TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

Beispiel Block 2

Stream I/O
Parallele Prozesse: fork, exec und wait
Interprozesskommunikation mit Unnamed Pipes

Christian El Salloum
SS 2011

1

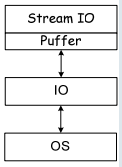
File Descriptors

- Verweis auf Eintrag in Tabelle offener Dateien (file descriptor table)
- Standard I/O
  - STDIN_FILENO= 0 (usually same as fileno(stdin))
  - STDOUT_FILENO = 1
  - STDERR_FILENO = 2
- Funktionen: open(2),close(2),read(2),write(2),...

2

Stream IO in C

- Stream IO baut auf File-Deskriptoren auf
  - `#include <stdio.h>`
- Stream Datentyp: `FILE`
- Gepuffert (siehe *fflush(3)*)
- Konvention: Befehle beginnen mit ‚f' *fopen(3), fdopen(3), fwrite(3), fprintf(3), …*
- *stdin, stdout, stderr* sind vordefinierte Streams



3

fopen(3)

```
FILE *fopen(const char *path, const char *mode);
```

Die Datei *path* wird geöffnet, und mit Stream (Rückgabewert) verbunden

mode:

„*r*“ nur lesen
“*w*“ nur schreiben (existierenden Inhalt löschen)
“*a*“ nur schreiben (am Ende anhängen)
„*r*+“/“*w*+“/“*a*+“ lesen und schreiben

4

fdopen(3)

```
FILE *fdopen(int fildes, const char *mode);
```

Assoziiert einen Stream mit einem Filedescriptor

```
FILE* f; int fd;
int fd = socket(AF_INET, SOCK_STREAM,0);
…
f = fdopen(fd, "r+");
fprintf(f,"Meine Prozess-ID ist: %d\n",
        getpid());
```

5

fflush(3), fclose(3)

```
int fflush(FILE *stream);
int fclose(FILE *stream);
```

- **`fflush`** erzwingt das Schreiben von gepufferten Daten
- **`fclose`** ruft **`fflush`** auf und schließt den Stream sowie den zugrundeliegenden Deskriptor.

6

Lesen/Schreiben

Funktion	
fread	Lesen von n Elementen a size Bytes
fgets	Lesen einer Zeile
fgetc	Lesen eines Zeichens
fwrite	Schreiben von n Elementen a size Bytes
fputs	Schreiben eines C-Strings
fprintf	Formatiertes Schreiben
fputc	Schreiben eines Zeichens
fseek	Positionieren des Dateipositionszeigers

7

ferror(3), feof(3)

```
int ferror(FILE *stream);
int feof(FILE *stream);
int clearerr(FILE *stream);
```

- **`ferror`** ergibt den Fehlerstatus des Streams zurück (0 ~ error flag nicht gesetzt).
- **`feof`** fragt ab, ob das End-Of-File Flag des Streams gesetzt ist (wird beispielsweise von fgets gesetzt, wenn das Ende der Datei erreicht wird)

8

Beispiel

```
#define SIZE 512
int main(int argc, char **argv)
{ char buffer[SIZE];
  FILE *f;
  …
  f = fopen(argv[1],"r");
  while (fgets(buffer,SIZE,f)!=NULL)
    { fputs(buffer,stdout); }
  if (ferror(f)) bail_out("IO Error");
  return 0; }
```

9

Prozesse

- Hierarchie
- Jeder Prozess hat Vaterprozess
- Ausnahme: `init`
- Jeder Prozess hat eine eindeutige ID (`pid_t`)

```
init─┬─acpid
     ├─ahc_dv_0
     ├─ahc_dv_1
     ├─bash
     ├─clock-applet
     ├─crond
     ├─cups-config-dae
     ├─cupsd
     ├─2*[dbus-daemon-1]
     ├─dbus-launch
     ├─dhcpd
     ├─gdm-binary─┬─gdm-binary─┬─X
     │            │            └─gdmgreeter
     │            └─gdm-binary──  ...
     ├─2*[sendmail]
     ├─sesam_server─┬─sesam_server
     │              └─sesam_server─  ...
     ├─smbd──5*[smbd]
     ├─sshd─┬─sshd──sshd──bash──pine
     │      ├─sshd──sshd──bash──pine
     .      .
```

