The Linux Operating System

Lecture on

The Linux Operating System

Using the penguin for fun and profit

Walter Kriha

Goals

- Learn the basic Linux concepts: everything is a file, getting help with “man” and “info”, users and groups.
- Learn about the core Linux components: kernel, modules, environment, interprocess communication
- Learn about the Linux shell (bash)
- How network graphics works: learning X Windows
- Understand the differences between a Windows system and Linux. When to use what.
- The final goal is to enable you to work with the Linux certification materials.

Linux is – in good old Unix tradition – a rather open system. This makes it an ideal candidate to learn about operating systems: both source code and documentation is ample and free.

Procedure

-An example session:

-Booting Linux: what is bootstrapping and how does it work

- Entering Multi-user mode: runlevels

- Work with shell and XWindows

- The Linux filesystem.

- Linux administration

- Linux architecture: kernel and environment, daemons and tools

This lecture assumes that you have not been exposed to a Unix or other multi-user system yet. We will start with the basics therefor. Later sessions will deal with filesystem, virtual memory and processes in detail.

A short history of Linux

1991 Linus Thorvalds starts working on a 386 based operating system. It was published on the internet and transformed into an open source project with thousands of developers.

1994 Release 1.0 stable was published

2001 Release 2.4.2 stable published, now running on many different processors (alpha, Sparc, Ultra, M68k, PowerPC and x86)

2003 Kernel version somewhere around 2.4.20 and going for 64 bit processors. Almost every kind of operating system technology (firewalls, journaling file-systems, kerberos etc.) have been implemented on Linux. A majority of web applications runs on the LAMP (Linux, Apache, MySql, PHP) platform and dominates the web server market.

With Linux desktop versions the operating systems tries to take market share from Microsoft by attacking “from below”. Companies and governments are increasingly thinking about switching to a Linux platform for price and security reasons.

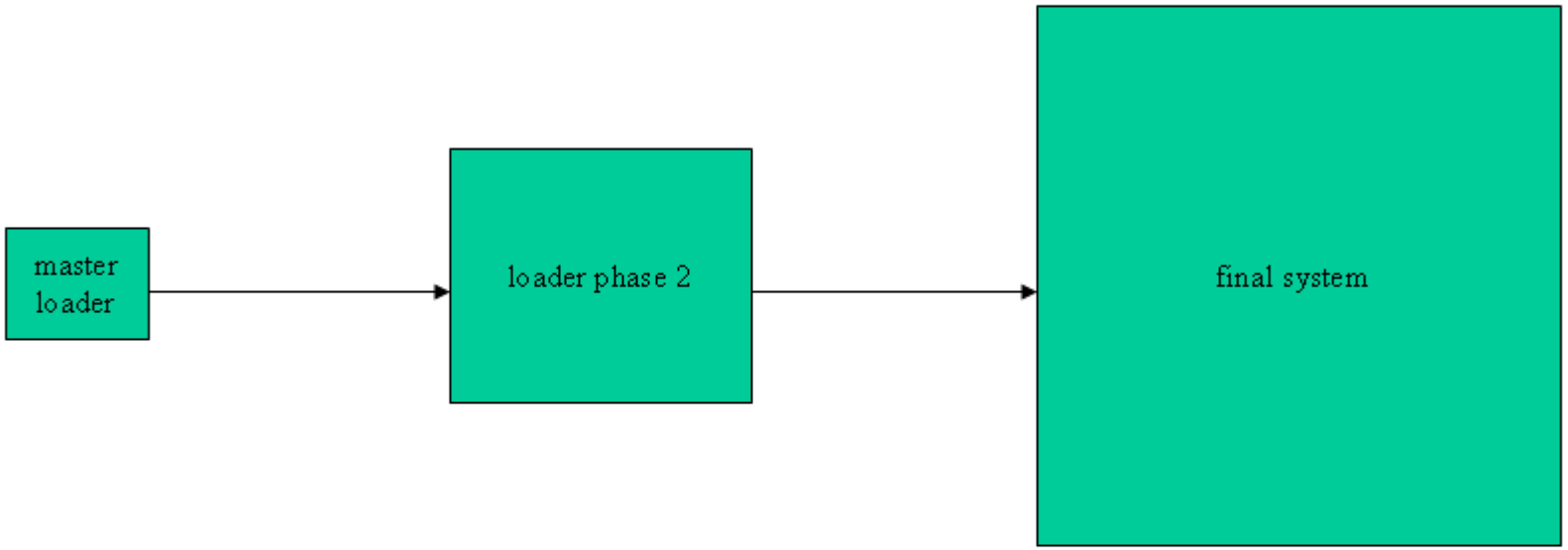
Linux is NOT a revolutionary technology. But it is certainly a very successful one.

Linux Architecture

1. Philosophies
2. Kernel design
3. Daemons and servers
4. Filesystems (journaling)
5. Network graphic architecture
6. Device Management and loadable modules

Linux is not at all revolutionary in many aspects. But avoiding the latest ideas of operating system builders also made Linux a fairly understandable and stable system.

Bootstrapping

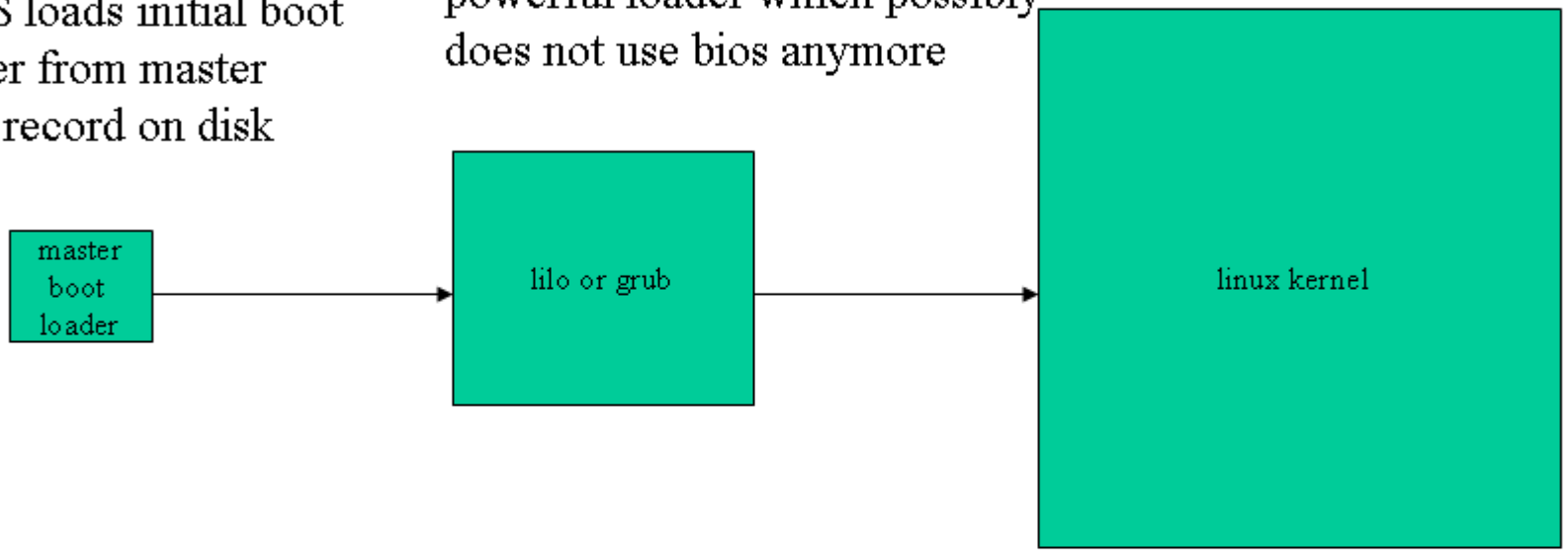


bootstrapping is the process of starting a large and complicated system via a number of very small steps. A characteristic feature of bootstrapping is that the wonderful and powerful functions of a large system cannot be used to start the system itself – they are simply not yet available because the system is not running.

Loading Linux (1)

BIOS loads initial boot loader from master boot record on disk

The initial loader loads a more powerful loader which possibly does not use bios anymore



Linux loaders are lilo or grub which are both found under /boot. The difference is that lilo knows exactly at which block of the partition the linux kernel starts and how big it is. It does NOT understand the linux filesystem and takes the information about the kernel from when the kernel was installed under /boot and lilo was re-run. Grub understands the filesystem and can locate the kernel within.

Loading Linux (2): System Check and Autoconfiguration

During system start the following functions are performed:

-determine CPU type, RAM etc.

- stop interrupts and configure memory management and kernel stack

- Initialize rest of kernel (buffers, debug etc.)

- Start autoconfiguration of devices from configuration files and via probing hardware addresses.

Probing is done via device driver routines. It means that certain memory locations are checked for the presence of a device. The system catches errors and then assumes that there is nothing mapped to this location. During regular operation such errors would cause a kernel panic.

