SP Programmiertipps

aus dem FSI-Informatik Forum

(https://fsi.informatik.uni-erlangen.de/forum/thread/4717-Haeufig-gemachte-Fehler-und-allgemeine-Hinweise)

Inhaltsverzeichnis

Häufig gemachte Fehler und allgemeine Hinweise

Da mich die Korrektur der ersten Aufgabe schon dazu veranlasst hat, meinen Benutzertitel in diesem Forum umzuändern, tue ich hier zum Ausgleich mal etwas für die Allgemeinheit und stelle eine Liste von Fehlern zusammen, die mir beim Korrigieren aufgefallen sind - in der Hoffnung, dass sie dem Einen oder Anderen zur Erleuchtung gereichen wird:

1. (Aufgabe 1) Negative Werte nicht ignoriert

Es stand deutlich in der Aufgabenstellung, dass Eingaben < 0 in der Funktion append_element() ignoriert werden sollen. Warum das gefordert war, sollte einleuchten, wenn man sich klar macht, dass die Funktion remove_element() bei einer leeren Liste den negativen Wert -1 zurückliefern soll. Wenn man -1 als regulären Wert in der Schlange zuließe, gäbe es dann natürlich eine Überschneidung.

Allgemein gilt: Lest euch die Aufgabenstellung genau durch, und prüft nochmals, wenn ihr fertig seid, ob euer Programm genau das tut, was vom Aufgabensteller verlangt wird!

2. Fehlende oder unvollständige Fehlerbehandlung

Murphys Gesetz besagt: Alles, was irgendwie schief gehen kann, wird auch schief gehen. Also: Bei allen Bibliotheksfunktionen, die ihr aufruft, den Rückgabewert überprüfen und im Fehlerfall eine Meldung ausgeben. Je nach Art und Schwere des Fehlers hat es keinen Zweck mehr, das Programm fortzusetzen - falls z.B. malloc() fehlschlägt, muss irgendetwas ziemlich kaputt sein, also sollte man mit exit(EXIT_FAILURE); die logische Konsequenz daraus ziehen.

3. Speicherlecks

Ein Speicherleck entsteht, wenn man sich mit malloc() (oder einer verwandten Funktion) Speicher geholt hat, aber diesen nicht mehr mit free() zurückgibt. Das ist zwar eher unkritisch, weil sämtlicher Speicher beim Beenden des Programms sowieso wieder freigegeben wird, aber es ist schlechter Programmierstil und könnte z.B. bei Programmen, die über eine längere Zeit laufen und immer wieder dynamisch Speicher nachfordern, leicht dazu führen, dass der RAM des Rechners irgendwann plötzlich voll ist.

Ob eure Anwendung Speicherlecks enthält, findet ihr mit dem Programm valgrind heraus, das in der Tafelübung vorgestellt wurde. Wenn ihr euer Programm mit dem zusätzlichen Flag -g kompiliert, kann valgrind übrigens noch genauere Informationen anzeigen.

(Falls ihr ein Programm an einer Stelle unerwartet wegen eines Fehlers beendet, müsst ihr übrigens den Speicher nicht unbedingt aufräumen.)

4. (Aufgabe 1) Bug beim Aufrufen von remove_element() auf eine leere Warteschlange

Hier wird es etwas komplizierter.

Der einfachste Weg zu überprüfen, ob die verkettete Liste leer ist, ist das Vergleichen des head-Zeigers mit NULL.

Damit das auch wirklich funktioniert, muss man aber eine der folgenden Möglichkeiten verwenden:

a) Wenn vor dem Entfernen nur noch ein Element existiert (head == tail), head explizit auf NULL setzen.
b) Fall a) nicht extra behandeln, sondern einfach head = head->next; verwenden. Das funktioniert einwandfrei, so lange man oben in seiner append_element() für das neu angelegte Element den next-Zeiger auf NULL setzt.

Tut man dies nicht, dann ist head bei einer leeren Liste nicht zwangsläufig ein NULL-Zeiger, sondern er enthält irgendwelche zufälligen Daten, die vorher schon drinstanden. Wenn man Glück hat, enthielt er durch Zufall den Wert 0x00000000 (= NULL) - falls nicht, hat man ein Problem, weil der Zeiger ins Nirvana verweist und das Programm deswegen mit einem Speicherzugriffsfehler abstürzt.

Solche Fehler sind leider relativ schwer zu finden, weil sie nicht immer auftreten. Hier kommt wieder valgrind ins Spiel!

