

# Übungen zu Systemprogrammierung 1

## Ü3 – Freispeicherverwaltung

---

Sommersemester 2020

Dustin Nguyen, Jonas Rabenstein, 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

# Agenda



4.1 Freispeicherverwaltung

4.2 Implementierung

4.3 make

4.4 gdb

4.5 Aufgabe 3: halde

4.6 Gelerntes anwenden



4.1 Freispeicherverwaltung

4.2 Implementierung

4.3 make

4.4 gdb

4.5 Aufgabe 3: halde

4.6 Gelerntes anwenden

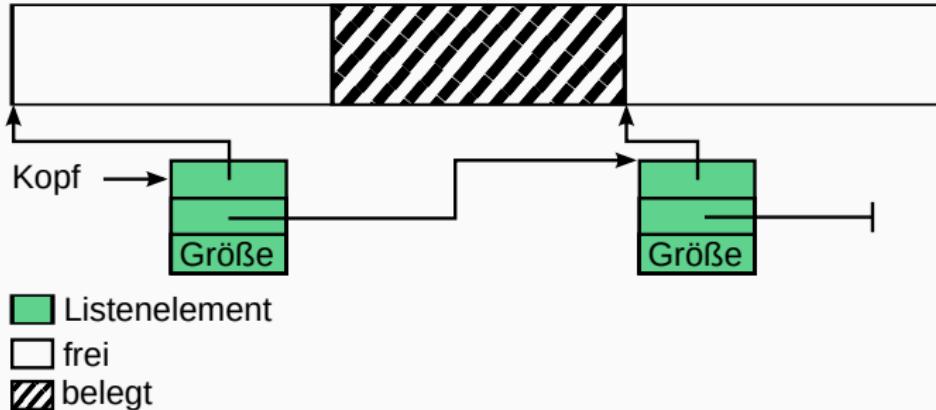
# Dynamische Speicherverwaltung (in C)



- 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: –



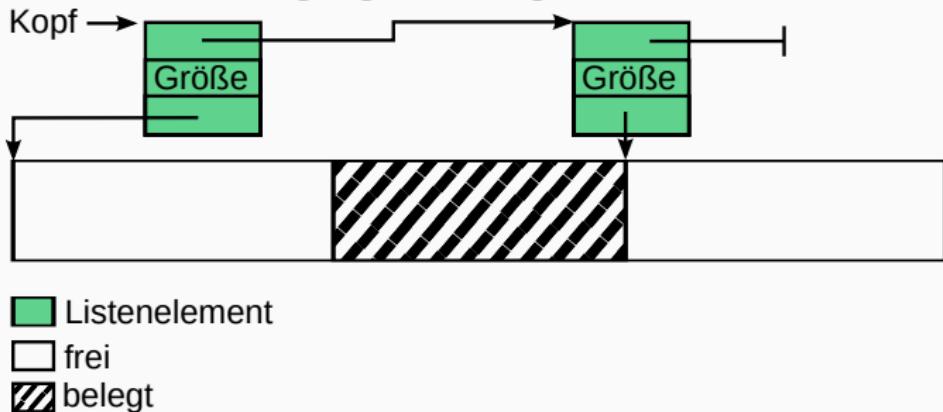
- Ziel: Speicherbereiche, die zur Laufzeit in beliebiger Größe angefordert werden können
- Skizze: Zustand eines teilweise belegten Heaps



- 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): einfache 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

