

# GNU/Linux Application Programming

WANG Xiaolin

`wx672ster+linux@gmail.com`

August 26, 2019

# Reference Books



VENKATESH B, ANGRAVE L, et Al. *CS241 System Programming Coursebook*. University of Illinois, 2019.



MATTHEW N, STONES R. *Beginning linux programming*. 4th ed. 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/slides/)

 [https://cs2.swfu.edu.cn/~wx672/lecture\\_notes/linux-app/src/](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
```

System Programming <https://github.com/angrave/SystemProgramming/wiki>

Beej's Guides <http://beej.us/guide/>

BLP4e <http://www.wrox.com/WileyCDA/WroxTitle/productCd-0470147628,descCd-DOWNLOAD.html>

TLPI <http://www.man7.org/tlpi/>

## 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 on it?

✉ [wx672ster+linux@gmail.com](mailto:wx672ster+linux@gmail.com)

📖 Preferably in English

📖 in [stackoverflow](#) style

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



OVERSIMPLIFIED PROGRAMS  
AHEAD!

# Part I

## 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:

```
④ aptitude search xxx
```

```
④ apt-cache search xxx
```

```
④ apt-file search xxx
```

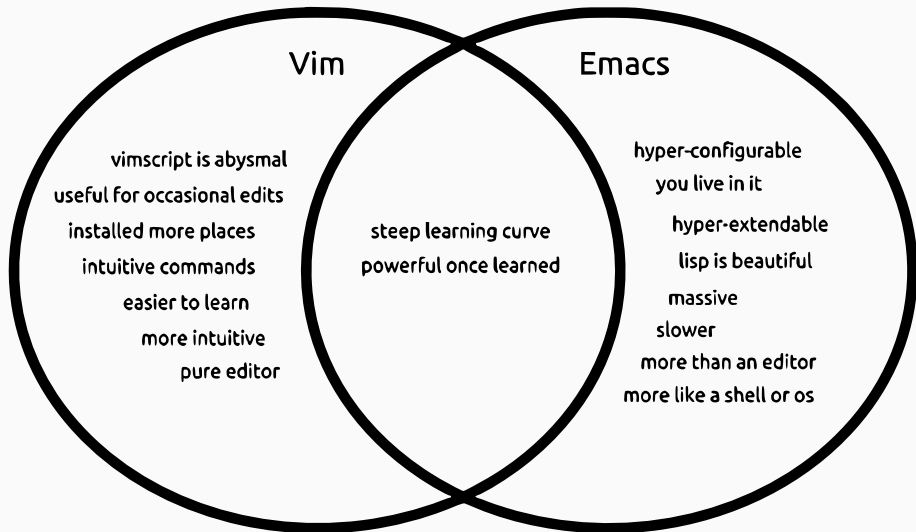
```
④ sudo apt install packagename
```

```
G Google "linux command xxx"
```

# Text Editors



vs.





# Help Your Editor

## Suffix matters

\$ vim ✗

\$ vim hello ✗

\$ vim hello.c ✓

\$ vim hello.py ✓

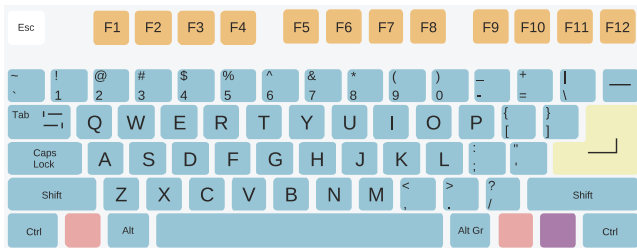
\$ emacs ✗

\$ emacs hello ✗

\$ emacsclient hello.c ✓

\$ emacsclient hello.py ✓

## Keyboard

 vimtutor

Ctrl + h t

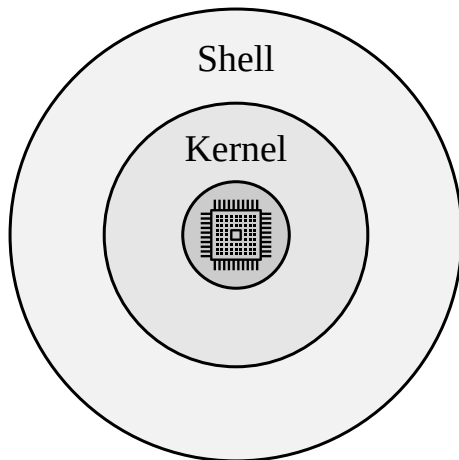
## Part II

### Shell Basics

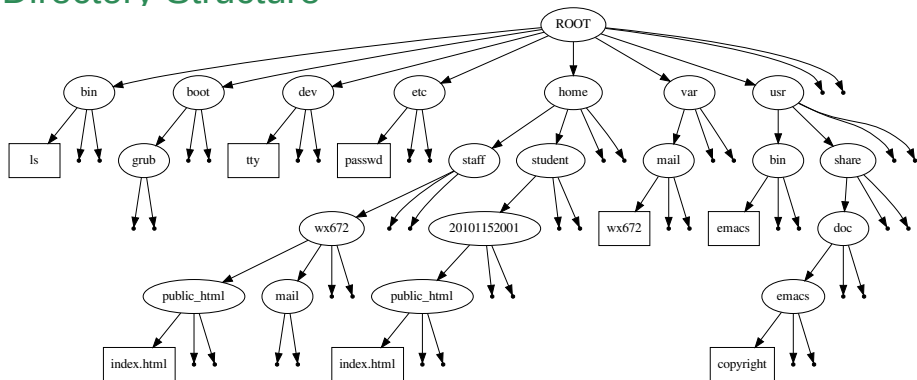
# 1 Shell Basics

# Shell

- ▶ A command line interpreter
- ▶ A programming language



## Directory Structure



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

# 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

# Redirection

## Redirecting output

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

```
$ ps aux >> output.txt
```

## Redirecting input

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



# 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** + **f** : forward

**Ctrl** + **n** : next

**Ctrl** + **r** : reverse search

**Ctrl** + **k** : kill (cut to end)

**Ctrl** + **d** : delete a character

**Ctrl** + **e** : end of line

**Ctrl** + **b** : backward

**Ctrl** + **p** : previous

**Ctrl** + **u** : cut to beginning

**Ctrl** + **y** : yank (paste)

 : completion

## Tmux

**Ctrl** + **a** **c** : create window

**Ctrl** + **a** **n** : next window

**Ctrl** + **a** **-** : split window

**Ctrl** + **a** **j** : go down

**Ctrl** + **a** **l** : go right

**Ctrl** + **a** **Ctrl** + **a** : switch window

**Ctrl** + **a** **p** : previous window

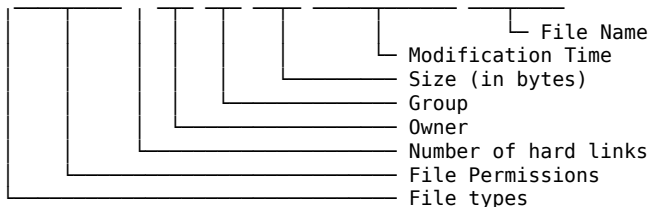
**Ctrl** + **a** **|** : split window

**Ctrl** + **a** **k** : go up

**Ctrl** + **a** **h** : go left

# Understanding "ls -l"

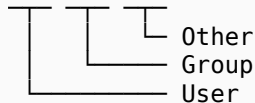
```
-rw----- 1 sam sam 57 Apr 17 1998 weather.txt
drwxr-xr-x 6 sam sam 102 Oct 9 1999 web_page
-rw-rw-r-- 1 sam sam 7648 Feb 11 20:41 web_site.tar
-rw----- 1 sam sam 574 Dec 16 1998 file.txt
```



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 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 ?.jpg
```

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

```
$ rm ?.*
```

```
$ rm 2*
```

```
$ 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
```

/proc

Allow higher-level access to driver and kernel information

```
$ cat /proc/cpuinfo
```

```
$ cat /proc/meminfo
```

```
$ cat /proc/version
```

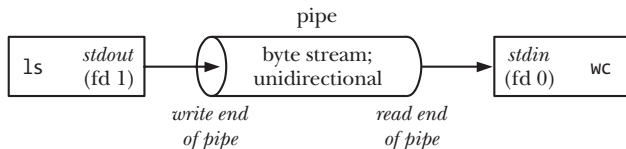
```
$ cat /proc/1/status
```

```
# echo 100000 > /proc/sys/kernel/pid_max
```

# Pipe

Chain processes together

```
$ ls | wc -l
```



## Unnamed pipe

```
$ unicode skull | head -1 | cut -f1 -d' ' | sm -
```

## Named pipe

1. 

```
$ mkfifo mypipe
```
2. 

```
$ gzip -9 -c < mypipe > out.gz
```
3. 

```
$ cat file > mypipe
```



## 2 Shell Programming

## \$ — Give Me The Value Of ...

`$var` Give me the value of variable “var”

`$(echo hello)` Give me the value (output) of command “echo hello”

`$((1+1))` Give me the value (result) of “1+1”

`$$` Give me the value of special variable “\$”

`$?` Give me the value of special variable “?”

`$0` Give me the value of special variable “0”

`@` Give me the value of special variable “@”

# 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 = $#"
```

```
8  echo -e "\targv[0] = $0"
```

```
9
10 i=1
11 for arg in $@; do
12     echo -e "\targv[$i] = $arg"
13     let i++
14 done
```

```
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 !
```



Re-write it in C

# 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<=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 }
```

# Array

```
1  #!/bin/bash
2
3  IMGDIR="$HOME/Pics/2009Summer/wallpapers/2009summer-1280x768"
4
5  files=($IMGDIR/*.jpg)
6
7  # get the length of array ${files[@]}
8  n=${#files[@]}
9
10 # get a random array element
11 wallpaper="${files[RANDOM % n]}"
12
13 # set it as wallpaper
14 qiv -z $wallpaper
```



Change wallpaper every 5 mins?



## Part III

# Linux Programming Environment

### 3 C Programming Environment

# Program Languages

## Machine code

The **binary numbers** that the CPUs can understand.

100111000011101111001111 ... and so on ...

People don't think in numbers.

## Assembly language — friendly to humans

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

**Assemblers** translate the ASM programs to machine code

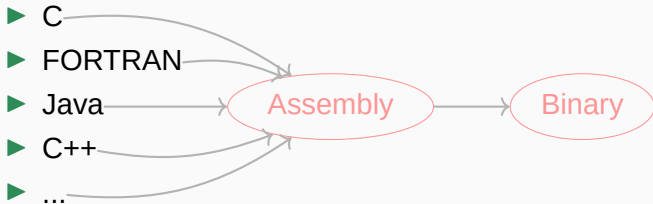
## High level languages

Even easier to understand by humans. Examples:

- ▶ C
- ▶ FORTRAN
- ▶ Java
- ▶ C++
- ▶ ...

## High level languages

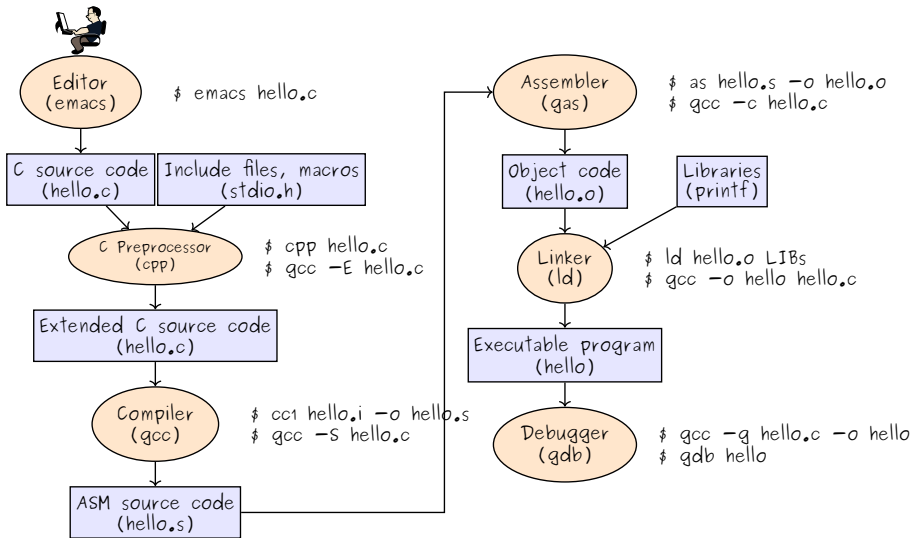
Even easier to understand by humans. Examples:



**Compilers** do the translation work

## 3.1 The Tool Chain

# Compilation



# Compiler vs. Interpreter

## hello.c

```
1 #include <stdio.h>
2 int main()
3 {
4     printf("Hello, world!\n");
5     return 0;
6 }
```

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

```
$ ./hello
```

## hello.sh

```
1 #!/bin/bash
2 echo 'Hello, world!'
```

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

```
$ ./hello.sh
```

## hello.py

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

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

```
$ ./hello.py
```



## 3.2 Header Files

# Header Files

## Why?

```
1 #include "add.h"
2
3 int triple(int x)
4 {
5     return add(x, add(x,x));
6 }
```

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

## Why not?

```
1 int add(int, int);
2
3 int triple(int x)
4 {
5     return add(x, add(x,
6         ↪ x));
7 }
```

## In the header files...

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

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

## 3.3 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:

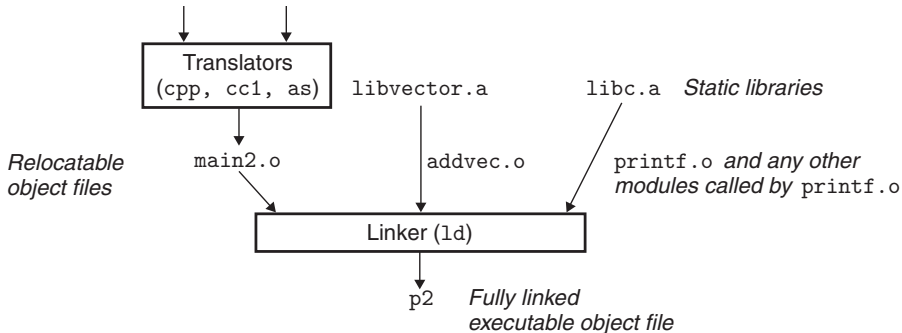
```
$ gcc -o hello hello.c /usr/lib/libm.a
```

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

## Static Linking

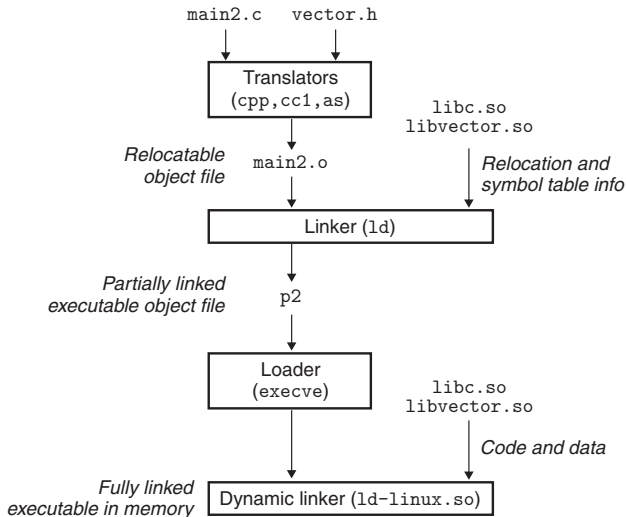
- ▶ The entire program and all data of a process must be in physical memory for the process to execute
- ▶ The size of a process is thus limited to the size of physical memory

Source files `main2.c` `vector.h`



## 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

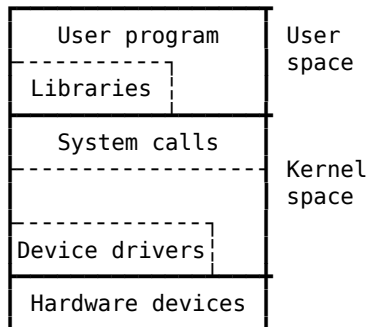
\$ man 7 libc

\$ man 3 intro

\$ man gcc

\$ info gcc

🌀 sudo apt install gcc-doc



## 3.4 Error Handling

## errno.h

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/stat.h>
4  #include <fcntl.h>
5
6  int main(int argc, char *argv[])
7  {
8      if ( open(argv[1], O_RDONLY) == -1 ){
9          perror("open");
10         exit(EXIT_FAILURE);
11     }
12     return 0;
13 }
```

\$ man errno

\$ man errno.h

\$ man perror


## 3.5 The Make Utility



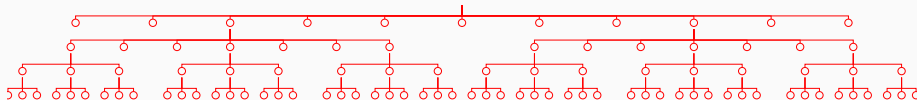
# The Make Utility

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 | |  command
```

## Example

```
1 | hello: hello.c
2 | |  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
```

## 3.6 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
```

## 3.7 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

## 3.8 A Sample GNU Package

# How to “Do one thing, and do it well”?

