Inhaltsverzeichnis

Vorwort

Aus unsere LUG wurde im Februar 2011 die Bitte an mich herangetragen, einen Vortrag über Gentoo zu halten.
Dieser Bitte bin ich sehr gerne nachgekommen, da mir die Verbreitung von Gentoo ein besonderes Anliegen ist.

Schließlich soll dies nicht nur ein weiter Vortrag ausschließlich über Gentoo und seine Eigenschaften werden. Wichtig ist mir möglichst konkret die Unterschiede und Vorteile zwischen dieser quell-basierenden Distribution und den binären Distributionen herauszustellen. Somit soll der Blick auf die wesentlichen konzeptionellen Unterschiede gelenkt werden.

Für diese Vergleiche entschloss ich mich, stellvertretend für die binären Distributionen, keinen geringeren als Debian heranzuziehen. Dabei sollen weder die eine noch die andere Distribution abgewertet werden. Stattdessen sollen die erweiterten Möglichkeiten herausgestellt werden, die sich ergeben, wenn man beim Paketbau sich der Quellen bedient und diese komfortabel in das Paketmanagement integriert.

Insofern soll dieser Beitag als Rückbesinnung auf die Wurzeln von Open-Source-Software verstanden werden und auch in Frage stellen, ob alle Entwicklungen der letzten Jahre tatsächlich in die richtige Richtung führten.

Die „Eierlegende-Wollmilchsau“ gibt es natürlich nicht und daher stellt auch Gentoo nur einen Kompromiss dar. So bleibt ein abschließende Beurteilung, ob die Vor- oder Nachteile überwiegen, jedem selbst überlassen.

Entdeckt also die Möglichkeiten ...

Entwicklung der Linux-Distributionen

Linus Torvalds veröffentlichte erstmals am 17. September 1991 seinen ersten Kernel-Quellcode.

Die ersten Linux-Distributionen erblickten im Jahre 1992 das Licht der Welt.
Seither ist es ihr primäre Ziel, den Zugang zu Linux zu erleichtern.
Daher sind sie maßgeblich am Erfolg der Verbreitung und Beliebtheit von Linux beteiligt.

Trotz dieser Bemühungen, waren in früheren Zeiten noch viele Grundlagen-Kenntnisse erforderlich, um Linux mit vollständiger Hardwareunterstützung auf einem PC installieren zu können.
Das ist wohl auch der Grund dafür, dass Linux von jeher ein gewisser Geek-Status anhaftet.

Durch die stetig verbesserte Hardware-Erkennung des Kernels mit seiner modularen Treiberunterstützung, sowie der Weiterentwicklung benutzerfreundlicher Installer und einfach zu bedienender Paketmanager, funktioniert i.d.R. das meiste "Out of the Box".

Eine Linux-Installation kann heutzutage meist schneller und einfacher als eine vergleichbare Windows-Installation ablaufen, weshalb dieser Vorgang sehr einsteigerfreundlich ist und keine besonderen Vorkenntnisse mehr erfordert.

Die Einfachheit einer Linux-Installation hat jedoch zwei Seiten.
Zwar sorgt es auf der einen Seite für eine rasche Verbreitung und zunehmende Beliebtheit von Linux.
Auf der anderen Seite begünstigt es eine breite Userbasis ohne grundlegende oder weiterführende Kenntnisse. Daraus resultiert eine geringe Kontrolle über das System, wenn der Anwender lediglich über die bereitgestellten GUIs in der Lage ist, dieses zu administrieren.

Erweiterte Kenntnisse sind jedoch unabdingbar, wenn man die Vorgaben seiner Distribution verändern oder überwinden möchte.
Abhängig von der konzeptionellen Auslegung der Distribution, stößt man dabei jedoch früher oder später an Grenzen, die es einem schwer bis unmöglich machen, gewisse Wünsche oder Änderungen umzusetzen.
Beispielsweise weil diese Vorhaben mit dem Konzept der Distribution, dem Paketmanager, oder den implementierten Automatismen kollidieren, oder sich damit einfach nicht umsetzten lassen.

Nach der Installation beschränken sich die meisten User auf die weitere Paketauswahl, oder auf die Auswahl und Konfiguration von Diensten und nehmen somit billigend, oder in Unkenntnis der erweiterten Möglichkeiten in Kauf, dass ihr System ansonsten fremdbestimmt wird.

Somit hören sie gerade dort auf, wo sich Linux am stärksten von unfreien *BS* (*Betriebs-System*) abhebt, und wo es eigentlich seinen Ursprung und seine größten Stärken hat, nämlich in der individuellen Anpassbarkeit an besondere Bedürfnisse, oder an die verwendete Hardware.

Um diese ketzerische Behauptung zu untermauern, bedarf es weitere Erläuterungen;

Um Linux einfach und universell auf möglichst vielen PCs installieren zu können, muss ein solches System recht generisch aufgebaut sein, damit es auf möglichst vielen Plattformen lauffähig ist.
Ihre Hardware-Kompatibilität wird sich daher stets am kleinsten gemeinsamen Nenner der jeweilig unterstützten Architektur orientieren.

Im Gegensatz dazu gibt es auch für spezielle Hardware oder Zielplattformen angepasste Distributionen, die genau für diesen einen Verwendungszweck optimiert wurden.
Viele davon lassen sich noch relativ leicht voneinander abgrenzen, z .B. für Server, Desktop, Router, embedded-Systems/ Receiver, Mobiltelefone, Spiele-Konsolen, etc.
Oftmals sind diese aber nicht flexibel genug ausgelegt, um abweichende Anforderungen umsetzen zu können, da sie meist nur für genau einen Anwendungszweck designt wurden.

Bei vielen Distributionen sind die Grenzen dagegen oftmals fließend. Die Wahl hängt dann oft von persönlichen Präferenzen wie z.B. der bevorzugten Desktop-Umgebung, der Programmauswahl, des Paketmanagements oder vom offiziellen Support durch den Distributor ab.

Die Anzahl der Distributionen nimmt stetig zu und macht gerade Einsteigern, wegen der Qual der Wahl, oftmals sehr zu schaffen.

So gibt es für nahezu jeden Verwendungszweck eine speziell angepasste Distribution, deren Unterschiede oftmals so gravierend sind, dass diese häufig eine erneute Einarbeitung nach sich zieht.
So unterscheiden sich nicht nur deren Arbeitsweisen, sondern auch deren zugrunde liegende Verzeichnisstrukturen und Paketformate, woraus teilweise Inkompatibilitäten resultieren.

All diesen Distributionen ist jedoch gemein, dass sie fremdbestimmt sind, denn der Nutzer hat keinen Einfluss und meist auch keine Kenntnis mit welchen Features die Pakete vom Distributor erstellt wurden. So beschränkt sich seine Wahlmöglichkeit lediglich auf deren Installation/Deinstallation sowie deren nachträgliche Konfiguration.

Selbstverständlich kann man auch bei binären Distributionen nachsehen mit welchen Optionen ihre Pakete erzeugt wurden, oder die Quellen herunterladen, um Pakete nach eigenen Wünschen von Hand zu kompilieren, jedoch erfordert es auf Anwenderseite mehr Sachkenntnis.
Außerdem entspricht es nicht der konzeptionellen Auslegung solcher Distributionen, weshalb diese Vorgehensweise meist ohne deren Unterstützung und daher manuell durchgeführt werden muss.

Sollte dabei das Paketmanagement umgangen werden, so kann dieses auch negative Auswirkungen nach sich ziehen.

Werden stattdessen erst einmal systemeigene Pakete erstellt, damit diese dann über den Paketmanager installiert werden können, so bleibt der Vorgang immer noch relativ aufwendig. Außerdem müssen dann auch die Quellen aller Abhängigkeiten des zu kompilierenden Programms vorhanden sein bzw. herunter geladen werden.

Deshalb gilt dieser Vorgang meist als archaisch, umständlich und daher wenig attraktiv. Deshalb wird er wohl von den meisten Usern gemieden oder für den Fall, dass es keinen anderen Ausweg gibt, nur für einzelne Pakete umgesetzt.

Das ist vermutlich der Grund, weshalb auf Anwenderseite so wenig Gebrauch davon gemacht wird. Zumal ein Vorteil für den User bei solchen Konzepten schwer zu erkennen ist.

Dass es aber auch komfortabel und mit einem sehr hohen Maß an Kontrolle und Flexibilität geht, zeigt u.a. das Konzept von Gentoo.
Dort ist kein manuelles **./configure && make && make install** pro Paket erforderlich, wie sich vielleicht mancher irrtümlich denkt, der schon einmal von Hand ein Paket kompiliert hat, und dann den Aufwand mit der Anzahl der Pakete multipliziert hat.
Stattdessen ist die Kompilation von Software mit allen Abhängigkeiten bei Gentoo vollständig automatisiert und für manch einen hat es etwas Magisches, dabei zuzusehen, wie sich sein persönliches Linux-System selbst aufbaut.

Linux als aufbauendes Selbststudium

Es gibt einige Linux-Projekte, denen man nachsagt, sie seien besonders gut zum Selbststudium von Linux geeignet. Besonders herauszustellen ist hier wohl *Linux From Scratch* (*LFS*).
Wer sich damit sein eigenes Linux zusammengebaut hat, dem sind vermutlich alle Grundlagen von Linux so detailliert bekannt, wie es mit kaum einem anderen Projekt der Fall wäre.

Nach dem Selbststudium hat *LFS* dann jedoch schnell ausgedient. Ein damit erstelltes System zu erweitern, umzubauen oder aktuell zu halten, möchte kaum jemand auf sich nehmen, da es zu zeitintensiv und unkomfortabel ist.

Gemessen an *LFS* ist der Umfang des Selbststudiums für Gentoo noch vergleichsweise gering. Es ermöglicht Dank der systematischen Dokumentation relativ schnell Erfolge.
So gelangt man vergleichsweise schnell zu seinem persönlichen Wunschsystem. Man kann quasi im laufenden Betrieb seine Kenntnisse bei Bedarf weiter vertiefen und darauf aufbauen.
Durch das *Rolling Release System* kann man sein System stets aktuell halten, ohne jemals wieder neu installieren zu müssen.

Entdecke die Möglichkeiten!

So wie fast jedes *GUI* nur eine Teilmenge an Funktionalität der **Kommandozeile** abbildet, so stellt auch jede Distribution nur einen Teil der Möglichkeiten von Linux seinem Nutzer in vereinfachter Form via *GUI* zur Verfügung. Das gilt gleichermaßen für den Installer.
Um sich mehr Möglichkeiten zu eröffnen, ist also meist die Benutzung der Kommandozeile erforderlich.

Vordergründig büßt man dafür einiges an Komfort ein, letztlich führt es aber zu mehr Kontrolle und somit zu mehr Flexibilität, wodurch sich u. U. viele neue Möglichkeiten eröffnen.

Jede Distribution ist doch letztlich immer ein Kompromiss zwischen **Komfort** und **Kontrolle**.

Mehr **Komfort** für den Anwender erreicht man Beispielsweise durch:
- eine Boot-CD mit bestmöglicher Hardware-Erkennung und einem intuitiven Installer.
- sinnvolle Vorgaben möglichst vieler Einstellungen, um sie dem User abzunehmen.
- viele Automatismen (z.B. bei Installation und Updates).
- Einstellmöglichkeiten für den User, die bequem über ein *GUI* zugänglich sind.

Mehr **Kontrolle** über sein System und somit mehr Möglichkeiten zur Beeinflussung, erreicht man durch das Aneignen weiterführender Kenntnisse und ihrer praktischen Anwendung.

Die Verwendung der **Kommandozeile** kann u.a. von Vorteil sein wenn:
- bei der Installation die Hardware-Erkennung fehlschlägt, oder die vorhandene Hardware nicht oder unzureichend unterstützt wird.
- der Installer das vorhandene oder gewünschte Partitionsschema oder Dateiformat nicht unterstützt, oder mir einen ungewollten Bootloader aufzwingen will.
- ein schlechter Automatismus zum Update, Installation oder Konfiguration fehlschlägt, oder eher einschränkend, statt unterstützend wirkt.
- der Paketmanager nicht die gewünschten Optionen zur Paketkonfiguration zur Verfügung stellt.
- die Distribution nicht die gewünschten Optionen zur Konfiguration oder Optimierung bereitstellt.
- zur Lösung von jeglichen Aufgaben für die es kein *GUI* gibt, oder bei der weniger manchmal mehr ist.

Gentoo

Daniel Robbins begann 1999 die Entwicklung der Gentoo Distribution unter dem Code-Namen „Enoch“. Mit der Version 1.0 fand dann die Umbenennung in **Gentoo** statt.

Bevor Daniel Robbins 2004 das Projekt verließ, gründete er die Gentoo Foundation, Inc. und ernannte deren erstes „Board of Trustees“. Hauptziele der Gentoo Foundation sind die Wahrung rechtlicher Interessen, als auch die Sicherstellung der Finanzierung. Damals kamen erste Gerüchte auf, dass dies der Untergang der Distribution sein könnte.
Aus schlecht unterrichteten Kreisen kamen Anfang 2008 ähnliche Gerüchte über den Untergang von Gentoo auf. Denn dem Projekt wurde der Stiftung-Status entzogen, weil notwendige Papiere auf dem Postweg verloren gingen und daher nicht fristgerecht eintrafen. Innerhalb eines Monats erhielt Gentoo ihren Status jedoch wieder zurück. Weitere Details über Gentoo's Historie hier unter: Gentoo Linux feiert 10. Geburtstag!

Gentoo war ursprünglich von Daniel Robbins für seine, und somit auch für die Bedürfnisse von Entwicklern konzipiert worden, um viele, der seinerzeit, bestehenden Beschränkungen aufzuheben oder möglichst komfortabel zu umgehen.
Einige Vorteile für Entwickler dabei:
Sie haben sofort ein laufendes Build-Enviroment, sowie sämtliche Quellen verfügbar, um damit sofort problemlos arbeiten zu können.
Sie können eigene Patches in lokalen Overlays entwickeln und verwalten, um diese regelmäßig und auch update-persistent, auf Quellpakete anwenden zu können.
Sie können verschiedene Compiler-Versionen parallel installieren und zwischen ihnen wechseln.
Man kann verschiedene Versionen der gleichen Programme oder Libraries parallel in sogenannten Slots installieren.
Das sind vermutlich längst nicht alle Gründe, warum auch heute noch viele Entwickler Gentoo als Plattform ihrer Wahl nutzen.

Ich bin der Meinung, was für Entwickler gut ist, kann für ambitionierte Anwender nicht schlecht sein! Selbst wenn sie sich nicht mit Programmierung beschäftigen, so können auch solche Anwender ihre Vorteile aus Gentoo's flexiblem Konzept gewinnen.

Gentoo ist unter diversen Architekturen lauffähig. Dazu zählen: Alpha, AMD64, ARM, Itanium, MIPS, PA-RISC, PowerPC, S/390, SH, UltraSparc und x86. Gentoo ist ebenfalls auf der Xbox, der Wii und auf der PlayStation 3 einsatzfähig.

Es gibt auch Projekte, bei denen der Linux-Kernel und einige GNU-Bibliotheken/Programme durch einen FreeBSD- (Gentoo/FreeBSD) bzw. OpenBSD-Kernel und deren Basis-Bibliotheken/Programme ersetzt wurde. Zudem kann man Gentoo unter verschiedenen Unix-ähnlichen Betriebssystemen in ein Unterverzeichnis installieren. Diese Installationsvariante wird Gentoo Prefix genannt. Unterstützt werden u. a. Mac OS X, Solaris und Microsoft Windows mit Hilfe der Microsoft Windows Services for UNIX.

Auch speziell gehärtete und an besondere Sicherheitsanforderungen angepasste Systeme, lassen sich verhältnismäßig einfach mit Gentoo erstellen, da es dafür bereits vordefinierte Profile gibt.

Gentoo zählt man allgemein zur Kategorie der quellbasierenden Distributionen. Es existiert bereits seit 12 Jahren und hat damit wohl bewiesen, dass es sich um keine Eintagsfliege unter den Distros handelt.

In der Sprache der Inuit, trägt die schnellste aller Pinguin-Arten den Namen Gentoo.

Da diese Distribution besonders viele Möglichkeiten der Optimierung zur Verfügung stellt, ist dieser Bezug durchaus zutreffend.

So lässt sich angefangen beim Kernel, weiter über einzelne Pakete, bis hin zum gesamten System, alles an die entsprechende Zielhardware anpassen und optimieren.

