Technische Universität Braunschweig

Institute of Operating Systems and Computer Networks
Reliable System Software

Übung Betriebssysteme (BS)

Tafelübung 2: Zeiger und Speicher

Sören Tempel

Wintersemester 2025

Preamble

Technische Universität Braunschweig

- Gestern war die Deadline für Übungsblatt #1
- Heute wird Übungsblatt #2 veröffentlicht
- Außerdem:
  - Veröffentlichung der Musterlösung für Übungsblatt #1
  - Wahl der Code-Interviewtermine bis Sonntag 23:59 Uhr
- Ab Montag finden dann die ersten Code-Interviews statt

Wdh. Zeitlicher Ablauf

Bearbeitungsphase

- ~2 Wochen
- Programmieraufgaben

Anmeldung

1. Aufforderung (per Mail)
2. Termin wählen (StudIP)

Interview-Phase

- ~2 Wochen
- Einzel-Interviews

Überblick: Tafelübung 2: Zeiger und Speicher

Tafelübung 2: Zeiger und Speicher

- Übungsblatt 2
- Freispeicherverwaltung
- Debugging

Übungsblatt 2

Dynamische Speicherverwaltung

- Beispiel in Java:

```
WaitingHuman somebody = new WaitingHuman("Mrs. Somebody");
WaitingHuman nobody = new WaitingHuman("Mr. Nobody");

somebody.add(nobody);
```

  - In Java: Neues Listenelement wird mit Hilfe von `new` instanziiert
    - Reservieren eines Speicherbereiches für das Objekt
    - Initialisieren des Objektes durch Ausführen des Konstruktors

Dynamische Speicherverwaltung

Technische Universität Braunschweig

- Beispiel in Java:

```
WaitingHuman somebody = new WaitingHuman("Mrs. Somebody");
WaitingHuman nobody = new WaitingHuman("Mr. Nobody");

somebody.add(nobody);
```

  - In Java: Neues Listenelement wird mit Hilfe von **new** instanziiert
    - Reservieren eines Speicherbereiches für das Objekt
    - Initialisieren des Objektes durch Ausführen des Konstruktors

- In C: Anlegen eines Listenelementes mittels malloc(3)

```
struct listelement *newElement;
newElement = malloc( sizeof(struct listelement) );
if( newElement == NULL ) {
        // Fehlerbehandlung
}
```

  - Zurückgegebener Speicher ist nicht initialisiert, hat undefinierten Wert
  - Inititialisierung muss per Hand erfolgen

Dynamische Speicherverwaltung

Technische Universität Braunschweig

- Explizite Initialisierung mit definiertem Wert: memset(3)

```
memset(newElement, 0, sizeof(struct listelement));
```

Dynamische Speicherverwaltung

Technische Universität Braunschweig

- Explizite Initialisierung mit definiertem Wert: memset(3)

```
memset(newElement, 0, sizeof(struct listelement));
```

- Mit `0` vorinitialisierter Speicher kann mit calloc(3) angefordert werden

```
struct listelement *newElement;
newElement = calloc( 1, sizeof(struct listelement) );
if ( newElement == NULL ) { /* Fehler */ }
```

Dynamische Speicherverwaltung

- Explizite Initialisierung mit definiertem Wert: memset(3)

```
memset(newElement, 0, sizeof(struct listelement));
```

- Mit `0` vorinitialisierter Speicher kann mit calloc(3) angefordert werden

```
struct listelement *newElement;
newElement = calloc( 1, sizeof(struct listelement) );
if ( newElement == NULL ) { /* Fehler */ }
```

- Keinen Garbage-Collection-Mechanismus
  - Speicherbereich muss von Hand mittels free(3) freigegeben werden
  - Nur Speicher, der mit einer der Funktionen malloc(3), calloc(3) oder realloc(3) angefordert wurde, darf mit free(3) freigegeben werden!
- Achtung:
  - Zugriff auf freigegebenen Speicherbereich ist undefiniert
  - Mehrfach Freigabe von Speicherbereichen ist ebenfalls undefined Behavior

Freispeicherverwaltung

Freispeicherverwaltung

Technische Universität Braunschweig

↪ Wir bauen uns unsere eigene Speicherverwaltung

02t-speicher 2025-11-07

Freispeicherverwaltung

Technische Universität Braunschweig

↪ Wir bauen uns unsere eigene Speicherverwaltung

- Notwendige Überlegungen:
  - Woher bekommt `malloc` den Speicher?
  - Wohin bringt `free` den Speicher?
  - Wie funktioniert die Speicherverwaltung?

Freispeicherverwaltung: Interface

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

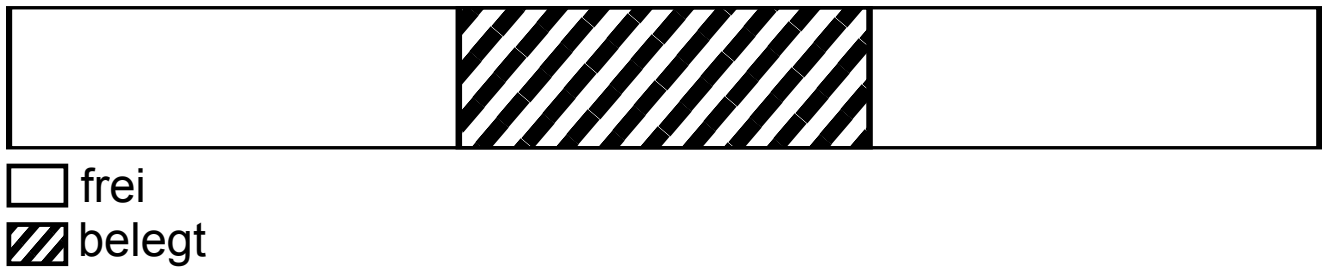
Anforderungsanalyse

Technische Universität Braunschweig

- Ziel: Speicherbereiche erhalten, die zur Laufzeit in beliebiger Größe angefordert werden können

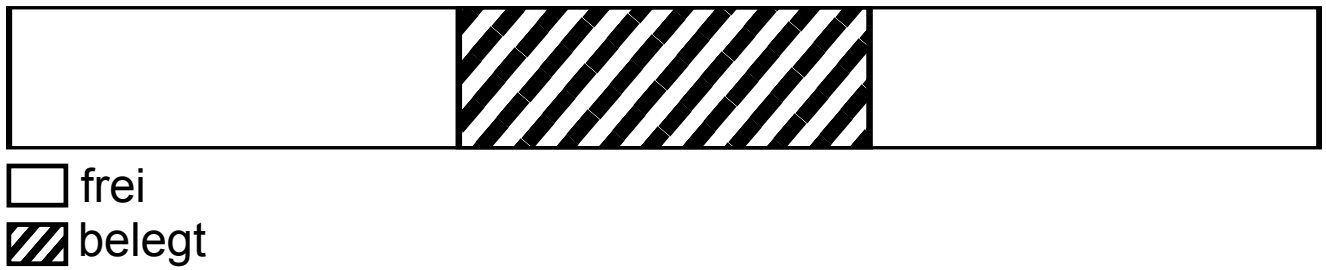
Anforderungsanalyse

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



Anforderungsanalyse

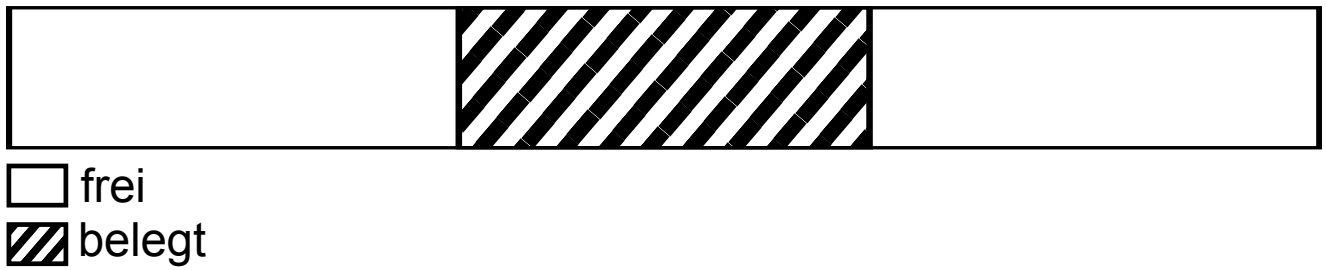
- Ziel: Speicherbereiche erhalten, die zur Laufzeit in beliebiger Größe angefordert werden können
- Skizze: Zustand eines teilweise belegten Heaps/Halde



  - Welche Informationen muss eine Freispeicherverwaltung bereit halten?

Anforderungsanalyse

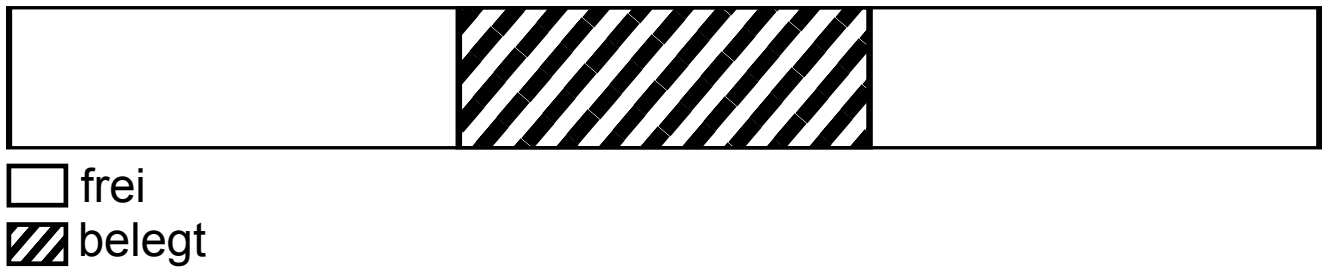
- Ziel: Speicherbereiche erhalten, die zur Laufzeit in beliebiger Größe angefordert werden können
- Skizze: Zustand eines teilweise belegten Heaps/Halde



  - Welche Informationen muss eine Freispeicherverwaltung bereit halten?
    - für freie Blöcke: Größe und Lage des Speicherbereichs
    - für belegte Blöcke: Größe des Speicherbereichs

Anforderungsanalyse

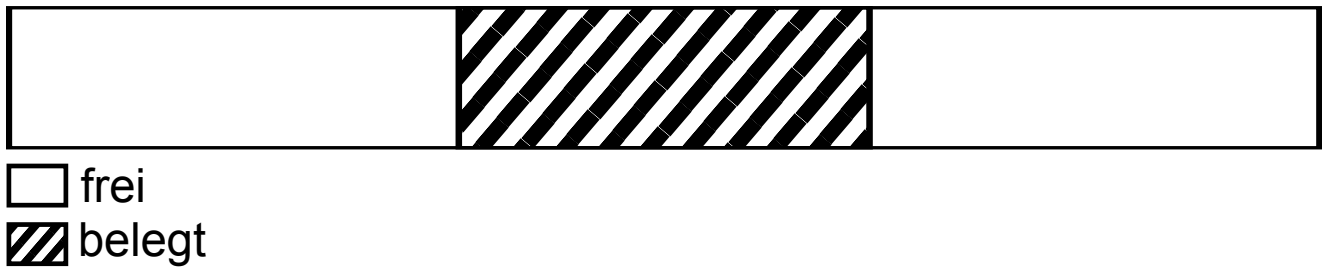
- Ziel: Speicherbereiche erhalten, die zur Laufzeit in beliebiger Größe angefordert werden können
- Skizze: Zustand eines teilweise belegten Heaps/Halde



  - 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?

