

last time

overall course themes

logistics

static versus dynamic linking

- dynamic (.so, .dll, .dylib): libraries loaded at runtime

- static (.a, .lib): library code copied to executable file

steps for building applications + libraries

on lab due times

when submission is allowed, moved to 8:59am next day

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

“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

program: main.o extra.o

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

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

program: main.o extra.o

► \$(CC) \$(LDFLAGS) -o \$@ \$^

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

program: main.o extra.o

▶ \$(CC) \$(LDFLAGS) -o \$@ \$^

%.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
```

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

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

```
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, ...

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

privileged instructions

can't let **any program** run some instructions

example: talk to I/O device

allows machines to be shared between users (e.g. lab servers)

processor has two modes:

- kernel mode — privileged instructions work

- user mode — privileged instructions cause exception instead

only *trusted* OS code runs in kernel mode

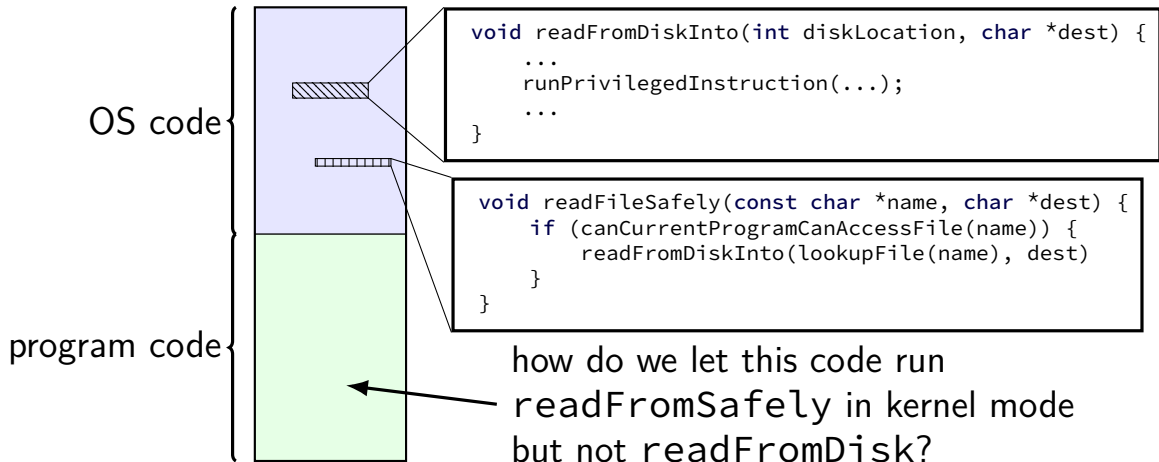
kernel mode

extra one-bit register: “are we in kernel mode”

processor switches to kernel mode to run OS

OS switches processor back to user mode when running normal code

calling the OS?



controlled entry to kernel mode (1)

special instruction: “system call”

runs OS code in kernel mode at location specified earlier

OS sets up at boot

location can't be changed without privileged instruction

controlled entry to kernel mode (2)

OS needs to make specified location:

figure out what operation the program wants

calling convention, similar to function arguments + return value

be “safe” — not allow the program to do ‘bad’ things

example: checks whether current program is allowed to read file before reading it

requires exceptional care — program can try weird things

Linux x86-64 system calls

special instruction: `syscall`

runs OS specified code in kernel mode

Linux syscall calling convention

before `syscall`:

`%rax` — system call number

`%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9` — args

after `syscall`:

`%rax` — return value

on error: `%rax` contains -1 times “error number”

almost the same as normal function calls

Linux x86-64 hello world

```
.globl _start
.data
hello_str: .asciz "Hello, World!\n"
.text
_start:
    movq $1, %rax # 1 = "write"
    movq $1, %rdi # file descriptor 1 = stdout
    movq $hello_str, %rsi
    movq $15, %rdx # 15 = strlen("Hello, World!\n")
    syscall

    movq $60, %rax # 60 = exit
    movq $0, %rdi
    syscall
```

approx. system call handler

```
sys_call_table:  
    .quad handle_read_syscall  
    .quad handle_write_syscall  
    // ...  
  
handle_syscall:  
    ... // save old PC, etc.  
    pushq %rcx // save registers  
    pushq %rdi  
    ...  
    call *sys_call_table(,%rax,8)  
    ...  
    popq %rdi  
    popq %rcx  
    return_from_exception
```

