3. Programmierwerkzeuge

AlDaBi Praktikum

David Weese

Sascha Meiers
WS 2013/14

Inhalt

- Build System
- Debugger
- Entwicklungsumgebung
- Profiler
- Laufzeit messen
- Memory Debugger

- Bemerkungen zur P-Aufgabe

BUILD SYSTEM

David Weese

Sascha Meiers
WS 2013/14

Allgemeines

Programmbau mit Make:

- Änderungen implizieren oft mehrere Zwischenschritte, um das Programm/Produkt zu bauen:
  - Übersetzen von Quelldateien in Objektdateien
  - Binden von Objektdateien zu ausführbaren Programmen
  - Erzeugen der Dokumentation aus den Quelldateien
- Zwischenschritte können von anderen abhängen (Abhängigkeitsgraph)

- Makefiles definieren:
  - welche Komponenten es gibt
  - wovon sie abhängen
  - Schritte zur Konstruktion der Komponenten

Makefile: Regeln

- für ein oder mehere Komponenten
- Abhängigkeiten
- Befehle

```
ziel1 ziel2 … zieln: quelle1 quelle2 … quellem
   kommando1
   kommando2
   kommando3
…
```

- **Befehle müssen mit TABS eingerückt sein!**

Beispiel: duden

- Programm duden besteht aus zwei Komponenten:
  grammatik.c und woerterbuch.c

- Für beide Komponenten:
  C-Quelldatei wird mit Hilfe von cc in Objektdatei übersetzt

- Binden der Objektdateien zum ausführbaren Programm

- Zu tun wäre:

```
cc grammatik.c -c -o grammatik.o
cc woerterbuch.c -c -o woerterbuch.o
cc grammatik.o woerterbuch.o -o duden
```

Makefile für duden

```
duden: grammatik.o woerterbuch.o
	cc grammatik.o woerterbuch.o -o duden

grammatik.o: grammatik.c
	cc grammatik.c -c -o grammatik.o

woerterbuch.o: woerterbuch.c
	cc woerterbuch.c -c -o woerterbuch.o

clean:
	rm grammatik.o woerterbuch.o duden
```

oder kürzer (implizite Regeln)

```
duden: grammatik.o woerterbuch.o
	cc grammatik.o woerterbuch.o -o duden

clean:
	rm grammatik.o woerterbuch.o duden
```

Funktionsweise von make

- Berechne Abhängigkeitsgraphen
- Für Zielkomponente A
  - bestimme Komponenten $A_1, \ldots, A_n$ von denen A abhängt
  - rufe Algorithmus für alle A_i rekursiv auf
  - falls A nicht existiert, oder ein A_i neu gebaut/verändert wurde: erzeuge A mit Kommandos
- Erkennen von Änderungen einer Datei
  - Datum der letzten Änderung wird verwaltet
  - Datei geändert, falls jünger als von ihr abhängige Komponente

Warum eigentlich?

- Alles neu zu kompilieren kann sehr lange dauern

Xkcd.com

- Dank make werden nur die wirklich notwendigen Komponenten kompiliert
- Und: einfacher Aufruf `make duden`

Features von **make**

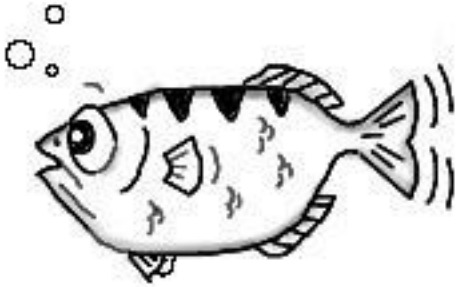
- Variablen `var=wert`
- Referenzierung durch `$(var)`
- Implizite Regeln
  - Definition von Standardregeln für Komponenten mit best. Namensmuster
  - Vordefiniert ist bspw:

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

- Implizite Variablen für erstes Ziel `$@` oder Quelle `$<`
- If-Anweisungen und Makros

