BS Praktikumsaufgabe 04

Ein Kernelmodul mit Stoppuhrfunktion

Version 0.8

Alexander Mendel
Karl-Fabian Witte

erstellt am 20. Februar 2023

Es soll ein Kernelmodul / Treiber mit Stoppuhrfunktion auf der Virtuellen Maschine implementiert werden. Dabei zählt ein Gerät vorwärts `/dev/timerf` und eins rückwärts `/dev/timerr`. Als Zeiteinheit dienen die sogenannten `jiffes`. (Ein Linux eigener Integerwert, welcher nach einer bestimmten Anzahl von Takten um Eins erhöht wird.) Ziel ist es, ein wenig in den Abgründen des Linuxkernels zu schnuppern und sich mit den Konzepten der Treiber-/Modulprogrammierung vertraut zu machen.

Inhaltsverzeichnis

1 Entwurf

Es wird versucht sich an dem Buch "Linux Device Drivers" von Corbet, Rubini und Kroah-Hartman zu halten. Zudem sollen sollen nur die zwei "Geräte" `timerr` und `timerf` erlaubt sein und diese auch nur jeweils in einmaliger Instanz.

1.1 Der Automat

Die Geräte sind Automaten, die auf folgende Befehle reagieren (z.B. `echo s > /dev/timerr`):

s : Start:

- `timerf`: `READY` → `RUNNING`
- `timerr`: `LOADED` → `RUNNING`

p : Pause: `RUNNING` → `PAUSE`

c : Continue: `PAUSE` → `RUNNING`

r : Reset: (alle internen Zeitwerte zurücksetzten): → `READY`

l<value> : Laden (nur `timerr`): `READY` → `LOADED`

Dabei wird der Zustand der Geräte in der Funktion `timer_write` entsprechend geändert und die Werte werden gesetzt. Es wird mit `copy_from_user` in einen

Kernelpuffer gefüllt und dann mit `strchr` ausgewertet. Die Ausgabe des aktuellen Zeitwertes wird in `timer_read` berechnet und ausgegeben. Bei der Berechnung der Zeitwerte mit den jiffis ist zu beachten, dass diese nicht negativ werden können, jedoch gibt es Macros im jiffi header, welche den Vergleich zweier Zahlen zuverlässig im Zeitraum einiger Tage abnehmen. Hier wird mit `copy_to_user` die Ausgabe zum Benutzer geschickt. Sowohl `timer_write` als auch `timer_read` haben einen lokalen dynamischen Puffer, welcher mit einem Block auf einem Semaphoren (hier Mutex) sicher allokiert und wieder entfernt wird. Kontroverse Entwurfsentscheidung: Wenn die Zeit der runterzählende Uhr abgelaufen ist, bleibt diese im Zustand RUNNING.

1.1.1 Hilfsfunktionen

Sowohl für `timer_read` als auch für `timer_write` wurden Hilfsfunktionen erstellt. Für `read` wurde die Berechnung der Zeit in `calc_time` gekapselt. Dort wird mit den eher starren Zeitvariablen hantiert um die richtige Zeit zu berechnen, welche dann zurückgegeben wird. Dabei wird auf das makro `time_befor64` die Rückwärtsuhr verwendet, um zu testen, ob die Zeit noch nicht erreicht wurde.

In der `timer_write` wird mit `get event` zunächst der String aus dem Userspace nach Befehlen durchsucht und ein entsprechendes `enum event` zurückgegeben. Da der Load aufruf auch ein Argument verlangt, wird eine Struktur mit `event` und `u64` zurückgegeben. In der Funktion `update_state` wird dann mit einer Fallunterscheidung der Zustände der Automat entsprechend der Events umgeleitet und die Zeitwerte gesetzt. Nur beim Austritt aus dem Pausezustand wird der Offset erneut berechnet, von der aus die Zeit in `read` berechnent wird. Ansonsten finden alle Zeitberechnungen innerhalb `timer_read` statt.

