Memory (Stack and Heap)

Praktikum „C-Programmierung“

Nathanael Hübbe, Eugen Betke, Michael Kuhn, Jakob Lüttgau, Jannek Squar

Wissenschaftliches Rechnen
Fachbereich Informatik
Universität Hamburg

2018-12-03

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

1 Introduction

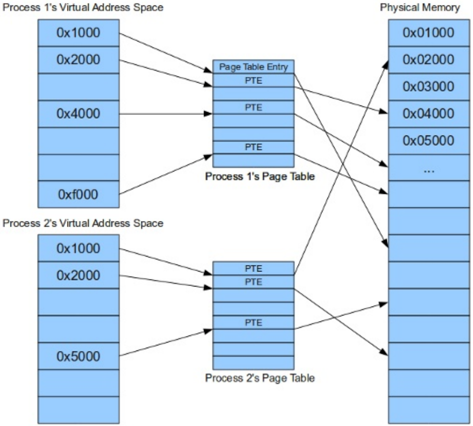
2 Heap vs. Stack

3 Memory management in C
- Library
- Common techniques

4 Pitfalls

5 Quellen

Virtual address space



- Modern operating systems manage memory allocation
- A page table maps physical memory to virtual address space
- Each process see contiguous memory space

Memory layout of C programs

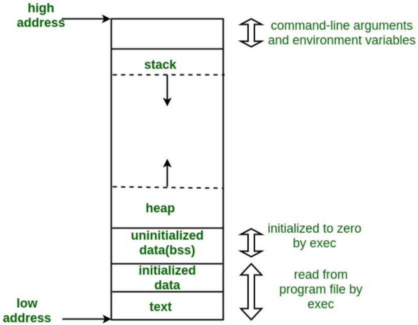


Abbildung: Memory Layout [1]

- Text segment
  - Contains executable instructions
  - Often read only and shared by processes
- DATA:
  - Data segment, initialized by programmer
- BSS:
  - Data segment, uninitialized by programmer
  - Initialized to arithmetic 0
  - Stores global and static variables
- Stack
  - Stores automatic variables
  - Typically grows from higher addresses towards zero
- Heap
  - Dynamically allocated space
  - Begins at the end of BSS segment

1 Introduction

2 Heap vs. Stack

3 Memory management in C
- Library
- Common techniques

4 Pitfalls

5 Quellen

Stack vs. Heap [2]

- Stack
  - very fast access
  - don’t have to explicitly de-allocate variables
  - space is managed efficiently by CPU, memory will not become fragmented
  - local variables only
  - limit on stack size (OS-dependent)
  - variables cannot be resized
- Heap
  - variables can be accessed globally
  - no limit on memory size
  - (relatively) slower access
  - no guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed
  - you must manage memory (you’re in charge of allocating and freeing variables)
  - variables can be resized using realloc()

1 Introduction

2 Heap vs. Stack

3 Memory management in C
- Library
- Common techniques

4 Pitfalls

5 Quellen

C memory management functions

`malloc`	allocates memory
`calloc`	allocates and zeroes memory
`realloc`	expands previously allocated memory block
`free`	deallocates previously allocated memory

- The memory management functions
  - allocate and deallocated memory on the heap
  - are defined in header `<stdlib.h>`

malloc

```c
void *malloc(size_t size);
```

- Allocates size bytes and returns a pointer to the allocated memory.
- The memory is not initialized.
- If size is 0, then malloc() returns either NULL, or a unique pointer value that can later be successfully passed to free().

calloc

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

- Allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory
- The memory is set to zero
- If nmemb or size is 0, then calloc() returns either NULL, or a unique pointer value that can later be successfully passed to free()

realloc

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

- Changes the size of the memory block pointed to by ptr to size bytes
- The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes.
- If the new size is larger than the old size, the added memory will not be initialized.

```
    p = realloc(NULL, size);
    // the same as
    p = malloc(size);
```

```
    realloc(ptr, 0);
    // the same as
    free(ptr)
```

free

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

- Frees the memory space pointed to by ptr
- ptr must have been returned by a previous call to malloc(), calloc(), or realloc().
- Otherwise, or if free(ptr) has already been called before, undefined behavior occurs.
- If ptr is NULL, no operation is performed.

