

## Aplicații. Implementarea alocării dinamice

7 decembrie 2005

Programarea calculatoarelor 2. Curs 10

Marius Minea

### 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

Programarea calculatoarelor 2. Curs 10

Marius Minea

### Gestionarea și structura blocurilor de memorie

- inițial, un singur bloc cu întreaga memorie 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
- pentru un bloc alocat, e necesar doar bitul de alocare + dimensiunea

Programarea calculatoarelor 2. Curs 10

Marius Minea

### Sisteme cu blocuri de dimensiuni fixe. Sisteme *buddy*

– dacă permitem alocarea blocurilor de orice dimensiuni, structurile de date și algoritmi devin mai complicați și ineficienți

– soluția: se selectează un șir  $s_1, s_2, \dots, s_n$  de dimensiuni permise orice solicitare e rotunjită în sus la cea mai mică lungime cuprinzătoare ⇒ 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).

Programarea calculatoarelor 2. Curs 10

Marius Minea

### Exemplu de structură de date

Conceptual:

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

Valoarea `size = b.szhi << 3 + b.szlo` reprezintă un indice în tabela cu dimensiunile permise ⇒ pe 5 biți se pot codifica 32 de dimensiuni

E natural ca blocurile să fie aliniate la multipli de 8 octeți ⇒ pointerii (pe 32 de biți) se obțin cu: `(struct block *) (b.prev << 3)`

Transformarea întregilor în pointeri e frecventă în rutine de nivel scăzut dar nu e portabilă și nu se recomandă în aplicații obișnuite

Programarea calculatoarelor 2. Curs 10

Marius Minea

Optimizat prin codificare pe biți:

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

### 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.

Programarea calculatoarelor 2. Curs 10

Marius Minea