Study-Guide: Kernel

Dieses Thema enthält zwei Kapitel, die sich um den Umgang mit dem Kernel selbst drehen. Dabei ist der Umgang mit Kernelmodulen während der Laufzeit und das Erstellen eines eigenen Kernels das Lernziel. Weitergehende Techniken, wie etwa das Patchen eines Kernels wird erst für LPIC Level 2 interessant.Verwalten/Abfragen von Kernel und Kernelmodulen zur Laufzeit Konfiguration, Erstellung und Installation eines angepaßten Kernels und seiner Module

Seite:	-= LinuxLernSystem =- (http://www.lpi-test.de)
Kurs:	LPIC-1 [102]
Buch:	Study-Guide: Kernel
Gedruckt von:	André Scholz
Datum:	Dienstag, 1 November 2005, 10:34 Uhr

Inhaltsverzeichnis

1.105 - Kernel

Dieses Thema enthält zwei Kapitel, die sich um den Umgang mit dem Kernel selbst drehen. Dabei ist der Umgang mit Kernelmodulen während der Laufzeit und das Erstellen eines eigenen Kernels das Lernziel. Weitergehende Techniken, wie etwa das Patchen eines Kernels wird erst für LPIC Level 2 interessant.

- Verwalten/Abfragen von Kernel und Kernelmodulen zur Laufzeit
- Konfiguration, Erstellung und Installation eines angepaßten Kernels und seiner Module

1.105.1 - Verwalten/Abfragen von Kernel und Kernelmodulen zur Laufzeit

Beschreibung: Prüfungskandidaten sollten in der Lage sein, Kernel und ladbare Kernelmodule zu verwalten und/oder abzufragen. Dieses Lernziel beinhaltet die Verwendung von Kommandozeilen-Utilites, um Informationen über den gerade laufenden Kernel und die Kernelmodule zu erhalten. Ebenfalls enthalten ist das manuelle Laden und Entladen von Modulen nach Bedarf. Dies beinhaltet auch die Bestimmung, wann Module entladen werden können und welche Parameter ein Modul akzeptiert. Prüfungskandidaten sollten in der Lage sein, das System so zu konfigurieren, daß Module auch durch andere als ihre Dateinamen geladen werden können.

Die wichtigsten Dateien, Bezeichnungen und Anwendungen:

- `/lib/modules/`*`kernel-version`*`/modules.dep`
- `/etc/modules.conf & /etc/conf.modules`
- **depmod**
- **insmod**
- **lsmod**
- **rmmod**
- **modinfo**
- **modprobe**
- **uname**

Kernelmodule

Der Linux-Kernel war anfangs (in der Version 1.x.x) ein rein monolithisches Gebilde. Das hatte zur Folge, daß jede Hardware-Unterstützung fest in den Kernel eingebunden werden musste. So entstanden entweder sogenannte Allround-Kernel, die riesengroß waren und Unterstützung für die abenteuerlichste Hardware zur Verügung stellten, oder es musste für jede neue Hardware, die in einen Rechner eingebaut wurde, ein neuer Kernel erstellt werden. Beides war keine praxistaugliche Lösung.

Aus diesem Grund wurde der Kernel modularisiert. Das heißt, daß bestimmte Teile des Kernels nicht mehr beim Compilieren fest eingebunden werden müssen, sondern später, während der Laufzeit, ge- und auf Wunsch auch wieder entladen werden können. Diese Fähigkeit ermöglicht es, daß Gerätetreiber eben nicht immer fest eingebunden werden müssen, sondern je nach Bedarf nachgeladen werden können. Distributoren können so schlanke Kernel ausliefern, die trotzdem alle Fähigkeiten von Linux unterstützen, weil die jeweiligen Fähigkeiten bei Bedarf geladen werden können. Die Teile des compilierten Kernels, die nicht fest eingebunden sind, sondern ge- und entladen werden können, werden *Kernelmodule* genannt.

Die Kernelmodule des Linux-Kernels werden im Dateisystem unter `/lib/modules/`*`Kernelversion`* abgespeichert, wobei *Kernelversion* für die Versionsnummer des Kernels steht, für den die Module passen. Die Module bestehen aus Objektdateien (Endung `.o`), die in diesem Verzeichnis in diverse Unterverzeichnisse sortiert sind.

Kernelmodule sind Teile des Kernels, zwar ausgelagert, aber doch Teile des Ganzen. Wenn solche Module ein- und ausgehängt werden, so spielen bestimmte Abhängigkeiten (dependencies) eine Rolle. Beispielsweise kann der Kernel nichts mit einem Modul anfangen, das ihm die Fähigkeit verleihen soll, mit SCSI-CDROMs umzugehen, wenn er gar nicht weiß, was SCSI ist. Das Modul für die SCSI-CDROMs ist also abhängig vom generellen Modul für SCSI. Diese Abhängigkeiten müssen den Programmen bekannt sein, die die Module laden und entladen. Aus diesem Grund existiert das Programm **depmod**, das die Abhängigkeiten aller Module untersucht und die Ergebnisse in die Datei `/lib/modules/`*`Kernelversion`*`/modules.dep` schreibt. In dieser Datei stehen dann - in klar lesbarer Form - die entsprechenden Abhängigkeiten jedes einzelnen Moduls. Jedesmal, wenn neue Module ins Modulverzeichnis kopiert werden, muß anschließend dieses Programm aufgerufen werden.

