GNU/Linux Application Programming

WANG Xiaolin

wx672ster+linux@gmail.com

July 31, 2019

Textbooks

- MATTHEW N, STONES R. Beginning linux programming. John Wiley & Sons, 2008.
- COOPER M. Advanced Bash Scripting Guide 5.3 Volume 1. Lulu.com, 2010.
- RAYMOND E S. The art of Unix programming. Addison-Wesley, 2003.
- STEVENS W R, RAGO S A. Advanced programming in the UNIX environment. Addison-Wesley, 2013.
- LOVE R. Linux System Programming: Talking Directly to the Kernel and C Library. O'Reilly Media, Inc., 2007.
- KERRISK M. The Linux Programming Interface: A Linux and UNIX System Programming Handbook. No Starch Press, 2010.
- BRYANT R E, O'HALLARON D R. Computer Systems: A Programmer's Perspective. 2nd ed. Addison-Wesley, 2010.

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

𝔼 Preferably in English

in stackoverflow style

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

Getting Started

Linux Programs

Where to find them? `/bin, /usr/bin, /usr/local/bin, ~/bin, …`

```
$ echo $PATH
```

How to find them? `which, whereis, type`

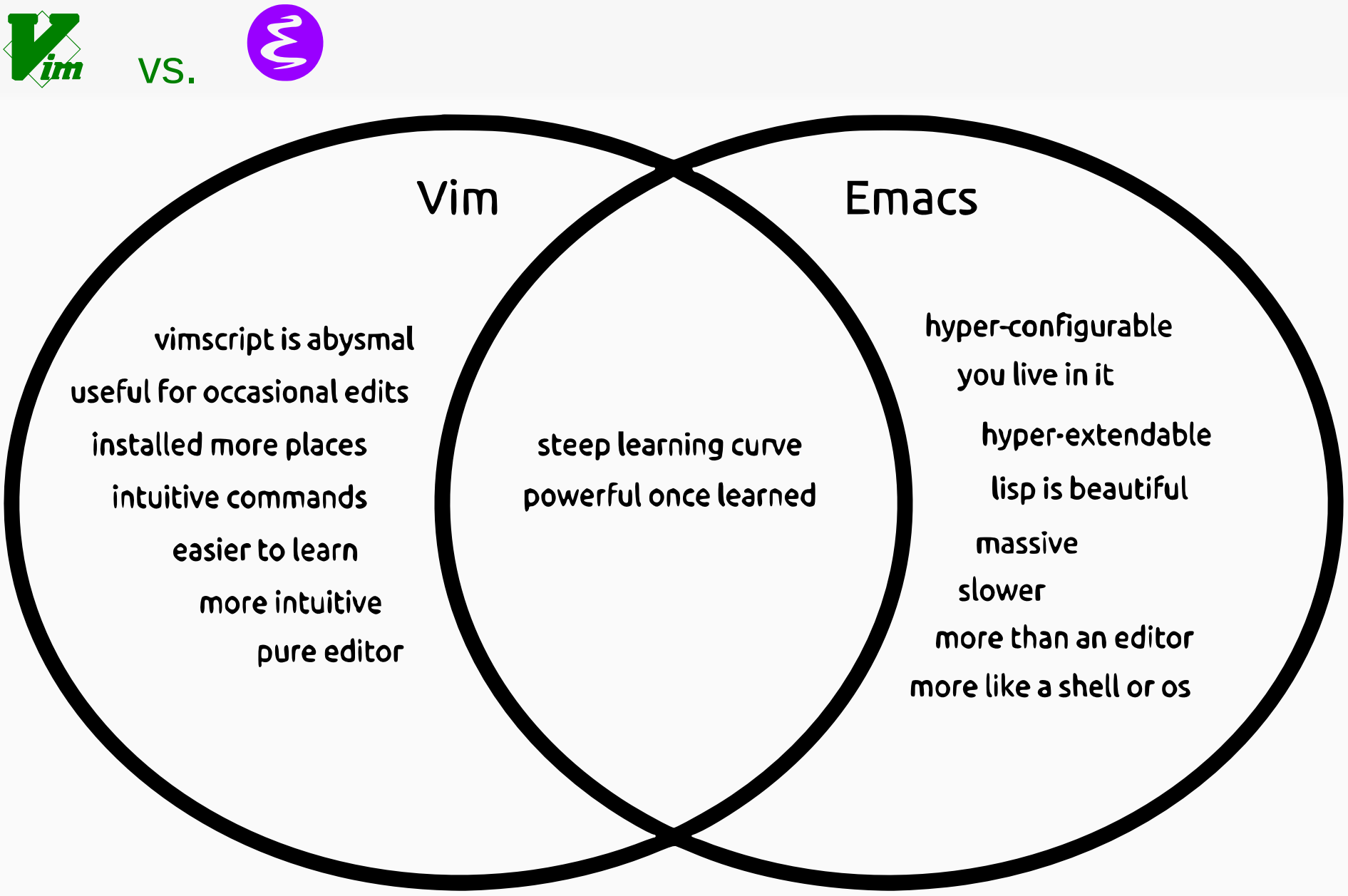
Command not found?

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

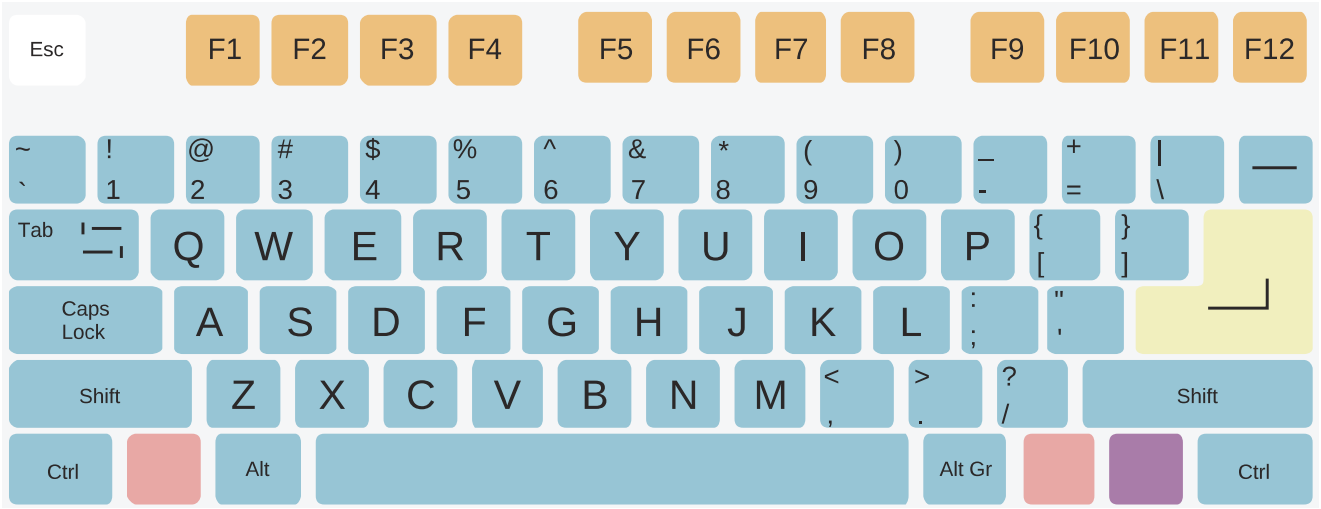


Help Your Editor

Suffix matters

```
$ vim ✘
$ vim hello.c ✔
$ vim hello.py ✔
$ emacs ✘
$ emacsclient hello.c ✔
$ emacsclient hello.py ✔
```

Keyboard



- vimtutor
- C-h t

Compiler vs. Interpreter

hello.c

```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
```

Compilation

Editor (emacs)

$ emacs hello.c

C source code (hello.c)

Include files, macros (stdio.h)

C Preprocessor (cpp)

$ cpp hello.c
$ gcc -E hello.c

Extended C source code (hello.c)

Compiler (gcc)

$ cc1 hello.i -o hello.s
$ gcc -S hello.c

ASM source code (hello.s)

Assembler (gas)

$ as hello.s -o hello.o
$ gcc -c hello.c

Object code (hello.o)

Libraries (printf)

Linker (ld)

$ ld hello.o LIBs
$ gcc -o hello hello.c

Executable program (hello)

Debugger (gdb)

$ gcc -g hello.c -o hello
$ gdb hello

Header Files

Why?

```c
#include "add.h"

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

Why not?

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

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

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

In the header files…

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

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

Library Files

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

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

Shared libraries `.so` files. The preferred ones.

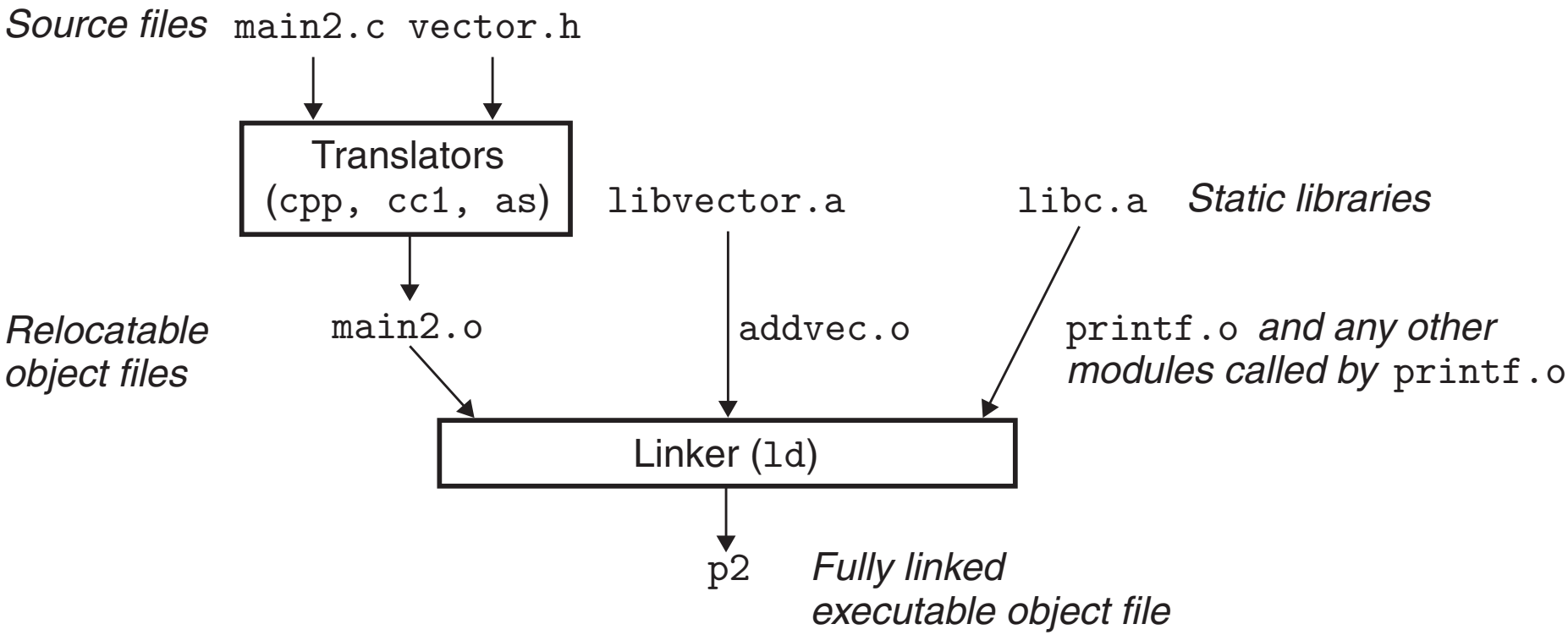
```
$ find /usr/lib -name "*.so.*"
```

Examples:

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

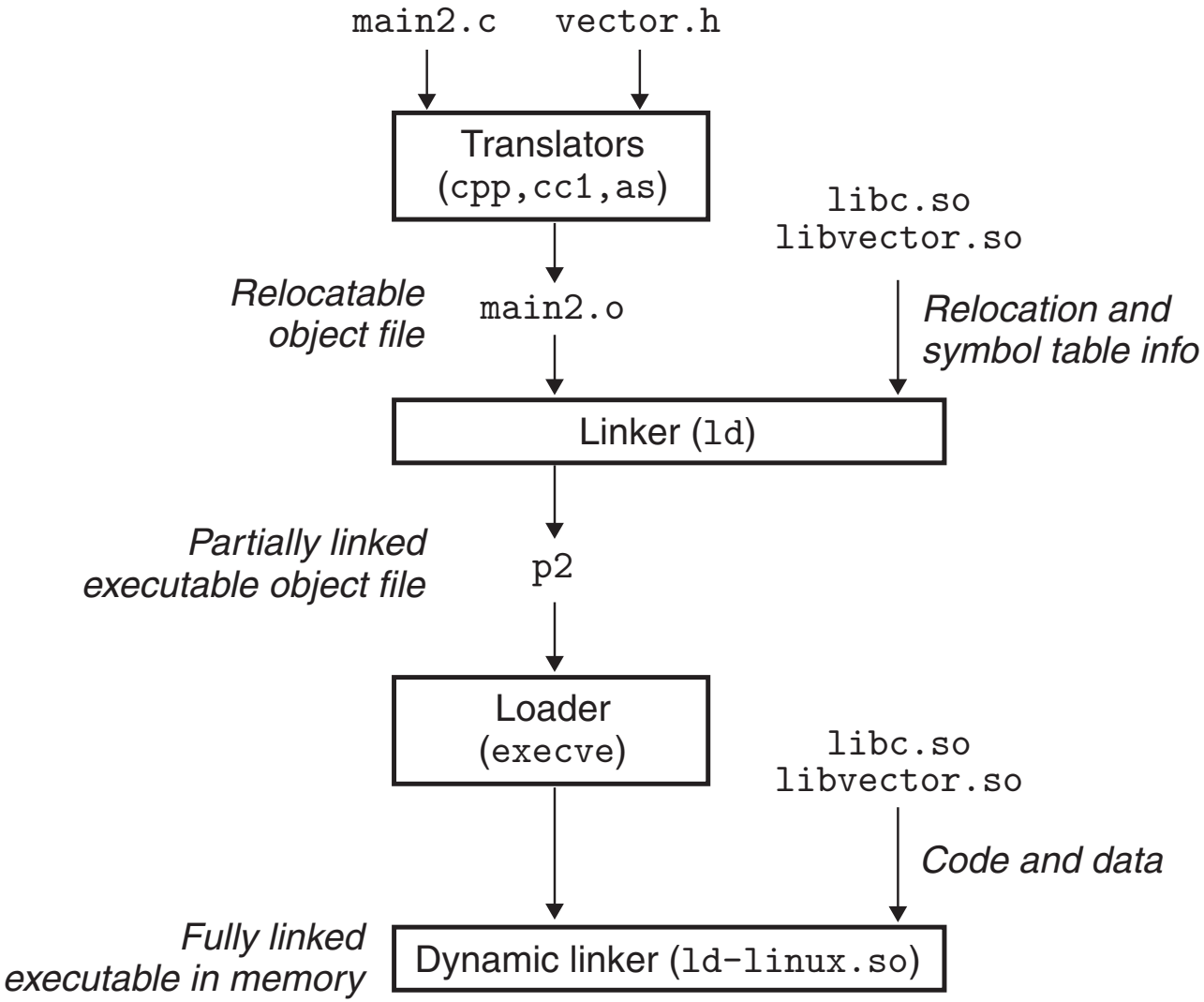
Static Linking

- 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

```c
#include "lib.h"

int main(int argc, char* argv[])
{
  int i=1;

  for (; i<argc; i++)
    {
      hello(argv[i]);
      hi(argv[i]);
    }
  return 0;
}
```

lib.h

```c
#include <stdio.h>

void hello(char *);
void hi(char *);
```

hello.c

```c
#include <stdio.h>

void hello(char *arg)
{
  printf("Hello, %s!\n", arg);
}
```

hi.c

```c
#include <stdio.h>

void hi(char *arg)
{
  printf("Hi, %s!\n", arg);
}
```

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

```c
#include "hello.h"

int main(int argc, char *argv[])
{
  if (argc != 2)
    printf ("Usage: %s needs an argument.\n", argv[0]);
  else
    hi(argv[1]);
  return 0;
}
```

hello.h

```c
#include <stdio.h>
#include <stdlib.h>

int hi(char*);
```

hi.c

```c
#include "hello.h"

int hi(char* s)
{
  printf ("Hi, %s\n",s);
  return 0;
}
```

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
```

