Introduction to Unix/Linux

Operating Systems UE 2022W

David Lung, Florian Mihola, Andreas Brandstätter, Axel Brunnbauer, Peter Puschner

Technische Universität Wien
Computer Engineering
Cyber-Physical Systems

2022-10-04

Motivation

- What is an Operating System?
- UNIX, Linux, . . . ?
- Why C?

What is an Operating System?

The operating system as . . .

1. An extended machine
   - Provide simpler and easier to use abstractions of the underlying hardware
   - Provide services that programs can obtain by a special interface

What is an Operating System?

The operating system as ...

1. An extended machine
   - Provide simpler and easier to use abstractions of the underlying hardware
   - Provide services that programs can obtain by a special interface
2. A resource manager
   - Multiplexing/sharing resources in time and in space
   - Create the illusion that a program has exclusive access to the resources

What is an Operating System?

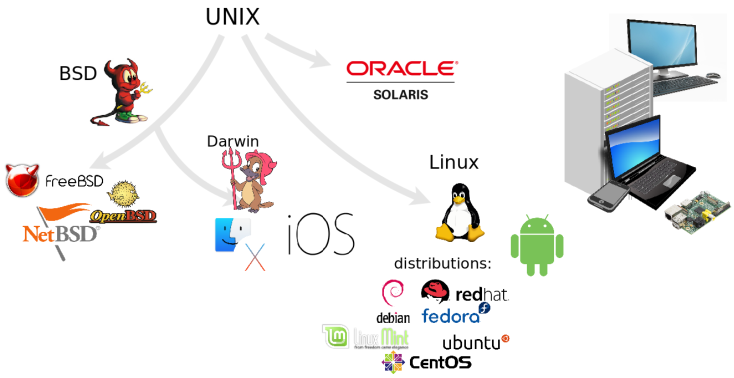
The operating system as . . .

1. An extended machine
   - Provide simpler and easier to use abstractions of the underlying hardware
   - Provide services that programs can obtain by a special interface
2. A resource manager
   - Multiplexing/sharing resources in time and in space
   - Create the illusion that a program has exclusive access to the resources

Important mechanisms: Processes, virtual memory, file system, . . .

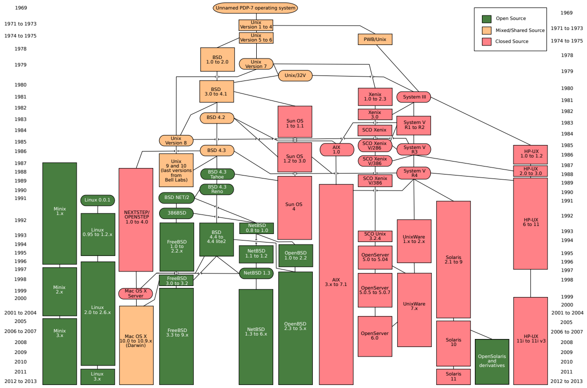
Unix

Unix-family OS everywhere



Unix

Evolution of Unix and Unix-like systems



Source: Wikipedia

The C Programming Language

Why?

The C Programming Language

Why?

- “Java [Python, Ruby, . . .] is much more powerful and high-level.”

The C Programming Language

Why?

- "Java [Python, Ruby, ...] is much more powerful and high-level."
  - Actually, most high-level languages and interpreters are implemented in C

The C Programming Language

Why?

- “Java [Python, Ruby, ...] is much more powerful and high-level.”
  - Actually, most high-level languages and interpreters are implemented in C
- More powerful - Close to hardware, explicit memory and resource management
  - Full control of what’s going on

The C Programming Language

Why?

- "Java [Python, Ruby, . . .] is much more powerful and high-level."
  - Actually, most high-level languages and interpreters are implemented in C
- More powerful - Close to hardware, explicit memory and resource management
  - Full control of what's going on
- Constructs that map efficiently to machine instructions
  - Compiled to fast and efficient code
  - First compiler for a new architecture is typically a C compiler

The C Programming Language

Why?

- "Java [Python, Ruby, ...] is much more powerful and high-level."
  - Actually, most high-level languages and interpreters are implemented in C
- More powerful - Close to hardware, explicit memory and resource management
  - Full control of what's going on
- Constructs that map efficiently to machine instructions
  - Compiled to fast and efficient code
  - First compiler for a new architecture is typically a C compiler
- Arbitrary memory address access and pointer arithmetic
  - Perfect fit for systems programming

The C Programming Language

Why?

