

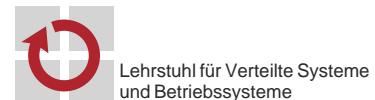
Übungen zu Systemprogrammierung 1

Ü3 – Freispeicherverwaltung

Sommersemester 2018

Christian Eichler, Jürgen Kleinöder

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



Aufgabe 1: lilo

Aufgabenstellung

- Einfach verkettete Liste aus Ganzzahlen (`int`)
- 2 Funktionen:

```
int insertElement(int value);  
int removeElement(void);
```

Agenda

- 4.1 Besprechung Aufgabe 1: lilo
- 4.2 Freispeicherverwaltung
- 4.3 Implementierung
- 4.4 make
- 4.5 gdb
- 4.6 Aufgabe 3: halde
- 4.7 Gelerntes anwenden

Vorstellen der Lösung von...

4.1 Besprechung Aufgabe 1: lilo

4.2 Freispeicherverwaltung

4.3 Implementierung

4.4 make

4.5 gdb

4.6 Aufgabe 3: halde

4.7 Gelerntes anwenden

4-4

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: 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

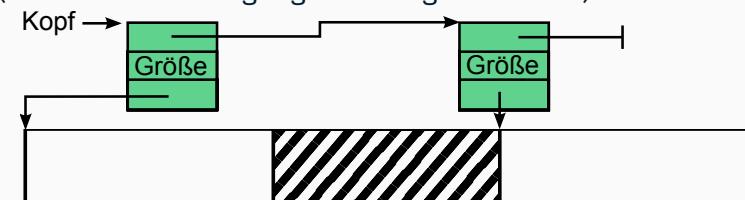
4-5

- 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: –

4-6

Konzept: Verkettete Liste zur Allokation

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



■ Listenelement
■ frei
■ belegt

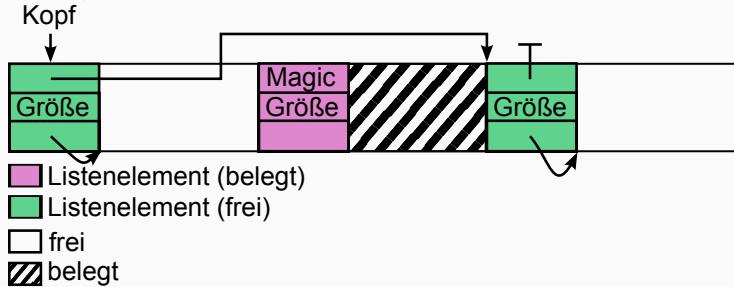
- Freie Blöcke werden in einer verketteten Liste gespeichert

- Wiederholung Übung 1

- Wie wird eine verkettete Liste in C implementiert?