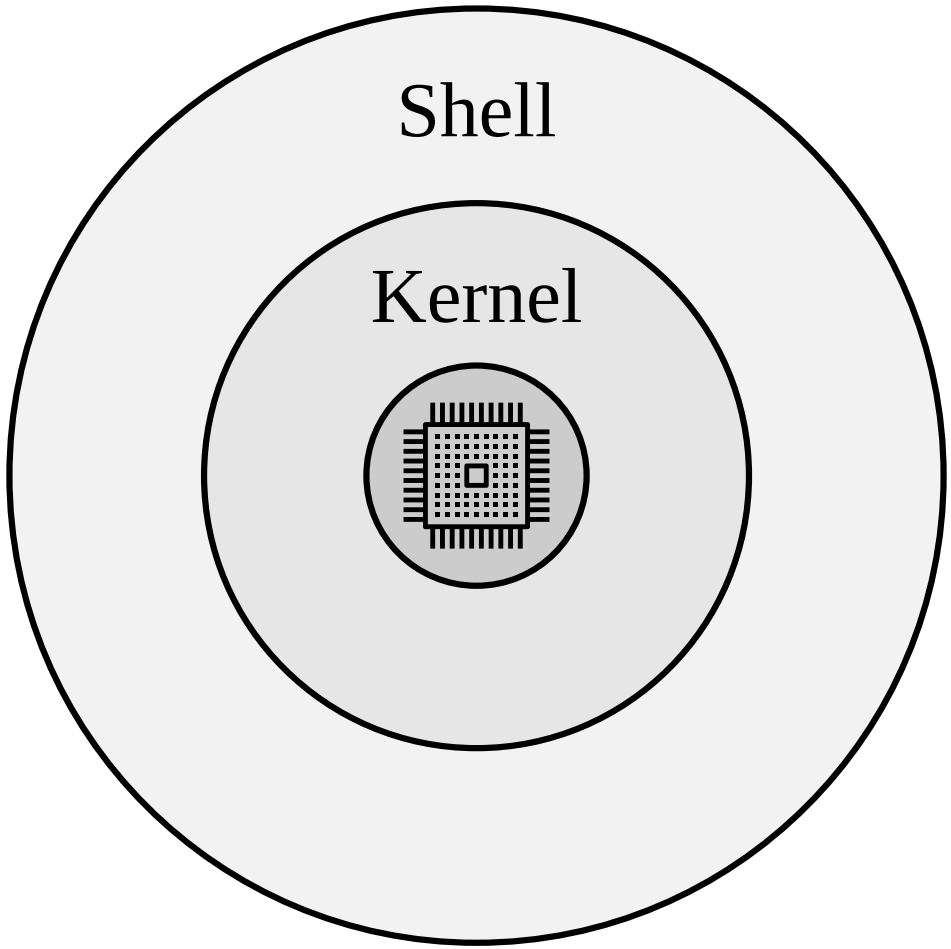
Getting Help

```
sudo apt install gcc-doc
$ man gcc
$ info gcc
```

Shell Basics

Shell

- A command line interpreter
- A programming language



Redirection

Redirecting output

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

Redirecting input

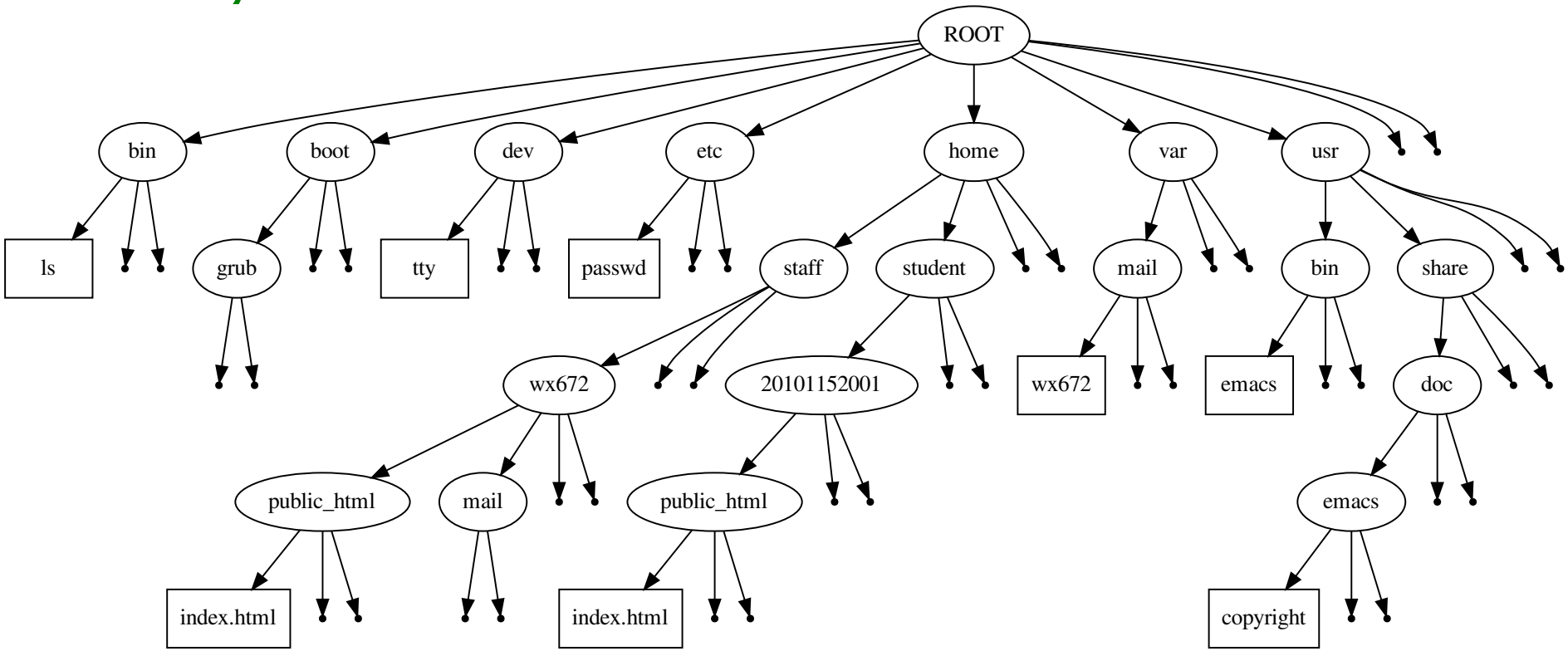
```
$ more < output.txt
```

Pipe

Chain processes together

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

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, &`	forground?	`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

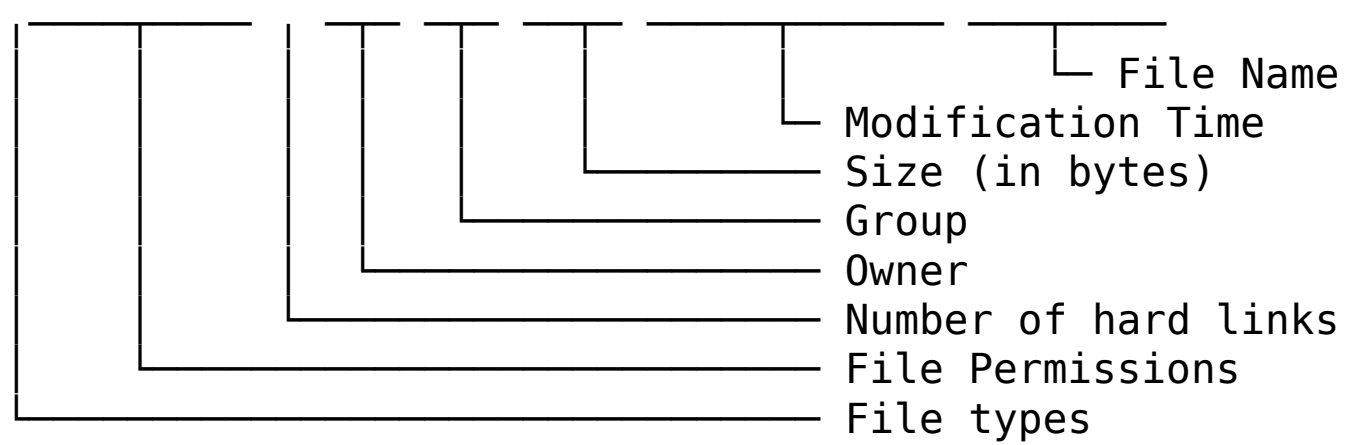
`Ctrl-a:`	beginning of line	`Ctrl-e:`	end of line
`Ctrl-f:`	forward	`Ctrl-b:`	backward
`Ctrl-n:`	next	`Ctrl-p:`	previous
`Ctrl-r:`	reverse search	`Ctrl-u:`	cut to beginning
`Ctrl-k:`	kill (cut to end)	`Ctrl-y:`	yank (paste)
`Ctrl-d:`	delete a character	`TAB:`	completion

Tmux

`C-a c:`	create window	`C-a C-a:`	switch window
`C-a n:`	next window	`C-a p:`	previous window
`C-a -:`	split window	`C-a |:`	split widnow
`C-a j:`	go down	`C-a k:`	go up
`C-a l:`	go right	`C-a h:`	go left

Understanding "`ls -l`"

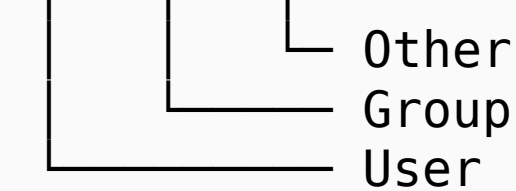
```
-rw------- 1 sam sam   57 Apr 17  1998 weather.txt
drwxr-xr-x 6 sam sam  102 Oct  9  1999 web_page
-rw-rw-r-- 1 sam sam 7648 Feb 11 20:41 web_site.tar
-rw------- 1 sam sam  574 Dec 16  1998 file.txt
```



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
```




```
$ chmod 755 foo    $ chmod 644 foo
$ chmod 000 foo    $ chmod 777 foo
$ chmod a-r foo    $ chmod u+x foo
$ chmod g+w 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
```

Shell Programming

Shell Programming

why? handy, powerful

which? bash

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, …, $@, $#`

```bash
#!/bin/bash

echo "You said:"

echo -e "\t$@"
echo
echo -e "\targc = $#"
echo -e "\targv[0] = $0"

i=1
for arg in $@; do
  echo -e "\targv[$i] = $arg"
  let i++
done
```

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  int i;
  printf("You said:\n\t");

  for(i=1; i<argc; i++)
    printf("%s ",argv[i]);

  printf("\n\n\targc = %d\n", argc);

  for(i=0; i<argc; i++)
    printf("\targv[%d] = %s\n",i,argv[i]);

  return 0;
}
```

Parameter Substitution

Default value

```
$ echo ${s:=abc}
$ echo ${s:=xyz}
```

```
$ echo ${v:-8}
$ 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 I

```
$ (( 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 ✔
```

Tests II

```
$ [[ 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
```

Tests III

```
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<=0; i++, j++)); do
  echo $i-$j
  echo $(($i-$j))
  done
$ for ((i=1; i<=10; i++)) { echo $i; }
$ 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

```
#!/bin/bash

[ -z "$1" ] && echo "Usage: `basename $0` [dhb]<number>" && exit 0;

case "$1" in
        [dD]*)
                NUM=$(echo $1 | cut -b 2-)
                printf "\tDec\tHex\tBin\n"
                printf "\t%d\t0x%02X\t%s\n" $NUM $NUM $(bc <<< "obase=2;$NUM")
                ;;
        [hH]*)
                NUM=$(echo $1 | cut -b 2-)
                NUM=$(echo $NUM | tr [:lower:] [:upper:])
                printf "\tHex\t\tDec\t\tBin\n"
                printf "\t0x%s\t\t%s\t\t%s\n" $NUM $(bc <<< "ibase=16;obase=A;$NUM") \
                        $(bc <<< "ibase=16;obase=2;$NUM")
                ;;
        0[xX]*)
                NUM=$(echo $1 | cut -b 3-)
                NUM=$(echo $NUM | tr [:lower:] [:upper:])
                printf "\tHex\t\tDec\t\tBin\n"
                printf "\t0x%s\t\t%s\t\t%s\n" $NUM $(bc <<< "ibase=16;obase=A;$NUM") \
                        $(bc <<< "ibase=16;obase=2;$NUM")
                ;;
        [bB]*)
                NUM=$(echo $1 | cut -b 2-)
                printf "\tBin\t\tHex\t\tDec\n"
                printf "\t%s\t\t0x%s\t\t%s\n" $NUM $(bc <<< "ibase=2;obase=10000;$NUM") \
                        $(bc <<< "ibase=2;obase=1010;$NUM")
                ;;
            *)
                printf "Dec\tHex\tBin\n"
                printf "%d\t0x%08X\t%08d\n" $1 $1 $(bc <<< "obase=2;$1")
                ;;
esac
```

select

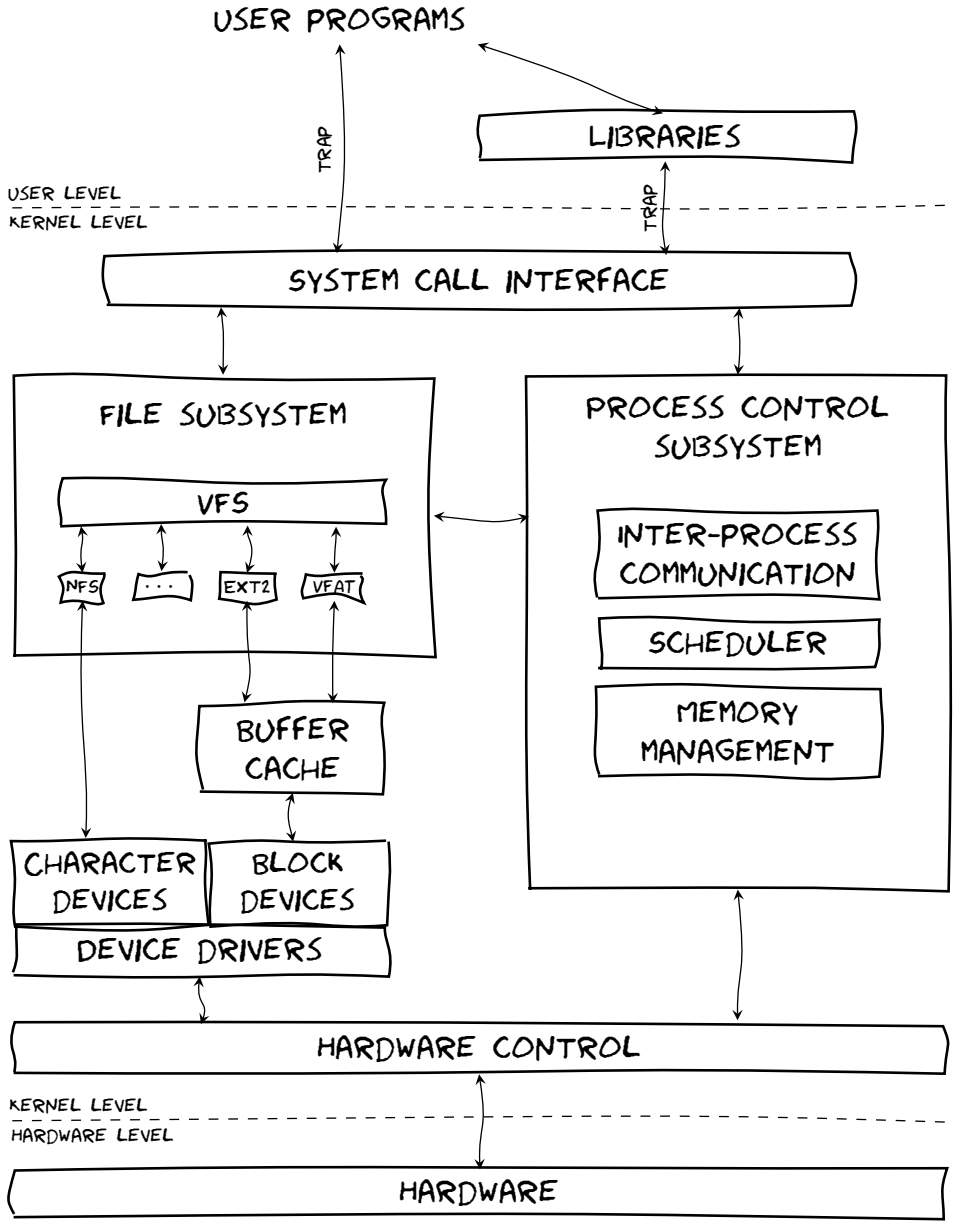
```
1  #!/bin/bash
2
3  PS3='Your favorite OS? '
4
5  select OS in "Linux" "Mac OSX" "Windows"
6  do
7     [[ "$OS" = "Linux" ]] && echo wise guy.
8     [[ "$OS" = "Mac OSX" ]] && echo rich guy.
9     [[ "$OS" = "Windows" ]] && echo patient guy.
10    break
11 done
```

Functions

```
#!/bin/bash

function screencapture(){
  ffmpeg -f x11grab -s 1920x1080 -r 30 -i :0.0 \
         -c:v libx264 -crf 0 -preset ultrafast screen.mkv
}

w2pdf(){
  libreoffice --convert-to pdf:writer_pdf_Export "$1"
}

rfc(){
    [[ -n "$1" ]] || {
                cat <<EOF
  rfc - Command line RFC viewer
  Usage: rfc <index>
EOF
                return 1
        }
        find /usr/share/doc/RFC/ -type f -iname "rfc$1.*" | xargs less
}
```

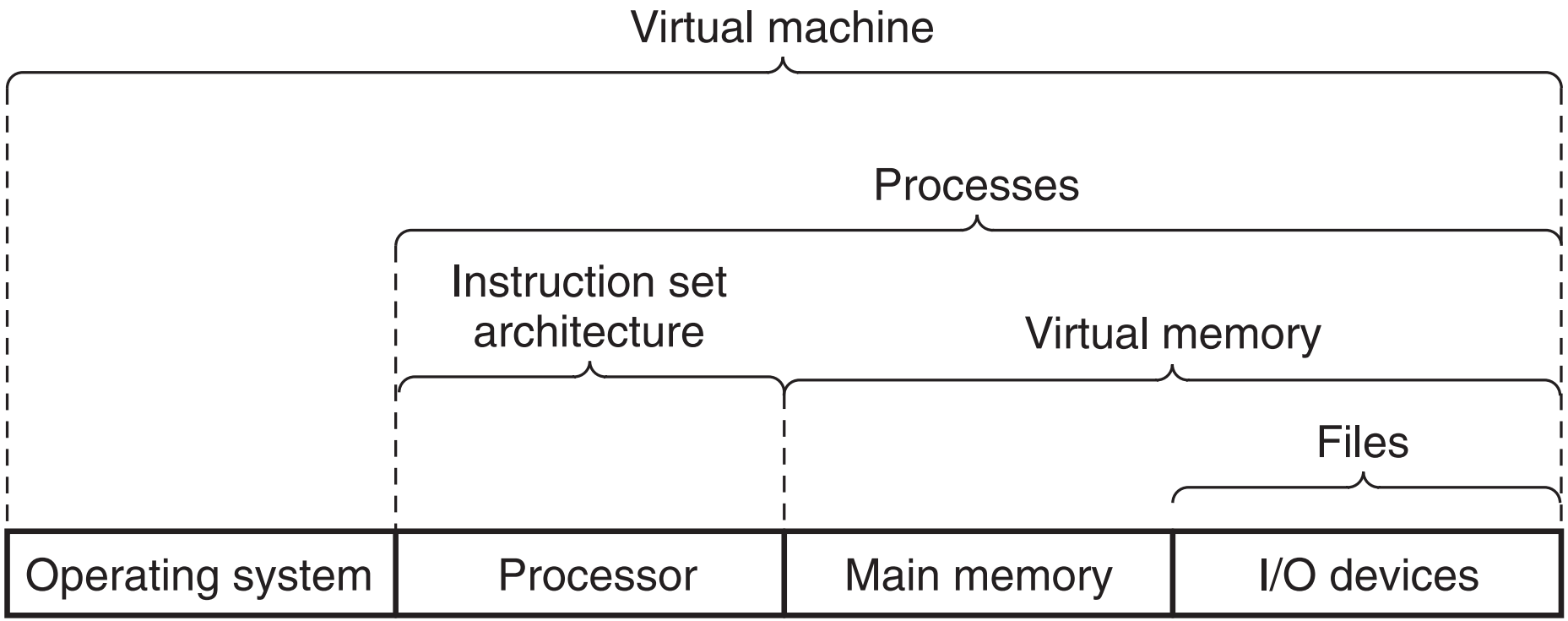
OS Basics

What's in the OS?

