Betriebssysteme - 2. Test

Einführung in Unix

- **Wie ist ein UNIX System aufgebaut?**
  Hardware (CPU, Speicher, …)
  Software (OS: Kernel + Treiber, Benutzerprogramme)
  Mehrbenutzer- und Mehrprozessbetriebssystem
- **Wie ist ein UNIX Kernel aufgebaut?**
  - Process management component: signal handling, process/thread creation/termination, CPU scheduling
  - Memory management component: virtual memory, paging page replacement, page cache
  - IO and networking component: virtual file system, terminals, file systems, sockets
- **Was ist ein Betriebssystem?**
  Erweiterte Maschine (Abstraktion zur Verringerung der Komplexität der Hardware, Schnittstellen und APIs für Applikationen), Ressourcenmanager (Aufteilen und verwalten von Ressourcen über Raum und Zeit)
- **Was passiert beim Login?**
  Anmeldung am System, Authentifizierung des Benutzers (Benutzer ist anwesend, Sitzung beginnt, erstes Programm (shell) startet)
- **Was sind schlechte Passwörter?**
  keine, kurze, leicht zu erratende, Namen, rückwärts geschrieben Wörter
- **Wie findet man fremde Passwörter heraus?**
  Zuschauen, Keylogger, Login fälschen, social engineering, ausprobieren
- **Was ist ein Prozess?**
  Ablauf eines Programms, Speicherabbild des Programms, Speicher für Programmdaten, Kontext (OS, CPU, Ressourcen)
- **Welche Prozesszustände gibt es in UNIX?**
  created, ready, running, sleep, dead
- **Prozesshierarchien sichtbar durch pstree**
- **Was ist ein Dämon?**
  Programm, das im Hintergrund läuft und Dienste zur Verfügung stellt
  entkoppelt vom aufrufenden Prozess (parent ist Hauptprozess init)
  zB.: httpd, crond, cupsd, sshd
- **Was ist ein Zombie?**
  Prozess, der beendet, aber noch im System registriert ist (solange bis parent den Status abfragt)
- **Wo existieren Prozesse?**
  Kernel space (process descriptor) und user space (Datensegment, Programmcode)
- **Was ist eine Taskliste?**
  Kernel hat eine Liste von aktiven Prozessen, Ausgabe mit “ps”

- **Wie werden Prozesse verwaltet?**
  Prozesstabelle in Linux besteht aus task_struct, 1 Eintrag pro laufendem Prozess
- **Wie können Programme in UNIX gestartet werden?**
  - indirekt über die Shell:
    - Programmname eingeben (User)
    - neuen Prozess registrieren (Shell)
    - Programmcode laden (Kernel)
    - Scheduler übergibt Prozessor an neuen Prozess (Kernel)
  - direkt über system call (z.B. execve)
- **Welche Rechte und Ressourcen hat ein neuer Prozess?**
  Er erbt sie vom aufrufenden Prozess
- **Was ist ein Scheduler und wie arbeitet er?**
  Wählt nächsten Prozess, wenn sich mehrere die CPU teilen müssen, arbeitet nach dem Scheduling Algorithmus. Ziel: Rechenzeit der CPU optimal nutzen
- **Was sind Vorder- und Hintergrundprozesse?**
  Vordergrundprozess blockiert Eingabe an der Shell, Hintergrundprozess läuft transparent für den Benutzer. Hintergrundprozess kann mit “&” abgesetzt werden. Bezieht sich nur auf die Shell, für den Scheduler sind alle Prozesse gleich. Alle von der Shell aufgerufenen Prozesse sind in deren jobs-Liste
- **Was sind Terminals?**
  Kernel hat viele pyhsische und virtuelle Terminals, jeder Prozess ist über Pipes mit einem Terminal lokal (ttyX) oder remote (ptsX) verbunden.
- **Welche Standard Ein- und Ausgaben gibt es?**
  - Standardeingabe (stdin, 0), mit Tastatur verbunden, umgeleitet durch “<”
  - Standardausgabe (stdout, 1), mit Bildschirm verbunden, umgeleitet durch “>”
  - Standarderror (stderr, 2), mit Bildschirm verbunden, umgeleitet durch “2>”
- **Was sind unnamed pipes?**
  “|”, verbinden Prozesse mit einem unidirektionalen FIFO Kanal
- **Was sind Signale?**
  Asynchrone Ereignisse, bewirken Unterbrechung des Programmablaufs
- **Wie kann ein Prozess auf ein Signal reagieren?**
  ausführen einer Funktion, ignorieren des Signals, default Aktion
- **Welche Signale gibt es?**
  - SIGKILL: Prozessabbruch, kann nicht behandelt werden
  - SIGTERM: Wie SIGKILL, kann aber behandelt werden
  - SIGSEGV: unerlaubter Speicherzugriff
  - SIGPIPE: es wird in Pipe geschrieben, deren Ausgang geschlossen ist
  - SIGCHILD: Kindprozess terminiert oder gestoppt (geht an Elternprozess)
- **Welche Eigenschaften haben Filenamen?**
  max 255 Zeichen, formatfrei, baumförmig organisiert, root ist “/”, “everything is a file”
- **Was geschieht beim Mounten von Dateisystemen?**
  Zusammenfassen mehrerer Verzeichnisse. Eingbundenes System lokal oder übers Netzwerk verfügbar, oder in einem File (z.B. ISO)

