Funzioni per l'allocazione di memoria

Anno Accademico 2010-2011

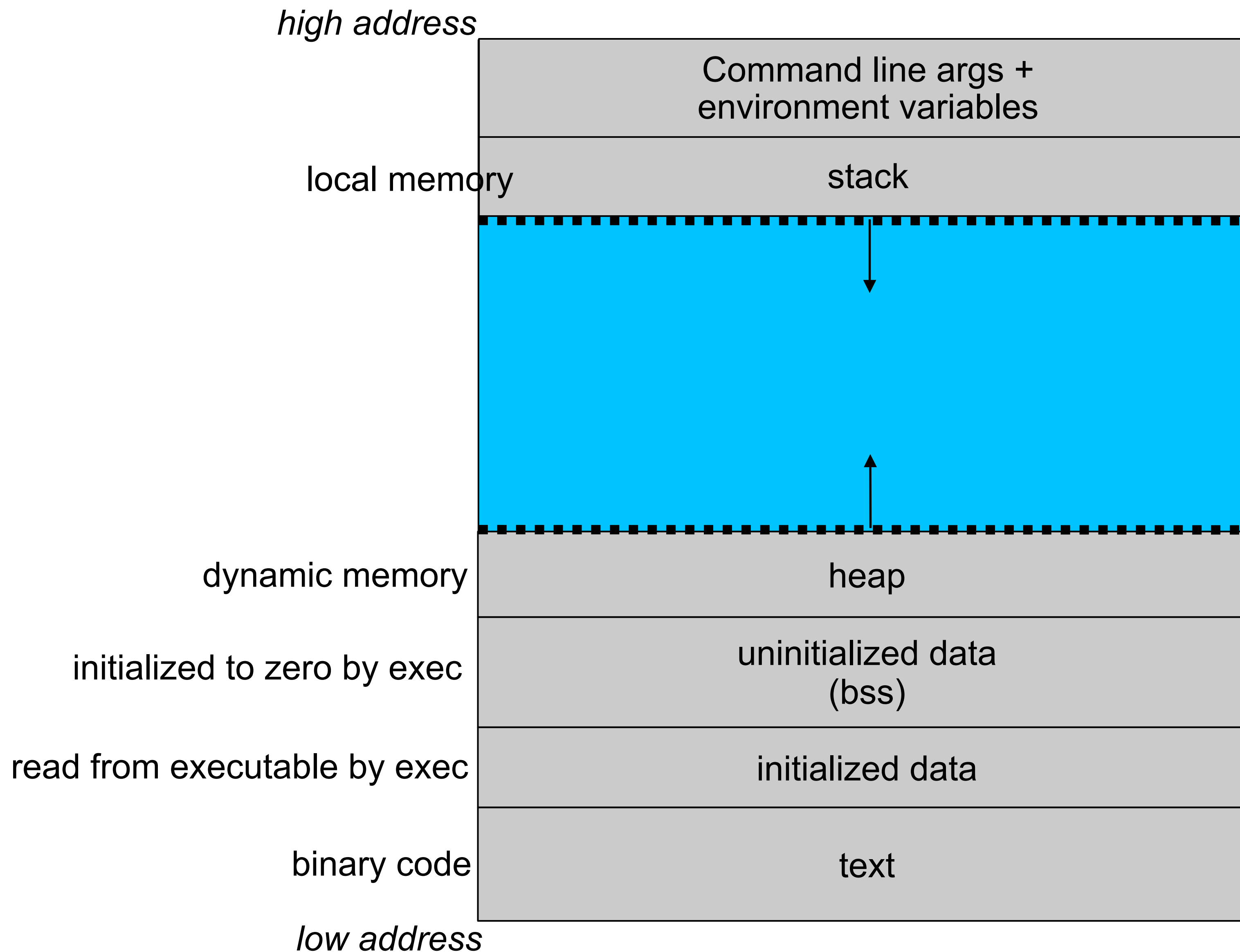
Prof. Claudio Cilli

Layout di memoria di un processo

Un programma C caricato in memoria è composto dalle seguenti parti:

- ***Text segment***: istruzioni in codice macchina eseguite dalla CPU. Generalmente è condiviso (sharable) cosicché esiste una singola copia in memoria ed è read-only per prevenire modifiche accidentali.
- ***Initialized data segment*** (o semplicemente ***data segment***): contiene variabili che sono esplicitamente inizializzate nel programma (globali e statiche inizializzate).
- ***Uninitialized data segment*** (chiamato anche ***segmento* bss** (block started by symbol)*)*: i dati in questo segmento sono inizializzati dal kernel a 0 o NULL pointer prima che il programma inizi l'esecuzione (globali e statiche non inizializzate).
- ***Stack***: contiene la variabile automatiche e le informazioni salvate ogni volta che viene effettuata una chiamata a funzione.
- ***Heap***: memoria dinamica che il programma richiede durante la sua esecuzione.