```
$ apt source hello
```

## 4 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;          $ env
8                                      $ man 3 getenv
9      while (*env) {                $ man 3 putenv
10         printf("%s\n", *env);
11         env++;
12     }
13
14     return 0;
15 }
```

# 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

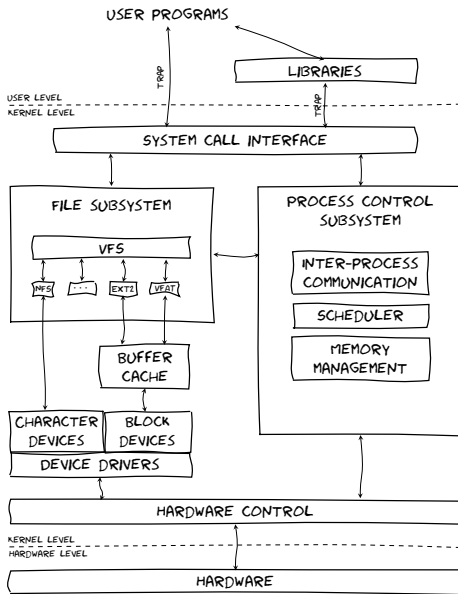
```
1  #include <syslog.h>
2  #include <sys/stat.h>
3  #include <fcntl.h>
4
5  int main(int argc, char *argv[])
6  {
7      if ( open(argv[1], O_RDONLY) < 0 )
8          syslog(LOG_ERR | LOG_USER, "%s - %m\n", argv[1]);
9      else
10         syslog(LOG_INFO | LOG_USER, "%s - %m\n", argv[1]);
11    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

## logger.sh

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

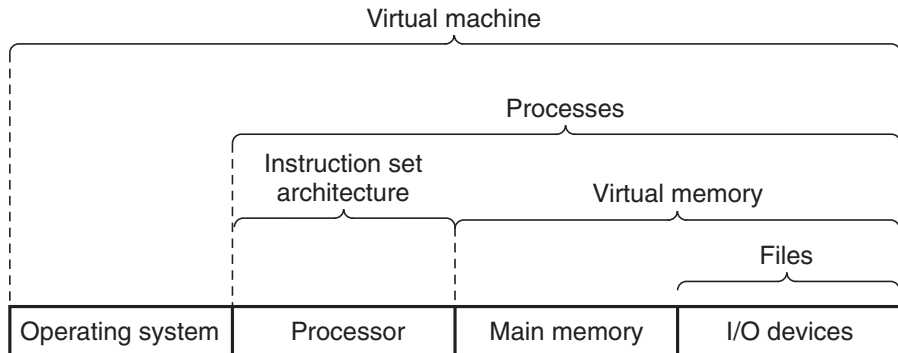
## 5 OS Basics

# Operating System

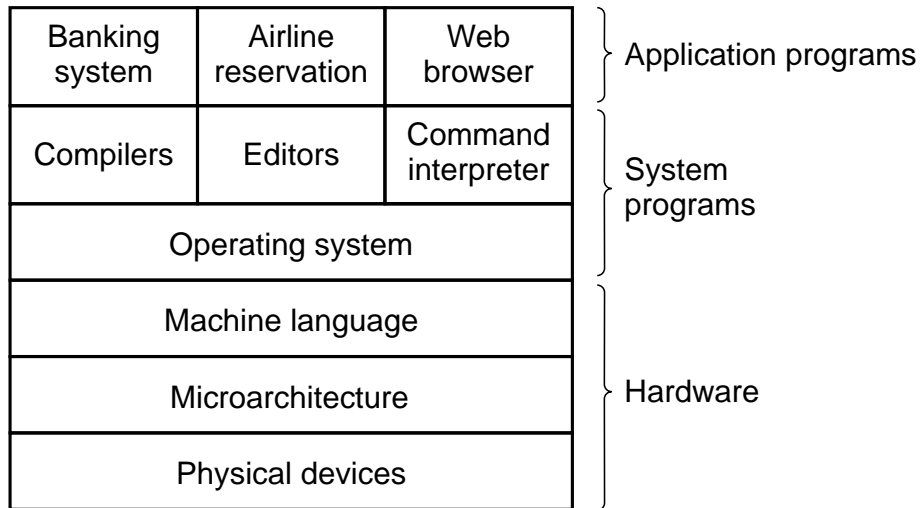


# Abstractions

To hide the complexity of the actual implementations



# A Computer System



## 5.1 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

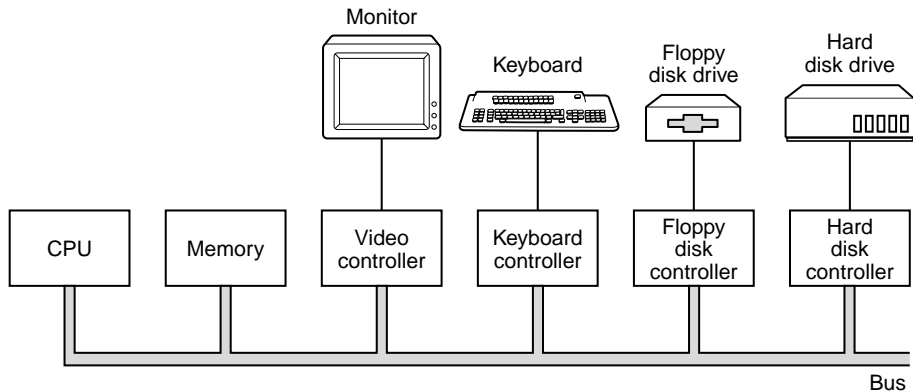
**Stack pointer (SP):** ➡ the top of the current stack in memory

**Program status (PS):** holds

- condition code bits
- processor state



# System Bus



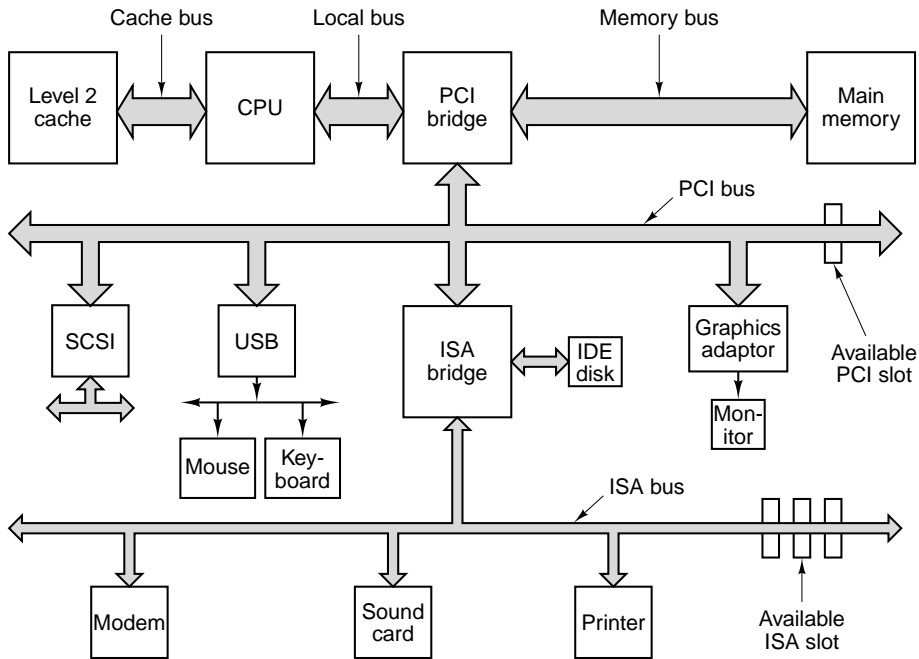
**Address Bus:** specifies the memory locations (addresses) for the data transfers

**Data Bus:** holds the data transferred. Bidirectional

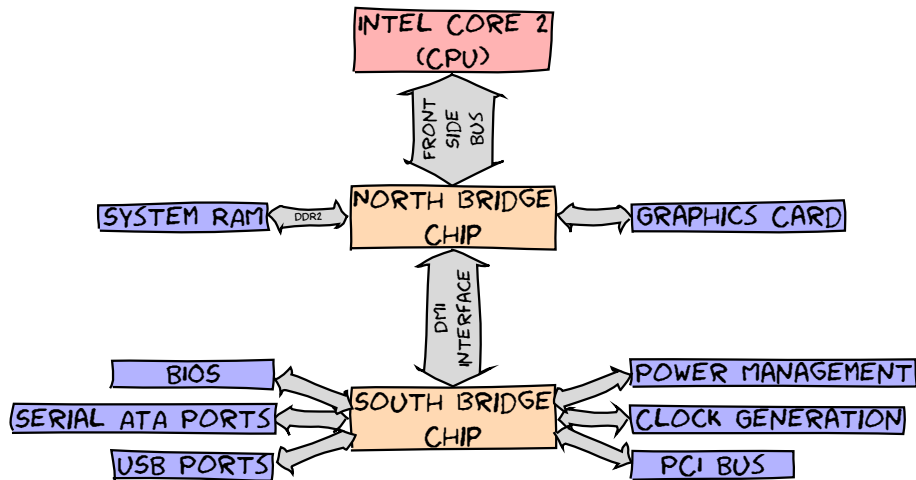
**Control Bus:** contains various lines used to route timing and control signals throughout the system

# 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



- ▶ 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.

0xFFFFFFFF	Reset vector	0xFFFFFFFF	JUMP to 0xF0000	4GB
			Unaddressable memory, real mode is limited to 1MB.	4GB-16B
0x100000			System BIOS	1MB
0xF0000			Ext. System BIOS	960KB
0xE0000			Expansion Area (maps ROMs for old peripheral cards)	896KB
0xC0000			Legacy Video Card Memory Access	768KB
0xA0000			Accessible RAM (640KB is enough for anyone – old DOS area)	640KB
0				0

## 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](#)

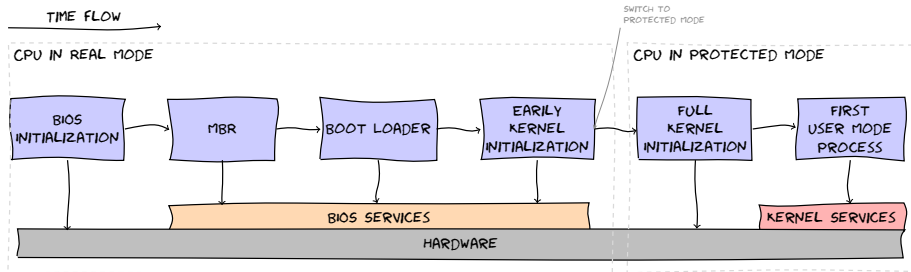
## 5.2 Bootstrapping



# 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 (0xfffffff0)

➡ POST ➡ HW init ➡ Find a boot device (FD,CD,HD...) ➡ Copy **sector zero (MBR)** to RAM (0x00007c00)

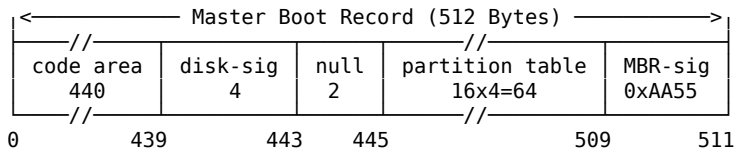
## 2. MBR – the first 512 bytes, contains

- ▶ 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
```

## 5.3 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**

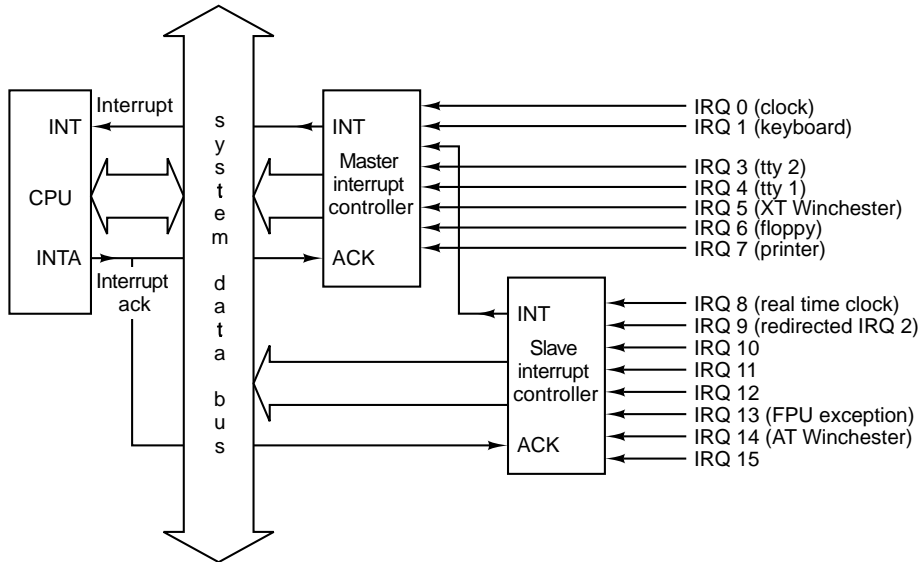
Trap (exception) is a software-generated INT caused by an error or by a specific request from an user program

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

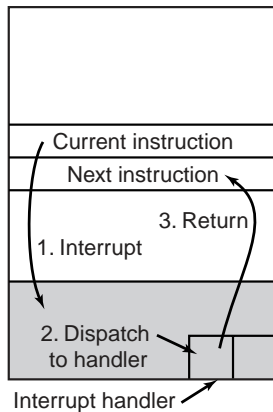
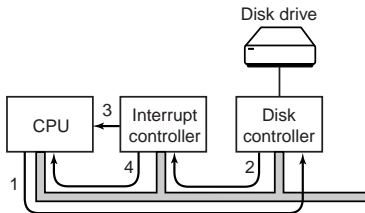
```
$ less /proc/devices
```