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

- 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

4-7

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

4-9

4.1 Besprechung Aufgabe 1: lilo

4.2 Freispeicherverwaltung

4.3 Implementierung

4.4 make

4.5 gdb

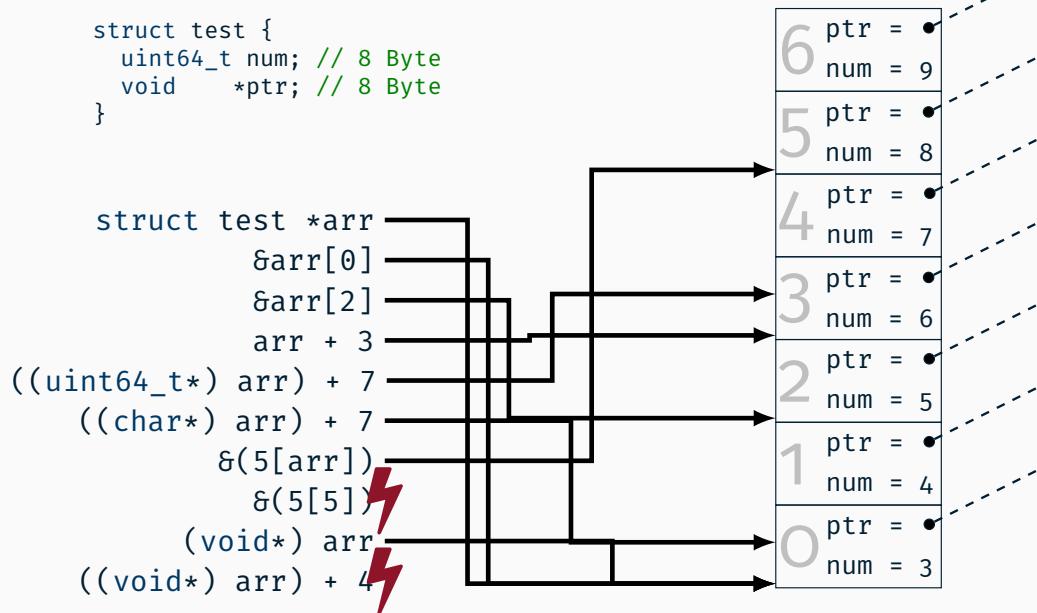
4.6 Aufgabe 3: halde

4.7 Gelerntes anwenden

Einschub: Zeiger und Zeigerarithmetik

```
struct test {
    uint64_t num; // 8 Byte
    void    *ptr; // 8 Byte
}
```

```
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
```



4-10

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

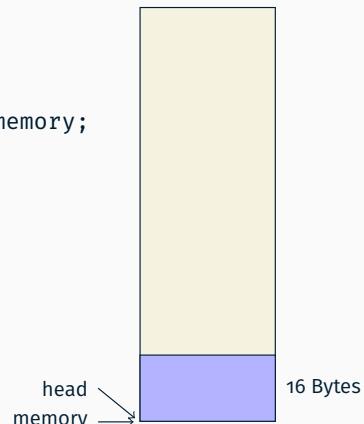
4-11

Initialisierung

- Speicher statisch alloziert
`static char memory[100];`

- struct mblock** reinlegen

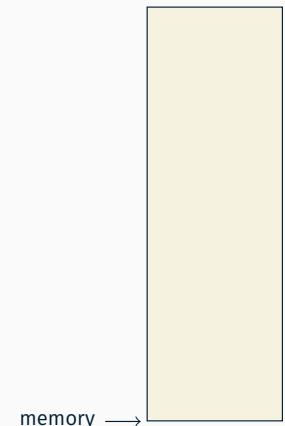
```
struct mblock* head = (struct mblock*) memory;
```



4-12

Initialisierung

- Speicher statisch alloziert
`static char memory[100];`



4-12

Initialisierung

- Speicher statisch alloziert
`static char memory[100];`

- struct mblock** reinlegen

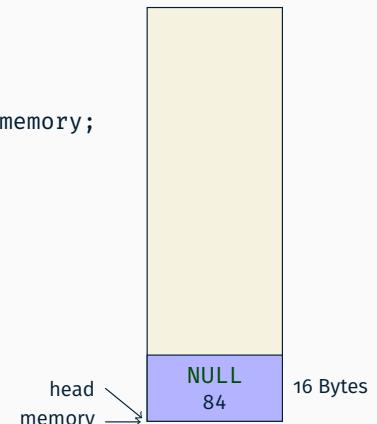
```
struct mblock* head = (struct mblock*) memory;
```

- struct mblock** initialisieren

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

- zwei Zeiger mit unterschiedlichem Typ auf den gleichen Speicherbereich

- unterschiedliche Semantik beim Zugriff
(Zeigerarithmetik, Strukturkomponenten)