Anforderungsanalyse

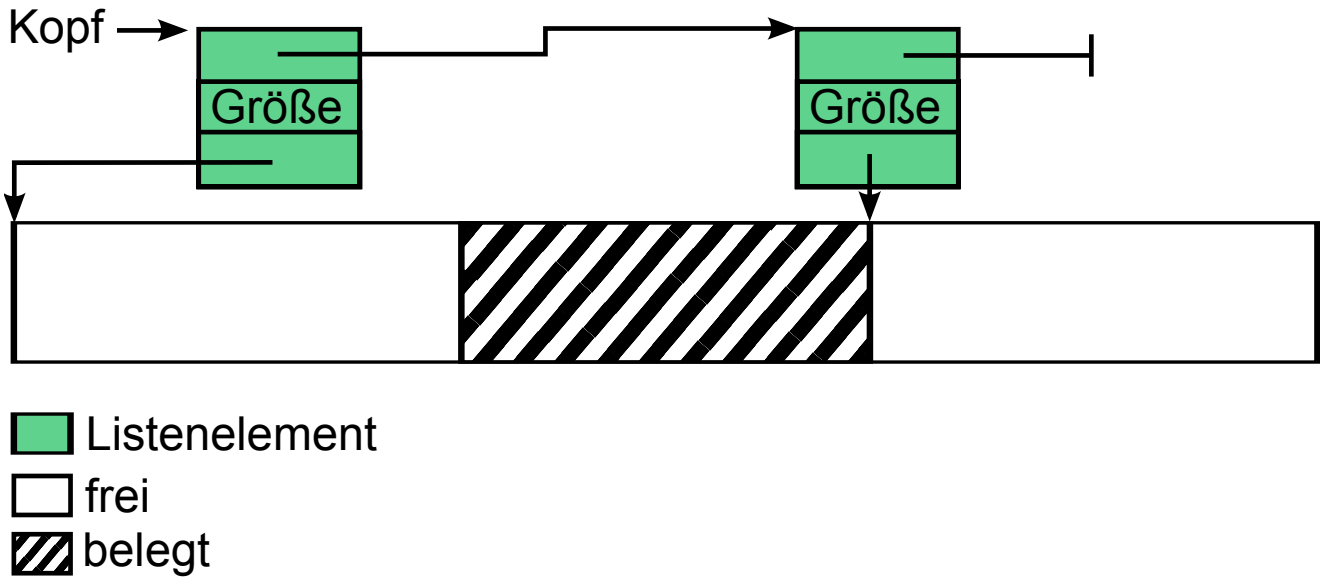
- Ziel: Speicherbereiche erhalten, die zur Laufzeit in beliebiger Größe angefordert werden können
- Skizze: Zustand eines teilweise belegten Heaps/Halde



  - 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: Verkettete Liste zur Allokation

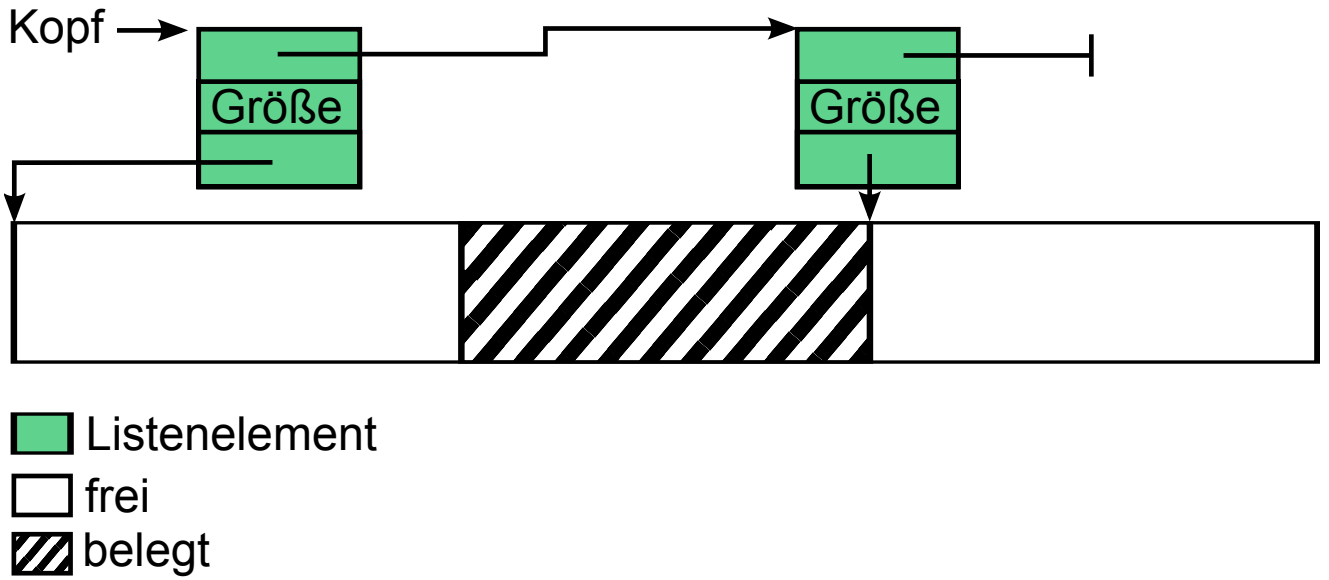
- Konzept einer Freispeicherverwaltung auf Basis einer verketteten Liste (ohne Berücksichtigung der belegten Blöcke!)



  - Freie Blöcke werden in einer verketteten Liste gespeichert

Konzept: Verkettete Liste zur Allokation

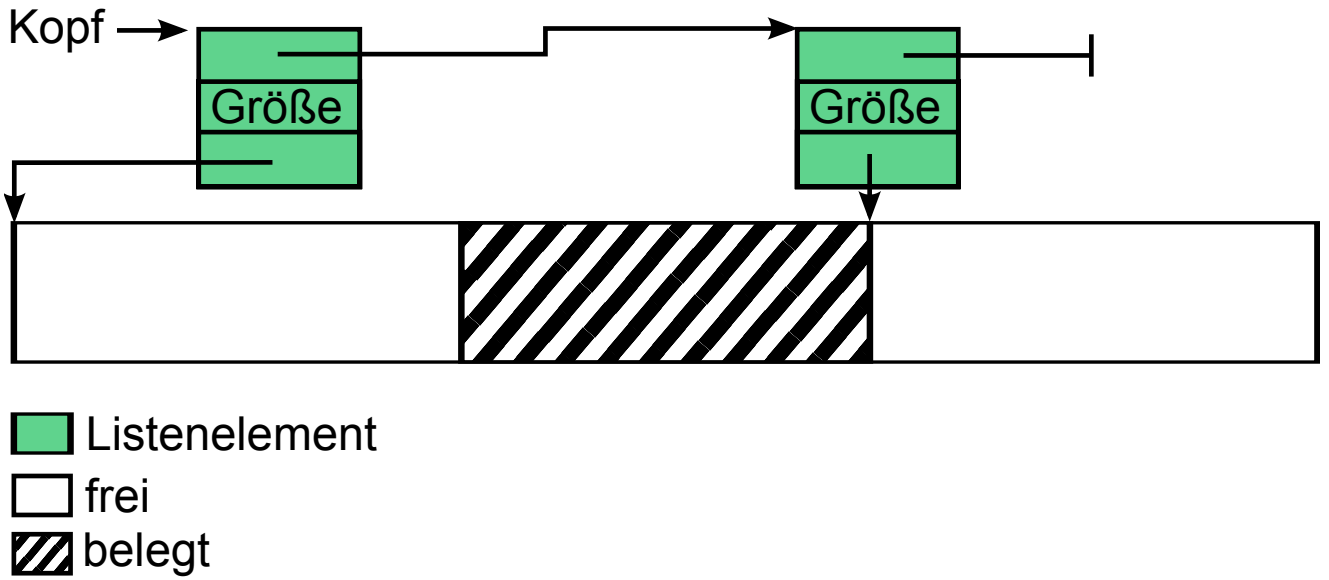
- Konzept einer Freispeicherverwaltung auf Basis einer verketteten Liste (ohne Berücksichtigung der belegten Blöcke!)



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

Konzept: Verkettete Liste zur Allokation

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

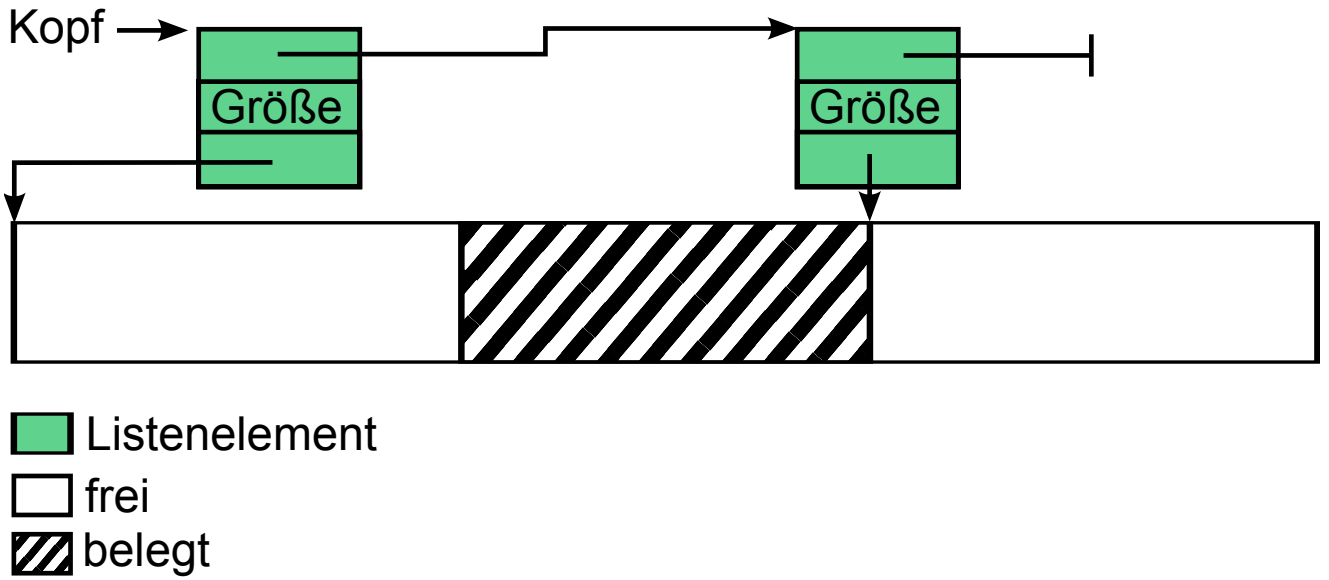


  - Freie Blöcke werden in einer verketteten Liste gespeichert
- Wie wird eine verkettete Liste in C implementiert?