Natürlich ist allein die Möglichkeit zur Optimierung noch kein Garant für ein spürbar schnelleres System, sondern hängt auch von einer Menge von Faktoren ab. Auch die Entscheidung zwischen Aufwand und Nutzen ist sehr individuell.
Beispielsweise kann es je nach Rechner ½ Stunde bis zu einem Tag dauern ein OpenOffice zu kompilieren. Der zu erwartende Nutzen, ist jedoch relativ gering, denn was spielt es letztlich für eine Rolle, ob das Programm anschließend 25% schneller geladen ist, wenn der eigentliche Flaschenhals die Schreibgeschwindigkeit des Users ist. Bei Gentoo trägt man diesem Umstand dadurch Rechnung, dass man OpenOffice zusätzlich auch als Binär-Paket zur Verfügung stellt.

Natürlich gibt es auch Szenarien, bei denen eine Geschwindigkeitssteigerung im mindestens zweistelligen Bereich, durch Optimierung, erzielt werden kann.
Number-Crunching, oder Multimedia-Encoding können z.B. ganz erheblich davon profitieren.

Aber selbst wenn beispielsweise eine Webserver-Anwendung, die täglich Millionen von Anfragen möglichst schnell beantworten soll, durch gezielte Optimierung nur 5% schneller arbeitet, so handelt es sich auch hier um einen signifikanten Geschwindigkeitsgewinn.

Je nach Anwendungsfall kann Optimierung also von großem Nutzen für den User sein.

Doch letztlich entscheidet der Anwender was für ihn das Beste ist. Gentoo stellt ihm dafür die Mittel bereit, seine Anforderungen möglichst effizient umzusetzen.
So kann er sein System genauso generisch konfigurieren wie er möchte, um eine möglichst breite Basis an Hardware zu unterstützen.

So sollte also die Hoffnung auf mögliche Geschwindigkeitsvorteile, nicht das einzige Motiv sein, zu Gentoo wechseln zu wollen.
Je nach Zielsystem oder Programm sind die Unterschiede vom Anwender womöglich gar nicht zu spüren. Die aufgewendete Zeit zum Kompilieren und Konfigurieren dagegen sehr wohl.
So mag sich mancher später fragen, ob er hier zum Narren gehalten wurde, und ob nicht der deutsche Name für Gentoo „Eselspinguin“ die Lage besser beschreibt.
Dass dem aber nicht so ist, und Gentoo darüber hinaus noch viele weitere Vorzüge zu bieten hat, wird im weiteren Verlauf hoffentlich noch deutlich werden.

Man kann die Sache aber auch von einem anderen Standpunkt ausgehend betrachten.
Wer die vorhandene Hardware maximal auszunutzen möchte und sei es auch nur um die Gewissheit zu haben, dass kein Hardware-Potential brach liegt, oder einfach nur weil es machbar ist, der könnte sich gerade deshalb auch ganz bewusst für Gentoo entscheiden.

Die Diskussion um mögliche Geschwindigkeitsgewinne durch Optimierung an die vorhandene Hardware sorgt immer wieder für hitzige Debatten, weshalb ich diese mit folgendem Resümee abschließen möchte, dem sich hoffentlich jeder anschließen kann:
So lange man keine gravierenden Fehler bei der Optimierung macht, wird das System anschließend zumindest nicht langsamer als vorher laufen ;-)

Weniger ist manchmal mehr

Ganz anders sieht es dagegen bei der Optimierung der Programme durch Reduzierung der Abhängigkeiten auf das Wesentliche aus.
Der Performance-Gewinn kann dabei, im Vergleich zur Hardwareoptimierung, noch um einiges höher ausfallen.

Wie so oft besteht auch hier die Kunst im Weglassen!

Für Manchen gar nicht so einfach, denn dazu muss man genau wissen, was man will oder braucht. Genau hierin besteht die Schwierigkeit bzw. die Herausforderung.
Umgesetzt wird das ganze durch Gentoo's herausragendes Feature der USE-Flags, auf das wir später noch näher eingehen werden.

Ein entsprechend leichtgewichtiges System verhält sich fast wie im richtige Leben ;-)
So würde bei gleicher Leistung das leichtere Auto schneller beschleunigen als das vollbeladene.
Genauso verhalten sich auch einige Applikationen, wenn man sie von ihrem Ballast befreit.

Java-Unterstützung ist häufig für viele Programme eine Performance-Bremse oder Sicherheitslücke. Warum also Systemweit unterstützen, wenn man vielleicht nur in einem oder zwei Programmen Java-Funktionalität nutzen möchte.
Pulseaudio kostet nicht nur zusätzliche Ressourcen, es soll auch einigen Programmen, vor allem Spielen, Probleme mit der Stabilität bereiten. Vielleicht mögt ihr es auch einfach nicht, oder die Entscheidung eures Distributors es nun standardmäßig zu implementieren.
Bei Gentoo bräuchtet ihr nun einfach das USE-Flag **-pulseaudio** zu setzen und anschließend alle Pakete, die diese Funktionalität eingebaut haben, nun ohne diese zu rekompilieren. Diese würde dann durch das absetzten eines simplen Befehls wie **emerge -N world** umgesetzt werden. **-N** steht hier für **--newuse**.
So einfach kann die Welt sein, wenn man Gentoo erst einmal installiert hat ;-)

Weitere Beispiele:
Multimessanger arbeiten wesentlich flüssiger, wenn man sie ihrer nicht benötigten Protokolle beraubt.
Programme ohne Debug-symbols zu kompilieren, sorgt ebenfalls für ihre Beschleunigung. Sollte man ernsthafte Schwierigkeiten mit einem Programm haben und die Debug-Informationen zur Fehlersuche benötigen oder einen detaillierten Bugreport einsenden wollen, so kann man immer noch gezielt dieses eine Programm wieder mit Debug-Informationen übersetzen, bis man das Problem im Griff hat.

Auch wird durch die Reduzierung der Komplexität auf das Wesentliche, häufig nach der Installation einiges einfacher bzw. übersichtlicher zu konfigurieren, da man sich mit weniger Optionen herumschlagen muss.

Oft wird gerne auf den wichtigen Aspekt von Open-Source-Software hingewiesen, sich der Quellen bedienen zu können, diese nach belieben zu sichten, zu studieren, und verändern zu dürfen.
Doch wie viele User machen von diesem Grundrecht der Selbstbestimmung Gebrauch?

Viele sagen sie brauchen die Quellen nicht, denn sie wollen nichts programmieren, oder sich nicht mit Programmierung auseinander setzten.
Jedoch von der einfachsten Art der Nutzung der Quellen, nämlich der Anpassung an eigene Bedürfnisse oder eigene Hardware, wird von kaum einem User Gebrauch gemacht.

Dabei ist gerade diese Möglichkeit und Freiheit, für mich eines ***der*** technischen und rechtlichen Abgrenzungsmerkmale gegenüber proprietären BS, und daher als besonders erstrebenswert anzusehen.
Ich frage mich manchmal, warum nicht mehr Open-Source-User von diesen erweiterten Möglichkeiten der Einflussnahme Gebrauch machen, statt sich lediglich auf die Auswahl von zu installierenden Programmen zu beschränken.

Geschieht dies
1. aus Bequemlichkeit oder der Überzeugung, dass der Mehraufwand die Mühe nicht wert sei oder
2. lediglich in Unkenntnis der vielfältigen Möglichkeiten selbst über die Fähigkeiten seines Systems bestimmen zu können?

Trifft auf euch ersteres zu, dann werdet ihr vermutlich nicht zu Gentoo finden.
Trifft jedoch eher letzteres zu und ihr erkennt darin einen Mehrwert für euch, dann werden euch die Hürden der Installation hoffentlich nicht davon abbringen, zu eurem ganz persönlichen Wunschsystem zu gelangen.
Voraussichtlich werdet ihr euer System dabei genauer kennen lernen als je zuvor und gleichzeitig viele allgemeingültige Linux-Kenntnisse erwerben.

Wer mit Gentoo beginnt, lernt Linux! (und wird nicht, wie bei vielen anderen Distributionen, nur deren spezifische Kenntnisse erwerben).

Gentoo – die Distribution, die keine ist

Viele nennen Gentoo auch die "Metadistribution", wegen seines Paketmanagements unter Verwendung von Ebuilds bzw. der darin enthaltenen Metainformation zum Bau der Pakete.

Andere wiederum sagen Gentoo ist ein Framework zum Bau eines eigenen Linux-Systems oder Distribution.

Jedenfalls verzichtet Gentoo bewusst auf jegliche Vorauswahl von Kernel, Paketen oder Programmen und letztlich sogar auf die Zuhilfenahme eines Installers, weshalb man eigentlich nicht von einer Distribution sprechen kann.

Stattdessen stellt es die nötigen Werkzeuge und Dokumentationen bereit, um den Anwender beim Bau seiner individuellen Installation bestmöglich zu unterstützen und anzuleiten, ohne ihn dabei zu bevormunden oder einzuschränken (*Gentoo is all about choice*).

Allerdings stellt diese Wahlfreiheit den Anwender womöglich vor eine bisher ungeahnte Qual der Wahl, mit der man erst einmal zurecht kommen muss und die außerdem noch eine gewisse Sachkenntnis erfordert.

Deshalb muss man gleich zu Beginn oder besser noch vor der Gentoo-Installation, mit dem Selbststudium allgemeiner sowie Gentoo-spezifischer Grundlagen beginnen, um die Installation erfolgreich und zügig bewältigen zu können.
Genauso wie vor der ersten praktischen Fahrstunde mindestens eine Einweisung in die Fahrzeugbedienung, besser aber noch einige Stunden Theorie liegen sollten.
Hier sieht wohl jeder ein, dass es wenig Sinn ergäbe, sofort los zu fahren, ohne gewisse Vorkenntnisse zu besitzen.

So ist der geneigte Gentoo-User quasi genötigt das umfangreiche Handbuch zu lesen, bei dem er sich Allgemeingültiges, sowie Gentoo-spezifisches Wissen aneignet, um die Installation erfolgreich durchführen und anschließend das System nach seinen Wünschen zusammenstellen zu können (*The Gentoo way*).
Häufig hört man stattdessen: "Gentoo erfordert eine steile Lernkurve". Diese Aussage liefert aber keine Begründung für die dahinter stehende Philosophie und scheint mir daher zu kurz gefasst.

Daher rate ich auch von jeglichem Versuch ab, sich an den Hürden der Installation, durch irgendwelche Kurz-Installations-Anleitungen, vorbei zu schummeln. Dieses würde früher oder später zu einer Bauchlandung führen werden, da das erforderliche Basiswissen fehlt.
Oder anders ausgedrückt:
Nur der beschwerliche Pfad führt zu einer höheren Stufe der Erkenntnis.

Deshalb möchte ich hier auch keine Kurz-Installation beschreiben, sondern dazu anhalten, so wie im Gentoo-Handbuch beschrieben, vorzugehen.
Stattdessen möchte ich in groben Zügen erklären, was bei der Installation eigentlich geschieht, welche Logik dahinter steht und somit versuchen den Blick auf das Wesentliche zu lenken, statt den Anwender sequenziell irgendwelche Befehlszeilen abarbeiten zu lassen.

Gentoo-spezifische Begrifflichkeiten

Portage ist die Paketverwaltung von Gentoo und ermöglicht den automatischen Bau der einzelnen Pakete aus ihren Quelltexten. Dabei stützt es sich auf den sogenannten **Portage tree.** Das ist ein Verzeichnis, welches Informationen zu jedem einzelnen Programm und jeder Bibliothek in Form von Scripts namens **Ebuilds** bereitstellt.
Die **Ebuilds** steuern den kompletten Ablauf der Installation. Vom Download der Quelltexte, über die Verifikation der Unverfälschtheit der Dateien mit Hilfe von Prüfsummen und Anwendung von distributions-spezifischen Patches, sowie die Berücksichtigung der sogenannten USE-Flags bis zur Kompilation des Paketes in einer Sandbox, um es darauf hin zu installieren.
Dabei werden, sofern diese Option nicht explizit ausgeschaltet wurden, etwaige Abhängigkeiten zu anderen Paketen beachtet und diese, falls nötig, ebenfalls aktualisiert oder installiert.
Der **Portage-Baum** wird mit Hilfe von rsync auf den aktuellen Stand der Distribution gebracht.

Emerge
Das Programm, das auf den Portage Baum zugreift, um Software zu installieren, heißt **emerge**.
Es ist ein Terminal-Programm und er lässt sich gut konfigurieren, um es dem eigenen Arbeitsablauf anzupassen.
Emerge verwaltet den Software-Bestand mit all seinen Abhängigkeiten, d. h. man kann damit die Software installieren, de-installieren und aktualisieren.
Beim Installieren folgt es den Informationen aus den **Ebuilds**.

Ebuilds

Ebuilds sind sogenannte Paketdefinitionen, also lediglich beschreibende Dateien (Metadaten) zum Bau und Installation der Programme. Sie enthalten in Textform Informationen und Verweise zum:

- Installieren von benötigter Software und Bibliotheken
- Herunterladen der Quelltexte
- Überprüfung auf Unverfälschtheit
- Entpacken
- Konfigurieren
- Kompilieren
- Installieren in eine Sandbox
- Installieren ins laufende System

Alle von Gentoo bereitgestellten Ebuilds werden nach Kategorien strukturiert und bilden zusammengenommen den Portage-Baum.
Der Portage-Tree wird laufend von den Gentoo Entwicklern online aktualisiert, sodass man nach einem Abgleich mit dem eigenen Baum, sehr schnell Zugriff auf die neuesten Versionen hat.
Eine eventuell erwünschte Erweiterung des Portage-Tree, durch eigene oder fremde Ebuilds, kann mittels sogenannte Overlays (Überlagerungen) erfolgen.

Die Gentoo Installation

Eckpfeiler der Installation zum besseren Verständnis der Abläufe (kein Handbuch-Ersatz).

Die Philosophie hinter Gentoo, dem Benutzer alle Freiheiten zu lassen, wird schon bei der Wahl der Installationsmethode klar.

Da es keinen einschränkenden Installer gibt, darf der Benutzer von Anfang an entscheiden mit welchem Linux-Installationsmedium er beginnen möchte, mit welchem Tool und auf welche Art er sein **Installationsmedium vorbereiten** und auf welche Art er seine Internetverbindung herstellen möchte. Die Installation wird am sinnvollsten direkt über das Internet durchgeführt. Dies hat den Vorteil, dass die Pakete sehr aktuell sind. Eine Installation von CD als Paketquelle ist zwar möglich, aber aus meiner Sicht nicht zu empfehlen.

Auf eurem PC läuft bereits ein Linux, das erhalten werden soll? Dann nutzt doch einfach dieses zur Installation von Gentoo aus eurer gewohnten Umgebung heraus.
Falls noch kein Linux installiert ist, könnt ihr eine beliebig vorhandene Linux Live-CD verwenden, die in der Lage ist, eure Hardware ausreichend gut zu erkennen und es euch ermöglicht eine Internetverbindung einzurichten.
Solltet ihr noch eine Live-CD benötigen, empfehle ich die Verwendung der SystemRescueCd, da diese schlank und universell genug ist, um die meisten Anforderungen einer x86 32Bit/64Bit Installation zu erfüllen und darüber hinaus alle Tools enthält auf deren Verwendung im Gentoo-Handbuch eingegangen wird.
Mit ihr kann man dann auch eine X-Oberfläche starten, um z.B. mit gparted komfortabel seine Festplatte zu partitionieren, mit Firefox online das Gentoo-Handbuch zu lesen, während man in einem Terminal Gentoo installiert. Natürlich kann man genauso gut alles per Text-Konsole installieren.

Grundsätzlich ist zu beachten, dass während einer Installation die gleiche Architektur laufen muss (z.B. 32- oder 64-Bit), die installiert werden soll, damit das spätere chrooten funktioniert. Im Fall der SystemRescueCd müsste man daher besonders darauf achten, je nach Wunsch der zu installierenden Architektur, den 32- oder den 64-Bit Kernel zu booten.

Als nächstes wird das zukünftige root-Verzeichnis eingehangen, damit dorthin das herunterzuladende Stage3-Archiv gespeichert werden kann, welches kompatibel zur Zielhardware und zur gewünschten Architektur von den Gentoo-Mirror-Servern ausgewählt werden muss.