2 Die Funktionen

Damit unser Modul auch richtig funktioniert, müssen entsprechende Funktionen über eine bestimmte API in den Kernel eingebunden werden. Hier soll zunächst kurz über die Einbindung gesprochen werden und später werden die Abläufe in den Funktionen besprochen Um das Module und de Treiber zu initialisieren, wird eine entsprechende Funktion einem dem Makro `module_init` übergeben. Beim entfernen des Modules muss eine entsprechende Funktion dem Makro `module_exit` übergeben werden.

```
1    module_init(timer_init);
2    module_exit(timer_cleanup);
```

`timer_init` wird beim laden des Modules ausgeführt, und `timer_cleanup` entsprechend beim entfernen.

Wie sich der Treiber verhält, wird in der Struktur `struct file_operations` festgelegt. Bei Zeichen orientierten Modulen wird `.owner` mit der symbolischen Konstanten `THIS_MODULE` belegt. Unsere Devices sollen vom Benutzer gelesen werden `.read` (`cat /dev/timerf`) sowie auch Befehle entgegennehmen `.write` können (`echo <cmd> > /dev/timerr`). DA das Device als Datei in Linux existiert, muss es bei solchen operationen geöffnet `.open` und wieder geschossen werden `.realaese`:

```
1  struct file_operations timer_fops = {
2      .owner   = THIS_MODULE,
3      .read    = timer_read,    /* Berechnung und Ausgabe der Zeit an User*/
4      .write   = timer_write,   /* Zustandsänderung */
5      .open    = timer_open,    /* Uebergabe der Geraeteinfos ans file */
6      .release = timer_release, /* eigentlich nix */
7  };
```

2.1 init

Die Funktion `timer_init` soll das Modul und die Devices dem Betriebssystem nutzbar gemacht werden. Dafür wird zunächst die Majornummer dynamisch ermittelt, der Modulmutex erzeugt und darauf werden die Strukturen mit dem Informationen für die beiden Device mit der Funktion `timer_dev_init` gefüllt.

2.1.1 timer_dev_init

Die Struktur `timer_dev` für die beiden Stoppuhren werden hier mit den Anfangwerten gefüllt. Zuerst wird der Speicher der Struktur alloziert. Die beiden Treiber werden über ihre Minornummer unterschieden, welche hier auch übergeben wird. Ein Mutex soll zudem den dynamisch allozierten, lokalen Kernelbuffer vor spontaner Terminierung schützen, damit der allozierte Speicherbereich wieder garantiert wieder freigegeben wird. Das interessanteste ist die Character Device Struktur, welche hier erzeugt und der `timer_dev` übergeben wird. Mit dieser Struktur wird das Device als Zeichenbezogenes Gerät von dem Betreibsystem behandelt.

2.2 cleanup

`timer_cleanup` befreit den Speicher in umgekehrter Reihenfolge, wie er in `timer_init` alloziert wurde.

2.3 open

`timer_open` macht zunächst ein `down_trylock` auf den Modulmutex `open_sem` und gibt ggf. einen Fehlerwert zurück. Beim erfolgreichen "Locken" wird der Filestruktur die Datenstruktur `timer_dev` des agierenden Gerätes bekannt gemacht.

2.4 release

`timer_release` befreit einfach den `open_sem`.

2.5 read

`timer_read` gibt die aktuelle Zeit der Stoppuhr wieder. Die Funktion wird erst nicht mehr aufgerufen, wenn eine negative Errornummer oder eine Null übergeben wird. Es wird jedoch nur der Zahlenwert der Rückgabe auch in den Userspace dann gedruckt. Deswegen wird die Funktion mindestens zweimal aufgerufen, wo jedoch beim letzten Mal Null zurückgegeben wird. Wir wissen, da die der Ausgabestring ein festest Format hat, wie viele Bytes übergeben werden müssen, und testen diese Anzahl mit dem Offset, welcher an die Stelle zeigt, an der wir das letzte mal aufgehört haben. Wenn der Offset kleiner als die erwünschte Anzahl ist, wird der Gerätemutex gelockt und ein lokaler Buffer alloziert. Dann wird die momentane Zeit in `calc_time` berechnet. Mit der Zeit wird in den Buffer die Message geschrieben und mit `copy_to_user` zum User transferiert. Danach wird der Offset neu berechnet, der Speicher des Buffers freigegeben und der Mutex wieder gelöst.

