Systemnahe Software II
SS 2016

Andreas F. Borchert
Universität Ulm

11. April 2016

Inhalte:

- Prozesse unter UNIX
- Signale
- Interprozess-Kommunikation mit einem besonderen Schwerpunkt auf TCP/IP

Syllabus: Ziele

- Eingehendes Verständnis der POSIX-Schnittstellen und Abstraktionen für Prozesse, Signale, Kommunikation und Synchronisierung.
- Sichere Programmierung mit C in diesen Bereichen und das Erkennen von potentiellen Sicherheitslücken.
- Grundkenntnisse in TCP/IP und der Gestaltung von Internet-Protokollen.
- Eingehendes Verständnis der Muster zur Verarbeitung paralleler Sitzungen über TCP/IP.

Syllabus: Voraussetzungen

- Teilnahme an Systemnahe Software I. Dazu gehören insbesondere
  - Grundlagen in C einschließlich der dynamischen Speicherverwaltung,
  - Grundkenntnisse der POSIX-Schnittstellen im Bereich von Ein- und Ausgabe (*open*, *read*, *write* und die darüber liegende Schicht der *stdio*) und
  - Grundkenntnisse in der sicheren Programmierung in C (mitsamt der *stralloc*-Bibliothek von Dan Bernstein)
- Freude daran, etwas auch an einem Rechner auszuprobieren und genügend Ausdauer, dass nicht beim ersten Fehlversuch aufgegeben wird.

Warum ist sichere Programmierung wichtig?

- Wir beschäftigen uns im Rahmen der Vorlesung auch mit Netzwerkanwendungen und der Umsetzung von Netzwerkprotokollen.
- Kontakte über das Netzwerk sind normalerweise weltweit über das Internet möglich.
- Entsprechend tragen wir die Verantwortung dafür, keine offenen Scheunentore zu hinterlassen.
- Das bedeutet, dass wir bei jeder Code-Zeile und bei jedem Detail genau wissen müssen, was wir da tun, welche Gefahren lauern und wie wir diese abwehren.
- Die Folgen können sonst unabsehbar sein wie beim Heartbleed-Bug...

- Das Heartbeat-Protokoll wurde in Ergänzung zum SSL-Protokoll definiert: RFC 6520
- Das Protokoll soll zwei Probleme lösen:
  - Eine schnellere Alternative zu *SO_KEEPALIVE*
  - Ein alternativer Ansatz zur *Path MTU Discovery*, nachdem die ursprünglich dafür gedachten ICMP-Pakete allzu häufig von Firewalls weggefiltert werden
- Im Rahmen des Protokolls können Pings geschickt werden mit Daten (Payload) und einer zufällig gewählten Ergänzung. Solche Pings werden dann beantwortet, wobei der Payload zusammen mit anderen zufälligen Daten zurückgeschickt wird.
- Der Payload hat eine variable Länge. Deswegen findet sich im Header eines Heartbeat-Pakets ein Feld mit zwei Bytes, das den Umfang der Payload-Daten spezifiziert.

ssl/ssl3.h

```
typedef struct ssl3_record_st
        {
/*r */  int type;               /* type of record */
/*rw*/  unsigned int length;    /* How many bytes available */
/*r */  unsigned int off;       /* read/write offset into 'buf' */
/*rw*/  unsigned char *data;    /* pointer to the record data */
/*rw*/  unsigned char *input;   /* where the decode bytes are */
/*r */  unsigned char *comp;    /* only used with decompression - malloc()ed *
/*r */  unsigned long epoch;    /* epoch number, needed by DTLS1 */
/*r */  unsigned char seq_num[8]; /* sequence number, needed by DTLS1 */
        } SSL3_RECORD;
```

- Eine typische Datenstruktur für einen Kommunikationspuffer, entnommen aus openssl-1.0.1f
- *data* zeigt auf (die bereits entschlüsselten) Daten, die wir über das Netzwerk erhalten haben.
- *length* gibt an, wieviele Bytes in *data* zum Lesen zur Verfügung stehen.

ssl/d1-both.c

```
        unsigned char *p = &s->s3->rrec.data[0], *pl;
        /* ... */
        /* Read type and payload length first */
        hbtype = *p++;
        n2s(p, payload);
        pl = p;
```

- *s−>s3−>rrec* ist vom Typ *SSL3_RECORD* und repräsentiert das eingelesene Datenpaket, in dem sich ein Heartbeat-Paket befindet.
- *p* zeigt auf den Anfang des Datenbereichs des eingelesenen Pakets.
- Dort ist zu Beginn der Typ des Heartbeat-Pakets (ein Byte) und der Umfang des beigefügten Payloads (zwei Bytes).
- *n2s* konvertiert zwei Bytes vom Netzwerk in *network byte order* in eine ganze Zahl (*short*).
- *payload* kann hier ein beliebiger Wert zwischen 0 und 65535 sein, der vollkommen frei von der anderen Seite gewählt werden kann.

ssl/d1-both.c

```
        buffer = OPENSSL_malloc(1 + 2 + payload + padding);
        bp = buffer;

        /* Enter response type, length and copy payload */
        *bp++ = TLS1_HB_RESPONSE;
        s2n(payload, bp);
        memcpy(bp, pl, payload);
        bp += payload;
        /* Random padding */
        RAND_pseudo_bytes(bp, padding);

        r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT,
           buffer, 3 + payload + padding);
```

- Hier wird ein Antwort-Paket geschnürt (in Reaktion zu einem Ping), bei der die erhaltene Payload zurückzuschicken ist mitsamt einer Ergänzung aus zufälligen Daten (*padding*).
- Mit Hilfe von *memcpy* wird von *pl* (zeigt an den Anfang der erhaltenen Payload) nach *bp* kopiert.
- Kopiert werden *payload* Bytes. Es wird nirgends überprüft, ob noch *payload* Bytes hinter *pl* belegt sind...