Linux system call examples

`mmap`, `brk` — allocate memory

`fork` — create new process

`execve` — run a program in the current process

`_exit` — terminate a process

`open`, `read`, `write` — access files

`socket`, `accept`, `getpeername` — socket-related

system call wrappers

library functions to not write assembly:

open:

```
movq $2, %rax // 2 = sys_open
// 2 arguments happen to use same registers
syscall
// return value in %eax
cmp $0, %rax
jl has_error
ret
```

has_error:

```
neg %rax
movq %rax, errno
movq $-1, %rax
ret
```

system call wrappers

library functions to not write assembly:

open:

```
movq $2, %rax // 2 = sys_open
// 2 arguments happen to use same registers
syscall
// return value in %eax
cmp $0, %rax
jl has_error
ret
```

has_error:

```
neg %rax
movq %rax, errno
movq $-1, %rax
ret
```

system call wrapper: usage

```
/* unistd.h contains definitions of:  
    O_RDONLY (integer constant), open() */  
#include <unistd.h>  
int main(void) {  
    int file_descriptor;  
    file_descriptor = open("input.txt", O_RDONLY);  
    if (file_descriptor < 0) {  
        printf("error: %s\n", strerror(errno));  
        exit(1);  
    }  
    ...  
    result = read(file_descriptor, ...);  
    ...  
}
```

system call wrapper: usage

```
/* unistd.h contains definitions of:  
    O_RDONLY (integer constant), open() */  
#include <unistd.h>  
int main(void) {  
    int file_descriptor;  
    file_descriptor = open("input.txt", O_RDONLY);  
    if (file_descriptor < 0) {  
        printf("error: %s\n", strerror(errno));  
        exit(1);  
    }  
    ...  
    result = read(file_descriptor, ...);  
    ...  
}
```

strace hello_world (1)

strace — Linux tool to trace system calls

run on assembly program we saw earlier:

```
$ strace -o trace.txt ./hello_world
```

```
$ cat trace.txt
```

```
execve("./hello_world", ["./hello_world"],  
        0x7ffeedafdf0a0 /* 28 vars */) = 0  
write(1, "Hello, World!\n\0", 15)      = 15  
exit(0)                                = ?  
+++ exited with 0 +++
```

strace hello_world (2)

```
#include <stdio.h>
int main() { puts("Hello, World!"); }
```

when statically linked:

```
execve("./hello_world", [ "./hello_world" ], 0x7ffeb4127f70 /* 28 vars */)
    = 0
brk(NULL)
    = 0x22f8000
brk(0x22f91c0)
    = 0x22f91c0
arch_prctl(ARCH_SET_FS, 0x22f8880)
    = 0
uname({sysname="Linux", nodename="reiss-t3620", ...}) = 0
readlink("/proc/self/exe", "/u/cr4bd/spring2023/cs3130/slide"..., 4096)
    = 57
brk(0x231a1c0)
    = 0x231a1c0
brk(0x231b000)
    = 0x231b000
access("/etc/ld.so.nohwcap", F_OK)
    = -1 ENOENT (No such file or
                                directory)
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 4), ...}) = 0
write(1, "Hello, World!\n", 14)
    = 14
exit_group(0)
    = ?
+++ exited with 0 +++
```

aside: what are those syscalls?

execve: run program

brk: allocate heap space

arch_prctl(ARCH_SET_FS, ...): thread local storage pointer
may make more sense when we cover concurrency/parallelism later

uname: get system information

readlink of /proc/self/exe: get name of this program

access: can we access this file?
(file indicates whether to use 'advanced' process features)

fstat: get information about open file

exit_group: variant of exit

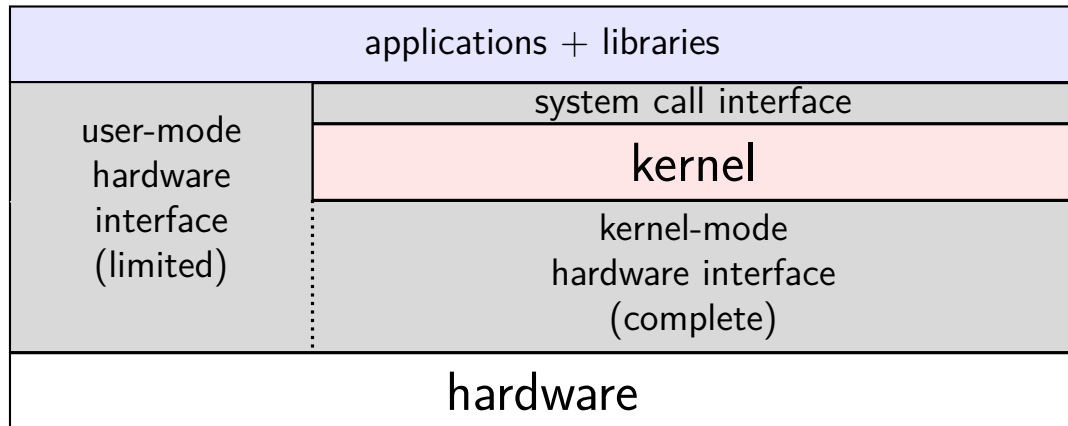
strace hello_world (2)

```
#include <stdio.h>
int main() { puts("Hello, World!"); }
```

