

Computer Systems and Organization 2

themes

automating building software

- libraries, taking advantage of incremental compilation

sharing machines

- multiple users/programs on one system

parallelism and concurrency

- doing two+ things at once

under the hood of sockets

- layered design of networks

- implementing secure communication

under the hood of fast processors

- caching, (hidden) parallelism, avoiding idle time

themes

automating building software

- libraries, taking advantage of incremental compilation

sharing machines

- multiple users/programs on one system

parallelism and concurrency

- doing two+ things at once

under the hood of sockets

- layered design of networks
- implementing secure communication

under the hood of fast processors

- caching, (hidden) parallelism, avoiding idle time

make

```
$ ./foo.exe
```

```
...
```

```
...
```

```
$ edit readline.c
```

```
$ make
```

```
clang -g -O -Wall -c readline.c -o readline.o
```

```
ar rcs terminal.o readline.o libreadline.a
```

```
clang -o foo.exe foo.o foo-utility.o -L. -lreadline
```

```
$
```

themes

automating building software

libraries, taking advantage of incremental compilation

sharing machines

multiple users/programs on one system

parallelism and concurrency

doing two+ things at once

under the hood of sockets

layered design of networks

implementing secure communication

under the hood of fast processors

caching, (hidden) parallelism, avoiding idle time

address translation



address translation



address translation

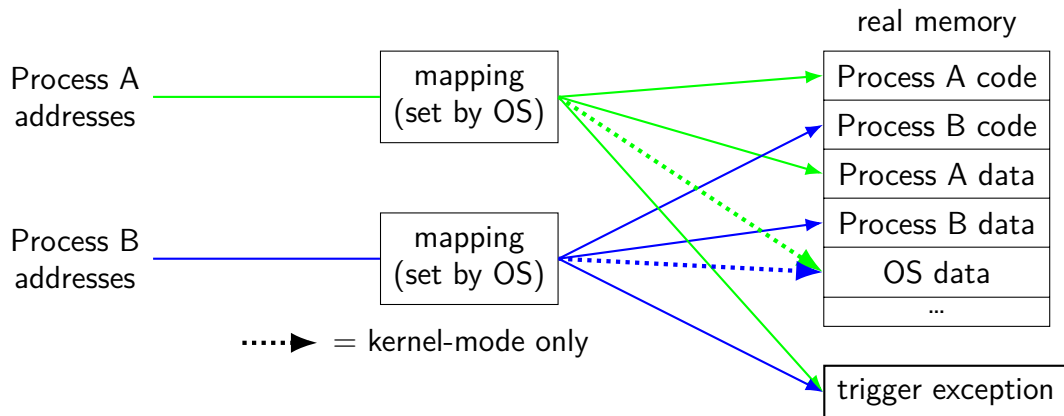


address translation



address spaces

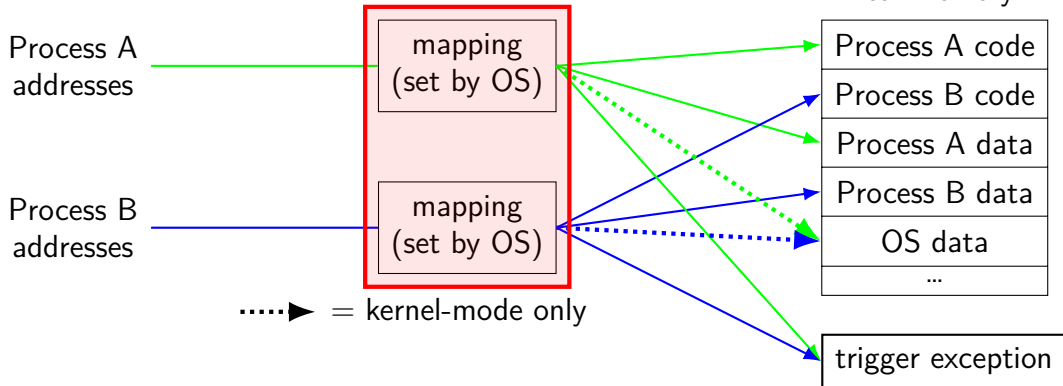
illusion of **dedicated memory**



address spaces

illusion of **dedicated memory**

chose one during context switch



themes

automating building software

libraries, taking advantage of incremental compilation

sharing machines

multiple users/programs on one system

parallelism and concurrency

doing two+ things at once

under the hood of sockets

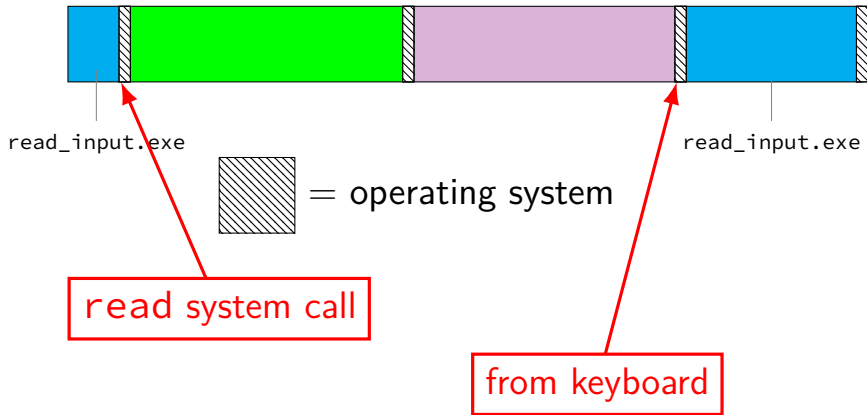
layered design of networks

implementing secure communication

under the hood of fast processors

caching, (hidden) parallelism, avoiding idle time

keyboard input timeline



time multiplexing

processor:



time multiplexing

processor:



...

```
call get_time
```

```
// whatever get_time does
```

```
movq %rax, %rbp
```

———— million cycle delay ————

```
call get_time
```

```
// whatever get_time does
```

```
subq %rbp, %rax
```

...

time multiplexing



...

```
call get_time
```

```
// whatever get_time does
```

```
movq %rax, %rbp
```

———— million cycle delay ————

```
call get_time
```

```
// whatever get_time does
```

```
subq %rbp, %rax
```

...

multiple cores+threads

core 1:



core 2:



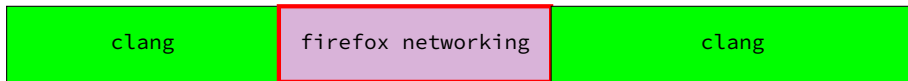
multiple cores? each core still divided up

multiple cores+threads

core 1:



core 2:



one program with multiple *threads*

themes

automating building software

libraries, taking advantage of incremental compilation

sharing machines

multiple users/programs on one system

parallelism and concurrency

doing two+ things at once

under the hood of sockets

layered design of networks

implementing secure communication

under the hood of fast processors

caching, (hidden) parallelism, avoiding idle time

permissions

```
$ ls /u/other/secret
```

```
ls: cannot open directory '/u/other/secret': Permission denied
```

```
$ shutdown
```

```
shutdown: Permission denied
```

themes

automating building software

libraries, taking advantage of incremental compilation

sharing machines

multiple users/programs on one system

parallelism and concurrency

doing two+ things at once

under the hood of sockets

layered design of networks

implementing secure communication

under the hood of fast processors

caching, (hidden) parallelism, avoiding idle time

layers

application	HTTP, SSH, SMTP, ...	application-defined meanings
transport	TCP, UDP, ...	reach correct program, reliability/streams
network	IPv4, IPv6, ...	reach correct machine (across networks)
link	Ethernet, Wi-Fi, ...	coordinate shared wire/radio
physical	...	encode bits for wire/radio

names and addresses

name	address
logical identifier	location/how to locate
variable counter	memory address 0x7FFF9430
DNS name www.virginia.edu	IPv4 address 128.143.22.36
DNS name mail.google.com	IPv4 address 216.58.217.69
DNS name mail.google.com	IPv6 address 2607:f8b0:4004:80b::2005
DNS name reiss-t3620.cs.virginia.edu	IPv4 address 128.143.67.91
DNS name reiss-t3620.cs.virginia.edu	MAC address 18:66:da:2e:7f:da
service name https	port number 443
service name ssh	port number 22

secure communication?

how do you know who your socket is to?

who can read what's on the socket?

what can you do to restrict this?