- GNU Make Manual - http://www.gnu.org/software/make/manual/

DEBUGGER



David Weese

Sascha Meiers
WS 2013/14

Fehlersuche mit **gdb** und **ddd**

- Ein Debugger führt ein Programm kontrolliert aus
  - Programm in definierter Umgebung ausführen
  - Programm unter bestimmten Bedingungen anhalten lassen
  - Zustand eines angehaltenen Programms untersuchen
  - Zustand eines angehaltenen Programms verändern
- GNU Debugger **gdb**
  - interaktives Programm mit Kommandozeilensteuerung
- Data Display Debugger **ddd**
  - graphische Benutzeroberfläche zu gdb

- Programmbeispiel:
  - Es sollen Zahlen von der Kommandozeile eingelesen, sortiert und wieder ausgegeben werden

```c
#include <stdio.h>
#include <stdlib.h>

void shell_sort(int a[], int size) {
   int i, j;
   int h = 1;

   do {
    h = h * 3 + 1;
   } while (h <= size);

   do {
    h /= 3;
    for (i = h; i < size; i++) {
       int v = a[i];
       for (j = i; j >= h && a[j-h] > v; j -= h)
        a[j] = a[j - h];
       if (i != j)
          a[j] = v;
    }
   } while (h != 1);
}
```

```c
int main (int argc, char *argv[]) {
   int *a;
   int i;

   a = (int *) malloc((argc - 1) * sizeof(int));
   for (i = 0; i < argc - 1; i++)
    a[i] = atoi(argv[i+1]);

   shell_sort(a, argc);

   for (i=0; i < argc - 1; i++)
    printf ("%d ", a[i]);
   printf ("\n");

   free (a);
   return 0;
}
```

Beispielprogramm sample.c

Ausführung von sample

- Manchmal geht's:

```
$ ./sample 1 8 5 3 4
1 3 4 5 8
$
```

- ... und manchmal nicht:

```
$ ./sample 1 8 5 3 4 7
0 1 3 4 5 7
$
```

Programme debuggen mit **ddd**

- Übersetzen mit *Debugging-Informationen*:
  - -g für Debugsymbole
  - Es empfiehlt sich keine Optimierung zu benutzen (-O, -O2, -O3)

```
$ cc -g sample.c -o sample
$
```

- Starten der Debugging-Sitzung:

```
$ ddd ./sample&
$
```

- Breakpoint in Zeile 31 und **run** mit **1 8 5 3 4** als Kommandozeile
- *View->Data Window* anzeigen
- Im *Data Window* Rechtsklick *New Display* und ***a @ 6** hinzufügen
- Schrittweise debuggen

Debugger steuern

- Ausführung steuern:
  - **Run** startet Debugger
  - **Step** führt einzelne Zeile aus und springt in Subroutinen
  - **Next** führt einzelne Zeile aus und überspringt Subroutinen
  - **Until** springt aus Schleifen raus
  - **Finish** springt aus Subroutinen zurück zum Aufrufer

- Variablen/Ausdrücke anzeigen:
  - Variable anzeigen `print i`
  - Arrayelement anzeigen `print a[3]`
  - Die ersten 6 Elemente eines Arrays anzeigen `print a[0]@6`

- Ausdrücke können als *New Display* im *Data Window* hinzugefügt werden

Siehe da!

shell_sort(a, argc) muss zu

shell_sort(a, argc – 1) zu geändert werden.

Funktionsweise des **gdb**

- Betriebssystem erlaubt Kontrolle der Programmausführung
- Verbindung zwischen Programmspeicher und Originalquelltext
  - Debugging-Informationen enthalten Symbolnamen, Typinfos und Zeilennummern

```
$ cc -S -g sample.c
$ less sample.s
…
        .globl  main
        .type   main, @function
main:
.LFB1:
        .loc 1 29 0
```

← **main**-Funktion beginnt in Zeile 29

Features von Debuggern

- Ändern der Programmausführung
  - Die Kommandos return und jump

- Ändern des Programmcodes

- Post-Mortem-Debugging
  - Untersuchen des letzten Zustands vor Programmabsturz

```
$ gdb ./sample core
```

- gdb Documentation - http://sourceware.org/gdb/documentation/
- ddd Documentation - http://www.gnu.org/manual/ddd/

ENTWICKLUNGSUMGEBUNG

David Weese

Sascha Meiers
WS 2013/14

Allgemeines

- Frei verfügbare C++ IDEs:
  - Microsoft Visual Studio Express (www.microsoft.com)
  - Eclipse (www.eclipse.org)
  - Kdevelop (www.kdevelop.org)
  - Xcode (developer.apple.com/tools/xcode/)
  - Emacs, Anjuta, Vim...

