U5 5. Übung

- Besprechung 3. Aufgabe (smash)
- Erstellen von C-Funktionsbibliotheken
- RCS

U5-1 Überblick C-Funktionsbibliotheken

- statische Bibliotheken
  - ➤ Archiv, in dem mehrere Objekt-Dateien (.o) zusammengefasst werden
  - ➤ beim statischen Binden eines Programms werden die benötigten Objekt-Dateien zu der ausführbaren Datei hinzukopiert
  - ➤ Bibliothek ist bei der Ausführung des Programms nicht mehr sichtbar
- dynamische, gemeinsam genutzte Bibliotheken (shared libraries)
  - ➤ Zusammenfassung von übersetzten C-Funktionen
  - ➤ beim Binden werden Referenzen auf die Funktionen offen gelassen
  - ➤ Shared Library ist nur einmal im Hauptspeicher vorhanden
  - ➤ Shared Library wird in virtuellen Adressraum dynamisch gebundener Programme beim Laden eingeblendet,
    noch offene Referenzen werden danach gebunden

U5-1 Symboltabellen

- Zugriff auf Funktionen und Variablen über symbolische Namen
- Eine Übersetzungseinheit (C-Datei) **definiert** und **verwendet** Symbole
  - Hauptprogramm definiert das Symbol *main* für die main-Funktion
  - Programm ruft (verwendet) eine Funktion der C-Bibliothek (z.B. malloc)
  - C-Datei definiert eine globale (nicht-static) Variable
  - Programm verwendet eine globale Variable, die u.U. in einer anderen Übersetzungseinheit definiert wurden
- Der Namensraum ist flach
  - jeder Name muss eindeutig sein
  - es darf auch keine Funktion mit dem Namen einer globalen Variable geben
- Kompilierte Übersetzungseinheit (Objekt-Datei) enthält Symboltabelle
  - Liste mit Symbolen, die von der Einheit verwendet werden
  - Liste von Symbolen, die von der Einheit definiert werden

U5-2 Statische Bibliotheken

- Statisches Binden: Binden der Übersetzungseinheiten zum Binärabbild
- Offene Symbolreferenzen werden aufgelöst
  - definiert in anderen Übersetzungseinheiten
  - Suche in Programmbibliotheken
- GCC sucht beim Binden implizit in der Standard C-Bibliothek (**libc.a**)
- weitere Bibliotheken können vom Entwickler angegeben werden

1 Statische Bibliotheken erstellen

- **Archiv** (Dateiendung **.a**) von Objekt-Dateien (**.o**-Dateien)
  ☞ Erstellen einer Bibliothek mit dem Kommando **ar**

```
ar -r -c -s libexample.a bar.o foo.o
```

- Jede Objekt-Datei enthält wiederum eine Symboltabelle
  ☞ Anzeige mit dem Kommando **nm libexample.a** bzw. **nm bar.o**

- Die Bibliothek kann dem Linker als Symbolquelle angeboten werden
  - Parameter -L*pfad*: Suche nach Bibliotheken (.a-Dateien) in *pfad*
  - Parameter -l*libname*: Binden mit der Bibliothek *libname*
    ☞ diese wird in einer Datei lib*libname*.a in den Suchpfaden gesucht

- Der Linker bindet dann alle Objekt-Dateien, die **bis dahin** unaufgelöste Symbole definieren, zum Binärabbild dazu

- Die Reihenfolge von Objekt-Dateien und Bibliotheken ist wichtig

2 Beispiel: Kompilieren mit statischen Bibliotheken

```
#include <bar.h>

void main ( void ) {
        bar ( 42 );
}
```

main.c

```
#ifndef BAR_H
#define BAR_H

void bar ( int );

#endif
```

bar.h (Schnittstelle)

```
#include <bar.h>
void bar ( int param ) {
        /* do stuff */
}
```

bar.c (Implementierung)

- Module exportieren eine Schnittstelle (Header-Datei)
  - Funktionsdeklarationen und ggf. Deklarationen (*extern*) globaler Variablen
- Beim Übersetzen muss der Compiler den Typen eines Symbols kennen
  - Einbinden der Schnittstellenbeschreibung mit *#include <bar.h>*
  - Die Adresse der Funktion bzw. Variable ist an dieser Stelle nicht notwendig
- Parameter -I*pfad*: Teilt Compiler zusätzlichen Suchpfad für Headerdateien mit

```
gcc -c <...> -I/myincludes main.c
```

3 Beispiel: Binden mit statischen Bibliotheken

```
#include <bar.h>

void main ( void ) {
        bar ( 42 );
}
```

main.c

```
#ifndef BAR_H
#define BAR_H

void bar ( int );

#endif
```

bar.h (Schnittstelle)

```
#include <bar.h>
void bar ( int param ) {
        /* do stuff */
}
```

bar.c (Implementierung)