themes

automating building software

libraries, taking advantage of incremental compilation

sharing machines

multiple users/programs on one system

parallelism and concurrency

doing two+ things at once

under the hood of sockets

layered design of networks

implementing secure communication

under the hood of fast processors

caching, (hidden) parallelism, avoiding idle time

2004 CPU

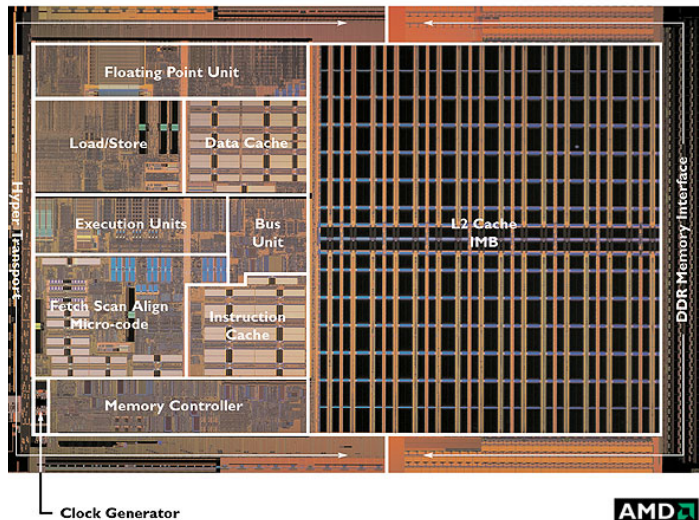


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU

▲ Registers

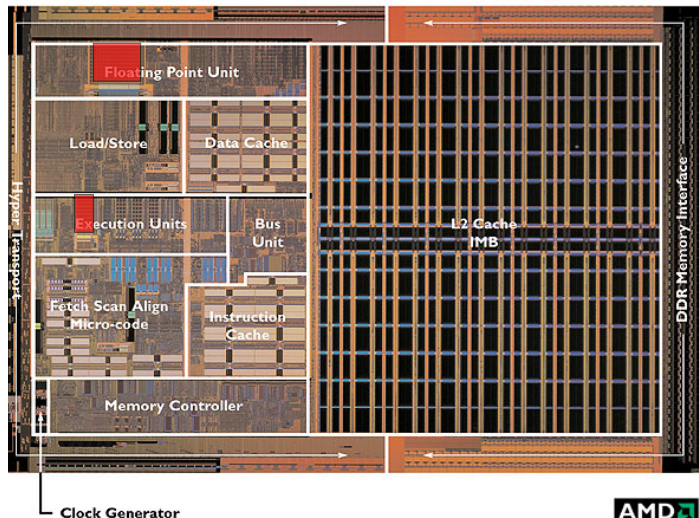


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU

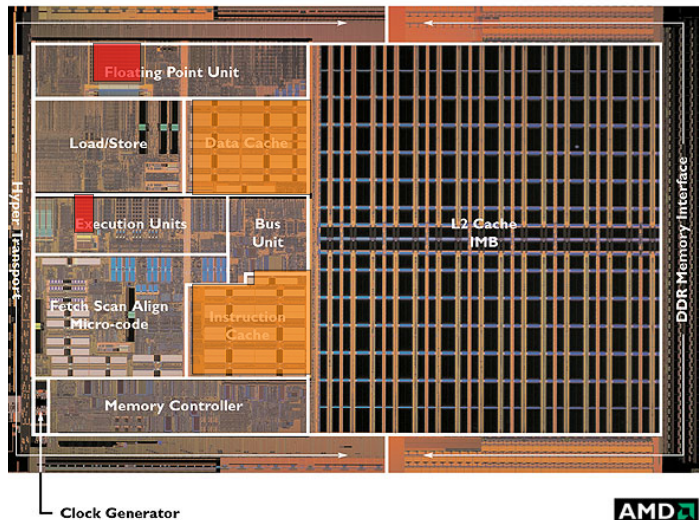


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU

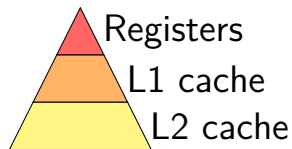
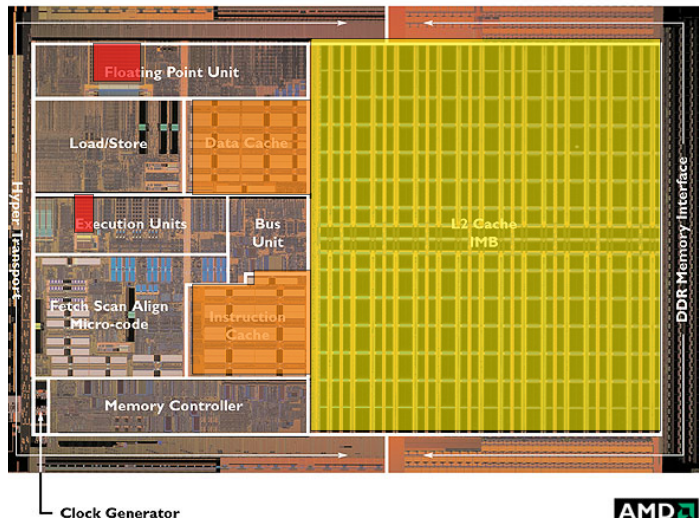


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU



Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU

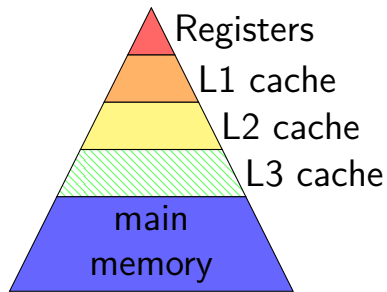
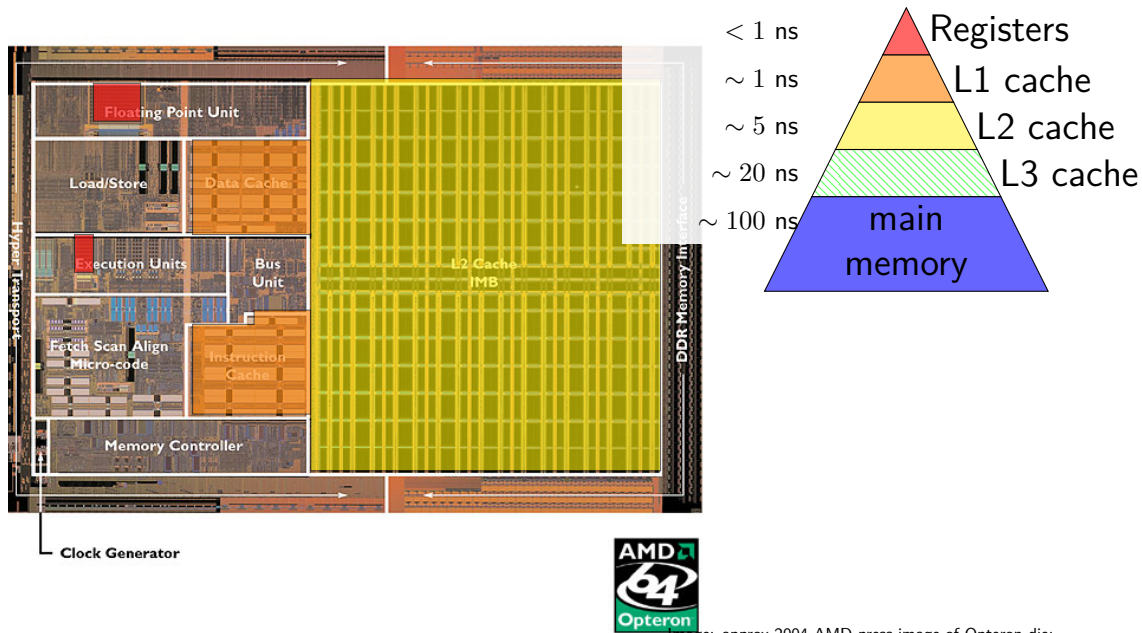


