

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 13 – 13. bis 17. Mai 2013

http://www4.cs.fau.de/Lehre/SS13/V_SP1



Agenda

- 4.1 Freispeicherverwaltung
- 4.2 Implementierung
- 4.3 make
- 4.4 gdb
- 4.5 Aufgabe 3: halde
- 4.6 Gelerntes Anwenden



Agenda

4.1 Freispeicherverwaltung

4.2 Implementierung

4.3 make

4.4 gdb

4.5 Aufgabe 3: halde

4.6 Gelerntes Anwenden



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: –



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

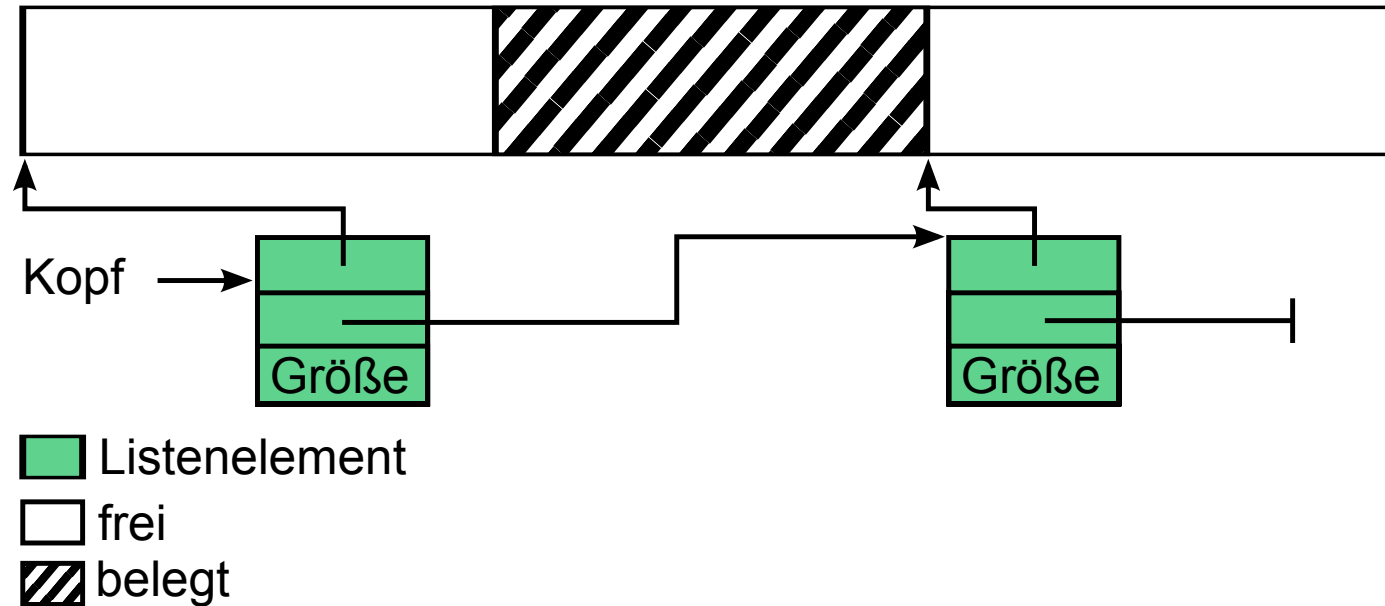


- Welche Informationen muss eine Freispeicherverwaltung bereit halten?
 - für freie Blöcke: Lage aller freien Blöcke und Größe und Lage des Speicherbereichs
 - für belegte Blöcke: Größe des Speicherbereichs
- Welche Datenstruktur ist für eine Freispeicherverwaltung geeignet?
 - KISS (Keep it small and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

