Introduction to Computer Networks and the Internet
COSC 264

Unices: History and User Perspective

Dr. Andreas Willig

Dept. of Computer Science and Software Engineering
University of Canterbury, Christchurch

UoC, 2019

Module Aims and Scope

- Broad overview on Unices, their history and user perspective
- We will mostly use Linux as a representative
- Major references: [1], [2]

Outline

UNIX and Unix-like Operating Systems

- UNIX is:
  - A trademark owned by The Open Group
    `http://www3.opengroup.org/`
  - Defined by the “Single UNIX specification”
  - A set of operating systems conforming to this specification
- Unix is:
  - A collection of related operating systems being compatible to certain programming interfaces (e.g. IEEE POSIX)
  - A superset of UNIX operating systems
- All Unices are:
  - multi-tasking: several processes and programs can run in parallel
  - multi-user: different users can work at the same time
  - Interactive: users submit commands and receive responses on-line

Major Steps of UNIX History

1969	Beginning	Ken Thompson and Dennis Ritchie (†) from Bell Labs start writing the first version in Assembler on a DEC PDP-7
1973	Fourth Ed.	UNIX became portable after re-write in C
1975	Sixth Ed.	V6 starts being used outside Bell Labs (e.g. at Universities), first BSD derived from V6
1980	Xenix	Microsoft introduces Xenix
1983	System V	UNIX System Development Lab formed, System V released
1984	4.2 BSD	UC Berkeley releases 4.2 BSD, including first TCP/IP stack. System V Version 2 released, up to 100,000 registered users

Major Steps of UNIX History (2)

1986	4.3 BSD	UC Berkeley releases 4.3 BSD, including name server. Sun delivers NFS, IBM announces AIX. 250,000 installations.
1987	SVR3	SVR3 is released, SGI releases IRIX
1988	POSIX.1	POSIX.1 is published, specifying (parts of) Unix by system call interfaces
1989	SVR4	SVR4 unifies features from System V, BSD and Xenix. Motif 1.0 appears. 1.2 Million installations
1990	OSF/1	DEC introduces OSF/1, Bell Labs presents Plan 9
1991	Linux	Linus Torvalds starts Linux development on 386 platform. Sun presents Solaris (SVR4 based), replacing SunOS (BSD based)

Major Steps of UNIX History (3)

1993	4.4 BSD	Final release of BSD, provides starting point for BSD derivates (FreeBSD, NetBSD, OpenBSD)
1994	Single UNIX	The Single UNIX specification is released
1997	UNIX V2	New version of Single UNIX includes real-time, threads and 64-bit processors
1999	Linux 2.2	Linux kernel V 2.2, DEC releases Tru64
2001	UNIX V3	Single UNIX V3 unites various specifications, including IEEE POSIX. Linux kernel 2.4 released
2003	Linux 2.6	First version of kernel 2.6 released (maintained until 2011)
2007	Mac OS X	Apples Mac OS X certified to UNIX V3

Major Steps of UNIX History (4)

- This history is an excerpt from
  `http://www.unix.org/what_is_unix/history_timeline.html`
- Contemporary open-source Unices include:
  - Linux (`www.kernel.org`), various distributions (Debian, Ubuntu, OpenSUSE, Fedora, RHEL, ...)
  - FreeBSD (`www.freebsd.org`)
  - NetBSD (`www.netbsd.org`)
  - OpenBSD (`www.openbsd.org`)
- Contemporary commercial Unices (and UNIXes) include:
  - Solaris (Oracle, previously Sun)
  - AIX (IBM)
  - HP-UX (HP)
  - Mac OS X (Apple)

Linux and GNU

- Started by Linus Torvalds in 1991
- Linux is an open-source Unix-like operating system kernel
- A Linux distribution (e.g. Fedora) adds a lot of things to the kernel:
  - Compilers and various libraries
  - Shells, GUIs, editors, . . .
  - Various command-line and other tools
  - . . .

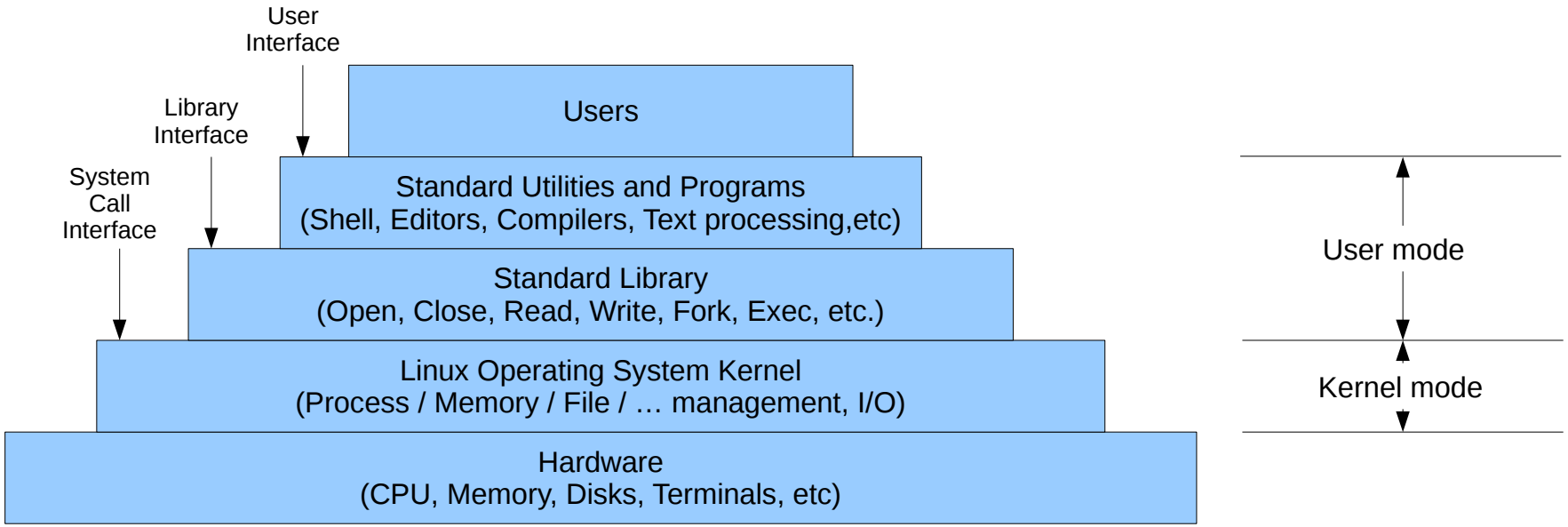
  which are collectively referred to as **userland**
- Many userland tools are provided by the Free Software Foundation / GNU project
- Linux is **not** certified as a UNIX, but Unix-like

