Entwicklungsumgebung zur Systemprogrammierung

Die Poolumgebung unter Linux beinhaltet zahlreiche Werkzeuge zur Systemprogrammierung. Zur Editierung von Systemprogrammen in der Programmiersprache C / C++ kann u.a.

- Standard-Editoren: vi, Vim, nano, joe
- Editoren mit graphischer Benutzungsschnittstelle: JEdit oder gedit
- Integrierte Entwicklungsumgebungen: CLion, Visual Studio Code, Codeblocks oder Eclipse

verwendet werden. Hier sollte sich jeder des bevorzugten Editors bedienen. Neuerdings unterstützt auch Visual Studio Tools zur Entwicklung von Linux-Software aus einer Windows-Umgebung heraus. Zum ggw. Zeitpunkt kann die Nutzung aber noch nicht uneingeschränkt empfohlen werden.

Die Compilation von C-Sourcen kann mit dem GNU-C-Compiler erfolgen.

Zum Test der Umgebung sollte man einmal das folgende prüfen:

- Erstellen eines C-Programms (z. B. **`HelloWorld.c`**) mit dem Editor der Wahl.
- Kompilation des Programms mit dem GNU-C-Compiler. Dieses kann z. B. in einem Terminalfenster erfolgen.
  `gcc -Wall –o hello HelloWorld.c`
- (Alternativ kann auch ein **`Makefile`** erstellt werden und über den Make-Mechanismus kompiliert und das ausführbare Programm erstellt werden)
- Starten des Programms in einem Terminalfenster.
  `hello`

Im Terminal erscheint dann die Ausgabe des Programms. Für das Debugging kann das Programm **gdb** verwendet werden. Der Debugger ist in die o. g. Entwicklungsumgebungen integriert.

Übersicht API zur POSIX-Systemprogrammierung

Um eine gewisse Portabilität des Codes zumindest zwischen verschiedenen Unix-/Linux-Systemen zu erreichen, sollten für die Systemprogrammierung POSIX-kompatible Schnittstellen für Systemaufrufe verwendet werden. Die wichtigsten Funktionen, die über Bibliotheken und diverse Header-Dateien (*unistd.h, sys/types.h, sys/wait.h, sys/stat.h, signal.h* etc.; siehe die Manual-Seiten zu den Systemfunktionen) verfügbar gemacht werden, werden in im Folgenden aufgelistet.

Für die Umsetzung der Funktionen (z. B. das Kopieren von Dateien auf einem Dateisystem), die von Kommandos auf Shell-Ebene aufgerufen werden, werden vom Betriebssystem Basisroutinen verwendet. Die Basisroutinen laufen dabei im Kernel-Mode und haben deshalb vollen Zugriff auf Betriebsmittel und die damit verbundenen Daten.

Die Abarbeitung des Aufrufs einer Routine wird über Interrupts initiiert:

1. die Register des Kernels werden auf dem Stack gesichert,
2. anschließend wird die Routine (auch Service-Routine genannt) ausgeführt und
3. schließlich wird einer Rückgabewert (ggf. Fehlercode) erzeugt. Die **`errno`**-Variable wird dabei durch die Wrapperfunktion beschrieben.

Prozessmanagement:

Aufruf	Beschreibung
`pid = fork()`	Erzeugt einen neuen Kind- vom Elternprozess
`pid = waitpid(pid, &statloc, options)`	Wartet auf Beendigung eines Kindes
`s = execve(name, argv, environp)`	Speicherabbild des aktuellen Prozesses ersetzen
`exit(status)`	Prozess beenden und Status zurückgeben

Dateimanagement:

Aufruf	Beschreibung
`fd = open(file, how, …)`	Datei öffnen (zum Lesen, Schreiben)
`s = close(fd)`	Offene Datei schließen
`n = read(fd, buffer, nbytes)`	Daten aus Datei in Puffer lesen
`n = write(fd, buffer, nbytes)`	Daten aus Puffer in Datei schreiben
`position = lseek(fd, offset, whence)`	Dateizeiger bewegen
`s = stat(name, &buf)`	Status einer Datei ermitteln

Verzeichnis- und Dateimanagement:

Aufruf	Beschreibung
`s = mkdir(name, mode)`	Erzeugen eines neuen Verzeichnisses
`s = rmdir(name)`	Löschen eines leeren Verzeichnisses
`s = link(name1, name2)`	Neuer Eintrag **`name2`** zeigt auf **`name1`**
`s = unlink(name)`	Verzeichniseintrag löschen
`s = mount(spezial, name, ag)`	Dateisystem einhängen
`s = umount(special)`	Eingehängtes Dateisystem entfernen

Verschiedenes:

Aufruf	Beschreibung
`s = chdir(dirname)`	Wechsel des aktuellen Verzeichnisses
`s = chmod(name, mode)`	Änderung der Dateirechte
`s = kill(pid, signal)`	Signal an einen Prozess schicken
`seconds = time(&seconds)`	Zeit seit dem 1. Januar 1970 in Sekunden abfragen. Schaltsekunden werden nicht mitgezählt.