Info

Ein Stage3-Archiv beinhaltet ein minimalistisches Gentoo-Grundsystem (baselayout) inklusive Tools wie z.B. rsync, wget, python, portage, sowie zuzüglich der GNU Toolchain.
Die Version dieses Basissystems (*engl.* base system) entspricht dem Paket sys-apps/baselayout, dessen Version sich auch aus der Datei **/etc/gentoo-release** auslesen lässt. Es ist die Grundlage des Betriebssystems und als die eigentliche Version einer Gentoo-Installation anzusehen.
Zu einem vollwertigem Gentoo-System fehlt dem Stage3 allerdings mindestens noch der Portage-Tree, ein Kernel, ein Root-Account, ein Bootloader, sowie die entsprechenden Einstellungen.
Seit 2005 gibt es diese architektur-spezifischen Stages, die auf wöchentlicher Basis skriptgesteuert aus dem stabilen Zweig des Portage-Tree erzeugt werden, und die eigentlich nur für eine Neuinstallation benötigt werden.
Vor dieser Zeit musste man, noch ausgehend von minimalistischen Stage1-Archiven, sich sein eigenes Stage3-Archiv erstellen, was ähnlich aufwendig und schwierig war, wie sich mit *LFS* ein Grundsystem zu erstellen.

Der Vorteil einer Installation von einem Stage1 war, dass man damit direkter zu seinem für Hard- und Software optimierten Grundsystem gelangte. Dafür war der Vorgang des Bootstrapping etwas schwieriger und daher für Einsteiger um einiges fehleranfälliger.
Aus diesem Grund empfiehlt die Installationsanleitung seitdem die Verwendung eines Stage3-Archives, minimalere Stages werden daher nicht mehr angeboten.

Ein Stage3 ist anfangs auch generisch konfiguriert und orientiert sich daher ebenfalls am kleinsten gemeinsamen Nenner der gewählten Architektur.
Um diese Bestandteile ebenfalls zu optimieren, wäre nach dem Verändern der Parameter das Rekompilieren dieser Pakete (system) erforderlich. Ansonsten würde sich die Optimierung lediglich auf alle, nach dem Verändern der Parameter kompilierten Pakete auswirken. Spätestens nach einem Versions-Update würden jedoch dann auch diese Bestandteile optimiert werden, es sei denn, man belässt die generischen Einstellungen, weil man sämtliche Architekturen unterstützen möchte.

Das Stage3-Archiv wird in der Partition entpackt, die später zum Wurzel-Verzeichnis des Systems werden soll. **`tar xvjpf stage3-*.tar.bz2`**
Hierbei ist die Angabe der korrekten Parameter von **tar** wichtig, damit beim Auspacken und Anlegen der Verzeichnisstruktur die Rechte, der darin enthaltenen Dateien, erhalten bleiben.

Nun lädt man einen der täglich aktualisierten Snapshots des Portage-Tree herunter. Entpackt diesen ebenfalls, jedoch unter Berücksichtigung abweichender Parameter!
Man beachte den Wegfall von Parameter **p (Preserve permissions)**, die absolute Pfadangabe, sowie den Wechsel ins Zielverzeichnis durch Verwendung des großen **–C (VERZEICHNIS wechseln)**, um das Archiv dort zu entpacken.
`tar xvjf /mnt/gentoo/portage-latest.tar.bz2 -C /mnt/gentoo/usr`

Compiler Einstellungen vornehmen (gcc C und C++ Compiler)

Die Variablen **CHOST**, **CFLAGS** und **CXFLAGS** beeinflussen das Verhalten des GNU Compilers gcc und legen durch seine Compiler-Flags fest, mit welchem Grad der Optimierung dessen Code erzeugt wird (-O Flags + Zusätze).
Außerdem bestimmt ihr damit auf welchem Maschinentyp, Prozessorfamilie oder Prozessor das kompilierte Binary lauffähig sein wird und wie gut es die erweiterten Hardwarefeatures ausnutzen kann.

Ihr habt schließlich gutes Geld für den Prozessor bezahlt, warum solltet ihr ihn dann nicht richtig nutzen dürfen? ;-)

An dieser Stelle bestimmt ihr also, ob ihr euer System generisch oder hardwarespezifisch gebaut werden soll.
Wenn ihr nicht gerade vor habt, als Distributor tätig zu werden oder das gleiche System auf vielen verschiedenen PCs einsetzen zu wollen, dann sollte nichts dagegen sprechen, alle noch zu kompilierenden Pakete hardwarespezifisch für euer eigenes System erstellen zu lassen.

Beim Grad der Optimierung sollte man jedoch konservativ bleiben (-O2). Eine allzu aggressive Optimierung kann zu Fehlern beim Kompilieren oder beim auszuführenden Programm führen, die sich nicht ohne weiteres der massiven Optimierung zuordnen lassen.

Seit der gcc-Version **4.2.x** gibt es sogar eine automatische Erkennung der Hardwarefeatures, die sich z. B. mittels CFLAGS="-O2 **-march=native** -pipe" aktivieren ließe. Wer sich also nicht lange mit den bestmöglichen Settings für sein System herumschlagen möchte, der nimmt erstmal diese Einstellung.
Das empfiehlt sich jedoch nur für den Fall, dass man kein Crosscompiling nutzen möchte und funktioniert außerdem nicht für jede Hardware optimal (auch hier, wie bei allen Automatismen gilt wieder mein Leitspruch: „Wenn du willst das etwas richtig gemacht wird, mach es selbst.“

Als sicherer Anhaltspunkt zum Einstieg empfiehlt sich das Wiki: http://en.gentoo-wiki.com/wiki/Safe_Cflags, es ist aber nicht immer auf dem neuesten Stand und wer sichergehen möchte, die optimalen Settings für seine Hardware zu finden, der sollte nach weiterführender Dokumentation Ausschau halten, wie z. B.
http://www.gentoo.de/doc/de/gcc-optimization.xml
http://gcc.gnu.org/onlinedocs/gcc-4.3.3/gcc/i386-and-x86_002d64-Options.html#i386-and-x86_002d64-Options
http://de.wikipedia.org/wiki/Internet_Streaming_SIMD_Extensions

Compiler und deren Qualität des erzeugten Codes sind eine Wissenschaft für sich.
Noch vor Jahren stand der proprietäre und kostenpflichtige Intel Compiler im Ruf, bis zu 50% effektiveren Code als der gcc zu produzieren.
Auf der anderen Seite ist es mit ihm nicht möglich, einen aktuellen Linux-Kernel zu übersetzten.
Wie weit der gcc diesbezüglich aufgeschlossen hat, vermag ich nicht zu sagen, doch hier sieht man welches Potential in der stetigen Weiterentwicklung des gcc steckt.
Schaut man bei neuen GCC Releases in die aktualisierten Infos unter Status/ Changes, Stichwort:
"GCC 4.x Release Series - Changes, New Features and Fixes; Caveats," so kann man sich regelmäßig von den Verbesserungen und Performance-Steigerungen des erzeugten Codes überzeugen.

Gerade jüngere Prozessorgenerationen wie z.B. Atom oder Arm profitieren erst bei neueren gcc-Versionen von den dort neu aufgenommenen "**New Targets**". Natürlich nur, wenn der Code hardwarespezifisch erzeugt wurde. Die generischen Einstellungen können sich hardwarespezifische Features nicht zu nutze machen.

Als Gentoo-User kann man auch nachträglich von den Verbesserungen des gcc profitieren. Dazu braucht man nur den gcc updaten, das Profil des neuen Compilers aktivieren und die neuen Compiler Flags hinzufügen. Es sei denn, man nutzt ohnehin die automatische Erkennung/Aktivierung der Hardwarefeatures mittels CFLAGS="-O2 **-march=native** -pipe". Anschließend kann man entweder sein gesamtes System in einem Rutsch rekompilieren (**`emerge -e system world`**) oder der Evolution seinen Lauf lassen, also im Rahmen der künftigen Updates nach und nach davon profitieren.

Der Kernel ist übrigens von diesen Einstellungen ausgenommen, denn er bestimmt durch seine eigene Konfiguration, welche Parameter er dem gcc beim kompilieren übergibt.

Für besonders Ambitionierte gibt es auch noch Möglichkeiten der Optimierung des Linkers mittels LDFLAGS. Siehe: http://en.gentoo-wiki.com/wiki/Safe_LDFLAGS als kleinen Einstieg in das Thema.
Mittlerweile sind jedoch in Gentoo's Standardprofil sichere und performante Vorgaben gewählt, weshalb man nicht zwingend aktiv werden muss.

Weiter geht's dann im Handbuch damit, fehlende Parameter der **make.conf** zu ergänzen, um solche Einträge wie: MAKEOPTS, GENTOO_MIRRORS, die dort ausführlich beschrieben sind.

Kopieren der DNS-Informationen aus der laufenden Umgebung mit funktionierender Internetanbindung (/**etc/resolv.conf**) in die künftige Chroot-Umgebung mittels: **`cp -L /etc/resolv.conf /mnt/gentoo/etc/`**

Ebenfalls s. Handbuch; danach wird das chrooten beschrieben.

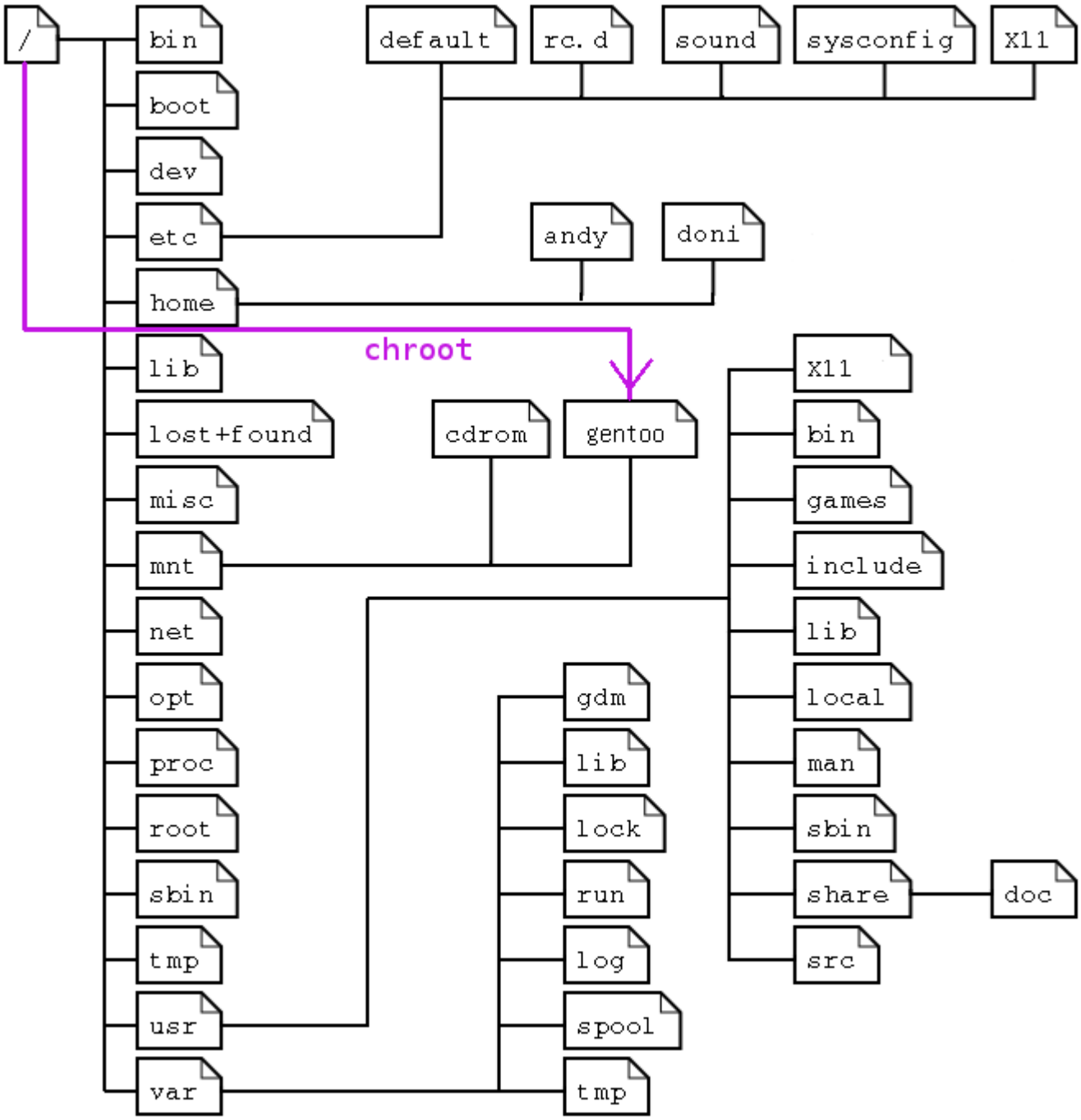
chroot – Wofür ist das gut?

Wie der Name schon fast verrät, bedeutet es Change root oder zu deutsch; wechseln der root-Umgebung. Das ist nichts Gentoo-spezifisches, sondern zählt zu den Standard-Tools unter Linux und ist für viele Dinge nützlich, wie z. B. für sicherheitskritische Anwendungen oder Tests, da man dadurch Anwendungen in geschlossene Umgebungen einsperren kann.
Die im chroot laufenden Anwendungen, sehen dabei nur die Verästelungen innerhalb und unterhalb ihres Verzeichnisses, welches zum neuen root gesetzt wurde. Auf die darüber liegenden Verzeichnisse haben sie dagegen keinen Zugriff mehr.

Chrooten ist also nicht nur für die Gentoo-Installation erforderlich, sondern es kann einem auch bei Problemen helfen, bei denen man sich aus dem System ausgesperrt hat, etwas zerschossen wurde und sich nun von dem System nicht mehr booten lässt. Das gilt gleichermaßen für alle Distributionen.

Bei Gentoo wird dieses elementare Wissen also gleich zu Anfang durch praktische Anwendung vermittelt, da es in den Installationsablauf integriert ist. Jedoch erläutert das Handbuch keine weiteren Details, weshalb ich das hiermit tun möchte.

Ein einfaches **`chroot /mnt/gentoo`** würde zwar das alte Wurzelverzeichnis zum neuen unter **/mnt/gentoo** werden lassen (Grafik s. u.). Jedoch nutzt das nicht viel. Dort sind wir derart abgeschottet, denn die nötige Funktionalität des laufenden Linux-Systems fehlt.



Daher müssen wir vor dem chrooten dafür sorgen, das in der künftigen root-Umgebung das Pseudo-Dateisystem **/proc** und eine Instanz der Device-Nodes **/dev** dort zur Verfügung stehen werden.
Dann erst sind die Voraussetzungen geschaffen, dass wir in der neuen root-Umgebung den laufenden Linux-Kernel so nutzen können, als hätten wir direkt davon in unsere chroot-Umgebung gebootet.
Das ist auch der Grund, warum die Architektur des laufenden Kernels, zu der in unserer neuen Umgebung kompatibel sein muss!
Nach dem Betreten der neuen Umgebung, muss diese noch aktualisiert werden, damit sie von den Änderungen in Kenntnis gesetzt wird.

Das alles geschieht durch die Eingabe der nachfolgenden Befehlszeilen. Eventuell anders lautende Mountpoints sind anzupassen.

```
(# mount /dev/sda3 /mnt/gentoo)          #Nur nötig, falls noch nicht eingehangen.
(# mount /dev/sda1 /mnt/gentoo/boot)     #Falls vorhanden, aber noch nicht eingehangen.
(# swapon /dev/sda2)                     #Falls vorh. oder bei Speichermangel benötigt.

# mount -t proc none /mnt/gentoo/proc
# mount -o bind /dev /mnt/gentoo/dev
# chroot /mnt/gentoo /bin/bash
# env-update
# source /etc/profile
# export PS1="(chroot) $PS1"             #Dient nur um den Prompt besser kenntlich zu machen.
```

Danach kann man dann wie gewohnt, in dieser neuen Root-Umgebung arbeiten. Jedoch nur mit den Programmen, die dort zur Verfügung stehen. Das gilt aber nur für die Konsole oder das X-Terminal, über das man gechrootet hat. Auf den anderen Konsolen steht, wie gewohnt, das ursprünglich gestartete System zur Verfügung.

Im Handbuch wird nicht explizit erwähnt, dass man mittels chroot jederzeit eine abgebrochene Gentoo-Installation wieder aufnehmen kann, ohne alle vorangegangenen Schritte wiederholen zu müssen.
Nur die in Klammern gesetzten Befehle sind dann vorher erforderlich, um danach im Handbuch beim Kapitel chroot fortfahren zu können.