- **Wozu dient das VFS?**
  Virtuelles Filesystem, abstrahiert alle verschiedenen verwendeten Filesysteme und bietet eine einheitliche Schnittstelle
- **Wie ist eine Partition strukturiert?**
  Boot Block, Super Block, Inode Liste, Daten
- **Welche Arten von Zugriffen gibt es?**
  user, group, other
  read, write, execute
  Ändern mit chmod, nur durch Eigentümer oder root möglich
- **Welche Systemvariablen gibt es?**
  $HOME: home directory
  $PATH: Suchpfad für Programme
  $PS1: prompt
  $PS2: 2. prompt
  $$: Prozessnummer für temp files
  $?: Status des letzten Kommandos
- **Welche sind die wichtigsten Shell Befehle?**
  cat, chmod, cp, grep, ln, ls, gcc, ipcs, make, mkdir, ps, pstree, rm, rmdir, sort

Einführung in C

- **Deklaration**
  Variablen haben einen Typ, muss bei der Deklaration dem Compiler bekanntgemacht werden
  Deklaration darf mehrmals vorkommen, nicht jede Deklaration ist eine Definition
- **Definition**
  Reservieren von Speicherplatz für Variablen, darf nur 1x vorkommen
  Jede Definition ist eine Deklaration
  int i; // Deklaration + Definition
  extern int j; // nur Deklaration, Definition anderswo
  int k = 21; // Deklaration + Definition + Initialisierung
- **Wie groß sind verschiedene Typen?**
  char: 1 Byte (muss nicht 8 Bit sein)
  short int: min. 16 Bit
  int: 32 oder 64 Bit
  long int: min. 32 Bit
  long long int: min. 64 Bit
  signed/unsigned: vorzeichenbehaftet/vorzeichenlos (float ist immer vorzeichenbehaftet)
  sizeof: Ermittlung des Speicherbedarfs
- **Welche Attribute können Variablen haben?**
  const - Konstante
  static - fester Speicherplatz (behält Zustand)
  volatile - kann außerhalb des Programmkontexts geändert werden