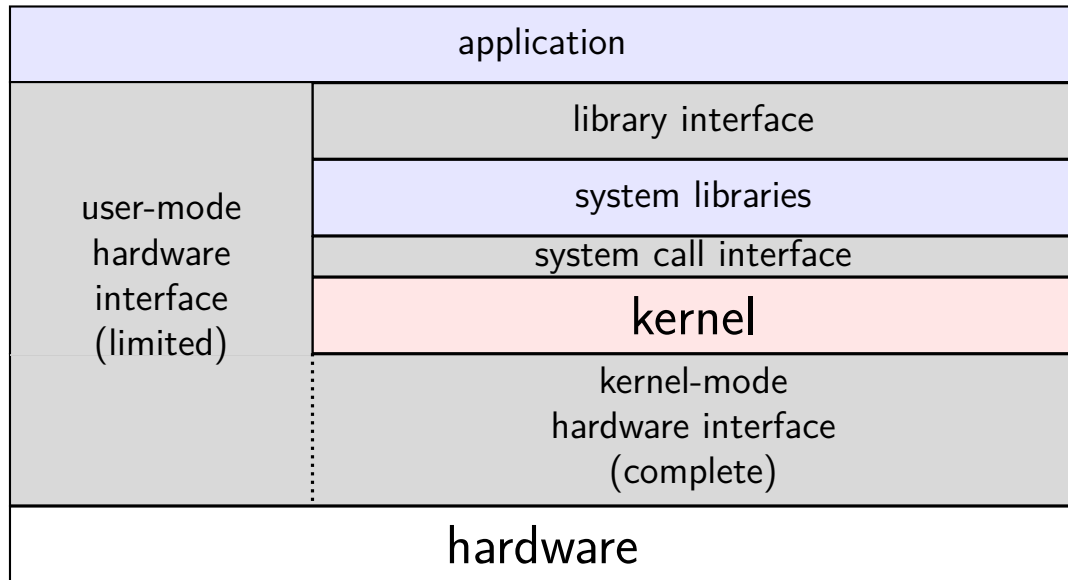
when dynamically linked:

```
execve("./hello_world", ["./hello_world"], 0x7ffcfe91d540 /* 28 vars */)
    = 0
brk(NULL)
    = 0x55d6c351b000
access("/etc/ld.so.nohwcap", F_OK)
    = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)
    = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=196684, ...}) = 0
mmap(NULL, 196684, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f7a62dd3000
close(3)
    = 0
access("/etc/ld.so.nohwcap", F_OK)
    = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20\35\2\0\0\0\0\0"
    = 832
...
close(3)
    = 0
write(1, "Hello, World!\n", 14)
    = 14
exit_group(0)
    = ?
```

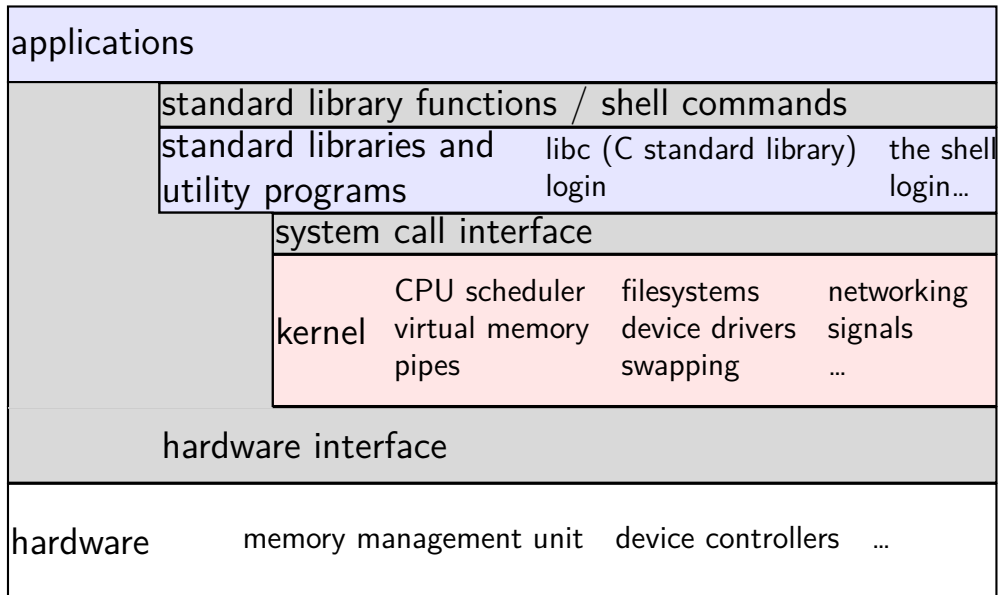

hardware + system call interface



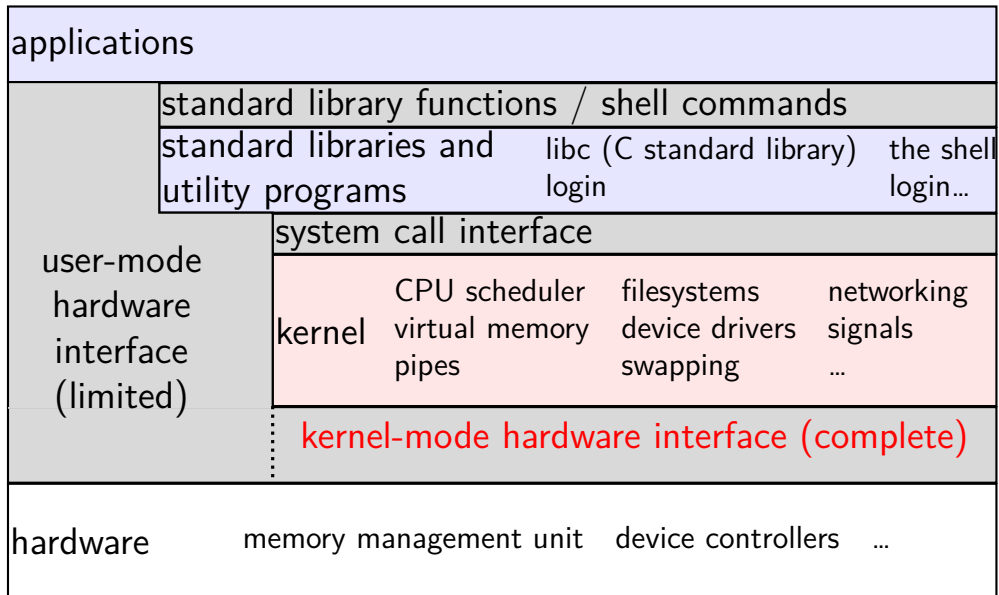
hardware + system call + library interface



