Laboratorio di Algoritmi e Strutture Dati

La gestione della memoria dinamica Heap

Prof. Luigi Lamberti
2005

Cenni sui Processi

Un **Programma** è un insieme di Istruzioni memorizzato in un file con le costanti del problema.

Quando il **Sistema Operativo** porta in esecuzione un programma:

1. Legge l'intestazione del file con le richieste del programma.
2. Alloca in memoria spazio sufficiente per collocare:
   a. il codice eseguibile del programma
   b. le costanti, le variabili globali e le variabili dichiarate static;
   c. uno **Stack** per le variabili locali, per il passaggio di parametri alle sub, e per gli indirizzi di ritorno dalle stesse;
   d. una zona di memoria "dinamica" Heap.
3. Riserva tutte le altre eventuali risorse.
4. Inizia l'esecuzione.

Un Programma in stato di esecuzione è un **Task** o **Processo**.

Al termine del Processo tutte le risorse occupate vengono rilasciate.

Allineamento dei dati

Secondo il Microprocessore e il S.O. utilizzato, l'accesso ottimale alla memoria avviene a gruppi di 2, 4, 8, 16 byte.

Le variabili devono essere dichiarate in modo da allinearsi agli indirizzi multipli 2, 4, 8, 16 byte.

Ad esempio, se l'accesso ottimale è a 4 byte, la dichiarazione

```
char a, b, s[31];
```

andrebbe sostituita con

```
char a, b, a1, b1, s[32];
```

Alcuni compilatori possono ottimizzare per spazio o per velocità: nel secondo caso allineano automaticamente i dati inserendo spazi vuoti.

Memoria Heap

Un **Heap** è un lotto di memoria assegnato ad un processo, per i dati, la cui esistenza o dimensione non è conosciuta a priori.

Quando sono richiesti blocchi di memoria, un processo

1. richiama una funzione per allocare memoria nell'heap
2. utilizza lo spazio ottenuto
3. libera la memoria con un'altra funzione.

Se necessario, il processo può chiedere heap aggiuntivi al S.O.

Il S.O. gestisce la memoria in **paragrafi** minimi (16, 32 o 64 byte); ogni richiesta viene arrotondata al multiplo successivo del paragrafo, oltre un paragrafo per informazioni accessorie.

L'uso continuo di allocazione e rilascio di memoria causa un **frazionamento** dello spazio disponibile con conseguente peggioramento delle prestazioni.

Valutare costi e benefici della gestione Heap, sostituendola, se è il caso, con un'assegnazione iniziale di tutto lo spazio necessario, o con la creazione dinamica di spazio sullo Stack.

Memoria Heap in C

```
#include <malloc.h>

void *malloc ( size_t Size );

void *calloc
          ( size_t Numero, size_t Size );

void *realloc
   (void *BlockAddress, size_t NewSize );

void free ( void *BlockAddress );
```

void *malloc (size_t Size);

Alloca **Size** byte di memoria, e restituisce :

- un puntatore **void** sullo spazio allocato correttamente;
- **NULL** se la memoria disponibile è insufficiente.

Per assegnare lo spazio a tipi diversi da void, usare un operatore **cast** sul valore restituito.

Occorre sempre controllare il valore restituito da malloc, anche se la memoria richiesta è piccola.

Il blocco di memoria occupato può essere più grande di **Size** (allineamento dati e informazioni accessorie).

Esempio

```
#include <stdio.h>
#include <malloc.h>

void main( void )
{ char *Buffer;

  Buffer = (char *) malloc ( 200 );
  if ( Buffer == NULL )
  { printf( "Memoria insufficiente\n" );
  }
  else
  { printf( "Spazio allocato\n" );
    gets  ( Buffer );
    printf( "%s\n", Buffer );
    free( Buffer );
    printf( "Memoria liberata\n" );
  }
}
```

void *calloc (size_t Numero, size_t Size);

Alloca memoria Heap per un array di **Numero** elementi, ciascuno di dimensioni **Size** byte, e restituisce:

- un puntatore **void** sullo spazio allocato correttamente;
- **NULL** se la memoria disponibile è insufficiente.

Per assegnare lo spazio a tipi diversi da void, usare un operatore **cast** sul valore restituito.

Ogni elemento è inizializzato a zero.

Controllare sempre il valore restituit da calloc, anche se la memoria richiesta è piccola.

Esempio

```
#include <stdio.h>
#include <malloc.h>

void main( void )
{ long *Buffer;

  Buffer = (long *) calloc ( 40, sizeof(long) );
  if (Buffer == NULL )
  { printf ( "Memoria insufficiente\n" );
  }
  else
  { printf ( "Disponibili 40 longint\n" );
    . . . . . .
    free ( Buffer );
  }
}
```

void *realloc (void *BlockAddress, size_t NewSize);

Ridimensiona un blocco Heap già allocato, portandolo a **NewSize** byte; restituisce un puntatore void al blocco di memoria riallocato ed eventualmente spostato.

Il contenuto del blocco non è modificato per un numero di byte pari alla minore tra vecchia e nuova dimensione.

Se **BlockAddress** è **NULL** e **NewSize** è diverso da zero, realloc è equivalente a malloc.

Se non c'è abbastanza memoria disponibile per espandere il blocco alla dimensione richiesta, realloc restituisce **NULL** e il blocco non è modificato.

Se **BlockAddress** è un puntatore valido e **NewSize** è **zero**, realloc libera il blocco e restituisce **NULL**.

void free (void *BlockAddress);

Rilascia un blocco di memoria heap, di indirizzo **BlockAddress**, precedentemente allocato con chiamate a malloc, calloc o realloc.

Se **BlockAddress** è **NULL**, il puntatore è ignorato e free esegue un immediato return.

Tentativi di liberare memoria non allocata da malloc, calloc o realloc, ovvero già liberata, può influire sulle successive allocazioni e causare **errori imprevedibili**.

La funzione free non restituisce alcun valore.

ATTENZIONE

Rilasciare lo spazio non più necessario per evitare il collasso del processo per mancanza di memoria.

Un errore comune è la perdita dell'indirizzo del blocco:

```
int x, *Pt;
Pt = (int *) malloc ( 200 ); // occupa 50 int
. . . .
Pt  = &x;   // Il puntatore perde il blocco
*Pt = 12345;
. . . .
free ( Pt );  // errore nel gestore dell'Heap
```