– Lösung zur Praktikumsaufgabe 8 –

Thema: *Prozesserzeugung, -überlagerung und -beendigung mittels C*

1. Hauptproblem bei dieser Aufgabe ist die Tatsache, dass `random()` deterministische Folgen von Zufallszahlen generiert. Für wechselnde Folgen muss der Generator mittels `srandom()` mit *unterschiedlichen* Werten initialisiert werden. Wenn keine besonderen Anforderungen (hinsichtlich statistischer Parameter) an die Güte der Zufallszahlen gestellt werden, kann man sich behelfen, in dem man zur Initialisierung die Systemzeit heranzieht. Die Funktion `time()` ist z. B. geeignet, da sie die verstrichenen Sekunden seit dem 1. 1. 1970 zurückliefert.

   Listing 1: Lösung von Aufgabe 1)

   ```c
   #include <stdio.h>
   #include <stdlib.h>
   #include <time.h>

   int main(int argc, char* argv[])
   {
     unsigned int ret;

     srandom((unsigned int) time(NULL));
     ret = random();
     printf("Returning with %d.\n", ret);
     exit(ret);
   }
   ```

   Drei Aufrufe ergeben beispielsweise:

   ```
   $ ../prak07/aufgabe-07-01b.sh ./aufgabe-08-01
   Returning with 1996004282.
   ./aufgabe-08-01 returns 186
   $ ../prak07/aufgabe-07-01b.sh ./aufgabe-08-01
   Returning with 782648803.
   ./aufgabe-08-01 returns 227
   $ ../prak07/aufgabe-07-01b.sh ./aufgabe-08-01
   Returning with 180963834.
   ./aufgabe-08-01 returns 250
   ```

   Offenbar (und nicht weiter verwunderlich) kann man mittels `exit()` auch nicht mehr als 1 Byte an die rufende Umgebung übermitteln.

2. Die Nutzung von `atoi()` zur Umwandlung einer Zeichenkette in einen Integer ist problematisch, da diese Funktion keinen Umwandlungsfehler (z. B., wenn eine unsinnige Zeichenkette übergeben wurde) detektieren kann. Besser ist die Nutzung von `strtoul()`, wie in der Musterlösung demonstriert.

Listing 2: Lösung von Aufgabe 2)

```
/*
   demonstrates:
   - parameter passing to main
   - fork()
   - handling large numbers of child processes
 */

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

#define MAXSONS 100000

int main(int argc, char* argv[])
{
  unsigned int sons, c;
  pid_t pid[MAXSONS], tmppid;
  char *conv_error;

  if (argc != 2) {
    printf("Usage: %s <number of child procsses>\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  sons=strtoul(argv[1], &conv_error, 10);
  if (*conv_error != '\0') {
    printf("Could not convert argument %s to integer. Exit.\n",
        argv[1]);
    exit(EXIT_FAILURE);
  }

  if (sons > MAXSONS) {
    printf("Allowed maximum of %li child processes exceeded.\n", (
        long) MAXSONS);
    exit(EXIT_FAILURE);
  }

  printf("Creating %d sons.\n", sons);

  for (c=0; c<sons; c++) {
    tmppid = fork();
    if (tmppid == -1) {
      printf("Creating the %ith child failed. Exiting.\n", c+1);
      /* TODO: kill all children */
      exit(EXIT_FAILURE);
    }
    if (tmppid == 0) {  /* Child */
      sleep(10);
      exit(EXIT_SUCCESS);
    }
    else {  /* Father */
      pid[c] = tmppid;
    }
```

```
    }

    /* Father waits for children to terminate */
    for (c=0; c<sons; c++) {
      wait(NULL);
    }
    exit(EXIT_SUCCESS);
  }
```

Abhängig von der Speicherausstattung des Systems kommt es bei hinreichend großen Mengen zu erzeugender Prozesse zu Fehlern:

Ein 7 Jahre altes Notebook (P4, 1 GHz, 256 MiB RAM) war nur in der Lage, etwa 3980 Kindprozesse zu erzeugen. Bei dieser Anzahl stieg der Load des Prozessors kurzzeitig bis auf 130, obwohl alle Prozesse (bis auf den Vater) ja schlafen. Mein Arbeitsplatzrechner (Core 2 Duo, 2.1 GHz, 2 GiB RAM) ist ohne weiteres in der Lage, 32000 Prozesse und mehr zu erzeugen.