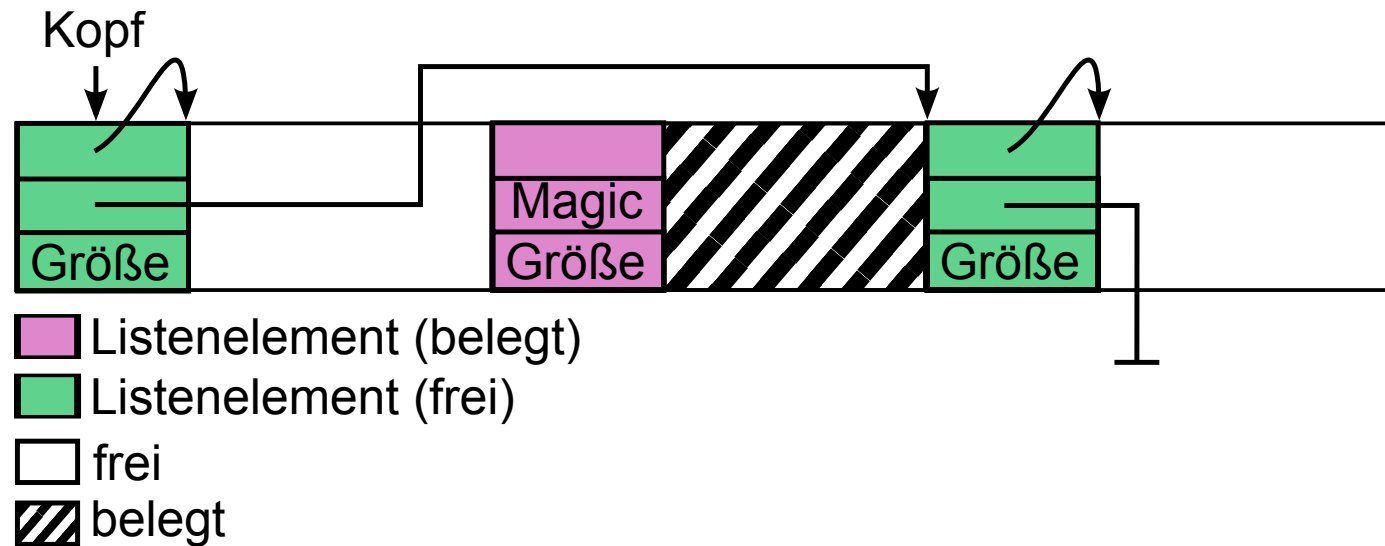
- Wie wird eine verkettete Liste in C implementiert?

```
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

- 4.1 Freispeicherverwaltung
- 4.2 Implementierung**
- 4.3 make
- 4.4 gdb
- 4.5 Aufgabe 3: halde
- 4.6 Gelerntes Anwenden



■ 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:

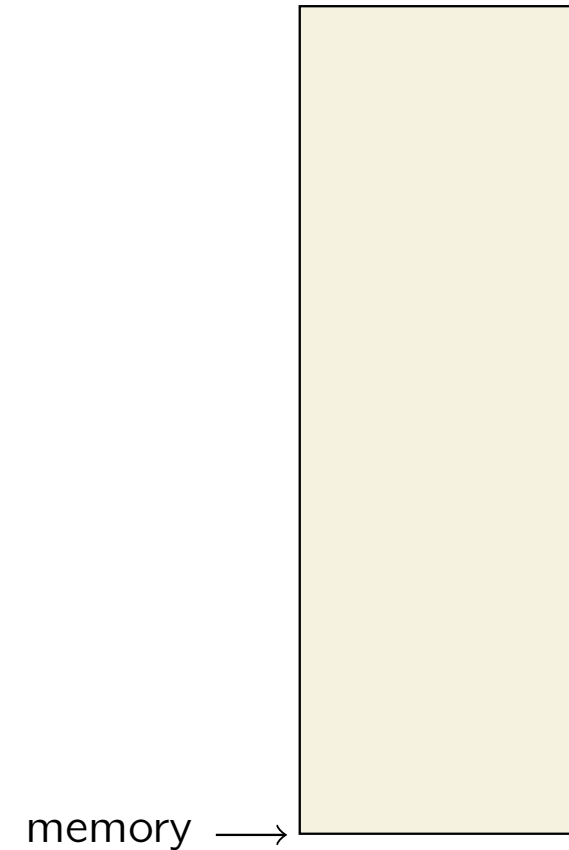
- Freispeicherverwaltung verwaltet 100 Byte statisch allokierten Speicher
- Verwendung von absoluten Größen (Annahme: 64-Bit Architektur)
 - Größe eines Zeigers: 8 Byte
 - Größe der struct mblock: 16 Byte



Initialisierung

- Speicher statisch allokiert

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



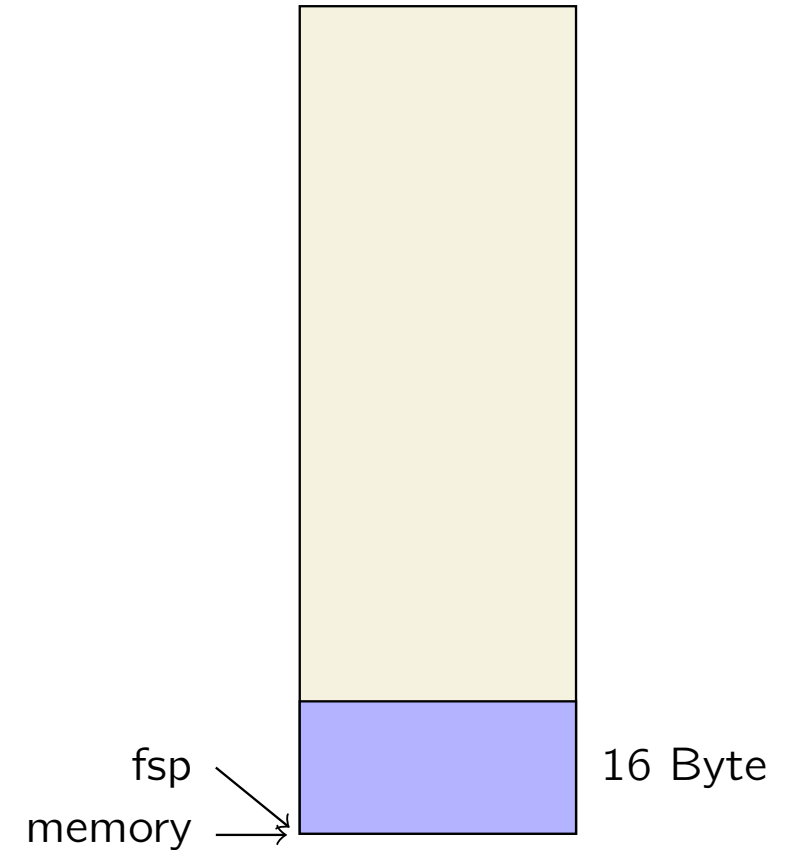
Initialisierung

- Speicher statisch allokiert