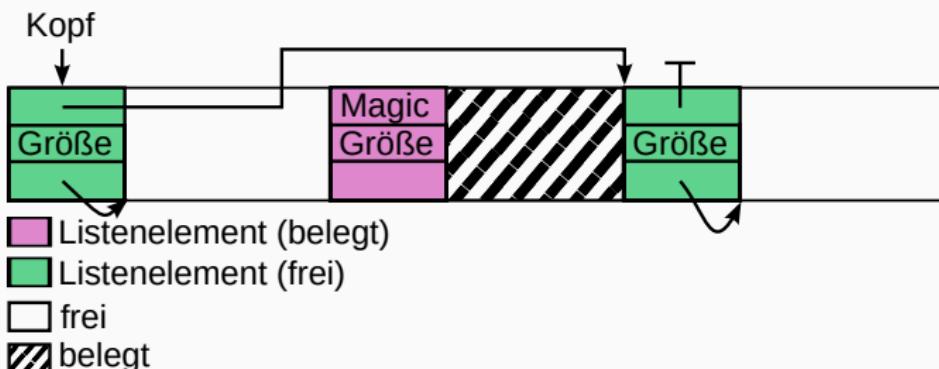
- Wiederholung Übung 1

- Wie wird eine verkettete Liste in C implementiert?

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

# 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



4.1 Freispeicherverwaltung

## 4.2 Implementierung

4.3 make

4.4 gdb

4.5 Aufgabe 3: halde

4.6 Gelerntes anwenden



# Implementierung

## ■ 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):

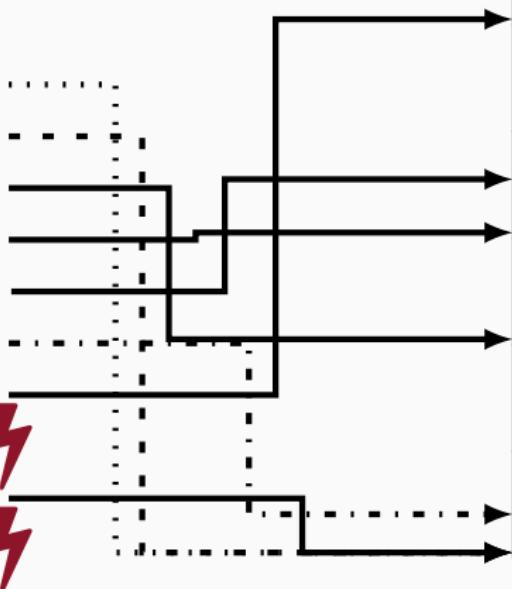
- mem\_area ist ein Feld beliebiger Länge
- In unserem Fall: mem\_area ist ein konstanter „Verweis“ auf das Ende der Struktur
- mem\_area selbst hat die Größe 0