- "Java [Python, Ruby, ...] is much more powerful and high-level."
  - Actually, most high-level languages and interpreters are implemented in C
- More powerful - Close to hardware, explicit memory and resource management
  - Full control of what's going on
- Constructs that map efficiently to machine instructions
  - Compiled to fast and efficient code
  - First compiler for a new architecture is typically a C compiler
- Arbitrary memory address access and pointer arithmetic
  - Perfect fit for systems programming
- Operating system kernels are mostly written in C

The C Programming Language

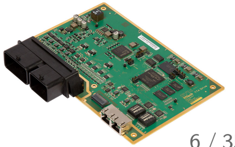
Why?

- "Java [Python, Ruby, . . .] is much more powerful and high-level."
  - Actually, most high-level languages and interpreters are implemented in C
- More powerful - Close to hardware, explicit memory and resource management
  - Full control of what's going on
- Constructs that map efficiently to machine instructions
  - Compiled to fast and efficient code
  - First compiler for a new architecture is typically a C compiler
- Arbitrary memory address access and pointer arithmetic
  - Perfect fit for systems programming
- Operating system kernels are mostly written in C
- Embedded systems are mostly programmed in C

The C Programming Language

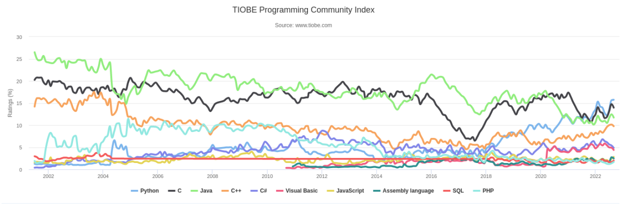
Why?

- "Java [Python, Ruby, . . .] is much more powerful and high-level."
  - Actually, most high-level languages and interpreters are implemented in C
- More powerful - Close to hardware, explicit memory and resource management
  - Full control of what's going on
- Constructs that map efficiently to machine instructions
  - Compiled to fast and efficient code
  - First compiler for a new architecture is typically a C compiler
- Arbitrary memory address access and pointer arithmetic
  - Perfect fit for systems programming
- Operating system kernels are mostly wri
- Embedded systems are mostly programm

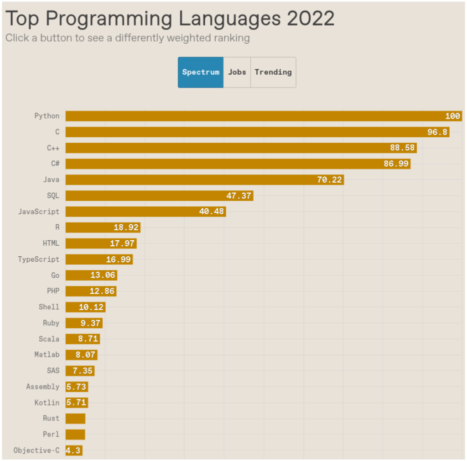


Relevance of C I

- Appeared 1972 when UNIX was ported to C
- Has not lost popularity and importance!
- Tools (compiler, debugger, profiler, . . .) improved over time



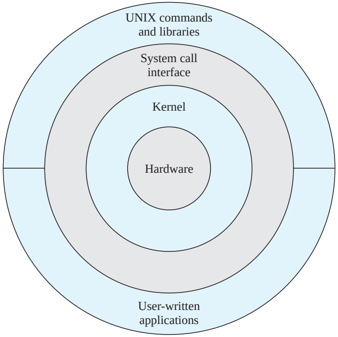
Relevance of C II



Source: IEEE Spectrum Rating of programming languages, 2022
`https://spectrum.ieee.org/top-programming-languages-2022`

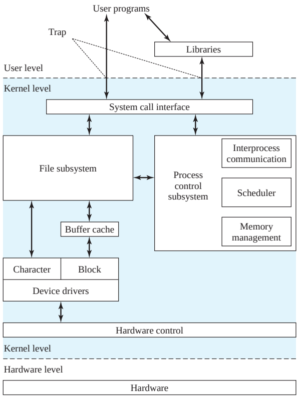
Unix Anatomy

- User space – Kernel space
  - Kernel routines run in privileged mode (*kernel mode*), includes device drivers
  - User processes request kernel services with *system calls*
- Multi-process and multi-user operating system
  - Run more than one program concurrently
  - Users share resources
- Requires authentication (login)