10

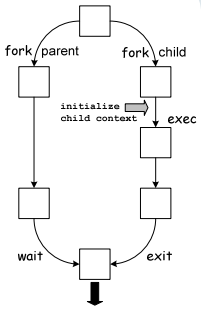
Erstellen von Prozessen

- Prozesse werden üblicherweise mit *fork(2)* erzeugt.
  - Weitere Möglichkeiten *clone(2)*, *vfork(2)*
  - In der Übung ist ausschließlich *fork(2)* zu verwenden
- *exec(3)* überschreibt den aktuellen Prozess durch ein anderes Programm.
- *wait(3)* wartet auf die Terminierung eines Kindes

11

fork() / exec() / wait()

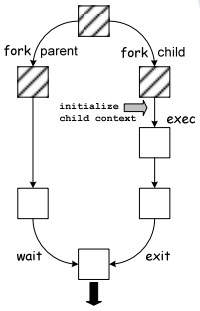
- **`fork()`**
  - erzeugt neuen Prozess
- **`exec*(„program“)`**
  - ersetzt *image* eines Prozesses durch neues Programm
- **`exit(status)`**
  - Beendet Prozess
- **`wait*(&status)`**
  - wartet auf Beendigung des Kindprozesses



12

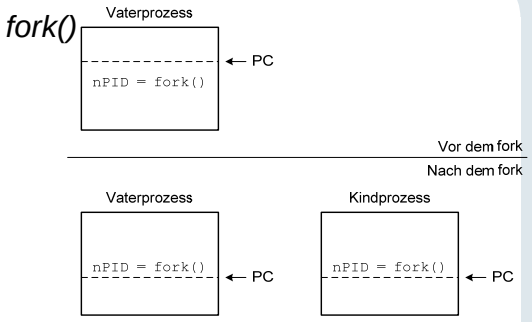
fork()

- Erzeugt einen neuen Prozess
- Neuer Prozess ist eine identische Kopie des aufrufenden Prozesses (bis auf PID, ...)
- Erzeugender Prozess ist Vater des erzeugten Prozesses
- Beide Prozesse laufen parallel und führen dasselbe Programm aus



13

fork()



14

fork()

Unterscheidung zwischen Vater- und Kindprozess durch den Rückgabewert von *fork():*

- `-1` im Fehlerfall
- `0` im Kindprozess (child)
- `>0` im aufrufenden Prozess (parent)

15

fork()

Kindprozess erbt von Elternprozess

- offene Dateien (gemeinsamer Zugriff!)
- Dateipuffer
- Signaldefinitionen
- **momentane** Variablenwerte

Jedoch gilt:

- Variablen sind lokal (keine Beeinflussung)
- Signale können lokal umgesetzt werden
- Kommunikation (IPC) via Pipes, Sockets, Shared Memory, ...

16

C-Interface

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

17

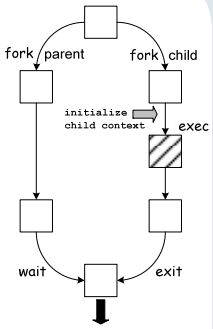
Verwendung von *fork()*

```
pid_t nPid;
...
switch (nPid = fork()) {
  case -1:  BailOut ( "can't fork" );
            break;
  case 0:   /* child: ChildProcess()   */
            ...
            exit(EXIT_SUCCESS);
            break;
  default:  /* parent: ParentProcess() */
            ...
            break;
}
```

18

exec()

- Erlaubt es einem Prozess, ein anderes Programm auszuführen
- Startet ein neues Programm innerhalb eines Prozesses
- PID bleibt gleich



19

exec()

```
int execl(const char *path, const char *arg,
  ...);
int execlp(const char *file, const char *arg,
  ...);
int execle(const char *path, const char *arg,
  ... , char * const envp[]);
int execv(const char *path, char *const
  argv[]);
int execvp(const char *file, char *const
  argv[]);
int fexecve(int fd, char *const argv[], char
  *const envp[]);
```

20

exec()

- Exekutiert das unter `filename` abgespeicherte Image
- Argumentübergabe beachten!

```
#include <unistd.h>

int execv(char *pathfilename, char *argv[] );
int execvp(char *filename, char *argv[] );

char *cmd[] = { "ls", "-l", (char *)0 };
(void) execv ("/bin/ls", cmd);
(void) execvp ("ls", cmd);
BailOut ( "can't exec" );
```

