

Übungen zu Systemprogrammierung 1

Ü5 – Freispeicherverwaltung

Wintersemester 2022/23

Luis Gerhorst, Jonas Rabenstein, Dustin Nguyen, Christian Eichler, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg



Lehrstuhl für Verteilte Systeme
und Betriebssysteme



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT



4.1 Freispeicherverwaltung

4.2 Implementierung

4.3 gdb

4.4 Aufgabe 5: halde

4.5 Gelerntes anwenden



4.1 Freispeicherverwaltung

4.2 Implementierung

4.3 gdb

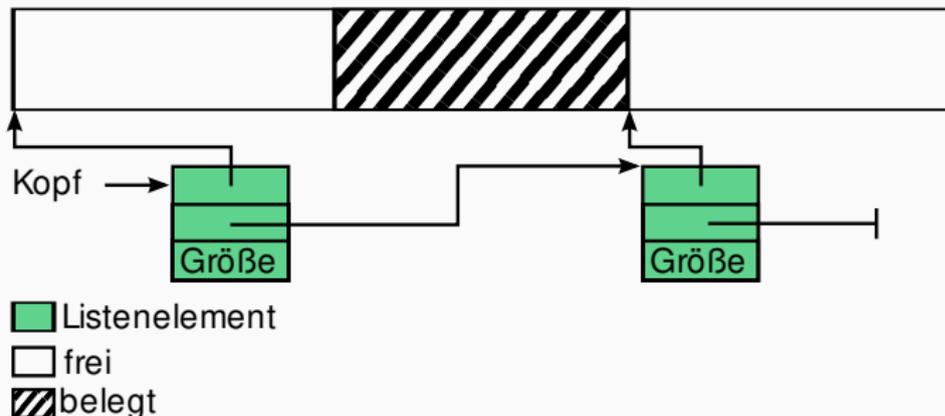
4.4 Aufgabe 5: halde

4.5 Gelerntes anwenden



- Anforderung von Speicher: `void *malloc(size_t size);`
 - Parameter: Größe des angeforderten Speichers
 - Rückgabewert: Zeiger auf einen Speicherbereich
- **Explizite** Freigabe: `void free(void *ptr);`
 - Parameter: Zeiger auf freizugebenden Speicherbereich
 - Rückgabewert: –

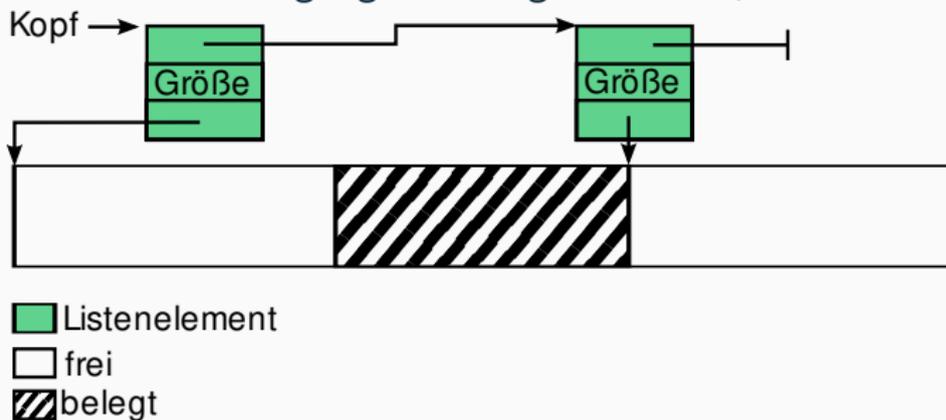
- Ziel: Speicherbereiche, die zur Laufzeit in beliebiger Größe angefordert werden können
- Skizze: Zustand eines teilweise belegten Heaps



- Welche Informationen muss eine Freispeicherverwaltung bereit halten?
 - für freie Blöcke: Größe und Lage des Speicherbereichs
 - für belegte Blöcke: Größe des Speicherbereichs
- Welche Datenstruktur ist für eine Freispeicherverwaltung geeignet?
 - KISS (Keep it small and simple): einfach verkettete Liste