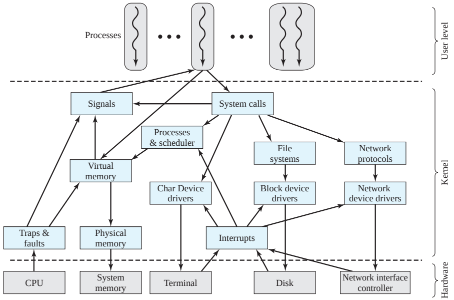


General Unix Architecture (Source: W. Stallings, “Operating Systems. Internals and Design Principles”)

Unix Anatomy



Traditional UNIX Kernel



Linux Kernel Components

(Source: W. Stallings, "Operating Systems. Internals and Design Principles")

The Shell

- In the simplest case, a shell is started after login

The Shell

- In the simplest case, a shell is started after login
- A user program that
  - Reads and interprets user input interactively (commands)
  - Starts other user programs
  - Executes *shell-scripts*

The Shell

- In the simplest case, a shell is started after login
- A user program that
  - Reads and interprets user input interactively (commands)
  - Starts other user programs
  - Executes *shell-scripts*
- Shell prompt:

```
jdoe@ti1:~$ _
```

Unix Philosophy

Doug McIlroy, 1978: (summarized)

- Write programs that do one thing and do it well. (DOTADIW)

Unix Philosophy

Doug McIlroy, 1978: (summarized)

- Write programs that do one thing and do it well. (DOTADIW)
- Write programs to work together.

Unix Philosophy

Doug McIlroy, 1978: (summarized)

- Write programs that do one thing and do it well. (DOTADIW)
- Write programs to work together.
- Write programs to handle text streams, because that is a universal interface.

Unix Philosophy

Doug McIlroy, 1978: (summarized)

- Write programs that do one thing and do it well. (DOTADIW)
- Write programs to work together.
- Write programs to handle text streams, because that is a universal interface.

= combining small, sharp tools and the use of a common underlying format (the line-oriented, plain text file) to accomplish larger tasks

Executing programs

```
$ echo Hi there
Hi there
```

Executing programs

```
$ echo Hi there
Hi there
```

```
$ date
Tue Oct  6 11:15:00 CEST 2020
```

Executing programs

```
$ echo Hi there
Hi there
```

```
$ date
Tue Oct  6 11:15:00 CEST 2020
```

```
$ date --iso-8601
2020-10-06
```

Executing programs

```
$ echo Hi there
Hi there
```

```
$ date
Tue Oct  6 11:15:00 CEST 2020
```

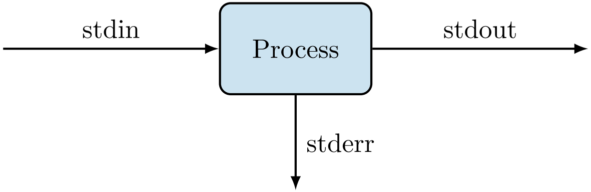
```
$ date --iso-8601
2020-10-06
```

```
$ rev
Hello class
ssalc olleH
```

<Ctrl-D> (EOF token)

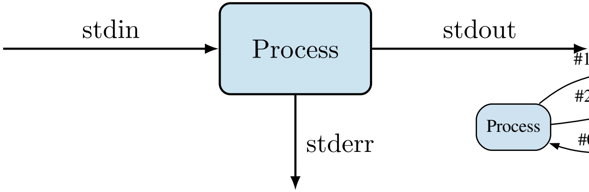
Standard Input/Output

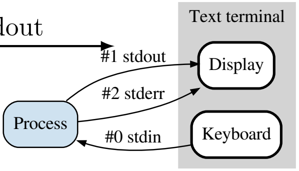
- A process communicates with its environment by following channels:
  - Standard input (stdin, 0), redirect with <
  - Standard output (stdout, 1), redirect with >
  - Standard error (stderr, 2), redirect with 2>



Standard Input/Output

- A process communicates with its environment by following channels:
  - Standard input (stdin, 0), redirect with <
  - Standard output (stdout, 1), redirect with >
  - Standard error (stderr, 2), redirect with 2>
- When started in a shell, the standard I/O is connected to the terminal.





Redirection and Pipes

Redirection to/from files

```
$ echo "Hello class" > somefile
$ cat somefile
Hello class
$ rev < somefile
ssalc olleH
```

Redirection and Pipes

Redirection to/from files

```
$ echo "Hello class" > somefile
$ cat somefile
Hello class
$ rev < somefile
ssalc olleH
```

```
$ echo "More text." >> somefile
$ cat somefile
Hello class
More text.
$ nl somefile
     1  Hello class
     2  More text.
```

Redirection and Pipes (ctd.)

