Übungen zu Systemprogrammierung 1 (SP1)

Ü4 – Freispeicherverwaltung

Jens Schedel, Christoph Erhardt, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

SS 2014 – 05. bis 09. Mai 2014

http://www4.cs.fau.de/Lehre/SS14/V_SP1

Agenda

Agenda

Dynamische Speicherverwaltung

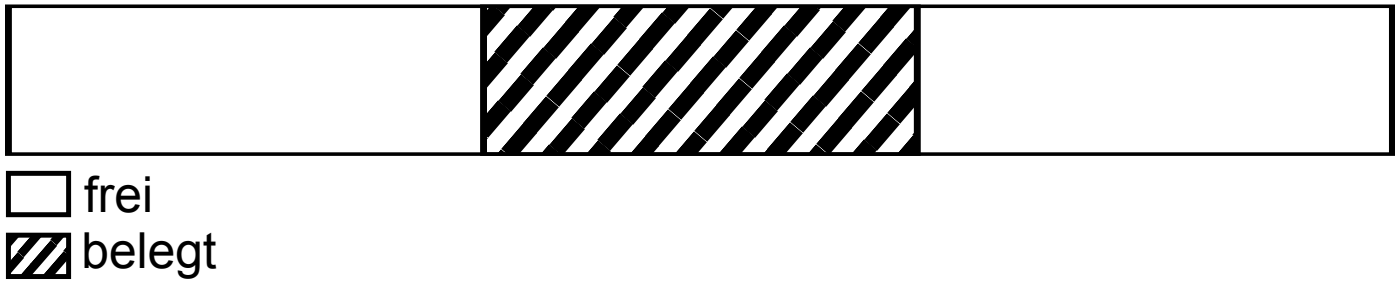
Auszug aus Wikipedia

„Der dynamische Speicher, auch Heap (engl. für ‚Halde‘, ‚Haufen‘), Haldenspeicher oder Freispeicher ist ein Speicherbereich, aus dem zur Laufzeit eines Programms zusammenhängende Speicherabschnitte angefordert und in beliebiger Reihenfolge wieder freigegeben werden können.”

- In C
  - Anforderung des Speichers mit Hilfe von `malloc(3)`
    - Parameter: Größe des angeforderten Speichers
    - Rückgabewert: Zeiger auf einen Speicherbereich
  - Explizite Freigabe mit Hilfe von `free(3)`
    - Parameter: Zeiger auf freizugebenden Speicherbereich
    - Rückgabewert: –

Anforderungsanalyse

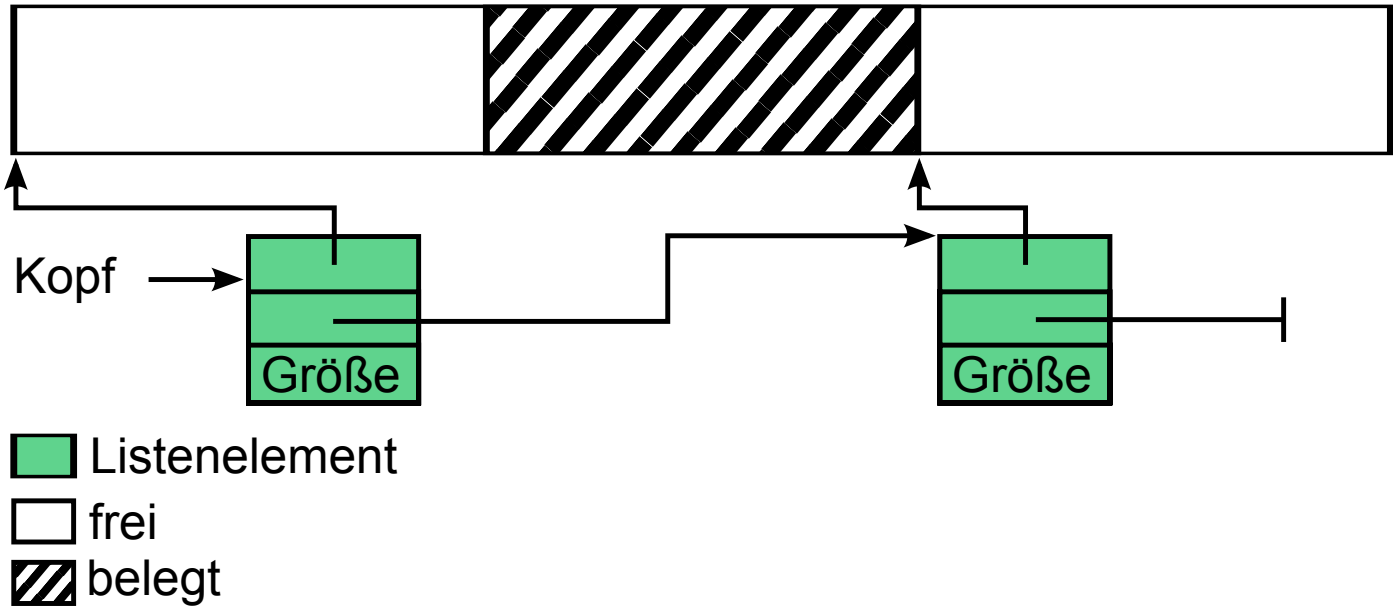
- Ziel: Speicherbereiche, die zur Laufzeit in beliebiger Größe angefordert werden können
- Skizze: Zustand eines teilweise belegten Heaps



