

GNU/Linux Application Programming

Lecture Handouts

WANG Xiaolin

wx672ster+linux@gmail.com

August 6, 2019





Contents

1	Getting Started	2
2	Shell Basics	3
3	Shell Programming	6
4	C Programming Basics	10
4.1	Programming Environment	10
4.2	OS Basics	17
5	The Linux Environment	25
6	Working With Files	27
6.1	File	27
6.2	Directory	34
7	Processes and signals	37
8	IPC	42
9	User Interface	42
9.1	Dialog, Zenity	42
9.2	Ncurses	42
9.3	GTK+	42
9.4	Qt	42
10	Terminal	42
11	IDE	42

References

- [1] BRYANT R E, O'HALLARON D R. Computer Systems: A Programmer's Perspective. 2nd ed. USA: Addison-Wesley, 2010.
- [2] COOPER M. Advanced Bash Scripting Guide 5.3 Volume 1. Lulu.com, 2010.
- [3] MATTHEW N, STONES R. Beginning linux programming. John Wiley & Sons, 2008.
- [4] RAYMOND E S. The art of Unix programming. Addison-Wesley, 2003.
- [5] STEVENS W R, RAGO S A. Advanced programming in the UNIX environment. Addison-Wesley, 2013.

Course Web Links

 <https://cs6.swfu.edu.cn/moodle>
 https://cs2.swfu.edu.cn/~wx672/lecture_notes/linux-app/slides/
 https://cs2.swfu.edu.cn/~wx672/lecture_notes/linux-app/src/
 <https://cs3.swfu.edu.cn/tech>

/etc/hosts

```
202.203.132.241 cs6.swfu.edu.cn
202.203.132.242 cs2.swfu.edu.cn
202.203.132.245 cs3.swfu.edu.cn
```

Homework

Weekly tech question

1. What was I trying to do?
2. How did I do it? (steps)
3. The expected output? The real output?
4. How did I try to solve it? (steps, books, web links)
5. How many hours did I struggle?

✉ *wx672ster+linux@gmail.com*

🌐 Preferably in English

📖 in stackoverflow style

Or simply show me the tech questions you asked on any website

1 Getting Started

Linux Commands

Where to find them? /bin, /usr/bin, /usr/local/bin,
~/bin, ...
\$ echo \$PATH

How to find them? which, whereis, type

Command not found?

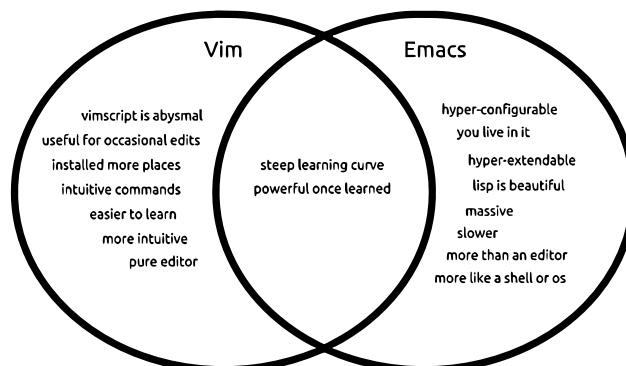
First double check your spelling

Then try:

```
🔍 aptitude search xxx
🔍 apt-cache search xxx
🔍 apt-file search xxx
🔍 sudo apt install packagename
🔍 Google "linux command xxx"
```

Text Editors

 vs. 

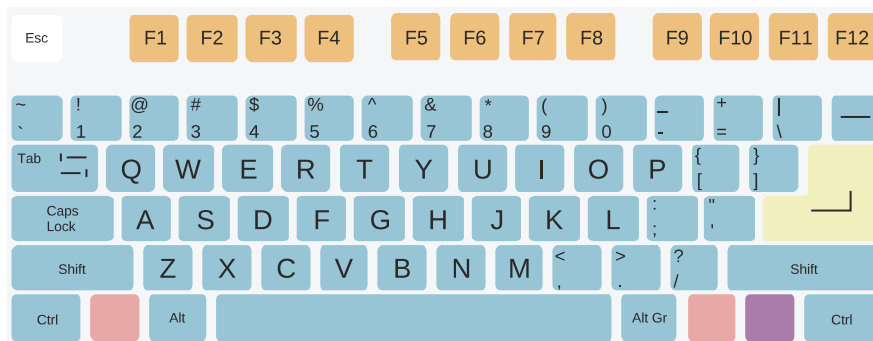




Help Your Editor

Suffix matters

```
$ vim ✗  
$ vim hello ✗  
$ vim hello.c ✓  
$ vim hello.py ✓  
$ emacs ✗  
$ emacs hello ✗  
$ emacsclient hello.c ✓  
$ emacsclient hello.py ✓
```

Keyboard

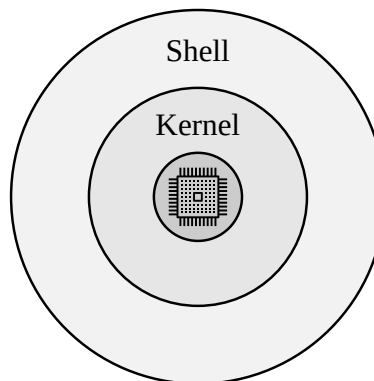


 vimtutor
 **Ctrl** + **h** **t**

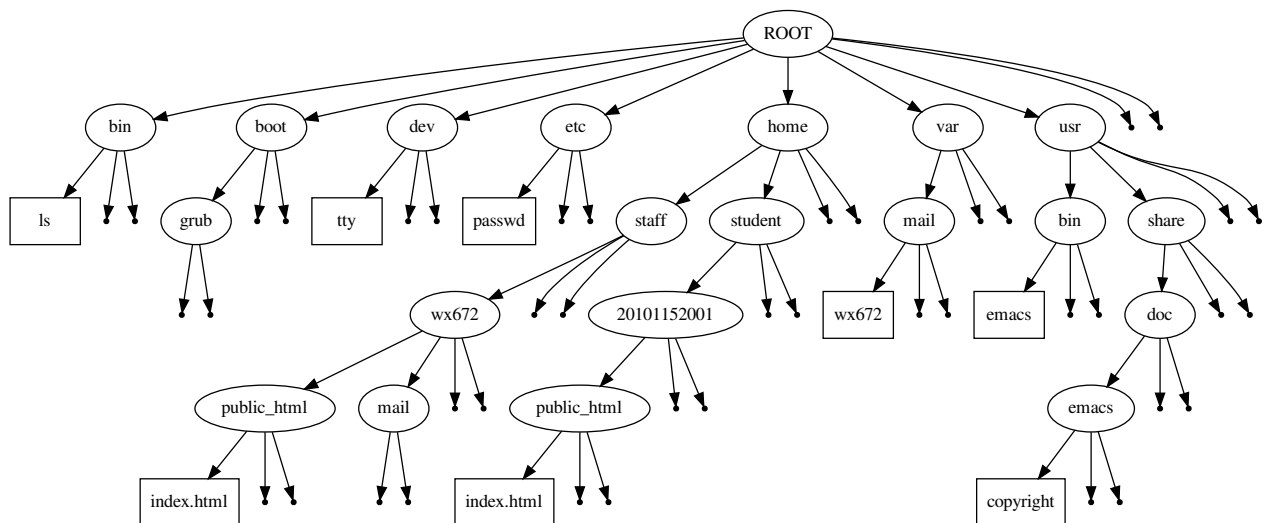
2 Shell Basics

Shell

- ☒ A command line interpreter
- ☒ A programming language



Directory Structure



Todo	How
Where am I?	pwd
What's in it?	ls
Move around?	cd
Disk usage?	du, df
USB drive?	lsblk, mount
New folder?	mkdir

File Operations

Ways to create a file

- Using an editor (vim, emacs, nano...), or
- `$ cat > filename`
- `$ echo "hello, world" > filename`
- `$ touch filename`

More file operations:

Todo	How	Todo	How
Copy?	cp	Move/Rename?	mv
Delete?	rm	What's it?	file
Link?	ln	Permission?	chmod, chown
Count?	wc	Archive?	tar, gzip, 7z, ...
Sort?	sort, uniq	Search?	find, grep

Process Operations

Todo	How	Todo	How
Kill?	kill, Ctrl-c	suspend?	Ctrl-z
background?	bg, &	foreground?	fg, jobs
status?	ps, top		

System Info

Todo	How	Todo	How
who?	w, who, whoami	how long?	uptime
software?	apt, aptitude, dpkg	kernel?	uname, lsmod
hardware?	lspci, lsusb, lscpu	memory?	free, lsmem

APT —  package management

Todo	How
upgrading?	apt update && apt upgrade
install?	apt install xxx
remove?	apt purge xxx
search?	apt search xxx
details?	apt show xxx
friendly UI?	aptitude

CLI Shortcuts

Ctrl + a : beginning of line	Ctrl + e : end of line
Ctrl + f : forward	Ctrl + b : backward
Ctrl + n : next	Ctrl + p : previous
Ctrl + r : reverse search	Ctrl + u : cut to beginning
Ctrl + k : kill (cut to end)	Ctrl + y : yank (paste)
Ctrl + d : delete a character	Ctrl + Tab : completion

Tmux

Ctrl + a c : create window	Ctrl + a Ctrl + a : switch window
Ctrl + a n : next window	Ctrl + a p : previous window
Ctrl + a - : split window	Ctrl + a I : split window
Ctrl + a j : go down	Ctrl + a k : go up
Ctrl + a l : go right	Ctrl + a h : go left

Understanding “ls -l”

-rw-----	1	sam	sam	57	Apr	17	1998	weather.txt
drwxr-xr-x	6	sam	sam	102	Oct	9	1999	web_page
-rw-rw-r--	1	sam	sam	7648	Feb	11	20:41	web_site.tar
-rw-----	1	sam	sam	574	Dec	16	1998	file.txt

								File Name
								Modification Time
								Size (in bytes)
								Group
								Owner
								Number of hard links
								File Permissions
								File types

d - directory
- - regular file
l - soft link
c - character device
b - block device
s - socket
p - named pipe (FIFO)

9-bit permission

```

7 5 5
111 101 101
rwx r-x r-x

```

```

└── Other
└── Group
└── User

```

```

$ chmod 755 foo
$ chmod 000 foo
$ chmod a-r foo
$ chmod g+w foo
$ chmod 644 foo
$ chmod 777 foo
$ chmod u+x foo
$ chmod go=rx foo

```

Wildcard Expansion

Character	Meaning	Example
?	any one	\$ ls ????.txt
*	zero or more	\$ ls *.c
[]	or	\$ ls *. [ch]
{}	and	\$ ls *. {c,h,cpp}

Example

```
$ touch {2,3,4,234}. {jpg,png} && ls
```

output:

```

2.jpg 234.jpg 3.jpg 4.jpg
2.png 234.png 3.png 4.png

```

```
$ rm [234].jpg
$ rm {2,3,4,234}.jpg
$ rm 2*

$ rm ?.jpg
$ rm ?.*
$ rm *
```

Everything Is A File

```
$ cat /dev/null > /var/log/messages # empty a file
$ : > /var/log/messages # no new process
$ ls > /dev/null
$ dd if=/dev/zero of=/tmp/clean bs=1k count=1k
$ dd if=/dev/urandom of=/tmp/random bs=1k count=1k
```