# Einschub: Zeiger und Zeigerarithmetik



```
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:

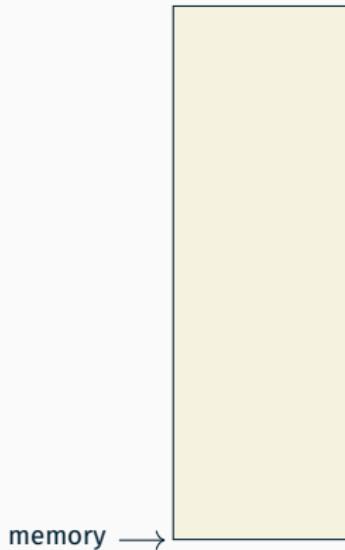
- Freispeicherverwaltung verwaltet 100 Bytes statisch allozierten Speicher
- Verwendung von absoluten Größen (Annahme: 64-Bit-Architektur)
  - Größe eines Zeigers: 8 Bytes
  - Größe der `struct mblock`: 16 Bytes



# Initialisierung

- Speicher statisch alloziert

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





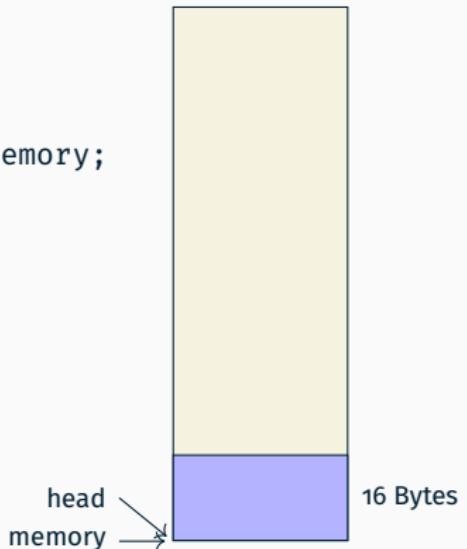
# Initialisierung

- 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

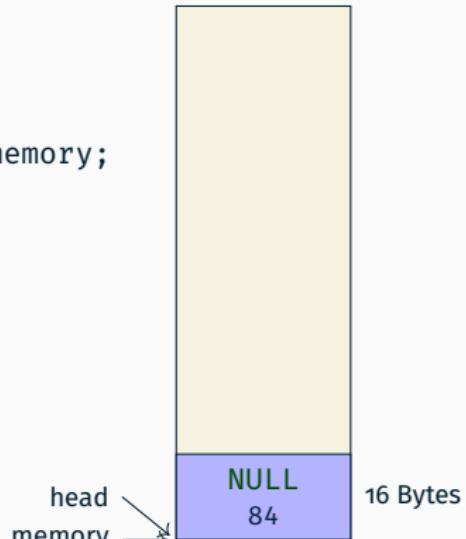
```
struct mblock *head = (struct mblock *)memory;
```

- struct mblock initialisieren

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

- ! zwei Zeiger mit unterschiedlichem Typ auf den gleichen Speicherbereich

- unterschiedliche Semantik beim Zugriff  
