Betriebssysteme
Übungen

Barry Linnert

Wintersemester 2017/18

Automatisierung mit Make

- Ständiges Zusammensuchen von auszuführenden Kommandos mühselig
  - Einhaltung von Abhängigkeiten bei der Reihenfolge notwendig
  - Überblick darf nicht verloren gehen
- Automatisierung über Skripte/Werkzeuge löst diese Probleme
- Gute Skripte/Werkzeuge
  - führen nur tatsächlich notwendige Kommandos aus, d. h. unveränderte Teile werden wiederverwendet
  - erlauben die parallele Ausführung mehrerer Kommandos
- GNU Make ist ein solches Werkzeug; ein Makefile spezifiziert
  - Ziele und deren Abhängigkeiten
  - Aktionen zum Erreichen von Zielen

Makefiles

- Zentraler Bestandteil: Regeln
  - Regel beschreibt Generierung eines Ziels durch Befehlen
  - Ziel kann Quelle in einer anderen Regel sein: Abhängigkeitsbaum/-wald
  - Post-order Traversierung generiert Ziel; Abbruch der Traversierung wenn Befehl fehlschlägt
  - Ziel und Quellen referenzieren normalerweise Dateien: Befehle werden nur ausgeführt, wenn Quelle jünger als Ziel
  - Pattern-Rules mittels Platzhalter „%“

```
Ziel: Quelle1 Quelle2 ...
        Befehl1
        Befehl2
        ...
```

- Achtung:
  - Befehle müssen mit genau einem Tabulator-Zeichen eingerückt sein
  - Genau ein Befehl pro Zeile (Fortsetzung auf der nächsten Zeile mittels Backslash)
  - Jeder Befehl wird in eigener Shell ausgeführt.

Makefiles

- Variablen
  - erlauben Parametrisierung von Regeln
  - Zuweisung via „=“, „+=“, u. a.
  - referenziert mittels $(name)
  - können Regel-spezifisch sein, z. B.:

    $@ aktuelles Ziel

    $^ alle Quellen des aktuellen Ziels

    $< erste Quelle des aktuellen Ziels
- Referenzen auf andere Variablen
  - werden erst bei tatsächlicher Verwendung aufgelöst
  - erlauben „nachträgliche“ Änderungen an Teilen von Variablen
  - erzwungene Auflösung durch Zuweisung mittels „:=“

Makefiles

- Ausschnitt Demo-Makefile

```
LSCRIPT = kernel.lds
OBJ = start1.o led1.o
CC = $(CROSS)gcc
LD = $(CROSS)ld
OBJCOPY = $(CROSS)objcopy
CROSS = arm-none-eabi-
CFLAGS = -Wall -Wextra -ffreestanding
CFLAGS += -mcpu=arm920t -O2
%.o: %.S
     $(CC) $(CFLAGS) -MMD -MP -o $@ -c $<
%.o: %.c
     $(CC) $(CFLAGS) -MMD -MP -o $@ -c $<
kernel: $(LSCRIPT) $(OBJ)
     $(LD) -T$(LSCRIPT) -o $@ $(OBJ) $(LIBGCC)
kernel.bin: kernel
     $(OBJCOPY) -Obinary --set-section-flags \
     .bss=contents,alloc,load,data $< $@
kernel.img: kernel.bin
     mkimage -A arm -T standalone -C none -a 0x20000000 -d $< $@
```

Makefiles

- Auswahl weiterer Features:
  - diverse Funktionen, bspw. zur Manipulation von Text
  - erstes Ziel ist Standard-Ziel, i. d. R. all
  - parallele Ausführung unabhängiger Regeln (make -j<n>)
  - Meta-Ziele (die keine Datei repräsentieren)
  - Katalog vorgefertigter impliziter Regeln
- Abhängigkeiten zwischen Quelldateien
  - C-Datei muss neu kompiliert werden, wenn genutzte Header-Datei geändert wurde
  - statt manueller Notation: durch GCC generieren lassen
  - Aktivieren und Steuern über diverse -M Parameter
  - Inkludieren der generierten Abhängigkeiten