Danach geht's im Handbuch weiter mit:

Portage-Tree updaten mit: `# emerge --sync`

Auswahl des richtigen Profils `# Das Installationsprofil wählen`

Die Grunddefinition eines zu installierenden Systems, legt man bei Gentoo mit Hilfe so genannter *Profile* fest. Sie definieren z. B. welche Pakete für das Funktionieren der Installation unverzichtbar sind, welche Software unter keinen Umständen installiert sein darf, welches Paket den Zuschlag erhält, wenn mehr als eines die gewünschte Funktionalität bereitstellt und sie geben gewisse Eigenschaften vor, die neu eingespielte Pakete aufweisen sollen.

Als nächstes machen wir uns mit dem Konzept der USE-Flags vertraut, ausführlich im Handbuch nachzulesen.

Was sind USE-Flags?

USE-Flags sind das herausragende Alleinstellungsmerkmal, mit dem sich Gentoo von anderen Distribution unterscheidet.
Die Idee hinter den USE-Flags ist so einfach wie genial:
Anpassung der Programme an die exakten Wünsche und Anforderungen des Anwenders!

Wenn man Gentoo, eine andere Distribution oder gar ein anderes *BS* installieren möchte, so wird man gewisse Entscheidungen in Abhängigkeit von seinen Anforderungen und seiner Umgebung treffen müssen. Die Konfiguration eines Server unterscheidet sich schließlich von der einer Workstation. Eine Office-Workstation unterscheidet sich von einer Workstation für 3D-Rendering oder Videoschnitt.

So verzichten Administratoren von Servern gerne auf eine graphische Oberfläche. Während Desktop-Benutzer gerne OpenOffice im Design ihrer Desktopumgebung haben möchten, aber keine OpenGL Unterstützung benötigen, möchte ein 3D-Designer oder ein Videoeditor kaum darauf verzichten.

Mit den USE-Flags bietet Portage eine Art Schalter, um solche Optionen bei der Installation zu aktivieren oder zu deaktivieren.
Die Entwickler eines Paketes haben meist in den Quellcode Optionen eingebaut, damit unerwünschte Funktionen abgeschaltet werden können oder der Funktionsumfang durch Einbindung externer Software erweitert werden kann.

Die USE-Flags bilden eine Abstraktionsschicht zur Konfiguration, der beim Kompilieren verwendeten Paket-Optionen, die anschließend die Funktionalität der erzeugten Pakete bestimmen.

Während bei den binären Distributionen, die jeweiligen Paketbetreuer entscheiden, welche Funktionen der Anwender wohl benötigen mag, überträgt Gentoo diese Entscheidungen auf den Anwender.

Wegen der Fülle, der zur Verfügung stehenden USE-Flags, entsteht besonders für Einsteiger zu Beginn ein erhöhter Aufwand zum Sichten dieser und zum Informieren über deren Funktionalitäten.
Durch die zu treffende Auswahl der entsprechenden USE-Flag spezifiziert man die Eigenschaften seines individuellen Wunsch-Systems.
Über erweiterte USE-Flags werden darüber hinaus, die zu unterstützende Hardware, die gewünschten Sprach-Unterstützungen und die zu akzeptierenden Software-Lizenzen definiert.
Erst dadurch wird Portage beim Kompilieren eine jeden Programms in die Lage versetzt, die Wunschkonfiguration des Users zu berücksichtigen.

Bei späteren Updates oder Änderungen fällt der Aufwand dagegen nicht mehr ins Gewicht, da man Änderungen an den Paketoptionen übersichtlich angezeigt bekommt und bei Bedarf bequem eingreifen kann.

Die Implementation des An- und Abschalten von Funktionen kann dabei vom Ebuild-Skript individuell umgesetzt werden , – i. d. R. durch die Anwendung von Configure-Optionen oder Patches.

Die USE-Flags lassen sich mit Hilfe von Konfigurationsdateien sowohl zentral für alle, als auch für einzelne Pakete steuern.

Woher kommen die USE-Flags?

Die Software-Entwickler implementieren in ihre Programme häufig bestehende Funktionalitäten externer Programme zuzüglich, der selbst geschriebenen Funktionen.
Viele davon sind optional und können durch entsprechend gesetzte Optionen (**enable/ disable**) einbezogen werden, was wiederum gewisse Abhängigkeiten nach sich zieht.

Wie der Programmierer diese Option betitelt, ist aber weitestgehend ihm überlassen.
So entsteht eine stetig wachsende Zahl an Optionen und deren Benennung, die eigentlich kaum zu überblicken ist.

Würde man selbst die Standard-Vorgabeoptionen aller verwendeten Quellpakete (enabled/ disabled) beeinflussen wollen, so käme man vor dem Kompilieren nicht umhin, diese Vorgaben zu sichten, um sie anschließend, in selbst geschriebenen Buildsripts pro Paket, nach eigenen Wünschen zu setzen.
Bei jedem Paketupdate wäre wiederum dessen neuer Funktionsumfang auf Änderungen zu prüfen, um anschließend das bereits vorhandene Buildscript anzupassen.

Diese Sisyphusarbeit, den gesamten Paketumfang einer umfangreichen Distribution mit selbst erstellten Buildscripts abzudecken, würde wohl kein Mensch alleine auf sich nehmen wollen.
Doch gemeinsam sind wir stark und so kommt es, dass die Gentoo-Paketmaintainer mittlerweile einen, sich täglich aktualisierenden, Pool von mehr als 26000 aktuellen Ebuilds, der nahezu alle Bedürfnisse abdeckt, verwalten.

Welche USE-Flags gibt es?

Im wesentlichen gibt es zwei Arten von USE-Flags: **globale** und **lokale** USE-Flags.

- Ein **lokales** USE-Flag wird zu Beginn seiner Aufnahme in Portage als neue Konfigurations-Option lediglich von einem einzelnen Paket verwendet.
  Im Verlauf der Weiterentwicklung können naturgemäß neue Pakete hinzustoßen, die ebenfalls von dem gleichen, lokalen USE-Flag Gebrauch machen werden. Der Status **lokal**, des USE-Flag ändert sich dadurch aber noch nicht.

- Ein **globales** USE-Flag wird von mehreren Paketen systemweit benutzt. USE-Flags werden erst nach Absprache der Entwickler untereinander und unter folgenden Voraussetzungen in den Stand der **globalen** USE-Flags erhoben:

  - Es müssen mindestens 5 Pakete davon Gebrauch machen.

  - Sie sollen allgemeine und übergreifende Ausrichtungen haben. So können z. B. Client- und Server USE-Flags niemals zu globalen USE-Flags werden, da sie spezifische und entgegengesetzte Ausrichtungen haben.

Sowohl **globale** als auch **lokale** USE-Flags lassen sich systemweit über die USE-Variable in der **/etc/make.conf** oder paketspezifisch, z. B. über die **/etc/portage/package.use**, setzen.

Eine Liste von allen verfügbaren **globalen** USE-Flags findet man unter dem Pfad:
`/usr/portage/profiles/use.desc`, die Liste der **lokalen** USE-Flags liegt dagegen hier:
`/usr/portage/profiles/use.local.desc` oder alle gemeinsam Online.

Auf eine weitere Unterkategorie, die erweiterten USE-Flags, wollen wir später noch eingehen.
Doch zuvor ein paar Beispiele, die den Nutzen und die Wirkungsweise der USE-Flags verdeutlichen sollen.

Wirkungsweise der USE-Flags.

Das Paket **app-office/openoffice** bietet u. a. das USE-Flag **cups**, damit OpenOffice mittels CUPS drucken kann. Besitzt man keinen Drucker, so braucht man diese Funktion vermutlich weder in openoffice noch in irgend einem anderen Programm.
So deaktiviert man dieses USE-Flag systemweit mit **-cups**, in der **/etc/make.conf,** damit cups in kein einziges Programm hinein kompiliert noch installiert wird. Das spart nicht nur Zeit beim Kompilieren, sondern auch Platz auf der Platte und bei den Binaries.

Aktiviert man hingegen das USE Flag **kde**, damit OpenOffice das Aussehen und die Bedienung von KDE Programmen annimmt, so wird zusätzlich, vor OpenOffice, das KDE Grundsystem als Folge der gesetzten Abhängigkeiten installiert, falls es nicht ohnehin schon vorhanden ist.

Hätte man dagegen bereits systemweit in der make.conf **java**-Unterstützung aktiviert, würde diese aber lediglich für openoffice deaktivieren wollen, so müsste man nur für diese Programm **-java** in der **/etc/portage/package.use** deklarieren, z. B. so: **`echo "app-office/openoffice -java" >> /etc/portage/package.use`**

Da diese Entscheidungen, selbst bei gleichen Anforderungen, von jedem User unterschiedlich getroffen werden, gleicht, durch die Fülle von USE-Flags, keine Gentoo-Installation der anderen.

Die meisten Distributionen kompilieren ihre Pakete mit eingeschalteter Unterstützung für alles, was möglich ist. Dies vergrößert die Programme, verlängert deren Startzeit und erhöht letztendlich die Abhängigkeiten. Mit Gentoo könnt ihr selbst bestimmen, mit welchen Optionen ein Paket übersetzt werden soll.

Deshalb ist Gentoo nicht nur eine hervorragende Distribution für Individualisten, sondern auch für User die selbst über ihre Konfiguration bestimmen wollen.

USE-Flags von Paketen anzeigen

Mit emerge kann man sich die USE-Flags von Paketen und ihre aktuelle Konfiguration anzeigen lassen, z. B. für sys-apps/portage:

`emerge --pretend --verbose <Paketname>` **#oder in Kurzform:**

`emerge -pv <Paketname>`

Alternativ gibt es viele weitere praktische Werkzeuge zum Anzeigen und Verwalten der USE-Flags, z. B. aus dem Paket **app-portage/gentoolkit:**

`euse` http://de.gentoo-wiki.com/wiki/Gentoolkit/euse

`equery` http://de.gentoo-wiki.com/wiki/Gentoolkit/equery#hasuse

oder aus dem Paket **app-portage/ufed**

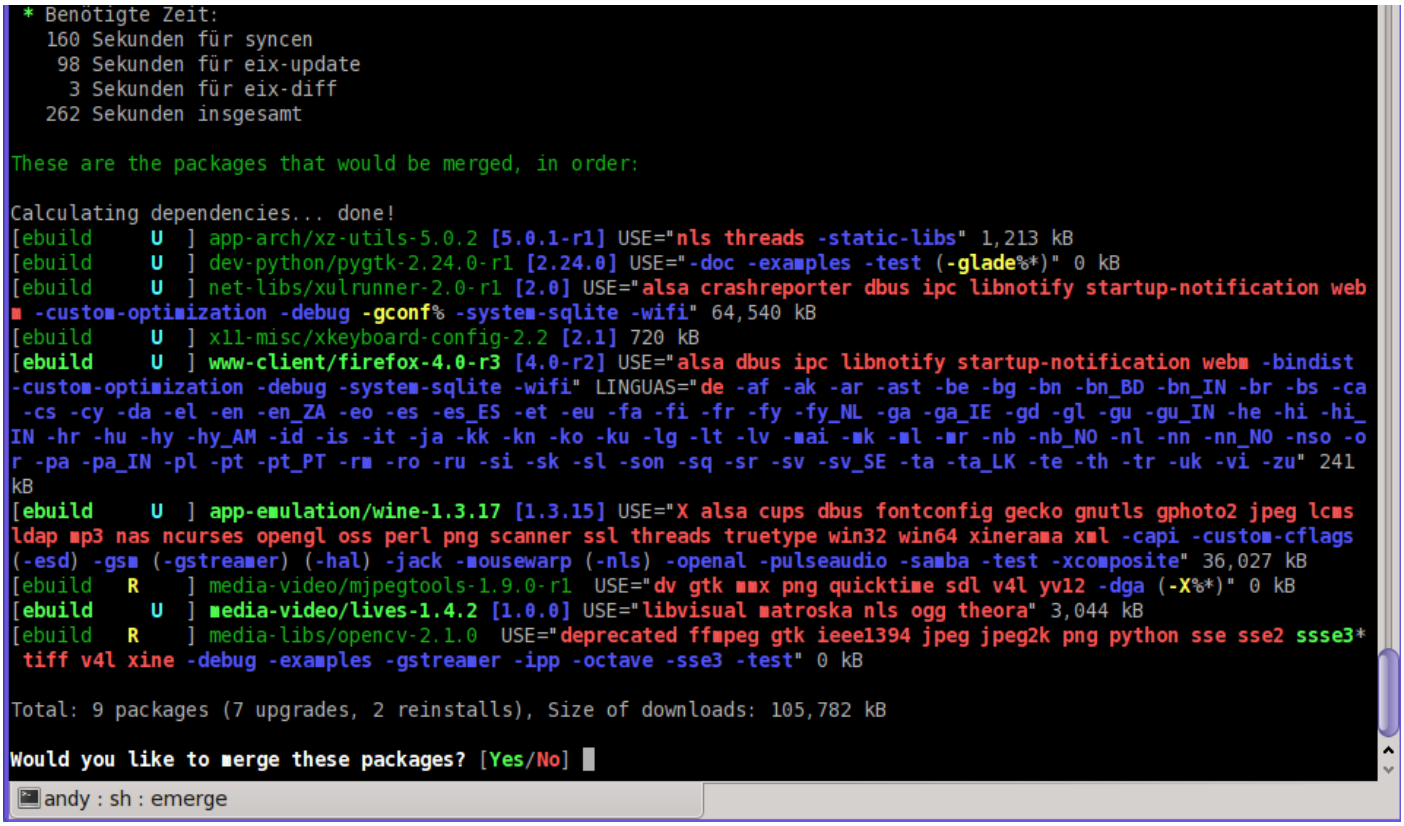
- **ufed**: Ein ncurses basierender Editor zum Selektieren der USE-Flags mit Info über diese.

Kennzeichnung von aktiven, inaktiven und geänderten USE Flags.

Blau - (vorangestellt) → inaktives Feature des Paketes.
Rot ohne Vorzeichen → aktives Feature des Paketes.

Grün nachträglich manuelle Änderung dieser Flags von bereits installierten Paketen, egal ob local/global.
* hinter Grün → davon ändern sich jedoch nur die mit *gekennzeichnete Flags durch ein remerge.

Gelb % → es gibt geänderte Flags innerhalb des neuen Paketes, verglichen mit der bereits installierten Version.
- (vorangestellt) → wegfallendes Feature.
* hinter Gelb → davon ändern sich jedoch nur die mit *gekennzeichnete Flags durch das Update aufgrund der Settings.
ohne Vorzeichen → hereinkommendes Feature.



emerge sortiert die USE-Flags in der Anzeige entsprechend ihrem Status (aktiviert/deaktiviert).
Die aktivierten Flags stehen vorn.
Wer die Flags alphabetisch sortiert haben möchte, kann die Option **--alphabetical** verwenden.

Nun, da das Prinzip der USE-Flags klar sein sollte, fahrt ihr fort, wie im Gentoo-Handbuch beschrieben (auch beim Handbuch auf die korrekte Architektur achten!), mit diversen vorzunehmenden Einstellungen, wie z. B.:

- **Spezifizieren Ihrer Locales**
- **Setzen der Zeitzone**
- **Installieren der Kernelquellen**

- **Konfiguration und Erstellung des Kernels (**könnte vom Umfang her einen eigenen Vortrag füllen).

  Auch hier wird es sich früher oder später zeigen, das die Einarbeitung in die eigene Kernel-Konfiguration, gut investierte Zeit ist, jedenfalls wenn ihr auch hier gerne über eure Konfiguration selbst bestimmen und Bescheid wissen möchtet. Auch hier könnt ihr wieder darüber bestimmen, ob euer Kernel nur auf eurer eingesetzten Hardware laufen soll, auf der gesamten Prozessorfamilie, oder auf allen unterstützten Prozessoren.
  Ob er die verwendete Hardware selbst erkennen soll, oder ob ihr im diese automatische Erkennung abnehmen wollt. Das verkürzt die Boot-Zeit, verringert die Bauzeit für den Kernel, und reduziert den Platzbedarf auf der Festplatte, da nur die benötigten Module, statt aller, vorhanden sein müssen. Die automatische Hardwareerkennung funktioniert nicht immer einwandfrei, das kennt ihr von eventuell benötigtem Blacklisting von Modulen her. Dieses Problem stellt sich nicht, wenn ihr nur benötigte Module in euren Kernel integriert. Auch kann dieser Minimalismus die Stabilität und die Sicherheit eures Kernels erhöhen.
  So bestimmt die gewählte Konfiguration nicht nur welche Hardware unterstützt wird, sondern auch maßgeblich das Verhalten des Kernels, z. B. ob Realtime-Kernel oder nicht, Kernel Based Modesetting, welcher Scheduler, welche Protokolle unterstützt werden etc.
  Auch hier gibt es im Grunde wieder nur ein Quellpaket pro Version, aber 1000 Möglichkeiten dies zu konfigurieren.
- Wem das zu viele Baustellen auf einmal sind, der möchte vielleicht lieber mit Genkernel beginnen.

