2. Programmierwerkzeuge

AlDaBi Praktikum

David Weese
© 2010/11

Enrico Siragusa
WS 2011/12

Inhalt

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

- Bemerkungen zur P-Aufgabe

BUILD SYSTEM

Allgemeines

Programmbau mit Make:

- Änderungen implizieren oft mehrere Zwischenschritte, um das Programm/ Produkt zu bauen:
  - Übersetzen von Quelldateinen 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, …, 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

Features von **make**

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

```
%.o : %.c
    $(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/

ENTWICKLUNGSUMGEBUNG

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, ...

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

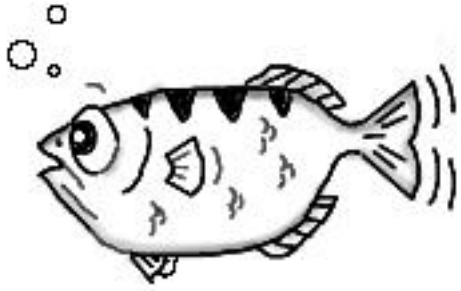
Eclipse

- Verfügbar für Windows, Linux, MacOS X, ...
- Entwicklungsumgebung für C, C++, Java, Perl, Python, ...

- Wie kann man mit Eclipse:
  - ein neues Projekt anlegen
  - Dateien hinzufügen
  - das Projekt kompilieren
  - den Debugger benutzen

- Eclipse CDT - https://www.eclipse.org/cdt/
- Subclipse – http://subclipse.tigris.org/update_1.8.x
- Linux Tools - http://download.eclipse.org/technology/linuxtools/update

DEBUGGER



Fehlersuche mit **gdb**

- 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

- 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 7
1 3 4 5 7 8
$
```

- ... und manchmal nicht:

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

Programme debuggen mit **gdb**

- Übersetzen mit *Debugging-Informationen*:

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

- Starten der Debugging-Sitzung:

```
$ gdb sample
(gdb)
```

- Breakpoint in Zeile 31 und **run** mit **1 8 5 3 4** als Kommandozeile

```
(gdb) breakpoint 31
(gdb) run 1 8 5 3 4
(gdb)
```

- Schrittweise debuggen

```
(gdb) watch *a @ 6
(gdb) step
```

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`

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

```
$ gcc -S -g sample.c
$ less sample.s
…
.globl main
        .type   main, @function
main:
.LFB6:
        .loc 1 26 0
```

← **main**-Funktion beginnt in Zeile 26

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/
- gdb mit STL Documentation - http://gcc.gnu.org/onlinedocs/libstdc++/manual/debug.html

MEMORY DEBUGGER

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:

```
$ g++ -g example.cpp -o example
$
```

- Starten der Debugging-Sitzung:

```
$ valgrind --leak-check=yes ./example 3
$
```

Grenzüberschreitung ...

- Fehlerhafter Zugriff jenseits der Array-Grenzen
  Auszug aus example.cpp:

```
45: int *i = new int[10];
46: i[10] = 13;
47: cout << i[-1] << endl;
48: delete[] i;
```

… ergibt

```
==17056== Invalid write of size 4
==17056==    at 0x401026: test_3() (example.cpp:46)
==17056==    by 0x401159: main (example.cpp:135)
==17056==  Address 0x51E9058 is 0 bytes after a block of size 40 alloc'd
==17056==    at 0x4A1B858: malloc (vg_replace_malloc.c:149)
==17056==    by 0x401019: test_3() (example.cpp:45)
==17056==    by 0x401159: main (example.cpp:135)
==17056==
==17056== Invalid read of size 4
==17056==    at 0x401034: test_3() (example.cpp:47)
==17056==    by 0x401159: main (example.cpp:135)
==17056==  Address 0x51E902C is 4 bytes before a block of size 40 alloc'd
==17056==    at 0x4A1B858: malloc (vg_replace_malloc.c:149)
==17056==    by 0x401019: test_3() (example.cpp:45)
==17056==    by 0x401159: main (example.cpp:135)
0
==17056==
==17056== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 8 from 1)
==17056== malloc/free: in use at exit: 0 bytes in 0 blocks.
==17056== malloc/free: 1 allocs, 1 frees, 40 bytes allocated.
==17056== For counts of detected errors, rerun with: -v
==17056== All heap blocks were freed -- no leaks are possible.
```

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**

- Ist 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/

PROFILER

Laufzeitanalyse mit **gprof**

- Profiler - Programmierwerkzeuge, die das Laufzeitverhalten von Software analysieren
- Tuning - systematische Steigerung der Leistung eines Programms
- Laufzeitanalyse
  - an welchen Stellen sind Leistungssteigerungen möglich
  - was sind die Effekte einer Optimierung

- GNU Profiler gprof
  - wertet Programmprofile aus
  - Programm erstellt Programmprofil während des Programmablaufs
  - Programmprofil gibt für jede Funktionen an
    - wie oft ausgeführt
    - wie lange ausgeführt

So geht‘s

- Übersetzen mit *eingerichteter Profilierung*:

  $ g++ –pg sample.cpp –o sample

  $

- Starten des Programms, Profil wird in Datei gmon.out geschrieben:

  $ ./sample

  $ 100000 ints read

  $

- Analysieren des Profils mit **gprof**:

  $ gprof sample

  $

Das flache Profil

- gibt an, wie sich die Laufzeit auf die einzelnen Funktionen verteilt:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 69.07      0.11     0.11        1   110.51   110.51  shell_sort(int*, int)
 31.40      0.16     0.05        1    50.23    50.23  quick_sort(int*, int,
  0.00      0.15     0.00        1     0.00     0.00  global constructors
  0.00      0.15     0.00        1     0.00     0.00  __static_initializati
```

Das strukturierte Profil

- gibt für jede Funktion f an
  - Anteile von f und der von f aufgerufenen Funktionen an der Gesamtlaufzeit (`%time`)
  - von welcher Funktion f aufgerufen wurde
  - welche Funktionen f aufgerufen hat

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