Loading Linux (3): Start processes

1. init: the first process and the only one started by the kernel itself. Starts other processes e.g. getty's waiting for logins from terminals
2. Swapper and other system processes (yes, the kernel depends on processes running in user mode)

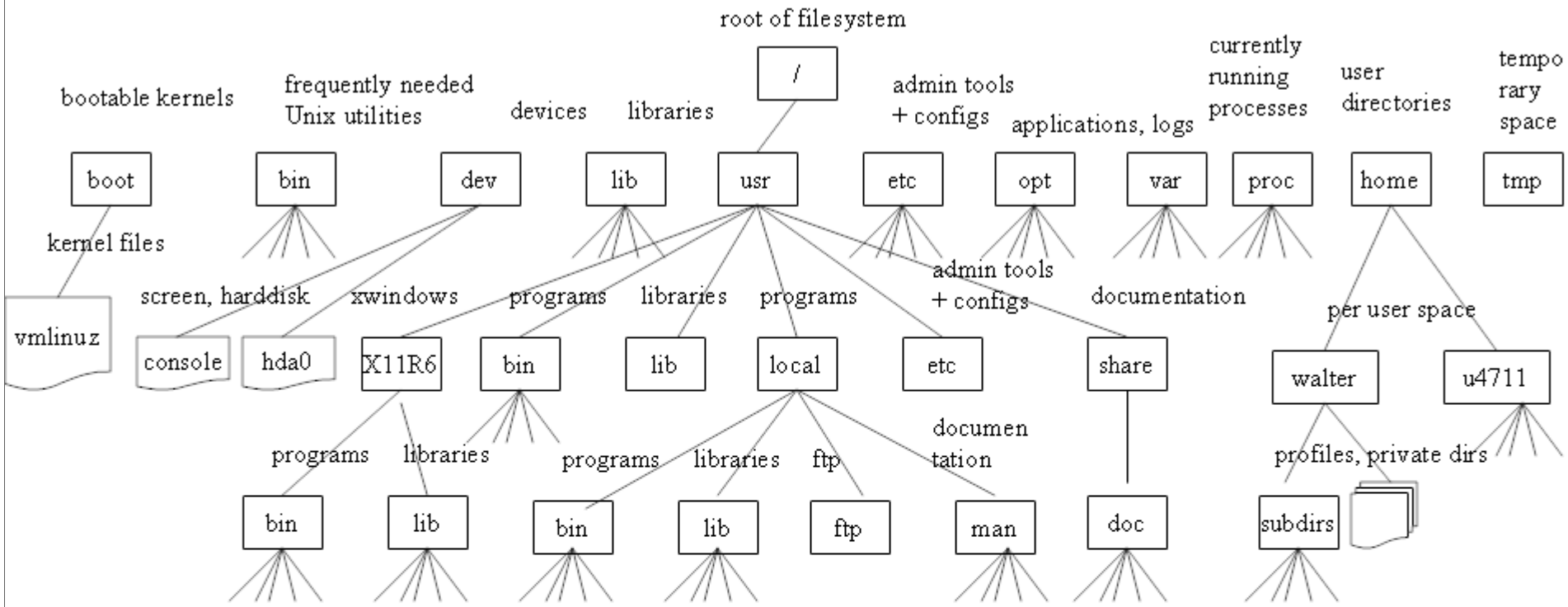
Init is the parent of all processes. Killing it usually causes an immediate shutdown of the whole system.

Loading Linux (4): Go to runlevel

1. System configuration scripts under /etc/rc.d/ are executed (shell scripts)
2. Depending on the configured „runlevel“ the system either boots into single-user mode or multi-user mode with or without networking and with or without X Window system. (The runlevel can be specified at kernel load-time)

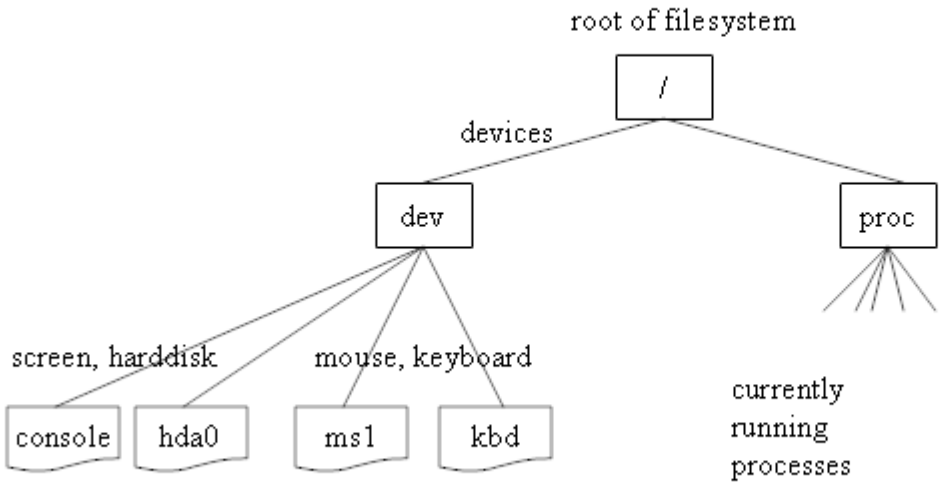
Shell scripts basically initialize the whole system once the kernel itself is running.

Filesystem Organization



Linux follows many Unix conventions: bin contains binaries (programs), lib libraries (shared libs with .so extension), etc administration stuff. A multi-user system needs to separate global and personal directories somehow. This has grown historically and is far from being consistent. An important issue is where new programs are installed. You need administration rights to install them into global places because this would affect other users. If you need a name and you want to signify that the name itself is not important, Linux/Unix people use „foo“ or „bar“ or „foobar“. Different Unix and Linux distributions have a different filesystem layout but the Linux Standard Process defines a core layout.

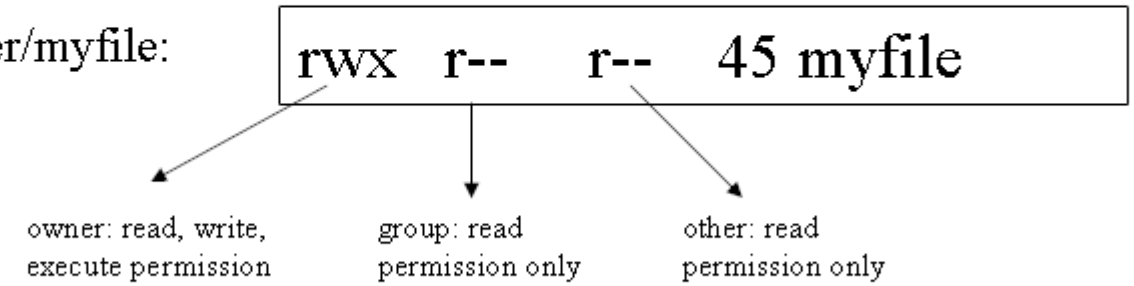
Everything is a file (1)



Note that even non-files are made visible as files by mapping them into the filesystem. Unix users treat devices just like files. This means that the standard Linux file utilities (e.g. cat) can be used on devices as well. Cat /dev/tty1 reads data from a terminal. Cat /tmp/somefile would read data from some file. The same is true for processes. They show up as entries under /proc. The running Linux system can be configured by writing values into the /proc space, e.g ip settings. with „echo 1 >/proc/ip/....“ File permissions are also available with non-file entities.

Everything is a file (2)

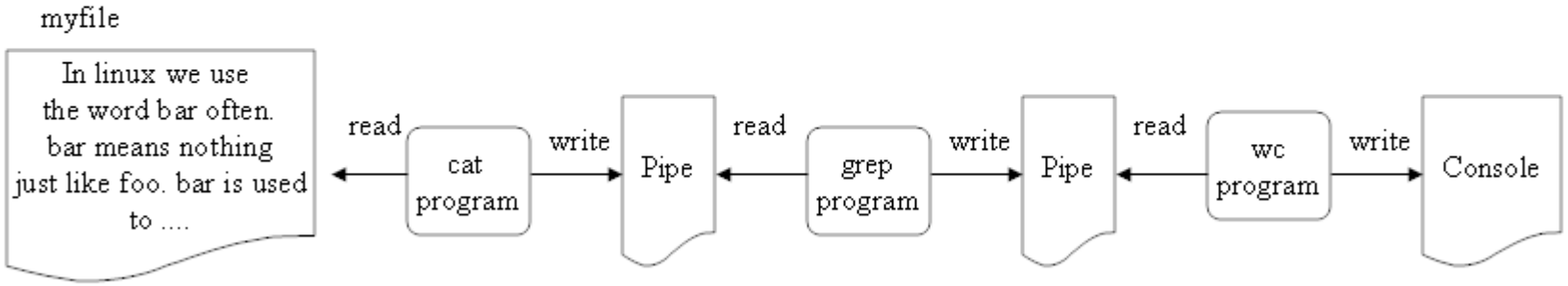
ls –l /home/walter/myfile:



Linux knows three basic permissions: read, write, execute. And it knows three owner types: the real owner, her group(s), anybody else (other). With Linux/Unix you will get errors during command execution which are in many cases simple permission problems. Check whether you have permission to read/write/execute the object you want to use. „ls“ is pretty much the same as the „dir“ command in MS-DOS/Windows.