- als Einstieg einige informative Links für Kernel-Newbees:

  - http://www.gentoo.org/doc/de/kernel-config.xml

  - http://de.gentoo-wiki.com/wiki/Kernel/Konfiguration

  - http://www.gentoo.de/doc/de/kernel-upgrade.xml

  - http://www.kernel-seeds.org/

- Konfiguration des restlichen Systems, wie z. B: **fstab**, Netzwerkkonfiguration, Wahl des Systemloggers oder des Cron-Daemons und Tools zur Datei-Indizierung. Achtung: hier ist das Handbuch nicht mehr ganz aktuell, denn das Paket heißt nicht mehr slocate, sondern mlocate. Auswahl des Bootloaders treffen, installieren, konfigurieren.

- **Neustart des Systems, Benutzerkonto anlegen und in die benötigten Gruppen eintragen.**

Gentoo wäre nicht Gentoo, wenn es hier irgendwelche Vorgaben machen würde.
Deshalb gibt es auch keine bestimmte Ausrichtung auf einen system-logger, cron-daemon, file-system, kernel-Quellen, WindowManager, Desktop-Umgebung etc...

Man versucht einfach alles zu unterstützen und überlässt dem User die Wahl (Gentoo is all about choice).
Zur revisionssicheren Verwaltung der .conf-Dateien empfehle ich das Portage-Tool **dispatch-conf**, welches u. a. im Handbuch Kapitel *4. a. dispatch-conf* beschrieben ist.
Das Handbuch bis zum Schluss durchzulesen, ist ohnehin sehr empfehlenswert, um weitere Details zu Funktionsweisen der Distribution zu erfahren.

Als Anlaufstelle für kompetente Hilfe solltet ihr euch nicht scheuen, egal ob deutsch- oder englischsprachig, das Forum in Anspruch zu nehmen. „Hier werden Sie geholfen ;-)“ http://www.gentoo.de/

Am Anfang war der Quellcode!

Open-Source-Software wird naturgemäß **zuerst als Quellcode** veröffentlicht, denn genau das ist es schließlich, was Open-Source ausmacht!

Die vor-kompilierten binären Pakete der Distributoren folgen dann Tage, Wochen, oder gar Monate später. Manche weniger populären Pakete schaffen es nie in das Repository mancher Distribution.
Das kann sich als Nachteil erweisen, wenn möglichst aktuelle Treiber oder Programme benötigt oder gewünscht werden.

Betrachtet man die Quellpakete als Ursprung, so sind diese vollständig und beinhalten einen gewissen Funktionsumfang und Abhängigkeiten. Einige davon sind zur korrekten Funktion zwingend erforderlich, andere sind optional und lassen sich durch zu übergebende Parameter an- oder abwählen.
Eine Regel, über die standardmäßig aktivierten oder deaktivierten Features, lässt sich nicht ableiten und wäre daher stets individuell zu prüfen, um davon Kenntnis zu erlangen (siehe auch ./configure –help).

Das vergleichbare Binär-Paket könnte dagegen in seiner Funktionalität beschnitten worden sein, indem bestimmte unterstützte Features abgeschaltet wurden.
Unterm Strich sind jedoch meist viele Features aktiviert, die häufig gar nicht benötigt werden, dafür jedoch eine Menge an ungenutzten Abhängigkeiten nach sich ziehen.
Diese belegen nicht nur ungenutzten Platz, sondern sie müssen auch teilweise mit geladen werden, was den Speicherbedarf und die Ladezeit erhöhen kann. Auch können sie bei ungünstiger Vorkonfiguration zusätzliche Sicherheitslücken aufwerfen.

Wenn der Distributor mehrere Hardware-Architekturen unterstützt, so baut er für jede dieser Zielarchitekturen separate Pakete, aus ein- und derselben Quelle, welche dann separiert in verschiedenen Repositories abgelegt werden.
Durch die Wahl der Architektur bzw. des architekturspezifischen Installers, merkt der User nichts von alldem, denn er bewegt sich mit seinem Paketmanager nur innerhalb seines architekturspezifischen Verzeichnisses.

Aber auch hier kommt es gelegentlich vor, dass sich in den Repositories scheinbar mehrere Variationen eines Paketnamens finden lassen, deren Namensgebung sich nicht immer auf Anhieb erschließt und den User oftmals verunsichert und somit zum Recherchieren nötigt.
In Extremfällen weicht dieser Name sogar stark vom Original Quellpaketnamen ab, wodurch er sich dann manchmal nicht mehr so einfach ermitteln lässt.

Hinter diesen Variationen verbergen sich meist mit unterschiedlichen Features kompilierte Binärpakete ein- und desselben Quellpaketes, die sich gegenseitig ausschließen oder an bestimmte Voraussetzungen gekoppelt sind.

Diese Namensvielfalt geschieht nicht unbedingt aus Willkür des Distributors, sondern ist eher als Versuch anzusehen, unterschiedliche Wünsche auf Anwenderseite zu befriedigen, oder aber durch Modularisierung der Pakete, deren Abhängigkeiten zu reduzieren.
So betrachtet müsste man alle diese Mehrfachnennungen ebenfalls abziehen, um auf die vergleichbare Quellpaketauswahl zu schließen.

Nehmen wir zu Beginn das Debian-Paket conky, welche unter lenny(oldstable) in der Version 1.6.0-2 vorlag, und dort noch schlicht conky hieß.

Mit welchen Features es erstellt wurde geht aus der derzeitigen online-Paketsuche nicht hervor, dort heißt es zur Zeit:
Paket: conky (1.6.0-2)
auf torsmo basierender Systemmonitor für X11 mit vielen Einstellmöglichkeiten
Conky ist ein ursprünglich auf dem Code von torsmo basierender Systemmonitor für X. Mittlerweile hat sich Conky weit von seinem Vorgänger entfernt. Conky kann praktisch alles anzeigen, entweder auf Ihrem Desktop oder in seinem eigenen Fenster. Neben vielen eingebauten Objekten bietet es die Möglichkeit der Ausführung von Programmen und Skripten, deren Ausgabe dann angezeigt wird.

Benötigte man seinerzeit ein nicht einkompiliertes Feature, so musste man das die Quellpakete von conky, aller zugehörigen Abhängigkeiten herunterladen, um es selbst mit geänderten Einstellungen zu kompilieren. Anscheinend hat man dem bei squeeze Rechnung getragen.

Die Pakete für das aktuelle stable (squeeze) lauten nun wie folgt:

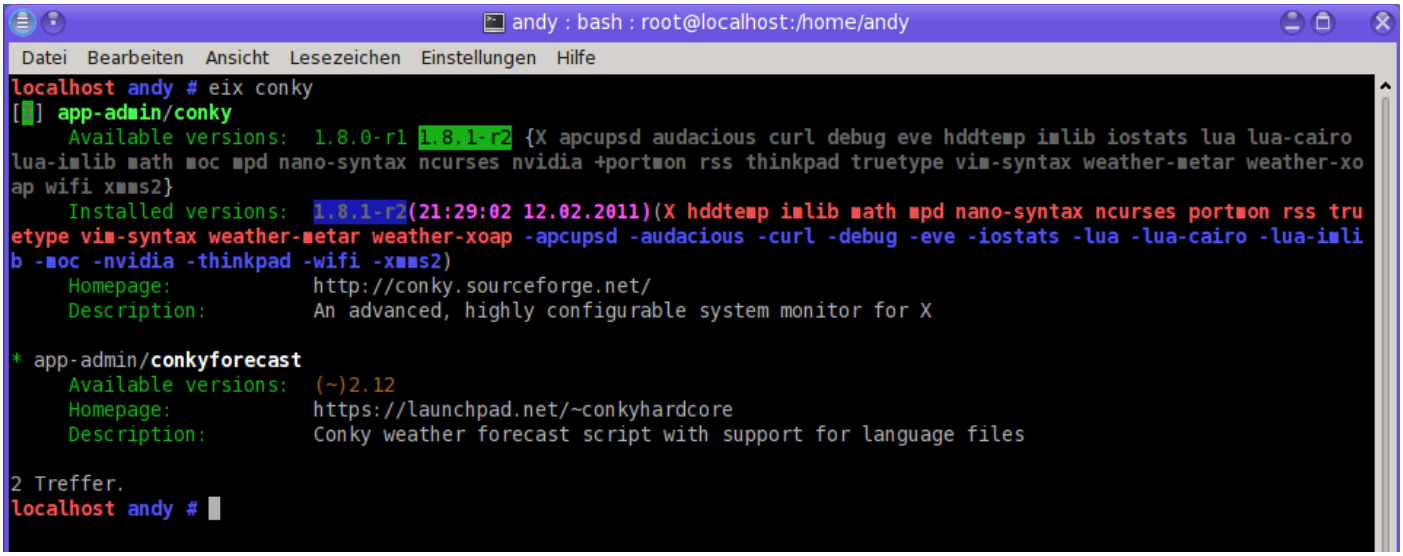
Paket: conky-all (1.8.0-1)
highly configurable system monitor (all features enabled)
Conky is a system monitor that can display just about anything, either on your root desktop or in its own window. Conky has many built-in objects, as well as the ability to execute external programs or scripts (either external or through built-in lua support).
This is a full conky with **most** compile options enabled:
X11, XDamage, XDBE, Xft, MPD, MOC, OpenMP, math, hddtemp, portmon, RSS, Weather, wireless, IBM, nvidia, eve-online, Imlib2, ALSA mixer, apcupsd, I/O stats, argb, Lua and the cairo and imlib2 lua bindings.
Use this if you are not sure what conky flavour you need.

Paket: conky-cli (1.8.0-1)
highly configurable system monitor (basic version)
Conky is a system monitor that can display just about anything, either on your root desktop or in its own window. Conky has many built-in objects, as well as the ability to execute external programs or scripts (either external or through built-in lua support).
This is a basic package that can be useful in servers or piped with dzen2. It includes the following support:
MPD, MOC, math, apcupsd, ncurses and I/O stats.

Paket: conky-std (1.8.0-1)
highly configurable system monitor (default version)
Conky is a system monitor that can display just about anything, either on your root desktop or in its own window. Conky has many built-in objects, as well as the ability to execute external programs or scripts (either external or through built-in lua support).
This package should be a good compromise for most users that do not need special features. It includes the following support:
X11, XDamage, XDBE, Xft, MPD, MOC, math, hddtemp, portmon, wireless, ALSA mixer, apcupsd, I/O stats, argb and Lua.

Bei Gentoo ist diese oftmals verwirrende Namensvielfalt nicht nötig, da die Features der Pakete über die USE-Flags gesteuert werden (dafür verwirren euch vermutlich zu Anfang die vielen USE-Flags).
Im Normalfall wird man also nur die Features (USE-Flags) auswählen, die man wirklich braucht und sich dadurch aller überflüssigen Abhängigkeiten entledigen.
Verdeutlichen möchte ich hier die Unterschiede in der Benennung zwischen USE-Flags und den Deklarationen der Paketoptionen bei Debian, sowie Gentoo's Erläuterungen zu den USE-Flags, um auf deren Funktionalität schließen zu können.

In u.a. Screenshot sehen wir, dass uns das Paket Namens **conky** in den stabilen Versionen **1.8.0-r1** und **1.8.1-r2** zur Verfügung steht.
Installiert ist Version **1.8.1-r2** mit den Rot gekennzeichneten aktiven(+) USE-Flags, die Blauen (-) wurden deaktiviert.



localhost andy # **equery uses conky**

[Legend : U - final flag setting for installation]
[: I - package is installed with flag]
[Colors : set, unset]
* Found these USE flags for app-admin/conky-1.8.1-r2:
U I
+ + X : Adds support for X11
- - apcupsd : enable support for sys-power/apcupsd
- - audacious : enable monitoring of music played by media-sound/audacious
- - curl : Adds support for client-side URL transfer library
- - debug : Enable extra debug codepaths, like asserts and extra output. If you want to get meaningful backtraces see http://www.gentoo.org/proj/en/qa/backtraces.xml
- - eve : enable support for the eve-online skill monitor
+ + hddtemp : Enable monitoring of hdd temperature (app-admin/hddtemp)
+ + imlib : Adds support for imlib, an image loading and rendering library
- - iostats : enable support for per-task I/O statistics
- - lua : enable if you want Lua scripting support
- - lua-cairo : enable if you want Lua Cairo bindings for Conky (also enables lua support)
- - lua-imlib : enable if you want Lua Imlib2 bindings for Conky (also enables lua and imlib support)
+ + math : enable support for glibc's libm math library
- - moc : enable monitoring of music played by media-sound/moc
+ + mpd : enable monitoring of music controlled by media-sound/mpd
+ + nano-syntax : enable syntax highlighting for app-editors/nano
+ + ncurses : Adds ncurses support (console display library)
- - nvidia : enable reading of nvidia card temperature sensors via media-video/nvidia-settings
+ + portmon : enable support for tcp (ip4) port monitoring
+ + rss : Enables support for RSS feeds
- - thinkpad : enable support for **IBM**/Lenovo notebooks
+ + truetype : Adds support for FreeType and/or FreeType2 fonts
+ + vim-syntax : Pulls in related vim syntax scripts
+ + weather-metar : enable support for metar weather service
+ + weather-xoap : enable support for metar and xoap weather service
- - wifi : Enable wireless network functions
- - xmms2 : enable monitoring of music played by media-sound/xmms2
localhost andy #

Zum besseren Vergleich hier nochmal die bei Debian's conky-all aktivierten Optionen:

X11, XDamage, XDBE, Xft**, MPD, MOC,** OpenMP**, math, hddtemp, portmon, RSS, Weather, wireless, IBM, nvidia, eve-online, Imlib2,** ALSA mixer**, apcupsd, I/O stats,** argb**, Lua and the cairo and imlib2 lua bindings.**

Legende der gewählten Farben:

X bei Gentoo beinhaltet XDamage, XDBE, Xft argb von Debian.
Gentoos audacious xmms2 Unterstützung gibt es bei Debian nicht.
OpenMP Unterstützung gibt es bei beiden nicht, aber Debian weist sie noch aus.

Die rot markierte Angabe bei Debian scheint mir falsch, da OpenMP-Support entfernt wurde, was auch das changelog der u.a. conky -Website belegt.

Newest lines in ChangeLog

```
2009-07-18
        * Added compilation switch --enable-xoap to be able to use
        www.weather.com as a source of weather data (this avoids adding
        libxml2 as a required dependency for users that wish to use noaa
        and not weather.com)
        * Added support for X alignment across multi-lined objects (i.e., using
        $alignr with $exec)
        * Disabled OpenMP code until GCC's implementation stabilizes
        * www.weather.com can now be used as well as a source of weather data
```

Info

Die Implementation des An- und Abschalten von Funktionen kann vom „ebuild“-Skript individuell umgesetzt werden. In der Regel werden die Optionen mit Hilfe von Configure-Optionen oder Patches realisiert.

Ein **Ebuild Quelltext** von **Conky** als Beispiel, findet sich ganz am Ende meiner Beschreibung, für alle, die sich selbst von meinen oben gemachten Aussagen überzeugen möchten, oder die einfach nur mal sehen möchten wie so ein Ebuild aussieht.

Es sei ausdrücklich erwähnt, dass man sich mit dem Inhalt oder dem Schreiben von Ebuilds als normaler Anwender nicht auseinander setzen muss, da ständig mehr als 26000 aktuelle verfügbar sind.
Was dort nicht verfügbar ist, findet sich dann meist in einem Overlay.
Ansonsten kann man ein eventuell nicht verfügbares Ebuild, für das neueste Paket seiner Wahl, natürlich auch selbst schreiben. Wie das geht wird im Development-Bereich der englischsprachigen Doku beschrieben.

Anzahl der Pakete als Maßstab?

Debian nennt auf seiner Website die Rekord verdächtige Zahl von 29000 Paketen für seine Distribution. Diese Zahl kann aber nur zum Vergleich der Binär-Distributionen untereinander herangezogen werden und sichert Debian dadurch den ersten Platz in Sachen Paket-Anzahl.

