BERUFSAKADEMIE SACHSEN
STAATLICHE STUDIENAKADEMIE
DRESDEN
UNIVERSITY OF COOPERATIVE EDUCATION

6

OOP – Linux

Anwendungs-entwicklung unter/für Linux

mit Skriptmaterial von Dr.-Ing. M. Feldmann

Prof. Dr.-Ing. Tenshi Hara
tenshi.hara@ba-dresden.de

Gliederung der Vorlesung

Inhalte

- GCC
- Make
- GDB
- Manpages
- POSIX am Beispiel (exec, fork, pipe)
- Docker für DevOps

Zentrale Werkzeuge

GCC

- GNU Compiler Collection
- Sammlung enthält Compiler für die Programmiersprachen C, C++, Java, Objective-C, Fortran, Ada und Go
- Alternative: Clang

GDB

- GNU Debugger
- De-facto-Standard-Debugger von Linux-Systemen
- besitzt keine eigene grafische Oberfläche, nutzt die Standard-Ein-/Ausgabe

Make

- automatisiert alle Arbeitsschritte zur Erzeugung von Programmen aus mehreren Quellcodedateien (Übersetzung, Linken, Dateien kopieren etc.)

GCC

- Programm zur Berechnung von Primzahlen zwischen 1 und 50
- abgespeichert in Datei `primzahl.c`

```
#include <stdio.h>
int pruefe_primzahl(int zahl) {
  int teiler=2;
  while (teiler*teiler <= zahl) {
    if (zahl % teiler == 0)
      return(0);
    teiler++;
  }
  return(1);
}
int main( ) {
  int zahl;
  for (zahl=1; zahl<=50; zahl++)
    if (pruefe_primzahl(zahl))
      printf("%d\n", zahl);
  return(0);
}
```

- Kompilieren und Übersetzen in ausführbare Binärdatei:

```
user@linux$ gcc primzahl.c
            -o primzahl
```


Erzeugt Ausgabedatei `primzahl` und legt ausführbaren Programm-Code in diese ab

Aufteilung des Programmcodes

Datei primzahl.c mit main-Funktion

```
#include <stdio.h>
#include "primzahl_fct.h"
int main( ) {
  int zahl;
  for (zahl=1; zahl<=50; zahl++)
    if (pruefe_primzahl(zahl))
      printf("%d\n", zahl);
  return(0);
}
```

Datei primzahl_fct.h mit Deklaration der Funktion pruefe_primzahl(int)

```
int pruefe_primzahl(int zahl);
```

Datei primzahl_fct.c mit Definition der Funktion pruefe_primzahl(int)

```
#include <math.h>
int pruefe_primzahl(int zahl) {
  int teiler=2;
  while (teiler <= sqrt(zahl)) {
    if (zahl % teiler == 0)
      return(0);
    teiler++;
  }
  return(1);
}
```

Programm erstellen mit make

- Makedatei für Beispiel (gespeichert als Datei „Makefile“):

```
primzahl:       primzahl.o primzahl_fct.o
                gcc primzahl.o primzahl_fct.o -lm \
                -o primzahl

primzahl.o:     primzahl.c primzahl_fct.h
                gcc -c primzahl.c

primzahl_fct.o: primzahl_fct.c
                gcc -c primzahl_fct.c

clean:
        rm -f primzahl.o primzahl_fct.o
```

- Aufruf:

```
user@linux$ make primzahl
```

Debugging mit GDB

- Ändern der Zeile „`int teiler=2;`“ in „`int teiler=0;`“
- Übersetzen des Quellcodes mit GCC:

```
user@linux$ gcc -Wall -g primzahl.c -o primzahl
```

  - Option `-Wall`: Compiler soll alle Warnungen ausgeben
  - Option `-g`: Das Resultat wird mit Zusatzinformationen für das Debugging angereichert

- Aufruf des GDBs mit dem Programm:

```
user@linux$ gdb primzahl
```

- Innerhalb des GDBs kann das Programm mit dem Kommando *run* ausgeführt werden

Debugging mit GDB – Breakpoints

- Mittels GDB können mit dem Kommando *break* an verschiedenen Stellen während der Programmausführung Breakpoints gesetzt werden:

```
(gdb) break pruefe_primzahl
```