- IDE (Integrated Development Environment) besteht aus:
  - Texteditor
  - Compiler
  - Linker
  - Debugger

Windows

MS Vidual Studio

- Express-Version für alle frei verfügbar
- VS 2003/2005/2008/2010 frei für Studenten über FU-MSDNAA
- FU-MSDNAA - https://msdnaa.mi.fu-berlin.de/
- MSDN Visual C++ Reference - Übersicht, Language Reference

Unix Systeme

Workflow auf der Konsole:

- Text editieren mit nano, vim o.Ä.
  - Oder mit GUI-Programmen: gedit, Emacs, kate, gvim, …
  - Viele unterstützen *code highlighting*, manche *code completion*
- Compilieren mit g++, clang++ etc.
  - Noch komfortabler: make
  - Bei ganz großen Projekten: cmake (erzeugt Makefiles)
  - make kann z.B. in vim aus dem Editor aufgerufen werden
- Debugging mit gdb
  - Oder graphischen Interfaces: kgdb, ddd,…

Komplette IDE:

- z.B. Eclipse

Mac

- Die meisten Konsolenprogramme stehen ebenfalls zur Verfügung
- Xcode
  - Gut ausgestattete IDE
  - Enthält neben Text-Editor, Compiler und Debugger auch Profiling-Instrumente

PROFILER

David Weese

Sascha Meiers
WS 2013/14

Laufzeitanalyse

- Profiler
  - Programmierwerkzeuge, die das Laufzeitverhalten von Software analysieren

- Laufzeitanalyse
  - In welchen Funktionen (Befehlen) wird die meiste Laufzeit verbraucht?
  - An welchen Stellen sind Leistungssteigerungen möglich?
  - Wie wirken sich Änderungen am Code auf die Laufzeit aus?

- Profile
  - Programm erstellt Programmprofil während des Programmablaufs
  - Programmprofil gibt für jede Funktionen an
    - wie oft ausgeführt
    - wie lange ausgeführt

GNU Profiler **gprof**

- Übersetzen mit *eingerichteter Profilierung* (Optimierung optional)

  $ g++ **–pg** aufgabe2.cpp –o aufgabe2
- Starten des Programms, Profil wird in Datei **gmon.out** geschrieben:

  $ ./aufgabe2
- Analysieren des Profils mit **gprof**:

  $ gprof aufgabe2

- Flaches Profil: Verteilung der Laufzeit auf Funktionen
- Strukturiertes Profil:
  - Zusätzlich Aufruf-Abhängigkeiten zwischen den Funktionen

- C/C++-Programme optimieren mit dem Profiler gprof (linuxfocus.org)
- gprof Documentation - http://sourceware.org/binutils/docs/

Wie funktioniert gprof?

- Einbauen von Befehlen zur Zeitmessung
  - Vor und nach jedem Funktionsaufruf
  - Zählt die Funktionsaufrufe (und von wo aus diese kommen)
  - Zeit die in der Funktion verbracht wird, wird gemessen

- Nachteile:
  - Code wird verfälscht
  - Sprung in die Funktion und heraus wird nicht bedacht
  - Parallelität der CPU wird ignoriert

- Alternative: Sample based methods
  - Prüfen in regelmäßigen Abständen von außen, wo sich das Programm gerade befindet
  - Siehe: perf

Performance Analyser **perf**

- „Performance counter for Linux“
- Zählt für jede Funktion verschiedene Events
  - Laufzeit in CPU-Zyklen (CPU time, keine Wall-clock-time)
  - cache misses
  - branch mispredictions
- Sample based:
  - Perf zieht in einer gewissen Frequenz „Proben“ wo sich das Progamm gerade befindet
  - Test an großen Daten sinnvoll (längere Laufzeit)
  - Frequenz kann beeinflusst werden über -F