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

- struct mblock reinlegen

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



Initialisierung

- Speicher statisch allokiert

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

- struct mblock reinlegen

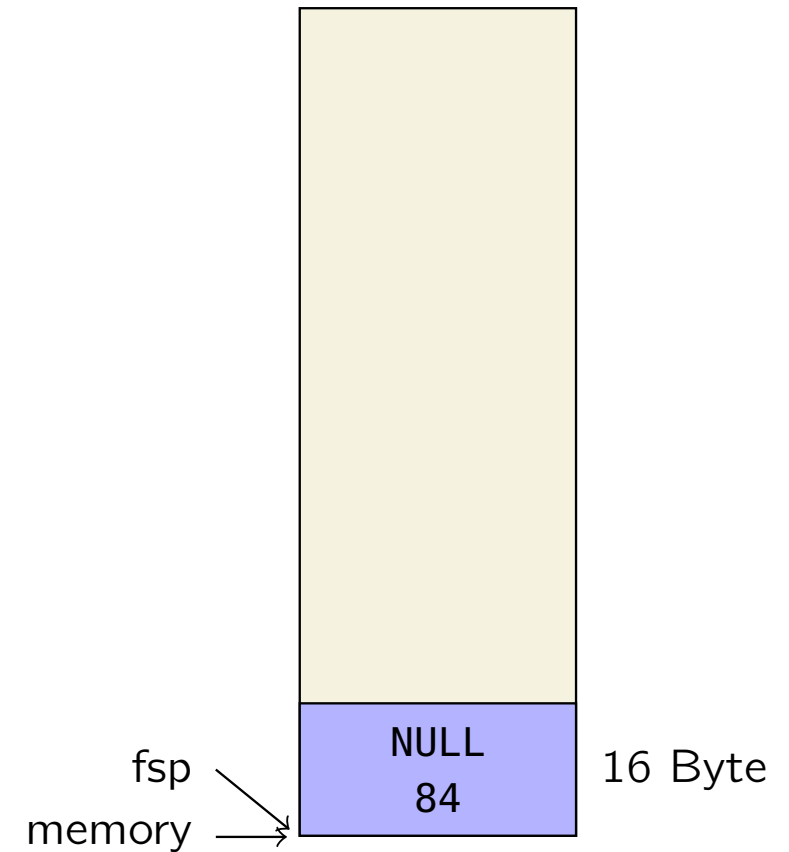
```
mblock* fsp = (struct mblock*) memory;
```

- struct mblock initialisieren

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

- ! zwei Zeiger mit unterschiedlichem Typ auf den gleichen Speicherbereich

- unterschiedliche Semantik beim Zugriff (Zeigerarithmetik, Strukturkomponenten)

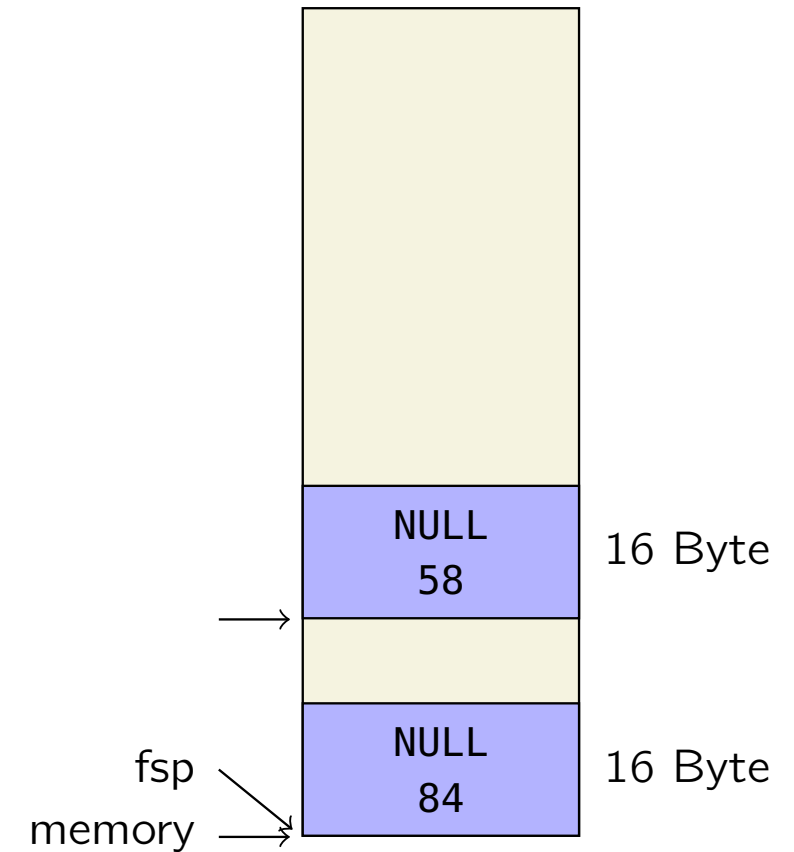


Speicheranforderung im Detail

■ Speicheranforderung von 10 Byte

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

