

# Systemprogrammierung

## Vom C-Programm zum laufenden Prozess

29. Mai 2011

## Übersetzen - Objektmodule

### ■ 1. Schritt: Präprozessor

- ◆ entfernt Kommentare, wertet Präprozessoranweisungen aus
    - fügt include-Dateien ein
    - expandiert Makros
    - entfernt Makro-abhängige Code-Abschnitte (*conditional code*)
- Beispiel:

```
#define DEBUG
...
#ifdef DEBUG
    printf("Zwischenergebnis = %d\n", wert);
#endif DEBUG
```

- ◆ Zwischenergebnis kann mit `cc -P datei.c` als `datei.i` erzeugt werden  
oder mit `cc -E datei.c` ausgegeben werden

## Übersetzen - Objektmodule (2)

- 2. Schritt: Compilieren
  - ◆ übersetzt C-Code in Assembler
  - ◆ Zwischenergebnis kann mit `cc -S datei.c` als `datei.s` erzeugt werden
- 3. Schritt: Assemblieren
  - ◆ assembliert Assembler-Code, erzeugt Maschinencode (Objekt-Datei)
  - ◆ standardisiertes Objekt-Dateiformat: ELF (Executable and Linking Format) (vereinfachte Darstellung) - in nicht-UNIX-Systemen andere Formate
    - Maschinencode
    - Informationen über Variablen mit Lebensdauer *static* (ggf. Initialisierungswerte)
    - Symboltabelle: wo stehen welche globale Variablen und Funktionen
    - Relokierungsinformation: wo werden welche "nicht gefundenen" globalen Variablen bzw. Funktionen referenziert
  - ◆ Zwischenergebnis kann mit `cc -c datei.c` als `datei.o` erzeugt werden

## Binden und Bibliotheken

- 4. Schritt: Binden
  - ◆ Programm `ld` : ( *linker* ), erzeugt ausführbare Datei ( *executable file* )
    - ebenfalls ELF-Format (früher a.out-Format oder COFF)
  - ◆ Objekt-Dateien (`.o`-Dateien) werden zusammengebunden
    - noch nicht abgesättigte Referenzen auf globale Variablen und Funktionen in anderen Objekt-Dateien werden gebunden (Relokation)
  - ◆ nach fehlenden Funktionen wird in Bibliotheken gesucht
- statisch binden
  - ◆ alle fehlenden Funktionen werden aus Bibliotheken genommen und in die ausführbare Datei einkopiert
    - ausführbare Datei ggf. sehr groß
    - Funktionen die in vielen Programmen benötigt werden (z. B. `printf`) werden überall einkopiert

## Binden und Bibliotheken (2)

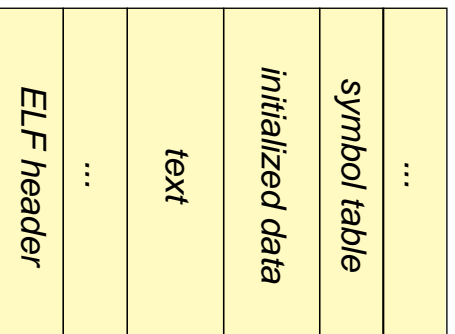
- dynamisch binden
  - ◆ Funktionen aus gemeinsam nutzbaren Bibliotheken (*shared libraries*) werden nicht in die ausführbare Datei inkopiert
    - ausführbare Datei enthält weiterhin nicht-relokierte Referenzen
    - ausführbare Dateien sind kleiner, mehrfach genutzte Funktionen sind nur einmal in der shared library abgelegt
    - Relokation erfolgt beim Laden

## Programme und Prozesse

- **Programm:** Folge von Anweisungen (hinterlegt beispielsweise als ausführbare Datei auf dem Hintergrundspeicher)
- **Prozess:** Programm, das sich in Ausführung befindet, und seine Daten (Beachte: ein Programm kann sich mehrfach in Ausführung befinden)
  - ein Prozess ist erst mal ein **abstraktes Gebilde** (= Funktionen und Datenstrukturen zur Verwaltung von Programmausführungen)
  - im objektorientierten Sinn eine *Klasse*
- **Prozessinstanz** (Prozessinkarnation): eine physische Instanz des abstrakten Gebildes "Prozess"
  - eine konkrete Ausführungsumgebung für ein Programm (Speicher, Rechte, Verwaltungsinformation)
  - im objektorientierten Sinn die *Instanz*
- Sprachgebrauch in der Praxis etwas schlampig: mit "Prozess" wird meistens eine Prozessinkarnation gemeint

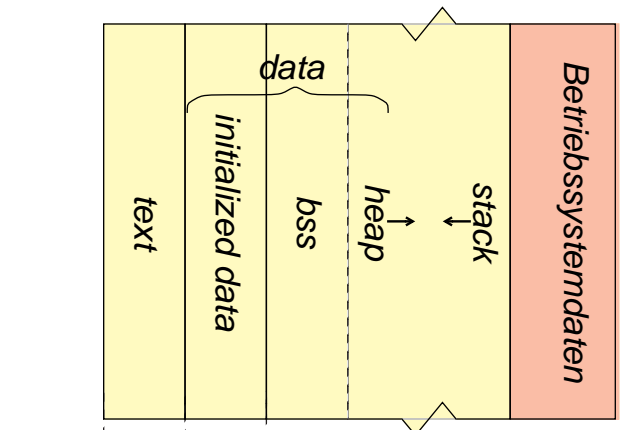
### 3.1 Speicherorganisation eines Programms