Programme zum Umgang mit den Modulen

Es gibt jetzt eine Reihe von Befehlen, die es ermöglichen mit Kernelmodulen umzugehen. Diese Befehle werden jetzt der Reihe nach kurz beschrieben.

Auflistung der geladenen Module

Um herauszufinden, welche Module bereits geladen sind, existiert der Befehl **lsmod**. Er zeigt alle geladenen Module im Format Name, Größe, Wie-oft-gebraucht und Liste der Module, die das Modul benötigen. Dieser Befehl dient also hauptsächlich dazu, zu überprüfen, ob ein bestimmtes Modul geladen ist oder nicht. **lsmod** wird ohne Parameter aufgerufen.

Laden von einzelnen Modulen

Um ein einzelnes Modul zu laden, kann der Befehl **insmod** verwendet werden. Als Parameter erwartet **insmod** den Namen des zu ladenden Moduls. Der Name wird **ohne** die Endung `.o` angegeben. Auch Pfade werden keine benötigt. Das Programm weiß ja selbst, in welchem Verzeichnis die ladbaren Module des aktuellen Kernels zu finden sind. Um also beispielsweise das Modul `/lib/modules/`*`Kernelversion`*`/kernel/drivers/usb/printer.o` zu laden genügt die Angabe

```
insmod printer
```

Anhand der aktuellen Kernelversionsnummer ermittelt **insmod** das notwendige Basisverzeichnis für die Module und kann dort über die Datei `modules.dep` den genauen Pfad zum gewünschten Modul ermitteln.

Wenn ein Modul selbst noch Parameter erwartet, z.B. die I/O-Adresse einer ISA-Netzwerkkarte, so werden sie nach dem Modulnamen angegeben. Diese Parameter haben immer die Form

Symbolname=Wert

Für die ISA-Netzwerkkarte könnte das z.B. der Befehl

```
insmod ne io=0x300 irq=5
```

sein. Damit wird das Modul `ne` geladen (ein Modul für eine NE2000 Netzwerkkarte) und bekommt als Parameter die Werte für IO-Adresse und IRQ übergeben.

Das Programm **insmod** läd nur das angegebene Modul. Wenn dieses Modul andere Module benötigt, so werden sie **nicht** mitgeladen.

Laden eines Moduls mit allen nötigen Zusatzmodulen

Damit man auch Module laden kann, und automatisch alle notwendigen Zusatzmodule mitgeladen werden, existiert der Befehl **modprobe**. Er hat heute den Befehl **insmod** völlig verdrängt, denn er ist wesentlich intelligenter. Die Anwendung entspricht der von **insmod**. Als Parameter wird das zu ladenden Modul, wiederum ohne Pfad und ohne der Endung `.o` angegeben. Auch eventuell notwendige Parameter werden wie bei **insmod** angegeben.

Im Gegensatz zu **insmod** überprüft **modprobe** die Abhängigkeiten der Module untereinander (über die Informationen aus `modules.dep` und läd alle Module, die das angegebene Modul benötigt, um richtig zu funktionieren. Ein Beispiel:

Wenn wir das Modul für ein am Parallelport angeschlossenes Zip-Laufwerk (ppa) laden wollen, so benötigt dieses Modul, das generellen Zugriff auf Parallelports ermöglicht (parport). es reicht zu schreiben

```
modprobe ppa
```

und **modprobe** erkennt, daß eine nicht erfüllte Abhängigkeit vorliegt, läd zuerst das Modul `parport` und erst danach das Modul `ppa`.

Anstatt Parameter für bestimmte Module direkt anzugeben, unterstützt **modprobe** (nicht **insmod**) eine Datei, die die notwendigen Modulparameter enthält. Diese Datei heißt entweder `/etc/conf.modules` (alt) oder `modules.conf`

(aktuell) und wird weiter unten noch genauer beschrieben.

Um genau zu sein, benutzt **modprobe** das Programm **insmod** um die notwendigen Module zu laden. **modprobe** ist also eine Art High-Level Programm zum Laden der Module oder ein Frontend für **insmod**.

Entfernen von geladenen Modulen

Wenn ein Modul nicht mehr benötigt wird, so kann es mit dem Befehl **rmmod** wieder aus dem Kernel entfernt werden. **rmmod** kann keine Module entfernen, die gerade in Gebrauch sind oder die von anderen geladenen Modulen benötigt werden. Allerdings ist es möglich, mit dem Kommandozeilenparameter `-r` dafür zu sorgen, daß **rmmod** so funktioniert, wie ein umgekehrtes **modprobe**, also nicht nur das angegebene Modul entfernt, sondern alle, die nur deshalb geladen sind, damit das zu entfernende Modul funktioniert.

