Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

The Linux kernel as flexible product-line architecture

M. de Jonge

REPORT SEN-R0205 FEBRUARY 28, 2002

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

The Linux Kernel as Flexible Product-Line Architecture

Merijn de Jonge
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
`http://www.cwi.nl/~mdejonge/`

ABSTRACT
The Linux kernel source tree is huge ($>$ 125 MB) and inflexible (because it is difficult to add new kernel components). We propose to make this architecture more flexible by assembling kernel source trees dynamically from individual kernel components. Users then, can select what component they really need. Hardware vendors can easily support the Linux platform by providing additional separately developed kernel components.

We demonstrate how the kernel's build process can be setup for this approach and how individual build processes look like. We use a technique called *source tree composition* to assemble source trees of components. We demonstrate how it is automated by the tool `autobundle`. We also propose a kernel component base as central repository of kernel components. It forms a central meeting point for kernel developers and users.

1998 ACM Computing Classification System: D.2.2, D.2.7, D.2.9, D.2.10, D.2.11, D.2.12, D.12.13, D.2.m

Keywords and Phrases: source tree composition, reuse, kernel, linux, components, software product lines

Note: Work carried out under CWI project SEN1.2, Domain-Specific Languages, sponsored by the Telematica Instituut.

1. INTRODUCTION

The implementation of the Linux kernel can be considered as a product-line architecture since different products (a kernel together with drivers and subsystems for a particular hardware architecture) can be derived from a single source tree.

Unfortunately, this architecture makes the current kernel source tree *huge*: a kernel distribution is about 28 MB, an unpacked kernel source tree is about 125 MB. The kernel is so voluminous because everything is included: architecture independent code as well as code for *all* different architectures, *all* available subsystems, and for *all* available drivers.

The product-line architecture also makes the kernel *inflexible* because it cannot be extended or adapted easily. This is because i) build and configuration processes of all kernel components are integrated in a single build/configuration process [2]; ii) the variability of the kernel (the architecture to build a kernel for, and the set of drivers and subsystems to build) is completely defined by the (fixed) set of kernel components of a particular kernel distribution.

Inflexibility and its great size yields a number of problems: i) a huge source tree is required when only a small portion is really needed to compile for a single architecture with a small number of drivers; ii) adding new drivers is difficult because it requires adapting the kernel's build and configuration process [7]; iii) embedding new kernel components in the kernel's source tree requires acceptance by the kernel maintainers [8]; iv) a huge source tree needs to be maintained (it requires coding style [4], coding conventions etc. which might hamper code reuse); v) new drivers can only be provided as patch until they are included in the kernel's source tree; vi) it is difficult to maintain a driver separately when it is also included in the kernel's source tree; vii) it is difficult for hardware vendors to supply new drivers. Drivers have to be accepted by the kernel maintainers, or they should be provided as patch which requires users to have particular versions of the kernel source tree available. Alternatively, pre-compiled drivers can be provided, but they require particular versions of the kernel to be running.

We propose to make the product-line architecture more flexible: i) drivers, subsystems, and architecture specific code are developed and distributed as separate *source code components*; ii) specialised kernel source trees are *generated* from selections of such components; iii) a *kernel component base* collects kernel components; iv) hardware vendors can easily support the Linux OS by *donating* new drivers to this kernel component base.

The paper is structured as follows. We analyse the structuring of the Linux kernel in components and kernel configuration in Section 2. We discuss automatic composition of kernel source trees in Section 3. Composition of compiled kernel components to form executable kernels is discussed in Section 4. Implementation of the dynamic product-line architecture is addressed in Section 5. Section 6 describes results and future work.

2. KERNEL CONFIGURATION

Configuring the Linux kernel consists of selecting drivers for available hardware and of desired functionality such as support for different file systems and network protocols. This configuration is performed at compile time by issuing the command `make config` and determines the kernel components that should be built. A number of different types of kernel components can be distinguished:

Core kernel components. These are hardware and platform independent components which form the base of each running kernel and are required by each different kernel configuration. The Linux scheduler and the platform independent signal handling routines are examples of this type of components.

Architecture specific components. These contain functionality which requires different implementations on different hardware architectures. Configuration for a specific architecture occurs once at compile time. After building a kernel it cannot be configured dynamically for a different architecture. Examples are boot procedures and IRQ handling.

Kernel subsystem components. These are optional hardware and platform independent kernel components. The kernel can be configured to add support for several subsystems. Some subsystems can be dynamically added to, or removed from a running kernel. The network subsystem is an example of this type of kernel components.