Layout di memoria di un processo



Osservazioni sulla memoria di un processo (I)

- Il formato di un eseguibile Linux è tale che soltanto le variabili inizializzate ad un valore diverso da zero occupano spazio nel file eseguibile su disco.
- Quando viene allocata memoria nello heap, lo spazio di memoria del processo cresce (utilizzare comando **ps** per verificare).
- Sebbene sia possibile restituire la memoria rilasciata al sistema e quindi restringere lo spazio di memoria del processo, ciò non viene quasi mai attuato (vedi oltre: **`malloc`** pool).
- Lo heap cresce “verso l'alto”, cioè ogni nuova allocazione di memoria dinamica ottiene un intervallo con indirizzi numericamente più grandi.

Osservazioni sulla memoria di un processo (II)

- Nello stack, oltre le variabili automatiche (o locali), vengono generalmente memorizzati i parametri passati alla funzione chiamata e il suo valore di ritorno e l'indirizzo dove riprendere l'esecuzione. Tale meccanismo permette l'esecuzione di chiamate ricorsive.
- Le variabili nello stack spariscono non appena la funzione che le contiene ritorna al chiamante. Lo spazio verrà riutilizzato per successive chiamate a funzione.
- In genere lo stack cresce "verso il basso", cioè verso indirizzi numericamente minori.
- Anche se teoricamente è possibile che lo stack e lo heap vadano a sovrapporsi, il kernel previene questa situazione.

Comando size

- Il comando **`size`** riporta la dimensione dei segmenti text, data e bss.

```
$ ls -l myprog
-rwxr-xr-x    1 penguin   birds        12320 Oct 24 16:45 myprog
$ size myprog
   text     data      bss      dec      hex filename
   1458      276        8     1742      6ce myprog
$ strip myprog
$ ls -l myprog
-rwxr-xr-x    1 penguin   birds         3480 Oct 24 16:50 myprog
$ size myprog
   text     data      bss      dec      hex filename
   1458      276        8     1742      6ce myprog
```

- Nell'esempio sopra, la differenza tra la dimensione dell'eseguibile (12320 bytes) e quella di ciò che viene effettivamente caricato in memoria (1742 bytes) è dovuta in gran parte alla tabella dei simboli, una lista di nomi di variabili di programma e di funzioni. Il comando **`strip`** rimuove i simboli, risparmiando un notevole spazio su disco, ma rendendo impossibile il debug del programma. Anche dopo la rimozione dei simboli, il file continua ad avere dimensione maggiore in quanto mantiene informazioni addizionali sul programma, ad esempio quali shared libraries utilizzare.

Allocazione di memoria

- Esistono tre funzioni specificate dall´ANSI C (**standard C library**) per l'allocazione di memoria: **`malloc, calloc, realloc`**
- Esse sono implementate mediante la sottostante **`system call sbrk`**
- La system call **`sbrk`** non viene generalmente chiamata direttamente dall'applicazione in quanto:
  - non è una funzione di allocazione della memoria general-purpose
  - il suo unico compito è di aumentare o diminuire lo spazio di memoria associato ad un processo (il segmento heap).
- Molte implementazioni allocano una memoria leggermente maggiore di quella richiesta per mantenere alcune utili informazioni del tipo dimensione del blocco allocato, puntatore al prossimo blocco, ecc.
- E possibile sostituire la **`malloc`** con un proprio meccanismo di allocazione