Linux

- Some key Linux properties and features:
  - Monolithic kernel with loadable modules (e.g. device drivers)
  - It clearly distinguishes between kernel and userland
  - It supports multi-tasking and multi processors
  - It supports dynamic shared libraries
  - It provides memory protection for processes
  - It runs on many different platforms, including:
    - x86
    - Itanium
    - s390
    - SPARC
  - It does not have a default graphical user interface (there are several), but text shells are available everywhere
    - And on smaller embedded boxes often the only interface
- From now on we focus mostly on Linux

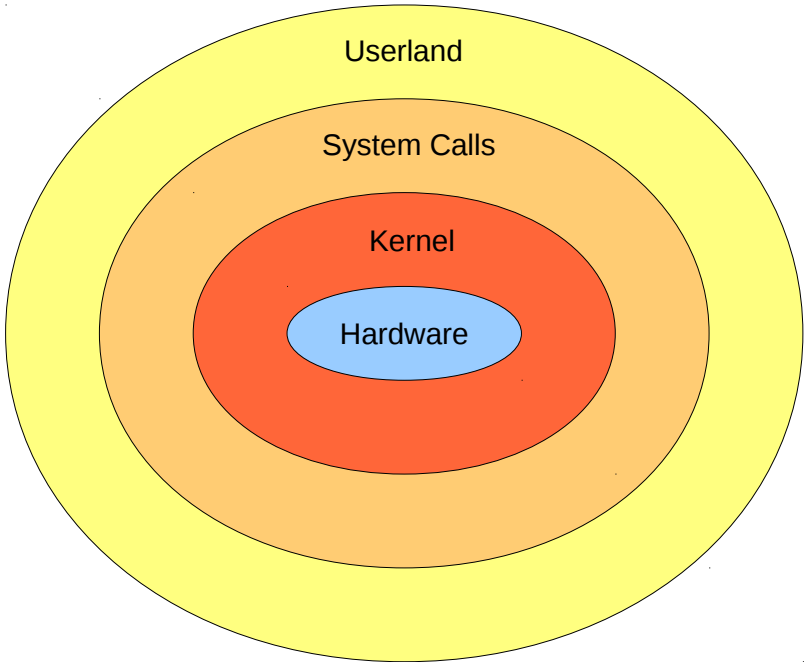
Outline

Conceptual Layers and Interfaces



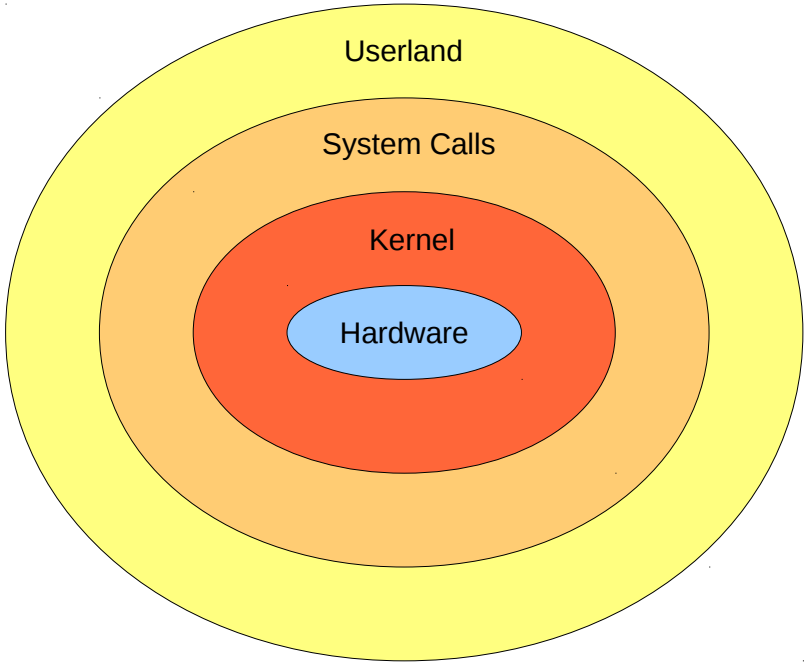
(From: [2, Fig. 10-1])