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

Konzept: Verkettete Liste zur Allokation

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



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

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

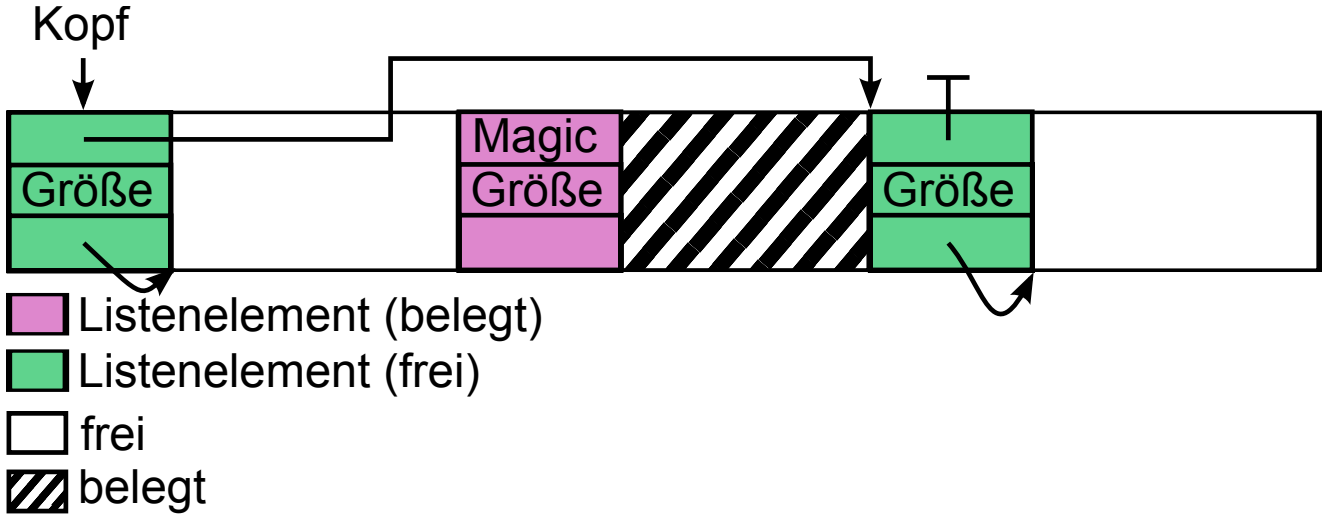
Speicher für die Listenelemente

Technische Universität Braunschweig

- Woher den Speicher für die Listenelemente nehmen?

Speicher für die Listenelemente

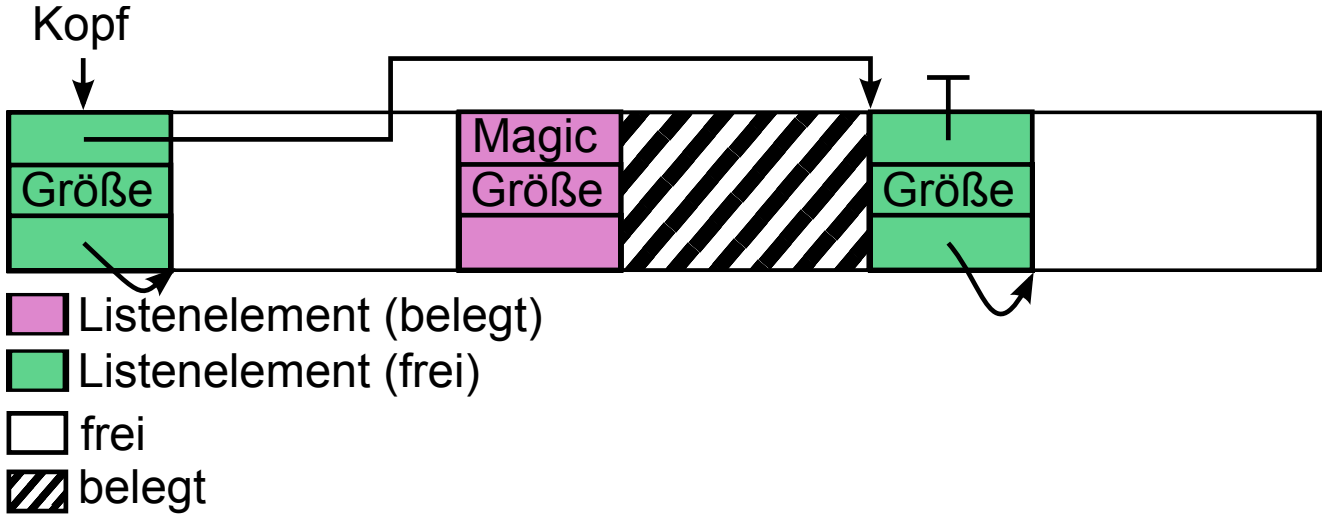
- Woher den Speicher für die Listenelemente nehmen?



  - Listenelemente werden innerhalb des verwalteten Speichers am Anfang des jeweiligen Speicherbereichs abgelegt

Speicher für die Listenelemente

- 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

Implementierung

Technische Universität Braunschweig

- 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 *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 ⇒ zu beachten bei `sizeof(struct mblock)`

Beispiel auf den Folien

Technische Universität Braunschweig

- Schrittweises Abarbeiten des folgenden Codestückes:

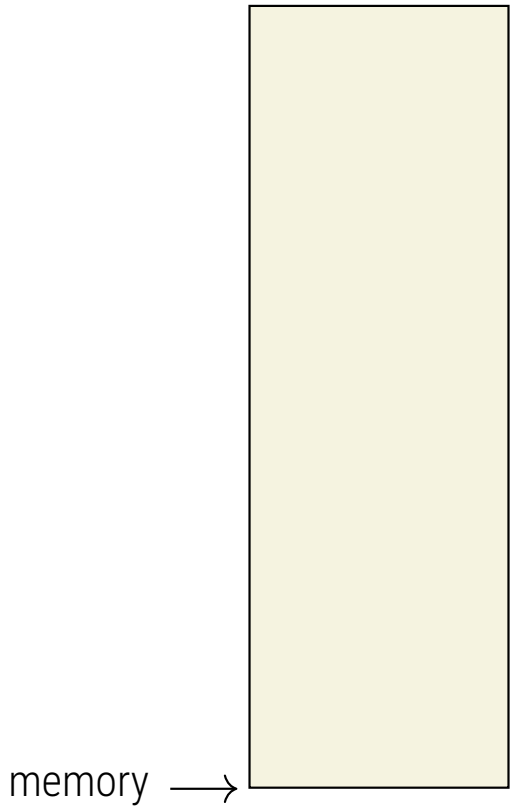
```
char *m1 = malloc(10);
char *m2 = malloc(20);

free(m2);
```

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

Initialisierung

- Speicher statisch alloziert

```
static char memory[100];
```



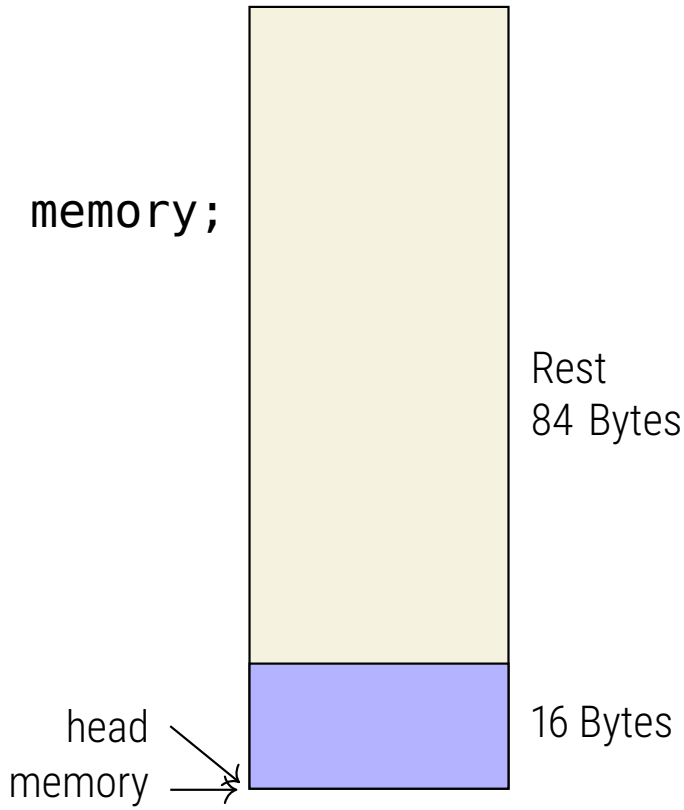
Initialisierung

- Speicher statisch alloziert

```
static char memory[100];
```

- `struct mblock` reinlegen

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



Initialisierung

- Speicher statisch alloziert

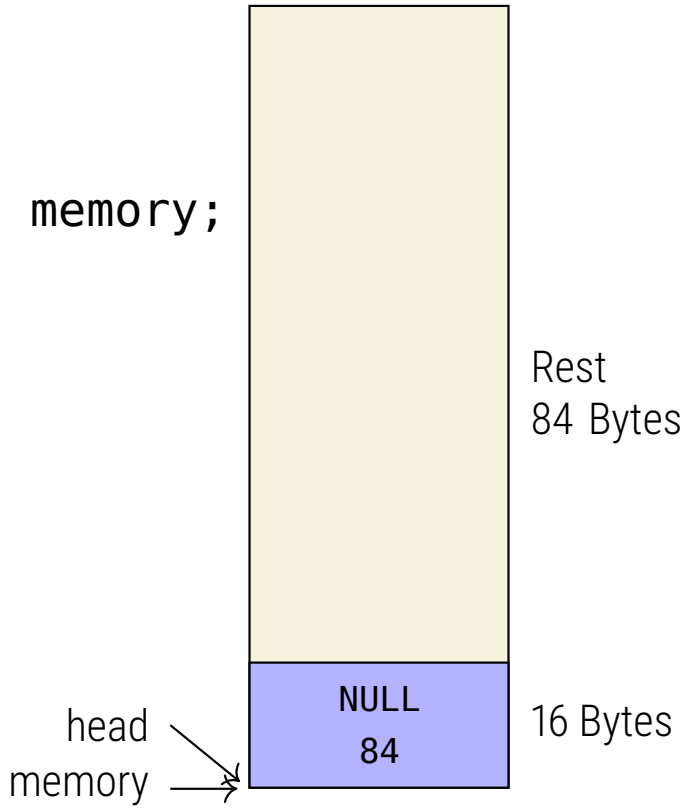
```
static char memory[100];
```

- `struct mblock` reinlegen

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

- `struct mblock` initialisieren

```
head->next = NULL;
head->size = 84; // 100 - 16 = 84
```