Everything is a file (3)

cat /home/walter/myfile | grep „bar“ | wc –l



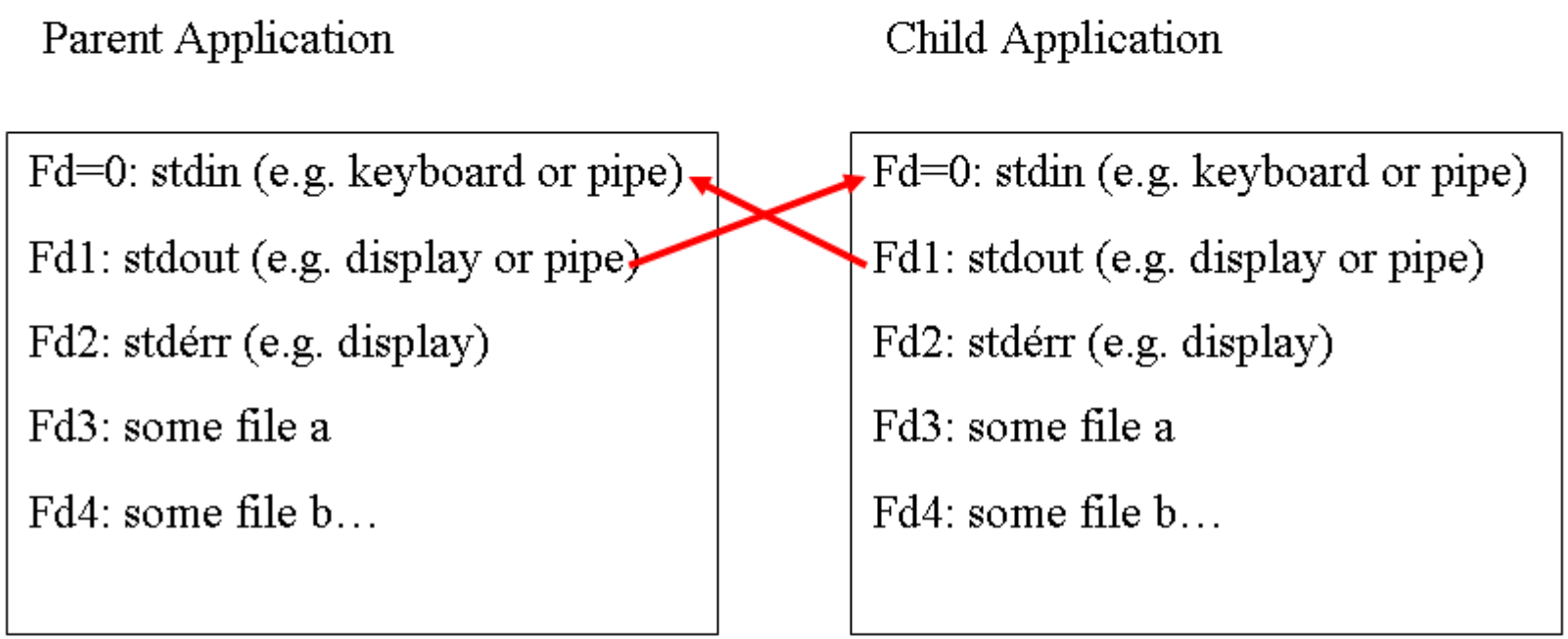
Because everything is a file it can be used/programmed with the same functions (open, close, read, write, ioctl). A pipe is a communication channel which connects programs that are connected through a parent/child relation. Again, it looks like a file so programs read and write to a pipe. Even the final display (console) works like a file and can be written to.

The command from above could also be written like that:

grep „bar“ > foo; wc –l < foo > /dev/console

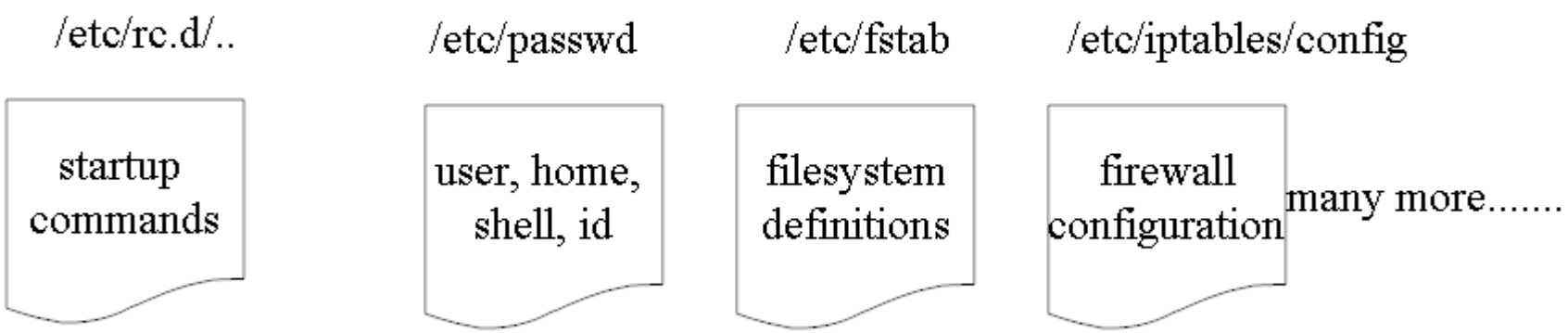
The difference is only that the results of intermediate processing will be stored in files instead of dynamically shipped to the next program. The following program opens the file and reads it. The difference is execution speed. The command btw. searches (greps) for „bar“ strings in myfile and counts the resulting lines with „word count (wc). In an object-oriented sense this is a high degree of polymorphism.

Stdin, Stdout, Stderr, Pipe



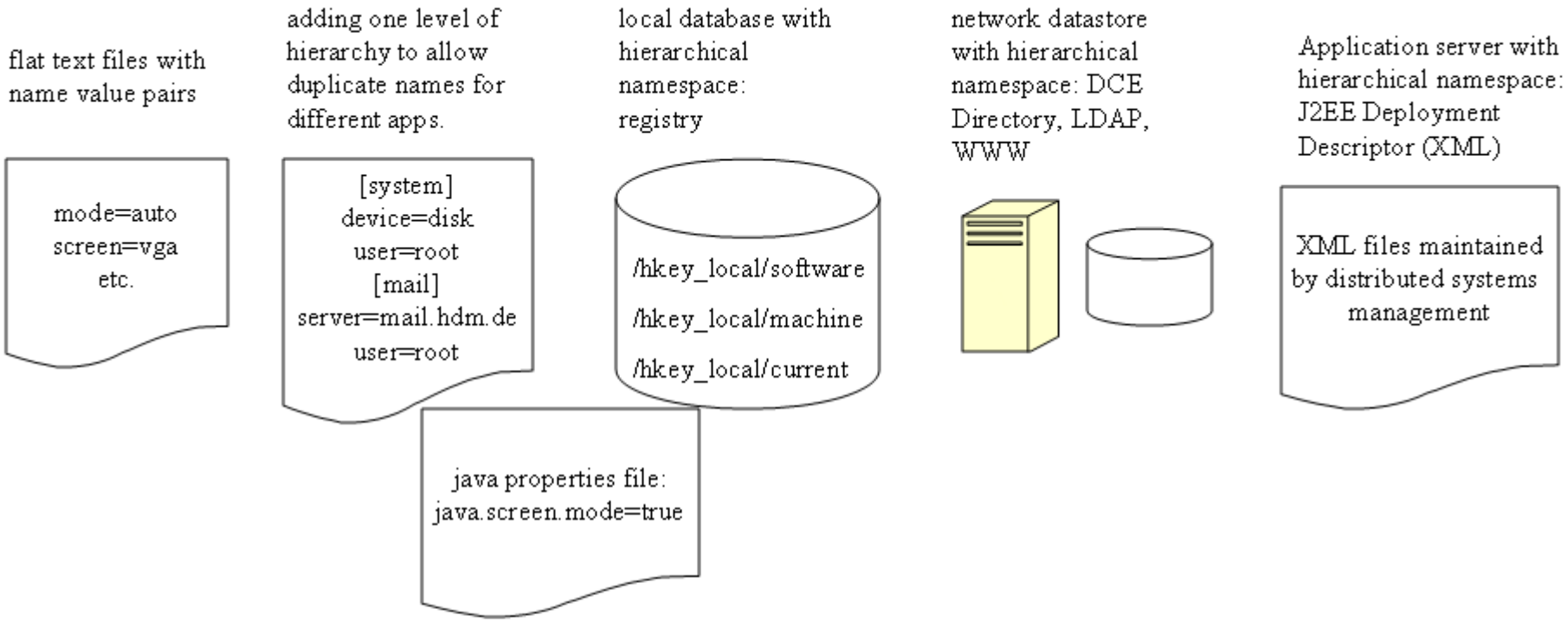
Pipes are interprocess communication channels which connect parent and child. The shell substitutes the original stdout of the father with an outgoing end and the original stdin of the child with the incoming end of the same pipe. If the parents now writes to “stdout” it really writes right to the childs inchannel – without realizing it! Without the need to reprogram anything! The back channel is set-up the same way by the shell. The shell basically configures the routes for process communication.

Everything is a file (4)



Linux systems are configured via configuration files. There are many of them. System administrators need to change and adjust those files. Many tasks can be done using GUI frontends nowadays but a real Linux admin will always want to see the resulting files. Because configuration is file based many file/text manipulation utilities (awk, sed, ed etc.) can be used to automate administration processes. Again: polymorphism at its best. Introducing a database format (e.g. windows registry) forces one to create a lot of new admin tools as well because the interface has now changed.