- 10 Bytes hinter dem fsp-mblock einen neuen mblock anlegen
- ...und initialisieren

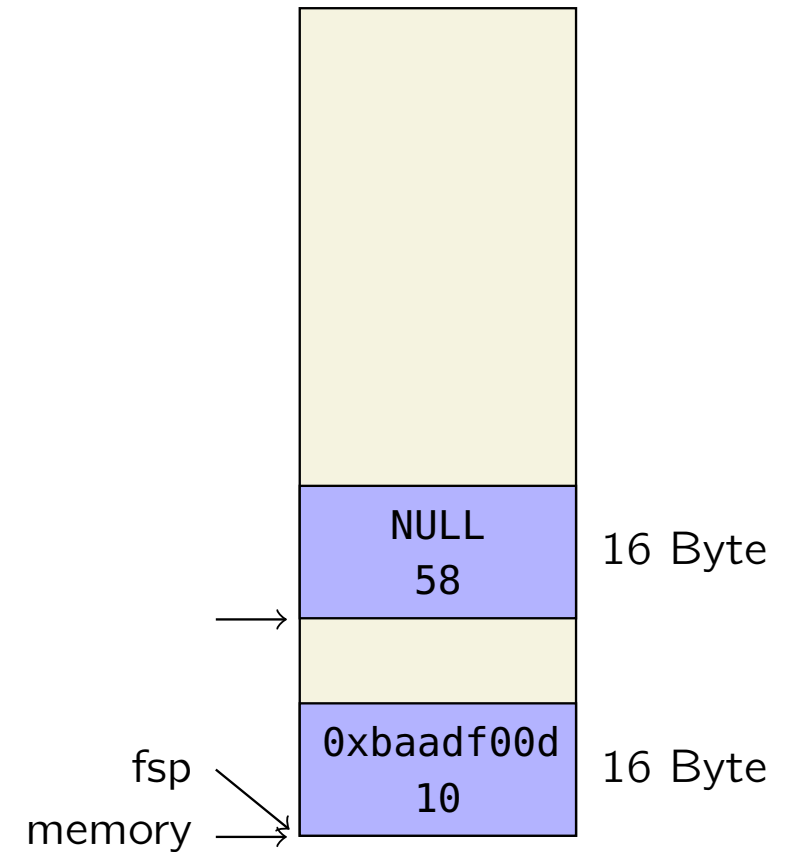


Speichieranforderung im Detail

■ Speichieranforderung von 10 Byte

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

- 10 Bytes hinter dem fsp-mblock einen neuen mblock anlegen
- ...und initialisieren
- Bisherigen fsp-mblock anpassen
 - als belegt markieren
 - Größe des Speicherbereichs aktualisieren

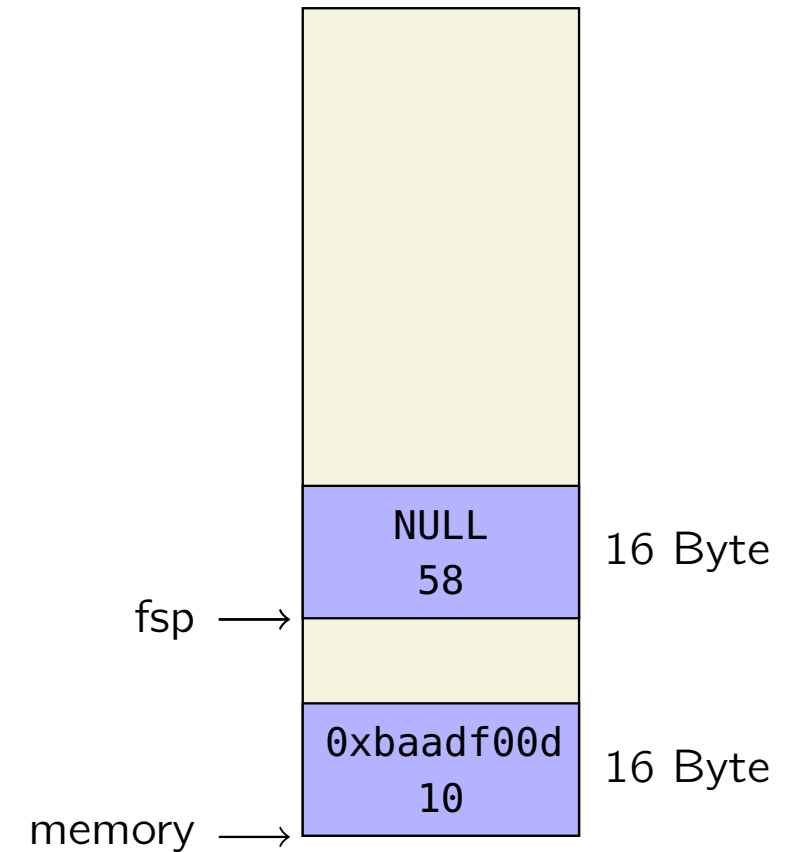


Speichieranforderung im Detail

■ Speichieranforderung von 10 Byte

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

- 10 Bytes hinter dem `fsp-mblock` einen neuen mblock anlegen
- ...und initialisieren
- Bisherigen `fsp-mblock` anpassen
 - als belegt markieren