- Welche Informationen muss eine Freispeicherverwaltung bereit halten?
  - für freie Blöcke: Lage aller freien Blöcke und 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!)



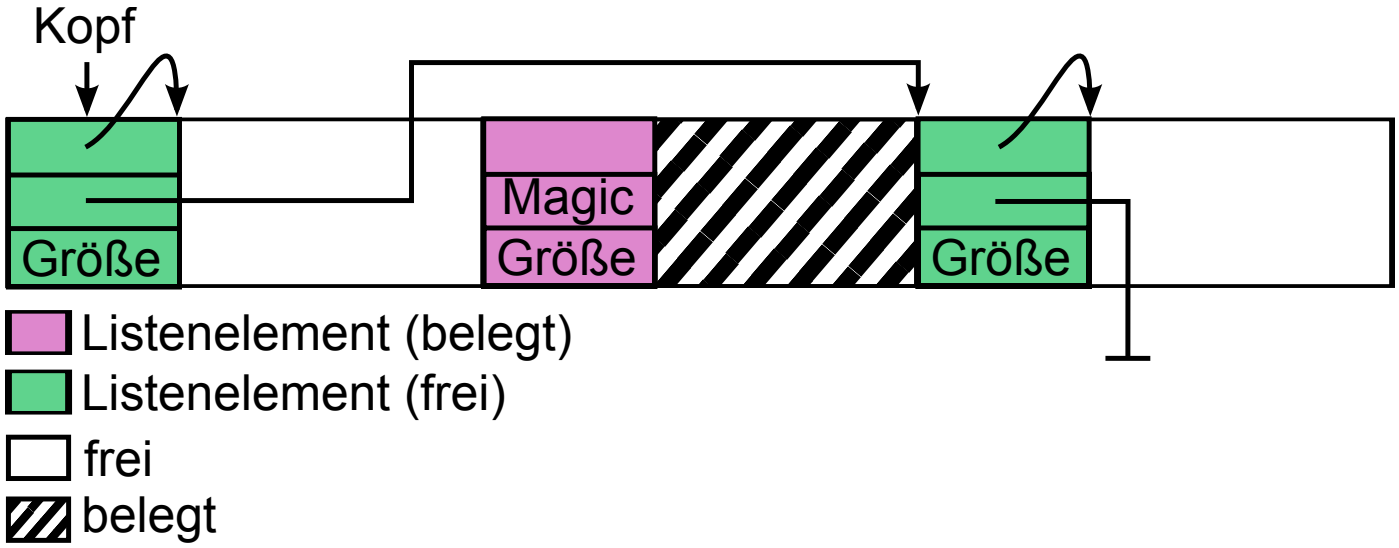
  - Freie Blöcke werden in einer verketteten Liste gespeichert

- Wiederholung Aufgabe 1 (lilo)
  - Wie wird eine verkettete Liste in C implementiert?

    `insertElement()` → `malloc()` → `insertElement()` → `malloc()` →
    `insertElement()` → `malloc()` → `insertElement()` → `malloc()` →
    `insertElement()` → `malloc()` → `insertElement()` → ...

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

Agenda

Implementierung

Listenelementdefinition in C

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

- Verwendung von FAM (Flexible Array Member):
  - `mem_area` ist eigentlich ein Feld beliebiger Länge
  - In unserem Fall: `mem_area` ist ein konstanter „Zeiger" auf das Ende der Struktur
  - `mem_area` selbst hat die Größe 0

