

Unix System Administration and Programming (COSC1131/1133)

Lecture 7

Installing Software

Lecture Outline

- History of Unix
- Basic scripting
 - Variables
 - Control flow
 - Parameters and arguments
 - Debugging

Installing Software

- A common way of installing Linux software is in the form of packages
 - Packages are archives containing a list of files
 - Software repositories contain packages compiled for each distribution and version
- Package dependency
 - Many Linux packages depend on (i.e. use) other packages
 - E.g. emacs uses Pearl
 - Version dependency: Packages may require specific versions of other packages
 - See also: “dependency hell”
- Package managers
 - Automatically download and install files to the location specified by the package
 - Automatically uninstall software, based on installation information
 - Keep track of package dependencies, both when installing and when removing a package

Linux Package Managers (1)

- Two package formats are in common use on Linux systems.
- Red Hat, SUSE, and several other distributions use RPM, the Red Hat Package Manager.
- Ubuntu uses the separate but equally popular .deb format (named after the Debian distribution, on which Ubuntu was originally based).
- The two formats are functionally similar.
- At the lowest level of both, there are the tools that install, uninstall, and query packages: **rpm** for RPM and **dpkg** for .deb.

Linux Package Managers (2)

- On top of these commands are systems that know how to find packages on the Internet, analyze interpackage dependencies, and upgrade all the packages on a system.
- **yum**, the Yellowdog Updater Modified, works with the RPM system.
- The Red Hat Network is specific to Red Hat Enterprise Linux and uses RPM.
- The Advanced Package Tool (**APT**) originated in the .deb universe but works well with both .deb and RPM packages.



Linux Package Managers

Graphical front ends	package kit yum extender kyum	aptitude synaptic
Retrieving, managing dependencies and configuring	yum	apt
Basic installer, uninstaller, package querying	rpm rpmbuild	dpkg
Distro	Red Hat	Debian

RPM (low-level)

- The **rpm** command installs, verifies, and queries the status of packages.
- It formerly built them as well, but this function has now been removed.
 - This is now a separate command: **rpmbuild**.
- **rpm** options have complex interactions and can be used together only in certain combinations.
- It's most useful to think of **rpm** as if it were several different commands that happen to share the same name.
- **rpm --help** lists all the options broken down by mode
- Common options are **-i** (install), **-U** (upgrade), **-e** (erase), and **-q** (query).
 - For example, the command **rpm -qa** lists all the packages installed on the system.
- **rpm** doesn't resolve dependencies.

YUM(High-level)

- Yellowdog Updater Modified (YUM) is used in for RPM-compatible Linux operating systems, such as Red Hat Enterprise Linux, Fedora, CentOS, etc.
- **yum** has a command-line interface
 - Several other tools provide graphical user interfaces to yum functionality (e.g. PackageKit, Yum Extender, Kyum) 
- **yum** uses XML repositories
 - This has become the standard for RPM-based repositories
- **yum** can also synchronize packages to repositories
 - It uses metadata (information about the package)
 - Command: **distro-sync**
- A separate tool, **createrepo**, sets up yum repositories,  generating the necessary XML metadata.
- The **mrepo** tool (formerly known as Yam) can help in the creation and maintenance of repositories.

Revision Control Systems

- Version control tools help multiple users to make simultaneous changes to a collection of documents/files, without clobbering each others' work or resulting in version confusion.
- It's important to keep track of the configuration changes you make so that when these changes cause problems, you can easily revert to a known-good configuration.

Subversion

- **svn** command uses Subversion
- In the Subversion model, a central server or directory acts as a project's authoritative repository.
- By default, the Subversion server is a module in the Apache web server, which is convenient for distributed software development but perhaps not so good for administrative uses.
- Fortunately, the Subversion community provide an alternative type of server in the form of a daemon called **svnserve**.
- **svnserve** should have its own user account and be run from **inetd**.

Git

- Git is a distributed revision control system with an emphasis on speed.
- Every Git working directory is a full-fledged repository with complete history and full revision tracking capabilities, not dependent on network access or a central server.
- Instead of checking out a particular version of a project's files, you simply copy the repository (including its entire history).
- Your commits to the repository are local operations, so they're fast and you don't have to worry about communicating with a central server.
- Git uses an intelligent compression system to reduce the cost of storing the entire history, and in most cases this system is quite effective. In many cases, working requirements for storage space are even lower than with Subversion.