 - Größe des Speicherbereichs aktualisieren
- `fsp`-Zeiger auf neues Kopfelement setzen

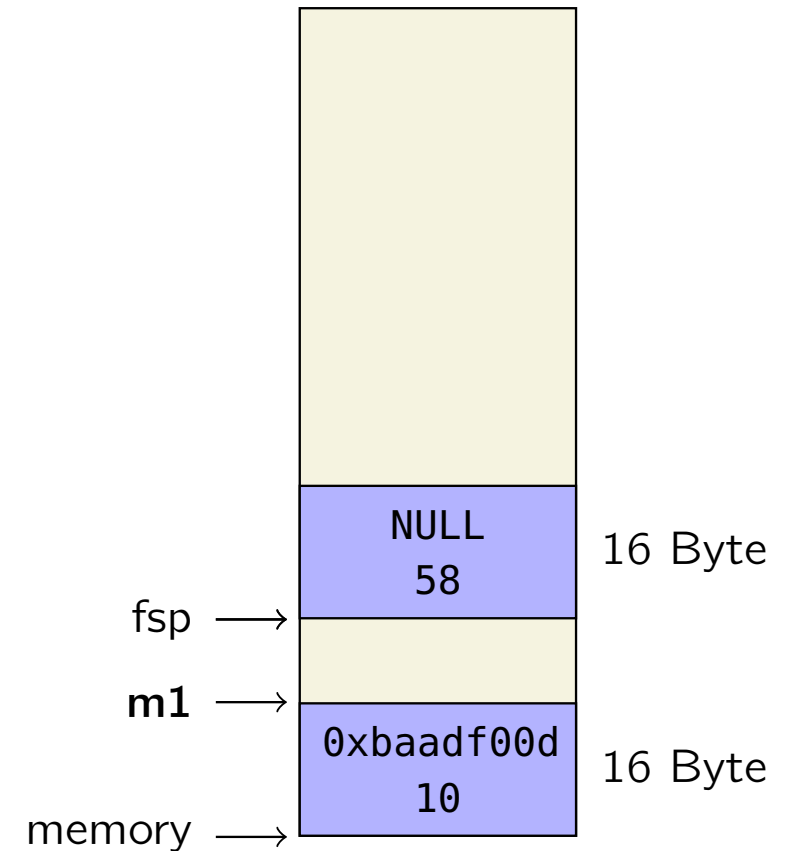


Speicheranforderung im Detail

■ Speicheranforderung von 10 Byte

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

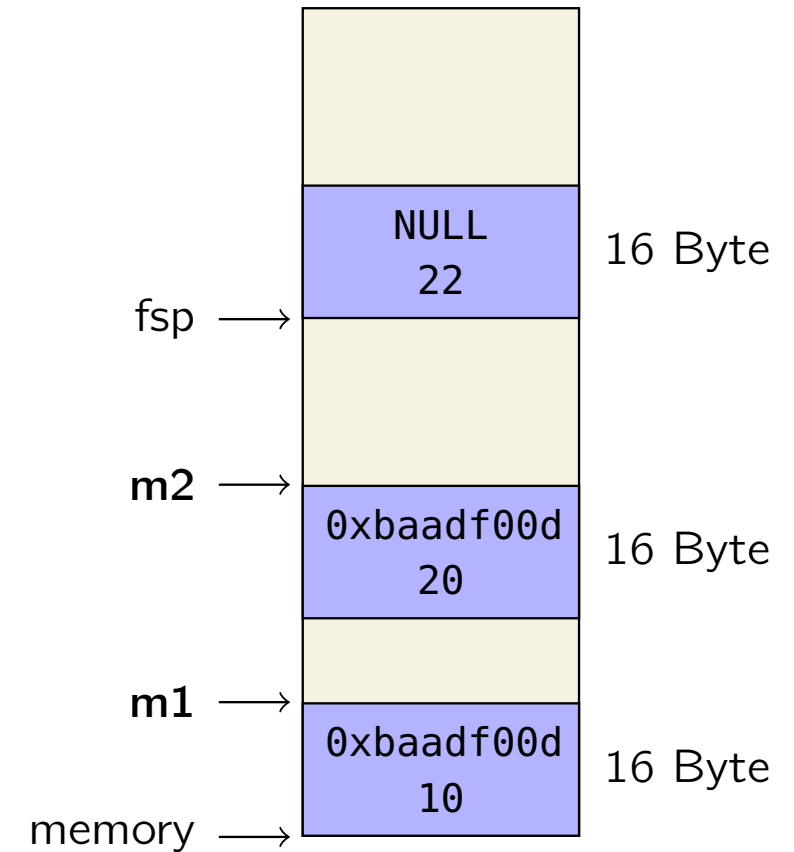
- 10 Bytes hinter dem `fsp-mblock` einen neuen `mblock` anlegen
- ...und initialisieren
- Bisherigen `fsp-mblock` anpassen
 - als belegt markieren
 - Größe des Speicherbereichs aktualisieren
- `fsp`-Zeiger auf neues Kopfelement setzen
- Zeiger auf die reservierten 10 Byte zurückgeben



Speichieranforderung im Detail

■ Situation nach 2 malloc()-Aufrufen

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

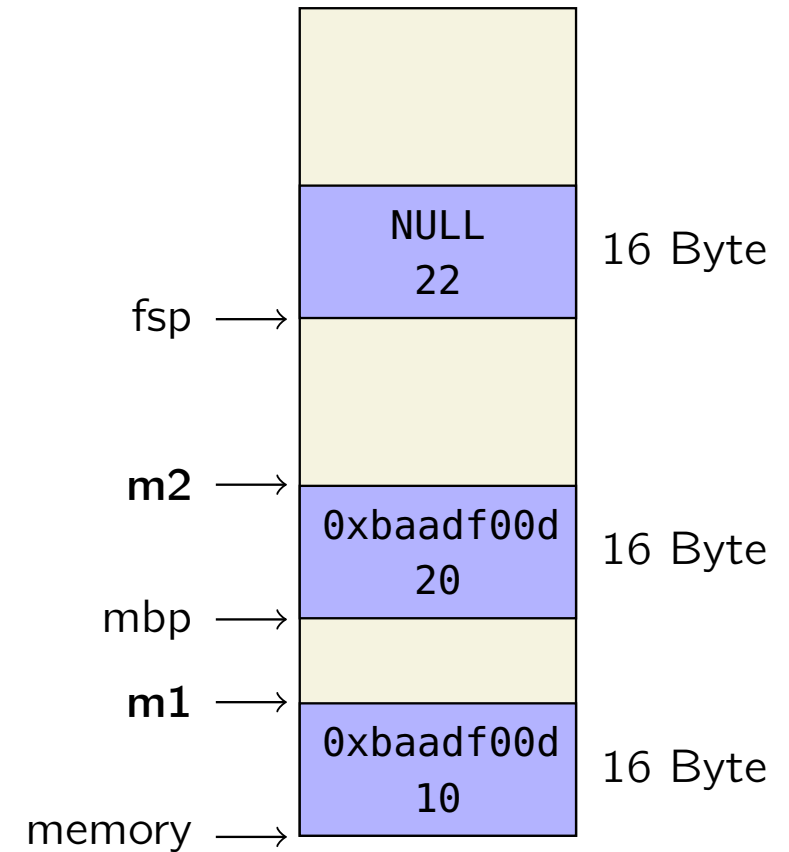


Speicherfreigabe

■ Freigabe von m2