Bekanntes Vorgehen: Programm mit dem zusätzlichen Flag -g kompilieren und das Lieblings-Werkzeug eines jeden C-/C++-Programmierers anwerfen.

Wenn valgrind irgendeine Meldung der Art Read error, Write error, Conditional jump depends on uninitialized data o.ä. ausspuckt, ist etwas faul. Dank der obigen Compileroption wird genau angezeigt, in welcher Zeile des Quellcodes der Fehler auftritt - hierdurch sollte man (mit etwas Erfahrung) das Problem schnell eingrenzen können.

Zu valgrind gibt es auch noch eine Alternative namens "Electric fence", die auf eine etwas andere Art und Weise arbeitet. Hierfür muss man seine Anwendung mit dem Flag -lefence linken. Danach das Programm ganz normal ausführen. Falls Efence einen Fehler findet, gibt es eine Meldung aus und beendet den Prozess sang- und klanglos.

Da hänge ich mich doch glatt mal dran. Ein paar Sachen, die zwar mehr schlechter Stil (über den man bekanntlich streiten kann) als konkrete Fehler sind, aber dennoch (mindestens bei mir *g*), in der Hoffnung, einen Blick für solche Sachen zu schärfen, Punkte kosten können. Zumindest in den späteren Aufgaben (> 3), wenn man das oft genug vorher gepredigt hat.

5. static bei globalen Variablen/Funktionen vergessen

Globale Funktionen/Variablen sind ja bekanntlich (weil global) von überall aus veränderbar. Wenn also ein anderer Programmierer ein Modul schreibt, und darin auf meine globalen Variablen zugreift, dann bekomme ich das wahrscheinlich nicht mit und wundere mich dann über sich plötzlich "von selbst" verändernde Werte o.ä. Genau deswegen gibt es das Prinzip der Datenkapselung bzw. des Information Hiding. Jeder bekommt nur das zu sehen und zu verwenden, was er unbedingt braucht.

Deswegen: Alles was nicht unbedingt global sein MUSS, gar nicht erst global machen (Tipp: Man kann Funktionen auch Werte als Parameter mitgeben - über Zeiger sogar "by reference", also innerhalb der Funktion veränderbar). Und wenn es denn global sein muss, dann wenigstens "lokal global", also static. Damit ist die Variable/Funktion nur innerhalb dieses Moduls (dieser .c-Datei) sichtbar und man kann halbwegs überblicken was damit passiert. Ausnahme natürlich: Funktionen, die von außerhalb aufgerufen werden sollen. Z.B. append_element() und remove_element() der queue.

6. Funktionen, die keinen Parameter erfordern, nicht explizit als void deklariert

Das ist eine sehr "interessante" Eigenart von C. Wenn man eine Funktion dieser Art deklariert:

```
void func();
```

Dann kann sie prinzipiell mit beliebig vielen Parametern aufgerufen werden:

```
func();

func(1);

func("bla");

func(1,2,3);
```

All das kompiliert anstandslos und funktioniert auch. Das produziert an sich natürlich erst mal noch keinen Fehler, aber es kann zu Fehlverhalten im Sinne von nicht erwartetem Verhalten führen. Man ruft eine Funktion mit irgendwelchen Parametern auf und erwartet dass sie was damit macht. Macht sie aber natürlich nicht, weil der Programmierer (wahrscheinlich) eine Funktion ohne jeglichen Parameter schreiben wollte. Also bitte so deklarieren:

```
void func(void);
```

7. Unnötige if-Verschachtelungen

Was in der queue relativ häufig vorkam:

```
void append_element(int value) {

        if (value >= 0) {

                /* code */

        }

}
```

Das ist natürlich korrekt. Und dafür gibt's auch keinen Punktabzug. Aber einen netten Kommentar. *g* Das macht den Code doch nur unnötig unübersichtlich. Wenn möglich, vermeidet lange {}-Blöcke. Besonders wenn sie mal über mehr als eine Seite gehen, wird es sehr schwer, das zu überblicken. Eine einfache Abhilfe hier wäre:

```
void append_element(int value) {

        if (value < 0)

                return:

        /* code */

}
```

8. Unerwünschte Funktionalität

Der Airhardt hat ja schon auf das Einhalten der Aufgabenstellung hingewiesen. Was aber auch noch ein Problem ist, ist unnötige bzw. nicht geforderte Funktionalität. Natürlich verstehen wir, dass man sich auch mal an einer Aufgabe austoben will und Spaß dran hat, noch dies oder das zu implementieren. Aber man kann's auch übertreiben (siehe z.B. eine Shell mit eingebauten Kommandos und allem Komfort...).