extern
signed, unsigned

- **Was sind Pointer?**
  Sie zeigen auf eine Adresse. Damit kann man Variableninhalte ansprechen, ähnlich wie über den Namen.
  Deklaration mit Typ *name. Hier wird kein Speicherbereich für den Variableninhalt alloziert, sondern nur Platz für die Adresse.
  Zugriff auf den Inhalt, auf den ein Pointer zeigt, mit * (“Dereferenzierungsoperator)
  Zugriff auf Adresse mit & (Adressoperator)
  int *a;
  a → Inhalt von a (eine Adresse)
  *a → Inhalt auf den a zeigt
  &a → Adresse von a
  int b;
  b → Inhalt von b (zugewiesener Wert)
  &b → Adresse von b
- **Was ist der Präprozessor und was macht er?**
  Erledigt einfache Ersetzungen im Code, wird vor dem kompilieren aufgerufen.
- **Wie kann Typumwandlung geschehen?**
  implizit oder explizit (casten)
  Bei Casten auf einen kleineren Typ werden die obersten Bits abgeschnitten
- **Was ist void und wofür wird es verwendet?**
  “leerer Datentyp”
  Ergebnistyp von Funktionen, Kennzeichnen von Zeigern, denen kein Typ zugeordnet ist, leere Parameterlisten von Funktionen
- **Was sind Structs?**
  Fassen Variablen zu einer logischen Einheit zusammen, Größe = Summe der Elemente plus Platz für Alignment
  Pointer auf Structs werden mit “->” dereferenziert
- **Was sind Unions?**
  Keine Ahnung, versteh ich nicht
- **Was sind enums?**
  sprechende Alias-Namen
- **Wie kann man einem vorhandenen Typ eine neue Bezeichnung zuweisen?**
  typedef
- **Warum ist es sinnvoll, Elemente in Structs der Größe nach zu ordnen?**
  Weil manche Compiler zur Platzoptimierung umordnen, manche nicht. Wenn dann beide Programme auf eine shmem zugreifen, kann das zu Problemen führen.
- **Was ist der Unterschied zwischen () und (void) als Funktionsparameter?**
  int foo() nimmt eine unbestimmte Anzahl an Parametern
  int foo(void) nimmt keine Parameter
- **Was bedeutet inline bei Funktionen?**
  Funktion soll so schnell wie möglich ausgeführt wrden. Implementierung ist nicht

vorgeschrieben, meistens wird der Code vom Compiler direkt hineinkopiert statt dem Funktionsaufruf

- **Wie verhalten sich Werteparameter und Variablenparameter?**
  Werteparameter sind lokal und werden nicht verändert, bei Variablenparametern wird ein Pointer übergeben. Der Inhalt, auf den der Pointer zeigt, kann verändert werden.
- **Funktionspointer**
  Pointer können auch auf Funktionen zeigen
- **Wozu dient modulare Programmierung?**
  Verbessert Lesbarkeit, Wiederverwendbarkeit und Wartbarkeit
- **Wie modularisiert man ein Programm?**
  Modul wird in Header (*.h) und Implementierung (*.c) geteilt. Header enthalten Prototypen und Konstanten, keine Definitionen von Funktionen (Implementierung in c-File)
- **Was bedeutet static bei Funktionen?**
  Funktion ist nur innerhalb ihrer c-Datei sichtbar
- **Wie können Module eingebunden werden?**
  #include ...

Traps & Pitfalls

- **Unterschiede zwischen =, ==, &, &&, |, ||, “”, ‘’**
  Sollt klar sein.
- **/***
  Kann Division durch den wert einens pointers sein (x = a/*b)
  Kann Beginn eines Kommentars sein /* blablabla */
- **a =- 1;**
  Alte Kompiler interpretieren dies als a = a - 1;
- **Was ist ein Token?**
  Ein atomares Parse - Element (Ein kleinster Ausdruck, z. B. a = 1, aber nicht p->a)
- **Wie wird ein Funktionspointer deklariert?**
  <return_value>* <pointer_name>(<type_of_arg1>, <type_of_arg2, ...);
- **Wie wird eine Funktion aufgerufen, deren Adresse auf dem Speicherwert x liegt?**
  (*(void(*)())x)();
- **Wie ist die Auswertereihenfolge von if (flag & asdf != qwer) {...}?**
  if (flag & (asdf != qwer)) {...}
- **Wie ist die Auswertereihenfolge von r = h<<4 + 1;?**
  r = h << 5;
- **Was passiert wenn gleich nach einem If ein ; kommt (if (blablabla);) ?**
  Falls das if einen Body hat wird dieser auf keinen Fall ausgeführt
- **Was passiwert, wenn bei einem Switch das break; vergessen wird?**
  Der darauf folgende case-Block wird ausgeführt