User Space



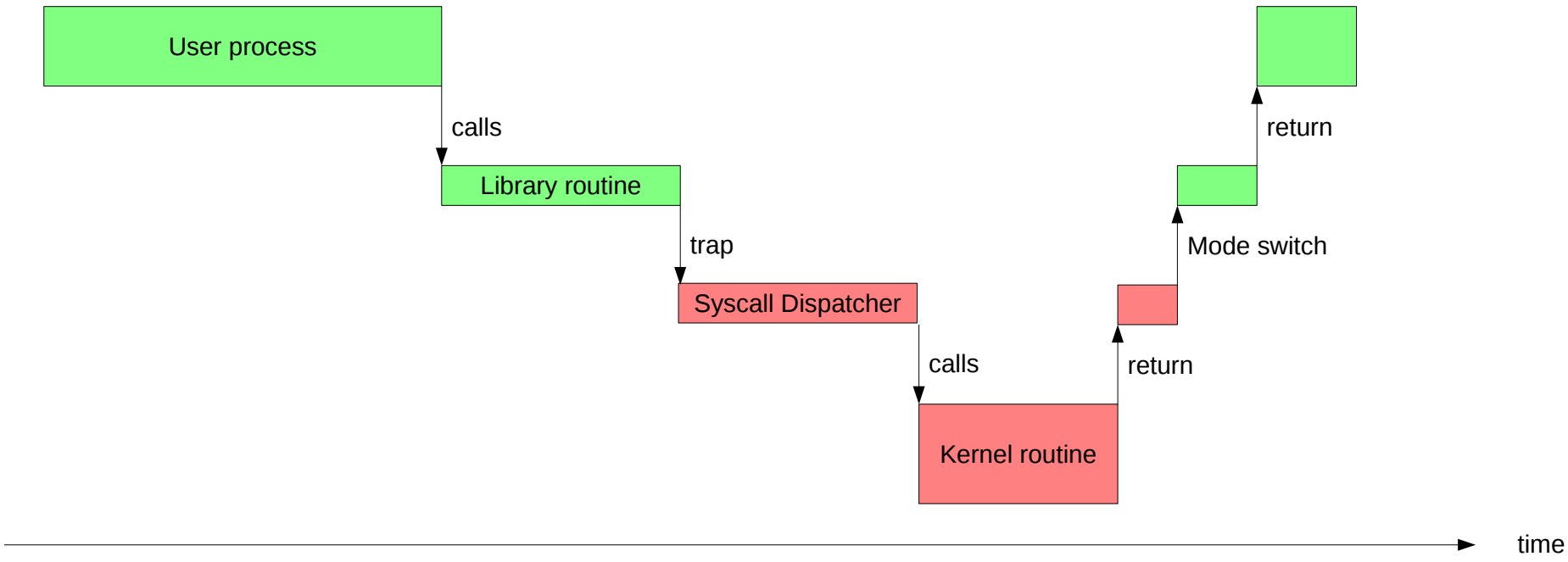
- User programs run in user space
- They have own memory areas separated from other user programs, user processes cannot interfere with each other
- User programs have no direct hardware access, instead send requests to kernel through system calls

Kernel Space



- Kernel has hardware access privileges, serves requests of user processes, returns responses
- Linux kernel runs own kernel threads for management purposes
- Kernel data structures are not accessible by userland processes (memory protection!) but are accessible by all kernel threads
- Kernel and userland processes run in different CPU modes – processes running in user mode trap when they directly access hardware or foreign memory areas

Mechanics of System Calls

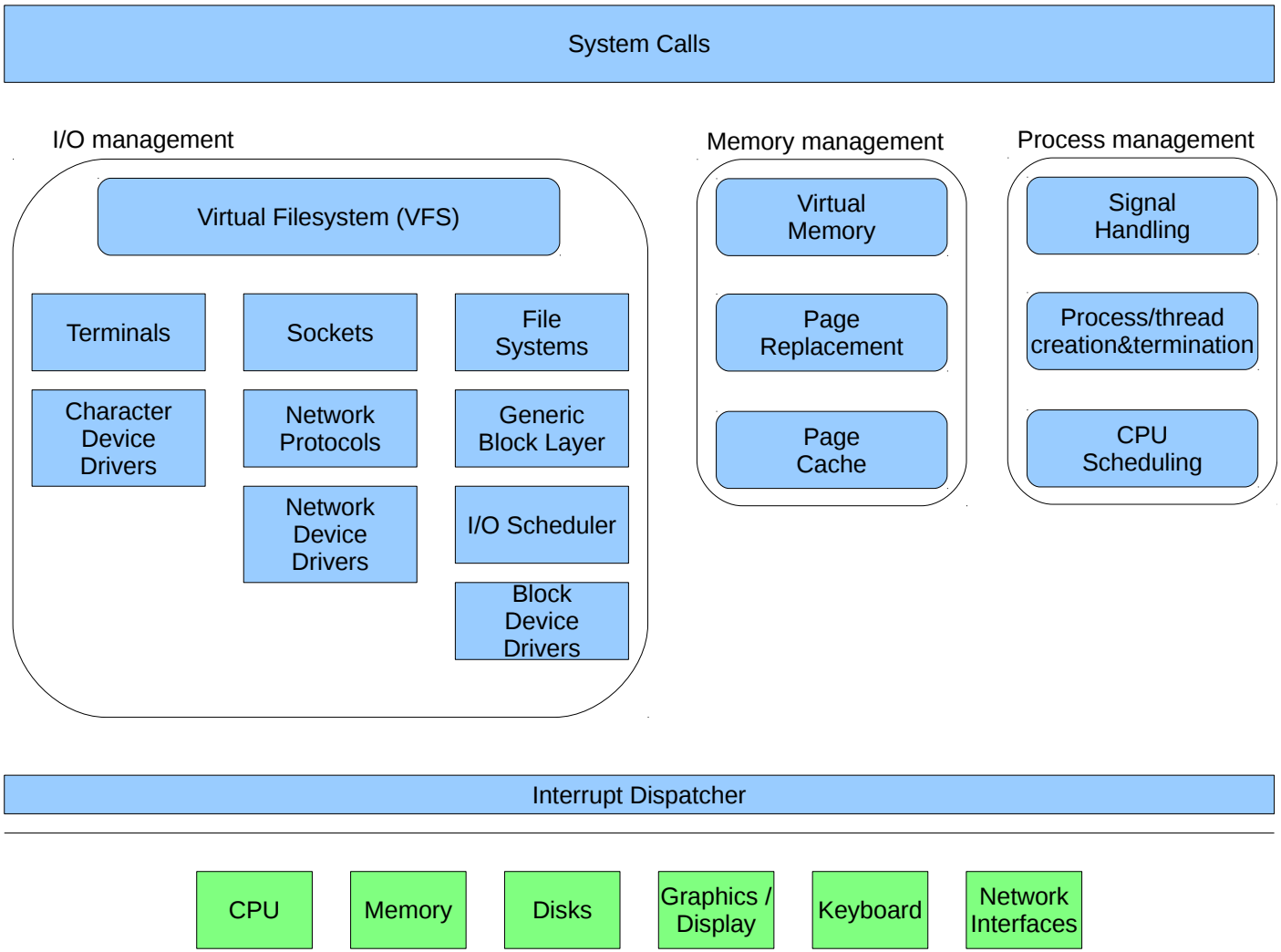


- User process calls a library routine (e.g. from C standard library, `glibc`)
- Library routine packs parameters into special data structure and traps, i.e. creates an exception (e.g. using invalid CPU instruction)
- Kernel exception handler invokes system call dispatcher
- System call dispatcher determines kernel routine to call, calls it
- Kernel routine returns to library routine (mode switch), then to user process