Wenn es aber um den Vergleich der Anzahl der Pakete zwischen Binär- und Quell-basierenden Distributionen geht, so darf man hier Äpfel nicht mit Birnen vergleichen.

Genauso falsch wäre es, die Anzahl der Ebuilds zum Vergleich heranzuziehen, da es für jede Versionsnummer eines Programms, ein eigenes Ebuild gibt.
Oder noch besser, die theoretische Anzahl an Kombinationen der zu erstellenden Pakete, die sich aus der Zahl der Pakete, multipliziert mit der Anzahl der USE-Flags, ergeben würde.
Das sind die Möglichkeiten die dem Gentoo-User zur Verfügung stehen, und die dafür sorgen, dass quasi alle Gentoo-Installationen, so individuell wie ihre Nutzer sind (Gentoo is all about choice).

Hier zeigt sich, warum es aus Gentoo-Sicht keinen Sinn macht, alle diese möglichen Pakete tatsächlich auch Binär zu erzeugen. Die Datenmenge wäre viel zu Groß, die Vielfalt nicht mehr überschaubar bzw. beherrschbar. Es gäbe zu viele Nachteile, bei vollständiger Beibehaltung dieser Vielfalt und das alles nur um dem User die Kompilierzeit zu ersparen, die in heutiger Zeit immer weniger Gewichtung hat.
Hier kommt man als Binär-Distributor unweigerlich an einen Scheideweg, an dem man genötigt ist bestimmte Entscheidungen dem User Abzunehmen, um die Komplexität auf ein beherrschbares Maß zu reduzieren. Deshalb ist so ein Binär-Paket-System grundsätzlich mehr oder minder fremdbestimmt.

Für einen objektiven Vergleich der Paketauswahl, müsste man also die Anzahl der Quellpakete von Debian, der Anzahl der Pakete (nicht Ebuilds) von Gentoo gegenüberstellen.

Leider konnte ich selbst keine aktuelle Erhebung bei Debian durchführen, da sich mir der Zugriff auf das Debian Developer's Packages Overview http://qa.debian.org/developer.php nicht so recht erschloss. Vergleichbare Erhebungen aus dem Jahr 2009 für Debian scheinen mir jedoch relativ verlässlich:

16756 source packages (**32958** binary packages), count input file bezogen von dieser Site:
http://asdfasdf.debian.net/~tar/bugstats/?zack%40debian.org

Aktuelle Zahlen von Gentoo belegen findet man hier: http://packages.gentoo.org/categories/
Categories: 154 Packages: **14607** Ebuilds: 27197

Wie man sieht scheint der Unterschied so groß nicht. Augenscheinlich ist dagegen das durchschnittliche Verhältnis von Binärpaketen zu Quellpaketen von ca. 2:1. (aktuell bei Debian ca. 29000 Binaries /2)

Wie man daraus ersieht, verfügen beide Distributionen über eine nahezu ebenbürtige Programmauswahl.

Allen Erbsenzählern, die nun auf eine eventuelle Differenz zugunsten Debians abzielen möchten, entgegne ich folgende subjektive Behauptung:
Mir scheint es auch wichtig, wie einfach und risikolos der User sich all dieser Pakete bedienen kann. Abhängig davon in welchem Release-Zweig sich der User befindet (z.B. *stable*), wird er neu in die Repositories (z. B. testing oder experimental) eingeflossenen Pakete gar nicht wahrnehmen.
Schließlich benötigen diese den festgelegten Reifungsprozess, bis sie erstmalig bei *stable* eintrudeln.
Möchte der User sie vorher, also aus experimental oder testing nutzen, könnte er sein System gefährden wie bereits Eingangs erwähnt. Noch mehr gilt das für die Verwendung fremder Paketquellen.

Bei Gentoo kann man sich dagegen recht unproblematisch aus den unstable oder sogar fremden Quellen (Overlays) bedienen.

Probleme sind erfahrungsgemäß eher während der Kompilation, also noch vor der Installation zu erwarten, oder aber beim ausführen dieser wenig getesteten Programme. Das gefährdet jedoch i.d.R nicht das gesamte System, sondern betrifft dann höchstens diese wenig getesteten Programme.
Das eröffnet dem User auf einfache und sichere Art, den flexiblen Zugriff auf genau die stabile oder aktuelle Version, die er sich wünscht.

Viele dieser Overlays dienen Entwicklern als Spielwiese, zum testen eigener Ebuilds, bevor diese dann in den Portage-Tree aufgenommen werden.
Dort finden sich auch viele auf Anwendungsgebiete spezialisierte Overlays, wie z.B. science, VDR, etc.
Auch Anhänger des alten KDE-3.5x werden hier fündig, da nach der abgeschlossenen Migration zu KDE-4.x, die alte Version vollständig dorthin ausgelagert wurde.
So könnte man problemlos ein brandaktuelles Grundsystem, mit einem alten KDE-3.5.x betreiben.

Zur einfachen und übersichtlichen Verwaltung der Overlays, gibt es ein bewährtes Tool Namens Layman.
Es wird vom Gentoo-Entwickler Tobias Scherbaum programmiert und gepflegt, den vielleicht Einige durch sein Buch "Gentoo – Die Metadistribution" kennen.

Kompilieren / Paketbau

Quellcode > Compiler = Objektdatei > Linker = binary oder executable

Um ein ausführbares Programm zu erstellen (Binary oder Executable), benötigt man dessen Quellcode sowie die notwendigen Tools um den Code in ein Binary zu konvertieren.
Diese Set von Tools nennt man **Toolchain,** da der Code diese Tools wie in einer Kette (engl. chain) durchläuft, mit dem Ziel, am Ende ein ausführbares Programm zu erhalten.

Toolchain

Üblicherweise besteht diese Werkzeugset (**tool chain**) aus einem **Text Editor** (zum schreiben des Codes), einem Compiler (zum übersetzten dieses Quellcodes in Maschinenlesbare Sprache, also Maschinencode), und einem Linker (um den Maschinencode von verschiedenen Quellen bzw. Orten, inklusive den eventuell vor-kompilierten dynamischen Bibliotheken[shared libraries], zu einem einzigen ausführbaren Programm mit Bibliotheken zu verbinden).
Hier weitere Details zu möglichen Komponenten einer Toolchain oder der GNU-Toolchain.

Beim Bau der Pakete aus dem Quellcode, werden diese nach dem übersetzen dem Linker übergeben.
Dieser linked u.a. die im Header des Sourcecode deklarierten Aufrufe der Systembibliotheken, gegen die auf dem Build-System installierten Versionen dieser Systembibliotheken, weshalb fortan eine Abhängigkeit zu diesen besteht.
Auf dem System auf dem das Paket erfolgreich gebaut wurde, sind daher alle Abhängigkeiten erfüllt, ganz gleich ob es sich dabei um das System des Users, oder um das Referenzsystem des Distributors handelt.

Wenn nun der Nutzer einer binären Distribution ein Paket installiert, welches gegen andere Versionen von Systembibliotheken gelinkt ist, als die seines eigenen, bzw. seines zugrunde liegenden Referenzsystems, dann wird er früher oder später auf Fehler stoßen.

So etwas kommt meist dadurch zustande, dass der User Repositories unterschiedlicher Releases einer Distribution beliebig mischt, oder zusätzlich fremde Repositories einbindet.

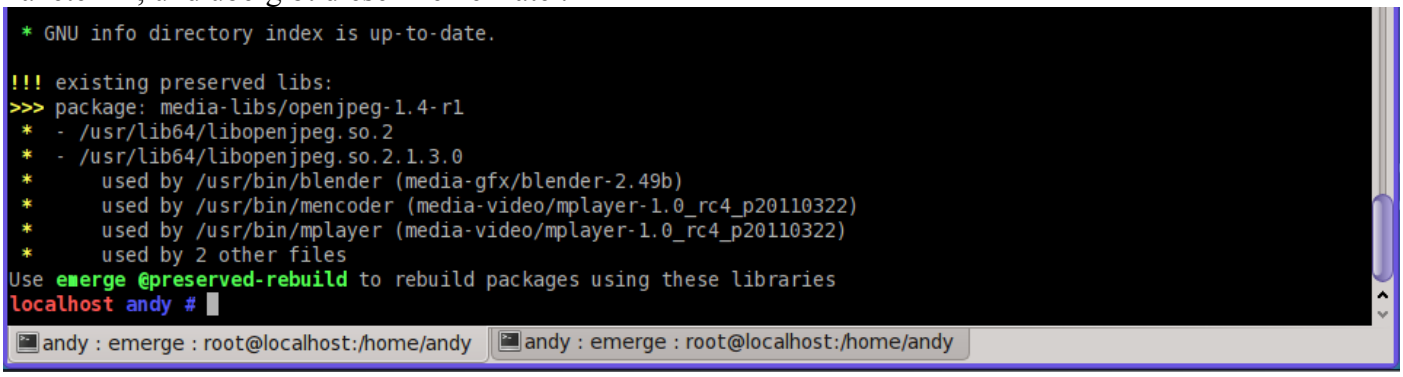
Wann diese Fehler dann in Erscheinung treten hängt von der Art der Bindung (statisch oder dynamisch), sowie von der Art der Bibliothek ab (Quelltextbibliothek, statische- oder dynamische Bibliothek).
Abhängig davon kann der Fehler bei der Installation des Paketes, beim Start des Programms, oder beim ersten Aufruf einer bestimmten Funktion, auftreten.
Im günstigsten Fall können solche Fehler durch manuelles relinken mit **ldd** korrigiert werden.
Oder sie können im Vorfeld durch parallele Installation verschiedener Versionen dieser Library an verschiedene Orte vermieden werden.
Im ungünstigsten Fall müsste das Paket neu gegen die auf dem System vorhandenen Bibliotheken gebaut werden.

Der User muss diesem Umstand durch entsprechende Maßnahmen mit der nötigen Sachkenntnis Rechnung tragen, um sein System konsistent zu halten, wenn er von den Vorgaben seiner Distro abweichen möchte.

Solange der User sich aus dem Standard-Repository seiner Distribution bedient ist die Konsistenz des Systems weitestgehend gewährleistet.
Der Distributor wiederum entwickelt sein System in einem jüngeren Zweig solange weiter, bis er es für ausgereift genug hält(freeze), es dem User als Ganzes zur Verfügung zu stellen.
Erst dann kann dieser sein System als ganzes zuverlässig Upgraden, weil dabei dann die zentralen Bestandteile der Distribution gemeinsam ausgetauscht werden.
Das ist der Grund, warum die meisten Binär-Distributionen mit Release-Zyklen arbeiten, bzw. ihre Aktualisierung stufenweise erfolgt.

Gentoo dagegen arbeitet nicht versionsorientiert, sondern nach dem Rolling Release Prinzip, bei dem eine kontinuierliche Aktualisierung des Systems erfolgt.
Eventuelle Migrationsprobleme ergeben sich daher lediglich für einzelne Programmpakete, nicht aber für eine ganze Distributionsversion.
Der Gentoo-User kann also über jede verwendete Version eines Programms unabhängiger und risikoarmer entscheiden.

Daher kann man auch beliebig stabile mit Testing-Paketen mischen, oder auch einzelne Systemrelevante Libraries relativ unabhängig austauschen.
Nach deren erfolgreicher Kompilation werden diese erneut, gegen die auf dem System tatsächlich vorhandenen Libraries, gelinkt, wodurch deren Abhängigkeiten wiederum erfüllt sind.
Selbstverständlich kann es dabei auch zu Inkonsistenzen kommen, z. B. wenn Libraries aktualisiert werden, und bereits vorhandenen Programme noch gegen ältere Versionen dieser gelinkt sind.
Portage ab Version 2.2, weist jedoch am Ende von Installationsvorgängen auf die zu rekompilierenden Pakete hin, und übergibt diese in eine Datei.



Nach dem Aufruf von **emerge @preserverd-rebuild** werden diese rekompiliert, danach ist das System wieder konsistent.
Natürlich kann man sein System jederzeit manuell auf Konsistenz prüfen und reparieren lassen mittels des Tools **revdep-rebuild**.

Obwohl Gentoo auch mit verschiedenen binären Paketformaten umgehen kann, gibt es im Portage-Tree kaum eigenen binäre Paketdateien, von wenigen Sonderfällen abgesehen (aktuell ca. 85 Pakete).
Wozu auch, würde es doch Gentoo's Vorteile ad absurdum führen.

Gentoo kann aus seinen Benutzerspezifisch konfigurierten und installierten Programmen auch binäre Pakete erzeugen. Entweder nachträglich selektiv, oder standardmäßig nach der Kompilation bei der Installation eines jeden Paketes. Dies kann beispielsweise zur Backup-Funktionalität dienen, z. B. beim Wiederherstellen von Paketen ohne erneutes kompilieren, oder aber zum Deployment auf andere Rechner (binhost).
So kann man relativ leicht an eigene Bedürfnisse angepasste Pakete zentral auf einer Maschine bauen, konfigurieren und testen, um diese anschließend auf Rechner einer geschlossenen Nutzergruppe zu deployen. So wird man zum eigenen Paket-Maintainer oder je nach Umfang zum eigenen Distributor.

Flexible Maskierung

Anhand von Maskierungen ermittelt Portage, welche Paketversionen für das System installierbar sind.
So ist experimentelle Software in der Regel immer maskiert, damit auf einem stabilen System nur getestete Pakete installiert werden können und damit auch stabil bleibt.

Alle Ebuilds enthalten sogenannte **Keywords** (Schlüsselwörter) zur Bestimmung der Architektur und Stabilität.
Damit wird zwischen **stabilen(+)** Paketen, **maskierten(~)** für unstable Pakete, und gesondert maskierten Paketen **(M) = hardmasked**, für experimentelle Pakete unterschieden.

Stabile Paketversionen sind ausreichend getestet, auch im Zusammenspiel mit anderen Paketen.
Maskierte (~) Pakete sind weitestgehend stabil, sollten aber noch ausführlicher getestet werden.
Probleme sind nicht bekannt, können aber auftreten.
In Fällen, in denen Pakete die Stabilität des Gesamtsystems beeinträchtigen könnten, oder neuere Versionen sehr starke Veränderungen aufweisen, übernehmen die Entwickler sie häufig in hart maskierter Form in den Portage-Baum. Die Keywords werden nach einem festgelegten Reifungsprozess, sprich mit steigender Stabilität, nach und nach angepasst und schließlich entfernt.

Durch die Kennzeichnung der Ebuilds mittels dieser Keywords, bedarf es bei Gentoo keiner Trennung der Pakete unterschiedlicher Entwicklungsstände oder Architekturen in separate Repositories.
Stattdessen befinden sich die Ebuilds zum Bau der stabilen, instabilen und experimentellen Pakete gleichzeitig im Portage Tree.

Die unterschiedlichen Entwicklungsstände die sich in den entsprechenden Versionsnummern widerspiegeln, sind durch bestimmte Präfixe / Suffixe, und zusätzlich anhand von farbliche Kennungen zu unterscheiden.

Legend

- \+ - stable
- ~ - unstable
- **M** - hardmask

Mit dieser Klassifizierung der Pakete kann nun jeder selber entscheiden, ob er lieber ein stabiles System oder ein aktuelleres, dafür aber weniger getestetes System haben möchte.

Die grundsätzliche Auslegung des Systems in stable oder unstable, wird beispielsweise durch die Variable ACCEPT_KEYWORDS="x86" in der **make.conf** Datei gesteuert.
Diese stünde für eine stabil ausgelegte x86 Architektur, sie ließe also standardmäßig nur die Installation stabiler Pakete zu.
Würde man nun einzelne instabile(~) Pakete installieren wollen, so sollte man diese in die Datei **/etc/portage/package.keywords** wie folgt aufnehmen: **net-www/apache ~x86**
Sollten zur Installation, wegen der resultierenden Abhängigkeiten, weitere unstable Pakete erforderlich sein, so würde Portage einen entsprechenden Hinweis darauf geben, und die Installation verweigern.
Dann wären auch diese Pakete in die **package.keywords** einzutragen um deren Installation zu ermöglichen.
Natürlich gibt es verschiedene Vorgehensweisen ein Paket zu demaskieren und verschiedene Tools (autounmask) zur komfortablen Unterstützung dafür.
Hier soll es jedoch zum besseren Verständnis der Funktionsweise, nur der manuelle Weg beschrieben werden.

