last time

make and Makefiles

- target: prereq (newline)(tab) commands
- suffix/pattern rules
- variables CC/CFLAGS/...

kernel mode versus user mode

- limit operations to OS code
- OS code checks "is this allowed"

system calls

- controlled entry into kernel mode
- starts at OS-specified location

things programs on portal shouldn't do

read other user's files

modify OS's memory

read other user's data in memory

hang the entire system

things programs on portal shouldn't do

read other user's files

modify OS's memory

read other user's data in memory

hang the entire system

memory protection

reading from another program's memory?

Program A	Program B

`0x10000: .word 42`
`// ...`
`// do work`
`// ...`
`movq 0x10000, %rax`
	`// while A is working:`
	`movq $99, %rax`
	`movq %rax, 0x10000`
	`...`

memory protection

reading from another program's memory?

Program A	Program B

`0x10000: .word 42`
`// ...`
`// do work`
`// ...`
`movq 0x10000, %rax`
	`// while A is working:`
	`movq $99, %rax`
	`movq %rax, 0x10000`
	`...`

result: `%rax` (in A) is ...

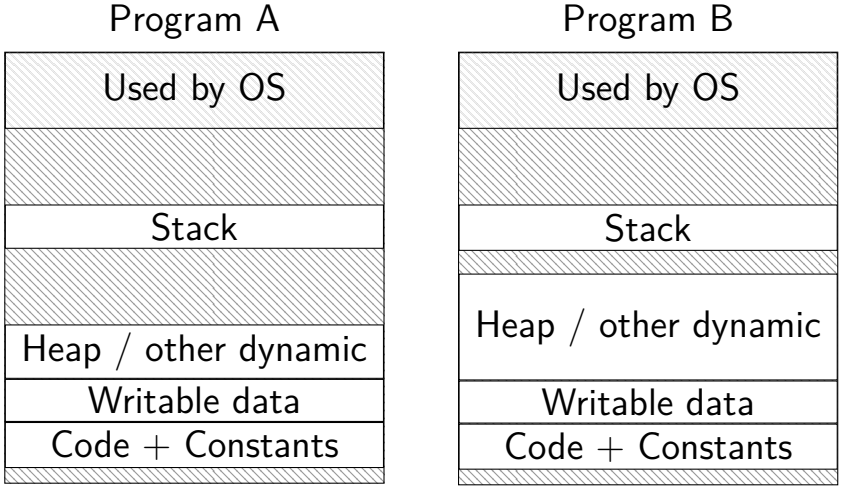
A. 42 B. 99 C. 0x10000

D. 42 or 99 (depending on timing/program layout/etc)

E. 42 or 99 or program might crash (depending on ...)

F. something else

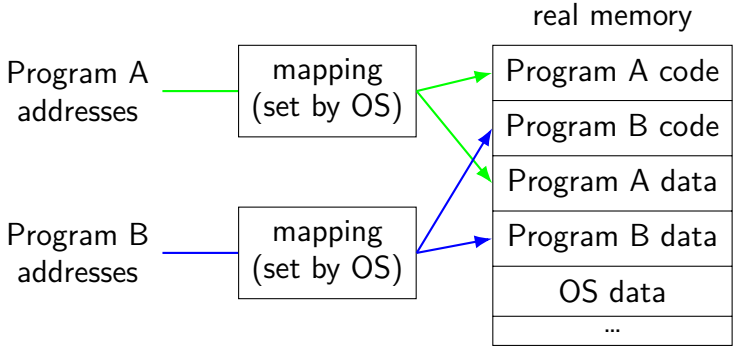
program memory (two programs)



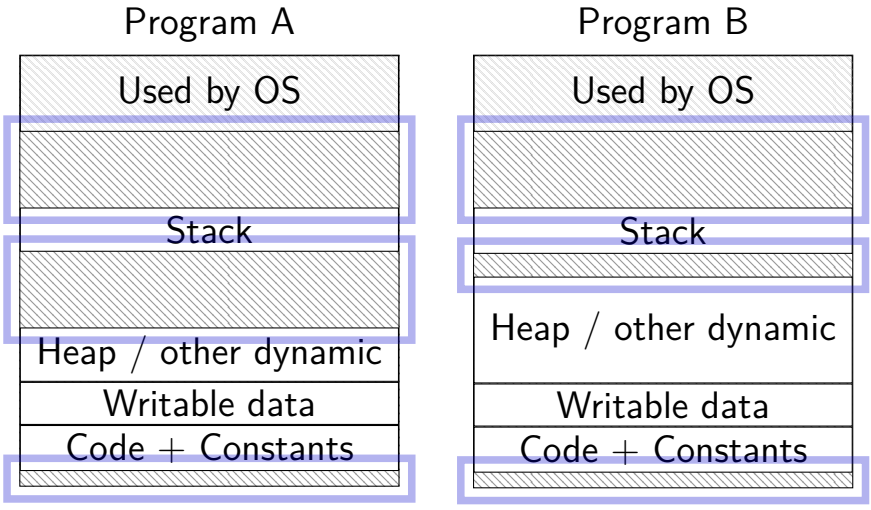
address space

programs have illusion of own memory

called a program's address space



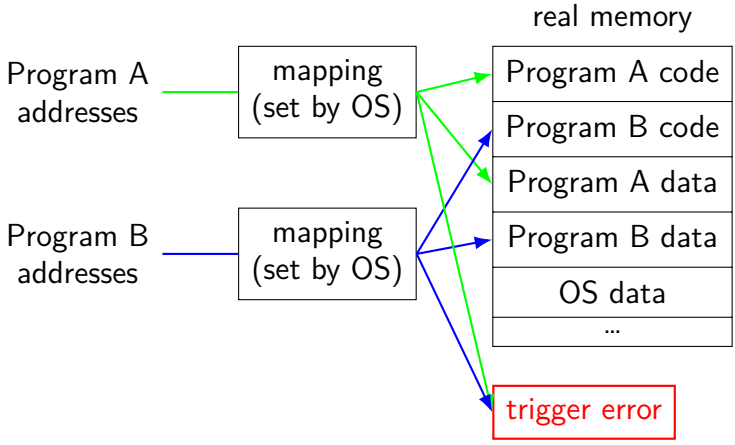
program memory (two programs)



address space

programs have illusion of own memory

called a program's address space



address space mechanisms

topic after exceptions

called virtual memory

mapping called page tables

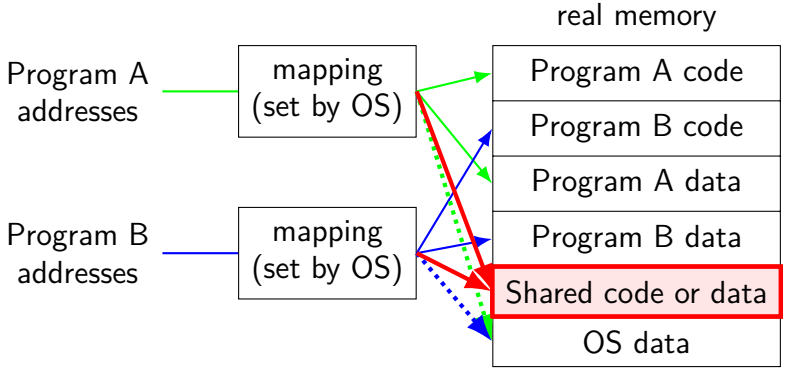
mapping part of what is changed in context switch

shared memory

recall: dynamically linked libraries

would be nice not to duplicate code/data...