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU



some performance examples

example1:

```
    movq $1000000000000, %rax
loop1:
    addq %rbx, %rcx
    decq %rax
    jge loop1
    ret
```

about 30B instructions

my desktop: approx 2.65 sec

example2:

```
    movq $1000000000000, %rax
loop2:
    addq %rbx, %rcx
    addq %r8, %r9
    decq %rax
    jge loop2
    ret
```

about 40B instructions

my desktop: approx 2.65 sec

some performance examples

example1:

```
    movq $1000000000000, %rax
loop1:
    addq %rbx, %rcx
    decq %rax
    jge loop1
    ret
```

about 30B instructions

my desktop: approx 2.65 sec

example2:

```
    movq $1000000000000, %rax
loop2:
    addq %rbx, %rcx
    addq %r8, %r9
    decq %rax
    jge loop2
    ret
```

about 40B instructions

my desktop: approx 2.65 sec

logistics

labs

attend lab in person and get checked off by TA, *or*

(most labs) submit something to submission site and we'll grade it

submit to submission site? don't care if you attend the lab

more strict about submissions without checkoffs

in-person lab checkoff of incomplete lab at least 50% credit

if both checkoff + submission, will use higher score

some labs will basically require attendance

or contact me for other arrangements if you can't (sick, etc.)

logistically won't work otherwise — e.g. code review

lab collaboration and submissions

please collaborate on labs!

when working with others on lab and submitting code files

please indicate who you worked with in those files
via comment or similar

lab space

if labs are full, might kick out students from 'wrong' lab section

for 3:30pm, please come to registered room

for 5pm, 6:30pm, based on registration
should only need one room

plan: those labs Rice 130 only

will send announcement/have something posted on unused room

homeworks

several homework assignments

done individually

generally due on Fridays

(tentative dates on schedule)

homework/lab automatic testing

some homeworks/labs have automatic testing

with some delay after you submit

- usually 10s of minutes

- depending on assignment, number of submissions in queue

- if you submit very early, testing program might not be setup yet

when testing program doesn't understand/can't test something,
left for manual grading (“not yet graded”)

intention is that testing results are not surprises

if you did some manual testing (no hidden requirements, etc.)

if you think testing program made a mistake,
please submit regrade request

warmup assignment

first homework

due week from Friday 8 Sep @ 11:59pm

write C function to split a string into array of strings
with dynamic memory allocation

write C program to call function using input/command-line
arguments

write Makefile for it (next topic, next week's lab)

quizzes

released evening after Thursday lecture
starting *next* week

due 15 minutes before lecture on Tuesdays

about lecture and/or lab from the prior week

4–6 questions

individual, open book, open notes, open Internet

okay: looking up resources/tutorials/etc.

not okay: asking Stack Overflow the quiz question

not okay: IMing your friend the quiz question

on help on quiz questions

I and the TAs won't answer quiz questions...

but we will answer questions about the lecture material, etc.

(and TAs (not you) are responsible for knowing
what they can't answer

but we'd prefer you don't try to test those limits)

going over past quizzes

have in past gone over quiz Qs in lecture

either when a lot missed it

or on request in lecture

also fine office hour/Piazza question

readings

in lieu of textbook, have readings

mostly written by Prof Tychnovich (now at UIUC) with edits by me

on website; should be indicated with corresponding lecture

lecture + assignment sync

generally:

quiz after lecture and/or lab coverage

labs after lecture coverage

homework after lab coverage

means homework (and sometimes quiz)

may be relatively delayed from lecture coverage

exams

1 final exam

no midterms — instead:

- quizzes count a lot

- slightly more homework/lab than pilot

development enviroment

official: department machines via SSH or NX (remote desktop)

you can also use your own machines, but...

we will test your code on x86-64 Linux

I haven't checked assignments on a Windows or OS X machine

getting help

office hours — calendar will be posted on website

mix of in-person and remote, indicated on calendar

remote OH will use Discord + online queue

in-person OH may or may not — indicated on whiteboard, probably

Piazza

use private questions if homework code, etc.

emailing me (preferably with '3130' in subject)

collaboration (1)

labs — you can/should work with other students
everyone should understand the work submitted

homeworks — individual

write your own code / do not share your code
can ask/look up conceptual questions of others
others includes other students, Q&A sites, code generation tools, etc.
cite any sources you use (comments in code)

collaboration (2)

quizzes — individual

but open book+notes+etc.