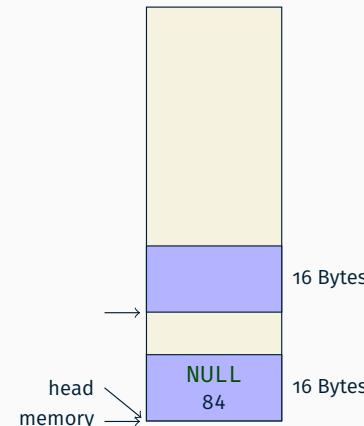
4-12

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



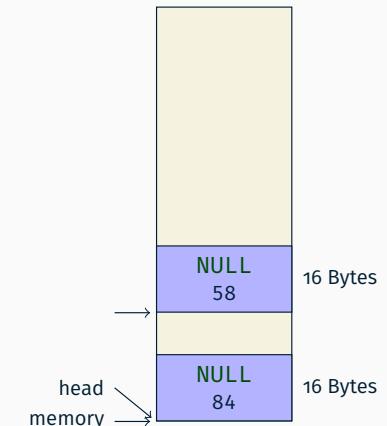
4-13

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



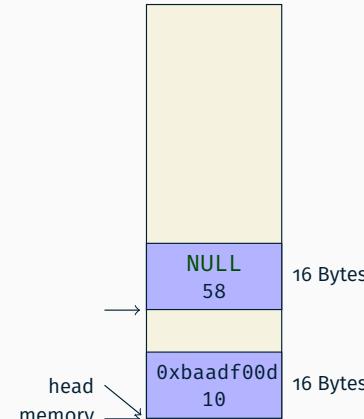
4-13

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



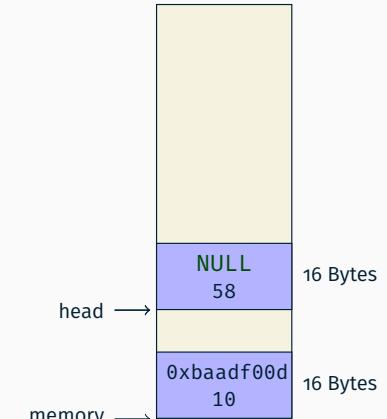
4-13

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

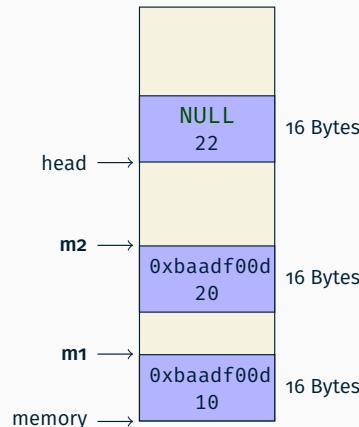


4-13

Speicheranforderung im Detail

■ Situation nach 2 malloc() -Aufrufen

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



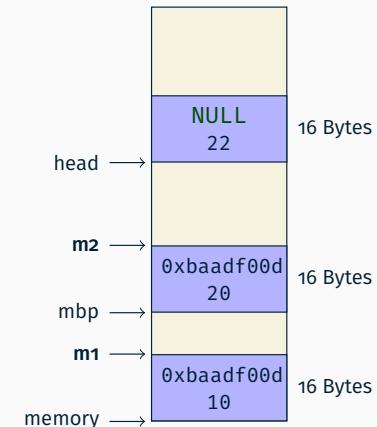
4-14

Speicherfreigabe

■ Freigabe von m2

```
free(m2);
```

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



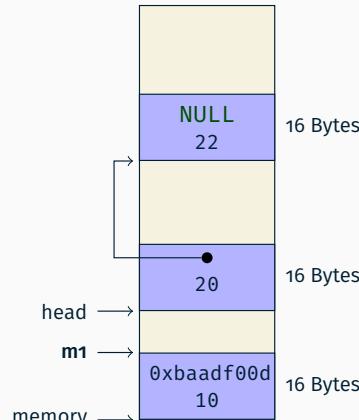
4-15

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



4-15

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

■ sinnvolle Implementierung erfordert geeignete Speichervergabestrategie

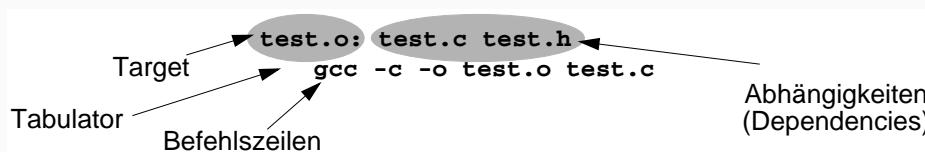
- Implementierung erheblich aufwändiger – Resultat aber entsprechend effizienter
- Strategien werden im Abschnitt Speicherverwaltung in SP2 behandelt (z. B. First-Fit, Best-Fit, Worst-Fit oder Buddy-Verfahren)

4-16

- 4.1 Besprechung Aufgabe 1: lilo
- 4.2 Freispeicherverwaltung
- 4.3 Implementierung
- 4.4 make
- 4.5 gdb
- 4.6 Aufgabe 3: halde
- 4.7 Gelerntes anwenden

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 erstellendes 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

- 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)

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)`

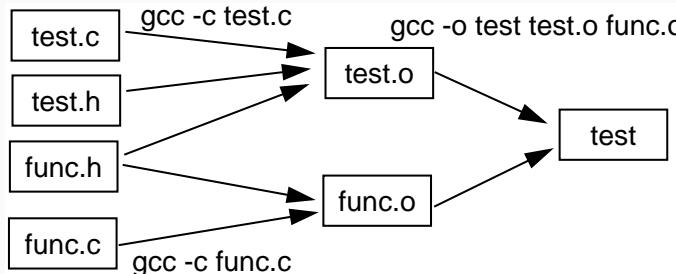
- Erzeugung neuer Makros durch Konkatenation

`ALLOBJS = $(OBJS) hallo.o`