Und grundsätzlich sollte man sich mal Gedanken über das Thema machen - gerade im Hinblick aufs Berufsleben, auf das die Uni ja ab und zu auch vorbereiten soll. Wenn ich dazu mal den juk (recht frei) zitieren darf: "Überlegt euch mal, wer das bezahlen soll. Der Auftraggeber hat bestimmte Features gefordert und ihr implementiert ihm noch 10 andere. Auf wessen Rechnung soll die Arbeitszeit gehen, die ihr dafür braucht?" Klar ist es bei SoS1 in erster Linie euer Problem, wenn ihr lieber noch was in eine fertige Aufgabe einbaut als das Sommerwetter zu nützen. Und mehr Übung im Programmieren schadet natürlich auch nicht. Aber bedenkt dabei: Mehr Code bietet mehr Potential für Fehler. Ein Programm, das volle Punkte gegeben hätte, erweitert mit einem unnötigen Teil, der einen Fehler enthält, gibt am Ende Punktabzug. Wenn ihr eine 20-seitige Lösung abgebt, macht ihr euch außerdem nicht gerade bei eurem Übungsleiter beliebt. UND: Es gibt einige Fächer (v.a. im Informatik-Hauptstudium), die für nicht verlangte Funktionalität rigoros einiges an Punkten abziehen!

9. Häufige Fehler im Makefile

- Das Pseudo-Target install muss die ausführbare Datei als Abhängigkeit haben, damit das Programm automatisch gebaut wird, falls die Datei noch nicht existiert.
- Das Pseudo-Target clean darf keine Abhängigkeiten haben. Wenn ich in meinem Verzeichnis aufräumen will, wäre es ziemlich sinnlos, erst das Programm zu bauen und gleich im Anschluss wieder zu löschen. Außerdem muss jeder rm-Aufruf mit der Option -f versehen werden, damit der Vorgang nicht abgebrochen wird, falls eine der zu löschenden Dateien nicht existiert.
- Alle Pseudo-Targets (Targets, die keine Datei desselben Namens erzeugen), also in diesem Fall install und clean müssen mit .PHONY: install clean explizit als solche gekennzeichnet werden. Warum das wichtig ist, könnt ihr ausprobieren, indem ihr eine neue Datei namens clean anlegt und dann make clean aufruft.

10. Strings und String-Funktionen

Immer dran denken: Jeder C-String ist durch das Zeichen '\0' terminiert, deswegen braucht man immer n + 1 Bytes an Platz.

Alle Funktionen, die irgendwas mit Strings arbeiten und eine Größenangabe als Argument fordern (z.B. fgets() oder strncpy()) schreiben immer ein abschließendes '\0', d.h. wenn ihr mit strncpy() 100 Zeichen kopieren wollt, müsst ihr den Zielpuffer 101 Bytes groß machen und der Funktion den Wert 101 als Größe übergeben.

An dieser Stelle ist mir noch aufgefallen, dass in den abgegebenen Aufgaben oft strncpy() verwendet wurde (und das teilweise falsch - s.o.), obwohl es problemlos mit strcpy() funktioniert hätte. Als Tipp: strcpy() ist nur dann gefährlich, wenn man nicht weiß, ob der zu kopierende String (+ '\0') in den Zielpuffer passt. Wenn ihr also genau wisst, dass euer Zielpuffer groß genug ist (z.B. weil ihr in mit der Größe strlen(string) + 1 allokiert habt), könnt ihr gefahrlos strcpy() verwenden.

11. Verwendung von fgets()

Die einfachste Möglichkeit, um zu prüfen, ob eine Zeile länger als 100 Zeichen ist, geht in etwa so (wichtig ist auch die Fehlerabfrage!):

```
char buffer[102];

if (fgets(buffer, 102, stdin) == NULL) {

        if (ferror(stdin)) {

                /* Fehler beim Lesen aufgetreten -> Programm abbrechen */

        } else {

                /* Dateiende -> Schleife verlassen */

        }

}

if (buffer[strlen(buffer) - 1] != '\n') {

        /* <= 100 Zeichen -> Zeile verarbeiten */

} else {

        /* > 100 Zeichen -> Rest der Zeile einlesen und wegwerfen */

}
```

Wie und warum das funktioniert, versteht ihr am besten, wenn ihr euch mal den ungefähren Inhalt des Puffers in den Grenzfällen auf Papier malt.