Initialisierung

- Speicher statisch alloziert

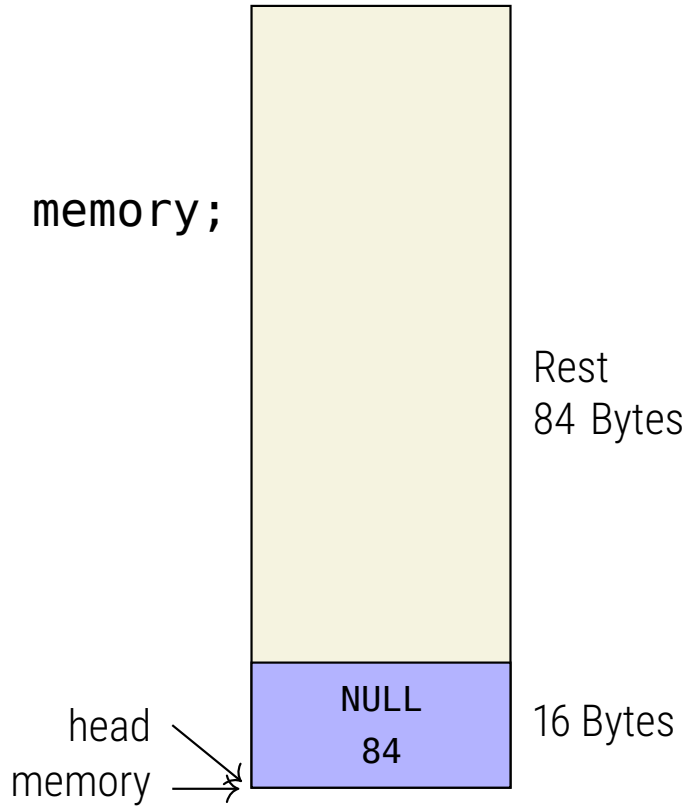
```
static char memory[100];
```

- `struct mblock` reinlegen

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

- `struct mblock` initialisieren

```
head->next = NULL;
head->size = 84; // 100 - 16 = 84
```



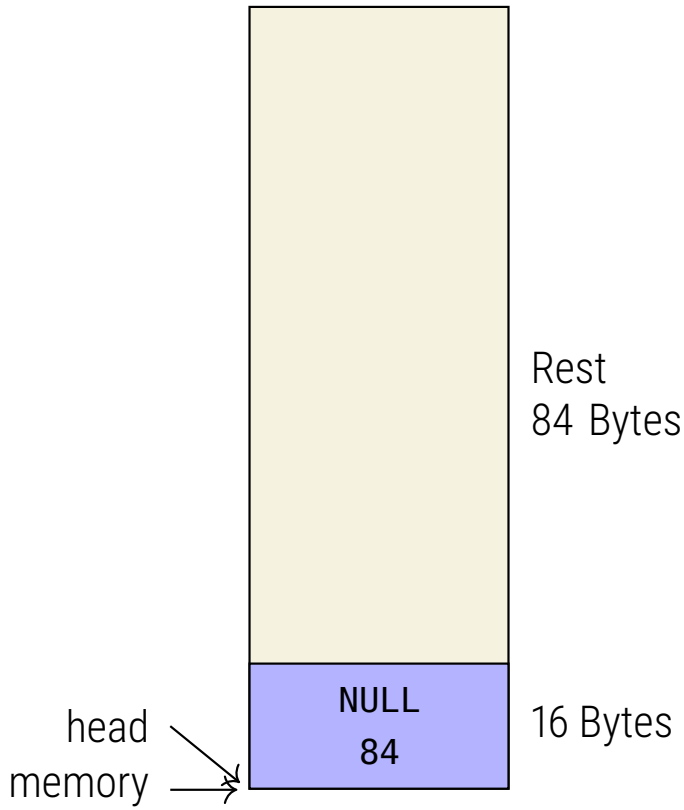
! zwei Zeiger mit unterschiedlichem Typ auf den gleichen Speicherbereich

- `char *` (`memory`) und `struct mblock*` (`head`)
- unterschiedliche Semantik beim Zugriff (Zeigerarithmetik, Strukturkomponenten)

Speicheranforderung im Detail

- Speicheranforderung von 10 Bytes

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

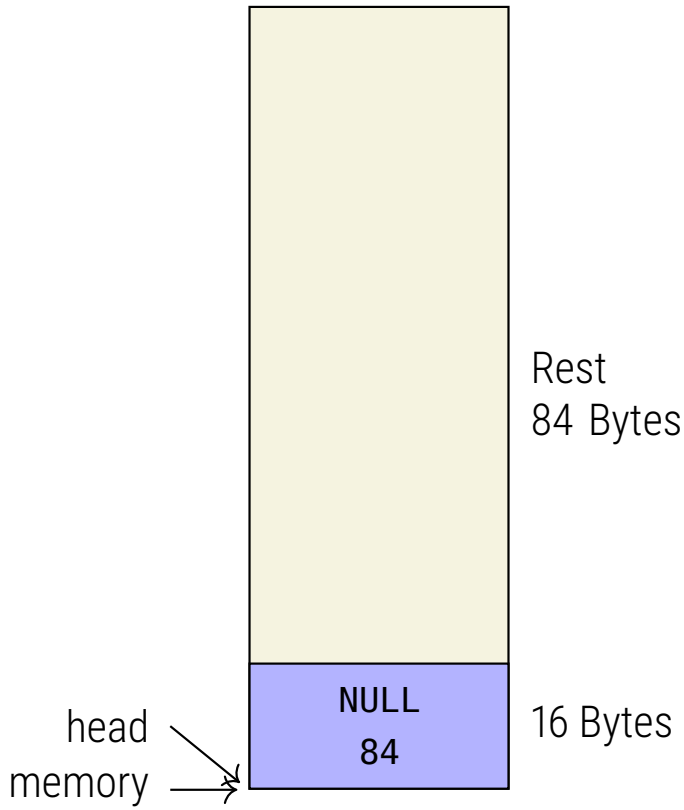


Speicheranforderung im Detail

- Speicheranforderung von 10 Bytes

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

1. Freispeicherliste nach mblock mit ausreichend Speicher durchsuchen

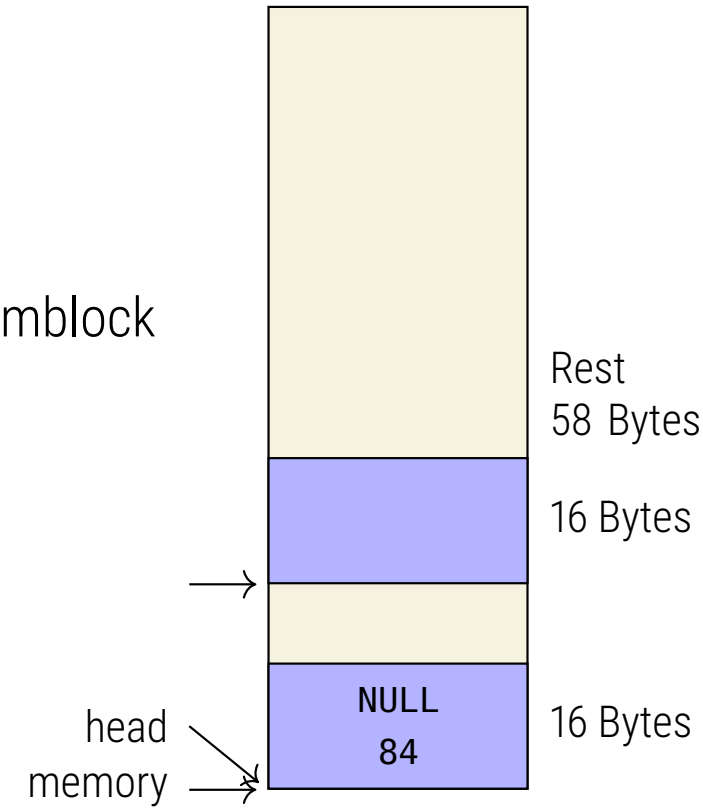


Speicheranforderung im Detail

- Speicheranforderung von 10 Bytes

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

1. Freispeicherliste nach mblock mit ausreichend Speicher durchsuchen
2. 10 Bytes hinter dem **head**-mblock einen neuen mblock anlegen

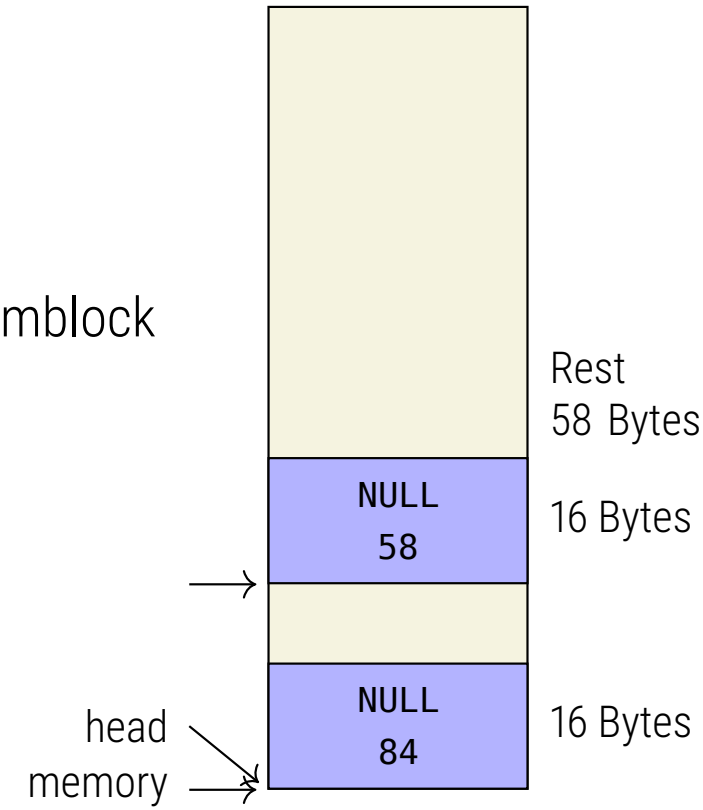


Speicheranforderung im Detail

- Speicheranforderung von 10 Bytes

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

1. Freispeicherliste nach mblock mit ausreichend Speicher durchsuchen
2. 10 Bytes hinter dem **head**-mblock einen neuen mblock anlegen
3. ... und initialisieren