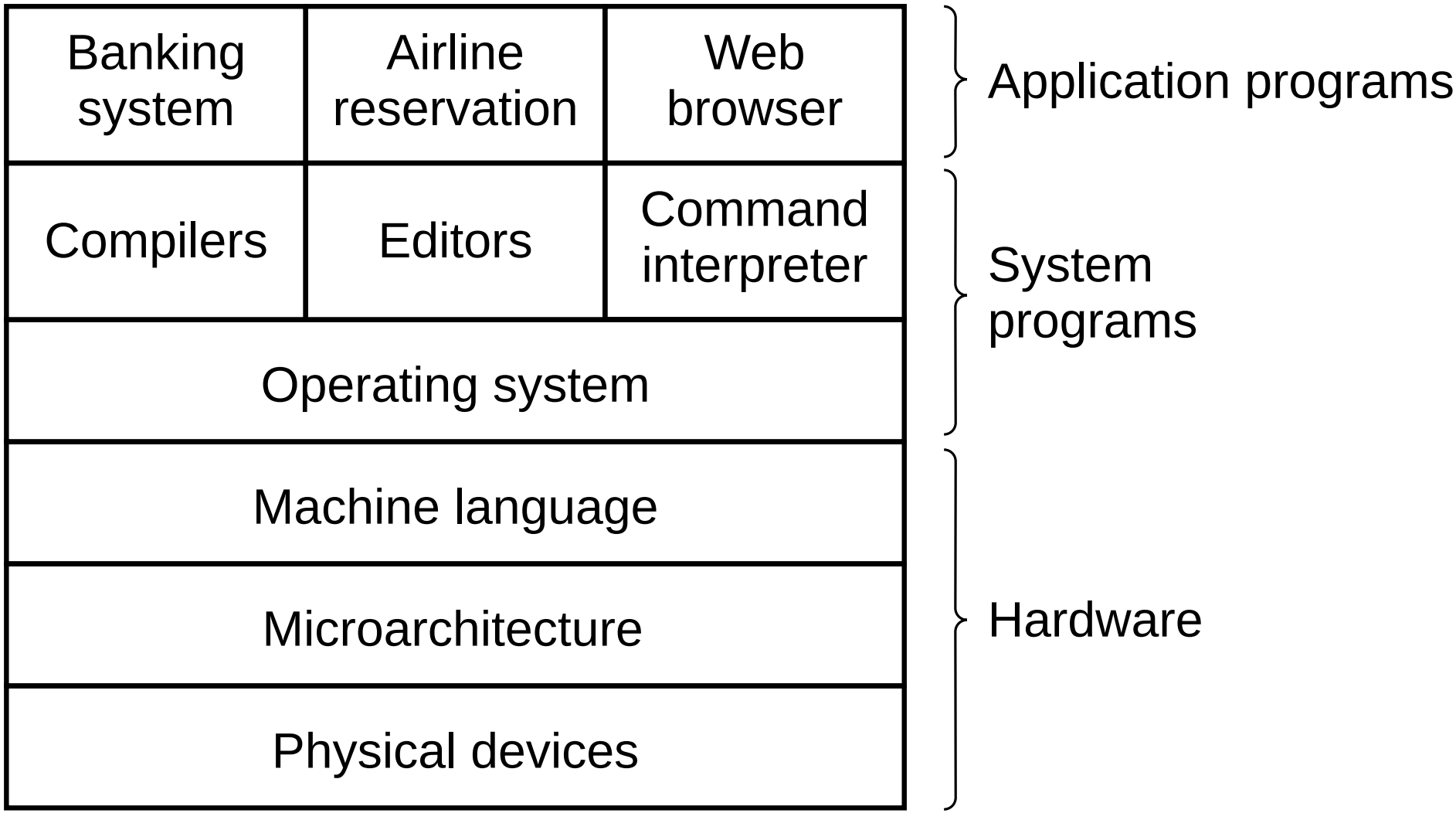


Abstractions

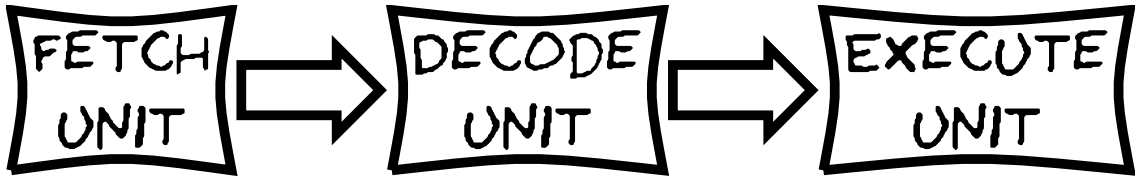
To hide the complexity of the actual implementations



A Computer System



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

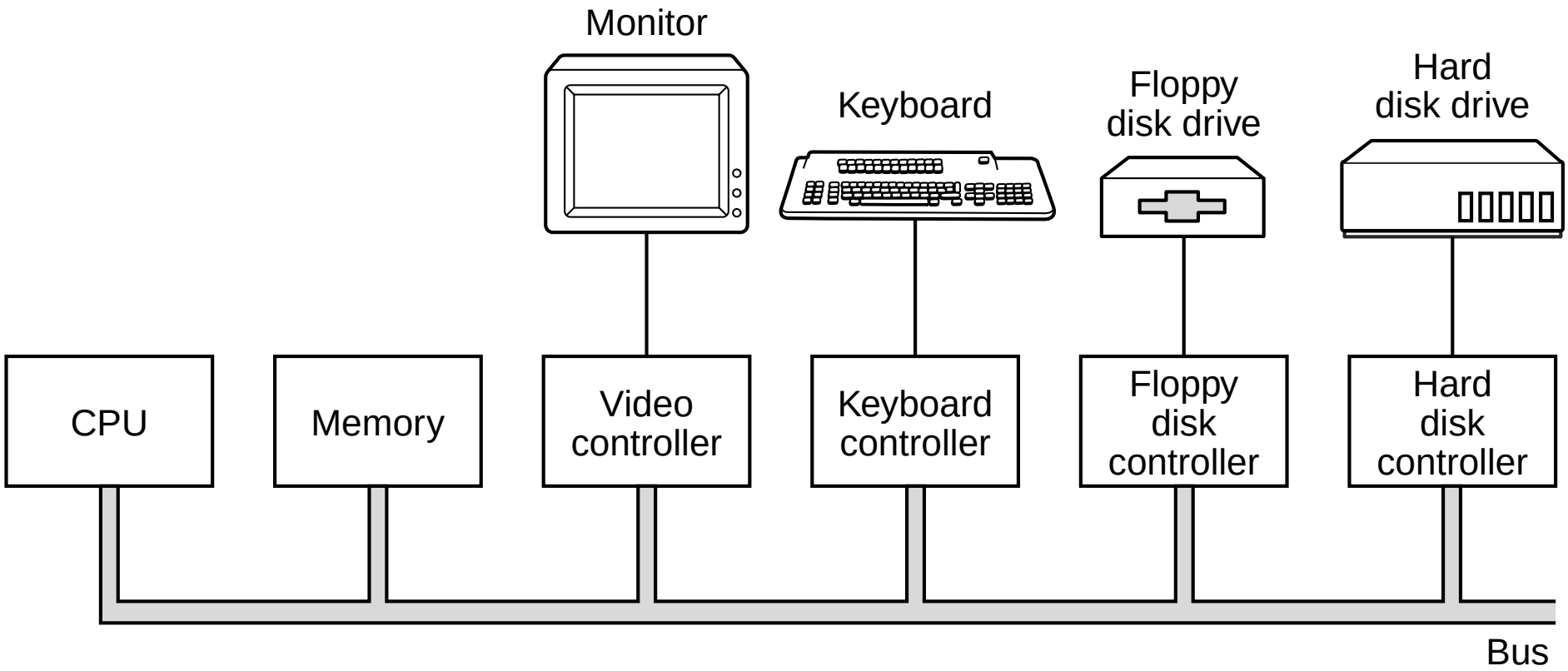
Program counter (PC): keeps the memory address of the next instruction to be fetched

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

Program status (PS): holds

- condition code bits
- processor state

System Bus



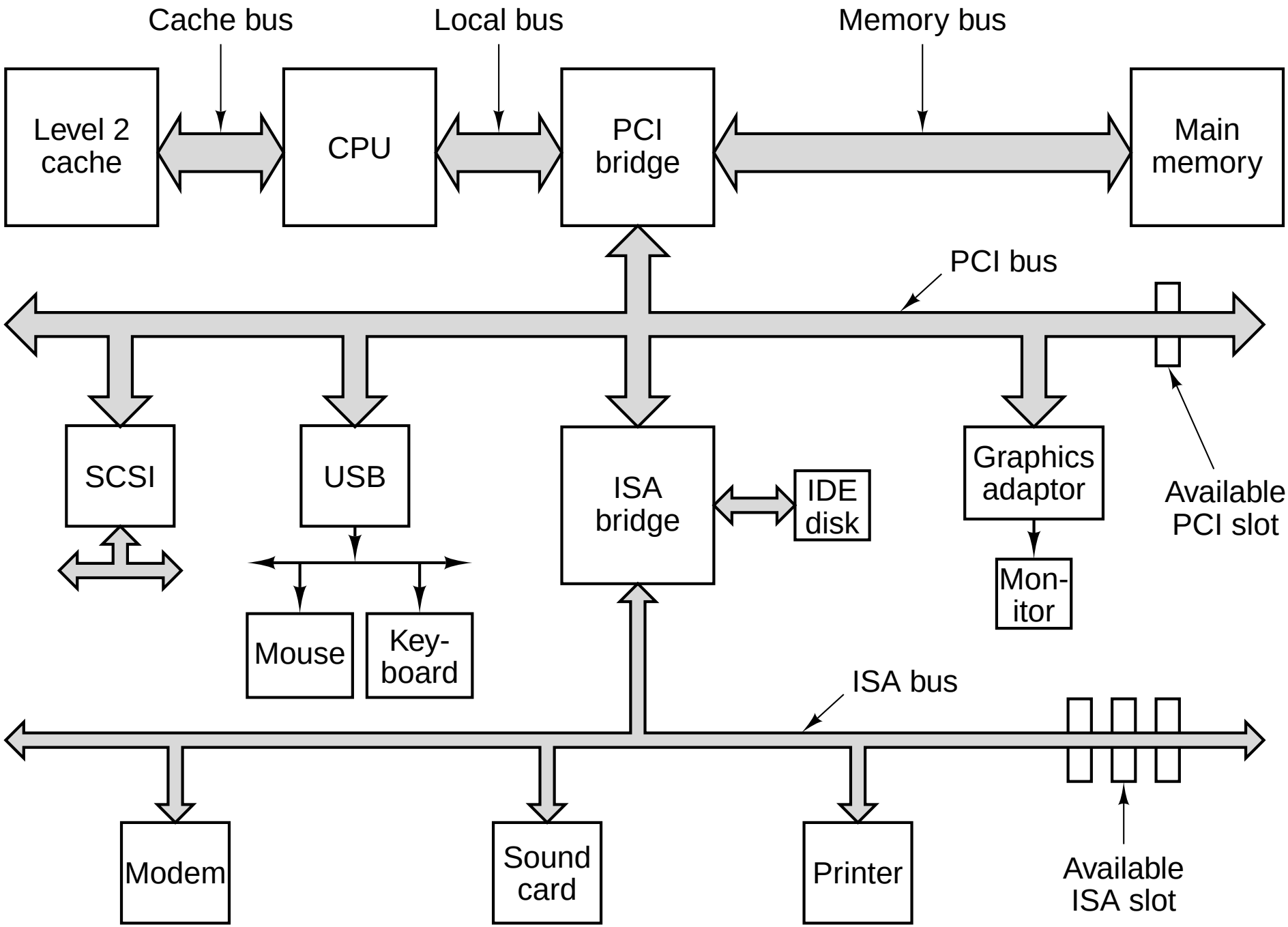
Address Bus: specifies the memory locations (addresses) for the data transfers

Data Bus: holds the data transfered. Bidirectional

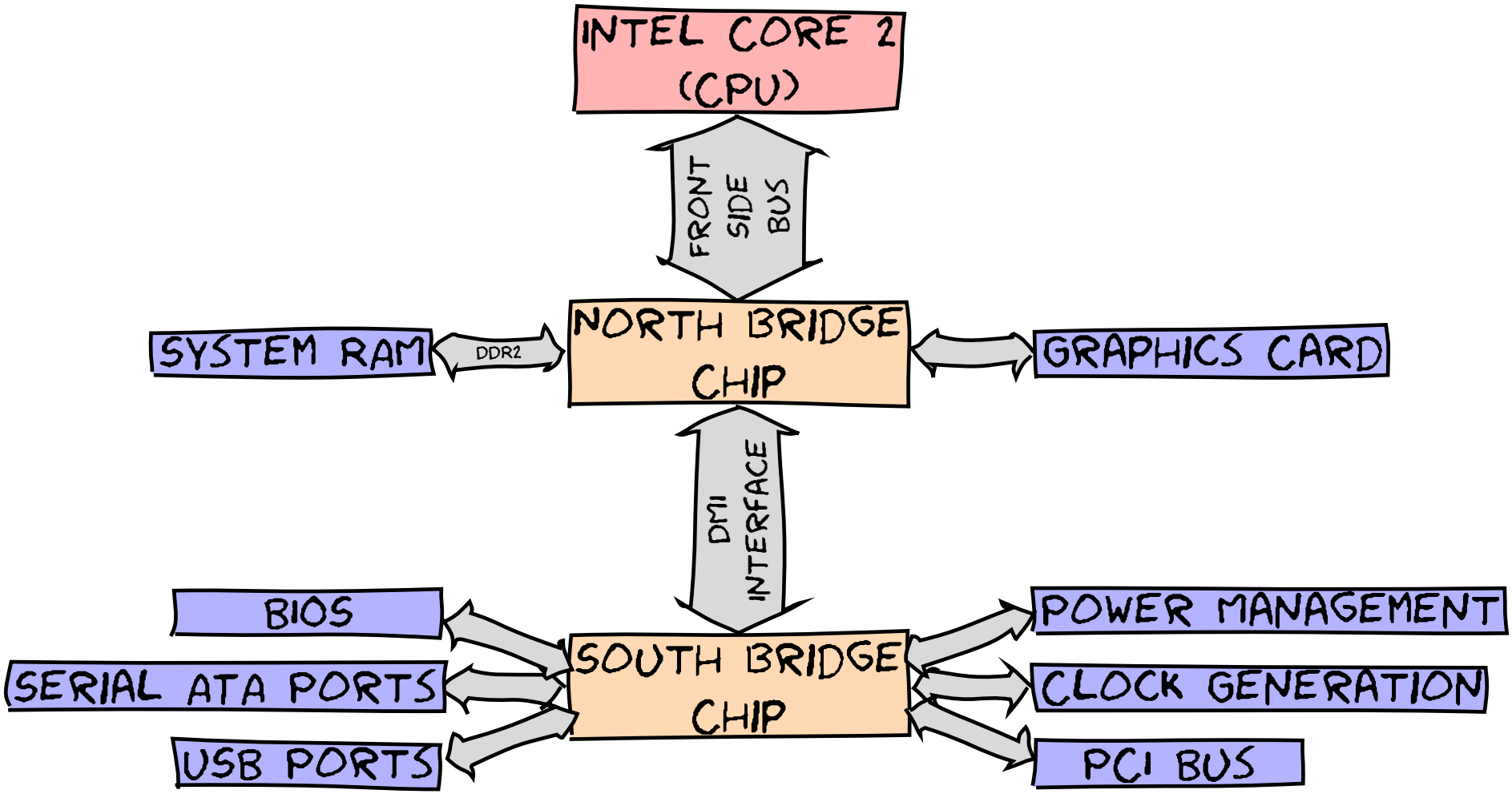
Control Bus: contains various lines used to route timing and control signals throughout the system

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


Cache bus
Local bus
Memory bus
Level 2 cache
CPU
PCI bridge
Main memory
PCI bus
SCSI
USB
ISA bridge
IDE disk
Graphics adaptor
Available PCI slot
Mon-itor
Mouse
Key-board
ISA bus
Modem
Sound card
Printer
Available ISA slot

Motherboard Chipsets



- 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 ($2^{33} \times 8B$)

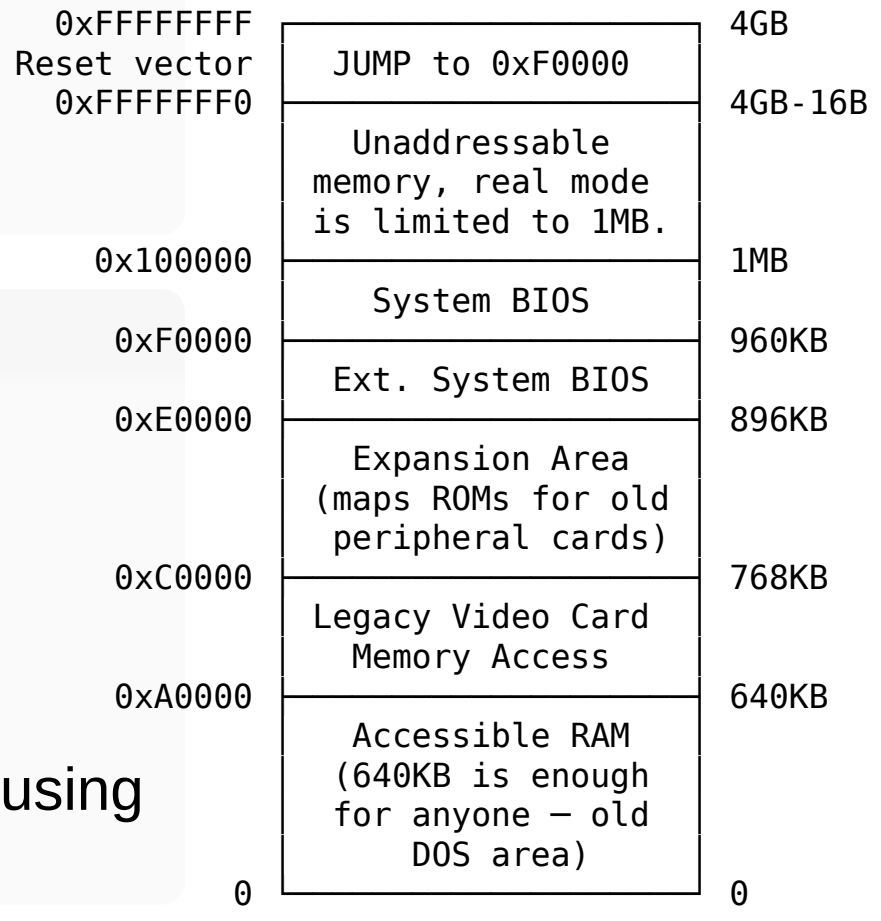
Some physical memory addresses are mapped away!