Kann ein Lesen (und Weitergeben) des Speicherinhalts jenseits des Eingabe-Puffers ein Problem darstellen?

- Ja! Ziemlich anschaulich erklärt es Randall Munroe in xkcd: `http://www.xkcd.com/1354/`
- Bruce Schneier dazu:

  "Catastrophic" is the right word. On the scale of 1 to 10, this is an 11.

- Jede Woche gibt es zwei Vorlesungsstunden an jedem Montag von 16-18 Uhr in der Helmholtzstraße 18, Raum E.20.
- Die Übungen finden am Dienstag von 16-18 Uhr in der Helmholtzstraße 18, Raum 1.20 statt.
- Da die Vorlesung am Pfingsmontag ausfällt, wird sie am Dienstag, den 17. Mai, am Übungstermin nachgeholt.
- Webseite: `https://www.uni-ulm.de/mawi/mawi-numerik/lehre/sose16/vorlesung-systemnahe-software-ii.html`

Syllabus: Übungen

- Es gibt ein- und gelegentlich auch zweiwöchige Übungsblätter.
- Die Aufgaben werden in Gruppen von idealerweise drei Studenten gelöst und im Rahmen eines gemeinschaftlichen Testats dem zugehörigen Tutor vorgestellt.
- Die Organisation der Tutorenzuteilungen findet bei den Übungen am 12. April statt.
- Bitte melden Sie sich für die Vorlesung bei SLC an.
- Sie sollten, sofern noch nicht vorhanden, sich um einen Shell-Zugang bei uns bemühen.

Syllabus: Prüfung

- Voraussetzung hierfür sind mindestens 50% der Übungspunkte (Vorleistung).
- Eine Probeklausur wird gegen Semesterende zur Verfügung stehen, die in Bezug auf den Umfang, den Schwierigkeitsgrad und die Breite der Aufgabenstellungen mit der schriftlichen Prüfungen übereinstimmen wird.
- Die erste Prüfung findet am Donnerstag, den 21. Juli, statt.
- Für die zweite Prüfung ist Donnerstag, der 6. Oktober, vorgesehen.
- Die Prüfung ist offen, d.h. eine Anmeldung zur zweiten Prüfung ist auch ohne Teilnahme an der ersten möglich.

Syllabus: Skript

- Es gibt ein Skript, das auf der Webseite kapitelweise veröffentlicht wird.
- Parallel gibt es Präsentationen (wie diese), die ebenfalls als PDF zur Verfügung gestellt werden.
- Wenn Sie das Skript oder die Präsentationen ausdrucken möchten, nutzen Sie dazu bitte die entsprechenden Einrichtungen des KIZ. Im Prinzip können Sie dort beliebig viel drucken, wenn Sie genügend Punkte dafür erworben haben.
- Das Druck-Kontingent, das Sie bei uns kostenfrei erhalten (das ist ein Privileg und kein natürliches Recht), darf für die Übungen genutzt werden, jedoch nicht für das Ausdrucken von Skripten oder Präsentationen.

Syllabus: Sprechstunden

- Sie sind eingeladen, mich jederzeit per E-Mail zu kontaktieren:
  E-Mail: `andreas.borchert@uni-ulm.de`
- Meine reguläre Sprechzeit ist am Mittwoch 10:00-11:30 Uhr. Zu finden bin ich in der Helmholtzstraße 20, Zimmer 1.22.
- Zu anderen Zeiten können Sie auch gerne vorbeischauen, aber es ist dann nicht immer garantiert, dass ich Zeit habe. Gegebenenfalls lohnt sich vorher ein Telefonanruf: 23572.

Syllabus: Nicht verzweifeln! 16

- Immer wieder kann es mal vorkommen, dass es zu scheinbar unlösbaren Problemen bei einer Übungsaufgabe kommt.
- Geben Sie dann bitte nicht auf. Nutzen Sie unsere Hilfsangebote.
- Sie können (und sollen) dazu gerne Ihren Tutor oder Tutorin kontaktieren oder bei Bedarf gerne auch mich.
- Schicken Sie bitte in so einem Fall alle Quellen zu und vergessen Sie nicht, eine präzise Beschreibung des Problems mitzuliefern.
- Das kann auch am Wochenende funktionieren.

Syllabus: Feedback

- Feedback ist ausdrücklich erwünscht.
- Es besteht insbesondere auch immer die Möglichkeit, auf Punkte noch einmal einzugehen, die zunächst noch nicht klar geworden sind.
- Vertiefende Fragen und Anregungen sind auch willkommen.
- Ich spule hier nicht immer das gleiche Programm ab. Jede Vorlesung und jedes Semester verläuft anders und das hängt auch von Ihnen ab!

- Definition von Ritchie und Thompson, den Hauptentwicklern von UNIX:
  A process is the execution of an image.
- Zum *image* zählen der übersetzte Programmtext (Maschinencode und vorinitialisierte Daten) und der Ausführungskontext.

Ein Programm wird in einem bestimmten Kontext ausgeführt. Zu diesem Kontext gehören

- der Adressraum, in dem unter anderem der Programmtext (als Maschinencode) und die Daten untergebracht sind,
- pro Thread ein Satz Maschinenregister einschließlich der Stackverwaltung (Stack-Zeiger, Frame-Zeiger) und dem PC (*program counter*, verweist auf die nächste auszuführende Instruktion) und
- weitere Statusinformationen, die vom Betriebssystem verwaltet werden wie beispielsweise Informationen über geöffnete Dateien.