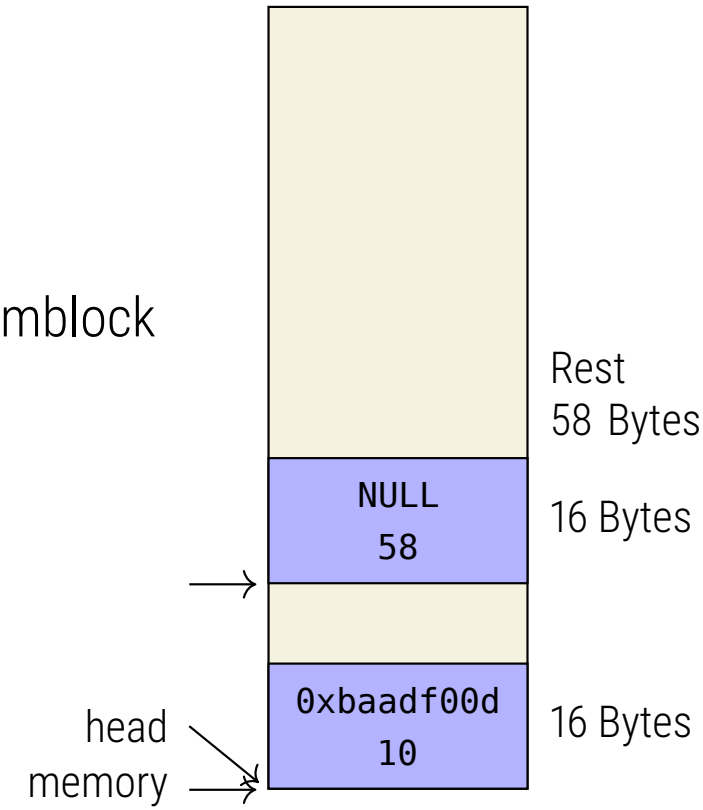


Speicheranforderung im Detail

- Speicheranforderung von 10 Bytes

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

1. Freispeicherliste nach mblock mit ausreichend Speicher durchsuchen
2. 10 Bytes hinter dem **`head`**-mblock einen neuen mblock anlegen
3. ... und initialisieren
4. Bisherigen **`head-mblock`** anpassen
   - als belegt markieren (**`0xbaadf00d`**)
   - Größe des Speicherbereichs aktualisieren

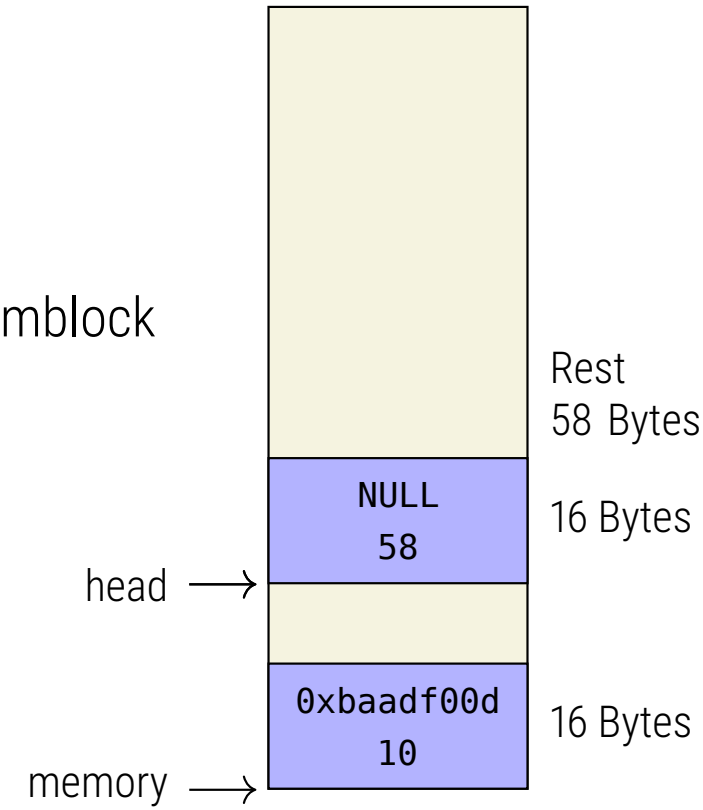


Speicheranforderung im Detail

- Speicheranforderung von 10 Bytes

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

1. Freispeicherliste nach mblock mit ausreichend Speicher durchsuchen
2. 10 Bytes hinter dem **head**-mblock einen neuen mblock anlegen
3. ... und initialisieren
4. Bisherigen **head-mblock** anpassen
   - als belegt markieren (**0xbaadf00d**)
   - Größe des Speicherbereichs aktualisieren
5. **head**-Zeiger auf neues Kopfelement setzen

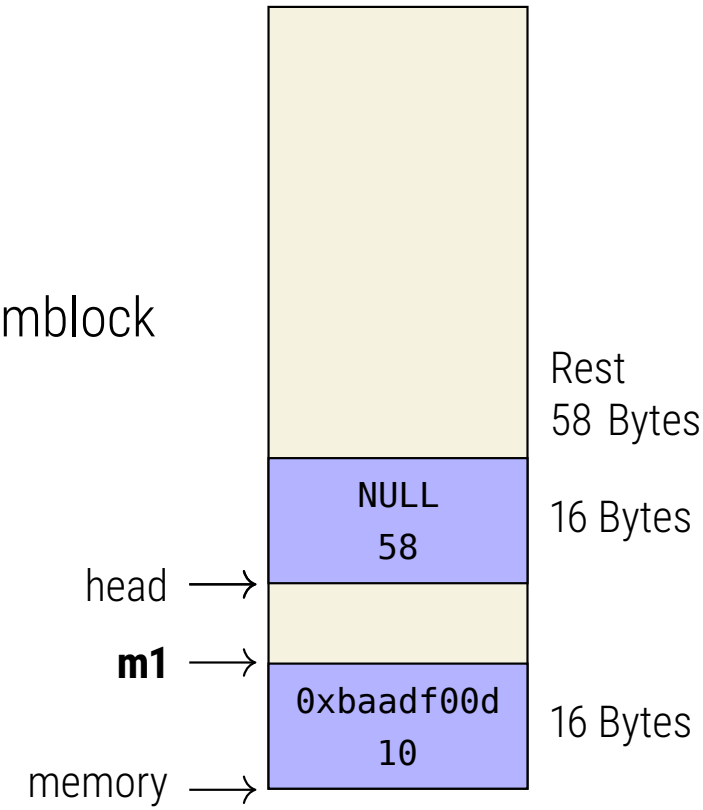


Speicheranforderung im Detail

- Speicheranforderung von 10 Bytes

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

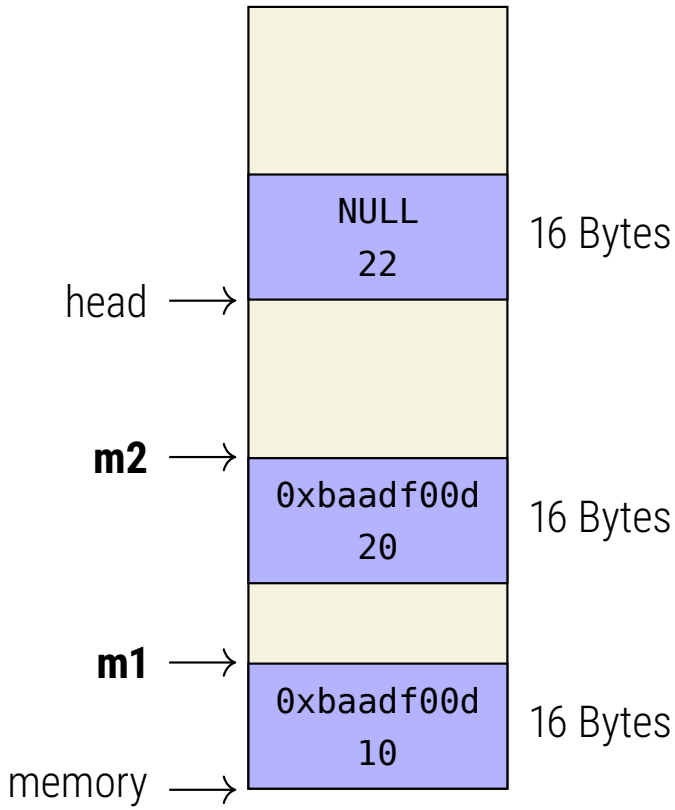
1. Freispeicherliste nach mblock mit ausreichend Speicher durchsuchen
2. 10 Bytes hinter dem **head**-mblock einen neuen mblock anlegen
3. ... und initialisieren
4. Bisherigen **head-mblock** anpassen
   - als belegt markieren (**0xbaadf00d**)
   - Größe des Speicherbereichs aktualisieren
5. **head**-Zeiger auf neues Kopfelement setzen
6. Zeiger auf die reservierten 10 Bytes zurückgeben (**m1**)



Speicheranforderung im Detail

- Situation nach 2 `malloc()`-Aufrufen

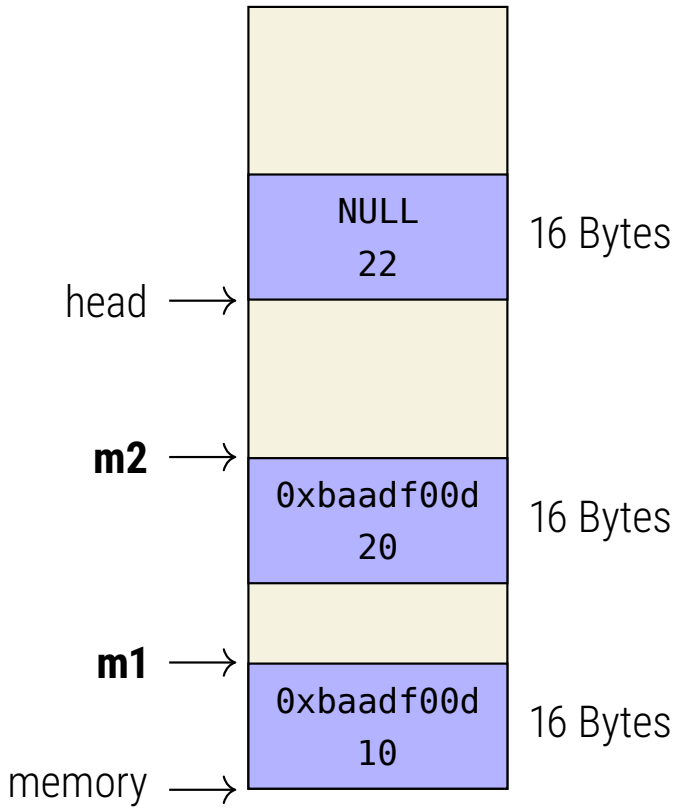
```
char* m1 = malloc(10);
char* m2 = malloc(20);
```