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

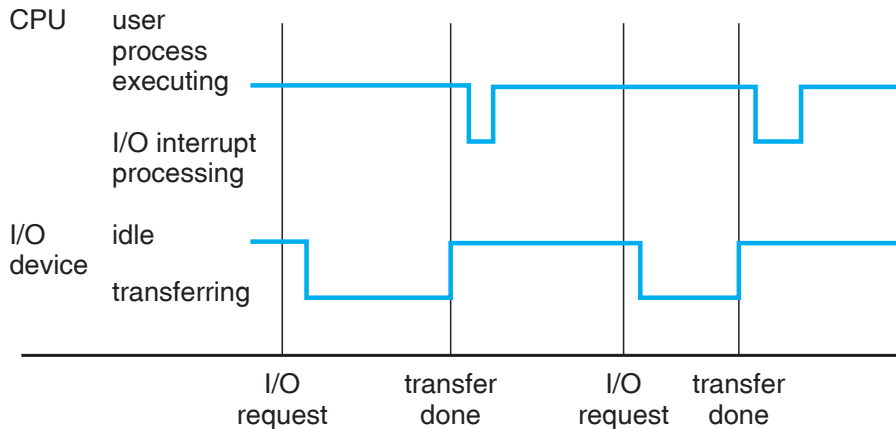
# Programmable Interrupt Controllers



# Interrupt Processing



# Interrupt Timeline





## 5.4 System Calls

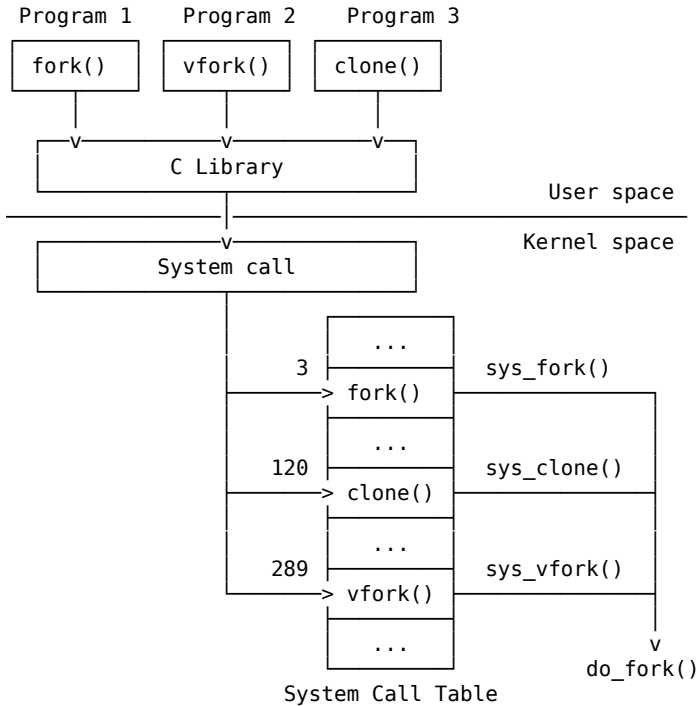
# System Calls

## A System Call

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

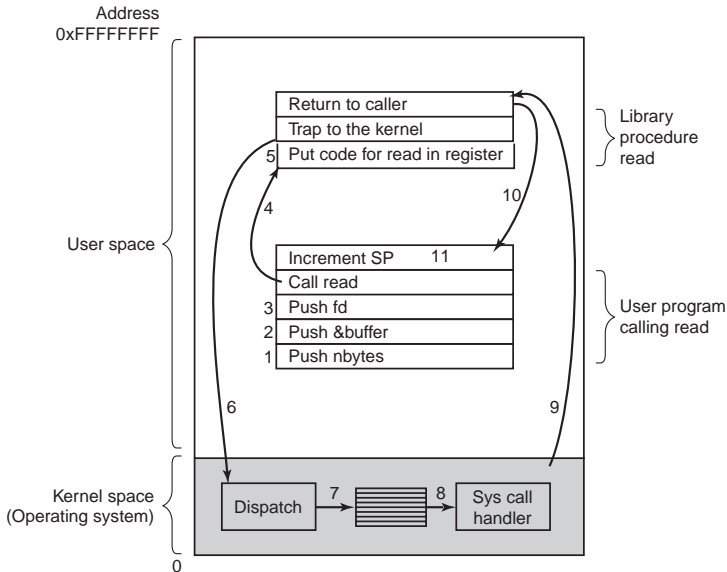
```
$ man 2 intro
```

```
$ 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.

- ▶  $256 \text{ entries} \times 4\text{B} = 1\text{KiB}$
- ▶ 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()

```
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



## execve()

```
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 2 execve

\$ man 3 exec

## Part IV

### Working With Files

## 6 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)
```

## 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

► No need to open() STDIN, STDOUT, and STDERR



cp

```
#define BUF_SIZE 4096
#define OUTPUT_MODE 0700

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);

    if ( (out = creat(argv[2], OUTPUT_MODE)) < 0 ) exit(3);

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

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

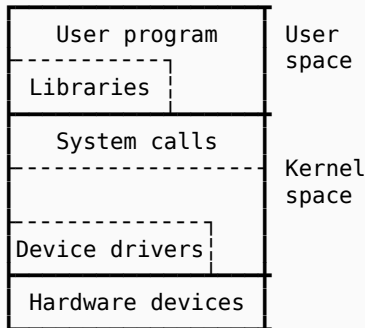
# stdio — The Standard I/O Library

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()

```
1  #include <unistd.h>
2  #include <sys/stat.h>
3  #include <fcntl.h>
4  #include <stdio.h>
5
6  int main()
7  {
8      char c;
9      int in;
10     in = open("/tmp/1m.test", O_RDONLY);
11
12     while (read(in, &c, 1) == 1);
13
14     return 0;
15 }
```

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

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

### 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
```

## cp — With stdio

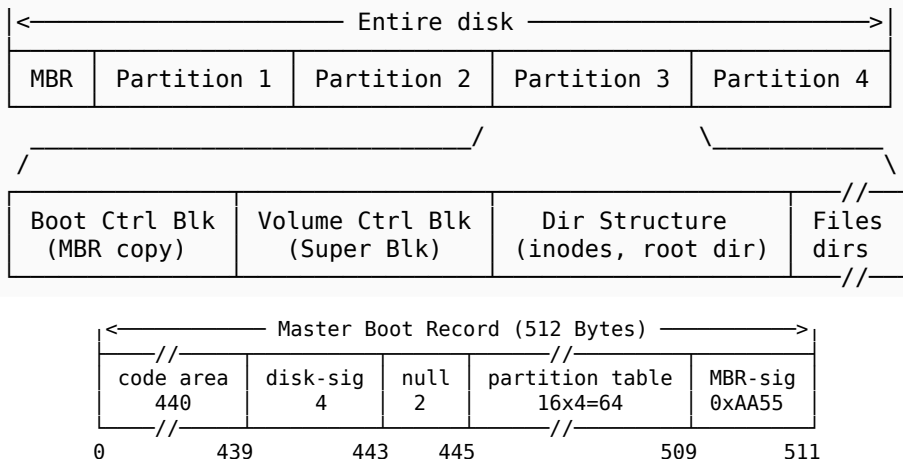
```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6      FILE *in, *out;
7      int c=0;
8
9      if (argc != 3) exit(1);
10
11     in = fopen(argv[1], "r");
12     out = fopen(argv[2], "w");
13
14     while ( (c = fgetc(in)) != EOF )
15         fputc(c, out);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



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

# 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).

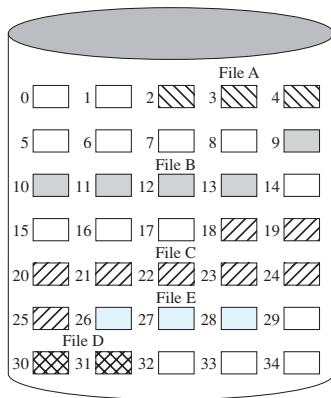
## Superblock

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



File Allocation Table

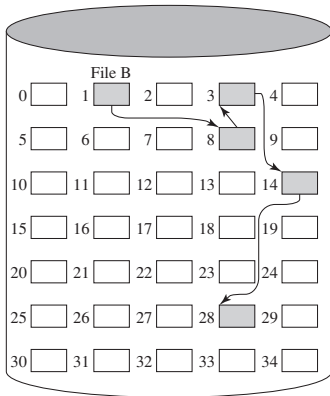
File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

## Contiguous Allocation

- 😊 simple
- 😊 good for read only

😞 fragmentation





File Allocation Table

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

## Linked List (Chained) Allocation

A pointer in each disk block



no waste block

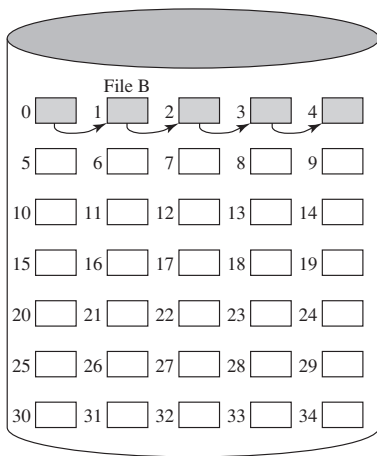


slow random access



not  $2^n$

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



File Allocation Table

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

## FAT: Linked list allocation with a table in RAM

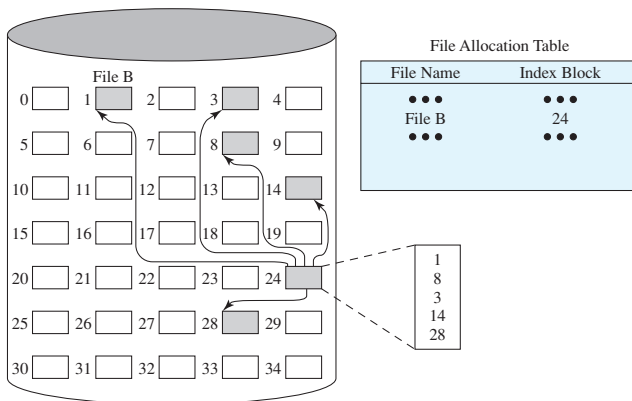
- ▶ Taking the pointer out of each disk block, and putting it into a table in memory
- ▶ fast random access (chain is in RAM)
- ▶ is  $2^n$
- ▶ the entire table must be in RAM

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

Physical  
block

0		
1		
2	10	
3	11	
4	7	← File A starts here
5		
6	3	← File B starts here
7	2	
8		
9		
10	12	
11	14	
12	-1	
13		
14	-1	
15		← Unused block

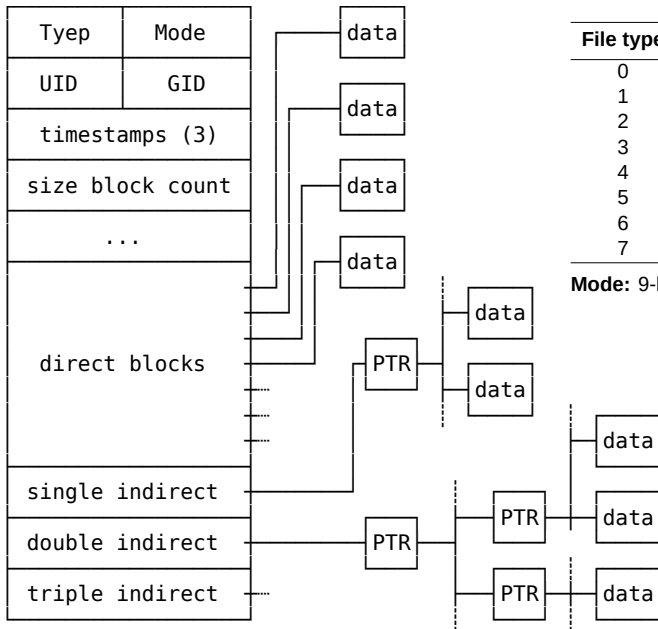
# Indexed Allocation



**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
:	:

**Directory inode (128B)**

Type	Mode
User ID	Group ID
File size	# blocks
# links	Flags
Timestamps (x3)	
Direct blocks (x12)	
Single indirect	
Double indirect	
Triple indirect	

**Directory block**

.	inode #
..	inode #
passwd	inode #
fstab	inode #
...	...

**Indirect block**

Direct blocks (x512)
----------------------

**File inode (128B)**

Type	Mode
User ID	Group ID
File size	# blocks
# links	Flags
Timestamps (x3)	
Direct blocks (x12)	
Single indirect	
Double indirect	
Triple indirect	

**File data block**

Data
------

Block # of  
block with  
512 double  
indirect  
entries

Block # of  
block with  
512 single  
indirect  
entries

Block #s of  
more  
directory  
blocks

# 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

Root directory

1	.
1	..
4	bin
7	dev
14	lib
9	etc
6	usr
8	tmp

Looking up  
usr yields  
i-node 6

I-node 6  
is for /usr

Mode size times
132

I-node 6  
says that  
/usr is in  
block 132

Block 132  
is /usr  
directory

6	.
1	..
19	dick
30	erik
51	jim
26	ast
45	bal

/usr/ast  
is i-node  
26

I-node 26  
is for  
/usr/ast

Mode size times
406

I-node 26  
says that  
/usr/ast is in  
block 406

Block 406  
is /usr/ast  
directory

26	.
6	..
64	grants
92	books
60	mbox
81	minix
17	src

/usr/ast/mbox  
is i-node  
60

```
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`

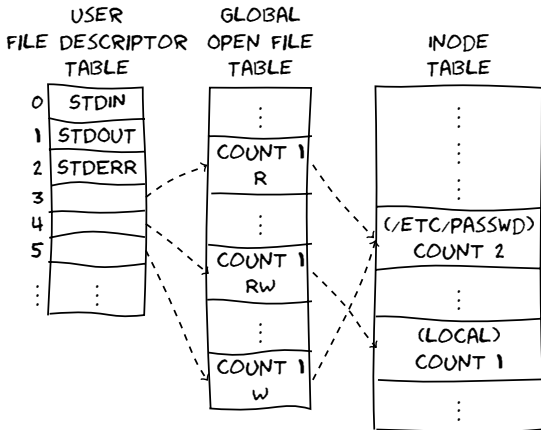
```
$ lsof -p 1
```

```
$ man 2 open
```



## A process executes the following code:

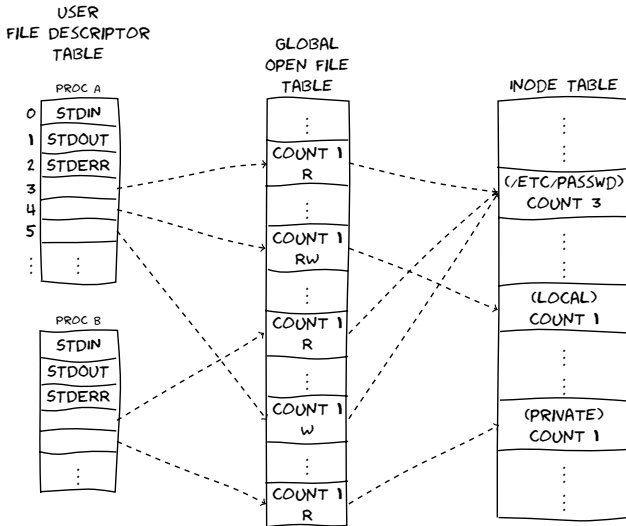
```
fd1 = open("/etc/passwd", O_RDONLY);  
fd2 = open("local", O_RDWR);  
fd3 = open("/etc/passwd", O_WRONLY);
```



## One more process B:

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

```
fd2 = open("private", O_RDONLY);
```

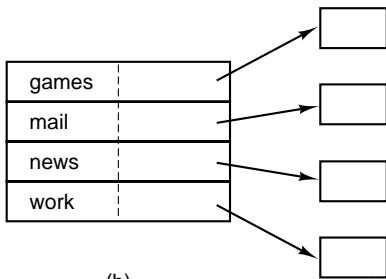


## 7 Directories

# Implementing Directories

games	attributes
mail	attributes
news	attributes
work	attributes

(a)



(b)

Data structure  
containing the  
attributes

## (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)

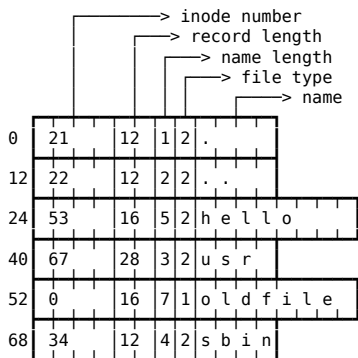
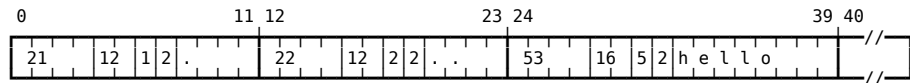
## Directory entry in `glibc`

```
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

```
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()

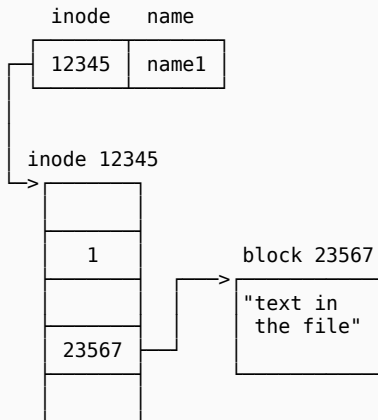
```
1  #include <sys/stat.h>
2  #include <sys/types.h>
3  #include <unistd.h>
4  #include <stdio.h>
5
6  int main(int argc, char *argv[])
7  {
8      char s[100];
9      if ( mkdir(argv[1], S_IRUSR|S_IXUSR) == 0 )
10         chdir(argv[1]);
11     printf("PWD = %s\n", getcwd(s,100));
12     rmdir(argv[1]);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



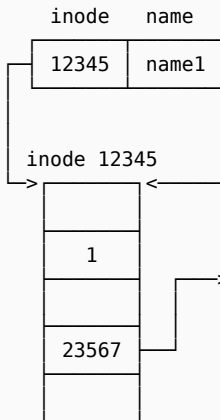
# Hard Links

## Hard links ➡ the same inode

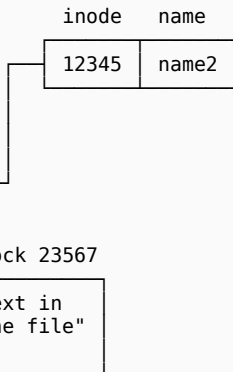
directory entry in /dirA



dir entry in /dirA

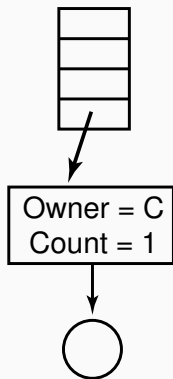


dir entry in /dirB



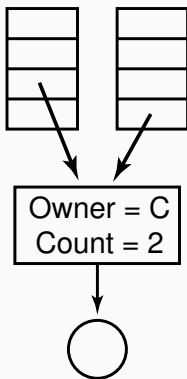
## Drawback

C's directory



(a)

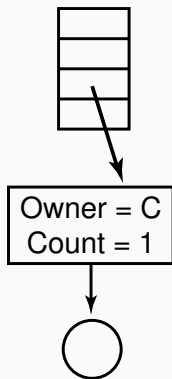
B's directory



(b)

C's directory

B's directory



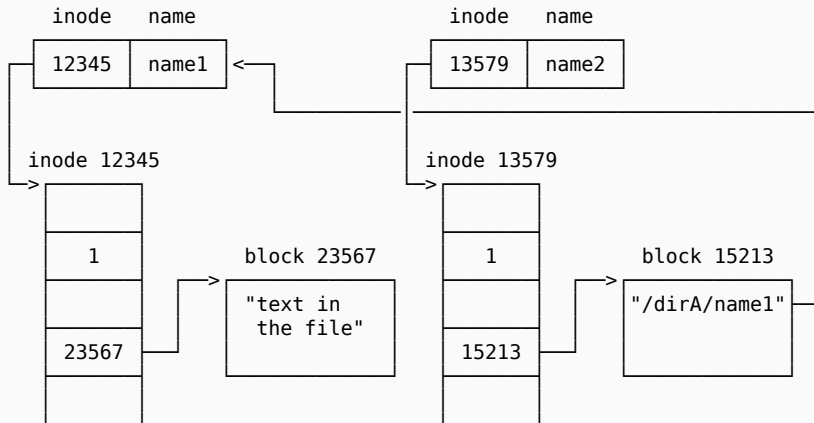
(c)

# Symbolic Links

A symbolic link has its own inode ➡ a directory entry

directory entry in /dirA

directory entry in /dirB



**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]); */
```

## Part V

# Processes and Threads

## 8 Virtual Memory

# Programs

A program is a file sitting in your hard disk. Two forms:

- ▶ Source code, e.g. `hello.c`, human readable
- ▶ Executable code, e.g. `a.out`, machine readable

Binary format identification Usually ELF

Machine-language instructions Program algorithm

Entry-point address Where to find `main()`?