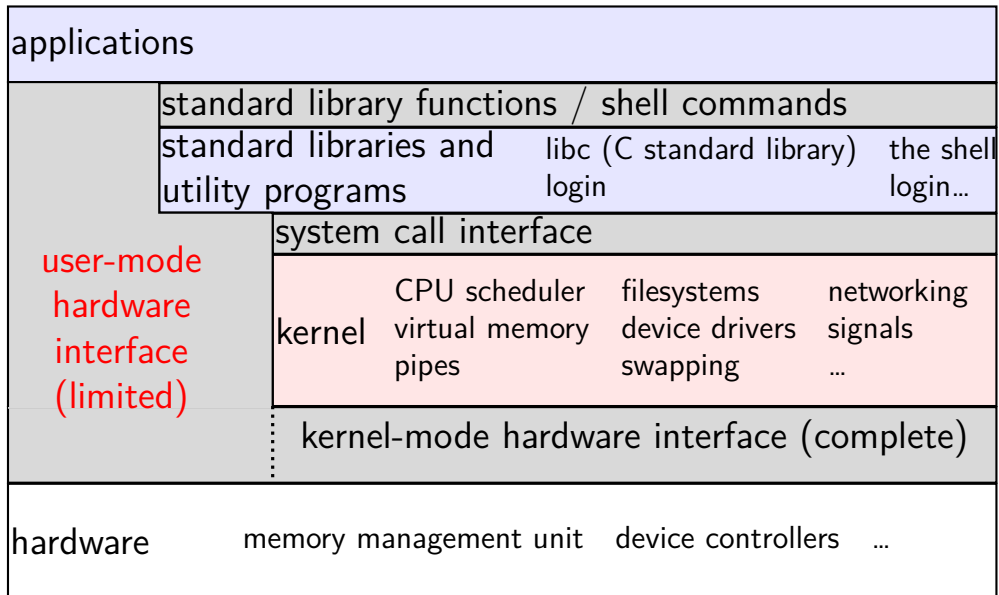
the classic Unix design



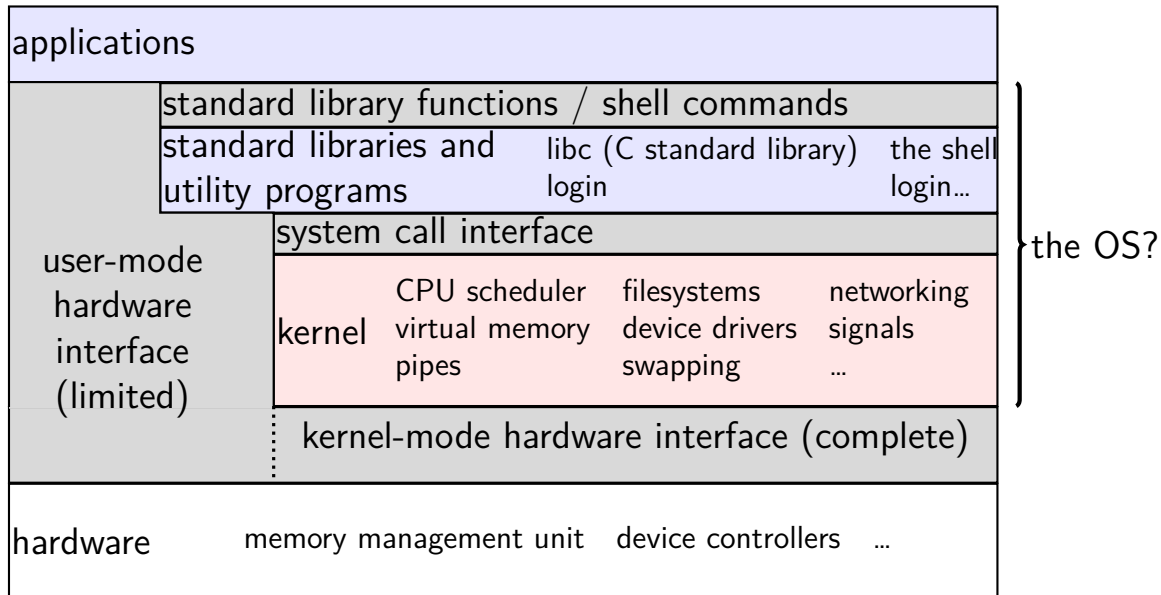
the classic Unix design



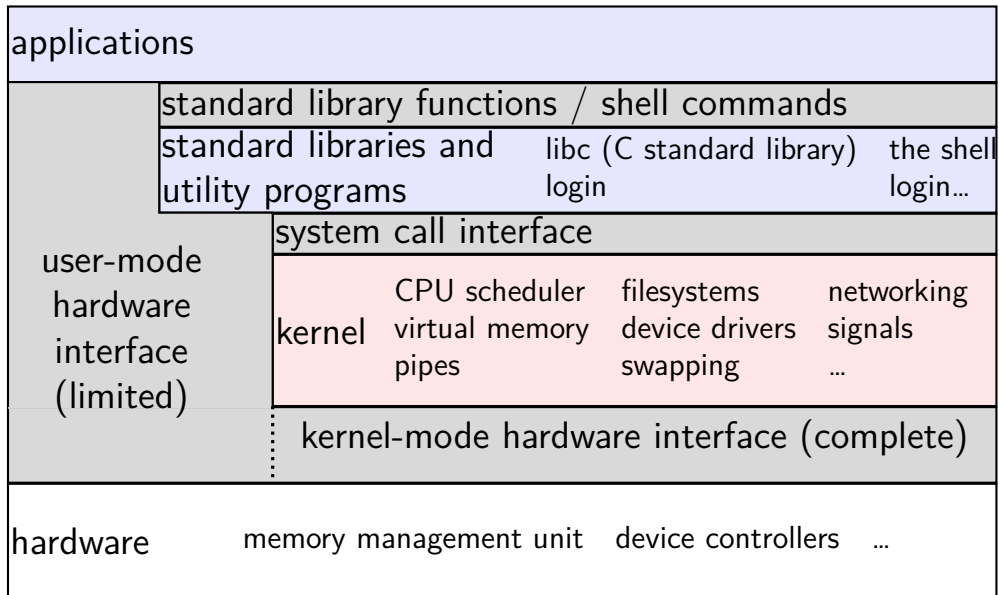
the classic Unix design



the classic Unix design



the classic Unix design



} the OS?

aside: is the OS the kernel?

OS = stuff that runs in kernel mode?

OS = stuff that runs in kernel mode + libraries to use it?

OS = stuff that runs in kernel mode + libraries + utility programs
(e.g. shell, finder)?

OS = everything that comes with machine?

no consensus on where the line is

each piece can be replaced separately...

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