Auch dieses Programm erwartet den Namen des zu entfernenden Moduls ohne Endung und Pfad.

Auch das Programm **modprobe** kann verwendet werden, um Module wieder zu entfernen. Mit dem Parameter `-r` werden Module entfernt, die auf der Kommandozeile angegeben wurden.

Modulparameter erkennen und einstellen

Module sind häufig Gerätetreiber für bestimmte Hardware. In vielen Fällen ist es notwendig, für die Ansteuerung der Hardware bestimmte Parameter wie IO-Port, IRQ usw. anzugeben, wenn ein Modul geladen werden soll. Wie oben gesehen, werden Parameter für Module immer in der Form

Symbolname=Wert

angegeben. Dazu muß aber klar sein, welche Symbolnamen das jeweilige Modul kennt, also welche Parameter es versteht. Um das herauszufinden gibt es das Programm **modinfo**. Dieses Programm gibt Informationen über ein Modul aus, unter anderem welche Parameter angegeben werden können. Auch **modinfo** arbeitet mit den Modulnamen ohne Endung und Pfad. Über Kommandozeilenparameter kann angegeben werden, welche Informationen ausgegeben werden sollen, ohne solche Schalter werden die Informationen über

- Dateiname und Pfad der Moduldatei (-n)
- Beschreibung (-d)
- Author (-a)
- Lizenz (-l)
- Parameter (-p)

ausgegeben. Um z.B herauszufinden, welche Parameter das Modul 3c509 (ein Gerätetreiber für eine 3Com Netzwerkkarte) benötigt, können wir schreiben

```
modinfo -p 3c509
```

Die entsprechende Ausgabe des Programms wäre:

```
debug int, description "EtherLink III debug level (0-6)"
irq int array (min = 1, max = 8), description "EtherLink III
   IRQ number(s) (assigned)"
xcvr int array (min = 1, max = 8), description "EtherLink III
   tranceiver(s) (0=internal, 1=external)"
max_interrupt_work int, description "EtherLink III maximum events
   handled per interrupt"
nopnp int, description "EtherLink III disable ISA PnP support (0-1)"
```

Daraus können wir entnehmen, daß dieses Modul die Parameter `debug, irq, xcvr, max_interrupt_work` und `nopnp` versteht. Alle diese Parameter erwarten ganzzahlige (int) Werte. Wenn hier die Angabe `array` steht, so ist gemeint, daß wenn mehrere gleichartige Karten existieren, die alle dieses Modul benötigen, dann an Stelle der normalen Angabe einer Zahl, eine durch Komma getrennte Liste von Zahlen verwendet wird. Bei drei solcher Netzwerkkarten hieße der Parameter, der den IRQ einstellt also `irq=3,5,7`. Die erste Karte benutzt IRQ3, die zweite 5 und die dritte 7.

Die Datei /etc/modules.conf

Um den Kernelmodulen ihre Parameter an einer festen Stelle angeben zu können, wurde **modprobe** so programmiert, daß es die Datei `/etc/modules.conf` (oder `/etc/conf.modules`) abarbeitet und dort entsprechende Parameter für die Module entnimmt.

Die Datei kann auch noch mehr, als nur Parameter für Module zur Verfügung stellen. Insbesondere können damit

- Parameter für Module festgelegt werden,
- Aliasnamen für Module vergeben werden,
- Module angegeben werden, die vor oder nach dem eigentlichen Modul geladen werden sollen,
- Module, die vor oder nach dem Laden des eigentlichen Moduls entfernt werden sollen

Die Datei hat ein einfaches Format. Jede Zeile, die nicht mit einem Kommentarzeichen (#) beginnt oder leer ist, wird als Anweisung interpretiert. Um einem Modul bestimmte Parameter zu setzen, wird der Befehl **options** benutzt. Er hat die folgende Syntax:

options*Modulname Optionen*

Ein Eintrag für unsere NE2000 Netzwerkkarte, könnte also so aussehen:

```
options ne io=0x320 irq=7
```

Jedesmal, wenn jetzt mit **modprobe** das Modul `ne` geladen wird, werden automatisch diese Optionen mitgeladen.

Um ein Modul auch mit anderem Namen anzusprechen, können wir Aliasnamen vergeben. Das bedeutet, daß mit **modprobe** auch Module geladen werden, die es gar nicht gibt, sondern die nur Aliasnamen sind. Die Syntax ist

alias*Aliasname Modulname*

Diese Methode wird gerne benutzt um bestimmte Hardware an einen Begriff zu binden, der mehr aussagt und allgemeiner handhabbar ist als der Modulname. So kann z.B. der Aliasname `eth0` an die tatsächliche Netzwerkkarte gebunden werden. Der Eintrag

```
alias eth0 ne
alias eth1 off

options ne io=0x320 irq=7
```

würde es also ermöglichen, daß wir (oder ein init-script) den Befehl

```
modprobe eth0
```

eingeben und so automatisch das Modul `ne` mit den angegebenen Parametern geladen wird. Versucht aber jemand mit **modprobe** das Modul `eth1` zu laden, dann weigert sich **modprobe** irgendein Modul zu laden, da der Wert des Aliases **off** ist.