The evolution of flexibility: configuration



Configuration information is one of the oldest ways to make applications or systems flexible. Behavior control is extracted from code and put into files or later databases or network stores. Things that changed: from file access (single process) to concurrent, remote access to database. From simple non-standardized information formats to highly structured XML hierarchies. Centralized maintenance of large numbers of applications and systems is now possible. The consequence is that we lost the simple way to edit config files with any text editor. We now have to use network enabled tools which understand the database access languages. But we won concurrency, remoteness and the power to represent arbitrary structures. (hardware also uses „morphware and configware“ to configure FPGAs)

The file interface (API)

1. fd = creat(„filename“, mode) // exclusive access etc.
2. fd = open(„filename“, mode, ..) // open file for read and/ or write
3. status = close(fd); // no name, only handle
4. number = read(fd, buffer, nbytes) // reads bytes into buffer from file
5. number = write(fd, buffer, nbytes) // writes bytes from buffer into file
6. position = lseek(fd, offset, whence) // move file pointer (no real disk seek)
7. status = stat(„filename“, &buf) // read file status into buf structure
8. status = fstat(fd, &buf) // same with file descriptor
9. status = pipe(&fd[0]) // create a pipe
10. status = fcntl(fd, cmd,..)

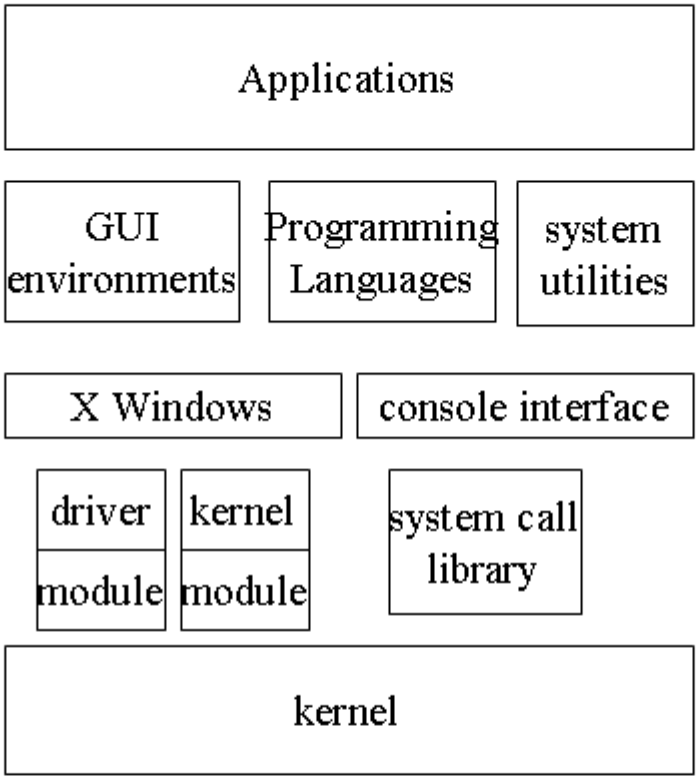
This table (after Tanenbaum pg. 738) shows the file related system calls. Every object with this type of interface can be treated as a regular file by countless unix utilities. Can you explain the function of „fd“ – the so called file descriptor?

User and Super-User

- There is one Unix Administrator account available in a Linux system. It is called „root“ (like the top node in the linux filesystem). It has ALL permissions for every system resource.
- There can be many other user accounts present. Those users can own resources either directly (they created them) or indirectly via group memberships
- Groups are collections of users.
- Resources have an associated access control list which knows 3 different principals: owner, group, other

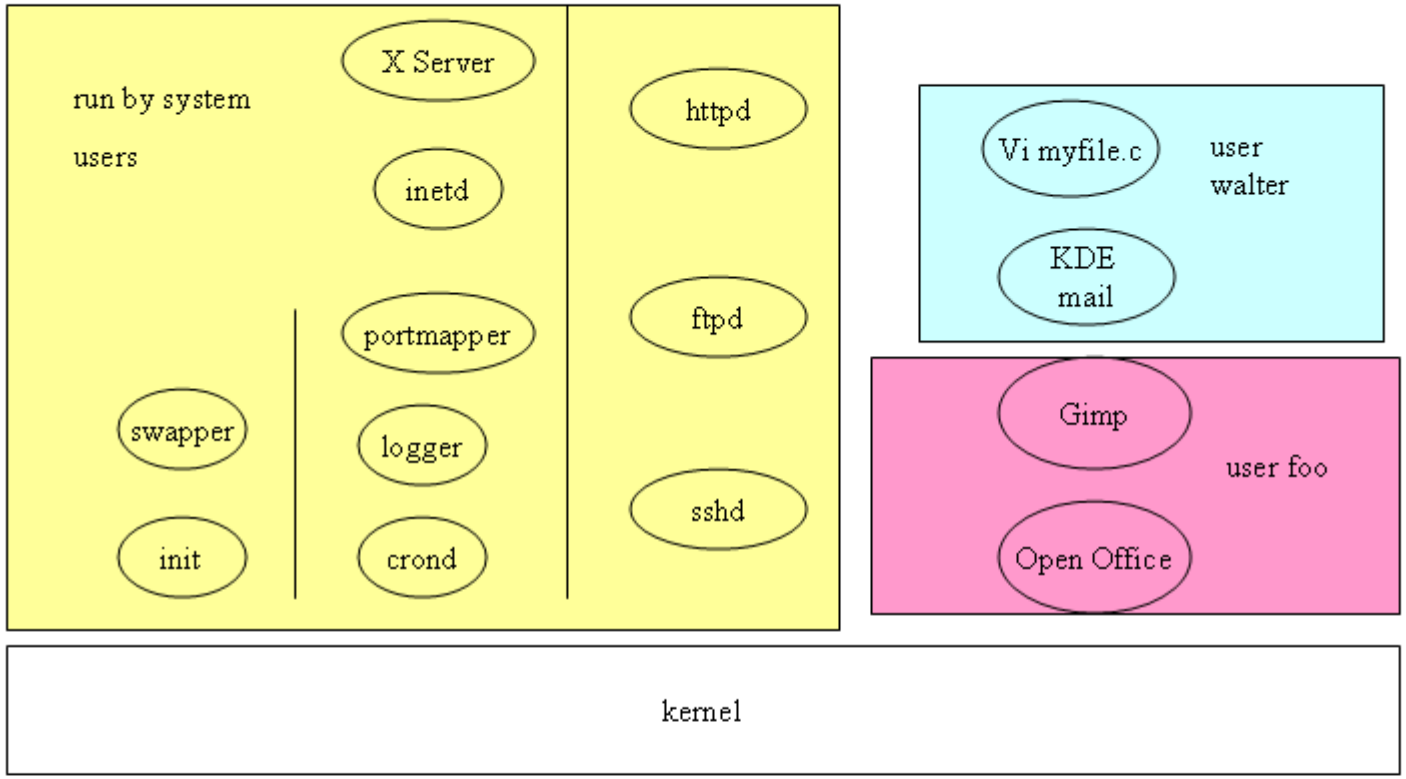
Critique: An allmighty super-user is a danger for the whole system. If „root“ makes configuration mistakes the security of the system is compromised. If a regular user downloads a virus or trojan it will usually only affect her own resources and not those of the other users. If users work under the „root“ account – even if they own the system – this security is gone. In later lectures we will talk about other (non-ACL based) security philosophies.

Linux OS components



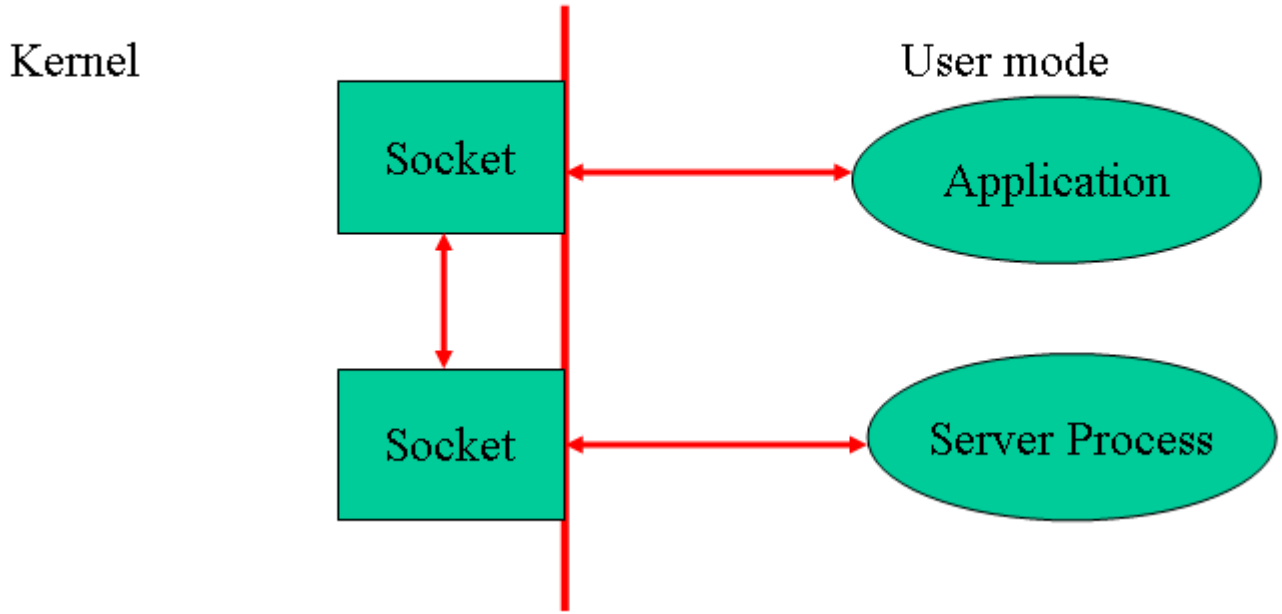
The linux kernel is monolithic but can be extended dynamically using loadable modules. These modules can be drivers for hardware or filtering components for the firewall framework netfilter etc. A standard console GUI is available and is used heavily in non-GUI applications like running a linux firewall. X Windows based applications can display there output either on the local screen or on some other X Window based station in the network.