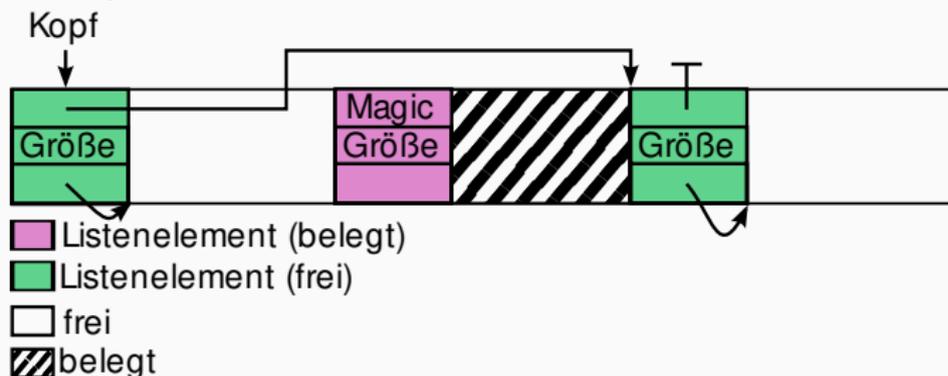
- Konzept einer Freispeicherverwaltung auf Basis einer verketteten Liste (ohne Berücksichtigung der belegten Blöcke!)



- Freie Blöcke werden in einer verketteten Liste gespeichert
- Wiederholung Übung 1
 - Wie wird eine verkettete Liste in C implementiert?

```
insertVal() → malloc() → insertVal() → malloc() →  
insertVal() → malloc() → insertVal() → malloc() →  
insertVal() → malloc() → insertVal() → malloc() →  
insertVal() → ...
```

- Woher den Speicher für die Listenelemente nehmen?



- Listenelemente werden innerhalb des verwalteten Speichers am Anfang des jeweiligen Speicherbereichs abgelegt
- Listenelemente auch in belegten Blöcken vorhanden, aber nicht verkettet
 - Verweis auf nächstes Listenelement wird zur Realisierung eines Schutzmechanismus eingesetzt
 - Abspeichern eines wohldefinierten magischen Wertes und Überprüfung des Wertes vor dem Freigeben



4.1 Freispeicherverwaltung

4.2 Implementierung

4.3 gdb

4.4 Aufgabe 5: halde

4.5 Gelerntes anwenden



■ Listenelementdefinition in C

```
struct mblock {  
    struct mblock *next; // Zeiger zur Verkettung  
    size_t size;         // Größe des Speicherbereichs  
    char mem_area[];     // Anfang des Speicherbereichs  
};
```

■ Verwendung von FAM (Flexible Array Member):

- mem_area ist **ein Feld beliebiger Länge**
- In unserem Fall: mem_area ist ein konstanter „Verweis“ auf das Ende der Struktur
- mem_area selbst hat die Größe 0



```
struct test {  
    uint64_t num; // 8 Byte  
    void *ptr; // 8 Byte (auf amd64)  
}
```

```
struct test *arr .....  
    &arr[0] - - - - -  
    &arr[2] -----  
    arr + 3 -----  
    ((uint64_t*) arr) + 7 -----  
    ((char*) arr) + 7 -----  
    &(5[arr]) -----  
    &(5[5]) ----- ⚡  
    (void*) arr ----- ⚡  
    ((void*) arr) + 4 -----
```

6	ptr = ●
	num = 9
5	ptr = ●
	num = 8
4	ptr = ●
	num = 7
3	ptr = ●
	num = 6
2	ptr = ●
	num = 5
1	ptr = ●
	num = 4
0	ptr = ●
	num = 3



- Schrittweises Abarbeiten des folgenden Codestückes:

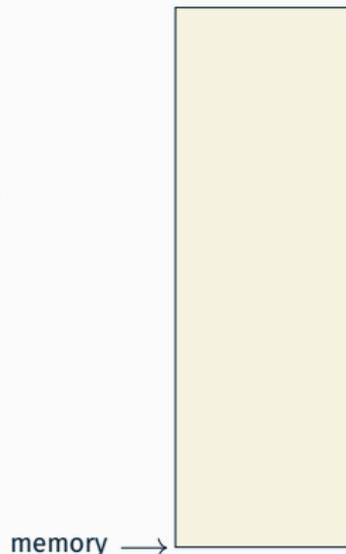
```
char *m1 = (char *)malloc(10);  
char *m2 = (char *)malloc(20);  
  
free(m2);
```

- Annahmen:

- Freispeicherverwaltung verwaltet 100 Bytes Speicher
- Verwendung von absoluten Größen (Annahme: 64-Bit-Architektur)
 - Größe eines Zeigers: 8 Bytes
 - Größe der `struct mblock`: 16 Bytes

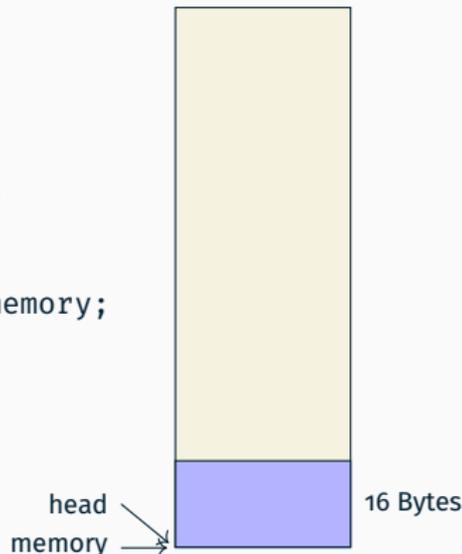


- Speicher vom Betriebssystem anfordern
 - durch statische Allokation zur Ladezeit
`char memory[100];`
 - durch Systemaufruf zur Laufzeit
`char *memory = mmap(NULL, 100, ...);`



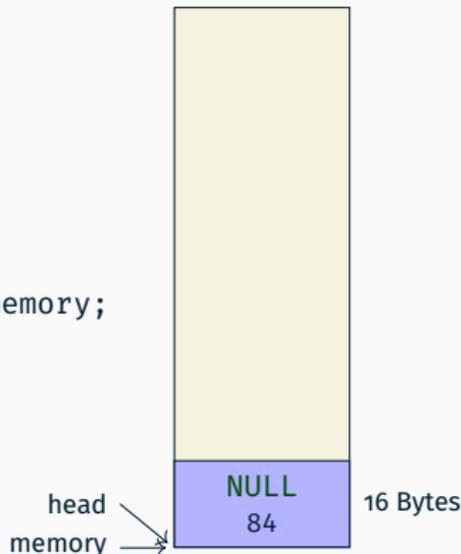


- Speicher vom Betriebssystem anfordern
 - durch statische Allokation zur Ladezeit
`char memory[100];`
 - durch Systemaufruf zur Laufzeit
`char *memory = mmap(NULL, 100, ...);`
- `struct mblock` reinlegen
`struct mblock *head = (struct mblock *)memory;`





- Speicher vom Betriebssystem anfordern
 - durch statische Allokation zur Ladezeit
`char memory[100];`
 - durch Systemaufruf zur Laufzeit
`char *memory = mmap(NULL, 100, ...);`
- `struct mblock` reinlegen
`struct mblock *head = (struct mblock *)memory;`
- `struct mblock` initialisieren
`head->next = NULL;`
`head->size = 84;`



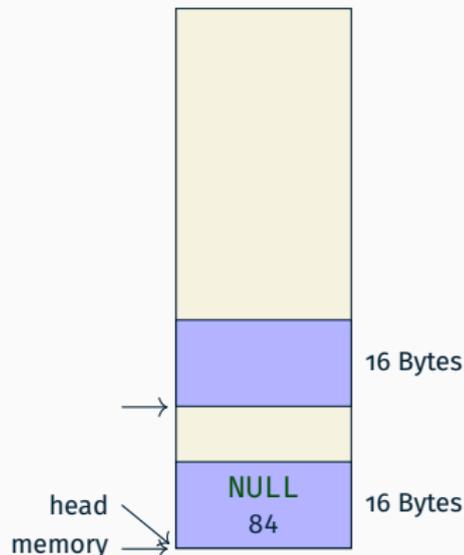
- ! zwei Zeiger mit unterschiedlichem Typ auf den gleichen Speicherbereich
 - unterschiedliche Semantik beim Zugriff (Zeigerarithmetik, Strukturkomponenten)