System Calls

- Linux supports very many system calls, see
  - `man 2 intro` for an introduction to system calls
  - `man 2 syscalls` for a list of supported system calls

Linux Functional Architecture



(adapted from: [2, Fig. 10.3])

Outline

Conventions

- Commands entered as normal user:

```
$ ls
file.a  file.b  file.c
```

  - Lines starting with `$` (command prompt) contain commands entered into the shell
  - Following line contains output of given command

- Commands entered as root user:

```
# ls
file.a  file.b  file.c
```

  - Lines starting with `#` (command prompt) contain commands entered into the shell as root user

Basic Concepts

- **Shells** let users enter commands, then they run those commands and display output to user
- Unices traditionally provide **command-line shells**, where users type text commands, output is also displayed as text
- Nowadays graphical interfaces are sometimes also referred to as shell, but here we focus on textual shells
- Different shells have emerged over time, including:
  - `bash` (Bourne Again SHell)
  - `csh` (C SHell)
  - `tcsh` (TENEX C shell, an extended C shell)
  - `zsh` (Z shell)
- We work with the `bash` shell from now on

Basic Concepts (2)

- A key design concept of Unices is:
  - One individual command provides only one or very few related functionalities, but implements them very well
  - Unices provide a large set of individual commands
  - Several commands can be combined using various combinators, for example piping allows to provide the output of one command as input to the next command
  - Combination of commands is helped by some conventions:
    - Files have no other internal structure, are just a sequence of bytes / characters
    - Many files are stored in text formats (human-readable), binary representations are less useful for piping

The Most Important Command for Beginners

- Unices often provide online documentation ('manual') for many available commands
- The command `man` displays these manual pages
- Examples:
  - `man ls` displays the manual page for the `ls` command
  - `man man` explains how to use the `man` command
- `man` pages are categorized into sections, sometimes pages of the same name exist in different sections, then the page name must be preceded by section number
- Linux distributions contain lots of GNU software, and GNU prefers the `info` command for providing documentation

Shell Capabilities

- Most shells include the following functionalities:
  - Accepting user inputs and executing commands
  - Chaining of commands via pipes, I/O redirection
  - Ability to run shell scripts
  - Command / filename completion, command line editing
  - Maintain a command history, recall earlier commands
  - Providing contextual information in **environment variables**

Exercise

Log into a Linux box, open a text terminal, start the `bash` and familiarize yourself with command line editing, the file completion facility, and the command history! See the `READLINE` section of the `bash` manual page.

Chaining Commands Together: Pipes

- Many commands take input data, almost all generate output data
- Commands can obtain input data in two different ways:
  - First example:

    ```
    $ grep hello myfile
    ```

    - Searches for occurences of the string `hello` in file `myfile`
    - Shell passes the filename `myfile` as parameter to the `grep` command, which opens the file and reads its contents
  - If no input filenames are given, the `grep` command reads input from a special file called **standard input** (`stdin`)
  - `stdin` is a special file not existing in the file system, but is created within the shell and made accessible to a command

Chaining Commands Together: Pipes (2)

- All commands write their output to another special channel called the **standard output** (`stdout`)
- Again, to a command `stdout` looks like a (write-only) file, but it does not exist in the filesystem, only within the shell
- With the pipe symbol ('`|`') the `stdout` of one command can be linked to the `stdin` of another command
- Examples:

```
$ ls
file1  file2  file3
```

  here

  - The `ls` command prints all filenames in current directory and writes them in sorted order (ascending) to its `stdout`
  - The shell reads the data from the `stdout` of the `ls` command and prints it into the terminal

Chaining Commands Together: Pipes (3)

- Another example:

```
$ ls | sort -r
file3
file2
file1
```

  where the command `sort -r` reads data from its `stdin`, sorts it and outputs it in reverse order to its `stdout`
- In this example the shell:
  - creates one **anonymous pipe** which contains a buffer, and connects the `stdout` of the first command (`ls`) to `stdin` of the second command (`sort -r`) via the anonymous pipe
  - starts the two commands (`ls` and `sort -r`) in parallel
  - reads data from `stdout` of second command and prints it
  - **Question**: Why is a buffer useful? How would you synchronize?
- A second output channel `stderr` is for error reporting

Chaining Commands Together: Pipes (4)

- More generally, you can think of a command like:

  ```
  $ cmd1 | cmd2 | cmd3 | cmd4 | cmd5
  ```

  as source data (generated by `cmd1`) flowing from the left to the right through a sequence of filters
- A simple command can be either of:
  - a `bash` builtin command, e.g. `export`
  - an executable program located in one of the directories the `PATH` environment variable points to

Exit Codes

- As a convention, each simple command (like `ls`) returns a separate **exit code** to the calling program (here: the shell):
  - Exit code is a 16 bit integer
  - If exit code is zero: program completed successfully
  - If exit code is non-zero: program experienced error, exit code indicates (program-specific) type of error
- The exit code of a piped command

  ```
  $ cmd1 | cmd2 | cmd3 | cmd4 | cmd5
  ```

  is determined as follows:
  - If any of the commands `cmdx` exits with a non-zero exit code, the execution of the chain is aborted and the exit code of the chain is the exit code of the offending command
  - Otherwise the chain has exit code of zero