<pre>0x10000: .word 42 // ... // do work // ... movq 0x10000, %rax</pre>	<pre><i>// while A is working:</i> movq \$99, %rax movq %rax, 0x10000 ...</pre>

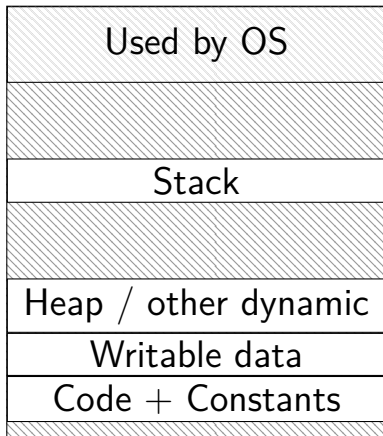
memory protection

reading from another program's memory?

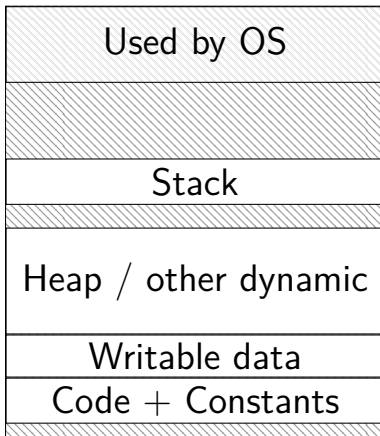
Program A	Program B
<pre>0x10000: .word 42 // ... // do work // ... movq 0x10000, %rax</pre>	<pre><i>// while A is working:</i> movq \$99, %rax movq %rax, 0x10000 ...</pre>
<p>result: %rax (in A) is ...</p> <p>A. 42 B. 99 C. 0x10000</p> <p>D. 42 or 99 (depending on timing/program layout/etc)</p> <p>E. 42 or 99 or program might crash (depending on ...)</p> <p>F. something else</p>	

program memory (two programs)