we can!



one way to set shared memory on Linux

```
/* regular file, OR: */
int fd = open("/tmp/somefile.dat", O_RDWR);
/* special in-memory file */
int fd = shm_open("/name", O_RDWR);
...
/* make file's data accessible as memory */
void *memory = mmap(NULL, size, PROT_READ | PROT_WRITE,
                    MAP_SHARED, fd, 0);
```

mmap: "map" a file's data into your memory

will discuss a bit more when we talk about virtual memory

part of how Linux loads dynamically linked libraries

things programs on portal shouldn't do

read other user's files

modify OS's memory

read other user's data in memory

hang the entire system

an infinite loop

```
int main(void) {
    while (1) {
        /* waste CPU time */
    }
}
```

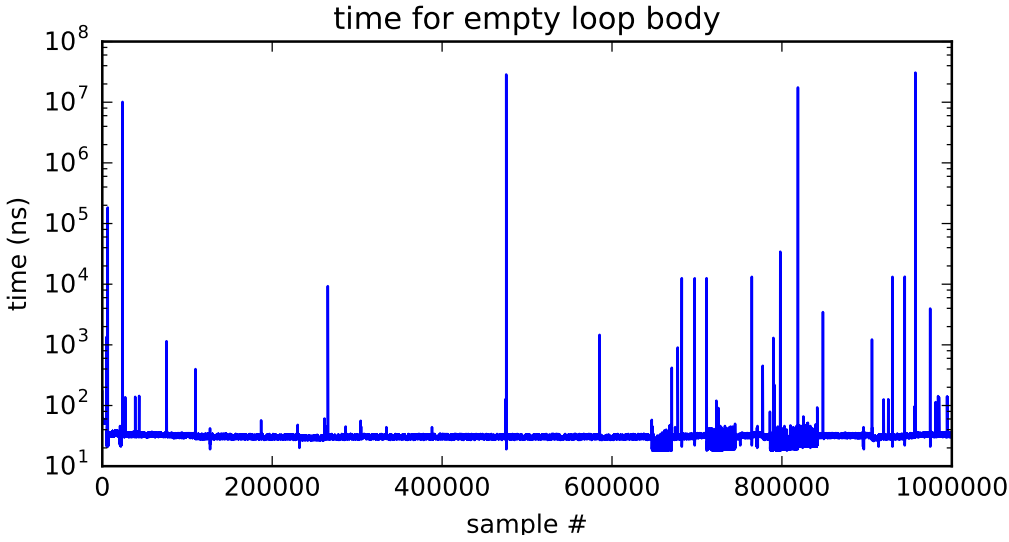
If I run this on a shared department machine, can you still use it?
...if the machine only has one core?

timing nothing

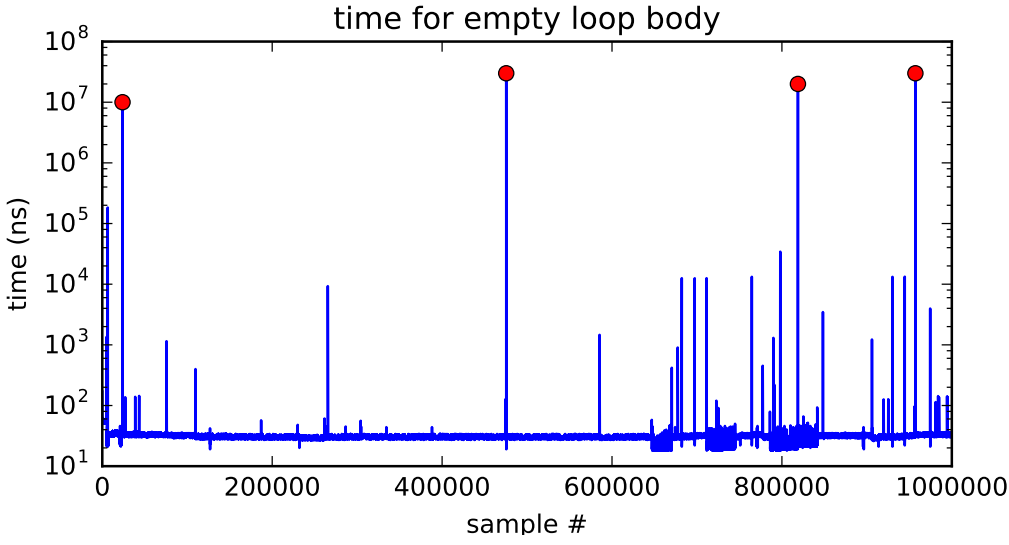
```
long times[NUM_TIMINGS];
int main(void) {
    for (int i = 0; i < N; ++i) {
        long start, end;
        start = get_time();
        /* do nothing */
        end = get_time();
        times[i] = end - start;
    }
    output_timings(times);
}
```

same instructions — same difference each time?

doing nothing on a busy system



doing nothing on a busy system



time multiplexing

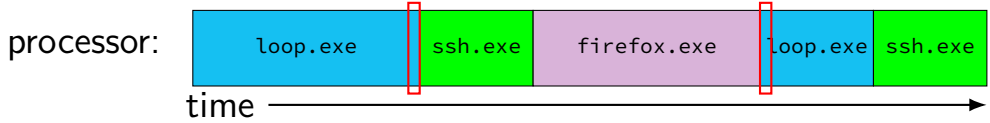
time multiplexing


```
...
call get_time
    // whatever get_time does
movq %rax, %rbp
```

million cycle delay

```
call get_time
    // whatever get_time does
subq %rbp, %rax
...
```

time multiplexing




```
...
call get_time
    // whatever get_time does
movq %rax, %rbp
```