- **Zu welchen if gehört das else: if(bla) if(blabla) to_blabla; else do_bla;**
  Zum zweiten, weil to_balbla; gehört zum zweiten, danach ist der if-Block fertig (weil keine {} verwendet wurden)
- **char* p = “asdf”; Was wird bei p[2] zurückgegeben?**
  d
- **int a[] = {1, 2, 3}; int* p = &a[0]; Was wird bei sizeof(a), sizeof(p), sizeof(*p) zurückgegeben?**
  Die Größe des Arrays a (3*sizeof(int)), die größe des pointers p (sizeof(int*)), die größe eines int (sizeof(int))
- **Für welche Operatoren ist die Auswertereihenfolge vorgegeben?**
  “&&”, “||”, “?:”, “,” (Ich nehme an für +, -, /, * auch aber davon stand nix da …)
- **Auf welche Stelle eines Pointer-Arrays muss der an free(.) übergebene pointer zeigen?**
  Auf die erste, sonst kommt ein seg fault
- **Was ist der aquivalende Ausdruck zum Makro “#define SQR(x) x*x” wenn SQL(a+b) aufgerufen wird?**
  a + b * a + b = a + a*b + b

Entwicklung in C

- **Was passiert bei der Programmerstellung?**
  Übersetzen: Übersetzungseinheit → Objektdatei
  Linken: Objektdatei → ausführbare Datei
- **Was sind Makefiles?**
  Automatisieren die Programmerstellung, spezifizieren Abhängigkeiten
- **Was sind die Standardziele von Makefiles?**
  make all, make clean
- **Welche Variablen gibt es in Makefiles?**
  CC (C Compiler)
  CFLAGS (C Compiler Optionen)
  LDFLAGS (Linker Optionen)
- **Was sind manpages?**
  Sammlung von Hilfe- und Dokumentationsseiten
- **Speicherverwaltung in Linux**
  Blabla ist mir egal
- **Was ist dynamisch angeforderter Speicher? Wie funktioniert das?**
  Speicher wird zur Laufzeit angefordert und freigegeben, wird von C nicht direkt unterstützt (Bibliotheksfunktionen).
  malloc() - Anforderung von Speicher, Rückgabewert: Startadresse des Speicherbereichs
  free() - Freigeben des Speichers
  realloc() - Ändern der Größe des angeforderten Speichers

- **Wann ist eine Adresse gültig?**
  Wenn sie von malloc, realloc oder calloc zurückgegeben wurde und nicht durch free oder realloc freigegeben wurde
- **Was ist mmap?**
  Bildet Dateien, lokalen Speicher oder Shared Memory im Adressraum des aktuellen Prozesses ab. Vorteile: effizientes Lesen, Bearbeiten von Dateien, Reallokieren großer Speicherbereiche, Interprozesskommunikation mittels Shared Memory
- **Wie funktionieren Zusicherungen?**
  Vorbedinungen, Invarianten und Nachbedingungen werden dynamisch überprüft - assert()
  In C: assert.h einbinden, Programm bricht bei Verletzung ab
- **Wie kann ein Programm analysiert werden?**
  statisch (Code anschauen, Compilerwarnungen, …), z.B. mit splint
  dynamisch (während Ablauf prüfen, z.B. mit dmalloc oder valgrind
  debugging (Debugnachrichten oder interaktiv mit Debugger, z.B. gdb)
- **Was sind die Aufgaben eines Debuggers?**
  Prozess an bestimmten Punkten aufhalten (Breakpoints), Speicher und Prozessorregister inspizieren, Informationen aufbereiten, Register und RAM manipulieren

Erstes Übungsbeispiel

- **Wie können Argumente an main übergeben und ausgelesen werden?**
  int main(int argc, char **argv);
  argc → Anzahl der Elemente von argv
  argv → Array der Kommandozeilenparameter. Erstes Element ist Programmname
  Optionsbehandlung mit getopt()
- **Was ist ein Dateisystem unter Linux?**
  Hierarchische Struktur von Dateien, “everything is a file”, z.B. gewöhnliche Dateien, Verzeichnisse, Geräte, Named Pipes, Sockets, Verweise
- **Welche Elemente kommen in der primären Hierarchie vor?**
  bin: wesentliche Befehle für Benutzer
  etc: Konfigurationsdateien
  dev: Gerätedateien
  lib: wesentliche Bibliotheken
  home: Benutzerverzeichnisse
  media: Einhängepunkte für Wechselmedien
  opt: Zusatzsoftware
  proc: Virtuelles Dateisystem
  sbin: wesentliche Systemkommandos
  usr: zweite Hierarchie für rechnerspezifische Dateien
  var: Hierarchie für veränderliche Datendateien
- **Was sind File Descriptors?**
  Verweis auf Index in der Tabelle offener Dateien (file descriptor table)