12. Verwendung von perror()

Viele Leute haben die Funktion perror() verwendet, um Fehlermeldungen auszugeben, falls eine zu lange Zeile eingelesen wurde. perror() gibt aber nicht einfach so eine Meldung nach stderr aus, sondern tut in etwa das hier:

```
void perror(const char *s)  {

        fprintf(stderr, "%s: %s\n", s, strerror(errno));

}
```

Im Klartext: perror() gibt den übergebenen String auf stderr aus, gefolgt von einem Doppelpunkt und der textuellen Beschreibung des zuletzt aufgetretenen Fehlers, dessen Fehlercode in der Variable errno gespeichert ist. Bei den Leuten, die perror() falsch verwenden, sieht es dann z.B. so aus:

Diese Zeile ist länger als 100 Zeichen!: Success

13. Uninitialisierter Speicher

Immer daran denken: Speicher, der von malloc() und realloc() (der neue Teil natürlich) kommt und alle lokalen Variablen sind IMMER uninitialisiert. Da kann alles Mögliche drin stehen. Also aufpassen, ob ihr irgendwo die Variable verwendet, ohne ihr vorher etwas zugewiesen zu haben.

14. Böse Sachen mit argv anstellen

argv ist ein Aufrufparameter von main(). Als solcher liegt es auf dem Stack. Versucht nicht, daran mit realloc() herumzubasteln! In argv bekommt man seine Aufrufparameter übergeben und es ist kein Speicherplatz für eigene Dinge...

15. man-pages lesen!

Das gilt nicht nur für Aufgabe 3 (mini_sh), wo ihr euch die Funktionsweise von getcwd(), strtok() und chdir() eigenständig erarbeiten solltet, sondern ganz allgemein.

Ein paar Dinge, die so ganz deutlich in der man-page drinstehen und trotzdem immer wieder falsch gemacht wurden:

- getcwd(): The buf argument should be a pointer to an array at least PATH_MAX bytes long.
- In der Seite zu wait() findet man ganz unten ein wunderschönes Beispiel, wie man die Makros zur Analyse des Rückgabestatus verwendet.
- In allen man-pages steht genau, was die jeweilige Funktion im Fehlerfall zurückgibt. Diese Rückgabe muss man dann natürlich auch abfragen, um festzustellen, ob ein Fehler aufgetreten ist!

Auch gerade bei der aktuellen malloc()-Aufgabe solltet ihr auch genau lesen, wie das Verhalten der von euch zu implementierenden Funktionen spezifiziert ist (Stichwort spezielle Parameter und Fehlerbehandlung).

16. Statische vs. dynamische Arrays

Schön, wenn ihr etwas gelernt habt und es dann auch gleich exzessiv anwenden wollt.

Im Falle von malloc() und Konsorten ist das aber nicht immer optimal. In den allermeisten Situationen reicht es völlig aus, statt

```
char *buffer = malloc(1337 * sizeof(char));
```

ganz einfach

```
char buffer[1337];
```

zu schreiben.

Dadurch erspart ihr euch nicht nur einen ganzen Haufen Arbeit, weil ihr weder eine Fehlerüberprüfung braucht noch euch darum kümmern müsst, den Speicher später wieder mit free() freizugeben - sondern auch etliche potenzielle Fehlerquellen.

Das Ganze funktioniert natürlich nur unter zwei Bedingungen:

1. Statische Arrays liegen im Stack-Segment und sind deswegen nur so lange gültig, wie sich der Kontrollfluss des Programms in der Funktion befindet, in der das Array deklariert wurde (oder in einer Funktion, die direkt oder indirekt von dieser Funktion aus aufgerufen wurde).
2. Die (maximale) Größe des Arrays muss zum Zeitpunkt der Programmierung bekannt sein, weil man sie zur Laufzeit nicht mehr verändern kann.

Bei der mini_sh] konnte man z.B. komplett ohne dynamische Arrays auskommen:

```
char  dir[PATH_MAX]; /* fuer getcwd(); wegen PATH_MAX s.o. */

char  input[1026];   /* fuer fgets(); 1024 Zeichen + '\n' + '\0' */

char *args[513];     /* Ergebnisse von strtok() */
```

args ist ein Feld von Zeigern auf die einzelnen Teilstrings der zerlegten Eingabe. Theoretisch kann die Eingabezeile maximal in 512 Teile zerlegt werden - nämlich dann, falls immer abwechselnd ein Zeichen und dann ein Leerzeichen kommt. Sobald das Array zusammengebaut wurde, muss noch ein abschließendes NULL dahinter gehängt werden (das wurde gerne vergessen!), deswegen insgesamt 513.