Compiling versus Package Installation

- Compiling allows more control over the options for an application
- Compiling an application means that it will be linked to the installed libraries on the system
- A package system provides easier control of installation, removal and upgrading of applications
- But some package systems do insufficient dependency checking

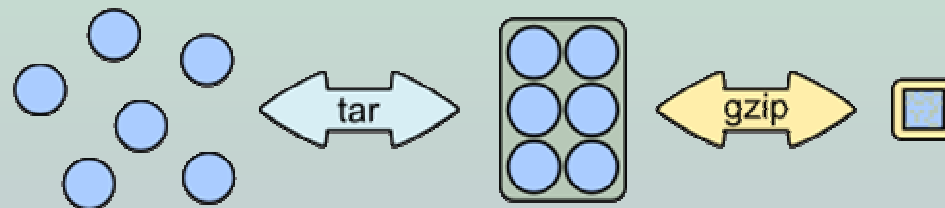
Compiling and Installing Software from Source

- Software usually comes in compressed format, such as **tar.gz** and **tar.bz2**
- General steps in compiling and installing software
 - Step 1: Unpacking
 - Step 2: Configuring
 - Step 3: Building
 - Step 4: Installing
- Example:

```
$ tar xvzf package.tar.gz (or tar xvjf  
package.tar.bz2)  
$ cd package  
$ ./configure  
$ make  
$ make install
```

Unpacking Software Packages(1)

- **tar**: Derived from tape archive, and commonly referred to as "tarball"
- Tar archiving is often used together with a compression method, such as **gzip**, to create a compressed archive. As shown, the combination of the files in the archive are compressed as one unit.



Unpacking Software Packages(2)

- `.tgz` is equivalent to `.tar.gz`
- `.tbz` and `.tb2` are equivalent to `.tar.bz2`
- Unpack gzip file
`$ tar xvzf pkg.tar.gz`
- Unpack bzip file
`$ tar xvjf pkg.tar.bz2`
- `tar` command options:
 - `-C`, directory DIR
 - `-f`, file F
 - `-j`, bzip2
 - `-p`, preserve-permissions
 - `-v`, verbose
 - `-z`, gzip
 - `-x`, extract

Configuring Source Code

- Source code needs to be configured before actual compilation happens.
- **README** and **INSTALL** files often provide step-by-step instructions for compiling from source
 - Not always complete or accurate, but provide a good starting point for other requirements
- Configuring is done by running the **configure** script.
 - When you run the **configure** script, you don't actually compile anything yet.
 - **configure** just checks your system and assigns values for system-dependent variables.
 - These values are used for generating a **Makefile**.
- The **Makefile** is used for generating the actual binary.

Sample `configure` Command Line Options

- **`--help`**
 - Lists commonly used options (not necessarily all, though)
- **`--prefix=/path`**
 - Sets compile to work with a given base path. Files are installed to paths underneath this base (e.g. executable programs are installed to `$PREFIX/bin` and `$PREFIX/sbin`)
- **`--enable-feature` Or `--with-feature`**
 - Sets compilation to include a given feature
- **`--disable-feature` Or `--without-feature`**
 - Sets compilation to exclude a given feature

Building Software

- We have to build the binary (the executable program) from the source code. This is done by running the **make** command.
- A **Makefile** tells **make** how to build the application. Without it, **make** doesn't know what to do.
- A **Makefile** can be generated by using the **configure** script, but there are other ways as well.
- When you run **make**, you'll see messages filling the screen.
 - This step may take some time, depending on the size of the program and the speed of your computer.
- If everything goes as it should, your executable is finished and ready to run after **make** has done its job.