Beispiel mit **perf**

- ./aufgabe2.cpp small.fa ACGACT
  - Liest eine Text-Datei small.fa ein, baut ein Suffix-Array und sucht die Sequenz ACGACT darin

- Kompilieren mit Debugging-Informationen und Optimierung
  - `g++ -g -O2 aufgabe2.cpp -o aufgabe2`
  - Keine Profile-Symbole (`-pg`)

- Aufnehmen (mit root Rechten)
  - `perf record -F 10000 ./aufgabe2 small.fa ACGACT`
- Anzeigen:
  - `perf report`

- Achtung beim inlining von Funktionen (O2 / O3)

Tuning

- Auswerten von Laufzeitprofilen
  - Problematische Funktionen/Befehle identifizieren
- Änderungen an den problematischen Stellen
  - Cache-Optimierung, Vorberechnungen, uvm.
  - Mehr zu Effizienz in einer der nächsten Vorlesungen
- Profilanalyse und Tuning sollten aber einer der letzten Schritte sein

„Premature Optimization is the source of all evil“

Donald Knuth

MEMORY DEBUGGER

David Weese

Sascha Meiers
WS 2013/14

Speicheranalyse mit **valgrind**

- Valgrind ist ein leistungsfähiges Toolset für
  - Profiling
  - Memory Debugging
  - Memory/Cache Profiling
  - Thread Debugging
- Memory Debugger – sucht Fehler im Speicher-Management von Programmen
- Zu testendes Programm mit Debugging-Informationen übersetzen. (Parameter "-g" beim gcc)

So geht’s

- Übersetzen mit Debugging-Informationen:

```
$ cc -g sample.c -o sample
$
```

- Starten der Debugging-Sitzung:

```
$ valgrind --leak-check=yes ./sample 1 5 8 6 3
$
```

Da war doch was…

- Wir wissen bereits wo der Fehler liegt:
  - Aufruf von mit
    shell_sort(a, argc)
  - Korrekt wäre
    shell_sort(a, argc-1)

- Findet valgrind den Fehler?

Valgrind Ausgabe

```
==20359== Invalid read of size 4
==20359==    at 0x80484F4: shell_sort (sample.c:18)
==20359==    by 0x80485EB: main (sample.c:38)
==20359==  Address 0x41fc040 is 0 bytes after a block of size 24 alloc'd
==20359==    at 0x402BE68: malloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==20359==    by 0x8048593: main (sample.c:33)
==20359==
==20359== Invalid write of size 4
==20359==    at 0x804851E: shell_sort (sample.c:20)
==20359==    by 0x80485EB: main (sample.c:38)
==20359==  Address 0x41fc040 is 0 bytes after a block of size 24 alloc'd
==20359==    at 0x402BE68: malloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==20359==    by 0x8048593: main (sample.c:33)
==20359==
0 1 3 4 5 6
==20359==
==20359== HEAP SUMMARY:
==20359==     in use at exit: 0 bytes in 0 blocks
==20359==   total heap usage: 1 allocs, 1 frees, 24 bytes allocated
==20359==
==20359== All heap blocks were freed -- no leaks are possible
==20359==
==20359== For counts of detected and suppressed errors, rerun with: -v
==20359== ERROR SUMMARY: 4 errors from 2 contexts (suppressed: 0 from 0)
```

Fehlerarten

- Valgrind erkennt:
  - Illegale Zugriffe:
    - Speicher wurde nicht initialisiert
    - Zugriff außerhalb reservierten Speichers
  - Allocation/Deallocation Mismatches:
    - mit **new** alloziert und mit **free** (anstatt **delete**) freigegeben
    - Feld mit **delete** (anstatt **delete**[]) freigegeben
  - Memory Leaks – nicht freigegebener Speicher
  - Double Frees – doppelt freigegebener Speicher