Data Initialized variables

Symbol and relocation tables Address of variables, functions...

Shared-library Where to find `printf()`?

More ...

# Process

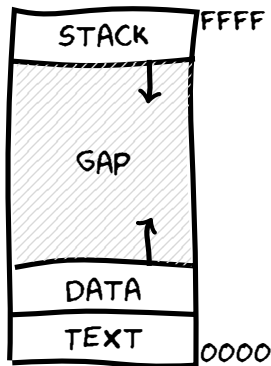
A **process** is an instance of a program in execution

## 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:

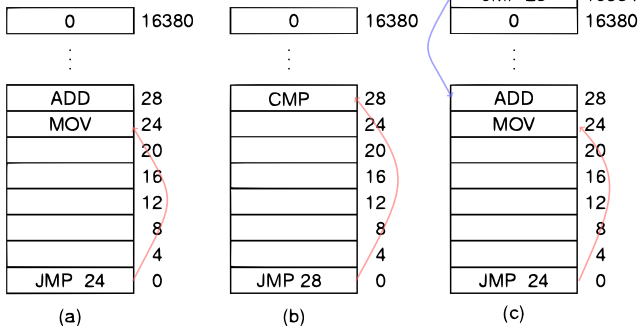
- ▶ sex is not really common among processes
- ▶ each process has just one parent





# Problem With Real Mode

EXPOSING  
PHYSICAL MEMORY  
TO  
PROCESSES  
IS NOT  
A GOOD IDEA



# Protected mode

We need

- ▶ Protect the OS from access by user programs
- ▶ Protect user programs from one another

**Protected mode** is an operational mode of x86-compatible CPU.

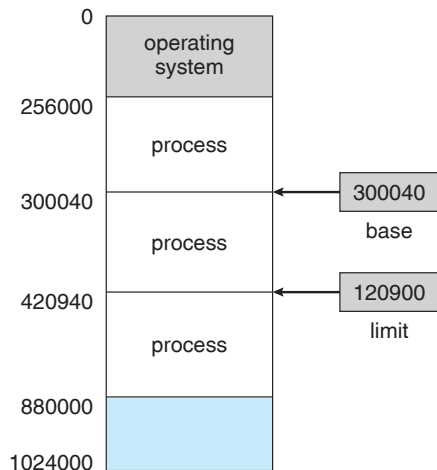
- ▶ The purpose is to protect everyone else (including the OS) from your program.

# Memory Protection

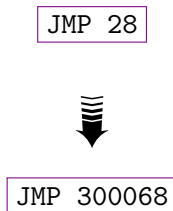
## Logical Address Space

**Base register** holds the smallest legal physical memory address

**Limit register** contains the size of the range

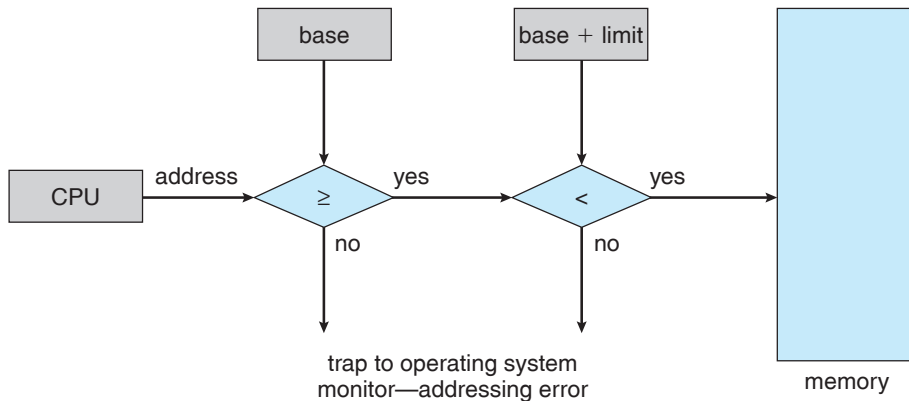


A pair of base and limit registers define the logical address space

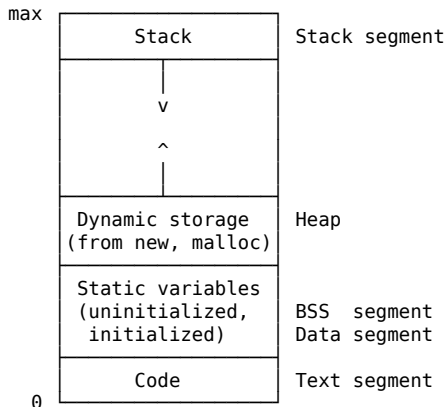


# Memory Protection

## Base and limit registers



# UNIX View of a Process' Memory



**text:** program code

**data:** initialized global and static data

**bss:** uninitialized global and static data

**heap:** dynamically allocated with malloc, new

**stack:** local variables

THE SIZE OF A PROCESS  
(TEXT + DATA + BSS) IS  
ESTABLISHED AT COMPILE TIME

# Stack vs. Heap

Stack	Heap
compile-time allocation	run-time allocation
auto clean-up	you clean-up
inflexible	flexible
smaller	bigger
quicker	slower

## How large is the ...

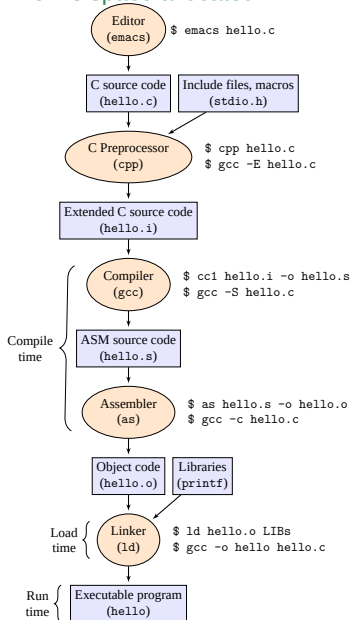
**stack:** `ulimit -s`

**heap:** could be as large as your virtual memory

**text|data|bss:** `size a.out`

# Multi-step Processing of a User Program

When is space allocated?



**Static:** before program start  
running

- Compile time
- Load time

**Dynamic:** as program runs

- Execution time

# Address Binding

Who assigns memory to segments?

Static-binding: before a program starts running

Compile time: **Compiler** and **assembler** generate an object file for each source file

Load time:

- ▶ **Linker** combines all the object files into a single executable object file
- ▶ **Loader** (part of OS) loads an executable object file into memory at location(s) determined by the OS
  - invoked via the `execve` system call

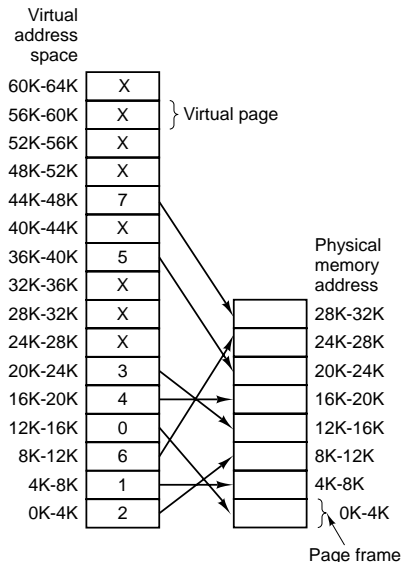
Dynamic-binding: as program runs

- ▶ Execution time:
  - ▶ uses `new` and `malloc` to dynamically allocate memory
  - ▶ gets space on stack during function calls



# Virtual Memory

Logical memory can be much larger than physical memory



## Address translation

*virtual address*  $\xrightarrow{\text{page table}}$  *physical address*

*Page 0*  $\xrightarrow{\text{map to}}$  *Frame 2*

$0_{\text{virtual}}$   $\xrightarrow{\text{map to}}$   $8192_{\text{physical}}$

$20500_{\text{vir}}$   $\xrightarrow{\text{map to}}$   $12308_{\text{phy}}$   
 $(20k + 20)_{\text{vir}}$   $\xrightarrow{\text{map to}}$   $(12k + 20)_{\text{phy}}$

# Paging

## Address Translation Scheme

Address generated by CPU is divided into:

Page number(p): an index into a page table

Page offset(d): to be copied into memory

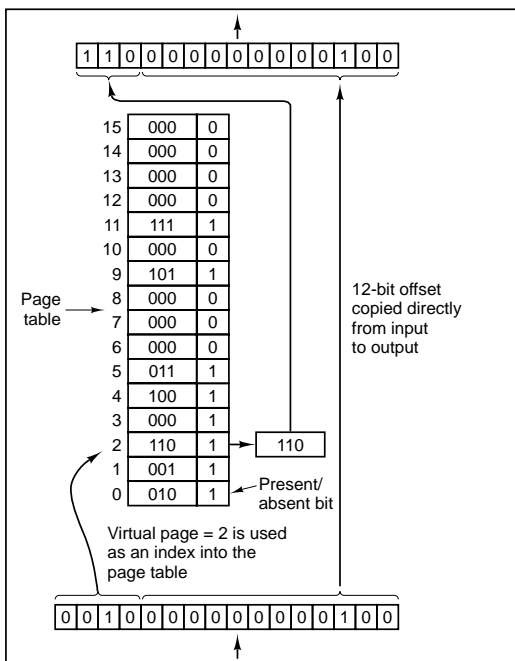
Given **logical address space** ( $2^m$ ) and **page size** ( $2^n$ ),

$$\text{number of pages} = \frac{2^m}{2^n} = 2^{m-n}$$

Example: addressing to 0010000000000100

$$\underbrace{\overbrace{0010}^{m-n=4} \overbrace{0000000000100}^{n=12}}_{m=16}$$

page number = 0010 = 2,    page offset = 000000000100

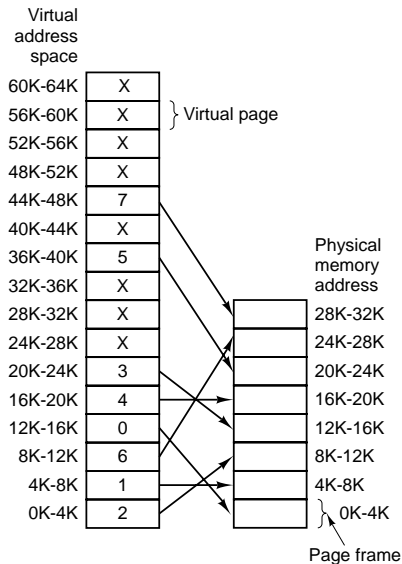


Outgoing  
physical  
address  
(24580)

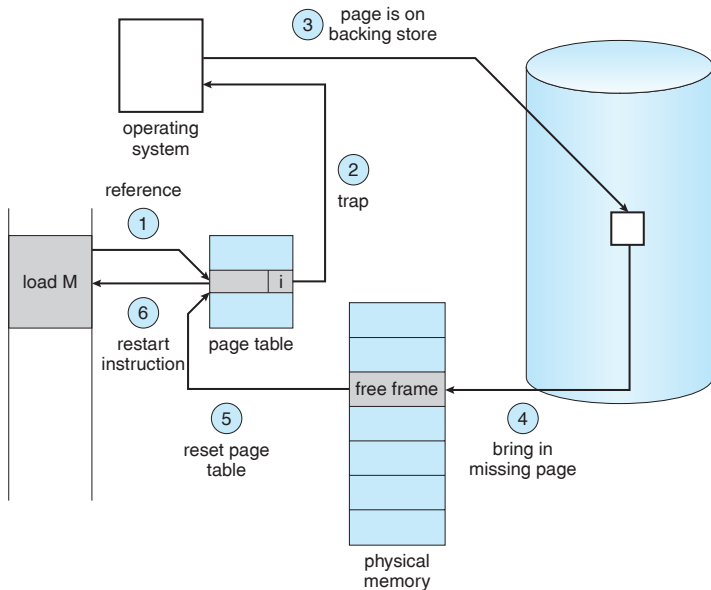
Virtual pages: 16  
Page size: 4k  
Virtual memory: 64K  
Physical frames: 8  
Physical memory: 32K

Incoming  
virtual  
address  
(8196)

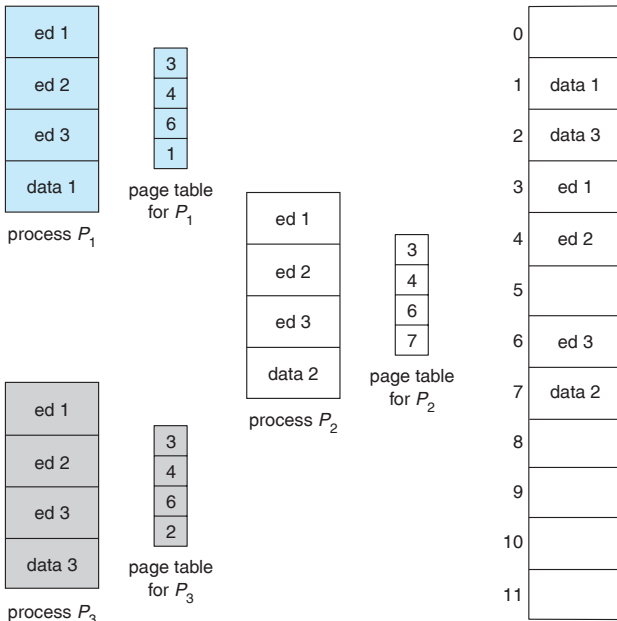
# Page Fault



# Page Fault Handling



# Shared Pages



# Page Table Entry

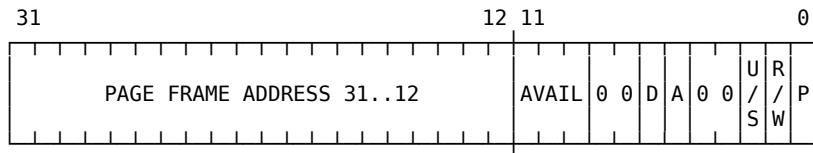
## Intel i386 Page Table Entry

- ▶ Commonly 4 bytes (32 bits) long
- ▶ Page size is usually 4k ( $2^{12}$  bytes). OS dependent

\$ getconf PAGESIZE

- ▶ Could have  $2^{32-12} = 2^{20} = 1M$  pages

Could address  $1M \times 4KB = 4GB$  memory



P – PRESENT

R/W – READ/WRITE

U/S – USER/SUPERVISOR

A – ACCESSED

D – DIRTY

AVAIL – AVAILABLE FOR SYSTEMS PROGRAMMER USE

NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.