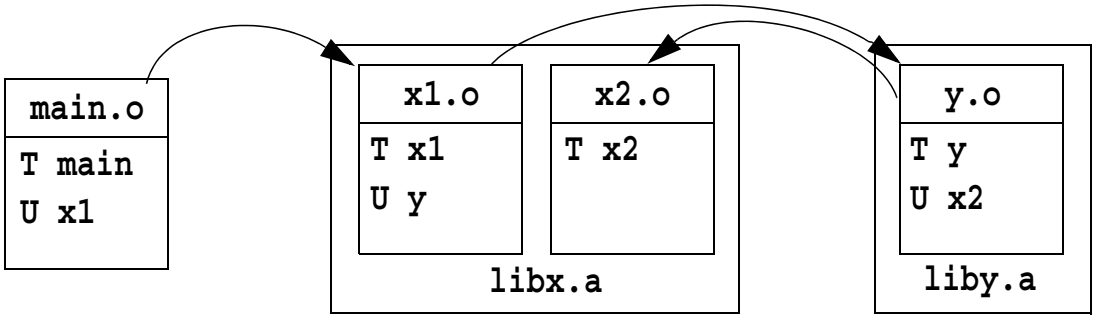
- Modul *bar* **definiert** Symbol *bar* (`Funktion void bar(int);`)
- Hauptprogramm ruft die Funktion `bar` auf (**verwendet** Symbol *bar*)
- Annahme: Modul *bar.o* ist Teil der Bibliothek *libexample.a* in */libdir*
- Übersetzung mit Kommando:

```
gcc -static -L/libdir -o main <...> main.o -lexample
```

- Falsch (warum?):

```
gcc -static -L/libbdir -o main <...> -lexample main.o
```

4 Beispiel 2: Binden mit statischen Bibliotheken



- Wie lautet der Bindeaufruf in diesem Beispiel?

```
gcc -static -L/libdir -o main main.o -lx -ly
```

- Beim Einbinden von `y.o` tritt ein neues undefiniertes Symbol `x2` auf
- `x2` wird nicht mehr aufgelöst, da `libx.a` nicht erneut durchsucht wird
- Lösung: Mehrfachangabe von `-lx`

```
gcc -static -L/libdir -o main main.o -lx -ly -lx
```

U5-3 Shared Libraries

- Kein Dateiarchiv sondern eine ladbare Funktionssammlung
  - ➤ Erzeugen mit gcc
- Code der Funktionen liegt nur einmal im Hauptspeicher, kann aber in verschiedenen Anwendungen an unterschiedlichen Adressen im virtuellen Adressraum positioniert sein
  - ➤ keine absoluten Adressen (Sprünge, Unterprogrammaufrufe) im Code erlaubt -> PIC (*position independent code*)
  - ➤ muss beim Kompilieren der Quellen berücksichtigt werden

```
gcc -fPIC -c file1.c
gcc -fPIC -c file2.c
...
```

- Bibliothek wird durch Binden mehrerer .o-Dateien erzeugt

```
gcc -shared -o libutil.so file1.o file2.o ...
```

1 Shared Libraries

- Beim Binden einer Anwendung werden Funktionen nicht aus Bibliothek kopiert

```
gcc prog.c -L. -lutil -o prog
```

  - ➤ Aufruf analog zum statischen Binden (aber Option -static hat dort verhindert, dass dynamisch gebunden wird)
  - ➤ Bibliothek `libutil.so` wird gesucht

- Endgültiges Binden erfolgt erst beim Laden
  - ➤ Beim Laden von prog (exec) wird zunächst der *dynamic linker/loader* (`ld.so`) geladen
  - ➤ ld.so lädt prog und die Bibliothek (wenn noch nicht im Hauptspeicher vorhanden) und bindet noch offene Referenzen
  - ➤ Bibliothek wird von `ld.so` in mehreren Directories gesucht (über Environment-Variable `LD_LIBRARY_PATH` einstellbar)

- Reihenfolge von Bibliotheken und Objekt- bzw. Quelldateien unwichtig

- Mehrfachaufführung einer Bibliothek nicht notwendig

U5-4 Revision Control System – RCS

1 Einführung

- RCS ist ein Versionskontrollsystem, das
  - Änderungen an Dateien mit dem Namen des Ändernden, dem Zeitpunkt und einem Kommentar speichert
  - Zugriffe auf Versionen kontrolliert und koordiniert
  - eindeutige Identifizierung verwendeter Versionen erlaubt
  - redundante Speicherung von Versionen vermeidet
    - ➥ es wird jeweils die letzte Version einer Datei gespeichert
    - ➥ zusätzlich werden sog. *reverse deltas* (Beschreibungen, wie aus Version n Version n-1 erzeugt wird) abgelegt

