Solaris Compiler Tutorial

Erik Trauschke
erik.trauschke@freenet.de
http://erisch.homeunix.net

“Warum sollte man Programme aus den Sourcen heraus kompilieren wollen, wo es doch an so vielen Stellen im Internet so tolle Pakete oder Binärdistributionen gibt?",

das hab ich bis jetzt schon sehr oft gehört. Und es stimmt zum Teil auch. Es hat auch wesentliche Vorteile, fertige Pakete zu benutzen:

- ich muss die Software nur installieren und fertig
- es dauert nicht so lange wie selbstkompilieren
- ich muss nicht wissen, wie man kompiliert

Nur was ist wenn es genau das Programm, was ich brauche, nicht als fertiges Paket (welches ja auch irgendeiner mal kompilieren musste) gibt oder dass das Paket nicht so funktioniert wie es soll, wenn ich einhundertundeins Abhängigkeiten mitinstallieren muss, obwohl ich die garnicht brauche, wenn mein Verzeichnisbaum einem Chaos gleicht, weil ich Pakete von 5 verschiedenen Distributoren installiert habe, die alle irgendwoanders liegen ...

Gut, so schlimm wird es in den meisten Fällen nicht sein. Trotzdem gibt es noch andere Gründe warum selbstkompilieren vorteilhaft sein kann:

- ich kann die Software für meinen Rechner optimieren
- ich kann entscheiden, was ich bei einem Programm wirklich brauche (weniger Abhängigkeiten, kleiner Binaries)
- in den meisten Fällen funktioniert das Programm nachher so wie man es sich vorstellt

Noch etwas Allgemeines zum Tutorial:

Als Betriebssystem benutze ich hier Solaris (Version unerheblich, sollte von 7 bis 11 keinen Unterschied machen). Trotzdem ist der größte Teil auch für andere Unices sowie für Linux und BSD gültig. Allerdings kann es sein, dass diverse Programme, die ich anspreche, auf anderen OS nicht verfügbar sind oder einfach anders heißen. Desweiteren nutze ich als Shell die bash, wenn jemand eine andere Shell nutzt, kann es sein, dass einige Befehle anders ausgeführt werden müssen (z.B. statt export muss man setenv oder set benutzen).
Ich habe bisher unter Linux, Solaris, Free/Net/OpenBSD, Irix und HP-UX Programme kompiliert und bis auf wenige Systemeigenheiten sind die Grundprinzipien alle gleich.

So, nun aber genug der Vorrede, jetzt gehts los. Folgende Themen werden behandelt:

1. Werkzeuge

Bevor wir anfangen, benötigen wir natürlich noch die richtigen Werkzeuge. Und die bestehen nicht nur aus dem Compiler. Es gibt eine Menge anderer Tools die nötig und/oder sehr hilfreich sind. Nun haben wir nur ein Problem: Eigentlich wollen wir unsere Programme selbst kompilieren, haben aber noch gar keine Werkzeuge dafür. Hier helfen uns jetzt die Pakete, die andere Leute schon "gepackt" haben. Bei Solaris finden wir alles, was wir fast zum Starten brauchen auf der Companion CD.

Ich halte es dann immer so, dass ich mit der Grundausrüstung von der Companion CD mir alle Tools die ich später brauche selbst kompiliere.

Kommen wir nun zum Wesentlichen: Die Programme die hier aufgelistet sind, müssen nicht zwangsläufig genutzt werden, haben sich aber bei mir sehr gut bewährt.
Dieses Kapitel beschränkt sich auf die Vorstellung der einzelnen Programme, eine genaue Beschreibung wie diese zu benutzen sind, würde hier den Rahmen sprengen. Von Einigen der hier aufgeführten Programme, kriegt man im besten Fall gar nichts mit, da sie von anderen Werkzeugen aufgerufen werden und nur dann auffallen, wenn sie nicht richtig funktionieren.

Folgende Werkzeuge möchte ich hier etwas näher vorstellen:

- Sun Studio Compiler
  alternativ dazu
- GNU Compiler Collection GCC
- GNU make
- GNU tar
- GNU sed und GNU grep
- libiconv und GNU gettext
- andere nützliche Programme

Sun Studio Compiler

Dieser Abschnitt steht für alle professionellen Compiler die es von den Anbietern eigener Hardware (z.B. Sun, SGI, HP, Intel, ...) gibt. Diese Compiler sind exakt auf die verschiedenen Prozessoren zugeschnitten und erzeugen meist auch schnelleren Code, weil dieser die Features der jeweiligen CPU vollständig ausnutzen kann. Das Problem ist nur, dass die meisten dieser Compiler nicht wie der GCC als OpenSource zugänglich sind, sondern teuer (sehr teuer, wird sich für den Hobby-Programmierer kaum lohnen, die Preise gehen in den 4-stelligen Bereich) gekauft werden müssen. Sun bietet aber ihre Entwicklungsumgebung Sun Studio kostenlos zur Benutzung an, sofern man Mitglied in der OpenSolaris Community ist.
Die Benutzung dieses Compilers bereitet aber teilweise Schwierigkeiten, da die meisten OpenSource Programme für den GCC optimiert wurden und sich manchmal mit dem Sun Compiler nicht ohne manuelle Eingriffe übersetzen lassen. Zum einen wären da die Compilerflags die teils völlig anders lauten als die des GCC und außerdem gibt es gewisse Preprozessordirektiven die nur vom GCC richtig interpretiert werden können.
Ich empfehle aber jedem, der mit Solaris arbeitet und sich zutraut den ein oder anderen C/C++ Quelltext händisch anzupassen, diesen Compiler zu benutzen, weil er einfach die optimalsten Binaries vor allem für SPARC Maschinen erzeugt.