- Zu einem Prozess können mehrere Ausführungsfäden (*Threads*) gehören, die ebenfalls vom Betriebssystem verwaltet werden. Entsprechend gibt es nicht nur Status-Informationen auf Prozess-Ebene, sondern auch (in einem geringeren Umfang) auf Thread-Ebene.
- Alle wesentlichen Status-Informationen wie etwa User-ID, die Gruppenzugehörigkeiten, die geöffneten Dateien und der Adressraum sind allen Threads eines Prozesses gemein.
- Deswegen wird ein Prozess auch als Rechtegemeinschaft betrachtet.

printpid.c

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

int main() {
   printf("%d\n", (int) getpid());
}
```

- Jeder Prozess hat unter UNIX eine gleichbeibende identifizierende positive ganze Zahl, die mit *getpid*() abgefragt werden kann.
- Bei der Mehrheit der UNIX-Systeme liegt die Prozess-ID im Bereich von 1 bis 32767. Die Eindeutigkeit ist jedoch nur zu Lebzeiten garantiert. Sobald ein Prozess beendet wird, kann die gleiche Prozess-ID später einem neuen Prozess zugeordnet werden. Alle gängigen UNIX-Systeme vergeben Prozess-IDs reihum, wobei bereits vergebene Prozess-IDs übersprungen werden.

- Ein Prozess kann sich jederzeit mit *exit*() beenden und dabei einen Statuswert im Bereich von 0 bis 255 angeben.
- Die *exit*-Funktion kann in C-Programmen auch implizit aufgerufen werden: Ein **return** in der *main*-Funktion führt zu einem entsprechenden *exit* und wenn das Ende der *main*-Funktion erreicht wird, entspricht dies einem *exit*(0).
- Ein Exit-Wert von 0 deutet dabei eine erfolgreiche Terminierung an; andere Werte, insbesondere *EXIT_FAILURE*, werden als Misserfolg gewertet. Diese Konventionen orientieren sich zwar an UNIX, sind aber auch Bestandteil der ISO-Standards 9899-1999 und 9899-2011.

- Neue Prozesse können nur in Form eines Klon-Vorganges mit Hilfe des Systemaufrufs *fork*() erzeugt werden.
- Der Adressraum, die Maschinenregister und fast der gesamte Status des Betriebssystems für den erzeugenden Prozess werden dupliziert.
- Das bedeutet, dass beide Prozesse (der *fork*() aufrufende Prozess und der neu erzeugte Prozess) einen zu Beginn gleich aussehenden Adressraum vorfinden. Änderungen werden jedoch nur bei jeweils einem der beiden Prozesse wirksam.
- Um dies effizient umzusetzen und um einen hohen Kopieraufwand bei der *fork*-Operation zu vermeiden, kommt hier eine Verzögerungstechnik zum Zuge: *copy on write*.

- Einige Statusinformationen beim Betriebssystem betreffen beide Prozesse. So werden offene Dateiverbindungen vererbt und können nach dem Aufruf von *fork* gemeinsam genutzt werden.
- Dies bezieht sich aber nur auf Dateiverbindungen, die zum Zeitpunkt des *fork*-Aufrufs eröffnet waren und nicht auf Dateien, die später von einem der beiden Prozesse neu eröffnet werden.
- Einige Statusinformationen des Betriebssystems werden *nicht* weitergegeben. Dazu gehören beispielsweise Locks und anhängige Signale.
- Die Manualseite *fork(2)* zählt alle Statusinformationen auf, die weitergegeben werden.

clones.c

```
#include <stdio.h>
#include <unistd.h>
int main() {
   printf("I am feeling lonely!\n");
   fork();
   printf("Hey, I am cloned!\n");
}
```

- Ein neuer Prozess beginnt nicht irgendwo mit einem neuen Programmtext bei *main*().
- Stattdessen finden wir nach *fork*() zwei weitgehend übereinstimmende Kopien eines Prozesses vor, die alle den gleichen Programmtext hinter dem Aufruf von *fork*() fortsetzen.
- Deswegen wird in diesem Beispiel das zweite *printf* doppelt ausgeführt.

clones.c

```
#include <stdio.h>
#include <unistd.h>
int main() {
   printf("I am feeling lonely!\n");
   fork();
   printf("Hey, I am cloned!\n");
}
```

```
doolin$ clones | cat
I am feeling lonely!
Hey, I am cloned!
I am feeling lonely!
Hey, I am cloned!
doolin$
```

- Warum erhalten wir jetzt die Ausgabe „I am feeling lonely!“ nun doppelt?

clones.c

```
#include <stdio.h>
#include <unistd.h>
int main() {
   printf("I am feeling lonely!\n");
   fork();
   printf("Hey, I am cloned!\n");
}
```

- Erfolgt die Ausgabe direkt auf ein Terminal, wird zeilenweise gepuffert. In diesem Falle erfolgt die Ausgabe des ersten *printf*() noch vor dem Aufruf von *fork*().
- Falls jedoch voll gepuffert wird — dies ist bei der Ausgabe in eine Datei oder in eine Pipeline der Fall — dann erfolgt vor dem *fork*() noch keine Ausgabe. Stattdessen wird der Puffer von *stdout* durch *fork*() dupliziert, womit die doppelte Ausgabe der ersten Zeile provoziert wird.

clones2.c

```
#include <stdio.h>
#include <unistd.h>
int main() {
   printf("I am feeling lonely!\n");
   fork();
   fork();
   fork();
   printf("Hey, I am cloned!\n");
}
```

- Die doppelte Ausgabe eines ungeleerten Puffers lässt sich durch die rechtzeitige Leerung des Puffers mit Hilfe von *fflush*() vermeiden.

Wie können Ursprungsprozess und Klon getrennte Wege gehen? 29

clones3.c

```
#include <stdio.h>
#include <unistd.h>
int main() {
   pid_t parent;

   printf("I am feeling lonely!\n"); fflush(stdout);
   parent = getpid();
   fork();
   if (getpid() == parent) {
      printf("I am the parent process!\n");
   } else {
      printf("I am the child process!\n");
   }
}
```

- Damit der ursprüngliche Prozess und der mit *fork* erzeugte Klon getrennte Wege verfolgen können, müssen sie sich voneinander unterscheiden können. Ein naheliegendes Mittel ist hier die Prozess-ID, da der ursprüngliche Prozess seine behält und der Klon eine neue erhält.

Der Rückgabewert von *fork*

fork.c

