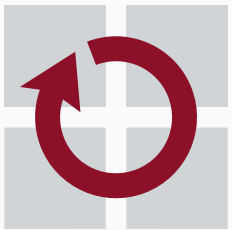
Übungen zu Systemprogrammierung 1

Ü5 – Freispeicherverwaltung

Sommersemester 2025

Luis Gerhorst, Thomas Preisner, Tobias Häberlein, Jürgen Kleinöder

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



Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Technische Fakultät

Agenda

4.1 gdb

4.2 Freispeicherverwaltung

4.3 Implementierung

4.4 Aufgabe 5: halde

4.5 Gelerntes anwenden

Agenda

Debugger: gdb

- 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
- 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>`

Demo

```c
/* 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++;
  }
}
```

Befehlsübersicht

- Programmausführung beeinflussen
  - Breakpoints setzen:
    - **b**reak [<Dateiname>:]<Funktionsname> [if <Bedingung>]
    - **b**reak [<Dateiname>:]<Zeilennummer> [if <Bedingung>]
  - Starten des Programms mit **r**un (+ evtl. Befehlszeilenparameter)
  - An laufendes Programm anhängen mit **a**ttach <pid>
  - Fortsetzen der Ausführung bis zum nächsten Stop mit **c**ontinue
  - schrittweise Abarbeitung auf Ebene der Quellsprache mit
    - **s**tep (läuft in Funktionen hinein)
    - **n**ext (behandelt Funktionsaufrufe als einzelne Anweisung)
    - **fin**ish (beendet aktuelle Funktion/Schleife/etc.)
  - Breakpoints anzeigen: **i**nfo **b**reakpoints
  - Breakpoint löschen: **d**elete breakpoint#
- Quellcode:
  - Quellcode an aktueller Position anzeigen: **l**ist
  - Zwei Fenster mit Quellcode und gdb-Kommandozeile: layout src
  - focus cmd und focus src um ein bestimmtes Fenster zu fokussieren.

Befehlsübersicht

- Variableninhalte anzeigen/modifizieren
  - Anzeigen von Variablen mit: **p**`rint (type) expr`
    - `expr` ist ein C-Ausdruck, im einfachsten Fall der Name einer Variable
    - Typecasts sind über `type` möglich
  - Automatische Anzeige von Variablen bei jedem Programmstopp (Breakpoint, Step, ...): `display expr`
  - Setzen von Variablenwerten mit `set <variablenname>=<wert>`
- Ausgabe des Funktionsaufruf-Stacks (backtrace): **b**`ack`**t**`race`
- 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
- Bildschirm leeren: STRG+L

Agenda

Dynamische Speicherverwaltung

- Ziel: Anforderung von Speicherbereichen beliebiger Größe zur Laufzeit.
- Jedes Maschinenprogramm hat zur Laufzeit seinen eigenen Haldenspeicher (Heap).
  - Speicherbereich im Adressraum des Programms
  - Wird dem Programm durch das Betriebssystem zugewiesen

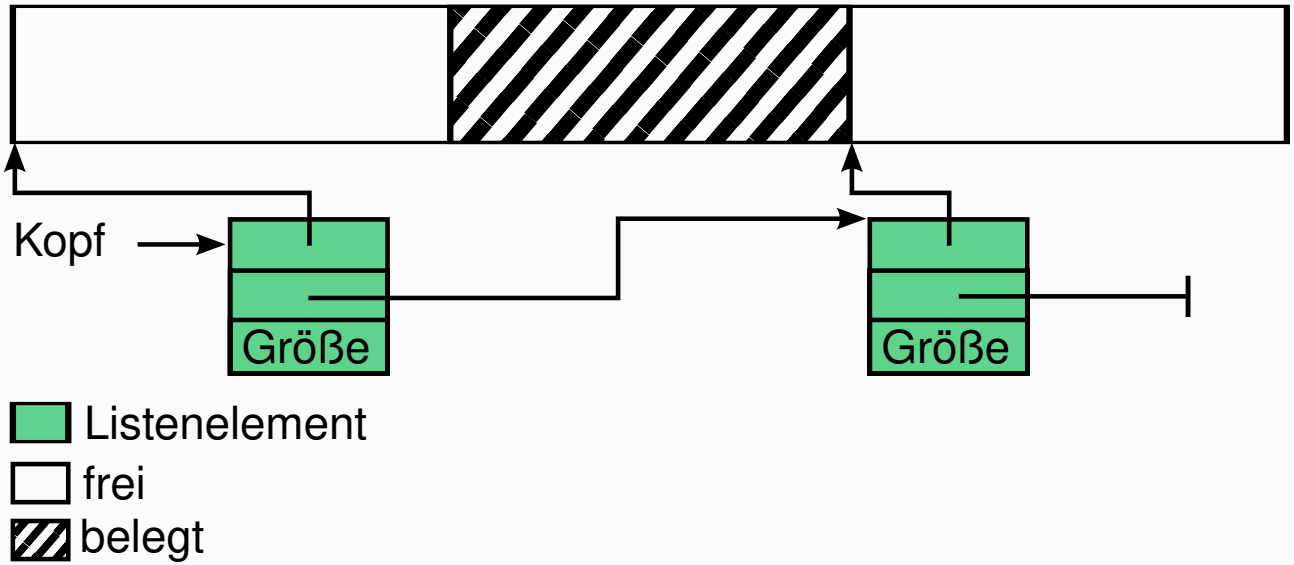
  $\Rightarrow$ Für die Verwaltung des Heaps ist jedes Programm selbst zuständig
  - Verwaltung wird in C durch Bibliotheksfunktionen übernommen
- Heap enthält
  - Strukturen zur Kennzeichnung freier und belegter Speicherbereiche (Freispeicherliste / Lochliste)
  - Die zu speichernden Daten selbst