Speicherfreigabe

- Freigabe von `m2`

```
free(m2);
```

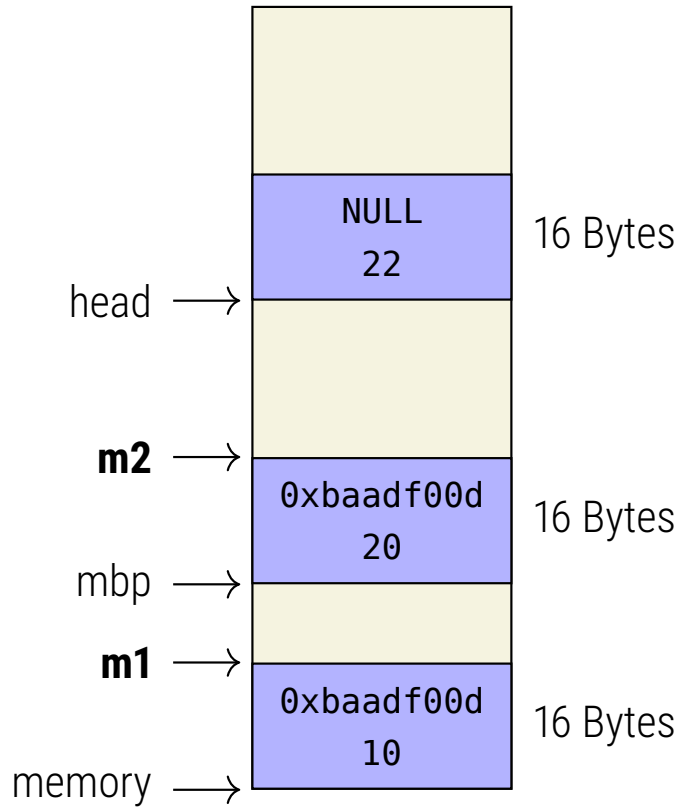


Speicherfreigabe

- Freigabe von `m2`

```
free(m2);
```

  - Zeiger `mbp` auf zugehörigen `mblock` ermitteln

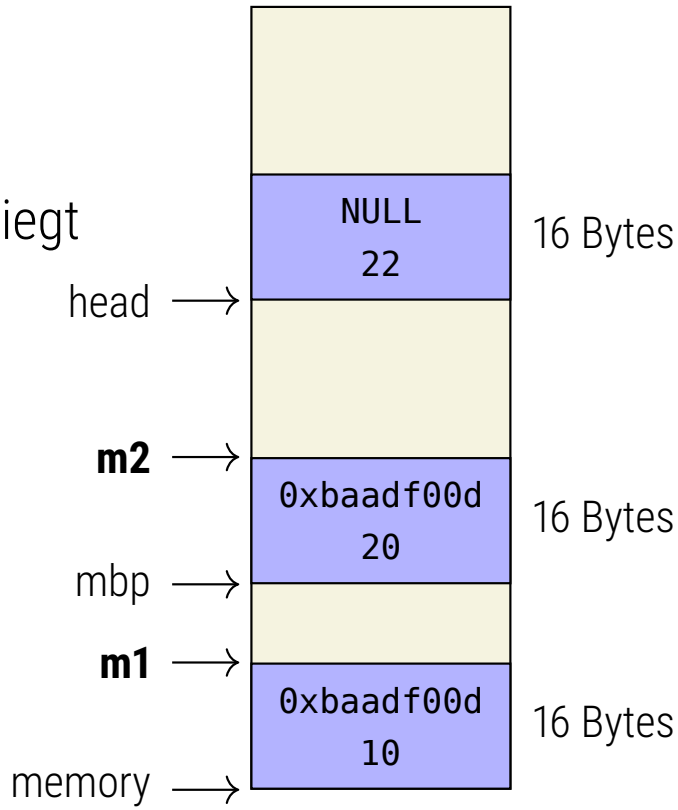


Speicherfreigabe

- Freigabe von `m2`

```
free(m2);
```

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

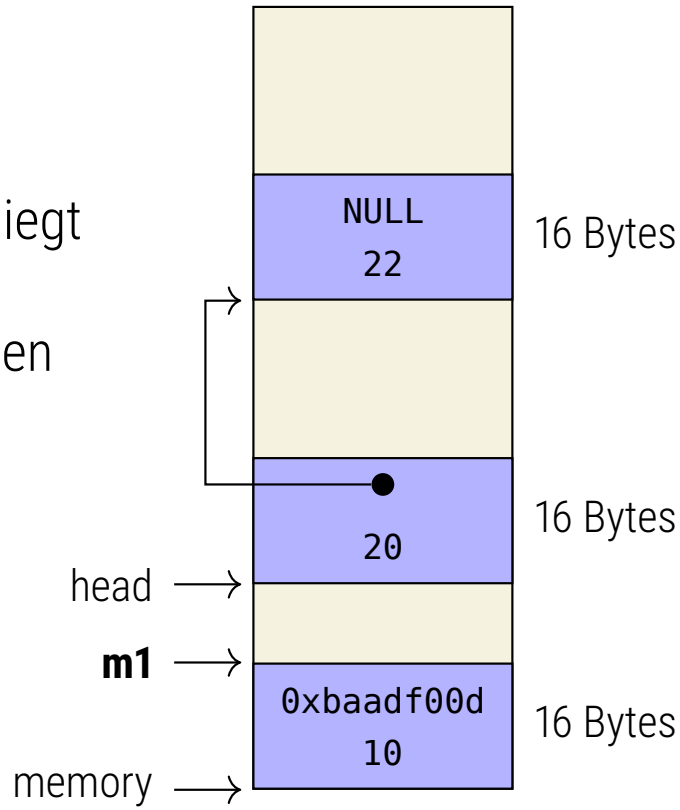


Speicherfreigabe

- 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

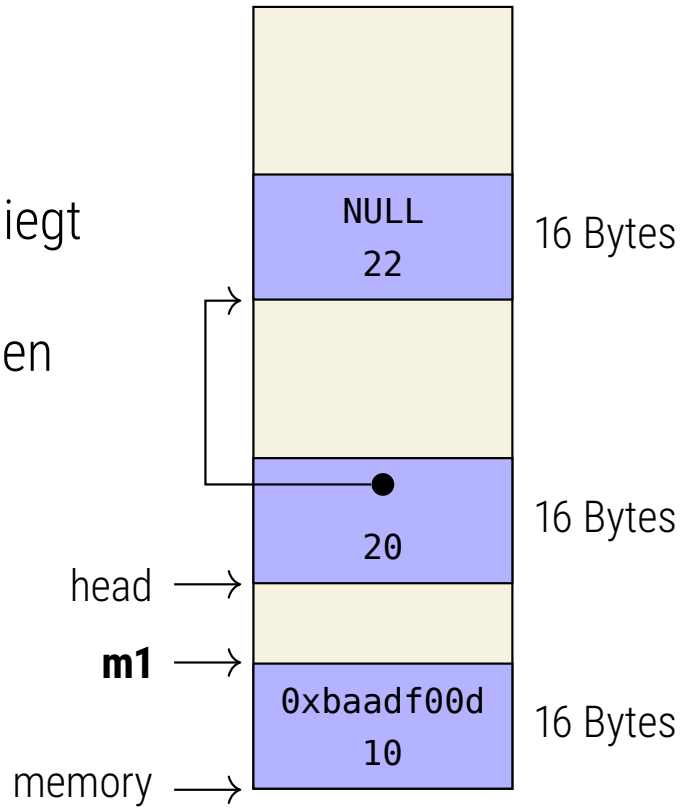


Speicherfreigabe

- 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
- Problem:
  - Speicher wird *fragmentiert*
    - Suchzeit für Lücken steigt
    - Freie Blöcke werden immer kleiner



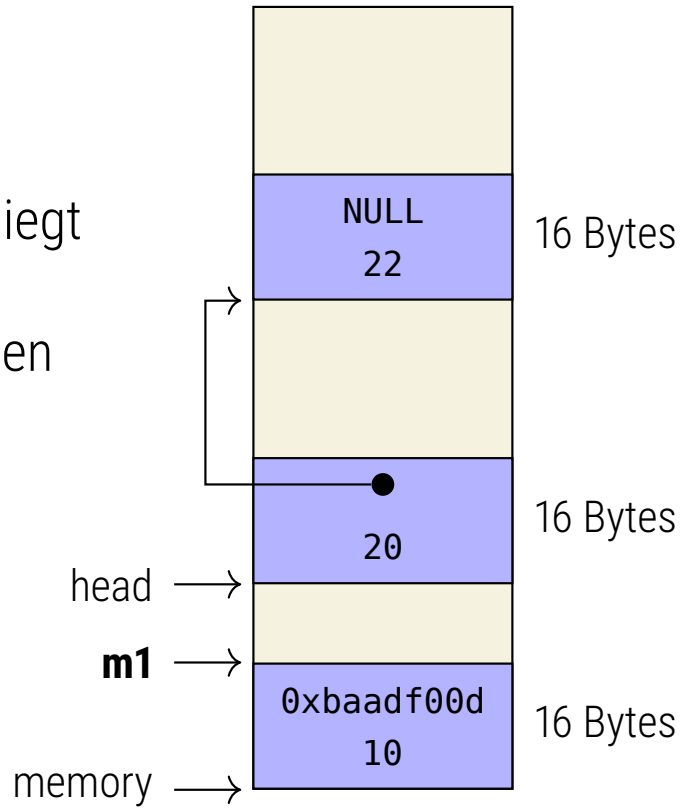
Speicherfreigabe

- 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
- Problem:
  - Speicher wird *fragmentiert*
    - Suchzeit für Lücken steigt
    - Freie Blöcke werden immer kleiner