Beispiel auf den Folien

- Schrittweises Abarbeiten des folgenden Codestückes

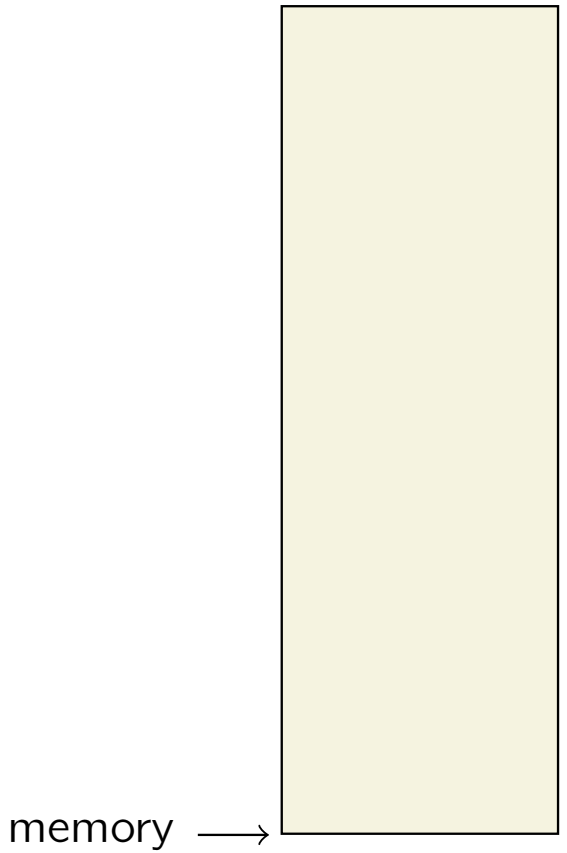
```
char *m1 = (char *) malloc(10);
char *m2 = (char *) malloc(20);

free(m2);
```