Makefile Structure

- **Makefile**

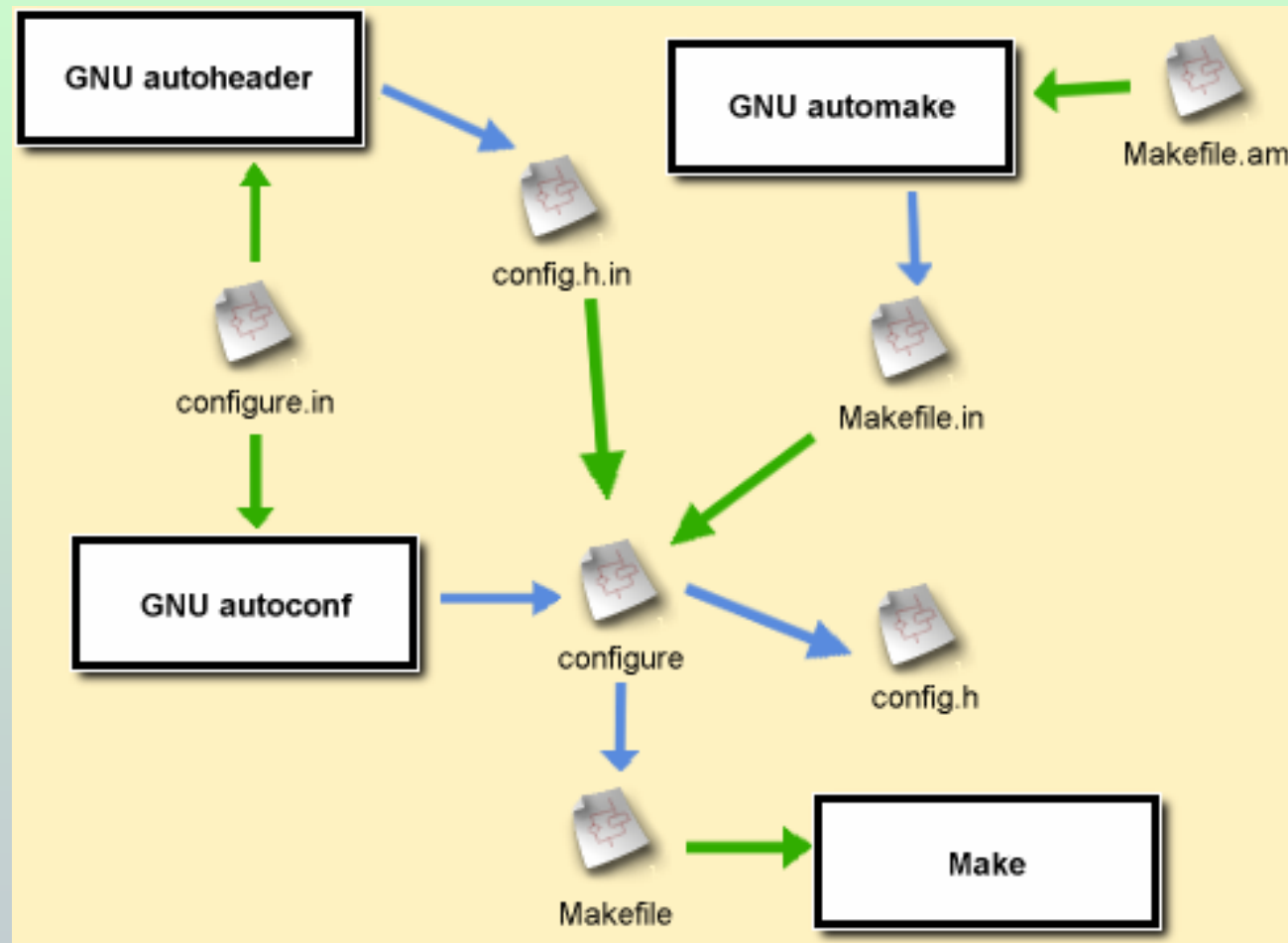
- It is a “recipe” generally used to generate executable code from some source file(s)
- works on a dependency basis (i.e. to build “a”, you need to first build “b” and “c”)

- Make is not a standard (unfortunately)

Variations include

- GNU's Make
- POSIX Make

The Software Building Process



GNU Build System^{🗨️} 🗨️

- Used for building packages for a system
- **Autotools** create a GNU build system for your package