Linux process view



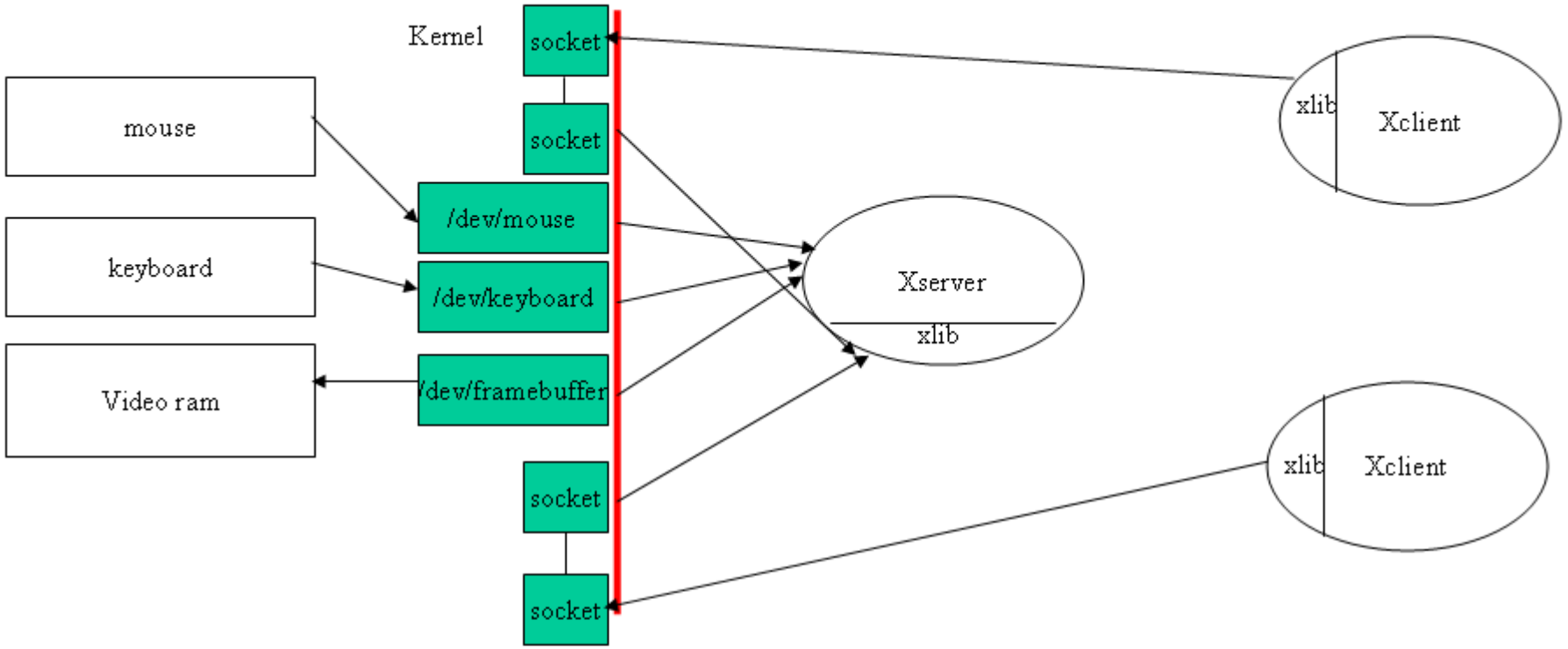
The kernel is NOT a process. It runs in the context of whatever process switched to kernel mode via the system call interface. Other OS (micro-kernel e.g.) run processes in kernel mode or put kernel functions into user processes. Init and swapper are the most important processes for the OS. Init starts new processes and swapper takes care of memory requirements. Next is a group of basic servers which are almost always needed. Unix servers are often named like xxxxd with the „d“ standing for „daemon“ – a process running without GUI in the background. Web Server (httpd) ftd server (ftpd) etc. are all optional. Finally users run processes like editors (vi, emacs) or office programs.

Interprocess Communication with Sockets



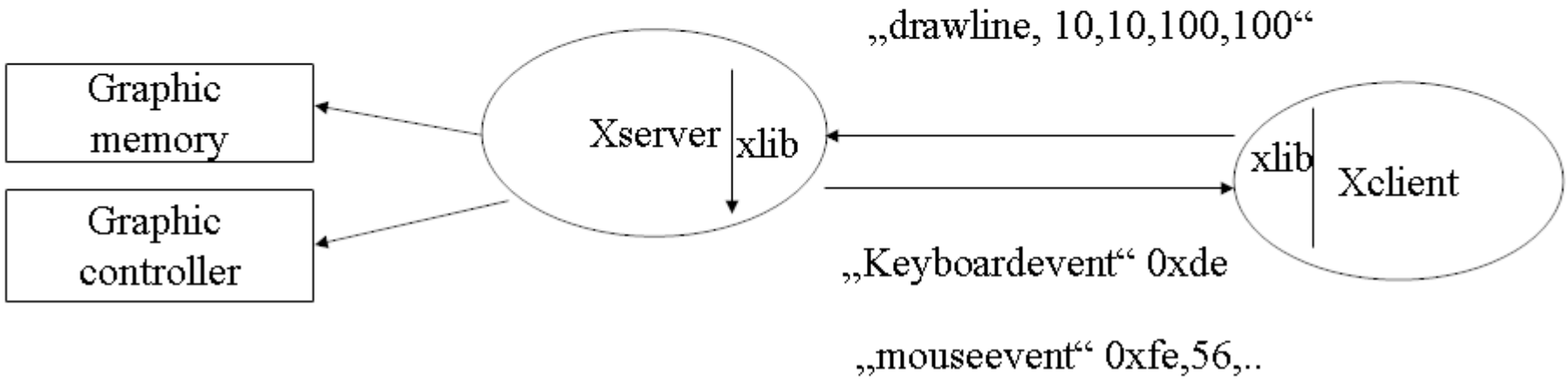
Sockets are transport endpoints defined through „hostname“, „port number“ and „protocol“. They work across machines and also locally. Typically a server process waits on a server socket for requests from clients. A client opens a socket to the server and sends requests. The connection is bi-directional and either stream or packet oriented (protocol tcp or udp). Sockets are very convenient IPC mechanisms. Lately the DBUS architecture provides an IPC layer on top of sockets for the communication of desktop applications.

The X window system: IPC



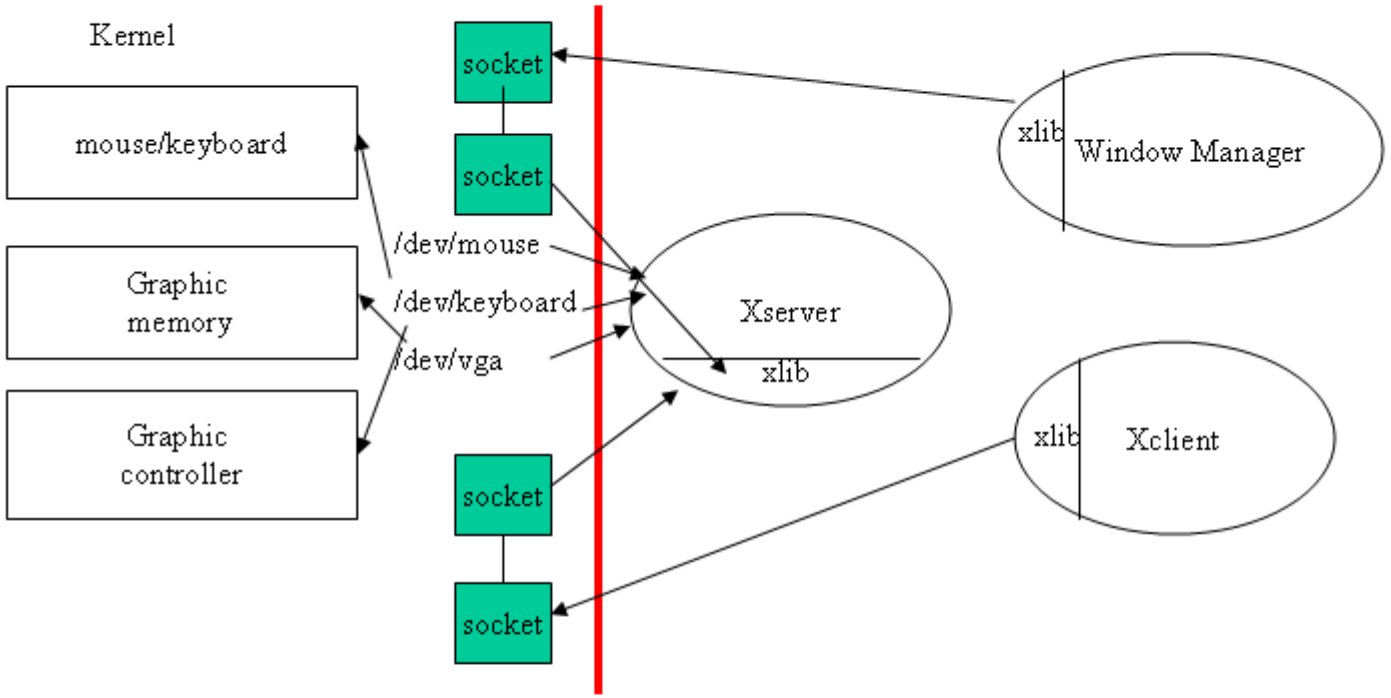
The graphic subsystem resides completely in USER space. Xclients talk via sockets to an Xserver. The X server is the only one who controls the graphic hardware (screen, keyboard, mouse, controller) through the device driver interfaces (/dev/xxx...). Clients know NOTHING about graphic hardware: The Xserver will render every command either by writing to the video memory (if only dumb hardware is present) or by issuing commands to the intelligent graphic controller. Notice: The communication endpoints are network wide: X Windows works across machines!