Würde man sein gesamtes System stattdessen auf unstable Paketen aufbauen, oder nachträglich darauf ändern wollen, so bräuchte man stattdessen in der **/etc/make.conf** lediglich eine Tilde voranstellen, also ACCEPT_KEYWORDS="~x86" und sein gesamtes System updaten.
In der **package.keywords** wären dann keine Einträge erforderlich, um stets die aktuellste Software installieren zu können.
Jedoch raten die meisten eher von dieser Vorgehensweise ab, wegen dem durchgängig wenig getesteten System, und der damit häufiger zu erwartenden Probleme und wegen des höheren Sicherheitsrisikos.

Auch ist es möglich, nur einzelne Paketversionen aus dem hart maskierten Bestand (experimental) freizugeben.

Vielleicht möchte man sich vorab informieren warum das Paket maskiert wurde.
Mit -B3 erhalten wir die drei Kommentarzeilen vor dem eigentlichen Eintrag.
Sie liefern eine kurze Begründung, warum das entsprechende Paket maskiert ist.

```
grep -B 3 x11-drivers/nvidia-drivers /usr/portage/profiles/package.mask
# Doug Goldstein <cardoe@gentoo.org> (24 Jan 2011)
# Masked beta versions
>=x11-drivers/nvidia-drivers-270
# Michal Januszewski <spock@gentoo.org> <1 Dec 2010)
# Beta NVIDIA drivers.  This is the first 260.x release that works on GF330M,
# so it might be useful for some users.
=x11-drivers/nvidia-drivers-260.19.26
```

Oder noch simpler mit dem Tool **equery**:

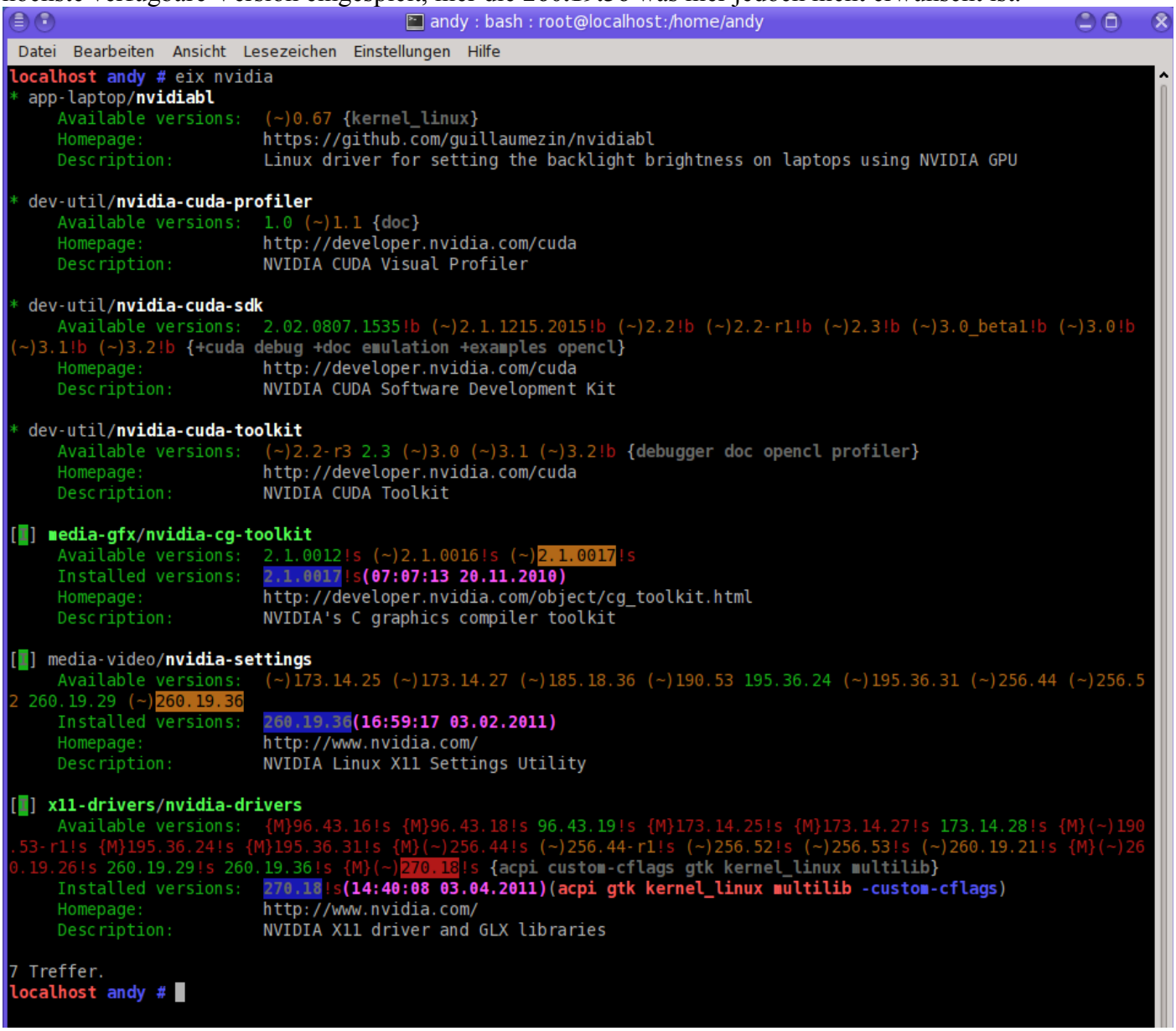
```
equery c nvidia-drivers
*nvidia-drivers-270.41.03 (12 Apr 2011)
  12 Apr 2011; Justin Lecher <jlec@gentoo.org>
  +nvidia-drivers-270.41.03.ebuild:
  Non-Maintainer Version Bump, #357113
  09 Apr 2011; Jeroen Roovers <jer@gentoo.org> nvidia-drivers-96.43.19.ebuild:
  Install nvidia-settings appropriate for this version (bug #304255).
```

Das Format der Datei package.unmask ist noch etwas einfacher als das der package.keywords.
Es reicht, den Paketnamen ohne weitere Zusätze in der **/etc/portage/package.unmask** anzugeben:
andy # **echo "=x11-drivers/nvidia-drivers-270.18" >> /etc/portage/package.unmask**

Das Blockieren von bestimmten Paketen (oder Versionen) kann mittels eigener Maskierungen durchgeführt werden, indem diese auf die gleiche Art, mit gleicher Syntax, in die **/etc/portage/package.mask** eingetragen werden.

Ein praxisgerechtes Beispiel für die Handhabung eigener Maskierungen.

Man besitzt z. B. eine sehr alte Nvidia Grafikkarte der GeForce4-er Serie.
Auf der Website des Herstellers erfährt man, dass es im Grunde nur einen Treiber-Namen gibt, der Abhängig von der Versionsnummer verschiedene Modellserien unterstützt.
Das gleiche Szenario spiegelt sich beim Ebuild im Portage-Tree wieder, dort lautet das Paket seit Urzeiten **x11-drivers/nvidia-drivers** und liegt wie üblich in verschiedene Versionen vor.
Für besagte GeForce4 Karte wäre die 96.xx.xx Versionen des Treibers zu verwenden.
Jetzt könnte man zwar eine bestimmte Version, z. B. die letzte stabile der 96.er Reihe direkt mit folgendem Befehl installieren: **emerge =nvidia-drivers-96.43.19.** Jedoch würde beim nächsten Update automatisch die höchste verfügbare Version eingespielt, hier die **260.19.36** was hier jedoch nicht erwünscht ist.



Stattdessen sollte man für diese Karten **>=x11-drivers/nvidia-drivers-97.00** in die Datei **/etc/portage/package.mask** eintragen, wodurch alle Pakete größer = 97.00 maskiert werden.

Ältere Karten der GeForce FX 5-Serie sollten die 173.xx.xx Treiber, zum Beispiel nvidia-drivers-173.14.28, verwenden. Für diese Karten sollten man **>=x11-drivers/nvidia-drivers-174.00** in die Datei **/etc/portage/package.mask** eintragen. Das verhindert, dass neuere Versionen des Treibers, welche inkompatibel zu dieser Karte sind, installiert werden, aber trotzdem Updates der 173.-er Serie aktualisiert werden dürfen. Neuere Karten, wie die GeForce 400, 300, 200, 100, 9, 8, 7 und 6 Serien, sollten die neuesten Treiber verwenden und benötigen solange keine Maskierung, bis Nvidia das nächste mal die Unterstützung dieser Karten, eventuell in künftigen neueren Treiber-Serien aufgibt.

Der obige Screenshot zeigt mit **`eix nvidia`** übrigens alle Pakete aus dem Portage-Tree an, deren Pakete mit dem Namen nvida beginnen. Nur 7 Treffer, das erscheint euch etwas dürftig?
Schließlich seid ihr von eurer Distro an eine viel größerer Auswahl gewohnt.

Wieder ein schönes Beispiel dafür, warum weniger manchmal mehr ist! Vergleichen wir...

Die letzten 6 Pakete der 7 o.a. Pakete (+2 Versionen der Ebuilds für die 96er und 173er Serie), entsprechen übrigens der gleichen Anzahl wie die, der Debian-Suche nach Paketen, deren Name *nvidia* enthält, in Suite *stable*, allen Bereichen, und auf allen Architekturen: 58 Treffer**,** bzw. eigentlich eher dieser:
Suche nach Paketen, deren Name *nvidia* enthält, in allen Suites, allen Bereichen, und auf allen Architekturen: 94 Treffer.

Das erste Paket des Screenshot oben (nvidiabl) konnte ich jedoch nicht bei Debian finden, bei Ubuntu dagegen schon.

Warum ist das so?

Nun, wenn man den nvidia-Treiber unter Gentoo installiert, baut er sich automatisch gegen die aktivierten kernel-quellen, und erzeugt/installiert dabei das nötige Kernel-Modul selbst. Somit benötigen wir unter Gentoo keiner der **nvidia-kernel-2.6xxx-Pakete**.

Hat man eine Grafikkarte die OpenGL, OpenCL und VDPAU unterstützt, und hat die USE-Flags **opengl** und **opencl vdpau** gesetzt, dann macht der nvidia-Treiber nach seiner Installation ebenfalls davon Gebrauch, da die entsprechenden Libraries bereits im System vorhanden sind.
Daher finden sich im Portage-Tree auch keine **libgl1- und libglx-xxx-Pakete,** keine **nvidia-libopencl1xxx** und auch keine **nvidia-libvdpau1xxx-Pakete,** wie hier bei Debian.

Daher gibt es auch zu den meisten dieser 58/94 Binären Pakete kein entsprechendes Quell-Paket mit gleichem Namen. Hier hat der Distributor, zwecks Modularisierung, viele kleine Binär-Pakete mit Libraries für spezifischen Support erstellt und zur besseren Unterscheidung versucht, möglichst Aussagekräftige Namen zu vergeben.

Wegen der hohen Anzahl und der geringen Unterschiede der Paketnamen, sowie dem schlecht ersichtlichen Verwendungszweck, leidet die Übersichtlichkeit doch sehr.
Deshalb ist für mich Gentoo klarer, weil logischer strukturiert und übersichtlicher, denn obwohl ich mit weniger Pakettreffern erschlagen werden, habe ich doch die größere Auswahl.

Mir geht es hier abermals darum zu zeigen, dass sich die Paketauswahl stark reduziert, wenn man die Pakete auf ihre Quell-Namen reduzieren würde, um sie objektiv mit der Anzahl der Gentoo-Pakete vergleichen zu können.

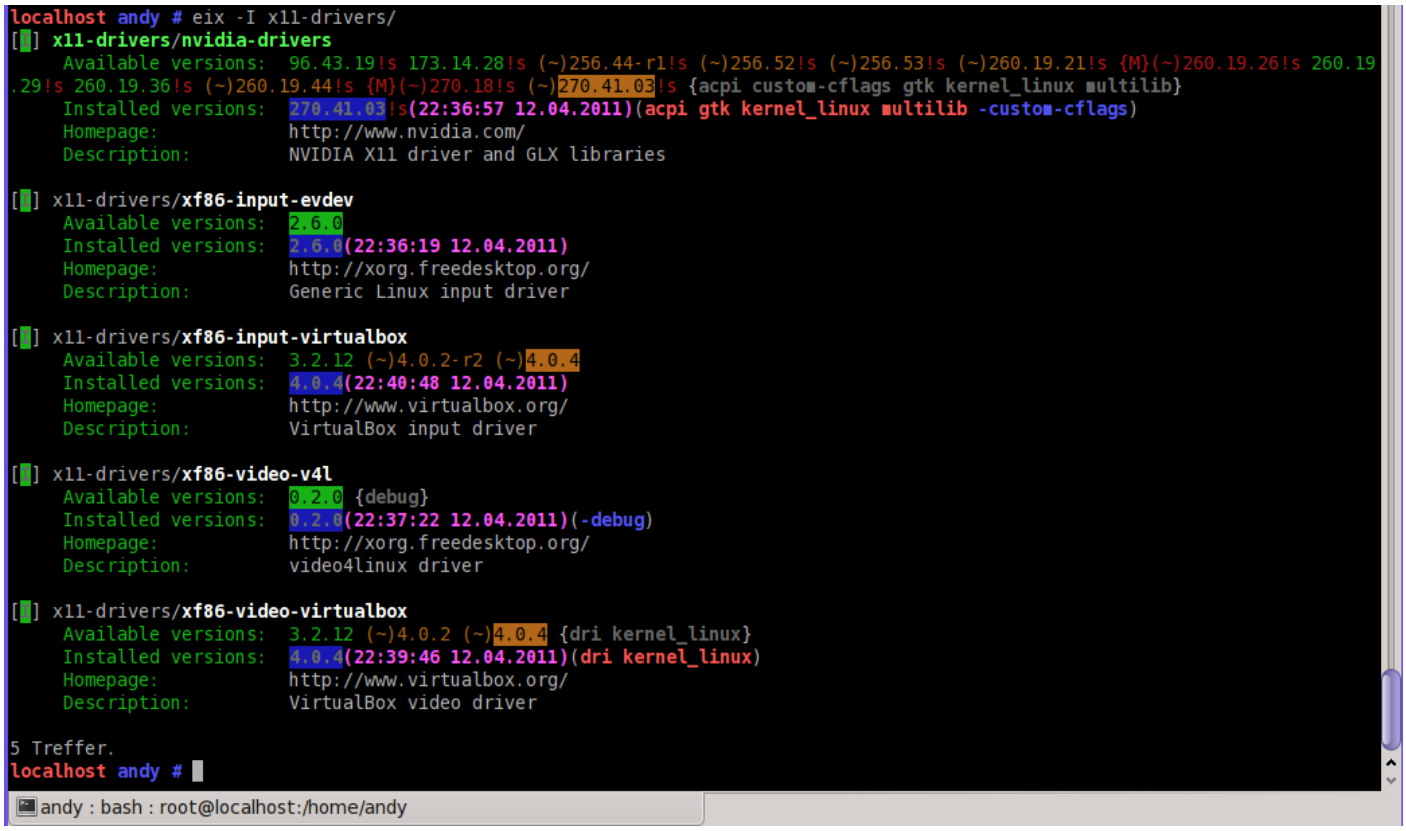
Nun handelt es sich bei dem obigen Beispiel ja um einen proprietären Treiber, der daher nur über wenige USE-Flags verfügt.
Zur Ergänzung des obigen Beispiels möchte ich noch die Verwendung des X-Servers aufzeigen, der u.a. ja noch zur Anzeige von Grafik erforderlich wäre.

Bei den binären Distros wurde noch vor kurzer Zeit ein "großer" X-Server mit implementierter Unterstützung für sämtliche Grafikkarten dieser Erde installiert, obwohl ja meist nur eine einzelne physikalisch vorhanden war. Das war ein ziemlicher Ressourcen-Killer.
Mittlerweile haben das auch viele Distributionen erkannt, und tragen dem Rechnung indem sie den X-server in gemeinsame Kernkomponenten und Hardwarespezifische Pakete zerlegen.

Somit kommt wieder eine Fülle von Paketen zustande, aus denen der User wählen kann, um sein nun modularisiertes X-System zusammen zustellen.
Ihre Suche nach Paketen, deren Name *xserver* enthält, in Suite(s) *stable*, allen Bereichen, und auf allen Architekturen: 76 Treffer.
Ihre Suche nach Paketen, deren Name *xserver* enthält, in allen Suites, allen Bereichen, und auf allen Architekturen: 100 Treffer.