Dynamische Speicherverwaltung in C

- 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: –

Initialisierung des Heaps

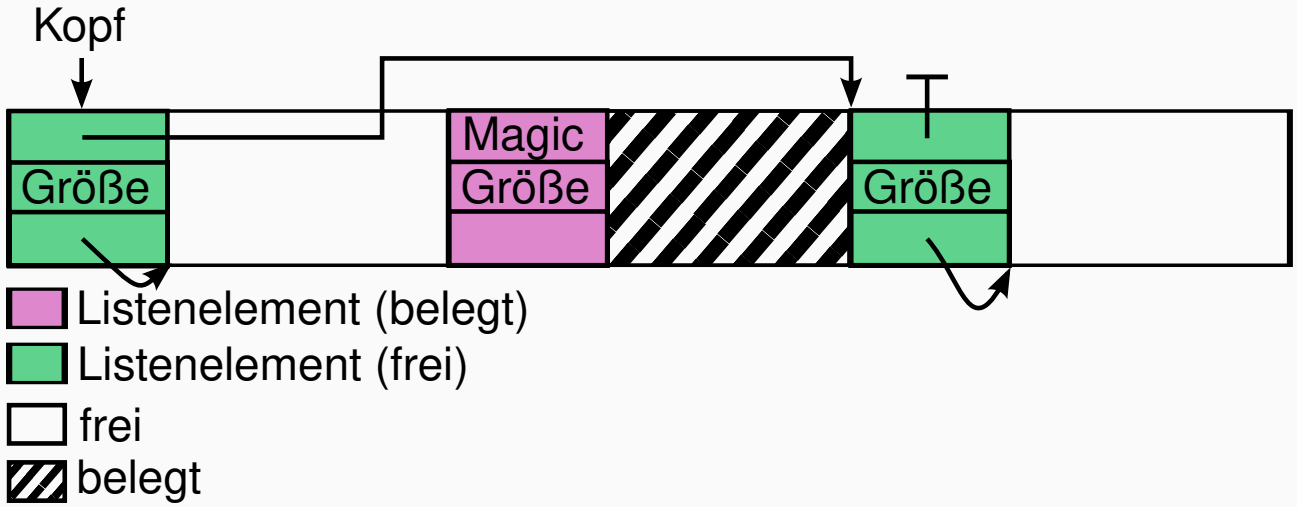
- Skizze: Zustand eines teilweise belegten Heaps



  - Welche Informationen muss eine Freispeicherverwaltung bereit halten?
    - für freie Blöcke (Löcher): 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

Speicher für die Listenelemente

- Wo wird die Liste selbst gespeichert?



  - Verwaltungs-Listenelemente am Anfang des jew. Speicherbereichs
  - auch in belegten Blöcken vorhanden, aber nicht verkettet

    ⇒ Realisierung eines Schutzmechanismus:
    - Abspeichern eines (wohldefinierten) magischen Wertes in Verweis auf nächstes Listenelement
    - Belegter Block kann nur freigegeben werden, wenn der magische Wert im entsprechenden Feld steht.