Further Command Combinators and Operators

- In the following a command is either:
  - A simple command, or
  - A piped command `cmd1 | cmd2 | ... | cmdn`, perhaps surrounded by brackets
- Some standard combinators and operators:

`cmd > file`	The `stdout` of `cmd` is not printed but re-directed into regular file `file`
`cmd » file`	`stdout` of `cmd` is appended to file `file`
`cmd < file`	`stdin` of `cmd` is taken from file `file`
`cmd1 ; cmd2`	runs first `cmd1`, and then `cmd2` no matter the exit code of `cmd1`
`cmd1 && cmd2`	runs first `cmd1`, and if this has zero exit code, runs `cmd2`
`cmd1 || cmd2`	runs `cmd2` if `cmd1` has non-zero exit code

Further Command Combinators and Operators (2)

- When the shell executes the command

  `` `cmd1` ``

  enclosed in backticks, it first runs the command `cmd`, substitutes its `stdout` into the initial command string and executes the resulting string
- For example:

```
$ which ls
/bin/ls
$ ls
file1  file2  file3
$ `which ls`
file1  file2  file3
```

- Another example:

```
$ echo Today is `date '+%d.%m.%y (%a)'`
Today is 11.06.12 (Mon)
```

Input Processing, Expansions and Quoting

- When the user submits a command, the shell:
  - first splits the command line into tokens ("word splitting")
  - then applies a range of expansions to the command line, yielding a sequence of modified command lines, and
  - executes the resulting command line
- Expansions include:
  - Command substition: `` `cmd` `` is replaced by `stdout` of `cmd`
  - Tilde expansion: '~' is replaced by the name of the users home directory, for example the command

    ```
    $ ls ~
    ```

    is expanded to become the command

    ```
    $ ls /home/awillig
    ```

  - Filename expansions (see next slide)
- See the `bash` documentation for full range of expansions and their precise order of processing

Filename Expansion

- Shell checks each word whether one of '*', '?', or '[' occurs
- Words containing '*' are interpreted as wildcard pattern for filenames and replaced by all matching filenames where '*' represents zero or more arbitrary characters, e.g.:

```
$ ls
file1  file2  file3  testfile  filtra  foo-bar-1 foo-bar-2
$ ls f*
file1  file2  file3  filtra foo-bar-1 foo-bar-2
$ ls f*1
file1  foo-bar-1
$ ls file*
file1  file2  file3
$ ls *file
testfile
$ ls *file*
file1  file2  file3  testfile
$ ls *fil*
file1  file2  file3  testfile  filtra
$ ls /bin/f*
/bin/false  /bin/fgconsole  /bin/fgrep  /bin/fuser
```

Filename Expansion(2)

- Words containing '?' are replaced by all matching filenames where '?' represents exactly one arbitrary character, e.g.:

```
$ ls
file1  file2  file3  fale  fole  foo-bar-1
$ ls file?
file1  file2  file3
$ ls f?le
fale  fole
$ ls f?le?
file1  file2  file3
```

- '[...]' matches exactly one of the characters mentioned in '...', e.g.:

```
$ ls
file1  file2  file3  filea
$ ls file[0-9]
file1  file2  file3
$ ls file[0-9a-z]
file1  file2  file3  filea
```

Some Simple Commands

`grep`	Searches for occurences of text matching a regular expression in one or more files (or `stdin`)
`sed`	Regexp-based search-and-replace
`vi`	a powerful screen editor, available almost everywhere
`emacs`	another, even more powerful editor
`find`	search directories recursively for file(s) satisfying given conditions
`more`	display contents of a file, one screenful at a time
`top`	shows continuously resource consumption (CPU, memory, etc.) of all running processes
`sort`	sorts its input (in-memory), writes sorted result to `stdout`
`uniq`	replaces subsequent duplicate lines by one
`date`	writes (formatted) date and time to `stdout`
`fortune`	displays fortune cookies
`tar`	creates (tape) archive files
`gzip`	compression
`od`	dump files in octal and other formats
`mv`, `cp`, `rm`	move, copy, remove / delete files

Environment

- The shell maintains a set of environment variables, which can be set and cleared by shell users (and shell scripts)
- Shell variables are for example used to:
  - hold configuration data (e.g. `EDITOR` specifies which editor to run when other programs like `svn` invoke an editor)
  - point towards directories, for example:
    - `PATH` contains a list of directories in which the shell searches for executables
    - `LD_LIBRARY_PATH` lists directories in which executables search for dynamic libraries
    - `TEXINPUTS` lists directories where LaTeX searches for files
  - store runtime data, for example:
    - `USER` contains login name of current user
    - `GROUP` contains group name of current group
    - `HOME` contains home directory of current user
- Shell variables are also expanded by the shell, you can refer to them by `$VARNAME` or `${HOME}`

Environment (2)

- Important commands for environment variables:
  - `printenv` without parameters prints all variables
  - `printenv VAR` prints value of variable `VAR`
  - `export MYVAR=VALUE` assigns value `VALUE` to variable `MYVAR`

Final Words about Shells

- We haven't even seen the tip of the iceberg, the `bash` (and the other shells) have many many more capabilities
- Shells provide entire programming languages (including loops, assignments, etc.), can be used to write shell scripts
  - This is what you explore on the first problem sheet
- Shell scripts are useful for system administration tasks, writing larger scripts usually not a good idea

Outline

Basic Concepts

- Unices are multi-user operating systems:
  - Several users can run programs / processes in parallel at the same time and independently of each other
  - Users can be subjected to resource limitations (e.g. disk quotas) and accounting
- Some implications:
  - Users must be identified with unique user id's
  - Users must authenticate, OS provides authentication mechanism (login procedure)
  - Users should be protected against mal-behaving other users, including memory protection, own mail folders etc.
- Several users can be added to a **group**, group members can have access to resources (files, devices, etc.) belonging to this group

