

Systemnahe Software II

SS 2019

Andreas F. Borchert
Universität Ulm

23. April 2019

Inhalte:

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

- 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.

- 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.

```
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 14–16 Uhr in der Helmholtzstraße 22, Raum E.04.
- Die Übungen finden am Dienstag von 14–16 Uhr in der Helmholtzstraße 18, Raum E.44, statt.
- Die Vorlesung fällt am 10. Juni 2019 (Pfingstmontag) aus und wird dann am 11. Juni 2019 zum Zeitpunkt der Übungen in der Helmholtzstraße 22, Raum E.03, nachgeholt.
- Webseite:
<https://www.uni-ulm.de/mawi/mawi-numerik/lehrenumerik/sommersemester-2019/vorlesung-systemnahe-software-ii/>

- Es gibt ein- und gelegentlich auch zweiwöchige Übungsblätter.
- Wir haben weder Tutoren noch Korrekteure. Entsprechend können Ihre Lösungen nicht korrigiert werden.
- Es besteht aber die Möglichkeit, Lösungen elektronisch einzureichen. Das ermöglicht ein allgemeines Feedback in den Übungen.
- Gelegentlich werden elektronische Einreichungen auch mit Testsuites verknüpft sein.
- Bitte melden Sie sich für die Vorlesung bei SLC an.
- Sie sollten, sofern noch nicht vorhanden, sich um einen Shell-Zugang bei uns bemühen.

- Eine Vorleistung für die Teilnahme zur Prüfung ist nicht mehr erforderlich.
- Die Prüfungen erfolgen mündlich zu individuell vereinbarten Terminen.

- 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.

- 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 1.23.
- Zu anderen Zeiten können Sie auch gerne vorbeischaun, aber es ist dann nicht immer garantiert, dass ich Zeit habe. Gegebenenfalls lohnt sich vorher ein Telefonanruf: 23572.

- Immer wieder kann es mal vorkommen, dass es zu scheinbar unlösbaren Problemen bei einer Übungsaufgabe kommt.
- Geben Sie dann bitte nicht auf! Kontaktieren Sie mich bitte stattdessen.
- 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.

- 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), *errno* und Signalmaske 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 (wie beispielsweise die Signalmaske).
- 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 Rechtsgemeinschaft betrachtet.

printpid.c

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

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

- Jeder Prozess hat unter UNIX eine gleichbleibende 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.

```
heim$ cat /proc/sys/kernel/pid_max
32768
heim$
```

- Bei Linux-Systemen ist der Wert abrufbar und kann bei 64-Bit-Systemen auf bis zu 2^{22} erhöht werden (*PID_MAX_LIMIT*).
- Andere Systeme wie Solaris erlauben hier ebenso bei Bedarf höhere Werte.
- Da die Prozess-IDs sequentiell vergeben werden, sind sie nicht wirklich geeignet für
 - ▶ die Generierung sicherheitsrelevanter Seed-Werte,
 - ▶ die Benennung temporärer Dateien oder
 - ▶ die Erzeugung von Session-IDs.

- 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.
- Der Standard IEEE Std 1003.1 zählt in der Manualseite zu *fork(2)* 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"); fflush(stdout);
    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.

- Grundsätzlich ist *fork* nicht unproblematisch, weil der gesamte Zustand der Datenstrukturen dupliziert wird.
- Dies verschärft sich, wenn Datenstrukturen mit Ressourcen (wie etwa Dateideskriptoren oder Locks) verknüpft sind.
- Besonders problematisch ist dies in Verbindung mit Threads, da hier *fork* ohne besondere Vorkehrungen völlig asynchron zu den anderen Threads ausgeführt wird.
- Eine saubere Lösung existiert hier nicht. Wenn Threads mit *fork* kombiniert werden, sollte auf *fork* möglichst unmittelbar ein *exec* folgen und dazwischen müsste alles *async-signal-safe* sein (mehr dazu später).
- Im POSIX-Standard gibt es *pthread_atfork* aus der Thread-Bibliothek, das jedoch entgegen der ursprünglichen Intention nicht geeignet ist für Thread-Ressourcen (wie etwa ein Mutex), aber für andere Datenstrukturen (wie etwa die der *stdio*) verwendet werden kann.

clones3.c

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

void flush_all() {
    fflush(0); /* flush all streams */
}

int main() {
    /* invoke flush_all implicitly before fork() is executed */
    pthread_atfork(flush_all, 0, 0);

    printf("I am feeling lonely!\n");
    fork();
    printf("Hey, I am cloned!\n");
}
```

- `pthread_atfork` hat drei Funktionsparameter des Typs **void (*)(void)**.
- Die erste Funktion wird vor `fork` implizit aufgerufen, die anderen beim Elternprozess bzw. Kindprozess nach `fork`.
- Ein Nullzeiger kann angegeben werden, wenn eine der Funktionen entfällt.