The X window system: API



The X Windows protocol is implemented in the X Windows library (xlib). Example: „Draw line, 10, 10, 100, 100, gc) would tell an X Server to draw a line between the coordinates and use the graphical context given for that. Depending on the hardware the XServer either has to translate the command into pixel values (e.g. Bresenham) or just turn around and issue a (perhaps slightly different formatted) command to intelligent graphics hardware. The Xclient receives mouse and keyboard events from the server. This make it clear: the XServer controls the viewing station. The graphics application can run on a machine WITHOUT any graphics hardware or display!

The X Desktops and Window Managers



X Windows does only provide mechanism, no policy, e.g. how a desktop or window manager should look. Separating mechanism and policy allows different policies to be implemented on the same platform. Reality has shown that users do not really appreciate this feature... Who wants a different user interface in every car?? This was one of the frequent cases where a clever technical idea did not meet the users demands.

Using Linux

1. Logging in: about users and administrators. Groups and permissions
2. Getting help: how to find help. Tools and tricks.
3. Where am I? about homes, the filesystem and navigation
4. Using the shell: why command lines still make sense
5. The GUI: using KDE
6. Processes and how to handle them. Concurrency, tools (ps)

While learning some basic steps we will start talking about architectural issues as well.

Finding Help

• man <command>: uses the „man“ program to find information on utilities or library functions. Use man man to get information on man itself.

• info <command>: the gnu help system

• <command –help>: displays available command line options

• kde or gnome help icon: displays help in a GUI

• which <command>: tells the absolute path of a command that will be executed if typed into at the shell command line. In other words it tells you which command really runs if several commands with the same name are installed in the file system. Simple takes the PATH environment variable and searches all directories for the specified command.

• whatis/apropos: commands which tell you about features or utilities. Require the whatis database to be installed.

• /usr/share/doc: documentation on many installed programs

Finding Files

- find . –name „*.bak“ –type f –exec rm {} \; -print

File management is an extremely common job for linux admins and users. The powerful „find“ command lets you specify

a) patterns

b) file types

c) dates

of files you are looking for.

In addition it allows you to specify a processing that should be applied to the files found (only those that match the pattern of course). In the example above „rm“ will be invoked for every file found and the file will be deleted (if you have the proper permissions). Search starts at the current directory.

Finding Text Patterns

- grep „class“ *.java

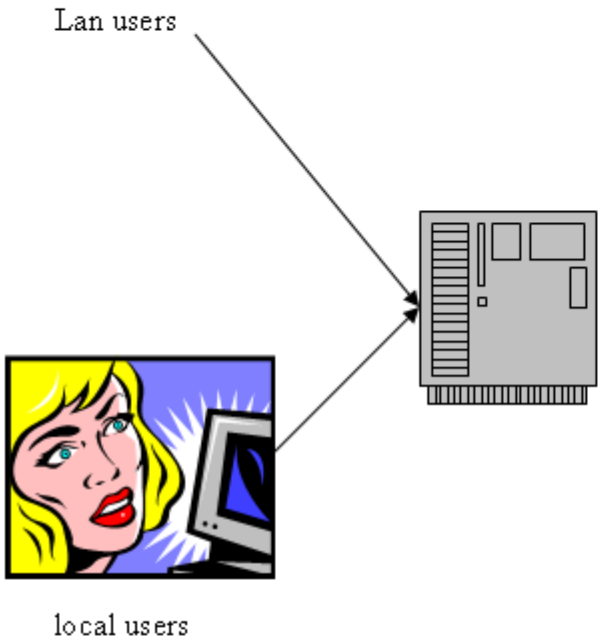
In programming searching for patterns is a frequently performed operation. The „grep“ command lets you specify a pattern using regular expressions and searches for this pattern in all the files given on the commandline (in this case alle .java files in the current directory. „grep –i“ will work case-insensitive. Don‘t forget that you can stack grep operations like grep „x“ *.c | grep „y“. This will first extract all lines containing x and then from this subset extract all lines containing y as well.

Finding Programs

- echo $PATH <return>
- /usr/local/bin;/usr/bin;.;
- which ls // searches the path for an executable
- /usr/bin/ls // tells you that ls would be executed from /usr/bin

If the shell needs to find a program to start it uses the PATH environment variable. This is quite similiar to how MS command shell works. It is also quite similiar to how Java locates loadable .class files. The mechanism is always the same: An environment variable contains several absolute or relative pathnames. The shell or java (using CLASSPATH) starts searching for the requested program in the first path location (/usr/local/bin above) and only continues searching in the rest of the variable if the program ist NOT found. The FIRST instance of the program or class found will be used. This is a) a nice feature that can be used to hide an instance of a program or class by putting a newer version of it in a path location in front of the original location. This is b) a wonderful way to shoot yourself in the foot if you forget that you are now relying on the order of PATH statements. The same mechanism also works on MS Windows with dynamic link libraries and is even more dangerous there. A wrong .dll library put somewhere „for a quick test“ has cost countless hours of debugging already.

Logging in

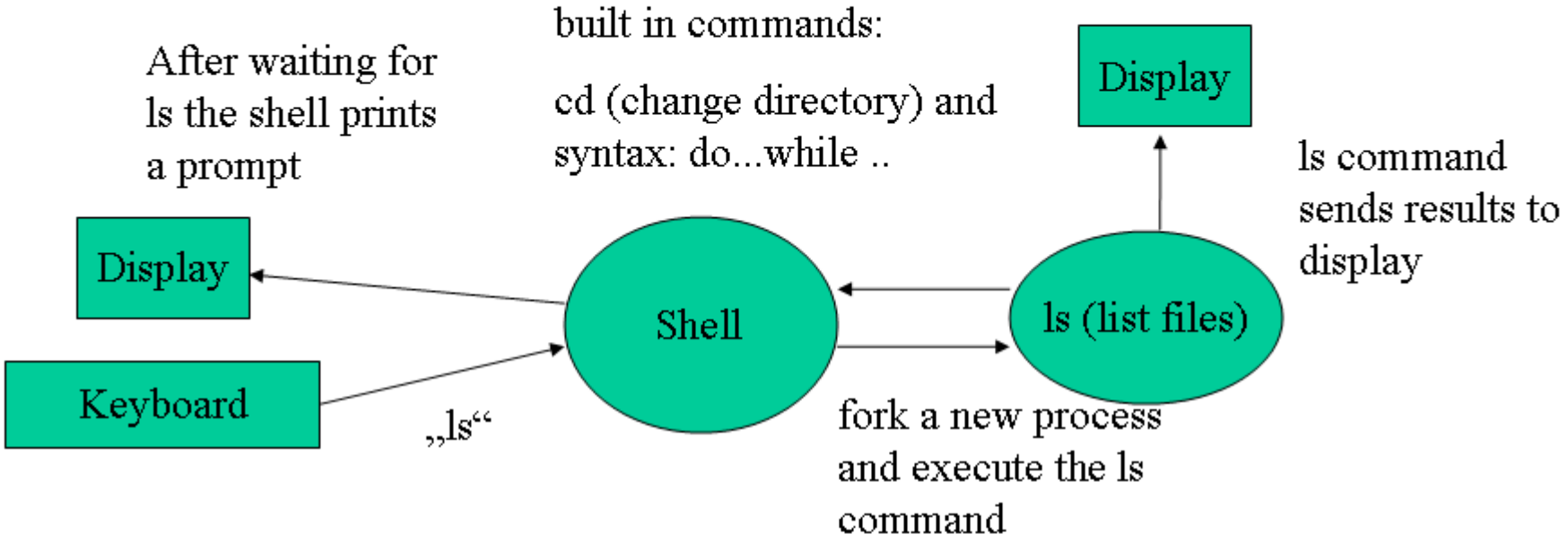


1. Authenticate user, compare credential with stored ones
2. Create shell with current directory at /home/userX
3. Process .profile, .bashrc etc. configuration files in users home directory
4. Start GUI environment (optional)
5. Check on every access to a resource that user is authorized

During log-in the users personal environment is set up. Look at the .xxx files in your home directory.