Die Sun Studio Collection kann man von der OpenSolaris Homepage beziehen:
http://www.opensolaris.org/os/community/tools/sun_studio_tools/

GCC

Die GNU Compiler Collection kennt wohl jeder, der irgendwas mit Linux oder Unix zu tun hat. Sie enthält neben dem normalen C Compiler auch noch Compiler für C++, Java, Fortran und noch weitere mehr. Desweiteren bringt sie eine handvoll Laufzeitbibliothken und Headerdateien, einen Assembler und einen Linker mit.
Wenn man noch sehr unerfahren im Kompilieren ist, ist er der beste Einstieg, weil die meisten Programme für ihn zugeschnitten sind. Deswegen wird man mit ihm die wenigsten Probleme haben. Mit den professionellen Compilern hat man schnell sehr viel Arbeit damit, die Makefiles und Quelltexte zu editieren um einen fehlerfreien Durchlauf zu erzielen.
Allerdings ist der GCC sehr auf die x86-Architektur optimiert. Bei anderen Architekturen wie beispielsweise SPARC oder MIPS ist die Unterstützung zwar da, aber meiner Meinung nach längst nicht so ausgereift wie das für x86 der Fall ist.
Den GCC gibt es für fast alle Betriebssysteme als Binärpaket. Für Solaris findet man ihn zum Beispiel auf der Companion CD, bei sunfreeware.com, blastwave oder anderen Paketdistributoren.

http://gcc.gnu.org

GNU make

GNU make oder make allgemein ist ein Tool, was es uns extrem vereinfacht, selbst komplexe Projekte mit nur einer handvoll Befehlen zu übersetzen. Die Aufgaben, die es zu erledigen hat liest make aus der sog. Makefile. Wie man damit umgeht erkläre ich später.
Hier kommt es vor allem auf das GNU an. Ein make bringen die meisten Unices mit, wenn man die Development Pakete bei der Installation mit auswählt. Nur funktionieren nicht alle make-Varianten gleich. Manche Programme lassen sich nur mit GNU make übersetzen, andere wiederum nur mit dem make was beim OS dabei war (das ist allerdings die Ausnahme). Informationen dazu findet man in den README oder INSTALL Files die fast immer im Quellverzeichnis des Programmes zu finden sind.
Ich hab mir angewöhnt, das GNU make als gmake abzuspeichern und das make des Betriebssystems so zu belassen. Das ist eine weitverbreitete Methode, die sich auch bei mir bewährt hat. Soweit nichts anderes in den readmes zu finden ist, sollte man dann immer gmake benutzen um etwas zu kompilieren.

http://www.gnu.org/software/make

GNU tar

Jetzt werden sich sicherlich einige fragen warum ich das noch extra brauche, weil tar wirklich zu jedem Unix dazugehört. Allerdings hat es gewaltige Vorteile, das GNU tar noch einmal selbst zu kompilieren und installieren. Meistens funktionieren nämlich bei den tars die beim OS dabei sind die -j und -z Schalter nicht, die es uns erlauben ohne Verwendung von langen Pipekonstrukten komprimierte tar-Dateien zu entpacken. Ein anderer Grund sind diverse Inkompatibilitäten (ich hatte schon tar-Archive die sich mit dem Solaris tar nicht entpacken ließen).
Außerdem ist tar sehr schnell kompiliert und hat kaum Abhängigkeiten, so dass es keine Mühe macht es schnell durch den den Compiler zu ziehen.

http://www.gnu.org/software/tar

GNU sed und grep

Auch diese Programme sind bei den meisten Unices dabei, sind aber teilweise inkompatibel mit den GNU Verwandten. Ich hatte schon Programme, die aus unerfindlichen Gründen nicht kompilieren wollten und nach dem benutzen von GNU sed oder GNU grep plötzlich fehlerfrei durchliefen.
Um all dem gleich von vornherein aus dem Weg zu gehen installieren wir die beiden kleinen Helferlein lieber gleich.

http://www.gnu.org/software/sed
http://www.gnu.org/software/grep

GNU libiconv und GNU gettext

libiconv ist eine Bibliothek, die zur Umwandlung von Textcodierungen in Unicode nötig ist. Die iconv Funktion benötigen sehr viele Programme. Auch hier gibt es meist schon eine Implementierung die beim Betriebssystem dabei ist und auch hier gilt wieder das was ich oben sagte: man spart sich Ärger wenn man die GNU libiconv installiert.
gettext wird für NLS (Native Language Support) benötigt. Das heißt, dass die Programme ihre Meldungen nicht nur in Englisch sondern auch z.B. in Deutsch anzeigen. Dieses Tool braucht man also nicht unbedingt, ich finde es aber sehr hilfreich.
Nur eines sollte man beachten: Diese beiden Programme befinden sich ganz unten im Abhängigkeitsbaum, d.h. wenn ich einmal angefangen habe, meine Programme mit libiconv/gettext zu kompilieren und irgendwann mal auf die Idee komme eins von diesen zu entfernen geht garnix mehr. Und es sind bestimmt 95% aller Programme die libiconv/gettext Unterstützung einkompilieren, wenn es vorhanden ist.

http://www.gnu.org/software/libiconv
http://www.gnu.org/software/gettext

andere nützliche Programme

Es gibt noch eine Reihe kleinerer und größerer Tools die für den Anfang weniger eine Rolle spielen aber für den einen oder anderen Fall nötig werden. Als erstes wäre da cvs zu nennen. Dieses Tool ermöglicht uns, einen kompletten Source-Tree von einem cvs-Server zu ziehen. Das ist nötig, wenn es für unsere Software kein Source-Archiv gibt oder wir die aktuellste Version eines Programmes haben wollen.
Wenn wir einmal mit cvs angefangen haben werden aber schnell noch andere Werkzeuge unverzichtbar: die GNU autotools. Diese werden benötigt um Makefiles und configure Dateien zu erstellen oder zu aktualisieren, wenn wir beispielsweise Quelletxte ändern.

http://www.gnu.org/software/cvs
http://www.gnu.org/software/autoconf

So, die Werkzeuge haben wir, jetzt müssen wir noch unser System konfigurieren ...