Pipes connect processes with a unidirectional FIFO

```
$ cat somefile | rev | nl
     1  ssalc olleH
     2  .txet eroM
```

In these examples, redirection and pipes are set up by the shell.

Processes

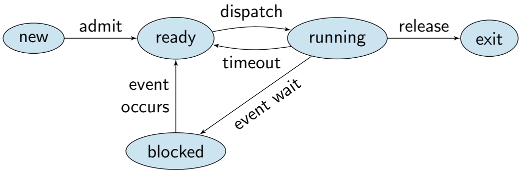
- The execution of a program is a process.

Processes

- The execution of a program is a process.
  - + program code
  - + program data (variables, . . .)
  - + context (state, program counter, processor registers, . . .)

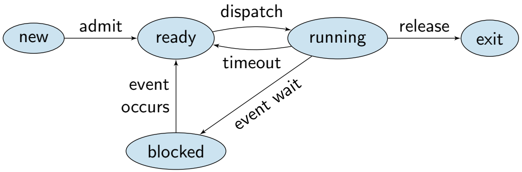
Processes

- The execution of a program is a process.
  - + program code
  - + program data (variables, ...)
  - + context (state, program counter, processor registers, ...)
- A Unix system executes many processes concurrently. Process states:



Processes

- The execution of a program is a process.
  - + program code
  - + program data (variables, . . .)
  - + context (state, program counter, processor registers, . . .)
- A Unix system executes many processes concurrently.
  Process states:



  - `ps` – snapshot of current processes
  - `pstree` – display process hierarchy

Process Management

- Programs executed on the shell are child processes of the shell.

Process Management

- Programs executed on the shell are child processes of the shell.
  - $<$**Ctrl+Z** $>$ stops currently active job
  - `jobs` – status of processes started in the current shell
  - `fg` *n* – continue job *n* in foreground
  - `bg` *n* – continue job *n* in background
  - '`&`' at the end of a command starts it in the background

Process Management

- Programs executed on the shell are child processes of the shell.
  - <**Ctrl+Z** > stops currently active job
  - `jobs` – status of processes started in the current shell
  - `fg` *n* – continue job *n* in foreground
  - `bg` *n* – continue job *n* in background
  - '`&`' at the end of a command starts it in the background

- Multiple commands:

Command sequence	Resulting behaviour
`cmd1 ; cmd2`	Execute commands subsequently
`cmd1 && cmd2`	Execute `cmd2` only if `cmd1` succeeds
`cmd1 || cmd2`	Execute `cmd2` only if `cmd1` fails
`cmd1 & cmd2`	Start `cmd1` in background and `cmd2` in foreground
`(cmd1 ; cmd2)`	Execute both commands in a subshell

Process Management

- Programs executed on the shell are child processes of the shell.
  - <**Ctrl+Z** > stops currently active job
  - `jobs` – status of processes started in the current shell
  - `fg` *n* – continue job *n* in foreground
  - `bg` *n* – continue job *n* in background
  - '`&`' at the end of a command starts it in the background
- Multiple commands:

Command sequence	Resulting behaviour
`cmd1 ; cmd2`	Execute commands subsequently
`cmd1 && cmd2`	Execute `cmd2` only if `cmd1` succeeds
`cmd1 || cmd2`	Execute `cmd2` only if `cmd1` fails
`cmd1 & cmd2`	Start `cmd1` in background and `cmd2` in foreground
`(cmd1 ; cmd2)`	Execute both commands in a subshell

- Example:

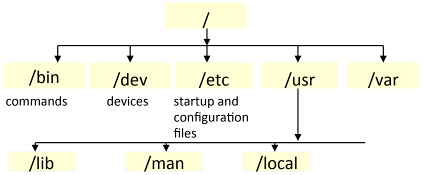
```
$ (sleep 10; date) > outfile &
```

Filesystem Organisation

- Hierarchical structure of files
- Wide range of input/output resources are simple streams of bytes exposed through the filesystem name space $\rightarrow$ "everything is a file"
  - Documents
  - Directories
  - Character-, block special files (devices; e.g. hard-drives, keyboards, printers)
  - Named pipes
  - Sockets (e.g. TCP/IP sockets, UNIX domain sockets)
  - Symbolic links

Filesystem Organisation

- Hierarchical structure of files
- Wide range of input/output resources are simple streams of bytes exposed through the filesystem name space → "everything is a file"
  - Documents
  - Directories
  - Character-, block special files (devices; e.g. hard-drives, keyboards, printers)
  - Named pipes
  - Sockets (e.g. TCP/IP sockets, UNIX domain sockets)
  - Symbolic links