- 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.



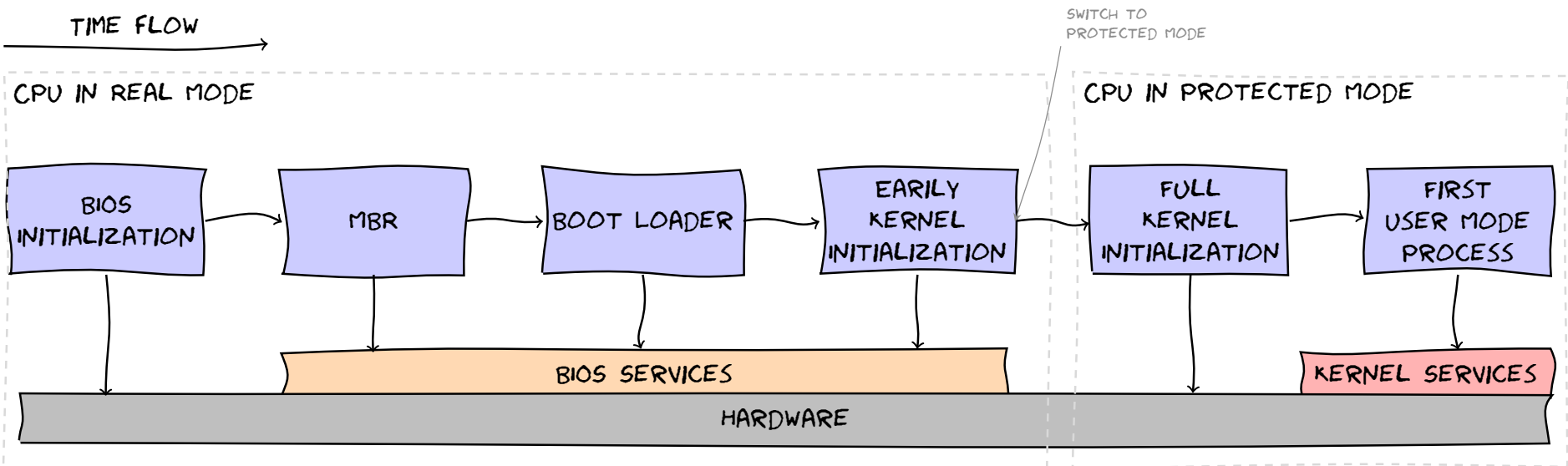
the northbridge

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

Bootstrapping

Can you pull yourself up by your own bootstraps?

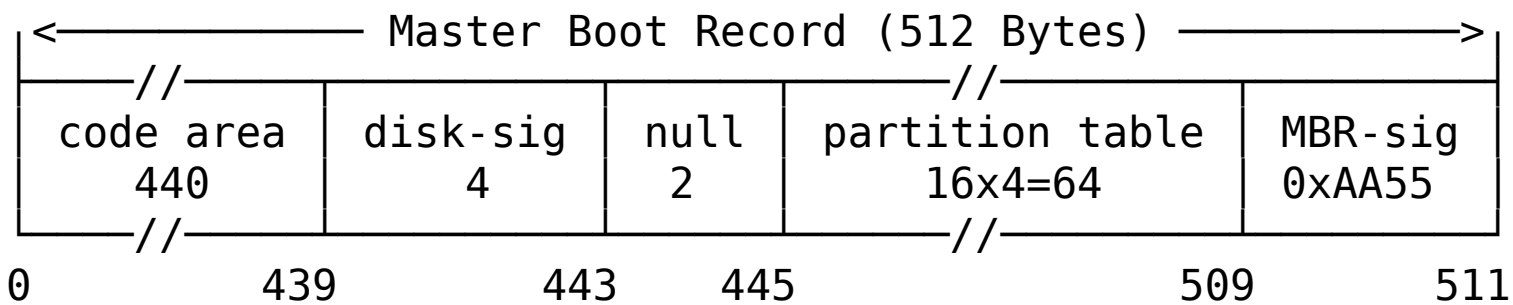
A computer cannot run without first loading software but must be running before any software can be loaded.



Intel x86 Bootstrapping

1. BIOS (`0xffffff0`)

   ➠ 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 ($16 \times 4 = 64\,B$)
   - 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
```

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

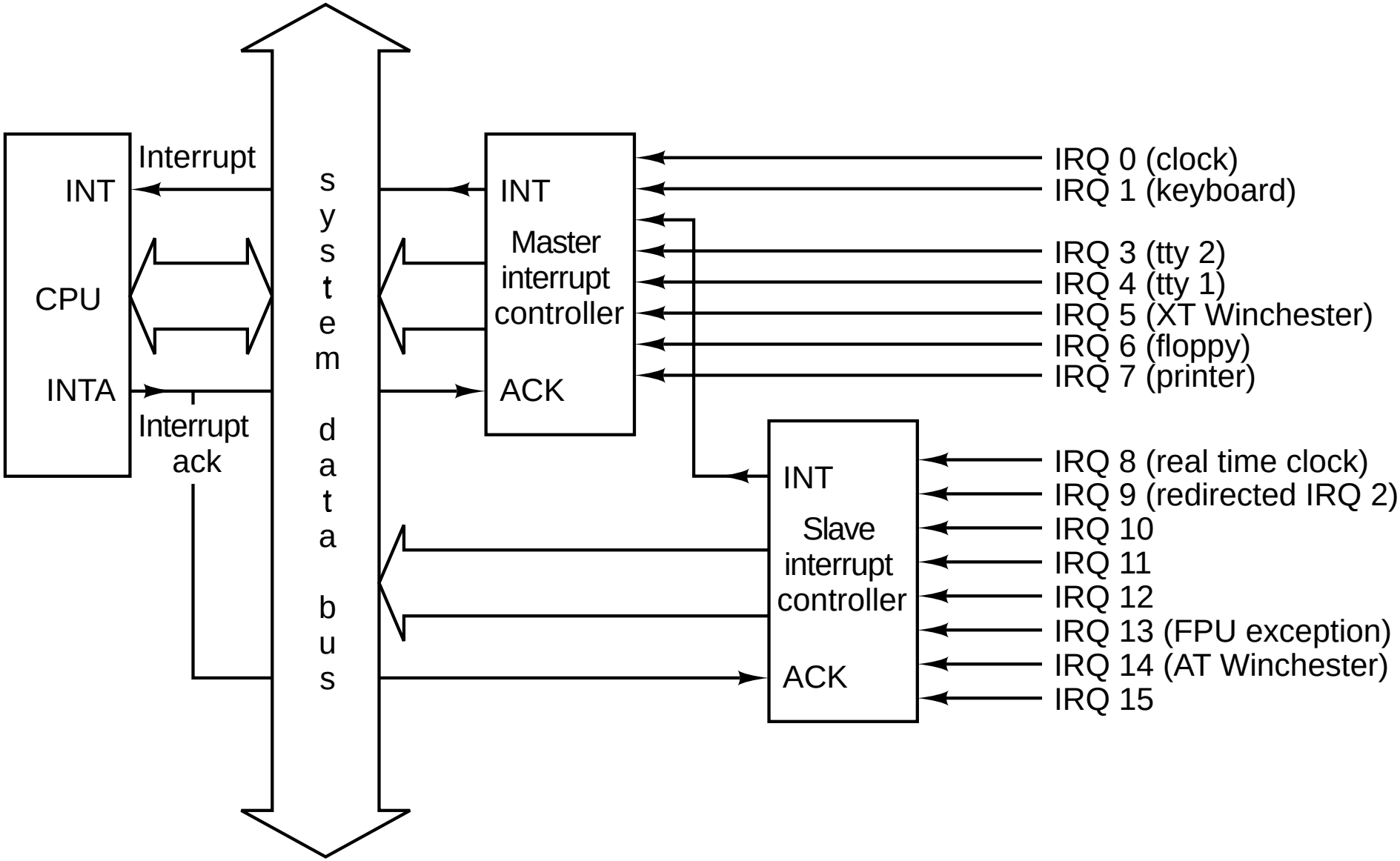
SW INT by executing a system call

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

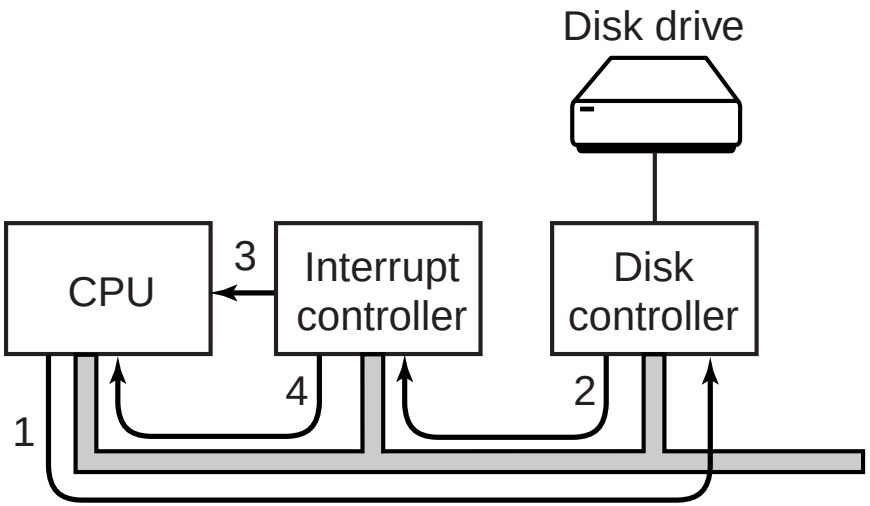
Interrupt vector is an array of pointers ☛ the memory addresses of interrupt handlers. This array is indexed by a unique device number

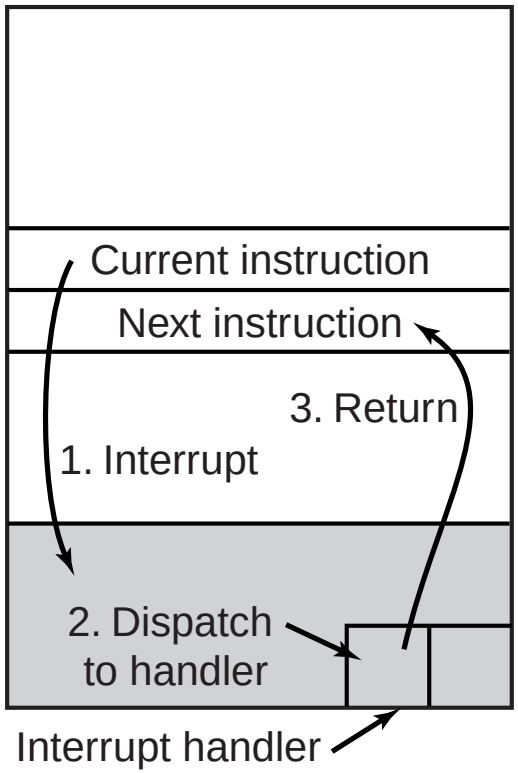
```
$ less /proc/devices
$ less /proc/interrupts
```

Programmable Interrupt Controllers

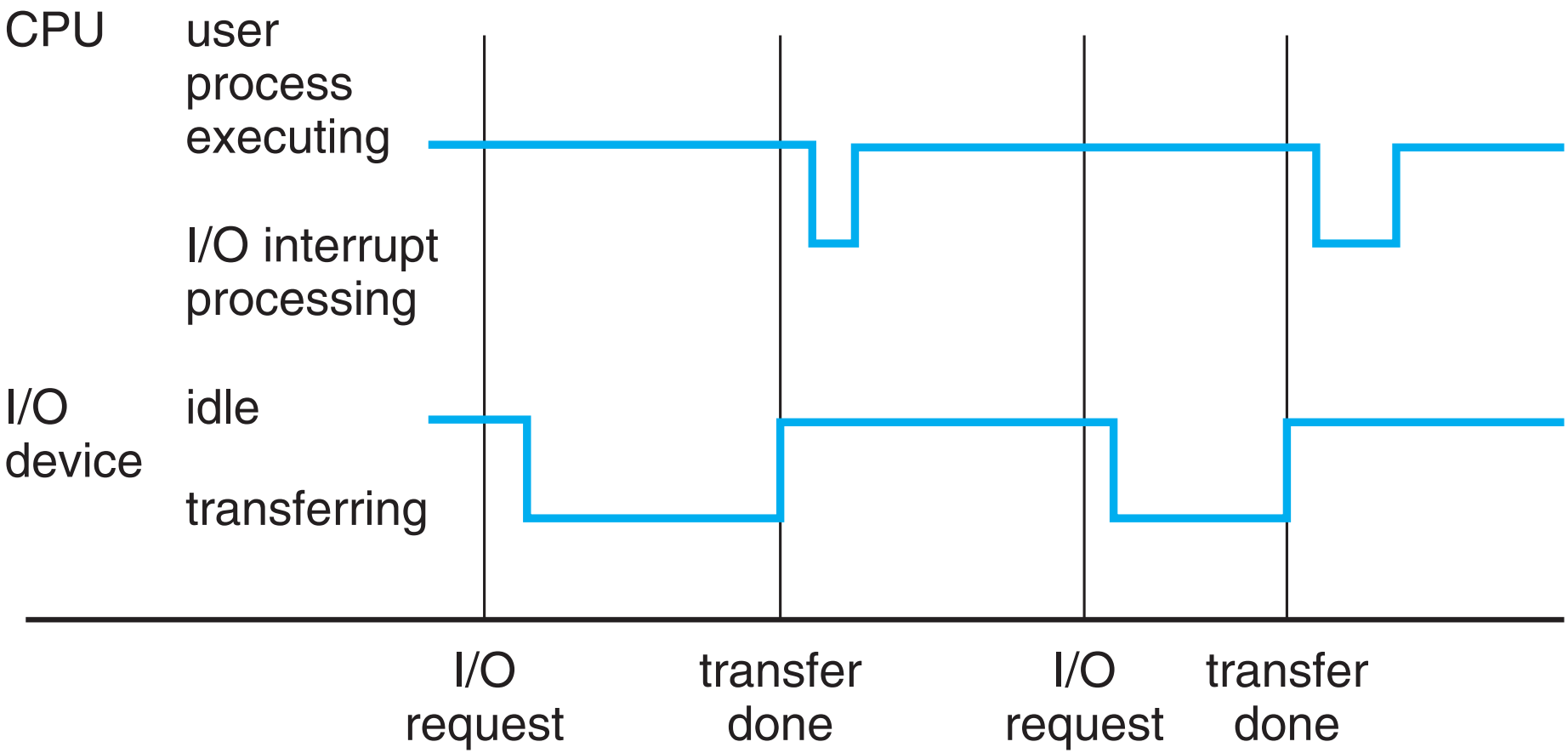


Interrupt Processing





Interrupt Timeline

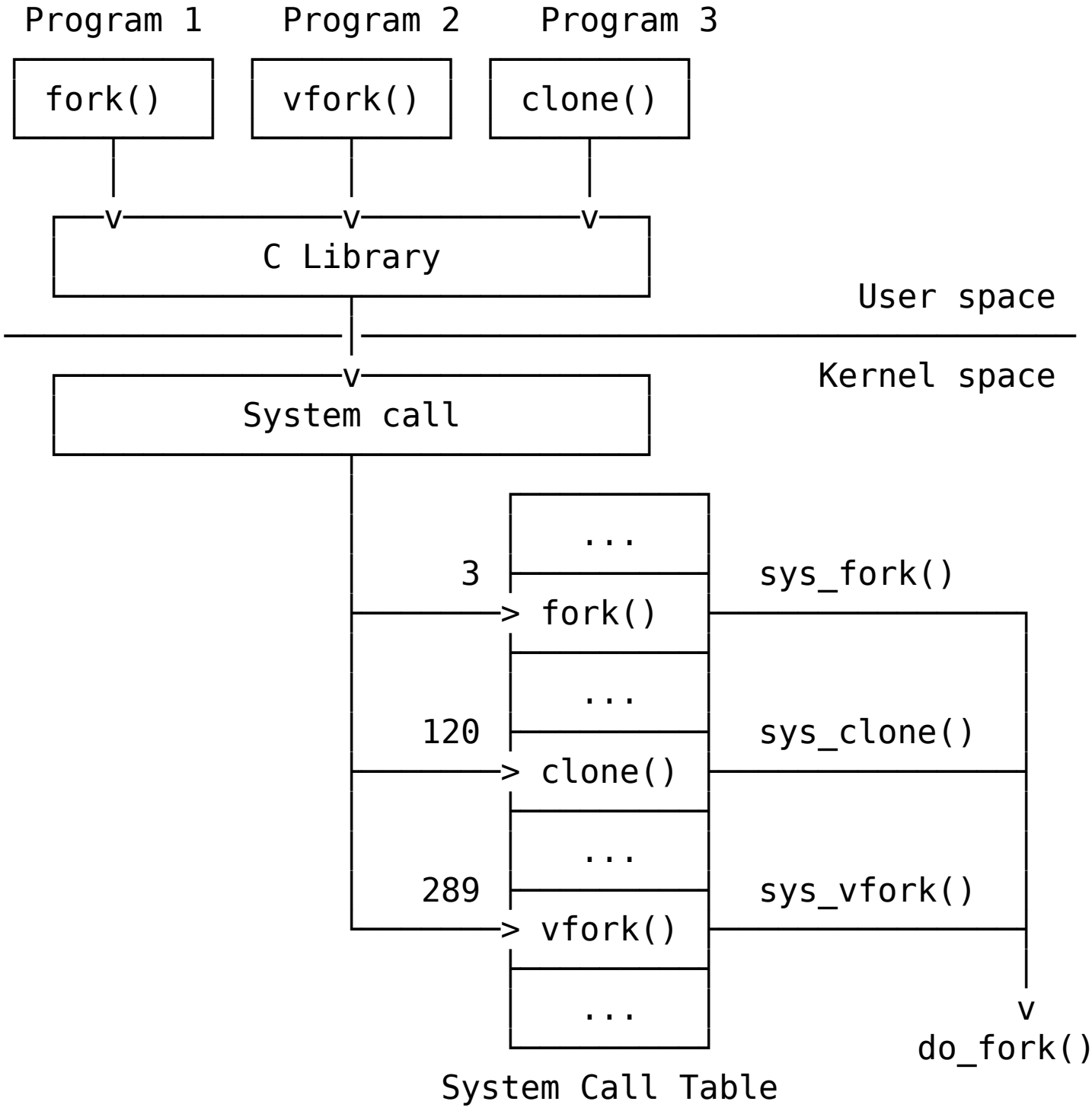


System Calls

A System Call

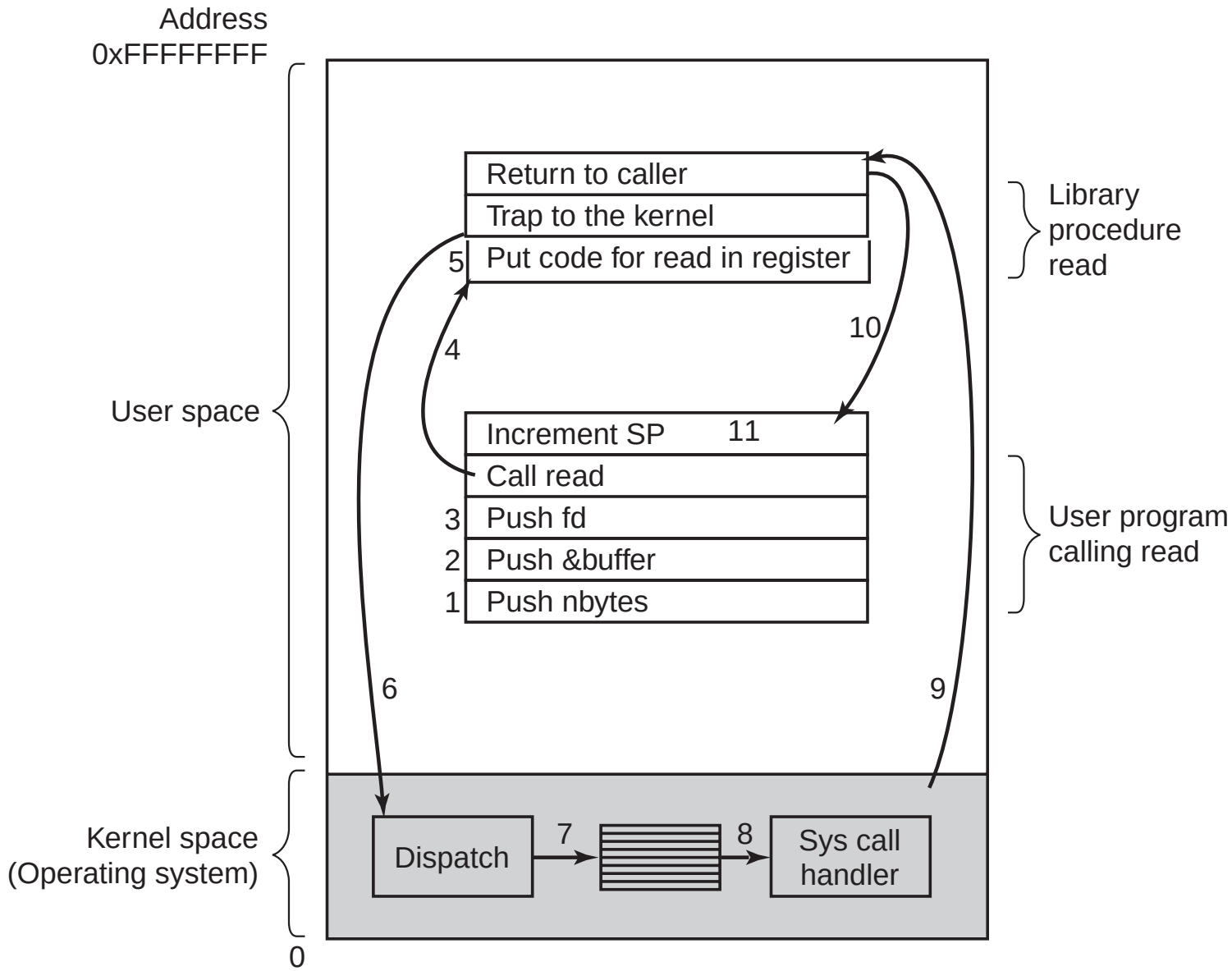
- is how a program requests a service from an OS kernel
- provides the interface between a process and the OS