```
#include <stdio.h>
#include <unistd.h>
int main() {
   pid_t pid;

   pid = fork();
   if (pid == -1) {
      perror("unable to fork"); exit(1);
   }
   if (pid == 0) {
      /* child process */
      printf("I am the child process: %d.\n", (int) getpid());
      exit(0);
   }
   /* parent process */
   printf("The pid of my child process is %d.\n", (int) pid);
}
```

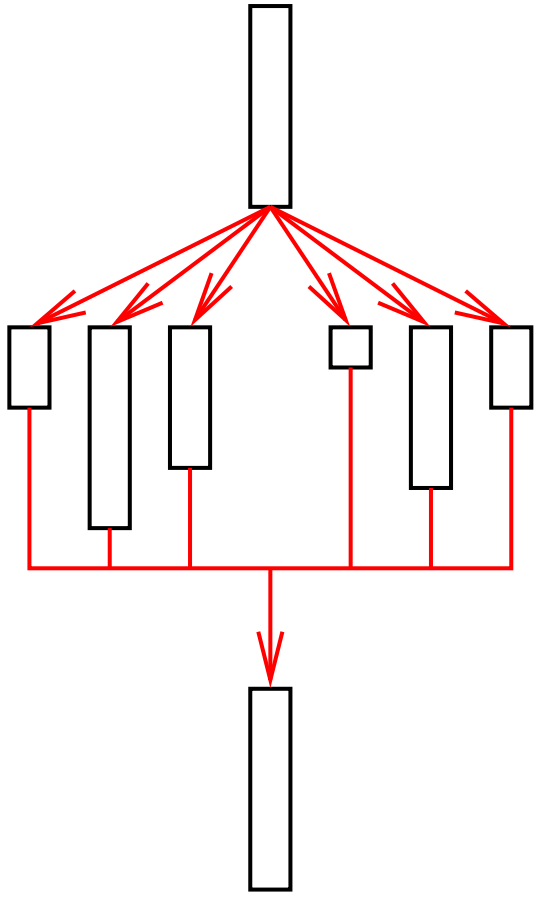
- *fork*() liefert -1 im Falle von Fehlern, 0 für den neu erzeugten Prozess und die Prozess-ID des neu erzeugten Prozesses beim alten Prozess.

fork.c

```
#include <stdio.h>
#include <unistd.h>
int main() {
   pid_t pid;

   pid = fork();
   if (pid == -1) {
      perror("unable to fork"); exit(1);
   }
   if (pid == 0) {
      /* child process */
      printf("I am the child process: %d.\n", (int) getpid());
      exit(0);
   }
   /* parent process */
   printf("The pid of my child process is %d.\n", (int) pid);
}
```

- Ein explizites *exit*() beim neu erzeugten Prozess verhindert, dass der Klon hinter der **if**-Anweisung den für den Erzeuger vorgesehenen Programmtext ausführt.



zu Beginn nur ein Prozeß

Erzeugen neuer Prozesse

Warten, bis alle neu erzeugten Prozesse beendet sind

- Es mag Fälle geben, bei denen neue Prozesse erzeugt und dann „vergessen“ werden. Im Normalfall jedoch stößt das weitere Schicksal des neuen Prozesses auf Interesse und insbesondere ist es nicht unüblich, dass der erzeugende Prozess auf das Ende der von ihm erzeugten Prozesse warten möchte.
- Dies macht insbesondere dann Sinn, wenn mehrere Prozesse erzeugt werden, die parallel Teilprobleme des Gesamtproblems lösen. Dann wartet der erzeugende Prozess nach Erzeugung all der Unterprozesse, bis sie alle ihre Teilaufgaben erledigt haben. Dieses Muster wird „Fork and Join“ genannt.

Synchronisierung bei der Prozessterminierung

forkandwait.c

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

int main() {
   pid_t child, pid; int stat;

   child = fork();
   if (child == -1) {
      perror("unable to fork"); exit(1);
   }
   if (child == 0) {
      /* child process */
      srand(getpid());
      exit(rand());
   }

   /* parent process */
   pid = wait(&stat);
   if (pid == child) {
      if (WIFEXITED(stat)) {
         printf("exit code of child = %d\n", WEXITSTATUS(stat));
      } else {
         printf("child terminated abnormally\n");
      }
   } else {
      perror("wait");
   }
}
```

forkandwait.c

```
if (child == 0) {
   /* child process */
   srand(getpid());
   exit(rand());
}
```

- Der neu erzeugte Prozess initialisiert den Pseudo-Zufallszahlengenerator mit *srand* und holt sich dann mit *rand* eine pseudo-zufällige Zahl ab.
- Da der Exit-Wert nur 8 Bit und entsprechend nur die Werte von 0 bis 255 umfasst, werden die höherwertigen Bits der Pseudo-Zufallszahl implizit weggeblendet.

forkandwait.c

```
/* parent process */
pid = wait(&stat);
if (pid == child) {
   if (WIFEXITED(stat)) {
      printf("exit code of child = %d\n", WEXITSTATUS(stat));
   } else {
      printf("child terminated abnormally\n");
   }
} else {
   perror("wait");
}
```

- Die Funktion *wait* wartet auf die Terminierung eines beliebigen Unterprozesses, der noch *nicht* von *wait* zurückgeliefert wurde.
- Falls es einen solchen Prozess nicht mehr gibt, wird -1 zurückgeliefert.
- Ansonsten liefert *wait* die Prozess-ID des terminierten Prozesses und innerhalb von *stat* den zugehörigen Status.