(Zeigerarithmetik, Strukturkomponenten)



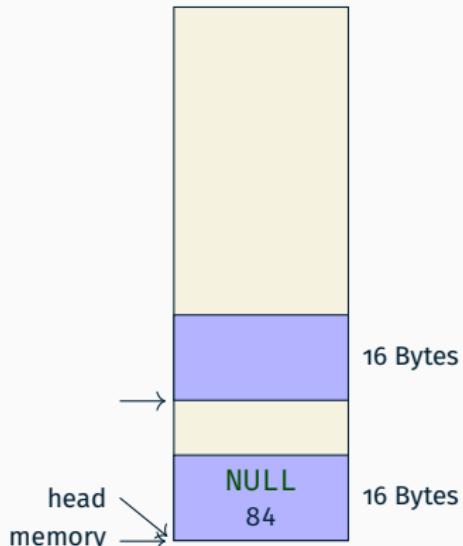


# Speicheranforderung im Detail

## ■ 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



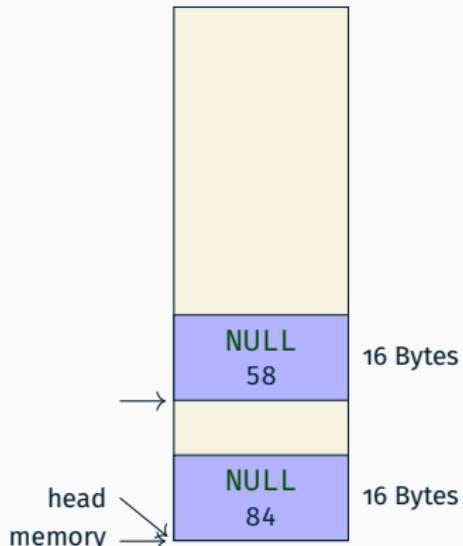


# Speicheranforderung im Detail

## ■ 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



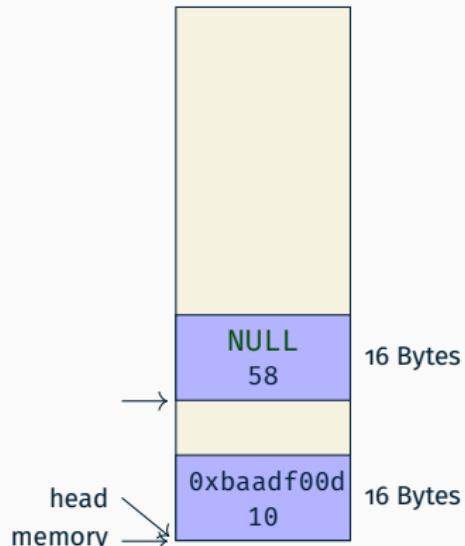


# Speicheranforderung im Detail

## ■ 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



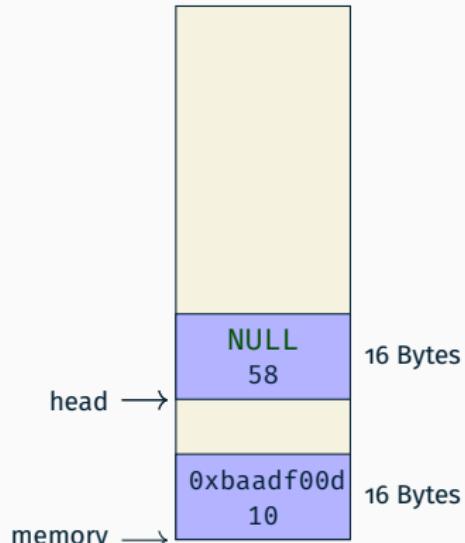


# Speicheranforderung im Detail

## ■ 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

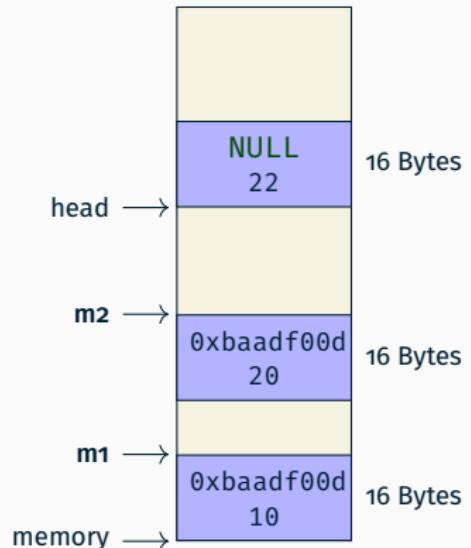




# Speicheranforderung im Detail

- Situation nach 2 malloc( )-Aufrufen

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



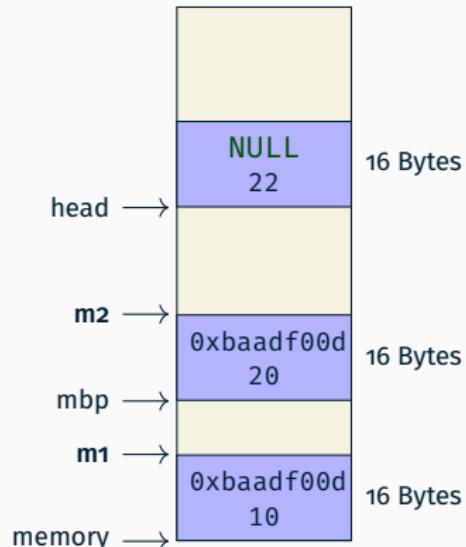


# Speicherfreigabe

## ■ Freigabe von m2