2 Einführung (2)

- RCS besteht aus einer Reihe von Kommandos, die es dem Benutzer erlauben
  - Dateien unter RCS-Kontrolle zu stellen und Kopien aller Versionen zu bekommen, die danach erstellt wurden
  - eine Version zum Editieren zu entnehmen und diese gegen gleichzeitige Änderungen zu sperren
  - Neue Versionen (mit Kommentar) zu erzeugen
  - Unbrauchbare Änderungen rückgängig zu machen
  - Zustandsinformation von Dateien abzufragen
    - ➤ Zeilenweise Unterschiede zwischen verschiedenen Versionen auszugeben
    - ➤ Log-Informationen über Versionen: Urheber, Datum, usw.

3 Terminologie

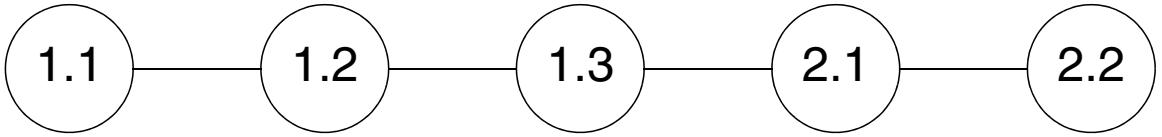
- Delta
  - Menge von zeilenweisen Änderungen an der Version einer Datei unter der Kontrolle von RCS
    (die Begriffe “Version" und “Delta” werden oft synonym gebraucht)
- Revision-Id
  - Jede Version erhält zur Identifikation eine Identifikation zugewiesen:

    `Release-Nummer.Level-Nummer`
- RCS-Datei
  - enthält die neueste Version und alle vorhergehenden Versionen in Form von Deltas zusammen mit Verwaltungsinformationen
  - der Dateiname endet auf **`,v`** , die RCS-Datei ist entweder im Unterdirectory **`RCS`**, oder im gleichen Directory wie die Arbeitsdatei abgelegt
- Arbeitsdatei
  - Kopie einer Version aus der RCS-Datei

4 Nummerierung von Versionen

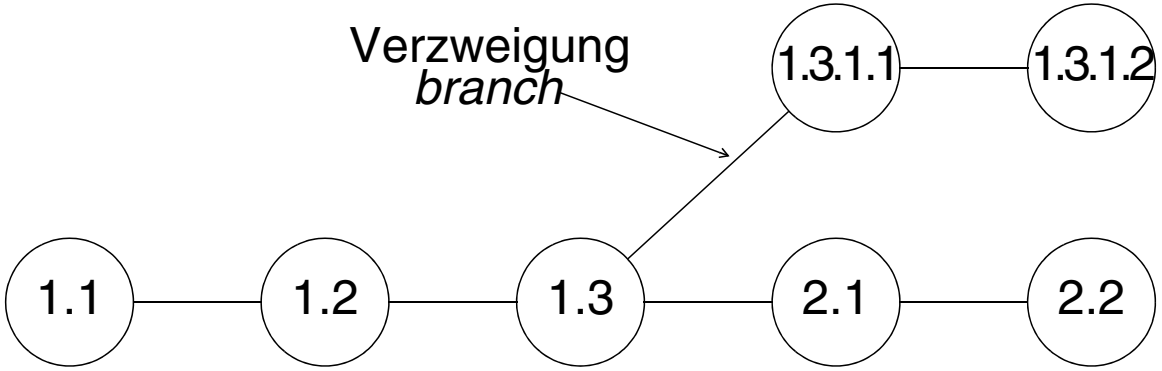
- Versionen werden ausgehend von der Ur-Version nummeriert:

```
release.level
```



- Versionen in einer Verzweigung erhalten

```
release.level.branch.branchlevel
```



5 Kommandos — Überblick

ci(1)	***check in*** speichert die Arbeitsdatei als neue Version in der RCS-Datei ab falls noch nicht vorhanden, wird ein neue RCS-Datei erzeugt
co(1)	***check out*** extrahiert eine existierende Version aus der RCS-Datei (nur zum Lesen oder exklusiv zum Schreiben)
rcs(1)	Modifikation von RCS-Datei-Attributen
rlog(1)	Ausgabe von *log*-Information und RCS-Datei-Attributen
ident(1)	extrahiert RCS-Identifikatoren aus einer Datei
rcsclean(1)	nicht-modifizierte Arbeitsdateien löschen
rcsdiff(1)	*diff* zwischen Versionen einer RCS-Datei
rcsmerge(1)	erzeugt aus zwei Versionen (insbes. bei Verzweigungen) eine neue Version