Redirection

Redirecting output

```
$ ls -l > output.txt
$ ps aux >> output.txt
```

Redirecting input

```
$ more < output.txt
```

Pipe

Chain processes together

```
$ ps aux | sort | less
```

3 Shell Programming

Variables

```
$ a=8; b=2
$ a=a+5; a=$a+5 ☹️
$ let a=a+5; let a+=5 😊
$ let b=b+a; let b+=a 😊
$ echo a; echo $a
$ (( a=5, b=6, a+=b )) 😊
$ (( b=a<5?8:9 )) 😊
$ r=$(( RANDOM%100 )) 😊
$ echo "$a" # partial quoting
$ echo '$a' # full quoting
$ a=$(ls -l); echo $a; echo "$a"
$ a=hello; b=world; let a+=b ☹️
```

Positional Parameters

`$0`, `$1`, `$2`, ..., `$@`, `$#`

```
1  #!/bin/bash
2
3  echo "You said:"
4
5  echo -e "\t$@"
6  echo
7  echo -e "\targc = $#"
```

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5      int i;
6      printf("You said:\n\t");
7
8      for(i=1; i<argc; i++)
9          printf("%s ",argv[i]);
10
11     printf("\n\n\targc = %d\n", argc);
12
13     for(i=0; i<argc; i++)
14         printf("\targv[%d] = %s\n",i,argv[i]);
15
16     return 0;
17 }
```

Parameter Substitution

Default value

```
$ echo ${s:=abc}          $ echo ${v:-8}
$ echo ${s:=xyz}          $ echo ${v:-10}
```

Example

```
1  #!/bin/bash
2
3  echo -n Hello, ${1:-world}
4  echo !
```

Parameter Substitution

Substring removal

```
$ for f in *.pbm; do ppm2tiff $f ${f%pbm}.tif; done
```

Substring replacement

```
$ for f in *.jpg; do mv $f ${f/jpg/JPG}; done
```

Environmental Variables

Each process has an environment

\$PATH	\$PWD	\$HOME	\$UID	\$USER
\$GROUPS	\$SHELL	\$TERM	\$DISPLAY	\$TEMP
\$HOSTNAME	\$HOSTTYPE	\$IFS	\$EDITOR	\$BROWSER
\$HISTSIZE	\$FUNCNAME	\$TMOUT	...	

```
export HISTSIZE=2000
export BROWSER='/usr/bin/x-www-browser'
export EDITOR='vim'
export ALTERNATE_EDITOR="vim"
export PDFVIEWER='/usr/bin/zathura'

$ env
$ declare
```

Tests

```
$ (( 5 < 6 )) && echo should be
$ [[ 1 < 2 ]] && echo of course
$ [[ $a -lt $b ]] && echo yes || echo no
$ if [[ $a -lt $b ]]; then echo yes; else echo no; fi
$ if test $a -lt $b; then echo of course; fi
$ if a = 5; then echo a=$a; fi # whitespace matters ✗
$ if a=5; then echo a=$a; fi ☹
$ if test a=5; then echo a=$a; fi ☹
$ if test a = 5; then echo a=$a; fi ☹
$ if test $a = 5; then echo a=$a; fi ✓
$ test $a = 5 && echo a=$a ✓
$ [[ $a = 5 ]] && echo a=$a ✓
$ [[ cmp a b ]] && echo same file ✗
$ if test cmp a b; then echo same file; fi ✗
$ if cmp a b; then echo same file; fi ✓
$ [[ -f ~/.bash_aliases ]] && . ~/.bash_aliases
$ [[ -x /usr/bin/xterm ]] && /usr/bin/xterm -e tmux &
$ [[ "$pass" != "$MYPASS" ]] && echo 'Wrong password!' && exit 1
```

```
$ help test
```

```
1  #!/bin/bash
2
3  words=$@
4  string=linux
5  if echo "$words" | grep -q "$string"
6  then
7      echo "$string found in $words"
8  else
9      echo "$string not found in $words"
10 fi
```

Loops

```
for ARG in LIST; do COMMAND(s); done
$ for i in 1 2 3; do echo -n i="$i "; done
$ for i in {1..10}; do echo $i; done
$ for i in $(seq 10); do echo $i; done
$ for ((i=1; i<=10; i++)); do echo $i; done
$ for ((i=1, j=1; i<=10; i++, j++)); do
    echo $i-$j ☹️
    echo $((i-$j)) 😊
done
$ for ((i=1; i<=10; i++)) { echo $i; } # C style
$ for i in hello world; do echo -n "$i "; done
```

Loops

```
while CONDITION; do COMMAND(s); done
$ a=0; while [[ a < 10 ]]; do echo $a; ((a++)); done ☹️
$ while [[ $a < 10 ]]; do echo $a; ((a++)); done ☹️
$ while [[ $a -lt 10 ]]; do echo $a; ((a++)); done ✓
$ while [ $a -lt 10 ]; do echo $a; ((a++)); done ✓
$ while (( a < 10 )); do echo $a; ((a++)); done ✓
$ until (( a = 10 )); do echo $a; ((a++)); done ☹️
$ until (( a == 10 )); do echo $a; ((a++)); done ✓
$ while read n; do n2 $n; done
$ while read n; do n2 $n; done < datafile
$ until (( n == 0 )); do read n; n2 $n; done
```

case


```

1  #!/bin/bash
2
3  [ -z "$1" ] && echo "Usage: `basename $0` [d|h][<number>]" && exit 0;
4
5  case "$1" in
6      [dD]*)
7          NUM=$(echo $1 | cut -b 2-)
8          printf "\tDec\tHex\tBin\n"
9          printf "\t%d\t0x%02X\t%s\n" $NUM $NUM $(bc <<< "obase=2;$NUM")
10         ;;
11      [hH]*)
12          NUM=$(echo $1 | cut -b 2-)
13          NUM=$(echo $NUM | tr [:lower:] [:upper:])
14          printf "\tHex\t\tDec\t\tBin\n"
15          printf "\t0x%s\t\t%s\t\t%s\n" $NUM $(bc <<< "ibase=16;obase=A;$NUM") \
16              $(bc <<< "ibase=16;obase=2;$NUM")
17          ;;
18      0[xX]*)
19          NUM=$(echo $1 | cut -b 3-)
20          NUM=$(echo $NUM | tr [:lower:] [:upper:])
21          printf "\tHex\t\tDec\t\tBin\n"
22          printf "\t0x%s\t\t%s\t\t%s\n" $NUM $(bc <<< "ibase=16;obase=A;$NUM") \
23              $(bc <<< "ibase=16;obase=2;$NUM")
24          ;;
25      [bB]*)
26          NUM=$(echo $1 | cut -b 2-)
27          printf "\tBin\t\tHex\t\tDec\n"
28          printf "\t%s\t\t0x%s\t\t%s\n" $NUM $(bc <<< "ibase=2;obase=10000;$NUM") \
29              $(bc <<< "ibase=2;obase=1010;$NUM")
30          ;;
31      *)
32          printf "Dec\tHex\tBin\n"
33          printf "%d\t0x%08X\t%08d\n" $1 $1 $(bc <<< "obase=2;$1")
34          ;;
35  esac

```

select

```

1  #!/bin/bash
2
3  PS3='Your favorite OS? '
4
5  select OS in "Linux" "Mac OSX" "Windows"
6  do
7      [[ "$OS" = "Linux" ]] && echo wise guy.
8      [[ "$OS" = "Mac OSX" ]] && echo rich guy.
9      [[ "$OS" = "Windows" ]] && echo patient guy.
10     break
11 done

```

Functions

```

1  #!/bin/bash
2
3  function screencapture(){
4      ffmpeg -f x11grab -s 1920x1080 -r 30 -i :0.0 \
5          -c:v libx264 -crf 0 -preset ultrafast screen.mkv
6  }
7
8  w2pdf(){
9      libreoffice --convert-to pdf:writer_pdf_Export "$1"
10 }
11
12 rfc(){
13     [[ -n "$1" ]] || {
14         cat <<EOF
15         rfc - Command line RFC viewer
16         Usage: rfc <index>
17     EOF
18         return 1
19     }
20     find /usr/share/doc/RFC/ -type f -iname "rfc$1.*" | xargs less
21 }

```

4 C Programming Basics

4.1 Programming Environment

Program Languages

Machine code

The *binary numbers* that the CPUs can understand.

100111000011101111001111 ... and so on ...

Assembly language — friendly to humans

People don't think in numbers.

```

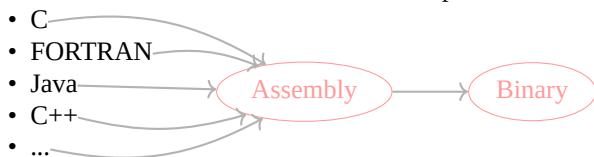
1  MOV A,47 ;1010 1111
2  ADD A,B  ;0011 0111
3  HALT     ;0111 0110

```

The ASM programs are translated to machine code by *assemblers*.

High level languages

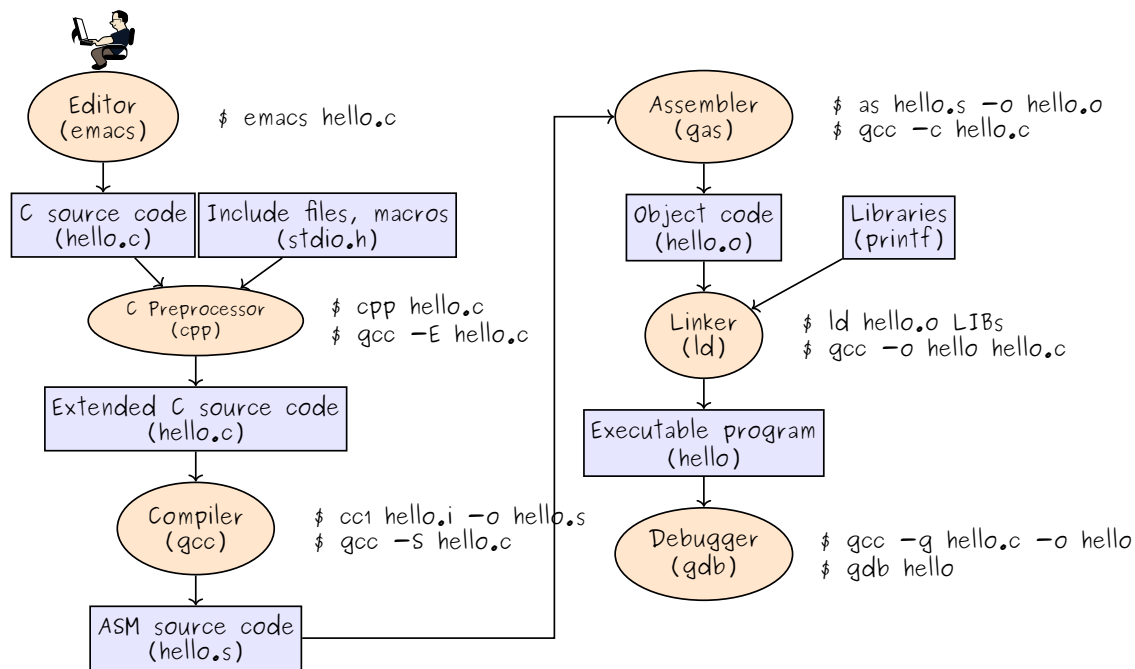
Even easier to understand for humans. Examples:



Compilers do the translation work.

4.1.1 The Tool Chain

Compilation



Source code written by programmer in high-level language, in our case in C. We write c source code with a *text editor*, such as emacs, vim, etc.

Preprocessing is the first pass of any C compilation. It processes include-files, conditional compilation instructions and macros.

cpp The GNU C preprocessor

```
$ gcc -E hello.c
```

Compilation is the second pass. It takes the output of the preprocessor, and the source code, and generates assembly source code.

gcc/g++ GNU C/C++ compiler

```
$ gcc -S hello.c
```

Assembly is the third stage of compilation. It takes the assembly source code and produces an assembly listing with offsets. The assembler output is stored in an object file.

as the portable GNU assembler

```
$ gcc -c hello.c
```

Linking is the final stage of compilation. It combines object code with predefined routines from libraries and produces the executable program.

ld The GNU linker

```
$ gcc hello.c -lm
```

Wrapper The whole compilation process is usually not done 'by hand', but using a wrapper program that combines the functions of preprocessor(cpp), compiler(gcc/g++), assembler(as) and linker(ld).