can/should have help reviewing lecture/readings/etc.

legitimate questions for office hours

don't ask other students, stack overflow, gen AI tools, etc. the quiz questions

don't try to find exactly the quiz question on stack overflow

feedback

anonymous feedback on Canvas

would appreciate feedback (esp. when I can do something)

(but not a good way to ask for regrades, etc.)

late policy

no late quizzes

one quiz dropped (unconditionally)

90% credit for 0–72 hours late homeworks

for labs that allow submission only

lab submission due time is 11:59am the next day

90% credit for 0–24 hours late

no late lab checkoffs except by special arrangement

excused lateness

special circumstances?

illness, emergency, etc.

contact me, we'll figure something out

please don't attend lab/etc. sick!

attendance

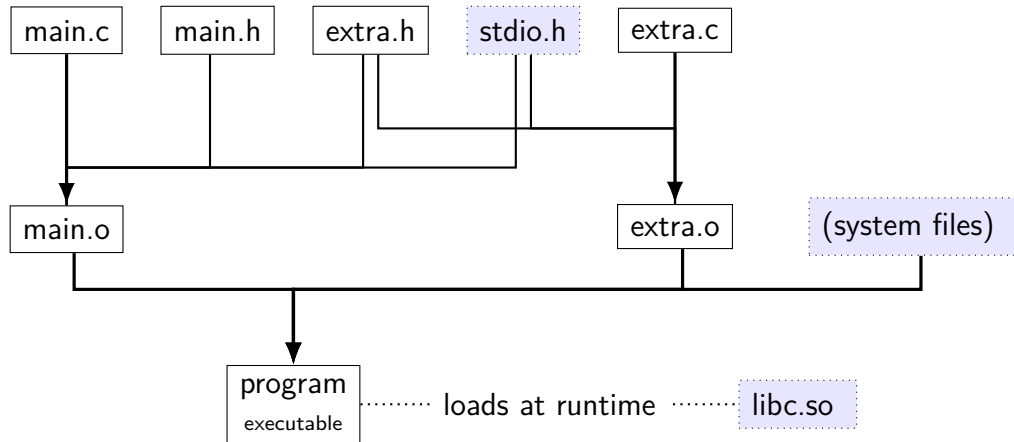
I won't take attendance in lecture

I will attempt to have lecture recordings

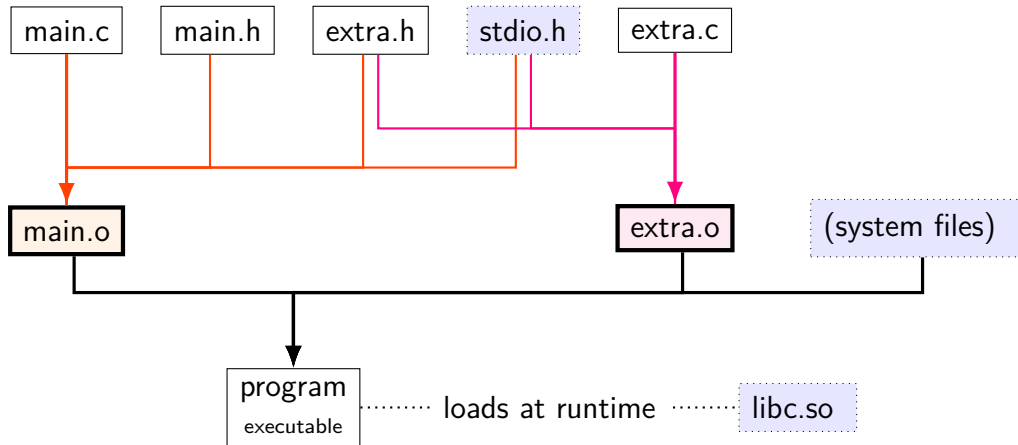
sometimes there may be issues with the recording

building

files in building C programs [dynamic linking]

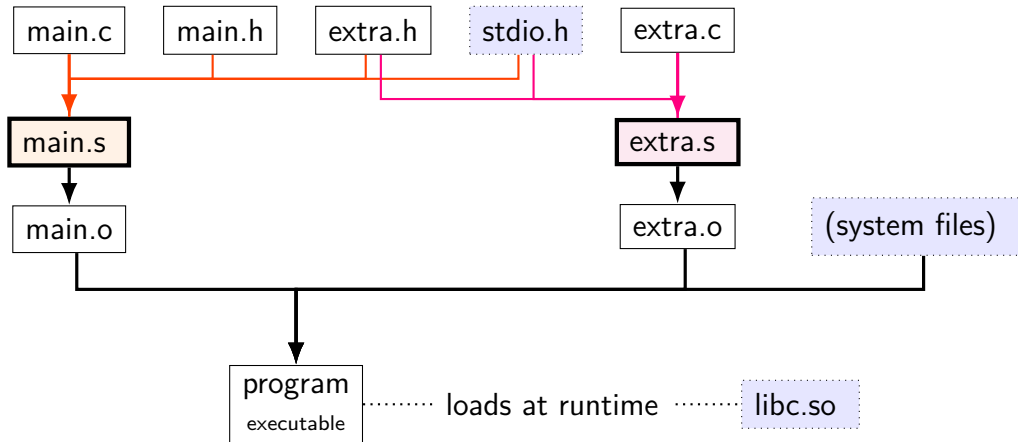


files in building C programs [dynamic linking]