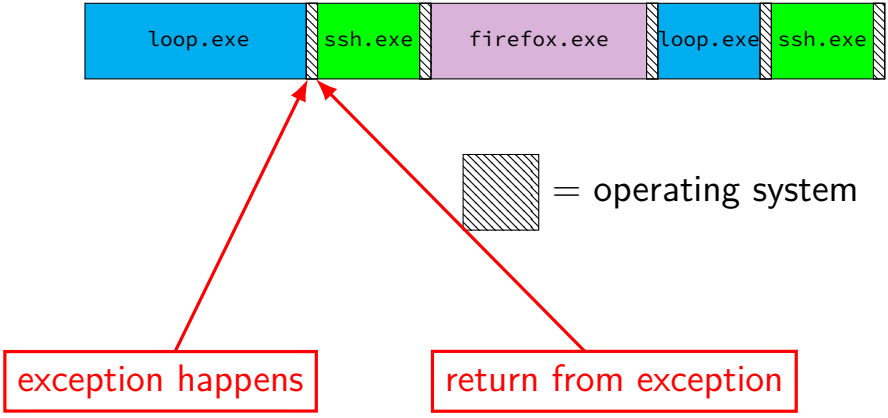
million cycle delay

```
call get_time
    // whatever get_time does
subq %rbp, %rax
...
```

time multiplexing really

 = operating system

time multiplexing really



threads

thread = illusion of own processor

own register values

own program counter value

threads

thread = illusion of own processor

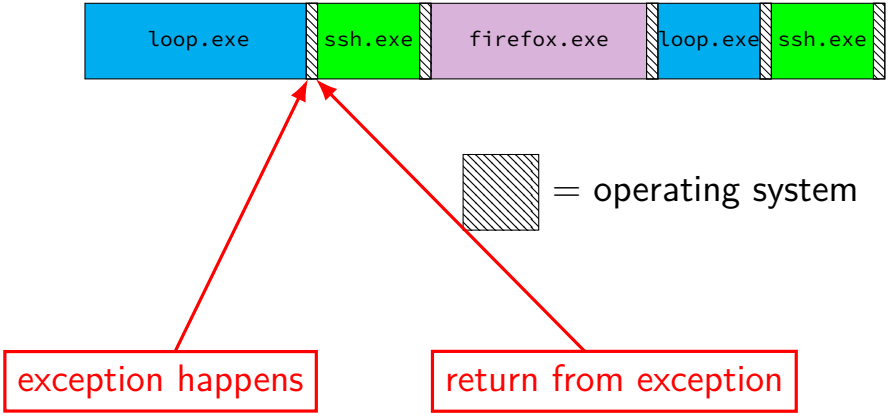
own register values

own program counter value

actual implementation:
many threads sharing one processor

> problem: where are register/program counter values when thread not active on processor?

time multiplexing really



OS and time multiplexing

starts running instead of normal program

- mechanism for this: exceptions (later)

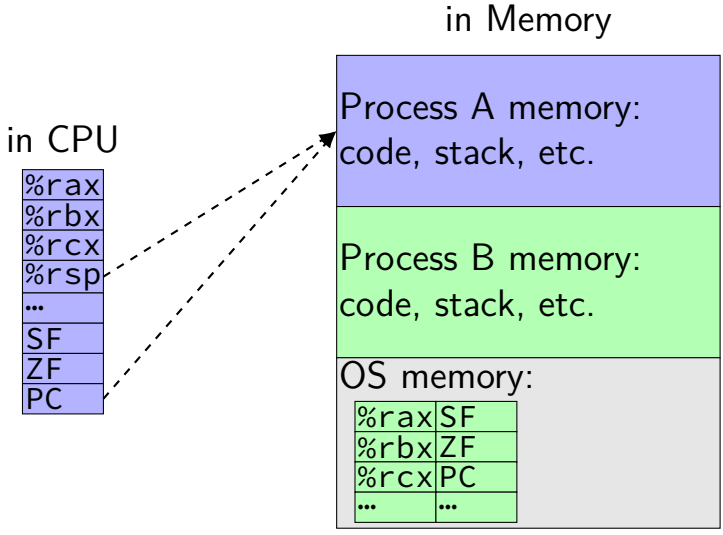
saves old program counter, registers somewhere

sets new registers, jumps to new program counter

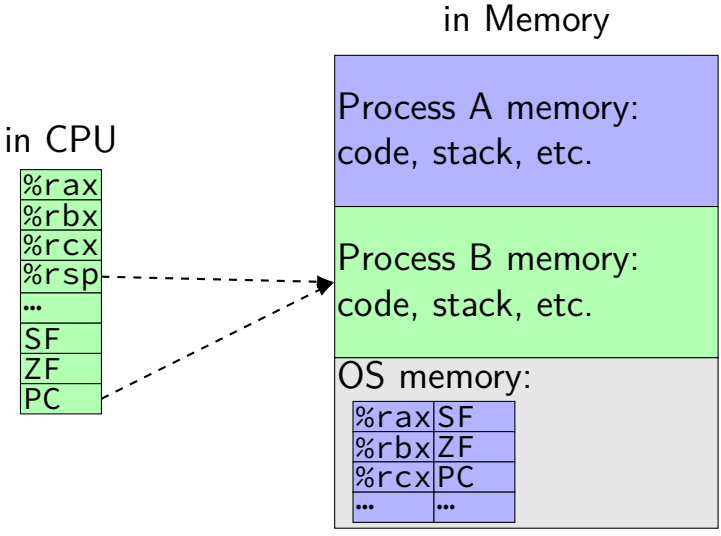
called context switch

- saved information called context

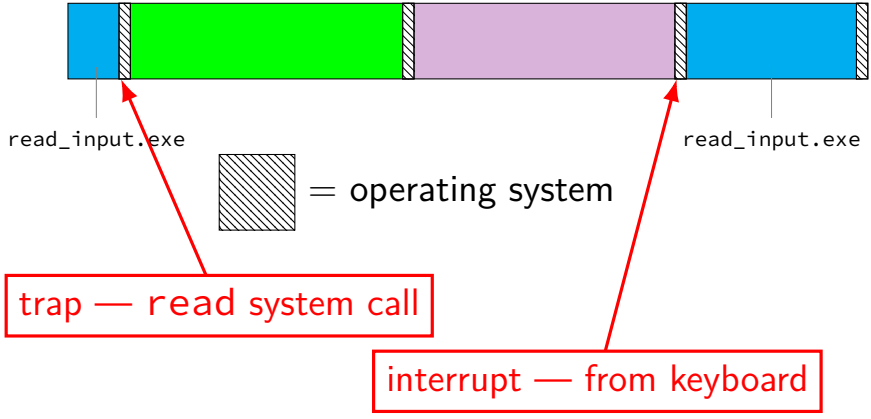
contexts (A running)



contexts (B running)



keyboard input timeline



types of exceptions

externally-triggered
- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, ...
- hardware is broken (e.g. memory parity error)

intentionally triggered exceptions
- system calls — ask OS to do something

errors/events in programs
- memory not in address space ("Segmentation fault")
- privileged instruction
- divide by zero
- invalid instruction

asynchronous
not triggered by running program

synchronous
triggered by current program

terms for exceptions

terms for exceptions aren't standardized

our readings use one set of terms

interrupts = externally-triggered
faults = error/event in program
trap = intentionally triggered

all these terms appear differently elsewhere

exception implementation

detect condition (program error or external event)

save current value of PC somewhere

jump to exception handler (part of OS)