- Gängige Makros:

- CC C-Compiler-Befehl
- CFLAGS Optionen für den C-Compiler

- Rechner beim Erzeugen von ausführbaren Dateien „entlasten“



- Zwischenprodukte verwenden und somit Übersetzungszeit sparen

- 4.1 Besprechung Aufgabe 1: lilo
- 4.2 Freispeicherverwaltung
- 4.3 Implementierung
- 4.4 make
- 4.5 gdb
- 4.6 Aufgabe 3: halde
- 4.7 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
 - mit GCC-Flag `-g` übersetzen
- Aufruf des Basis-Debuggers mit `gdb <Programmname>`
- Inklusive Visualisierung des Quelltextes: `cgdb <Programmname>`

Beispiel

```
static void initArray(long *array, unsigned int size) {  
    for(int i = 0; i <= size; ++i) {  
        array[i] = 0;  
    }  
}  
  
int main(int argc, char *argv[]) {  
    long buf[7];  
    long *array = buf;  
  
    initArray(buf, sizeof(buf)/sizeof(long));  
  
    while(array != buf + sizeof(buf)/sizeof(long)) {  
        printf("%ld\n", *array);  
        array++;  
    }  
  
    return 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 <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: Stoppt bei jedem Zugriff (kombiniert watch und rwatch)
 - Anzeigen und Löschen analog zu den Breakpoints

Agenda

4.1 Besprechung Aufgabe 1: lilo

4.2 Freispeicherverwaltung

4.3 Implementierung

4.4 make

4.5 gdb

4.6 Aufgabe 3: halde

4.7 Gelerntes anwenden

4-25

4-26

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

4-28

- 4.1 Besprechung Aufgabe 1: lilo
- 4.2 Freispeicherverwaltung
- 4.3 Implementierung
- 4.4 make
- 4.5 gdb
- 4.6 Aufgabe 3: halde
- 4.7 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

■ „Hier könnte Ihre Lösung stehen“

- ⇒ In den Tafelübungen wird die Lösung gezeigt
- ⇒ Besucht die Tafelübungen, es lohnt sich! ;-)