Login name and User-ID

- Each user is identified by a 16-bit **user-id** (UID)
- To a UID belongs a **login name** and further attributes:
  - login name = the name you use for logging into the system
- Linux keeps login information in file `/etc/passwd`, e.g.:

```
$ cat /etc/passwd | grep awillig
awillig:x:1000:1000:Andreas Willig,,,:/home/awillig:/usr/bin/tcsh
```

  where:
  - `awillig` is the login name
  - `x` is **not** a password (but used to be in older Linux versions, now encrypted passwords reside in `/etc/shadow`)
  - The first `1000` is the UID
  - The second `1000` is group-id (GID) of users default group – each user has its own group that is his default group
  - Further fields indicate full name and details of user, its home directory and its login shell

Login name and User-ID (2)

- A user can change some of its login settings:
  - Command `chsh` specifies a new login shell
  - Command `chfn` changes full user name and details
  - Command `passwd` changes password of user
- Further useful commands related to users:
  - `whoami` prints the own login name
  - `who` tells who is currently logged into the system (and how many shells they use)
  - `quota` (when installed) shows how much space on disks is allocated to a user and how much he uses

The Root User

- There is a special user, the super-user or **root** user – it:
  - has UID 0 and GID (group-id) 0
  - has full privileges: it can create, edit or remove any file of any user, change file attributes, change disk quotas, etc.
  - is the only user allowed to carry out certain activities, e.g. changing system configuration

Warning and Example

Never, ever work as root unless you really have to (e.g. for configuration work). You can destroy parts or all of your system!

```
# cd /
# rm -r tmp/*
```

One additional blank erases **all** files in the system, there is nothing like a 'recycle bin' ...

Groups

- Users can join several distinct groups
- All group members have same access right to resources belonging to the group, e.g. files
- Existing groups are listed in file `/etc/group`, e.g.:

```
$ cat /etc/group
...
floppy:x:25:awillig
tape:x:26:
sudo:x:27:
audio:x:29:awillig,pulse
...
```

where

  - First entry of line gives name of group
  - Second entry gives group password
  - Third entry gives GID
  - Fourth entry lists members of group (list of UIDs)

Groups (2)

- A user can be member of several groups, but at any time has only one “current” group, which is applied when evaluating access rights
- When logging in, its current group is its default group (the one listed in `/etc/passwd`)
- The command `groups` lists the groups that current user belongs to (according to `/etc/group`), current group is listed first
- User can change its current group with command `newgrp`

Users, Groups, and File Permissions

- Unices distinguish three different operations on a file:
  - read
  - write
  - execute (i.e. run as program)
- Unices distinguish three different parties on a file:
  - The owner, which is an individual user (represented by UID)
  - The group to which file belongs (represented by GID)
  - The rest of the world
- Permissions specify for each party the allowed operations

Users, Groups, and File Permissions (2)

- Let us look at a file permission:

```
$ ls -al /etc/passwd
-rw-r--r-- 1 root root 1661 Feb 25 16:06 /etc/passwd
```

- First character indicates file type:
  - '-' is a regular file
  - 'd' is a directory
  - 'l' is a symbolic link
  - 'b' is a block special file
  - ...
- Next nine characters give the mode bits:
  - First three show rights to read, write, execute for file owner
    - Here: owner may read and write, but not execute
  - Next three show rights to read, write, execute for group
  - Final three show rights to read, write, execute for world
- There is also a **sticky bit**, you find out about it ...

Users, Groups, and File Permissions (3)

- Continued:

```
$ ls -al /etc/passwd
-rw-r--r-- 1 root root 1661 Feb 25 16:06 /etc/passwd
```

- First `root` specifies owner of the file
- Next `root` specifies group of the file
- `1661` is the size of the file in bytes
- `Feb 25 16:06` is time and date of last modification of file
- last is the name of the file

Beware

This is only the 'typical' output, there are many more variants (see the `man` or `info` pages for `ls`).

Users, Groups, and File Permissions – Quiz

- Suppose that:

```
$ whoami
awillig
$ groups
awillig cdrom floppy audio dip video plugdev netdev bluetooth
$ ls -al /etc/passwd
-rw-r--r-- 1 root root 1661 Feb 25 16:06 /etc/passwd
```

- Do i have read access / write access to this file?
- And how in the example:

```
$ whoami
awillig
$ groups
awillig cdrom floppy audio dip video plugdev netdev bluetooth
$ ls -al /etc/passwd
-rw-r----x 1 root root 1661 Feb 25 16:06 /etc/passwd
```

Outline

Files

- Files are just sequences of bytes, content is not interpreted
  - Further structure can be added by additional libraries, e.g. for sound files, JPEG files, etc.
- Unices represent (almost) everything as file, including
  - Regular files
  - Block devices (e.g. hard disks) and character devices (e.g. serial interfaces)
  - Pipes and named pipes
  - Sockets

Remark

These entities are accessed through the common file API

Filenames

- Files have names of up to 255 characters length, all ASCII characters except NUL (`0x0`) are allowed
  - Note: Entering filenames with non-printable characters in a shell is a completely different matter . . .
- Unices do not enforce any further structure in filenames
- Many applications follow the convention that filenames consist of a basename and an extension, for example