Agenda

Implementierung

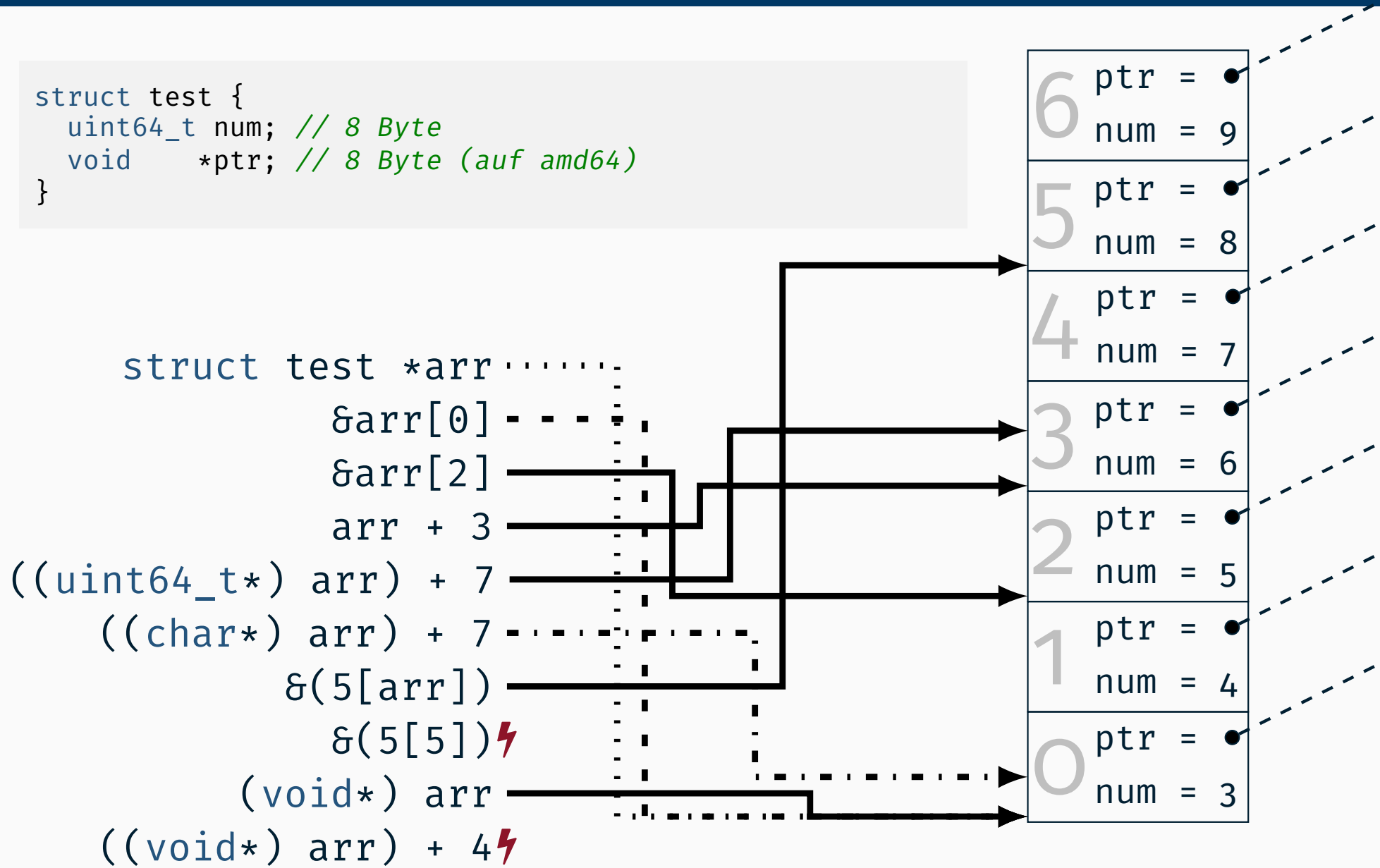
- Listenelementdefinition in C

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

- Verwendung von FAM (Flexible Array Member):
  - `memory` ist **ein Feld beliebiger Länge**
  - In unserem Fall: `memory` ist ein konstanter „Verweis” auf das Ende der Struktur
  - `memory` selbst hat die Größe `0`

Einschub: Zeiger und Zeigerarithmetik

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



Beispiel auf den Folien

- Schrittweises Abarbeiten des folgenden Codestückes:

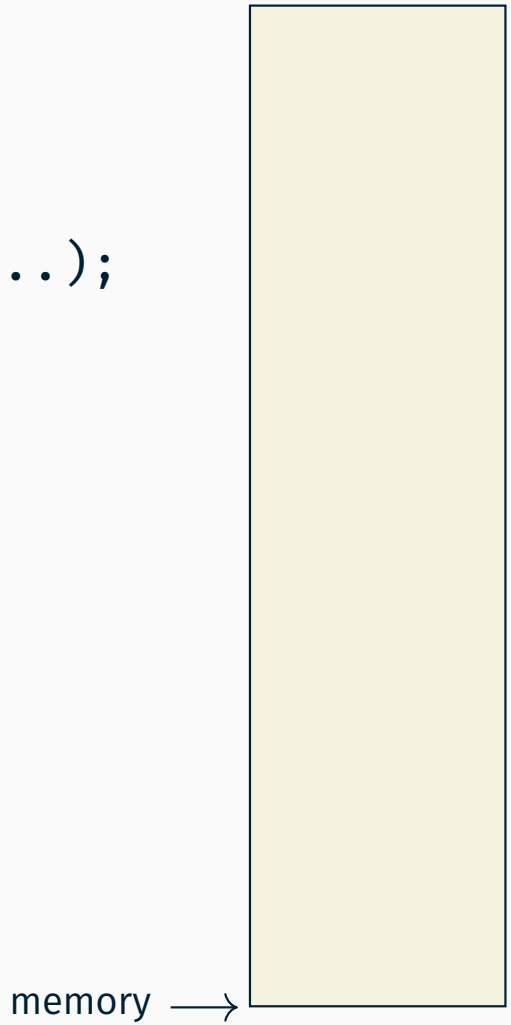
```
char *m1 = malloc(16);
char *m2 = malloc(32);

free(m2);
```

- Annahmen:
  - Freispeicherverwaltung verwaltet 128 Byte an Speicher
  - Verwendung von absoluten Größen (Annahme: 64-Bit-Architektur)
    - Größe eines Zeigers: 8 Bytes
    - Größe der `struct mblock`: 16 Bytes

Initialisierung

- Speicher vom Betriebssystem anfordern:
  - durch statische Allokation zur Ladezeit:
    ```
    char memory[128];
    ```
  - durch Systemaufruf zur Laufzeit:
    ```
    char *memory = mmap(NULL, 128, ...);
    ```



Initialisierung

- Speicher vom Betriebssystem anfordern:
  - durch statische Allokation zur Ladezeit:
    ```
    char memory[128];
    ```
  - durch Systemaufruf zur Laufzeit:
    ```
    char *memory = mmap(NULL, 128, ...);
    ```
- `struct mblock` reinlegen:

  ```
  struct mblock *head = (struct mblock *)memory;
  ```