- **Was ist der Unterschied zwischen fopen und fdopen?**
  fopen: Datei wird geöffnet und mit Stream verbunden
  fdopen: assoziiert Strem mit einem file descriptor
- **Wie hängen fflush und fclose zusammen?**
  fclose ruft fflush auf, fflush erzwingt das Schreiben gepufferter Daten, fclose schließt Stream und Deskriptor
- **Welche sind die gängigsten I/O-Operationen?**
  fread, fgets, fgetc, fwrite, fputs, fputc, fprintf, fseek, ferror, feof
- **Signale**
  Siehe weiter oben
- **Wie können Signale behandelt werden und was ist dabei zu beachten?**
  Ereignis generiert Signal
  Signal ist pending
  Signal wird delivered (kann über Signalmaske des Prozesses blockiert werden)
  Standardaktionen: ignorieren, terminieren, core dump (Speicherabbild), anhalten, fortsetzen
  sigaction() konfiguriert Reaktion des Programmes, SIGKILL und SIGSTOP können nicht ignoriert werden
  Signalhandler sollten so einfach wie möglich sein, nicht alle Funktionen sind zugelassen (z.B. kein malloc, printf, exit). Der Handler kann wiederum durch ein Signal unterbrochen werden → Blockieren von Signalen über Maske)
- **Was sind Sockets?**
  Kommunikationsmechanismus, Kommunikationssendepunkt
  Client und Server kommunizieren durch I/O auf File Descriptor, der dem Socket zugeordnet wird
  Schnittschnelle zur Tansportschicht eines Kommunikationsprotokolls
- **Wie sehen die Funktionsaufrufe im Detail aus?**

Server	Client
socket() Erstellt Endpoint	socket()
bind() ordnet Socket Adresse zu	
listen() hört auf Verbindungen	
accept() blockert bis zum Verbindungsaufbau	
	connect()
	send()
recv()	
send()	
	recv()...

Zweites Übungsbeispiel

- **Welche Eigenschaften haben Prozesse in Linux?**
  Zustand, Scheduling, Identifikation, Speicherverwaltung, Signale, Prozessverwandschaften, Process Control Block, Kernelstack, Dateidesktriptorentabelle, Berechtigungen, Accounting Informationen, Timerverwaltung,

Interprozesskommunikation
Jeder Prozess hat einen Elternprozess (außer init), jeder Prozess hat eine eigene ID

- **Welche Möglichkeiten gibt es, einen Prozess zu erstellen?**
  fork oder clone
- **Welche wichtigen Funktionen gibt es im Zusammenhang mit Prozessen?**
  fork → erzeugt neuen Prozess
  exec → ersetzt image eines Prozesses durch ein neues Programm
  exit → beendet Prozess
  wait → wartet auf Terminierung des Kindprozesses
- **Welchen Rückgabewert kann fork() haben?**
  -1 im Fehlerfall, 0 im Kindprozess, sonst irgendwas > 0
- **Welche Arten von exec() gibt es?**
  execl, execlp, execle (environment kann verändert werden), execv, execvp, fexecve (akzeptiert file descriptor)
  p → sucht in $PATH nach Programmname
  l → variable Argumentanzahl
  v → Argumentarray (?)
- **Wie entstehen Zombies?**
  Kindprozess terminiert, Vaterprozess hat wait noch nicht ausgeführt. Kindprozess wird auf Zustand “Zombie” gesetzt, bis Vaterprozess wait ausführt
- **Wie entstehen Orphans?**
  Vaterprozess terminiert, Kindprozess wurde noch nicht beendet → Kindprozess verwaist und wird dem init-Prozess vererbt. init entfernt den Prozesseintrag nach Beendigung des Orphans.
- **Wie hängen wait und waitpid zusammen?**
  Wait wartet auf Kindprozess, waitpid wartet auf Prozess mit ID pid.
- **Auf welche Arten kann festgestellt werden, ob ein Kindprozess beendet wurde?**
  synchron: wait, waitpid
  asynchron: SIGCHLD wird an den Elternprozess gesendet, wenn das Kind beendet wurde, kann mit einem Signal Handler behandelt werden
- **Was sind Pipes und welche Eigenschaften haben sie?**
  Kommunikationskanale zwischen verwandten Prozessen
  unidirektional (für I/O braucht man 2 Pipes), Stream von Daten, implizite Synchronisation
- **Wie werden Pipes deklariert und verwendet?**
  Eine Pipe ist ein Array mit Größe 2. Deskriptor pipe[0] ist Leseende, pipe[1] ist Schreibende. Nicht verwendete Enden müssen geschlossen werden.
  pipe() → öffnet Pipe, nimmt Array als Parameter
  fdopen() → erstellt einen Stream mit Filedeskriptor (FILE *)
  dup2(old, new) → lenkt Streams um
- **Wie synchronisieren Pipes?**
  Lesen von leerer Pipe und schreiben auf volle Pipe blockiert
  Lesen von Pipe ohne offene Schreibenden liefert EOF
  Schreiben auf Pipe ohne offene Leseenden liefert SIGPIPE