Funktionsweise von **valgrind**

- Virtuelle Machine mit Just-In-Time Compilierung
  - Übersetzt Binärcode des Programms in plattform-unabhängigen Byte-Code
  - Dieser sog. Ucode wird von Valgrind-Tools modifiziert
  - Danach rückübersetzt und ausgeführt
- Dadurch lassen sich beliebige Programme analysieren
- Laufzeit ist aber um ein Vielfaches größer
- Valgrind Documentation - http://valgrind.org/docs/

LAUFZEIT MESSEN

David Weese

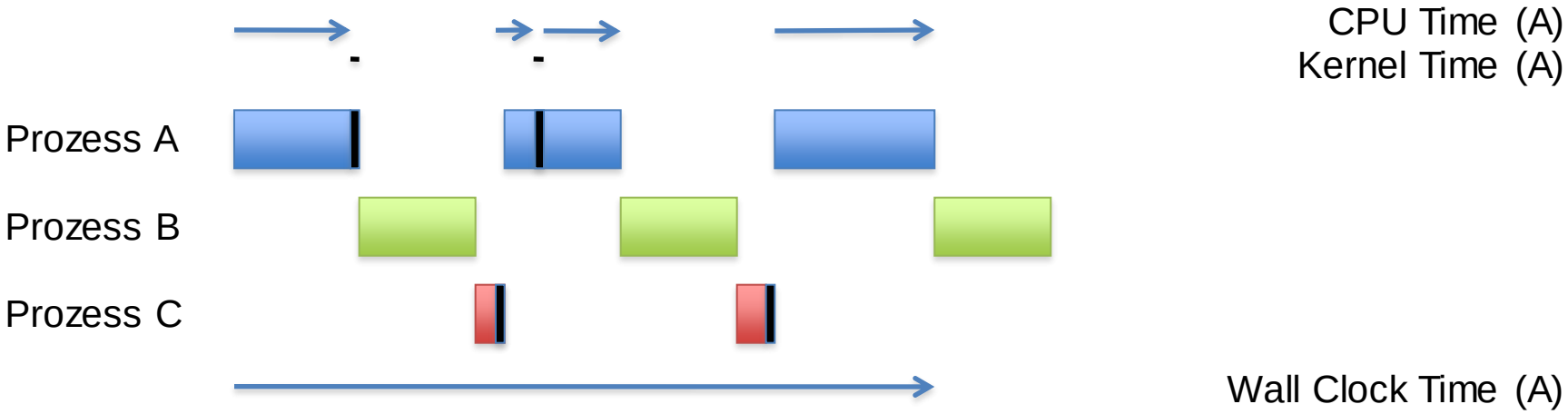
Sascha Meiers
WS 2013/14

Was ist Laufzeit?

- Laufzeit = Running Time
  - Zeit, die für die Ausführung eines Algorithmus benötigt wird
  - Anwender interessiert meist die phys. Laufzeit = Wall Clock Time
- Laufzeitanalyse unterscheidet verschiedene Zeiten:
  - **Wall Clock** Time
    - Gesamtzeit messbar mit herkömmlicher Uhr
  - **CPU** Time
    - Zeit, die ein Prozessor ausschliesslich mit dem Algorithmus beschäftigt war
  - **Kernel** Time
    - Zeit, die ein Prozessor im Kernel verbracht hat (I/O, Interrupts)
  - **Total CPU** Time
    - Multi-Prozessor-System kann mit mehreren Prozessoren parallel rechnen
    - Summe der CPU Times aller beteiligten Prozessoren

Was ist Laufzeit? (II)

- In einem Multitasking Betriebssystem gilt:
  - jeder laufende Prozess bekommt nur einen bestimmten Time-Slot, bevor der nächste Prozess an der Reihe ist (Scheduler verteilt Slots)
  - prozentuale CPU-Nutzung (siehe top/Task Manager) = aktuelles Verhältnis der Time-Slots eines Prozesses zu Time-Slots aller Prozesse
  - Wall Clock Time ≥ CPU Time + Kernel Time



Wie misst man Laufzeit?