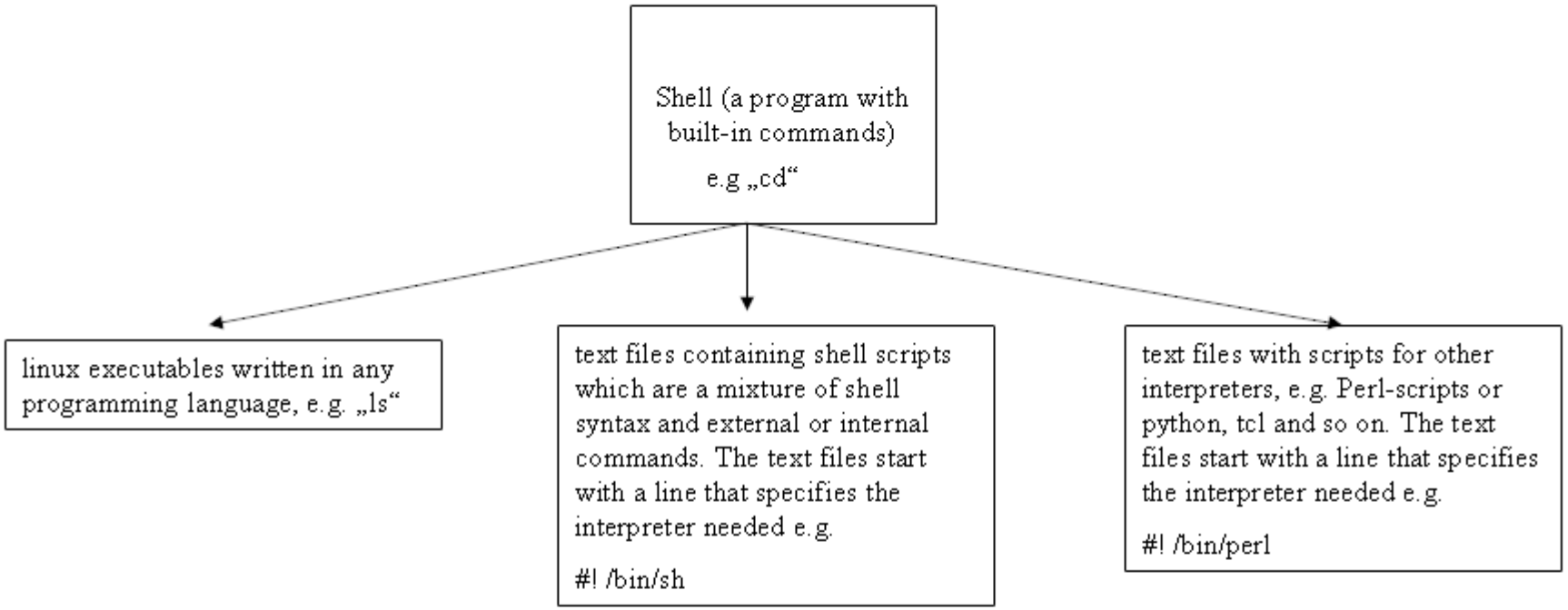
A „user“ is only an operating system user if she has an account on the machine. The account can be a network account or a local account. If a web server runs on a machine and gets some requests, those requests do NOT come from OS users. The only known user to the OS would be the web servers identity (usually „nobody“ or „www“).

The bash shell: an interpreter



A shell is a command interpreter. It reads from the keyboard and displays prompts at the console. Unix commands are built in a way that allows command chaining and also to bundle them in shell scripts: lists of commands to be executed either sequentially or in parallel. Commands usually do NOT ask back: „do you really want to ...“ or they have a „force“ parameter which suppresses warnings. Unix commands also do not include presentation information – which makes the results possible input to the next command. These features made many people think that Unix was only for „gurus“ – misunderstanding that using textual representations is only a means to achieve better automation.

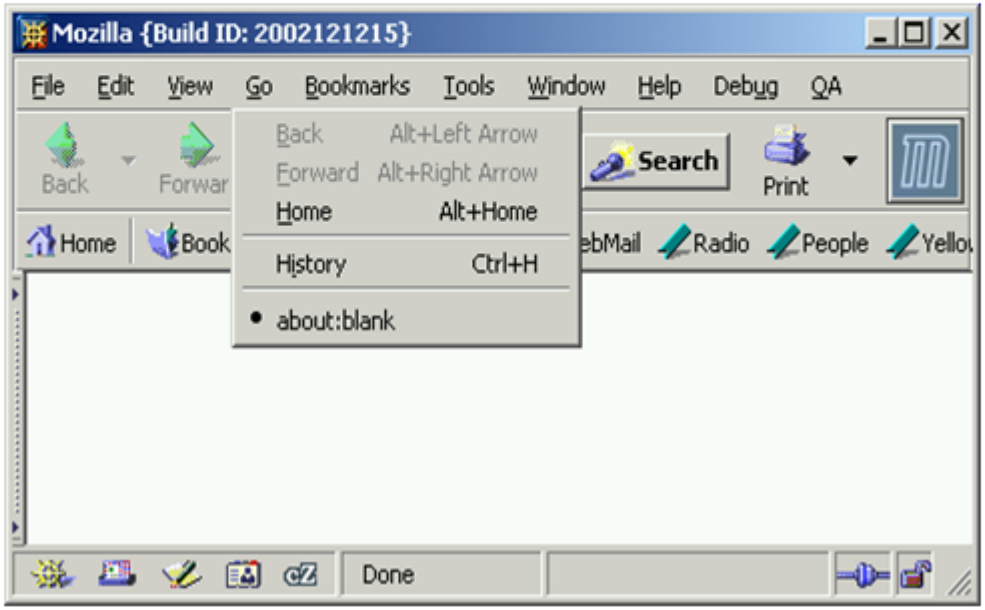
Extensibility: built-in and external commands



Most shell commands are external programs which do NOT belong to the shell code in any way. The reson why shell and programs give the impression of a tightly integrated application is simply that all programs are written to conform to certain specifications e.g. with respect to how they treat input and output.

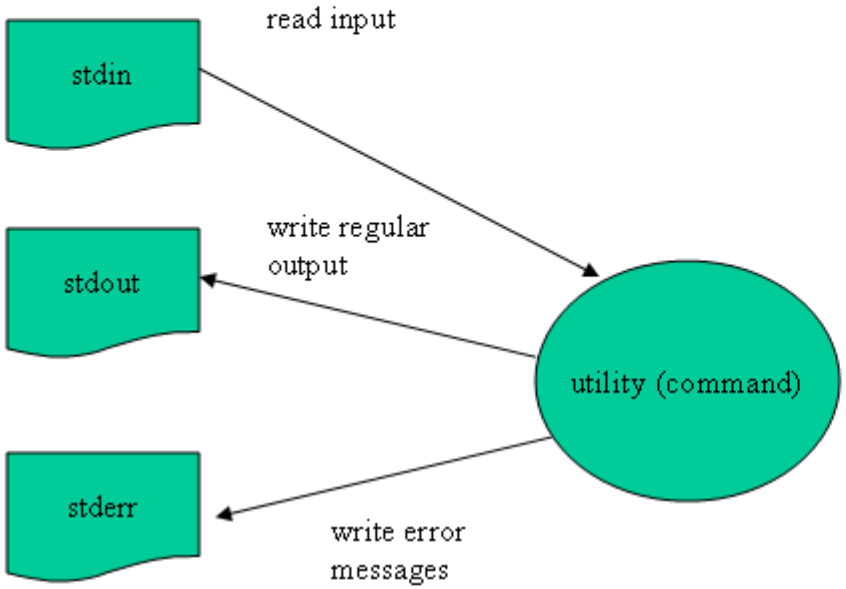
In modern speak: The operating systems and the shell form a framework with interfaces that other programs must conform to. This allows the shell to work with any new program that complies to those interfaces.

Shells vs. GUI menu systems



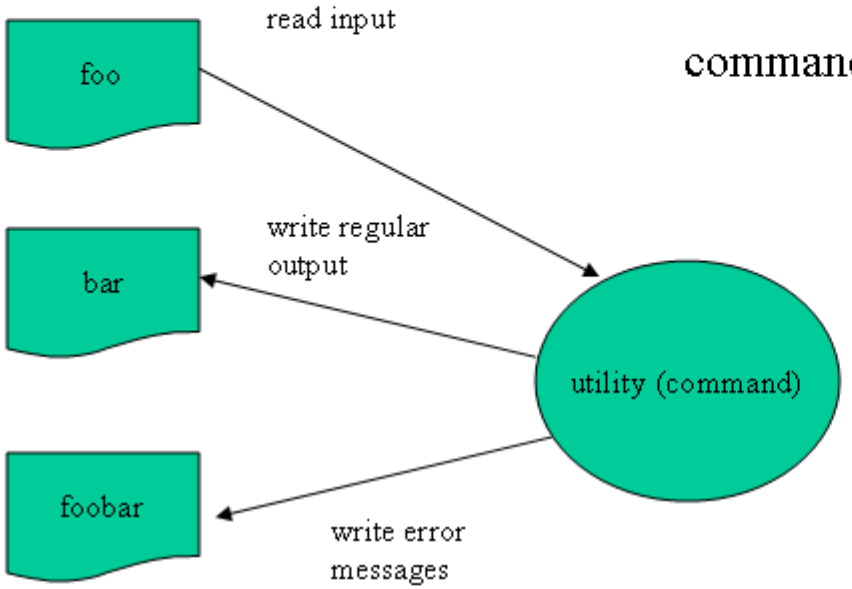
A menu system can also start external programs. But it cannot connect those programs into pipelines of processes and it has no syntax to express program statements. In other word it does not provide a turing complete language. But it is extensible the same way the shell is: All menu entries must conform to the same interface, usually an action or command object pattern with a do() method which is called by the menu system when the entry is selected. Just like the shell calls fork-exec when a command is given on the command line or a script.