Main components

- **Autoconf**: A tool for producing configure scripts for building, installing and packaging software (needs a Bourne shell)
 - After running a series of tests on the target system the configure script generates header files and a **Makefile** from templates, thus customizing the software package for the target system.
- **Automake**: A tool that produces portable **Makefile.in** files
 - It is written in Perl and must be used with GNU Autoconf
 - Automake contains the following commands:
 - **aclocal**: A program for generating **automake** macros
 - **automake**
- **Libtool**: generic library support script

configure.in Example

`AC_INIT(my_prog.c)`

`dn1` find and test the C compiler

`AC_PROG_CC`

`dn1` Determining a C compiler to use

`AC_LANG_C`

`dn1` Do compilation using cc and use .c extension

`AC_PROG_MAKE_SET dn1` Define variable **MAKE**

`AC_HEADER_STDC dn1` Check if the system has ANSI header files

`AC_CHECK_FUNC (function, [action-if-found], [action-if-not-found])`

`dn1` Example: `AC_CHECK_FUNC (atoi, , AC_MSG_ERROR(oops! no atoi ?!?)`

`VERSION="0.0.1"`

`AC_SUBST(VERSION)`

`dn1` substitute a particular variable into the output files

`dn1` read Makefile.in and write Makefile

`AC_OUTPUT(Makefile)`

Makefile.in Example

```
CC = @CC@
```

```
VERSION = @VERSION@
```

```
CFLAGS = @CFLAGS@
```

```
all: my_prog-bin
```

```
my_prog-bin: my_prog.c
```

```
    $(CC) $(CFLAGS) -DVERSION=\"$(VERSION)\" my_prog.c -o  
    my_prog
```

```
clean: rm -f my_prog
```

```
distclean: rm -f my_prog config.* Makefile
```

Essentially, **configure** will replace the variables between the @ signs

Sample configure output

```
creating cache ./config.cache
checking for gcc... gcc
checking whether the C compiler (gcc ) works... yes
checking whether the C compiler (gcc ) is a cross-compiler...
no
checking whether we are using GNU C... yes
checking whether gcc accepts -g... yes
checking whether make sets ${MAKE}... yes
checking how to run the C preprocessor... gcc -E
checking for ANSI C header files... yes
checking for atol... yes
updating cache ./config.cache
creating ./config.status
creating Makefile
```


Building Software

- You need to be **root** or use **sudo** to be able to install software.
- **make install** command installs the program
- It shows some messages on screen
- If the **configure** command is run without any options, most of the time it installs the executable in some place, which is already in your PATH
 - If so, you can just run the program by typing its name. But if the file is not in one of PATH directories, you can simply find the file and copy there.
- **make clean** removes all unnecessary files created during installation by **Makefile**
- Make sure you keep your **Makefile**. It's needed if you later decide to uninstall the program.

Devices and Hardware in Linux

- In Linux and UNIX each and every hardware device is treated as a file.
- Device files are
 - *special* files.
 - stored in the **/dev** directory
- A device file
 - provides access to hardware devices so that end users do not need to get technical details about hardware
 - is an *interface* to a device driver
 - allows software to interact with the device driver using standard input/output system calls, which simplifies many tasks

Classes of Devices

- Character devices

- Can be accessed as a stream of bytes (e.g. keyboard)
- Usually implement at least open, read, write close operations (e.g. `/dev/console`)
- Similar to files, but most devices allow only moving forward in the byte stream (sequential access)
- E.g.

```
crw-----.  1 root root      5,   1 Jun 20 19:55 console
```

- Block devices

- Handle larger data units (blocks) e.g. file systems, but accessing each character is also often possible
- Are usually addressable devices (e.g. disk)
- No restrictions on moving back and forth in the data
- E.g.

```
brw-rw----.  1 root disk      8,   1 Jun 20 19:55 sda1
```

- Network interfaces

- Stream-oriented
- Handle blocks (packets)

Pseudo-devices

- **/dev/null**: Accepts and discards all input; produces no output.
- **/dev/zero**: Produces a continuous stream of NULL (zero value) bytes.
- **/dev/random**: Produces a variable-length stream of pseudo-random numbers.
 - High-quality randomness, i.e. high entropy (but still pseudo-random)
 - Blocks until a 'random' noise is generated
- **/dev/urandom**: Produces a variable-length stream of pseudo-random numbers.
 - Re-uses the noise to produce more pseudorandom bits: less entropy
 - Non-Blocking

Kernel Tasks

- Process management
 - Creating, destroying processes, communication between processes (signals, pipes, interprocess communication)
- Memory management
 - Allocating and releasing memory, etc.
- File systems management
- Device control
 - Device-specific operations performed by device drivers
- Networking
 - Collecting, identifying and dispatching incoming packets, organising the delivery of messages

Application Level

User Programs

Kernel Level

System Calls

Process
Management

Memory
Management

Device
Control

File
Systems

Network

Architecture-
Dependent
Code

Memory
Manager

Character
Devices

Block
Devices

Network
Subsystem

CPU

RAM

Console,
Serial Ports

Disks,
CD, DVD

Network
interfaces

Hardware Level

What is a Device Driver?