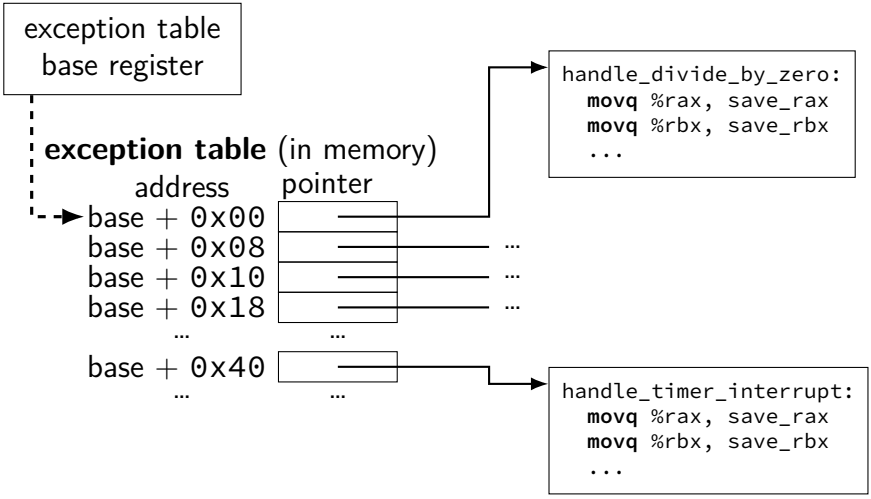
- jump done without program instruction to do so

exception implementation: notes

I describe a simplified version

real x86/x86-64 is a bit more complicated
(mostly for historical reasons)

locating exception handlers



running the exception handler

hardware saves the old program counter (and maybe more)

identifies location of exception handler via table

then jumps to that location

OS code can save anything else it wants to , etc.

which of these require exceptions? context switches?

A. program calls a function in the standard library

B. program writes a file to disk

C. program A goes to sleep, letting program B run

D. program exits

E. program returns from one function to another function

F. program pops a value from the stack

The Process

process = thread(s) + address space

illusion of dedicated machine:

- thread = illusion of own CPU
- address space = illusion of own memory

signals

Unix-like operating system feature

like exceptions for processes:

can be triggered by external process

- kill command/system call

can be triggered by special events

- pressing control-C
- other events that would normal terminate program
  - 'segmentation fault'
  - illegal instruction
  - divide by zero

can invoke signal handler (like exception handler)

exceptions v signals

(hardware) exceptions	signals
handler runs in kernel mode	handler runs in user mode
hardware decides when	OS decides when
hardware needs to save PC	OS needs to save PC + registers
processor next instruction changes	thread next instruction changes

exceptions v signals

(hardware) exceptions	signals
handler runs in kernel mode	handler runs in user mode
hardware decides when	OS decides when
hardware needs to save PC	OS needs to save PC + registers
processor next instruction changes	thread next instruction changes

...but OS needs to run to trigger handler
most likely "forwarding" hardware exception

exceptions v signals

(hardware) exceptions	signals
handler runs in kernel mode	handler runs in user mode
hardware decides when	OS decides when
hardware needs to save PC	OS needs to save PC + registers
processor next instruction changes	thread next instruction changes

signal handler follows normal calling convention
not special assembly like typical exception handler

exceptions v signals

(hardware) exceptions	signals
handler runs in kernel mode	handler runs in user mode
hardware decides when	OS decides when
hardware needs to save PC	OS needs to save PC + registers
processor next instruction changes	thread next instruction changes

signal handler runs in same thread ('virtual processor') as process was using before

not running at 'same time' as the code it interrupts

base program

```
int main() {
    char buf[1024];
    while (fgets(buf, sizeof buf, stdin)) {
        printf("read %s", buf);
    }
}
```

base program

```
int main() {
    char buf[1024];
    while (fgets(buf, sizeof buf, stdin)) {
        printf("read %s", buf);
    }
}
```

---

some input
read some input
more input
read more input
(control-C pressed)
(program terminates immediately)

base program

```
int main() {
    char buf[1024];
    while (fgets(buf, sizeof buf, stdin)) {
        printf("read %s", buf);
    }
}
```

---

some input
read some input
more input
read more input
(control-C pressed)
(program terminates immediately)

new program

```
int main() {
    ... // added stuff shown later
    char buf[1024];
    while (fgets(buf, sizeof buf, stdin)) {
        printf("read %s", buf);
    }
}
```

---

some input
read some input
more input
read more input
(control-C pressed)
Control-C pressed?!
another input **read another input**

new program

```
int main() {
    ... // added stuff shown later
    char buf[1024];
    while (fgets(buf, sizeof buf, stdin)) {
        printf("read %s", buf);
    }
}
```

---

some input
read some input
more input
read more input
(control-C pressed)
Control-C pressed?!
another input **read another input**

new program

```
int main() {
    ... // added stuff shown later
    char buf[1024];
    while (fgets(buf, sizeof buf, stdin)) {
        printf("read %s", buf);
    }
}
```

---

some input
read some input
more input
read more input
(control-C pressed)
Control-C pressed?!
another input **read another input**

example signal program

```
void handle_sigint(int signum) {
    /* signum == SIGINT */
    write(1, "Control-C pressed?!\n",
        sizeof("Control-C pressed?!\n"));
}

int main(void) {
    struct sigaction act;
    act.sa_handler = &handle_sigint;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGINT, &act, NULL);

    char buf[1024];
    while (fgets(buf, sizeof buf, stdin)) {
        printf("read %s", buf);
    }
}
```

example signal program

```
void handle_sigint(int signum) {
    /* signum == SIGINT */
    write(1, "Control-C pressed?!\n",
        sizeof("Control-C pressed?!\n"));
}

int main(void) {
    struct sigaction act;
    act.sa_handler = &handle_sigint;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGINT, &act, NULL);

    char buf[1024];
    while (fgets(buf, sizeof buf, stdin)) {
        printf("read %s", buf);
    }
}
```

example signal program

```
void handle_sigint(int signum) {
    /* signum == SIGINT */
    write(1, "Control-C pressed?!\n",
        sizeof("Control-C pressed?!\n"));
}

int main(void) {
    struct sigaction act;
    act.sa_handler = &handle_sigint;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGINT, &act, NULL);

    char buf[1024];
    while (fgets(buf, sizeof buf, stdin)) {
        printf("read %s", buf);
    }
}
```

SIGxxxx

signals types identified by number...

constants declared in `<signal.h>`

constant	likely use
SIGBUS	"bus error"; certain types of invalid memory accesses
SIGSEGV	"segmentation fault"; other types of invalid memory accesses
SIGINT	what control-C usually does
SIGFPE	"floating point exception"; includes integer divide-by-zero
SIGHUP, SIGPIPE	reading from/writing to disconnected terminal/socket
SIGUSR1, SIGUSR2	use for whatever you (app developer) wants
SIGKILL	terminates process (cannot be handled by process!)
SIGSTOP	suspends process (cannot be handled by process!)
...	...