Kernel drivers. These are hardware specific components that control hardware devices. Like kernel subsystem components, a running kernel can dynamically be configured to add new, or to remove existing drivers.

Basically, the configuration of the Linux kernel thus consists of two types of configurations:

Configuring what components to build. This configures the kernel's build process such that only the drivers, subsystems, etc. according to a user's selection are built.

Configuring how to build a kernel from them. This configuration is concerned with linking kernel modules into the kernel after they have been built.

Currently, both types of configurations are combined in a single kernel build process. This implies that build knowledge of all possible kernel components is already contained in this build process. Consequently, adding a new component requires adapting this build process. We propose to separate both types of configurations to yield a more flexible, extendible kernel:

Composition of kernel source trees. First, select the kernel components that are desired and then automatically generate a dedicated kernel source tree for this selection. The generated kernel source tree contains only the source trees of the selected (and required) components. It also contains a single build and configuration procedure that merges the build/configuration procedures of the individual components.

Composition of kernel components. Then, build the kernel components and link them into a kernel executable. Kernel modules can be linked either statically or dynamically (as loadable modules).

```
package
identification
         name=scsi
      version=0.1
     location=http://www.cwi.nl/~mdejonge/kernel-base/
         info=http://www.kernel.org/
  description='basic scsi driver package'
     keywords=linux, kernel, scsi, driver
configuration interface
requires
  kernel-base 0.1
```

Figure 1: A package definition for the generic SCSI driver.

3. Composition of Kernel Source Trees

Automatic assembling a kernel's source tree from those source code components that are really needed allows users to obtain a kernel specialised for their needs. Hardware vendors can easily provide support for the Linux platform by suppling new drivers as additional source code components.

Automatic assembling of source trees (called *source tree composition* [3]) consists of the following phases:

- Source code components are (formally) described in package definitions (see Figure 1).
- Users choose the components of need (by selecting package definitions).
- Components are merged into a self-contained source distribution.
- The distribution is unpacked, configured, and built.

Source tree composition performs dependency resolution to calculate which additional packages are required. It performs version checking to determine which versions of components to use. It integrates individual source trees into a single one. It also merges individual build and configuration processes. Finally, source tree composition supports generation of self-contained source distributions from particular component collections.

With source tree composition, individual kernel components can be developed and maintained separately. Merging them in different kernel configurations is completely automated. Source tree composition simplifies the configuration and build processes of such collections of source code components.

To enable dynamic assembling of Linux kernel source trees, we propose to develop kernel components individually, to define package definitions for each of them, and to use `autobundle`[1] to automatically assembling them to form kernel source trees. In Section 5 we discuss the implications for the development of kernel components.

4. Composition of Kernel Components

After kernel components have been built (compiled), they have to be linked to form an executable kernel. Three types of linking are distinguished in the kernel:

Statically linked with explicit initialisation. Components are linked into the kernel at build time. Explicit invocation of their initialisation routines is performed conditionally depending on whether the component is selected during configuration or not.

Statically linked with implicit initialisation Components are linked statically. Their initialisation routines are executed via a dynamic table of function pointers (heavily depending on specific gcc features [1]).

[1] `autobundle` implements automatic source tree composition. It is Free Software and available at: `http://www.cwi.nl/~mdejonge/autobundle`

Hence, there is no direct function invocation. Kernel drivers are typical examples of components that are linked this way.

Dynamic linked with explicit initialisation Initialisation is performed explicitly when a component is dynamically loaded. The kernel executable does not contain direct function invocations to such components. Components linked this way are also known as *dynamic loadable modules*.

Source tree composition requires independent build processes. Hence, a component's build process has no knowledge about the build process of another component. Consequently, there is no information available about what components to link statically in the kernel, and what the names of these components are.

For dynamic linked components this is no problem. They can be built, installed, and be loaded at run time. For static linked components we need a mechanism to register the names of kernel component to link (and their link order). We also need a mechanism to support conditional code which execution depends on component selection. In the next section we discus both mechanisms.

5. IMPLEMENTATION

In this section we briefly describe the implementation of an artificial componentized kernel. This kernel is structured similar to the Linux kernel and is used to test the concepts presented in this paper. We are currently applying these techniques to the 'real' Linux kernel.

We only consider the build process of the kernel and its components because, in case of Linux, this what needs to be adapted to fit in the product-line architecture that we propose. The architecture of the Linux kernel itself is already flexible enough to deal with different configurations and component compositions. Except for small changes related to conditional code, the source code of the kernel therefore needs no adaption.

Automake/Autoconf. The build processes of kernel components in our architecture make use of the tools `autoconf` [5] and `automake` [6]. These tools are required by `autobundle` because they provide standard build and configuration procedures (see [3] for details).