■ Speicheranforderung von 10 Bytes

```
char *m1 = (char *)malloc(10);
```

- Freispeicherliste nach mblock mit ausreichend Speicher durchsuchen
- 10 Bytes hinter dem head-mblock einen neuen mblock anlegen

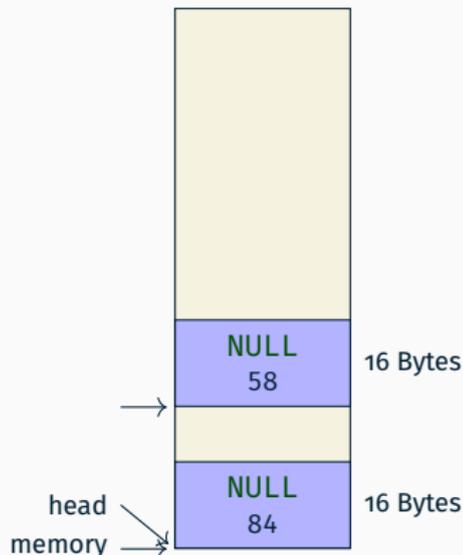




■ Speicheranforderung von 10 Bytes

```
char *m1 = (char *)malloc(10);
```

- Freispeicherliste nach mblock mit ausreichend Speicher durchsuchen
- 10 Bytes hinter dem head-mblock einen neuen mblock anlegen
- ... und initialisieren

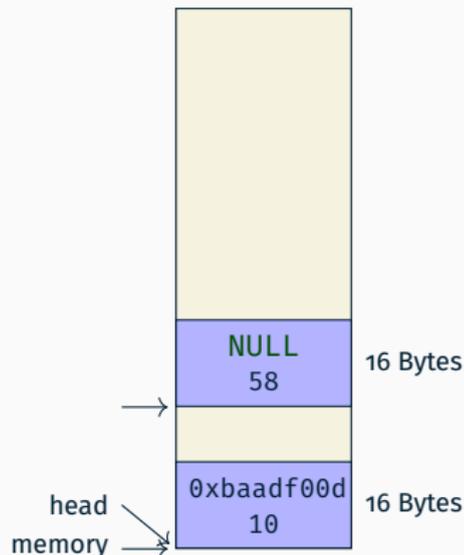




■ Speicheranforderung von 10 Bytes

```
char *m1 = (char *)malloc(10);
```

- Freispeicherliste nach mblock mit ausreichend Speicher durchsuchen
- 10 Bytes hinter dem head-mblock einen neuen mblock anlegen
- ... und initialisieren
- Bisherigen head-mblock anpassen
 - als belegt markieren
 - Größe des Speicherbereichs aktualisieren