Relazione tra malloc e (s)brk

user process

application code

memory allocation function `malloc`

sbrk (wrapper of brk)

kernel

brk

Rilascio della memoria

- Quando non si intende più far riferimento al segmento di memoria allocata bisogna rilasciarlo mediante la funzione **`free`**
- Normalmente la memoria rilasciata non viene ritornata al kernel, cioè il processo non diminuisce la sua dimensione in memoria; lo spazio rilasciato infatti viene tenuto a disposizione di eventuali prossime richieste di allocazione di memoria *(**`malloc`** pool)*
- Se un processo non rilascia i blocchi allocati mediante **`malloc`** (o simili), l'utilizzo della memoria cresce, generando *memory leakage,* cioè un aumento non controllato dell'utilizzo della memoria da parte di un processo

System call malloc e calloc

`void *malloc(size_t size);`

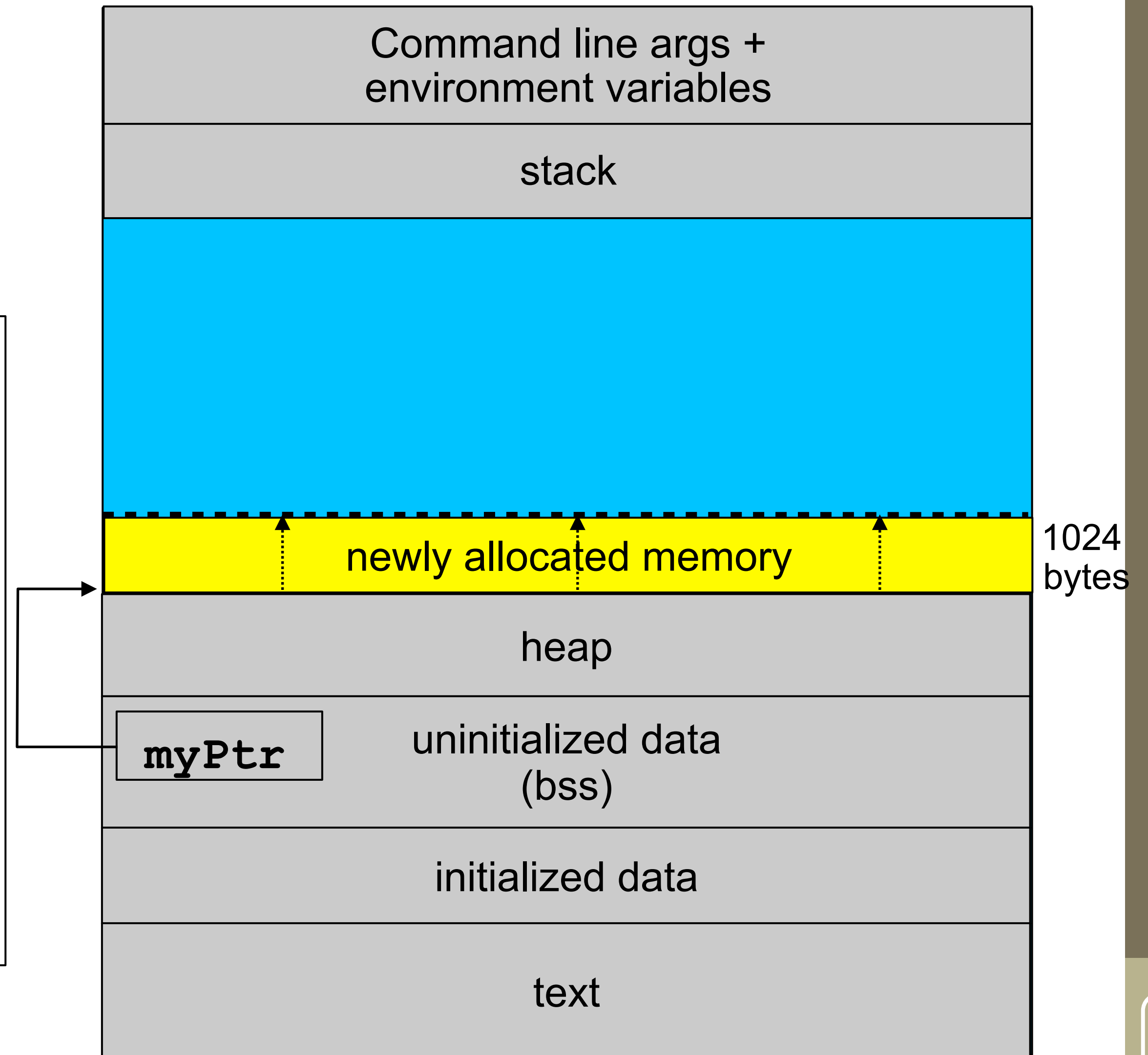
- **`malloc`** alloca uno spazio di memoria di **`size`** bytes.
- La memoria allocata NON viene inizializzata (il suo contenuto è indeterminato)

`void *calloc(size_t nmemb, size_t size);`

- **`calloc`** alloca memoria per un array di **`nmemb`** elementi, ciascuno di dimensione **`size`** bytes
- La memoria allocata viene inizializzata a 0.