Initialisierung

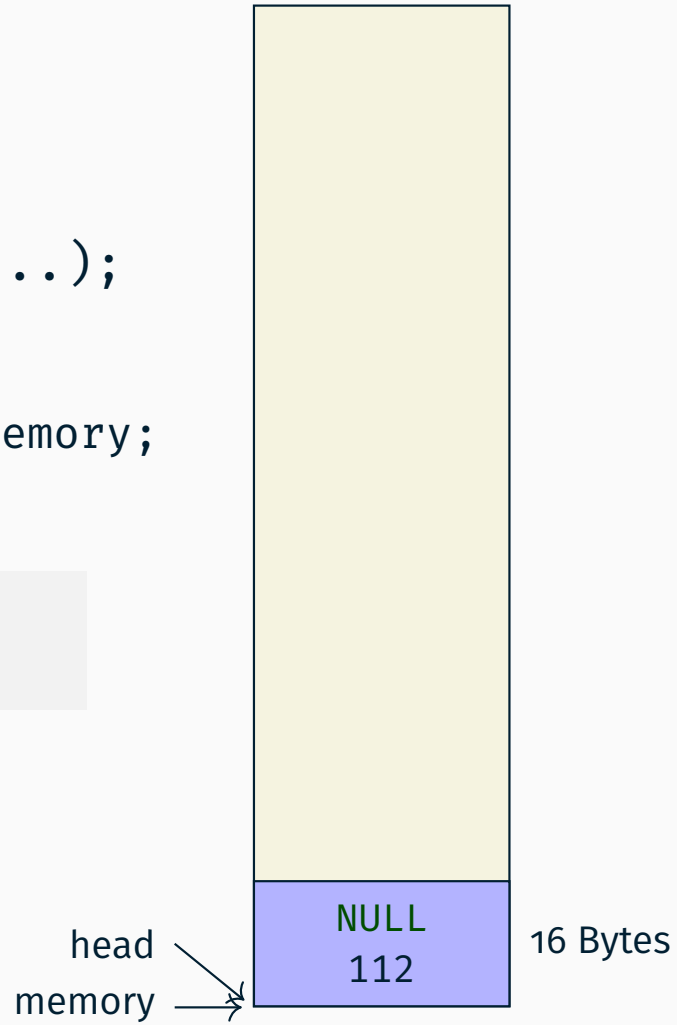
- Speicher vom Betriebssystem anfordern:
  - durch statische Allokation zur Ladezeit:
    `char memory[128];`
  - durch Systemaufruf zur Laufzeit:
    `char *memory = mmap(NULL, 128, ...);`
- `struct mblock` reinlegen:

```
struct mblock *head = (struct mblock *)memory;
```