- A piece of software that translates between the hardware commands understood by the device and the programming interface used by the kernel.
- It
 - is a collection of functions to access the hardware
 - Doesn't force any policies on the user (but configuration files specify default behaviour)
 - can be invoked by several applications simultaneously (reentrant code)

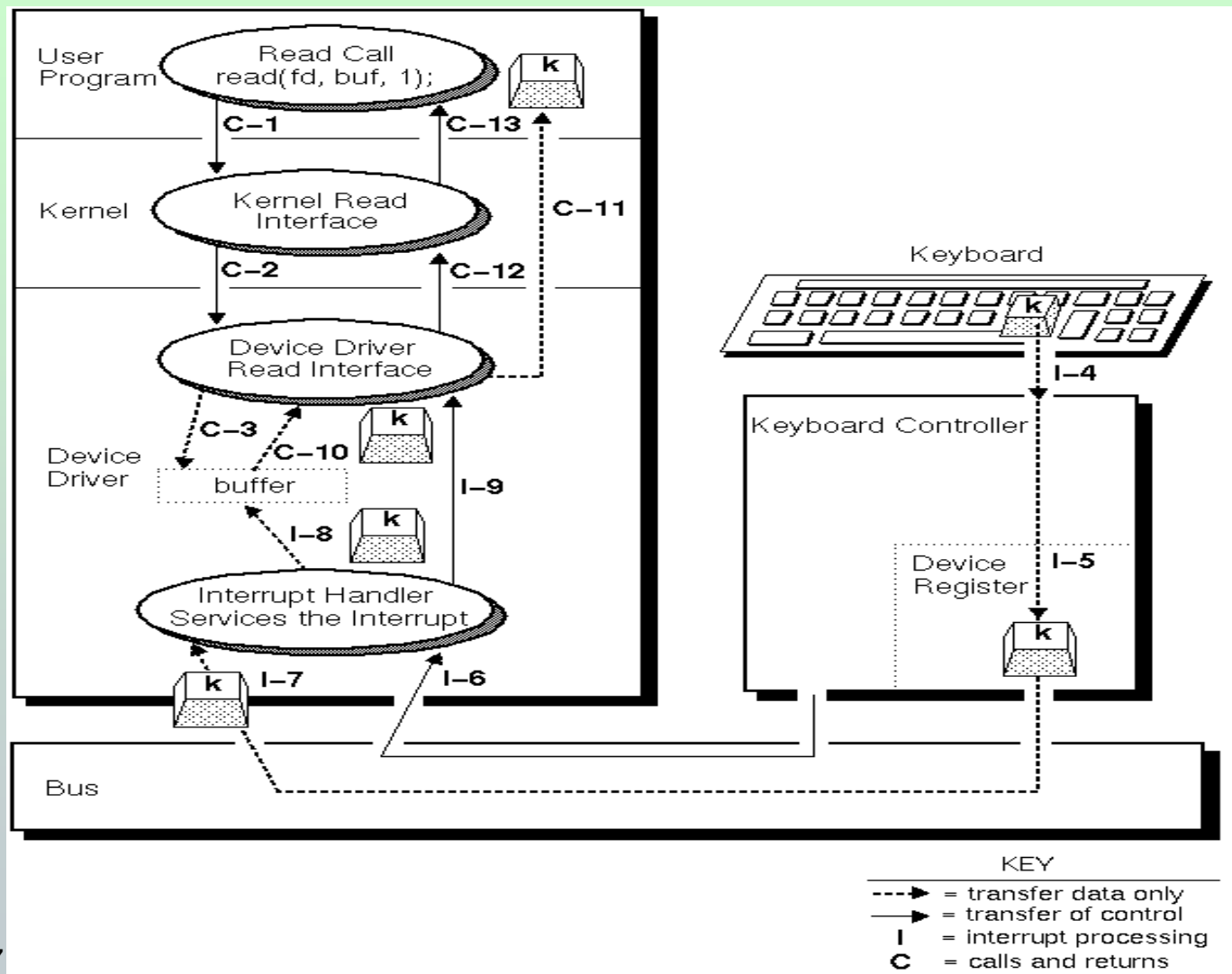
Drivers and the Kernel

- Drivers are kernel processes.
 - In most cases, device drivers are part of the kernel.
- A driver can be accessed from within the kernel and from user space.
- The driver layer helps keep the kernel reasonably device independent.
- Device drivers are system specific
 - Depend on the operating system
 - Drivers for other operating systems (e.g. Windows) will not work on Linux
 - Depend on the architecture (i385, x86, ...)
 - Often specific to a particular range of kernel versions