Program A



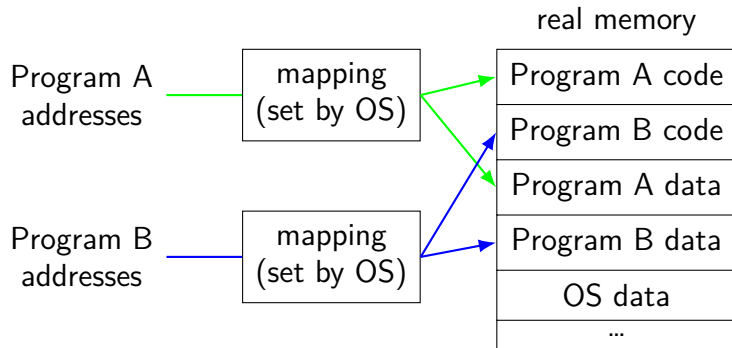
Program B



address space

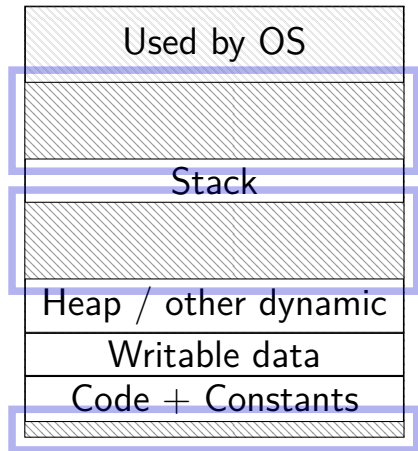
programs have **illusion of own memory**

called a program's **address space**

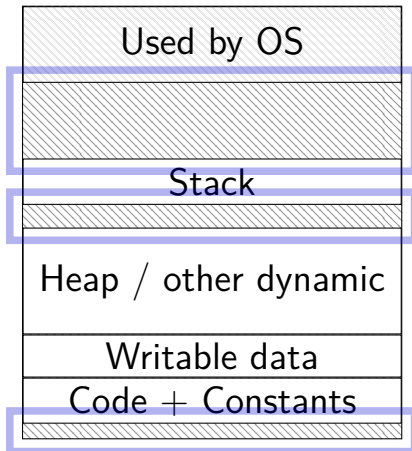


program memory (two programs)

Program A



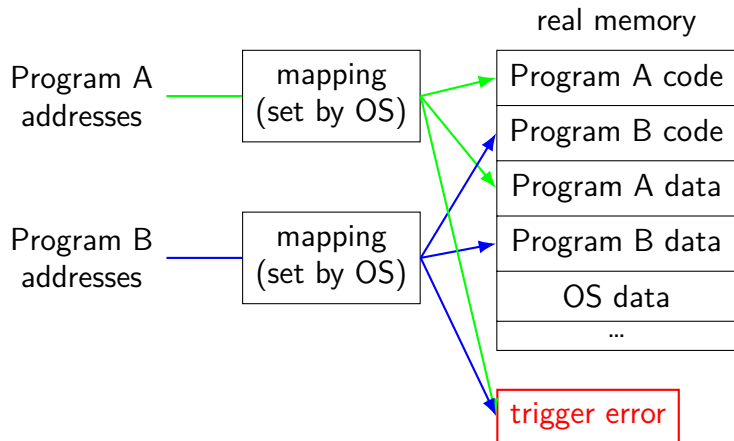
Program B



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**

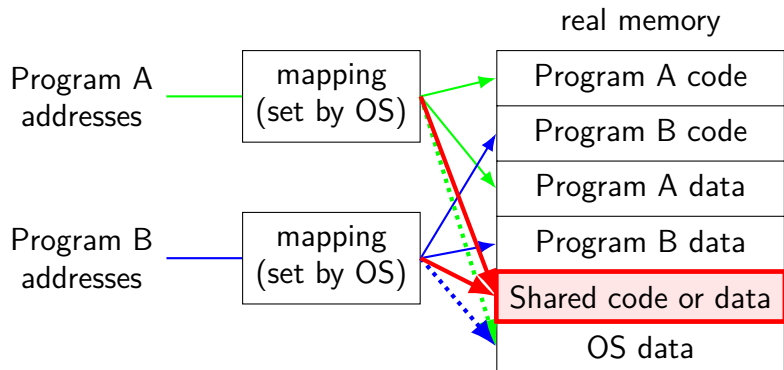
mapping part of what is changed in context switch