- Anschließend können die Variablenbelegungen ausgelesen werden
- Beispiel: Ändern der Zeile „`int teiler=2;`“ in „`int teiler;`“ und anschließendes Übersetzen

```
(gdb) break pruefe_primzahl
Breakpoint 1 at 0x80483ea: file primzahl_full.c, line 5.
(gdb) run
Starting program: /home/tenshi/cScripting/primzahlErr

Breakpoint 1, pruefe_primzahl (zahl=1) at primzahl_full.c:5
5               While (teiler*teiler <= zahl) {
(gdb) print teiler
$1 = 134520820
```

Manpage zu Programm anlegen

- In Datei `PROGRAMMNAME.NUMMER_FUER_MAN-KATEGORIE` kann in spezieller Syntax eine Manpage definiert werden: `primzahl.1`
- Nachdem Datei mit `gzip` gepackt wurde, kann sie in das Verzeichnis der Manpages abgelegt werden, z.B.: `/usr/share/man/man1`
- Syntax für Manpages:

`.\"`	Kommentarzeile
`.TH`	Titel-/Kopf-/Fußzeile definieren
`.SH`	Überschrift für den nächsten Absatz
`.fi`	Blocksatz in der Folge verwenden
`.nf`	Normale Formatierung in der Folge verwenden
`\fR`	normaler Text
`\fB`	Fettschrift

Manpage zu Programm Anlegen

```
.TH primzahl 1 "September 2017" Vorlesung UNIX an der BA
.SH NAME
primzahl \- Berechnet Primzahlen von 1 bis 50.
.SH SYNTAX
primzahl
.SH BESCHREIBUNG
.fi
Das Programm \fBPrimzahl\fR berechnet alle Primzahlen von 1 bis 50. Es
dient als Beispielprogramm für die Vorlesung UNIX an der BA Dresden.
```

```
user@linux$ gzip primzahl.1
```

```
user@linux$ sudo cp primzahl.1.gz /usr/share/man/man1
```

```
primzahl(1)                  an                  primzahl(1)

NAME
       primzahl - Berechnet Primzahlen von 1 bis 50.

SYNTAX
       primzahl

BESCHREIBUNG
       Das Programm Primzahl berechnet alle Primzahlen von 1
       bis 50.  Es dient als Beispielprogramm für die Vorle-
       sung UNIX an der BA Dresden.

UNIX-Vorlesung          September 2017          primzahl(1)
: Manual page primzahl(1) line 1/18 (END)
```

Ausgewählte POSIX-Funktionen – Exec

- `exec` ist Oberbegriff für fünf verschiedene Funktionen, mit denen ein Programm ausgeführt werden kann:
  - `execl`, `execlp`, `execle`, `execv`, `execvp`
  - Unterscheiden sich in Form der Parameterübergabe
- Programm wird als Zeichenkette angegeben
- Im Gegensatz zur Funktion `system` wird keine neue Shell gestartet, sondern das Programm wird im aktuellen Prozess ausgeführt
- Beispiel: Syntax von `execve`

```
execve("PFADZUPROGRAMM", ARGUMENTE, UMGEBUNGSVARIABLEN)
```

Ausgewählte POSIX-Funktionen – Exec

Beispielprogramm:

```
#include <stdio.h>
#include <unistd.h>
int main(){
        char *argv[4], *env[2];

        argv[0] = "/bin/ls";
        argv[1] = "--color";
        argv[2] = "/home/user";
        argv[3] = NULL;
        env[0] = "LS_COLORS=fi=00:di=01";
        env[1] = NULL;

        execve("/bin/ls", argv, env);

        perror("execve() failed");
}
```