```
free(m2);
```

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

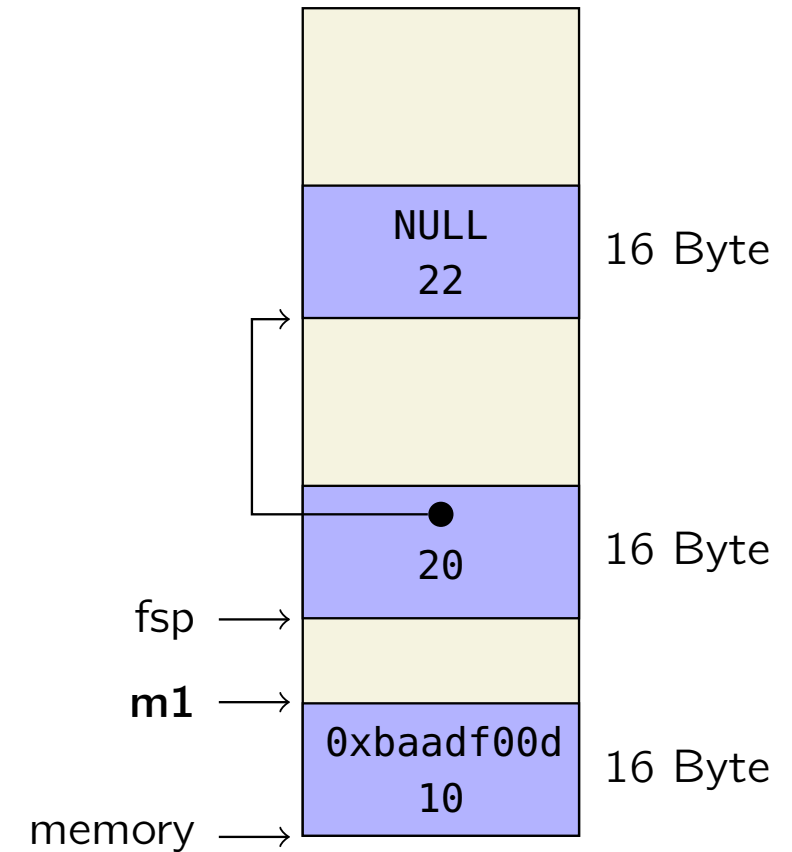


Speicherfreigabe

■ Freigabe von m2