Kernel base package. general functionality is contained in the kernel base package. This functionality is required by all kernel components. It includes general build rules, build utilities, and general kernel include files (such as `init.h`).

Supporting conditional code. The kernel base package generates the file `kernel-config.h` at compile time. This file defines for each source code component `c` that is bundled the symbol `KC_c`. Code that should only be executed when a component `c` is bundled, can be embedded within `#ifdef KC_c ... #endif` preprocessor code.

Registering components to be linked statically. The base package implements a mechanism to statically link a varying collection of components in the kernel. The mechanism is based on registering kernel components during the kernel's build process. Each component that needs to be linked statically, registers itself using the tool `regdrv`. A general build rule in the base package performs this action during component installation:

```
install-data-local:
   regdrv $(driverdir) $(DRIVER) \
      $(DRIVER_PRIO)
```

Kernel components can be registered with priorities to force a particular link order (by setting `DRIVER_PRIO` to a non-zero value). The Linux kernel contains a similar feature, by fine-tuning the build order of components [1]. Our approach makes link ordering more explicit.

During the final link stage the tool `kernel-linker` is used to obtain a list of all registered kernel components:

[3]The kernel base is available at `http://www.cwi.nl/~mdejonge/kernel-base/`.

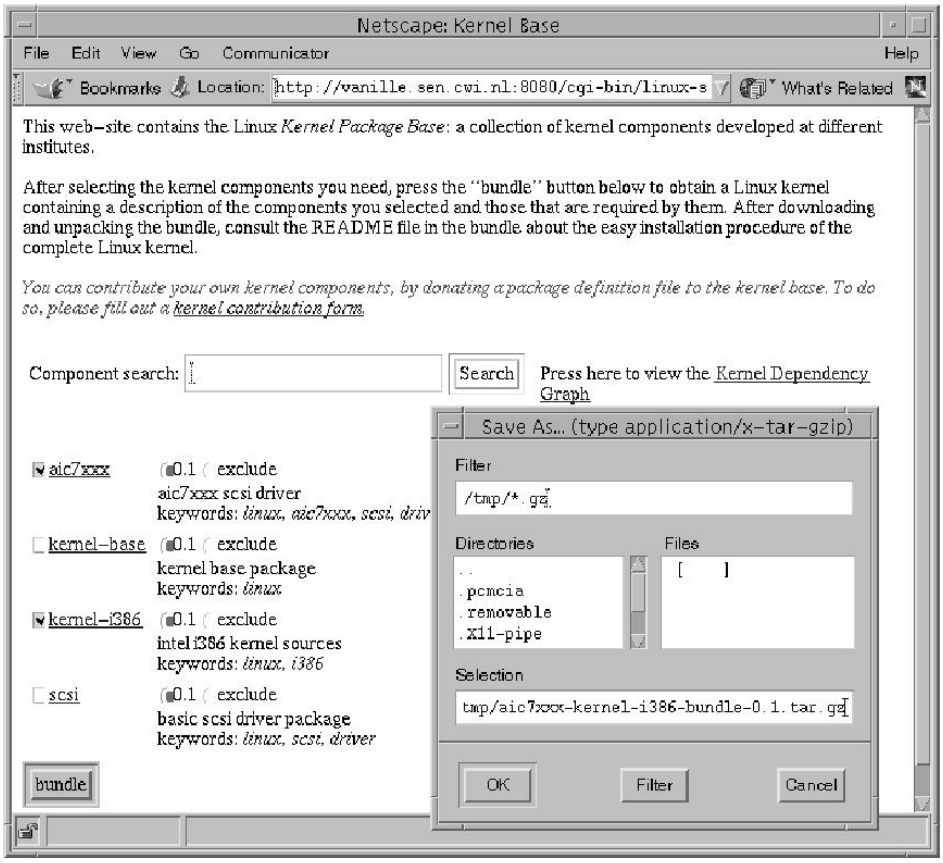

Figure 2: Screenshot of an experimental online kernel base[3] from which kernel components can be selected and kernel distributions can be generated.

```
KERNEL_OBJECTS := \
   $(shell kernel-linker $(driverdir))

kernel.o: $(KERNEL_OBJECTS)
        $(LD) -r $(KERNEL_OBJECTS) -o $@
```

Developing a component. Figure 1 contains an example package definition for the generic SCSI package. It defines a dependency upon version 0.1 of the kernel-base package, it briefly describes the package, and it expresses where the package can be retrieved. The location field of a package definition allows component distributions to reside anywhere. Only package definitions are stored centrally to make components available.