2.5.1 `calc_time`

In dieser Funktion wird die Zeit berechnet. Da wir meist statische Zustandsvariablen haben, wird hier wild gerechnet. Dafür werden vier lokale Variablen angelegt. Eine, welche die momentane Zeit beinhaltet `now`. `time` ist die Variable, welche den

Rückgabewert bunkert.`pause` ist ein Kontainer für die in Pausedauer, wenn sich das Gerät in der Pause befindet. Für die rückwärts laufende Uhr wurde noch eine Variable `goal` definiert, welche den Zielzeitpunkt der Stoppuhr beinhaltet. Mit diesen Variablen wird dann die Zeit berechnet. Wenn das Ziel `goal` erreicht wurde, soll nur Null zurückgegben werden. Dafür wird mit `time_befor64` der Jiffiesbibliothek die momentane Zeit `now` gegen `goal` getetet.

2.6 write

Die Funktion `timer_write` betritt erstmal einen vom "privaten" Gerätemutex geschützten Bereich und erzeugt einen Kernelbuffer, um die Informationen mit `copy_from_user` in diesen zu speicher. Das erneute Aufrufen der Funktion hört erst bei einer negativen Errornummer oder bei der gesammten Anzahl der übermittleten Bytes auf. Bei erfolgreicher Übermittlung aller Bytes vom User- zum Kernelspace, wird die Funktion `update_state` aufgerufen, welche zunächst aus dem Buffer mit der Funktion `get_event` den String in ein Event umwandelt. Danach wird der Status des Automaten neu gesetzt. Zurück in `timer_write` wird der Speicherplatz des Buffers wieder frei gegeben und der Mutex hoch gesetzt.

2.6.1 `get_event` und `update_state`

Für `get_event` wird ein statisches und konstantes Array erzeugt, welches alle gültigen Befehle in der entsprechenden Reihenfolge des Enums `event` enthält. Über eine Schleife über das Array, wird mit Hilfe von `strchr` in dem Buffer nach dem Befehl gesucht. Wird ein Befehlszeichen identifiziert, wird ein Zähler inkrementiert und das Event (hier momentaner Laufvariablenwert) der Rückgabevariable zugewiesen. Da bei dem Ladenevent ein Zahlenwert mit übergeben werden muss, wird dieser in so einem Fall mit `sscanf` ausgelesen. Schlägt diese Funktion fehl, wird das Flag `is_time_valid` gelöscht. Am ende wird nochmal geprüft, ob nur ein Befehl im String enthalten war und ob das Flag setzt ist. Wenn eines nicht zu traf, dann wird ein unbekanntes Event übergeben. Zu Debugzwecken wird hier der Bufferinhalt ausgegeben. Der Rückgabewert ist eine Struktur aus Eventwert und Zeitwert. `updte_state` überprüft für je nach Staus alle möglichen Events. Wenn jedoch ein unbekanntes oder das ein reset Event passiert ist, werden diese vorher abgefangen und behandelt. Meist wird nur ein Zeitwert mit `get_jiffi64` und der Status neu gesetzt. Nur bei der Transition vom Pausestatus zum Runningstatus wird die Dauer Pause auf die Offsetvariable dazugerechnt.

