Speicher

Teil 1: Memory API

Prof. Dr.-Ing. Andreas Heil

Licensed under a Creative Commons Attribution 4.0 International license. Icons by The Noun Project.

v1.0.1

Lernziele und Kompetenzen

- Unterschiedliche Adressierung von Programminstruktionen, Heap und Stack **verstehen**

Speicherarten: Stack

- Wird implizit (automatisch) reserviert
- Compiler reserviert für die Variable `x` entsprechend Speicher auf dem Stack
- Speicher wird bei Aufruf von `func` alloziert und beim Verlassen der Routine wieder freigegeben

```
void func() {
  int x; // declares an integer on the stack
  ...
}
```

Speicherarten: Heap

- Speicher muss explizit durch Entwickler alloziert werden
- Hinweis: Compiler reserviert Speicher für Pointer, z.B. `int *x`, auf dem Stack
- Prozess fordert Speicher auf dem Heap für ein Integer an
- Zurück kommt die Speicheradresse, an der der Integer Wert auf dem Heap liegt

```
void func() {
  int *x = (int *) malloc(sizeof(int));
  ...
}
```

Speicher reservieren

Speicher für eine Fließkommazahl reservieren:

```
double *d = (double *) malloc(sizeof(double));
```

- Compiler kennt die Größe des Datentyps
- Sie auch? 8 Byte, 32-Bit Fließkommazahlen, war schon dran, oder?

Array für 10 Integer-Werte reservieren

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

Speicher freigeben

Allozierten Speicher wieder freigeben

```
double *x = malloc(10 * sizeof(int));
…
free(x);
```

Typische Fehler beim Umgang mit Speicher (1)

Vergessen Speicher zu reservieren

```
char *src = “hello world”;
char *dst;                  // Speicher nicht reserviert
strcpy (dst, scr);
```

▶ Resultiert in sog. » Segmentation Fault«

Korrekt wäre:

```
char *src = “hello world”;
char *dst = (char *) malloc(strlen(src) + 1);
strcpy (dst, scr);
```

Typische Fehler beim Umgang mit Speicher (2)

Nicht genügend Speicher reserviert

```
char *src = “hello world”;
char *dst = (char *) malloc(strlen(src)); // String um 1 Zeichen zu kurz
strcpy (dst, scr);
```

- Das kann laufen, kann aber auch abstürzen
- Je nachdem ob `malloc` hier ggf. ein Byte mehr alloziert
- Verlassen sollten Sie sich darauf allerdings nicht… 🙈

Typische Fehler beim Umgang mit Speicher (3)

Speicher reserviert, aber vergessen zu initialisieren

- Egal ob initialisiert oder nicht, es wird auf jeden Fall etwas aus dem Speicher gelesen
- Und zwar das was vorher drin war 😯
- Nennt sich dann »Uninitialized Read«

Typische Fehler beim Umgang mit Speicher (4)

Speicher nicht freigegeben

- Ein Klassiker
- Hatten wir schon einmal zu Beginn der Vorlesungsreihe
- Herzlichen Glückwunsch, Sie haben ein Speicherleck (engl. memory leak) gebaut 🤦
- Kann man auch bei höheren Programmiersprachen erreichen, indem Referenzen nicht »aufgeräumt« werden

Typische Fehler beim Umgang mit Speicher (5)

Speicher freigegeben obwohl er noch benötigt wird

- Klingt schon so, als wäre das keine gute Idee
- Nennt sich »Dangling Pointer«
- GGf. noch benötigte Daten können ab dann durch erneutes `malloc` überschreiben werden

Typische Fehler beim Umgang mit Speicher (5)

Speicher mehrfach freigeben

- Man sollte denken, das sollte kein Unterschied machen
- Ergebnis ist allerdings nicht exakt definiert
- Nennt sich »Double Free«
- Immer wieder gut, um die zugrundeliegenden Bibliotheken zur Speicherverwaltung maximal zu verwirren 😵

malloc, free und das Betriebssystem

- `malloc` und `free` beziehen sich immer nur auf den virtuellen Adressraum eines Prozesses
- Auch wenn ein Speicherleck gebaut wird und der gesamte Heap voll läuft gilt:
  - Das Betriebssystem holt sich nach Prozessende den gesamten Speicher zurück
  - Kann aber Probleme bei langlaufenden Prozessen (Web Server o.ä. machen)
- Viel größeres Problem wenn im Betriebssystem selbst ein Speicherleck enthalten ist
- `malloc` und `free` sind selbst keine SysCalls
- `brk` und `sbrk` sind SysCalls zum Ändern des Heaps
- `mmap` zum Erzeugen eines neuen Speicher-Mappings in den virtuellen Adressraum

Referenzen

Bildnachweise