```
$ gcc -Wall hello.c -lm -o hello
```

Compiler vs. Interpreter

```
hello.c
#include <stdio.h>
int main()
{
    printf("Hello, world!\n");
    return 0;
}
```

```
$ gcc -o hello hello.c
$ ./hello
```

```
hello.sh
#!/bin/bash
echo 'Hello, world!'
```

```
$ chmod +x hello.sh
$ ./hello.sh
```

```
hello.py
```

```
#!/usr/bin/python
print "Hello, world!"
```

```
$ chmod +x hello.py
$ ./hello.py
```

4.1.2 Header Files

Header Files

Why?

```
#include "add.h"
```

```
int triple(int x)
{
    return add(x, add(x,x));
}
```

- Ensure everyone use the same code
- Easy to share, upgrade, reuse

Why not?

```
int add(int, int);
```

```
int triple(int x)
{
    return add(x, add(x, x));
}
```

In the header files...

- function declarations
- macro definitions
- constants
- system wide global variables

```
$ ls /usr/include/
```

4.1.3 Library Files

Library Files

Static libraries .a files. Very old ones, but still alive.

```
$ find /usr/lib -name "*.a"
```

Shared libraries .so files. The preferred ones.

```
$ find /usr/lib -name "*.so.*"
```

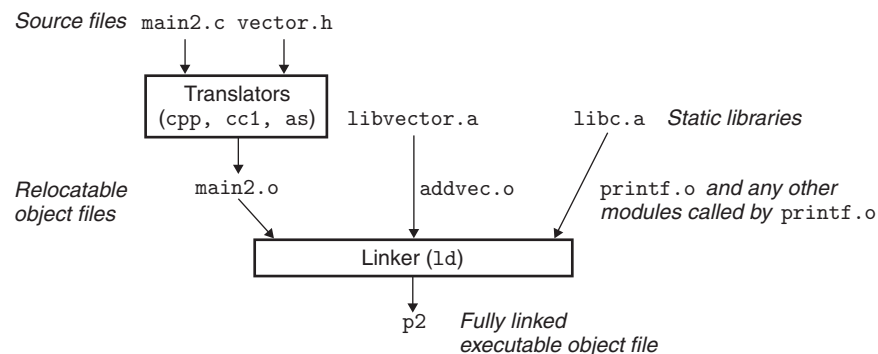
Examples:

```
$ gcc -o hello hello.c /usr/lib/libm.a
```

```
$ gcc -o hello hello.c -lm
```

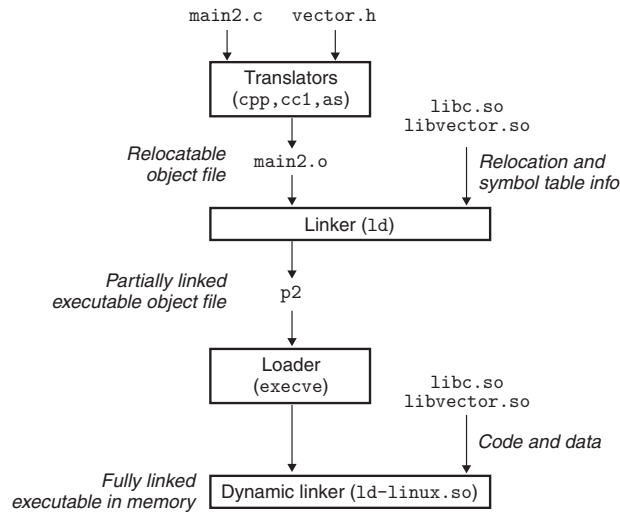
Static Linking

- The entire program and all data of a process must be in physical memory for the process to execute
- The size of a process is thus limited to the size of physical memory



Dynamic Linking

- Only one copy in memory
- Don't have to re-link after a library update



Build A Static Library

Source codes

main.c

```

1  #include "lib.h"
2
3  int main(int argc, char* argv[])
4  {
5      int i=1;
6
7      for (; i<argc; i++)
8      {
9          hello(argv[i]);
10         hi(argv[i]);
11     }
12     return 0;
13 }

```

lib.h

```

1  #include <stdio.h>
2
3  void hello(char *);
4  void hi(char *);

```

hello.c

```

1  #include <stdio.h>
2
3  void hello(char *arg)
4  {
5      printf("Hello, %s!\n", arg);
6  }

```

hi.c

```

1  #include <stdio.h>
2
3  void hi(char *arg)
4  {
5      printf("Hi, %s!\n", arg);
6  }

```

Build A Static Library

Step by step

1. Get *hello.o* and *hi.o*
\$ gcc -c hello.c hi.c
2. Put *.o into *libhi.a*
\$ ar crv libhi.a hello.o hi.o
3. Use *libhi.a*
\$ gcc main.c libhi.a

Build A Static Library

Makefile

```

1  main: main.c lib.h libhi.a
2      gcc -Wall -o main main.c libhi.a
3
4  libhi.a: hello.o hi.o
5      ar crv libhi.a hello.o hi.o
6
7  hello.o: hello.c
8      gcc -Wall -c hello.c
9
10 hi.o: hi.c
11     gcc -Wall -c hi.c
12
13 clean:
14     rm -f *.o *.a main

```

Build A Shared Library

Source codes

hello.c

```

1  #include "hello.h"
2
3  int main(int argc, char *argv[])
4  {
5      if (argc != 2)
6          printf ("Usage: %s needs an argument.\n", argv[0]);
7      else
8          hi(argv[1]);
9      return 0;
10 }

```

hello.h

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int hi(char*);

```

hi.c

```

1  #include "hello.h"
2
3  int hi(char* s)
4  {
5      printf ("Hi, %s\n",s);
6      return 0;
7  }

```

Build A Shared Library

Step by step

1. Get *hi.o*
\$ gcc -fPIC -c hi.c
2. Get *libhi.so*
\$ gcc -shared -o libhi.so hi.o
3. Use *libhi.so*
\$ gcc -L. -Wl,-rpath=. hello.c -lhi
4. Check it
\$ ldd a.out

Build A Shared Library

Makefile

```

1  # http://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html
2  # http://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html
3  #
4  # gcc -fPIC -c hi.c
5  # gcc -shared -o libhi.so hi.o
6  # gcc -L/current/dir -Wl,option -Wall -o hello hello.c -lhi
7  #
8  # -L          - tells ld where to search libraries
9  # -Wl,option - pass option as an option to the linker (ld)
10 # -rpath=dir - Add a directory to the runtime library search path
11
12 hello: hello.c hello.h libhi.so
13      gcc -L. -Wl,-rpath=. -Wall -o hello hello.c -lhi
14 libhi.so: hi.o hello.h
15      gcc -shared -o libhi.so hi.o
16 hi.o: hi.c hello.h
17      gcc -fPIC -c hi.c
18 clean:
19      rm *.o *.so hello

```

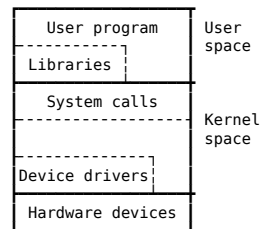
GNU C Library

Linux API > POSIX API

```

$ man 7 libc
$ man 3 intro
$ man gcc
$ info gcc
🔴 sudo apt install gcc-doc

```



4.1.4 The Make Utility

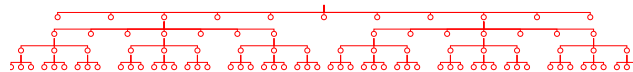
The Make Utility

To compile a single C program:

```
$ gcc hello.c -o hello
```

✓ OK, But...

What if you have a large project with 1000+ files?



Linux 4.9 source tree: 3799 directories, 55877 files

make: help you maintain your programs.

Makefile

```

1 | target: dependencies
2 |   ──TAB──> command

```

Example

```

1 | hello: hello.c
2 |   ──TAB──> gcc -o hello hello.c

```

```
$ info make makefiles
```

Makefile

```
1 edit: main.o kbd.o command.o display.o \  
2     insert.o search.o files.o utils.o  
3     gcc -Wall -o edit main.o kbd.o command.o display.o \  
4     insert.o search.o files.o utils.o  
5  
6 main.o: main.c defs.h  
7     gcc -c -Wall main.c  
8 kbd.o : kbd.c defs.h command.h  
9     gcc -c -Wall kbd.c  
10 command.o: command.c defs.h command.h  
11     gcc -c -Wall command.c  
12 display.o : display.c defs.h buffer.h  
13     gcc -c -Wall display.c  
14 insert.o: insert.c defs.h buffer.h  
15     gcc -c -Wall insert.c  
16 search.o: search.c defs.h buffer.h  
17     gcc -c -Wall search.c  
18 files.o: files.c defs.h buffer.h command.h  
19     gcc -c -Wall files.c  
20 utils.o: utils.c defs.h  
21     gcc -c -Wall utils.c  
22  
23 clean:  
24     rm edit main.o kbd.o command.o display.o \  
25     insert.o search.o files.o utils.o
```

```
./  
├── command.c  
├── display.c  
├── files.c  
├── insert.c  
├── kbd.c  
├── main.c  
├── search.c  
├── utils.c  
├── buffer.h  
├── command.h  
├── defs.h  
└── Makefile
```

4.1.5 Version Control

git

To create a new local git repo

In your source code directory, do:

```
$ git init  
$ git add .  
$ git commit -m "something to say..."
```

To clone a remote repo



Example:

```
$ git clone https://github.com/wx672/lecture-notes.git  
$ git clone https://github.com/wx672/dotfile.git
```

Most commonly used git Commands

```
$ git add filename[s]  
$ git rm filename[s]  
$ git commit  
$ git status  
$ git log  
$ git diff  
$ git push  
$ git pull  
$ git help {add,rm,commit,...}
```

```
$ man gittutorial  
$ man gittutorial-2
```

```
 sudo apt install git  
 https://github.com
```

4.1.6 Manual Pages

Man page

Layout


```

1  NAME
2      A one-line description of the command.
3  SYNOPSIS
4      A formal description of how to run it and what
5      command line options it takes.
6  DESCRIPTION
7      A description of the functioning of the command.
8  EXAMPLES
9      Some examples of common usage.
10 SEE ALSO
11     A list of related commands or functions.
12 BUGS
13     List known bugs.
14 AUTHOR
15     Specify your contact information.
16 COPYRIGHT
17     Specify your copyright information.

