GNU/Linux
Application Programming

WANG Xiaolin

wx672ster+linux@gmail.com

September 18, 2019

Reference Books

- VENKATESH B, ANGRAVE L, et Al. *CS241 System Programming Coursebook*. University of Illinois, 2019.
- MATTHEW N, STONES R. *Beginning linux programming*. John Wiley & Sons, 2008.
- COOPER M. *Advanced Bash Scripting Guide 5.3 Volume 1*. Lulu.com, 2010.
- RAYMOND E S. *The art of Unix programming*. Addison-Wesley, 2003.
- STEVENS W R, RAGO S A. *Advanced programming in the UNIX environment*. Addison-Wesley, 2013.
- LOVE R. *Linux System Programming: Talking Directly to the Kernel and C Library*. O'Reilly Media, Inc., 2007.
- KERRISK M. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, 2010.
- BRYANT R E, O'HALLARON D R. *Computer Systems: A Programmer's Perspective*. 2nd ed. USA: Addison-Wesley, 2010.

Course Web Links

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

/etc/hosts

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

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/

Homework

Weekly tech question

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

✉ `wx672ster+linux@gmail.com`

𝔼 Preferably in English

in stackoverflow style

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

OVERSIMPLIFED 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?

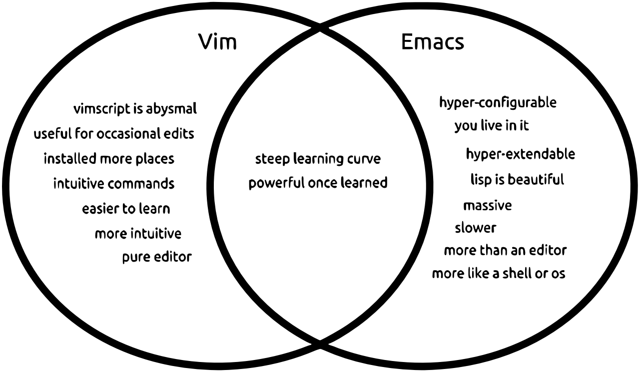
First double check your spelling

Then try:

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

Text Editors

vs.



uemacs Linus Torvalds' editor

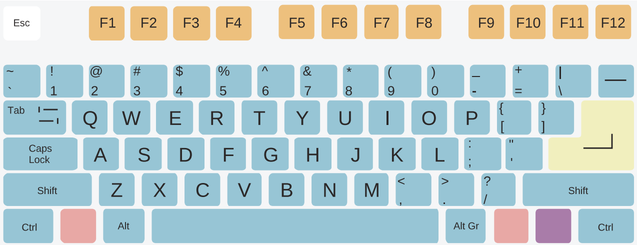
https://github.com/torvalds/uemacs

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

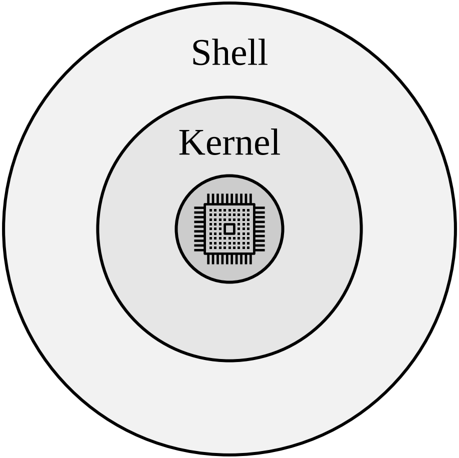
Part II

Shell Basics

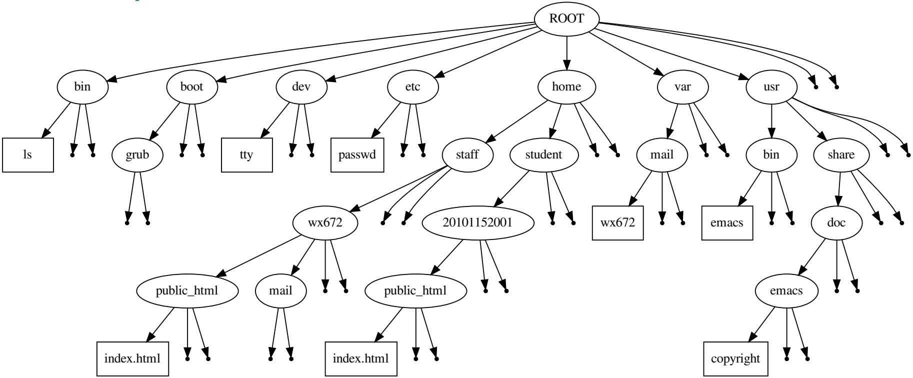
1 Shell Basics

Shell

- A command line interpreter
- A programming language



Directory Structure



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

File Operations

Ways to create a file

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

More file operations:

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

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`, `&`	forground?	`fg`, `jobs`
status?	`ps`, `top`		

System Info

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

APT — package management

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

CLI Shortcuts

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

Tmux

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

Understanding "`ls -l`"

```
-rw------- 1 sam sam   57 Apr 17  1998 weather.txt
drwxr-xr-x 6 sam sam  102 Oct  9  1999 web_page
-rw-rw-r-- 1 sam sam 7648 Feb 11 20:41 web_site.tar
-rw------- 1 sam sam  574 Dec 16  1998 file.txt
|_________ | ___ ___ ____ ____________ ________
|    |     |  |   |   |        |           |
|    |     |  |   |   |        |           └─ File Name
|    |     |  |   |   |        └─ Modification Time
|    |     |  |   |   └─ Size (in bytes)
|    |     |  |   └─ Group
|    |     |  └─ Owner
|    |     └─ Number of hard links
|    └─ File Permissions
└─ File types
```

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

9-bit permission

```
 7   5   5
111 101 101
rwx r-x r-x
___ ___ ___
 |   |   └─ Other
 |   └─── Group
 └─────── User
```

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

Wildcard Expansion

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

Example

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

output:

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

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

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

Everything Is A File

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

Generating random numbers

```
$ echo $RANDOM # 0 ~ 32767; pseudorandom!
$ r=$(($RANDOM % 100)) # 0 ~ 100
$ RANDOM=8; for i in $(seq 10); do echo $RANDOM; done
$ od -A n -N1 -t d /dev/urandom
```

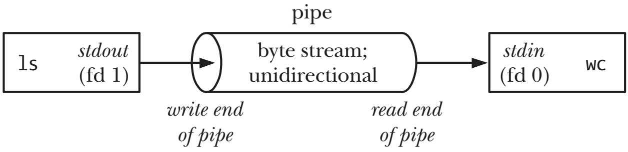
/proc

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

Positional Parameters

$0, $1, $2, …, $@, $#

```bash
#!/bin/bash

echo "You said:"

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

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

```c
#include <stdio.h>

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

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

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

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

  return 0;
}
```

Parameter Substitution

Default value

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

```
$ echo ${v:-8}
$ echo ${v:-10}
```

Example

```bash
#!/bin/bash

echo Hello, ${1:-world}!
```

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 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 ls,cd,…
$ if a = 5; then echo a=$a; fi ✘ # whitespace matters
$ if a=5; then echo a=$a; fi 😐
$ test $a = 5 && echo a=$a ✔
$ [[ $a = 5 ]] && echo a=$a ✔
```

Tests II

```
$ if cmp a b; then echo same file; fi ✔
$ [[ cmp a b ]] && echo same file ✘
$ if test 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

```bash
#!/bin/bash

words=$@
string=linux
if echo "$words" | grep -q "$string"
then
  echo "<$string> found in <$words>"
else
  echo "<$string> not found in <$words>"
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)) ☺
    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 -lt 10 ]]; do echo $a; ((a++)); done ✔
$ while  [ $a -lt 10 ] ; do echo $a; ((a++)); done ✔
$ while [[  a < 10 ]]; do echo $a; ((a++)); done ✘
$ while [[ $a < 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

```bash
#!/bin/bash

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

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

select

```bash
#!/bin/bash

PS3='Your favorite OS? '

select OS in "Linux" "Mac OSX" "Windows"
do
  [[ "$OS" = "Linux" ]] && echo wise guy.
  [[ "$OS" = "Mac OSX" ]] && echo rich guy.
  [[ "$OS" = "Windows" ]] && echo patient guy.
  break
done
```

Functions

```
#!/bin/bash

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

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

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

Array

```
#!/bin/bash

IMGDIR="$HOME/Pics/2009Summer/wallpapers/2009summer-1280x768"

files=($IMGDIR/*.jpg)

# get the length of array ${files[@]}
n=${#files[@]}

# get a random array element
wallpaper="${files[RANDOM % n]}"

# set it as wallpaper
qiv -z $wallpaper
```

Change wallpaper every 5 mins?

Subshells

```
$ read first second
  hello world
$ echo $first $second

$ read first second <<<'hello world'
$ echo $first $second
```

Subshell examples

```
$ echo hello world | (read f s; echo $f $s)
$ echo $f $s

$ x=out; (x=in; echo $x)
```

Part III

Linux Programming Environment

3 C Programming Environment

Program Languages

Machine code

The binary numbers that the CPUs can understand.

`1001110000111011110011111 ... and so on ...`

People don't think in numbers.

Assembly language — friendly to humans

```
MOV A,47 ;1010 1111
ADD A,B  ;0011 0111
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:

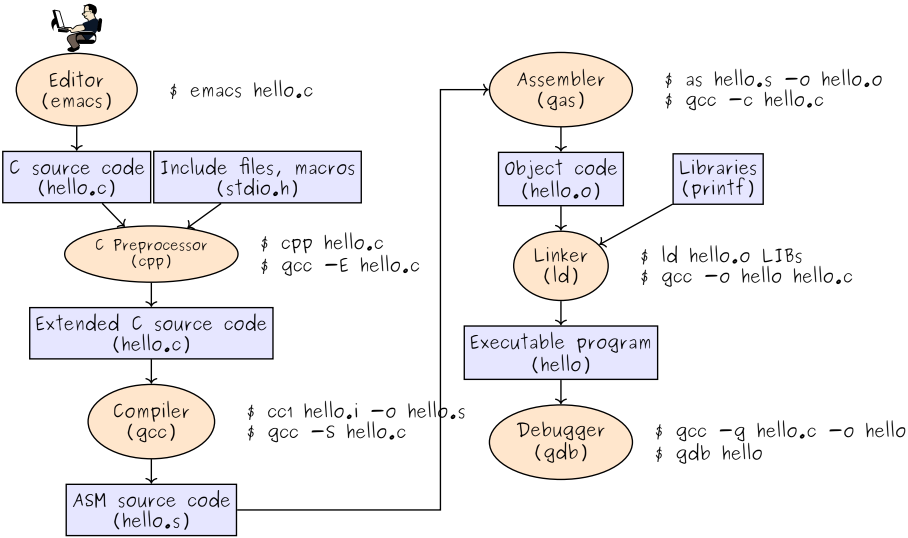
- C
- FORTRAN
- Java
- C++
- ...

Assembly $\rightarrow$ Binary

Compilers do the translation work

3.1 The Tool Chain

Compilation



Compiler vs. Interpreter

hello.c

```c
#include <stdio.h>

int main(void)
{
  puts("Hello, world!");
  return 0;
}
```

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

hello.sh

```bash
#!/bin/bash
echo Hello, world!
```

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

hello.py

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

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

3.2 Header Files

Header Files

Why?

```c
#include "add.h"

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

Why not?

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

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

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

In the header files…

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

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

3.3 Library Files

Library Files

A library is a collection of pre-compiled object files which can be linked into programs.

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.*"
```

Example

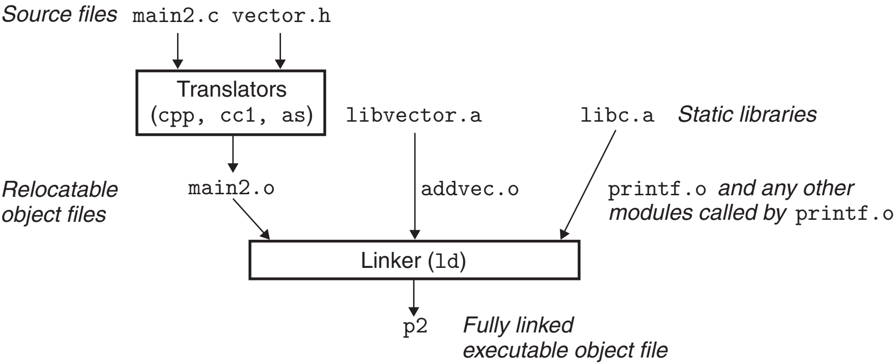
calc.c

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

int main (void)
{
  double x = sqrt (2.0);
  printf ("The square root of 2.0 is %f\n", x);
  return 0;
}
```

```
$ gcc -o calc calc.c $(find /usr/lib -name libm.a) ✔
$ gcc -o calc calc.c -lm ✔
$ gcc -o calc -lm calc.c ☹
```

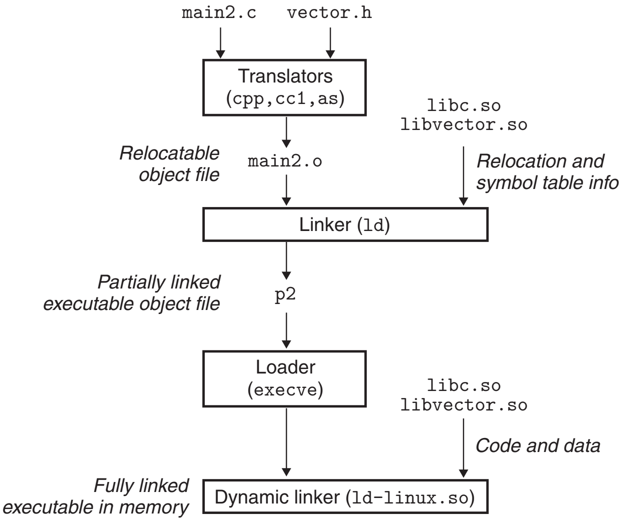
Static Linking

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



Dynamic Linking

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



Build A Static Library

Source codes

main.c

```c
#include "lib.h"

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

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

lib.h

```c
#include <stdio.h>

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

hello.c

```c
#include <stdio.h>

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

hi.c

```c
#include <stdio.h>

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

Build A Static Library

Step by step

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

Build A Static Library

Makefile

```
main: main.c lib.h libhi.a
	gcc -Wall -o main main.c libhi.a

libhi.a: hello.o hi.o
	ar crv libhi.a hello.o hi.o

hello.o: hello.c
	gcc -Wall -c hello.c

hi.o: hi.c
	gcc -Wall -c hi.c

clean:
	rm -f *.o *.a main
```