```
$ man 2 intro
$ man 2 syscalls
```


Program 1
Program 2
Program 3
fork()
vfork()
clone()
C Library
User space
Kernel space
System call
...
3
sys_fork()
fork()
...
120
sys_clone()
clone()
...
289
sys_vfork()
vfork()
...
do_fork()
System Call Table

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

File management

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

Directory and file system management

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

Miscellaneous

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

System Call Examples

fork()

```c
#include <stdio.h>
#include <unistd.h>

int main()
{
  printf("Hello World!\n");
  fork();
  printf("Goodbye Cruel World!\n");
  return 0;
}
```

```
$ man 2 fork
```

exec()

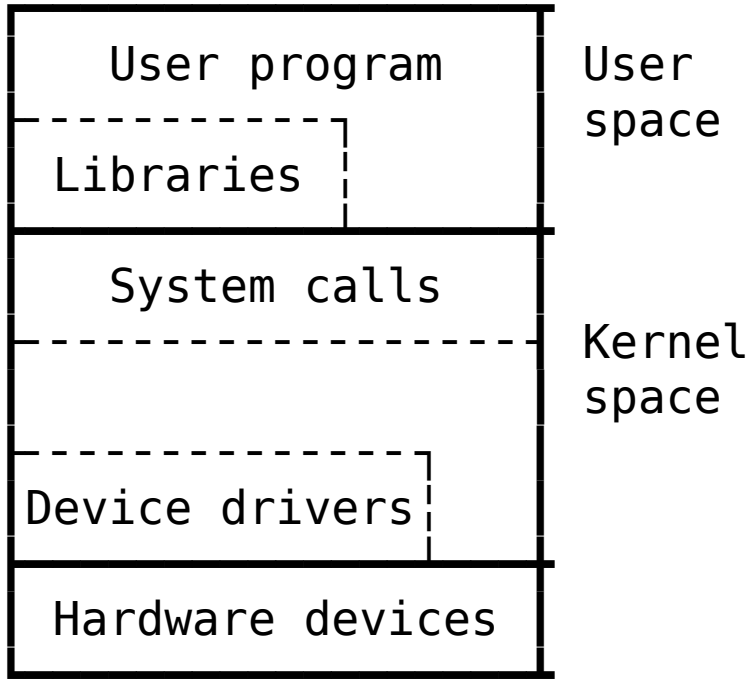
```c
#include <stdio.h>
#include <unistd.h>

int main ()
{
  printf("Hello World!\n");
  if(fork() != 0 )
    printf("I am the parent process.\n");
  else {
    printf("A child is listing the directory contents...\n");
    execl("/bin/ls", "ls", "-al", NULL);
  }
  return 0;
}
```

```
$ man 3 exec
```

GNU C Library

Linux API > POSIX API

```
$ man 7 libc
$ man 3 intro
$
$
$
```



Working With Files

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

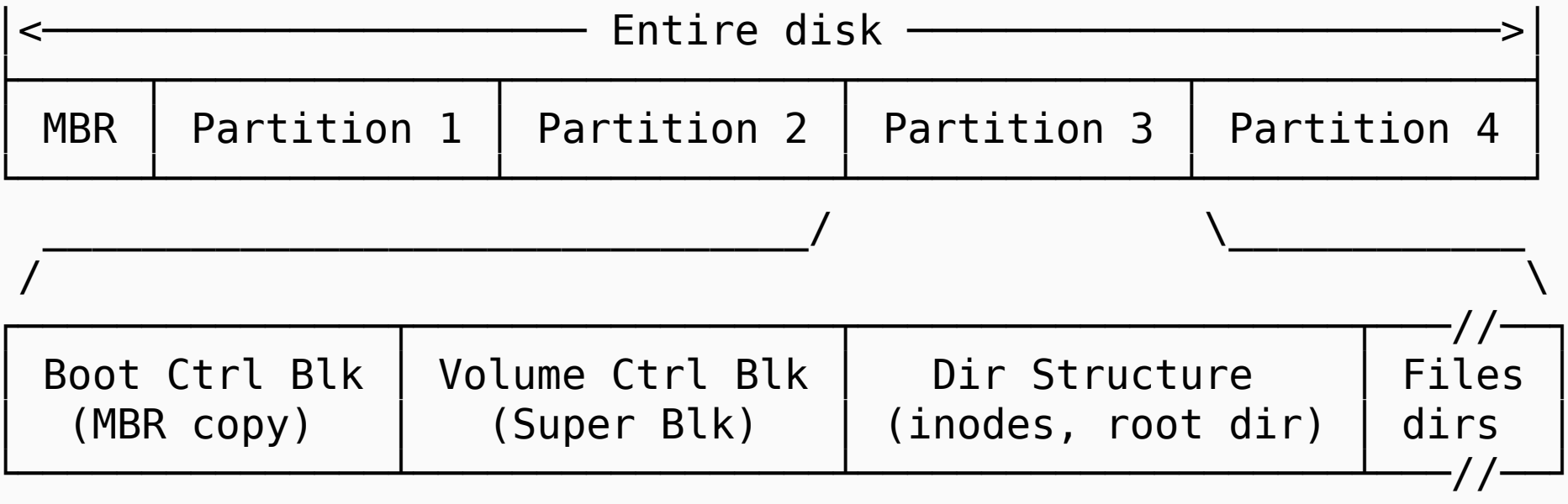
POSIX file system calls

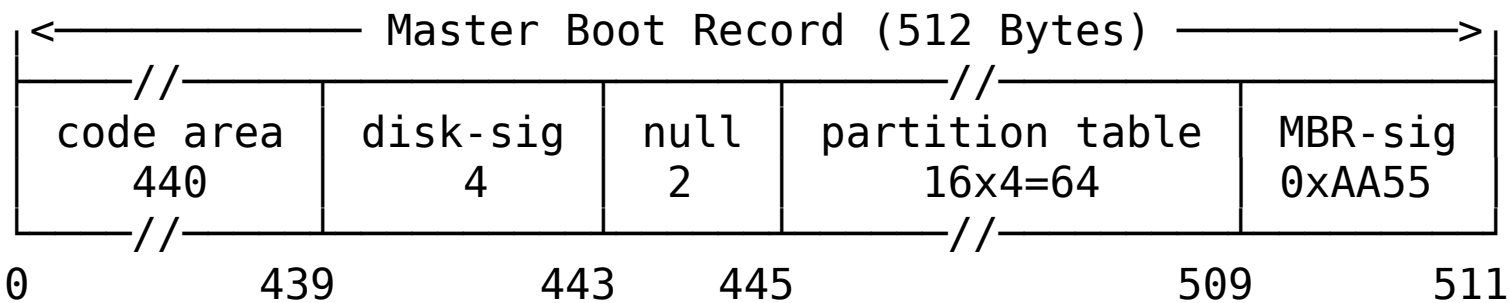
```
creat(name, mode)
open(name, flags)
close(fd)
link(oldname, newname)
unlink(name)
truncate(name, size)
ftruncate(fd, size)
stat(name, buffer)
fstat(fd, buffer)
```