```

Man Page

Groff source code

```

1  .\" Text automatically generated by txt2man
2  .TH untitled "06 August 2019" "" ""
3  .SH NAME
4  \fBA one-line description of the command.
5  .SH SYNOPSIS
6  .nf
7  .fam C
8  \fBA formal description of how to run it and what command line options it takes.
9  .fam T
10 .fi
11 .fam T
12 .fi
13 .SH DESCRIPTION
14 \fBA description of the functioning of the command.
15 .SH EXAMPLES
16 Some examples of common usage.
17 .SH SEE ALSO
18 \fBA list of related commands or functions.
19 .SH BUGS
20 List known bugs.
21 .SH AUTHOR
22 Specify your contact information.
23 .SH COPYRIGHT
24 Specify your copyright information.

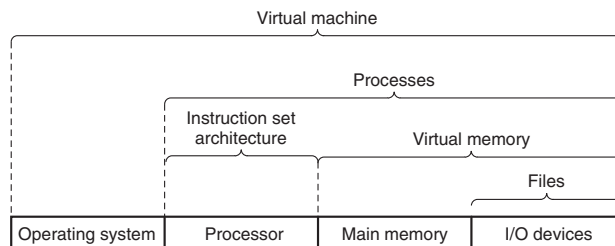
```

\$ man 7 groff
 \$ man txt2man
 \$ man a2x
 \$ ls /usr/share/man

4.2 OS Basics

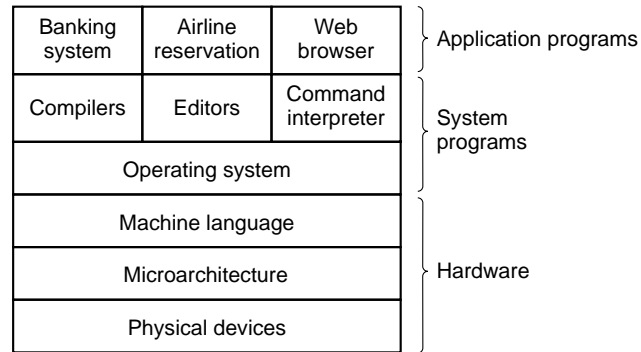
Abstractions

To hide the complexity of the actual implementations



See also: [Computer Systems: A Programmer's Perspective, Sec. 1.9.2, *The Importance of Abstractions in Computer Systems*].

A Computer System



4.2.1 Hardware

CPU Working Cycle



1. Fetch the first instruction from memory
2. Decode it to determine its type and operands
3. execute it

Special CPU Registers

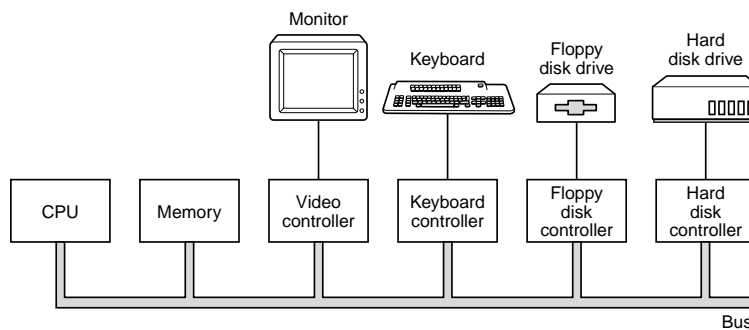
Program counter (PC): keeps the memory address of the next instruction to be fetched

Stack pointer (SP): points to the top of the current stack in memory

Program status (PS): holds

- condition code bits
- processor state

System Bus



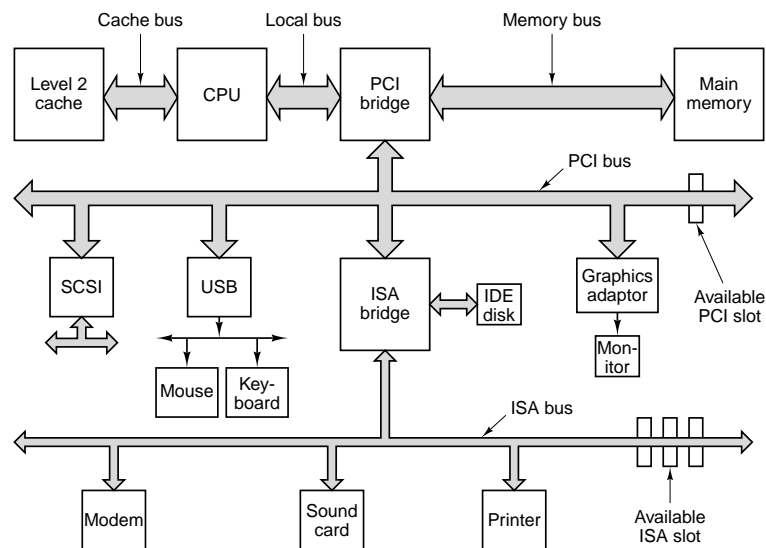
Address Bus: specifies the memory locations (addresses) for the data transfers

Data Bus: holds the data transferred. Bidirectional

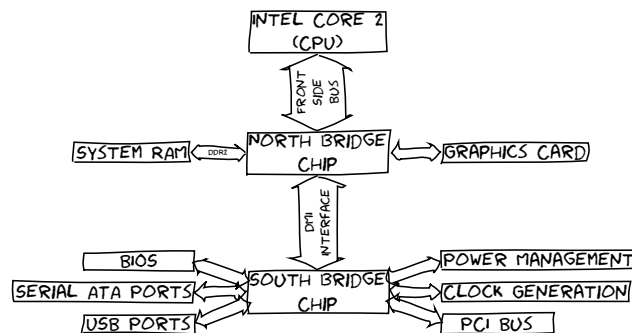
Control Bus: contains various lines used to route timing and control signals throughout the system

Controllers and Peripherals

- Peripherals are real devices controlled by controller chips
- Controllers are processors like the CPU itself, have control registers
- Device driver writes to the registers, thus control it
- Controllers are connected to the CPU and to each other by a variety of buses



Motherboard Chipsets



See also: *Motherboard Chipsets And The Memory Map* ¹.

- The CPU doesn't know what it's connected to
 - CPU test bench? network router? toaster? brain implant?
- The CPU talks to the outside world through its pins
 - some pins to transmit the physical memory address
 - other pins to transmit the values
- The CPU's gateway to the world is the *front-side bus*

Intel Core 2 QX6600

- 33 pins to transmit the physical memory address
 - so there are 2^{33} choices of memory locations
- 64 pins to send or receive data
 - so data path is 64-bit wide, or 8-byte chunks

This allows the CPU to physically address 64GB of memory ($2^{33} \times 8B$)

¹<http://duartes.org/gustavo/blog/post/motherboard-chipsets-memory-map>

See also: *Datasheet for Intel Core 2 Quad-Core Q6000 Sequence* ².

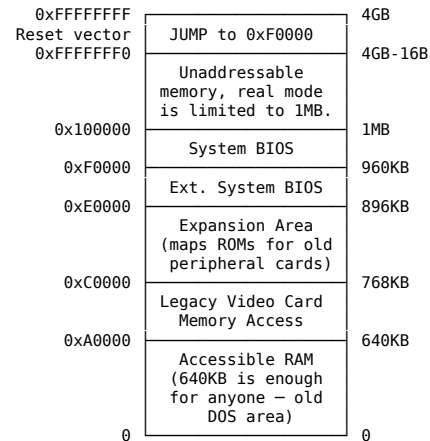
Some physical memory addresses are mapped away!

- only the addresses, not the spaces
- Memory holes
 - 640 KiB ~ 1 MiB
 - /proc/iomem

Memory-mapped I/O

- BIOS ROM
- video cards
- PCI cards
- ...

This is why 32-bit OSes have problems using 4 GiB of RAM.



the northbridge

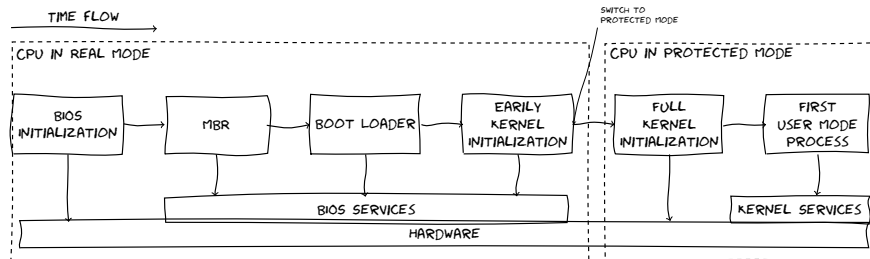
1. receives a physical memory request
 2. decides where to route it
 - to RAM? to video card? to ...?
 - decision made via the *memory address map*
- When is the memory address map built? `setup()`.

4.2.2 Bootstrapping

Bootstrapping

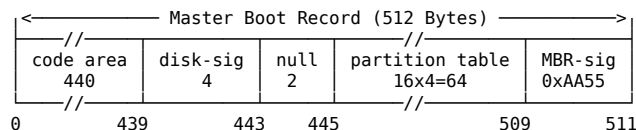
Can you pull yourself up by your own bootstraps?

A computer cannot run without first loading software but must be running before any software can be loaded.



Intel x86 Bootstrapping

1. BIOS (0xfffff0)
 - ➡ POST ➡ HW init ➡ Find a boot device (FD,CD,HD...) ➡ Copy sector zero (MBR) to RAM (0x00007c00)
2. MBR – the first 512 bytes, contains
 - Small code (< 446 B), e.g. GRUB stage 1, for loading GRUB stage 2
 - the primary partition table (16 × 4 = 64 B)
 - its job is to load the second-stage boot loader.
3. GRUB stage 2 — load the OS kernel into RAM
4. startup
5. init — the first user-space process



```
$ sudo hd -n512 /dev/sda
```

²<http://download.intel.com/design/processor/datashts/31559205.pdf>

4.2.3 Interrupt

Why Interrupt?