Einer meiner Lieblinge:

17. Magic numbers are evil

Wann immer es sich vermeiden lässt, sollte man im Code keine direkten numerischen Literale verwenden. Das macht es nicht nur schwerer, den Sinn der Zahl zu verstehen, sondern auch nachträgliche Änderungen (wenn die Zahl mehrfach vorkommt) werden fehleranfällig.

Bei allen sizeof()s sollte das hoffentlich einsichtig sein (z.B. nicht 4 sondern sizeof(char *)...), aber dann:

Die Puffergröße in der Shell für Eingaben ist z.B. idealerweise 1026. Also am Anfang vom Programm ein #define BUFSIZE 1026 und dann überall wo man das braucht BUFSIZE verwenden und nicht 1026.

Die Returnwerte sind auch ein guter Kandidat. Bei der main() am besten konsequent IMMER EXIT_SUCCESS oder EXIT_FAILURE und NIE 0 oder 1 verwenden. Und wenn man eigene Rückgabewerte für main definieren möchte (darf man ja auch), dann die auch oben als Konstanten (Makros) definieren.

Aber auch wenn man eine eigene Funktion schreibt, die mehrere oder gar viele verschiedene Rückgabewerte mit eigener Semantik hat. Dann ruhig #define FOOSUCCESS 0, #define FOONOMEM 1, #define FOOBADWEATHER 2 usw.

18. Datentypen haben auch Rechte

Wenn eine Funktion einen bestimmten Datentyp erwartet, dann hat das meistens einen Grund. Die ganzen Speicherverwaltungsfunktionen z.B. erwarten alle eine Größenangabe vom Typ size_t. Sehr beliebt ist aber, ihnen eine Variable vom Typ int zu übergeben. Gut, size_t ist meistens auch int. Aber was wenn nicht? Wenn size_t kleiner als int ist? Dann bekommt malloc() einen völlig anderen Wert übergeben, als man erwartet. Deswegen muss man natürlich nicht Zahlen casten - malloc((size_t)400) ist unnötig, weil das der Compiler eh macht (hier natürlich 17. lesen und überlegen, ob an der Stelle 400 stehen sollte). Also bitte in der Manpage nachschauen, was eine Funktion (oder Struktur) für Datentypen erwartet, und diese auch verwenden.

Zu dem Thema passt auch unsigned/signed. Wenn man die miteinander mischt, warnt einen der Compiler meistens. Bevor man jetzt die Warnung einfach durch einen expliziten Cast unterdrückt, sollte man noch mal kurz nachdenken, ob an der Stelle nicht vielleicht doch ein Bug versteckt ist. Z.B. daran denken dass unsigned int viel größere (positive) Zahlen darstellen kann als int. Das gibt sehr lustige Effekte wenn man den Bereich eines Datentyps überschreitet.

Und weiter geht's mit Aufgabe 4:

19. man-pages lesen, Teil 2

Gut, das war diesmal bei weitem nicht so extrem, wie ich befürchtet hatte. Ich kann mir vorstellen, dass die mitgelieferten Testcases da dem einen oder anderen von euch den richtigen Weg gezeigt haben.

Konkret geht es mir hier um die diversen Sonderfälle, die bei den Eingabeparametern zu beachten sind: malloc(0), free(NULL), realloc(NULL, n) und realloc(p, 0).

Außerdem sollte im Fehlerfall das Programm explizit nicht abgebrochen und auch keine Fehlermeldung ausgegeben werden, sondern nur die errno gesetzt und ein NULL-Zeiger zurückgegeben. Um die endgültige Fehlerbehandlung muss sich dann der Benutzer kümmern, der eure Funktionen in seinem eigenen Programm verwendet!

20. Fehler in realloc() und calloc()

Mit realloc() kann man einen Speicherblock nicht nur vergrößern, sondern auch verkleinern. Falls man nun den Inhalt des alten Blocks in den neuen Block kopiert, muss man dabei als Byteanzahl immer das Minimum der alten und der neuen Blockgröße angeben. Sonst könnten jeweils folgende Situationen auftreten:

a) Man kopiert immer die neue Anzahl von Bytes. Falls man den Puffer vergrößern will und der alte Block am Ende des 1-MB-Segments liegt, liest man bei memcpy() über dessen Rand hinaus und sorgt dadurch für einen Programmabsturz.
b) Man kopiert immer die alte Anzahl von Bytes. Falls man den Puffer verkleinern will, wird durch memcpy() der nachfolgende Block und seine Verwaltungsstruktur teilweise überschrieben.