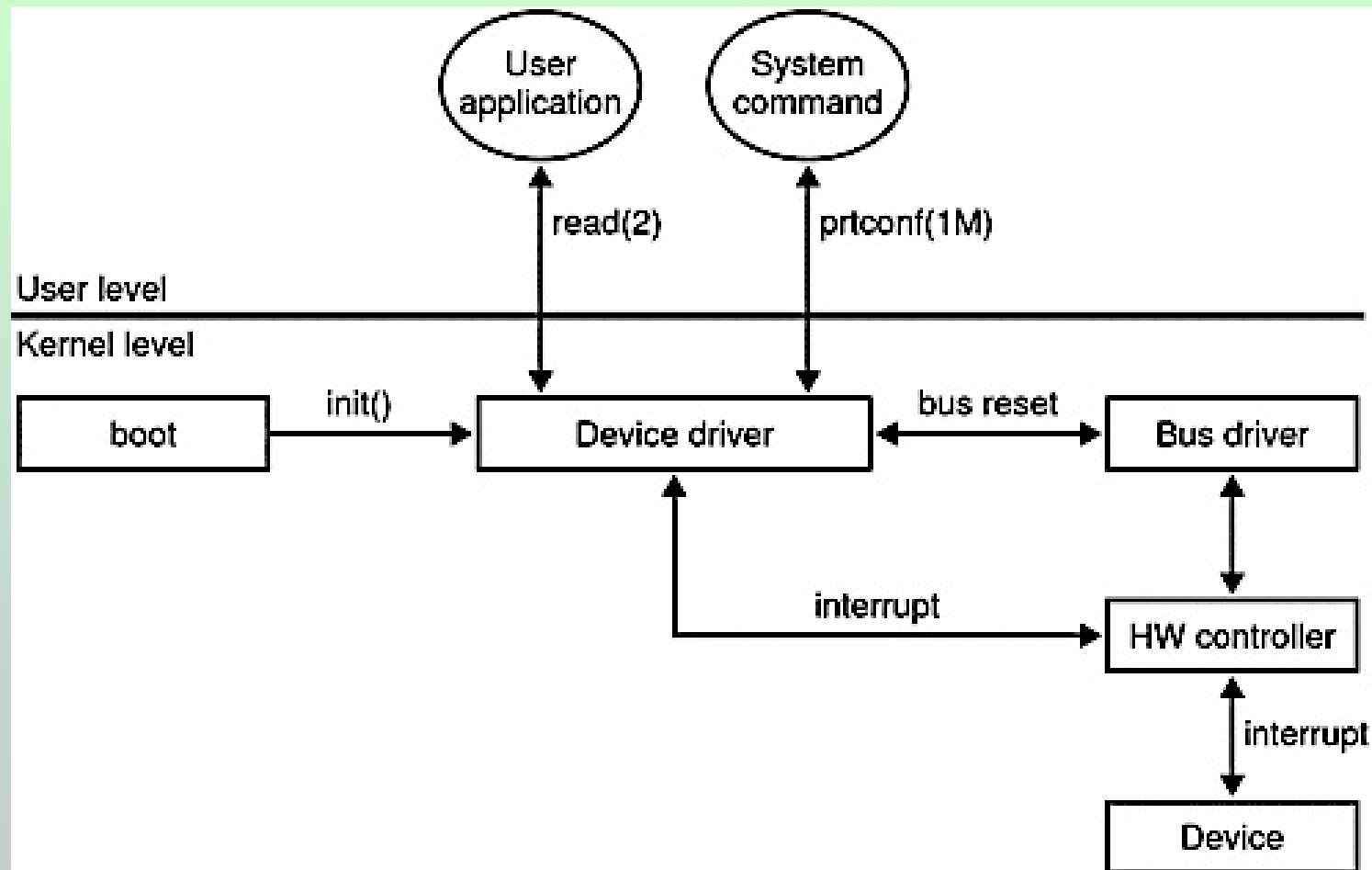
Device Driver Tasks

- Autoconfiguration
 - Determine availability
- I/O operations
 - E.g. open, read, write, close
- Interrupt handling
 - Respond to device requests in a timely manner
- Special requests
 - **ioctl** calls manipulate the parameters of device files
- Re-initialisation
 - E.g. when the system bus is reset
- User-level requests to the **sysconfig** utility
 - Allow system manager to reconfigure the device

Character Input Example



How a Driver Works



Device Numbers

- Device drivers are listed in a kernel table
- Devices have major and minor numbers
- The major number of a device
 - identifies the driver (offset in the kernel's table)
 - are fixed in Linux
 - E.g. IDE: 3, 22, 33, 34, SCSI: 8
 - for new drivers can be allocated manually, or dynamically by the system
 - can be read from **/proc/devices**
- Within each device driver there are routines used for accessing device functions (e.g. read, write, etc)
- The minor number is a flag passed on to the driver
 - The interpretation of the minor number is up to the device driver
 - E.g. disk #2

Linux Device Naming Conventions

- Input/output device examples
 - **lp**: *line printers*
 - **tty**: *terminals (tty1-tty8)*
 - **fb**: *frame buffer (for graphics hardware)*
- External storage device examples
 - **hd**: *IDE devices*
 - **hda** → *first registered device*
 - **hdc2** → *second partition of third IDE disk*
 - **sd**: *SCSI, PATA/SATA drivers, USB, Firewire, etc.*
 - **sda** → *first registered device*
 - **sdb3** → *third partition of second SATA/SCSI registered device*

A Typical `/proc/devices` File Excerpt

Character devices:

```
1 mem
4 tty
5 /dev/console
6 lp
7 vcs
10 misc
13 input
14 sound
21 sg
180 usb
```

Block devices:

```
1 ramdisk
8 sd
11 sr
65 sd
66 sd
```

Device File Creation

- Block or character device files can be created manually with the **mknod** command, with the syntax:
 - **mknod** *filename type major minor*
 - *filename* is the device file to be created
