Tafelübung zu BS

1. Prozesse verwalten

Olaf Spinczyk
Arbeitsgruppe Eingebettete Systemsoftware

Lehrstuhl für Informatik 12
TU Dortmund

olaf.spinczyk@tu-dortmund.de
http://ess.cs.uni-dortmund.de/~os/

Agenda

- Besprechung Aufgabe 0
- Fortsetzung Grundlagen C-Programmierung
- Aufgabe 1: Prozesse verwalten
  - Tastatureingaben mit scanf
  - Vergleich von Strings mit strcmp
  - Fehlerbehandlung
  - Unix-Prozessmanagement
    - fork()
    - execlp()
    - wait()
    - waitpid()
  - Signal-Handler

Besprechung Aufgabe 0

- → Foliensatz Besprechung

Grundlagen C-Programmierung

- → Foliensatz C-Einführung (Folie 32-46)

Tastatureingaben mit scanf(3)

hier: manpage section

```
int scanf(Formatstring,Variablenzeiger1,
                       Variablenzeiger2,…);
```

- liest Zeichen aus dem Eingabestrom (z.B. von der Tastatur) und konvertiert Teile davon in Variablenwerte
- kehrt zurück, wenn Formatstring abgearbeitet ist, oder wegen eines unpassenden Zeichens abgebrochen wurde
- benötigt **`#include`** `<stdio.h>`
- Parameter
  - *Formatstring* wie bei `printf()` mit *Umwandlungsspezifikatoren*
  - *Zeiger* auf die einzulesenden Variablen, Datentypen entsprechend der Reihenfolge im Formatstring
- Rückgabewert:
  - Anzahl der erfolgreichen Umwandlungen

scanf mit int - Beispiel

```c
#include <stdio.h>

int main() {
    int eastwood;
    printf("Bitte eine ganze Zahl eingeben> ");

    if (scanf("%d",&eastwood)<1) {
        printf("Fehler bei scanf!\n");
        return 1;
    }

    printf("Die Zahl ist %d.\n",eastwood);
    return 0;
}
```

```
streic00@lithium:~/example$ ./scanf_example.elf
Bitte eine ganze Zahl eingeben> 42
Die Zahl ist 42.
```

```
streic00@lithium:~/example$ ./scanf_example.elf
Bitte eine ganze Zahl eingeben> Pferd
Fehler bei scanf!
```

```
streic00@lithium:~/example$ ./scanf_example.elf
Bitte eine ganze Zahl eingeben> 42Pferd
Die Zahl ist 42.
```

scanf mit strings - Beispiel

```c
#include <stdio.h>

int main() {
    char guevara[42];
    printf("Bitte ein Tier eingeben> ");

    if (scanf("%41s",guevara)<1) {
        printf("Fehler bei scanf!\n");
        return 1;
    }

    printf("Das Tier ist: %s.\n",guevara);
    return 0;
}
```

```
streic00@lithium:~/example$ ./scanf_example.elf
Bitte ein Tier eingeben> Pferd
Das Tier is: Pferd.
```

```
streic00@lithium:~/example$ ./scanf_example.elf
Bitte ein Tier eingeben> 42
Das Tier ist: 42
```

- %s erwartet **array** argument (nicht &array!)
- %Ns liest string der Länge N in Puffer
- strings enden mit '\0'. Scanf fügt dies selbst ein. Platz lassen!

Vergleich von Strings mit strcmp(3)

- Strcmp vergleicht zwei Strings
  - bei Ungleichheit: Rückgabewert größer oder kleiner 0
  - bei Gleichheit: Rückgabewert 0

```
#include <stdio.h>
#include <string.h>

int main() {
    char tier[42];
    printf("Bitte ein Tier eingeben> ");

    if (scanf("%41s",tier)<1) {
        printf("Fehler bei scanf!\n");
        return 1;
    }

   if ((strcmp("Pferd", tier) != 0)) {
      printf("%s ist kein Pferd!\n", tier);
      return 1;
   }

    return 0;
}
```

```
mm@ios:~/example$ ./strcmp
Bitte ein Tier eingeben> Hund
Hund ist kein Pferd!
```

Unterschiede: Strings vs. Chars

- **char** verhält sich eher wie **int**
  - Kann man mit ==, <, > usw. vergleichen
  - Nimmt man in scanf mit %c entgegen

```
#include <stdio.h>
#include <string.h>

int main() {
    char tier[42];
    printf("Bitte ein Tier

    if (scanf("%41s",tier)
        printf("Fehler bei
        return 1;
    }

   if ((strcmp("Pferd", t
      printf("%s ist kein
      return 1;
   }

    return 0;
}
```

```
#include <stdio.h>
#include <string.h>

int main() {
    char buchstabe;
    printf("Buchstaben eingeben> ");

    if (scanf("%c",&buchstabe)<1) {
        printf("Fehler bei scanf!\n");
        return 1;
    }

   if (buchstabe != 'P') {
      printf("%c ist kein P!\n", buchstabe);
      return 1;
   }

    return 0;
}
```