Außerdem muss man sowohl in realloc() als auch in calloc() den Rückgabewert des intern verwendeten malloc()-Aufrufs abfragen. Es kann ja jederzeit sein, dass dieser fehlschlägt, weil nicht genügend Platz frei ist. Wenn man das nicht abfängt, sondern einfach drauflos schreibt (in realloc() mit memcpy(), in calloc() mit memset()), gibt es einen kleinen Programmabsturz...

21. Testcases verwenden

Ich persönlich war vom Ergebnis meiner Gruppe extrem positiv überrascht, deswegen schätze ich, dass die meisten Leute die mitgelieferten Testcases auch verwendet haben. Auf vielfachen Wunsch einer einzelnen Person erwähne ich das aber trotzdem: Wer diesen Luxus nicht ausgenutzt hat, ist selber schuld!

Und falls der Testcase mitten in der Ausführung abgestürzt, ist der gdb euer bester Freund. Die drei häufigsten Ursachen für Crashes:

- Zugriff auf NULL-Zeiger (entweder beim "Suchzeiger" oder beim "Schleppzeiger").
- Beim Durchlaufen der Schleife stößt man auf einen 0x00beef00-Zeiger, weil man einen Fehler im free() hat.
- Man landet mit einem Zeiger jenseits der 1-MB-Grenze. Das findet man leicht raus, indem man von dieser Adresse den Wert des newmem-Zeigers abzieht.

22. Dokumentation nicht vernachlässigen

Mit einem kleinen bisschen Hinsetzen und Nachdenken wäre die Bug-Finde-Teilaufgabe eigentlich relativ simpel zu lösen gewesen. Zur Not nimmt man halt mal einen Stift und ein Blatt analoges Papier und pinselt sich hin, welcher Zeiger wohin zeigt und was in welchem Speicherblock steht.

In meinen Gruppen gab es auf die Teilaufgaben c) und d) insgesamt drei Punkte, das ist dann doch immerhin ein Viertel der Gesamtpunktzahl!

23. Makefiles: "Umweg" über .o-Datei gehen

Dies ist der Grund, warum bei Aufgabe 5 so gut wie niemand die vollen 4,0 Punkte bekommen hat. Zwar ziemlich pingelig, aber hoffentlich einzusehen: Wenn man in seinen beiden Targets wsort-static und wsort jeweils die wsort.c als Abhängigkeit angegeben hat, bürdet man dem Compiler unnötige Arbeit auf, denn:

- Für beide Anwendungen müsste die wsort.c jeweils neu kompiliert werden, obwohl beide sich bekanntlich nur in der Verwendung der halde-Bibliothek unterscheiden.
- Falls man an seiner Bibliothek etwas ändert, würde das gesamte Programm neu kompiliert werden, obwohl es nur neu gelinkt werden muss.

Deshalb wird in der Musterlösung ein zusätzliches Target wsort.o erstellt und dieses für die Anwendungen als Abhängigkeit angegeben.

An dieser Stelle nochmals der Hinweis: Alle Targets, die keine gleichnamige Datei als Ausgabe erzeugen, müssen als .PHONY markiert werden.

24. SetUID, SetGID und Sticky-Bit nicht vergessen!

Das wurde im printdir gerne weggelassen, obwohl es eigentlich eindeutig in den Übungsfolien steht.

25. Fehlerbehandlung, die Zweite

Obwohl das immer wieder bis zur Vergasung wiederholt wurde, mache ich das hiermit zum x-ten Male deutlich: Bei allen Funktionen, die laut man-page fehlschlagen können, muss der Rückgabewert abgefragt und falls nötig eine

Fehlerbehandlung durchgeführt werden!

Einzige Ausnahme: Bei Funktionen wie z.B. sigaddset(), die nur dann einen Fehler meldet, wenn man ihr trotteligerweise eine ungültige Signalnummer übergeben hat, kann man die Fehlerabfrage weglassen. Ansonsten gilt (Gebetsmühle): Fehlermeldung nach stderr ausgeben (meistens bietet sich perror() an) und dann den Fehler je nach Situation angemessen behandeln.