Prozessverwaltung unter POSIX

Erzeugen eines neuen Prozesses

In einem POSIX konformen System werden neue Prozesse mit dem **`fork()`** Befehl erzeugt. Der resultierende Prozess ist eine exakte Kopie (Daten, Instruktionen, Zustand, auch PC werden exakt kopiert) des erzeugenden Prozesses (ergo: Elternprozess), der den Aufruf tätigt. Die Kopie unterscheidet sich nur in der Prozess-Id vom Elternprozess.

Syntax

```
#include <unistd.h>
pid_t fork(void);
```

`fork()` erzeugt einen **neuen** Prozess (Kindprozess). Einige Eigenschaften des Kindprozesses:

- Der Kindprozess erhält eine vollständige Kopie des Speichers des Elternprozesses.
- Der Kindprozess erhält eine eindeutige Id.
- Der Kindprozess erbt alle offenen Dateien des Elternprozesses.
- Eltern- und Kindprozess laufen jeweils nach dem **`fork()`** Aufruf in einem eigenen Kontext weiter.

Rückgabewert

- **`-1`**: falls **`fork()`**fehlschlägt. Der Rückgabewert muss immer geprüft werden! Der Fehlergrund kann **`errno`** entnommen werden.
- **`0`**: Der neu erzeugte Kindprozess bekommt an dieser Stelle die 0 zurückgeliefert. Somit kann hieran kann der Kindprozess erkannt werden.
- **`> 0`**: Der Elternprozess erhält die Prozess-Id des Kindprozesses zurück.

Synchronisation des Elternprozesses mit den Kindprozessen

Zur zeitlichen Synchronisation von Abfolgen von Befehlsaufrufen kann es notwendig sein, in einem Prozess auf die Beendigung eines anderen Prozesses zu warten.

Syntax

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int* stat_loc, int options);
```

Parameter

`pid`:

- **`> 0`**: Warte auf den Prozess mit der ID **`pid`**.
- **`= 0`**: Warte auf die Termination eines Prozesses aus der gleichen Prozessgruppe.
  Eine Prozessgruppe ist eine Menge von Prozessen, die zusammen Signale (zum Beenden eines Prozesses kann z. B. das Signal **`SIGKILL`** ausgelöst werden) empfangen können. Ein Kindprozess ist in derselben Prozessgruppe wie sein Elternprozess. Prozesse können eigene Prozessgruppen starten.
- **`-1`**: Warte auf einen (beliebigen) Kindprozess.

- **< -1**: Warte auf einen Prozess, dessen Gruppen ID dem Betrag von **pid** entspricht.

stat_loc:

- Ein Zeiger auf eine **int** Variable, in der der Status des Prozesses, auf die gewartet wurde, geschrieben wird.

Der Wert des Parameters ist von der jeweiligen Betriebssystem-Version abhängig. Deshalb muss bei einer portablen Programmierung die Auswertung des Parameters über Makros erfolgen. Es stehen dabei unter anderem folgende Makros zur Verfügung (für mehr: siehe Manual Pages ….):

- **WIFEXITED(stat_loc)**: liefert **TRUE**, wenn der Kindprozess sich normal beendet hat.
- **WEXITSTATUS(stat_loc)**: liefert den exit-Code des Kindprozesses (untere 8 Bit), falls dieser sich normal beendet hat.
- **WIFSIGNALED(stat_loc)**: liefert **TRUE**, wenn der Kindprozess durch ein Signal (ausgelöst z. B. per **<CTRL>-<C >** oder durch den Befehl **kill <pid>**) beendet wurde.
- …

Sofern der Rückgabewert nicht berücksichtigt werden muss, kann ein NULL-Zeiger übergeben werden.

options:

- Dieser Parameter beschreibt, wie sich der **waitpid()** Befehl verhält. Die möglichen Optionen sind Bitwerte, die Oder-verknüpft werden können. Z. B.:
  WNOHANG: waitpid()
  suspendiert den Elternprozess nicht, sondern kehrt direkt wieder zurück. Diese Einstellung könnte genutzt werden, um Prozesse nebenläufig im Hintergrund zu starten. Der abfragende Prozess blockiert dann nicht.

Weitere Optionen können den Manual Pages oder der unten angegebenen Literatur entnommen werden. Es ist zu bemerken, dass für dieses Praktikum keine Optionen benutzt werden (es wird eine 0 übergeben) müssen.

Rückgabewert

Die Prozess-Id des Prozesses, auf den gewartet wurde.

(Alternativ kann auch der **wait()** Befehl benutzt werden, der auf einige Parameter verzichtet und damit weniger flexibel ist.)

Beenden eines Prozesses

Syntax