```
free(m2);
```

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



- 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)



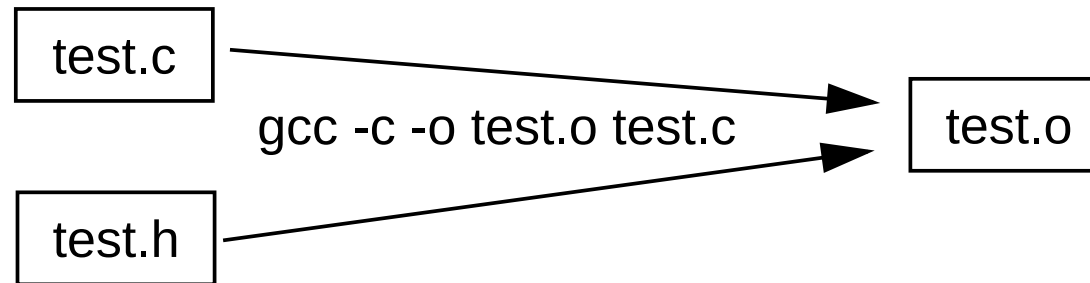
Agenda

- 4.1 Freispeicherverwaltung
- 4.2 Implementierung
- 4.3 make
- 4.4 gdb
- 4.5 Aufgabe 3: halde
- 4.6 Gelerntes Anwenden



Make – Teil 1

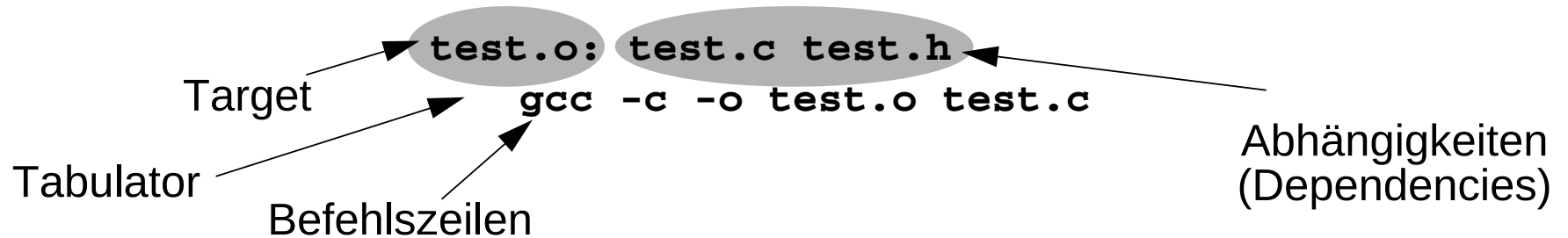
- Grundsätzlich: Erzeugung von Dateien aus anderen Dateien
 - für uns interessant: Erzeugung einer .o-Datei aus einer .c-Datei



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



■ 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



- 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 Konkatination