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

Initialisierung

- Speicher statisch allokiert

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



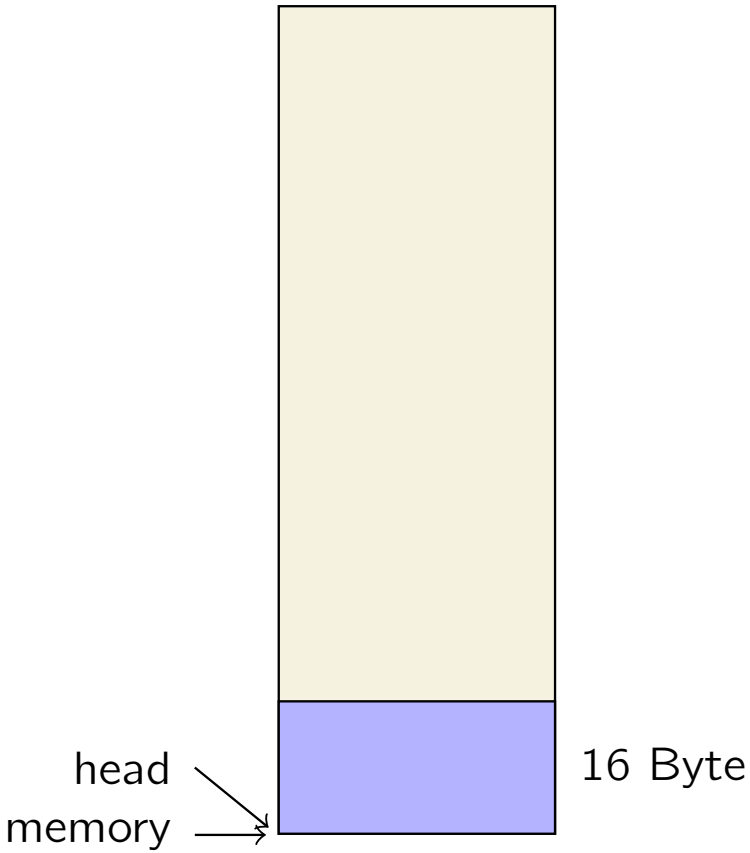
Initialisierung

- Speicher statisch allokiert

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

- `struct mblock` reinlegen

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



Initialisierung

- Speicher statisch allokiert

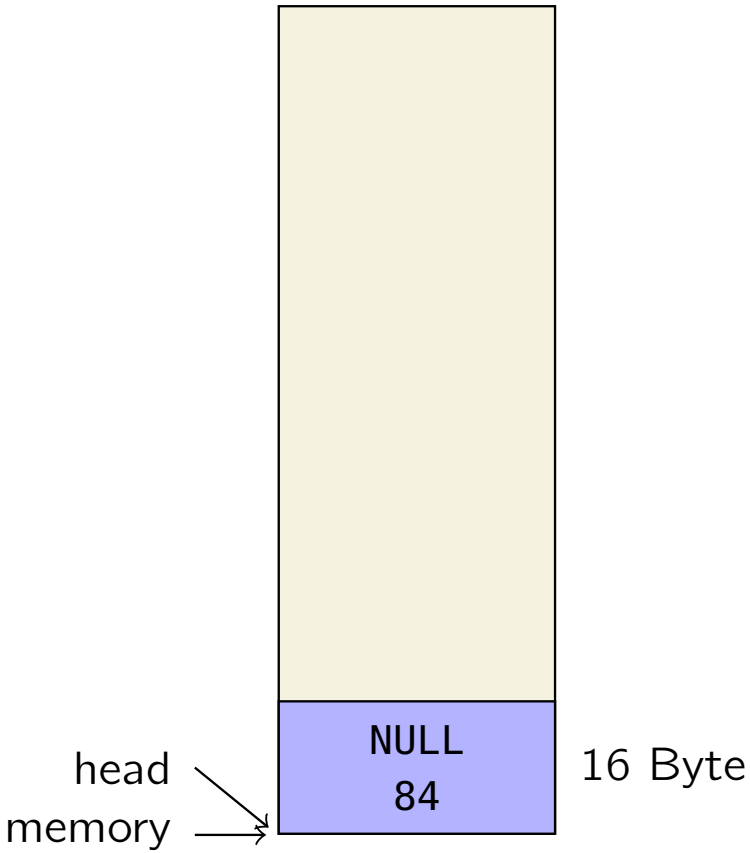
```
static char memory[100];
```

- `struct mblock` reinlegen

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

- `struct mblock` initialisieren

```
head->next = NULL;
head->size = 84;
```



! zwei Zeiger mit unterschiedlichem Typ auf den gleichen Speicherbereich

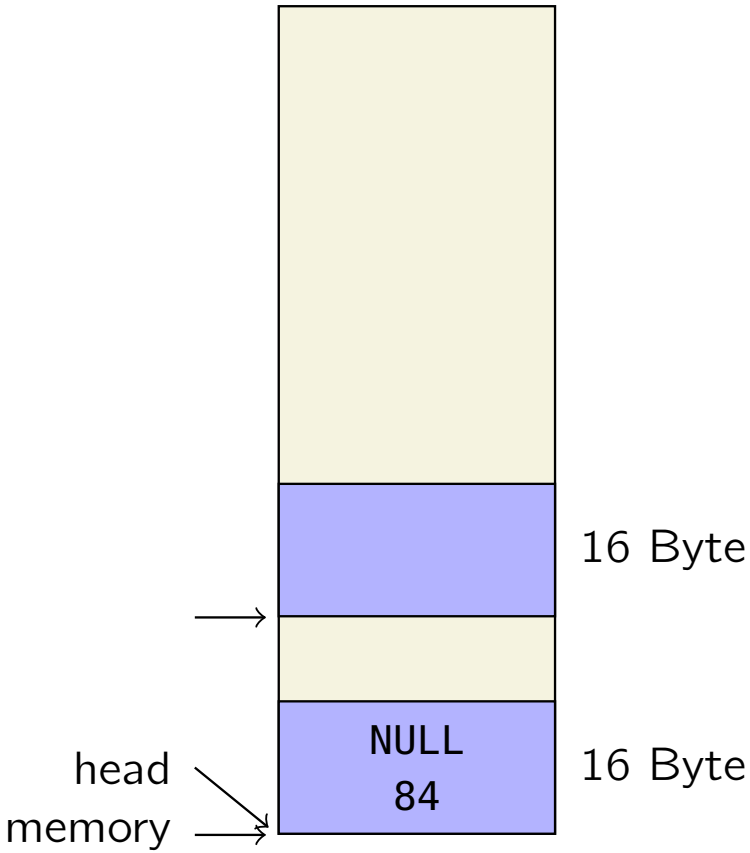
- unterschiedliche Semantik beim Zugriff (Zeigerarithmetik, Strukturkomponenten)