Der in *stat* abgelegte Status des Unterprozesses besteht aus mehreren Komponenten, die angeben,

- wie ein Prozess sein Leben beendete (durch *exit*() oder durch ein Signal (bei einem Crash oder Verwendung von *kill*()) oder ob der Prozess nur gestoppt wurde,
- welcher Wert bei *exit*() angegeben wurde, falls *exit*() benutzt wurde und
- welches Signal das Leben des Prozesses terminierte bzw. stoppte, falls der Prozess nicht mit *exit*() endete.

- Was geschieht mit dem Rückgabewert bei *exit*() und dem sonstigen Endstatus eines Prozesses, wenn der übergeordnete Prozess nicht zeitig *wait*() aufruft?
- Das UNIX-System lässt solche toten Prozesse noch in seiner Verwaltung weiterleben, so dass der Endstatus noch bewahrt wird, aber die nicht mehr benötigten Ressourcen freigegeben werden.
- Prozesse, die sich in diesem Stadium befinden, werden als Zombies bezeichnet.

genzombie.c

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

int main() {
   pid_t child = fork();
   if (child == -1) {
      perror("fork"); exit(1);
   }
   if (child == 0) exit(0);
   printf("%d\n", child);
   sleep(60);
}
```

- Der neu erzeugte Prozess verabschiedet sich hier sofort mit *exit*(), während der übergeordnete Prozess mit Hilfe eines *sleep*()-Aufrufes sich für 60 Sekunden zur Ruhe legt.
- Während dieser Zeit verbleibt der Unterprozeß im Zombie-Status.

```
doolin$ genzombie&
[1]     24489
doolin$ 24490

doolin$ ps -ylp 24489,24490
 S   UID   PID  PPID  C PRI NI   RSS     SZ    WCHAN TTY      TIME CMD
 S   120 24489 23591  0  64 28   616    936        ? pts/31   0:00 genzombi
 Z   120 24490 24489  0   0                            0:00 <defunct>
doolin$
```

- In der ersten Spalte gibt *ps* bei dieser Aufrufvariante den Status eines Prozesses an.
- „*Z*“ steht dabei für Zombie, „*S*“ für schlafend.
- Weitere Varianten sind „*O*“ für gerade arbeitend, „*R*“ für arbeitsbereit und „*T*“ für gestoppt.

orphan.c

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

int main() {
   pid_t child;
   child = fork();
   if (child == -1) {
      perror("fork"); exit(1);
   }
   if (child == 0) {
      printf("Hi, my parent is %d\n", (int) getppid());
      sleep(5);
      printf("My parent is now %d\n", (int) getppid());
      exit(0);
   }
   sleep(3);
   exit(0);
}
```

- Wenn sich der übergeordnete Prozess verabschiedet, dann wird ihm der Prozess mit der Prozess-ID 1 als neuer übergeordneter Prozess zugewiesen.

- Der Prozess mit der Prozess-ID 1 spielt eine besondere Rolle unter UNIX. Es ist der erste Prozess, der vom Betriebssystem selbst erzeugt wird. Er führt den unter */etc/init* oder /sbin/init zu findenden Programmtext aus.
- Dieser Prozess startet weitere Prozesse anhand einer Konfigurationsdatei (bei uns unter */etc/inittab*) und ruft ansonsten *wait*() auf, um den Status der von ihm selbst erzeugten Prozesse oder den von Waisenkindern entgegenzunehmen.
- Auf diese Weise wird dann auch der Zombie-Status eines Prozesses beendet, wenn es zum Waisenkind wird.

Mit *fork*() ist es möglich, neue Prozesse zu erzeugen. Allerdings teilen die neuen Prozesse sich den Programmtext mit ihrem Erzeuger. Wie ist nun der Wechsel zu einem anderen Programmtext möglich? Die Lösung dafür ist der Systemaufruf *exec*(), der

- den gesamten virtuellen Adressraum des aufrufenden Prozesses auflöst,
- an seiner Stelle einen neuen einrichtet mit einem angegebenen Programmtext,
- sämtliche Maschinenregister für den Prozess neu initialisiert und
- Statusinformationen des Betriebssystems weitgehend unverändert belässt

Wechsel zum Programm *date*

datum.c

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

int main() {
   execl(
      "/usr/bin/date",  /* path of the program */
      "/usr/bin/date",  /* name of the program, i.e. argv[0] */
      "+%d.%m.%Y",      /* first argument, i.e. argv[1] */
      0                 /* terminate list of arguments */
   );
   /* not reached except if execl failed */
   perror("/usr/bin/date"); exit(1);
}
```

- Dieses Programm ersetzt seinen eigenen Programmtext durch den von *date*.

datum.c

```
execl(
   "/usr/bin/date",  /* path of the program */
   "/usr/bin/date",  /* name of the program, i.e. argv[0] */
   "+%d.%m.%Y",      /* first argument, i.e. argv[1] */
   0                 /* terminate list of arguments */
);
```

- *execl* erlaubt die Angabe beliebig vieler Kommandozeilenargumente in der Form einzelner Funktionsparameter. Mit einem Nullzeiger wird die Liste der Parameter beendet.
- Dabei ist zu beachten, dass der Pfadname des auszuführenden Programms und der später unter *argv*[0] zu findende Kommandoname getrennt angegeben werden. Normalerweise sind beide gleich, es gibt aber auch Ausnahmen.

datum.c