 - *type* is **c** for a character device or **b** for a block device
 - *major* and *minor* are the major and minor device numbers.
- If you are manually creating a device file that refers to a driver that's already present in your kernel, check the documentation for the driver to find the appropriate major and minor device numbers.

Hardware Compatibility

- A hardware compatibility list (HCL) is a list of computer hardware that is compatible with a particular operating system or device management software.
- HCL is a database of hardware models and their compatibility with a certain operating system.
- Before buying hardware or selecting an OS for existing hardware, you should carefully study the HCL and only select compatible hardware.
- Using incompatible hardware may result in unexpected behaviour of the system or sudden crashes.
- Each vendor provides their own HCL
- <http://www.linux-drivers.org> is a good source of checking different distros HCL

Adding Drivers to the Kernel

- Drivers can be added to the system while building and compiling the kernel.
- Drivers used during boot have to be part of the kernel
 - (Initially in the boot process, it is the kernel that takes care of everything)
 - Drivers not included in the kernel will be treated as modules and are accessible only after the boot process completes
- Missing drivers for important devices can cause serious problems, e.g.
 - SATA hard drives
 - Without SATA drivers in the kernel, it is not possible to load boot data from SATA disks, so the system crashes with a “kernel panic”
 - USB keyboard and mouse.
 - Without drivers you can’t type to login or even restart the system, so it freezes

Installing Drivers

- On Linux systems, device drivers are typically distributed in one of three forms:
 - A patch against a specific kernel version
 - A loadable module
 - An installation script or package that installs the driver
- The most common form is an installation script or package.
- Sometimes you need to patch the kernel to get support for a specific driver

Loadable Kernel Modules (LKMs)