Ausgewählte POSIX-Funktionen – fork

`fork`: erzeugt einen neuen Kindprozess des aktuellen Prozesses

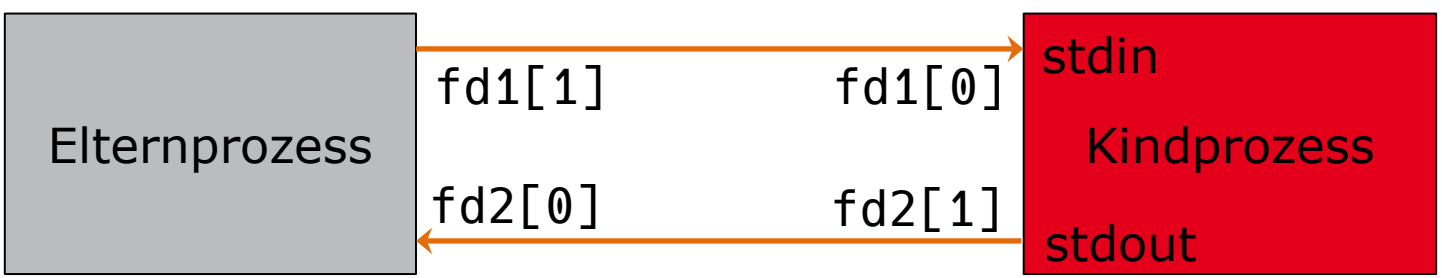
- Eingabe: keine
- Rückgabe: die Prozess-Id (pid) des neu erzeugten Prozesses
- Alternativen für parallele Ausführung: `clone( )`, Threads

Programmlogik
für Kind-Prozess


```
#include <stdio.h>
#include <unistd.h>
int main( ) {
  int pid_child;
  if((pid_child = fork( )) == 0)
    printf("Dies ist eine Ausgabe des Kind-Prozesses.\n");
  else if(pid_child > 0)
    printf("Ein Kind-Prozess mit der Id %d wurde erzeugt.\n",pid_child);
  else
    perror("fork( ) ist fehlgeschlagen...");
}
```

Ausgewählte POSIX-Funktionen – Pipes

- über Pipes können verschiedene Prozesse miteinander zum unidirektionalen Austausch verbunden werden
- Funktion `pipe` erzeugt eine virtuelle Datei, in die ein Prozess schreiben und aus der ein anderer lesen kann:
  `int pipe(int pipefd[2])`
  - Eingabeparameter: Feld, in das die Funktion den Deskriptor für die Eingabe und den Deskriptor für die Ausgabe schreibt
  - `pipefd[0]` realisiert die Eingabe, `pipefd[1]` die Ausgabe
- besonders interessant, um die Standardeingabe und Standardausgabe von Kindprozessen umzuleiten



Ausgewählte POSIX-Funktionen – Pipes

Beispiel (Teil 1):

```
int main( ) {
  int fd1[2], fd2[2], ref;
  char buffer[80];
  if ((pipe(fd1) != 0) || (pipe(fd2) != 0)) {
    perror("pipe2: pipe( ) failed");
    return(1);
  }
  if (fork( ) == 0) {
    close(fd1[1]);
    close(fd2[0]);
    if ((dup2(fd1[0], STDIN_FILENO) == -1)
    || (dup2(fd2[1], STDOUT_FILENO) == -1)) {
      perror("pipe2: dup2( ) failed");
      return(1);
    }
    close(fd1[0]);
    close(fd2[1]);
    execlp("sort", "sort", NULL);
    perror("pipe2: execlp( ) failed");
    return(1);
  }
```

benötigt als Import:
- `stdio.h`
- `unistd.h`
- `sys/wait.h`

dup2 dupliziert einen File-Deskriptor

nicht benötigte Deskriptoren können geschlossen werden

Ausgewählte POSIX-Funktionen – Pipes

Übertragt Daten an Kindprozess (`fd1[1]` ist mit `stdin` des Kindprozesses verbunden)

Beispiel (Teil 2):

```
  close(fd1[0]);
  close(fd2[1]);

  write(fd1[1], "These\nlines\nshall\nbe\nsorted\n", 28);
  close(fd1[1]);
  wait(NULL);

  if ((ref = read(fd2[0], buffer, 79)) == -1)
    perror("pipe2: read() failed");
  else
    write(STDOUT_FILENO, buffer, ref);

  close(fd2[0]);
  return(0);
}
```

Lesen der Ausgabe des Kindprozesses

Docker für DevOps

- Docker basiert auf Container-Technologien und einem Image-Repository (bzw. mehreren Repositories)