```
execl(
   "/usr/bin/date",  /* path of the program */
   "/usr/bin/date",  /* name of the program, i.e. argv[0] */
   "+%d.%m.%Y",      /* first argument, i.e. argv[1] */
   0                 /* terminate list of arguments */
);
/* not reached except if execl failed */
perror("/usr/bin/date"); exit(1);
```

- Normalerweise geht es im Programmtext nach einem Aufruf von *execl*() nicht weiter, weil im Erfolgsfalle das Programm ausgetauscht wurde. Nur bei einem Fehler (weil z.B. das *date*-Kommando nicht gefunden wurde) wird das Programm hinter dem Aufruf von *execl*() fortgesetzt.

- Auf den ersten Blick erscheinen diese vier Systemaufrufe seltsam. Warum ist eine Kombination aus *fork*() und *exec*() notwendig, um einen neuen Prozess mit einem neuen Programmtext in Gang zu setzen?
- Wäre es nicht besser und einfacher, nur einen einzigen Systemaufruf dafür zu haben?
- Die Frage verschärft sich, wenn berücksichtigt wird, dass in der Zeit der frühen UNIX-Implementierungen die Technik des „*copy on write*“ noch nicht zur Verfügung stand. Stattdessen war es bei *fork*() notwendig, den gesamten Speicher zu kopieren.
- Bei BSD wurde deswegen zeitweise *fork1*() eingeführt, das diesen Kopiervorgang unterdrückte, um die typische Kombination von *fork*() und *exec*() nicht zu teuer werden zu lassen.

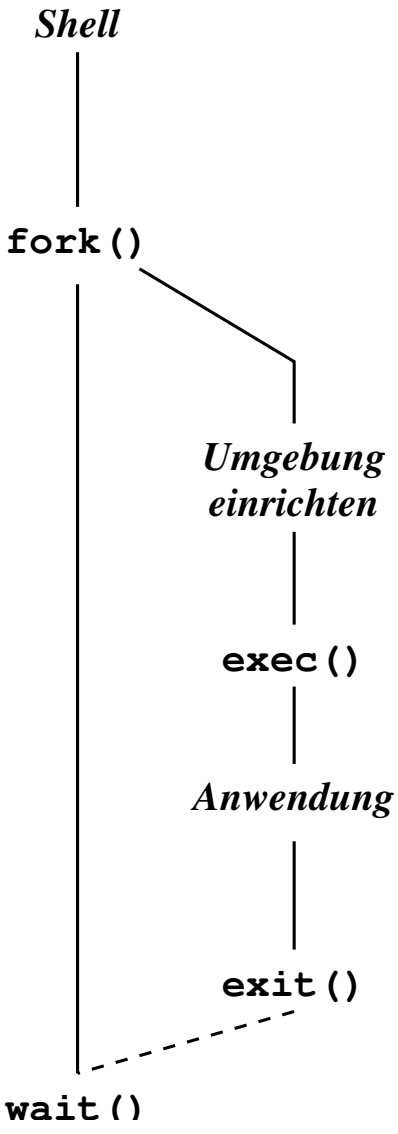
Alternativen zu *fork* und *exec*


```
//IS198CPY JOB (IS198T30500),'COPY JOB',CLASS=L,MSGCLASS=X
//COPY01   EXEC PGM=IEBGENER
//SYSPRINT DD  SYSOUT=*
//SYSUT1   DD  DSN=OLDFILE,DISP=SHR
//SYSUT2   DD  DSN=NEWFILE,
//           DISP=(NEW,CATLG,DELETE),
//           SPACE=(CYL,(40,5),RLSE),
//           DCB=(LRECL=115,BLKSIZE=1150)
//SYSIN    DD  DUMMY
```

- UNIX ist keinesfalls das erste Betriebssystem, das Prozesse unterstützte. Die älteren Systeme boten in der Tat die Kombination aus *fork*() und *exec*() in einem Systemaufruf an.
- Das Beispiel zeigt ein Kopierkommando in der JCL (Job Command Language) aus der IBM-Mainframe-Welt (von der Wikipedia übernommen). Hieran zeigt sich, dass dies die Kommandosprache deutlich verkompliziert. Der Haken liegt darin, dass Prozesse häufig eine Umgebung erwarten, die mehr umfaßt als eine Kommandozeile. Wichtiger Bestandteil der Umgebung sind bereits im Vorfeld eingerichtete Ein- und Ausgabeverbindungen und die Zuteilung von Ressourcen.

- So sieht die traditionelle Erzeugung eines Prozesses aus:
  - Erzeuge einen neuen Prozess mit einem gegebenen Programmtext mit einem Systemaufruf, der *fork*() und *exec*() kombiniert.
  - Einrichtung der Umgebung für den neuen Prozess.
  - Start des neuen Prozesses.
- Entsprechend ist es notwendig, alle wichtigen Systemaufrufe für die Einrichtung einer Umgebung einschließlich dem Öffnen von Ein- und Ausgabeverbindungen in zwei Varianten zu unterstützen: Die eine Variante bezieht sich auf den eigenen Prozess, die andere für einen untergeordneten Prozess, der noch nicht gestartet wurde.

Prozesserzeugung bei der UNIX-Shell



- Die Trennung in *fork*() und *exec*() erlaubt die Konfiguration der Umgebung des aufzurufenden Programms innerhalb der Shell mit ganz normalen Systemaufrufen.

```
clonard$ tinysh
% date
Mon Apr 28 13:10:54 MEST 2008
% date >out
% cat out
Mon Apr 28 13:11:06 MEST 2008
% awk {print$4} <out
13:11:06
% clonard$
```

- Die kleine Shell *tinysh* erlaubt
  - den Aufruf von Kommandos mit beliebig vielen Parametern, die durch Leerzeichen getrennt werden,
  - die Umlenkung der Standard-Ein- und Ausgabe, wobei auch das Anhängen unterstützt wird und
  - die Auswertung des *wait*-Systemaufrufs.
- Die Konfiguration des aufzurufenden Programms erfolgt hier zwischen *fork* und *exec*.

tinysh.c

```
int main() {
   stralloc line = {0};
   while (printf("%% "), readline(stdin, &line)) {
      strlist tokens = {0};
      stralloc_0(&line); /* required by tokenizer() */
      if (!tokenizer(&line, &tokens)) break;
      if (tokens.len == 0) continue;
      pid_t child = fork();
      if (child == -1) {
         perror("fork"); continue;
      }
      if (child == 0) {
         // setup child and argv.list ...
         execvp(cmdname, argv.list);
         perror(cmdname);
         exit(255);
      }