  `program.c`

  is a C program
- However, nothing stops you from having filenames like

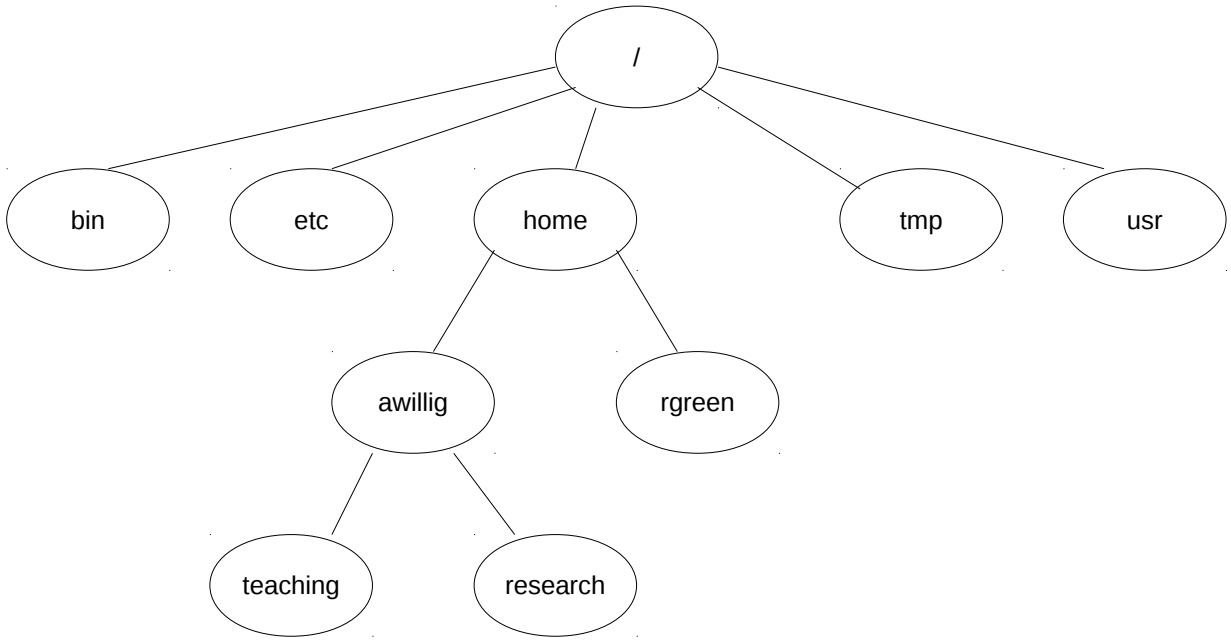
  `program.tar.gz`

  or

  `pro?*m.tar.gz`

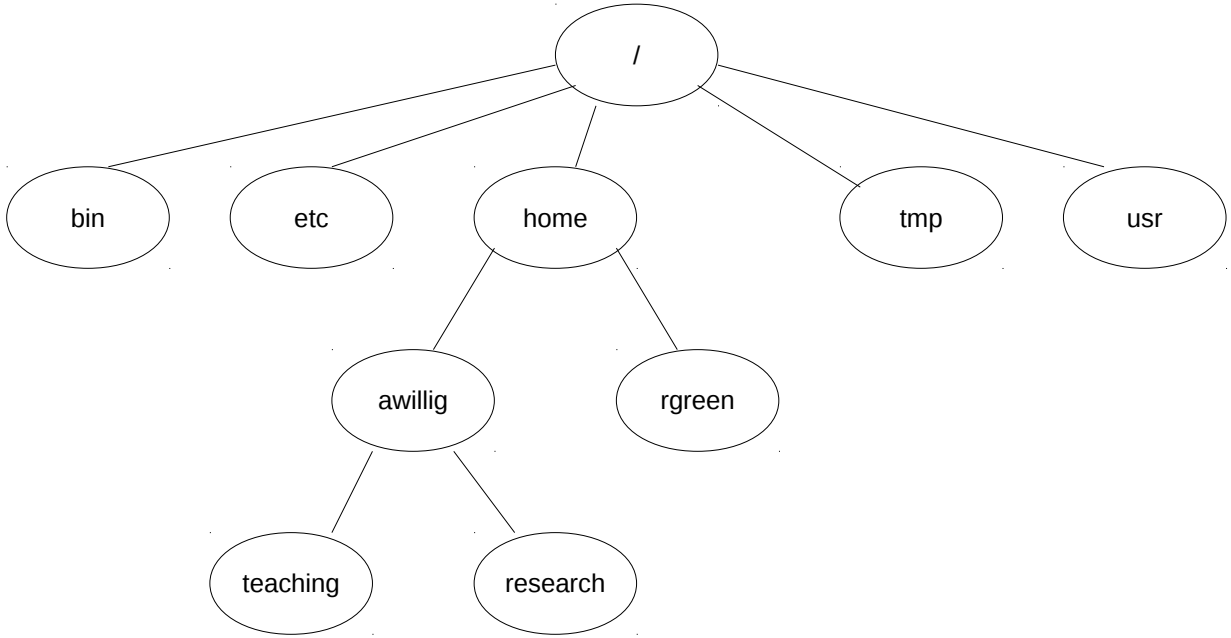
  To enter the last into the shell, you need **escaping**

Directories and File Hierarchy



- A directory contains a list of files and their attributes
- Some of these files are themselves directories
- Within a directory all files must have different names
- This allows to create a hierarchical structure or **directory tree**

Directories and File Hierarchy (2)



- The root directory is denoted as '/'
- A **full filename** or **pathname** for a file in a directory lists all parent directories in "descending" order, starting from `/`, and separated by `/` characters
- Example: file `slideset-unices.tex` in directory `teaching` has pathname `/home/awillig/teaching/slideset-unices.tex`

Absolute and Relative Pathnames

- An **absolute pathname** of a file lists all parent directories, like in the example

  `/home/awillig/teaching/slideset-unices.tex`

- As an alternative, users can specify one particular directory as their **current directory** or **working directory** and can express filenames relative to the working directory
- Example: suppose the working directory is `/home/awillig`, then the same file can be referred to as

  `teaching/slideset-unices.tex`

  which is an example of a **relative pathname**
- Note that relative paths do not start with '/'
- There are two special files in each directory:
  - '.' refers to this directory itself
  - '..' refers to the parent directory in the tree

Absolute and Relative Pathnames (2)