Filesystem Hierarchy Standard[1] I

- / Primary hierarchy (root directory)
  - `/bin`: Essential command binaries (for all users)
  - `/etc`: Configuration files
  - `/dev`: Devices
  - `/lib`: Libraries essential for the binaries in `/bin` and `/sbin`
  - `/home`: Users' home directories
  - `/media`: Mount points for removable media
  - `/mnt`: Temporarily mounted file systems
  - `/opt`: Optional application software packages
  - `/proc`: Virtual filesystem providing process and kernel information as files
  - `/sbin`: Essential system binaries

Filesystem Hierarchy Standard[2] II

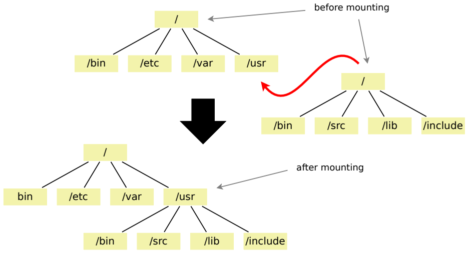
- `/usr`: Secondary hierarchy for shareable, read-only data (contains the majority of multi-user utilities and applications)
- `/usr/local`: Tertiary hierarchy for local data, specific to the host
- `/var`: Variable files, whose content is expected to continually change during normal operation of the system (log files, spools, temporary e-mails)

[1] `http://www.pathname.com/fhs/`

[2] `http://www.pathname.com/fhs/`

Mounting File Systems

- All files and directories appear under the root directory, even if they are stored on different physical or virtual devices
- File system to be mounted is either:
  - locally available (hard-drive partitions, removable media)
  - a network resource (e.g. using NFS),
  - or contained in a file itself (e.g. loop device for ISO-Images)
- Advantage: different file systems concurrently in use



Navigating through the File System

- cd – change directory

```
$ cd
$ pwd
/home/jdoe
$ mkdir test
$ cd test
$ pwd
/home/jdoe/test
$ echo "Hello class" > textfile
$ ls -l
-rw-rw-r--  1 jdoe jdoe   12 Oct  6 11:15 textfile
$ cd .. ; pwd
/home/jdoe
```

Navigating through the File System

- cd – change directory

```
$ cd
$ pwd
/home/jdoe
$ mkdir test
$ cd test
$ pwd
/home/jdoe/test
$ echo "Hello class" > textfile
$ ls -l
-rw-rw-r--  1 jdoe jdoe   12 Oct  6 11:15 textfile
$ cd .. ; pwd
/home/jdoe
```

- Filenames
  - **absolute**: start with '/', from the root directory
    e.g. `$ cat /etc/passwd`
  - **relative**: do not start with '/', and are related to the current directory
    e.g. `$ cat ../tmpfile`
    (`.` is the current directory, `..` the parent directory)

Navigating through the File System

- cd – change directory

```
$ cd
$ pwd
/home/jdoe
$ mkdir test
$ cd test
$ pwd
/home/jdoe/test
$ echo "Hello class" > textfile
$ ls -l
-rw-rw-r--  1 jdoe jdoe   12 Oct  6 11:15 textfile
$ cd .. ; pwd
/home/jdoe
```

- Filenames
  - **absolute**: start with '/', from the root directory
    e.g. `$ cat /etc/passwd`
  - **relative**: do not start with '/', and are related to the current directory
    e.g. `$ cat ../tmpfile`
    (`.` is the current directory, `..` the parent directory)
- *Note:* use <**TAB**> for shell completion

Wildcards

- Pattern matching for filename specification

`*`	zero or more characters
`?`	a single character
`[xyz]`	one of ‘x’, ‘y’ or ‘z’
`[a-i]`	one in the range from ‘a’ to ‘i’

- Interpretation and expansion by the shell
- The operation of matching of wildcard patterns to multiple file or path names is referred to as globbing.
- Use wildcards as normal characters by quoting or a preceding backslash (\)

Wildcards

Examples

```
$ ls
myfile  prog  prog.c  proG.c  t1  t2
t3      t4    test1   test1.c
```

```
*
```

Wildcards

Examples

```
$ ls
myfile   prog   prog.c   proG.c   t1   t2
t3       t4     test1    test1.c
```

`*` *(all files listed above)*

Wildcards

Examples

```
$ ls
myfile   prog   prog.c   proG.c   t1   t2
t3       t4     test1    test1.c
```