Bei Gentoo wurde von jeher ohnehin nur die Hardwareunterstützung in den X-Server eingebaut, die auch benötigt wurde.
Da es aber auch hier (wie auch in den Quellen) eine Vielzahl von x11-Treibern gibt, die jeweils durch ein eigenes USE-Flag unterstützt werden, wurde es zunehmend schwerer diese aus der Fülle aller Flags zu unterscheiden.

`localhost andy # eix x11-drivers/`
74 Treffer. #Für alle x11-drivers, Liste viel zu lang, daher nur den Screenshot der installierten x11-drivers mit dem Befehl
`eix -I x11-drivers/`



Das wurde auch Gentoo irgendwann zu unübersichtlich, weshalb sie das Konzept der erweiterten USE-Flags ersannen.

Erweiterte USE-Flags

Die Vielzahl verschiedener Grafikkarten-Treiber des X-Servers waren der hauptsächliche Anlass für das Konzept der **erweiterten USE-Flags**.
Sie sorgen für eine Untergliederung aller USE-Flags in bestimmte Kategorien, und damit für mehr Übersichtlichkeit.
Alle verfügbaren expanded USE-Flags finden sich unter **USE_EXPAND** hier beschrieben:
/usr/portage/profiles/base/make.defaults

Sinnvollerweise sollte man sie in der zentralen make.conf Datei deklarieren, Beispiele:

LINGUAS="de en"
INPUT_DEVICES="evdev" **#keyboard, mouse** wurden von evdev abgelöst
VIDEO_CARDS="nvidia v4l" **# v4l**
ALSA_CARDS="hda-intel"
CAMERAS="agfa_cl20 casio_qv dimagev dimera3500 kodak_dc120 kodak_dc210 kodak_dc240 kodak_dc3200 kodak_ez200 konica_qm150 panasonic_coolshot panasonic_dc1000 panasonic_dc1580 panasonic_l859 polaroid_pdc320 polaroid_pdc640 polaroid_pdc700 ricoh_g3 sipix_blink sipix_blink2 sipix_web2 sony_dscf1 sony_dscf55 toshiba_pdrm11"
ACCEPT_LICENSE="$ACCEPT_LICENSE *"

```
localhost andy # eix xorg
[I] x11-base/xorg-drivers
     Available versions:  1.9 (~)1.10 {input_devices_acecad input_devices_aiptek input_devices_elographics input_devices_evdev
input_devices_fpit input_devices_joystick input_devices_keyboard input_devices_mouse input_devices_penmount input_devices_synap
tics input_devices_tslib input_devices_virtualbox input_devices_vmmouse input_devices_void input_devices_wacom video_cards_apm
video_cards_ark video_cards_ast video_cards_chips video_cards_cirrus video_cards_dummy video_cards_epson video_cards_fbdev vide
o_cards_fglrx video_cards_geode video_cards_glint video_cards_i128 video_cards_i740 video_cards_impact video_cards_intel video_
cards_mach64 video_cards_mga video_cards_neomagic video_cards_newport video_cards_nouveau video_cards_nv video_cards_nvidia vid
eo_cards_omapfb video_cards_qxl video_cards_r128 video_cards_radeon video_cards_rendition video_cards_s3 video_cards_s3virge vi
deo_cards_savage video_cards_siliconmotion video_cards_sis video_cards_sisusb video_cards_sunbw2 video_cards_suncg14 video_card
s_suncg3 video_cards_suncg6 video_cards_sunffb video_cards_sunleo video_cards_suntcx video_cards_tdfx video_cards_tga video_car
ds_trident video_cards_tseng video_cards_v4l video_cards_vesa video_cards_via video_cards_virtualbox video_cards_vmware video_c
ards_voodoo}
     Installed versions:  1.10(17:07:07 12.04.2011)(input_devices_evdev video_cards_nvidia video_cards_v4l -input_devices_aceca
d -input_devices_aiptek -input_devices_elographics -input_devices_fpit -input_devices_keyboard -input_devices_mouse -input_devi
ces_penmount -input_devices_synaptics -input_devices_tslib -input_devices_virtualbox -input_devices_vmmouse -input_devices_void
 -input_devices_wacom -video_cards_apm -video_cards_ark -video_cards_ast -video_cards_chips -video_cards_cirrus -video_cards_du
mmy -video_cards_epson -video_cards_fbdev -video_cards_fglrx -video_cards_geode -video_cards_glint -video_cards_i128 -video_car
ds_i740 -video_cards_impact -video_cards_intel -video_cards_mach64 -video_cards_mga -video_cards_neomagic -video_cards_newport
 -video_cards_nouveau -video_cards_nv -video_cards_omapfb -video_cards_qxl -video_cards_r128 -video_cards_radeon -video_cards_re
ndition -video_cards_s3 -video_cards_s3virge -video_cards_savage -video_cards_siliconmotion -video_cards_sis -video_cards_sisus
b -video_cards_sunbw2 -video_cards_suncg14 -video_cards_suncg3 -video_cards_suncg6 -video_cards_sunffb -video_cards_sunleo -vid
eo_cards_suntcx -video_cards_tdfx -video_cards_tga -video_cards_trident -video_cards_tseng -video_cards_vesa -video_cards_via -
video_cards_virtualbox -video_cards_vmware -video_cards_voodoo)
     Homepage:            http://www.gentoo.org/
     Description:         Meta package containing deps on all xorg drivers

[I] x11-base/xorg-server
     Available versions:  1.9.4 (~)1.9.5 (~)1.10.0.902 {dmx doc ipv6 kdrive minimal nptl static-libs tslib +udev xorg}
     Installed versions:  1.10.0.902(17:14:30 12.04.2011)(ipv6 nptl udev xorg -dmx -doc -kdrive -minimal -static-libs -tslib)
     Homepage:            http://xorg.freedesktop.org/
     Description:         X.Org X servers

* x11-base/xorg-x11
     Available versions:  7.4-r1
     Homepage:            http://xorg.freedesktop.org
     Description:         An X11 implementation maintained by the X.Org Foundation (meta package)

* x11-misc/Xorgautoconfig
     Available versions:  *0.2.4-r1
     Homepage:            http://dev.gentoo.org/~josejx/Xorgautoconfig.html
     Description:         Xorgautconfig generates xorg.conf files for PPC based computers.

[I] x11-misc/xorg-cf-files
     Available versions:  1.0.4
     Installed versions:  1.0.3(09:21:33 20.11.2010)
     Homepage:            http://xorg.freedesktop.org/
     Description:         Old Imake-related build files

5 Treffer.
localhost andy #
andy : bash : root@localhost:/home/andy
```

Wie man anhand der obigen beiden Screenshots sehen kann, wurde durch die Spezifizierung unserer Hardware mit den erweiterten USE-Flags durch:
INPUT_DEVICES="evdev"
VIDEO_CARDS="nvidia v4l"
unserer Vorgaben beim Bau des xorg-servers dahingehend berücksichtigt, dass gleich der Eingabetreiber für **event-dev** basierende Geräte wie z.B. Maus und Tastatur (**xf86-input-evdev**), der nvidia-Treiber (**nvidia-drivers**), und der Video for Linux Treiber (**xf86-video-v4l**) mit-installiert wurden.

Paketspezifische USE Flags verwalten

Statt eine einzelne Datei **/etc/portage/package.use** zu verwenden, lässt sich dieser Pfad auch als Verzeichnis anlegen.
Alle darin vorhandenen Dateien fügt Portage intern automatisch zusammen.
So lassen sich die Einstellungen thematisch besser gruppieren und verwalten, indem man z. B. **/etc/portage/pakage.use/**web-server dafür nutzt, USE-Flags spezifisch für den Webserver festzulegen, während man in **/etc/portage/package.use/**desktop die USE-Flags für Desktop relevante Pakete sammelt.
Nach dem gleichen Schema können auch für **/etc/portage/package.keywords** alternativ auch als Verzeichnis angelegt werden.

Übrigens, standardmäßig existiert keiner der /etc/portage/package.* Textdateien, bei Bedarf sind diese also anzulegen!

Warum Software selbst übersetzten?

Der Nutzen, der sich aus selbständiger Kompilierung von Paketen und deren Anpassung an eigene Hardware und Bedürfnisse ergeben kann, hat ohne Zweifel sehr viel mehr Vor- als Nachteile.

Die Nachteile sind schnell aufgezählt und sollen hier nicht verschwiegen werden.
- Zeitaufwand zum sichten der Paketoptionen und deren Selektion.
- Die zum kompilieren benötigte Zeit, die dabei verlorene Rechenleistung.

Allgemeine Vorteile:

- Mehr Kontrolle, denn ich bestimme selbst mit welchen Optionen meine Programme erstellt werden.
- Möglichkeiten zur Optimierung der Performance ausschöpfen bzw. selbst bestimmen, Paket-spezifisch.
- Mehr Flexibilität durch Unabhängigkeit zu fremdbestimmten Referenzsystem-Vorgaben.

Abhängig von der Art der gewählten Konfiguration:
- Weniger Abhängigkeiten= weniger Daten=kleinere Binaries=kürzere Ladezeiten – weniger Ballast.
- Ggf. mehr Sicherheit durch weniger laufende Dienste, sowie weniger unterstützte, aber nicht genutzte Protokolle.

Gentoo-spezifische Vorteile:
setzen sich zusammen aus allen o.g. allgemeinen Vorteilen, zuzüglich:

- Intelligente Paketverwaltung mit Berücksichtigung aller Abhängigkeiten
- Durch die große Auswahl an Ebuilds, braucht man eigentlich keine eigenen Buildscripte schreiben, und hat doch fast immer die neuesten Pakete zur Auswahl.
- Durch die Abstraktionsschicht der Paketoptionen zu den USE-Flags in den Ebuilds, und die intelligente Paketverwaltung, wird die **Systemweite** oder Paket-spezifische Steuerung sehr komfortabel, übersichtlich, und flexibel ermöglicht.
- Änderungen der Paketoptionen werden einem dadurch deutlich angezeigt (%Gelb*), dadurch braucht man nicht selbst umständlich und Paketweise sämtliche Optionen sichten bzw. verfolgen.
- Virtuelle Pakete und diverse Tools sorgen für Übersicht und Komfort auf Anwenderseite (z.B. eselect, eix, gentoolkit, ufed, etc.).

****

Kein Distributor weiß mit welcher Hardware ihr arbeitet.
Niemand kennt Eure Vorlieben und Wünsche besser als ihr selbst.

Daraus folgert mein Leitspruch:
Wenn du willst das etwas richtig gemacht wird, mach es selbst!

****

Quellen / Literaturempfehlungen

https://www.opensourcepress.de/openbooks.html Gunnar Wrobel Gentoo Linux

http://www.metadistribution.eu/ Tobias Scherbaum Gentoo Linux:Die Metadistribution

http://swift.siphos.be/linux_sea/index.html Sven Vermeulen Linux Sea

http://de.wikipedia.org/wiki/Wikipedia:Hauptseite

http://www.gentoo.org/

http://de.gentoo-wiki.com/wiki/Hauptseite

http://en.gentoo-wiki.com/wiki/Main_Page

http://www.debian.org/

http://asdfasdf.debian.net/~tar/bugstats/?zack@debian.org

cat conky-1.8.1-r2.ebuild

```
# Copyright 1999-2011 Gentoo Foundation
# Distributed under the terms of the GNU General Public License v2
# $Header: /var/cvsroot/gentoo-x86/app-admin/conky/conky-1.8.1-r2.ebuild,v 1.6 2011/03/06 12:11:08
klausman Exp $

EAPI=2

inherit eutils

DESCRIPTION="An advanced, highly configurable system monitor for X"
HOMEPAGE="http://conky.sourceforge.net/"
SRC_URI="mirror://sourceforge/${PN}/${P}.tar.bz2"

LICENSE="GPL-3 BSD LGPL-2.1 MIT"
SLOT="0"
KEYWORDS="alpha amd64 ppc ppc64 sparc x86"
IUSE="apcupsd audacious curl debug eve hddtemp imlib iostats lua lua-cairo lua-imlib math moc mpd
nano-syntax ncurses nvidia +portmon rss thinkpad truetype vim-syntax weather-metar weather-xoap wifi X
xmms2"

DEPEND_COMMON="
	X? (
		imlib? ( media-libs/imlib2 )
		lua-cairo? ( >=dev-lua/toluapp-1.0.93 x11-libs/cairo[X] )
		lua-imlib? ( >=dev-lua/toluapp-1.0.93 media-libs/imlib2 )
		nvidia? ( media-video/nvidia-settings )
		truetype? ( x11-libs/libXft >=media-libs/freetype-2 )
		x11-libs/libX11
		x11-libs/libXdamage
		x11-libs/libXext
		audacious? ( >=media-sound/audacious-1.5 dev-libs/glib )
		xmms2? ( media-sound/xmms2 )
	)
	curl? ( net-misc/curl )
	eve? ( net-misc/curl dev-libs/libxml2 )
	portmon? ( dev-libs/glib )
	lua? ( >=dev-lang/lua-5.1 )
	ncurses? ( sys-libs/ncurses )
	rss? ( dev-libs/libxml2 net-misc/curl dev-libs/glib )
	wifi? ( net-wireless/wireless-tools )
	weather-metar? ( net-misc/curl )
	weather-xoap? ( dev-libs/libxml2 net-misc/curl )
	virtual/libiconv
	"
RDEPEND="
	${DEPEND_COMMON}
	apcupsd? ( sys-power/apcupsd )
	hddtemp? ( app-admin/hddtemp )
	moc? ( media-sound/moc )
	nano-syntax? ( app-editors/nano )
	vim-syntax? ( || ( app-editors/vim app-editors/gvim ) )
	"
```

```
DEPEND="
	${DEPEND_COMMON}
	dev-util/pkgconfig
	"

src_prepare() {
	epatch "${FILESDIR}/${P}-nvidia-x.patch"
	epatch "${FILESDIR}/${P}-xmms2.patch"
	epatch "${FILESDIR}/${P}-secunia-SA43225.patch"
	epatch "${FILESDIR}/${P}-acpitemp.patch"
}

src_configure() {
	local myconf

	if use X; then
		myconf="--enable-x11 --enable-double-buffer --enable-xdamage"
		myconf="${myconf} --enable-argb --enable-own-window"
		myconf="${myconf} $(use_enable imlib imlib2) $(use_enable lua-cairo)"
		myconf="${myconf} $(use_enable lua-imlib lua-imlib2)"
		myconf="${myconf} $(use_enable nvidia) $(use_enable truetype xft)"
		myconf="${myconf} $(use_enable audacious) $(use_enable xmms2)"
	else
		myconf="--disable-x11 --disable-own-window --disable-argb"
		myconf="${myconf} --disable-lua-cairo --disable-nvidia --disable-xft"
		myconf="${myconf} --disable-audacious --disable-xmms2"
	fi

	econf \
		${myconf} \
		$(use_enable apcupsd) \
		$(use_enable curl) \
		$(use_enable debug) \
		$(use_enable eve) \
		$(use_enable hddtemp) \
		$(use_enable iostats) \
		$(use_enable lua) \
		$(use_enable thinkpad ibm) \
		$(use_enable math) \
		$(use_enable moc) \
		$(use_enable mpd) \
		$(use_enable ncurses) \
		$(use_enable portmon) \
		$(use_enable rss) \
		$(use_enable weather-metar) \
		$(use_enable weather-xoap) \
		$(use_enable wifi wlan)
}

src_install() {
	emake DESTDIR="${D}" install || die
	dodoc ChangeLog AUTHORS TODO || die
	dohtml doc/docs.html doc/config_settings.html doc/variables.html || die
```

```
	if use vim-syntax; then
		insinto /usr/share/vim/vimfiles/ftdetect
		doins "${S}"/extras/vim/ftdetect/conkyrc.vim || die

		insinto /usr/share/vim/vimfiles/syntax
		doins "${S}"/extras/vim/syntax/conkyrc.vim || die
	fi

	if use nano-syntax; then
		insinto /usr/share/nano/
		doins "${S}"/extras/nano/conky.nanorc || die
	fi
}

pkg_postinst() {
	elog "You can find a sample configuration file at ${ROOT%/}/etc/conky/conky.conf."
	elog "To customize, copy it to ~/.conkyrc and edit it to your liking."
	elog
	elog "For more info on Conky's features please look at the Changelog in"
	elog "${ROOT%/}/usr/share/doc/${PF}. There are also pretty html docs available"
	elog "on Conky's site or in ${ROOT%/}/usr/share/doc/${PF}/html."
	elog
	elog "Also see http://www.gentoo.org/doc/en/conky-howto.xml"
	elog
}
localhost conky #
```