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 */  
    }  
}
```

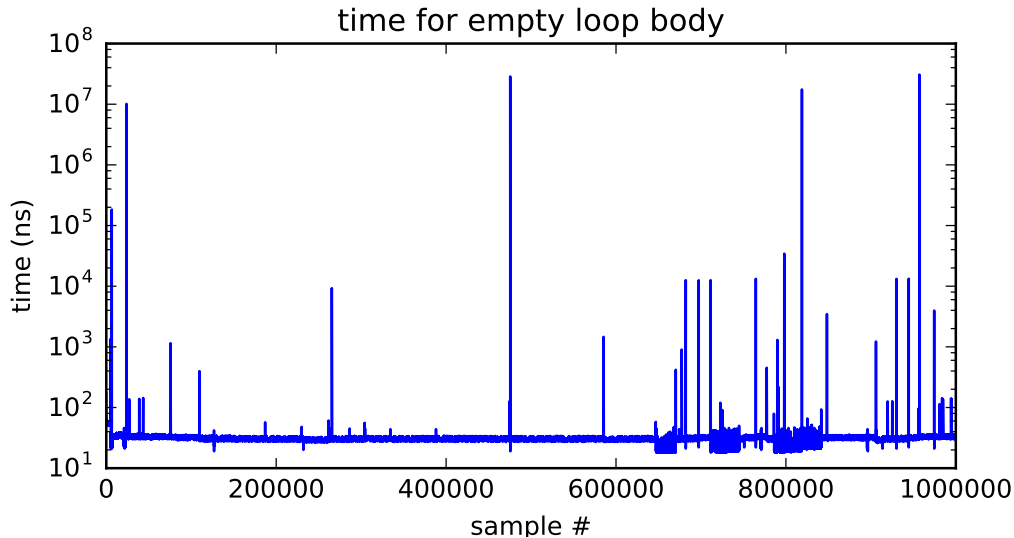
If I run this on a shared department machine, can you still use it?
...if the machine only has one core?

timing nothing

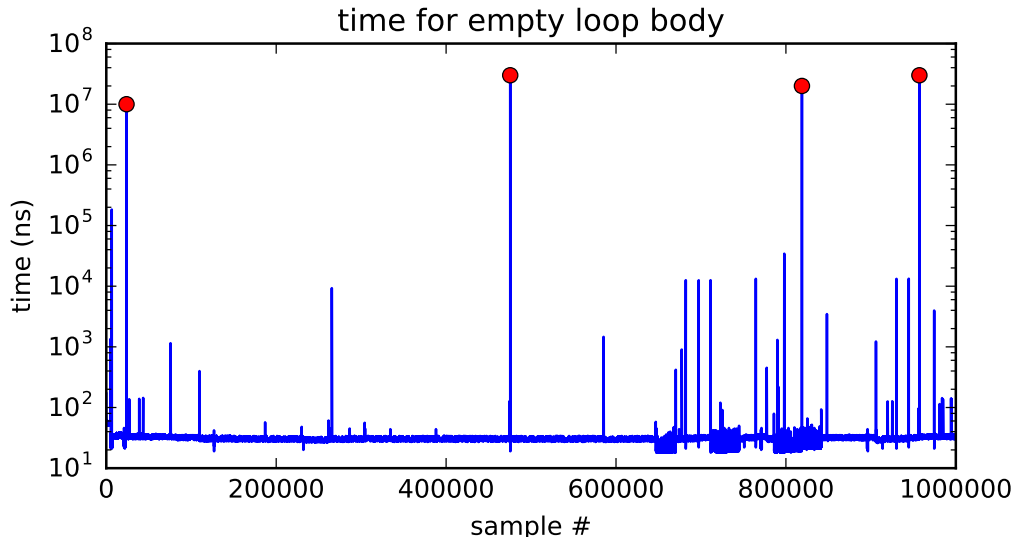
```
long times[NUM_TIMINGS];  
int main(void) {  
    for (int i = 0; i < N; ++i) {  
        long start, end;  
        start = get_time();  
        /* do nothing */  
        end = get_time();  
        times[i] = end - start;  
    }  
    output_timings(times);  
}
```

same instructions — same difference each time?

doing nothing on a busy system

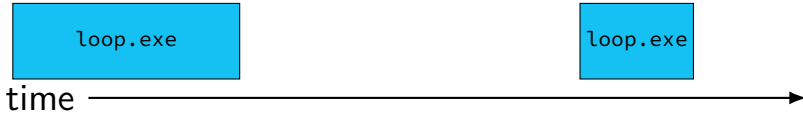


doing nothing on a busy system



time multiplexing

processor:



time multiplexing



...

```
call get_time
```

```
// whatever get_time does
```

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

———— million cycle delay ————

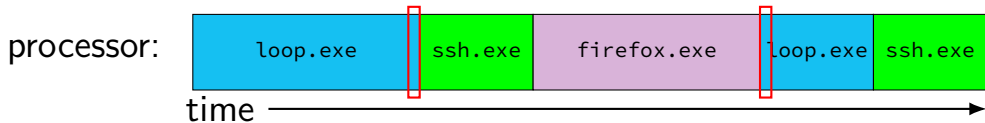
```
call get_time
```