"Angemessen" bedeutet, dass nicht prinzipiell immer sofort das Programm abgeschossen wird. Gegenbeispiel: Wenn beim printdir das Einlesen eines Verzeichnisses fehlschlägt (z.B. weil der Zugriff verweigert wurde), bietet es sich an, dieses Verzeichnis zu überspringen und mit dem nächsten fortzufahren.

Nur bei wirklich kritischen Fehlern wie einer fehlgeschlagenen Speicheranforderung muss das Programm beendet werden. In diesem Fall braucht man auch keine aufwändige Speicherbereinigung mehr durchzuführen.

26. Sortierung nach Änderungszeit

Wie man qsort() verwendet, sollte seit Aufgabe 2 bekannt sein. Die meisten Leute haben es auch richtig gemacht und sich ein Array aus Strukturen zusammengebastelt, die alle wichtigen Informationen (inklusive Modifikationszeit) aus den eingelesenen Verzeichniseinträgen enthalten. Die Vergleichsfunktion muss dann entsprechend zwei Zeiger auf solche Strukturen erhalten und deren Modifikationszeit-Einträge miteinander vergleichen. Wichtig ist hier, dass die jüngste Datei (die mit der höchsten Modifikationszeit) als erstes kommen sollte, d.h. das Array muss absteigend sortiert werden.

Kleine Randnotiz: So etwas wie return (e2->mTime - e1->mTime); funktioniert auf 32-Bit-Systemen problemlos, während es auf einem 64-Bit-System zu unerwarteten Ergebnissen kommen könnte, falls die eine Datei > 68 Jahre (INT_MAX Sekunden) jünger ist als die andere. Dann gäbe es nämlich einen Überlauf, weil zwei 64-Bit-Zahlen voneinander abgezogen werden und das Ergebnis in eine 32-Bit-Zahl gespeichert wird... (weiterführende Informationen)

Ganz streng genommen müsste man es also so machen:

```
if (e1->mTime > e2->mTime) {
        return -1;
} else if (e1->mTime < e2->mTime) {
        return 1;
} else {
        return 0;
}
```

Ein paar Leute haben sich auch nur die Dateinamen gemerkt und dann in der Vergleichsfunktion lstat() aufgerufen. Das ist konzeptionell nicht sehr toll, langsam, und stellt einen vor die Frage, was man macht, falls lstat() fehlschlagen sollte.

27. Noch mal ein altes Lieblingsthema

Spätestens nach dem fünften Mal, dass der Übungsleiter einem einen halben Punkt abgezogen hat, sollte die Information durchgesickert sein, dass Variablen und Hilfsfunktionen, die nur innerhalb des eigenen Moduls verwendet werden sollen, als static deklariert werden müssen.

Dummerweise wurde das in der mini_sh, die für Aufgabe 7 vorgegeben war, auch falsch gemacht...

28. Signalbehandlung

Was viele vergessen haben, obwohl in der Tafelübung deutlich darauf hingewiesen wurde: Wird in einer Signalbehandlungsfunktion ein Bibliotheks- oder Systemaufruf verwendet, der fehlschlagen kann, muss man vorher die errno sichern und anschließend wiederherstellen.

29. Nebenläufigkeitsprobleme in der job_sh

Eine kurze Auflistung, was man hätte machen sollen:

a) Die Jobliste mit jl_new() anlegen, bevor der SIGCHLD-Handler installiert wurde.
b) SIGCHLD vorübergehend blockieren, während man im Kommando "jobs" die Liste der aktuellen Hintergrundprozesse durchgeht.
c) Wenn ein Hintergrundprozess gestartet wird: SIGCHLD und SIGINT vor dem fork() blockieren, damit der Hintergrundprozess nicht abgeräumt werden kann, bevor er überhaupt in die Jobliste eingetragen wurde. Das SIGCHLD kann im Kindprozess sofort wieder deblockiert werden, das SIGINT erst dann, wenn es mittels sigaction() auf die Ignorierliste gesetzt wurde.