While a process is reading a disk file, can we do...

```
1 while(!done_reading_a_file())
2 {
3     let_CPU_wait();
4     // or...
5     lend_CPU_to_others();
6 }
7 operate_on_the_file();
```

Modern OS are Interrupt Driven

HW INT by sending a signal to CPU

SW INT by executing a *system call*

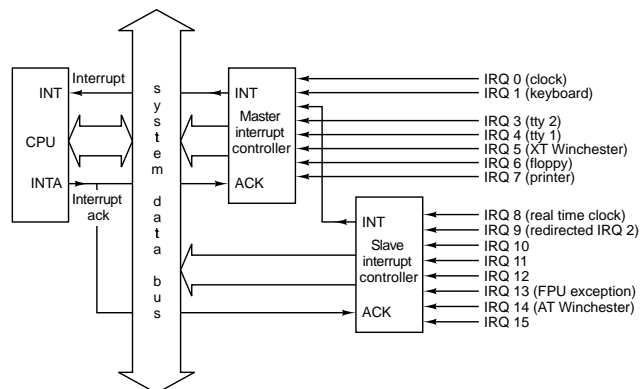
Trap (exception) is a software-generated INT caused by an error or by a specific request from an user program

Interrupt vector is an array of pointers to the memory addresses of *interrupt handlers*. This array is indexed by a unique device number

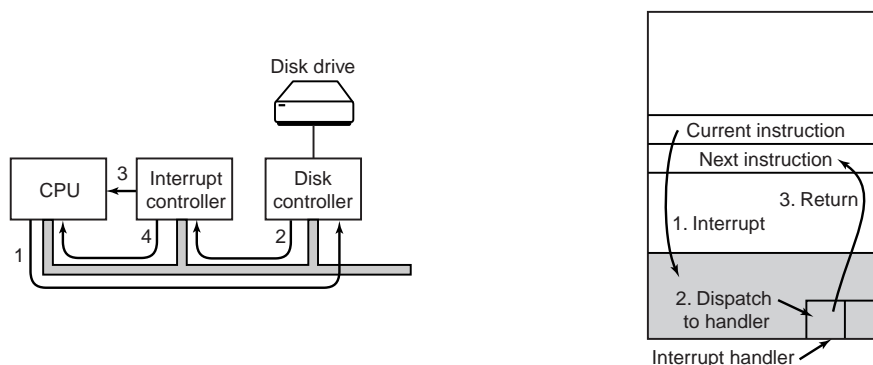
```
$ less /proc/devices
```

```
$ less /proc/interrupts
```

Programmable Interrupt Controllers

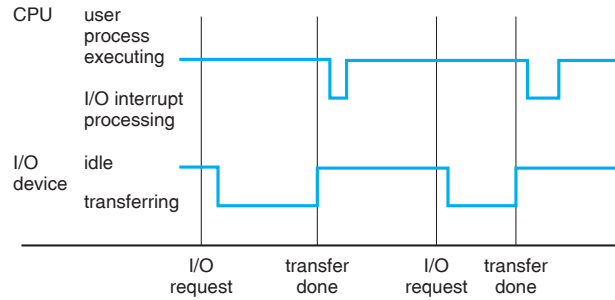


Interrupt Processing



Detailed explanation: in **tanenbaum2008modern**.

Interrupt Timeline



4.2.4 System Calls

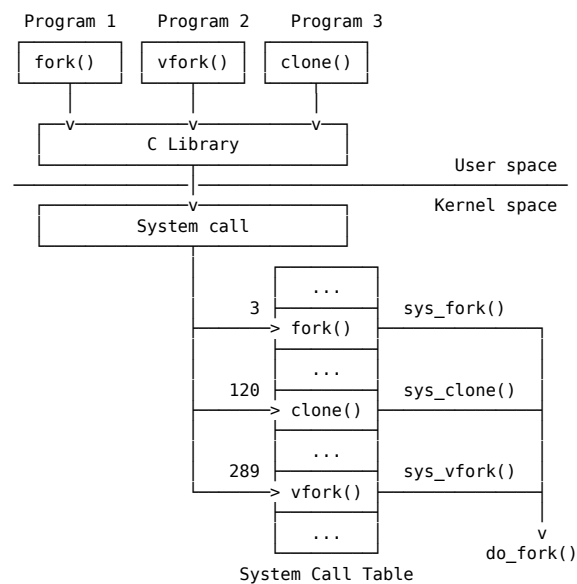
System Calls

A System Call

- is how a program requests a service from an OS kernel
- provides the interface between a process and the OS

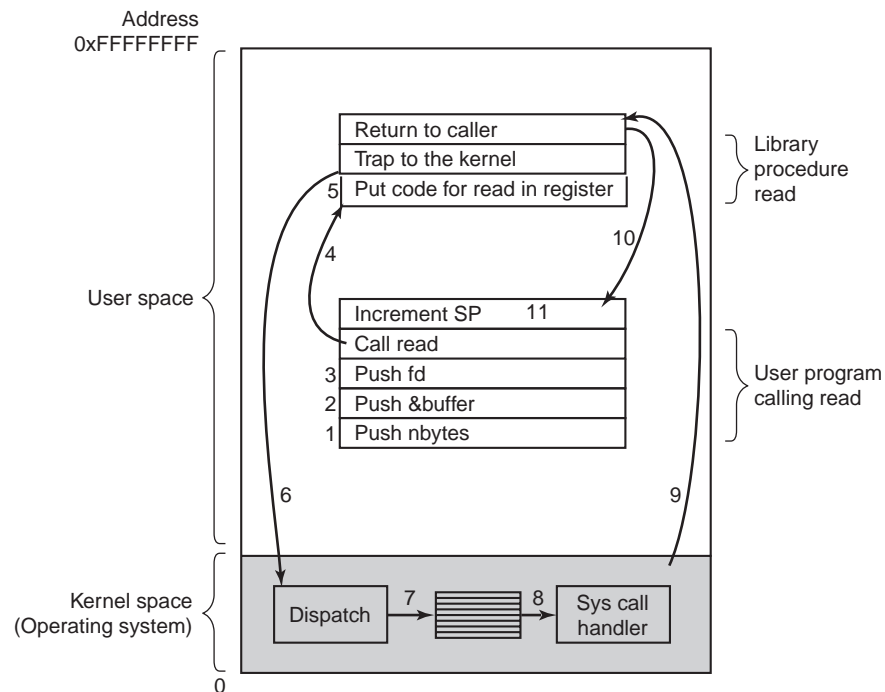
\$ man 2 intro

\$ man 2 syscalls



The 11 steps in making a system call

read(fd, buffer, nbytes)



Example

Linux `INT 80h`

Interrupt Vector Table: The very first 1KiB of x86 memory.

- 256 entries \times 4B = 1KiB
- Each entry is a complete memory address (segment:offset)
- It's populated by Linux and BIOS
- Slot 80h: address of the kernel services dispatcher (☛ sys-call table)

Example

```

1  Msg: db 'Hello, world'
2  MsgLen: equ $-Msg
3  mov eax,4          ; sys_write syscall = 4
4  mov ebx,1          ; 1 = STDOUT
5  mov ecx,Msg         ; offset of the message
6  mov edx,MsgLen      ; length of string
7  int 80h            ; call the kernel

```

```

$ nasm -f elf64 hello.asm -o hello.o
$ ld hello.o -o hello
$ ./hello

```

Process management	
Call	Description
<code>pid = fork()</code>	Create a child process identical to the parent
<code>pid = waitpid(pid, &statloc, options)</code>	Wait for a child to terminate
<code>s = execve(name, argv, environp)</code>	Replace a process' core image
<code>exit(status)</code>	Terminate process execution and return status

File management	
Call	Description
<code>fd = open(file, how, ...)</code>	Open a file for reading, writing or both
<code>s = close(fd)</code>	Close an open file
<code>n = read(fd, buffer, nbytes)</code>	Read data from a file into a buffer
<code>n = write(fd, buffer, nbytes)</code>	Write data from a buffer into a file
<code>position = lseek(fd, offset, whence)</code>	Move the file pointer
<code>s = stat(name, &buf)</code>	Get a file's status information

Directory and file system management	
Call	Description
<code>s = mkdir(name, mode)</code>	Create a new directory
<code>s = rmdir(name)</code>	Remove an empty directory
<code>s = link(name1, name2)</code>	Create a new entry, name2, pointing to name1
<code>s = unlink(name)</code>	Remove a directory entry
<code>s = mount(special, name, flag)</code>	Mount a file system
<code>s = umount(special)</code>	Unmount a file system

Miscellaneous	
Call	Description
<code>s = chdir(dirname)</code>	Change the working directory
<code>s = chmod(name, mode)</code>	Change a file's protection bits
<code>s = kill(pid, signal)</code>	Send a signal to a process
<code>seconds = time(&seconds)</code>	Get the elapsed time since Jan. 1, 1970

Fig. 1-18. Some of the major POSIX system calls. The return code *s* is -1 if an error has occurred. The return codes are as follows: *pid* is a process id, *fd* is a file descriptor, *n* is a byte count, *position* is an offset within the file, and *seconds* is the elapsed time. The parameters are explained in the text.

System Call Examples

`fork()`

```

1  #include <stdio.h>
2  #include <unistd.h>
3
4  int main ()
5  {
6      printf("Hello World!\n");
7      fork();
8      printf("Goodbye Cruel World!\n");
9      return 0;
10 }
```

`$ man 2 fork`

`exec()`


```

1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main()
5 {
6     printf("Hello World!\n");
7     if( fork() != 0 )
8         printf("I am the parent process.\n");
9     else {
10         printf("A child is listing the directory contents...\n");
11         execl("/bin/ls", "ls", "-al", NULL);
12     }
13     return 0;
14 }

```

`$ man 3 exec`

Quoted from [stackoverflow](#): What is the difference between the functions of the *exec* family of system calls:

There is no *exec* system call — this is usually used to refer to all the *execXX* calls as a group. They all do essentially the same thing: loading a new program into the current process, and provide it with arguments and environment variables. The differences are in how the program is found, how the arguments are specified, and where the environment comes from.

- The calls with *v* in the name take an array parameter to specify the *argv*[] array (*vector*) of the new program.
- The calls with *l* in the name take the arguments of the new program as a variable-length argument *list* to the function itself.
- The calls with *e* in the name take an extra argument to provide the *environment* of the new program; otherwise, the program inherits the current process's environment.
- The calls with *p* in the name search the *PATH* environment variable to find the program if it doesn't have a directory in it (i.e. it doesn't contain a / character). Otherwise, the program name is always treated as a path to the executable.

5 The Linux Environment

Command Line Options

getopt.c

```

1  #include <stdio.h>
2  #include <unistd.h>
3
4  int main(int argc, char* argv[]) {
5      int opt;
6
7      while ((opt = getopt(argc, argv, "hf:l")) != -1) {
8          switch (opt) {
9              case 'h':
10                 printf("Usage: %s [-h] [-f file] [-l]\n", argv[0]);
11                 break;
12             case 'l':
13                 printf("option: %c\n", opt);
14                 break;
15             case 'f':
16                 printf("filename: %s\n", optarg);
17                 break;
18             }
19         }
20     return 0;
21 }

```

`$ man 3 getopt`

Command Line Options

getopt.sh

```
1  #!/bin/bash
2
3  while getopts hf:l OPT; do
4      case $OPT in
5          h) echo "usage: `basename $0` [-h] [-f file] [-l]"
6              exit 1 ;;
7          l) echo "option: l" ;;
8          f) echo "filename: $OPTARG" ;;
9          esac
10     done
```

```
$ ./getopt.sh -h
$ ./getopt.sh -lf filename
$ ./getopt.sh -l -f filename
$ ./getopt.sh -f filename -l
```

Environment Variable

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  extern char** environ;
5
6  int main() {
7      char** env = environ;
8
9      while (*env) {
10         printf("%s\n", *env);
11         env++;
12     }
13
14     return 0;
15 }
```

\$ env
\$ man 3 getenv
\$ man 3 putenv

Time and Date

```
1  #include <time.h>
2  #include <stdio.h>
3
4  int main(void)
5  {
6      time_t t = time(NULL); /* long int */
7
8      printf("epoch time:\t%ld\n",t);
9      printf("calendar time:\t%s", ctime(&t));
10
11     return 0;
12 }
```

- January 1 1970 — start of the Unix epoch

```
$ man 3 time
$ man 3 ctime
```

Temporary Files

mkstemp.c

```
1  #include <stdlib.h>
2  #include <unistd.h>
3  #define _GNU_SOURCE
4  #include <stdio.h>
5
6  int main(int argc, char *argv[])
7  {
8      char c, *f;
9
10     asprintf(&f, "%sXXXXXX", argv[1]);
11     int tmp = mkstemp(f);
12
13     while ( read(0, &c, 1) == 1)
14         write(tmp, &c, 1);
15
16     unlink(f);
17     free(f);
18     return 0;
19 }
```

mktemp.sh