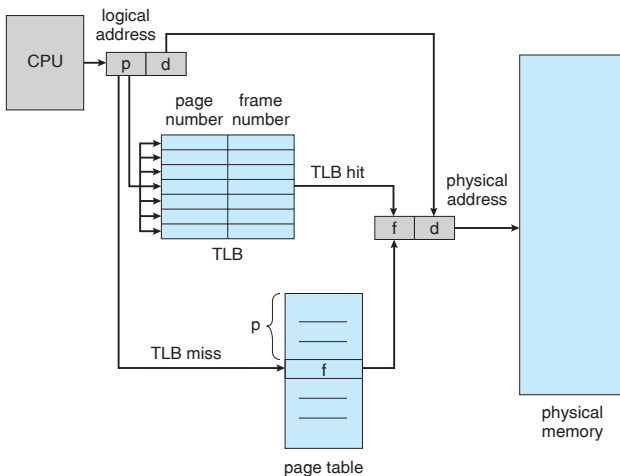
# Page Table

- ▶ Page table is kept in main memory
- ▶ Usually one page table for each process
- ▶ **Page-table base register (PTBR)**: A pointer to the page table is stored in PCB
- ▶ **Page-table length register (PRLR)**: indicates size of the page table
- ▶ Slow
  - ▶ Requires two memory accesses. One for the page table and one for the data/instruction.
- ▶ TLB



# Translation Lookaside Buffer (TLB)

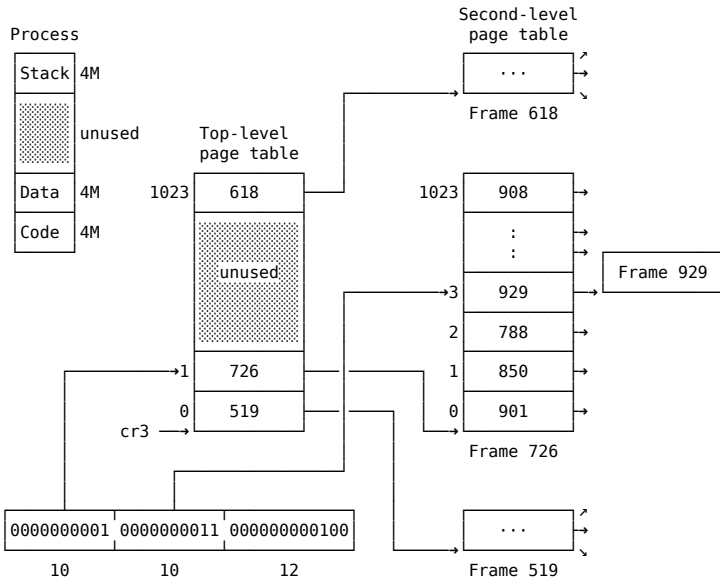
**80-20 rule** Only a small fraction of the PTEs are heavily read; the rest are barely used at all





# Two-Level Page Tables

## Example



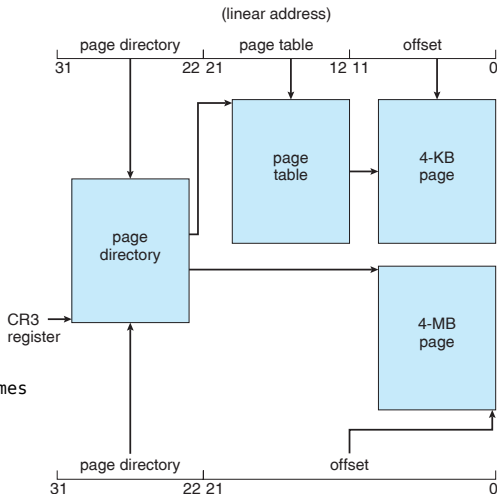
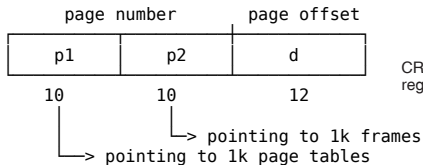
# Pentium Paging

Linear Address  $\Rightarrow$  Physical Address

Two page size in Pentium:

4K: 2-level paging

4M: 1-level paging



# Problem With 64-bit Systems

- Given:
- ▶ virtual address space = 64 *bits*
  - ▶ page size = 4 *KB* =  $2^{12}$  *B*

? How much space would a simple single-level page table take?

if Each page table entry takes 4 *Bytes*  
then The whole page table ( $2^{64-12}$  entries) will take

$$2^{64-12} \times 4 \text{ B} = 2^{54} \text{ B} = 16 \text{ PB} \quad (\text{peta} \Rightarrow \text{tera} \Rightarrow \text{giga})!$$

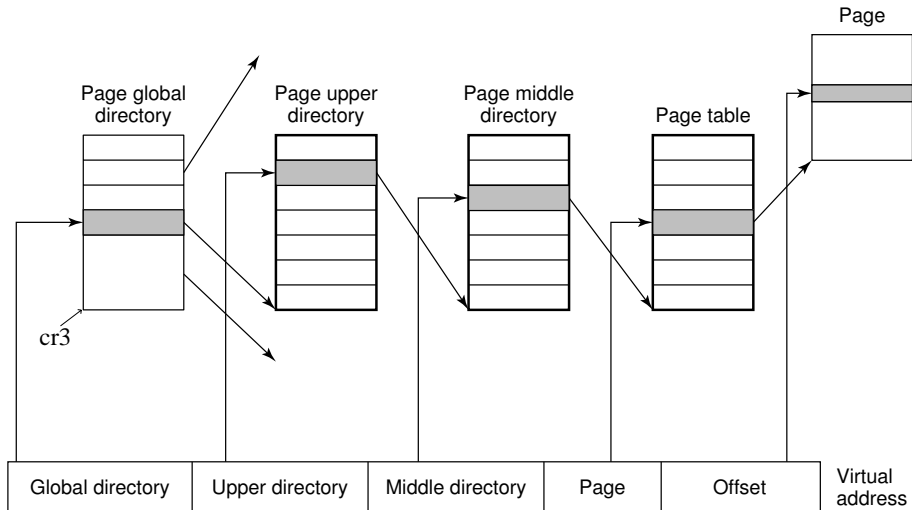
And this is for ONE process!

Multi-level?

if 10 *bits* for each level  
then  $\frac{64-12}{10} = 5$  levels are required  
5 memory access for each address translation!

# Paging In Linux

4-level paging for both 32-bit and 64-bit



## 4-level paging for both 32-bit and 64-bit

### ► 64-bit: four-level paging

1. Page Global Directory
2. Page Upper Directory
3. Page Middle Directory
4. Page Table

### ► 32-bit: two-level paging

1. Page Global Directory
2. Page Upper Directory — 0 bits; 1 entry
3. Page Middle Directory — 0 bits; 1 entry
4. Page Table

The same code can work on 32-bit and 64-bit architectures

Arch	Page size	Address bits	Paging levels	Address splitting
x86	4KB(12bits)	32	2	10 + 0 + 0 + 10 + 12
x86-PAE	4KB(12bits)	32	3	2 + 0 + 9 + 9 + 12
x86-64	4KB(12bits)	48	4	9 + 9 + 9 + 9 + 12

## 9 Process



## From kernel's point of view

A process consists of


User-space memory program code, variable...

Kernel data structures keep the state of the process

# Process Control Block (PCB)

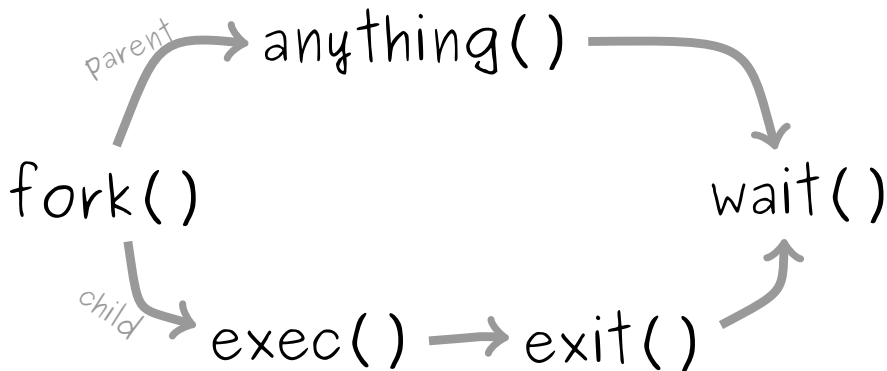
## Implementation

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 

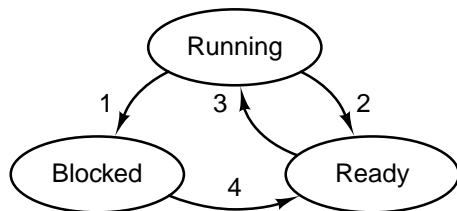
process state
PID
program counter
registers
memory limits
list of open files
...

## Process Creation



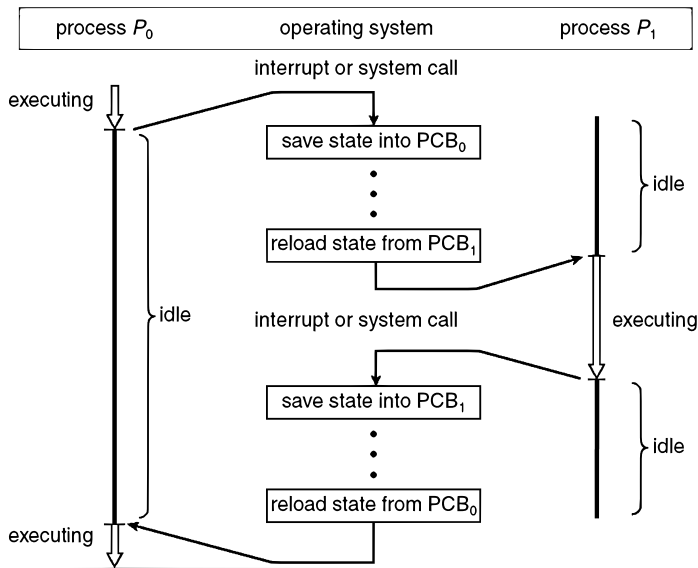
- ▶ When a process is created, it is almost identical to its parent
  - ▶ It receives a (logical) copy of the parent's address space, and
  - ▶ executes the same code as the parent
- ▶ The parent and child have separate copies of the data (stack and heap)

# 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



# 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 }
```

## 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 }
```

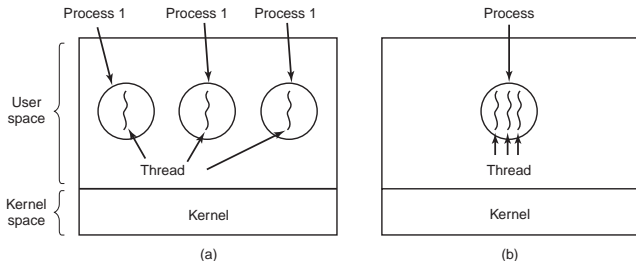
## 10 Thread



# 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

```
void *thread_function(void *arg) {
    int i;
    for ( i=0; i<10; i++ ) {
        printf("Thread says hi!, %d\n",i);
        sleep(1);
    }
    return NULL;
}

int main(void)
{
    pthread_t mythread;

    if( pthread_create(&mythread, NULL, thread_function, NULL) ) {}

    printf("Can you see my thread working?\n");

    if( pthread_join ( mythread, NULL ) ) {}

    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

```
#define NUMBER_OF_THREADS 5

void *hello(void *tid)
{
    printf ("Hello from thread %d\n", *(int*)tid);
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t t[NUMBER_OF_THREADS];
    int status, i;

    for (i=0; i<NUMBER_OF_THREADS; i++){
        printf("Main: creating thread %d ...", i);

        if( (status = pthread_create(&t[i], NULL, hello, (void *)&i)) ){
            puts("done.");
        }

        for (i=0; i<NUMBER_OF_THREADS; i++){
            printf("Joining thread %d ...", i);

            if( pthread_join(t[i], NULL) ){
                puts("done.");
            }
        }
        exit(0);
    }
```

# Linux Threads

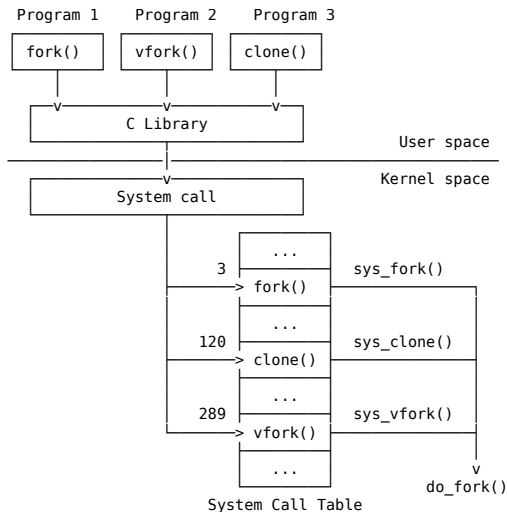
## To the Linux kernel, there is no concept of a thread

- ▶ 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:  $1.7\mu\text{s}$   
                                 fork:  $1.8\mu\text{s}$

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

## clone() Example

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



# Stack Grows Downwards

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



## 11 Signals

# Signals

- ▶ Signals are software interrupts
- ▶ Every signal has a name (SIGXXXX)
- ▶ One process can send a signal to another process

## Sending signals

```
$ [Ctrl]+[c], [Ctrl]+[z], ...
```

```
$ kill -signal <pid>
```

## Trapping signals

```
#!/ trap <command> <signals>
```

# Trap

```
1  #!/bin/bash
2
3  sigint(){
4      echo -e "Why Ctrl-c?\n-> "
5  }
6
7  trap sigint SIGINT
8
9  echo -n "-> "
10
11 while read CMD; do
12     $CMD
13     echo -n "-> "
14 done
```

```
#!/ trap "rm -rf $tmpfiles" EXIT
```



# Example

## SIGINT

```
#define MAXLINE 4096

void sig_int(int signo)
{
    printf("Why Ctrl-c?\n-> ");
}

int main(void)
{
    char   buf[MAXLINE];
    pid_t  pid;
    int     status;

    if (signal(SIGINT, sig_int) == SIG_ERR){}
    printf("-> ");

    while( fgets(buf, MAXLINE, stdin) != NULL ) {
        buf[strlen(buf) - 1] = '\0'; /* null */

        if ( (pid = fork()) == 0 ) { /* child */
            execlp(buf, buf, (char*)0);
            perror("execlp");
            exit(127);
        }

        if( (pid = waitpid(pid, &status, 0)) < 0 ) perror("waitpid");
        printf("-> ");
    }
    exit(EXIT_SUCCESS);
}
```

# Example

## SIGUSR1

```
void sig_usr(int);

int main(void)
{
    printf("PID = %d\n", getpid());

    if( signal(SIGUSR1, sig_usr) == SIG_ERR ){}

    for(;;) pause();
}

void sig_usr(int signo)
{
    if (signo == SIGUSR1)
        puts("received SIGUSR1.");
    else{}
}
```

```
$ kill -USR1 <PID>
```

# Example

## SIGALRM

```
1  #include <signal.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <stdlib.h>
5
6  void cry(int sig)
7  {
8      puts("C: I'm crying...");
9      kill(getppid(),sig);
10 }
11
12 void complain(int sig)
13 {
14     puts("P: You're noisy.");
15 }
16
17 int main()
18 {
19     if ( fork() == 0 ){
20         signal(SIGALRM, cry);
21         alarm(2);
22         pause();
23     }
24
25     signal(SIGALRM, complain);
26     pause();
27     exit(0);
28 }
```

## Part VI

# Interprocess Communication

# Interprocess Communication

## Example:

```
$ unicode skull | head -1 | cut -f1 -d' ' | sm -
```

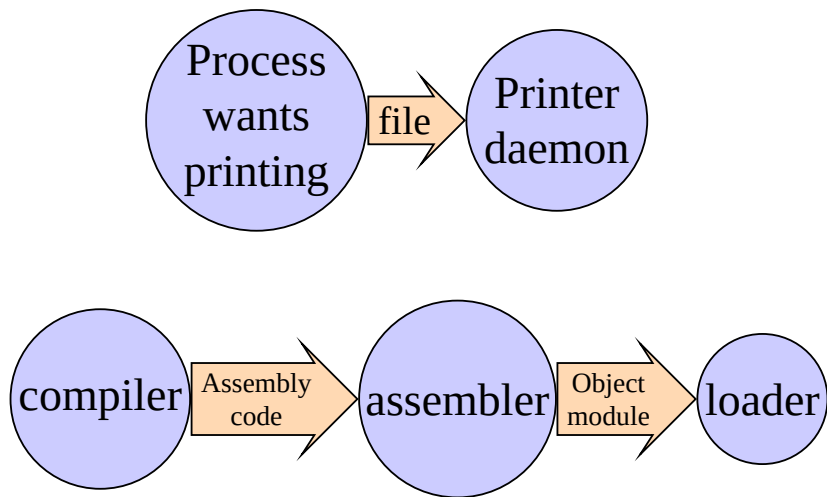
## IPC issues:

1. How one process can pass information to another
2. Be sure processes do not get into each other's way  
e.g. in an airline reservation system, two processes compete for the last seat
3. Proper sequencing when dependencies are present  
e.g. if A produces data and B prints them, B has to wait until A has produced some data

## Two models of IPC:

- ▶ Shared memory
- ▶ Message passing (e.g. sockets)

## Producer-Consumer Problem



# Producer-Consumer Problem

- ▶ Consumers don't try to remove objects from Buffer when it is empty.
- ▶ Producers don't try to add objects to the Buffer when it is full.