- `struct mblock` initialisieren:

```
head->next = NULL;
head->size = 112;
```

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

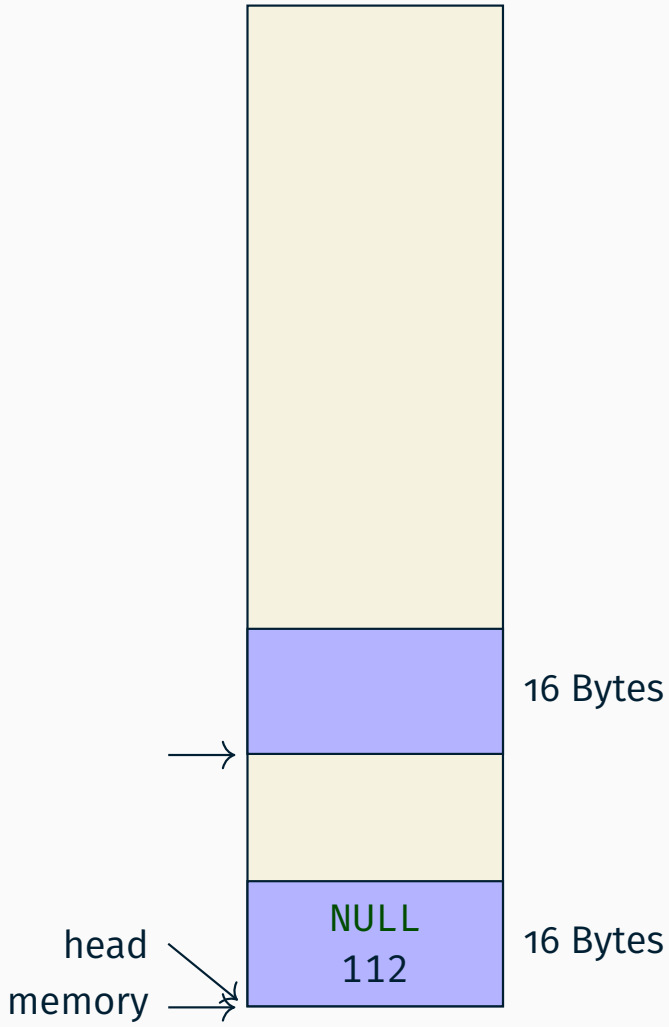


Speicheranforderung im Detail

- Speicheranforderung von 16 Bytes

```
char *m1 = malloc(16);
```

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

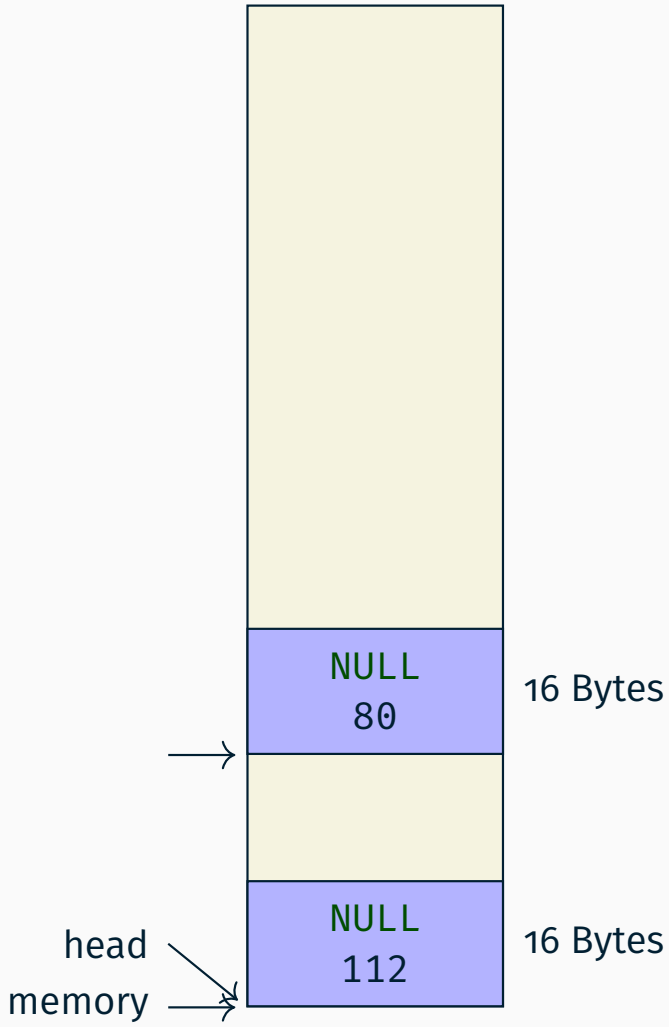


Speicheranforderung im Detail

- Speicheranforderung von 16 Bytes

```
char *m1 = malloc(16);
```

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

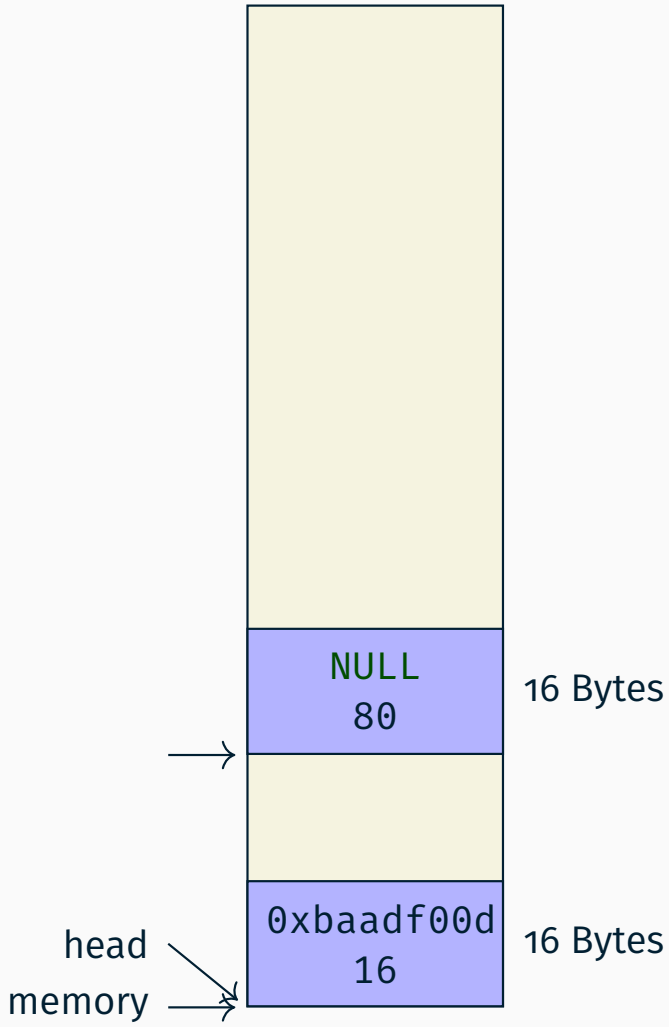


Speicheranforderung im Detail

- Speicheranforderung von 16 Bytes

```
char *m1 = malloc(16);
```

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