Build A Shared Library

Source codes

hello.c

```c
#include "hello.h"

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

hello.h

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

int hi(char*);
```

hi.c

```c
#include "hello.h"

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

Build A Shared Library

Step by step

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

Build A Shared Library

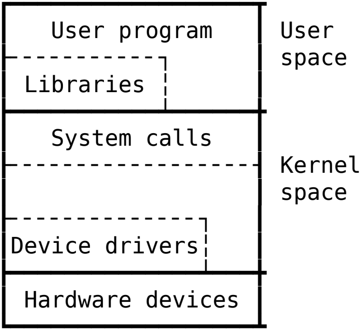
Makefile

```
# http://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html
# http://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html
#
# gcc -fPIC -c hi.c
# gcc -shared -o libhi.so hi.o
# gcc -L/current/dir -Wl,option -Wall -o hello hello.c -lhi
#
# -L         - tells ld where to search libraries
# -Wl,option - pass option as an option to the linker (ld)
# -rpath=dir - Add a directory to the runtime library search path

hello: hello.c hello.h libhi.so
	gcc -L. -Wl,-rpath=. -Wall -o hello hello.c -lhi
libhi.so: hi.o hello.h
	gcc -shared -o libhi.so hi.o
hi.o: hi.c hello.h
	gcc -fPIC -c hi.c
clean:
	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

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

int main(int argc, char *argv[])
{
  if ( open(argv[1], O_RDONLY) == -1 ){
    perror("open");
    exit(EXIT_FAILURE);
  }
  return 0;
}
```

```
$ man errno
$ man errno.h
$ man perror
```

3.5 The Make Utility

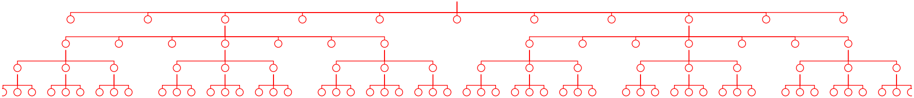
The Make Utility

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

```
target: dependencies
	command
```

Example

```
hello: hello.c
	gcc -o hello hello.c
```

```
$ info make makefiles
```

Makefile

```
edit: main.o kbd.o command.o display.o \
                insert.o search.o files.o utils.o
        gcc -Wall -o edit main.o kbd.o command.o display.o \
                insert.o search.o files.o utils.o

main.o: main.c defs.h
        gcc -c -Wall main.c
kbd.o : kbd.c defs.h command.h
        gcc -c -Wall kbd.c
command.o: command.c defs.h command.h
        gcc -c -Wall command.c
display.o : display.c defs.h buffer.h
        gcc -c -Wall display.c
insert.o: insert.c defs.h buffer.h
        gcc -c -Wall insert.c
search.o: search.c defs.h buffer.h
        gcc -c -Wall search.c
files.o: files.c defs.h buffer.h command.h
        gcc -c -Wall files.c
utils.o: utils.c defs.h
        gcc -c -Wall utils.c

clean:
        rm edit main.o kbd.o command.o display.o \
           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

```
NAME
    A one-line description of the command.
SYNOPSIS
    A formal description of how to run it and what
    command line options it takes.
DESCRIPTION
    A description of the functioning of the command.
EXAMPLES
    Some examples of common usage.
SEE ALSO
    A list of related commands or functions.
BUGS
    List known bugs.
AUTHOR
   Specify your contact information.
COPYRIGHT
    Specify your copyright information.
```

Man Page

Groff source code

```
.\" Text automatically generated by txt2man
.TH untitled  "06 August 2019" "" ""
.SH NAME
\fBA one-line description of the command.
.SH SYNOPSIS
.nf
.fam C
  \fBA formal description of how to run it and what command line options it takes.
.fam T
.fi
.fam T
.fi
.SH DESCRIPTION
\fBA description of the functioning of the command.
.SH EXAMPLES
Some examples of common usage.
.SH SEE ALSO
\fBA list of related commands or functions.
.SH BUGS
List known bugs.
.SH AUTHOR
Specify your contact information.
.SH COPYRIGHT
Specify your copyright information.
```

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

3.8 A Sample GNU Package

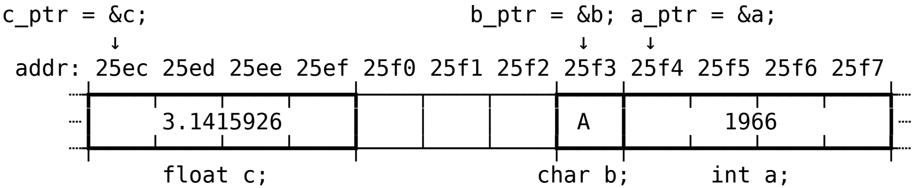
How to “Do one thing, and do it well”?

```
$ apt source hello
```

3.9 Pointers in C

Pointers

```c
int main(void)
{
  int   a = 1966;
  char  b = 'A';
  float c = 3.1415926;
  int   *a_ptr = &a; /* a pointer to an integer */
  char  *b_ptr = &b; /* a pointer to a  char    */
  float *c_ptr = &c; /* a pointer to a  float   */

  printf("&a = %p, sizeof(a) = %ld\n", a_ptr, sizeof(a));
  printf("&b = %p, sizeof(b) = %ld\n", b_ptr, sizeof(b));
  printf("&c = %p, sizeof(c) = %ld\n", c_ptr, sizeof(c));
  return 0;
}
```



Pointer Operators

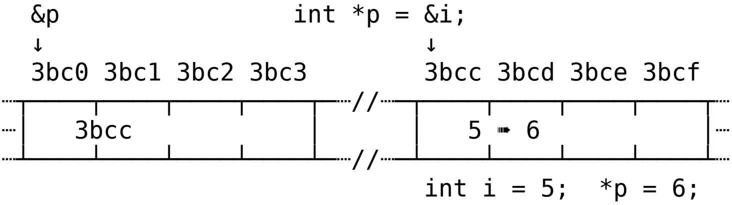
- & returns the address of a thing
- ∗ return the object (thing) to which a pointer points at

```
int thing; int *thing_ptr;
```

C Code	Description
`thing`	the variable named ‘thing’
`&thing`	address of ‘thing’ (a pointer)
`*thing`	✘ `thing` is not a pointer
`thing_ptr`	pointer to an int
`*thing_ptr`	the int variable at the address `thing_ptr` points to
`&thing_ptr`	odd, a pointer to a pointer

Example

```c
int main(void)
{
  int i = 5;
  int *p;
  p = &i; /* now p pointing to i */
  *p = 6; /* i = 6 */

  printf("&i = %p, i = %d, *p = %d\n", &i, i, *p);
  printf("&p = %p, p = %p\n", &p, p);
  return 0;
}
```



Invalid operation

```c
int main(void)
{
  int i = 5;
  printf("*i = %d\n", *i); /* Wrong! */

  return 0;
}
```

Invalid memory access

```c
int main(void)
{
    int *p = 5; /* should be (int *)5 */

    printf(" p = %p\n",  p); /*  p = 0x5 */
    printf("&p = %p\n", &p); /* &p = 0x7ffda48a2068 */
    printf("*p = %c\n", *p); /* Invalid memory access */
    return 0;
}
```

Call By Value

```
void inc_count(int count){
  ++count;
  printf("inc_count: &count = %p\n", &count);
  printf("inc_count: count = %d\n", count);
}

int main(){
  int count = 0;
  printf("main: &count = %p\n", &count);

  inc_count(count);
  printf("main: count = %d\n", count);

  return 0;
}
```

Call By Value

```c
void inc_count(int count){
  ++count;
  printf("inc_count: &count = %p\n", &count);
  printf("inc_count: count = %d\n", count);
}

int main(){
  int count = 0;
  printf("main: &count = %p\n", &count);

  inc_count(count);
  printf("main: count = %d\n", count);

  return 0;
}
```

Call by value: only the value of ‘`count`’ is handed to the function `inc_count()`

Solution 1: return

```c
int inc_count(int count)
{
    return ++count;
}

int main()
{
    int count = 0;

    count = inc_count(count);
    printf("%d\n", count);

    return 0;
}
```

Solution 2: Call by reference

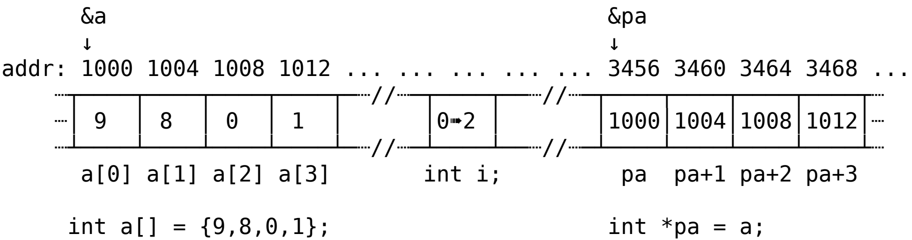
```c
void inc_count(int *count_ptr)
{
    ++(*count_ptr);
}

int main()
{
    int count = 0;

    inc_count(&count);
    printf("%d\n", count);

    return 0;
}
```

More efficient than solution 1 Imagining you are operating on a large data structure rather than an `int`

3.10 Pointers and Arrays

```c
int main(void)
{
  int a[] = {9,8,0,1};
  int i = 0;

  while (a[i] != 0)
    ++i;

  printf("ZERO at a[%d].\n", i);

  return 0;
}
```

```c
int main(void)
{
  int a[] = {9,8,0,1};
  int *pa = a;

  while ((*pa) != 0)
    ++pa;

  printf("ZERO at a[%ld].\n", pa - a);
  printf("pa = %p; a = %p\n",pa,a);
  return 0;
}
```



Passing Arrays to Functions

When passing an array to a function, C will automatically change the array into a pointer.

```
#define MAX 5
void init_array_1(int*);
void init_array_2(int*);

int main(void)
{
  int a[MAX], i;

  init_array_1(a);
  printf("init_array_1: ");
  for(i=0; i<MAX; i++)
    printf("a[%d]=%d, ", i, a[i]);
  puts("");

  init_array_2(a);
  printf("init_array_2: ");
  for(i=0; i<MAX; i++)
    printf("a[%d]=%d, ", i, a[i]);
  puts("");

  return 0;
}
```

```
void init_array_1(int a[])
{
  int i;

  for (i=0; i<MAX; ++i)
      a[i] = 0;
}

void init_array_2(int *ptr)
{
  int i;

  for (i=0; i<MAX; ++i)
      *(ptr + i) = i;
}
```

Strings — Arrays Of Characters

Strings are character arrays with the additional special character "`\0`" (`NUL`) at the end, e.g.:

```
char system[] = "Linux";
```

L	i	n	u	x	\0

The most common string functions

```
strcpy(string1, string2) /* copy string2 into string1 */
strcat(string1, string2) /* concatenate string2 onto
                            the end of string1 */
length = strlen(string)  /* get the length of a string */
strcmp(string1, string2) /* 0 if string1 equals string2,
                            otherwise nonzero */
```

Arrays Of Pointers

```
void print_msg(char *ptr_a[], int n)
{
  int i;
  for (i = 0; i < n; i++)
    printf(" %s", ptr_a[i]);

  puts(".");
}

int main()
{
  char *message[9] =
    {"Dennis", "Ritchie", "designed",
         "the",         "C", "language",
          "in",     "the",     "1970s"};
  print_msg(message, 9);
  return 0;
}
```

4 The Linux Environment

Command Line Options

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

  for(arg = 0; arg < argc; arg++) {
    if(argv[arg][0] == '-')
      printf("option: %s\n", argv[arg]);
    else
      printf("argv[%d]: %s\n", arg, argv[arg]);
  }
  return 0;
}
```

```
$ ./a.out -a -bc hello 'holy world'
```

getopt — The Standard Way

$ man 3 getopt

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

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

```
$ ./a.out -h -l -fhello
```

$ help getopts

```bash
#!/bin/bash

while getopts hf:l OPT; do
  case $OPT in
    h) echo "usage: `basename $0` [-h] [-f file] [-l]"
       exit 1 ;;
    l) echo "option: l" ;;
    f) echo "filename: $OPTARG" ;;
  esac
done
```

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

Environment Variable

```c
extern char** environ;

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

  while (*env) {
    printf("%s\n", *env);
    env++;
  }
  return 0;
}
```

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

Time and Date

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

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

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

  return 0;
}
```

- January 1 1970 — start of the Unix epoch

$ `man 3 time`

$ `man 3 ctime`

Temporary Files

mkstemp.c

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

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

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

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

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

mktemp.sh

```bash
#!/bin/bash

tmp=$(mktemp)

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


```

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

Logging

syslog.c

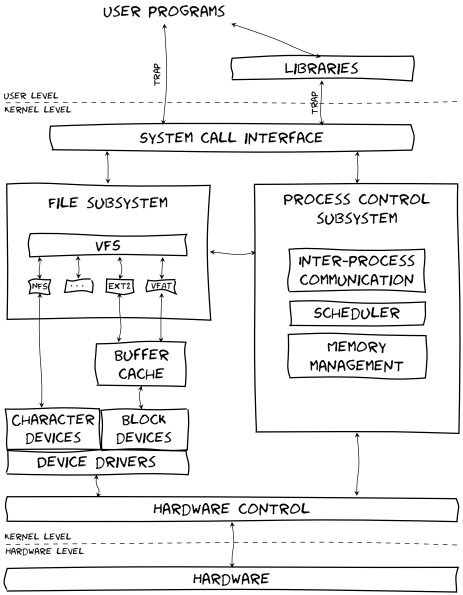
```c
#include <syslog.h>
#include <sys/stat.h>
#include <fcntl.h>

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

logger.sh