- Von aussen – Ausführungszeit eines Programms messen:
  - time *CMD ARG1 ARG2* ... (Unix/Mac OS X)
  - timeit *CMD ARG1 ARG2* ... (Windows Server 2003 Resource Kit)

- Von innen – Einzelne Teilabschnitte innerhalb eines C++ Programms messen:
  - `clock_t x = clock();`
  - `time_t  y = time(NULL);`
    - x ist die CPU Time seit Prozesserzeugung in *clock ticks*
    - y ist die Wall Clock Time in Sekunden seit dem 01.01.1970
    - Konstante CLOCKS_PER_SEC gibt an, wieviele clock ticks einer Sekunde entsprechen
    - Teil der Standard C Library (`#include <ctime>`)

- CPU Time wird bevorzugt bei Algorithmen ohne Festplattennutzung
- Wall Clock Time bei Algorithmen mit Festplattennutzung

Wie misst man Laufzeit? (II)

- Es gibt noch jede Menge plattformabhängige hochauflösende Funktionen:
  - Unter Windows:
    - QueryPerformanceCounter (CPU Time)
    - GetTickCount (Wall Clock Time)
    - GetProcessTimes (CPU Time, Kernel Time)
  - Unter Unix:
    - clock_gettime (CPU Time, Wall Clock Time)
    - gettimeofday (Wall Clock Time)
    - getrusage (CPU Time, Kernel Time)

BEMERKUNGEN ZUR P-AUFGABE

David Weese

Sascha Meiers
WS 2013/14

Bemerkungen zu Aufgabe 2

- Suffixarray
  - Enthält Positionen, keine Strings !!!

`vector<unsigned>`  `vector<string>` 

- Aufbau des Suffixarrays

```
string text = „mamaobama“;
vector<unsigned> sa;

for(unsigned i=0; i< text.size(); ++i) sa.push_back(i);

// Erzeuge unseren (vorher definierten) Vergleichs-Funktor
SuffixComparator less(text);

::std::sort(sa.begin(), sa.end(), less)
```

Bemerkungen zu Aufgabe 2 (II)

- Funktor zum Suffix-Sortieren
  - Ineffiziente Variante:

```
struct SuffixComparator {
        string const & T;
        SuffixComparator (string const & text) : T(text) {}
        operator()(unsigned a, unsigned b)
        {
                return T.substr(a) < T.substr(b);
        }
};
```

  - Besser: Iteratoren benutzen
  - Musterlösung anschauen

    https://svn.imp.fu-berlin.de/aldabi/WS13/material/aufgabe2.cpp

- Die Suche im Suffixarray wird uns wieder begenen
  - Dann wird es auf Effizienz ankommen

Bemerkungen zu Aufgabe 3

- Lokales Alignment mit dem Smith-Waterman-Algorithmus
  - Initialisierung, Dynamic Programming, Traceback

- Korrekte Ein- und Ausgabe beachten:
  - Vertikale Sequenz
  - Horizontale Sequenz
  - *Match Score*
  - *Mismatch Score*
  - *Gap Score*

```
./aufgabe3 senkrecht waagerecht 5 0 -2
enkrecht
|  |||||
e--recht
score:26
```

Bemerkungen zu Aufgabe 3 (II)

- Matrix allokieren
  - vector<vector<int> >
  - Oder: vector<int> und die Matrix *linearisieren*: [r][c] => [c*ROWS + r]

```
./aufgabe3 senkrecht waagerecht 5 0 -2
enkrecht
|  |||||
e--recht
score:26
```

- Traceback:
  - Bei mehreren gleich guten Pfaden die angegebene Reihenfolge beachten!

		w	a	a	g	e	r	e	c	h	t
	0	0	0	0	0	0	0	0	0	0	0
s	0	0	0	0	**0**	0	0	0	0	0	0
e	0	0	0	0	0	**5**	3	5	3	1	0
n	0	0	0	0	0	**3**	5	3	5	3	1
k	0	0	0	0	0	**1**	3	5	3	5	3
r	0	0	0	0	0	0	**6**	4	5	3	5
e	0	0	0	0	0	5	4	**11**	9	7	5
c	0	0	0	0	0	3	5	9	**16**	14	12
h	0	0	0	0	0	1	3	7	14	**21**	19
t	0	0	0	0	0	0	1	5	12	19	**26**