pt. obiecte diferite)
$\Rightarrow$ trebuie rezolvate (dezambiguate) pentru a permite regăsirea corectă

Tabele de dispersie deschise și închise

Tabele de dispersie închise (closed hashing)

– dacă la indicele `idx=h(x)` se găsește alt obiect `y`, se caută succesiv după o anumită regulă: *secvențial* (`idx++`), *liniar* (`idx+=i`), cu a doua funcție (`idx+=h2(x)`), până se găsește obiectul sau o intrare vidă

– nu pot conține mai multe obiecte decât dimensiunea tabloului

⇒ la depășire, obiectele trebuie redistribuite într-un tablou mai mare

– la ștergere, intrarea în tablou trebuie marcată "șters", nu "vid", pentru a permite căutarea corectă (până la găsire sau "vid")

Tabele de dispersie deschise (open hashing)

– o intrare în tablou: *listă* de obiecte cu aceeași valoare pentru `h`

⇒ hashing + căutare liniară în listă (scurtă pentru funcții bune)

– necesită alocare dinamică pentru elementele listei (v. exemplu)

– și aici, tablou cu dimensiune cel puțin comparabilă cu nr. de obiecte