↪ Lösung: Verschmelzung benachbarter Freispeicherblöcke

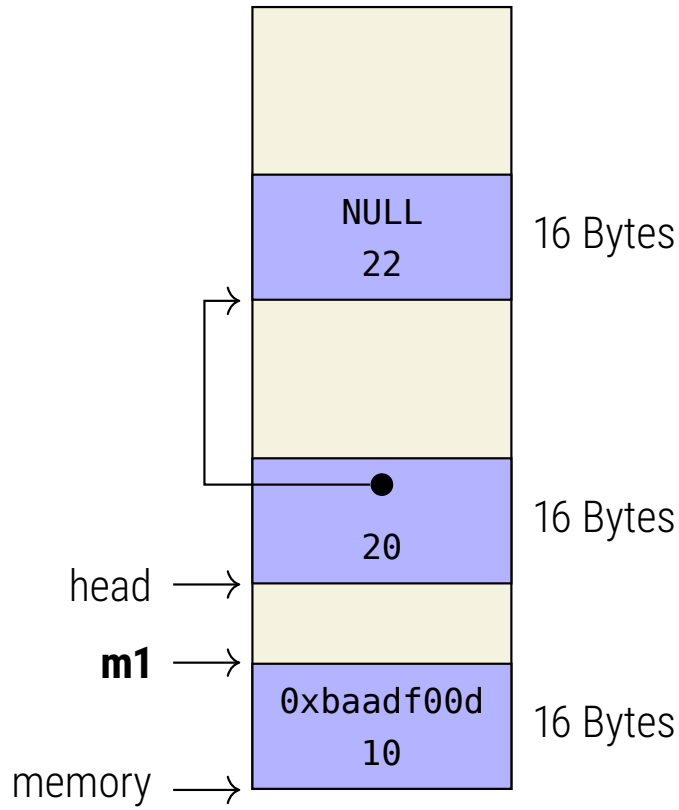


Speicherverschmelzung

- Freigabe von `m2` mit Verschmelzen:

```
free(m2);
```

- Blöcke verschmelzen:

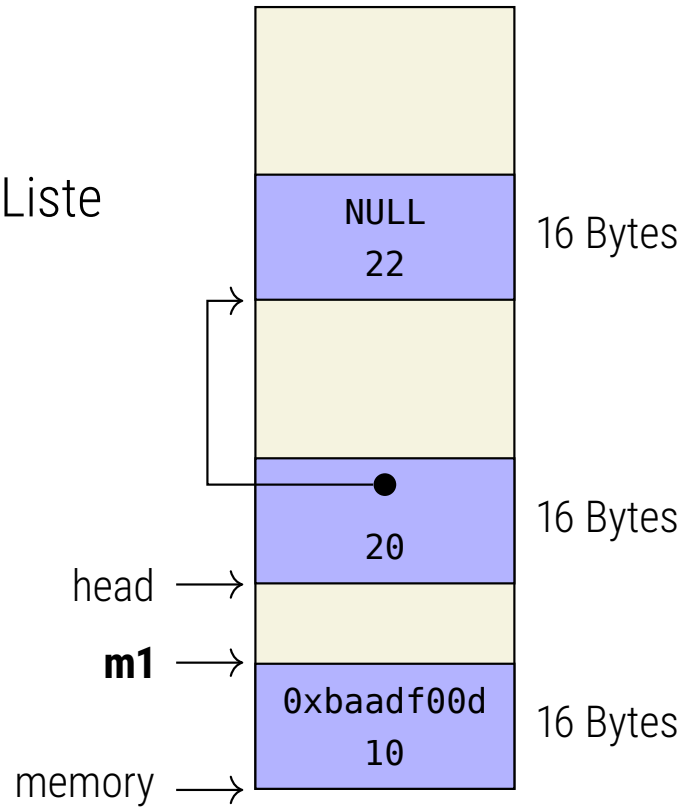


Speicherverschmelzung

- Freigabe von `m2` mit Verschmelzen:

```
free(m2);
```

- Blöcke verschmelzen:
  - Ausgebend vom vorherigen (oder ersten) Block Liste durchlaufen.

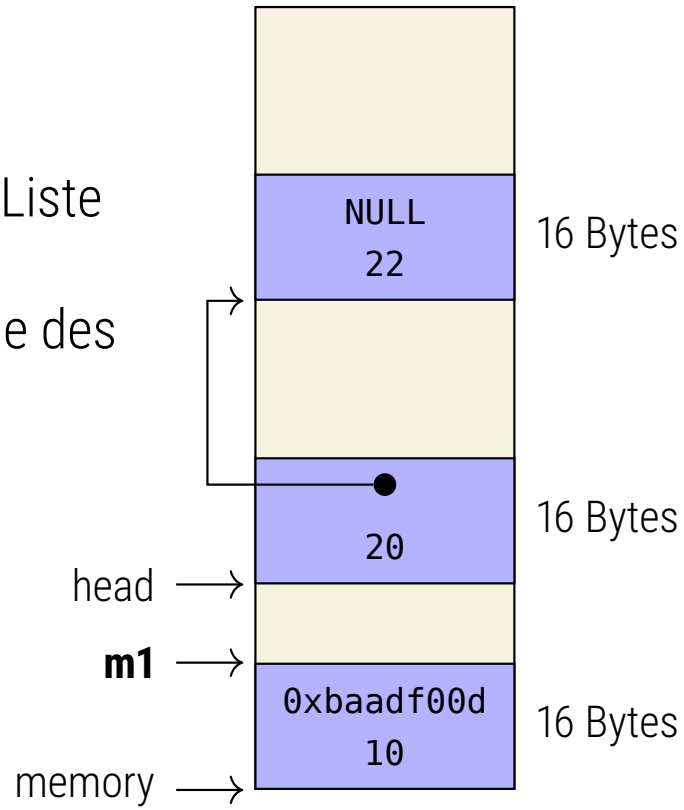


Speicherverschmelzung

- Freigabe von `m2` mit Verschmelzen:

```
free(m2);
```

- Blöcke verschmelzen:
  - Ausgebend vom vorherigen (oder ersten) Block Liste durchlaufen.
  - Überprüfen, ob der nächste Block direkt am Ende des Betrachteten anliegt.

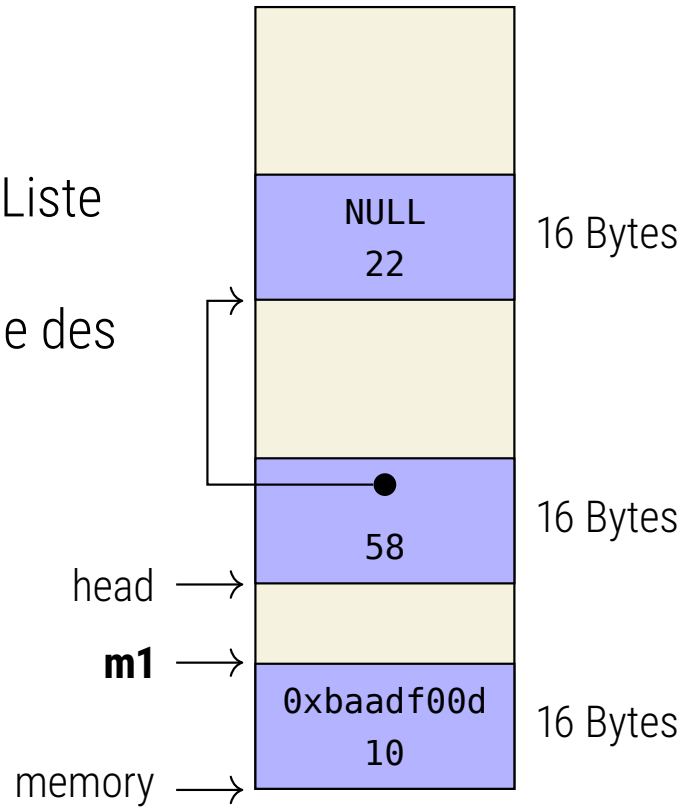


Speicherverschmelzung

- Freigabe von `m2` mit Verschmelzen:

```
free(m2);
```

- Blöcke verschmelzen:
  - Ausgebend vom vorherigen (oder ersten) Block Liste durchlaufen.
  - Überprüfen, ob der nächste Block direkt am Ende des Betrachteten anliegt.
  - Größe neu setzen: $20 + 22 + 16 = 58$

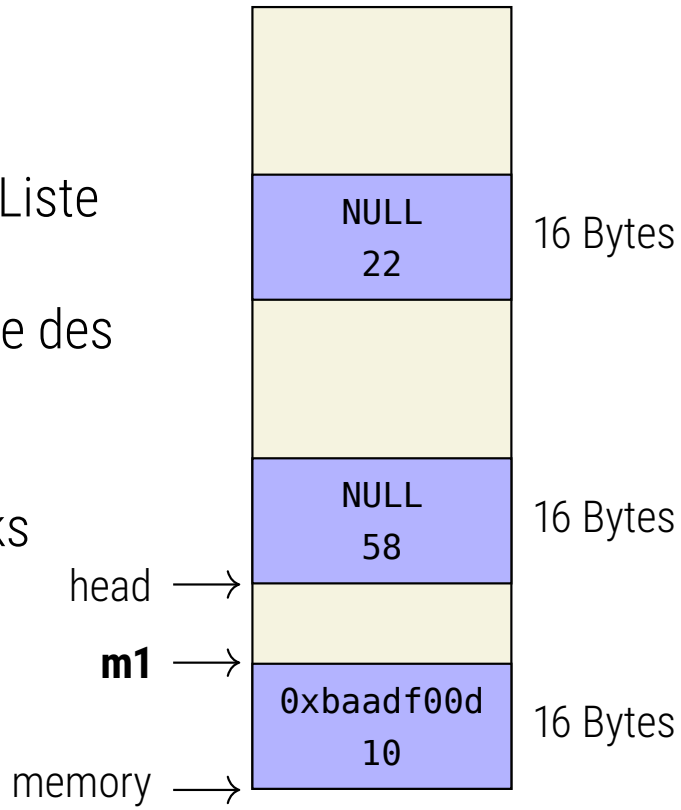


Speicherverschmelzung

- Freigabe von `m2` mit Verschmelzen:

```
free(m2);
```

- Blöcke verschmelzen:
  - Ausgebend vom vorherigen (oder ersten) Block Liste durchlaufen.
  - Überprüfen, ob der nächste Block direkt am Ende des Betrachteten anliegt.
  - Größe neu setzen: $20 + 22 + 16 = 58$
  - Zeiger auf Nachfolgerzeiger des nächsten Blocks setzen.