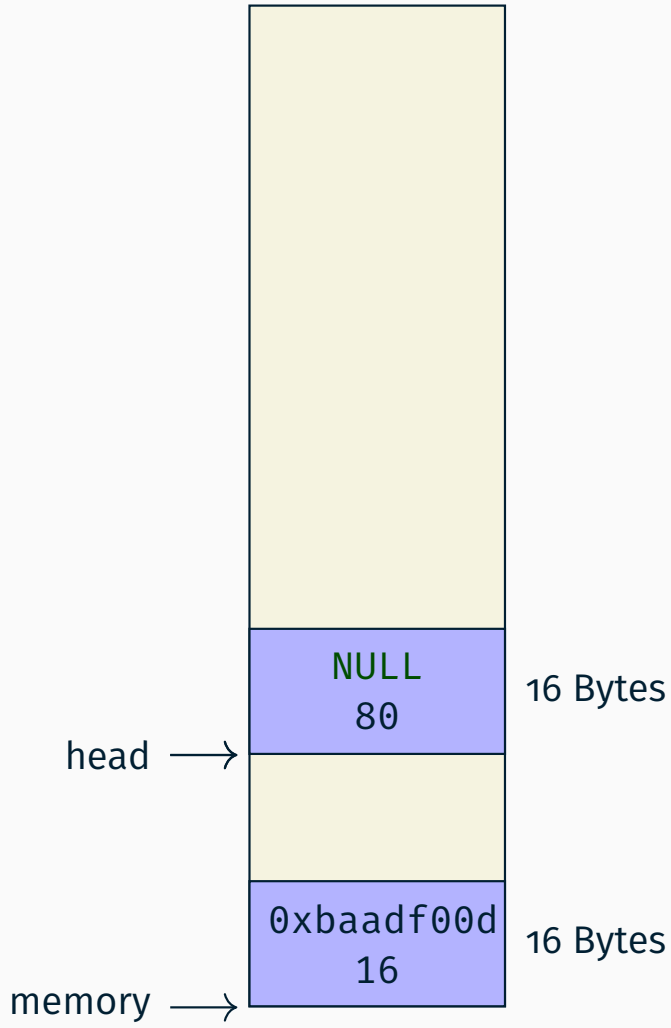


Speicheranforderung im Detail

- Speicheranforderung von 16 Bytes

```
char *m1 = malloc(16);
```

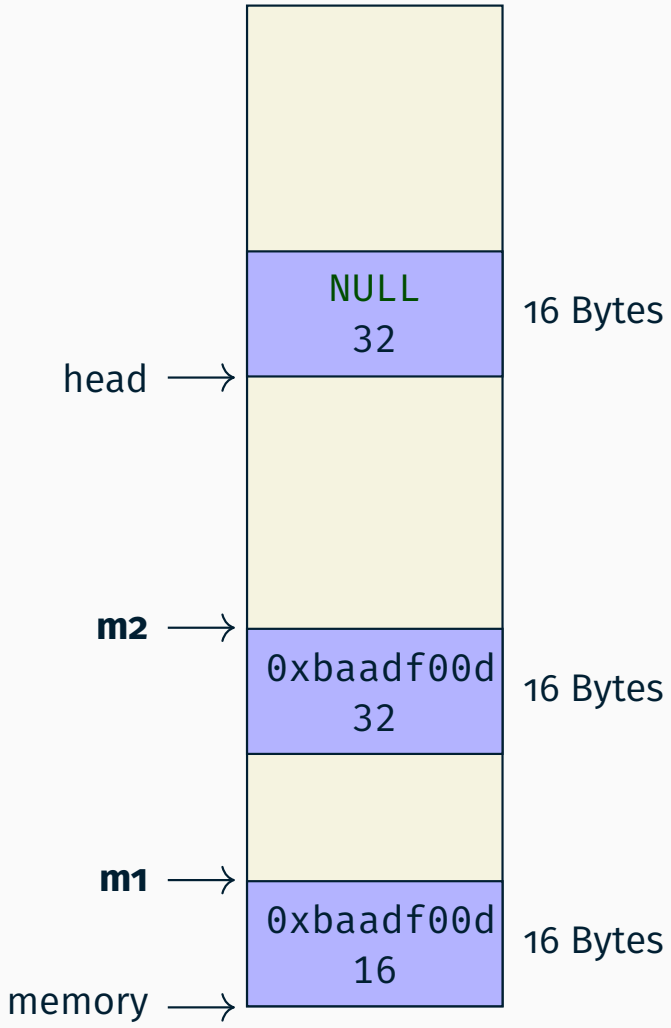
  - Freispeicherliste nach mblock mit ausreichend Speicher durchsuchen
  - 16 Bytes hinter dem head-mblock einen neuen mblock anlegen
  - ...und initialisieren
  - Bisherigen `head-mblock` anpassen
    - als belegt markieren
    - Größe des Speicherbereichs aktualisieren
  - head-Zeiger auf neues Kopfelement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



Speicheranforderung im Detail

- Situation nach 2 `malloc()`-Aufrufen

```
char *m1 = malloc(16);
char *m2 = malloc(32);
```