```
read(fd, buffer, byte_count)
write(fd, buffer, byte_count)
lseek(fd, offset, whence)
chown(name, owner, group
fchown(fd, owner, group)
chmod(name, mode
fchmod(fd, mode)
utimes(name, times)
```

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

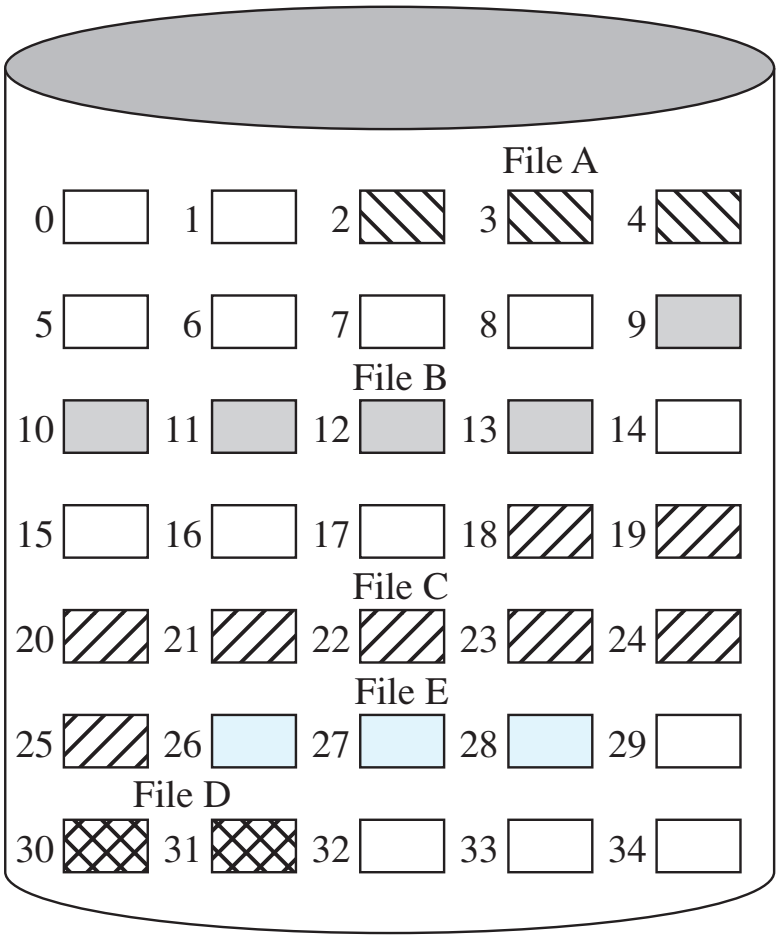
Keeps information about the file system

- 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

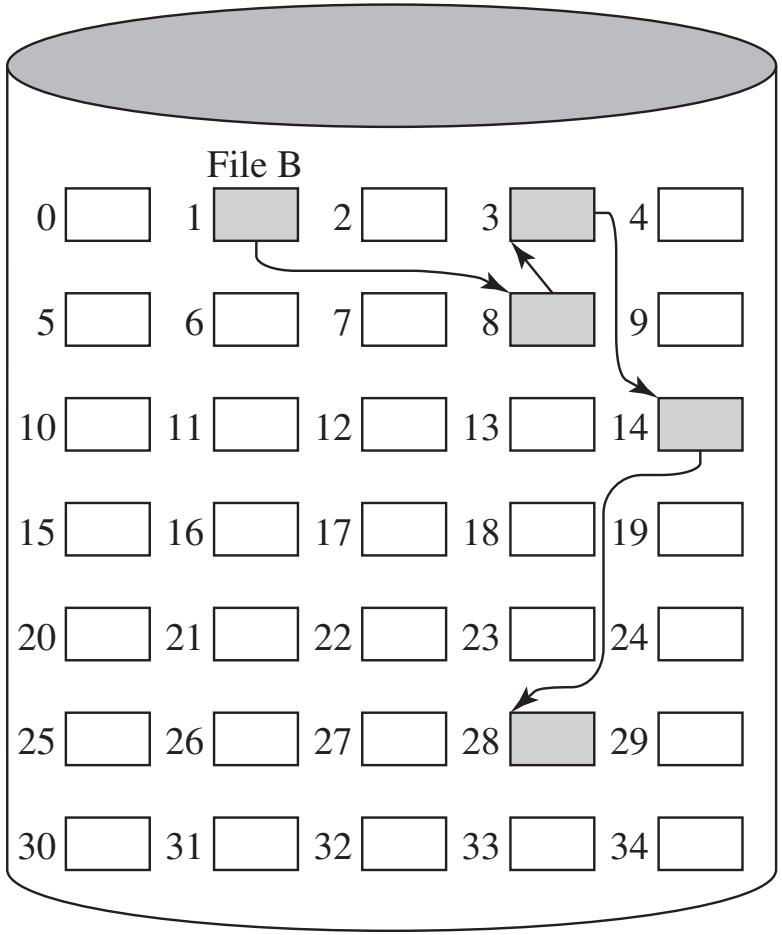


File Allocation Table

File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

- simple;
- good for read only;
- fragmentation

Linked List (Chained) Allocation A pointer in each disk block

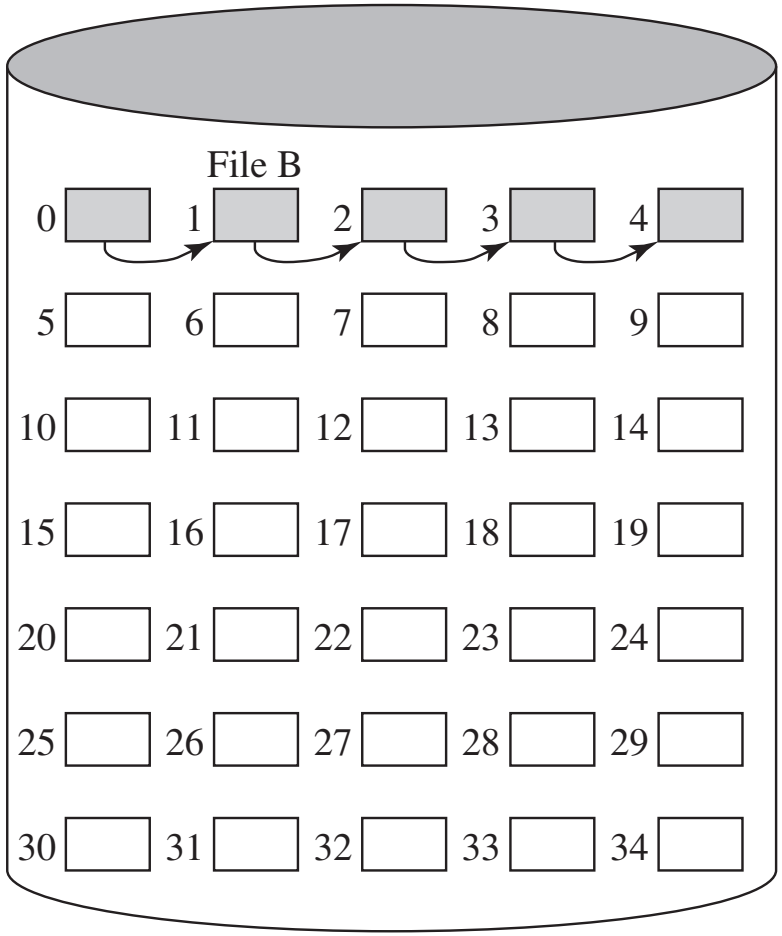


File Allocation Table

File Name	Start Block	Length
•••	•••	•••
File B	1	5
•••	•••	•••

- no waste block; access;
- slow random - not 2^n

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



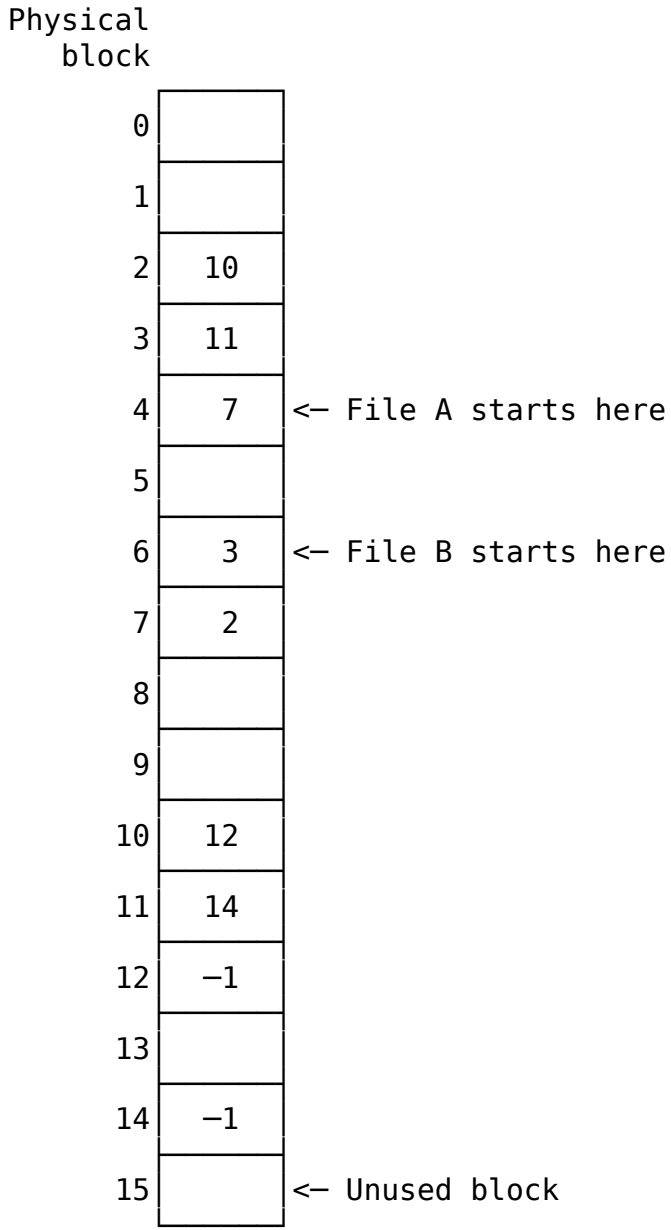
File Allocation Table

File Name	Start Block	Length
• • •	• • •	• • •
File B	0	5
• • •	• • •	• • •

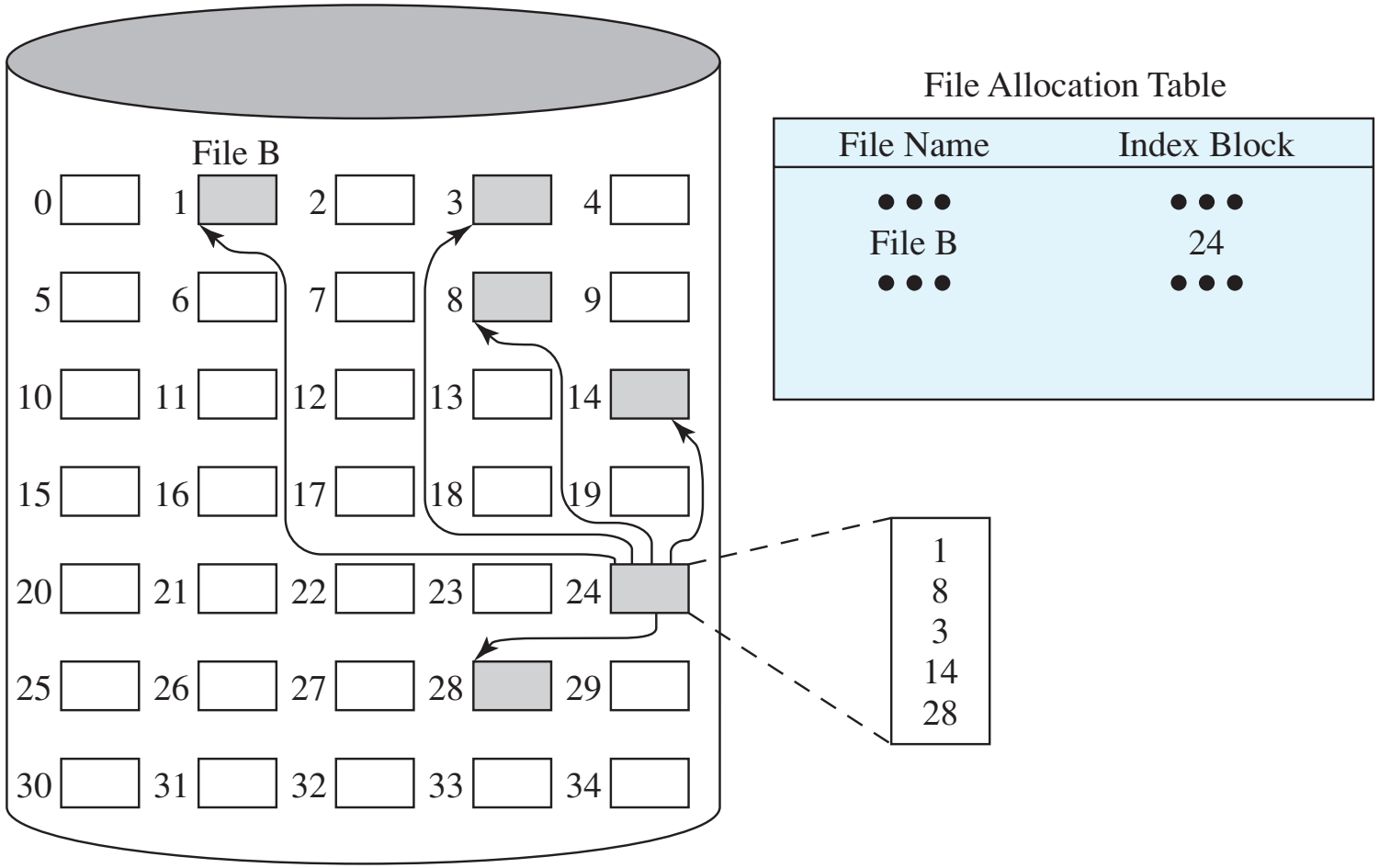
FAT: Linked list allocation with a table in RAM

- 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

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



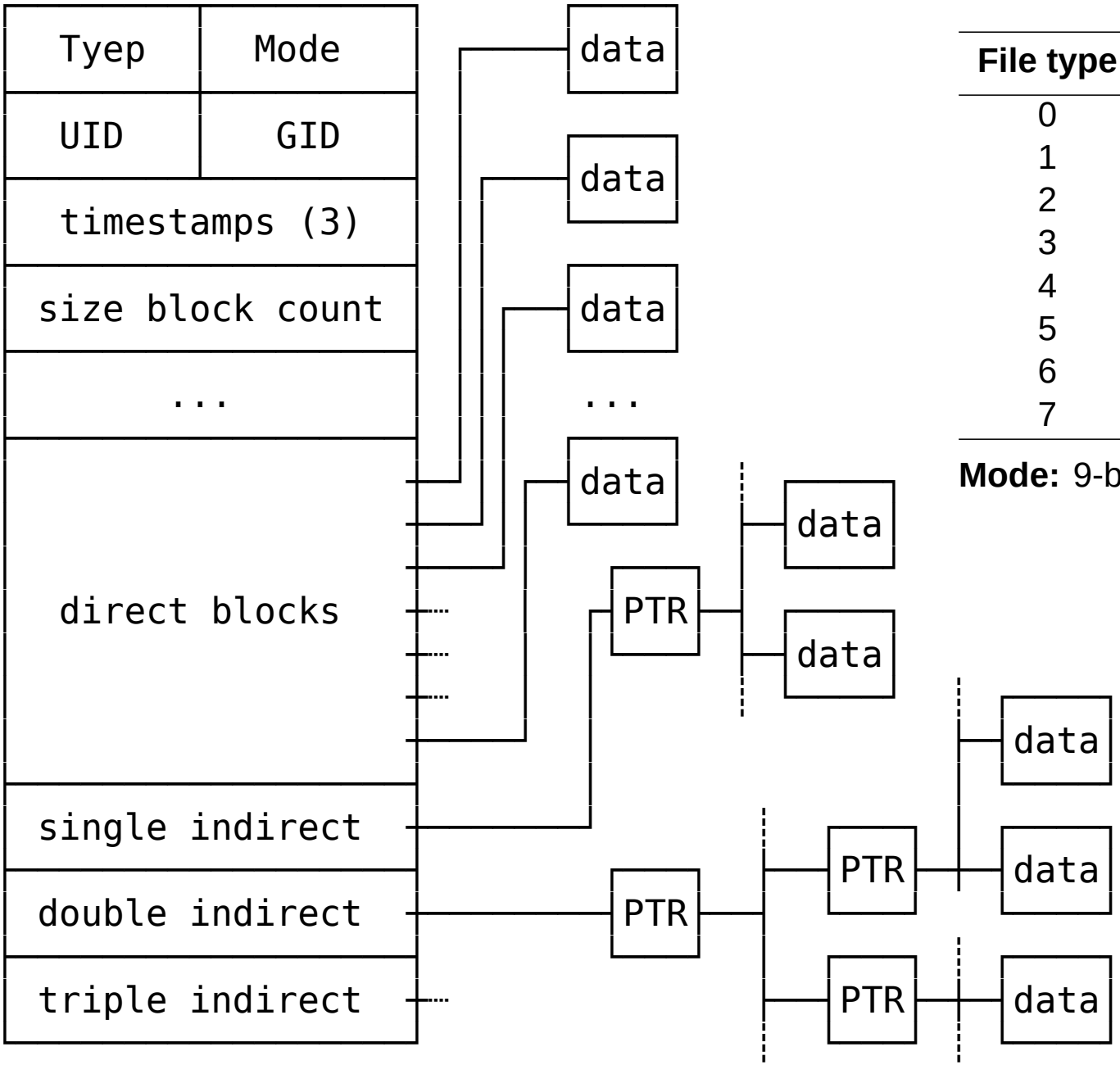
Indexed Allocation



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

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

I-node

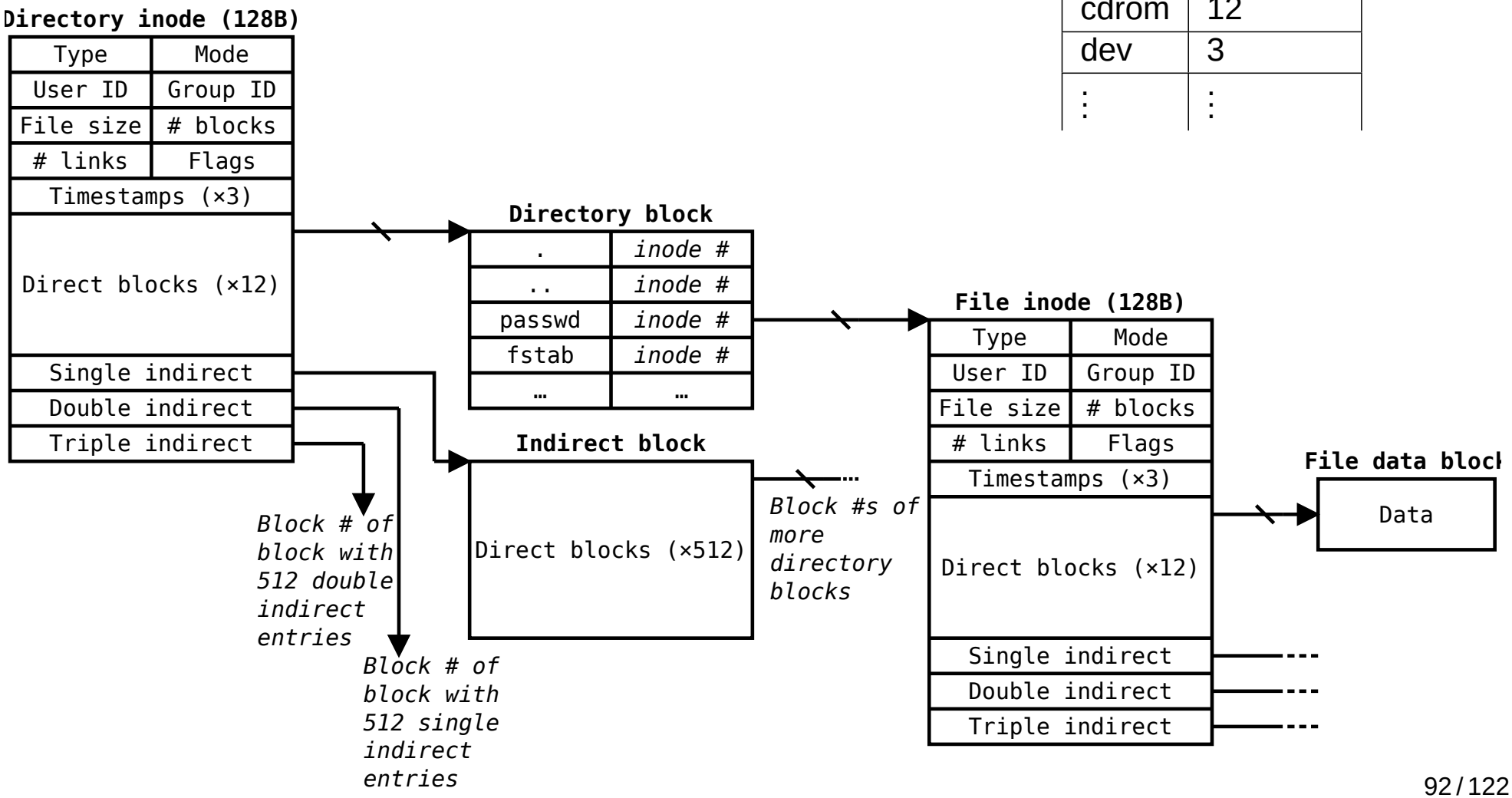


File type	Description
0	Unknown
1	Regular file
2	Directory
3	Character device
4	Block device
5	Named pipe
6	Socket
7	Symbolic link

Mode: 9-bit pattern

UNIX Treats a Directory as a File

.	2
..	2
bin	11116545
boot	2
cdrom	12
dev	3
⋮	⋮

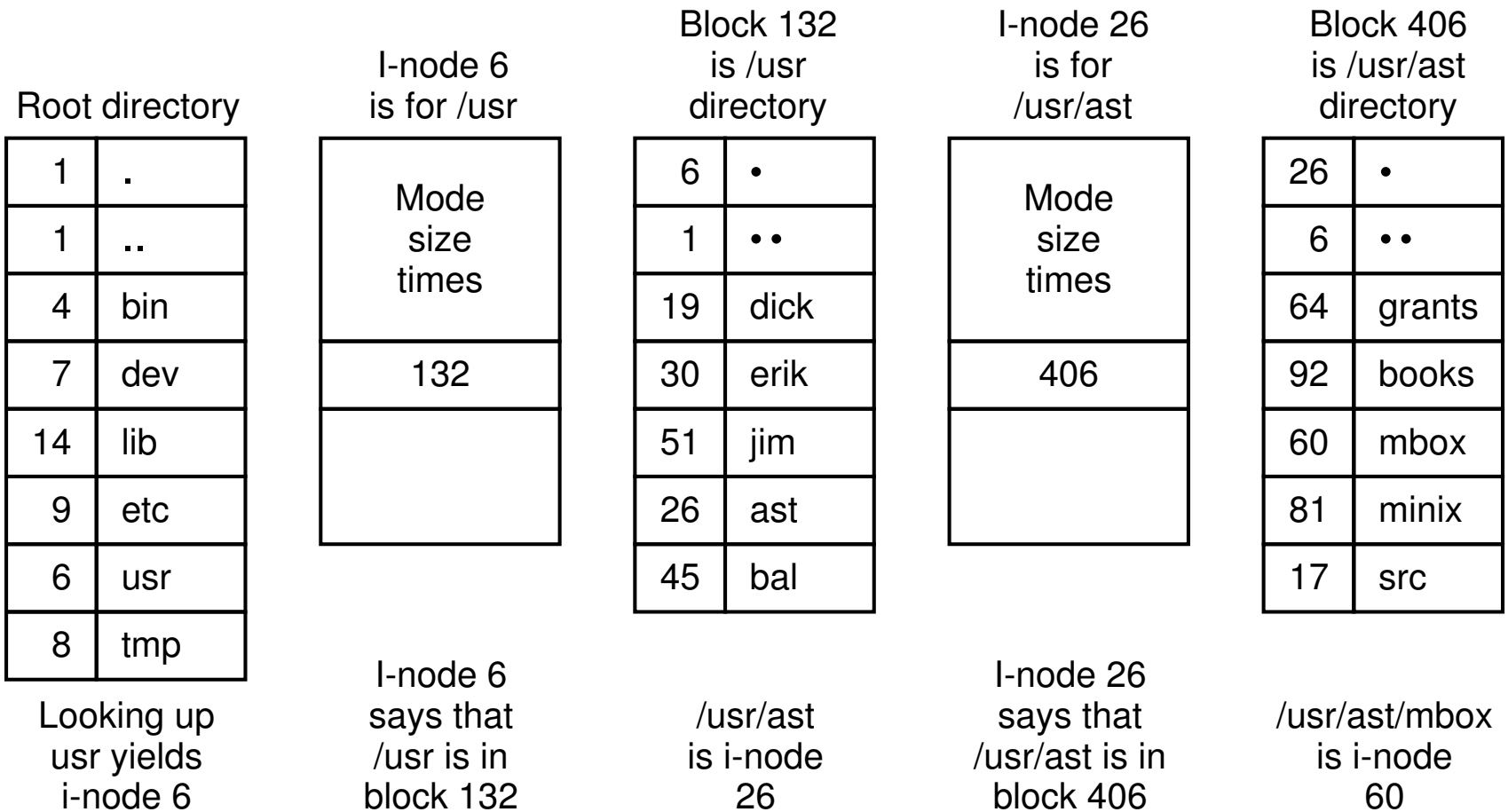


`open()`

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

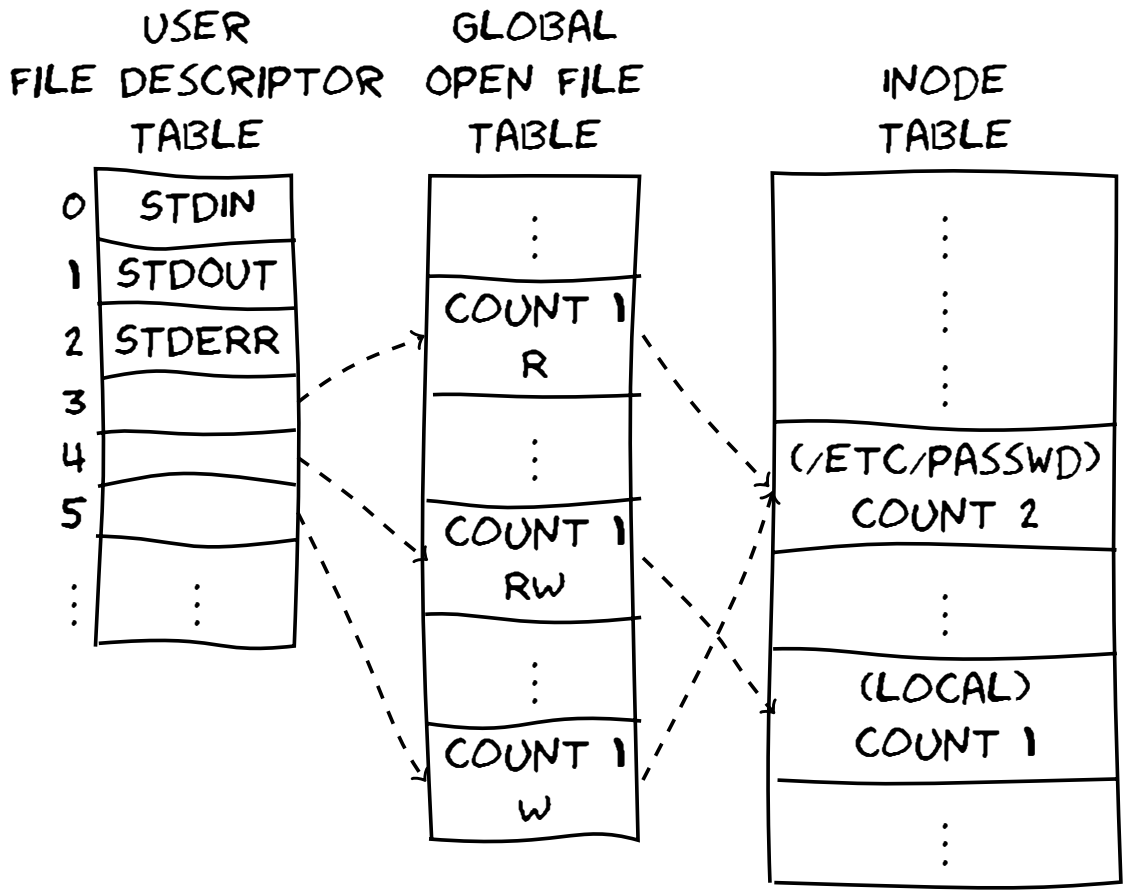
To see files opened by a process, e.g. `init`