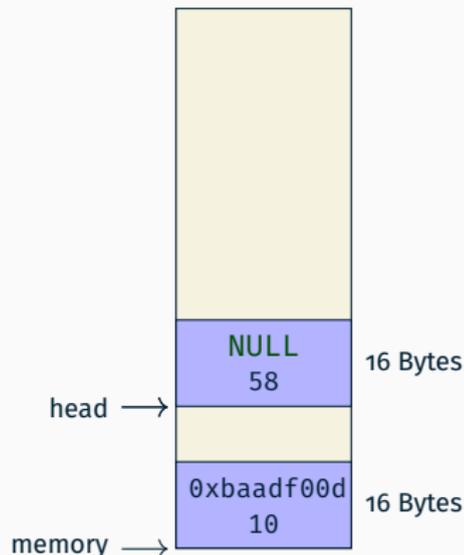




■ Speicheranforderung von 10 Bytes

```
char *m1 = (char *)malloc(10);
```

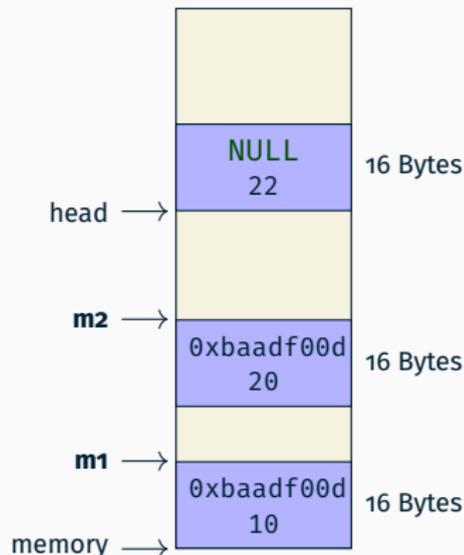
- Freispeicherliste nach mblock mit ausreichend Speicher durchsuchen
- 10 Bytes hinter dem head-mblock einen neuen mblock anlegen
- ... und initialisieren
- Bisherigen head-mblock anpassen
 - als belegt markieren
 - Größe des Speicherbereichs aktualisieren
- head-Zeiger auf neues Kopfelement setzen





■ Situation nach 2 malloc()-Aufrufen

```
char *m1 = (char *)malloc(10);  
char *m2 = (char *)malloc(20);
```

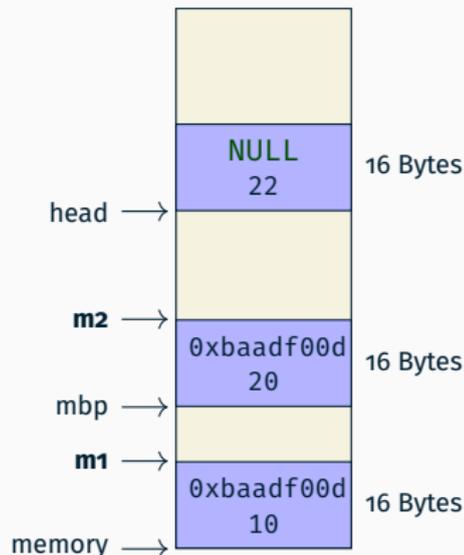




■ Freigabe von m2

```
free(m2);
```

- Zeiger mbp auf zugehörigen mblock ermitteln
- Überprüfen, ob ein gültiger, belegter mblock vorliegt (0xbaadf00d)

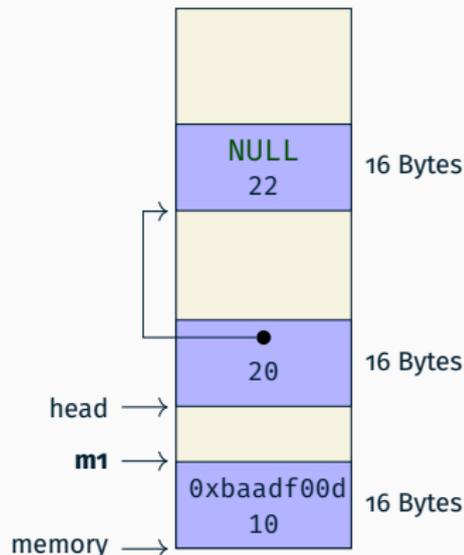




■ Freigabe von m2

```
free(m2);
```

- Zeiger mbp auf zugehörigen mblock ermitteln
- Überprüfen, ob ein gültiger, belegter mblock vorliegt (0xbaadf00d)
- head auf freigegebenen mblock setzen, bisherigen head-mblock verketten





- Mit dem Systemaufruf `mmap(2)` kann ein Programm seinen virtuellen/logischen Adressraum zur Laufzeit erweitern lassen:

```
#include <sys/mman.h>
```

```
void *mmap(void *addr, size_t length, int prot,  
           int flags, int fd, off_t offset);
```

- Beliebige freie Startadresse: `addr = NULL;`
- Les- und schreibbar, aber nicht ausführbar:
`prot = PROT_READ | PROT_WRITE; // nicht PROT_EXECUTE`
- Speicher soll nicht mit anderen Prozessen geteilt werden:
`flags = MAP_PRIVATE | MAP_ANONYMOUS;`
`fd = -1;`
`offset = 0;`
- Rückgabewert ist `MAP_FAILED` oder gültige Speicheradresse



