Structura cursului

- prima jumătate a semestrului
  - curs - elemente de programare C/C++
  - laborator - limbaj de asamblare
  - evaluare - test practic ASM (săptămâna a 8-a)
- a doua jumătate a semestrului
  - curs introducere în programarea Windows
  - laborator - proiect folosind biblioteca MFC
  - evaluare - notarea proiectului, pe etape

Cuprins - prima jumătate

I. Alocarea memoriei
II. Structuri
III. Conversii de tip
IV. Referinţe şi pointeri
V. Clase
VI. Metode virtuale

I. Alocarea memoriei

Tipuri de alocare a memoriei

- statică
  - dimensiunea zonelor de memorie - stabilită la momentul compilării
- dinamică
  - alocare/eliberare explicită
    - gestiunea este sarcina programatorului
    - risc - apariţia erorilor de programare
  - *garbage collection* - eliberare automată

Le ce nivele se pune problema

- nivelul programului
  - gestionarea memoriei proprii
- nivelul sistemului de operare
  - gestionarea memoriei întregului sistem
  - caz particular - încărcarea programelor în memorie
    - la lansare
    - transferuri din memoria virtuală (*swap*)

Alocare statică şi dinamică (1)

- depinde de punctul din care privim
- alocare statică la nivelul programului
  - spaţiul ocupat de variabile este cunoscut de la momentul compilării
  - pentru sistemul de operare, încărcarea programului în memorie reprezintă o problemă de alocare dinamică

Alocare statică şi dinamică (2)

- alocare dinamică la nivelul programului
  - folosim o funcţie de bibliotecă (ex. *malloc*)
  - aceasta gestionează o zonă de memorie dedicată
    - deci alocată static
  - ce se întâmplă când zona este plină?
    - următoarele cereri de alocare vor eşua
    - sau programul poate cere o zonă suplimentară de la sistemul de operare

Criterii de performanţă

1. timpul maxim necesar pentru o operaţie
   - unele cereri de alocare trebuie deservite foarte rapid
2. timpul mediu necesar pentru o operaţie
   - performanţa globală a sistemului
   - situaţiile cele mai defavorabile - cât mai rar
3. gradul de fragmentare
   - internă sau externă

Criterii de performanţă (cont.)

4. dimensiunea spaţiului de memorie ocupat de structurile de date ale alocatorului
   - reduce cantitatea de memorie disponibilă pentru programe
5. simplitatea în utilizare

I.1. Alocatorul cu hartă de resurse

Structură

- harta de resurse - tablou cu informaţii despre zonele de memorie
  - adresa de început
  - dimensiunea
  - starea (liber/ocupat)

Alocarea unei zone noi

- se parcurge harta de resurse
- se alege zona liberă cea mai potrivită ca dimensiune
- algoritmi de selecţie
  - First Fit
  - Best Fit
  - Worst Fit

Avantaje

- simplitate în concepţie
- zonele libere adiacente pot fi concatenate

Dezavantaje

- spaţiu mare ocupat de harta de resurse
- timp de căutare relativ mare
  - trebuie parcurse toate zonele, inclusiv cele ocupate
- structură de tip tablou - gestiune dificilă
  - inserări (alocări)
  - ştergeri (eliberări)

Îmbunătăţiri

a) Fiecare zonă conţine şi propria dimensiune
- mai puţine informaţii în harta de resurse
- câştig - la zonele libere
- nu consumă memorie suplimentară
- dimensiunea unei zone libere - cel puţin 4 octeţi
  - pentru a putea memora dimensiunea

Îmbunătăţiri

b) Fiecare zonă liberă conţine

- propria dimensiune
- adresa de început a următoarei zone libere

– se formează o listă înlănţuită

– harta de resurse - redusă

- informaţii despre zonele ocupate
- adresa de început a primei zone libere

– dimensiunea unei zone libere - cel puţin 8 octeţi

I.2. Alocatorul cu puteri ale lui 2

Structură

- zonele de memorie pot avea numai anumite dimensiuni - puteri ale lui 2
- câte o listă cu zonele libere pentru fiecare dimensiune posibilă
- zonele ocupate conţin şi propria dimensiune
- listă vidă - se sparge o zonă din lista superioară

Avantaje

- timp de răspuns redus
  - nu se face o căutare în harta de resurse
  - se calculează puterea lui 2 care incude dimensiunea dorită
  - se preia o zonă liberă din lista zonelor cu dimensiunea respectivă
    - prima găsită

Dezavantaje

- fragmentare internă
- structuri de date de 2^n octeţi - se alocă 2^{n+1}
  - se reţine şi propria dimensiune
  - situaţia apare des în nucleul sistemului de operare
- nu se pot concatena zonele libere adiacente