```
#include <stdlib.h>

void exit(int status);
```

- Beendet den aktuellen Prozess.

Parameter

- **status**: Wird an den wartenden (Eltern-)Prozess zurückgeliefert.

Achtung: Sollte kein Elternprozess mit **waitpid()** auf das durch den Aufruf von **exit** generierte Signal warten, wird der terminierende Prozess zu einem Zombie Prozess, er kann nicht korrekt beendet und seine Ressourcen freigeben werden.

Austausch des Speicherbildes

Nach der Erzeugung eines neuen Prozesses per **fork()** wird der Kindprozess in der Regel andere Aufgaben übernehmen, d. h. anderen Code ausführen. Dazu muss u. a. Maschinencode des Prozesses, also das Programm, welches ausgeführt wird, durch anderen Code ersetzt werden, wobei der Prozess-Kontext (z. B. User-Id des Prozess-Eigentümers) weitgehend erhalten bleibt.

Syntax

```
#include <unistd.h>

extern char **environ;

int exec<l|v>[e|p] (const char *path, … parameter [, char* const
envp[]]);
```

- Die **exec** – Befehlsfamilie lädt ein neues Programm. Der Speicherinhalt des laufenden Prozesses wird dabei ausgetauscht. Wird das neue Programm erfolgreich gestartet, so läuft es mit der Prozess Id des Prozesses, der **exec** aufgerufen hat.
- Es gibt 6 Möglichkeiten des Aufrufes:
  - **execl[e|p]**: Die Parameter, die der **main**-Funktion des zu startenden Programms übergeben werden, werden in Listenform transferiert. Der letzte Parameter der Liste muss ein Null- Zeiger sein.
    Verwendung: **execl(const char* path, char *arg0, char *arg1, …);**
  - **execv[e|p]**: Die Übergabeparameter, die der main-Funktion des zu startenden Programms übergeben werden, werden als Vektor angegeben. Das letzte Vektorelement muss ein Null-Zeiger sein.
    Verwendung: **execv(const char* path, char *argv[]);**
    (vergleiche **argv[]** als Parameter der **main()** Funktion.)
  - **exec<l|v>**: Nur der Pfad und die Argumente werden übergeben.
  - **exec<l|v>e**: Als letzter Parameter wird ein Vektor auf die Umgebungsvariablen übergeben. Die externe Variable **environ** ist ein Verweis auf ein Feld mit den Umgebungsvariablen.
  - **exec<l|v>p**: Der Pfad des übergebenen Befehls wird in der Umgebungsvariable **$PATH** gesucht.

Beispiele

```
char * cmd[] = {"ls", "-l", (char *)NULL}; /* gilt für alle */
                                           /*  Beispiele    */

/* Beispiel 1: */
execl("/bin/ls", "ls", "-l", (char *)NULL);
```

```
/* Beispiel 2: */
execv("/bin/ls", cmd);

/* Beispiel 3: */
execve("/bin/ls", cmd, environ);

/* Beispiel 4: */
execvp("ls", cmd);

/* Beispiel 5: */
execle("/bin/ls", "ls", "-l", (char *)NULL, environ);
```

Weitere POSIX-Funktionen

`time()`: Ausgabe der aktuellen Systemzeit. Zu beachten:

- Die Zeit wird bei Unix auf Basis eines Zeitstempels ermittelt. Dabei werden Sekunden ab dem 1.1.1970 (UTC) gemessen und in einem Zähler abgelegt.
- Durch die Speicherung der Zeit als 32 Bit (signed) Integer (max. darstellbarer Wert ist $2^{31} - 1$ = 2147483647; das entspricht einem Zeitbereich von 1970 ± 68 Jahre) bei älteren Unix- und bei eingebetteten Systemen kann es systembedingt am 19. Januar 2038 um 3:14:08 h (UTC) zu einem Überlauf kommen.
- Bei Verwendung der unsigned-Darstellung kommt es im Jahr 2106 zu einem Problem.
- Moderne Unix-Varianten verwenden zur Speicherung einen 64 Bit signed Integer Wert.

Referenzen

- Steve Grägert: POSIX-Programmierung mit UNIX,
  siehe Dateibereich der Veranstaltung
- W. Richard Stevens, Stephen A. Rago: Advanced Programming in the UNIX Environment. Second Edition, Addison-Wesley Professional, 2008. ISBN 0321525949
- Bruce Molay: Understanding Unix/Linux programming, Prentice Hall, 2003. ISBN 0140083968
- M. Mitchell, J. Oldham, A. Samuel: Advanced Linux Programming. New Riders Publishing; 2001 (Online verfügbar: http://www.advancedlinuxprogramming.com/alp-folder)
- Andrew S. Tanenbaum: Moderne Betriebssysteme. Pearson Studium, 3., aktualisierte Auflage.