```bash
#!/bin/bash

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

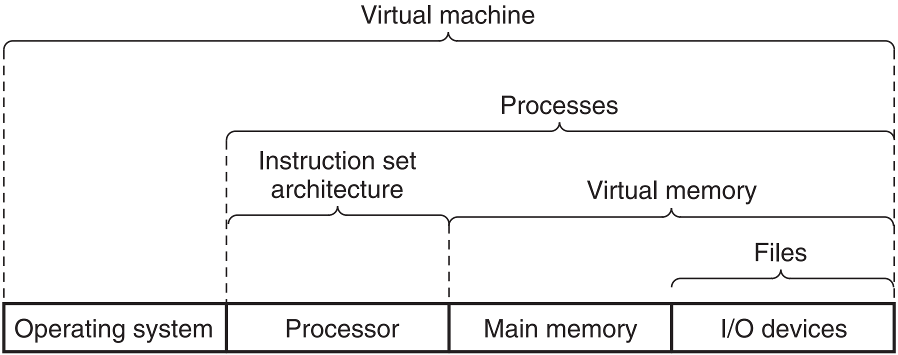
5 OS Basics

Operating System

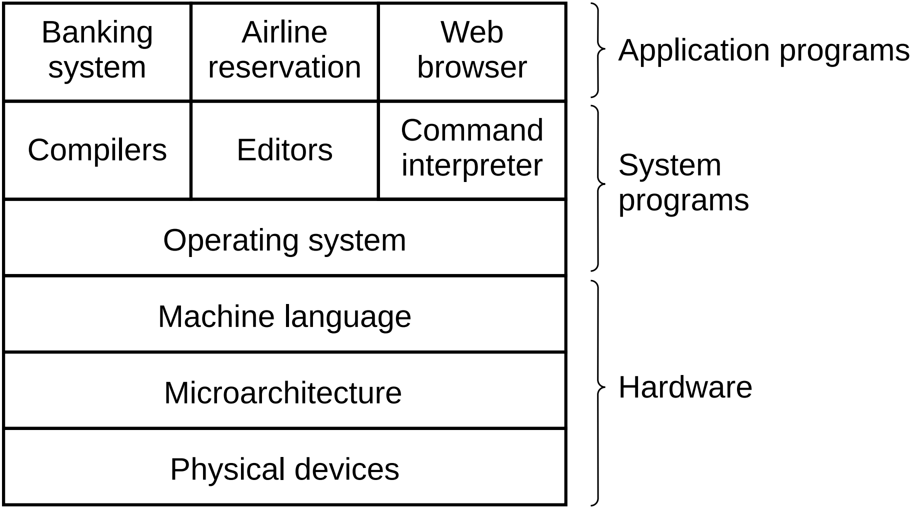


Abstractions

To hide the complexity of the actual implementations

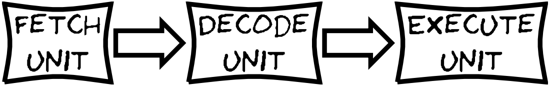


A Computer System



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

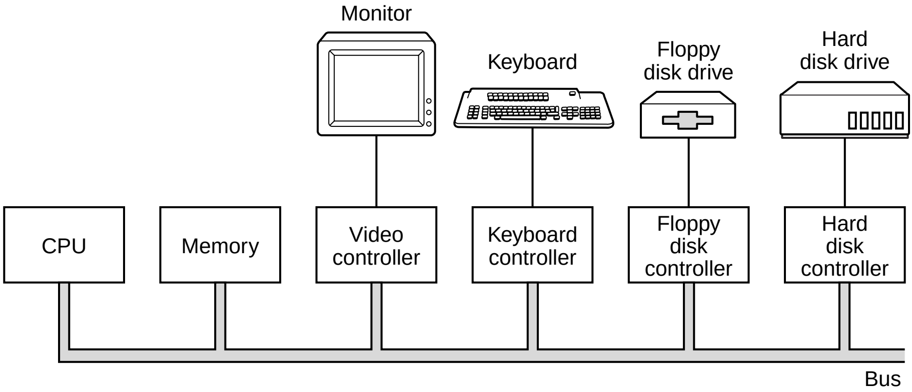
Program counter (PC): keeps the memory address of the next instruction to be fetched

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

Program status (PS): holds

- condition code bits
- processor state

System Bus



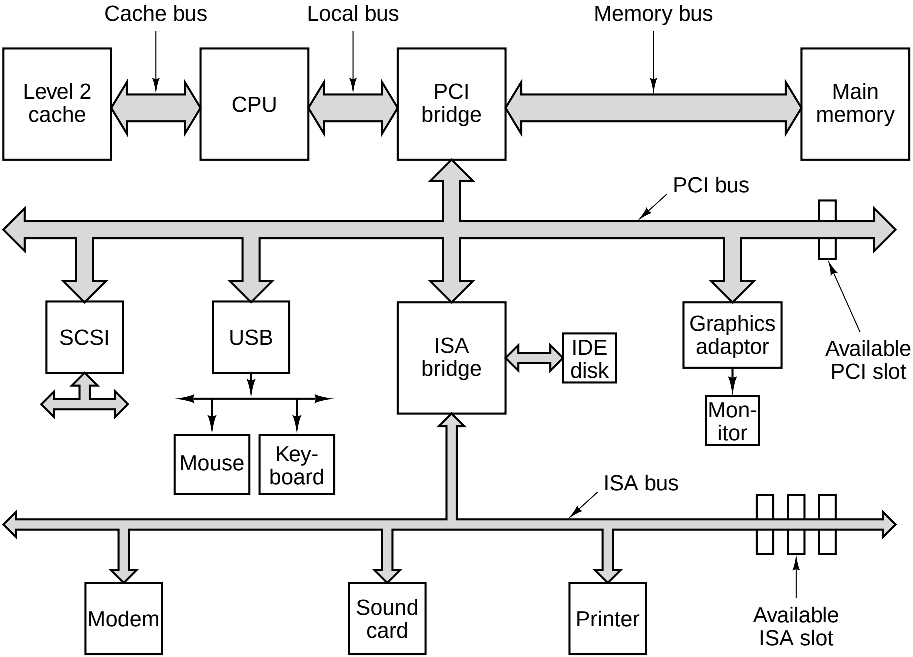
Address Bus: specifies the memory locations (addresses) for the data transfers

Data Bus: holds the data transfered. Bidirectional

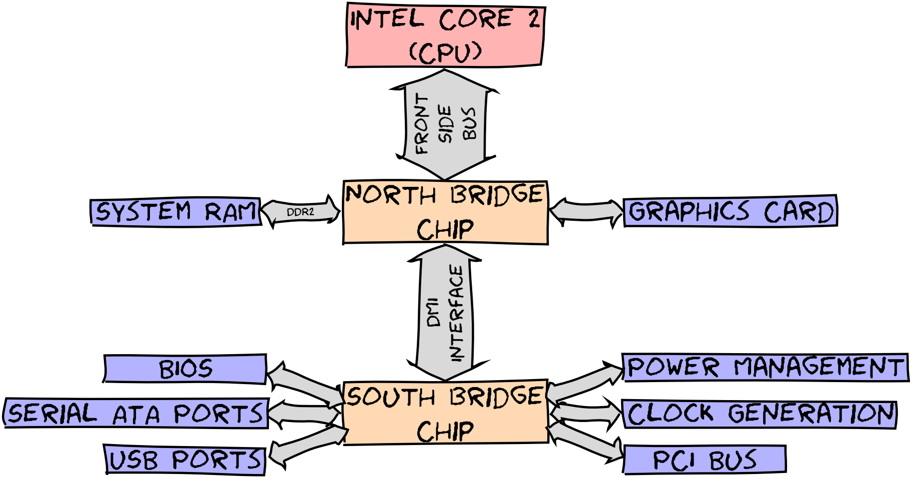
Control Bus: contains various lines used to route timing and control signals throughout the system

Controllers and Peripherals

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


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

Motherboard Chipsets



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

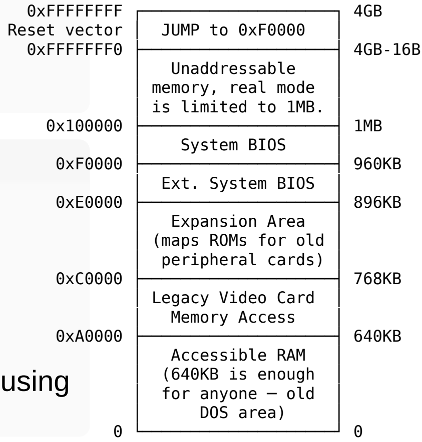
Intel Core 2 QX6600

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

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

Some physical memory addresses are mapped away!

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



Memory-mapped I/O

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

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

the northbridge

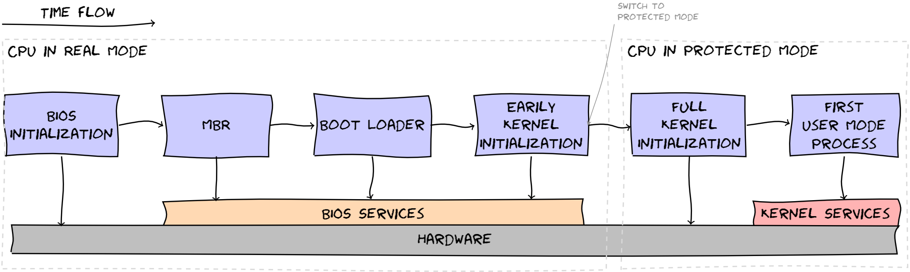
1. receives a physical memory request
2. decides where to route it
   - to RAM? to video card? to …?
   - decision made via the memory address map

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

```
|<------------ Master Boot Record (512 Bytes) ------------>|
|----//-------------------------------------//-------------|
| code area  | disk-sig  | null | partition table | MBR-sig |
|    440     |    4      |  2   |    16x4=64      | 0xAA55  |
|----//-------------------------------------//-------------|
0           439         443    445               509      511
```

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

5.3 Interrupt

Why Interrupt?

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

```
while(!done_reading_a_file())
{
  let_CPU_wait();
  // or...
  lend_CPU_to_others();
}
operate_on_the_file();
```

Modern OS are Interrupt Driven

HW INT by sending a signal to CPU

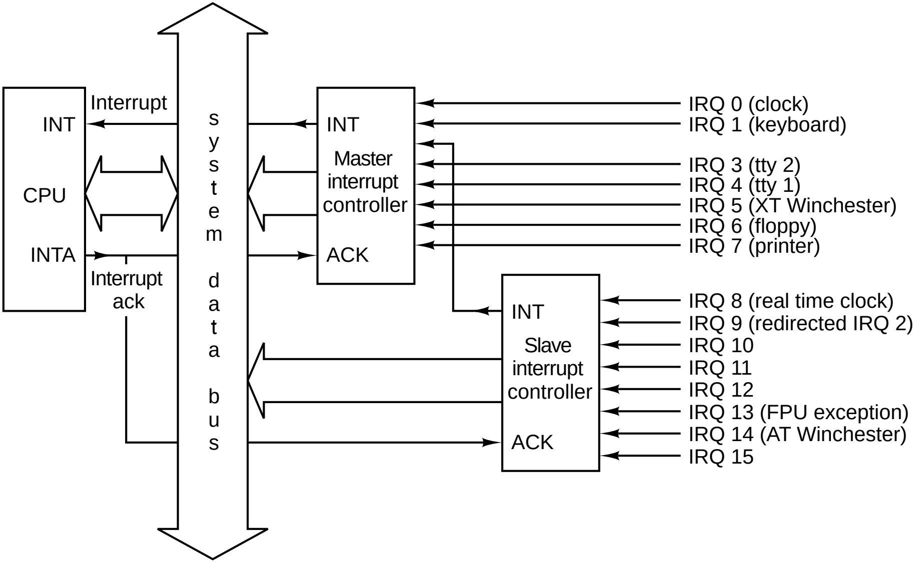
SW INT by executing a system call

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

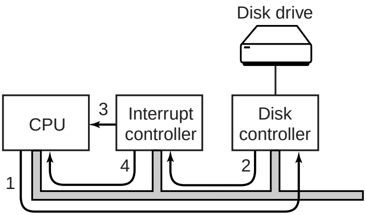
Interrupt vector is an array of pointers ☛ the memory addresses of interrupt handlers. This array is indexed by a unique device number

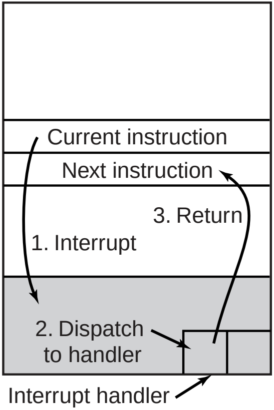
```
$ less /proc/devices
$ less /proc/interrupts
```

Programmable Interrupt Controllers

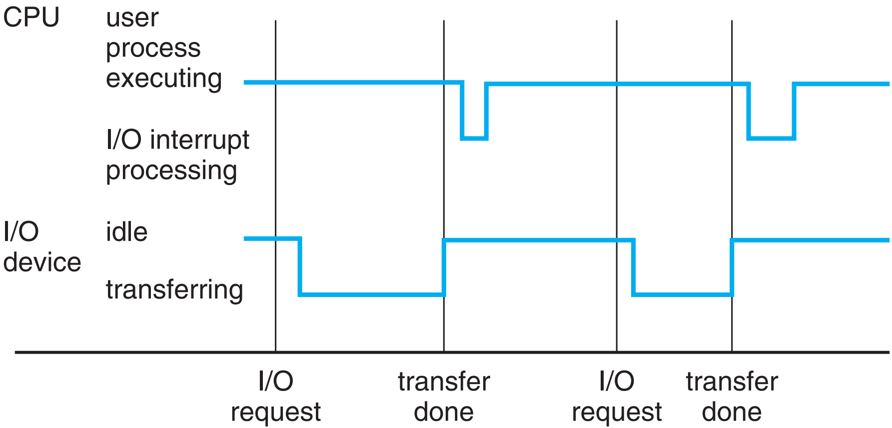


Interrupt Processing





Interrupt Timeline



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
```

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

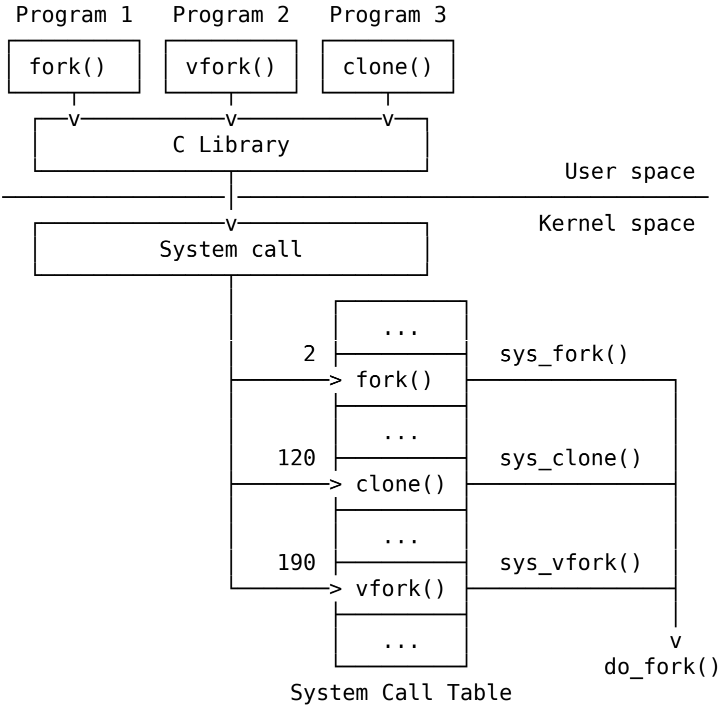
Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer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

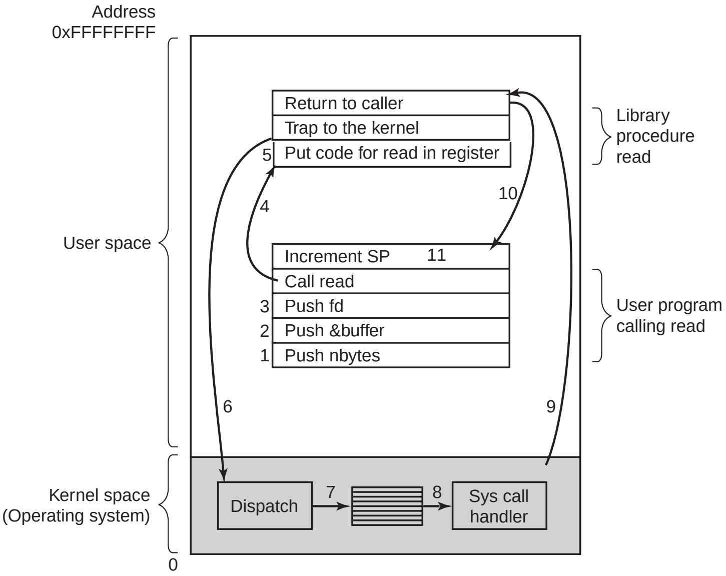
Miscellaneous

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


Program 1
Program 2
Program 3
fork()
vfork()
clone()
C Library
User space
Kernel space
System call
...
2
fork()
sys_fork()
...
120
clone()
sys_clone()
...
190
vfork()
sys_vfork()
...
do_fork()
System Call Table

The 11 steps in making a system call

`read(fd,buffer,nbytes)`



Example

Linux `INT 80h`

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

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

Example