Mit pre-install können Befehle angegeben werden, die vor dem Laden des Moduls ausgeführt werden sollen. Diese Befehle können (müssen aber nicht) andere Module laden. Die Syntax lautet:

pre-install*Modul Kommando*

Bevor das Modul geladen wird, wird erst das Kommando ausgeführt. Analog dazu gibt es die Befehle

post-install*Modul Kommando*
pre-remove*Modul Kommando*
post-remove*Modul Kommando*

die entsprechend nach dem Laden, vor dem Entfernen und nach dem Entfernen bestimmter Module bestimmte Kommandos ausführen. Alle vier Kommandos können auch auf Aliasnamen angewandt werden.

Der Befehl uname

Der Befehl uname gibt Informationen über den Kernel und das System aus. Um z.B. die aktuelle Kernelversion zu erfragen, genügt ein Aufruf von

```
uname -r
```

Das ist z.B. die Methode, mit der erfragt werden kann, in welchem Verzeichnis die Module des Kernels liegen. Es ist tatsächlich möglich zu schreiben

```
cd /lib/modules/`uname -r`/kernel
```

Durch die Kommandosubstitution wird das Konstrukt `` `uname -r` `` durch die Ausgabe des Befehls ersetzt und die entspricht genau dem, was als Verzeichnisnamen vorliegt.

Alle Informationen können mit `uname -a` ausgegeben werden, die restlichen Parameter entnehmen Sie der Handbuchseite.

1.105.2 - Konfiguration, Erstellung und Installation eines angepaßten Kernels und seiner Module

Beschreibung: Prüfungskandidaten sollten in der Lage sein, einen Kernel und ladbare Kernelmodule aus Quellcode anzupassen, zu erzeugen und zu installieren. Dieses Lernziel beinhaltet das Anpassen der aktuellen Kernelkonfiguration, das Erzeugen eines neuen Kernels und das Erzeugen von Kernelmodulen nach Bedarf. Ebenfalls enthalten ist die Installation des neuen Kernels sowie jedweder Module und das Sicherstellen, daß der Bootmanager den neuen Kernel und die dazugehörigen Dateien findet (generell in `/boot`, siehe Lernziel 1.102.2 für mehr Details über Bootmanager und deren Konfiguration).

Die wichtigsten Dateien, Bezeichnungen und Anwendungen:

- `/usr/src/linux/*`
- `/usr/src/linux/.config`
- `/lib/modules/`*`kernel-version`*`/*`
- `/boot/*`
- **make**
- make targets:**config, menuconfig, xconfig, oldconfig, modules, install, modules_install, depmod**

Eine der wesentlichen Unterschiede von Linux gegenüber praktisch allen anderen Betriebssystemen ist, daß der Quellcode verfügbar ist und es so möglich ist, seinen eigenen Kernel zu compilieren. Einen Kernel, der also genau an die Bedürfnisse des eigenen Rechners angepasst ist und keinen unnötigen Ballast enthält. Seit der Einführung des modularen Kernels gibt es zwar keine wirkliche Notwendigkeit mehr, den Kernel selbst zu erzeugen, es ist jedoch immer noch in manchen Fällen besser, einen eigenen Kernel zu benutzen, anstelle eines Standard-Kernels. Ob es sich nun um besonders sicherheitsrelevante Systeme handelt, oder bestimmte Fähigkeiten eines Standard-Kernels nicht erwünscht sind, jedesmal ist es die Lösung, einen entsprechenden eigenen Kernel zu erzeugen.

Einen eigenen Kernel compilieren

Der Vorgang der Compilierung eines Kernels ist kein Kunststück, es werden keine C-Kenntnisse benötigt. Ähnlich wie beim Übersetzen von Paketen, die im Quellcode vorliegen (siehe Abschnitt 1.102.3 der LPI101 Vorbereitung) wird wieder das Programm **make** benötigt, das anhand eines Makefiles den Kernel erstellt. Im Gegensatz zu einem "normalen" Programm ist aber etwas mehr Konfigurationsarbeit nötig, bevor der Übersetzungsvorgang gestartet werden kann.