2. Die Systemumgebung

Das ist sicher der Teil, der dem die meisten (so hab ich es zumindest festgestellt) die geringste Beachtung schenken, obwohl er maßgeblich über Erfolg oder Misserfolg entscheidet. Ich selbst hab das am eigenen Leib erfahren müssen. Als ich am Anfang selbst noch keine Ahnung von all dem hatte (gut, viel Ahnung hab ich heute auch nicht, ich kann nur gut erzählen ...) musste ich bei fast jedem Programm herumrudern um es zum Laufen zu bekommen. Da hab ich dann angefangen selbst in den Makefiles oder gar in den Quelltexten herumzueditieren. Kompiliere ich exakt diese Programme heute wieder, dann muss ich garnichts tun und das Kompilieren läuft ohne Macken durch. Komisch - nicht?
Aber wenn man erst erkannt hat, wozu die ganzen Flags und Systemvariablen da sind, läuft alles wie von selbst. Nur findet man nirgens eine Information darüber, ich musste das alles selbst durch ausprobieren und Foren abklappern herausfinden.
Nun folgen die 4 Säulen des erfolgreichen Kompilierens: (klingt wie im Ethikuntericht, passt aber gut)

1. Ordnung im Verzeichnisbaum
2. Der Suchpfad für Programme
3. Der Suchpfad für Bibliotheken
4. Die Flags
(5.) für Faule - ein Kompilierprofil

1. Ordnung im Verzeichnisbaum

Ich bin sicher kein ordentlicher Mensch (jeder der schon mal in meine Zimmer geschaut hat, kann das bestätigen) aber was Ordnung der Verzeichnisse angeht, sollte man schon eine gewisse Grundstruktur erkennen können.

Zum ersten brauchen wir ein Verzeichnis, wo wir unsere selbstkompilierten Sachen einmal installieren wollen. Und dies sollte sich auf ein Verzeichnis beschränken (bis auf ein paar Ausnahmen). Bei mir hat sich dafür /usr/local bewährt. In dieses Verzeichnis werden z.B. auch die Pakete von sunfreeware.com installiert, die von blastwave gehen nach /opt/csw und die von der Companion CD nach /opt/sfw. Im Endeffekt muss jeder selbst wissen, wo er seine Programme hininstallieren will, aber wenn man dann irgendwann nicht mehr weiß, wo nun welches Programm ist, kann das nicht von Vorteil sein. Außerdem verliert man so schnell den Überblick ob man ein Programm vielleicht schon installiert hat mit der Konsequenz dass man vielleicht irgendwann manche Programme mehrfach auf dem Rechner hat.
Desweiteren sollten wir uns ein Verzeichnis anlegen, in dem wir unsere Quellen kompilieren. Ich nutze dafür /usr/src. Hier muss man aber aufpassen. Unter Solaris ist /usr/src nur ein symbolischer Link auf ein anderes Verzeichnis. Das kann zu Problemen bei manchen Programmen führen. Deswegen dieses Verzeichnis daraufhin überprüfen (ein ls -l gibt darüber Auskunft). Ich habe darum den Link /usr/src entfernt und das Verzeichnis mit mkdir erzeugt.
Beim Kompilieren von neuer Software sollte man es vermeiden nach /usr zu installieren, weil dort die ganzen Programme des Betriebssystems liegen. Diese Verzeichnisse sollte man also in Ruhe lassen.

2. Der Suchpfad für Programme

Ich gehe mal davon aus, dass die grundsätzliche Funktion der $PATH Variable und wie man sie setzt bekannt ist. Es gibt aber ein paar Dinge, über die man vielleicht noch nicht so genau nachgedacht hat. Da wäre zuerst die Suchreihenfolge von Programmen in den verschiedenen Verzeichnissen. Prinzipiell gilt: Das Verzeichnis was als erstes in der $PATH Variable angegeben ist, wird auch zuerst durchsucht - ist ja irgendwie auch logisch. Welche Konsequenzen hat das nun für uns?
Wie ich weiter oben schon andeutete kann es vorkommen, dass wir einige Programme doppelt auf unserem Rechner vorliegen haben; einmal das was mit dem Betriebssystem mitkommt und dann das was wir uns selbst kompiliert haben. Ein gutes Beispiel dafür ist grep, sed oder make (wenn wir uns entschieden haben, die GNU Variante nicht gmake zu nennen). Nun könnte man sagen, dass man einfach das eine Programm, was man nicht mehr braucht einfach löscht. Das ist sicher möglich, aber nicht immer einfach, weil einfaches Löschen unser System inkonsistent machen kann, und wenn man versucht das zugehörige Paket zu deinstallieren, vielleicht auch Programme die man noch benötigt mit entfernt werden.
Um dem Ganzen aus dem Weg zu gehen, passt man also seinen $PATH so an, dass das Verzeichnis worin sich die selbstkompilierte Version befindet, vor dem Verzeichnis in dem dem sich die andere Version befindet, durchsucht wird. Um mal direkt zu werden: Bei mir steht /usr/local/(s)bin immer als Erstes im Path, weil die Programme in dem Verzeichnis nur existieren, weil ich sie wirklich brauche und benutzen will.
In den meisten Profildateien wird der $PATH meistens nur ergänzt, d.h. es entstehen solche Konstrukte wie

```
PATH=$PATH:/opt/sfw/bin
```

Das kann allerdings zur Folge haben, dass man am Ende ein Verzeichnis mehrfach einbindet, weil der $PATH von mehreren Profildateien verändert werden kann.
Um nun zum Kompilieren einen sauberen $PATH zu haben, empfehle ich diesen explizit zu setzen, genau so wie wir ihn brauchen:

```
# export PATH=/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin
```

Das ist natürlich nur ein Beispiel. Der $PATH muss natürlich so angepasst werden, dass ALLE Programme die wir zum Kompilieren brauchen darin enthalten sind.