Usage example

```c
/* Allocate space for an array with ten elements of type int. */
int *ptr = (int *) malloc(10 * sizeof (int));
if (ptr == NULL) {
    /* Memory could not be allocated, the program should
    handle the error here as appropriate. */
} else {
    /* Allocation succeeded.  Do something.  */
    free(ptr);
    ptr = NULL;
}
```

Allocation idiom

```
int *ptr1 = malloc(10 * sizeof(int));
```

Listing 1: Straight forward allocation

```
int *ptr2 = malloc(10 * sizeof(*ptr));
```

Listing 2: Allocation idiom

- Insted of `sizeof(int)` we used `sizeof(*ptr)`
- `sizeof()` automatically determines the correct size of `*ptr`
- If type changes, the malloc still provides right amount of memory

Single dynamic allocation with ownership semantics

```
{
    struct obj otmp;
    /* do stuff with otmp */
}

{
    struct obj *otmp;
    otmp = malloc(sizeof *otmp);
    /* do stuff with otmp */
    free(otmp);
}
```

- User is responsible for memory allocation and deallocation
- Memory is allocated and released in the same scope

Dynamic memory allocation in functions

```c
int *my_func(void) {
    int * x;
    x = (int *) malloc(25);
    return x;
}

int main(int argc, char** argv) {
    int *pi = my_func();
    /* do something with pi */
    free(pi);
    return 0;
}
```

- Memory is allocated by `malloc` in a function
- The user is responsible for deallocation

1 Introduction

2 Heap vs. Stack

3 Memory management in C
- Library
- Common techniques

4 Pitfalls

5 Quellen

Common pitfalls

Memory management is a source for bugs [4], e.g.

- Memory leaks
- Use after free
- Freeing not dynamically allocated memory
- Accessing not allocated memory
- Multiple free

Memory leaks

```c
int memory_leak() {
    int* ptr = malloc(sizeof(*ptr));
    return 0;
}
```

- Pointer get lost after function returns
- Memory stays allocated, until program exits

Use after free

```c
int *ptr = malloc(sizeof (int));
free(ptr);
*ptr = 7; /* Undefined behavior */
```

- Memory is used after it was freed
- Results in undefined behaviour

Freeing not dynamically allocated memory

```c
char *msg = "Default message";
int tbl[100];
free(msg);
free(tbl);
```

- Freeing memory that was not allocated by `malloc`, `calloc` or `realloc`
- Results in undefined behaviour

Accessing not allocated memory

```
int *my_func(void) {
    int x[25];
    return x;
}

int main(int argc, char** argv) {
    int *pi = my_func();
    *(pi + 1) = 5;
    free(pi);
    return 0;
}
```

- Not allocated memory is used after freed

Multiple free

```
void main(){
    int *p;
    p = (int *)malloc(10 * sizeof(int));
    f(p);
    free(p);
}
void f(int *g){
    printf("%d", g);
    free(g);
}
```

- The same allocation is freed several times
- Results in undefined behaviour

Summary

- From application’s perspective, memory is a contigous space partitioned in segments (Text, DATA, BSS, Stack and Heap)
- Memory management
  - On stack memory is managed for you
  - On heap the user is responsible for memory management
  - Memory management functions are defined in `<stdlib.h>`
- Common bugs
  - Memory leaks
  - Use after free
  - Freeing not dynamically allocated memory
  - Accessing not allocated memory
  - Multiple free

1 Introduction

2 Heap vs. Stack

3 Memory management in C
  - Library
  - Common techniques

4 Pitfalls

5 Quellen

Quellen I

[1] GeeksforGeeks. Memory Layout of C Programs. `https://www.geeksforgeeks.org/memory-layout-of-c-program/`. Accessed on 03.12.2014.

[2] Gribblelab. Memory: Stack vs Heap. `https://www.gribblelab.org/CBootCamp/7_Memory_Stack_vs_Heap.html`. Accessed on 03.12.2014.

[3] Turkeyland. Buffer Overflows and You. `https://turkeyland.net/projects/overflow/intro.php`. Accessed on 03.12.2014.

Quellen II

[4] Wikibooks. C Programming/stdlib.h/malloc. https://en.wikibooks.org/wiki/C_Programming/stdlib.h/malloc. Accessed on 03.12.2014.