```
SECTION .DATA
Msg:        db 'Hello, world!',10   ; 10 = ascii for LF
MsgLen:     equ $-Msg

SECTION .TEXT
GLOBAL _start

_start:
        mov eax, 4          ; write(
        mov ebx, 1          ;    STDOUT_FILENO,
        mov ecx, Msg        ;    "Hello, world!\n",
        mov edx, MsgLen     ;    sizeof("Hello, world!\n")
        int 80h             ; );

        mov eax, 1          ; exit(
        mov ebx, 0          ;    EXIT_SUCCESS
        int 80h             ; );
```

```
$ nasm -f elf64 hello.asm -o hello.o
$ ld hello.o -o hello
$ less /usr/include/asm/unistd_32.h
$ less /usr/include/asm/unistd_64.h
```

System Call Examples

```c
#include <unistd.h>

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

  return 0;
}
```

- Actually, write() is a wrapper function in glibc.

$ man 2 write

$ man 3 write

Don’t invoke syscall directly whenever possible

```c
int main(void) {
  register char* arg2 asm("rsi") = "hello, world!\n";

  /* rax: sys_write; rdi: STDOUT; */
  asm("mov $1, %rax; mov $1, %rdi; mov $14, %rdx; syscall;");

  return 0;
}
```

System Call Examples

```
$ man 2 fork
```

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

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

execve()

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

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

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

write()

```c
#include <unistd.h>

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

  return 0;
}
```

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

read()

```c
#include <unistd.h>

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

  read(0, buffer, 10);

  write(1, buffer, 10);

  return 0;
}
```

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

- No need to `open()` `STDIN`, `STDOUT`, and `STDERR`

cp

```c
#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 */
}
```

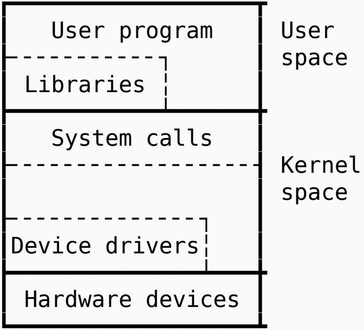
`stdio` — The Standard I/O Library

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()

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

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

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

  return 0;
}
```

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

fopen() — Buffered I/O

```c
#include <stdio.h>

int main(void)
{
  FILE *stream;

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

  while ( fgetc(stream) != EOF );

  fclose(stream);

  return 0;
}
```

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

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

cp — With stdio

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

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

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

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

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

        return 0;
}
```

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

File System Implementation

A typical file system layout

```
|<--------------------------- Entire disk --------------------------->|
|----------------------------------------------------------------------|
| MBR | Partition 1    | Partition 2    | Partition 3    | Partition 4 |
|----------------------------------------------------------------------|
    ___________________________________/                 \______________
   /                                                                    \
 --------------------------------------------------------------------//----
| Boot Ctrl Blk      | Volume Ctrl Blk    |   Dir Structure          | Files |
|  (MBR copy)        |  (Super Blk)       | (inodes, root dir)       | dirs  |
 --------------------------------------------------------------------//----

        |<-------------- Master Boot Record (512 Bytes) -------------->|
        |----//-----------------------------------//-------------------|
        | code area  | disk-sig  | null | partition table | MBR-sig    |
        |    440     |     4     |  2   |    16x4=64      | 0xAA55     |
        |----//-----------------------------------//-------------------|
        0           439         443    445               509          511
```

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).

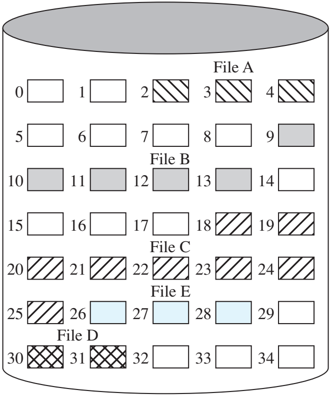
Superblock

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

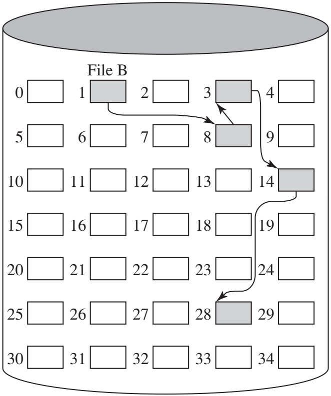


File Allocation Table

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

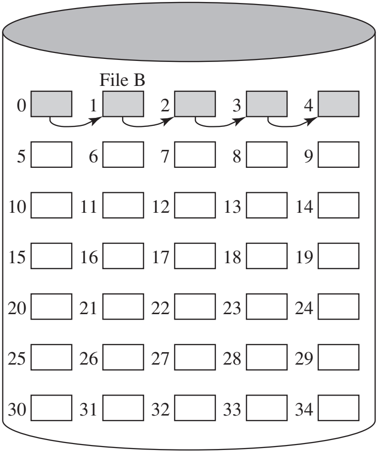
File Name	Start Block	Length
•••	•••	•••
File B	1	5
•••	•••	•••

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
• • •	• • •	• • •

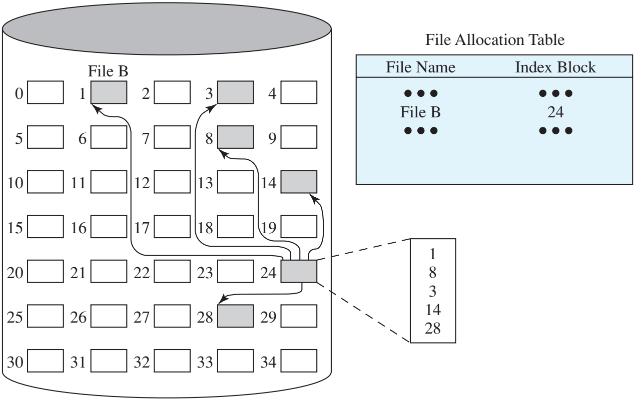
FAT: Linked list allocation with a table in RAM

- 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

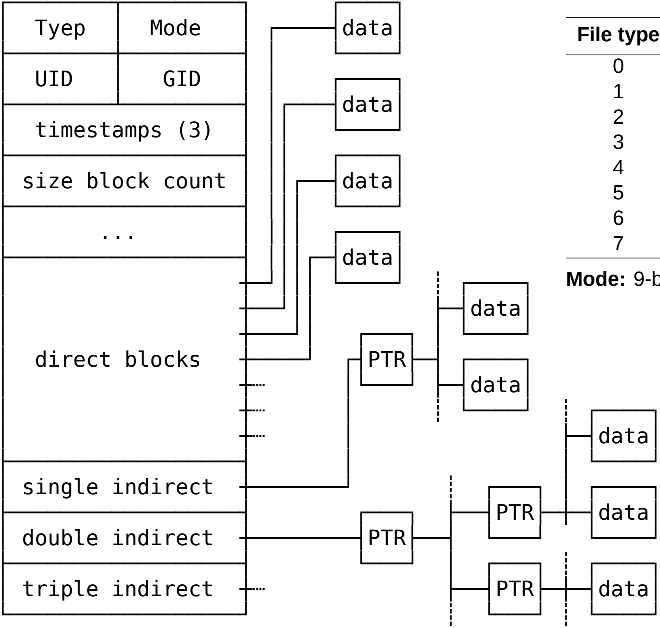
Indexed Allocation



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

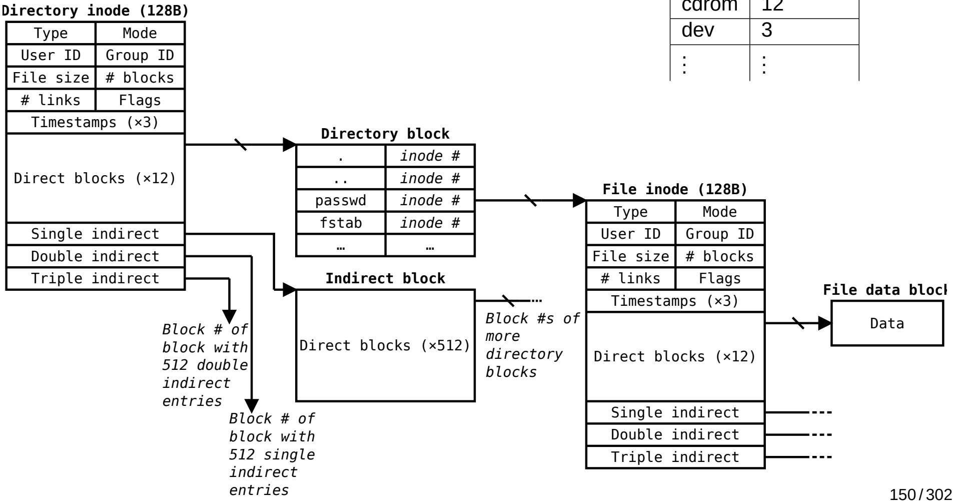


File type	Description
0	Unknown
1	Regular file
2	Directory
3	Character device
4	Block device
5	Named pipe
6	Socket
7	Symbolic link

Mode: 9-bit pattern

UNIX Treats a Directory as a File

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

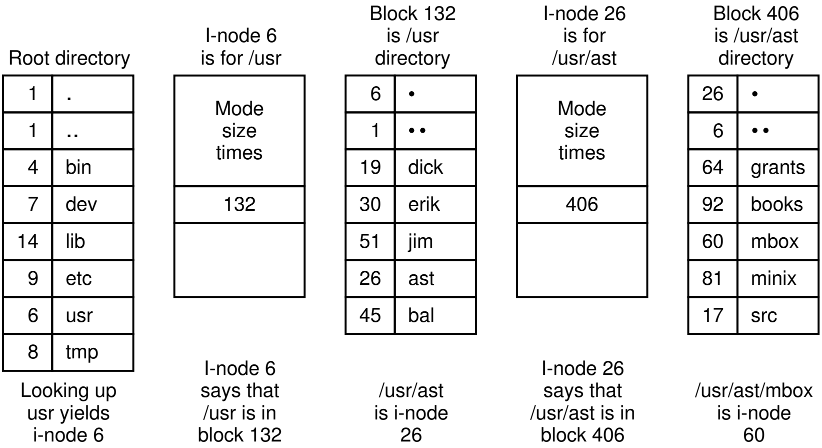


`open()`

Why? To avoid constant searching

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

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



`fd open(pathname, flags)`

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

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

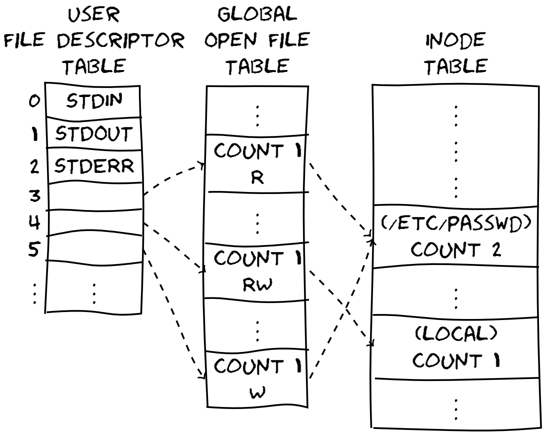
To see files opened by a process, e.g. `init`

```
$ lsof -p 1
```

```
$ man 2 open
```

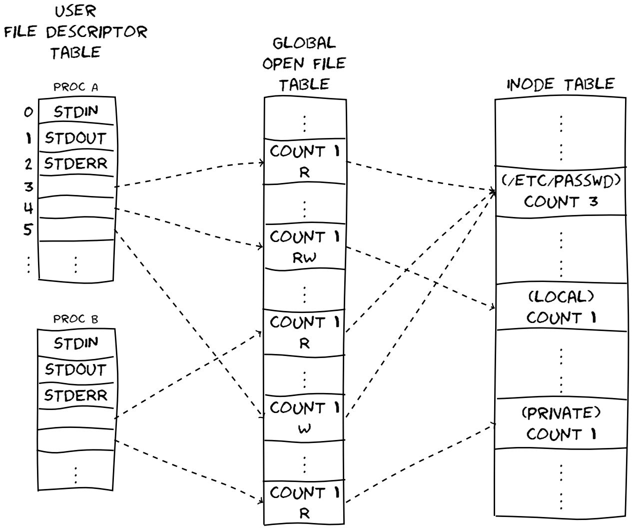
A process executes the following code:

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



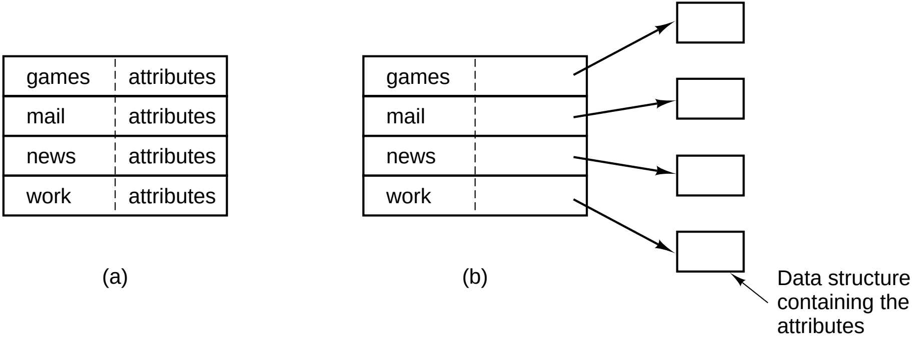
One more process B:

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



7 Directories

Implementing Directories



(a) A simple directory (Windows)

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

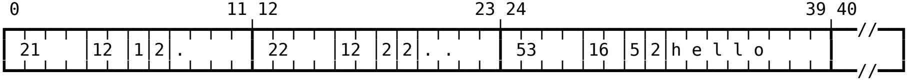
(b) Directory in which each entry just refers to an i-node (UNIX)

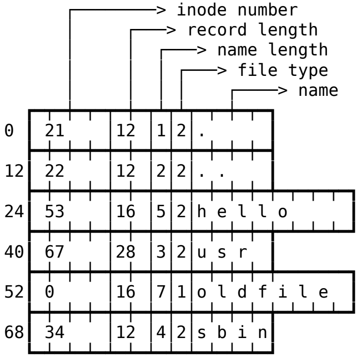
Directory entry in `glibc`

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

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

Ext2 Directories





- Directories are special files
- "." and ". ." first
- Padding to $4\times$
- inode number is 0 — deleted file

ls

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

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

  dp = opendir(argv[1]);

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

  closedir(dp);

  return 0;
}
```

The real `ls.c`?