- Mit dem Systemaufruf `mmap(2)` kann ein Programm seinen virtuellen/logischen Adressraum zur Laufzeit erweitern lassen:

```
#include <sys/mman.h>
```

```
void *mmap(NULL, size_t length, PROT_READ | PROT_WRITE,  
           MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
```



- Mit dem Systemaufruf `mmap(2)` kann ein Programm seinen virtuellen/logischen Adressraum zur Laufzeit erweitern lassen:

```
#include <sys/mman.h>
```

```
void *mmap(NULL, size_t length, PROT_READ | PROT_WRITE,  
           MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
```

- Mit diesen Parametern funktional vergleichbar mit `malloc(3p)`
 - aber *noch* langsamer
 - nur zum grobgranularen An-/Nachfordern geeignet
vgl. Vorlesung B VI.2/20



- Mit dem Systemaufruf `mmap(2)` kann ein Programm seinen virtuellen/logischen Adressraum zur Laufzeit erweitern lassen:

```
#include <sys/mman.h>
```

```
void *mmap(NULL, size_t length, PROT_READ | PROT_WRITE,  
          MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
```

- Mit diesen Parametern funktional vergleichbar mit `malloc(3p)`
 - aber *noch* langsamer
 - nur zum grobgranularen An-/Nachfordern geeignet
vgl. Vorlesung B VI.2/20

MAP_ANONYMOUS nicht in POSIX-1.2008, sondern Linux-spezifisch

- In dieser Aufgabe ausnahmsweise `-D_GNU_SOURCE` statt `-D_XOPEN_SOURCE=700`
- POSIX kann nicht basierend auf POSIX implementiert werden



- sehr einfache Implementierung – in der Praxis problematisch
 - Speicher wird im Laufe der Zeit stark fragmentiert
 - Suche nach passender Lücke dauert zunehmend länger
 - eventuell keine passende Lücke mehr vorhanden, obwohl insgesamt genug Speicher frei ist
 - in der Praxis: Verschmelzung benachbarter Freispeicherblöcke
- kein nachträgliches Vergrößern des Heaps
 - in der Praxis: erneut Speicher vom Betriebssystem nachfordern
- langsame Suche nach freiem Speicherbereich passender Größe
 - in der Praxis: Gruppierung der freien Speicherbereiche (Buckets)
- sinnvolle Implementierung erfordert geeignete Speichervergabestrategie
 - Implementierung erheblich aufwändiger – Resultat aber entsprechend effizienter
 - Strategien werden im Abschnitt Speicherverwaltung in SP2 behandelt (z. B. First-Fit, Best-Fit, Worst-Fit oder Buddy-Verfahren)



4.1 Freispeicherverwaltung

4.2 Implementierung

4.3 gdb

4.4 Aufgabe 5: halde

4.5 Gelerntes anwenden



- Ein Debugger dient zum Suchen und Finden von Fehlern in Programmen
- Im Debugger kann man u.a.
 - das Programm schrittweise abarbeiten
 - Variablen- und Speicherinhalte ansehen und modifizieren
 - core dumps (Speicherabbilder beim Programmabsturz) analysieren
 - Erlauben von core dumps (in der laufenden Shell): z.B. `limit coredumpsize 1024k` oder `limit coredumpsize unlimited`
- Das zu analysierende Programm sollte mit folgenden Optionen übersetzt werden
 - `-g`, damit es Debug-Symbole enthält
 - `-O0`, um Übersetzeroptimierungen auszuschalten (kann das Laufzeitverhalten beeinflussen)
- Aufruf des Basis-Debuggers mit `gdb <Programmname>`
- Inklusive Visualisierung des Quelltextes: `cgdb <Programmname>`



```
/* Mit folgenden Übersetzeroptionen kompilieren:  
* -O0 -g  
*/  
#include <stdio.h>  
  
static void initArray(long *array, size_t size) {  
    for (size_t i = 0; i <= size; i++) {  
        array[i] = 0;  
    }  
}  
  
int main(void) {  
    long *array;  
    long buf[7];  
  
    array = buf;  
    initArray(buf, sizeof(buf)/sizeof(long));  
  
    while (array != buf+sizeof(buf)/sizeof(long)) {  
        printf("%ld\n", *array);  
        array++;  
    }  
}
```