`malloc` e layout di memoria di un processo

```
void *myPtr;
main()
{
  size_t mySize = 1024;
  ...
  myPtr = malloc(mySize);
}
```



System call `realloc`

```
void *realloc(void *ptr, size_t newsize);
```

- `realloc` permette di modificare, incrementandola o decrementandola, un'area di memoria precedentemente allocata
- Nel caso di incremento di memoria, se c'è abbastanza spazio oltre la fine della regione esistente, `realloc` alloca lo spazio addizionale e ritorna lo stesso puntatore `ptr` che gli è stato passato; in caso contrario, alloca una nuova area, copia il contenuto di quella indirizzata dal puntatore `ptr` passato nella nuova, rilascia mediante `free` la vecchia area, e infine ritorna il puntatore alla nuova area
- Attenzione: dato che l'area di memoria potrebbe muoversi, evitare di avere prima della `realloc` variabili puntatore che referenziano quest'area
- `realloc(NULL, newsize)` è uguale a `malloc(newsize)`, mentre `realloc(ptr, 0)` è uguale a `free(ptr)`

System call `free`

```
void free(void *ptr);
```

- free permette di rilasciare un'area di memoria precedentemente allocata.
- La memoria rilasciata non viene effettivamente ritornata al kernel, ma rimane in un pool di memoria disponibile, in modo da poter essere utilizzata in una prossima allocazione mediante una delle chiamate **`*alloc`**

Note sull'allocazione di memoria (I)

- Tutte le funzioni di allocazione ritornano un valore di tipo **`void *:`** questo è un generico puntatore non tipizzato *(typeless)*
- Generalmente tale valore di puntatore viene assegnato mediante l'operatore cast (**`<tipo>`**) ad un puntatore tipizzato.
- Il tipo `size_t` è un tipo intero senza segno *(unsigned integral type)* che rappresenta un ammontare di memoria; nei sistemi moderni **`size_t`** è definito come **`unsigned long,`** ma è consigliabile utilizzare **`size_t`** per portabilità
- E' usuale utilizzare l'operatore C **`sizeof()`** in combinazione con le chiamate di allocazione di memoria, per determinare la dimensione in byte di un'area che deve contenere un array di elementi un certo tipo:

```
myPtr=(struct myStruct *)malloc(nelem*sizeof
(struct myStruct));
```

Note sull'allocazione di memoria (II)

- **`calloc`** e **`malloc`** ritornano un puntatore che ha un allineamento in memoria compatibile con qualsiasi tipo di variabile, oppure NULL se la chiamata fallisce.
- Se **`realloc`** fallisce, l'area originaria non viene modificata (né rilasciata, né spostata).
- Errori gravi, che determinano l'interruzione forzata del programma, durante le chiamate di allocazione o rilascio memoria sono per lo più dovuti alla corruzione dello heap, del tipo *overflowing* dei limiti della memoria allocata (scrittura dopo l'ultimo byte allocato) oppure rilascio di uno stesso puntatore per due volte.
- Versioni recenti della ***GNU libc*** includono un'implementazione di **`malloc`** che rileva e notifica eventuali memory leaks, mediante il settaggio della variabile di ambiente **`MALLOC_CHECK_`** (vedi man page **`malloc`**)

System call `alloca`

`void *alloca(size_t size);`

- **`alloca`** è simile alla **`malloc`**, ma lo spazio di memoria allocato risiede nello stack.
- Tale memoria è automaticamente rilasciata quando la funzione che ha chiamato **`alloca`** ritorna al chiamante.
- Consultare la *man page* per verificarne l'effettiva usabilità ed eventuali malfunzionamenti!