SIGxxxx

signals types identified by number...

constants declared in `<signal.h>`

constant	likely use
SIGBUS	"bus error"; certain types of invalid memory accesses
SIGSEGV	"segmentation fault"; other types of invalid memory accesses
SIGINT	what control-C usually does
SIGFPE	"floating point exception"; includes integer divide-by-zero
SIGHUP, SIGPIPE	reading from/writing to disconnected terminal/socket
SIGUSR1, SIGUSR2	use for whatever you (app developer) wants
SIGKILL	terminates process (cannot be handled by process!)
SIGSTOP	suspends process (cannot be handled by process!)
...	...

handling Segmentation Fault

```
...
void handle_sigsegv(int num) {
    puts("got SIGSEGV");
}

int main(void) {
    struct sigaction act;
    act.sa_handler = handle_sigsegv;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGSEGV, &act, NULL);

    asm("movq %rax, 0x12345678");
}
```

handling Segmentation Fault

```
...
void handle_sigsegv(int num) {
    puts("got SIGSEGV");
}

int main(void) {
    struct sigaction act;
    act.sa_handler = handle_sigsegv;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGSEGV, &act, NULL);

    asm("movq %rax, 0x12345678");
}
```

```
got SIGSEGV
got SIGSEGV
got SIGSEGV
got SIGSEGV
```

signal API

`sigaction` — register handler for signal

`kill` — send signal to process

`pause` — put process to sleep until signal received

`sigprocmask` — temporarily block some signals from being received

... and much more

output of this?

pid 1000

```
void handle_sigusr1(int num) {
    write(1, "X", 1);
    kill(2000, SIGUSR1);
    _exit(0);
}

int main() {
    struct sigaction act;
    act.sa_handler = &handler_usr1;
    sigaction(SIGUSR1, &act);
    kill(1000, SIGUSR1);
}
```

pid 2000

```
void handle_sigusr1(int num) {
    write(1, "Y", 1);
    _exit(0);
}

int main() {
    struct sigaction act;
    act.sa_handler = &handler_usr1;
    sigaction(SIGUSR1, &act);
}
```

If these run at same time, expected output?

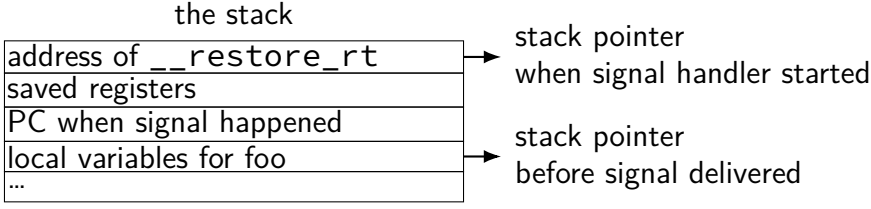
A. XY	B. X	C. Y
D. YX	E. X or XY, depending on timing	F. crash
G. (nothing)	H. something else	

x86-64 Linux signal delivery (1)

suppose: signal happens while `foo()` is running

OS saves registers to user stack

OS modifies user registers, PC to call signal handler



x86-64 Linux signal delivery (2)

```
handle_sigint:
    ...
    ret
...
__restore_rt:
    // 15 = "sigreturn" system call
    movq $15, %rax
    syscall
```

`__restore_rt` is return address for signal handler

sigreturn syscall restores pre-signal state

- needed to handle caller-saved registers
- also might unblock signals (like un-disabling interrupts)

signal handler unsafety (0)

```
void foo() {
    /* SIGINT might happen while foo() is running */
    char *p = malloc(1024);
    ...
}

/* signal handler for SIGINT
   (registered elsewhere with sigaction() */
void handle_sigint() {
    printf("You pressed control-C.\n");
}
```

signal handler unsafety (1)

```
void *malloc(size_t size) {
    ...
    to_return = next_to_return;
    /* SIGNAL HAPPENS HERE */
    next_to_return += size;
    return to_return;
}

void foo() {
    /* This malloc() call interrupted */
    char *p = malloc(1024);
    p[0] = 'x';
}

void handle_sigint() {
    // printf might use malloc()
    printf("You pressed control-C.\n");
}
```

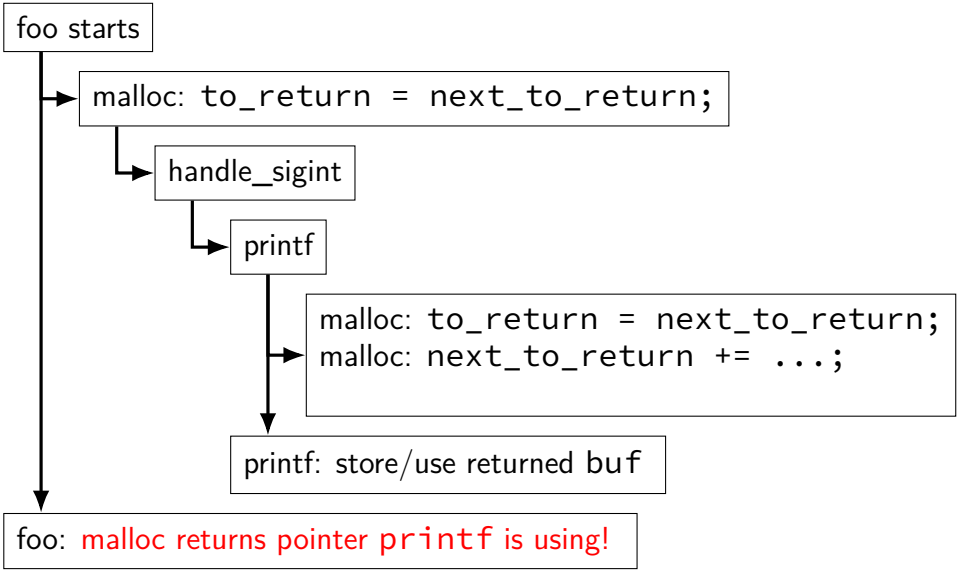
signal handler unsafety (1)

```
void *malloc(size_t size) {
    ...
    to_return = next_to_return;
    /* SIGNAL HAPPENS HERE */
    next_to_return += size;
    return to_return;
}

void foo() {
    /* This malloc() call interrupted */
    char *p = malloc(1024);
    p[0] = 'x';
}

void handle_sigint() {
    // printf might use malloc()
    printf("You pressed control-C.\n");
}
```