Problem bei scanf mit %c

- %c nimmt jeden char an
  - Also auch das Newline ('\n') bei Enter

```
mm@ios:~/example$ ./chars
Bitte einen Buchstaben eingeben> x
Noch einen Buchstaben> a: x, b:

mm@ios:~/example$
```

```
#include <stdio.h>

int main() {
    char a, b;
    printf("Bitte einen Buchstaben eingeben> ");
    if (scanf("%c",&a)<1) {
       /* Fehlerbehandlung */
    }

    printf("Noch einen Buchstaben> ");
    if (scanf("%c",&b)<1) {
       /* Fehlerbehandlung */
    }

    printf("a: %c, b: %c\n", a, b);
    return 0;
}
```

Problem bei scanf mit %c

- Kleine Abhilfe: Leerzeichen vor %c
  - Schluckt allen Whitespace, ist in der man-Page dokumentiert

```
mm@ios:~/example$ ./chars
Bitte einen Buchstaben eingeben> x
Noch einen Buchstaben> y
a: x, b: y
mm@ios:~/example$
```

```
#include <stdio.h>

int main() {
    char a, b;
    printf("Bitte einen Buchstaben eingeben> ");
    if (scanf(" %c",&a)<1) {
       /* Fehlerbehandlung */
    }

   printf("Noch einen Buchstaben> ");
    if (scanf(" %c",&b)<1) {
       /* Fehlerbehandlung */
    }

   printf("a: %c, b: %c\n", a, b);
    return 0;
}
```

Fehlerbehandlung

- Insbesondere bei Systemcalls können Fehler auftreten (Exceptions quasi)
- Wie geht man in C mit solchen Bedingungen um?
  - C kennt keine Exceptions
  - Rückgabewert nutzen?

  → globale Variable: `errno`

- typisches Schema:

```
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>

void main() {
    while (someSystemCall()==-1) {
        /*Spezialfaelle behandeln*/
        if (errno==EINTR) continue;
        /*allgemeiner Fall*/
        perror("someSystemCall");
        exit(EXIT_FAILURE);
    }
    /* alles ok, weitermachen */

    /* ... */

    return 0;
}
```

Fehlerbehandlung

- `int errno`
  - benötigt `#include <errno.h>`
  - enthält nach einem gescheiterten Bibliotheksaufruf den Fehlercode
  - kann ansonsten beliebigen Inhalts sein!
  - *dass* ein Fehler auftrat, wird *meistens* durch den Rückgabewert -1 angezeigt (manpage zum Systemcall lesen!)
- `void perror(const char *s)`
  - benötigt `#include <stdio.h>`
  - gibt eine zum aktuellen Fehlercode passende Fehlermeldung *auf dem Standard-Fehlerkanal* aus
- `void exit(int status)`
  - benötigt `#include <stdlib.h>`
  - setzt den Exit-Code und beendet das Programm augenblicklich

FALSCH

```
...
someSystemCall();
if (errno) {
        ...
```

Ergänzung

- **`void exit(int status)`** (Fortsetzung)
  - kann in kritischen Fehlerfällen zum Beenden benutzt werden
  - sonst: nur Rückkehr zur main() Funktion kann ordentliches Aufräumen ermöglichen

```
void somewhere_deep_in_the_code() {
    ...
    if (fatal_error())  /* z.B. Programmzustand unbekannt */
        exit(EXIT_FAILURE);
    ...
}
```

- **`unsigned int sleep(unsigned int seconds)`**
  - benötigt **`#include`** `<unistd.h>`
  - Rückgabewert:
    - Sekunden, die noch zu schlafen sind (normalerweise 0)
    - Andere Werte als 0 sollten bei uns aber (noch) nicht auftreten ;-)

```
sleep(5); /*sleep for 5 seconds*/
```

execlp (3)

- `int execlp(const char *path, const char *arg, ...);`
  - benötigt **`#include`** `<unistd.h>`
  - Überschreibt die Prozessdaten im Speicher durch Daten aus Datei
  - Prozess wird sofort neu gestartet
  - Identisches Prozessobjekt, PID bleibt gleich
  - Im Fehlerfall, Rückgabewert -1 und errno gesetzt.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
    char befehl[42];
    if (scanf("%41s", befehl) < 1)  /* Begrenze die Puffernutzung. */
        return 1;
    execlp(befehl, befehl, NULL); /* Kehrt niemals zurück. */
    return 1;                                   /* Falls doch: Fehler.*/
}
```

fork (2)

- `pid_t fork(void)`
  - benötigt **`#include`** `<unistd.h>` (**`#include`** `<sys/types.h>` für `pid_t`)
  - erzeugt eine Kopie des laufenden Prozesses
  - Unterscheidung ob Kind oder Vater anhand des Rückgabewertes (-1 im Fehlerfall)

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>   /*fuer pid_t benoetigt*/
int main() {
    pid_t retval;
    retval = fork();
    switch (retval) {
        case -1: perror("fork"); exit(EXIT_FAILURE);
        case  0: printf("I'm the child.\n"); break;
        default: printf("I'm the parent, my child's pid is %d.\n",retval);
    }
   return 0;
}
```

wait (2)