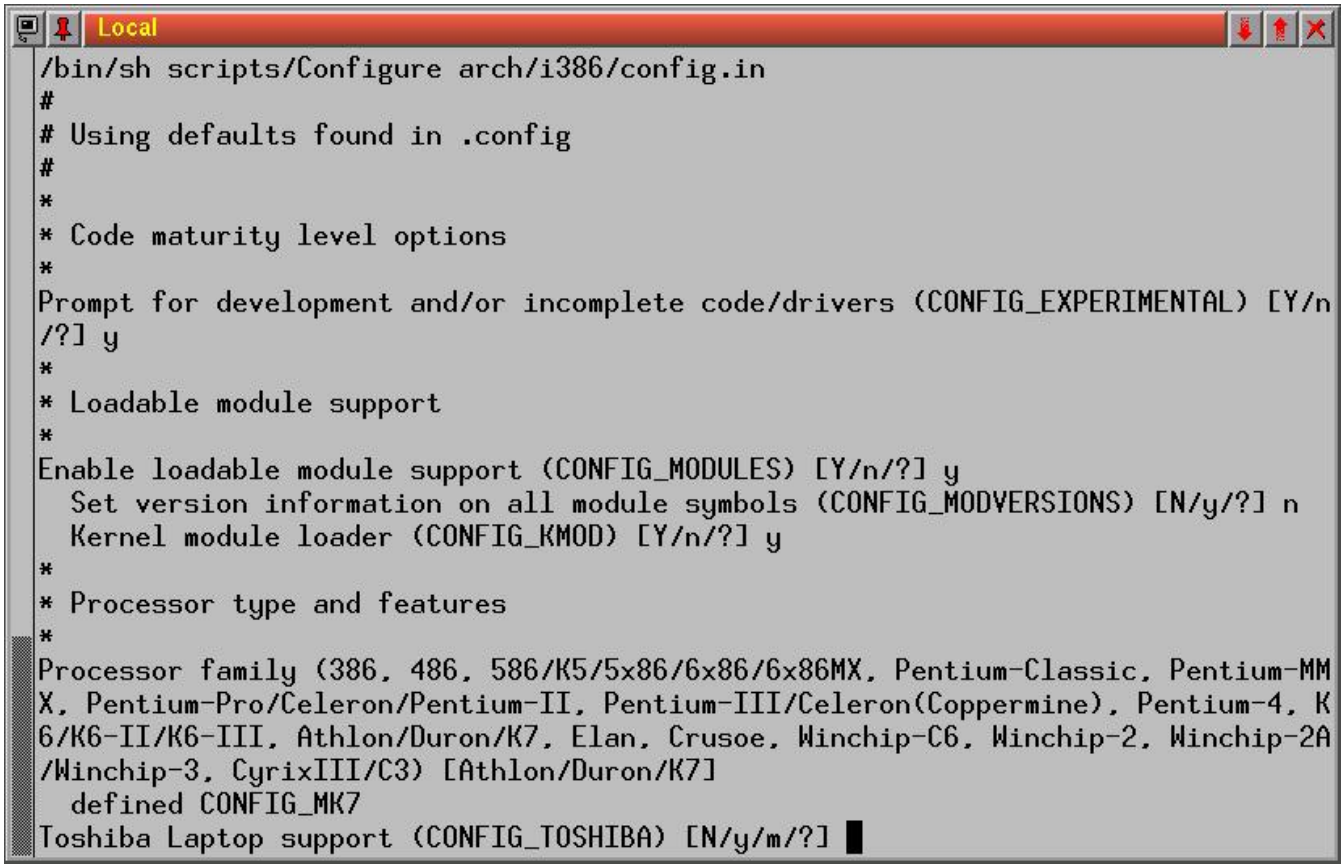
Ein Kernel besteht aus vielen hundert Quellcode-Dateien (`*.c`), die alle für bestimmte Fähigkeiten des Kernels stehen. Die wesentliche Arbeit bei der Neuerstellung eines Kernels ist es, festzulegen, welche dieser Fähigkeiten erwünscht sind, welche als Modul erstellt werden sollen und welche weggelassen werden sollen. Diese Aufgabe alleine wäre nahezu unlösbar, wenn es dafür keine speziellen Hilfsprogramme gäbe. Aber diese Hilfsprogramme existieren. Sie sind Teil des Kernel-Quellcodes selbst und werden auch über das Makefile (also mit dem Programm **make**) aufgerufen.