signal handler unsafety (2)

```
void handle_sigint() {
    printf("You pressed control-C.\n");
}

int printf(...) {
    static char *buf;
    ...
    buf = malloc()
    ...
}
```

signal handler unsafety: timeline



signal handler unsafety (3)

```
foo() {
  char *p = malloc(1024)... {
    to_return = next_to_return;
    handle_sigint() { /* signal delivered here */
      printf("You pressed control-C.\n") {
        buf = malloc(...) {
          to_return = next_to_return;
          next_to_return += size;
          return to_return;
        }
        ...
      }
    }
    next_to_return += size;
    return to_return;
  }
  /* now p points to buf used by printf! */
}
```

signal handler unsafety (3)

```
foo() {
  char *p = malloc(1024)... {
    to_return = next_to_return;
    handle_sigint() { /* signal delivered here */
      printf("You pressed control-C.\n") {
        buf = malloc(...) {
          to_return = next_to_return;
          next_to_return += size;
          return to_return;
        }
        ...
      }
    }
    next_to_return += size;
    return to_return;
  }
  /* now p points to buf used by printf! */
}
```

signal handler safety

POSIX (standard that Linux follows) defines "async-signal-safe" functions

these must work correctly no matter what they interrupt

...and no matter how they are interrupted

includes: `write`, `_exit`

does not include: `printf`, `malloc`, `exit`

blocking signals

avoid having signal handlers anywhere:

can instead block signals

`sigprocmask` system call

signal will become "pending" instead

OS will not deliver unless unblocked

analagous to disabling interrupts

alternatives to signal handlers

first, block a signal

then use system calls to inspect pending signals

example: `sigwait`

or unblock signals only when waiting for I/O

example: `pselect` system call

synchronous signal handling

```
int main(void) {
    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    sigprocmask(SIG_BLOCK, SIGINT);

    printf("Waiting for SIGINT (control-C)\n");
    if (sigwait(&set, NULL) == 0) {
        printf("Got SIGINT\n");
    }
}
```

setjmp/longjmp

```
jmp_buf env;

main() {
  if (setjmp(env) == 0) { // like try {
    ...
    read_file()
    ...
  } else { // like catch
    printf("some error happened\n");
  }
}

read_file() {
  ...
  if (open failed) {
      longjmp(env, 1) // like throw
  }
  ...
}
```

implementing setjmp/longjmp

setjmp:

- copy all registers to `jmp_buf`
- ... including stack pointer

longjmp

- copy registers from `jmp_buf`
- ... but change `%rax` (return value)

setjmp psuedocode

setjmp: looks like first half of context switch

```
setjmp:
  movq %rcx, env->rcx
  movq %rdx, env->rdx
  movq %rsp + 8, env->rsp // +8: skip return value
  ...
  save_condition_codes env->ccs
  movq 0(%rsp), env->pc
  movq $0, %rax // always return 0
  ret
```

longjmp psuedocode

longjmp: looks like second half of context switch

```
longjmp:
  movq %rdi, %rax // return a different value
  movq env->rcx, %rcx
  movq env->rdx, %rdx
  ...
  restore_condition_codes env->ccs
  movq env->rsp, %rsp
  jmp env->pc
```

setjmp weirdness — local variables

Undefined behavior:

```
int x = 0;
if (setjmp(env) == 0) {
    ...
    x += 1;
    longjmp(env, 1);
} else {
    printf("%d\n", x);
}
```

setjmp weirdness — fix

Defined behavior:

```
volatile int x = 0;
if (setjmp(env) == 0) {
    ...
    x += 1;
    longjmp(env, 1);
} else {
    printf("%d\n", x);
}
```

on implementing try/catch

could do something like setjmp()/longjmp()

but setjmp is slow

setjmp exercise

```
jmp_buf env; int counter = 0;
void bar() {
    putchar('Z');
    ++counter;
    if (counter < 2) {
        longjmp(env, 1);
    }
}
int main() {
    while (setjmp(env) == 1) {
        putchar('X');
    }
    putchar('Y');
    bar();
}
```

Expected output?

A. YZ B. XYZ C. YZYZ D. XYZXYZ

E. XYZYZ F. YZXYZ G. something else H. varies/might crash

on implementing try/catch

could do something like setjmp()/longjmp()

but setjmp is slow

low-overhead try/catch (1)

```
main() {
  printf("about to read file\n");
  try {
    read_file();
  } catch(...) {
    printf("some error happened\n");
  }
}
read_file() {
  ...
  if (open failed) {
      throw IOException();
  }
  ...
}
```

low-overhead try/catch (2)

```
main:
  ...
  call printf
start_try:
  call read_file
end_try:
  ret
```

```
main_catch:
  movq $str, %rdi
  call printf
  jmp end_try
```

```
read_file:
  pushq %r12
  ...
  call do_throw
  ...
end_read:
  popq %r12
  ret
```

lookup table

program counter range	**action**	**recurse?**
`start_try` to `end_try`	`jmp main_catch`	no
`read_file` to `end_read`	`popq %r12, ret`	yes
anything else	error	—

low-overhead try/catch (2)

```
main:
  ...
  call printf
start_try:
  call read_file
end_try:
  ret
```

```
main_catch:
  movq $str, %rdi
  call printf
  jmp end_try
```

```
read_file:
  pushq %r12
  ...
  call do_throw
  ...
end_read:
  popq %r12
  ret
```

lookup table

program counter range	action	recurse?
`start_try` to `end_try`	`jmp main_catch`	no
`read_file` to `end_read`	`popq %r12, ret`	yes
anything else	error	—

low-overhead try/catch (2)

```
main:
  ...
  call printf
start_try:
  call read_file
end_try:
  ret
```

```
main_catch:
  movq $str, %rdi
  call printf
  jmp end_try
```

```
read_file:
  pushq %r12
  ...
  call do_throw
  ...
end_read:
  popq %r12
  ret
```

lookup table

program counter range	action	recurse?
`start_try` to `end_try`	`jmp main_catch`	no
`read_file` to `end_read`	`popq %r12, ret`	yes
anything else	error	—

low-overhead try/catch (2)

```
main:
  ...
  call printf
start_try:
  call read_file
end_try:
  ret
```

```
main_catch:
  movq $str, %rdi
  call printf
  jmp end_try
```

```
read_file:
  pushq %r12
  ...
  call do_throw
  ...
```

not actual x86 code to run
track a "virtual PC" while looking for catch block

lookup table

program counter range	action	recurse?
`start_try` to `end_try`	`jmp main_catch`	no
`read_file` to `end_read`	`popq %r12, ret`	yes
anything else	error	—