116 A4 pages
5308 lines

Do one thing, and do it really well.

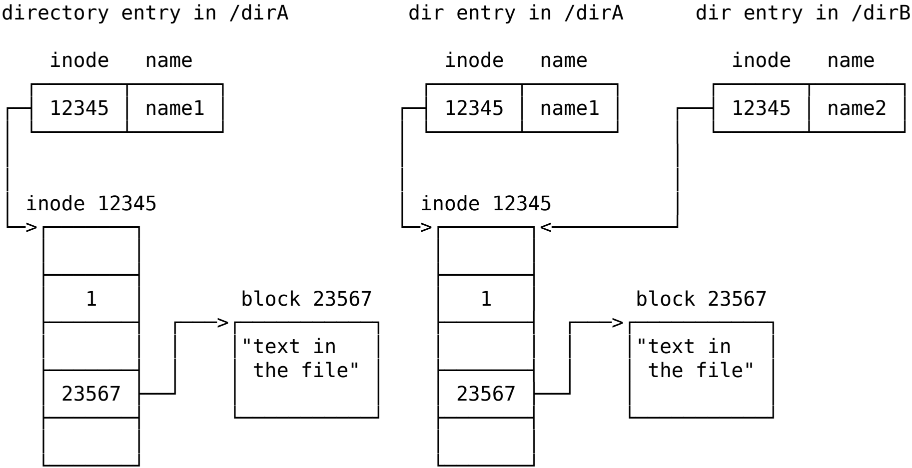
```
$ apt source coreutils
```

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

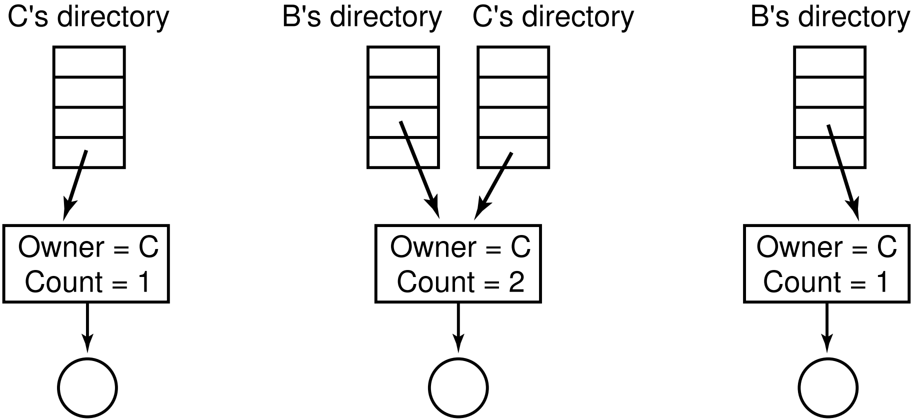
```c
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

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

Hard Links

Hard links ☛ the same inode



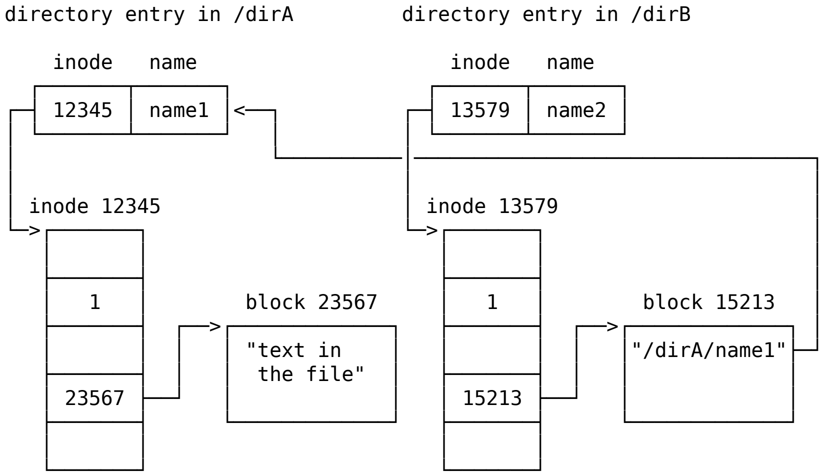
Drawback



(a) (b) (c)

Symbolic Links

A symbolic link has its own inode ☛ a directory entry



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

link(), unlink(), symlink()

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

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

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

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

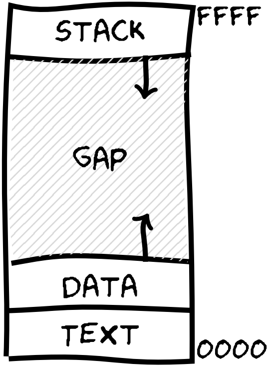
A process is an instance of a program in execution

Processes are like human beings:

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

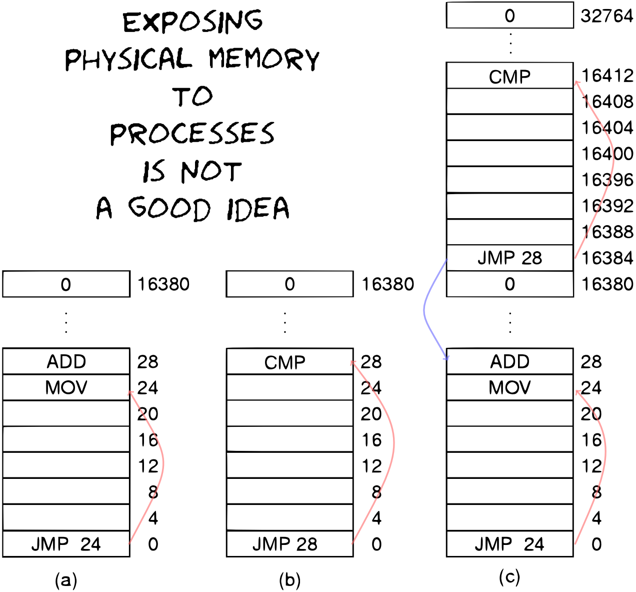
A small difference:

- 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.

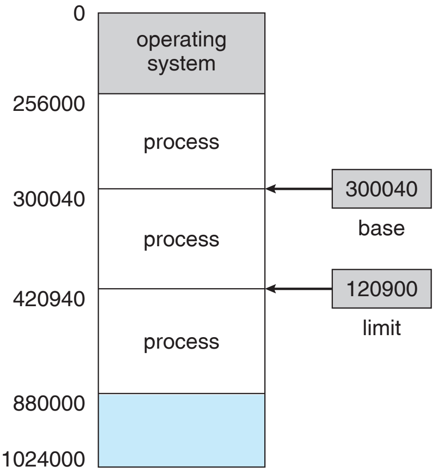
- The purpose is to protect everyone else (including the OS) from your program.

Memory Protection

Logical Address Space

Base register holds the smallest legal physical memory address

Limit register contains the size of the range



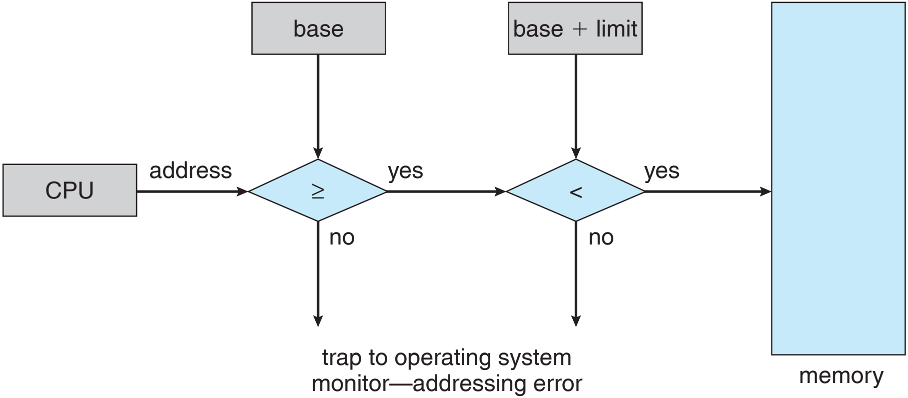
A pair of `base` and `limit` registers define the logical address space

`JMP 28`

`JMP 300068`

Memory Protection

Base and limit registers



UNIX View of a Process' Memory

```
max  +---------------------+
     |        Stack        |  Stack segment
     +---------------------+
     |          |          |
     |          v          |
     |                     |
     |          ^          |
     |          |          |
     +---------------------+
     |  Dynamic storage    |  Heap
     | (from new, malloc)  |
     +---------------------+
     |  Static variables   |
     |  (uninitialized,    |  BSS  segment
     |   initialized)      |  Data segment
     +---------------------+
     |        Code         |  Text segment
0    +---------------------+
```

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

- text: program code
- data: initialized global and static data
- bss: uninitialized global and static data
- heap: dynamically allocated with `malloc`, `new`
- stack: local variables

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 ...

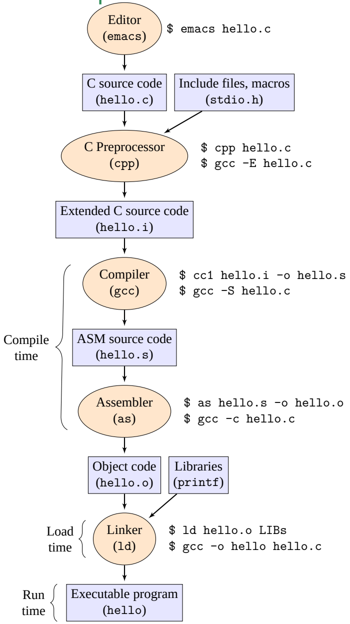
stack: `ulimit -s`

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:

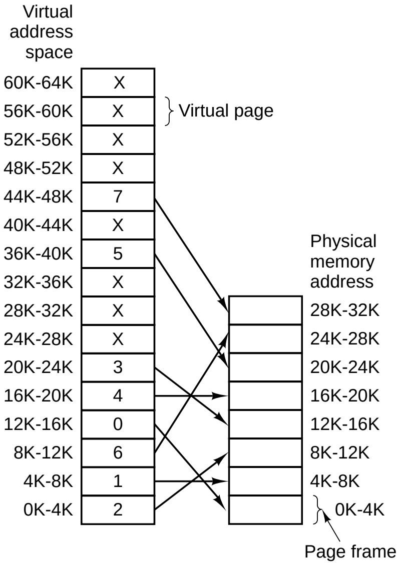
- Linker combines all the object files into a single executable object file
- Loader (part of OS) loads an executable object file into memory at location(s) determined by the OS
  - invoked via the `execve` system call

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

$$\textit{virtual address} \xrightarrow{\color{blue}{\textit{page table}}} \textit{physical address}$$

$$\textit{Page } 0 \xrightarrow{\textit{map to}} \textit{Frame } 2$$

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

$$\begin{matrix} 20500_{vir} \\ (20k+20)_{vir} \end{matrix} \xrightarrow{\textit{map to}} \begin{matrix} 12308_{phy} \\ (12k+20)_{phy} \end{matrix}$$

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

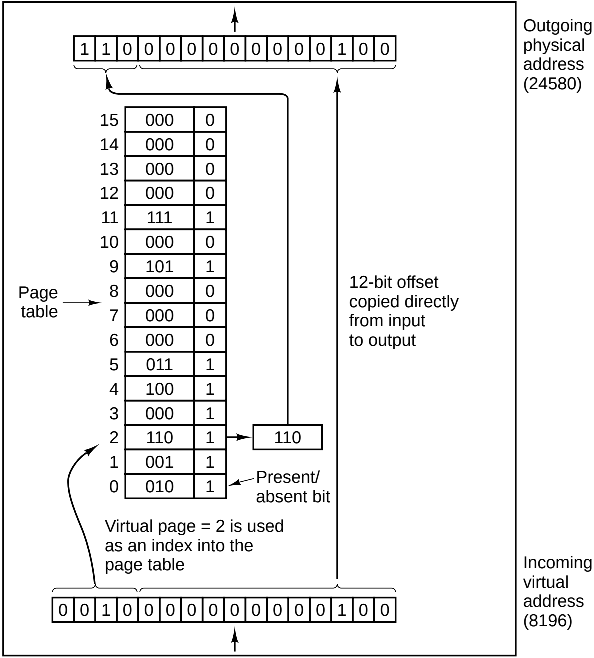
Given logical address space (2^m) and page size (2^n),

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

Example: addressing to 0010000000000100

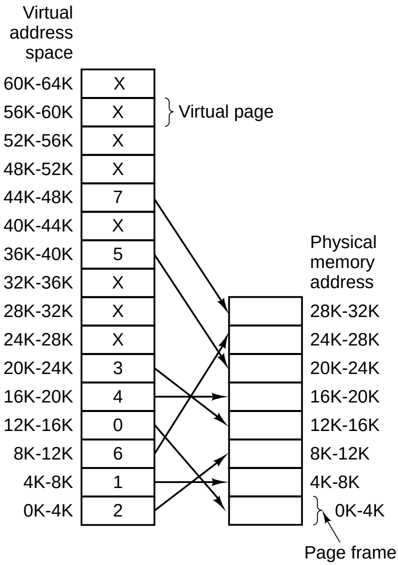
$$\underbrace{\overbrace{0\,0\,1\,0}^{m-n=4}\ \overbrace{0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0}^{n=12}}_{m=16}$$

$$\text{page number} = 0010 = 2, \quad \text{page offset} = 000000000100$$



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

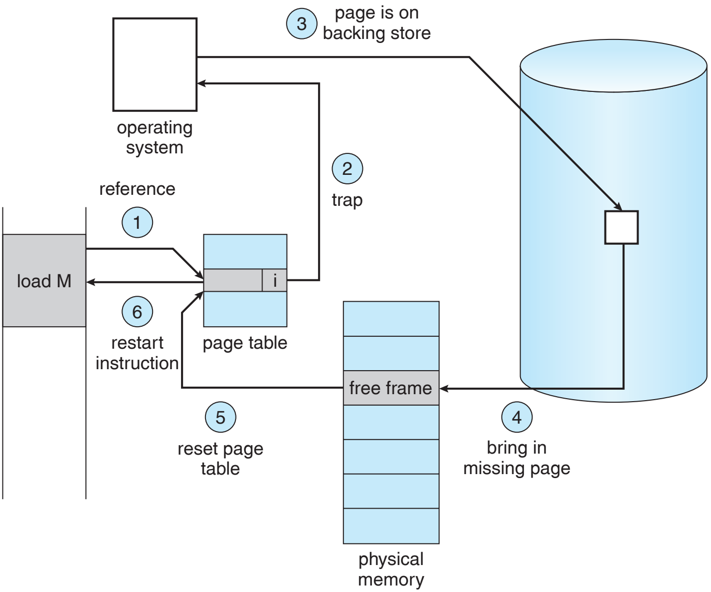
Page Fault



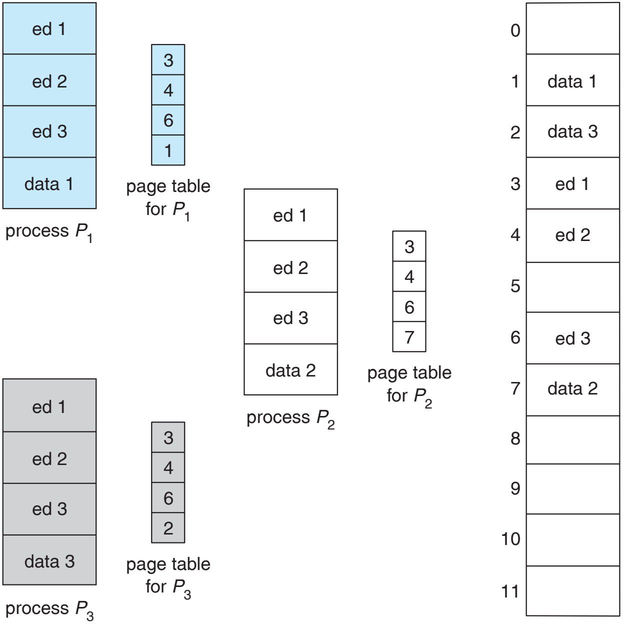
`MOV REG, 32780`?

➠ Page fault & swapping

Page Fault Handling



Shared Pages

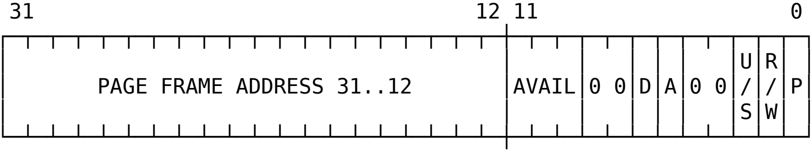


Page Table Entry

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



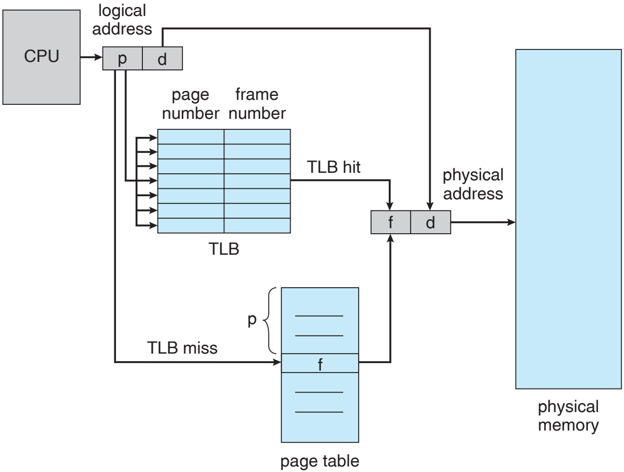

