34 Überblick über die 7. Übung

- make
- gdb
- (insure)

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.

test.c —— cc -c -o test.o test.c ——▶ test.o

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

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

Target
Tabulator
Befehlszeilen
Abhängigkeiten (Dependencies)

Beispiel

```
test: test.o func.o
    ld -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
```


Make (2)

- 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)

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)
```


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)

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)
```


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
  - **`LDFLAGS`** Optionen für den Linker
- Wichtige Regeln:
  - **`.c.o`** C-Datei in Objektdatei übersetzen
  - **`.c`** C-Datei übersetzen und linken

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 $<
```


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 $@: $?
    $(LD) $(LDFLAGS) -o $@ $(OBJS)

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


test.o: test.c $(HEADER)

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

Nützliche Konvention

- 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
```


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

Debuggen mit dem gdb

- 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)

Debuggen mit dem gdb

- Anzeigen von Variablen mit **p** <variablenname>
- Setzen von Variablenwerten mit **set** <variablenname>**=**<wert>
- Ausgabe des Stack-Traces: **bt**
- Navigieren zwischen den Stackframes: **up**, **down**

Emacs und gdb

- gdb läßt sich am komfortabelsten 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`**

Electric Fence

- Speicherprobleme (SIGSEGV!) lassen sich mit der Electric Fence-Bibliothek gut finden:

```
gcc -g -o hello hello.c -L/proj/i4sp/pub/efence -lefence
```

- Programm danach im Debugger laufen lassen
-