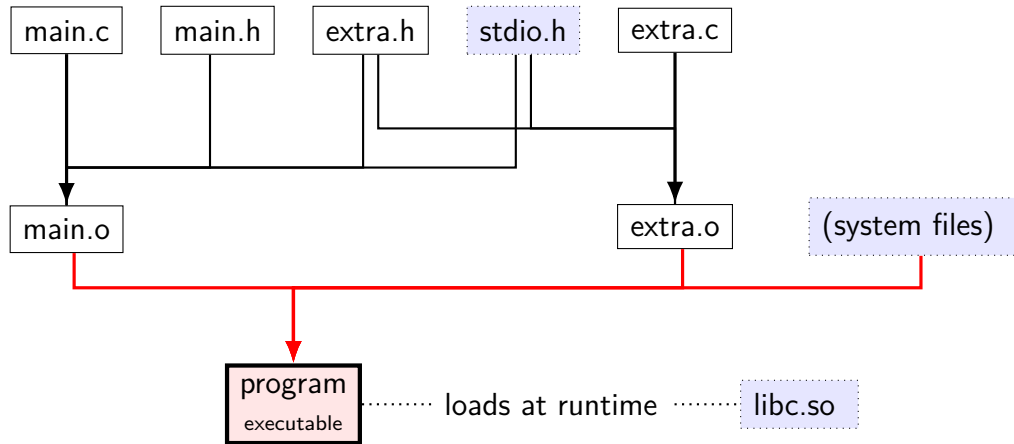
```
clang -c main.c  
clang -c extra.c
```

files in building C programs [dynamic linking]



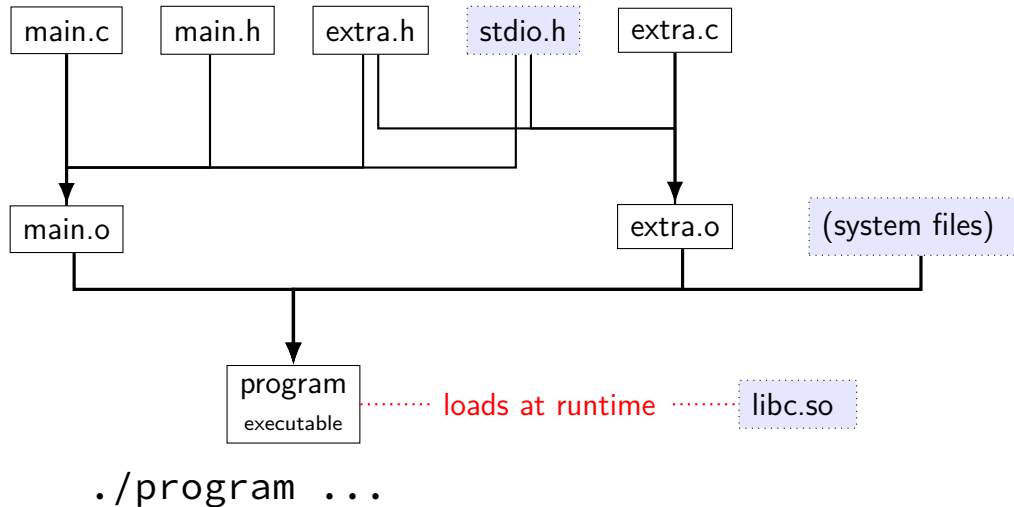
```
clang -S -c main.c  
clang -S -c extra.c
```

files in building C programs [dynamic linking]

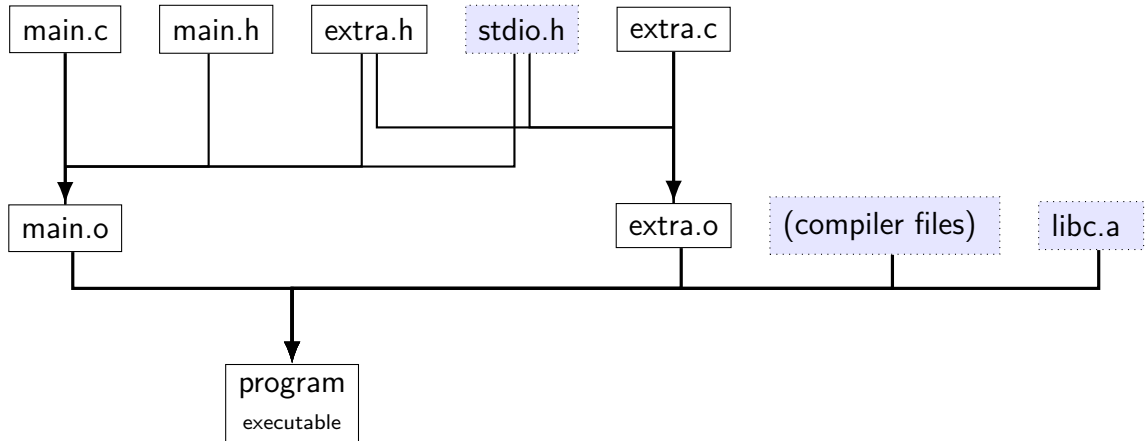


```
clang -o program main.o extra.o
```