21

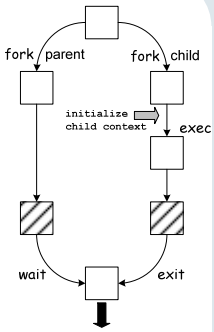
exec()

```
#include <unistd.h>

int execl(char *path_filename, char *arg0,
                          ..,
                          char *argN,(char*)0);
int execlp(char *filename, char *arg0,
                          ..,
                          char *argN,(char*)0);

(void) execl ( "/bin/ls", "ls", "-l", (char*) 0 );
(void) execlp ( "ls", "ls", "-l", (char*) 0 );
BailOut ( "can't exec" );
```

22

wait()

- Wartet bis Kindprozess terminiert
- Liefert PID des terminierten Prozesses
- Wenn kein Kindprozess existiert, -1 als Rückgabewert (auch bei EINTR)



23

exit()

Terminiert den aktuellen Prozess

```
void exit(int status);
```

`exit(EXIT_SUCCESS)`	keine Fehler
`exit(EXIT_FAILURE)`	Fehler aufgetreten

24

wait()

Liefert Status des terminierten Kindprozesses
`wait(&status)`

Es immer notwendig auf Terminierung des Kindes zu warten
- unter UNIX besetzen auch bereits terminierte Prozesse einen Eintrag in der Prozesstabelle
- falls kein Platz mehr frei ist, kann kein neuer Prozess mehr gestartet werden

25

waitpid(): verwandter Call to *wait()*

Warten auf das Terminieren eines Prozesses mit einer bestimmten Prozess-Id:

```
pid_t waitpid (pid_t Pid,
               int *pnStatus,
               int nOptions );
```

`waitpid(-1,&status,0)` equivalent zu `wait`
`waitpid(pid, &status, 0)` wartet auf Kind mit PID `pid`
`waitpid(-1,&status, WNOHANG)` blockiert nicht.

26

Zombies und Orphans

- Der Kindprozess terminiert und der Vaterprozess hat noch nicht `wait` ausgeführt
  - Der Kindprozess wird auf Zustand „Zombie“ gesetzt
  - Eintrag in der Prozesstabelle bleibt erhalten bis der Vaterprozess `wait` ausführt
- Der Vaterprozess terminiert und der Kindprozess läuft immer noch
  - Kindprozesse werden dem `INIT` Prozess vererbt

27

Wurde ein Kind beendet?

- `waitpid(-1,&status, WNOHANG)`
  - Blockiert nicht, holt Exit-Status falls ein Kind beendet wurde. (Polling)
  - Eleganter: Wenn ein Kind beendet wurde wird das Signal `SIGCHLD` an den Elternprozess gesendet.
- `signal(SIGCHLD, chterm);`
- `void chterm(int sig) { wait(…); … }`

28

Fallstricke

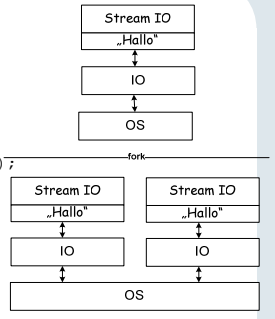
```
int main(…)
{
   fprintf(stdout,"Hallo");
   fork();
   return 0;
}
```

Ausgabe: `"HalloHallo"`
Warum?

29

Fallstricke

```
int main(…)
{
   fprintf(stdout,"Hallo");
   fork();
   return 0;
}
```



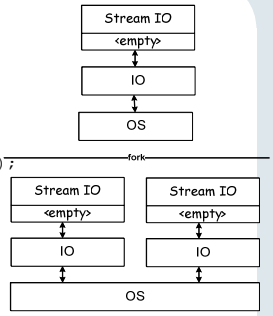
Ausgabe: `"HalloHallo"`
Warum?