```
1  #!/bin/bash
2
3  tmp=$(mktemp)
4
5  while read LINE; do
6      echo $LINE >> $tmp
7  done
8
9
$ man 3 mkstemp
$ man 3 tmpfile
$ man 3 asprintf
```

Logging

syslog.c

```
1  #include <syslog.h>
2  #include <sys/stat.h>
3  #include <fcntl.h>
4
5  int main(int argc, char *argv[])
6  {
7      if ( open(argv[1], O_RDONLY) < 0 )
8          syslog(LOG_ERR | LOG_USER, "%s - %m\n", argv[1]);
9      else
10         syslog(LOG_INFO | LOG_USER, "%s - %m\n", argv[1]);
11     return 0;
12 }
```

logger.sh

```
1  #!/bin/bash
2
3  [[ -f "$1" ]] && logger "$1 exists." || logger "$1 not found."
```

6 Working With Files

6.1 File

File

A logical view of information storage

User's view

A file is the smallest storage unit on disk.

- Data cannot be written to disk unless they are within a file

UNIX view

Each file is a sequence of 8-bit bytes

- It's up to the application program to interpret this byte stream.

File

What is stored in a file?

Source code, object files, executable files, shell scripts, PostScript...

Different type of files have different structure

- UNIX looks at contents to determine type
 - Shell scripts** start with “#!”
 - PDF** start with “%PDF...”
 - Executables** start with *magic number*
- Windows uses file naming conventions
 - executables** end with “.exe” and “.com”
 - MS-Word** end with “.doc”
 - MS-Excel** end with “.xls”

File Types

Regular files: ASCII, binary

Directories: Maintaining the structure of the FS

In UNIX, everything is a file

Character special files: I/O related, such as terminals, printers ...

Block special files: Devices that can contain file systems, i.e. disks

Disks — logically, linear collections of blocks; disk driver translates them into physical block addresses

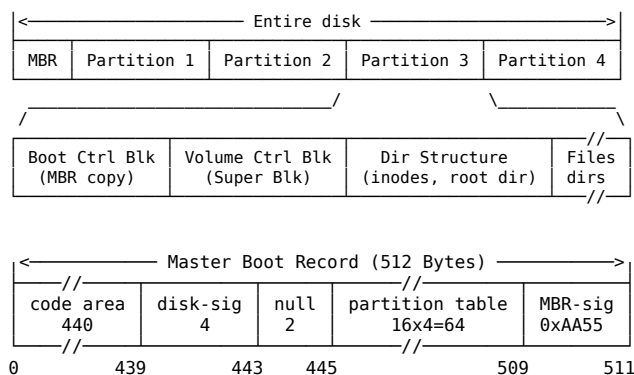
File Operations

POSIX file system calls

<code>creat(name, mode)</code>	<code>read(fd, buffer, byte_count)</code>
<code>open(name, flags)</code>	<code>write(fd, buffer, byte_count)</code>
<code>close(fd)</code>	<code>lseek(fd, offset, whence)</code>
<code>link(oldname, newname)</code>	<code>chown(name, owner, group)</code>
<code>unlink(name)</code>	<code>fchown(fd, owner, group)</code>
<code>truncate(name, size)</code>	<code>chmod(name, mode)</code>
<code>ftruncate(fd, size)</code>	<code>fchmod(fd, mode)</code>
<code>stat(name, buffer)</code>	<code>utimes(name, times)</code>
<code>fstat(fd, buffer)</code>	

File System Implementation

A typical file system layout



On-Disk Information Structure

Boot block a MBR copy

Superblock Contains volume details

number of blocks	size of blocks
free-block count	free-block pointers
free FCB count	free FCB pointers

I-node Organizes the files FCB (File Control Block), contains file details (metadata).

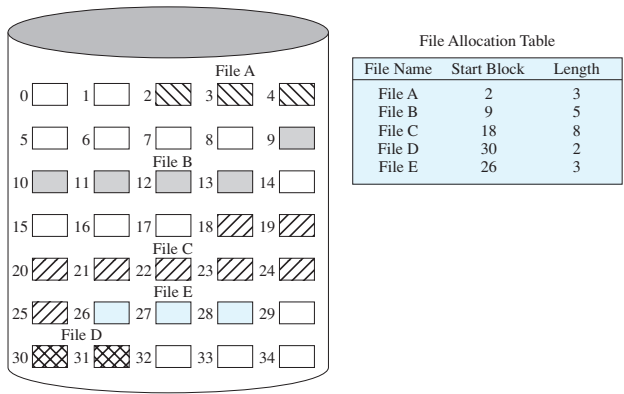
Superblock

Keeps information about the file system

- Type — ext2, ext3, ext4...
- Size
- Status — how it's mounted, free blocks, free inodes, ...
- Information about other metadata structures

```
$ sudo dumpe2fs /dev/sda1 | less
```

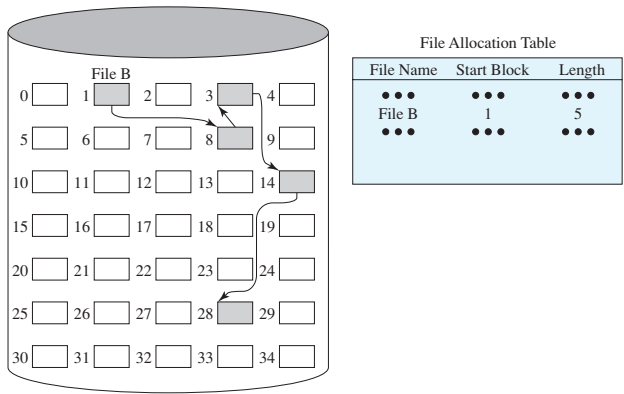
Implementing Files



Contiguous Allocation

- 😊 simple
- 😊 good for read only

☹ fragmentation



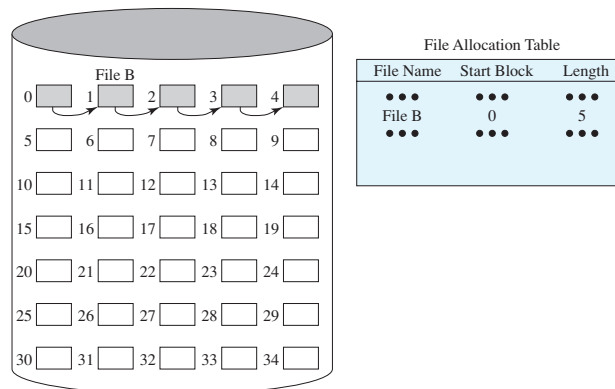
Linked List (Chained) Allocation

A pointer in each disk block

- 😊 no waste block
- 😞 slow random access

⊗ not 2^n

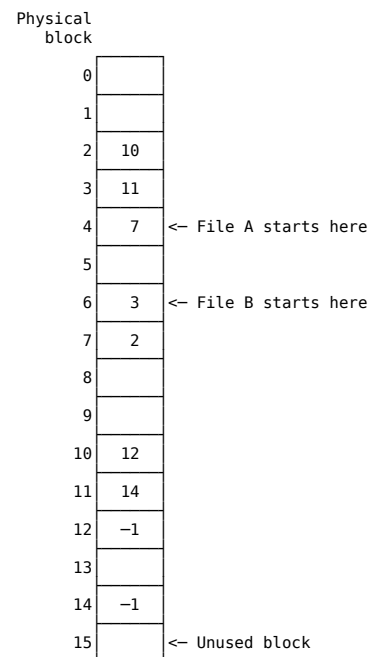
Linked List (Chained) Allocation Though there is no external fragmentation, consolidation is still preferred.



FAT: Linked list allocation with a table in RAM

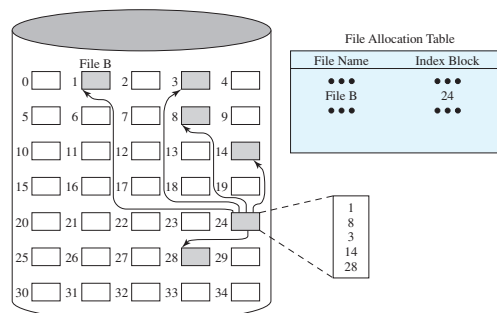
- Taking the pointer out of each disk block, and putting it into a table in memory
- fast random access (chain is in RAM)
- is 2^n
- the entire table must be in RAM

$$disk \nearrow \Rightarrow FAT \nearrow \Rightarrow RAM_{used} \nearrow$$



See also: [wiki:fat](https://en.wikipedia.org/wiki/fat).

Indexed Allocation

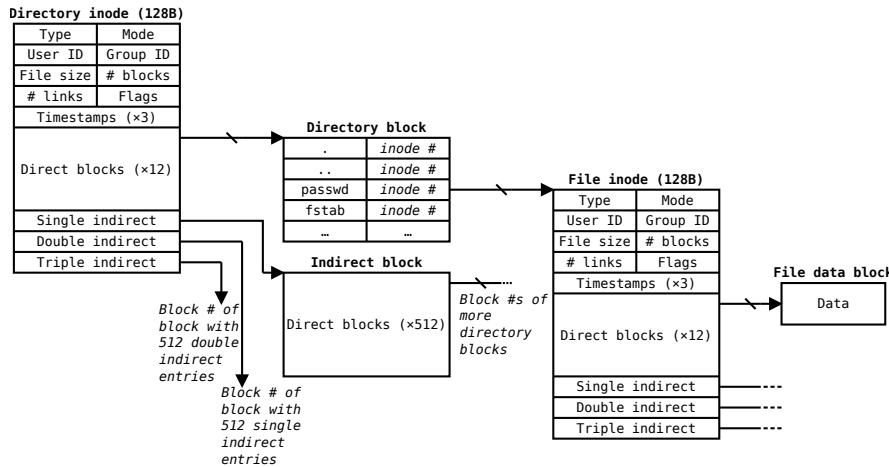


I-node A data structure for each file. An i-node is in memory *only if* the file is open

$$files_{opened} \nearrow \Rightarrow RAM_{used} \nearrow$$

See also: [wiki:inode](https://en.wikipedia.org/wiki/inode).

UNIX Treats a Directory as a File

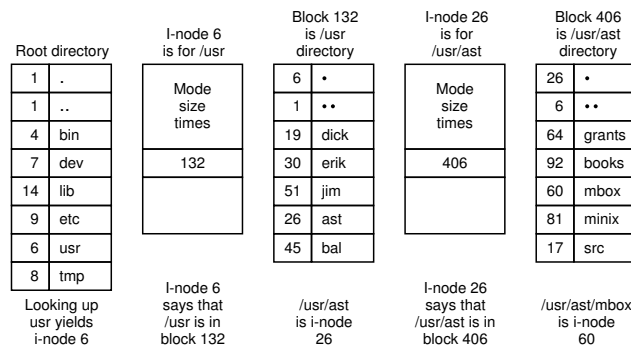


`open()`

Why? To avoid constant searching

- Without `open()`, every file operation involves searching the directory for the file.

The steps in looking up `/usr/ast/mbox`



`fd open(pathname, flags)`

A per-process *open-file table* is kept in the OS

- upon a successful `open()` syscall, a new entry is added into this table
- indexed by *file descriptor* (`fd`)
- `close()` to remove an entry from the table

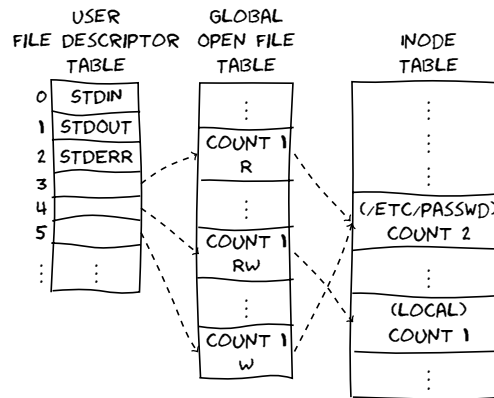
To see files opened by a process, e.g. `init`