      /* wait for termination of child */
      // ...
   }
} // main
```

sareadline.c

```
bool readline(FILE* fp, stralloc* sa) {
   sa->len = 0;
   for(;;) {
      if (!stralloc_readyplus(sa, 1)) return false;
      int ch = getc(fp);
      if (ch == EOF) return sa->len > 0;
      if (ch == '\n') break;
      sa->s[sa->len++] = ch;
   }
   return true;
} // readline
```

- Diese *readline*-Funktion erlaubt das Einlesen beliebig langer Zeilen.
- Mit *stralloc_readyplus* wird jeweils Platz für mindestens ein weiteres Zeichen geschaffen.
- Die resultierende Zeichenkette ist *nicht* durch ein Nullbyte terminiert.

Erzeugung der Liste mit Kommandozeilenparametern 54

- Die Funktion *execl* ist für die *tinysh* ungeeignet, da die Zahl der Kommandozeilenparameter nicht feststeht. Diese soll auch nicht durch das Programm künstlich begrenzt werden.
- Alternativ zu *execl* gibt es *execv*, das einen Zeiger auf eine Liste mit Zeigern auf Zeichenketten erwartet, die am Ende mit einem Null-Zeiger abzuschliessen ist.
- Die in der *tinysh* verwendete Funktion *execvp* (mit zusätzlichem p) sucht im Gegensatz zu *execv* nach dem Programm in allen Verzeichnissen, die die Umgebungsvariable *PATH* aufzählt.

Erzeugung einer Liste mit Zeigern auf Zeichenketten 55

strlist.h

```
#ifndef STRLIST_H
#define STRLIST_H

#include <stddef.h>
#include <stdbool.h>

typedef struct strlist {
   char** list;
   size_t len; /* # of strings in list */
   size_t allocated; /* allocated length for list */
} strlist;

/* assure that there is at least room for len list entries */
bool strlist_ready(strlist* list, size_t len);

/* assure that there is room for len additional list entries */
bool strlist_readyplus(strlist* list, size_t len);

/* truncate the list to zero length */
void strlist_clear(strlist* list);

/* append the string pointer to the list */
bool strlist_push(strlist* list, char* string);
#define strlist_push0(list) strlist_push((list), 0)

/* free the strlist data structure but not the strings */
void strlist_free(strlist* list);

#endif
```

Erzeugung einer Liste mit Zeigern auf Zeichenketten 56

strlist.h

```
typedef struct strlist {
   char** list;
   size_t len; /* # of strings in list */
   size_t allocated; /* allocated length for list */
} strlist;

bool strlist_ready(strlist* list, size_t len);
bool strlist_readyplus(strlist* list, size_t len);
void strlist_clear(strlist* list);
bool strlist_push(strlist* list, char* string);
void strlist_free(strlist* list);
```

- Die *strlist*-Bibliothek folgt weitgehend dem Vorbild der *stralloc*-Bibliothek.

Erzeugung einer Liste mit Zeigern auf Zeichenketten 57

strlist.c

```
/* assure that there is at least room for len list entries */
bool strlist_ready(strlist* list, size_t len) {
   if (list->allocated < len) {
      size_t wanted = len + (len>>3) + 8;
      char** newlist = (char**) realloc(list->list,
         sizeof(char*) * wanted);
      if (newlist == 0) return false;
      list->list = newlist;
      list->allocated = wanted;
   }
   return true;
}

/* assure that there is room for len additional list entries */
bool strlist_readyplus(strlist* list, size_t len) {
   return strlist_ready(list, list->len + len);
}
```

Erzeugung einer Liste mit Zeigern auf Zeichenketten 58

strlist.c

```
void strlist_clear(strlist* list) {
   list->len = 0;
}

/* append the string pointer to the list */
bool strlist_push(strlist* list, char* string) {
   if (!strlist_ready(list, list->len + 1)) return false;
   list->list[list->len++] = string;
   return true;
}

/* free the strlist data structure but not the strings */
void strlist_free(strlist* list) {
   free(list->list); list->list = 0;
   list->allocated = 0;
   list->len = 0;
}
```

tokenizer.h

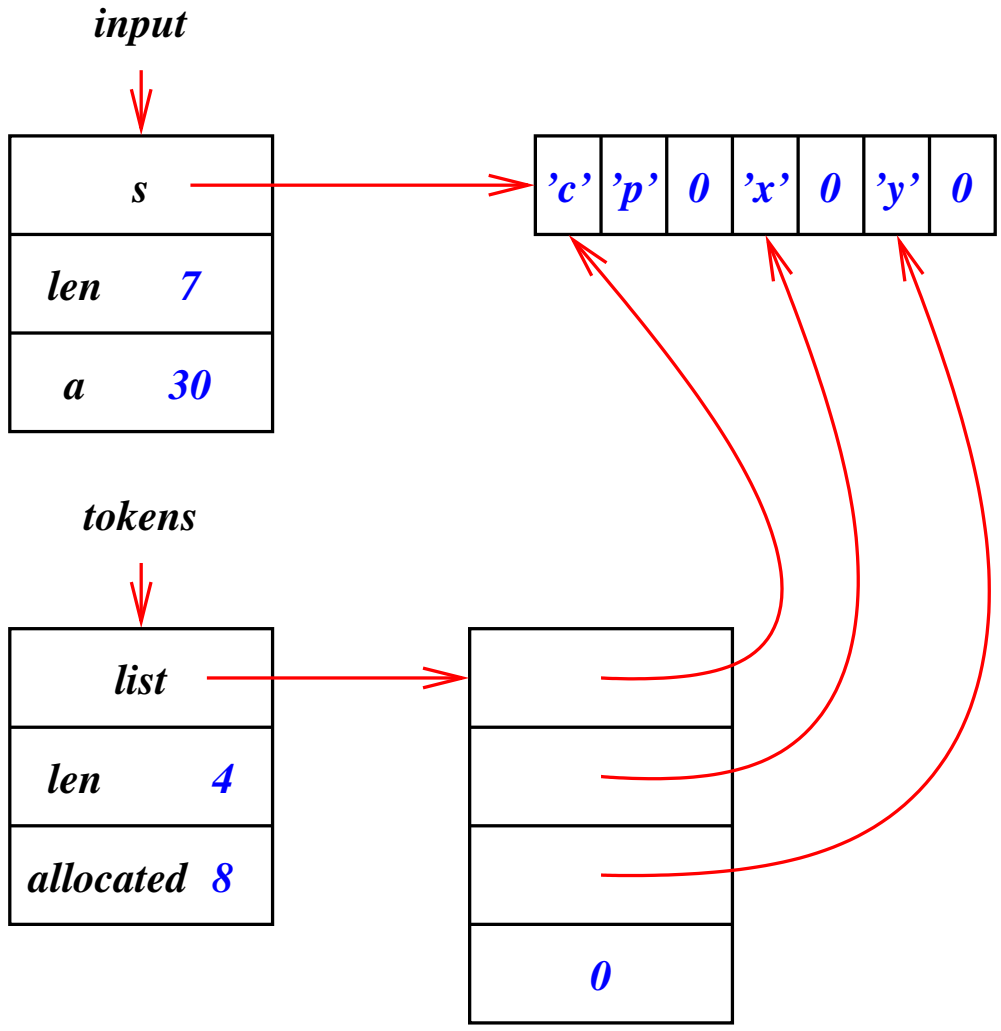
```
#ifndef TOKENIZER_H