Makefiles

- Ausschnitt Demo-Makefile

```
LIBGCC := $(shell $(CC) \
     -print-libgcc-file-name)
.PHONY: all
all: kernel

DEP = $(OBJ:.o=.d)

-include $(DEP)

.PHONY: install
install: kernel.img
     arm-install-image $<

.PHONY: clean
clean:
     rm -f kernel kernel.bin kernel.img
     rm -f $(OBJ)
     rm -f $(DEP)
```

Compiler-Warnungen für `printf`

- Compiler prüft Argumente von Funktionsaufrufen.
  - Nicht möglich bei variablen Argumentlisten!
- GCC unterstützt Sonderbehandlungen für bestimmte Funktionen
  - z. B. für `printf`: prüfe Argumente gegen Formatstring
  - Aktivieren über format-Funktionsattribut
  - hilft Fehler zu vermeiden

- Prüfung für printf-Formatstring

```
__attribute__((format(printf, 1, 2)))
void myprint(char *fmt, ...) {
/*
* 1: Position des Formatstrings in der Argumentliste
* 2: Position des ersten gegen den Formatstring zu prüfenden Arguments
*/
}
```

Fehlerbehandlung in C

- C kennt keine Exceptions und try-catch-finally Konstrukte
  - Fehler wird über Rückgabewert von Funktionen angezeigt
  - normalerweise: 0 für Erfolg, alles andere: Fehlercode
  - Rückgabewerte müssen geprüft werden
  - im Fehlerfall müssen ggf. Ressourcen freigegeben werden
  - wenn nicht: direktes `return`
- Zentralisierte Ressourcenfreigabe mittels `goto`
  - bei festgestelltem Fehler wird Rückgabewert gesetzt und an passende Stelle am Funktionsende gesprungen
- Keine weitere Verwendung von `goto`!

Fehlerbehandlung in C

```
int foo(void) {
      int ret = 0;
      ret = alloc_resource_a();
      if (ret) return ret;
      ret = alloc_resource_b();
      if (ret) goto err_free_a;
      ret = alloc_resource_c();
      if (ret) goto err_free_b;

      /*
      * Echte Arbeit hier. Falls ein Fehler festgestellt wird: ret setzen und nach
      * out springen
      *
      * Falls Ressourcen im Erfolgsfall belegt bleiben sollen: extra
      * Return vor Sprungmarke out.
      */
out:
      free_resource_c();
err_free_b:
      free_resource_b();
err_free_a:
      free_resource_a();
      return ret;
}
```

Bitmanipulation in C

- Binäre Operatoren:
  - Und `&`, Oder `|`, Exklusive-Oder `^`, Nicht `~`
  - Nicht verwechseln mit logischen Operatoren: `&& || !`
- Links- und Rechts-Shift: `<< , >>`
- Bits setzen: `v |= (1 << 5) | (1 << 8);`
- Bits löschen: `v &= ~(1 << 5);`
- Multiplikation, Division und Rest mit (konstanten) Zweierpotenzen werden von GCC automatisch in effizientere Form umgewandelt, z. B.
  - `x * 16` entspricht `x << 4`
  - `x / 512` entspricht `x >> 9`
  - `x % 32` entspricht `x & ((1 << 5) - 1)` entspricht `x & 0x1f`

Pointer in C, mal anders erklärt

Freie Universität Berlin

- `unsigned char **********a;`
  - enthält * → a ist ein Zeiger → a enthält Speicheradresse → a belegt (bei uns) 4 Byte
  - An welcher Adresse liegt a? `&a`
  - Welchen Typ hat der Ausdruck `&a`? Typ von a mit einem Sternchen mehr
  - Zugriff auf das, worauf a zeigt? `*a`
  - Welchen Typ hat der Ausdruck `*a`? Typ von a mit einem Sternchen weniger
- Aus Sicht des Pointers besteht der ganze Speicher aus einer Aneinanderreihung von Objekten des von ihm referenzierten Typs
  - Addition/Subtraktion von ganzen Zahlen bedeutet Verschiebung der Speicheradresse um entsprechend viele Elemente
  - Kurzschreibweise: statt `*(a + 10)` geht auch `a[10]`
  - bei Strukturen: statt `(*s).v` geht auch `s->v`