30

Fallstricke

```
int main(…)
{
   fprintf(stdout,"Hallo");
   fflush(stdout);
   fork();
   return 0;
}
```

Ausgabe: `"Hallo"`



31

Rückblende

fork() - Kindprozess erzeugen

exec() - neues Programm innerhalb eines Prozesses starten

exit() – den aktuellen Prozess beenden

wait() - im Vaterprozess auf die Terminierung des Kindprozesses warten

32

Pipes

- Kommunikationskanal zwischen *verwandten* Prozessen
- Eigenschaften:
  - Unidirektional (2 Pipes für Übertragung in beide Richtungen)
  - „Stream“ von Daten
  - Implizite Synchronisation

33

Verwendung von Pipes (1)

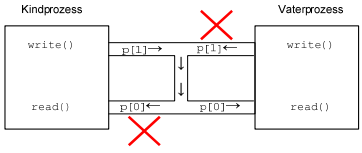
- Pipe wird mittels eines Feldes von zwei Integer-Elementen deklariert

  `int aFileDesc[2];`
- Deskriptor `aFileDesc[0]` ist Leseende
- Deskriptor `aFileDesc[1]` ist Schreibende
- Nicht verwendete Enden müssen geschlossen werden
  - Schreibender Prozess schließt das Leseende
  - Lesender Prozess schließt das Schreibende

34

Verwendung von Pipes (1)

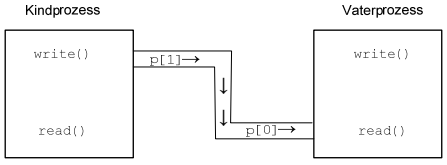
ohne geschlossene Enden



35

Verwendung von Pipes (1)

geschlossene Enden



36

Verwendung von Pipes (2)

- Öffnen einer Pipe mittels Systemaufruf *pipe()*
- *fdopen()* erstellt einen mit dem gegebenen Filedeskriptor assoziierten Stream (`FILE *`)
  - Verwendung von Funktionen, die auf Streams operieren
  - z.B. fopen(), fclose()
  - fclose()
- *dup2()* kann zum Umlenken von Dateien verwendet werden

37

C-Interface

```
#include <limits.h> /* for PIPE_BUF */
#include <unistd.h> /* prototype */

int aFileDesc[2];

if ( pipe ( aFileDesc ) != 0 )
{
  bail_out( "can't create pipe", 1 );
}
```

38

dup(2), dup2(2)

- `dup(`*int fildes*`)` dupliziert einen File-Deskriptor.
  - Der neue Deskriptor hat die niedrigste nicht verwendete ID
- `dup2(int old,int new)`
  - Schließt Filedeskriptor mit ID `new`
  - Dupliziert `old`, der neue Deskriptor erhält ID `new`

39

Umleiten der Standardein/ausgabe

- Anwendung: Kommunikation mit existierendem Programm (via fork/exec), welches über Standard eingabe / Standardausgabe kommuniziert
- Strategie: Umleiten der Standardeingabe (0) oder Standardausgabe (1) in neuem Prozess

40

C-Interface

```
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>
int fd;

fd = open(„log.txt",O_WRONLY|O_CREAT);
dup2(fd,                    /* old descriptor */
     fileno(stdout));       /* new descriptor */
close(fd);                  /* close old desc. */

(void) execlp("grep", "grep", „max", (char *) 0);
```

41

Synchronisation

- Lesen von leerer Pipe ist blockierend
- Schreiben auf volle Pipe ebenso
  - Achtung: Blockiert wenn vergessen Pipe zu schließen
- Lesen von Pipe ohne offene Schreibenden liefert EOF
- Schreiben auf Pipe ohne offene Leseenden liefert SIGPIPE Signal

42

Pipes: Pitfalls

- Pipes eignen sich gut für unidirektionale Kommunikation
- Bidirektional: Zwei Pipes
  - Fehleranfällige Synchronisation (deadlock)
- Synchronisation & Puffer
  - fflush() verwenden
  - Puffer konfigurieren (setbuf(3), setvbuf(3))

43

Rückblende

- Kommunikation zwischen verwandten Prozessen mittels Pipes.
- Implizite Synchronisation
- Nicht verwendete Enden schließen
- Filestream für Filedeskriptor der Pipe mittels fdopen()
- Umlenken von Pipes mittels *dup2()*

44

ENDE

Danke für die Aufmerksamkeit!

Real-Time Systems Group

05.04.2011 Block zum 2.Übungsbeispiel 45