Übungen zu Systemprogrammierung 2 (SP1+2)

Ü4 – Freispeicherverwaltung

Christoph Erhardt, Jens Schedel, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

WS 2013/14 – 28. bis 31. Oktober 2013

http://www4.cs.fau.de/Lehre/WS13/V_SP1+2

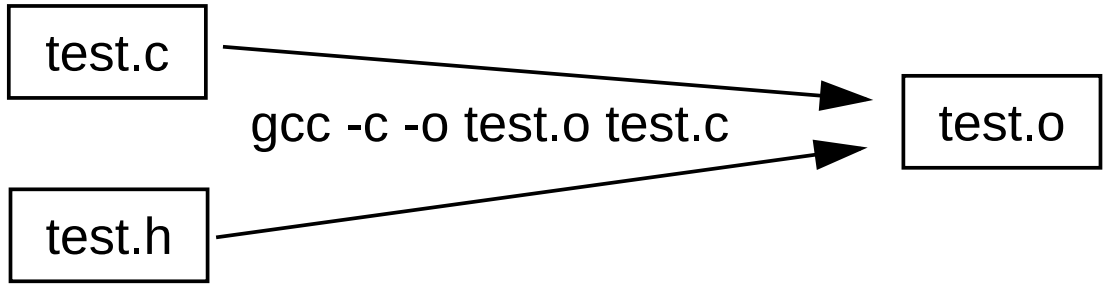
Agenda

4.1 make
4.2 gdb

Agenda

Make – Teil 1

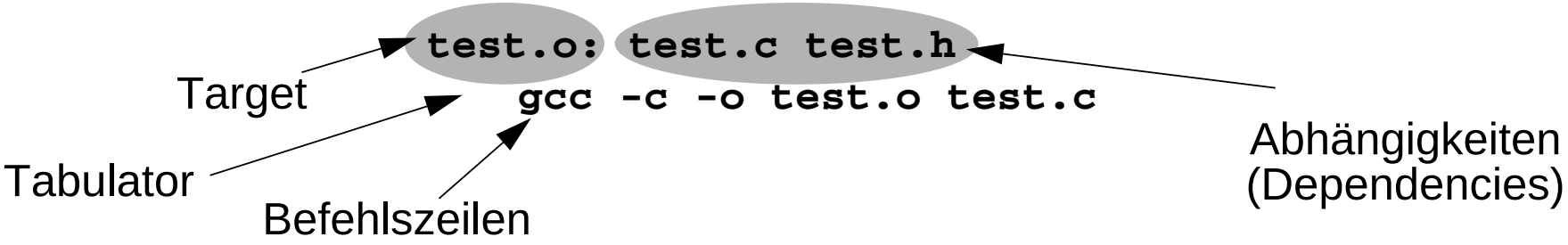
- Grundsätzlich: Erzeugung von Dateien aus anderen Dateien
  - für uns interessant: Erzeugung einer .o-Datei aus einer .c-Datei