lookup table tradeoffs

no overhead if throw not used

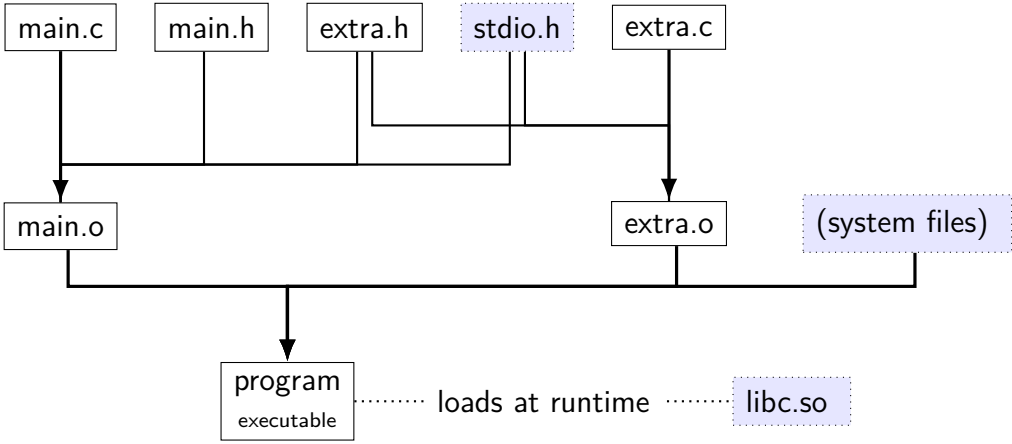
handles local variables on registers/stack, but...

larger executables (probably)

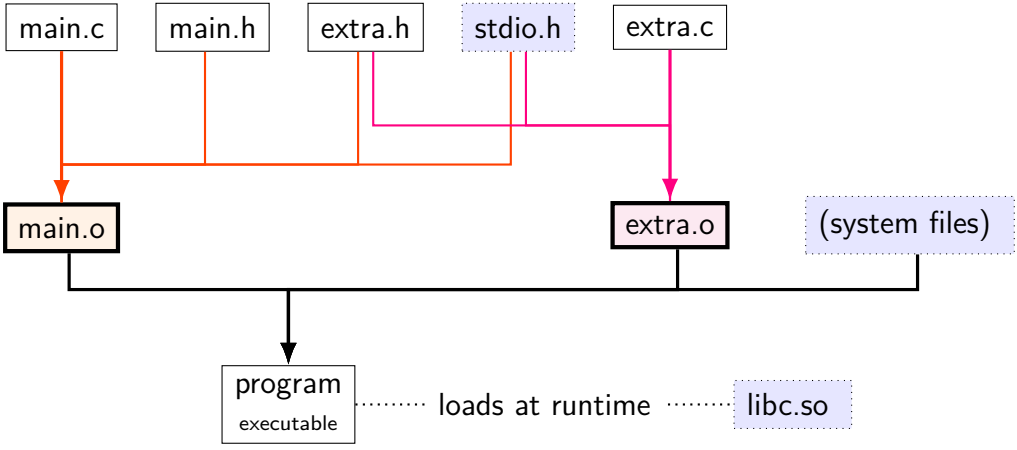
extra complexity for compiler

backup slides

files in building C programs [dynamic linking]

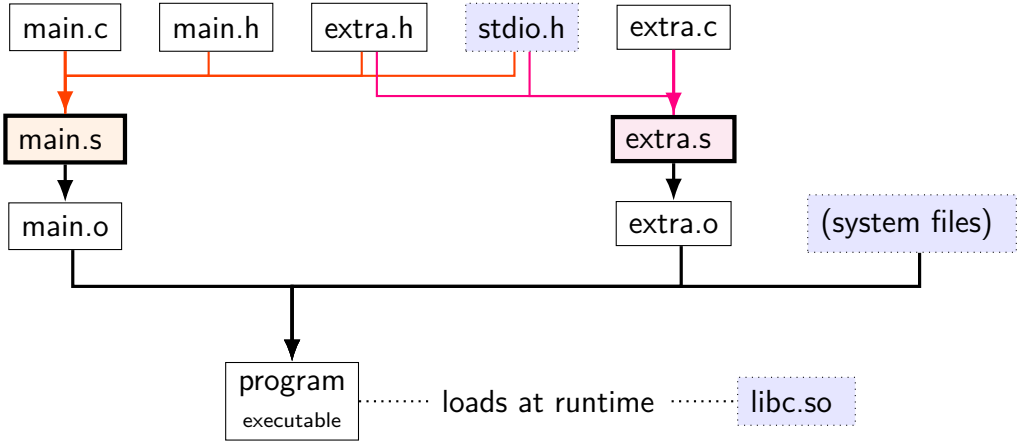


files in building C programs [dynamic linking]



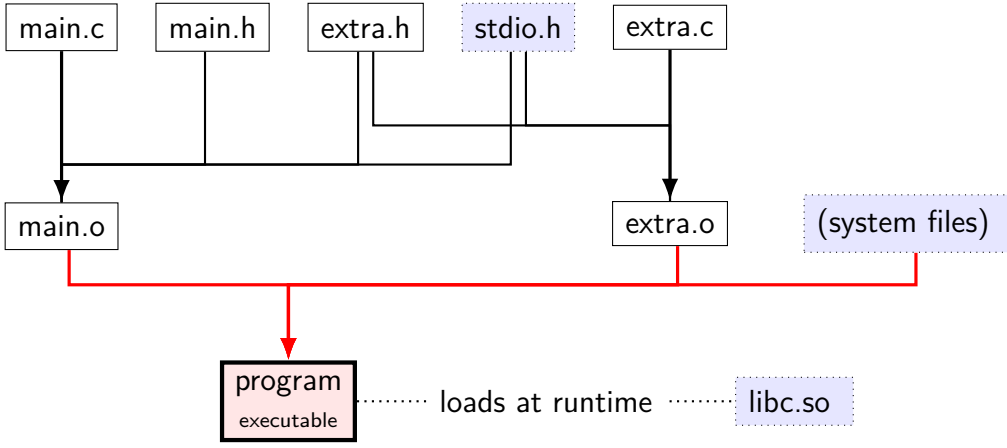

```
clang -c main.c
clang -c extra.c
```

files in building C programs [dynamic linking]