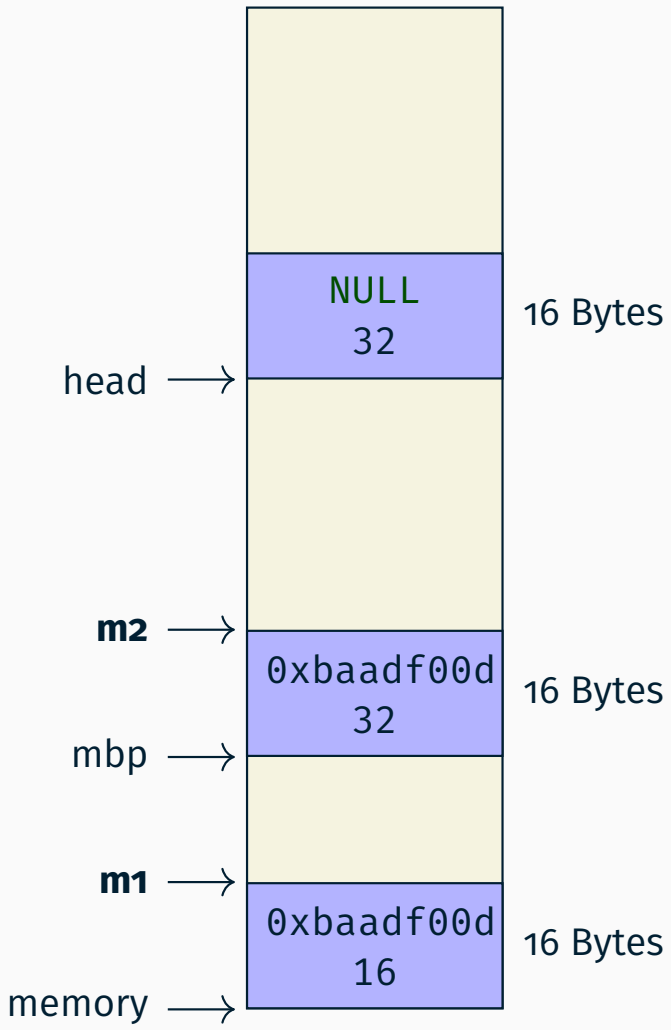


Speicherfreigabe

- Freigabe von `m2`

```
free(m2);
```

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

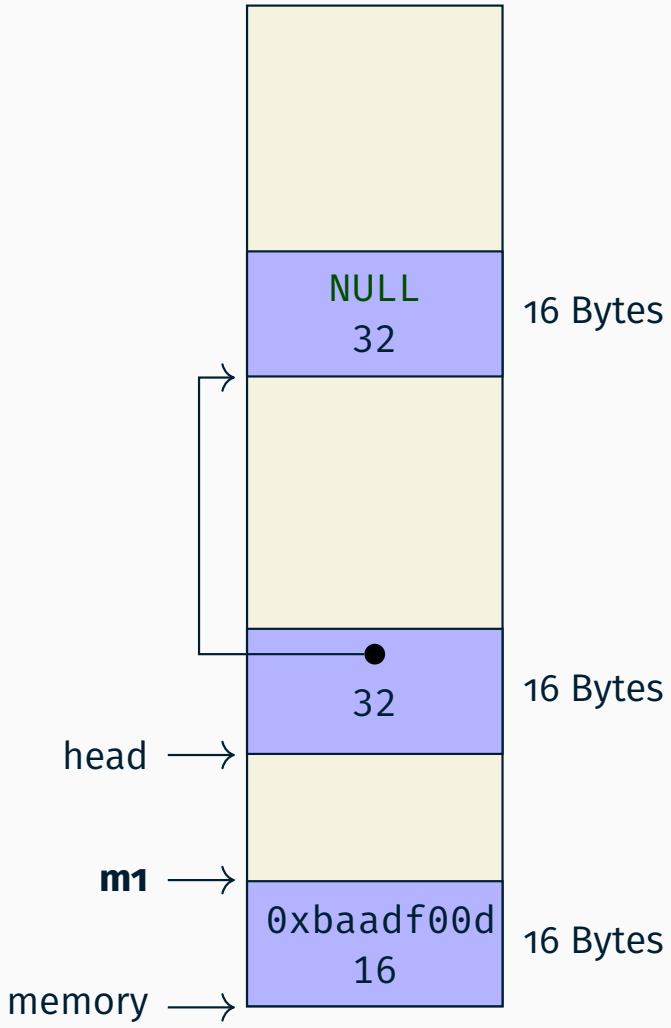


Speicherfreigabe

- 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



Systemaufruf zum Nachfordern von Speicher

- 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

Systemaufruf zum Nachfordern von Speicher

- 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);
```

Systemaufruf zum Nachfordern von Speicher

- 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 V.2/20

Systemaufruf zum Nachfordern von Speicher

- 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 V.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

Zusammenfassung

- 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)

Agenda

Ziele der Aufgabe

- Ziele der Aufgabe
  - Zusammenhang zwischen „nacktem Speicher” und typisierten Datenbereichen verstehen
  - Funktion aus der C-Bibliothek / POSIX selbst realisieren
    - Das Verhalten der zu implementierenden Funktionen soll dem in der POSIX-Manpage entsprechen
  - 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
  - Alignment wird von vorgegebener Funktion automatisch übernommen

Agenda

Aufgabenstellung

Skizzieren Sie den Aufbau des verwalteten Speicherbereichs (hier: 96 Bytes, `sizeof(struct mblock)` = 16 Bytes) nach jedem Schritt des jeweiligen Szenarios

1.

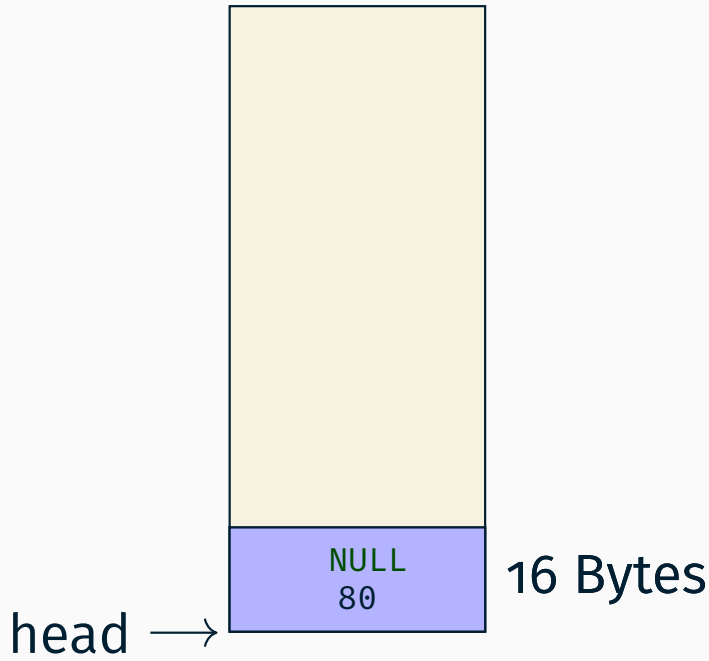
```
char *c1 = malloc(16);
char *c2 = malloc(16);
free(c1);
```

2.