```
1 while(TRUE){  
2     while(FULL);  
3     item = produceItem();  
4     insertItem(item);  
5 }
```

```
1 while(TRUE){  
2     while(EMPTY);  
3     item = removeItem();  
4     consumeItem(item);  
5 }
```

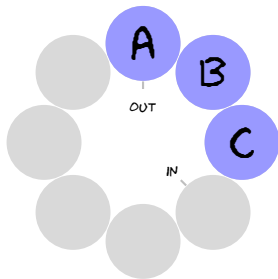
How to define full/empty?

# Bounded-Buffer Problem (Circular Array)

Front(out): the first full position

Rear(in): the next free position

Full or empty when “*front == rear*”?





## Common solution:

**Full:** when `“(in + 1)%BUFFER_SIZE == out”`

Actually, this is `“full - 1”`

**Empty:** when `“in == out”`

Can only use `“BUFFER_SIZE - 1”` elements

## Shared data:

```
1  #define BUFFER_SIZE 6
2  typedef struct {
3      ...
4  } item;
5  item buffer[BUFFER_SIZE];
6  int in = 0; //the next free position
7  int out = 0; //the first full position
```

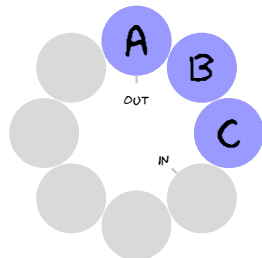
# Bounded-Buffer Problem

## Producer:

```
1  while (true) {  
2      /* do nothing -- no free buffers */  
3      while (((in + 1) % BUFFER_SIZE) == out);  
4  
5      produce(buffer[in]);  
6  
7      in = (in + 1) % BUFFER_SIZE;  
8  }
```

## Consumer:

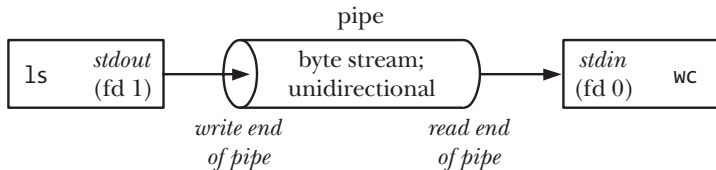
```
1  while (true) {  
2      while (in == out); /* do nothing */  
3  
4      consume(buffer[out]);  
5  
6      out = (out + 1) % BUFFER_SIZE;  
7  }
```



## 12 Pipes and FIFOs

# Pipe

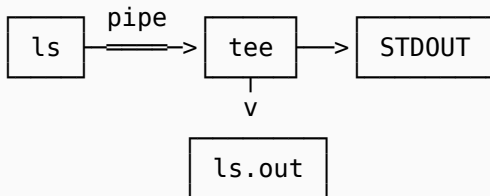
```
$ ls | wc -l
```

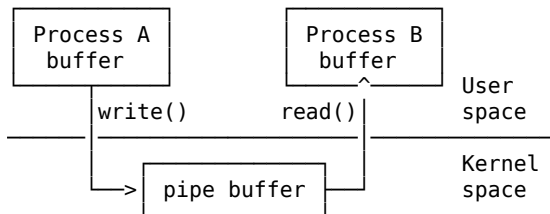


- ▶ A pipe is a byte stream
- ▶ Unidirectional
- ▶ `read()` would be blocked if nothing written at the other end

## tee

```
$ ls | tee ls.out
```



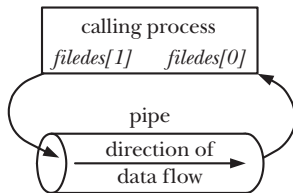


- ▶ No direct link between A and B (need system calls)
- ▶ A pipe is simply a buffer maintained in kernel memory

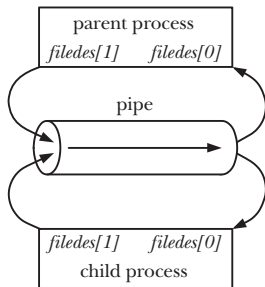
```
$ cat /proc/sys/fs/pipe-max-size
```

# pipe()

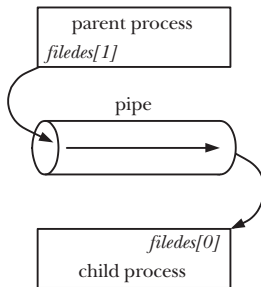
```
1  #include <unistd.h>
2
3  int pipe(int fd[2]);
```



## pipe() + fork()



a) After *fork()*



b) After closing unused descriptors

```

#define BUF_SIZE 10

int main(int argc, char *argv[]) /* Over-simplified! */
{
    int pfd[2]; /* Pipe file descriptors */
    char buf[BUF_SIZE];
    ssize_t numRead;

    pipe(pfd); /* Create the pipe */

    switch (fork()) {
    case 0: /* Child - reads from pipe */
        close(pfd[1]); /* Write end is unused */

        for(;;) { /* Read data from pipe, echo on stdout */
            if( (numRead = read(pfd[0], buf, BUF_SIZE)) == 0 )
                break; /* End-of-file */
            if( write(1, buf, numRead) != numRead ){}
        }
        puts(""); /* newline */

        close(pfd[0]); _exit(EXIT_SUCCESS);

    default: /* Parent - writes to pipe */
        close(pfd[0]); /* Read end is unused */

        if( (size_t)write(pfd[1], argv[1], strlen(argv[1])) != strlen(argv[1]) ){}

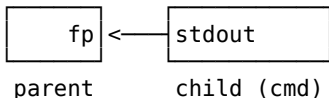
        close(pfd[1]); /* Child will see EOF */

        wait(NULL); /* Wait for child to finish */
        exit(EXIT_SUCCESS);
    }
}

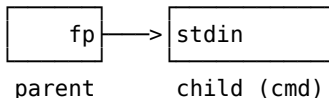
```

## popen()

```
fp = popen(cmd, "r");
```



```
fp = popen(cmd, "w");
```



`popen()` does a `fork()` and `exec()` to execute the `cmd` and returns STD I/O file pointer.

- `r` `fp` is readable (`stdout`)

- `w` `fp` is writable (`stdin`)



## Example

```
int main()
{
    FILE *fp;
    char buf[1025];
    int rc;

    memset(buf, '\\0', sizeof(buf));

    if( (fp = popen("ps ax", "r")) != NULL ) {
        rc = fread(buf, sizeof(char), 1024, fp);
        while (rc > 0) {
            buf[rc - 1] = '\\0';
            printf("Reading %d:-\\n %s\\n", 1024, buf);
            rc = fread(buf, sizeof(char), 1024, fp);
        }
        pclose(fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

\$ ps ax | cat

## Example

```
int main(int argc, char *argv[])
{
    FILE *fp;
    char buf[BUFSIZ + 1];

    sprintf(buf, argv[1]);

    if( (fp = popen("od -c", "w")) != NULL ) {
        fwrite(buf, sizeof(char), strlen(buf), fp);
        pclose(fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

```
$ echo -n hello | od -c
```

# Named Pipe (FIFO)

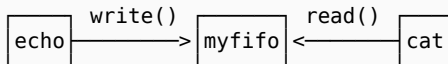
**PIPEs** pass data between related processes.

**FIFOs** pass data between any processes.

```
$ mkfifo myfifo
```

```
$ echo hello > myfifo
```

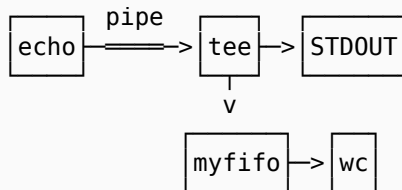
```
$ cat myfifo
```



## tee

```
$ echo hello | tee myfifo
```

```
$ wc myfifo
```



# IPC With FIFO

```
#define FIFO_NAME "/tmp/myfifo"

int main(int argc, char *argv[]) /* Oversimplified */
{
    int fd, i, mode = 0;
    char c;

    if (argc < 2) {}

    for(i = 1; i < argc; i++) {
        if (strcmp(++argv, "O_RDONLY", 8) == 0) mode |= O_RDONLY;
        if (strcmp(*argv, "O_WRONLY", 8) == 0) mode |= O_WRONLY;
        if (strcmp(*argv, "O_NONBLOCK", 10) == 0) mode |= O_NONBLOCK;
    }

    if (access(FIFO_NAME, F_OK) == -1) mkfifo(FIFO_NAME, 0777);

    printf("Process %d: FIFO(fd %d, mode %d) opened.\n",
        getpid(), fd = open(FIFO_NAME, mode), mode);

    if( (mode == 0) | (mode == 2048) )
        while( read(fd,&c,1) == 1 ) putchar(c);

    if( (mode == 1) | (mode == 2049) )
        while( (c = getchar()) != EOF ) write(fd,&c,1);

    exit(EXIT_SUCCESS);
}
```

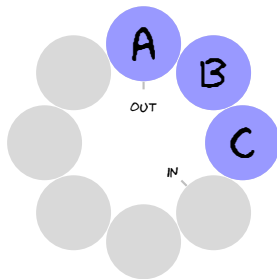
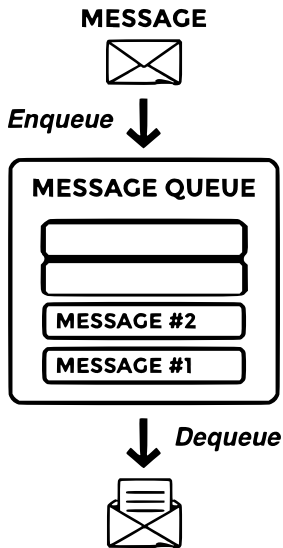
```
$ watch 'lsof -n.1 /tmp/myfifo'
$ ./a.out O_RDONLY
$ ./a.out O_WRONLY
$ ./a.out O_RDONLY O_NONBLOCK
$ ./a.out O_WRONLY O_NONBLOCK
```

## O\_NONBLOCK

- ▶ A read()/write() will wait on an empty blocking FIFO
- ▶ A read() on an empty nonblocking FIFO will return 0 bytes
- ▶ open(const char \*path, O\_WRONLY | O\_NONBLOCK);
  - ▶ Returns an error (-1) if FIFO not open
  - ▶ Okay if someone's reading the FIFO
- ▶ If opened with O\_RDWR, the result is undefined

## 13 Message Queues

# Message Queues



# Message Queues

## Send

```
int main(int argc, char **argv)
{
    mqd_t queue;
    struct mq_attr attrs;
    size_t msg_len;

    if (argc < 3){}

    queue = mq_open(argv[1], O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR, NULL);
    if (queue == (mqd_t)-1){}

    if (mq_getattr(queue, &attrs) == -1){}

    msg_len = strlen(argv[2]);
    if (msg_len > LONG_MAX || (long)msg_len > attrs.mq_msgsize){}

    if (mq_send(queue, argv[2], strlen(argv[2]), 0) == -1){}

    return 0;
}
```



# Message Queues

## Receive

```
int main(int argc, char **argv)
{
    mqd_t queue;
    struct mq_attr attrs;
    char *msg_ptr;
    ssize_t recvd;
    size_t i;

    if (argc < 2){}

    queue = mq_open(argv[1], O_RDONLY | O_CREAT, S_IRUSR | S_IWUSR, NULL);
    if (queue == (mqd_t)-1){}

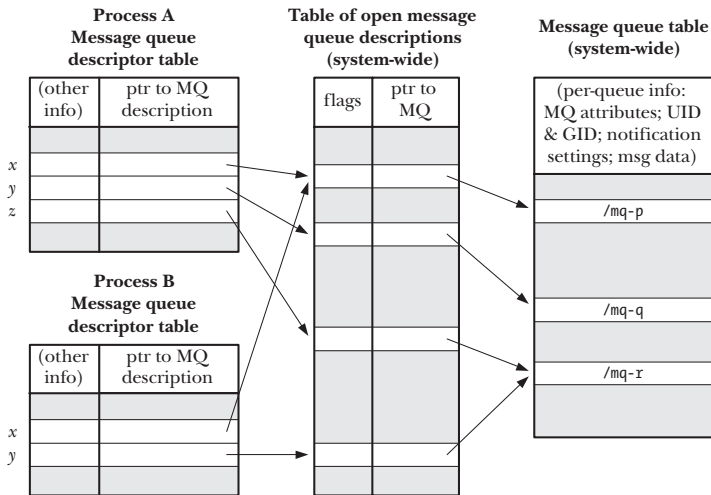
    if (mq_getattr(queue, &attrs) == -1){}

    msg_ptr = calloc(1, attrs.mq_msgsize);
    if (msg_ptr == NULL){}

    recvd = mq_receive(queue, msg_ptr, attrs.mq_msgsize, NULL);
    if (recvd == -1){}

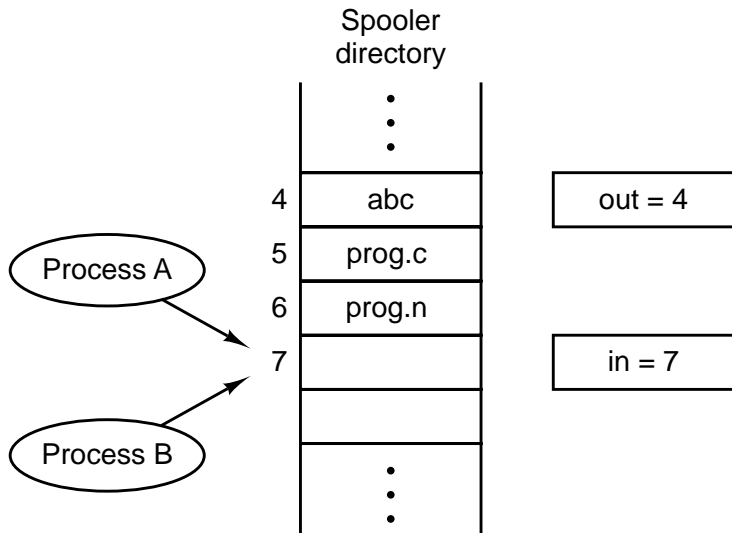
    printf("Message: ");
    for (i = 0; i < (size_t)recvd; i++)
        putchar(msg_ptr[i]);
    puts("");
}
```

# Relationship Between Kernel Data Structures



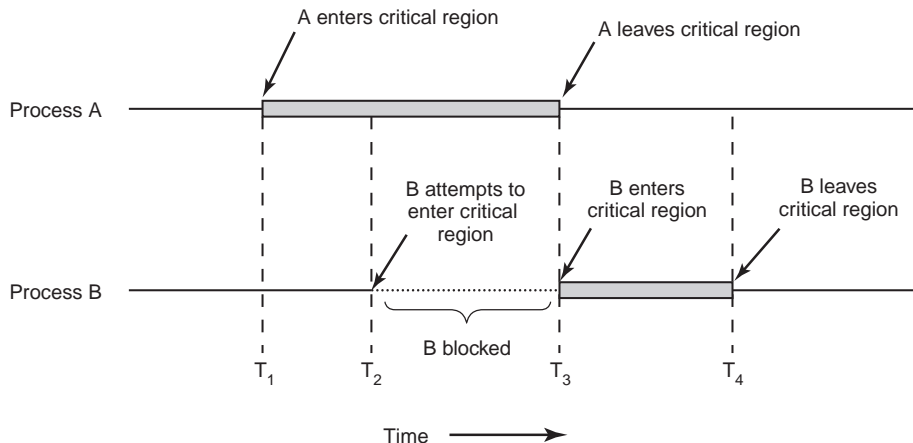
## 14 Semaphores

# Race Conditions



# Mutual Exclusion

**Critical Region** is a piece of code accessing a common resource.



## A solution to the critical region problem must satisfy three conditions

**Mutual Exclusion:** No two processes may be simultaneously inside their critical regions.

**Progress:** No process running outside its critical region may block other processes.

**Bounded Waiting:** No process should have to wait forever to enter its critical region.

# Mutual Exclusion With Busy Waiting

## Strict Alternation