```
$ lsof -p 1
```

```
$ man 2 open
```

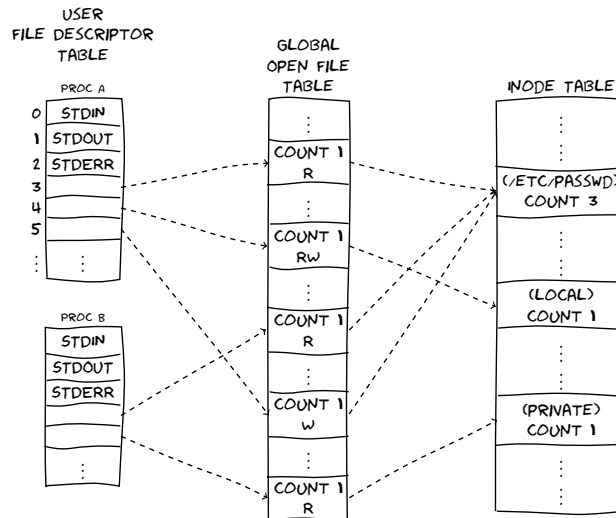
A process executes the following code:

```
fd1 = open("/etc/passwd", O_RDONLY);
fd2 = open("local", O_RDWR);
fd3 = open("/etc/passwd", O_WRONLY);
```



One more process B:

```
fd1 = open("/etc/passwd", O_RDONLY);
fd2 = open("private", O_RDONLY);
```



write()

```
1 #include <unistd.h>
2
3 int main(void)
4 {
5     write(1, "Hello, world!\n", 14);
6
7     return 0;
8 }
```

\$ man 2 write

\$ man 3 write

read()

```
1 #include <unistd.h>
2
3 int main(void)
4 {
5     char buffer[10];
6
7     read(0, buffer, 10);
8
9     write(1, buffer, 10);
10
11     return 0;
12 }
```

\$ man 2 read

\$ man 3 read

cp


```

1  #include <sys/types.h>  /* include necessary header files */
2  #include <fcntl.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5
6  #define BUF_SIZE 4096    /* use a buffer size of 4096 bytes */
7  #define OUTPUT_MODE 0700 /* protection bits for output file */
8
9  int main(int argc, char *argv[])
10 {
11     int in, out, rbytes, wbytes;
12     char buf[BUF_SIZE];
13
14     if (argc != 3) exit(1);
15
16     if ( (in = open(argv[1], O_RDONLY)) < 0 ) exit(2); /* open source file */
17
18     if ( (out = creat(argv[2], OUTPUT_MODE)) < 0 ) exit(3); /* create destination file */
19
20     while (1) { /* Copy loop */
21         if ( (rbytes = read(in, buf, BUF_SIZE)) <= 0 ) break; /* read a block of data */
22         if ( (wbytes = write(out, buf, rbytes)) <= 0 ) exit(4); /* write data */
23     }
24
25     close(in);
26     close(out);
27     if (rbytes == 0) exit(0); /* no error on last read */
28     else exit(5);           /* error on last read */
29 }

```

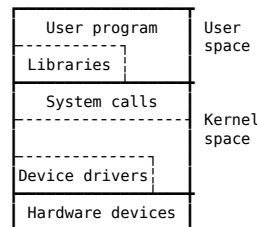
stdio — The Standard I/O Library

System calls: open(), read(), write(), close()...

Library functions: fopen(), fread(), fwrite(), fclose()...

Avoid calling syscalls directly as much as you can

- Portability
- Buffered I/O



open() vs. fopen()

open()

```

1  #include <unistd.h>
2  #include <sys/stat.h>
3  #include <fcntl.h>
4
5  int main()
6  {
7      char c;
8      int in;
9      in = open("/tmp/1m.test", O_RDONLY);
10
11     while (read(in, &c, 1) == 1);
12
13     return 0;
14 }

```

\$ strace -c ./open

fopen() — Buffered I/O

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      FILE *stream;
6
7      stream = fopen("/tmp/1m.test", "r");
8
9      while ( fgetc(stream) != EOF );
10
11     fclose(stream);
12
13     return 0;
14 }

```

\$ strace -c ./fopen

```
$ dd if=/dev/zero of=/tmp/1m.test bs=1k count=1024  
|https://stackoverflow.com/questions/1658476/c-fopen-vs-open|
```

cp — With stdio

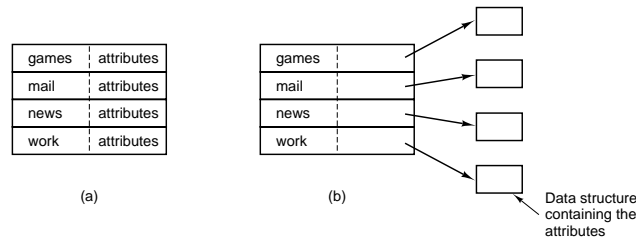
```
1  #include <stdio.h>  
2  #include <stdlib.h>  
3  
4  int main(int argc, char *argv[])  
5  {  
6      FILE *in, *out;  
7      int c=0;  
8  
9      if (argc != 3) exit(1);  
10  
11     in = fopen(argv[1], "r");  
12     out = fopen(argv[2], "w");  
13  
14     while ( (c = fgetc(in)) != EOF )  
15         fputc(c, out);  
16  
17     return 0;  
18 }
```

Homework: Try fread()/fwrite() instead.

|https://stackoverflow.com/questions/32742430/is-getc-a-macro-or-a-function| |https://stackoverflow.com/questions/9104568/macro-vs-function-in-c|

6.2 Directory

Implementing Directories



(a) A simple directory (Windows)

- fixed size entries
- disk addresses and attributes in directory entry

(b) Directory in which each entry just refers to an i-node (UNIX)

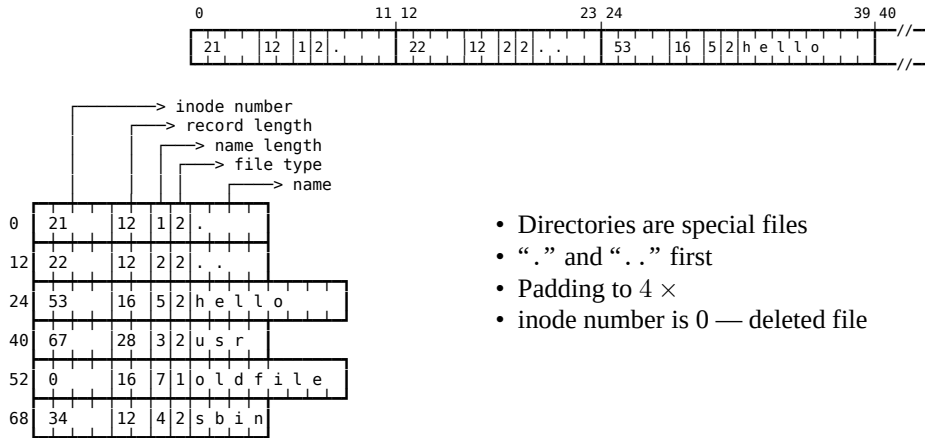
Directory entry in glibc

```
1  struct dirent {  
2      ino_t      d_ino;      /* Inode number */  
3      off_t      d_off;      /* Not an offset; see below */  
4      unsigned short d_reclen; /* Length of this record */  
5      unsigned char d_type;    /* Type of file; not supported  
6                               by all filesystem types */  
7      char       d_name[256]; /* Null-terminated filename */  
8  };
```

```
$ man readdir
```

```
$ view /usr/include/x86_64-linux-gnu/bits/dirent.h
```

Ext2 Directories



- Directories are special files
- “.” and “..” first
- Padding to 4 ×
- inode number is 0 — deleted file

ls

```

1  #include <sys/types.h>
2  #include <dirent.h>
3  #include <stddef.h>
4  #include <stdio.h>
5
6  int main(int argc, char *argv[])
7  {
8      DIR *dp;
9      struct dirent *entry;
10
11     dp = opendir(argv[1]);
12
13     while ( (entry = readdir(dp)) != NULL ){
14         printf("%s\n", entry->d_name);
15     }
16
17     closedir(dp);
18
19     return 0;
20 }
```

The real ls.c?

- 116 A4 pages
- 5308 lines

Do one thing, and do it really well.

\$ apt source coreutils

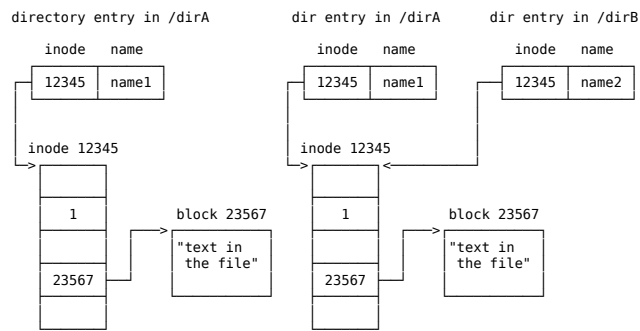
mkdir(), chdir(), rmdir(), getcwd()

```

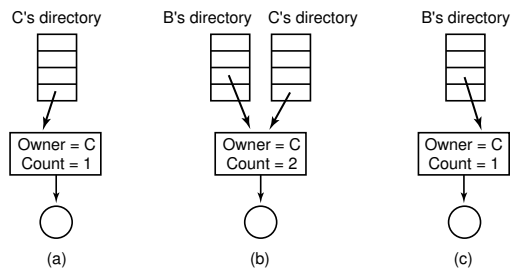
1  #include <sys/stat.h>
2  #include <sys/types.h>
3  #include <unistd.h>
4  #include <stdio.h>
5
6  int main(int argc, char *argv[])
7  {
8      char s[100];
9      if ( mkdir(argv[1], S_IRUSR|S_IXUSR) == 0 )
10         chdir(argv[1]);
11     printf("PWD = %s\n", getcwd(s,100));
12     rmdir(argv[1]);
13     return 0;
14 }
```

Hard Links

Hard links ← the same inode

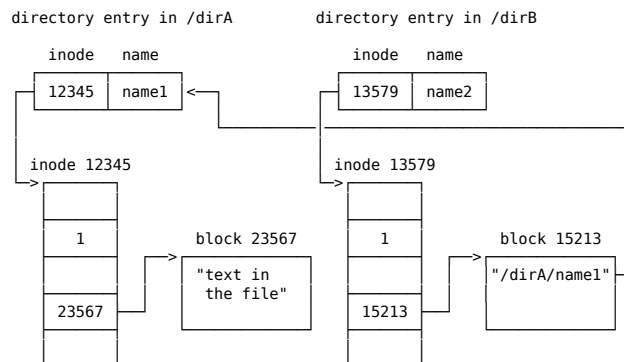


Drawback



Symbolic Links

A symbolic link has its own inode ➡ a directory entry



Fast symbolic link: Short path name (< 60 chars) needs no data block. Can be stored in the 15 pointer fields

`link()`, `unlink()`, `symlink()`

```

1  #include <unistd.h>
2  #include <stdio.h>
3
4  int main(int argc, char *argv[])
5  {
6      link(argv[1], argv[2]);
7      perror(argv[0]);
8      return 0;
9  }
10
11 /* symlink(argv[1], argv[2]); */
12 /* unlink(argv[1]); */

```

7 Processes and signals

Process

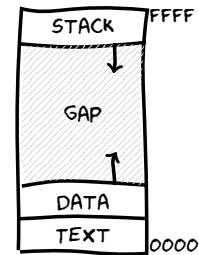
A **process** is an instance of a program in execution

Processes are like human beings:

- they are generated
- they have a life
- they optionally generate one or more child processes, and
- eventually they die

A small difference:


- sex is not really common among processes
- each process has just one parent



Process Control Block (PCB)

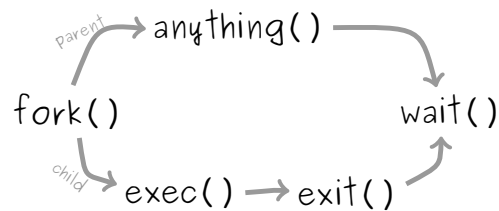
Implementation

A process is *the collection of data structures* that fully describes how far the execution of the program has progressed.