Wie können Ursprungsprozess und Klon getrennte Wege gehen?

32

clones4.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.

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);
}
```

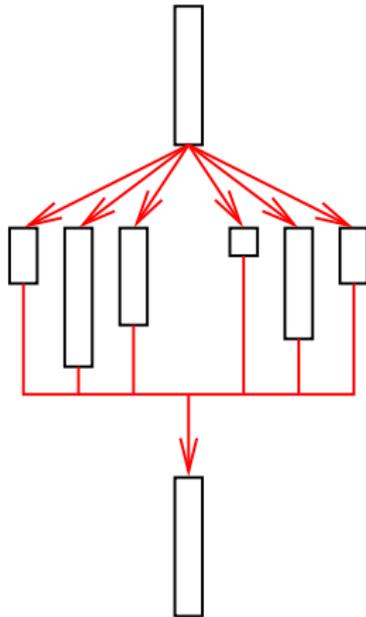
- *fork()* liefert -1 im Falle von Fehlern, 0 für den neu erzeugten Prozess und die Prozess-ID des neu erzeugten Prozesses beim alten Prozess.

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.

```
#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.

- Prozesse können auch zu Prozessgruppen zusammengefasst werden.
- Für die ID einer Prozessgruppe kann die eines Prozesses aus der Gruppe gewählt werden.
- Dies ist bei Fork-and-Join sinnvoll, da dann gezielt auf Prozesse einer Gruppe gewartet werden kann.

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

int main() {
    pid_t group = 0;
    for (int i = 0; i < 10; ++i) {
        pid_t pid = fork();
        if (pid < 0) {
            perror("fork"); exit(1);
        }
        if (pid == 0) {
            exit(getpid() % 255);
        }
        setpgid(pid, group);
        if (group == 0) {
            group = pid;
        }
    }
    int wstat;
    while (waitpid(-group, &wstat, 0) > 0) {
        if (WIFEXITED(wstat)) {
            printf(" %d", WEXITSTATUS(wstat));
        } else {
            printf(" X");
        }
    }
    printf("\n");
}
```

forkandjoin.c

```
setpgid(pid, group);  
if (group == 0) {  
    group = pid;  
}
```

- *pid* ist die Prozess-ID eines frisch erzeugten Kindprozesses, *group* ist zu Beginn 0.
- Mit *setpgid* kann die Prozess-ID von *pid* gesetzt werden. *pid* muss dabei der eigene Prozess und ein Kindprozess sein. Wenn *group* den Wert 0 hat, wird implizit als Gruppen-ID der Wert von *pid* gewählt.

```
int wstat;
while (waitpid(-group, &wstat, 0) > 0) {
    /* ... */
}
```

- *waitpid* bietet eine verallgemeinerte Schnittstelle im Vergleich zu *wait*.
- Der erste Parameter spezifiziert, worauf gewartet wird:
 - ▶ *pid* > 0: es wird auf den genannten Prozess gewartet
 - ▶ *pid* == 0: es wird auf einen Prozess aus der gleichen Prozessgruppe wie der eigene Prozess gewartet
 - ▶ *pid* == -1: es wird (wie bei *wait*) auf einen beliebigen Kindprozess gewartet
 - ▶ *pid* < -1: es wird auf ein Prozess der Prozessgruppe *-pid* gewartet.
- Der zweite Parameter ist (wie bei *wait*) ein Zeiger auf den zurückzuliefernden Status und der dritte Parameter erlaubt die Angabe von Optionen. Hier wäre ggf. *WNOHANG* interessant – dann ist *waitpid* nicht-blockierend und liefert -1, wenn noch keiner der spezifizierten Kindprozesse soweit ist.

- 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 -y lp 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.