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

Funktionsweise

- Regeldatei mit dem Namen `Makefile`



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

Makros

- In einem Makefile können Makros definiert werden

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

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

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

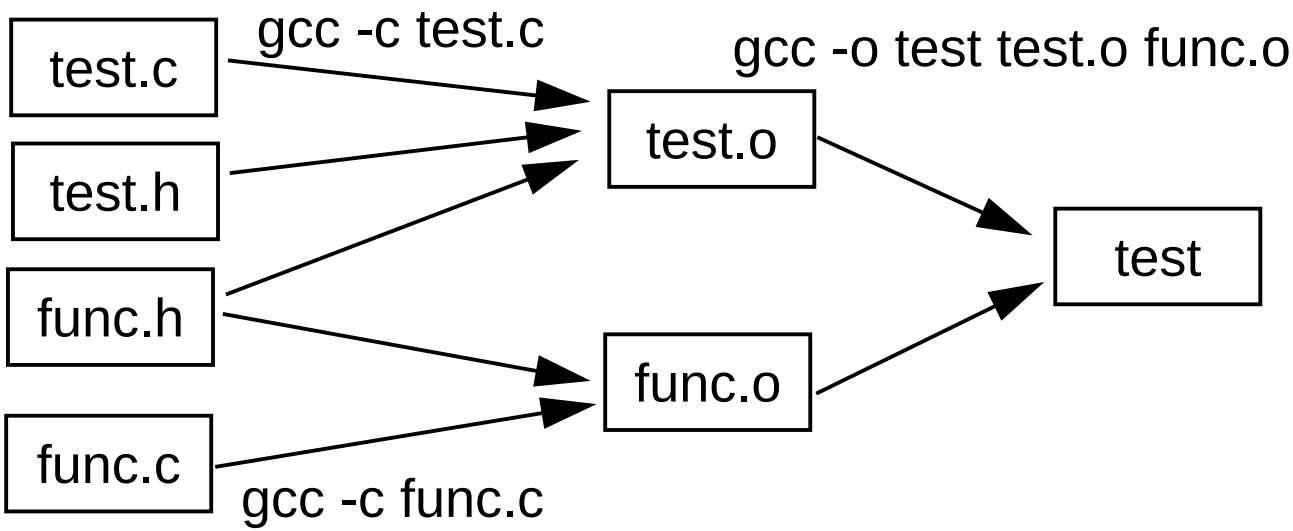
- Erzeugung neuer Makros durch Konkatenation

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

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

Schrittweises Übersetzen

- Rechner beim Erzeugen von ausführbaren Dateien „entlasten”



- Zwischenprodukte verwenden und somit Übersetzungszeit sparen

Agenda

Debugger: gdb

- Ein Debugger dient zum Suchen und Finden von Fehlern in Programmen
- Im Debugger kann man u.a.
  - das Programm schrittweise abarbeiten
  - Variablen- und Speicherinhalte ansehen und modifizieren
  - core dumps (Speicherabbilder beim Programmabsturz) analysieren
    - Erlauben von core dumps (in der laufenden Shell): z. B. `limit coredumpsize 1024k` oder `limit coredumpsize unlimited`
- Programm sollte Debug-Symbole enthalten
- Aufruf des Debuggers mit `gdb <Programmname>`

Beispiel

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

int main(int argc, char *argv[]) {
  long *array;
  long buf[7];
  array = buf;

  initArray(buf,sizeof(buf)/sizeof(long));

  while ( array != buf+sizeof(buf)/sizeof(long) ) {
    printf("%ld\n", *array);
    array++;
  }

  exit(EXIT_SUCCESS);
}
```


Befehlsübersicht

- Programmausführung beeinflussen
  - Breakpoints setzen:
    - `b [<Dateiname>:]<Funktionsname>`
    - `b <Dateiname>:<Zeilennummer>`
  - Starten des Programms mit `run` (+ evtl. Befehlszeilenparameter)
  - Fortsetzen der Ausführung bis zum nächsten Stop mit `c` (continue)
  - schrittweise Abarbeitung auf Ebene der Quellsprache mit
    - `s` (step: läuft in Funktionen hinein)
    - `n` (next: behandelt Funktionsaufrufe als einzelne Anweisung)
  - Breakpoints anzeigen: `info breakpoints`
  - Breakpoint löschen: `delete breakpoint#`

Befehlsübersicht

- Variableninhalte anzeigen/modifizieren
  - Anzeigen von Variablen mit: `p expr`
    - `expr` ist ein C-Ausdruck, im einfachsten Fall der Name einer Variable
  - Automatische Anzeige von Variablen bei jedem Programmstopp (Breakpoint, Step, ...): `display expr`
  - Setzen von Variablenwerten mit `set <variablenname>=<wert>`
- Ausgabe des Funktionsaufruf-Stacks (backtrace): `bt`
- Quellcode an aktueller Position anzeigen: `list`
- Watchpoints: Stoppt Ausführung bei Zugriff auf eine bestimmte Variable
  - `watch expr`: Stoppt, wenn sich der Wert des C-Ausdrucks `expr` ändert
  - `rwatch expr`: Stoppt, wenn `expr` gelesen wird
  - `awatch expr`: Stopp bei jedem Zugriff (kombiniert `watch` und `rwatch`)
  - Anzeigen und Löschen analog zu den Breakpoints