```
1 while(TRUE){  
2     while(turn != 0);  
3     critical_region();  
4     turn = 1;  
5     noncritical_region();  
6 }
```

```
1 while(TRUE){  
2     while(turn != 1);  
3     critical_region();  
4     turn = 0;  
5     noncritical_region();  
6 }
```

☹ One process can be blocked by another not in its critical region

# Mutual Exclusion With Busy Waiting

## Peterson's Solution

```
1 | int interest[0] = 0;  
2 | int interest[1] = 0;  
3 | int turn;
```

**P0**

```
1 | interest[0] = 1;  
2 | turn = 1;  
3 | while(interest[1] == 1  
4 |         && turn == 1);  
5 | critical_section();  
6 | interest[0] = 0;
```

**P1**

```
1 | interest[1] = 1;  
2 | turn = 0;  
3 | while(interest[0] == 1  
4 |         && turn == 0);  
5 | critical_section();  
6 | interest[1] = 0;
```



Wikipedia. *Peterson's algorithm* — Wikipedia, The Free Encyclopedia. 2015.



# Mutual Exclusion With Busy Waiting

Lock file

```
const char *mylock = "/tmp/LCK.test2";

int main() {
    int fd;

    for(;;){
        while( (fd = open(mylock, O_RDWR | O_CREAT | O_EXCL, 0444)) != -1 ){
            printf("Process(%d) - Working in critical region...\n", getpid());
            sleep(2);                /* working */
            close(fd);
            if ( unlink(mylock) == 0 ) puts("Done.\nResource unlocked.");
            sleep(3);                /* non-critical region */
        }
        printf("Process(%d) - Waiting for lock...\n", getpid());
    }
    exit(EXIT_SUCCESS);
}
```

☹ Lock file could be left in system after Ctrl + c



```
const char *mylock = "/tmp/LCK.test2";

void sigint(int signo){
    if ( unlink(mylock) == 0 ) puts("Quit. Lock released.");
    exit(EXIT_SUCCESS);
}

int main() {
    int fd;

    signal(SIGINT,sigint);

    for(;;){
        while( (fd = open(mylock, O_RDWR | O_CREAT | O_EXCL, 0444)) != -1 ) {
            printf("Process(%d) - Working in critical region...\n", getpid());
            sleep(2);           /* working */
            close(fd);
            if ( unlink(mylock) == 0 ) puts("Done.\nResource unlocked.");
            sleep(3);           /* non-critical region */
        }
        printf("Process(%d) - Waiting for lock...\n", getpid());
    }
    exit(EXIT_SUCCESS);
}
```

# What is a Semaphore?

- ▶ A locking mechanism
- ▶ An integer or ADT

```
1 down(S){  
2   while(S<=0);  
3   S--;  
4 }
```

```
1 up(S){  
2   S++;  
3 }
```

<i>Atomic Operations</i>	
P()	V()
Wait()	Signal()
Down()	Up()
Decrement()	Increment()
...	...

More meaningful names:

- ▶ `increment_and_wake_a_waiting_process_if_any()`
- ▶ `decrement_and_block_if_the_result_is_negative()`

# Using Semaphore For Signaling

- ▶ One thread sends a signal to another to indicate that something has happened
- ▶ It solves the serialization problem

Signaling makes it possible to guarantee that a section of code in one thread will run before that in another

1		statement a1	1		sem.wait()
2		sem.signal()	2		statement b1

What's the initial value of `sem`?

# Example

```
void *func(void *arg);
sem_t sem;

#define BUFSIZE 1024
char buf[BUFSIZE];

int main() {
    pthread_t t;

    if( sem_init(&sem, 0, 0) != 0 ) {}

    if( pthread_create(&t, NULL, func, NULL) != 0 ) {}

    puts("Please input some text. Ctrl-d to quit.");

    while( fgets(buf, BUFSIZE, stdin) )
        sem_post(&sem);

    sem_post(&sem);           /* in case of Ctrl-d */

    if( pthread_join(t, NULL) != 0 ) {}

    sem_destroy(&sem);

    exit(EXIT_SUCCESS);
}

void *func(void *arg) {
    sem_wait(&sem);
    while( buf[0] != '\0' ) {
        printf("You input %ld characters\n", strlen(buf)-1);
        buf[0] = '\0';      /* in case of Ctrl-d */
        sem_wait(&sem);
    }
    pthread_exit(NULL);
}
```

## i++ can go wrong!

```
static int glob = 0;

static void *threadFunc(void *arg) /* loop 'arg' times */
{
    int j;
    for (j = 0; j < *((int *) arg); j++) glob++; /* not atomic! */
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t t1, t2;
    int loops;

    loops = (argc > 1) ? atoi(argv[1]) : 10000000;

    if( pthread_create(&t1, NULL, threadFunc, &loops) != 0 ){}

    if( pthread_create(&t2, NULL, threadFunc, &loops) != 0 ){}

    if( pthread_join(t1, NULL) != 0 ){}

    if( pthread_join(t2, NULL) != 0 ){}

    printf("glob = %d\n", glob);
    exit(EXIT_SUCCESS);
}
```

# Atomic

## i++ is not atomic in assembly language

```
1  LOAD    [i], r0    ;load the value of 'i' into
2                      ;a register from memory
3  ADD     r0, 1       ;increment the value
4                      ;in the register
5  STORE   r0, [i]     ;write the updated
6                      ;value back to memory
```

Interrupts might occur in between. So, i++ needs to be protected with a mutex.

**Mutex** A semaphore that is initialized to 1. In case of:

**1:** A thread may proceed and access the shared variable

**0:** It has to wait for another thread to release the mutex

```
1 | mutex.wait()  
2 |     i++  
3 | mutex.signal()
```

```
1 | mutex.wait()  
2 |     i++  
3 | mutex.signal()
```



```

static int glob = 0;
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

static void *threadFunc(void *arg)
{
    int j;

    for (j = 0; j < *((int *) arg); j++) {
        if ( pthread_mutex_lock(&mtx) != 0 ){}
        glob++;
        if ( pthread_mutex_unlock(&mtx) != 0 ){}
    }

    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t t1, t2;
    int loops;

    loops = (argc > 1) ? atoi(argv[1]) : 10000000;

    if( pthread_create(&t1, NULL, threadFunc, &loops) != 0 ){}

    if( pthread_create(&t2, NULL, threadFunc, &loops) != 0 ){}

    if( pthread_join(t1, NULL) != 0 ){}

    if( pthread_join(t2, NULL) != 0 ){}

    printf("glob = %d\n", glob);
    exit(EXIT_SUCCESS);
}

```

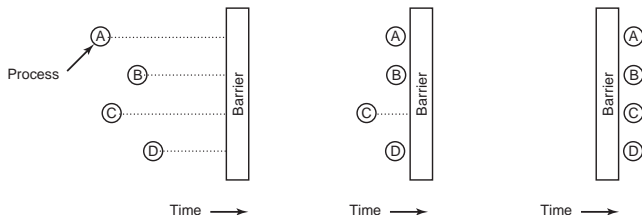
```
static int glob = 0;
static sem_t sem;

static void *threadFunc(void *arg)
{
    int j;

    for (j = 0; j < *((int *) arg); j++) {
        if (sem_wait(&sem) == -1) {}
        glob++;
        if (sem_post(&sem) == -1) {}
    }

    return NULL;
}
```

# Barrier



1. Processes approaching a barrier
2. All processes but one blocked at the barrier
3. When the last process arrives at the barrier, all of them are let through

## Synchronization requirement:

```
specific_task()  
critical_point()
```

No thread executes `critical_point()` until after all threads have executed `specific_task()`.

# Barrier Solution

```
1  n = the number of threads
2  count = 0
3  mutex = Semaphore(1)
4  barrier = Semaphore(0)
```

**count:** keeps track of how many threads have arrived

**mutex:** provides exclusive access to count

**barrier:** is locked ( $\leq 0$ ) until all threads arrive

When `barrier.value < 0`,

`barrier.value == Number of queueing processes`

```
1  specific_task();
2  mutex.wait();
3      count++;
4  mutex.signal();
5  if (count < n)
6      barrier.wait();
7  barrier.signal();
8  critical_point();
```

```
1  specific_task();
2  mutex.wait();
3      count++;
4  mutex.signal();
5  if (count == n)
6      barrier.signal();
7  barrier.wait();
8  critical_point();
```

# Barrier Solution

```
1 n = the number of threads
2 count = 0
3 mutex = Semaphore(1)
4 barrier = Semaphore(0)
```

**count:** keeps track of how many threads have arrived

**mutex:** provides exclusive access to count

**barrier:** is locked ( $\leq 0$ ) until all threads arrive

When `barrier.value < 0`,

`barrier.value == Number of queueing processes`

```
1 specific_task();
2 mutex.wait();
3   count++;
4 mutex.signal();
5 if (count < n)
6   barrier.wait();
7 barrier.signal();
8 critical_point();
```

```
1 specific_task();
2 mutex.wait();
3   count++;
4 mutex.signal();
5 if (count == n)
6   barrier.signal();
7 barrier.wait();
8 critical_point();
```

Only one thread can pass the barrier!

## Barrier Solution

```
1  specific_task();
2
3  mutex.wait();
4      count++;
5  mutex.signal();
6
7  if (count == n)
8      barrier.signal();
9
10 barrier.wait();
11 barrier.signal();
12
13 critical_point();
```

```
1  specific_task();
2
3  mutex.wait();
4      count++;
5
6      if (count == n)
7          barrier.signal();
8
9          barrier.wait();
10         barrier.signal();
11 mutex.signal();
12
13 critical_point();
```

## Barrier Solution

```
1  specific_task();
2
3  mutex.wait();
4      count++;
5  mutex.signal();
6
7  if (count == n)
8      barrier.signal();
9
10 barrier.wait();
11 barrier.signal();
12
13 critical_point();
```

```
1  specific_task();
2
3  mutex.wait();
4      count++;
5
6      if (count == n)
7          barrier.signal();
8
9      barrier.wait();
10     barrier.signal();
11     mutex.signal();
12
13     critical_point();
```

💀 Blocking on a semaphore while holding a mutex! 💀

```
barrier.wait();  
barrier.signal();
```

## Turnstile

This pattern, a `wait` and a `signal` in rapid succession, occurs often enough that it has a name called a *turnstile*, because

- ▶ it allows one thread to pass at a time, and
- ▶ it can be locked to bar all threads

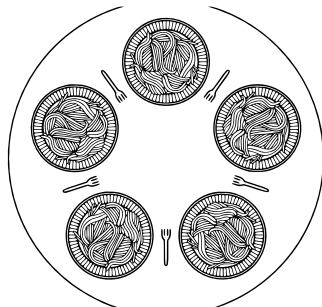


## 15 Classical IPC Problems

## 15.1 The Dining Philosophers Problem

# The Dining Philosophers Problem

```
1 while True:
2     think()
3     get_forks()
4     eat()
5     put_forks()
```



How to implement `get_forks()` and `put_forks()` to ensure

1. No deadlock
2. No starvation
3. Allow more than one philosopher to eat at the same time

# The Dining Philosophers Problem

## Deadlock

```
#define N 5                                /* number of philosophers */

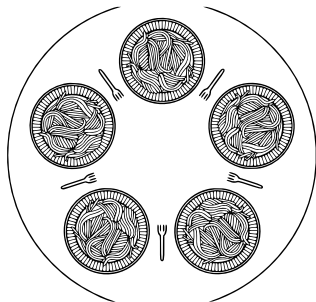
void philosopher(int i)                    /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think();                          /* philosopher is thinking */
        take_fork(i);                      /* take left fork */
        take_fork((i+1) % N);              /* take right fork; % is modulo operator */
        eat();                             /* yum-yum, spaghetti */
        put_fork(i);                       /* put left fork back on the table */
        put_fork((i+1) % N);               /* put right fork back on the table */
    }
}
```

- Put down the left fork and wait for a while if the right one is not available? Similar to CSMA/CD — Starvation

# The Dining Philosophers Problem

## With One Mutex

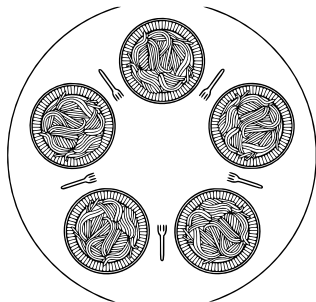
```
1  #define N 5
2  semaphore mutex=1;
3
4  void philosopher(int i)
5  {
6      while (TRUE) {
7          think();
8          wait(&mutex);
9          take_fork(i);
10         take_fork((i+1) % N);
11         eat();
12         put_fork(i);
13         put_fork((i+1) % N);
14         signal(&mutex);
15     }
16 }
```



# The Dining Philosophers Problem

## With One Mutex

```
1  #define N 5
2  semaphore mutex=1;
3
4  void philosopher(int i)
5  {
6      while (TRUE) {
7          think();
8          wait(&mutex);
9          take_fork(i);
10         take_fork((i+1) % N);
11         eat();
12         put_fork(i);
13         put_fork((i+1) % N);
14         signal(&mutex);
15     }
16 }
```



- Only one philosopher can eat at a time.
- How about 2 mutexes? 5 mutexes?

# The Dining Philosophers Problem

## AST Solution (Part 1)

A philosopher may only move into eating state if neither neighbor is eating

```
1  #define N 5                /* number of philosophers */
2  #define LEFT (i+N-1)%N    /* number of i's left neighbor */
3  #define RIGHT (i+1)%N     /* number of i's right neighbor */
4  #define THINKING 0        /* philosopher is thinking */
5  #define HUNGRY 1          /* philosopher is trying to get forks */
6  #define EATING 2          /* philosopher is eating */
7  typedef int semaphore;
8  int state[N];              /* state of everyone */
9  semaphore mutex = 1;       /* for critical regions */
10 semaphore s[N];            /* one semaphore per philosopher */
11
12 void philosopher(int i)    /* i: philosopher number, from 0 to N-1 */
13 {
14     while (TRUE) {
15         think( );
16         take_forks(i); /* acquire two forks or block */
17         eat( );
18         put_forks(i); /* put both forks back on table */
19     }
20 }
```

# The Dining Philosophers Problem

## AST Solution (Part 2)

```
1 void take_forks(int i)           /* i: philosopher number, from 0 to N-1 */
2 {
3     down(&mutex);                /* enter critical region */
4     state[i] = HUNGRY;           /* record fact that philosopher i is hungry */
5     test(i);                    /* try to acquire 2 forks */
6     up(&mutex);                 /* exit critical region */
7     down(&s[i]);                /* block if forks were not acquired */
8 }
9 void put_forks(i)               /* i: philosopher number, from 0 to N-1 */
10 {
11     down(&mutex);              /* enter critical region */
12     state[i] = THINKING;       /* philosopher has finished eating */
13     test(LEFT);                /* see if left neighbor can now eat */
14     test(RIGHT);               /* see if right neighbor can now eat */
15     up(&mutex);                /* exit critical region */
16 }
17 void test(i)                   /* i: philosopher number, from 0 to N-1 */
18 {
19     if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
20         state[i] = EATING;
21         up(&s[i]);
22     }
23 }
```



# The Dining Philosophers Problem

## AST Solution (Part 2)

```
1 void take_forks(int i)           /* i: philosopher number, from 0 to N-1 */
2 {
3     down(&mutex);                /* enter critical region */
4     state[i] = HUNGRY;           /* record fact that philosopher i is hungry */
5     test(i);                     /* try to acquire 2 forks */
6     up(&mutex);                  /* exit critical region */
7     down(&s[i]);                 /* block if forks were not acquired */
8 }
9 void put_forks(i)               /* i: philosopher number, from 0 to N-1 */
10 {
11     down(&mutex);                /* enter critical region */
12     state[i] = THINKING;         /* philosopher has finished eating */
13     test(LEFT);                  /* see if left neighbor can now eat */
14     test(RIGHT);                 /* see if right neighbor can now eat */
15     up(&mutex);                  /* exit critical region */
16 }
17 void test(i)                    /* i: philosopher number, from 0 to N-1 */
18 {
19     if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
20         state[i] = EATING;
21         up(&s[i]);
22     }
23 }
```

# The Dining Philosophers Problem

## More Solutions

- ▶ If there is at least one leftie and at least one rightie, then deadlock is not possible
- ▶ [Wikipedia: Dining philosophers problem](#)

## 15.2 The Readers-Writers Problem

# The Readers-Writers Problem

**Constraint:** no process may access the shared data for reading or writing while another process is writing to it.