- definiert durch das ELF-Format
- wichtigste Elemente (stark vereinfacht dargestellt)



<b>ELF header</b>	Identifikator und Verwaltungsinformationen (z. B. verschiedene <i>executable</i> Formate möglich)
<b>text</b>	Programmcode
<b>initialized data</b>	initialisierte globale und <i>static</i> Variablen
<b>symbol table</b>	Zuordnung der im Programm verwendeten symbolischen Namen von Funktionen und globalen Variablen zu Adressen (z. B. für Debugger)

### 3.2 Speicherorganisation eines Prozesses



<b>bss</b>	nicht initialisierte globale und <i>static</i> Variablen (wird vor der Vergabe an den Prozess mit 0 vorbelegt)
<b>heap</b>	dynamische Erweiterungen des bss-Segments ( <b>sbrk(2)</b> , <b>malloc(3)</b> )
<b>stack</b>	lokale Variablen, Funktionsparameter, Speicherbereiche für Registerinhalte, (wird bei Bedarf dynamisch erweitert)

<b>symbol table</b>	
<b>initialized data</b>	
<b>text</b>	
<b>...</b>	
<b>ELF-header</b>	

## Laden eines Programms

- in eine konkrete Ausführungsumgebung ("Prozessinkarnation") kann ein Programm geladen werden
  - Loader
- Laden statisch gebundener Programme
  - ◆ Segmente der ausführbaren Datei werden in den Speicher geladen
    - abhängig von der jeweiligen Speicherorganisation des Betriebssystems
  - ◆ Speicher für nicht-initialisierte globale und *static* Variablen (bss) wird bereitgestellt
  - ◆ Speicher für lokale Variablen (stack) wird bereitgestellt
  - ◆ Aufrufparameter werden in Stack- oder Datensegment kopiert, *argc* und *argv*-Zeiger werden entsprechend initialisiert
  - ◆ *main*-Funktion wird angesprungen

## Laden eines Programms (2)

- Laden dynamisch gebundener Programme
  - ◆ Spezielles Lade-Programm wird gestartet: *ld.so* ( *dynamic linker/loader* )  
Id:so erledigt die weiteren Aufgaben
    - Segmente der ausführbaren Datei werden in den Speicher geladen und Speicher für nicht-initialisierte globale und *static* Variablen (bss) wird angelegt
    - fehlende Funktionen werden aus *shared libraries* geladen (ggf. rekursiv)
    - noch offene Referenzen werden abgesättigt (Relokation)
    - wenn notwendig werden Initialisierungsfunktionen der *shared libraries* aufrufen (z. B. Klasseninitialisierungen bei C++)
    - Parameter für *main* werden bereitgestellt
    - *main*-Funktion wird angesprungen
    - bei Bedarf können auch während der Laufzeit des Programms auf Anforderung des Programms weitere Funktionen nachgeladen werden (z. B. für plugins)

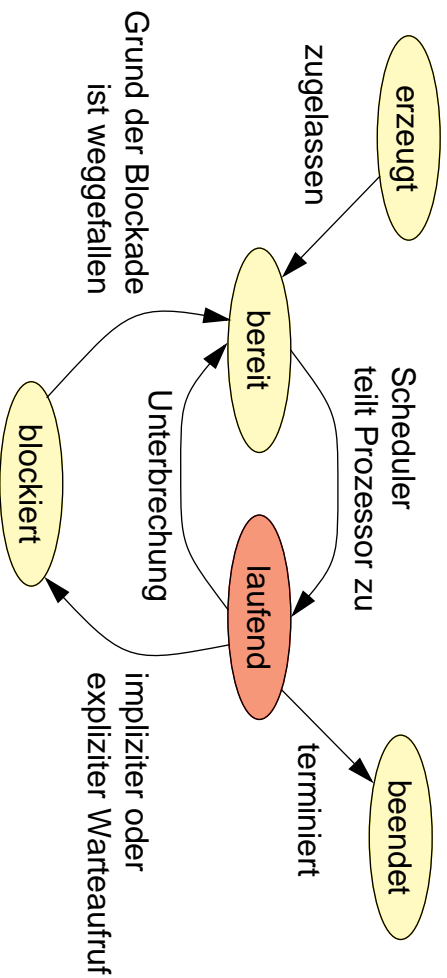
## Prozesse

### 5.1 Prozesszustände

- Ein Prozess befindet sich in einem der folgenden Zustände:
  - ◆ **Erzeugt (New)**  
Prozess wurde erzeugt, besitzt aber noch nicht alle nötigen Betriebsmittel
  - ◆ **Bereit (Ready)**  
Prozess besitzt alle nötigen Betriebsmittel und ist bereit zum Laufen
  - ◆ **Laufend (Running)**  
Prozess wird vom realen Prozessor ausgeführt
  - ◆ **Blockiert (Blocked/Waiting)**  
Prozess wartet auf ein Ereignis (z.B. Fertigstellung einer Ein- oder Ausgabeoperation, Zuteilung eines Betriebsmittels, Empfang einer Nachricht); zum Warten wird er blockiert
  - ◆ **Beendet (Terminated)**  
Prozess ist beendet; einige Betriebsmittel sind aber noch nicht freigegeben oder Prozess muss aus anderen Gründen im System verbleiben