Konfiguration des Kernels

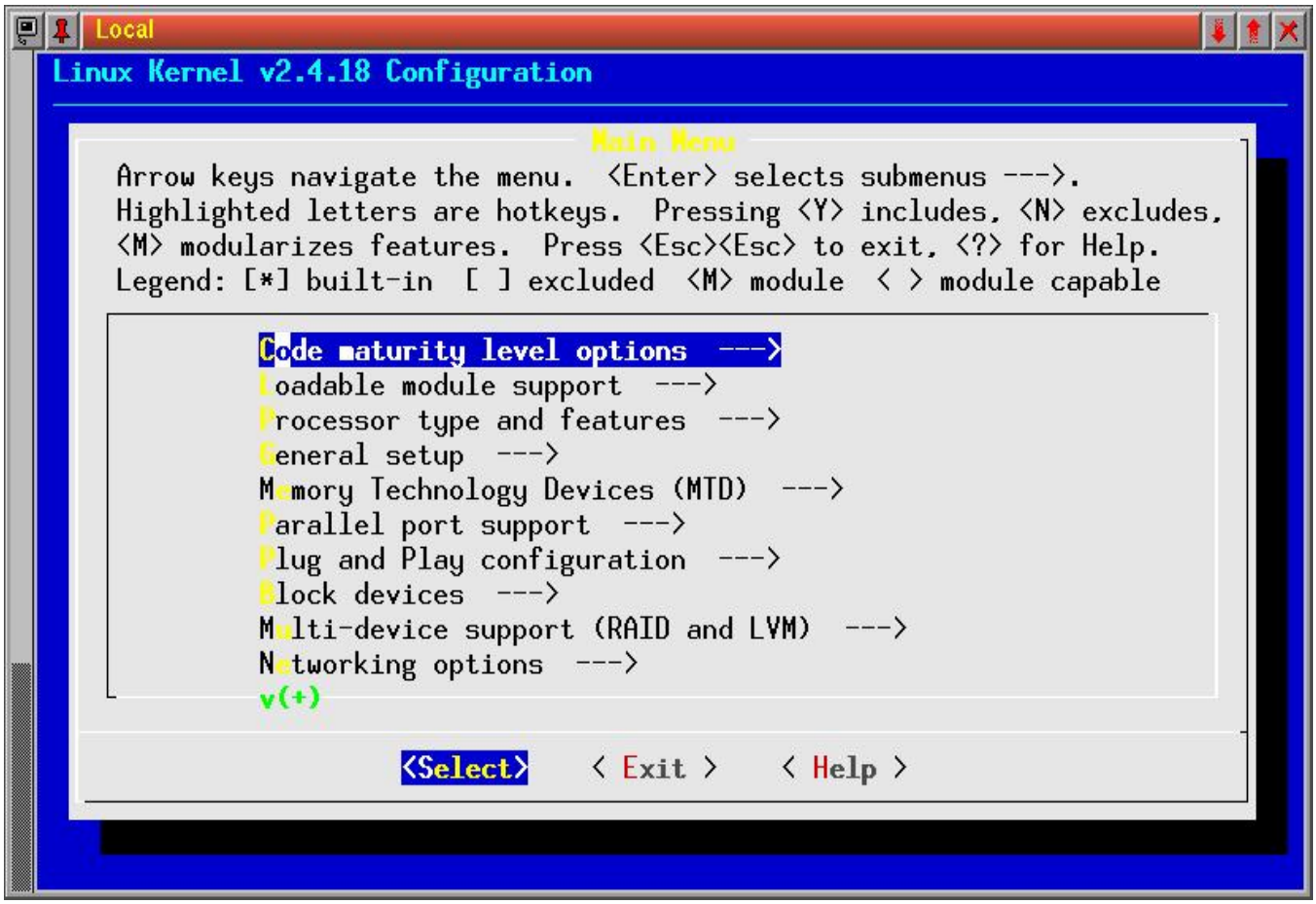
Die Quellen des Kernels werden in ein Verzeichnis unter `/usr/src` entpackt. Die Kernelversion, die compiliert werden soll, muß unter `/usr/src/linux` zu finden sein. Am einfachsten wird das über einen symbolischen Link erreicht. `/usr/src/linux` ist also ein Symlink auf das Verzeichnis, in dem wirklich die Kernelquellen liegen. Innerhalb dieses Verzeichnisses wird alle Konfigurations- und Übersetzungsarbeit geleistet.

Um jetzt den Konfigurationsvorgang zu starten, gibt es drei unterschiedliche Möglichkeiten:

- **make config**
  Startet ein sehr einfaches textorientiertes Programm, das der Reihe nach alle Einstellungen des Kernels abarbeitet und den Anwender auffordert zu entscheiden, ob das jeweilige Feature integriert, als Modul erzeugt oder weggelassen werden soll. Diese Methode ist sehr umständlich, weil immer alle Fragen von vorne bis hinten bearbeitet werden müssen. (Screenshot)