- Programmausführung beeinflussen
 - Breakpoints setzen:
 - `break [<Dateiname>:]<Funktionsname>`
 - `break <Dateiname>:<Zeilennummer>`
 - Starten des Programms mit `run` (+ evtl. Befehlszeilenparameter)
 - Fortsetzen der Ausführung bis zum nächsten Stop mit `c` (continue)
 - schrittweise Abarbeitung auf Ebene der Quellsprache mit
 - `step` (läuft in Funktionen hinein)
 - `next` (behandelt Funktionsaufrufe als einzelne Anweisung)
 - Breakpoints anzeigen: `info breakpoints`
 - Breakpoint löschen: `delete breakpoint#`



- Variableninhalte anzeigen/modifizieren
 - Anzeigen von Variablen mit: `p expr`
 - `expr` ist ein C-Ausdruck, im einfachsten Fall der Name einer Variable
 - Automatische Anzeige von Variablen bei jedem Programmstopp (Breakpoint, Step, ...): `display expr`
 - Setzen von Variablenwerten mit `set <variablenname>=<wert>`
- Ausgabe des Funktionsaufruf-Stacks (backtrace): `bt`
- Quellcode an aktueller Position anzeigen: `list`
- Watchpoints: Stoppt Ausführung bei Zugriff auf eine bestimmte Variable
 - `watch expr`: Stoppt, wenn sich der Wert des C-Ausdrucks `expr` ändert
 - `rwatch expr`: Stoppt, wenn `expr` gelesen wird
 - `awatch expr`: Stopp bei jedem Zugriff (kombiniert `watch` und `rwatch`)
 - Anzeigen und Löschen analog zu den Breakpoints



4.1 Freispeicherverwaltung

4.2 Implementierung

4.3 gdb

4.4 Aufgabe 5: halde

4.5 Gelerntes anwenden



■ Ziele der Aufgabe

- Zusammenhang zwischen „nacktem Speicher“ und typisierten Datenbereichen verstehen
- Funktion aus der C-Bibliothek / POSIX selbst realisieren
- Entwickeln eigener Testfälle für selbstgeschriebenen Code

■ Vereinfachungen

- First-Fit-ähnliche Allokationsstrategie
- 1 MiB Speicher nur einmal bei der Initialisierung vom Betriebssystem Anfordern
- freier Speicher wird in einer einfach verketteten Liste (unsortiert) verwaltet
- benachbarte freie Blöcke werden nicht verschmolzen
- `realloc` wird grundsätzlich auf `malloc`, `memcpy` und `free` abgebildet



4.1 Freispeicherverwaltung

4.2 Implementierung

4.3 gdb

4.4 Aufgabe 5: halde

4.5 Gelerntes anwenden



„Aufgabenstellung“

- Skizzieren Sie den Aufbau des verwalteten Speicherbereichs (hier: 64 Bytes, `sizeof(struct mblock) = 16 Bytes`) nach jedem Schritt des jeweiligen Szenarios

- Szenario 1:

```
char *c1 = (char *)malloc(5);  
char *c2 = (char *)malloc(7);  
free(c1);
```

- Szenario 2:

```
char *c1 = (char *)malloc(20);  
free(c1);  
char *c2 = (char *)malloc(4);
```