```
$ lsof -p 1
```

```
$ man 2 open
```

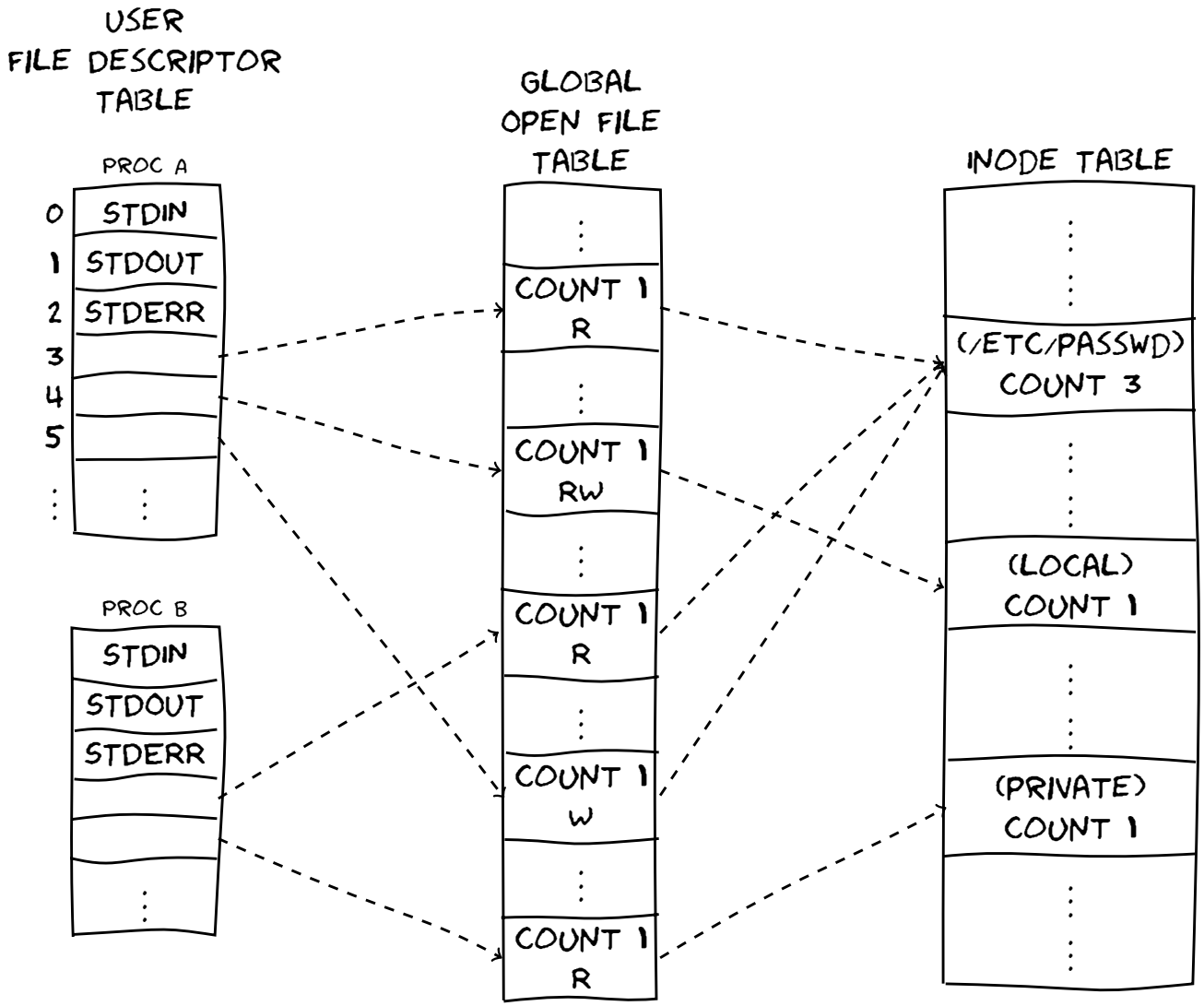
A process executes the following code:

```
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()

```c
#include <unistd.h>

int main(void)
{
  write(1, "Hello, world!\n", 14);

  return 0;
}
```

```
$ man 2 write
$ man 3 write
```

read()

```c
#include <unistd.h>

int main(void)
{
  char buffer[10];

  read(0, buffer, 10);

  write(1, buffer, 10);

  return 0;
}
```

```
$ man 2 read
$ man 3 read
```

cp

```c
#include <sys/types.h>   /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

#define BUF_SIZE 4096    /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700 /* protection bits for output file */

int main(int argc, char *argv[])
{
  int in, out, rbytes, wbytes;
  char buf[BUF_SIZE];

  if (argc != 3) exit(1);

  if ( (in = open(argv[1], O_RDONLY)) < 0 ) exit(2); /* open source file */

  if ( (out = creat(argv[2], OUTPUT_MODE)) < 0 ) exit(3); /* create destination file */

  while (1) { /* Copy loop */
    if ( (rbytes = read(in, buf, BUF_SIZE)) <= 0 ) break;   /* read a block of data */
    if ( (wbytes = write(out, buf, rbytes)) <= 0 ) exit(4); /* write data */
  }

  close(in);
  close(out);
  if (rbytes == 0) exit(0); /* no error on last read */
  else exit(5);             /* error on last read */
}
```

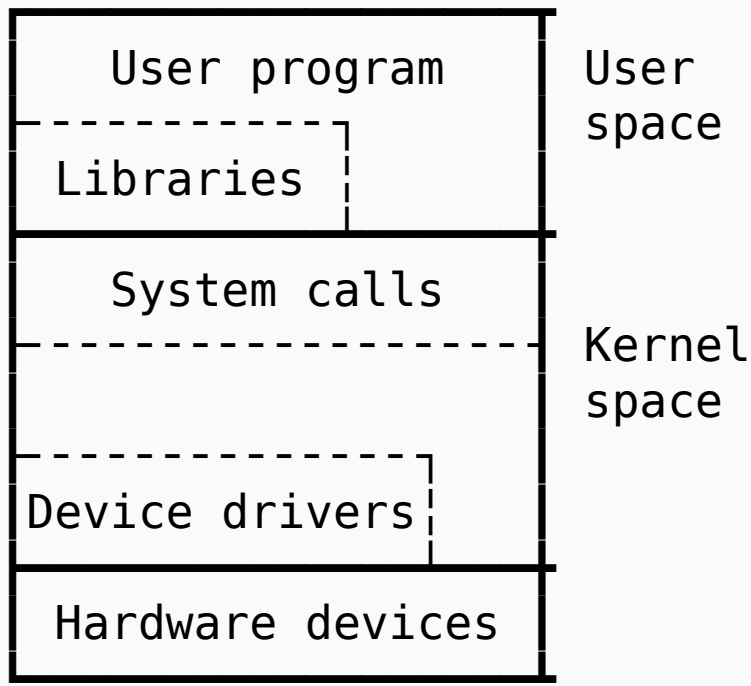
`stdio` — The Standard I/O Library

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()

```c
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
  char c;
  int in;
  in = open("/tmp/1m.test", O_RDONLY);

  while (read(in, &c, 1) == 1);

  return 0;
}
```

```
$ strace -c ./open
```

fopen() — Buffered I/O

```c
#include <stdio.h>

int main(void)
{
  FILE *stream;

  stream = fopen("/tmp/1m.test", "r");

  while ( fgetc(stream) != EOF );

  fclose(stream);

  return 0;
}
```

```
$ strace -c ./fopen
```

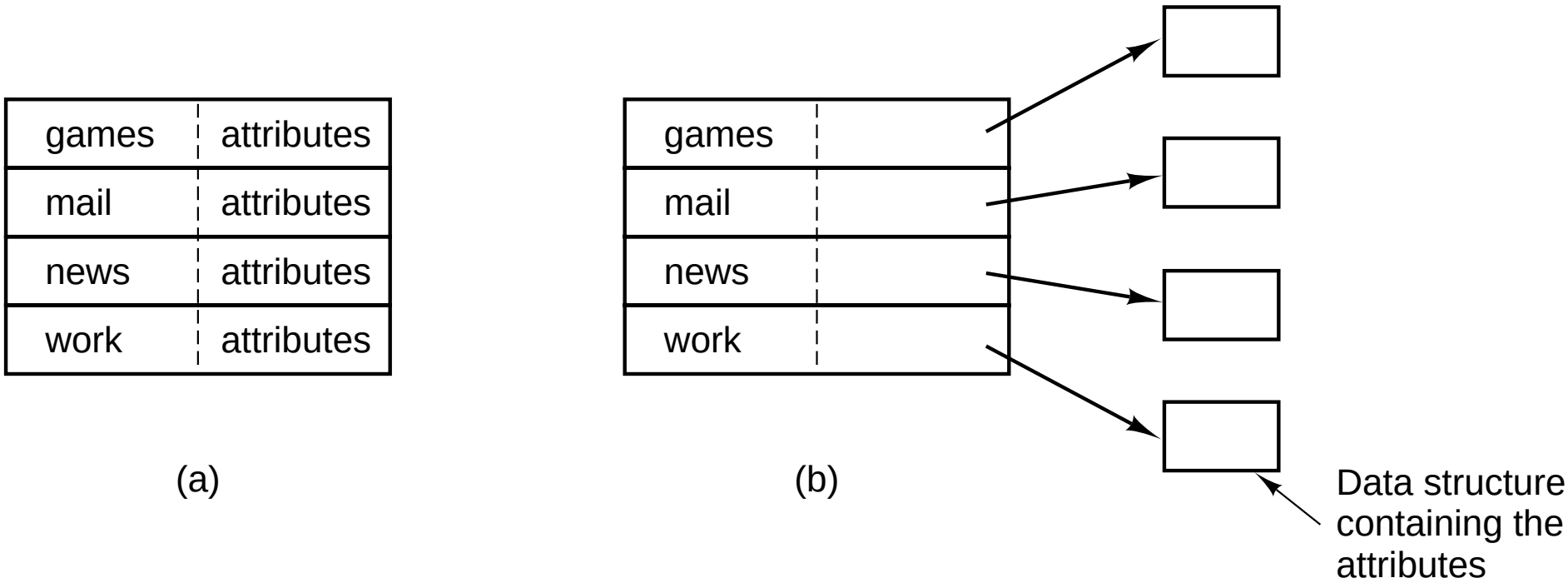
```
$ dd if=/dev/zero of=/tmp/1m.test bs=1k count=1024
```