- bei allen Kommandos kann als ***filename*** immer sowohl der Arbeitsdateiname oder der RCS-Dateiname angegeben werden

5 Kommandos — *ci(1)*

- *check in RCS-Revisions* — Erzeugen neuer Versionen
  - ***ci(1)*** übernimmt neue Versionen in RCS-Dateien
  - die neue Version wird aus der jeweiligen Arbeitsdatei entnommen, die Arbeitsdatei wird anschließend gelöscht
  - existierte zu der Arbeitsdatei noch keine RCS-Datei, wird eine neue RCS-Datei erzeugt

- Aufrufsyntax (nur die wichtigsten Optionen angegeben!):

```
ci [-rrev] [-lrev] [-urev] filename ...
```

`-r`***rev*** die neue Version erhält Version ***rev***
  - ***rev*** muß größer als die letzte existierende Version sein
  - soll eine neue *Release* erzeugt werden, genügt die Angabe der *Release*-Nummer (z. B. `-r5`)

`-l`***rev*** wie `ci -r`, anschließend wird automatisch ein `co -l` durchgeführt

`-u`***rev*** wie `ci -r`, anschließend erfolgt ein `co`

5 Kommandos — *ci(1)*

- Beispiel ***ci, rlog***

```
% ci prog.c
RCS/prog.c,v  <--  prog.c
initial revision: 1.1
enter description, terminated with single '.' or end of file:
NOTE: This is NOT the log message!
>> Program to demonstrate RCS
>> .
done
% rlog prog.c

RCS file: RCS/prog.c,v
Working file: prog.c
head: 1.1
branch:
locks: strict
access list:
symbolic names:
comment leader: " * "
keyword substitution: kv
total revisions: 1;     selected revisions: 1
description:
Program to demonstrate RCS
----------------------------
revision 1.1
date: 1992/07/20 11:56:43;  author: jklein;  state: Exp;
Initial revision
=============================================================
```

5 Kommandos — *co(1)*

- *check out RCS Revisions* — Versionen entnehmen

- ***co(1)*** entnimmt eine Version aus allen angegebenen RCS-Dateien
- die entnommene Version wird als Arbeitsdatei abgespeichert
- der Name der Arbeitsdatei ergibt sich aus dem Namen der RCS-Datei, wobei die Endung `,v` und ggf. der Pfad-Prefix `RCS/` weggelassen werden
  - Aufrufsyntax (nur die wichtigsten Optionen angegeben!):

    ```
    co [-rrev] [-lrev] [-urev] filename ...
    ```

    `-r`***rev*** extrahiert die neueste Version, der Versionsnummer kleiner oder gleich ***rev*** ist

    `-l`***rev*** wie `co -r`, extrahiert die Version für den Aufrufer exklusiv zum Schreiben (für weitere co-Aufrufe gesperrt)

    `-u`***rev*** wie `co -r`, falls eine Sperre der Version durch den Aufrufer existiert, wird diese aufgehoben

5 Kommandos — *co(1)*

- Beispiel ***co, rlog***

```
% co -l prog.c
RCS/prog.c,v  -->  prog.c
revision 1.1 (locked)
done
% rlog prog.c

RCS file: RCS/prog.c,v
Working file: prog.c
head: 1.1
branch:
locks: strict
        jklein: 1.1
access list:
symbolic names:
comment leader: " * "
keyword substitution: kv
total revisions: 1;     selected revisions: 1
description:
Program to demonstrate RCS
----------------------------
revision 1.1    locked by: jklein;
date: 1992/07/20 11:56:43;  author: jklein;  state: Exp;
Initial revision
=============================================================
%
```

6 Identifikation von RCS-Versionen

- RCS ersetzt bei einem ***check out*** im Text alle Vorkommen der Zeichenkette

  `$Id$`

  durch

  `$Id:` *filename revisionnumber date time author state locker*`$`

- ***co(1)*** sorgt dafür, dass diese Zeichenkette automatisch auf aktuellem Stand gehalten wird

- um diese Zeichenkette in Objekt-Code zu implantieren, reicht es, sie in als *String* im Programm anzugeben — in C z. B.

```
static volatile const char rcsid[] = "$Id$";
```

- mit dem Kommando ***ident(1)*** können solche RCS-Identifikatoren aus beliebigen Dateien extrahiert werden

  ➥ damit ist z. B. feststellbar, aus welchen Versionen der Quelldateien ein ausführbares Programm entstanden ist