```
    P - PRESENT
  R/W - READ/WRITE
  U/S - USER/SUPERVISOR
    A - ACCESSED
    D - DIRTY
AVAIL - AVAILABLE FOR SYSTEMS PROGRAMMER USE

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



Multilevel Page Tables

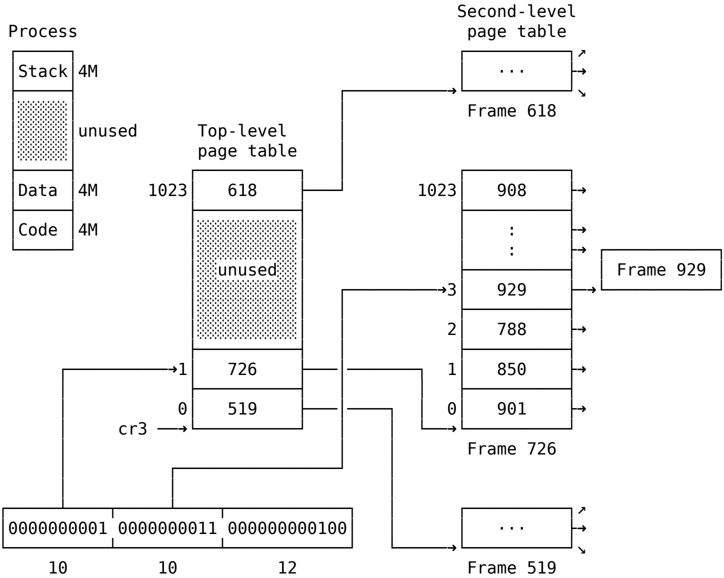
- a $1M$-entry page table eats $4M$ memory
- while 100 processes running, $400M$ memory is gone for page tables
- avoid keeping all the page tables in memory all the time

A two-level scheme

```
         page number        page offset
+----------+----------+------------+
|    p1    |    p2    |     d      |
+----------+----------+------------+
     10         10          12
      |          |
      |          └─> pointing to 1k frames
      └──> pointing to 1k page tables
```

Two-Level Page Tables

Example



Pentium Paging

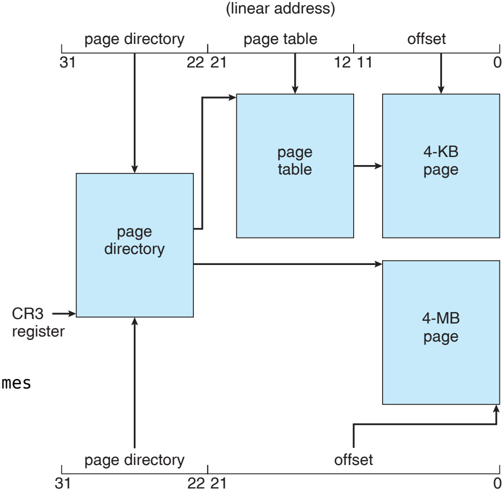
Linear Address ⇒ Physical Address

Two page size in Pentium:

4K: 2-level paging

4M: 1-level paging

```
   page number        page offset
+---------+---------+------------+
|   p1    |   p2    |     d      |
+---------+---------+------------+
    10        10          12
    |         |
    |         └─> pointing to 1k frames
    └─> pointing to 1k page tables
```



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

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

And this is for ONE process!

Multi-level?

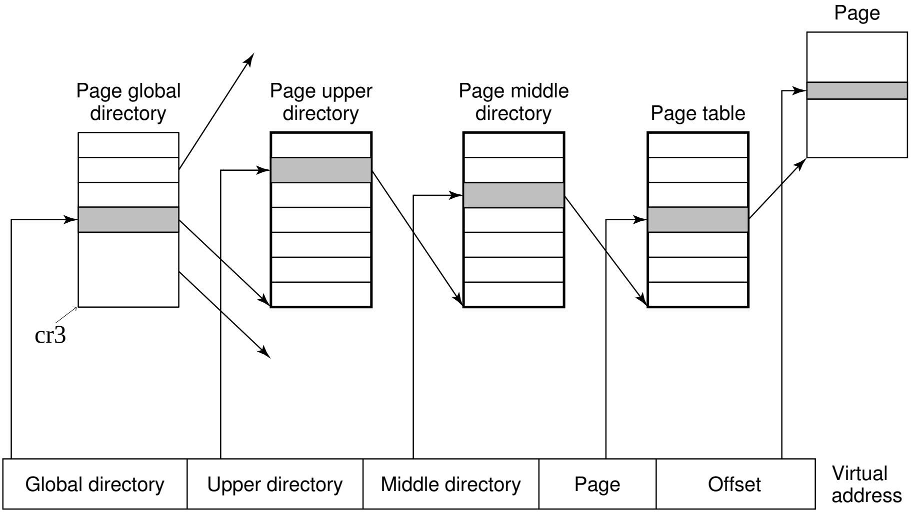
if $10\,bits$ for each level

then $\frac{64-12}{10} = 5$ levels are required

5 memory accress 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)

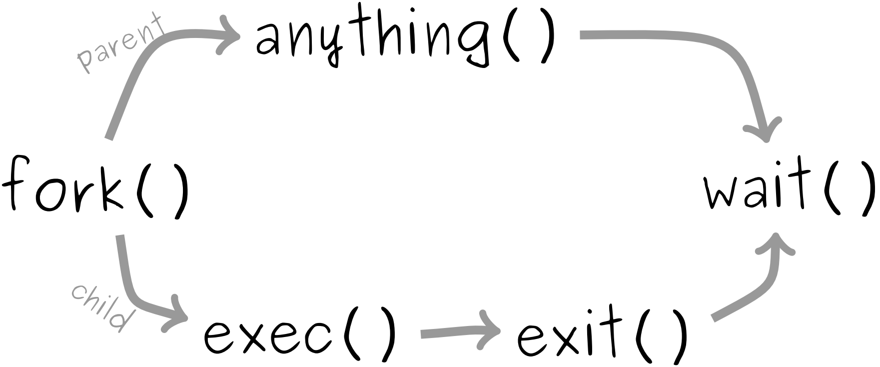
Implementation

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

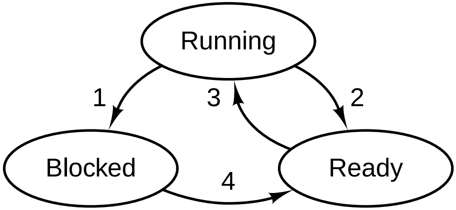
`process state`
`PID`
`program counter`
`registers`
`memory limits`
`list of open files`
`...`

Process Creation



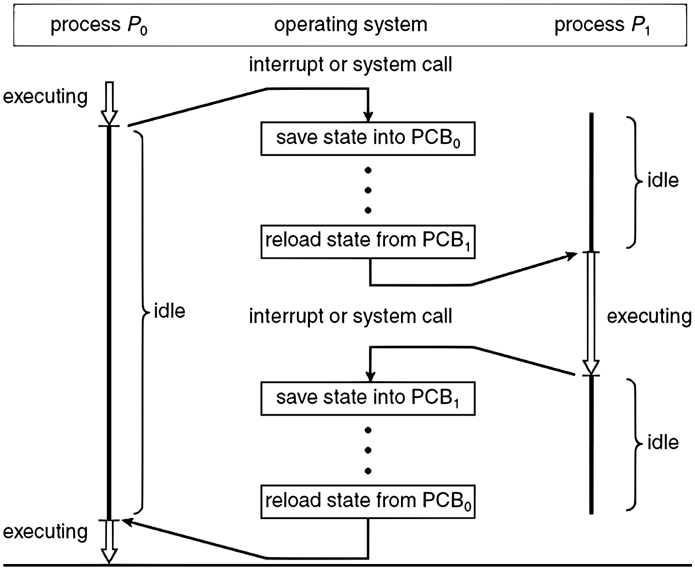
- When a process is created, it is almost identical to its parent
  - It receives a (logical) copy of the parent's address space, and
  - executes the same code as the parent
- The parent and child have separate copies of the data (stack and heap)

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

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

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

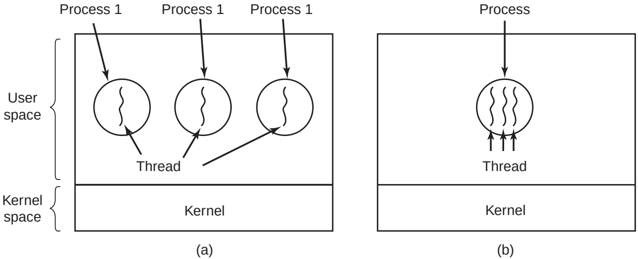
exec()

```c
int main()
{
  pid_t pid;
  /* fork another process */
  pid = fork();
  if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    exit(-1);
  }
  else if (pid == 0) { /* child process */
    execlp("/bin/ls", "ls", NULL);
  }
  else { /* parent process */
    /* wait for the child to complete */
    wait(NULL);
    printf ("Child Complete");
    exit(0);
  }
  return 0;
}
```

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
`pthread_create`	Create a new thread
`pthread_exit`	Terminate the calling thread
`pthread_join`	Wait for a specific thread to exit
`pthread_yield`	Release the CPU to let another thread run
`pthread_attr_init`	Create and initialize a thread's attribute structure
`pthread_attr_destroy`	Remove a thread's attribute structure

Pthreads

Example 1

```c
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

```c
#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

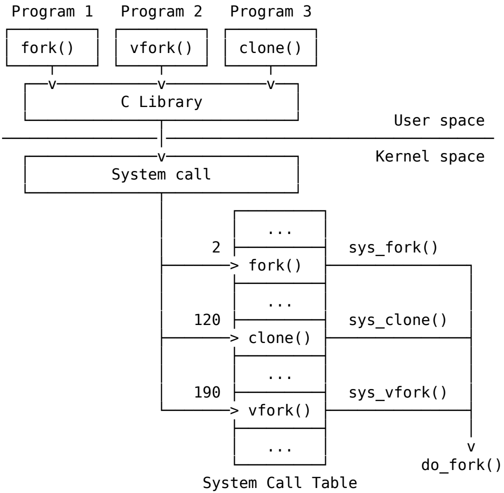
To the Linux kernel, there is no concept of a thread

- 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: $1.7\mu s$
fork: $1.8\mu s$

Linux Threads

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



clone()

```c
#include <sched.h>
int clone(int (*fn) (void *), void *child_stack,
          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
`CLONE_FS`	File-system info
`CLONE_VM`	Same memory space
`CLONE_SIGHAND`	Signal handlers
`CLONE_FILES`	The set of open files

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

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

clone() Example

```c
#include <unistd.h>
#include <sched.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>

int variable;

int do_something()
{
  variable = 42;
  _exit(0);
}
int main(void)
{
  void *child_stack;
  variable = 9;

  child_stack = (void *) malloc(8192);
  printf("The variable was %d\n", variable);

  clone(do_something, child_stack,
        CLONE_FS | CLONE_VM | CLONE_FILES, NULL);
  sleep(1);

  printf("The variable is now %d\n", variable);
  return 0;
}
```

clone() Example

```c
#include <unistd.h>
#include <sched.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>

int variable;

int do_something()
{
  variable = 42;
  _exit(0);
}
int main(void)
{
  void *child_stack;
  variable = 9;

  child_stack = (void *) malloc(8192);
  printf("The variable was %d\n", variable);

  clone(do_something, child_stack,
        CLONE_FS | CLONE_VM | CLONE_FILES, NULL);
  sleep(1);

  printf("The variable is now %d\n", variable);
  return 0;
}
```

Stack grows downwards

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

11 Signals

Signals

- Singals 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