```
char *c1 = malloc(32);
free(c1);
char *c2 = malloc(16);
```

3.

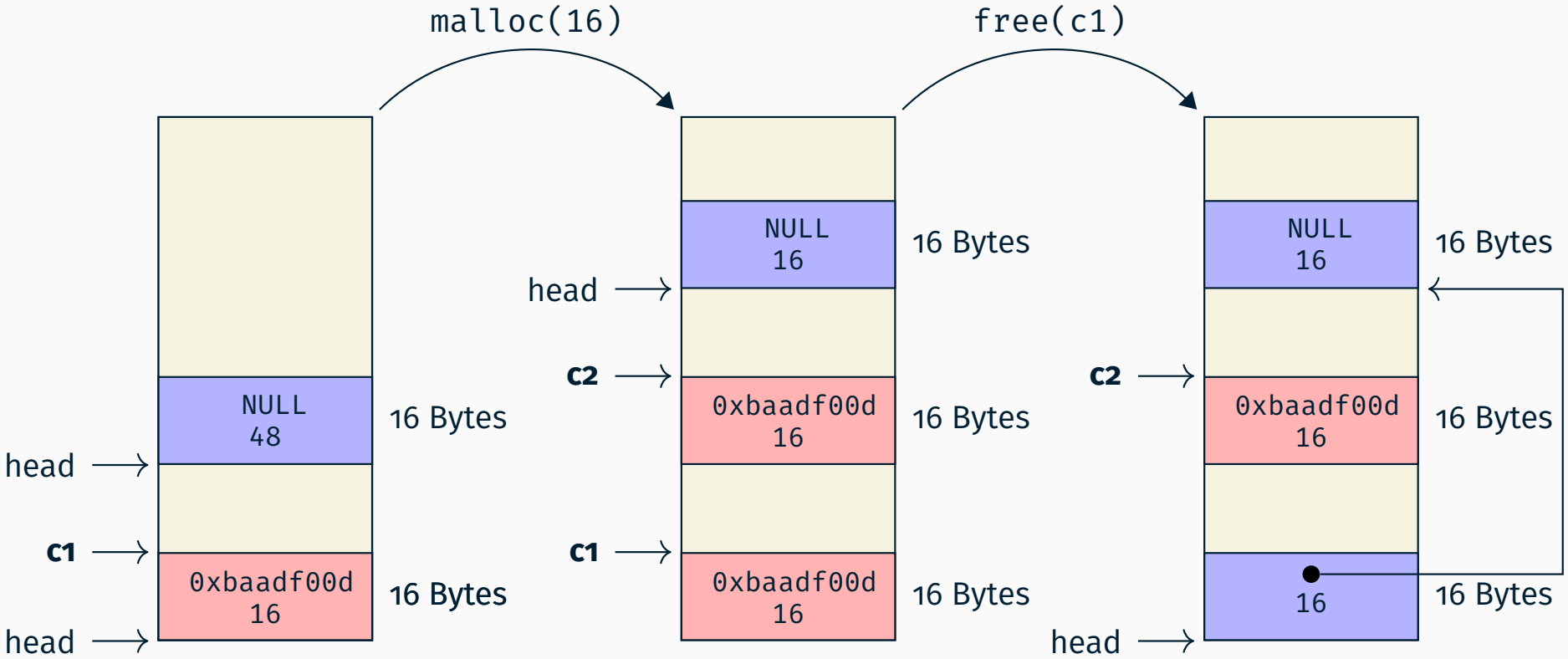
```
char *c1 = malloc(32);
char *c2 = malloc(32);
free(c1);
```



Lösung zu den Aufgaben

- Szenario 1:

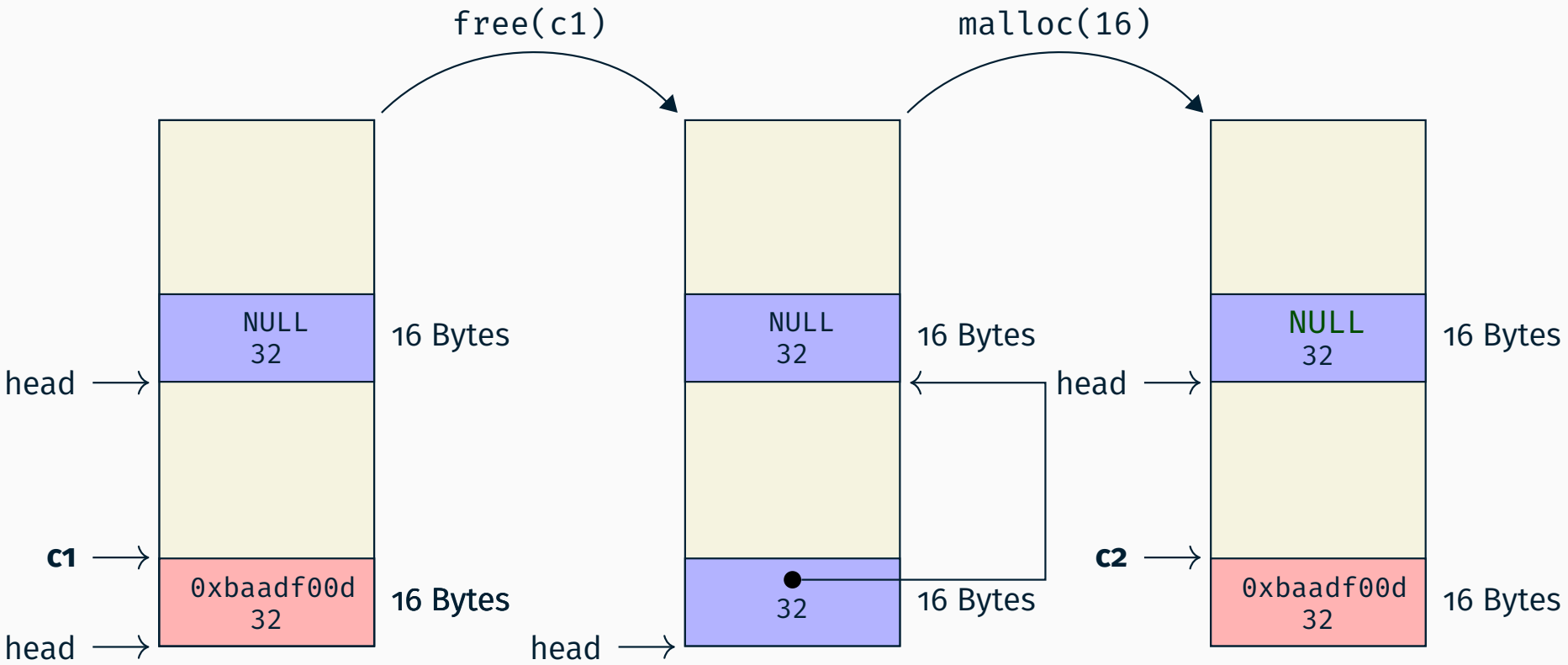
```
char *c1 = malloc(16);
char *c2 = malloc(16);
free(c1);
```



Lösung zu den Aufgaben

- Szenario 2:

```
char *c1 = malloc(32);
free(c1);
char *c2 = malloc(16);
```



Lösung zu den Aufgaben

- Szenario 3:

```
char *c1 = malloc(32);
char *c2 = malloc(32);
free(c1);
```