Alocatorul McKusick-Karels

- variantă a alocatorului cu puteri ale lui 2
- fiecare pagină de memorie conţine numai zone de aceeaşi dimensiune
- alocatorul reţine dimensiunea zonelor de memorie pentru fiecare pagină
  - zonele ocupate nu mai reţin propria dimensiune
  - structuri de 2^n octeţi - se alocă exact 2^n

Alocatorul *buddy* (1)

- variantă a alocatorului cu puteri ale lui 2
- poate concatena zonele libere adiacente
- dimensiunea unei zone: $D \cdot 2^n$

  D - dimensiune minimă (elementară)
- hartă de biţi - zone de dimensiune D

  1 - ocupat

  0 - liber

Alocatorul *buddy* (2)

- adresa de început a unei zone de memorie alocate - multiplu al dimensiunii sale
  - perechi de zone adiacente de dimensiuni egale
  - facilitează reunirea zonelor libere vecine
- timp maxim mare pentru concatenare
- interfaţă de programare complicată
  - eliberarea unei zone - trebuie precizată mărimea

Alocatorul *lazy buddy* (1)

- reduce timpul consumat pentru concatenarea zonelor adiacente
- numărul zonelor alocate dinamic
  - relativ constant în marea majoritate a timpului
    - concatenarea nu este necesară
  - foarte mare în anumite momente
    - multe cereri care trebuie servite într-un timp scurt

Alocatorul *lazy buddy* (2)

- concatenarea zonelor libere adiacente
  - nu se face la fiecare operaţie de eliberare
  - nu poate fi amânată până când devine strict necesară
    - timpul maxim de răspuns - prea mare
  - se face în funcţie de o anumită măsură a gradului de fragmentare

Alocatorul *lazy buddy* (3)

Starea unei zone

- liberă din punct de vedere local - imediat după eliberare
- liberă din punct de vedere global - după concatenare
- ocupată

Alocatorul *lazy buddy* (4)

- pentru fiecare dimensiune posibilă a zonelor de memorie

$$N = A + L + G$$

N - numărul total de zone de dimensiunea respectivă
A - numărul de zone ocupate (alocate)
L - numărul de zone libere din punct de vedere local
G - numărul de zone libere din punct de vedere global

Alocatorul *lazy buddy* (5)

$$S = N - 2L - G$$

- $S > 1 \rightarrow$ concatenarea nu este necesară
- $S = 1 \rightarrow$ concatenarea este necesară
- $S = 0 \rightarrow$ concatenarea este necesară şi trebuie realizată de urgenţă

Alocatorul *lazy buddy* (6)

Avantaj

- timp de răspuns mai redus în cazul unui număr mare de cereri venite într-un timp scurt

Dezavantaj

- timp mai mare consumat pentru operaţia de eliberare a zonelor de memorie

I.3. Alocatorul *slab*

Structură (1)

- pentru zone mici - puteri ale lui 2
- structuri de date mari - caracter tipizat
  - cunoaşte tipurile de obiecte cu care lucrează
  - gestiune mai eficientă
- lucrează cu blocuri continue de dimensiune fixă (*slabs*)

Structură (2)

- fiecare tip de obiecte - un cache cu blocuri
  - structură de tip listă înlănţuită
- zone refolosite pentru obiecte de acelaşi tip
  - nu mai trebuie reiniţializate
  - obiectele din nucleu - iniţializarea consumă mai mult timp decât alocarea

Operaţii asigurate de alocator (1)

- iniţializarea obiectelor
  - numai la preluarea de către cache-ul respectiv a unui bloc nou
  - se aplică asupra tuturor obiectelor din bloc
- alocarea unui obiect
- utilizarea obiectului

Operaţii asigurate de alocator (2)

- eliberarea spaţiului ocupat de un obiect
  - nu este necesară reiniţializarea la o nouă alocare
- distrugerea obiectelor
  - similar cu iniţializarea

Implementare (1)

- fiecare cache - liste cu blocurile componente
  - blocurile complet libere
  - blocurile complet ocupate
  - blocurile parţial ocupate
- blocurile libere pot trece de la un cache la altul

Implementare (2)

- obiectele nu încap exact într-un bloc
- rămâne spaţiu nefolosit în cadrul blocului
  - obiectele pot fi plasate în bloc de la un anumit deplasament
  - deplasamente diferite între blocuri - se reduce numărul de suprapuneri în cache-ul procesorului

II. Structuri

Tipul *struct*

- colecţie de variabile (membri)
  - accesate printr-un nume comun
- ar putea fi şi variabile independente
- atunci de ce le grupăm?
  - logica aplicaţiei impune tratarea lor comună
  - mai uşor de gestionat variabilele respective

Reprezentare internă

- membrii sunt plasaţi în memorie unul după altul
- la adrese consecutive
  - adresa de început a primului membru = adresa de început a structurii
  - adresa de început a fiecărui membru = suma dimensiunilor membrilor precedenţi
- sau nu?