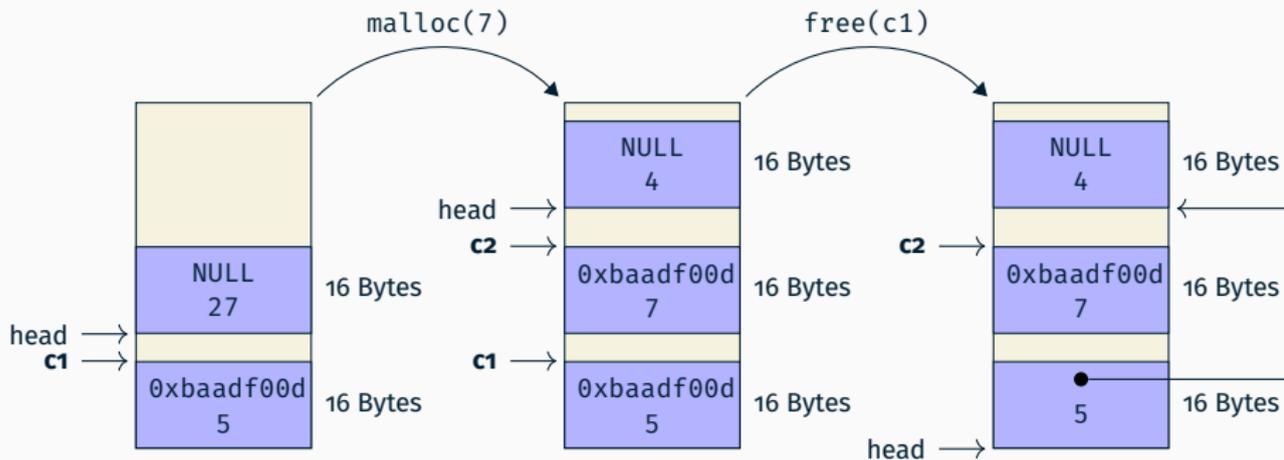
- Szenario 3:

```
char *c1 = (char *)malloc(18);  
char *c2 = (char *)malloc(14);  
free(c1);
```



■ Szenario 1:

```
char *c1 = (char *)malloc(5);  
char *c2 = (char *)malloc(7);  
free(c1);
```



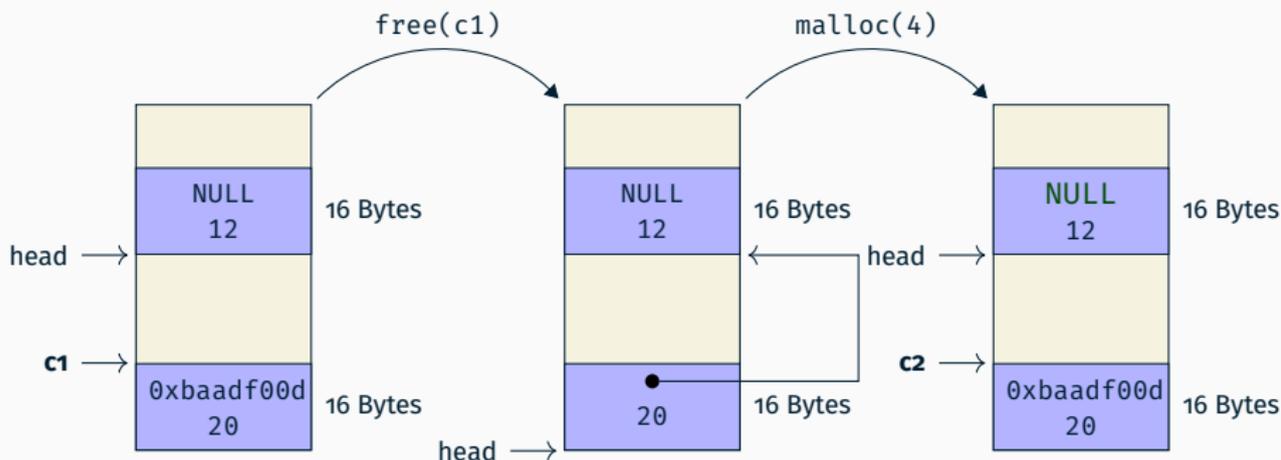


■ Szenario 2:

```
char *c1 = (char *)malloc(20);
```

```
free(c1);
```

```
char *c2 = (char *)malloc(4);
```





■ Szenario 3:

```
char *c1 = (char *)malloc(18);  
char *c2 = (char *)malloc(14);  
free(c1);
```