### 5.1 Prozesszustände (2)

#### ■ Zustandsdiagramm



Nach Silberschatz, 1994

- ◆ Scheduler ist der Teil des Betriebssystems, der die Zuteilung des realen Prozessors vornimmt.

## 5.2 Prozesserstellung (UNIX)

### ■ Erzeugen eines neuen UNIX-Prozesses

#### ◆ Duplizieren des gerade laufenden Prozesses

```
pid_t fork( void );
```

<pre>pid_t p; ... p = fork(); if( p == (pid_t)0 ) {     /* child */ } else if( pi==(pid_t)-1 ) {     /* parent */ } else {     /* error */     ... }</pre>	Vater
--	-------

## 5.2 Prozesserstellung (UNIX)

### ■ Erzeugen eines neuen UNIX-Prozesses

#### ◆ Duplizieren des gerade laufenden Prozesses

```
pid_t fork( void );
```

<pre>pid_t p; ... p = fork(); if( p == (pid_t)0 ) {     /* child */ } else if( pi==(pid_t)-1 ) {     /* parent */ } else {     /* error */     ... }</pre>	Vater	<pre>pid_t p; ... p = fork(); if( p == (pid_t)0 ) {     /* child */ } else if( pi==(pid_t)-1 ) {     /* parent */ } else {     /* error */     ... }</pre>	Kind
--	-------	--	------

## 5.2 Prozesserstellung (2)

- ◆ Der Kind-Prozess ist eine perfekte **Kopie** des Vaters
  - gleiches Programm
  - gleiche Daten (gleiche Werte in Variablen)
  - gleicher Programmzähler (nach der Kopie)
  - gleicher Eigentümer
  - gleiches aktuelles Verzeichnis
  - gleiche Dateien geöffnet (selbst Schreib-/Lesezeiger ist gemeinsam)
  - ...
- ◆ Unterschiede:
  - verschiedene PIDs
  - `fork()` liefert verschiedene Werte als Ergebnis für Vater und Kind

## 5.3 Ausführen eines Programms (UNIX)

- Prozess führt ein neues Programm aus

```
int execlve ( const char *path, char *const argv[],
             char *const envp[] );
```

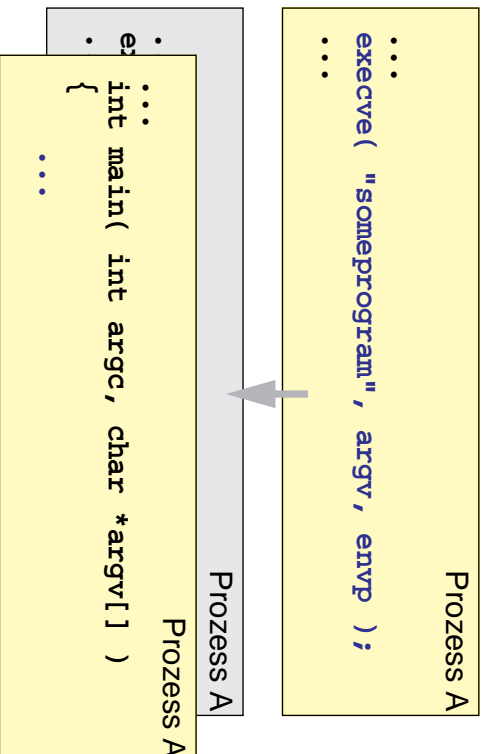
```
Prozess A
...
execlve ( "someprogram", argv, envp );
...
```



## 5.3 Ausführen eines Programms (UNIX)

- Prozess führt ein neues Programm aus

```
int execeve( const char *path, char *const argv[],
             char *const envp[] );
```



das vorher ausgeführte Programm ist dadurch endgültig beendet

- execeve kehrt im Erfolgsfall nie zurück

## 5.4 Operationen auf Prozessen (UNIX)

- Prozess beenden

```
void _exit( int status );
[ void exit( int status ); ]
```

- Prozess terminiert - exit kehrt nicht zurück

- Prozessidentifikator

```
pid_t getpid( void );           /* eigene PID */
pid_t getppid( void );        /* PID des Vaterprozesses */
```

- Warten auf Beendigung eines Kindprozesses

```
pid_t wait( int *statusp );
```

- Prozess wird so lange blockiert bis Kindprozess terminiert
- über den Parameter werden Informationen über den exit-Status des Kindprozesses zurückgeliefert