3. Der Suchpfad für Bibliotheken

Jetzt kommen wir zu einem sehr kniffligen Thema: dem Suchpfad für die Libraries. Um diesen festzulegen gibt es verschiedene Möglichkeiten. Die Vor- und Nachteile dieser Varianten möchte ich versuchen etwa zu erläutern.
Die erste und meines Erachtens auch eleganteste Methode ist die Benutzung des systemeigenen Linker/Loader-Konfigurationstools. Unter Solaris ist das crle, unter Linux nennt es sich ldconfig. Diese Tools setzen den Suchpfad systemweit, also nicht nur für bestimmte Nutzer. Wie man diese Tools benutzt steht in den man-files, deswegen werde ich hierrauf nicht weiter eingehen.
Die andere Möglichkeit ist die Benutzung des $LD_LIBRARY_PATH. Ich versuche diese Variante zu vermeiden, weil auf dieses Variable das Selbe zutrifft wie auf die $PATH Variable (und alle anderen Systemvariablen). Die Variablen können von verschiedenen Profildateien gesetzt werden und auch hier kann es passieren, dass Verzeichnisse mehrfach eingebunden werden. Das nächste ist, dass crle sowieso vorhanden ist und schon einige Einträge besitzt. Somit würden wir mit einem zusätzlichen $LD_LIBRARY_PATH nur Unordnung ins System bringen
Trotzdem hat der $LD_LIBRARY_PATH durchaus seine Daseinsberechtigung. Tools wie crle oder

ldconfig können nur mit root-Rechten ausgeführt werden. Wenn man also nur user auf einem System ist und verschiedene Bibliotheken irgendwo in seinem home-Verzeichnis installiert hat (weil man nur dort Schreibzugriff hat) kommt man um die Benutzung des $LD_LIBRARY_PATHs gar nicht herum.
Desweiteren setzt man ihn in Startscripten von umfangreichen Programmen ein, die ihre Bibliotheken, der Übersichtlichkeit halber, in anderen Verzeichnissen haben.
Falls es jemand ganz genau wissen will, kann er ja das hier mal lesen:

http://www.visi.com/~barr/ldpath.html

Der Autor dieser Seite geht sogar noch weiter. Er rät auch von der Benutzung von Tools wie crle und ldconfig ab. Seiner Meinung nach sollten alle Bibliothekspfade "hardcoded" in den Binaries stehen. Das setzt aber voraus, dass jeder die selbe Verzeichnisstruktur hat. Sobald man aber Programme auch anderen Leuten zum Installieren anbieten will, stößt man damit an Grenzen. Deswegen seh ich es ein wenig gelassener und hatte auch mit meiner Systemkonfiguration noch nie Probleme.

4. Die Flags

Mit den Flags können wir dem Compiler verschiedene Optionen mitgeben, beispielsweise über die Position von include-Dateien und Bibliotheken oder einfach zur Optimierung der zu kompilierenden Programme
Die wichtigsten Flags auf die ich hier eingehen möchte sind die CFLAGS, CXXFLAGS, CPPFLAGS und LDFLAGS. (es gibt natürlich noch mehr, aber die benötigt man nur in Ausnahmefällen)
Die eigentlichen Funktionen der einzelnen Flags beschreibe im nächsten Kapitel, hier soll es nur darum gehen, wie wir die Flags dem Compiler zugänglich machen.
Die Flags sind im Prinzip nichts anderes als Variablen, die entweder lokal in den Makefiles oder auch global als Systemvariablen definiert werden können. Für erste Variante müssten wir allerdings bei jedem Programm was wir kompilieren wollen die Flags von Hand jedesmal neu eintippen. Deswegen liegt es nahe sie einfach mit export (oder setenv, set, ... - je nach shell) als Systemvariable festzulegen:
export CFLAGS="-xtarget=native -xarch=v8plus"

Dabei sollte man immer die Anführungszeichen benutzen, das ist zwar nicht immer nötig (eigentlich nur wenn mehrere Optionen angegeben werden), aber wir ersparen uns wieder ne Menge Ärger wenn wir sie grundsätzlich setzen.

5. für Faule - ein Kompilierprofil

Da der Mensch von Natur aus faul ist, wäre es natürlich viel zu viel Arbeit, jedesmal wenn wir etwas kompilieren wollen, die ganzen Einstellungen von Hand einzugeben.
Deswegen ist es günstig sich eine Profildatei anzulegen, die alles auf einmal erledigt. Ich habe mir dafür in meinem home-Verzeichnis eine Datei namens .cprofile angelegt, die z.B. so aussieht:

```
CFLAGS="-xO3 -xtarget=native -xarch=v8plus"
CXXFLAGS="-xO3 -xtarget=native -xarch=v8plus"
CPPFLAGS="-I/usr/local/include"
LDFLAGS="-R/usr/local/lib -L/usr/local/lib -L/opt/SUNWspro/lib"
PATH=/usr/local/bin:/opt/SUNWspro/bin:/usr/ccs/bin:/usr/bin:/usr/sbin
export CFLAGS CXXFLAGS CPPFLAGS LDFLAGS PATH
```

Auf die Bedeutung der einzelnen Optionen gehe ich im nächsten Kapitel ein.

Bevor wir dann etwas Kompilieren wollen, müssen wir das Profil nur laden:

source ~/.cprofile

Ich würde nicht empfehlen, dieses Profil als Standard-Profil für den jeweiligen User zu benutzen, da der $PATH nur noch die zum Kompilieren nötigen Programmpfade enthält und somit möglicherweise einige Anwendungen nicht gefunden werden.
Dieses oder ein ähnliches Profil nutze ich nun schon geraume Zeit wenn ich Programme kompilieren möchte und seitdem ich das mache habe ich kaum noch Probleme mit Programmen die sich nicht übersetzen lassen.

Unser System ist nun auch unseren Wünschen entsprechend konfiguriert, wenden wir uns nun der Bedeutung der Compilerflags zu ...

4. Die Bedeutung der Flags

Für das was jetzt kommt habe ich selbst mindestens 2 Jahre gebraucht um es herauszufinden. Das Problem ist, dass man die richtige Benutzung der Flags nirgendwo in konzentrierter Form findet (zumindest hab ich bis jetzt noch nichts überzeugendes gefunden). Außerdem weiß man als Anfänger gar nicht, wonach man suchen muss. Um euch das Ganze zu ersparen gibts jetzt dieses Tutorial.
Wie schon im vorherigen Kapitel erwähnt, dienen die Flags dazu, dem Preprozessor, dem Compiler und dem Linker verschiedene Kommandos mitzugeben. Ich werde die wichtigsten Flags jetzt einzeln behandeln und Tips zum Umgang mit ihnen geben.