```
granularity: each sample hit covers 2 byte(s) for 6.22% of 0.16 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.16                 main [1]
                0.11    0.00       1/1           shell_sort(int*, int) [2]
                0.05    0.00       1/1           quick_sort(int*, int, int) [3]
-----------------------------------------------
                0.11    0.00       1/1           main [1]
[2]     68.7    0.11    0.00       1         shell_sort(int*, int) [2]
-----------------------------------------------
                                154478           quick_sort(int*, int, int) [3]
                0.05    0.00       1/1           main [1]
[3]     31.2    0.05    0.00       1+154478  quick_sort(int*, int, int) [3]
                                154478           quick_sort(int*, int, int) [3]
-----------------------------------------------
                0.00    0.00       1/1           __do_global_ctors_aux [12]
[10]     0.0    0.00    0.00       1         global constructors keyed to main [1
                0.00    0.00       1/1           __static_initialization_and_dest
-----------------------------------------------
                0.00    0.00       1/1           global constructors keyed to mai
[11]     0.0    0.00    0.00       1         __static_initialization_and_destruct
-----------------------------------------------
```

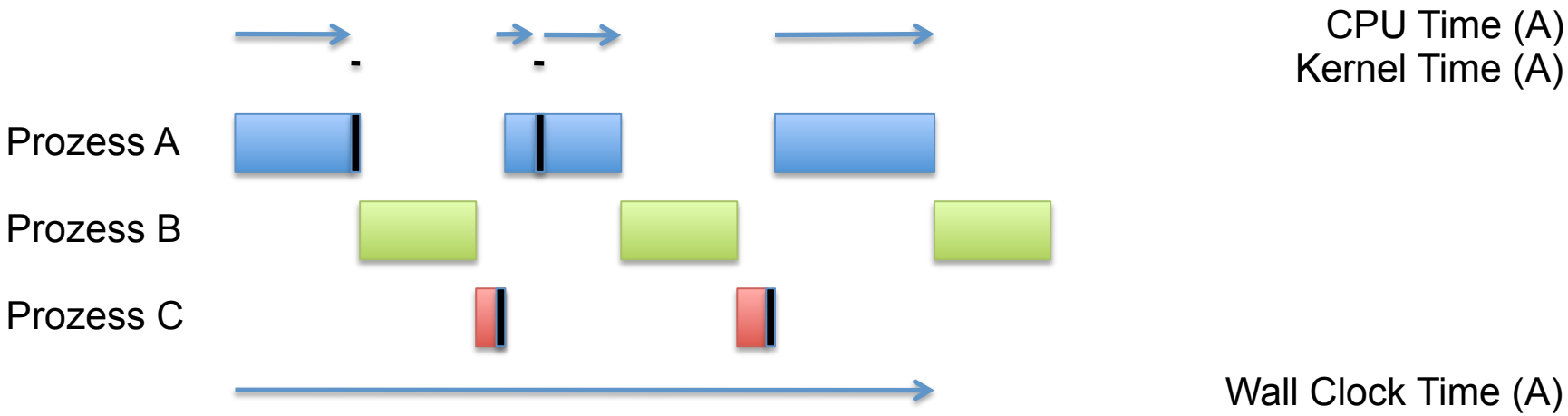
LAUFZEIT MESSEN

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

Bemerkungen zu Aufgabe 1

- Typische Fehler:
  - Text oder Pattern wird falsch indiziert
    - Matches an Textposition 0 werden nicht gefunden
    - Illegale Speicherzugriffe hinter dem Text
  - Shifttabelle wird falsch berechnet
    - `int shift[20];`
  - Falsches Ein-/Ausgabeformat (noch nicht geahndet)
    - es sollten die Argumente von `main(int argc, char * argv[])` genommen werden
    - nicht über `cin >> pattern` oder `getline(cin, pattern)`
    - Matchpositionen sollten ohne extra Text ausgegeben werden
  - Statische Feld für die Ergebnisse
    - `int results[256];`
  - Frist verpasst
    - Automatische SVN Snapshots am Montags um 15Uhr

Bemerkungen zu Aufgabe 1 (II)

- Mögliche Verbesserungen:
  - `vector<int>` ist effizienter als `map<char, int>`
  - Konversion `char` nach `int` geht so:
    - `int i = (int)P[j];  // C-style cast`
    - `int i = static_cast<int>(P[j]); // noch sauberer`
  - Länge eines C-Style Strings:
    - `#include <cstring>`
    - `char const * pattern = "HALLO";`
    - `int i = strlen(pattern);  // i == 5`
  - ... noch einfacher mit der string Klasse:
    - `#include <string>`
    - `string p = pattern;`
    - `int i = p.size();`
  - C style printf statt iostream cout
  - globale Variablen vermeiden, besser lokale Variablen durchreichen
  - besser strukturieren, Funktionen statt Spaghetti-Code

Hinweise zu Aufgabe 2

- Laufzeiten von versch. Algorithmen bestimmen
  - gemeint ist CPU Time
  - das Programm muss unter Linux laufen
    - am besten clock() verwenden
    - oder die unter Linux vorhanden
- Auf das richtige Ausgabeformat achten!
- Parameter werden wieder über die Kommandozeile übergeben!
- Die Lösung soll aufgabe2.cpp heißen (klein, nicht im Unterordner!)