```
1 semaphore mutex = 1;
2 semaphore noOther = 1;
3 int readers = 0;
4
5 void writer(void)
6 {
7     while (TRUE) {
8         wait(&noOther);
9         writing();
10        signal(&noOther);
11    }
12 }
```

```
1 void reader(void)
2 {
3     while (TRUE) {
4         wait(&mutex);
5         readers++;
6         if (readers == 1)
7             wait(&noOther);
8         signal(&mutex);
9         reading();
10        wait(&mutex);
11        readers--;
12        if (readers == 0)
13            signal(&noOther);
14        signal(&mutex);
15        anything();
16    }
17 }
```

# The Readers-Writers Problem

**Constraint:** no process may access the shared data for reading or writing while another process is writing to it.

```
1 semaphore mutex = 1;
2 semaphore noOther = 1;
3 int readers = 0;
4
5 void writer(void)
6 {
7     while (TRUE) {
8         wait(&noOther);
9         writing();
10        signal(&noOther);
11    }
12 }
```

```
1 void reader(void)
2 {
3     while (TRUE) {
4         wait(&mutex);
5         readers++;
6         if (readers == 1)
7             wait(&noOther);
8         signal(&mutex);
9         reading();
10        wait(&mutex);
11        readers--;
12        if (readers == 0)
13            signal(&noOther);
14        signal(&mutex);
15        anything();
16    }
17 }
```

# The Readers-Writers Problem

No starvation

```
1 semaphore mutex = 1;
2 semaphore noOther = 1;
3 semaphore turnstile = 1;
4 int readers = 0;
5
6 void writer(void)
7 {
8     while (TRUE) {
9         turnstile.wait();
10        wait(&noOther);
11        writing();
12        signal(&noOther);
13        turnstile.signal();
14    }
15 }
```

```
1 void reader(void)
2 {
3     while (TRUE) {
4         turnstile.wait();
5         turnstile.signal();
6
7         wait(&mutex);
8         readers++;
9         if (readers == 1)
10            wait(&noOther);
11        signal(&mutex);
12        reading();
13        wait(&mutex);
14        readers--;
15        if (readers == 0)
16            signal(&noOther);
17        signal(&mutex);
18        anything();
19    }
20 }
```

## 15.3 The Sleeping Barber Problem

# The Sleeping Barber Problem



## Where's the problem?

- ▶ the barber saw an empty room right before a customer arrives the waiting room;
- ▶ Several customer could race for a single chair;



# Solution

```
1  #define CHAIRS 5
2  semaphore customers = 0; // any customers or not?
3  semaphore bber = 0;      // barber is busy
4  semaphore mutex = 1;
5  int waiting = 0;        // queueing customers
```

```
1  void barber(void)
2  {
3      while (TRUE) {
4          wait(&customers);
5          wait(&mutex);
6          waiting--;
7          signal(&mutex);
8          cutHair();
9          signal(&bber);
10 }
11 }
```

```
1  void customer(void)
2  {
3      if(waiting == CHAIRS)
4          goHome();
5      else {
6          wait(&mutex);
7          waiting++;
8          signal(&mutex);
9          signal(&customers);
10         wait(&bber);
11         getHairCut();
12     }
13 }
```

## Solution2

```
1  #define CHAIRS 5
2  semaphore customers = 0;
3  semaphore bber = ?;
4  semaphore mutex = 1;
5  int waiting = 0;
6
7  void barber(void)
8  {
9      while (TRUE) {
10         wait(&customers);
11         cutHair();
12     }
13 }
```

```
1  void customer(void)
2  {
3      if (waiting == CHAIRS)
4         goHome();
5      else {
6         wait(&mutex);
7         waiting++;
8         signal(&mutex);
9         signal(&customers);
10        wait(&bber);
11        getHairCut();
12        signal(&bber);
13        wait(&mutex);
14        waiting--;
15        signal(&mutex);
16    }
17 }
```

## 16 Shared Memory

# Write

```
int main(int argc, char *argv[])
{
    int fd;
    size_t len;                /* Size of shared memory object */
    char *addr;

    if (argc != 3 || strcmp(argv[1], "--help") == 0){}

    if ( (fd = shm_open(argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR)) == -1 ){}

    len = strlen(argv[2]);
    if (ftruncate(fd, len) == -1){ /* Resize object to hold string */}
    printf("Resized to %ld bytes\n", (long)len);

    addr = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED) perror("mmap");

    if (close(fd) == -1) perror("close");

    printf("copying %ld bytes\n", (long) len);
    memcpy(addr, argv[2], len);    /* Copy string to shared memory */
    exit(EXIT_SUCCESS);
}
```

# Read

```
int main(int argc, char *argv[])
{
    int fd;
    char *addr;
    struct stat sb;

    if (argc != 2 || strcmp(argv[1], "--help") == 0){}

    if ( (fd = shm_open(argv[1], O_RDONLY, 0)) == -1 ){ }

    if (fstat(fd, &sb) == -1) perror("fstat"); /* Get object size */

    addr = mmap(NULL, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED) perror("mmap");

    if (close(fd) == -1) perror("close");

    write(STDOUT_FILENO, addr, sb.st_size);
    printf("\n");
    exit(EXIT_SUCCESS);
}
```

## 17 Sockets

# Message Passing

## Problem with semaphores

- ▶ Too low level
- ▶ Not suitable for distributed systems

## Message passing

- ▶ No conflicts, easier to implement
- ▶ Uses two primitives, `send()` and `receive()` system calls:
  - `send(destination, &message);`
  - `receive(source, &message);`

# Message Passing

## Design issues

- ▶ Message can be lost by network; — ACK
- ▶ What if the ACK is lost? — SEQ
- ▶ What if two processes have the same name? — socket
- ▶ Am I talking with the right guy? Or maybe a MIM? — authentication
- ▶ What if the sender and the receiver on the same machine? — Copying messages is always slower than doing a semaphore operation.



# Message Passing

## TCP Header Format

0				1				2				3			
Source Port								Destination Port							
Sequence Number															
Acknowledgment Number															
Data Offset		0	0	0	N S	C R E	U R G	A P S T	P R S Y N	F I N	Window				
Checksum										Urgent Pointer					
Options												Padding			

# Message Passing

## The producer-consumer problem

```
1  #define N 100  /* number of slots in the buffer */
2  void producer(void)
3  {
4      int item;
5      message m;          /* message buffer */
6      while (TRUE) {
7          item = produce_item(); /* generate something to put in buffer */
8          receive(consumer, &m); /* wait for an empty to arrive */
9          build_message(&m, item); /* construct a message to send */
10         send(consumer, &m);      /* send item to consumer */
11     }
12 }
13
14 void consumer(void)
15 {
16     int item, i;
17     message m;
18     for (i=0; i<N; i++) send(producer, &m); /* send N empties */
19     while (TRUE) {
20         receive(producer, &m); /* get message containing item */
21         item = extract_item(&m); /* extract item from message */
22         send(producer, &m); /* send back empty reply */
23         consume_item(item); /* do something with the item */
24     }
25 }
```

# A TCP Connection

```
wx672@cs3:~$ netstat -at | grep http | grep ESTAB
```

```
tcp    0    0  cs3.swfu.edu.cn:http    220.163.96.3:47179    ESTABLISHED
```

address		port	address		port
socket			socket		

a pair of sockets form a TCP connection

## Port numbers

Port range: 0 ~ 65535

Well-known ports: 0 ~ 1023

FTP	20/21	SSH	22	Telnet	23
SMTP	25	DNS	53	DHCP	67/68
HTTP	80	POP3	110	HTTPS	443
IMAP4	143				

# Sockets

To create a socket:

```
fd = socket(domain, type, protocol)
```

**Domain** Determines address format and the range of communication (local or remote). The most commonly used domains are:

Domain	Addr structure	Addr format
AF_UNIX	sockaddr_un	/path/name
AF_INET	sockaddr_in	ip:port
AF_INET6	sockaddr_in6	ip6:port

**Type** SOCK\_STREAM (☎), SOCK\_DGRAM (✉)

**Protocol** always 0

# Address Structure

- ▶ Different socket domain, different address format, different structure type
- ▶ One set of socket syscalls supports all socket domains

```
struct sockaddr {  
    sa_family_t sa_family;    /* Address family (AF_* constant) */  
    char        sa_data[14]; /* Socket address (size varies  
                               according to socket domain) */  
};
```

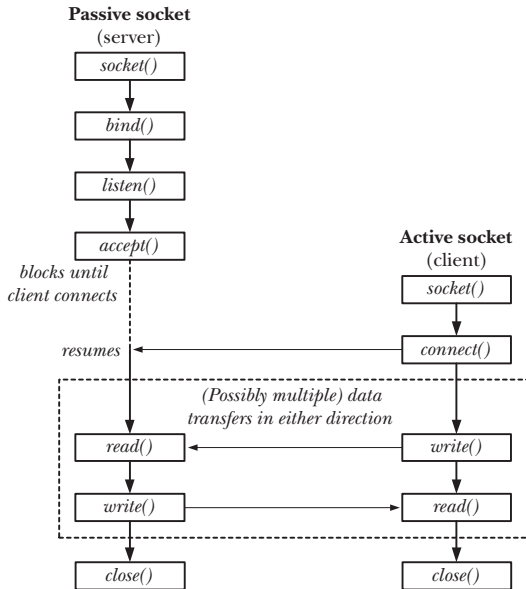
## Example

```
struct sockaddr_un addr;
```

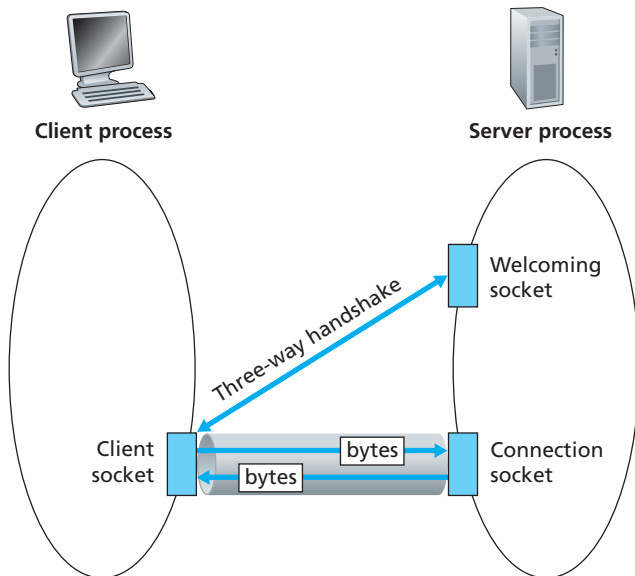
- ✓ `bind(sfd, (struct sockaddr *)&addr, sizeof(struct sockaddr_un));`
- ✗ `bind(sfd, &addr, sizeof(struct sockaddr_un));`

## 17.1 Stream Sockets

# Stream Sockets



# Two Sockets at the Server





# Socket System Calls

`socket()` creates a new socket

`bind()` binds a socket to an address (usually a well-known address on server side)

`listen()` waits for incoming connection requests

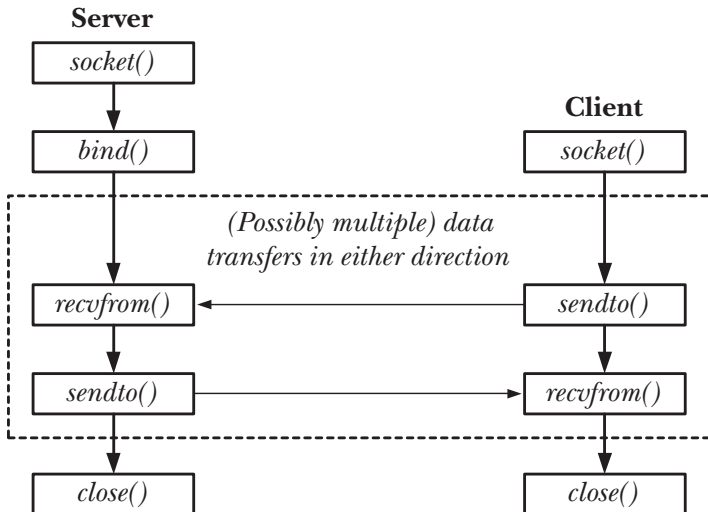
`connect()` sends a connection request to peer

`accept()` accepts a connection request

`send()/recv()` data transfer

## 17.2 Datagram Sockets

# Datagram Sockets



## 17.3 Unix Domain Sockets

# Unix Domain Sockets

```
struct sockaddr_un {  
    sa_family_t sun_family; /* Always AF_UNIX */  
    char sun_path[108]; /* Null-terminated socket pathname */  
};
```

## Stream server

```
#define SV_SOCKET_PATH  "/tmp/us_xfr"
#define BUF_SIZE 100
#define BACKLOG 5

int main(void)
{
    struct sockaddr_un addr;
    int sfd, cfd;
    ssize_t numRead;
    char buf[BUF_SIZE];

    if( (sfd = socket(AF_UNIX, SOCK_STREAM, 0)) == -1 ){}

    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, SV_SOCKET_PATH, sizeof(addr.sun_path) - 1);

    if( bind(sfd, (struct sockaddr *)&addr, sizeof(struct sockaddr_un)) == -1 ){}

    if(listen(sfd, BACKLOG) == -1){}

    for(;;) {
        if( (cfd = accept(sfd, NULL, NULL)) == -1 ){}

        while((numRead = read(cfd, buf, BUF_SIZE)) > 0)
            if(write(STDOUT_FILENO, buf, numRead) != numRead){}

        if(numRead == -1){}

        if(close(cfd) == -1){}
    }
}
```

## Stream client

```
#define SV_SOCKET_PATH "/tmp/us_xfr"
#define BUF_SIZE 100

int main(void)
{
    struct sockaddr_un addr;
    int sfd;
    ssize_t numRead;
    char buf[BUF_SIZE];

    if( (sfd = socket(AF_UNIX, SOCK_STREAM, 0)) == -1 ){}

    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, SV_SOCKET_PATH, sizeof(addr.sun_path) - 1);

    if (connect(sfd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un)) == -1){}

    while ((numRead = read(STDIN_FILENO, buf, BUF_SIZE)) > 0)
        if (write(sfd, buf, numRead) != numRead){}

    if (numRead == -1){}

    exit(EXIT_SUCCESS); /* Closes our socket; server sees EOF */
}
```

Datagram server



Datagram client

## 17.4 Internet Sockets

# Network Byte Order

**Big endian** The most significant byte comes first

**Little endian** The least significant byte comes first

Big endian

0x100 0x101 0x102 0x103

...	01	23	45	67	...
-----	----	----	----	----	-----

`int i = 0x01234567;`

Little endian

0x100 0x101 0x102 0x103

...	67	45	23	01	...
-----	----	----	----	----	-----

**Network byte order** is big endian

**Host byte order** Most architectures are big endian. x86 is an exception.

## Convert int between host and network byte order

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

# Socket Addresses

## IPv4

```
1  struct in_addr {                /* IPv4 4-byte address */
2      in_addr_t s_addr;           /* Unsigned 32-bit integer */
3  };
4
5  struct sockaddr_in {            /* IPv4 socket address */
6      sa_family_t  sin_family;    /* Address family (AF_INET) */
7      in_port_t    sin_port;      /* Port number */
8      struct in_addr sin_addr;     /* IPv4 address */
9      unsigned char __pad[X];     /* Pad to size of sockaddr (16 bytes) */
10 };
```

## IPv6

```
1  struct in6_addr {              /* IPv6 address structure */
2      uint8_t s6_addr[16];       /* 16 bytes == 128 bits */
3  };
4
5  struct sockaddr_in6 {          /* IPv6 socket address */
6      sa_family_t sin6_family;    /* Address family (AF_INET6) */
7      in_port_t    sin6_port;     /* Port number */
8      uint32_t     sin6_flowinfo; /* IPv6 flow information */
9      struct in6_addr sin6_addr;  /* IPv6 address */
10     uint32_t     sin6_scope_id; /* Scope ID (new in kernel 2.4) */
11 };
```