#define TOKENIZER_H
#include <stralloc.h>
#include "strlist.h"
bool tokenizer(stralloc* input, strlist* tokens);

#endif
```

- Die Funktion *tokenizer* zerlegt die Eingabezeile in *input* in einzelne (durch Leerzeichen getrennte) Wörter und fügt diese in die Liste *tokens*.
- Wesentlich ist hier, dass die einzelnen Zeichenketten nicht dupliziert werden, sondern innerhalb der Eingabezeile verbleiben. Zu diesem Zweck werden Leerzeichen durch Nullbytes ersetzt.

Zerlegung einer Eingabezeile



- Das Diagramm zeigt die resultierende Datenstruktur des Wortzerlegers am Beispiel „cp x y“.

Zerlegung einer Eingabezeile

tokenizer.c

```
/*
 * Simple tokenizer: Take a 0-terminated stralloc object and return a
 * list of pointers in tokens that point to the individual tokens.
 * Whitespace is taken as token-separator and all whitespaces within
 * the input are replaced by null bytes.
 * afb 4/2003
 */

#include <ctype.h>
#include <stdlib.h>
#include <stralloc.h>
#include "strlist.h"
#include "tokenizer.h"

bool tokenizer(stralloc* input, strlist* tokens) {
   char* cp;
   int white = 1;

   strlist_clear(tokens);
   for (cp = input->s; *cp && cp < input->s + input->len; ++cp) {
      if (isspace((int) *cp)) {
         *cp = '\0'; white = 1; continue;
      }
      if (!white) continue;
      white = 0;
      if (!strlist_push(tokens, cp)) return false;
   }
   return true;
}
```

Zerlegung einer Eingabezeile

tinysh.c

```
while (printf("%% "), readline(stdin, &line)) {
   strlist tokens = {0};
   stralloc_0(&line); /* required by tokenizer() */
   if (!tokenizer(&line, &tokens)) break;
   if (tokens.len == 0) continue;
   // ...
}
```

- Da der Wortzerleger nullbyte-terminierte Zeichenketten liefert, muss mit *stralloc_0* noch ein Nullbyte angehängt werden.
- Falls keine Wörter zu finden sind, wird sofort die nächste Zeile eingelesen.
- Die Erzeugung der Kommandozeilenparameterliste wird dem neu zu erzeugenden Prozess überlassen.

Konfiguration des neuen Prozesses

tinysh.c

```
if (child == 0) {
   strlist argv = {0}; /* list of arguments */
   char* cmdname = 0; /* first argument */
   char* path; /* of output files */
   int oflags;

   for (int i = 0; i < tokens.len; ++i) {
      switch (tokens.list[i][0]) {
         case '<':
            fassign(0, &tokens.list[i][1], O_RDONLY, 0);
            break;
         case '>':
            path = &tokens.list[i][1];
            oflags = O_WRONLY|O_CREAT;
            if (*path == '>') {
               ++path; oflags |= O_APPEND;
            } else {
               oflags |= O_TRUNC;
            }
            fassign(1, path, oflags, 0666);
            break;
         default:
            strlist_push(&argv, tokens.list[i]);
            if (cmdname == 0) cmdname = tokens.list[i];
      }
   }
   if (cmdname == 0) exit(0);
   strlist_push0(&argv);
   execvp(cmdname, argv.list);
   perror(cmdname); exit(255);
}
```

tinysh.c

```
/*
 * assign an opened file with the given flags and mode to fd
 */
void fassign(int fd, char* path, int oflags, mode_t mode) {
   int newfd = open(path, oflags, mode);
   if (newfd < 0) {
      perror(path); exit(255);
   }
   if (dup2(newfd, fd) < 0) {
      perror("dup2"); exit(255);
   }
   close(newfd);
} // fassign
```

- Mit dem Systemaufruf *dup2* lässt sich ein Dateideskriptor auf einen gegebenen anderen Deskriptor duplizieren, die dann beide auf den gleichen Eintrag in der *Open File Table* verweisen.
- So lassen sich neu eröffnete Datei-Verbindungen mit vorgegebenen Dateideskriptoren wie etwa 0 (stdin) oder 1 (stdout) verknüpfen.