Pointer in C, mal anders erklärt

Freie Universität Berlin

- Übergabe an Funktionen
  - Übergabe von Daten → Daten können verwendet werden
  - Übergabe einer Speicheradresse → Zugriff auf Daten an dieser Adresse
  - Übergabe eines Zeigers auf Speicheradresse → Speicheradresse selbst änderbar

- Adressoperator & funktioniert nur bei Dingen, die irgendwo im Speicher liegen
- Dereferenzoperator * funktioniert nur bei Typen mit Sternchen/„Speicheradressen“

- Richtig: `a &a[10] *a ***a &a *(&a) a[0][3] *a++ *++a`
- Falsch: `&(a + 10) *(unsigned int)a &&a &a++`

Pointer in C, mal anders erklärt

Freie Universität Berlin

- Arrays lassen sich wie (unveränderlicher) Zeiger verwenden:
  - `int b[10];` entspricht (nahezu)

    ```
    int * const b = <Speicheradresse von 10 int>;
    ```

- Verschachtelte Pointer eher anzutreffen bei verlinkten Strukturen:

  ```
  head->next->pcb.context->pc
  ```

Defensives Programmieren/ Robuster Code

Freie Universität Berlin

- Unvorhergesehenes vorhersehen
  - Daten kritisch betrachten
  - Schleifendurchläufe begrenzen
  - `default`-Fall in einem Switch-Statement
  - Rückgabewerte prüfen
  - Design by Contract
  - spezielle Werte für „ungültig“
- Sonstiges
  - kein Copy&Paste (auch nicht von eigenem Code): dupliziert Fehler
  - NULL-Pointer referenziert gültige Adresse (im Moment) (Compiler nimmt allerdings an, dass ein erfolgreich dereferenzierter Zeiger nicht NULL ist. Annahme mittels `-fno-delete-null-pointer-checks` abschalten.)

Defensives Programmieren/ Robuster Code

- Vor-/Nachbedingungen und Invarianten prüfen
  - dauerhaft an kritischen Stellen
  - vorübergehend zum Finden eines Fehlers
- Je nach Schwere des Fehlers unterschiedliche Reaktion
  - System wird anhalten mit Fehlermeldung für Systementwickler
  - . . .
  - Funktion gibt Fehlercode zurück (oder übergeht Fehler)

- Ausschnitt Linux-Kernel

```
#define BUG() do { \
     printk("BUG: failure at %s:%d/%s()!\n", __FILE__, __LINE__, __func__); \
     panic("BUG!"); \
} while (0)
#define BUG_ON(condition) do { if (unlikely(condition)) BUG(); } while(0)
```

Fehlersuche

Freie Universität Berlin

- Bei plötzlich auftretendem Fehler:
  - Reproduzierbarkeit prüfen
  - `make clean`
  - Compiler-Warnungen beseitigen
  - letzte funktionierende Version nochmal prüfen (Versionsverwaltung!)
  - Änderungen nochmal einzeln durchgehen
  - mehr Checks im Code (ggf. vorübergehende Checks)
  - Datenstrukturen dumpen
  - Debugger anwerfen

- Fehler oft nicht in neuem Code, sondern in altem Code, der durch neuen Code anders verwendet wird!

QEMU mit GDB

Freie Universität Berlin

- Mit Debug-Informationen kompilieren: `CFLAGS += -g`
  - (eventuell Optimierungen abschalten)

- QEMU starten mit:

  `qemu-bsprak -kernel kernel -S -gdb tcp::<port>`

  -S hält virtuelle CPU noch vor erster Instruktion an

  -gdb tcp::<port> erlaubt GDB-Verbindung via TCP

- GDB starten mit:

  `arm-none-eabi-gdb kernel`

- In GDB eingeben:

  `target remote :<port>`