Speicheranforderung im Detail

- Speicheranforderung von 10 Byte

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

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

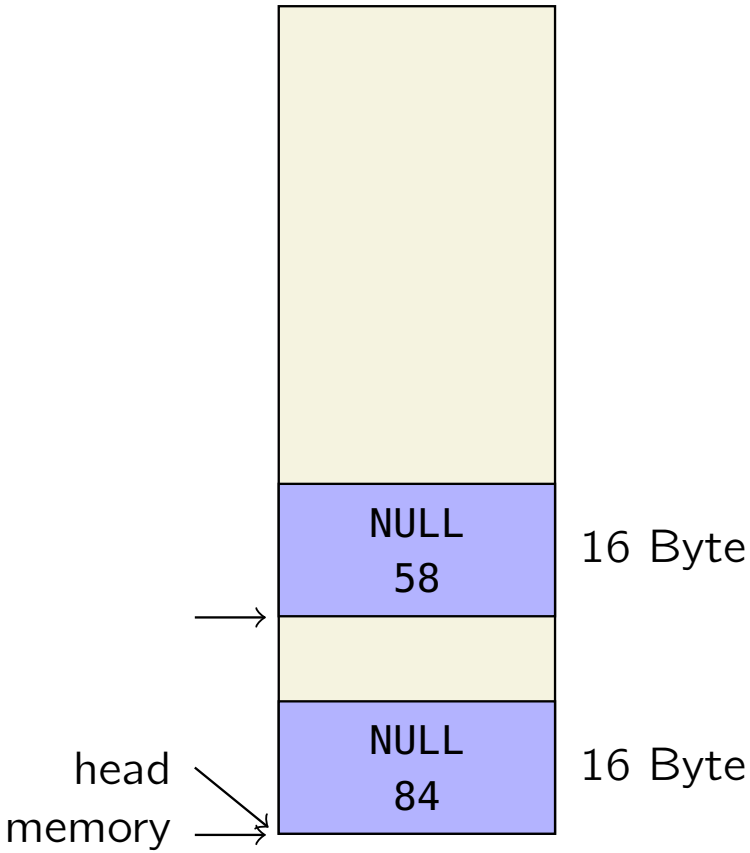


Speicheranforderung im Detail

- Speicheranforderung von 10 Byte

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

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

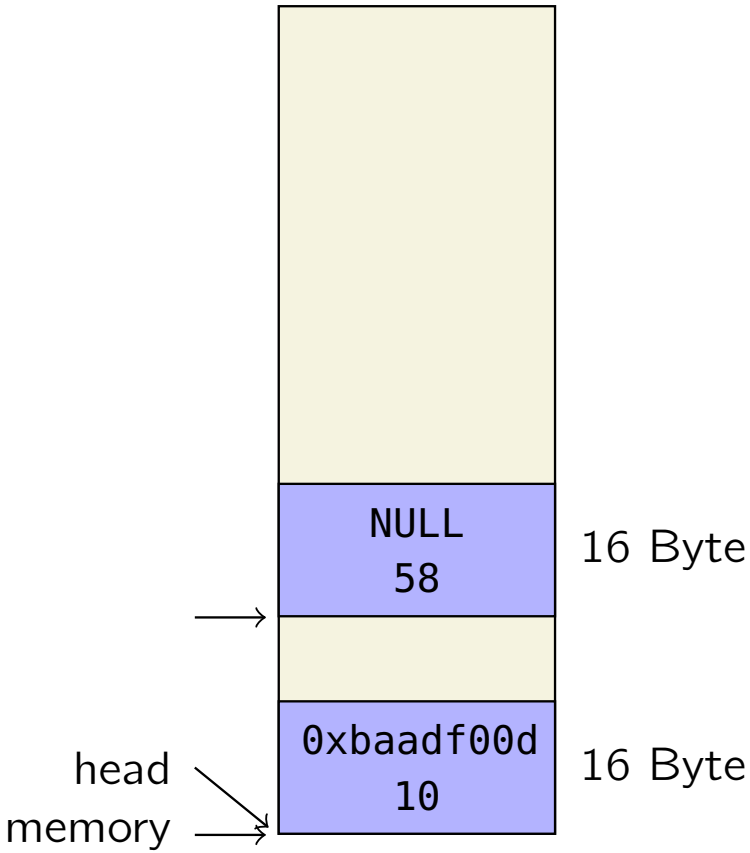


Speicheranforderung im Detail

Speicheranforderung von 10 Byte

```
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

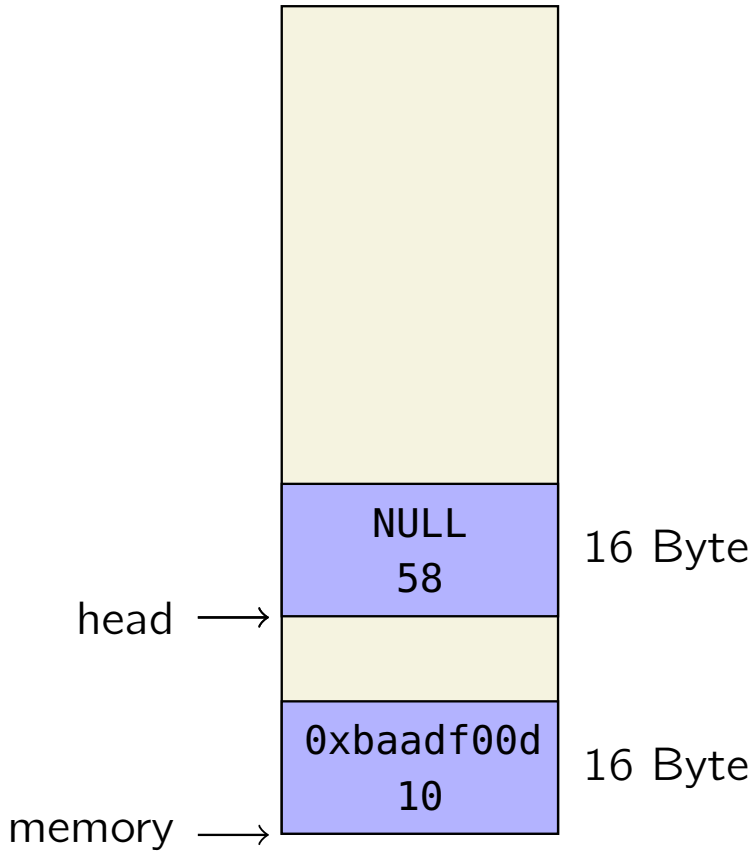


Speicheranforderung im Detail

- Speicheranforderung von 10 Byte