`*` *(all files listed above)*

`t*`

Wildcards

Examples

```
$ ls
myfile   prog   prog.c   proG.c   t1   t2
t3       t4     test1    test1.c
```

`*`	*(all files listed above)*
`t*`	t1 t2 t3 t4 test test.c

Wildcards

Examples

```
$ ls
myfile  prog  prog.c  proG.c   t1  t2
t3      t4    test1   test1.c
```

`*`	*(all files listed above)*
`t*`	t1 t2 t3 t4 test test.c
`t?`	

Wildcards

Examples

```
$ ls
myfile   prog   prog.c   proG.c   t1   t2
t3       t4     test1    test1.c
```

`*`	*(all files listed above)*
`t*`	t1 t2 t3 t4 test test.c
`t?`	

Wildcards

Examples

```
$ ls
myfile   prog   prog.c   proG.c   t1   t2
t3       t4     test1    test1.c
```

`*`	*(all files listed above)*
`t*`	t1 t2 t3 t4 test test.c
`t?`	t1 t2 t3 t4

Wildcards

Examples

```
$ ls
myfile   prog   prog.c   proG.c   t1   t2
t3       t4     test1    test1.c
```

`*`	*(all files listed above)*
`t*`	t1 t2 t3 t4 test test.c
`t?`	t1 t2 t3 t4
`t[12]`	

Wildcards

Examples

```
$ ls
myfile   prog   prog.c   proG.c   t1   t2
t3       t4     test1    test1.c
```

`*`	*(all files listed above)*
`t*`	t1 t2 t3 t4 test test.c
`t?`	t1 t2 t3 t4
`t[12]`	t1 t2

Wildcards

Examples

```
$ ls
myfile   prog   prog.c   proG.c    t1   t2
t3       t4     test1    test1.c
```

`*`	*(all files listed above)*
`t*`	t1 t2 t3 t4 test test.c
`t?`	t1 t2 t3 t4
`t[12]`	t1 t2
`pr*.c`	

Wildcards

Examples

```
$ ls
myfile   prog   prog.c   proG.c    t1   t2
t3       t4     test1    test1.c
```

`*`	*(all files listed above)*
`t*`	t1 t2 t3 t4 test test.c
`t?`	t1 t2 t3 t4
`t[12]`	t1 t2
`pr*.c`	proG.c prog.c

Wildcards

Examples

```
$ ls
myfile  prog  prog.c  proG.c  t1  t2
t3      t4    test1   test1.c
```

`*`	*(all files listed above)*
`t*`	t1 t2 t3 t4 test test.c
`t?`	t1 t2 t3 t4
`t[12]`	t1 t2
`pr*.c`	proG.c prog.c
`*[1-4].c`	

Wildcards

Examples

```
$ ls
myfile   prog   prog.c   proG.c    t1   t2
t3       t4     test1    test1.c
```

`*`	*(all files listed above)*
`t*`	t1 t2 t3 t4 test test.c
`t?`	t1 t2 t3 t4
`t[12]`	t1 t2
`pr*.c`	proG.c prog.c
`*[1-4].c`	test1.c

Permissions

`-rw-rw-r--` ???

Permissions

- Access permissions for each individual file (as file attribute) $\underbrace{\texttt{-}}_{\text{special}}\ \underbrace{\overset{4}{\texttt{r}}\overset{2}{\texttt{w}}\overset{1}{\texttt{x}}}_{\text{user}}\ \underbrace{\overset{4}{\texttt{r}}\overset{2}{\texttt{w}}\overset{1}{\texttt{x}}}_{\text{group}}\ \underbrace{\overset{4}{\texttt{r}}\overset{2}{\texttt{w}}\overset{1}{\texttt{x}}}_{\text{others}}$

Permissions

- Access permissions for each individual file (as file attribute) $\underbrace{\texttt{-}}_{\text{special}}\ \underbrace{\overset{4\,2\,1}{\texttt{rwx}}}_{\text{user}}\ \underbrace{\overset{4\,2\,1}{\texttt{rwx}}}_{\text{group}}\ \underbrace{\overset{4\,2\,1}{\texttt{rwx}}}_{\text{others}}$
- First character to indicate normal (-) or special file: directory (d), socket (s), symbolic link (l), pipe (p), character special device (c), block special device (b)

Permissions

`-rw-rw-r--` ???