Esempio 1 (I)

```
struct coord {          /* 3D coordinates */
    int x, y, z;
} *coordinates;
unsigned int count;     /* how many we need */
size_t amount;          /* total amount of memory */

/* ... determine count somehow... */
amount = count * sizeof(struct coord); /* how many bytes to
allocate */

coordinates = (struct coord *) malloc(amount); /* get the
space */
if (coordinates == NULL) {
    /* report error, recover or give up */
}
/* ... use coordinates ... */
free(coordinates);
coordinates = NULL;      /* not required, but a good idea */
```

Esempio 1 (II)

- Dopo che la memoria è allocata e un puntatore la referenzia, possiamo trattare il puntatore **`coordinates`** come se fosse un array, anche se in realtà è sempre un puntatore:

```
int cur_x, cur_y, cur_z;
size_t an_index;
an_index = 2;
cur_x = coordinates[an_index].x;
cur_y = coordinates[an_index].y;
cur_z = coordinates[an_index].z;
```

- Il compilatore genera il codice appropriato per indirizzare attraverso puntatore l'elemento dell'array e i membri della sua struttura mediante la notazione:

  `coordinates[elem_index].field_name`

Esempio 1 (III)

- Per inizializzare la memoria ritornata da **malloc** si può utilizzare la funzione **memset**:

```
memset(coordinates, 0, amount);
```

- L'altra possibilità è utilizzare **calloc**
- Un approccio all'utilizzo di **malloc** che garantisce l'allocazione di un corretto spazio di memoria anche quando la dichiarazione del puntatore potrebbe essere stata modificata è il seguente:

```
some_type *pointer;
pointer = malloc(count * sizeof(*pointer));
```

Esempio 2

```
int new_count;
size_t new_amount;
struct coord *newcoords;

/* set new_count, for example: */
new_count = count * 2;          /* double the storage */
new_amount = new_count * sizeof(struct coord);

newcoords = (struct coord *) realloc(coordinates,
    new_amount);
if (newcoords == NULL) {
    /* report error, recover or give up */
}

coordinates = newcoords;
/* continue using coordinates ... */
```

Esempio 3 (I)

```
void manage_table(void)
{
    static struct table *table;
    struct table *cur, *p;
    int i;
    size_t count;

    ...
    table = (struct table *)malloc(count * sizeof(struct table));
    /* fill table */
    cur = & table[i];        /* point at item with index i */
    ...
    cur->field = j;          /* use pointer */
    ...
    if (some condition) {    /* need to grow table */
        count += count/2;
        p = (struct table * realloc(table,count*sizeof(struct table));
        table = p;
    }

    cur->field = j;     /* PROBLEM 1: update table element */
    other_routine();    /* PROBLEM 2: other_routine() may call
                        manage_table() */
    cur->field = k;     /* PROBLEM 2: similar to 1 */
    ...
}
```

Esempio 3 (II)

```
table = (struct table *)malloc(count * sizeof(struct
table));
/* fill table */
...
table[i].field = j;   /* Update a member of the i'th
element */
...
if (some_condition) { /* need to grow table */
    count += count/2;
    p = (struct table *)realloc(table,count*sizeof(struct
table));
    table = p;
}

table[i].field = j; /* PROBLEM 1 goes away */
other_routine();    /* Recursively calls us, modifies table
*/
table[i].field = k; /* PROBLEM 2 goes away also */
```

Esempio 4

```
/* Simple implementation of calloc */

void *calloc(size_t nmemb, size_t size)
{
    void *p;
    size_t total;

    total = nmemb * size;   // Compute size
    p = malloc(total);      // Allocate the memory

    if (p != NULL)          // If it worked ...
        memset(p, '\0', total); // Fill it with zeros

    return p;     // Return value is NULL or pointer
}
```

GNU Coding Standards (by Richard Stallman)

- Check every call to **malloc** or **realloc** to see if it returned zero. Check **realloc** even if you are making the block smaller; in a system that rounds block sizes to a power of 2, **realloc** may get a different block if you ask for less space.
- In Unix, **realloc** can destroy the storage block if it returns zero. GNU **realloc** does not have this bug: if it fails, the original block is unchanged. Feel free to assume the bug is fixed. If you wish to run your program on Unix, and wish to avoid lossage in this case, you can use the GNU **malloc**.
- You must expect **free** to alter the contents of the block that was freed. Anything you want to fetch from the block, you must fetch before calling **free**.