Memoria Dinamica

Alessandro Pellegrini
a.pellegrini@ing.uniroma2.it

L’Heap

- Non è sempre noto a tempo di compilazione di quanta memoria avrà bisogno un programma:
  - Interazioni con l’utente
  - Interazione con altri sistemi esterni (applicazioni di rete, …)
  - Durata variabile dell’esecuzione dell’applicazione
  - File in input

- L’heap (mucchio) è un segmento del programma eseguibile dal quale è possibile ottenere delle aree di memoria a richiesta

Allocazione dinamica della memoria

- Esistono due funzioni della libreria standard, dichiarate in `stdlib.h`, che consentono di ottenere e rilasciare buffer di memoria in maniera dinamica dall'heap
  - `void *malloc(size_t size)`: alloca una quantità di memoria di dimensione `size` byte e restituisce un puntatore alla memoria allocata, o `NULL` in caso di errore. La memoria non è inizializzata. Se `size` è zero, `malloc()` restituisce `NULL` o un puntatore che può essere successivamente passato a `free()`.
  - `void free(void *ptr)`: libera l'area di memoria puntata da `ptr`, che deve essere stato precedentemente restituito da `malloc()` (o varianti).

Esempio

```
int *ptr = malloc(10 * sizeof(int));
if (!ptr) {
    /* Manage the error here */
} else {
    /* Allocation successful. Do whatever you want! */
    free(ptr);  /* When memory is not needed anymore, you free it. */
    ptr = NULL; /* Set the pointer to NULL, to avoid a “dangling” pointer */
}
```

- Un’alternativa all’allocazione iniziale:

```
int *ptr = malloc(10 * sizeof(*ptr));
```

Array flessibili

- Quando si definisce una struct, l'ultimo membro può essere un *flexible array* (a partire dal C99)
- Si tratta di un vettore in cui non viene specificata la dimensione, che permette di realizzare strutture a dimensione variabile:

```
struct list {
        struct list *next;
        size_t size;
        unsigned char payload[];
}
```

- Qual è la dimensione di payload? Come si specifica?

Esempio: lista di stringhe

directory: string-list

```
struct node_t {
        struct node_t *next;
        unsigned char payload[];
};

struct node_t *add_after(struct node_t *prev, char *str)
{
        size_t len = strlen(str) + 1;
        struct node_t *node = malloc(sizeof(*node) + len);
        memcpy(node->payload, str, len);
        node->next = prev->next;
        prev->next = node;

        return node;
}
```

Allocazione di vettori e riallocazione

directory: dynamic-vector

- Esistono due funzioni aggiuntive per allocare memoria dinamicamente:
  - `void *calloc(size_t nmemb, size_t size)`: restituisce un'area di memoria tale da contenere un vettore di nmemb elementi, ciascuno di dimensione size. La memoria è inizializzata a zero. Viene verificato se nmemb * size provoca overflow, caso in cui viene generato un errore.
  - `void *realloc(void *ptr, size_t size)`: viene "modificata" la quantità di memoria puntata da ptr, effettuando una nuova allocazione e spostando size byte da ptr al nuovo buffer allocato. Il buffer puntato da ptr viene liberato con free(ptr). Il nuovo buffer viene restituito dalla funzione.

Errori comuni: mancato controllo di allocazione

- `malloc()` non garantisce il successo dell'operazione:
  - se non c'è memoria disponibile
  - se il programma ha superato il limite di memoria che può utilizzare

- In questo caso, viene restituito `NULL`.

- Molto spesso i programmatori non verificano il valore restituito da `malloc()`:
  - È possibile accedere a un puntatore impostato a `NULL`
  - SEGFAULT!