```
free(m2);
```

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



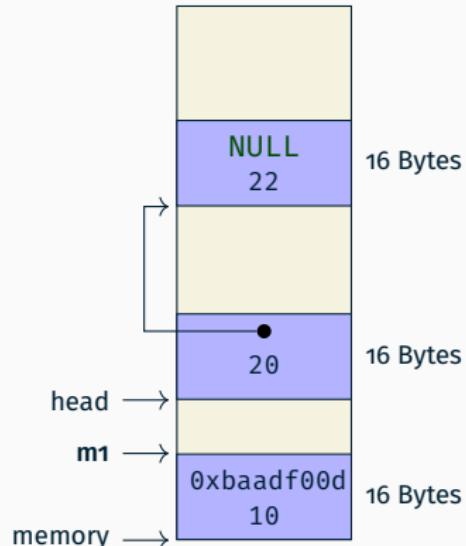


# Speicherfreigabe

## ■ 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 verkettten





# 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: Speicherseiten 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



4.1 Freispeicherverwaltung

4.2 Implementierung

**4.3 make**

4.4 gdb

4.5 Aufgabe 3: halde

4.6 Gelerntes anwenden

# Make – Warum?

## # .c-Dateien

lilo	1
vim 8.1	136
OpenSSH 7.9p1	269
Linux 4.19.1	> 26000

- ✗ von Hand übersetzen: zu aufwändig
- ✗ Dauer bei wiederholtem Übersetzen
- Automatisiertes Übersetzen **modifizierter Dateien**





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

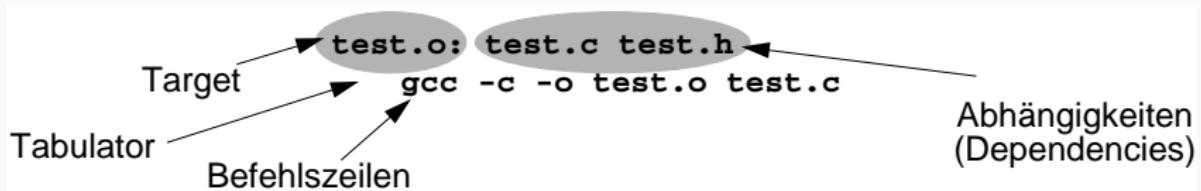


- Falls Quelle(n) sich ändert wird der Befehl neu ausgeführt
- Änderung auf Basis der Modifikationszeit



# Funktionsweise

## ■ Regeldatei mit dem Namen Makefile



- **Target (was wird erzeugt?)**
  - Name der zu erstellenden Datei
- **Abhängigkeiten (woraus?)**
  - Namen aller Eingabedateien (direkt oder indirekt)
  - Können selbst Targets sein
- **Befehlszeilen (wie?)**
  - Erzeugt aus den Abhängigkeiten das Target
- zu erststellendes Target bei make-Aufruf angeben: `make test.o`
  - Falls nötig baut make die angegebene Datei neu
  - Davor werden rekursiv alle veralteten Abhängigkeiten aktualisiert
  - Ohne Target-Angabe bearbeitet make das erste Target im Makefile



- 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

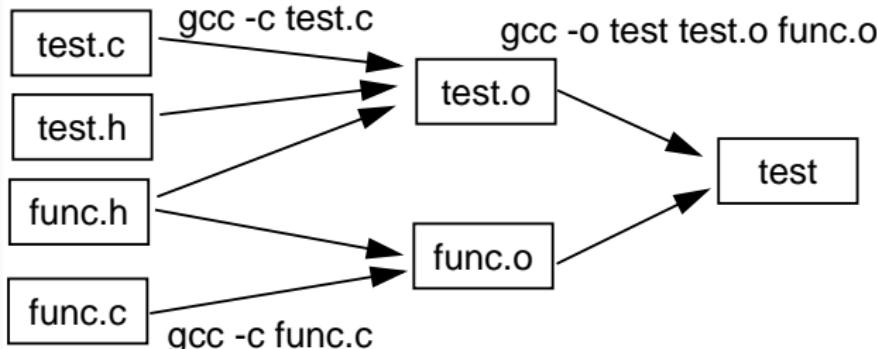
```
ALLOBJS = $(OBJS) hallo.o
```

- 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
- Beispiel:

```
test: test.o func.o
      gcc -o test test.o func.o
```

```
test.o: test.c test.h func.h
      gcc -c test.c
```

```
func.o: func.c func.h
      gcc -c func.c
```

# Agenda



4.1 Freispeicherverwaltung

4.2 Implementierung

4.3 make

**4.4 gdb**

4.5 Aufgabe 3: halde

4.6 Gelerntes anwenden



# 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
    - 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>`

# Demo



```
/* 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 [<Dateiname>:]<Funktionsname>
  - b <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
  - s (step: läuft in Funktionen hinein)
  - n (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 <variablename>=<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



# Agenda

4.1 Freispeicherverwaltung

4.2 Implementierung

4.3 make

4.4 gdb

**4.5 Aufgabe 3: halde**

4.6 Gelerntes anwenden



# 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 alloziert
- 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

4.1 Freispeicherverwaltung

4.2 Implementierung

4.3 make

4.4 gdb

4.5 Aufgabe 3: halde

**4.6 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:

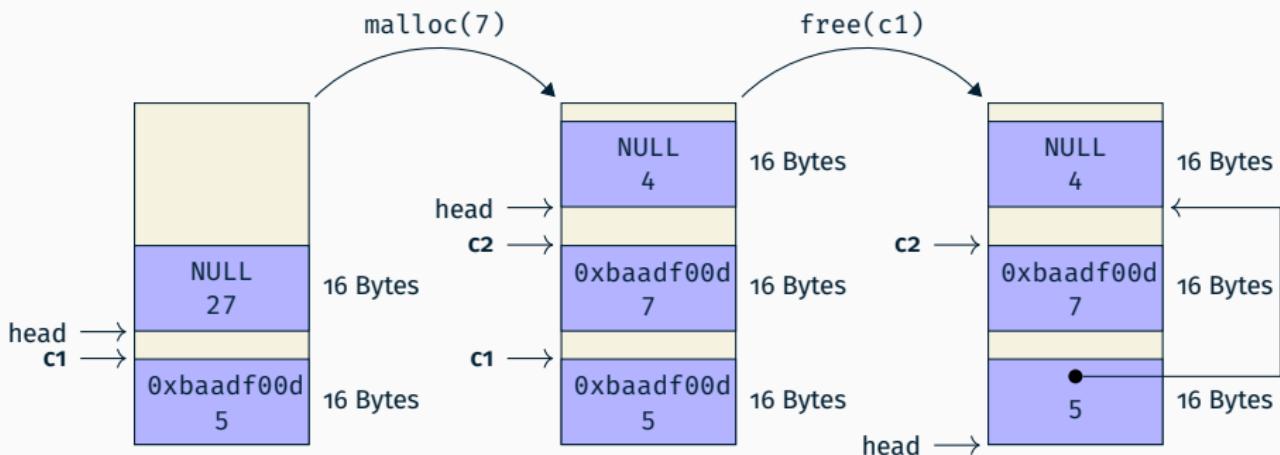
```
char *c1 = (char *)malloc(18);
char *c2 = (char *)malloc(14);
free(c1);
```

# Lösung zu den Aufgaben



## ■ Szenario 1:

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

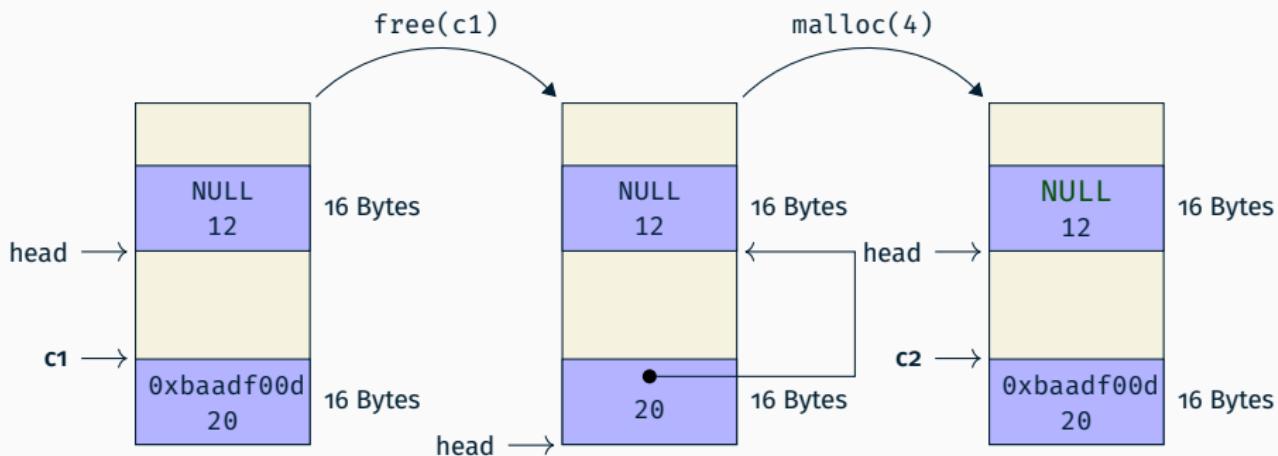


# Lösung zu den Aufgaben



## ■ Szenario 2:

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



# Lösung zu den Aufgaben



## ■ Szenario 3:

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

