Speicher (Stack and Heap)

Praktikum „C-Programmierung“

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

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

2019-11-11

Wissenschaftliches Rechnen
Fachbereich Informatik
Universität Hamburg

Virtueller Adressraum

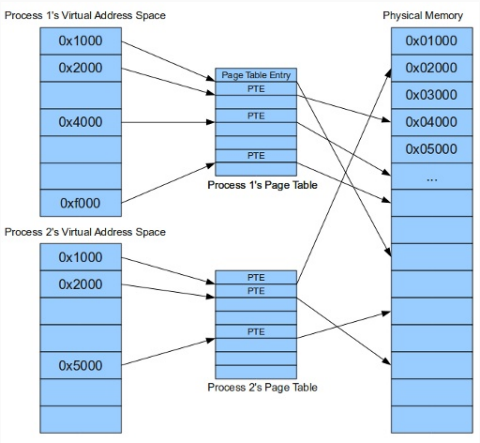

Abbildung 1: Memory mapping [3]

- Speicher wird von Betriebssystemen verwaltet
- Abbildung vom physikalischen Speicher auf virtuellen Adressraum
- Jedem Prozess wird virtuell kontinuierlicher Speicher zur Verfügung gestellt

Speicherlayout von C-Programmen

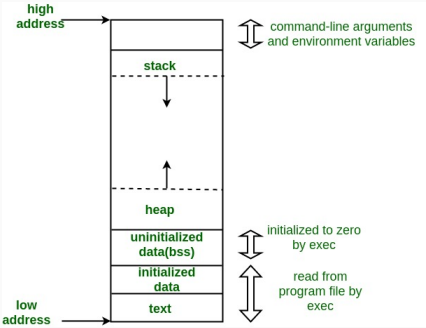

Abbildung 2: Speicherlayout [1]

- Textsegment
  - Beinhaltet ausführbaren Code
  - Oft nur Lesezugriff erlaubt und gemeinsame von mehreren Prozessen genutzt
- DATA:
  - Datensegment, initialisiert von Programmierer
- BSS (block started by segment):
  - Datensegment, nicht initialisiert von Programmierer
  - Initialisiert zu arithmetischen 0
  - Speichert globale und statische Variablen
- Heap
  - Dynamisch reservierter Speicher
  - Beginnt nach dem BSS Segment
- Stack
  - Speichert automatische Variablen
  - Wächst typischerweise von hohen Adressen gegen die 0-Adresse

Stack vs. Heap [2]

- Stack
  - Sehr schneller Zugriff
  - Der Speicherraum wird effizient von der CPU verwaltet
    - Der Speicher wird nicht fragmentiert
  - Variablen müssen nicht explizit freigegeben werden
  - Stackgröße ist beschränkt (betriebssystemabhängig)
  - Variablengröße kann nicht verändert werden
- Heap
  - Relativ langsame Zugriffszeiten
  - Effiziente Speichernutzung ist nicht garantiert
    - Nach längerer Laufzeit, wenn Speicherblöcke reserviert und freigegen werden, kann der Speicher fragmentiert werden
  - Der Nutzer ist für die Speicherverwaltung verantwortlich
  - Keine Begrenzung an Speichergröße
  - Variablengröße kann verändert werden (z.B. mit realloc())

Zeiger Definition

```c
int *pi; /* Zeiger auf ein Integer */
```

1. Datentyp des Zeiger (Datentyp des Wertes + Asterix) und Name
2. Beinhaltet eine Speicheradresse auf eine Position in virtuellen Adressraum
   - Ein nicht initialisierter Zeiger beinhaltet eine zufällige Adresse

```c
int *pi = NULL; /* Initialisierung */
```

- `NULL`-Zeiger zeigt auf eine ungültige Adresse `0`
- Typische Nutzung
  - Initialisierung von Zeigern
  - Fehlerbehandlung
  - Übergabe an Funktionen, wenn keine gültige Adresse vorhanden ist
  - ...

Zeigeroperatoren

```c
int i = 5;
int *pi = &i; /* Adressoperator */
int n = *pi; /* Inhaltsoperator */
```

- Der Adressoperator (`&`) gibt die Adresse der Variable zurück.
- Der Inhaltsoperator (`*`) gibt den Speicherinhalt auf den der Zeiger zeigt.

Adressenausgabe mit printf()

```c
int i = 5;
int *pi = &i;
printf("  i = %d\n", i);
printf("*pi = %d\n", *pi); /* Inhalt */
printf(" pi = %p\n", pi); /* Adresse */
```

Ausgabe:

```
  i = 5
*pi = 5
 pi = 0x7ffc6045c61c
```

Speicherverwaltungsfunktionen in C

`malloc`	Reserviert Speicher.
`calloc`	Reserviert Speicher und initialisiert mit 0.
`realloc`	Verändert die Speichergröße vom reservieren Speicherblock.
`free`	Gibt den reservierten Speicher frei.