Stdin, stdout, stderr



When a linux utility is started by the shell it gets three open files automatically. They are intended to provide input and output channels for the program. They can be anything but a file e.g. a pipe channel. The program does not care and uses them to read input and write results. Command line parameters can override this behavior but it is considered good unix style to write programs in this way. The principle behind is 100% OO and is called information hiding: The program does not know or care what those files really are.

Redirection



command <foo >bar 2>foobar

With the redirection characters „<„ for input and „>“ for output the standard channels stdin, stderr and stdout can be redirected. This is done by the shell which modifies the file-descriptor tables of parent and child process. The child program itself will NOT notice this if it happens at program creation time by the parent process. If the mapping is requested via command line parameters („e.g. specifying –o outputfile bar) the program will change the input and output channels to the ones requested and from that point on take all input from file „foo“ and write output to file „bar“ or „foobar“ in case of errors.

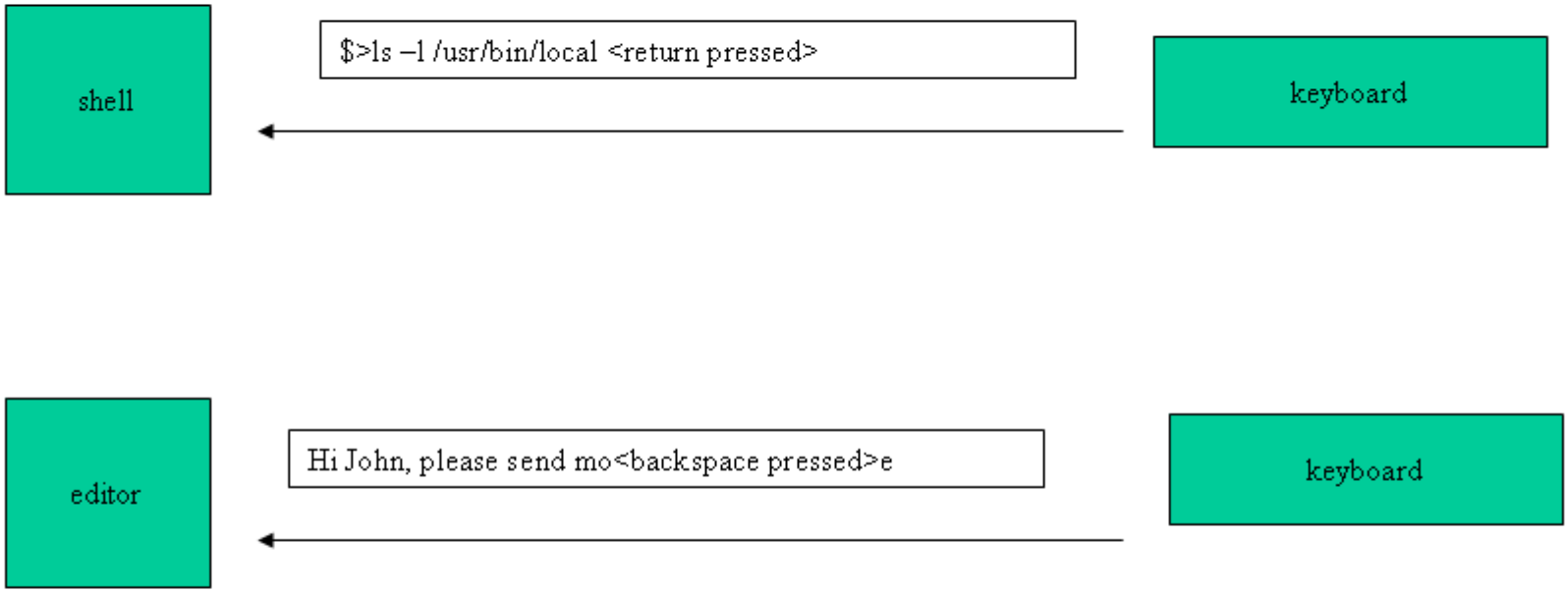
background processing with &

cat /usr/local/foo.txt & // read file foo.txt and write it to stdout. Do this in
// background

..shell waits for next command while „cat“ is still running and printing to the terminal

The ampersand character tells the shell to start the requested command in the background as a separate process and return immediately to the user for more input. Without output redirection the command above would clutter our screen with the content of foo.txt. A better solution would be: cat /usr/local/foo > /tmp/bar.txt &

Line Disciplines



Programs expect input in two fundamentally different ways. The shell is line-oriented and does not get the input until „return“ is pressed (actually newline). An editor wants to get every character the moment it is typed and not when return is pressed. The shell operates in „COOKED“ mode while the editor operates in „RAW“ mode. Unix input lines („tty“ for teletype) can be set in raw or cooked mode.

Frequently used commands

file system commands:

- ls –l (lists directory details)
- rmdir, mkdir (make and remove directories)
- cat, netcat (reads content)
- mknod (create device entry)
- ln (create links)
- tail –f (read the end from file)

finding things:

- man xxxx (find manual entry)
- info (info packages)
- find (recursively search directory tree)
- grep xxx file (look for pattern in file)
- diff fileA fileB (show differences between files)

Process management

- ps (process status)
- kill –HUP 3435 (stop process)

Tools

- vi, emacs (editors)
- pr, lpr (print utils)
- X... (GUI clients for most everything)
- netstat, ifconfig, ping, traceroute, tcp dump (network utilities)

A short overview: http://www.icgeb.res.in/~whotdr/0426_UNIX-guide.pdf

Administrating Linux

1. Creating new users and groups
2. System Configuration
3. Regular admin tasks and tools (cron, at etc.)
4. Extending the kernel: modules
5. Security configuration (firewall settings)
6. Your daily job: checking the logfiles (or if things go wrong)

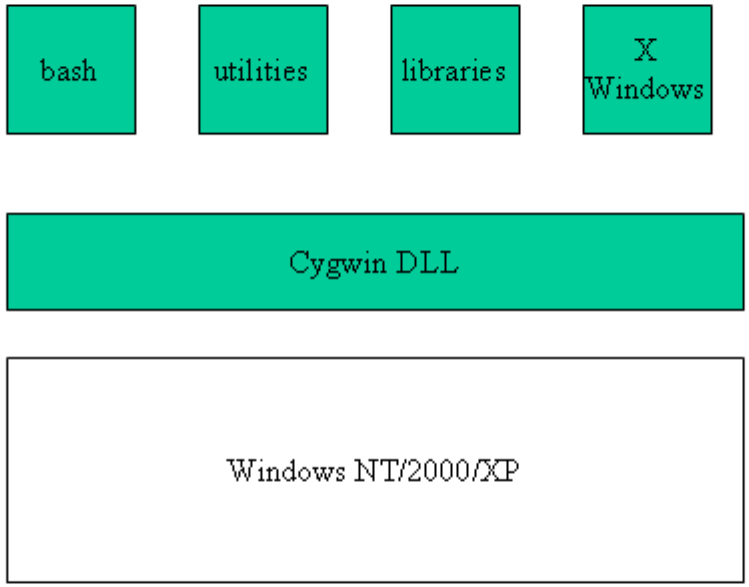
While learning some basic steps we will start talking about architectural issues as well.

Programming on Linux

1. C-compiler tools
2. Editors
3. Debugging programs
4. Makefiles
5. System Programming outlook

We will go much deeper when we start learning C on Linux to do some real system programming.

Unix environments on NT



Once you get used to the unix utilities you'd like to use them on other platforms as well, e.g. the gnu binutils and compiler to generate code for embedded control. The cygwin environment maps unix calls to the windows32 API. Almost all unix utilities are available. See www.cygwin.redhat.com

Resources (1)

- The linux documentation project. http://www.tldp.org Always check here first for good literature on linux. Excellent guides and tutorials on all aspects of linux.
- http://lwn.net/ linux weekly newsletter. The source for the linux guru.
- Sarwar, Koretsky, Sarwar, the linux textbook. A good collection of practical exercises with the most important linux tools.
- Klein, Linux Sicherheit. In German. Covers most aspects of building and running a secure Linux system (both host and network security)
- A quick guide to Unix, http://www.icgeb.res.in/~whotdr/0426_UNIX-guide.pdf Two pages with Unix commands – keep it with you at all times! Same stuff is available for vi and emacs.

Resources (2)

- Andrew Tanenbaum, Modern Operating Systems, 2nd edition. Case Study 1: Unix and Linux. As always from Tanenbaum an excellent explanation of Unix and Linux.
- **High-performance Linux clustering*** With the advent of clustering technology, supercomputers can now be created for a fraction of the cost of traditional high-performance machines. This article introduces the basic concepts of high-performance computing with Linux cluster technology. http://ibm.com/developerworks/ecma/campaign/er.jsp?id=127052
- **Anatomy of the Linux boot process*** This article describes the most common traits of embedded Linux distributions that people employ on x86 hardware and contrasts some of the different options frequently seen on non-x86 embedded systems. http://ibm.com/developerworks/ecma/campaign/er.jsp?id=127053
- Linux Kernel Device Driver Kit by Greg Kroah-Hartman http://kernel.org/pub/linux/kernel/people/gregkh/ddk/ How to create well behaved drivers for Linux. Driver development is a sore spot for Windows as well and causes frequent instabilities. Better architecture: micro-kernal design with user level drivers.