U2 2. Übung

U2-1 Überblick

- Aufgabe 2: **qsort(3)**, Wettbewerb
- Debugging mit GDB und valgrind
- Übersetzen von Projekten mit **make(1)**

U2-2 Aufgabe 2: Sortieren mittels qsort

1 Funktion *qsort(3)*

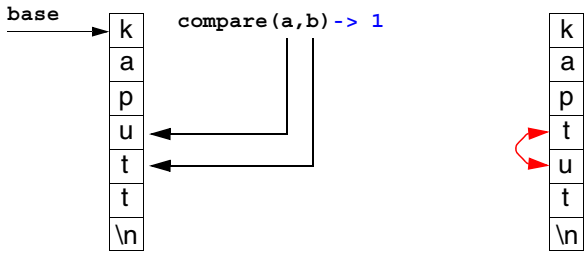
- Prototyp aus `stdlib.h`:

```
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

4 wsort-Wettbewerb

- Wer implementiert das schnellste **wsort**?
- Entkoppelt von Aufgabe 2 ☞ eigene Aufgabe **contest**
  - Projektverzeichnis `/proj/i4sp/$LOGIN/contest/`
  - Abgabe (bis 27.11.2008, 14:00): `/proj/i4sp/pub/abgabe contest`
  - `Makefile` welches ein Programm `wsort` aus beliebigen weiteren Dateien baut, unter Verwendung eurer gewünschten Compiler-Flags!
- Teilnahme allein oder in Teams beliebiger Grösse
- Die schnellste Lösung wird mit einem Kasten Zirndorfer prämiert!
- Ideenaustausch und Bekanntgabe von Zwischenzeiten im Forum
  ☞ `http://fsi.informatik.uni-erlangen.de/forum`
- Unser Pferd im Rennen: `psort` - unschlagbar? :-)
  - sollte `psort` überraschenderweise geschlagen werden winkt ein Überraschungspreis

4 wsort-Wettbewerb: Regeln

- Vollständige Funktionalität entsprechend Aufgabenstellung 2
- Korrektheit bei allen Eingaben
- Compilerflags sind frei wählbar
- Zeitmessung
  - Referenzrechner: faui06[a-o] (Core 2 Quad Q6600, 8 GB RAM)
  - Pro Wortliste werden mehrere Durchläufe in Reihe gestartet, die kürzeste Zeit gilt
  - Relevant ist die **Summe der Real-Zeiten** für **wlist5** und **wlist6**
  - Testen mit: `/proj/i4sp/pub/contest/runtest.sh`

4 wsort-Wettbewerb: Ansatzpunkte für Speedups

- Compiler-Flags
  Optimierungsstufe, Zielarchitektur
- Einlesen und Ausgeben
  Anzahl der Lese- bzw. Schreiboperationen
- Speicherverwaltung
  Wenige Kopieraktionen bzw. Reallokierungen
- Parallelisierung
  Arbeit auf mehrere Threads verteilen
- Sortierverfahren
  qsort(3) ist nicht langsam, aber nicht für alle Zwecke optimal

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` [<Dateiname>]:<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)
- Breakpoints anzeigen: `info breakpoints`
- Breakpoint löschen: `delete` <breakpoint-nummer>

2 Variablen, Stack

- Anzeigen von Variablen mit `p` ***`expr`***
- **expr** ist ein C-Ausdruck, im einfachsten Fall der Name einer Variable
- Automatische Anzeige von Variablen bei jedem Programmhalt (Breakpoint, Step, ...) mit `display` ***`expr`***
- Setzen von Variablenwerten mit `set` <variablenname>=<wert>
- Ausgabe des Funktionsaufruf-Stacks (backtrace): `bt`

3 Watchpoints

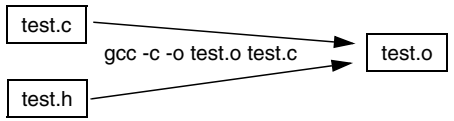
- Stoppt Ausführung bei Zugriff auf eine bestimmte Variable
- `watch` ***`expr:`*** Stoppt, wenn sich der Wert des C-Ausdrucks **expr** ändert
- `rwatch` ***`expr:`*** Stoppt, wenn **expr** gelesen wird
- `awatch` ***`expr:`*** Stopp bei jedem Zugriff (kombiniert `watch` und `rwatch`)
- **expr** ist ein C-Ausdruck, im einfachsten Fall ein Variablenname
- Anzeigen und Löschen analog zu den Breakpoints

U2-4 valgrind

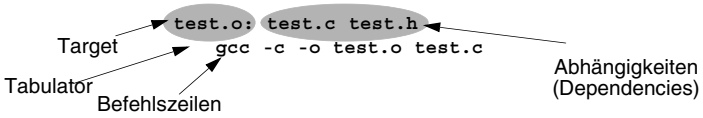
- Baukasten von Debugging- und Profiling-Werkzeugen (ausführbarer Code wird durch synthetische CPU auf Softwareebene interpretiert ➔ Ausführung erheblich langsamer!)
  - Memcheck: erkennt Speicherzugriff-Probleme
    - ➤ Nutzung von nicht-initialisiertem Speicher
    - ➤ Zugriff auf freigegebenen Speicher
    - ➤ 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

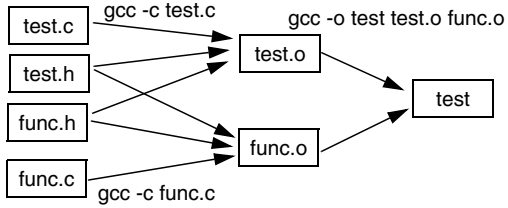
- Erzeugung von Dateien aus anderen Dateien.
  - z.B. Erzeugung einer .o-Datei aus einer C-Datei durch C-Compiler



- Ausführung von *Update*-Operationen
- Regeldatei mit dem Namen `Makefile`:
  - Targets (was?) und Abhängigkeiten (woraus?)
  - Befehlszeilen (wie?)



1 Beispiel mit mehreren Modulen