- Access permissions for each individual file (as file attribute) $\underbrace{\texttt{-}}_{\text{special}}\ \underbrace{\overset{4\,2\,1}{\texttt{rwx}}}_{\text{user}}\ \underbrace{\overset{4\,2\,1}{\texttt{rwx}}}_{\text{group}}\ \underbrace{\overset{4\,2\,1}{\texttt{rwx}}}_{\text{others}}$
- First character to indicate normal (-) or special file: directory (d), socket (s), symbolic link (l), pipe (p), character special device (c), block special device (b)
- Permission to <u>r</u>ead, <u>w</u>rite, and e<u>x</u>ecute for user/group/others

Permissions

`-rw-rw-r--` ???

- Access permissions for each individual file (as file attribute) $\underbrace{\texttt{-}}_{\text{special}}\underbrace{\overset{4\,2\,1}{\texttt{rwx}}}_{\text{user}}\underbrace{\overset{4\,2\,1}{\texttt{rwx}}}_{\text{group}}\underbrace{\overset{4\,2\,1}{\texttt{rwx}}}_{\text{others}}$
- First character to indicate normal (-) or special file: directory (d), socket (s), symbolic link (l), pipe (p), character special device (c), block special device (b)
- Permission to read, write, and execute for user/group/others
- `chmod` – change file mode bits (= permissions)

Permissions

`-rw-rw-r--` ???

- Access permissions for each individual file (as file attribute) $\underbrace{\texttt{-}}_{\text{special}}\ \underbrace{\overset{4\,2\,1}{\texttt{rwx}}}_{\text{user}}\ \underbrace{\overset{4\,2\,1}{\texttt{rwx}}}_{\text{group}}\ \underbrace{\overset{4\,2\,1}{\texttt{rwx}}}_{\text{others}}$
- First character to indicate normal (-) or special file: directory (d), socket (s), symbolic link (l), pipe (p), character special device (c), block special device (b)
- Permission to <u>r</u>ead, <u>w</u>rite, and e<u>x</u>ecute for user/group/others
- `chmod` – change file mode bits (= permissions)
  - with octal representation, e.g. `chmod 755 textfile`

Permissions

`-rw-rw-r--` ???

- Access permissions for each individual file (as file attribute) $\underbrace{\texttt{-}}_{\text{special}}\ \underbrace{\overset{4\,2\,1}{\texttt{rwx}}}_{\text{user}}\ \underbrace{\overset{4\,2\,1}{\texttt{rwx}}}_{\text{group}}\ \underbrace{\overset{4\,2\,1}{\texttt{rwx}}}_{\text{others}}$
- First character to indicate normal (-) or special file: directory (d), socket (s), symbolic link (l), pipe (p), character special device (c), block special device (b)
- Permission to <u>r</u>ead, <u>w</u>rite, and e<u>x</u>ecute for user/group/others
- `chmod` – change file mode bits (= permissions)
  - with octal representation, e.g. `chmod 755 textfile`
  - textual specification, e.g. `chmod ugo+x,g-w textfile`, `chmod u=rwx,go=rx textfile`

Permissions

`-rw-rw-r--` ???

- Access permissions for each individual file (as file attribute) $\underbrace{\texttt{-}}_{\text{special}}\underbrace{\overset{4\,2\,1}{\texttt{rwx}}}_{\text{user}}\underbrace{\overset{4\,2\,1}{\texttt{rwx}}}_{\text{group}}\underbrace{\overset{4\,2\,1}{\texttt{rwx}}}_{\text{others}}$
- First character to indicate normal (-) or special file: directory (d), socket (s), symbolic link (l), pipe (p), character special device (c), block special device (b)
- Permission to read, write, and execute for user/group/others
- `chmod` – change file mode bits (= permissions)
  - with octal representation, e.g. `chmod 755 textfile`
  - textual specification, e.g. `chmod ugo+x,g-w textfile`, `chmod u=rwx,go=rx textfile`
- Only user (owner) or root can change permissions

Shell Variables

- Only string type

Shell Variables

- Only string type
- Created at first assignment

```
$ FILE=/tmp/dummy.txt
```

Shell Variables

- Only string type
- Created at first assignment

```
$ FILE=/tmp/dummy.txt
```

- Usage:

```
$ ls /tmp
dummy.txt dummy.txt.bak
$ rm $FILE ${FILE}.bak
```

Shell Variables

- Only string type
- Created at first assignment

```
$ FILE=/tmp/dummy.txt
```

- Usage:

```
$ ls /tmp
dummy.txt dummy.txt.bak
$ rm $FILE ${FILE}.bak
```

- Export to environment for subsequently started processes:

```
$ export FILE
```

at assignment

```
$ export FILE=/tmp/dummy.txt
```

System Variables

- `$HOME` . . . Home directory

System Variables

- `$HOME` . . . Home directory
- `$USER` . . . User name

System Variables

- `$HOME` ... Home directory
- `$USER` ... User name
- `$?` ... Exit status of the last command