Drittes Übungsbeispiel

- **Wie wird flüchtiger Speicher unter Linux verwaltet?**
  In einem Filesystem Namens tmpfs
- **Wie verhält sich tmpfs?**
  Wie eine normales Dateisystem (swapping, Zugriffsrechte, …)
- **Wie werden Prozesse synchronisiert?**
  Über semaphoren.
- **Was ist ein Eventcounter?**
  Eine Ganzzahlige Variable zu Zählen von Vorkommnissen von Ereignissen. Der Anfangswert ist 0.
- **Wie können Eventcounter angelegt, gelesen und gelöscht werden?**
  eventcounter_t* create_eventcounter(key_t key);
  long earead(eventcounter_t* evc);
  int rm_eventcounter(eventcounter_t* evc);
- **Wie werden Eventcounter verwendet? (Keine Ahnung ob die Antwort stimmt)**
  eawait(<event_counter_variable>, <irgendein_long>): Das Programm wird erst weiter abgearbeitet, wenn <irgendein_long> >= <event_counter_variable> ist. Also wenn z. B. der ec = 5 ist und das long 7 müssen 2 Events eintreten (also der Eventcounter muss zwei mal erhöt werden) damit das Programm weiter ausgeführt wird.
  eadvance(eventcounter_t *evc): Der Eventcounter wird um 1 erhöht.
  Damit kann z. b. ein Abwechselndes arbeiten implementiert werden, z. B. der Prozess Aarbeitet bei allen gerade Eventcounter-Werten, der Prozess B bei allen ungeraden. der ec ist am Anfang null, Prozess sagt eawait(ec, 0) macht was sagt eadvance(ec) und sagt dann eawait(ec, 2). Nach dem eadvance(ec) kann der Prozess B anfangen, weil der mit eadvance(ec, 1) beginnt, macht dann seine sachen, ruft eadvance(ec) auf und macht dann eawait(ec,3); usw …
- **Was ist ein Sequencer?**
  Eine ganzzahlige Variable mit Anfangswert 0 die verwendet wird um die Abfolge von Ereignissen zu entscheiden.
- **Wie können Sequencer angelegt, gelöscht und abgeholt werden?**
  sequencer_t* create_sequencer(key_t key);
  int rm_sequencer(sequencer_t* sequencer);
  long sticket(sequencer_t* sequencer);
- **Wie wird ein Sequencer angewendet?**
  Das “Ticketabholen” ist nichts anderes als das zurückgeben und erhöhen einer Zählvariable. Dies ist aber ein atomares Ereignis, d. h. es bekommt tatsächlich jeder Aufrufende Prozess einen anderen Wert. Somit kann das Zeugs mit den Eventcountern sicherer gelöst werden.