- 1. Präprozessor - CPPFLAGS
- 2. C Compiler - CFLAGS
- 3. C++ Compiler - CXXFLAGS
- 4. Linker/Loader - LDFLAGS

1. Präprozessor – CPPFLAGS

Bei meinen ersten Kompilierversuchen unter Solaris hatte ich immer das Problem, dass include-Dateien von Third-Party Paketen (z.B. von sunfreeware) nicht gefunden wurden, weil sie halt nicht unter /usr/include sondern unter /usr/local/include lagen. Ich habe mir sprichwörtlich einen Wolf gesucht um herauszufinden, wie ich dieses Verzeichnis mit in die Liste der zu durchsuchenden Verzeichnisse einfügen konnte. Ich habe dann Systemvariablen wie C(PP)_INCLUDE_PATH gefunden, die aber nicht so funktionieren, wie ich mir das vorstellte (außerdem war ich zu dieser Zeit noch fest der Meinung, dass **CPP**FLAGS den **C** **P**lus **P**lus Compiler steuern).
Ihr werdet damit ab jetzt keine Probleme mehr haben. Die Optionen die man in den CPPFLAGS angibt, werden direkt an cpp, den Präprozessor, weitergeleitet. Welche Optionen möglich sind, kann man in den Manuals zu cpp finden, die wichtigste Funktion der CPPFLAGS ist aber das Einbinden von include-Verzeichnissen. Ich selbst habe bisher auch noch keine anderen Optionen benötigt.
Um nun Verzeichnisse zum include-Suchpfad hinzuzufügen benutzt der Präprozessor des Compilers die Option -I (das gilt übrigens für GCC und Sun Compiler, sowie für alle anderen mir bekannten Compiler). Dabei ist zu beachten, dass die hier angegebenen Verzeichnisse VOR den vom System vorgegebenen include-Verzeichnissen durchsucht werden. Will man das verhindern, muss man also diese Verzeichnisse noch zusätzlich mit -I einbinden, obwohl es eigentlich nicht nötig wäre.
Eine CPPFLAGS Definition könnte also etwa so aussehen:

CPPFLAGS="-I/usr/local/include -I/opt/sfw/include"

2. C Compiler – CFLAGS

Die CFLAGS dienen hauptsächlich zur Optimierung der kompilierten Binary. Im Allgemeinen (also fast immer) kann man die CFLAGS weglassen, ohne dass daraus Probleme entstehen. Die Flags die wirklich nötig sind werden im Regelfall von den Makefiles oder von libtool gesetzt (z.B. -kPIC bei der Erzeugung von shared libraries) und wir müssen uns darum nicht kümmern.
Jedoch liegt in der richtigen Optimierung einer der Hauptargumente, warum man überhaupt selbst kompiliert. Ein paar oft benutzte Optimierungsflags möchte ich hier etwas näher betrachten. Dabei wird grundsätzlich zwischen plattformunabhängigen und -unabhängigen Flags unterschieden. Die plattformabhängigen Optimierungen bedienen sich vor allem der verschiedenen Features einer speziellen CPU. Bei SPARC-CPUs wäre hier vor allem die VIS-Funktionen (Visual Instruction Set - verfügbar ab GCC 4.0 oder Sun Studio) zu erwähnen, die verschiedene Multimedia-Programme beschleunigen können (z.B. xine).
Die plattformunabhängigen Optimierungen regeln eher den Umgang mit Registern und Debuginformationen. Eine Übersicht über alle möglichen Optimierungsflags finden sich in den Manuals und Dokumentationen der Compiler. Ich möchte hier nur einige wenige ansprechen, die ich auch selbst für meine Binaries oft benutze.

Flags zur Benutzung mit Sun Studio

-xtarget=system_type
-xarch=cpu_arch
-xchip=cpu_type

Mit diesen Flags gibt den CPU Typ, für den das Programm kompiliert werden soll. Im Normalfall wird nur -xtarget benötigt, weil es die anderen Flags automatisch setzt. Für xtarget lassen sich folgende Werte einsetzen:

native(64)	-	optimiert das Programm für den Typ Rechner auf dem gerade kompiliert wird. Dieses ist für Programme die man nur für sich selbst verwenden möchte sicher der beste Weg. Per Default werden immer 32 Bit Binaries erzeugt, allerdings gibt man mit native64 an, dass man 64 Bit Binaries haben möchte.
generic(64)	-	erzeugt Code der auf allen SPARC Plattformen lauffähig sein sollte (natürlich funktioniert generic64 nur mit UltraSPARCs). Dies ist die Default Einstellung des Compilers.
system_type	-	Diese Einstellung optimiert den Code für genau die Plattform die mit system_type angegeben wurde. Hier sollte man aufpassen und sich immer in Errinnerung halten, dass zum Beispiel eine für ein UltraSPARC III System kompilierte Software, auf einer alten SPARCstation nicht mehr laufen wird.

Eine Liste mit allen möglichen Werten für system_type gibts hier:
http://docs.sun.com/source/819-3688/cc_ops.app.html#18765

Einen kleinen Hinweis möchte ich noch geben: Falls man auf einer UltraSPARC Maschine die Option -xtarget=native benutzt, bekommt man ständig den Hinweis, dass das Programm mit älteren Maschinen nicht mehr benutzbar ist. Um diese Warnung zu unterdrücken, kann man mit -xarch=v8plusa dem Compiler explizit mitteilen, dass man sich darüber im klaren ist und erhält dann keine Warnung mehr.

Folgende weitere Möglichkeiten gibt es für xarch:

generic(64), native64	-	für diese gilt das selbe wie für xtraget
v8a, v8	-	anwendbar für alte 32Bit SPARCstations, die v8a Variante kann für MicroSPARC basierte Rechner benutzt werden SS4/5
v8plus, v8plusa,	-	erzeugt 32Bit Code mit UltraSPARC Extensions (VIS 1.0)
v8plusb	-	erzeugt 32 Bit Code mit UltraSPARC III Extensions (VIS 2.0)
v9, v9a	-	erzeugt 64Bit Code mit UltraSPARC Extensions (VIS 1.0)
v9b	-	erzeugt 64Bit Code mit UltraSPARC III Extensions (VIS 2.0)

-xO

Dies ist das universelle Optimierungsflag. Hinter dem O kann man eine Zahl zwischen 1 und 5 angeben, 1 für eine geringe Optimierung, 5 für maximale Optimierung. Allerdings sollte man sich nicht dazu verleiten lassen, jetzt alle Programme mit -xO5 zu kompilieren. Das Problem ist, dass die Binary zwar (möglicherwise) schneller wird, aber die Kompilierzeit, der Speicherbedarf beim Kompilieren sowie beim Ausführen des Programms mitunter stark ansteigen kann.
Als gutes Mittelmaß hat sich -xO3 bewährt.