```bash
#!/bin/bash

sigint(){
  echo -e "Why Ctrl-c?\n-> "
}

trap sigint SIGINT

echo -n "-> "

while read CMD; do
  $CMD
  echo -n "-> "
done
```

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

Example

SIGINT

```c
#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

```c
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

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

void cry(int sig)
{
  puts("C: I'm crying...");
  kill(getppid(),sig);
}

void complain(int sig)
{
  puts("P: You're noisy.");
}

int main()
{
  if ( fork() == 0 ){
    signal(SIGALRM, cry);
    alarm(2);
    pause();
  }

  signal(SIGALRM, complain);
  pause();
  exit(0);
}
```

Part VI

Interprocess Communication

Interprocess Communication

Example:

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

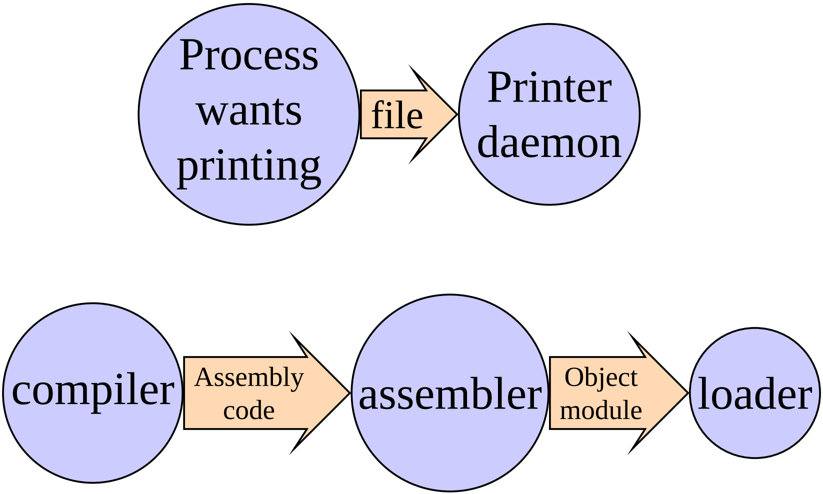
IPC issues:

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.

```
while(TRUE){
    while(FULL);
    item = produceItem();
    insertItem(item);
}
```

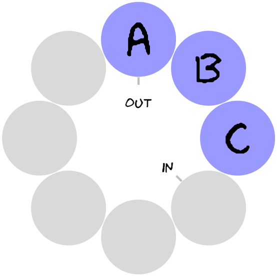
```
while(TRUE){
    while(EMPTY);
    item = removeItem();
    consumeItem(item);
}
```

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 "$(\texttt{in}+1)\%\texttt{BUFFER\_SIZE} == \texttt{out}$"

Actually, this is "$\texttt{full}-1$"

Empty: when "$\texttt{in} == \texttt{out}$"

Can only use "$\texttt{BUFFER\_SIZE}-1$" elements

Shared data:

```c
#define BUFFER_SIZE 6
typedef struct {
  ...
} item;
item buffer[BUFFER_SIZE];
int in = 0; //the next free position
int out = 0;//the first full position
```

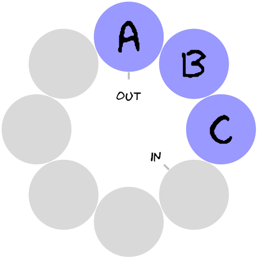
Bounded-Buffer Problem

Producer:

```
while (true) {
  /* do nothing -- no free buffers */
  while (((in + 1) % BUFFER_SIZE) == out);

  produce(buffer[in]);

  in = (in + 1) % BUFFER_SIZE;
}
```

Consumer:

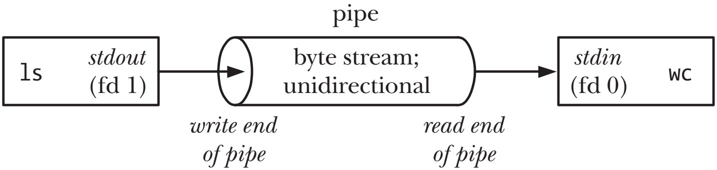
```
while (true) {
  while (in == out); /* do nothing */

  consume(buffer[out]);

  out = (out + 1) % BUFFER_SIZE;
}
```



12 Pipes and FIFOs

Pipe

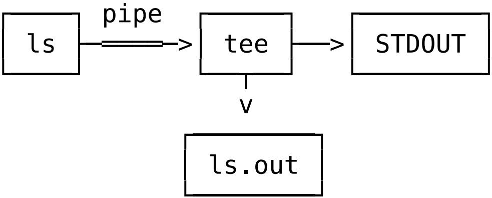
```
$ ls | wc -l
```

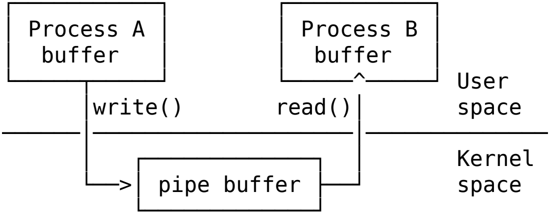


- A pipe is a byte stream
- Unidirectional
- `read()` would be blocked if nothing written at the other end

`tee`

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



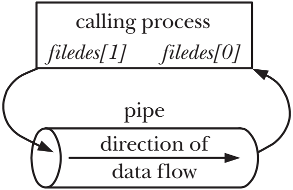


- No direct link between A and B (need system calls)
- A pipe is simply a buffer maintained in kernel memory

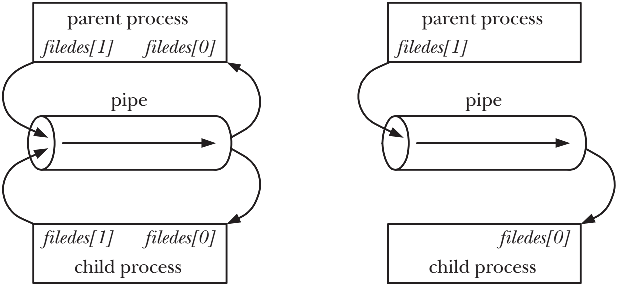
```
$ cat /proc/sys/fs/pipe-max-size
```

pipe()

```c
#include <unistd.h>

int pipe(int fd[2]);
```



pipe() + fork()



a) After *fork()*

b) After closing unused descriptors

```c
#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");
 ___________    ____________       ___________    ____________
|       fp  |<-| stdout     |     |       fp  |->| stdin      |
|___________|  |____________|     |___________|  |____________|

 parent         child (cmd)        parent         child (cmd)
```

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

```c
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

```c
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
```

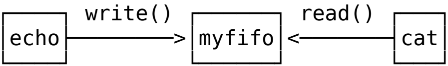
Named Pipe (FIFO)

PIPEs pass data between related processes.

FIFOs pass data between any processes.

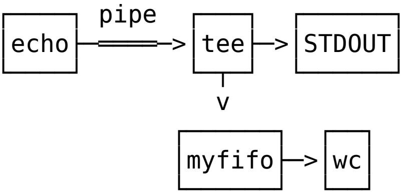
`$ mkfifo myfifo`

```
$ echo hello > myfifo
$ cat myfifo
```



tee

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



IPC With FIFO

```c
#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 (strncmp(*++argv, "O_RDONLY", 8) == 0)  mode |= O_RDONLY;
    if (strncmp(*argv, "O_WRONLY", 8) == 0)    mode |= O_WRONLY;
    if (strncmp(*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);
}
```

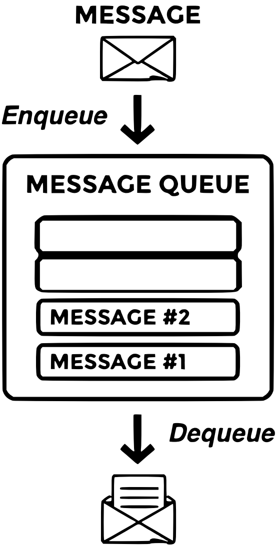
```
$ watch 'lsof -n.1 /tmp/myfifo'
$ ./a.out O_RDONLY
$ ./a.out O_WRONLY
$ ./a.out O_RDONLY O_NONBLOCK
$ ./a.out O_WRONLY O_NONBLOCK
```

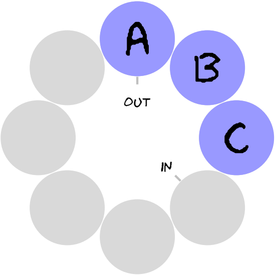
`O_NONBLOCK`

- 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

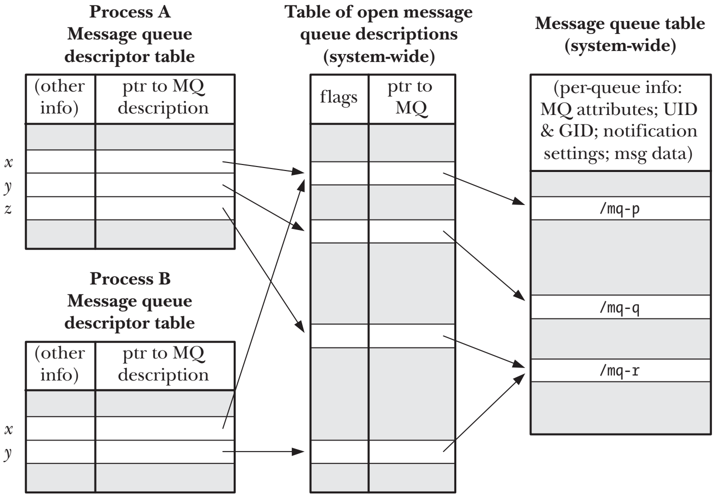
```c
int main(int argc, char **argv)
{
  mqd_t queue;
  struct mq_attr attrs;
  size_t msg_len;

  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

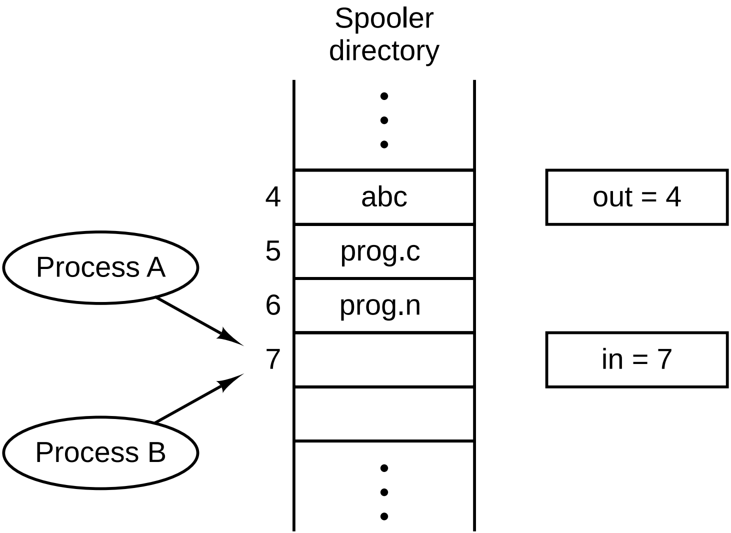
```
int main(int argc, char **argv)
{
  mqd_t queue;
  struct mq_attr attrs;
  char *msg_ptr;
  ssize_t recvd;
  size_t i;

  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



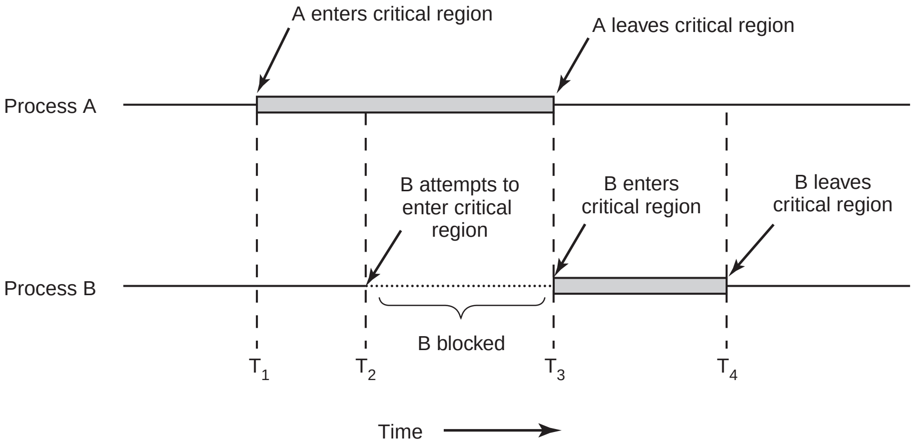
14 Semaphores

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

```
while(TRUE){
  while(turn != 0);
  critical_region();
  turn = 1;
  noncritical_region();
}
```

```
while(TRUE){
  while(turn != 1);
  critical_region();
  turn = 0;
  noncritical_region();
}
```

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

Mutual Exclusion With Busy Waiting

Peterson's Solution

```c
int interest[0] = 0;
int interest[1] = 0;
int turn;
```

P0

```c
interest[0] = 1;
turn = 1;
while(interest[1] == 1
              && turn == 1);
critical_section();
interest[0] = 0;
```

P1

```c
interest[1] = 1;
turn = 0;
while(interest[0] == 1
              && turn == 0);
critical_section();
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

✘

```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

```
down(S){
  while(S<=0);
  S--;
}
```

```
up(S){
  S++;
}
```

Atomic Operations	
`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

```
statement a1
sem.signal()
```

```
sem.wait()
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!

```c
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

```
LOAD    [i], r0   ;load the value of 'i' into
                  ;a register from memory
ADD     r0, 1     ;increment the value
                  ;in the register
STORE   r0, [i]   ;write the updated
                  ;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

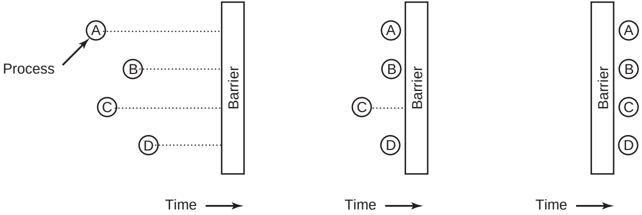
```
mutex.wait()
    i++
mutex.signal()
```

```
mutex.wait()
    i++
mutex.signal()
```

```c
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);
}
```

```c
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

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

count: keeps track of how many threads have arrived

mutex: provides exclusive access to `count`

barrier: is locked (≤ 0) until all threads arrive

When `barrier.value < 0`,

`barrier.value == Number of queueing processes`