files in building C programs [dynamic linking]



files in building C programs [static linking]



file extensions

name		
.c		C source code
.h		C header file
.s	(or .asm)	assembly file
.o	(or .obj)	object file (binary of assembly)
(none)	(or .exe)	executable file
.a	(or .lib)	statically linked library [collection of .o files]
.so	(or .dll or .dylib)	dynamically linked library ['shared object']

static libraries

Unix-like *static* libraries: libfoo.a

internally: archive of .o files with index

create: `ar rcs libfoo.a file1.o file2.o ...`

use: `cc ... -o program -L/path/to/lib ... -lfoo`

no space between `-l` and library name

`cc` could be `clang`, `gcc`, `clang++`, `g++`, etc.

`-L/path/to/lib` not needed if in standard location

shared libraries

Linux *shared* libraries: libfoo.so

create:

compile .o files with `-fPIC` (position independent code)

then: `cc -shared ... -o libfoo.so`

use: `cc ...-o program -L/path/to/lib ...-lfoo`

finding shared libraries (1)

```
$ ls
libexample.so  main.c
$ clang -o main main.c -lexample
/usr/bin/ld: cannot find -lexample
clang: error: linker command failed with exit code 1 (use -v to see invocation)
$ clang -o main main.c -L. -lexample
$ ./main
./main: error while loading shared libraries:
    libexample.so: cannot open shared object file: No such file or directory
```

finding shared libraries (1)

```
$ ls
libexample.so  main.c
$ clang -o main main.c -lexample
/usr/bin/ld: cannot find -lexample
clang: error: linker command failed with exit code 1 (use -v to see invocation)
$ clang -o main main.c -L. -lexample
$ ./main
./main: error while loading shared libraries:
  libexample.so: cannot open shared object file: No such file or directory
```

```
$ LD_LIBRARY_PATH=. ./main

or

$ export LD_LIBRARY_PATH=.
$ ./main

or

$ clang -o main main.c -L. -lexample -Wl,-rpath .
$ ./main
```

finding shared libraries (1)

```
cc ...-o program -L/path/to/lib ...-lfoo
```

on Linux: /path/to/lib only used to create program
program contains `libfoo.so` *without full path*

Linux default: `libfoo.so` expected to be in `/usr/lib`, `/lib`, and other 'standard' locations

possible overrides:

`LD_LIBRARY_PATH` environment variable
paths specified with `-Wl,-rpath=/path/to/lib` when creating executable

exercise (incremental compilation)

program built from main.c + extra.c

main.c, extra.c both include extra.h, stdio.h

```
clang -c main.c                # command 1
clang -c extra.c                # command 2
clang -o program main.o extra.o # command 3
```

What commands need to be rerun if...

Question A: ...main.c changes?

Question B: ...extra.h changes?

make

make — Unix program for “making” things...

...by running commands based on what's changed

what commands? based on *rules* in *makefile*

make rules

```
main.o: main.c main.h extra.h
►      clang -c main.c
```

before colon: target(s) (file(s) generated/updated)

after colon: prerequisite(s)

following lines prefixed by a tab character: command(s) to run

make will run the commands if any prerequisite is newer than the target

make rules

```
main.o: main.c main.h extra.h  
▶      clang -c main.c
```

before colon: **target(s)** (file(s) generated/updated)

after colon: prerequisite(s)

following lines prefixed by a tab character: command(s) to run

make will run the commands if any prerequisite is newer than the target

make rules

```
main.o: main.c main.h extra.h
►      clang -c main.c
```

before colon: target(s) (file(s) generated/updated)

after colon: prerequisite(s)

following lines prefixed by a tab character: command(s) to run

make will run the commands if any prerequisite is newer than the target

make rules

```
main.o: main.c main.h extra.h  
▶      clang -c main.c
```

before colon: target(s) (file(s) generated/updated)

after colon: prerequisite(s)

following lines prefixed by a **tab** character: command(s) to run

make will run the commands if any prerequisite is newer than the target

make rules

```
main.o: main.c main.h extra.h
```

```
▶          clang -c main.c
```

before colon: target(s) (file(s) generated/updated)

after colon: prerequisite(s)

following lines prefixed by a tab character: **command(s) to run**

make will run the commands if any prerequisite is newer than the target

make rules

```
main.o: main.c main.h extra.h  
▶      clang -c main.c
```

before colon: target(s) (file(s) generated/updated)

after colon: prerequisite(s)

following lines prefixed by a tab character: command(s) to run

make will run the commands if any prerequisite is newer than the target

make rules

```
main.o: main.c main.h extra.h
►      clang -c main.c
```

before colon: target(s) (file(s) generated/updated)

after colon: prerequisite(s)

following lines prefixed by a tab character: command(s) to run

make will run the commands if any prerequisite is newer than the target

...after making sure prerequisites up to date

make rule chains

program: main.o extra.o

▶ clang -o program main.o extra.o

extra.o: extra.c extra.h

▶ clang -c extra.c

main.o: main.c main.h extra.h

▶ clang -c main.c

to *make* program, first...

update main.o and extra.o if they aren't

running make

“make *target*”

- look in Makefile in current directory for rules

- check if *target* is up-to-date

- if not, rebuild it (and dependencies, if needed) so it is

“make *target1 target2*”

- check if both *target1* and *target2* are up-to-date

- if not, rebuild it as needed so they are

“make”

- if “*firstTarget*” is the first rule in Makefile,

- same as ‘make *firstTarget*’

exercise: what will run?