Linux Kernel Modul

- **Was ist ein Kernel Modul und wie ist es aufgebaut?**
  erweitert den Kernel um zusätzliche Funktionalität, Module können zur Laufzeit entwickelt, geladen und entfernt werden. Gut geeignet für Gerätetreiber. Module können aufeinander aufbauern.
  Ein Modul ist immer serviceorientiert: nach Einhängen nur noch Bearbeitung von Requests.
- **Was ist bei der Entwicklung zu beachten?**
  Keine externen Bibliotheken, viele Funktionen können aber trotzdem verwendet werden (linux/string.h etc.)
  Gleichzeitigkeit: Sequentieller Code kann immer unterbrochen werden, schon beim Laden
  Manuelle Freigabe von Ressourcen notwendig
  Auf Portabilität achten
  Alles, was im Userspace einfacher entwickelt werden kann, soll dort entwickelt werden
- **Was sind Device Drivers?**
  Gerät wird abstrahiert, File unter /dev außer Netzwerkgeräte. Der Treiber soll möglichst komplette Hardwarefunktionalität anbieten.
- **Welche Hauptklassen von Device Drivers gibt es?**
  Character Device (Stream), Block Device (Blocks), Network Device (Packets)
  Weiter Einteilung nach Subsystemen (USB, SCSI, Firewire, …)
- **Welche Securityaspekte sind zu beachten?**
  Der Kernel hat maximale Rechte im System. Typische Bugs: Buffer Overflow, Information Leagkage, Unchecked User Input
- **Wie ist der Linux Kernel lizenziert?**
  General Public License Version 2, deklariert durch MODULE_LICENSE Makro
  Proprietäre Module werden toleriert, erhalten aber nicht die komplette Kernel API
- **Was enthalten Kernel Quellen?**
  Den Kernel selbst, Build System basierend auf Unix Make, Interfaces für User-Space Applikationen, Kernel Module (Header Files), zahlreiche Architekturen und Device Driver
- **Wie können eigene Kernel Module kompiliert werden?**
  “two-faced” Makefile: Bei Ausführung im Modulquellverzeichnis Wechsel in Kernel Source Directory und Ausführung des eigentlichen Kernel Makefiles um das Modul zu übersetzen. Bei Kernel-Build Einbindung: wird zu Makefilefragment, das die gesamte Compile-Info für Kernel-Build Vorgang enthält
- **Wie können Kernel Module geladen und entfernt werden?**
  insmod und rmmod als root zur Laufzeit
  lsmod: Liste der geladenen Module
- **Was sind Character Devices?**
  Zeichenorientierte Geräte, die sich wie Files verhalten (Maus, Keyboard, Serial, …), implementieren meist open, close, read, write. Oft nur sequentiell lesbar und schreibbar

- **Was ist bei File Operationen zu beachten?**
  Device Treiber kann Defaultverhalten überschreiben, Funktionen werden bei den jeweiligen Operationen auf das Character Device aufgerufen.
  Ausführung im Kernelspace, im Kontext des aufrufenden Userspace Prozesses
  struct module *owner muss gesetzt werden, damit das Modul nicht aus dem System entfernt werden kann, solange Operationen ausgeführt werden
- **Wie kann synchronisiert werden?**
  Semaphoren, Spinlocks (deaktiviert Interrupts, kann in Interruptbehandlungsroutinen verwendet werden)
- **Wie funktioniert die Speicherverwaltung unter Linux?**
  sehr flexibel: page-oriented, vmalloc, caches, pools, …
  meistens geeignet: knalloc und kfree (nicht automatisch initialisiert und gelöscht)
- **Wie funktioniert der Datenaustausch zwischen Kernelspace und Userspace?**
  Bufferinhalte müssen manchmal transferiert werden → niemals ungeprüfte Pointer im Kernel dereferenzieren!
  asm/uaccess.h nutzen: wie memcpy, aber mit Prüfung des Userspacepointers, Rückgabewert ist Anzahl der nicht kopierten Bytes (Fehler wenn nicht 0)
- **Wie sieht der Lifecycle eines Character Devices aus?**
  Registrierung/Allozierung der Character Device Region
  Initialisierung der Character Device Struktur
  Registrierung des Character Devices im Kernel (ab hier kann das Device verwendet werden, auch wenn die module_init Funktion noch nicht fertig ist
  Deregistrierung des Character Devices
  Deregistrierung der Character Device Region

Ressourcenverwaltung

- **Wie können Ressourcen gelöscht werden?**
  Unsynchronisiert: Fehlerhafter PRozess löscht Ressourcen
  Synchronisiert: Eigener Kommunikationskanal
- **Warum könnten synchronisiertes Löschen verwendet werden?**
  Damit sichergestellt wird, dass kein weiterer Prozess als der Löschende auf die Ressourcen zugreift.