- Each process is represented by a *PCB*
- `task_struct` in 

process state
PID
program counter
registers
memory limits
list of open files
...

Process Creation



- When a process is created, it is almost identical to its parent
 - It receives a (logical) copy of the parent's address space, and
 - executes the same code as the parent
- The parent and child have separate copies of the data (stack and heap)

Forking in C

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main ()
5 {
6     printf("Hello World!\n");
7     fork();
8     printf("Goodbye Cruel World!\n");
9     return 0;
10 }
```

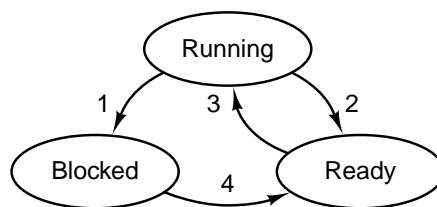
\$ man fork

`exec()`

```
1 int main()
2 {
3     pid_t pid;
4     /* fork another process */
5     pid = fork();
6     if (pid < 0) { /* error occurred */
7         fprintf(stderr, "Fork Failed");
8         exit(-1);
9     }
10    else if (pid == 0) { /* child process */
11        execlp("/bin/ls", "ls", NULL);
12    }
13    else { /* parent process */
14        /* wait for the child to complete */
15        wait(NULL);
16        printf ("Child Complete");
17        exit(0);
18    }
19    return 0;
20 }
```

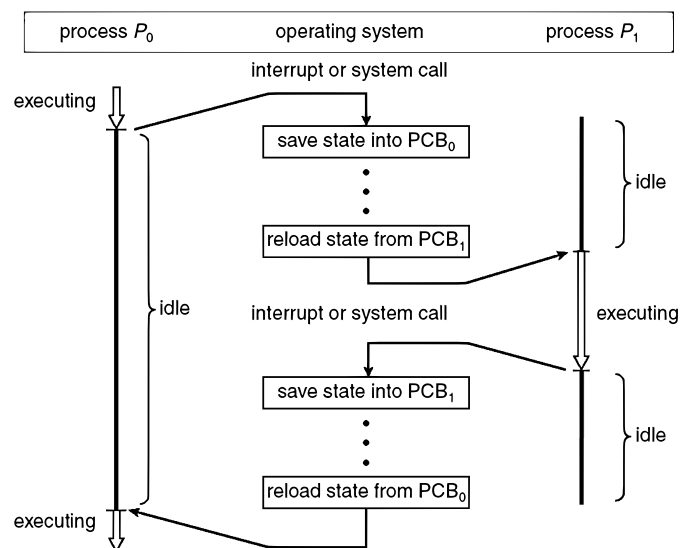
\$ man 3 exec

Process State Transition



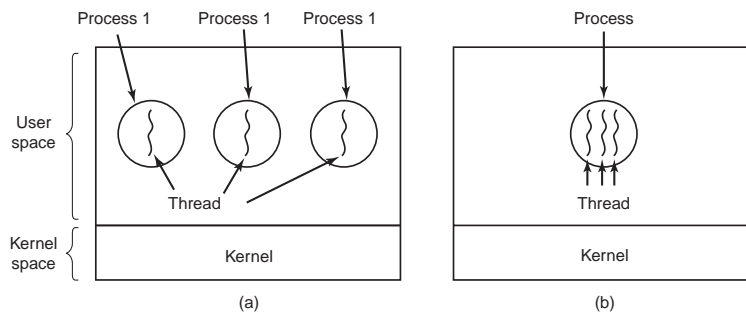
1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

CPU Switch From Process To Process



Process vs. Thread

- a single-threaded process = resource + execution
- a multi-threaded process = resource + executions



A process = a unit of resource ownership, used to group resources together;

A thread = a unit of scheduling, scheduled for execution on the CPU.

Threads

code, data, open files, signals...		
thread ID	thread ID	thread ID
program counter	program counter	program counter
register set	register set	register set
stack	stack	stack

POSIX Threads

IEEE 1003.1c The standard for writing portable threaded programs. The threads package it defines is called *Pthreads*, including over 60 function calls, supported by most UNIX systems.

Some of the Pthreads function calls

Thread call	Description
<code>pthread_create</code>	Create a new thread
<code>pthread_exit</code>	Terminate the calling thread
<code>pthread_join</code>	Wait for a specific thread to exit
<code>pthread_yield</code>	Release the CPU to let another thread run
<code>pthread_attr_init</code>	Create and initialize a thread's attribute structure
<code>pthread_attr_destroy</code>	Remove a thread's attribute structure

Pthreads

Example 1

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

void *thread_function(void *arg){
    int i;
    for( i=0; i<20; i++ ){
        printf("Thread says hi!\n");
        sleep(1);
    }
    return NULL;
}

int main(void){
    pthread_t mythread;
    if(pthread_create(&mythread, NULL, thread_function, NULL)){
        printf("error creating thread.");
        abort();
    }

    if(pthread_join(mythread, NULL)){
        printf("error joining thread.");
        abort();
    }

    exit(0);
}
```

Pthreads

`pthread_t` defined in `pthread.h`, is often called a "thread id" (`tid`);
`pthread_create()` returns zero on success and a non-zero value on failure;
`pthread_join()` returns zero on success and a non-zero value on failure;

How to use pthread?

- `#include <pthread.h>`
- ```
$ gcc thread1.c -o thread1 -pthread
$./thread1
```

## Pthreads

### Example 2

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 #define NUMBER_OF_THREADS 10
6
7 void *print_hello_world(void *tid)
8 {
9 /* prints the thread's identifier, then exits.*/
10 printf ("Thread %d: Hello World!\n", tid);
11 pthread_exit(NULL);
12 }
13
14 int main(int argc, char *argv[])
15 {
16 pthread_t threads[NUMBER_OF_THREADS];
17 int status, i;
18 for (i=0; i<NUMBER_OF_THREADS; i++)
19 {
20 printf ("Main: creating thread %d\n",i);
21 status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);
22
23 if(status != 0){
24 printf ("Oops. pthread_create returned error code %d\n",status);
25 exit(-1);
26 }
27 }
28 exit(NULL);
29 }
```

## Linux Threads

### To the Linux kernel, there is no concept of a thread

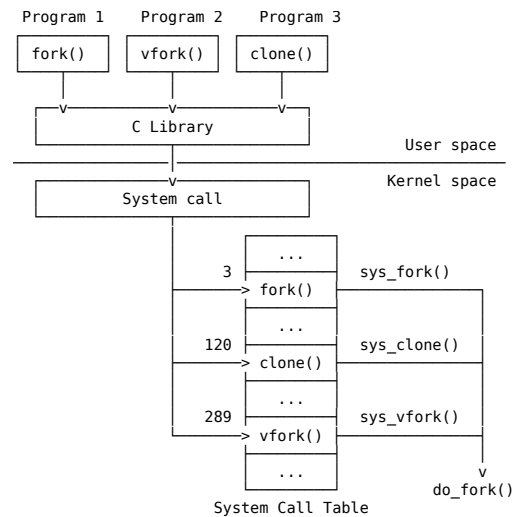
- Linux implements all threads as standard processes
- To Linux, a thread is merely a process that shares certain resources with other processes
- Some OS (MS Windows, Sun Solaris) have cheap threads and expensive processes.
- Linux processes are already quite lightweight

On a 75MHz Pentium      thread: 1.7 $\mu$ s  
                             fork: 1.8 $\mu$ s

## Linux Threads

`clone()` creates a separate process that shares the address space of the calling process. The cloned task behaves *much like* a separate thread.





`clone()`

```

1 #include <sched.h>
2 int clone(int (*fn) (void *), void *child_stack,
3 int flags, void *arg, ...);

```

**arg 1** the function to be executed, i.e. `fn(arg)`, which returns an `int`;

**arg 2** a pointer  $\blacktriangleright$  a (usually malloced) memory space to be used as the stack for the new thread;

**arg 3** a set of flags used to indicate how much the calling process is to be shared. In fact,

`clone(0) == fork()`

**arg 4** the arguments passed to the function.

It returns the PID of the child process or -1 on failure.

\$ man clone

## The `clone()` System Call

Some flags:

| flag          | Shared                |
|---------------|-----------------------|
| CLONE_FS      | File-system info      |
| CLONE_VM      | Same memory space     |
| CLONE_SIGHAND | Signal handlers       |
| CLONE_FILES   | The set of open files |

## In practice, one should try to avoid calling `clone()` directly

Instead, use a threading library (such as `pthread`) which use `clone()` when starting a thread (such as during a call to `pthread_create()`)

## `clone()` Example

```

1 #include <unistd.h>
2 #include <sched.h>
3 #include <sys/types.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <stdio.h>
7 #include <fcntl.h>
8
9 int variable;
10
11 int do_something()
12 {
13 variable = 42;
14 _exit(0);
15 }
16
17 int main(void)
18 {
19 void *child_stack;
20 variable = 9;
21
22 child_stack = (void *) malloc(16384);
23 printf("The variable was %d\n", variable);
24
25 clone(do_something, child_stack,
26 CLONE_FS | CLONE_VM | CLONE_FILES, NULL);
27 sleep(1);
28
29 printf("The variable is now %d\n", variable);
30 return 0;
31 }

```

## Stack Grows Downwards

```
child_stack = (void**)malloc(8192) + 8192/sizeof(*child_stack);
```

## 8 IPC

## 9 User Interface

### 9.1 Dialog, Zenity

### 9.2 Ncurses

### 9.3 GTK+

### 9.4 Qt

## 10 Terminal

- chap 62 of **Kerrisk:2010:LPI:1869911**
  - <http://tldp.org/HOWTO/Text-Terminal-HOWTO-16.html>
  - <http://pubs.opengroup.org/onlinepubs/9699919799/>
- \$ man infocmp

## 11 IDE

Makefile, git