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

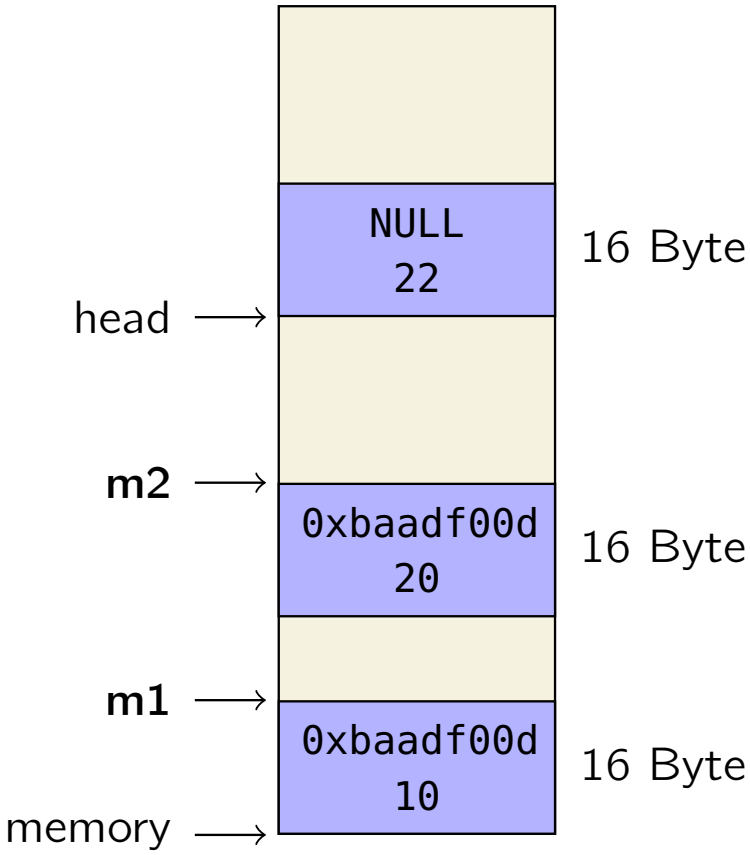
- Freispeicherliste nach mblock mit ausreichend Speicher durchsuchen
- 10 Bytes hinter dem **head-mblock** einen neuen mblock anlegen
- ...und initialisieren
- Bisherigen **head-mblock** anpassen
  - als belegt markieren
  - Größe des Speicherbereichs aktualisieren
- **head**-Zeiger auf neues Kopfelement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



Speicheranforderung im Detail

- Situation nach 2 `malloc()`-Aufrufen

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

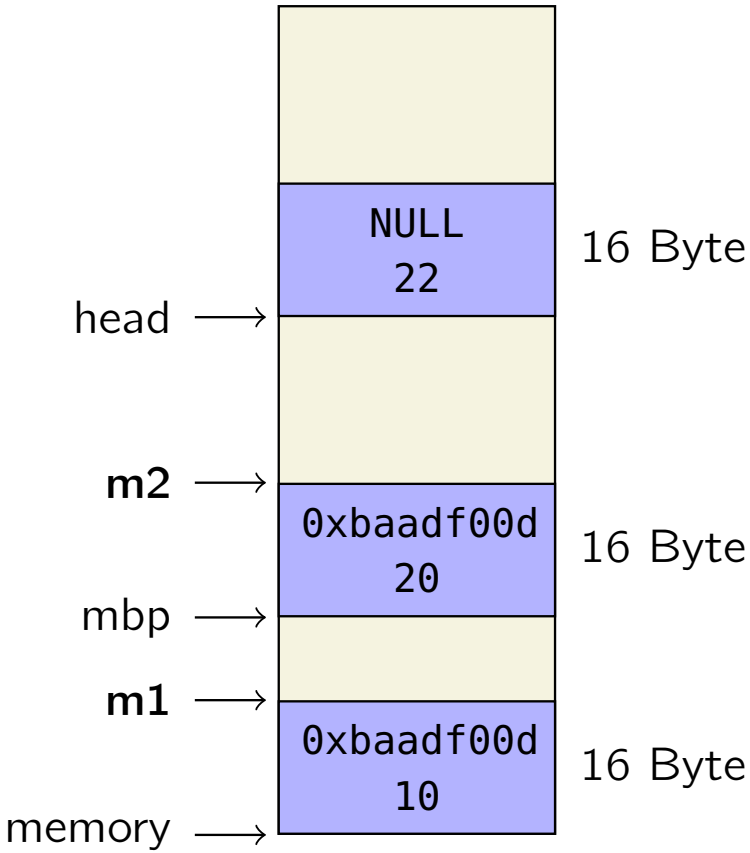


Speicherfreigabe

- Freigabe von `m2`