```
test: test.o func.o
    gcc -o test test.o func.o

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

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


2 Allgemeines

- Kommentare beginnen mit `#` (bis Zeilenende)
- Befehlszeilen müssen mit TAB beginnen
- Nach der letzten Befehlszeile einer Regel muss eine Zeile folgen, die weder mit TAB noch mit `#` beginnt
- das zu erstellende Target kann beim `make`-Aufruf angegeben werden (z.B. `make test`)
  - ohne explizite Target-Angabe bearbeitet make das erste Target im Makefile
- beginnt eine Befehlszeile mit `@` wird sie nicht ausgegeben
- beginnt eine Befehlszeile mit `-`, führt ein Fehlschlagen nicht zum Abbruch
- jede Zeile wird in einer neuen Shell ausgeführt
  - `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)
    gcc -o test $(SOURCE)
```

- Erzeugen neuer Makros durch Ersetzung in existierenden Makros

```
OBJS = $(SOURCE:.c=.o)
```

  - In allen Wörten, die auf den Suchstring **.c** enden, wird dieser durch **.o** ersetzt

- Erzeugung neuer Makros durch Konkatenation

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

4 Dynamische Makros

- `$@` Name des Targets (hier: *test*)

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

- `$*` Basisname des Targets (ohne Dateiendung, hier *test*)

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

- `$<` Name einer Abhängigkeit (in impliziten Regeln)

5 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 gcc verwendet, weil direkter Aufruf von ld die Standard-Bibliotheken nicht mit einbindet - gcc 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

6 Suffix Regeln

- Allgemeine Regel zur Erzeugung einer Datei mit einer bestimmten Endung aus einer gleichnamigen Datei mit einer anderen Endung.
- Beispiel: Erzeugung von **.o**-Dateien aus **.c**-Dateien

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

- Dateiendungen müssen deklariert werden, als Abhängigkeiten des Spezialtargets `.SUFFIXES`

```
.SUFFIXES: .c .o
```

- Explizite Regeln überschreiben die Suffix-Regeln

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

- Regeln ohne Kommandos können Abhängigkeiten überschreiben

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

  - die Suffix-Regel wird weiterhin zur Erzeugung herangezogen

7 Beispiel verbessert

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

test: $(OBJS)
    $(CC) $(LDFLAGS) -o $@ $(OBJS)

# Suffix-Regeln
.SUFFIXES: .c .o

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

# korrekte Abhaengigkeiten
test.o: test.c $(HEADER)

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

8 GNU Make Erweiterungen

- Funktionsumfang von POSIX.2 make sehr eingeschränkt
- viele Make Implementierungen mit z.T. inkompatiblen Erweiterungen
  - BSD Make (verschiedene Variationen)
  - Sun Make (Solaris)
  - Microsoft nmake
  - smake
  - GNU Make (gmake), installiert als *make* im CIP-Pool

9 Pseudo-Targets

- Dienen nicht der Erzeugung einer gleichnamigen Datei
- Deklaration als Abhängigkeiten des Spezial-Targets **.PHONY**

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

  - so deklarierte Targets werden immer gebaut, auch wenn eine gleichnamige Datei bereits existiert, die aktueller als die Abhängigkeiten ist
- Aufräumen mit `make clean`

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

- Projekt bauen mit `make all` (Konvention: `all` ist immer erstes Target)

```
all: test
```

- Installieren mit `make install`

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


10 Eingebaute Funktionen

- Ausgabe eines Shell-Kommandos einem Verzeichnis zuweisen

```
CURRENTDIR = $(shell pwd)
```

- Dateinamen nach einem Shell-Wildcard-Muster suchen

```
SOURCE = $(wildcard *.c)
```

11 Dynamische Makros

- `$^` Mit Leerzeichen getrennte Liste aller Abhängigkeiten

12 Einbinden anderer Makefiles

- `include`-Anweisung (am Zeilenanfang, ohne Tabulator)

```
include /proj/i4spic/xxx.mk
```

- die Datei wird an Stelle der include-Anweisung eingebunden
- Zusatzinfo für Fortgeschrittene:
  - inkludierte Dateien können make-Targets sein
  - Make wird diese dann wenn nötig erst aktualisieren
  - Makefiles können sich so selbst generieren
  - z.B. dynamische Erzeugung von Abhängigkeiten mit

```
gcc -MM test.c > test.dep
```

  - Einbinden der so erzeugten Dependencies

```
-include test.dep
```

    '**-**' unterdrückt hierbei die Warnung, wenn das **.dep**-File zunächst nicht vorhanden ist