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

pid_t wait(int *status);
```

- legt sich schlafen wenn Kindprozesse existieren, aber keiner davon Zombie ist
- Rückgabewert
  - -1 im Fehlerfall, bzw. wenn keine Kinder (mehr) existieren
  - ansonsten pid *eines* Zombies
- Parameter
  - Zeiger auf eine Statusvariable, vordefinierte *Makros* zum Auslesen
- Fehler
  - ECHILD: Prozess hat keine Kinder
  - EINTR: nichtblockiertes Signal wurde gefangen (sollte bei uns nicht auftreten)

wait - Beispiel

- Kindprozess erzeugen und auf dessen Tod warten

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {
    int status;
    switch (fork()) {
        case -1: perror("fork"); exit(EXIT_FAILURE);
        case  0: printf("I'm the child.\n");
                 exit(EXIT_SUCCESS);
        default: if (wait(&status)==-1) {
                     perror("wait");
                     exit(EXIT_FAILURE);
                 } else
                     printf("Child returned with exit status: %d\n",
                         WEXITSTATUS(status));
    }
    return 0;
}
```

Präprozessormakro zum Auswerten der Statusvariable, mehr davon in `man 2 wait`

waitpid (2)

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

pid_t waitpid(pid_t pid, int *status, int options);
```

- Probleme von `wait()`
  - erlöst *irgendeinen* Zombie von möglicherweise mehreren
  - evtl. will man sich nicht schlafen legen
- Parameter:
  - `pid`: PID des gesuchten Zombies (-1, wenn alle Kinder gemeint sind)
  - `status`: Zeiger auf Statusvariable (wie bei wait)
  - `options`: 0 (keine) oder mehrere durch bitweises ODER verknüpft
- Rückgabewert: im Prinzip wie `wait()`, aber...
  - -1, wenn es keine Kinder gibt, die `pid` entsprechen
  - 0, wenn waitpid sich schlafen legen würde, aber die Option `WNOHANG` gegeben ist.

waitpid - Beispiel

- Funktion zum „Einsammeln“ toter Kindprozesse

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>

void collect_dead_children() {/*zombies “erloesen”*/
    pid_t res;
    while ((res=waitpid(-1, NULL, WNOHANG)) > 0);
    if (res == -1)
       if (errno != ECHILD) {
           perror(“waitpid”);
           exit(EXIT_FAILURE);
       }
}
```

da uns der Status nicht interessiert...

vordefiniertes *Präprozessorsymbol*, mehr davon in `man 2 waitpid`

Signal-Handler, sigaction(2)

```
int sigaction(SignalNr, *neueAkt, *alteAkt);
```

- Im eigenen Prozess Signale verarbeiten (SIGINT, SIGCHLD, etc.)
- Parameter
  - SignalNr = das Signal das behandelt werden soll
  - *neueAkt = Pointer auf struct sigaction mit den Informationen zum neuen Handler (Eingabe!)
  - *alteAkt = Pointer auf struct sigaction mit den Informationen zum vorherigen Handler (Ausgabe!), ignoriert bei Übergabe von NULL
- Benötigt <signal.h>
- Rückgabewerte:
  - -1: Fehler bei der Ausführung, Fehlercode in errno
  - 0: Erfolgreiche Ausführung

Signal-Handler

```
struct sigaction {
    void     (*sa_handler)(int);   /* Zeiger auf die Handler-Funktion */
    sigset_t sa_mask;              /* Ignorierte Signale während Behandlung */
    int      sa_flags;             /* Optionen */
    void     (*sa_restorer)(void); /* veraltet -> IGNORIEREN */
}
```

```
#include <signal.h>
...
/* die Handler-Funktion für unser Signal */
void handle_abbruch(int sig) {
  printf("SIGINT gefangen: %d!\n", SIGINT==sig);
  exit(2);
}
int main() {
  struct sigaction action;
  action.sa_handler = &handle_abbruch; /* die Adresse des Handlers angeben */
  action.sa_flags = 0;                 /* keine besonderen Flags */
  sigemptyset(&action.sa_mask);  /* keinerlei Signale ignorieren */
  if (sigaction(SIGINT, &action, NULL)) { /* SIGINT-Handler registrieren */
    perror("sigaction"); return -1;    /* Rückgabewert != 0 => Fehler! */
  }
  while(1);
  return 0;
}
```

Zusammenfassung

- `man 3 scanf`
- `man 3 strcmp`
- `man 3 errno`
- `man 3 perror`
- `man 3 exit`
- `man 3 sleep`
- `man 3 execlp`
- `man 2 fork`
- `man 2 waitpid`