- Die Speicherverwaltungsfunktionen
  - reservieren und geben den Speicher auf dem Heap frei
  - sind definiert im Header `stdlib.h`

Speicherreservierung mit malloc()

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

- reserviert `size` Bytes und liefert einen Zeiger auf den reservierten Speicher
- gibt `NULL`, wenn Speicherreservierung scheitert
- initialisiert nicht den Speicher

Speicherreservierung mit calloc()

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

- reserviert `nmemb` Elemente mit der Größe `size` und gibt einen Zeiger auf den reservierten Speicherbereich zurück
- gibt `NULL`, wenn Speicherreservierung scheitert
- initialisiert den Speicher

Speicherreservierung mit realloc()

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

- verändert die Größe des Speicherbereichs von `ptr` zu auf die neue Größe von `size` Bytes
- verändert nicht den Inhalt im Speicherbereich innerhalb von `[0, min(old_size, new_size)]`
- initialisiert nicht den zusätzlich reservierten Speicher

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

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

Speicherfreigabe mit free()

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

- gibt den Speicherbereich frei, auf den `ptr*` zeigt
  - `ptr*` muss zuvor von `malloc()`, `calloc()` oder `realloc()` zurück gegeben sein
- hat keinen definierten Verhalten beim wiederholten Aufruf `free(ptr)`
- führt keine Operationen durch, wenn `ptr` ist NULL

Idiom für Speicherreservierung

```c
int *ptr1 = malloc(10 * sizeof(int));
int *ptr2 = malloc(10 * sizeof(*ptr)); // Idiom
```

- Anstatt von `sizeof(int)` benutzen wir `sizeof(*ptr)`
- `sizeof()` bestimmt automatisch die Größen von `*ptr`
- Vorteil, wenn Datentyp sich verändert, dann reserviert `malloc()` immer noch eine korrekte Speichermenge.

Beispiel: Arrays (Stack vs. Heap)

Stack

```
size_t size = 5;
int array[size];
/* Initialisierung */
for (int i = 0; i < size; ++i) {
 array[i] = i * i;
}
```

Heap

```
size_t size = 5;
int *array = malloc(size * sizeof(*array));
/* Initialisierung */
for (int i = 0; i < size; ++i) {
 array[i] = i * i;
}
free(array);
```

Beispiel: Fehlerbehandlung

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

if (ptr == NULL) {
 /* Error handling */
} else {
 /* Allocation succeeded. Do something. */
 free(ptr);
}
```

Beispiel: Pass-By-Pointer

```c
void print_array(int *array, size_t size, char* comment) {
 printf("%s\n", comment);
 for (int i = 0; i < size; ++i) {
  printf("%d ", array[i]);
 }
 printf("\n");
}
```

`print_array(...)` kann sowohl für Arrays auf dem Stack, als auch auf dem Heap verwendent werden.

Stack

```c
...
int array_s[size];
/* Statt array wird ein Zeiger übergeben */
print_array(array_s, size, "Stack");
...
```

Heap

```c
...
int *array_h = malloc(size * sizeof(array_h));
print_array(array_h, size, "Heap");
free(array_h);
...
```

Dynamische Speicherreservierung mit Eigentumssemantik

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

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

- Der Nutzer ist verantwortlich für Speicherreservierung und Freigabe.
- So ähnlich wie beim Stack, wird der Speicher im gleichen Gültigkeitsbereich reserviert und freigegeben.

Dynamische Speicherreservierung mit Funktionen

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

void func_b() {
 int *pi = func_a();
 /* do something with pi */
 free(pi);
}
```

- Ein Speicherbereich wird in einer Funktion dynamisch reserviert
- Der Nutzer ist für die Freigabe verantwortlich
- Nachteil: Nicht sofort ersichtlich, dass der Rückgabewert freigegeben werden muss

Häufige Fehler

Speicherverwaltung ist fehleranfällig [4], z.B. treten folgende Fehler häufig auf:

- Speicherlecks
- Nutzung nach `free()`
- Freigabe von nicht dynamisch reservierten Speicher
- Zugriff auf nicht reservierten Speicher
- Mehrfacher Aufruf von `free()`

Speicherlecks

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

- Zeiger geht nach dem Funktionsende verloren
- Der Speicher bleibt bis Programmende reserviert

Nutzung nach free

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

- Speichernutzung nach Aufruf von `free()`
- Das Verhalten ist nicht definiert

Freigabe von nicht dynamisch reservierten Speicher

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

- Freigabe vom nicht durch `malloc`, `calloc` or `realloc` reservierten Speicher
- Das Verhalten ist nicht definiert

Zugriff auf nicht reservierten Speicher

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

void func_b() {
 int *pi = func_a();
 *(pi + 1) = 5;
 free(pi); /* Undefined behavior */
}
```

- Nutzung vom nicht reservierten Speicher

Multiple free

```c
void func_a(int *g) {
 printf("%d", g);
 free(g);
}

void func_b() {
 int *p;
 p = (int *)malloc(10 * sizeof(int));
 func_a(p);
 free(p); /* Undefined behavior */
}
```

- Der Speicher wird mehrmals freigegen
- Das Verhalten ist nicht definiert

Summary

- Aus der Programmperspektive ist der Speicher kontinuierlich.
- Der Speicher ist aufgeteilt in Segment, insbesondere in Stack und Heap
- Speicherverwaltung
  - Speicher auf dem Stack wird automatisch verwaltet
  - Für die Speicherverwaltung auf dem Heap ist Benutzer verantwortlich
    - Zugriff auf den Inhalt funktioniert über Zeiger
- Speicherverwaltung ist fehleranfällig

Quellen

[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.

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