Im Vaterprozess kann SIGINT sofort deblockiert werden, während man SIGCHLD erst wieder dann freigeben kann, sobald man den Kindprozess in die Jobliste eingetragen hat.

d) Bei der vorgegebenen Shell gab es Probleme, wenn ein Hintergrundprozess fertig wurde, während ein Vordergrundprozess am Laufen war - genauer gesagt wurde der Exitstatus des Hintergrundprozesses zusammen mit der Befehlszeile des Vordergrundprozesses ausgegeben. Für diese Problematik gibt es prinzipiell zwei Lösungsmöglichkeiten, gedacht war folgende:

SIGCHLD vor dem fork() blockieren (damit im Fall, dass der Kindprozess vorzeitig stirbt, das SIGCHLD erst einmal zurückgehalten wird). Im Kindprozess kann man es dann sofort wieder deblockieren; im Vaterprozess wartet man mit sigsuspend(), bis der Signalhandler den Kindprozess aufgeräumt hat. Dazu müsste man sich global die PID und die Kommandozeile des Vaterprozesses merken und in der Signalbehandlungsfunktion überprüfen, ob die von waitpid() zurückgelieferte PID die des Vaterprozesses ist.

e) Wer es ganz genau nimmt, müsste beim Programmende vor dem Aufruf von jl_destroy() (den nicht wenige Leute vergessen haben) das SIGCHLD blockieren und danach nicht mehr deblockieren (oder es alternativ ignorieren).

30. SIGCHLD-Handler

Das Warten auf Kindprozesse muss in einer Schleife erfolgen, denn es kann vorkommen, dass nur ein Signal ausgelöst wird, obwohl mehrere Kinder gestorben sind. Man muss also waitpid() mit der PID -1 (auf alle Kindprozesse warten) und dem Options-Flag WNOHANG (nicht blockieren, falls kein Zombie gefunden wurde) aufrufen und den Rückgabewert überprüfen. Falls die Rückgabe 0 ist (oder -1 und errno == ECHILD) liegt kein Zombie (mehr) vor.

Allgemein sollte man beim Einrichten von Signal-Handlern das Flag SA_RESTART setzen, sonst müsste man streng genommen jeden einzelnen Systemaufruf, der mit EINTR fehlschlagen könnte, einzeln abprüfen und falls nötig neu aufsetzen.

31. Schließen nicht benötigter Sockets

Zitat der Aufgabenstellung: Beachten Sie, dass die offenen Filedeskriptoren an die Kind-Prozesse vererbt werden und alle Sockets solange bestehen bleiben, bis der letzte Prozess seine Filedeskriptoren geschlossen hat. Darum müssen Sie darauf achten, dass jeder Prozess nur die Filedeskriptoren geöffnet hält, die er auch wirklich braucht.

Das heißt im Klartext: accept() erzeugt bekanntlich bei der Verbindungsannahme einen neuen zusätzlichen Socket, der für die Kommunikation mit dem verbundenen Client verwendet werden kann, während man auf dem ursprünglichen Socket weiter auf eingehende Verbindungen lauschen kann.

Der Kindprozess soll die aktuelle Uhrzeit verschicken und sich nicht um weitere Verbindungsannahmen kümmern, deswegen muss er mit close() den "Horch-Socket" schließen. Der Vaterprozess seinerseits braucht den "Kommunikations-Socket" nicht mehr und muss ihn schließen.

Hier nochmals der Hinweis: close() schließt einen Dateideskriptor nur für den aufrufenden Prozess, aber nicht für andere Prozesse, die auf denselben Deskriptor zugreifen. Im Gegensatz dazu kann man mit shutdown() den Hahn komplett zudrehen - was uns gleich zum nächsten Punkt bringt.

32. Umgang mit shutdown()

Wer in seinem port_forward kein shutdown() an den richtigen Stellen verwendet hat und sein Programm ausgiebig getestet hat, dürfte festgestellt haben, dass in manchen Fällen der Kindprozess offen gehalten wurde, obwohl eine der zwei Gegenstellen eigentlich die Verbindung beendet hatte. Der Grund hierfür ist, dass der andere Rechner nicht mitbekommt, dass die Verbindung getrennt wurde und dadurch der zweite Forward-Thread endlos weiter läuft.

Um es einmal grafisch aus Sicht eines der zwei Threads darzustellen:

[Rechner A]--->[port-forward]--->[Rechner B]

Schlägt nun das Lesen von Rechner A fehl, dann ist die Verbindung getrennt. In diesem Fall muss Rechner B Bescheid bekommen, dass (zumindest in diese Richtung) keine Daten mehr laufen können, und zwar indem ein shutdown() mit dem Parameter SHUT_WR auf den Schreib-Socket durchgeführt wird.

Analoges gilt, falls das Schreiben auf Socket B fehlschlägt - dann muss Socket A in Leserichtung (SHUT_WR) geschlossen werden.