cp — With stdio

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        FILE *in, *out;
        int c=0;

        if (argc != 3) exit(1);

        in = fopen(argv[1], "r");
        out = fopen(argv[2], "w");

        while ( (c = fgetc(in)) != EOF )
                fputc(c, out);

        return 0;
}
```

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

Implementing Directories



(a) A simple directory (Windows)

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

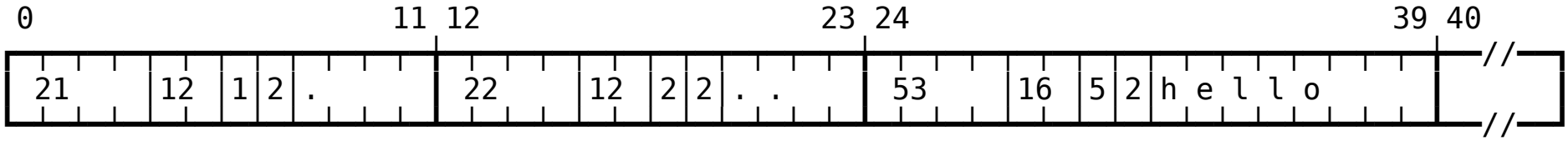
(b) Directory in which each entry just refers to an i-node (UNIX)

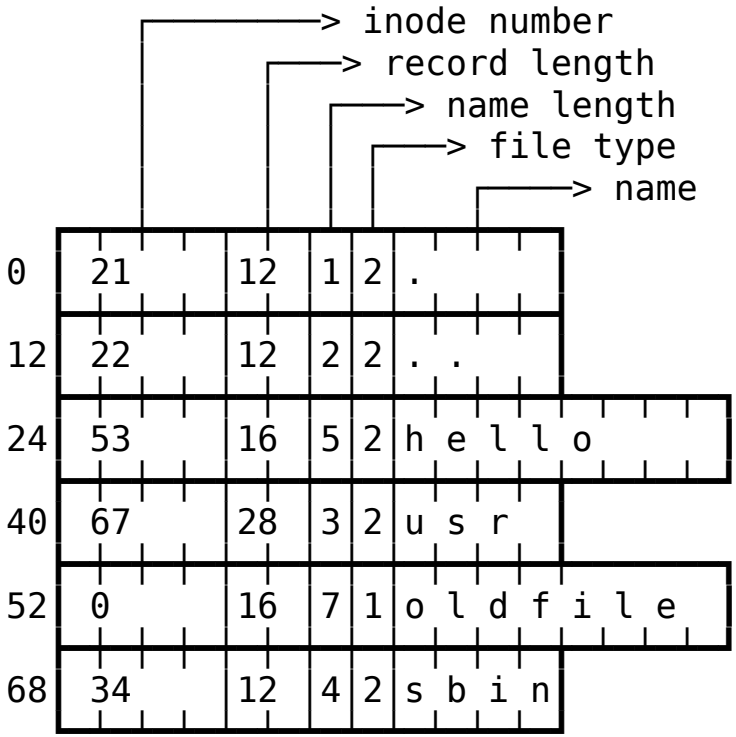
Directory entry in `glibc`

```
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 $4\times$
- inode number is 0 — deleted file

ls

```c
#include <sys/types.h>
#include <dirent.h>
#include <stddef.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  DIR *dp;
  struct dirent *entry;

  dp = opendir(argv[1]);

  while ( (entry = readdir(dp)) != NULL ){
    printf("%s\n", entry->d_name);
  }

  closedir(dp);

  return 0;
}
```

The real `ls.c`?

116 A4 pages
5308 lines

Do one thing, and do it really well.

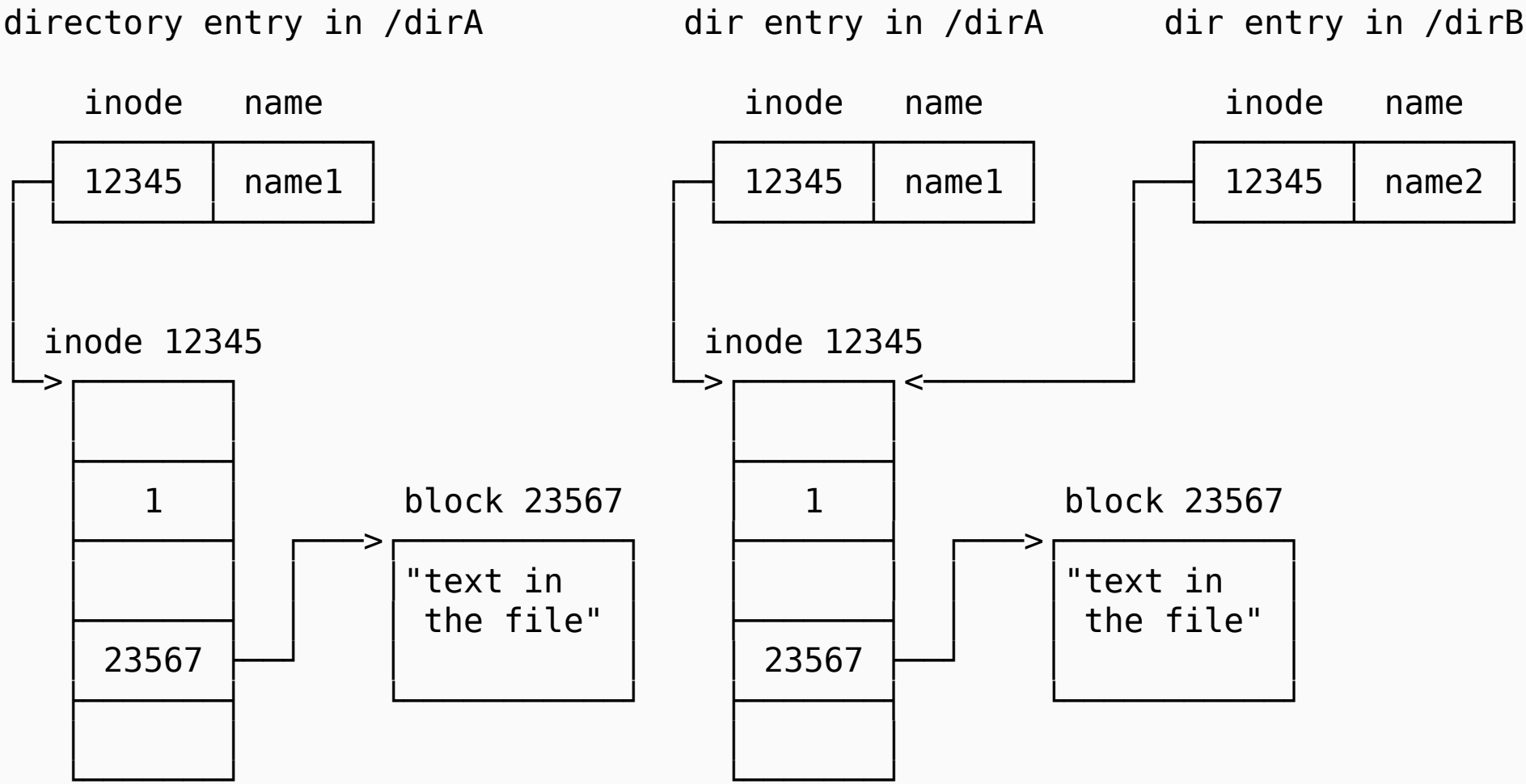
```
$ apt source coreutils
```

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

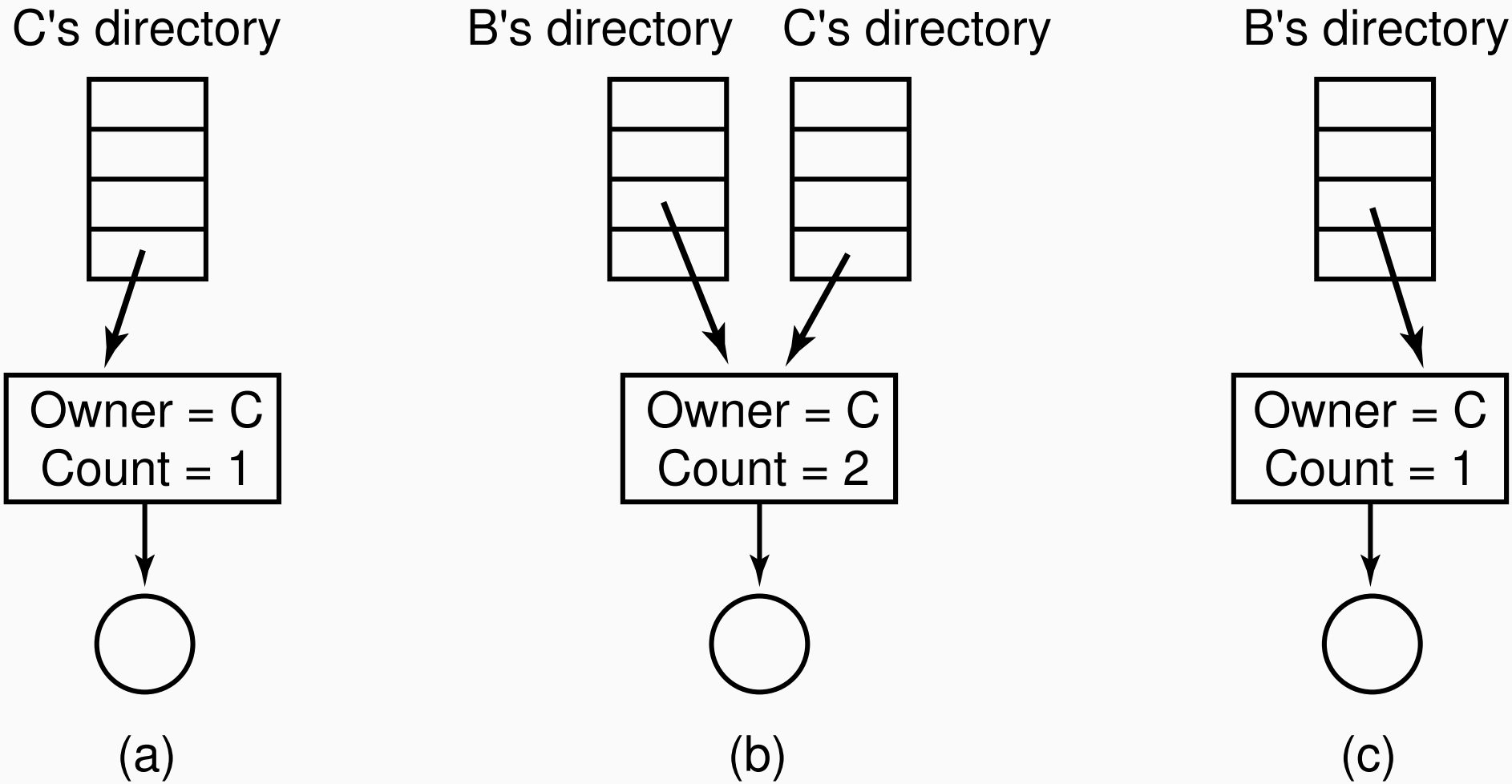
```c
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  char s[100];
  if ( mkdir(argv[1], S_IRUSR|S_IXUSR) == 0 )
    chdir(argv[1]);
  printf("PWD = %s\n", getcwd(s,100));
  rmdir(argv[1]);
  return 0;
}
```

Hard Links

Hard links ☛ the same inode

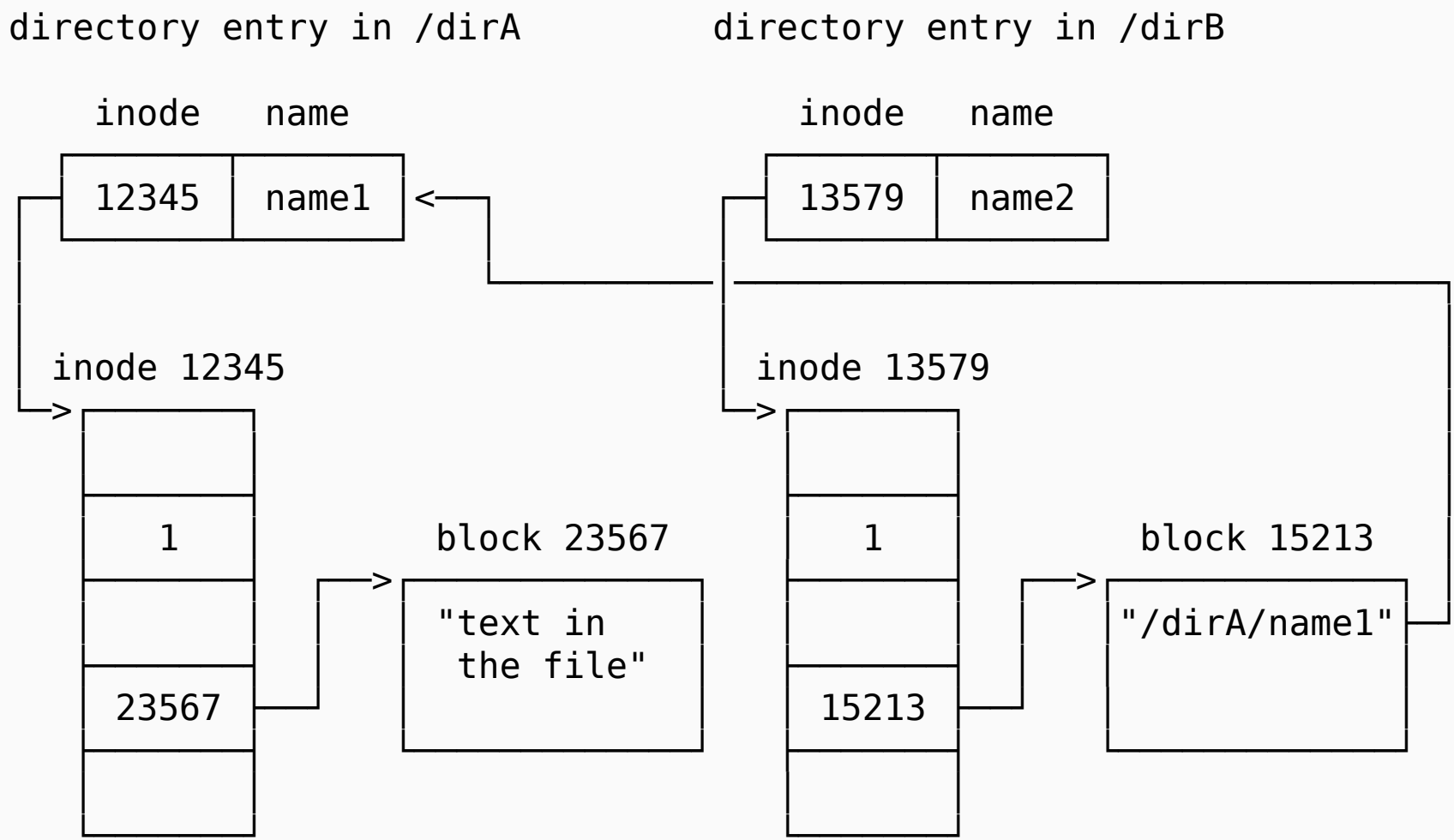


Drawback


C's directory
Owner = C
Count = 1
(a)
B's directory
C's directory
Owner = C
Count = 2
(b)
B's directory
Owner = C
Count = 1
(c)

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()

```c
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  link(argv[1], argv[2]);
  perror(argv[0]);
  return 0;
}

/* symlink(argv[1], argv[2]); */
/* unlink(argv[1]); */
```

The Linux Environment

Command Line Options

getopt.c

```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char* argv[]) {
  int opt;

  while ((opt = getopt(argc, argv, "hf:l")) != -1) {
    switch (opt) {
    case 'h':
      printf("Usage: %s [-h] [-f file] [-l]\n", argv[0]);
      break;
    case 'l':
      printf("option: %c\n", opt);
      break;
    case 'f':
      printf("filename: %s\n", optarg);
      break;
    }
  }
  return 0;
}
```

```
$ 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

```c
#include <stdlib.h>
#include <stdio.h>

extern char** environ;

int main() {
  char** env = environ;

  while (*env) {
    printf("%s\n", *env);
    env++;
  }

  return 0;
}
```

```
$ env
$ man 3 getenv
$ man 3 putenv
```

Time and Date

```c
#include <time.h>
#include <stdio.h>

int main(void)
{
  time_t t = time(NULL); /* long int */

  printf("epoch time:\t%ld\n",t);
  printf("calendar time:\t%s", ctime(&t));

  return 0;
}
```

- January 1 1970 — start of the Unix epoch

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

Temporary Files

mkstemp.c

```c
#include <stdlib.h>
#include <unistd.h>
#define _GNU_SOURCE
#include <stdio.h>

int main(int argc, char *argv[])
{
  char c, *f;

  asprintf(&f, "%sXXXXXX", argv[1]);
  int tmp = mkstemp(f);

  while ( read(0, &c, 1) == 1)
    write(tmp, &c, 1);

  unlink(f);
  free(f);
  return 0;
}
```

mktemp.sh

```sh
#!/bin/bash

tmp=$(mktemp)

while read LINE; do
  echo $LINE >> $tmp
done

```

```
$ man 3 mkstemp
$ man 3 tmpfile
$ man 3 asprintf
```

Logging

syslog.c

```c
#include <syslog.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
  if ( open(argv[1], O_RDONLY) < 0 )
    syslog(LOG_ERR | LOG_USER, "%s - %m\n", argv[1]);
  else
    syslog(LOG_INFO | LOG_USER, "%s - %m\n", argv[1]);
  return 0;
}
```

logger.sh

```sh
#!/bin/bash

[[ -f "$1" ]] && logger "$1 exists." || logger "$1 not found."
```

Terminal

IDE

Processes and signals

IPC

User Interface