```
free(m2);
```

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

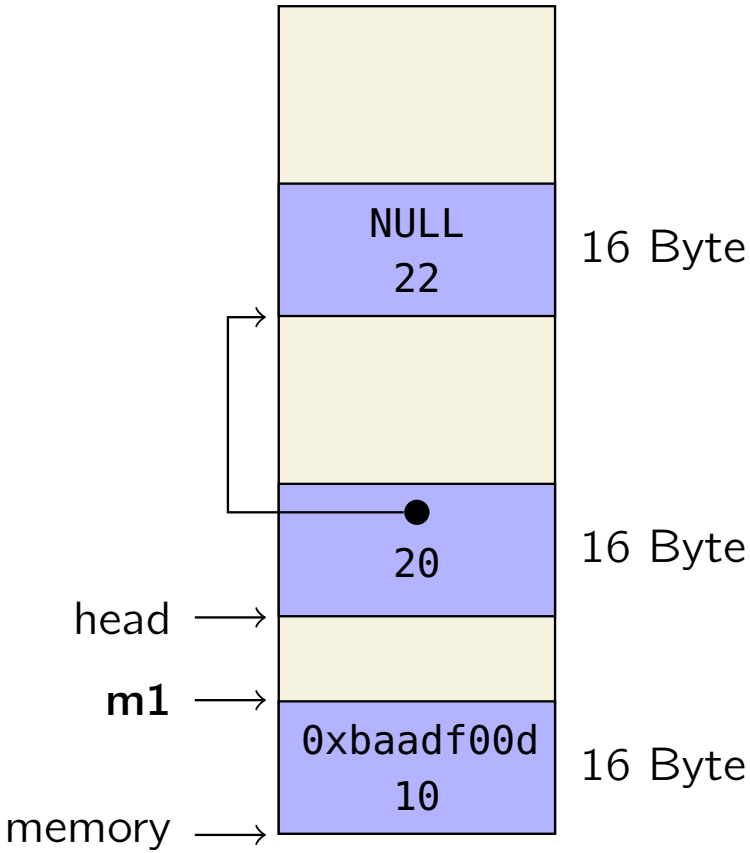


Speicherfreigabe

- 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



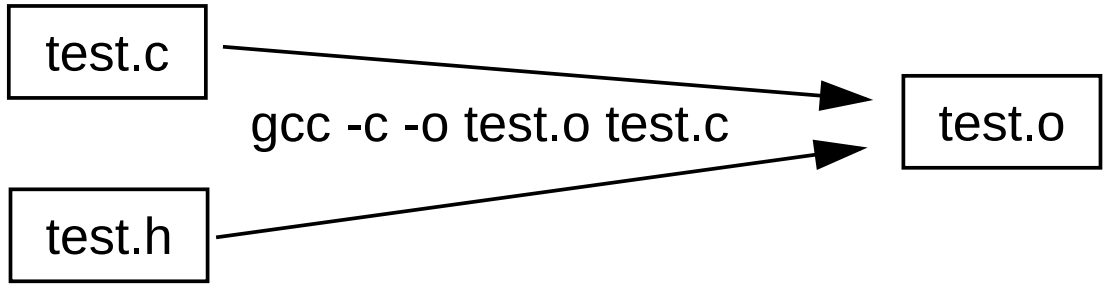
Zusammenfassung

- 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
- sinnvolle Implementierung erfordert geeignete Speichervergabestrategie
  - Implementierung erheblich aufwändiger - Resultat aber entsprechend effizienter
  - Strategien werden im Abschnitt Speicherverwaltung in der Vorlesung SP2 behandelt (z. B. First-Fit, Best-Fit, Worst-Fit oder Buddy-Verfahren)

Agenda

Make – Teil 1

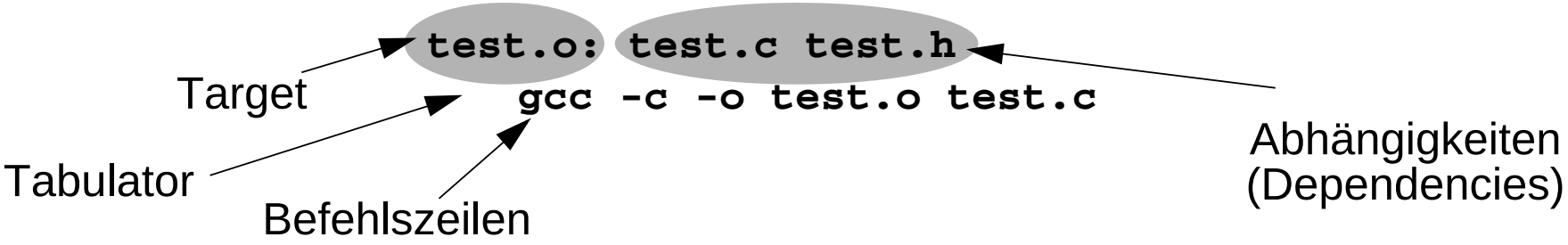
- Grundsätzlich: Erzeugung von Dateien aus anderen Dateien
  - für uns interessant: Erzeugung einer .o-Datei aus einer .c-Datei



- Ausführung von *Update*-Operationen (auf Basis der Modifikationszeit)

Funktionsweise

- Regeldatei mit dem Namen `Makefile`



  - Target (was wird erzeugt?)
    - erzeugt gleichnamige Datei
  - Abhängigkeiten (woraus?)
    - kann auch ein Target sein
  - Befehlszeilen (wie?)
- zu erstellendes Target bei `make`-Aufruf angeben: `make test.o`
  - ohne Target-Abgabe bearbeitet `make` das erste Target im Makefile

Makros

- In einem Makefile können Makros definiert werden