3 Racing Conditions

Wir haben mit dem Modulmutex und dem Devicemutex etwas übertrieben, da einer der beiden für unsere Zwecke gereicht hätte. Der Modulmutex ist hier der "Überflüssigere", da wir keine Dateiglobalen Variablen haben, die von beiden Geräten verändert werden. Doo

4 Debugging

Im Kernel zu debuggen ist relativ umständlich. Es kann nicht mit einem Debugger gearbeitet werden. Es muss auf die gute alte "print everywhere and everything" Methode zurückgegriffen werden. Dafür wurde ein Makro `PDEBUG` erstellt, welches mit der Kernel eigenen Format-Print-Funktion `printk` und dem dazugehörigen Makro `KERN_DEBUG` Meldungen an die Datei `/var/log/message` übergibt, welche wir dann in einer zusätzlichen Konsole mit `tail -f` "live" beobachten. Es werden mithilfe von `grep` oder `awk` die wichtigen Information heraus gefiltert. Es wurde zu dem eine Funktion `dev_dump` erstellt, um den momentanen Status der Timer zu erfahren. Diese wird immer nach jeder Statusänderung des Automaten ausgegeben. Zudem wir in der Funktion `get_event` der Buffer ausgegbeen, damit man diese Evententscheidung nachvollziehen kann. In `calc_time` werden die lokalen Variablen ausgegeben, da die Berechnung nachvollziehbar bleibt. Zudem werden bei jedem Funktionsaufruf der Name der Funktion ausgerufen, um den momentanen Programmzeiger/Verlauf zu verfolgen. Noch besser wäre, beschreibende Nachrichten in den Debugmitteilungen zu schreiben, die auch ein paar mehr Informationen des Zustandes der lokalen Variablen beinhalten.

5 Installierung

Das `Makefile` (welches unbedingt mit großen "M", sonst findet er das nicht vom Kernel aus) erstellt mit dem Befehl `make` das Modul. Die Compilerfalgs werden mit der vom Kernel eigenen Variable `ccflags-y` übergeben. Für das mühselige laden und löschen der "Geräte" wurde ein Shellskript `timer.sh` erstellt, welches mit den Optionen `-load, -unload` diese Arbeit abnimmt. Dabei ruft `load` die `unload`-Methode indirekt auf, um alte Geräte und Treiber zu entfernen. Danach wird dann der Treiber `insmod timer.ko` in den Kernel eingefügt, mit Hilfe von Linuxterminalhacks wird die Majornummer herausgeschrieben und mit dieser via `mknod` und der Minornummer die

Geräte installiert. Und da das noch nicht bequem genug ist, kann man als Superuser mit `make load` bzw. `make unload` das laden auch so bewerkstelligen.

6 Kommentare

Für die genauere und detailliertere Betrachtung, haben wir den Code extra viele Kommentare beigefügt, welche bei der Abnahme fehlten.

7 Sonstiges

Es wird für die Geräte die folgende Struktur verwendet, welche hier kurz aufgeführt wird.

```
1  struct time {
2      u64 offset;   /* startzeit + pausenzeiten */
3      u64 enter_pause; /* zeitpunkt beim Eintritt in den Pausenzustand */
4      u64 load;       /* geladene Zeit */
5  }
6
7  struct timer_dev {
8      int minor;              /* timerf or timerr */
9      char name[TIMER_NAME_LEN]  /* string for name */
10     struct time time;       /* time variables */
11     enum state state; /* current state */
12
13     struct semaphore sem;   /* mutex to avoid race conditions */
14     struct cdev cdev;       /* stuff for char devs */
15 };
16
```

mit dem Befehl `./timer.sh -setup` kann man sich zwei Konsolen öffnen, wovon eine das Logfile anzeigt und das andere für die Befehlseingabe bereit steht. `./timer.sh -test` wird ein testlauf durchgeführt, welcher nach unseren Ermessen die korrekte Funktionalität präsentiert, da fast alle (nicht immer die nicht erlaubten/bekannten) Befehle in jedem Zustand durchprobiert wuden (Robusthaitstest).

Es wird versucht, gänzlich auf `goto` zu verzichten, obwohl dieses im Buch für das Zusammenspiel zwischen `init` und `cleanup` sehr gut umgesetzt wurde. Für uns hat

sich das `goto` noch nicht als großer Nutzen gezeigt, da wir ja auch nicht auf Performance aus sind.