Flags zur Benutzung mit GCC:

-mcpu=cpu_type
-mtune=cputype

Damit gibt man an, für welche CPU unser Programm kompiliert werden soll. Mögliche Varianten sind:

- supersparc
- hypersparc
- ultrasparc
- ultrasparc3

Das sind nicht alle, aber die die für die meisten am interesanntesten sein sollten. Daneben gibt es noch die Möglichkeit die Architektur für -mcpu anzugeben (v7, v8, v9), jedoch finde ich es am sinnvollsten immer gleich die CPU Implementierung anzugeben.
Der Standardwert für -mcpu ist v7. Dieser Code ist auf allen SPARC-Plattformen lauffähig.
Während man mit -mcpu den Befehlssatz für die entsprechnede Archtitektur auswählt, optimiert man mit -mtune die Ablaufsteuerung (scheduling) der CPU. Am sinnvollsten gibt man bei hier noch einmal die selbe CPU-Implementierung an, wie bei -mcpu.

Noch ein Hinweis an Besitzer einer Micro- oder TurboSPARC Maschine: Für diese beiden CPUs kann man -mcpu/-mtune die Option "supersparc" benutzen. Ob eine HyperSPARC Optimierung auch funktioniert weiß ich nicht, das muss ich bei Gelegenheit mal ausprobieren.

-m32
-m64

Mit diesen Flags können wir angeben, ob wir 32 oder 64 bit Binaries erzeugen wollen. Gibt man keinen dieser Parameter an, wird meines Wissens nach 32 bit Code erzeugt.
Ich möchte mich hier nicht über Sinn oder Unsinn von 64 bit Binaries auslassen, kann aber soviel sagen: 64 Bit Binaries sind etwa 5-50% größer als ihre 32 Bit Pendanten (und es damit auch länger dauert sie in den Speicher zu laden) und in 90% der Fälle auch langsamer. Ich empfehle es daher niemanden all seine Programme ohne vorherige Überlegungen in 64 bit zu kompilieren, nur weil die Maschine das kann. Ich halte im Privat-Anwender Bereich die Einführung von 64 bit sowieso

für absolut sinnlos, aber das steht hier nicht zur Debatte
Wer mehr über die Performance von 64 bit Programmen erfahren will, dem empfehle ich diese Seite:

http://www.osnews.com/story.php?news_id=5768&page=1

-Ox

Dies ist der generelle Optimierungsparameter. Er fasst verschiedene Optimierungsflags zusammen. Das x steht dabei für 0...3 oder s. Dabei entspricht -O0 für gar keine Optimierung und -O3 für maximale Optimierung, -Os dagegen für eine möglichst geringe Größe der entstehenden Binary.
Es gilt hier das gleiche wie für den Sun Studio Compiler: es ist abzuraten, jetzt grundsätzlich immer mit -O3 zu optimieren. Außerdem lassen sich manche Programme mit einigen Flags die -Ox setzt überhaupt nicht mehr kompilieren. Desweiteren kann ein Programm, was zu stark optimiert wurde, auch fehlerhaft oder garnicht funktionieren.
Ich selbst nutze für meine Binaries meistens -O2, weil das für mich einen guten Kompromiss zwischen Optimierung, Größe und Stabilität darstellt.

-ffast-math

Hier noch ein Beispiel für ein Flag, was man ohne weiteres NICHT benutzen sollte. Es beschleunigt mathematische Operationen, kann aber dazu führen, dass die Berechnungen am Ende nicht mehr stimmen.
Vor allem bei Programmen, bei denen exakte Ergebnisse elementar wichtig sind, sollte dieses Flag nicht benutzt werden.

Unter Sun Studio gibt es ein ähnliches Beispiel:

-fast

Dies ist ein Macro, was verschiedene Optimierungsflags setzt. Auch hier wird von Standards abgewichen und man sollte vorsichtig mit diesem Flag umgehen.

3. C++ Compiler – CXXFLAGS

Der einzige Unterschied zwischen den CFLAGS und den CXXFLAGS ist, dass die CXXFLAGS dem C++ Compiler zugeführt werden, während die CFLAGS an den C Compiler weitergericht werden.
Im Allgemeinen kann man hier die selben Flags angeben, wie bei den CFLAGS. Nur sollte man das auch tun, weil sich der C++ Compiler für die CFLAGS überhaupt nicht interessiert. Dies ist ein häufiger Anfängerfehler und man wundert sich dann beim Betrachten der Compiler-Ausgaben wo denn die Optimierungsflags hinverschwunden sind.

4. Linker/Loader – LDFLAGS

Die Flags zur Steuerung des Linkers haben im Wesentlichen 3 Aufgaben:

- Definition der Bibliothekssuchpfade für das Kompilieren (-L/path)
- Definition der Bibliothekssuchpfade, die das Programm später nutzen soll (-R/path)
- Einbindung von Bibliotheken um Symbolreferenzen aufzulösen (-lxxx)

Den Unterschied zwischen den ersten beiden muss ich wahrscheinlich noch etwas näher erläutern. Wir haben ja im vorherigen Kapitel schon den Suchpfad für Bibliotheken angepasst. Nun muss man aber wissen, dass es sich dabei nur um den Suchpfad für den runtime-linker, also den Linker, der zur Laufzeit die shared-libraries in das auszuführende Programm einbindet, handelt.
Fürs Einbinden von Bibliotheken zur Kompilierzeit ist allerdings ein zusätzlicher Suchpfad erforderlich. Und diesen setzen wir mit dem -L Flag, gefolgt vom gewünschten Verzeichnis:

```
LDFLAGS="-L/usr/local/lib -L/opt/sfw/lib"
```