The Makefile of the package reuses general build rules defined in `kernel.Makefile` (which is contained in the kernel base package). These build rules can for instance build a component as statically linked or as dynamic loadable component. Parameters which define how to build the SCSI driver (such as the name of the driver and the constituent source files) can be defined declaratively in the Makefile:

```
DRIVER      = scsi.o
DRIVER_SRC  = scsi1.c scsi2.c
DRIVER_PRIO = scsi_0

include kernel.Makefile
```

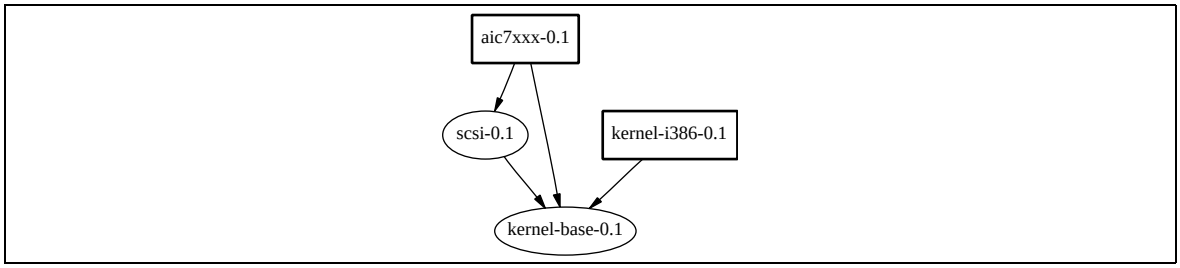

Figure 3: Dependency graph of a Linux kernel. By selecting the components `aic7xxx-0.1` and `kernel-i386-0.1`, `autobundle` automatically generates a source tree which also includes the (required) components `scsi-0.1` and `kernel-base-0.1`.

The symbols defined in `kernel-config.h` (see above) can be used when the implementation of the driver contains code that needs conditional execution.

Extending a component. The Linux kernel contains several generic drivers (such as the generic SCSI driver). Code from a generic driver is reused to implement others. Our architecture supports this by defining a dependency in a package definition. For example, we defined the SCSI package `aic7xxx` (which is also part of the Linux kernel) and defined a dependency upon the generic SCSI package that we described above. In the Makefile we instruct the preprocessor where to look for include files of the generic SCSI driver, by adding a line `INCLUDES += -I$(SCSI)/include`.

Building a kernel. Once package definitions for kernel components and component distributions have been made, different kernels can be assembled. Either via a generated online kernel base (see Figure 2), or by using the `autobundle` tool directly:

```
autobundle -o . \
   -p aic7xxx-0.1 -p kernel-i386-0.1
```

This command automatically assembles two kernel components and the components they depend on (see Figure 3). After unpacking the resulting distribution, the `README` file contains instructions about how to configure and build this kernel.

6. Concluding Remarks

Contributions. We analysed how the kernel and its build process are structured. We argued that separating linking of kernel source trees and linking of kernel components yields a more flexible product-line architecture. This architecture allows hardware vendors to easily add support for the Linux platform. It yields less code that needs to be maintained centrally. Finally, it no longer requires users to download a kernel source distribution of 28 MB, but only the sources they really need. We developed (part of) a new kernel build process which makes source tree assembling possible. We used `autobundle` to automatically assemble kernel source trees. We have shown the implementation of the build process for an artificial kernel with a similar structure as the Linux kernel. We also discussed a (generated) Linux kernel base from which kernel distributions are automatically assembled. Finally, we described how to build a specialised kernel source tree by using `autobundle`.

References

1. T. Aivazian. Linux kernel 2.4 internals, 2001. http://www.moses.uklinux.net/patches/lki.html.
2. M. E. Chastain. Linux kernel Makefiles, 2000. http://www.linuxhq.com/kernel/v2.4/doc/kbuild/makefiles.txt.html.
3. M. de Jonge. Source tree composition. In *Proceedings: Seventh International Conference on Software Reuse*, LNCS. Springer-Verlag, 2002. to appear.
4. Linux kernel coding style. http://www.linuxhq.com/kernel/v2.4/doc/CodingStyle.html.
5. D. Mackenzie and B. Elliston. Autoconf: Generating automatic configuration scripts, 1998. http://www.gnu.org/manual/autoconf/.
6. D. Mackenzie and T. Tromey. Automake, 2001. http://www.gnu.org/manual/automake/.
7. R. Russell et al. The kernel hacking HOWTO, 1999.
8. Submitting drivers for the Linux kernel. http://www.linuxhq.com/kernel/v2.4/doc/SubmittingDrivers.html.