```
ALLOBSJS = $(OBSJS) hallo.o
```

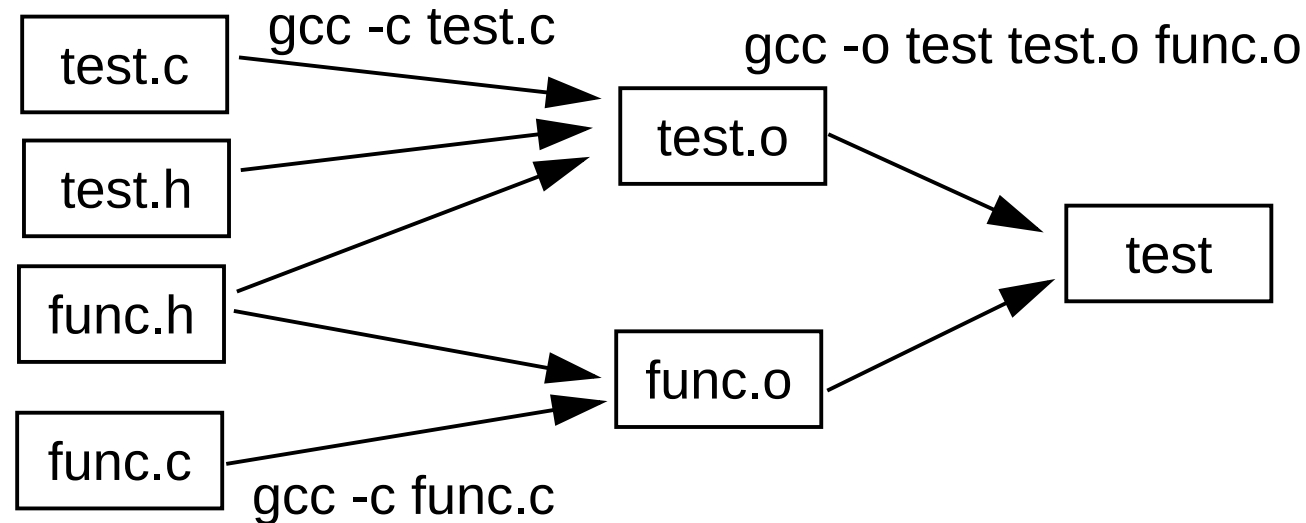
- Gängige Makros:

- `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

- 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`
- Programm sollte Debug-Symbole enthalten
- Aufruf des Debuggers mit `gdb <Programmname>`



Beispiel

```
void initArray(long *array, unsigned int size) {
    int i;
    for ( i=0; i<=size; i++ ) {
        array[i] = 0;
    }
}

int main(int argc, char *argv[]) {
    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++;
    }

    exit(EXIT_SUCCESS);
}
```



- 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 <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



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

- 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
- 128 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

- 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 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);
```