Mittels des Shell-Kommandos `ulimit` kann man die Anzahl bestimmter Ressourcen, die einem Nutzer maximal zur Verfügung stehen, limitieren. Alle Limits werden mit `ulimit -a` angezeigt. Im Labor Z136c können Sie beispielsweise maximal 16374 Nutzerprozesse erzeugen, was Sie mit Hilfe Ihres Programmes experimentell überprüfen können. Gleichzeitig reduziert der Einsatz dieser Limits die Gefährlichkeit von `fork()`-Bomben.

Wichtig ist, dass in dieser Lösung ein Vater mit vielen Söhnen koexistiert, die Generationstiefe ist zwei. In Aufgabe 5 werden wir sehen, dass auch eine andere „Verwandschaftstruktur“ möglich ist.

3.* Um Zombies zu sehen, muss man im einzelnen:

a) Kindprozesse erzeugen,

b) diese Kindprozesse sich wieder beenden lassen,

c) den Vater daran hindern, `wait()` aufzurufen,

d) die Prozessliste mit den Prozessstati anzeigen,

e) danach `wait()` für jedes Kind aufrufen.

Listing 3: Lösung von Aufgabe 3)

```
/*
   demonstrates:
     - a zombie process
 */

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

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

int main(int argc, char* argv[])
{
  pid_t spid;

  if ((spid = fork()) == -1) {
    printf("Creating a child failed. Exiting.\n");
    exit(EXIT_FAILURE);
  }
  if (spid == 0) {  /* Child */
    exit(EXIT_SUCCESS);
  }
  else {  /* Father */
    sleep(1); /* just to make sure the child is dead */
    printf("Father: child has exited.\n");
    printf("Father: executing 'ps a'\n");
    system("ps a");
    printf("Father: Press <Enter> to continue ...\n");
    getchar();
    wait(NULL); /* eliminate the zombie */
  }

  exit(EXIT_SUCCESS);
}
```

Die Ausgabe des Programmes für einen Testlauf ist:

```
robge@hadrian:~$ ./aufgabe-08-03
Father: child has exited.
Father: executing 'ps -a'
  PID TTY      STAT   TIME COMMAND
 3841 tty7     Ss+    5:10 /usr/bin/X -br :0 vt7 -auth /var/run/xauth/A:0-Q
 4771 pts/5    Ss     0:00 bash
 5016 pts/0    S      0:01 xpdf prak08-lsg.pdf
 9256 pts/2    S      0:02 emacs aufgabe-08-02.c
13447 pts/3    S+     0:00 ssh ilpro122.informatik.htw-dresden.de
13472 pts/2    S+     0:00 ./aufgabe-08-03
13473 pts/2    Z+     0:00 [aufgabe-08-03] <defunct>
13474 pts/2    R+     0:00 ps a
Father: Press <Enter> to continue ...
```

Den Zombie erkennen Sie am Zustand `Z+` sowie an der Angabe `<defunct>` nach dem Kommandonamen.

4.

Listing 4: Lösung von Aufgabe 4)

```c
/*
   demonstrates:
   - replacing process image with execl
 */
```

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

int main(int argc, char* argv[])
{
  int ret;

  if (argc != 2) {
    printf("Usage: %s <cmd>\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  printf("Starting %s ...\n", argv[1]);
  ret = execl(argv[1], argv[1], NULL);
  /* something went wrong */
  perror("execl");
  exit(EXIT_FAILURE);
}
```

Da `execl()` den Pfad nicht durchsucht, muss der genaue Pfad zur ausführbaren Datei übergeben werden, z. B. so:

```
robge@hadrian:~> ./aufgabe-08-04 pwd
Starting pwd ...
execl: No such file or directory
robge@hadrian:~> ./aufgabe-08-04 `which pwd`
Starting /bin/pwd ...
/home/local/robge/txt/job/htw/bs1/src/prak08
robge@hadrian:~>
```

Verwendet man anstatt `execl()` die Funktion `execlp()`, dann wird die Pfadvariable des Elternprozesses übernommen und entsprechend durchsucht. Ungünstig an dieser Lösung ist, dass man dem zu startenden Kommando nur sehr umständlich Parameter mitgeben kann (z. B. `ls -la`), da die `execl`-Varianten ihre Argumente als einzelne Parameter erhalten.

Die Lösung dieses Problems ist die Funktion `execvp()`, die den Namen des zu startenden Kommandos sowie dessen Parameter als Feld von Zeigern auf Zeichenketten (also ganz analog zu `main()`) übernimmt.

Listing 5: Übergabe von Parametern an das zu startende Programm mittels `execvp`

```c
/*
   demonstrates:
   - replacing process image with execvp which searches
     the PATH and takes arguments as array of pointers to char
 */

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

```c
#include <unistd.h>

int main(int argc, char* argv[])
{
  int ret;

  if (argc < 2) {
    printf("Usage: %s <cmd>\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  ret = execvp(argv[1], &argv[1]);
  /* something went wrong */
  perror("execl");
  exit(EXIT_FAILURE);
}
```

5. Anstatt eines Feldes (Aufgabe 2) für die Aufbewahrung der PID der erzeugten Söhne genügt nun eine einzige Variable, denn jeder Vater erzeugt nur einen einzigen Prozess.

Listing 6: Lösung von Aufgabe 5, iterativer Ansatz

```c
/*
  demonstrates
  - building a "chain" of processes (father->son->grandson->...)
  - iterative solution
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <wait.h>

int main(int argc, char* argv[])
{
  unsigned int c, generations;
  pid_t pid, tmppid;

  if (argc != 2) {
    printf("Usage: %s <generations>\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  generations = atoi(argv[1]);
  printf("%i generations to be started.\n", generations);

  for (c=0; c<generations; c++) {
    printf("Starting generation %i\n", c+1);
    pid = fork();
    if (pid == -1) {
      printf("Creating the %ith generation failed. Exiting.\n", c ↩
          +1);
      /* TODO: kill already started children */
      exit(EXIT_FAILURE);
    }
    if (pid != 0) {
```

```
        tmppid=wait(NULL);
        if (tmppid == -1) {
      printf("Waiting for son failed. Exiting.\n");
      exit(EXIT_FAILURE);
        }
        break; /* leave the loop */
      }
    }
    /* The grandest of sons will pass this first */
    sleep(10);
    printf("Son with PID %i ends.\n", getpid());
    exit(EXIT_SUCCESS);

  }
```

Die Ausgabe von `pstree` nach der Erzeugung von 4 Generationen ist (ausschnittsweise):

```
-+-galeon---2*[{galeon}]
         |-wmmon
         |-xterm---bash---less
         |-xterm---bash-+-emacs
         |              `-xpdf.bin
         |-xterm---bash
         `-xterm---bash-+-aufgabe-08-05-i---aufgabe-08-05-i---aufgabe-08-05-i---aufgabe-08-05-i---aufgabe-08-05-i
                        |-emacs
                        `-pstree
```

Genauso ist eine rekursive Lösung möglich:

Listing 7: Lösung von Aufgabe 5, rekursiver Ansatz

```
/*
   demonstrates:
   - building a "chain" of processes (father->son->grandson->...)
   - recursive solution
 */

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

int generations;

void rek_son(int currgen)
{
  pid_t ret;

  if (currgen == generations) {
    sleep(10);
    printf("Youngest Generation (%i) exits.\n", currgen);
    exit(EXIT_SUCCESS);
  }
  else {
    ret = fork();
    if (ret == -1) {
```

```
      /* TODO: Vaeter killen */
      exit(EXIT_FAILURE);
    }
    if (ret == 0) {
      rek_son(currgen+1);
    }
    wait(NULL);
    printf("Generation (%i) exits.\n", currgen);
    exit(EXIT_SUCCESS);
  }
}

int main(int argc, char* argv[])
{
  if (argc != 2) {
    printf("Usage: %s <cmd>\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  generations = atoi(argv[1]);
  printf("%i generations to be started.\n", generations);

  rek_son(0);

  /* must not be reached */
  printf("Dammit. I must not reach this statement.\n");
  exit(EXIT_FAILURE);
}
```