Exemplu

```
struct S {
    char a,b;
    int c;
    double d;
    char e;
    short f;
};
```

Exemplu (cont.)

```
S s;
printf("Dimensiune structura: %d\n",sizeof(s));
printf("start:\t%p\n",&s);
printf("a:\t%p\t%2d\n",&s.a,(int)&s.a-(int)&s);
printf("b:\t%p\t%2d\n",&s.b,(int)&s.b-(int)&s);
printf("c:\t%p\t%2d\n",&s.c,(int)&s.c-(int)&s);
printf("d:\t%p\t%2d\n",&s.d,(int)&s.d-(int)&s);
printf("e:\t%p\t%2d\n",&s.e,(int)&s.e-(int)&s);
printf("f:\t%p\t%2d\n",&s.f,(int)&s.f-(int)&s);
```

Rezultat - exemplu de afişare

```
Dimensiune structura: 24
start:   0012FF4C
a:       0012FF4C      0
b:       0012FF4D      1
c:       0012FF50      4
d:       0012FF54      8
e:       0012FF5C     16
f:       0012FF5E     18
```

Interpretare

- variabila c începe la deplasament 4
  - în loc de 2
- variabila f începe la deplasament 18
  - în loc de 17
- dimensiunea structurii - 24
  - ar fi trebuit să fie 17
  - sau 20, dacă ţinem cont de cele de mai sus

Alinierea adreselor (1)

- deplasamentul fiecărui membru este multiplu de dimensiunea sa
- motivaţie
  - transferurile cu memoria sunt mai simple
  - ocuparea memoriei este mai uşor de gestionat
- efect - între membrii structurii pot apărea spaţii nefolosite

Alinierea adreselor (2)

- dezavantaj
  - consum de memorie mărit
  - în principiu, ne putem permite
- alinierea poate fi dezactivată

`Project` → `Properties` → `C/C++` → `Code Generation` → `Struct Member Alignment:` `1 Byte` (în loc de `Default`)

Dimensiunea structurii (1)

- de multe ori este mai mare decât suma dimensiunilor membrilor
  - consecinţă a alinierii adreselor
  - pot fi spaţii neutilizate între unii membri
  - poate fi un spaţiu neutilizat la finalul structurii

Dimensiunea structurii (2)

- exemplu

```
S t[2];
printf("\nt[0]: %p\nt[1]: %p\n",&t[0],&t[1]);
```

- rezultat afişare

```
t[0]:   0012FF14
t[1]:   0012FF2C
```

De ce ne interesează?

- calculul adreselor elementelor din tablourile de structuri
  - vezi anterior
- sunt şi alte situaţii în care informaţia este importantă
  - exemplu - citirea unei imagini dintr-un fişier în format BMP

Structură header BMP

```
typedef struct {
  unsigned char magic[2];
  unsigned long filesz;
  unsigned short creator1;
  unsigned short creator2;
  unsigned long bmp_offset;
} bmpfile_header_t;
```

Citire header

```
bmpfile_header_t bh;
FILE *fBMP=fopen("f.bmp","r");
// citire header fisier
fread(&bh,sizeof(bh),1,fBMP);
// ...
fclose(fBMP);
```

Probleme

sizeof(bh)=16

- în fişier nu există aliniere
- trebuie citiţi 14 octeţi
- deplasamentele câmpurilor sunt la rândul lor incorecte
- e mai sigur să citim membrii unul câte unul

Funcţii cu parametri structuri

- limbajul C nu permite transmiterea structurilor ca parametri
  - doar pointeri la structuri
  - limbajul C++ o permite, dar cazul său îl vom discuta mai târziu
- accesarea membrilor
  - pe stivă avem doar un pointer, nu toată structura
  - atenţie la aliniere

Returnarea valorilor din funcţii

- unde trebuie pus rezultatul pentru a fi returnat?
- în mod uzual - registrul eax
  - sau al/ax pentru dimensiuni mai mici
- dar dacă vrem să returnăm un tip mai mare de 4 octeţi?

Unde apare problema

- tipurile elementare nu depăşesc 4 octeţi
- tipul double este returnat prin unitatea de virgulă mobilă
  - nu ne interesează aici
- structuri
  - nu pot fi returnate în limbajul C
    - doar pointeri la structuri
  - dar în C++ este posibil

Soluţii (1)

- depinde de compilator
  - aici discutăm despre Visual Studio
- structură cu dimensiune până în 8 octeţi
  - regiştrii edx (partea mai semnificativă) şi eax
- ce înseamnă parte mai semnificativă?
  - Intel - adresare *little-endian*
  - partea mai semnificativă dintr-un operand - cea aflată la o adresă mai mare în memorie

Soluţii (2)

- structură cu dimensiune mai mare de 8 octeţi
  - se mai transmite un parametru ascuns (nu apare în lista de parametri)
  - adresa structurii care trebuie completată cu rezultatul
  - acest parametru este plasat la adresa [ebp+8]
  - deci primul parametru "real" se găseşte acum la [ebp+12]