```
SOURCE = test.c func.c
```

- Verwendung der Makros mit `$(NAME)` oder `${NAME}`

```
test: $(SOURCE)
  gcc -o test $(SOURCE)
```

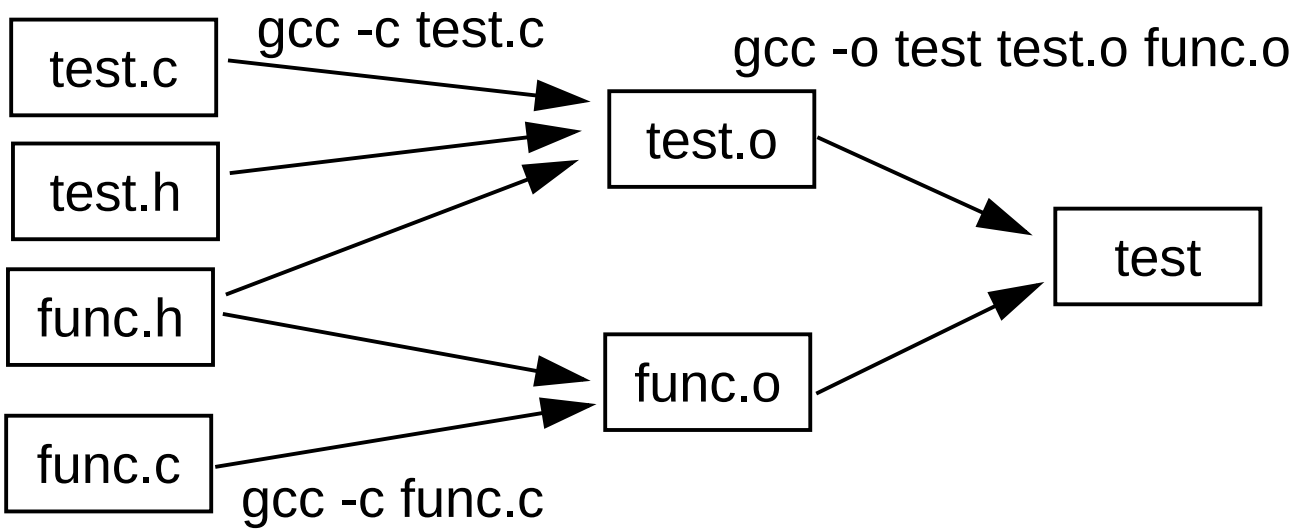
- Erzeugung neuer Makros durch Konkatenation

```
ALLOBJS = $(OBJS) hallo.o
```

- Gängige Makros:
  - `CC` C-Compiler-Befehl
  - `CFLAGS` Optionen für den C-Compiler

Schrittweises Übersetzen

- Rechner beim Erzeugen von ausführbaren Dateien „entlasten”



  - Zwischenprodukte verwenden und somit Übersetzungszeit sparen

Agenda

Ziele der Aufgabe

- Ziele der Aufgabe
  - Zusammenhang zwischen „nacktem Speicher” und typisierten Datenbereichen verstehen
  - Funktion aus der C-Bibliothek selbst realisieren
  - Umgang mit `make(1)`
  - Entwickeln eigener Testfälle für selbstgeschriebenen Code
- Vereinfachungen
  - First-Fit-ähnliche Allokationsstrategie
  - 1 MiB Speicher statisch allokiert
  - 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

Agenda

Aktive Mitarbeit!

„Aufgabenstellung“

- Skizzieren Sie den Aufbau des verwalteten Speicherbereichs (hier: 64 Byte, `sizeof(mblock)` = 16 Byte) nach jedem Schritt des jeweiligen Szenarios
  - Szenario 1:

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

  - Szenario 2:

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

  - Szenario 3:

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