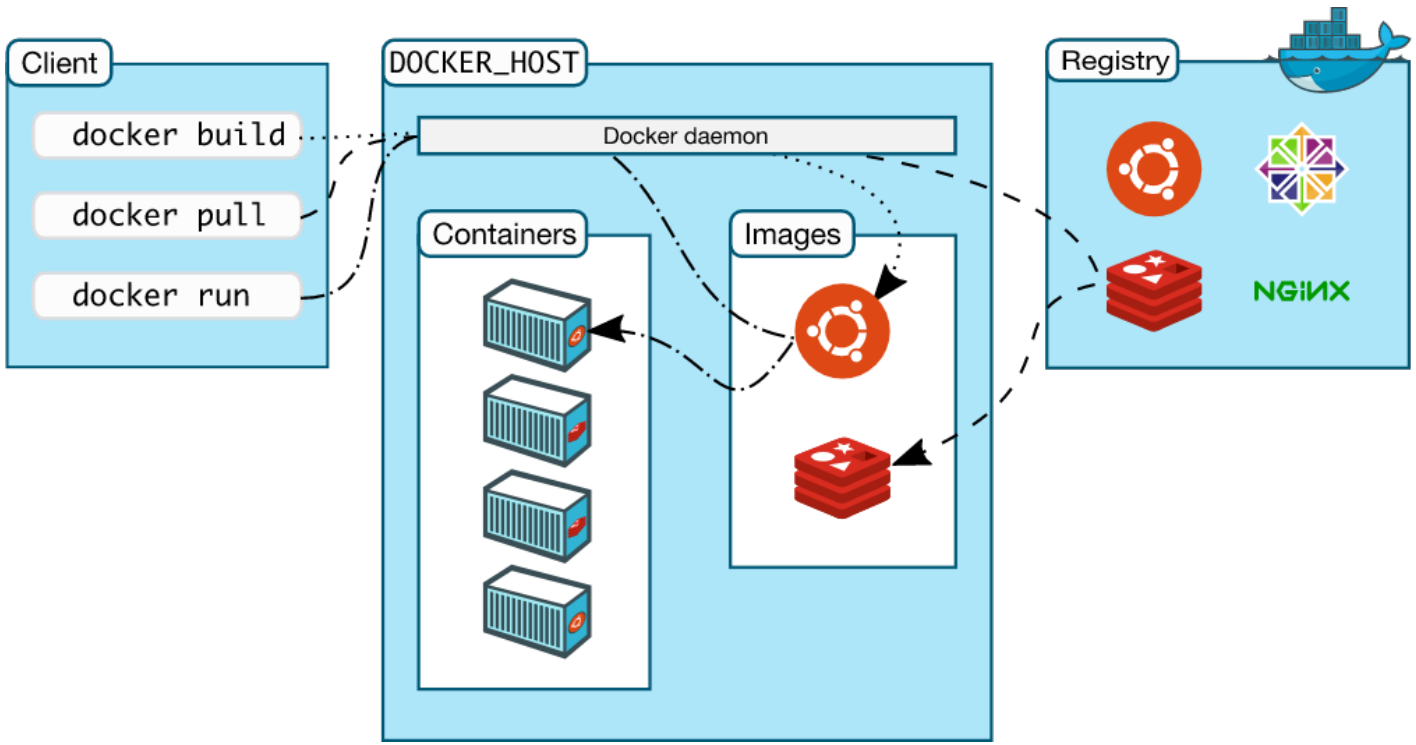


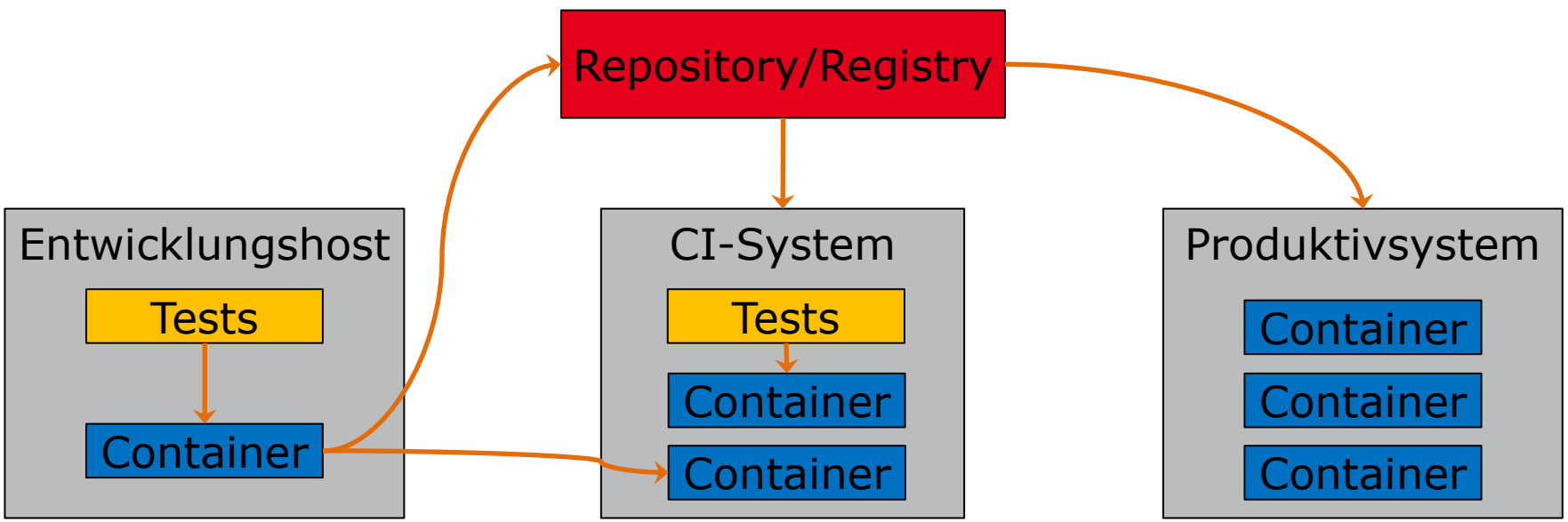
Abb. : https://docs.docker.com/engine/docker-overview/

- Docker nutzt Namespace-Isolierung, siehe z.B.:
  - `docker inspect --format '{{.State.Pid}}' <container_name_or_Id>`
  - `nsenter -t <contanier_pid> -n <command>`

Docker – Einsatzszenarien

Argumente für den Einsatz von Docker sind u.a.:

- Isolation von Ausführungskontext (verbesserte z.B. Sicherheit)
- einfache Testumgebung mit der Möglichkeit, Software für verschiedene Betriebssysteme zu testen
- leichtgewichtiger als reguläre Virtualisierungstechnologie
- Testumgebung soll identisch zur späteren Ausführungsumgebung im Produktivsystem sein (vgl. DevOps-Ansatz)



Erstellen eines Images

- eigene Images können mittels einer Beschreibungsdatei, dem sogenannten **Dockerfile** spezifiziert werden
- Syntax in Dockerfile ähnelt typischer Kommandozeilensystax, siehe: https://docs.docker.com/engine/reference/builder/
- einfaches Dokerfile:

- sofern Dockerfile in lokalem Verzeichnis abgelegt wurde, kann das zugehörige Image erstellt werden mittels:

```
user@linux$ sudo docker build -t autocompleter
```

Aufgaben

- Laden Sie https://upcn.eu/autocompleter.zip herunter und verschaffen Sie sich einen Überblick über die Anwendungsstruktur
- Folgen Sie den Anweisungen in der Datei `README.md` und setzen Sie die Anwendung in Betrieb; für diesen Schritt ist eine Anpassung der Datenbankkonfiguration in `config.json` erforderlich und es müssen Abhängigkeiten aufgelöst werden
- Installieren Sie Docker, z.B. nach dem folgenden Tutorial: https://docs.docker.com/engine/getstarted/linux_install_help/
- Erstellen Sie ein Dockerfile, durch das ein Docker-Image erstellt wird, durch das die Webanwendung automatisch ausgeführt werden kann
- Die folgenden syntaktischen Elemente im Dockerfile werden dazu benötigt: `FROM`, `RUN`, `EXPOSE`, `WORKDIR`, `ENTRYPOINT`
- Der Docker-Container soll gestartet werden können mittels:

```
sudo docker run -d -p 8080:8080 -it -v /home/ubuntu/autocompleter/
config.json:/home/ubuntu/autocompleter/config.json autocompleter
```

- Empfohlene weiterführende Lektüre: https://docs.docker.com/engine/getstarted-voting-app/