System Variables

- `$HOME` ... Home directory
- `$USER` ... User name
- `$?` ... Exit status of the last command
- `$PATH` ... Program path

```
$ echo $PATH
/usr/local/bin:/usr/bin:/usr/local/sbin:
/usr/sbin
```

→ If you create a program in a local directory and want to execute it:

```
$ ./myprogram
```

System Variables

- `$HOME` … Home directory
- `$USER` … User name
- `$?` … Exit status of the last command
- `$PATH` … Program path

```
$ echo $PATH
/usr/local/bin:/usr/bin:/usr/local/sbin:
/usr/sbin
```

$\rightarrow$ If you create a program in a local directory and want to execute it:

```
$ ./myprogram
```

- Print environment variables with `env`

Unix Commands I

Examples

- File management

ls	list directory contents
cd	change the working directory
pwd	print filename of working directory
cp, mv	move (rename) files
ln	make links between files
mkdir	make directories
rm, rmdir	remove files and directories
chmod, chown	change file mode bits, owner
du	estimate file space usage
file	determine file type

- Process management

jobs	display status of jobs in current shell session
fg, bg	run job in foreground/background
ps, pstree	snapshot of current processes/process hierarchy
kill	send a signal to a process

Unix Commands II

Examples

- Text processing

cat	concatenate files to standard output
sort	sort lines of text files
nl	number lines of files
wc	print line, word, and byte counts
cut	remove sections from each line
tr	translate or delete characters
tac	contatenate and print files in reverse
rev	reverse lines of a file
grep	print lines matching a pattern
sed	stream editor for filtering and transforming text

Unix Commands III

Examples

- Utilities

echo	print arguments to stdout
more, less	pager
date, cal	print current time and time/calendar
tar	archiving utility
make	build utility
ssh	SSH client (remote login program)
gcc	GNU compiler collection C compiler

- Editors vim, emacs
- . . . and many many more

- see `$ man` *`command`* for more information

Interprocess communication

- How can processes interact?

Interprocess communication

- How can processes interact?
  - stream of data (pipes, stream sockets)

Interprocess communication

- How can processes interact?
  - stream of data (pipes, stream sockets)
  - sending messages (message queues, datagram sockets)

Interprocess communication

- How can processes interact?
  - stream of data (pipes, stream sockets)
  - sending messages (message queues, datagram sockets)
  - accessing a shared resource (file, memory)

Interprocess communication

- How can processes interact?
  - stream of data (pipes, stream sockets)
  - sending messages (message queues, datagram sockets)
  - accessing a shared resource (file, memory)
- Classification

Interprocess communication

- How can processes interact?
  - stream of data (pipes, stream sockets)
  - sending messages (message queues, datagram sockets)
  - accessing a shared resource (file, memory)
- Classification
  - related vs. unrelated processes
    unrelated processes require named resources (system-wide namespace)

Interprocess communication

- How can processes interact?
  - stream of data (pipes, stream sockets)
  - sending messages (message queues, datagram sockets)
  - accessing a shared resource (file, memory)
- Classification
  - related vs. unrelated processes
    unrelated processes require named resources (system-wide namespace)
  - implicit vs. explicit synchronization
    ensure orderly execution and access to a shared resource

Outlook

What you will learn in this course

We assume...

You already know how to program in an imperative programming language.
This is not an introduction to programming!

Outlook

What you will learn in this course

We assume. . .

You already know how to program in an imperative programming language.
This is not an introduction to programming!

Educational objectives of the programming assignments

- How to write and compile a C program, use options and arguments, basic stream I/O (1a); communicate through stream sockets (1b)
- How to create child processes, communicate through unnamed pipes (2)
- Collaboration of unrelated processes through shared memory and synchronization with semaphores (3)

Summary

- C is still a highly relevant language
- Unix-based OS are ubiquitous
- Introduction to basic Unix concepts and the environment

Material

- Advanced Bash-Scripting Guide
  `http://www.tldp.org/LDP/abs/html/`

Homework

Work through slides “Introduction to C”.
The next lecture will deal with the features specific to C, so you should be familiar with the elements of the C language.