Dabei weiß ich nicht genau ob man auch eine Pfadliste nach -L angeben kann, anstatt für jeden Pfad ein eigenes Statement zu schreiben.
Um nun auch den Pfad für den runtime-linker zu setzen, nutzen wir das Flag -R. Damit "hardcoden" wir quasi den Bibiothekssuchpafd in die Binary. Jetzt werden sich sicher einige Fragen wozu man das tun sollte, wo man diesen doch mit crle festlegen kann. Das ist eine gute Frage die ich selbst auch nicht ausreichend beantworten kann. Es gibt aber Leute, die diese Variante für besonders "sauber" halten, weil man sich unabhängig von crle oder einem LD_LIBRARY_PATH macht. Allerdings muss ich sagen, dass diese Methode dann an ihre Grenzen stößt, wenn man selbstkompilierte Programme weitergeben will und diese nicht an ihrem vorgesehenen Ort installiert werden.
Im Endeffekt muss also jeder selbst entscheiden, ob er lieber einen Systemsuchpfad nutzen will oder die Bibliothekspfade in die Binary programmiert. Ich nutze da einen Mittelweg, indem ich zumindest den Pfad, in dem das Programm später mal liegen soll (z.B. /usr/local), bei -R angebe. Noch ein Tip für die, die grundstzlich "hardcoden" wollen: Man sollte dann bei -R möglichst viele verschiedene Verzeichnisse angeben, in denen Bibliotheken liegen können.

```
LDFLAGS="-R/usr/local/lib"
```

Die dritte Aufgabe der LDFLAGS war nun das Einbinden zusätzlichen Bibliotheken. Das wird nötig, wenn das Programm was wir kompilieren Gebrauch von shared libraries macht. Dann sind nämlich nicht alle Funktionen im Quelltext des Programms sondern in einer oder mehreren extra Bibliotheken. Binden wir diese Bibliotheken dann nicht mit ein, bringt der Linker einen Fehler über nichtaufgelöste Symbole (unresolved symbols).
Normalerweise werden diese Bibliotheken vom configure-script gefunden und automatisch eingebunden. Aber das funktioniert vor allem dann nicht immer zuverlässig, wenn das Programm

nicht auf die Plattform vorbereitet ist, auf der wir es kompilieren wollen.
Der Syntax für das Einbinden ist einfach, wenn man ihn einmal erklärt bekommen hat. Das Flag was wir hier nutzen ist -l gefolgt von dem Namen der Bibliothek, die wir einbinden wollen, allerdings ohne das "lib" am Anfang und ohne Dateiendung.
Die Dateien in den Bibliothekspfaden wie /usr/lib haben fast alle die selbe Struktur. Sie fangen mit "lib" an, dann kommt der Name der Bibliothek und dann die Dateiendung, wobei ".a" oder ".la" für eine statische und ".so" für eine dynamische (shared) Bibliothek steht.
Wollen wir also z.B. libiconv.so einbinden muss das etwa so aussehen:

LDFLAGS="-liconv"

Dabei sollte man noch beachten, dass nur Bibliotheken eingebunden werden können, die sich im Suchpfad befinden. Will man also eine Bibliothek einbinden, die sich nicht unter /usr/lib befindet, muss man vorher noch ein -L Flag setzen. Ansonsten meldet der Compiler, dass die Bibliothek nicht gefunden werden kann.

Die drei verschiedenen Flags kann man natürlich auch alle zusammen festlegen:

LDFLAGS="-L/usr/local -R/usr/local -liconv"

Die Werkzeuge sind da, Systemumgebung stimmt, die Flags sind gesetzt. Jetzt wirds Zeit das wir mit der eighentlichen Arbeit anfangen...

5. Das Kompilieren

Ich hoffe ich habe euch bis jetzt nicht allzusehr gelangweilt. Aber jetzt gehts richtig los.

Es gibt verschiedene Varianten, wie eine Software konfiguriert, kompiliert und installiert werden kann. Daher ist es auch logisch, wenn ich nicht auf jede dieser Varianten eingehen kann. Ich möchte mich hier an das am weitesten verbreitete Prozedere halten, das von fast allen GNU - Programmen angewendet wird: der altbekannte Dreisatz:

./configure
make
make install

Diese Variante nutzen fast alle OpenSource Programme, aber Ausnahmen bestätigen die Regel. Hat man allerdings dieses Prinzip in etwa verstanden, kommt man auch mit anderen Kompilierverfahren klar.
Generell gilt aber, bevor man mit dem Kompilieren beginnt sollte man unbedingt einen Blick in die readme-Dateien werfen, weil dort oft hilfreiche Hinweise zu finden sind ohne die man unter Umständen nicht weiterkommt. Diese befinden sich im Quellenverzeichnis von GNU Programmen (und auch dem Großteil anderer OpenSource Programme) und heißen INSTALL und README.
Ich werde nun zu jedem Teilschritt ein paar Bemerkungen machen. Ich kann hier natürlich nicht auf jede Kleinigkeit eingehen. Für ein komplettes Tutorial über den Kompilierprozess würde man sicher etliche Gigabyte an Text schreiben müssen. Aber ich hoffe das nach diesem kleinen Tutorial hier jeder in der Lage ist, zumindest ein paar gutmütige Programme übersetzt zu bekommen.

- 1. Konfigurieren - ./configure
- 2. Kompilieren - make
- 3. Installieren - make install

1. Konfigurieren - ./configure

Bevor man mit make dem Kompilierprozess in Gang bringen kann, muss das configure-script die Makefiles mit den richtigen Informationen füllen. Hier geht es darum, herauszufinden welche Funktionen, Variablen und Programme auf dem System vorhanden sind und wo sich Header und und Bibliotheken befinden. Außerdem muss festgestellt werden welche speziellen Compilerflags nötig sind um beispielsweise Unterstützung für threads zu benutzen. Das configure-script ist ansich nichts anderes als ein shell-script welches verschiedene Compilertests durchführt und je nachdem, ob es funktioniert oder einen Fehler gibt, das Makefile anpasst.
Zusätzlich können wir das script mit diversen Optionen ausführen, mit denen wir beispielsweise die Position von benötigten Programmen angeben oder spezielle Programmfunktionen aktivieren und deaktivieren können. Eine Übersicht über alle möglichen Optionen bekomt man, wenn man configure so aufruft:

./configure --help

Das unterstützen zwar nicht alle, aber doch die meisten configure-scripte. Vor allem bei sehr umfangreichen Programmen kann diese Liste sehr lang sein, es stehen aber zu allen Optionen kurze Erklärungen dazu.
Die wichtigste Option mit der man umgehen können sollte ist --prefix. Damit legen wir fest, in welches Verzeichnis das Programm später installiert werden soll. Um die Prefix-Option richtig

anwenden zu können, sollte man zumindest die grobe Verzeichnisstruktur, in der die Programme später untergegebracht werden, verstanden haben. Im Grunde hat kein Programm ein eigenes Verzeichnis (das ist nur in Ausnahmefällen sinnvoll) sondern befindet sich, neben anderen Programmen, aufgeteilt in verschiedenen Unterverzeichnissen. Die wichtigsten Unterverzeichnisse sind ./bin für die ausführbaren Binaries, ./lib für die benötigten Bibliotheken, ./man für die Manual Dateien und ./include für die Headerdateien. Alle diese Verzeichnisse befinden sich normalerweise in einem Hauptverzeichnis (vor allem viele Linux-Distributionen machen das nicht so, es hat alles seine Vor- und Nachteile),und genau dieses geben wir bei --prefix an. Die Aufteilung der einzelnen Dateien in die Unterverzeichnisse erledigt dann das configure-script oder das makefile.
Die Prefix-Option wird folgendermaßen angewandt:

```
# ./configure --prefix=/usr/local
```

Die Voreinstellung für --prefix ist meistens /usr/local, jedoch gebe ich es trotzdem immer an. So spart man sich böse Überraschungen. Das Problem ist nämlich, dass sich durch die Aufteilung in die unterschiedlichen Verzeichnisse die meisten Programme nicht so einfach deinstallieren lassen. Es ist zwar möglich, führt aber hier zuweit. Deswegen sollten wir vorher überlegen wohin wir unsere Programme installieren - d.h. von /usr sollten wir möglichst die Finger lassen, weil da die Programme, die das Betriebssystem mitbringt, liegen.

Desweiteren lassen sich beim Konfigurieren auch die Flags aus dem vorherigen Kapitel setzen. Das ist dann nützlich wenn wir ein Programm mit anderen Flags als wir sonst benutzen, kompilieren wollen. Dazu werden die Definitionen für die einzelenen Flags einfach als Option dem configure-script mitgegeben:

```
# ./configure CFLAGS="-Os -mcpu=ultrasparc3"
```

Dabei sollte man auch immer die Anführungszeichen benutzen, weil es sonst passieren kann, dass nur die erste Option übernommen wird.

2. Kompilieren – make

Ist das configure-script dann durchgelaufen (und hoffentlich ohne Fehler) kann das eigentliche Kompilieren beginnen. Und da wir die 500 Quelltextdateien nicht einzeln durch den Compiler ziehen wollen, nutzen wir das make-tool.
Make sorgt dafür, dass jede Quelltextdatei mit den richtigen Flags kompiliert wird und das alle notwendigen Header und Bibliotheken eingebunden werden. Die Instruktionen dafür holt sich make aus dem Makefile, welches sich im Quellenverzeichnis befindet und vom configure-script hoffentlich mit den richtigen Werten gefüllt wurde.
Wenn alles so funktioniert wie es soll, reicht es einfach make ohne Optionen aufzurufen:

```
# make
```

Eine interessante Option für allem für Besitzer von Mehrprozessormaschinen ist -j. Mit dieser Option gibt man an wieviele jobs make gleichzeitig abarbeiten soll. So lässt sich im Optimalfall die Kompilierdauer auf einem Dual-CPU System halbieren. Allerdings lassen sich nicht alle Programme parallel kompilieren. Dies ist zwar meist in den readmes vermerkt, jedoch sollte man auch sonst bei einem Fehler einfach nochmal versuchen, das Programm ohne die -j Option zu kompilieren.
Die optimale Geschwindigkeit beim Kompilieren erhält man übrigens, wenn man make einen job mehr ausführen lässt, als CPUs im System vorhanden sind. Bei einem Dual-System sieht das ganze also dann so aus:

```
# make -j3
```

Falls es nötig werden sollte, können wir auch make noch Compiler-Flags mitgeben. Das wird dann nötig, wenn die Entwickler bei der Erstellung der Makefiles "geschludert" haben und beispielsweise eine benötigte Bibliothek nicht eingebunden wird. Hier sollte man aber aufpassen, dass man die Flags nicht absolut setzt sondern nur ergänzt, ansonsten werden alle anderen vorher gesetzten Flags nicht mehr berücksichtigt. Um nun zum Beispiel beim Linken noch zusätzlich die Bibliothek libiconv.so einzubinden macht man folgendes:

```
# make LDFLAGS="$(LDFLAGS) -liconv"
```

Das ist zwar nicht ganz sauber, weil nun auch jede andere Quelldatei mit libiconv verknüpft wird, ist aber meistens nicht tragisch, da die Bibliothek nicht verwendet wird, wenn keine ihrer Funktionen benötigt wird.
Hierzu noch ein kleiner Tip: manchmal muss man ein bisschen tricksen, damit es funktioniert. So kann es sein, dass die LDFLAGS nicht berücksichtigt werden obwohl gelinkt wird. Hier muss man das Statement in die C(XX)FLAGS schreiben, damit es funktioniert. Und auch dieses Verfahren führt nicht zwangsläufig zum Erfolg, vor allem dann wenn die Dateien mit Hilfe von libtool kompiliert werden.

3. Installieren - make install

Wenn wir das Kompilieren überstanden haben, sollte es beim Installieren keine Probleme mehr geben. Zu erwähnen wäre noch, dass wir jetzt die passenden Nutzerechte haben müssen um in die Verzeichnisse schreiben zu dürfen. Das werden in den meisten Fällen root-Rechte sein.
Für das Installieren sollten keine weiteren Optionen mehr nötig sein, ein einfaches

```
# make install
```

reicht aus.
Nachdem alle Dateien installiert sind, können wir das Programm dann testen ob es auch so funktioniert wie wir uns das vorstellen.

Damit wären wir dann auch am Ende dieses Tutorials. Ich hoffe es hat dazu beigetragen, den Umgang mit dem Compiler ein wenig zu erleichtern.

Falls noch Fragen offen sind, einfach nachfragen: erik.trauschke@freenet.de