- Loadable kernel modules (LKM) are object files that are linked to the kernel runtime
 - They reduce the kernel size when a module is not being used
 - They are often used for adding device drivers to the kernel
 - They can be a security risk, e.g. modules can contain bugs or be malicious
 - Many rootkits are implemented as LKMs
 - You can disable module loading via `/proc/sys/kernel/modules_disabled`
- LKMs are conventionally stored in `/lib/modules` and the filenames end in `.ko`
 - RedHat 7 stores them in `/lib/modules/kernel_version/kernel/drivers/`

Working with LKMs

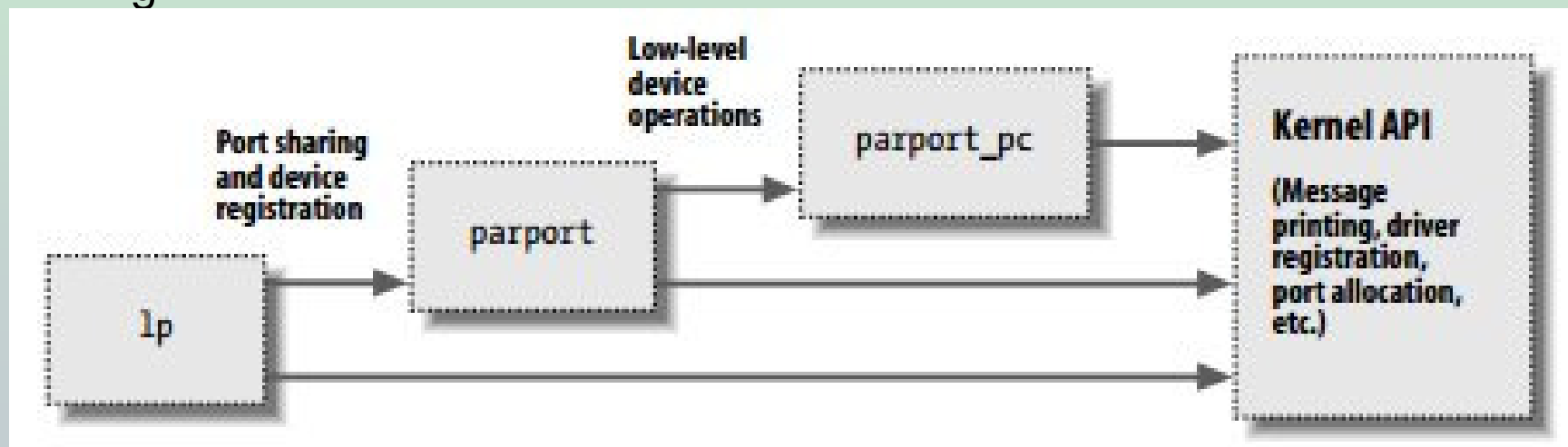
- Persistent module loading
 - Many kernel modules are loaded automatically at boot time
 - Additional modules to be loaded can be listed in `/etc/modules-load.d/program.conf` and will be loaded by the `systemd-modules-load.service` daemon (RedHat 7)
- Loading and unloading
 - To see currently loaded modules, use `lsmod`
 - For each module, it lists the name, the memory it uses and the number of processes using it and the name of those processes
 - `/proc/modules` gives a more verbose list of the same
 - To load a module, run `modprobe module_name` as `root`
 - To unload, use `modprobe -r module_name`
 - Other commands to load/unload: `insmod`, `rmmod`
 - To get information about a module: `modinfo <drivername>`

Module Operation

- Starting (**initialization_function**)
 - A module registers itself with the kernel (i.e. prepares for later invocation)
- Operation
 - Event-driven
 - Can use only functions built into the kernel
 - A fault (e.g. segmentation) can kill the process or the whole system
- Exit/shutdown (**cleanup_function**)
 - Gets executed before the module is unloaded
 - Must undo everything that was built up by it (e.g. free memory, interrupt levels)

Module Dependencies

- Kernel modules can refer to (functions in) other modules: this is called module stacking
 - Splitting modules into layers
 - improves code organisation, but
 - modules become dependent on other modules
- E.g.



Drivers in Kernel Space and User Space

- Drivers usually execute in kernel space

Advantages

- Additional CPU instructions may be available
- Each driver has its own memory mapping / address space protected from ordinary users

- Some drivers execute in user space

Advantages

- The full C library can be linked in
- If the driver hangs, the user can kill it
- Infrequently used drivers can be swapped out

UnixSysAd7 Easier to protect closed-source code