```
// 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
```

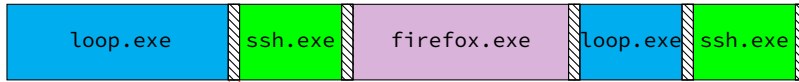
...

time multiplexing really



= operating system

time multiplexing really



= operating system

exception happens

return from exception

threads

thread = illusion of own processor

own register values

own program counter value

threads

thread = illusion of own processor

own register values

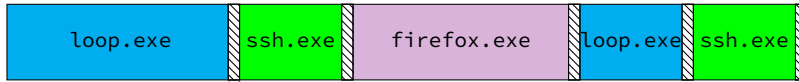
own program counter value

actual implementation:

many threads sharing one processor

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

time multiplexing really



= operating system

exception happens

return from exception

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**

context

all registers values

`%rax %rbx, ..., %rsp, ...`

condition codes

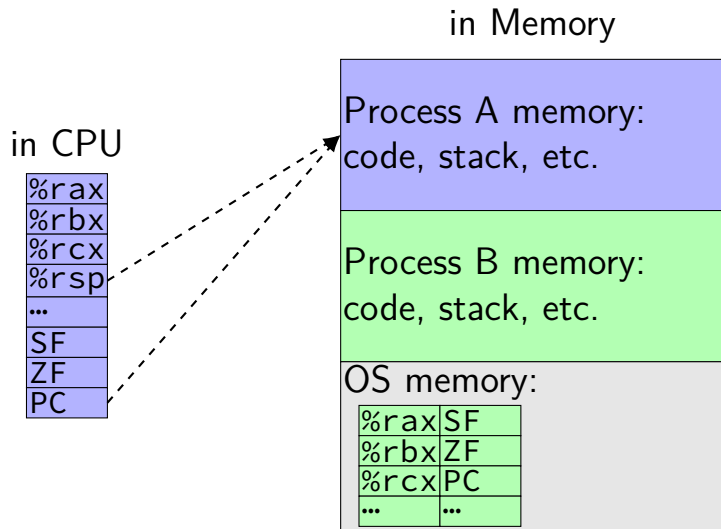
program counter

address space (map from program to real addresses)

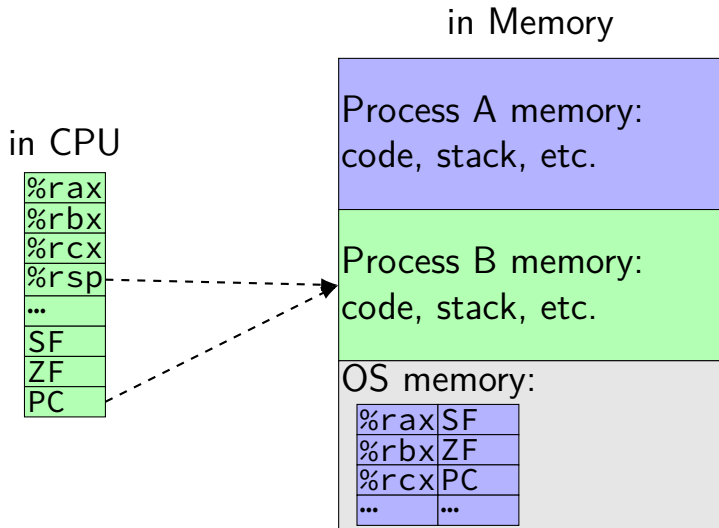
context switch pseudocode

```
context_switch(last, next):  
    copy_preexception_pc last->pc  
    mov rax, last->rax  
    mov rcx, last->rcx  
    mov rdx, last->rdx  
    ...  
    mov next->rdx, rdx  
    mov next->rcx, rcx  
    mov next->rax, rax  
    jmp next->pc
```

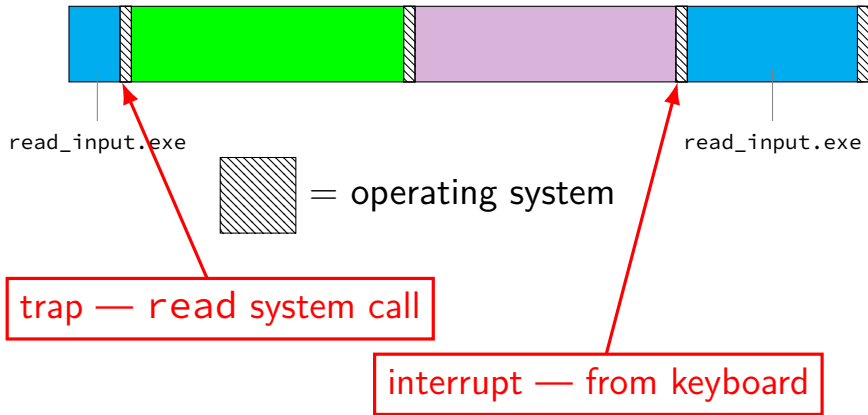
contexts (A running)



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)

asynchronous

not triggered by
running program

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

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)

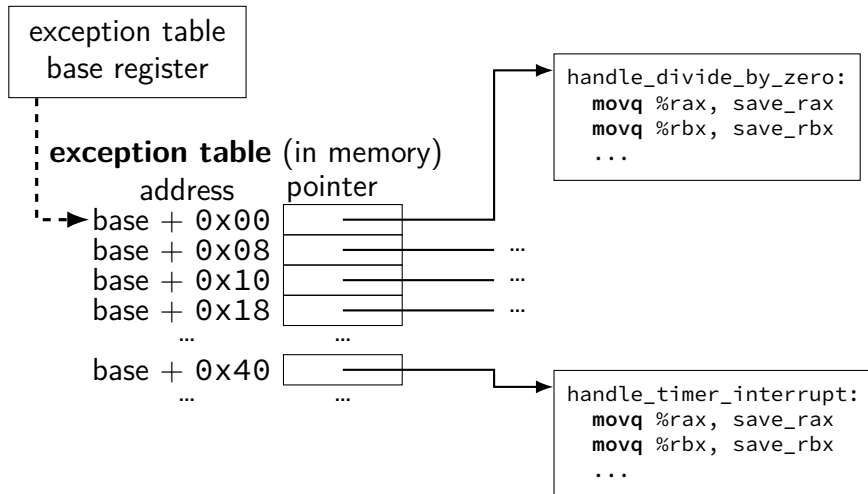
jump done without program instruction to do so

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