W: X Y

► buildW

X: Q

► buildX

Y: X Z

► buildY

W modified 1 minute ago

X modified 3 hours ago

Y does not exist

Z modified 1 hour ago

Q modified 2 hours ago

exercise: “make W” will run what commands?

A. none

B. buildY only C. buildW then buildY

D. buildY then buildW

E. buildX then buildY then buildW

F. buildX then buildW

G. something else

‘phony’ targets (1)

common to have Makefile targets that aren’t files

```
all: program1 program2 libfoo.a
```

“make all” effectively shorthand for “make program1
program2 libfoo.a”

no actual file called “all”

‘phony’ targets (2)

sometimes want targets that don’t actually build file

example: “make clean” to remove generated files
clean:

```
►          rm --force main.o extra.o
```

but what if I create...

clean:

► `rm --force main.o extra.o`

`all: program1 program2 libfoo.a`

Q: if I make a file called “all” and then “make all” what happens?

Q: same with “clean” and “make clean”?

marking phony targets

clean:

► `rm --force main.o extra.o`

`all: program1 program2 libfoo.a`

`.PHONY: all clean`

special .PHONY rule says “ ‘all’ and ‘clean’ not real files”

(not required by POSIX, but in every make version I know)

conventional targets

common convention:

target name	purpose
(default), all	build everything
install	install to standard location
test	run tests
clean	remove generated files

redundancy (1)

program: main.o extra.o

► clang -o program main.o extra.o

extra.o: extra.c extra.h

► clang -o extra.o -c extra.c

main.o: main.c main.h extra.h

► clang -o main.o -c main.c

what if I want to run clang with -Wall?

what if I want to change to gcc?

variables/macros (1)

CC = gcc

CFLAGS = -Wall -pedantic -std=c11 -fsanitize=address

LDFLAGS = -Wall -pedantic -fsanitize=address

LDLIBS = -lm

program: main.o extra.o

► \$(CC) \$(LDFLAGS) -o program main.o extra.o \$(LDLIBS)

extra.o: extra.c extra.h

► \$(CC) \$(CFLAGS) -o extra.o -c extra.c

main.o: main.c main.h extra.h

► \$(CC) \$(CFLAGS) -o main.o -c main.c

variables/macros (2)

```
CC = gcc
CFLAGS = -Wall
LDFLAGS = -Wall
LDLIBS = -lm
```

```
program: main.o extra.o
```

```
▶ $(CC) $(LDFLAGS) -o $@ $^ $(LDLIBS)
```

```
extra.o: extra.c extra.h
```

```
▶ $(CC) $(CFLAGS) -o $@ -c $<
```

```
main.o: main.c main.h extra.h
```

```
▶ $(CC) $(CFLAGS) -o $@ -c $<
```

aside: `$^` works on GNU make (usual on Linux), but not portable.

suffix rules

CC = gcc

CFLAGS = -Wall

LDFLAGS = -Wall

program: main.o extra.o

► \$(CC) \$(LDFLAGS) -o \$@ \$^

.c.o:

► \$(CC) \$(CFLAGS) -o \$@ -c \$<

extra.o: extra.c extra.h

main.o: main.c main.h extra.h

aside: \$^ works on GNU make (usual on Linux), but not portable.

pattern rules

```
CC = gcc
CFLAGS = -Wall
LDFLAGS = -Wall
LDLIBS = -lm
```

```
program: main.o extra.o
```

```
▶ $(CC) $(LDFLAGS) -o $@ $^ $(LDLIBS)
```

```
%.o: %.c
```

```
▶ $(CC) $(CFLAGS) -o $@ -c $<
```

```
extra.o: extra.c extra.h
```

```
main.o: main.c main.h extra.h
```

aside: these rules work on GNU make (usual on Linux), but less portable than suffix rules.

built-in rules

'make' has the 'make .o from .c' rule built-in already, so:

```
CC = gcc
CFLAGS = -Wall
LDFLAGS = -Wall
LDLIBS = -lm
```

```
program: main.o extra.o
```

```
▶      $(CC) $(LDFLAGS) -o $@ $^ $(LDLIBS)
```

```
extra.o: extra.c extra.h
```

```
main.o: main.c main.h extra.h
```

(don't actually need to write supplied rule!)

writing Makefiles?

error-prone to automatically all .h dependencies

-M option to gcc or clang

outputs Make rule

ways of having make run this

Makefile generators

other programs that write Makefiles

other build systems

alternatives to writing Makefiles:

other make-ish build systems

ninja, scon, bazel, maven, xcodebuild, msbuild, ...

tools that generate inputs for make-ish build systems

cmake, autotools, qmake, ...

backup slides