Errori comuni: memory leak

- Un memory leak è la perdita di controllo da parte del sistema su una porzione di memoria
- La libreria di sistema non sa se stiamo ancora usando un buffer: quando non lo usiamo più, dobbiamo rilasciarlo con `free()`
- Un memory leak si genera quando ad una chiamata a `malloc()` non corrisponde una chiamata a `free()`

- Gli effetti possono essere gravi:
  - Possiamo finire la memoria
  - Le applicazioni possono andare in crash
  - Il sistema può diventare instabile

Un esempio di memory leak

file: leak.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

bool leak(void)
{
        char *s = malloc(4096);

        if(s == NULL) {
                return false;
        } else {
                s[0] = 'A';
                return true;
        }
}

int main(void)
{
        while(leak());
        return 0;
}
```

Errori comuni: use after free

- Un puntatore passato a `free()` è una variabile che conserva l'indirizzo della memoria appena rilasciata
- Utilizzare questo puntatore dopo la chiamata a `free()` genera un *comportamento non definito*
- La memoria rilasciata potrebbe essere utilizzata dal sistema per altri scopi (particolarmente vero nel caso di programmazione concorrente)

```
int *ptr = malloc(sizeof(int));
free(ptr);
*ptr = 0; /* undefined behavior */
printf("%p", ptr); /* undefined behavior */
```

Errori comuni: use after free

- Ci sono alcune tecniche per ridurre la probabilità di accedere a *dangling pointer* (puntatori pendenti)

- Resettare esplicitamente il puntatore:

```
free(ptr);
ptr = NULL;
```

- Utilizzare correttamente gli scope delle variabili (laddove possibile):

```
int *ptr = NULL;
 {
   int *ptr = malloc(sizeof(int));
   free(ptr);
 }
*i = 200; /* the application will crash! */
```

Errori comuni: freeing wrong pointers

- `free()` accetta come parametro un puntatore a un buffer *allocato dinamicamente*
- Non ha però modo di verificare che il buffer puntato sia stato *effettivamente* restituito da una invocazione a `malloc()`

- Codice di questo tipo porta la libreria di allocazione di memoria dinamica in uno *stato non coerente*:

```
char *msg = "Una stringa globale";
int tbl[100];
free(msg); /* undefined behavior */
free(tbl);  /* undefined behavior */
```

Errori comuni: double free corruption

- Una *corruzione da doppia liberazione* si verifica quando si chiede alla libreria di liberare più volte lo stesso buffer di memoria

```
int *ptr = malloc(sizeof(int));
free(ptr);
free(ptr);
```

- Si tratta di un ulteriore *undefined behavior*
- Nel caso peggiore, l'applicazione potrebbe andare in crash molto tempo dopo, quando si chiede nuova memoria tramite `malloc()`

Ingrandiamo lo stack software: “valgrind”

- Valgrind è uno strumento per il debug di problemi di memoria, la ricerca dei memory leak ed il profiling del software.
- Si basa su un’architettura virtualizzata, all’interno della quale emula l’esecuzione di ciascuna istruzione assembly del programma
- Sfrutta i simboli di debug per individuare le istruzioni che:
  - leggono/scrivono fuori da un buffer allocato dinamicamente
  - effettuano doppie liberazioni di memoria
  - utilizzano variabili non inizializzate in delle istruzioni di confronto
- Si utilizza da riga di comando:

```
$ valgrind ./programma [parametri]
```

Matrici dinamiche

- Si può sfruttare il funzionamento dell'operatore [] applicato ai puntatori e l'utilizzo di memoria dinamica per creare "matrici dinamiche"
- Una matrice dinamica bidimensionale è effettivamente un vettore di puntatori a vettori

```
int **allocate_matrix(int rows, int cols)
{
    int i;
    int **ret = malloc(sizeof(int *) * rows);
    for(i = 0; i < rows; i++) {
        ret[i] = malloc(sizeof(int) *cols);
    }
    return ret; // Matrix is not initialized!
}
```

- Non c'è alcun vincolo sul fatto che le colonne della matrice debbano avere tutte la stessa dimensione