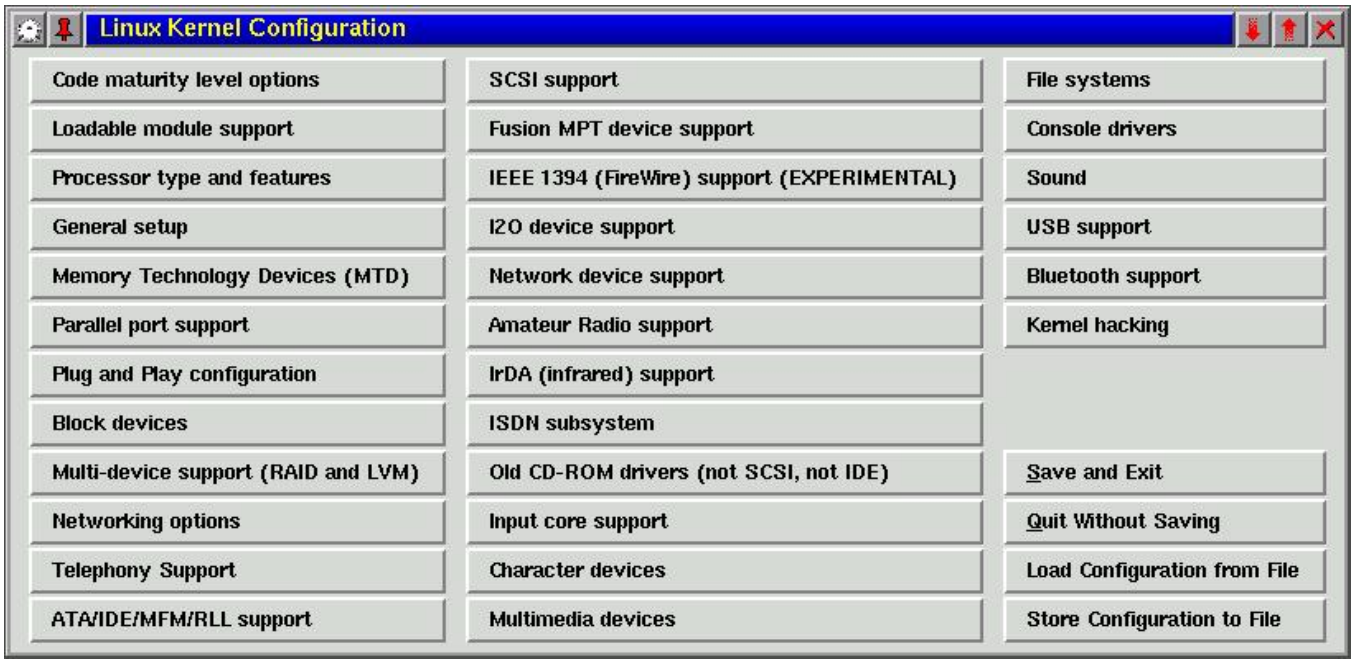


- **make menuconfig**
  Startet ein dialog-basiertes textorientiertes Programm, das auch die Beantwortung aller Fragen ermöglicht. Im Gegensatz zu **make config** erlaubt es aber das Navigieren durch alle Fragen, Wiederholung von bereits beantworteten Fragen oder nur die Veränderung einzelner Punkte. (Screenshot)



- **make xconfig**

Ein Tcl/tk basiertes graphisches Programm, das es wie **make menuconfig** erlaubt, durch die verschiedenen Fragen zu navigieren, nur eben als graphische Anwendung unter X11. (Screenshot)



Alle drei Programme tun im Prinzip das selbe. Sie ermöglichen es dem Anwender, bestimmte Fragen über die Kernelkonfiguration zu beantworten und speichern das Ergebnis in die Datei `/usr/src/linux/.config`

Diese Datei enthält die Ergebnisse der Konfiguration, also die Antworten auf alle Fragen, die während der Konfiguration beantwortet wurden. Die Form entspricht einer Shellvariablendefinition, also beispielsweise

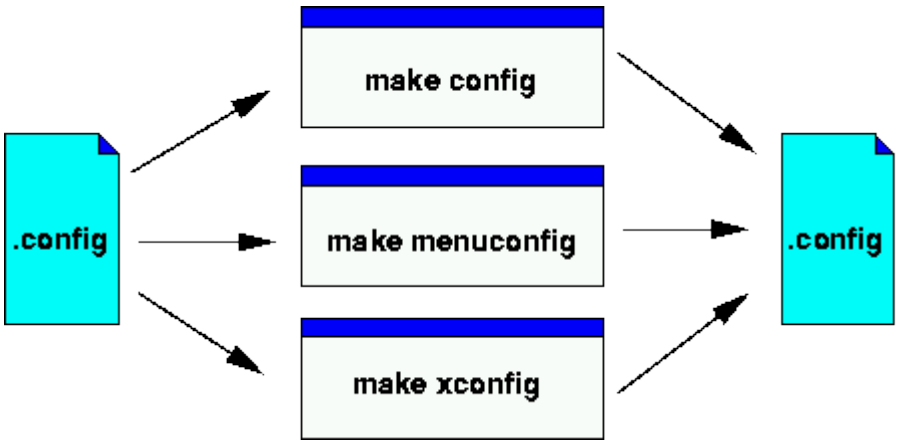
```
#
# Automatically generated make config: don't edit
#
CONFIG_X86=y
CONFIG_ISA=y
# CONFIG_SBUS is not set
CONFIG_UID16=y

#
# Code maturity level options
#
CONFIG_EXPERIMENTAL=y

#
# Loadable module support
#
CONFIG_MODULES=y
# CONFIG_MODVERSIONS is not set
CONFIG_KMOD=y

...
```

Die Datei `/usr/src/linux/.config` ist also die Datei, die die gesamte Kernelkonfiguration enthält. Es ist also möglich, verschiedene Konfigurationen herzustellen, indem einfach diese Datei umbenannt wird, um so für verschiedene Rechner jeweils einen Kernel anzupassen.

Die Beantwortung der Fragen während der Kernelkonfiguration ist ein langwieriger Prozess, bei der Version 2.4.18 sind es beispielsweise 656 Stück. Diese Fragen können hier nicht einzeln durchgesprochen werden. Viele Distributionen legen aber bereits eine `.config` Datei ihres speziellen Kernels bei, die dann benutzt werden kann, um einzelne Veränderungen zu machen.

Die Datei `.config` ist also auch die Datei, aus der die Voreinstellungen der Fragen entnommen werden.