- Examples:
  - suppose our working directory is `/home/awillig`
  - Then

    ```
    $ cp teaching/slideset-unices.tex ../rgreen/
    ```

    copies the file `teaching/slideset-unices.tex` into the home directory of another user (provided i have the rights to write there)
  - And

    ```
    $ cp teaching/slideset-unices.tex .
    ```

    copies the file `teaching/slideset-unices.tex` into my own home directory
  - Finally

    ```
    $ cp teaching/slideset-unices.tex ./unix-slides.tex
    ```

    copies the file `teaching/slideset-unices.tex` into my own home directory, but giving it another name there
  - Same thing, but shorter:

    ```
    $ cp teaching/slideset-unices.tex unix-slides.tex
    ```

The Filesystem Hierarchy Standard

- FHS defines important directories and contents for Linux
- Maintained by Linux foundation, see
  `http://www.linuxfoundation.org/collaborate/workgroups/lsb/fhs`
- Last version approved in 2004
- Major Linux distributions follow it (more or less)

The Filesystem Hierarchy Standard (2)

`/`	root directory
`/bin`	Essential binaries needed in single-user mode
`/boot`	Files needed by boot loader
`/dev`	represents all physical and logical devices as files
`/etc`	static configuration files for various system services
`/etc/X11`	configuration files for X11 window system
`/home`	home directories of users
`/lib`	dynamic libraries needed by binaries in `/bin` and `/sbin`
`/media`	mount point for removable media
`/mnt`	mount point for temporarily mounted file systems
`/opt`	optional software and data
`/root`	superusers home directory
`/sbin`	essential system binaries
`/tmp`	temporary files, usually not backed up or cleaned periodically

The Filesystem Hierarchy Standard (3)

`/usr`	secondary FS hierarchy, contains most applications and support services, none of this is required for booting
`/usr/bin`	Binaries
`/usr/include`	Standard include files (C header files)
`/usr/lib`	dynamic and static libraries
`/usr/share`	data files (e.g. documentation, fonts)
`/usr/src`	source code (e.g. kernel sources)
`/usr/X11Rx`	X Windows binaries (nowadays often elsewhere)
`/usr/local`	Root of a tertiary host-specific hierarchy, includes own `bin`, `include`, etc. sub-directories
`/var`	variable files (logfiles, spool files, etc.)
`/var/cache`	cache area for applications (e.g. information about installable software packages)
`/var/log`	logfiles
`/var/mail`	mailboxes of local users
`/var/spool`	Data to be processed: print queues, unprocessed mail, etc.

Home, Sweet Home

- Each user has a separate directory, its **home directory**
- Shells abbreviate this by '~'
- It is a private space for the user, allowing him to store files and create sub-directories
  - As a convention, the owner has all rights in its home directory, other users have no rights or only reading rights – but this can be changed by modifying the file attributes of the home directory (see also `umask` shell command)!
- It also usually contains user-specific configuration data for individual programs, for example:
  - `~/.bashrc` allows user-specific `bash` customisations
  - `~/.emacs` allows user-specific configurations for `emacs`
- Sidenote: filenames starting with '.' (**dotfiles**) are usually not displayed by `ls`, but this is only convention and can be changed with certain options

Important Commands for Files and Directories

`cp f1 f2`	copies file `f1` to file `f2`
`mv f1 f2`	moves file `f1` to file `f2`
`rm f`	removes file `f`
`rmdir dir`	removes directory `dir`, which must be completely empty
`ls`	lists all files in current directory
`ls f`	lists file `f` – when `f` is a directory, its contents is listed, otherwise file attributes are listed
`cd`	changes working directory to home directory
`cd dir`	changes working directory to directory `dir`
`pwd`	prints the current working directory
`mkdir f`	creates a new sub-directory `f`
`diff f1 f2`	compares two files and shows their differences

Hard and Symbolic Links

- A link allows a file in one directory *A* to appear in another directory as well *B*
  - Note that this does **not** refer to a copy but to a real link, so any changes made to the file in directory *B* will become visible in directory *A* as well
  - As a result, the file system is not tree-structured anymore but can become a directed acyclic graph
- Hard links:
  - A file entry in a directory refers to a data structure on disk, an **inode**, which represents the actual file
  - A hard link to a file `f` is created by letting another filename point to the same inode as `f` does
- Soft or symbolic links:
  - Links are a special type of file
  - A link file contains the pathname of the file that the link refers to
- In Unices links are created by the command `ln` and removed with `rm`

Mounting

- Several storage devices can be attached, e.g.:
  - hard disks
  - CD's
  - USB sticks
- Each device (or partition thereof) is formatted with a specific **file system**, e.g.:
  - ext2, ext3, ext4
  - btrfs
  - JFS, XFS
  - NFS
  - ISO 9660
- One device carries the **root filesystem**, it supplies:
  - `/`, i.e. the root directory
  - `/boot`, the bootable kernel
  - `/sbin`, `/bin` with essential binaries

Mounting (2)

- The filesystem of other devices can be added to the root filesystem by **mounting**:
  - A **mount point** is an empty directory in the root filesystem, e.g. `/mnt`
  - After mounting to `/mnt`, the filesystem of a new device is located under `/mnt`, i.e. a file `x.x` in the new devices root directory can now be found under `/mnt/x.x`
- Linux offers the `mount` command, which attaches a device to a mount point, e.g.:

  ```
  # mount /dev/sda2 /home
  ```

  mounts a SCSI hard disk represented by device file `/dev/sda2` to mount point `/home`
- When called without parameters, `mount` displays currently mounted filesystems and their mount points
- The file `/etc/fstab` (see also `man fstab`) specifies mount options (e.g. mount points, read/write/read-write, etc.) for various filesystems

[1] Daniel P. Bovet and Marco Cesati.
Understanding the Linux Kernel.
O'Reilly, Sebastopol, California, third edition, 2006.

[2] Andrew S. Tanenbaum.
Modern Operating Systems.
Pearson / Prentice Hall, Upper Saddle River, New Jersey, third edition, 2008.