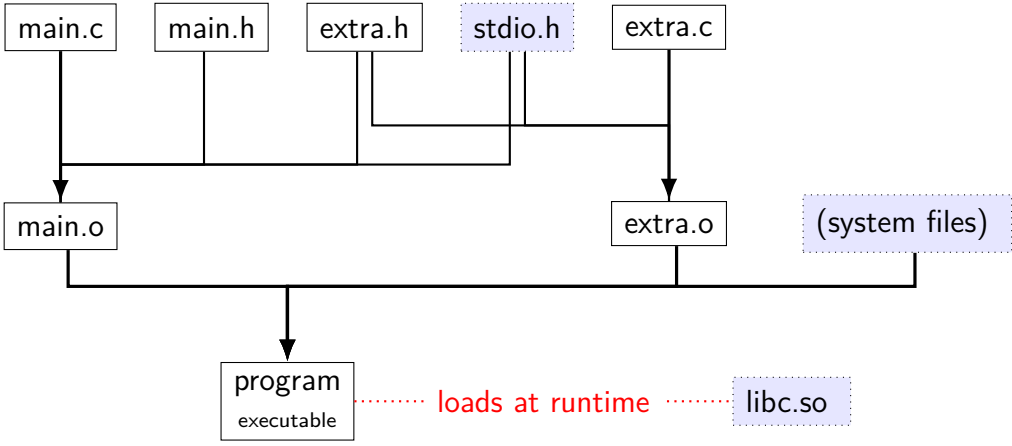

```
clang -S -c main.c
clang -S -c extra.c
```

files in building C programs [dynamic linking]



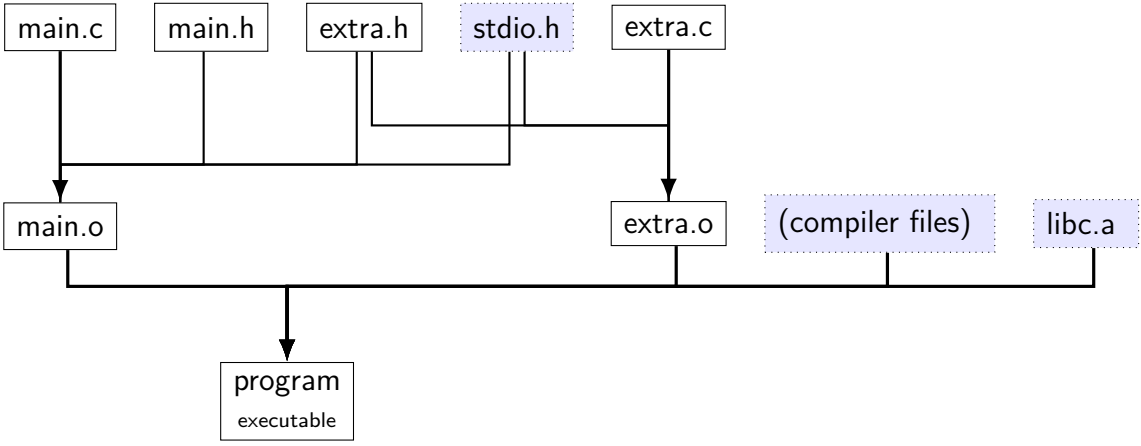

```
clang -o program main.o extra.o
```

files in building C programs [dynamic linking]




```
./program ...
```

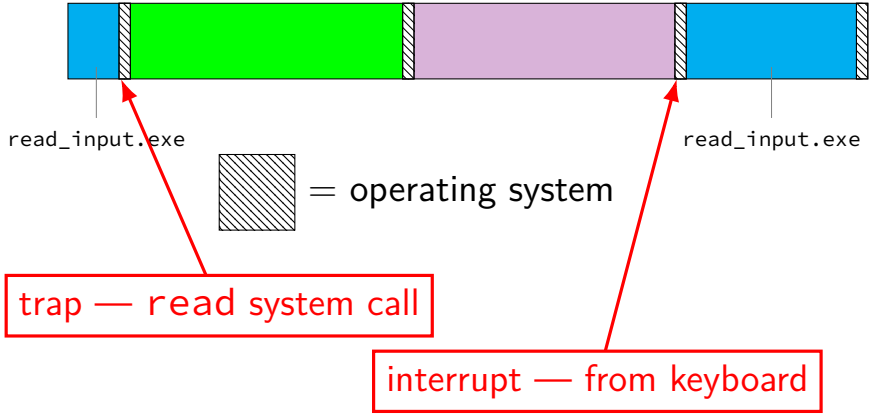
files in building C programs [static linking]



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`)	dynamically linked library ['shared object']

keyboard input timeline



exceptions in exceptions

```
handle_timer_interrupt:
  save_old_pc save_pc
  movq %r15, save_r15
  /* key press here */

  movq %r14, save_r14
  ...
```

exceptions in exceptions

```
handle_timer_interrupt:
  save_old_pc save_pc
  movq %r15, save_r15
  /* key press here */
  movq %r14, save_r14
  ...
```

```
handle_keyboard_interrupt:
  save_old_pc save_pc
  movq %r15, save_r15
  movq %r14, save_r14
  movq %r13, save_r13
  ...
```

exceptions in exceptions

```
handle_timer_interrupt:
  save_old_pc save_pc
  movq %r15, save_r15
  /* key press here */
  movq %r14, save_r14
  ...
```

```
handle_keyboard_interrupt:
  save_old_pc save_pc
  movq %r15, save_r15
  movq %r14, save_r14
  movq %r13, save_r13
  ...
```

oops, overwrote saved values?

interrupt disabling

CPU supports disabling (most) interrupts

interrupts will wait until it is reenabled

CPU has extra state:

- are interrupts enabled?
- is keyboard interrupt pending?
- is timer interrupt pending?

exceptions in exceptions

```
handle_timer_interrupt:
  /* interrupts automatically disabled here */
  movq %rsp, save_rsp
  save_old_pc save_pc
  /* key press here */
  jmpIfFromKernelMode skip_exception_stack
  movq current_exception_stack, %rsp
skip_set_kernel_stack:
  pushq save_rsp
  pushq save_pc
  enable_intterupts2
  pushq %r15
  ...
  /* interrupt happens here! */
  ...
```

exceptions in exceptions

```
handle_timer_interrupt:
  /* interrupts automatically disabled here */
  movq %rsp, save_rsp
  save_old_pc save_pc
  /* key press here */
  jmpIfFromKernelMode skip_exception_stack
  movq current_exception_stack, %rsp
skip_set_kernel_stack:
  pushq save_rsp
  pushq save_pc
  enable_intterupts2
  pushq %r15
  ...
  /* interrupt happens here! */
  ...
```

exceptions in exceptions

```
handle_timer_interrupt:
  /* interrupts automatically disabled here */
  movq %rsp, save_rsp
  save_old_pc save_pc
  /* key press here */
  jmpIfFromKernelMode skip_exception_stack
  movq current_exception_stack, %rsp
skip_set_kernel_stack:
  pushq save_rsp
  pushq save_pc
  enable_intterupts2
  pushq %r15
  ...
  /* interrupt happens here! */
  ...
```

```
handle_keyboard_interrupt:
  movq %rsp, save_rsp
```

disabling interrupts

automatically disabled when exception handler starts

also can be done with privileged instruction:

```
change_keyboard_parameters:
  disable_interrupts
  ...
  /* change things used by
     handle_keyboard_interrupt here */
  ...
  enable_interrupts
```