Schematisch können wir das also folgendermaßen darstellen:



Die drei verschiedenen Konfigurationsprogramme bekommen ihre Voreinstellung aus `.config` und schreiben die veränderten Ergebnisse wieder nach `.config`.

Normalerweise bringt der Wechsel von einer Kernelversion auf eine andere natürlich auch eine neue `.config` Datei mit. In der Regel gibt es ja in neueren Versionen auch neuere Einstellmöglichkeiten und damit neue Fragen und Antworten. Um eine bestehende `.config` Datei auch in einem neuen Kernel zu benutzen, kann der Befehl

- **make oldconfig**
  Benutzt eine alte Konfigurationsdatei und stellt nur die Fragen, die dort noch nicht beantwortet wurden weil es sie noch nicht gab.

benutzt werden. Diese Form bedingt natürlich, daß zunächst die alte `.config` Datei in das neue Kernelverzeichnis kopiert wurde.

Übersetzung des Kernels

Nachdem jetzt eingestellt ist, welche Teile des Kernels fest eingebaut werden sollen, welche als Modul erstellt werden sollen und welche einfach weggelassen werden sollen, muß der Kernel anhand dieser festgestellten Regeln compiliert werden. Dieser Vorgang läuft in drei Schritten ab, die alle wieder über das Programm **make** abgewickelt werden.

Der erste Schritt ist nur bei der ersten Kernelcompilation notwendig, bei späteren Übersetzungen kann er weggelassen werden. Es ist allerdings kein Schaden, wenn er auch dann ausgeführt wird. Es geht um die Erstellung der Abhängigkeitsinformationen (dependancies). Der Befehl lautet

- **make dep**
  Erstellt die Abhängigkeitsinformationen, welche `.c` Datei welche andere benötigt.

Der zweite Schritt ist die Übersetzung des Kernels selbst. Je nach gewünschtem Kernel werden hier verschiedene Befehle benutzt.

- **make zImage**
  Die alte Anweisung, die den Kernel komprimiert in einer Datei ablegt. Bei Intel-basierten Systemen liegt der fertige Kernel dann unter `/usr/src/linux/arch/i386/boot/zImage`.
- **make zdisk**
  Compiliert den Kernel und kopiert ihn auf eine Diskette, die so zur Bootdisk wird. Auch das die alte Methode.
- **make bzImage**
  Die moderne Methode, einen großen (b wie big) Kernel herzustellen und komprimiert abzulegen. Bei Intel-basierten Systemen liegt der fertige Kernel dann unter `/usr/src/linux/arch/i386/boot/bzImage`
- **make bzdisk**
  Die neue Methode, eine Bootdisk mit großem (big) Kernel zu erstellen.

Der Vorgang des Compilierens dauert eine Weile, abhängig vom verwendeten Rechner und dessen Ressourcen. Es werden jetzt alle .c Dateien, die durch die Konfigurationsinformation angewählt wurden, in Objektdateien (.o)

compiliert die dann später zum Kernel zusammengefasst (linked) werden.

Als dritter Schritt müssen jetzt noch die Module übersetzt werden, also die Teile des Kernels, die nicht fest eingebunden werden sollen, sondern während der Laufzeit ein- und aushängbar sind. Dazu dient der Befehl

- **make modules**
  Übersetzt die ladbaren Module. Die übersetzten Module bleiben in den Verzeichnissen, in denen ihr Quellcode liegt.

Damit ist der eigentliche Compiliervorgang abgeschlossen. Der Kernel und die Module sind jetzt übersetzt, stehen dem System aber noch nicht zur Verfügung, da sie ja noch unterhalb des Verzeichnisses `/usr/src/linux` liegen.

Installieren des neuen Kernels und der Module

Damit die übersetzten Module auch dahin kommen, wo sie später erwartet werden (in das Verzeichnis `/lib/modules/`*`Kernelversion`*), wird der Befehl

- **make modules_install**
  Kopiert die Moduldateien in eine Verzeichnisstruktur unterhalb von `/lib/modules/`*`Kernelversion`*.

ausgeführt. Dieser Befehl startet dann auch gleich automatisch das Programm **depmod** um die notwendigen Modulabhängigkeiten zu analysieren und in der Datei `/lib/modules/`*`Kernelversion`*`/modules.dep` abzulegen.

Um schließlich auch den Kernel selbst zu benutzen, muß die Datei `/usr/src/linux/arch/i386/boot/bzImage`, die ja den neuen Kernel enthält, unter beliebigem Namen (etwa bzImage.neu) in das Verzeichnis `/boot` kopiert werden. Auch die Datei `/usr/src/linux/System.map` sollte dorthin kopiert werden und zwar als `/boot/System.map.`*`Kernelversion`*. Jetzt kann entsprechend der verwendete Bootmanager konfiguriert werden, damit er den neuen Kernel als Bootoption anbietet.

Bei der Installation eines neuen Kernels ist es immer ratsam, zunächst einmal den alten (funktionierenden) Kernel nicht zu überschreiben. So bleibt die Möglichkeit bestehen, den alten Kernel zu laden, falls der neue nicht erwartungsgemäß funktioniert.