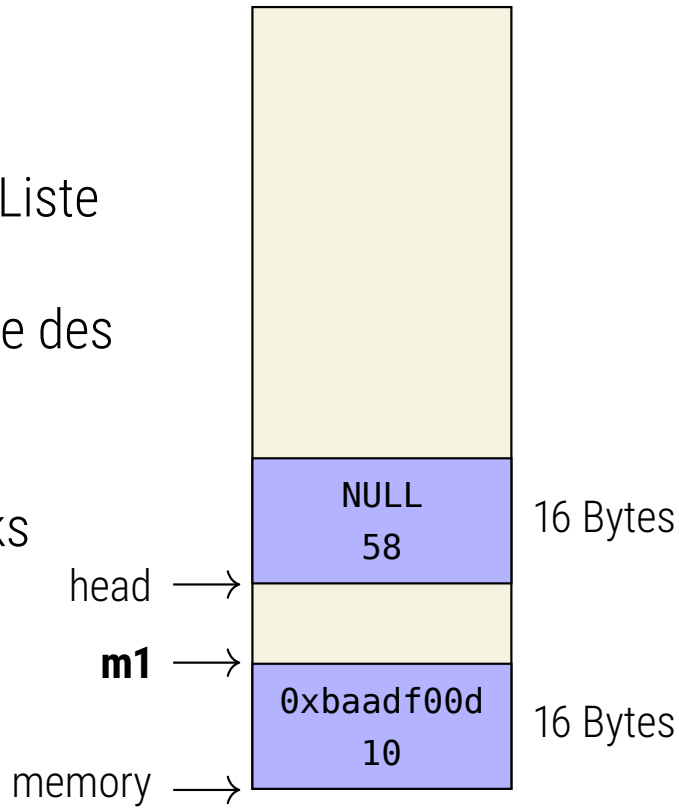


Speicherverschmelzung

- Freigabe von `m2` mit Verschmelzen:

```
free(m2);
```

- Blöcke verschmelzen:
  - Ausgebend vom vorherigen (oder ersten) Block Liste durchlaufen.
  - Überprüfen, ob der nächste Block direkt am Ende des Betrachteten anliegt.
  - Größe neu setzen: $20 + 22 + 16 = 58$
  - Zeiger auf Nachfolgerzeiger des nächsten Blocks setzen.



↪ Optional und nicht gefordert: Nullen des Speichers

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
- Kein nachträgliches Vergrößern des Heaps
  - In der Praxis: Speicherseiten vom Betriebssystem nachfordern
- Sinnvolle Implementierung erfordert geeignete Speichervergabestrategie
  - Implementierung erheblich aufwändiger – Resultat aber entsprechend effizienter
  - Mögliche Strategien wären z. B. First-Fit, Best-Fit, Worst-Fit oder Buddy-Verfahren

halde - Freispeicherverwaltung

Technische Universität Braunschweig

- Ziele der Aufgabe:
  - Besseres Verständnis von Zeigern aufbauen
  - Zusammenhang zwischen „nacktem Speicher” und typisierten Datenbereichen verstehen
  - Funktion aus der C-Bibliothek selbst realisieren

halde - Freispeicherverwaltung

Technische Universität Braunschweig

- Ziele der Aufgabe:
  - Besseres Verständnis von Zeigern aufbauen
  - Zusammenhang zwischen „nacktem Speicher” und typisierten Datenbereichen verstehen
  - Funktion aus der C-Bibliothek selbst realisieren
- Zu implementierende Schnittstelle:
  - `void *halde_malloc(size_t size);`
  - `void halde_free(void *ptr);`

halde - Freispeicherverwaltung

Technische Universität Braunschweig

- Ziele der Aufgabe:
  - Besseres Verständnis von Zeigern aufbauen
  - Zusammenhang zwischen „nacktem Speicher” und typisierten Datenbereichen verstehen
  - Funktion aus der C-Bibliothek selbst realisieren
- Zu implementierende Schnittstelle:
  - `void *halde_malloc(size_t size);`
  - `void halde_free(void *ptr);`
- Anforderungen:
  - First-Fit Allokationsstrategie
  - 1 MiB Speicher statisch alloziert
  - Freier Speicher wird in einer einfach verketteten Liste verwaltet

halde - Freispeicherverwaltung

- Ziele der Aufgabe:
  - Besseres Verständnis von Zeigern aufbauen
  - Zusammenhang zwischen „nacktem Speicher” und typisierten Datenbereichen verstehen
  - Funktion aus der C-Bibliothek selbst realisieren
- Zu implementierende Schnittstelle:
  - `void *halde_malloc(size_t size);`
  - `void halde_free(void *ptr);`
- Anforderungen:
  - First-Fit Allokationsstrategie
  - 1 MiB Speicher statisch alloziert
  - Freier Speicher wird in einer einfach verketteten Liste verwaltet
- Aufgabenteil A: `malloc`

halde - Freispeicherverwaltung

Technische Universität Braunschweig

- Ziele der Aufgabe:
  - Besseres Verständnis von Zeigern aufbauen
  - Zusammenhang zwischen „nacktem Speicher" und typisierten Datenbereichen verstehen
  - Funktion aus der C-Bibliothek selbst realisieren
- Zu implementierende Schnittstelle:
  - `void *halde_malloc(size_t size);`
  - `void halde_free(void *ptr);`
- Anforderungen:
  - First-Fit Allokationsstrategie
  - 1 MiB Speicher statisch alloziert
  - Freier Speicher wird in einer einfach verketteten Liste verwaltet
- Aufgabenteil A: `malloc`
- Aufgabenteil B: `free` ohne verschmelzen

02t-speicher 2025-11-07

halde - Freispeicherverwaltung

Technische Universität Braunschweig

- Ziele der Aufgabe:
  - Besseres Verständnis von Zeigern aufbauen
  - Zusammenhang zwischen „nacktem Speicher" und typisierten Datenbereichen verstehen
  - Funktion aus der C-Bibliothek selbst realisieren
- Zu implementierende Schnittstelle:
  - `void *halde_malloc(size_t size);`
  - `void halde_free(void *ptr);`
- Anforderungen:
  - First-Fit Allokationsstrategie
  - 1 MiB Speicher statisch alloziert
  - Freier Speicher wird in einer einfach verketteten Liste verwaltet
- Aufgabenteil A: `malloc`
- Aufgabenteil B: `free` ohne verschmelzen
- Aufgabenteil C: Verschmelzen freigegebener Blöcke

Debugging

Debugging mit dem GDB

Technische Universität Braunschweig

- Ein Debugger dient zum Suchen und Finden von Fehlern in Programmen

Debugging mit dem GDB

Technische Universität Braunschweig

- Ein Debugger dient zum Suchen und Finden von Fehlern in Programmen
- Im Debugger kann man unter anderem:
  - das Programm schrittweise abarbeiten
  - Variablen- und Speicherinhalte ansehen und modifizieren
  - core dumps (Speicherabbilder beim Programmabsturz) analysieren

Debugging mit dem GDB

Technische Universität Braunschweig

- Ein Debugger dient zum Suchen und Finden von Fehlern in Programmen
- Im Debugger kann man unter anderem:
  - das Programm schrittweise abarbeiten
  - Variablen- und Speicherinhalte ansehen und modifizieren
  - core dumps (Speicherabbilder beim Programmabsturz) analysieren
- Programm sollte Debug-Symbole enthalten
  - mit GCC-Flag `-g` übersetzen

Debugging mit dem GDB

Technische Universität Braunschweig

- Ein Debugger dient zum Suchen und Finden von Fehlern in Programmen
- Im Debugger kann man unter anderem:
  - das Programm schrittweise abarbeiten
  - Variablen- und Speicherinhalte ansehen und modifizieren
  - core dumps (Speicherabbilder beim Programmabsturz) analysieren
- Programm sollte Debug-Symbole enthalten
  - mit GCC-Flag `-g` übersetzen
- Aufruf des Basis-Debuggers mit `gdb <Programmname>`

Befehlsübersicht: Programmablauf

Technische Universität Braunschweig

- Starten des Programms mit `run` (+ evtl. Befehlszeilenparameter)
- Fortsetzen der Ausführung bis zum nächsten Stop mit `c` (continue)
- Verlassen der Funktion mit `finish`
- Beenden des GDB mit `quit`

Befehlsübersicht: Programmablauf

Technische Universität Braunschweig

- Starten des Programms mit `run` (+ evtl. Befehlszeilenparameter)
- Fortsetzen der Ausführung bis zum nächsten Stop mit `c` (continue)
- Verlassen der Funktion mit `finish`
- Beenden des GDB mit `quit`
- schrittweise Abarbeitung auf Ebene der Quellsprache mit
  - `s` (step: läuft in Funktionen hinein)
  - `n` (next: behandelt Funktionsaufrufe als einzelne Anweisung)

Befehlsübersicht: Programmablauf

Technische Universität Braunschweig

- Starten des Programms mit `run` (+ evtl. Befehlszeilenparameter)
- Fortsetzen der Ausführung bis zum nächsten Stop mit `c` (continue)
- Verlassen der Funktion mit `finish`
- Beenden des GDB mit `quit`
- schrittweise Abarbeitung auf Ebene der Quellsprache mit
  - `s` (step: läuft in Funktionen hinein)
  - `n` (next: behandelt Funktionsaufrufe als einzelne Anweisung)
- schrittweise Abarbeitung auf Ebene der Assembler Instruktionen mit
  - `si` (stepi: läuft in Funktionen hinein)
  - `ni` (nexti: behandelt Funktionsaufrufe als einzelne Anweisung)

Befehlsübersicht: Breakpoints

Technische Universität Braunschweig

- Breakpoints setzen:
  - `b [<Dateiname>:]<Funktionsname>`
  - `b <Dateiname>:<Zeilennummer>`
- Breakpoints anzeigen: `info breakpoints`
- Breakpoint löschen: `delete breakpoint#`

Befehlsübersicht: Breakpoints

Technische Universität Braunschweig

- Breakpoints setzen:
  - `b [<Dateiname>:]<Funktionsname>`
  - `b <Dateiname>:<Zeilennummer>`
- Breakpoints anzeigen: `info breakpoints`
- Breakpoint löschen: `delete breakpoint#`
- 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

02t-speicher 2025-11-07

Befehlsübersicht: Daten

Technische Universität Braunschweig

- 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>`

Befehlsübersicht: Daten

Technische Universität Braunschweig

- 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>`
- Quellcode an aktueller Position anzeigen: `list`

Befehlsübersicht: Daten

Technische Universität Braunschweig

- 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>`
- Quellcode an aktueller Position anzeigen: `list`
- Informationen über die Umbegung
  - `info args` Argumente der aktuellen Funktion
  - `info locals` Funktionslokale Variablen
  - `bt` Darstellung des Stacks (backtrace)
  - `bt full` Darstellung des Stacks mit Variablen

Demo

Technische Universität Braunschweig

```c
#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++;
        }
}
```

02t-speicher 2025-11-07