```
specific_task();
mutex.wait();
   count++;
mutex.signal();
if (count < n)
   barrier.wait();
barrier.signal();
critical_point();
```

```
specific_task();
mutex.wait();
   count++;
mutex.signal();
if (count == n)
   barrier.signal();
barrier.wait();
critical_point();
```

Barrier Solution

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

count: keeps track of how many threads have arrived

mutex: provides exclusive access to `count`

barrier: is locked (≤ 0) until all threads arrive

When `barrier.value < 0`,

```
barrier.value == Number of queueing processes
```

```
specific_task();
mutex.wait();
   count++;
mutex.signal();
if (count < n)
   barrier.wait();
barrier.signal();
critical_point();
```

```
specific_task();
mutex.wait();
   count++;
mutex.signal();
if (count == n)
   barrier.signal();
barrier.wait();
critical_point();
```

Deadlock!

Only one thread can pass the barrier!

Barrier Solution

```
specific_task();

mutex.wait();
    count++;
mutex.signal();

if (count == n)
    barrier.signal();

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

critical_point();
```

```
specific_task();

mutex.wait();
    count++;

    if (count == n)
        barrier.signal();

    barrier.wait();
    barrier.signal();
mutex.signal();

critical_point();
```

Barrier Solution

```
specific_task();

mutex.wait();
    count++;
mutex.signal();

if (count == n)
    barrier.signal();

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

critical_point();
```

```
specific_task();

mutex.wait();
    count++;

    if (count == n)
        barrier.signal();

    barrier.wait();
    barrier.signal();
mutex.signal();

critical_point();
```

Deadlock!

☠ 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

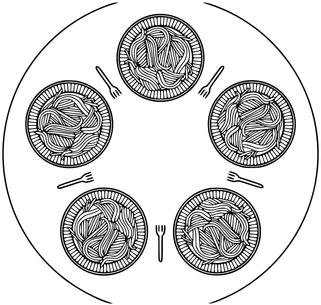
- it allows one thread to pass at a time, and
- it can be locked to bar all threads

15 Classical IPC Problems

15.1 The Dining Philosophers Problem

The Dining Philosophers Problem

```python
while True:
    think()
    get_forks()
    eat()
    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

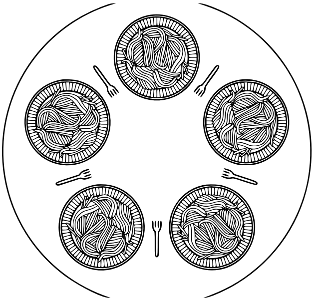
```
#define N 5                               /* number of philosophers */

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

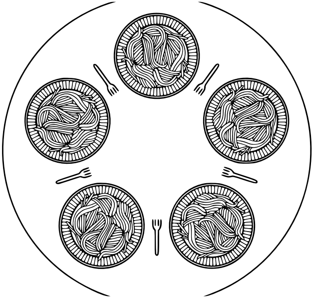
```c
#define N 5
semaphore mutex=1;

void philosopher(int i)
{
    while (TRUE) {
      think();
      wait(&mutex);
         take_fork(i);
         take_fork((i+1) % N);
         eat();
         put_fork(i);
         put_fork((i+1) % N);
      signal(&mutex);
    }
}
```



The Dining Philosophers Problem

With One Mutex

```c
#define N 5
semaphore mutex=1;

void philosopher(int i)
{
    while (TRUE) {
      think();
      wait(&mutex);
         take_fork(i);
         take_fork((i+1) % N);
         eat();
         put_fork(i);
         put_fork((i+1) % N);
      signal(&mutex);
    }
}
```



- 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

```c
#define N 5                 /* number of philosophers */
#define LEFT (i+N-1)%N      /* number of i's left neighbor */
#define RIGHT (i+1)%N       /* number of i's right neighbor */
#define THINKING 0          /* philosopher is thinking */
#define HUNGRY 1            /* philosopher is trying to get forks */
#define EATING 2            /* philosopher is eating */
typedef int semaphore;
int state[N];               /* state of everyone */
semaphore mutex = 1;        /* for critical regions */
semaphore s[N];             /* one semaphore per philosopher */

void philosopher(int i)  /* i: philosopher number, from 0 to N-1 */
{
     while (TRUE) {
          think( );
          take_forks(i); /* acquire two forks or block */
          eat( );
          put_forks(i);  /* put both forks back on table */
     }
}
```

The Dining Philosophers Problem

AST Solution (Part 2)

```
void take_forks(int i)              /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                   /* enter critical region */
    state[i] = HUNGRY;              /* record fact that philosopher i is hungry */
    test(i);                        /* try to acquire 2 forks */
    up(&mutex);                     /* exit critical region */
    down(&s[i]);                    /* block if forks were not acquired */
}
void put_forks(i)                   /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                   /* enter critical region */
    state[i] = THINKING;            /* philosopher has finished eating */
    test(LEFT);                     /* see if left neighbor can now eat */
    test(RIGHT);                    /* see if right neighbor can now eat */
    up(&mutex);                     /* exit critical region */
}
void test(i)                        /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

The Dining Philosophers Problem

AST Solution (Part 2)

```c
void take_forks(int i)          /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);               /* enter critical region */
    state[i] = HUNGRY;          /* record fact that philosopher i is hungry */
    test(i);                    /* try to acquire 2 forks */
    up(&mutex);                 /* exit critical region */
    down(&s[i]);                /* block if forks were not acquired */
}
void put_forks(i)               /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);               /* enter critical region */
    state[i] = THINKING;        /* philosopher has finished eating */
    test(LEFT);                 /* see if left neighbor can now eat */
    test(RIGHT);                /* see if right neighbor can now eat */
    up(&mutex);                 /* exit critical region */
}
void test(i)                    /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Starvation!

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.

```
semaphore mutex = 1;
semaphore noOther = 1;
int readers = 0;

void writer(void)
{
  while (TRUE) {
    wait(&noOther);
      writing();
    signal(&noOther);
  }
}
```

```
void reader(void)
{
  while (TRUE) {
    wait(&mutex);
      readers++;
      if (readers == 1)
          wait(&noOther);
    signal(&mutex);
    reading();
    wait(&mutex);
      readers--;
      if (readers == 0)
        signal(&noOther);
    signal(&mutex);
    anything();
  }
}
```

The Readers-Writers Problem

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

```
semaphore mutex = 1;
semaphore noOther = 1;
int readers = 0;

void writer(void)
{
  while (TRUE) {
    wait(&noOther);
      writing();
    signal(&noOther);
  }
}
```

```
void reader(void)
{
  while (TRUE) {
    wait(&mutex);
      readers++;
      if (readers == 1)
        wait(&noOther);
    signal(&mutex);
    reading();
    wait(&mutex);
      readers--;
      if (readers == 0)
        signal(&noOther);
    signal(&mutex);
    anything();
  }
}
```

Starvation!

The Readers-Writers Problem

No starvation

```
semaphore mutex = 1;
semaphore noOther = 1;
semaphore turnstile = 1;
int readers = 0;

void writer(void)
{
  while (TRUE) {
    turnstile.wait();
      wait(&noOther);
        writing();
      signal(&noOther);
    turnstile.signal();
  }
}
```

```
void reader(void)
{
  while (TRUE) {
    turnstile.wait();
    turnstile.signal();

    wait(&mutex);
      readers++;
      if (readers == 1)
          wait(&noOther);
    signal(&mutex);
    reading();
    wait(&mutex);
      readers--;
      if (readers == 0)
        signal(&noOther);
    signal(&mutex);
    anything();
  }
}
```

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

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

```c
void barber(void)
{
  while (TRUE) {
    wait(&customers);
    wait(&mutex);
      waiting--;
    signal(&mutex);
    cutHair();
    signal(&bber);
  }
}
```

```c
void customer(void)
{
  if(waiting == CHAIRS)
    goHome();
  else {
    wait(&mutex);
    waiting++;
    signal(&mutex);
    signal(&customers);
    wait(&bber);
    getHairCut();
  }
}
```

Solution2

```c
#define CHAIRS 5
semaphore customers = 0;
semaphore bber = ?;
semaphore mutex = 1;
int waiting = 0;

void barber(void)
{
  while (TRUE) {
    wait(&customers);
    cutHair();
  }
}
```

```c
void customer(void)
{
  if (waiting == CHAIRS)
    goHome();
  else {
    wait(&mutex);
    waiting++;
    signal(&mutex);
    signal(&customers);
    wait(&bber);
    getHairCut();
    signal(&bber);
    wait(&mutex);
      waiting--;
    signal(&mutex);
  }
}
```

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

<table>
<tr><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><td colspan="16">Source Port</td><td colspan="16">Destination Port</td></tr>
<tr><td colspan="32">Sequence Number</td></tr>
<tr><td colspan="32">Acknowledgment Number</td></tr>
<tr><td colspan="4">Data Offset</td><td>0</td><td>0</td><td>0</td><td>NS</td><td>CWR</td><td>ECE</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td colspan="16">Window</td></tr>
<tr><td colspan="16">Checksum</td><td colspan="16">Urgent Pointer</td></tr>
<tr><td colspan="24">Options</td><td colspan="8">Padding</td></tr>
</table>

Message Passing

The producer-consumer problem

```
#define N 100  /* number of slots in the buffer */
void producer(void)
{
    int item;
    message m;                   /* message buffer */
    while (TRUE) {
        item = produce_item();    /* generate something to put in buffer */
        receive(consumer, &m);    /* wait for an empty to arrive */
        build_message(&m, item);  /* construct a message to send */
        send(consumer, &m);       /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;
    for (i=0; i<N; i++) send(producer, &m);  /* send N empties */
    while (TRUE) {
        receive(producer, &m);    /* get message containing item */
        item = extract_item(&m);  /* extract item from message */
        send(producer, &m);       /* send back empty reply */
        consume_item(item);       /* do something with the item */
    }
}
```

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

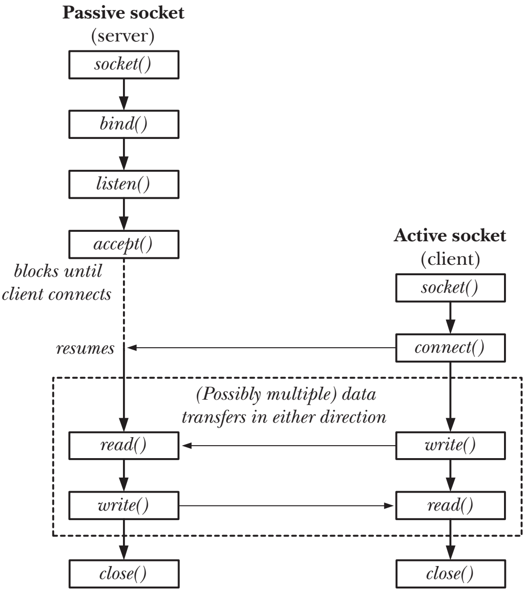
```
struct sockaddr_un addr;
```

✔ `bind(sfd, (struct sockaddr *)&addr, sizeof(struct sockaddr_un));`

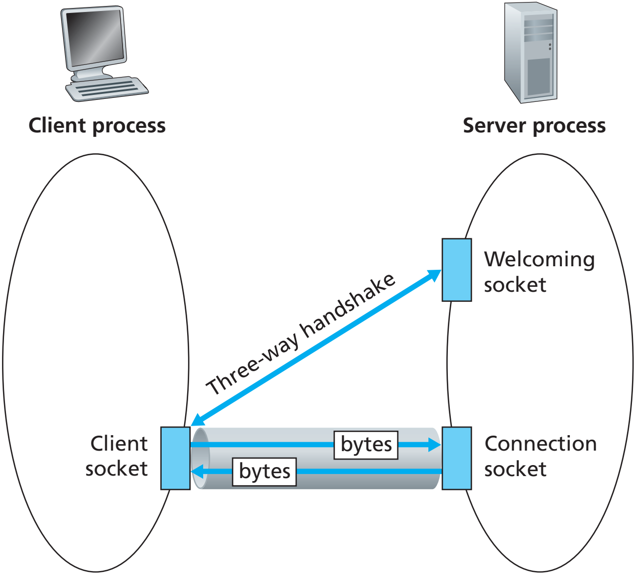
✘ `bind(sfd, &addr, sizeof(struct sockaddr_un));`

17.1 Stream Sockets

Stream Sockets



Two Sockets at the Server

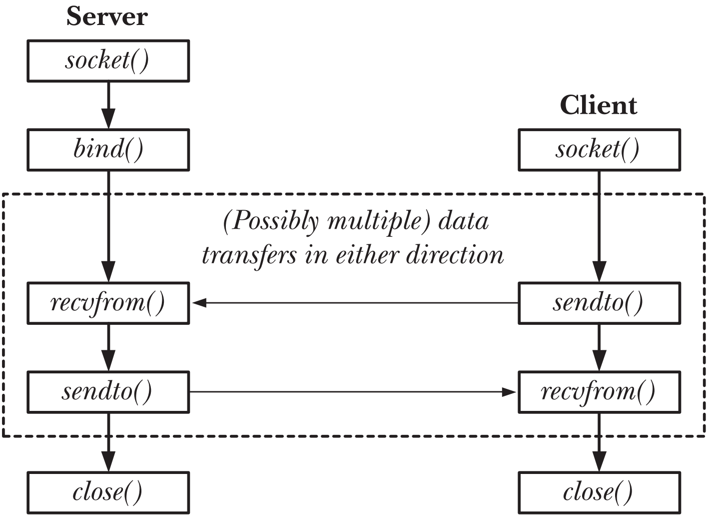


Socket System Calls

`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

```c
#define SV_SOCK_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_SOCK_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_SOCK_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_SOCK_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

```c
struct in_addr {                /* IPv4 4-byte address */
    in_addr_t s_addr;           /* Unsigned 32-bit integer */
};

struct sockaddr_in {            /* IPv4 socket address */
    sa_family_t    sin_family; /* Address family (AF_INET) */
    in_port_t      sin_port;   /* Port number */
    struct in_addr sin_addr;   /* IPv4 address */
    unsigned char  __pad[X];   /* Pad to size of sockaddr (16 bytes) */
};
```

IPv6

```c
struct in6_addr {                  /* IPv6 address structure */
    uint8_t s6_addr[16];           /* 16 bytes == 128 bits */
};

struct sockaddr_in6 {              /* IPv6 socket address */
    sa_family_t sin6_family;       /* Address family (AF_INET6) */
    in_port_t   sin6_port;         /* Port number */
    uint32_t    sin6_flowinfo;     /* IPv6 flow information */
    struct in6_addr sin6_addr;     /* IPv6 address */
    uint32_t    sin6__scope_id;    /* Scope ID (new in kernel 2.4) */
};
```