



# last time

course intro / logistics

building C programs (cc = clang/gcc, etc.)

`cc -c file.c` (makes `file.o`) — compile+assemble  
reads `file.c` and all the files it `#includes`

`cc file1.o file2.o ... -o executable` — link  
reads `.o` files + some system files

`...-Lpath -lname` — libraries:

static (`libname.a`): included in executable itself

dynamic (`libname.so`): found + loaded at program start — one copy on system

runtime search paths for dynamic libraries

# anonymous feedback (1)

Holding class in Mcleod is very hard for most of this class as I know at least DMT2, which a lot of us are in , is all the way in gilmer and is a more than 15 min to get here. I ask to please consider holding class somewhere else on grounds or starting class a few minutes later every day so that everyone has ample time to get here and be prepared to learn.

I'm pretty sure there's not an alternative room (I didn't volunteer for a long walk from Rice...)

disappointed to lose some lecture time, ...

# quiz demo

## anonymous feedback (2)

I was hoping you could do some introductions to some concepts before diving into the slides. I think it would help clarify things for us before we learn new content due to the large gap we have had since last talking about this subject. Also if you could continue some in class exercises and add examples to the slides that would be very helpful.

Is there a way we could get a C refresher. Are there any good resources to learn memory allocation etc? We did not have good practice with that in cs01

# warmup assignment

## C review Q

```
int array[4] = {10,20,30,40};  
int *p;  
p = &array[0];  
p += 2;  
p[1] += 1;
```

array =

- |                             |                  |               |
|-----------------------------|------------------|---------------|
| A. compile or runtime error | B. {10,20,30,41} |               |
| C. {10,20,32,41}            | D.               | {10,21,30,40} |
| E. {11,21,30,40}            | F.               | none of these |

# some avenues for review

review CSO1 stuff

labs 9–12 (of last Spring)

[https://www.cs.virginia.edu/~jh2jf/courses/  
cs2130/spring2023/](https://www.cs.virginia.edu/~jh2jf/courses/cs2130/spring2023/)

exercises we've used in the past:

implement strsep library function

implement conversion from dynamic array to linked list



## some pointer stuff

0x040

0x038

0x030

0x028

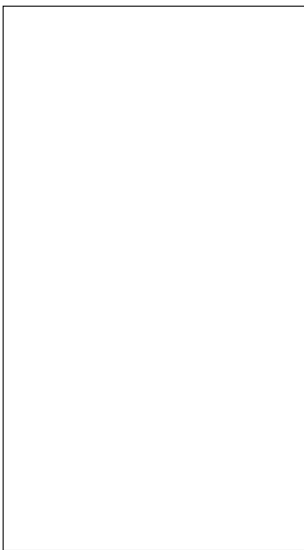
0x020

0x018

0x010

0x008

0x000



```
int array[3]={0x12,0x45,0x67};  
int single = 0x78;  
int *ptr;
```

## some pointer stuff

0x040

0x038

0x030

0x028

0x020

0x018

0x010

0x008

0x000

array[2]: 0x67
array[1]: 0x45
array[0]: 0x12
single: 0x78
ptr = ???

```
int array[3]={0x12,0x45,0x67};  
int single = 0x78;  
int *ptr;
```

## some pointer stuff

0x040	
0x038	
0x030	array[2]: 0x67
	array[1]: 0x45
	array[0]: 0x12
0x028	single: 0x78
0x020	ptr = ???
0x018	
0x010	
0x008	
0x000	

```
int array[3]={0x12,0x45,0x67};  
int single = 0x78;  
int *ptr;
```

~~\*ptr = 0xAB;~~ compile error

## some pointer stuff

0x040	
0x038	
0x030	array[2]: 0x67
	array[1]: 0x45
	array[0]: 0x12
0x028	single: 0x78
	ptr: 0x28
0x020	
0x018	
0x010	
0x008	
0x000	

```
int array[3]={0x12,0x45,0x67};  
int single = 0x78;  
int *ptr;
```

```
ptr = &single;
```

```
ptr = (int*) 0x28;  addr. of single
```

## some pointer stuff

0x040	
0x038	
0x030	array[2]: 0x67
	array[1]: 0x45
	array[0]: 0x12
0x028	single: 0x78
	ptr: 0x28
0x020	
0x018	
0x010	
0x008	
0x000	

```
int array[3]={0x12,0x45,0x67};  
int single = 0x78;  
int *ptr;
```

```
ptr = &single;  
ptr = (int*) 0x28;    addr. of single
```

~~ptr = 0x28; compile error~~

~~ptr = (int\*) single;~~  
pointer to unknown place

## some pointer stuff

0x040	
0x038	array[2]: 0x67
	array[1]: 0x45
0x030	array[0]: 0x12
0x028	single: 0xFF
	ptr: 0x28
0x020	
0x018	
0x010	
0x008	
0x000	

```
int array[3]={0x12,0x45,0x67};  
int single = 0x78;  
int *ptr;  
ptr = &single;
```

```
*ptr = 0xFF;
```

## some pointer stuff

0x040	
0x038	
0x030	array[2]: 0x67
	array[1]: 0x45
	array[0]: 0x12
0x028	single: 0x78
	ptr: 0x2C
0x020	
0x018	
0x010	
0x008	
0x000	

```
int array[3]={0x12,0x45,0x67};  
int single = 0x78;  
int *ptr;
```

```
ptr = array;  
ptr = &array[0];  
ptr = (int*) 0x2C;
```

## some pointer stuff

0x040	
0x038	
0x030	array[2]: 0x67
	array[1]: 0x45
	array[0]: 0x12
0x028	single: 0x78
	ptr: 0x2C
0x020	
0x018	
0x010	
0x008	
0x000	

```
int array[3]={0x12,0x45,0x67};  
int single = 0x78;  
int *ptr;
```

```
ptr = array;  
ptr = &array[0];  
ptr = (int*) 0x2C;
```

~~ptr = array[0]; compile error~~

~~ptr = (int\*) array[0];~~

pointer to unknown place



## some pointer stuff

0x040

0x038

0x030

0x028

0x020

0x018

0x010

0x008

0x000

array[2]: 0xFF
array[1]: 0x45
array[0]: 0x12
single: 0x78
ptr: 0x2C

```
int array[3]={0x12,0x45,0x67};  
int single = 0x78;  
int *ptr;  
ptr = &array[0];
```

```
ptr[2] = 0xFF;  
*(ptr + 2) = 0xFF;
```

```
int *temp1; temp1 = ptr + 2;  
*temp1 = 0xFF;
```

```
int *temp2; temp2 = &ptr[2];  
*temp2 = 0xFF;
```

## some pointer stuff

0x040	
0x038	array[2]: 0x67
	array[1]: 0x45
0x030	array[0]: 0x12
0x028	single: ...
	ptr: 0x2C
0x020	
0x018	
0x010	
0x008	
0x000	

```
int array[3]={0x12,0x45,0x67};  
int single = 0x78;  
int *ptr;
```

```
void change_arg(int *x) {  
    *x = compute_some_value();  
}  
...  
change_arg(&single);
```

# 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 runs commands if any prereq modified date after 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 runs commands if any prereq modified date after 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 runs commands if any prereq modified date after 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 runs commands if any prereq modified date after 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 runs commands if any prereq modified date after 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 runs commands if any prereq modified date after 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 runs commands if any prereq modified date after 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, scones, 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 operation: problem

how can hardware (HW) plus operating system (OS) allow:  
    read your own files from hard drive

but disallow:  
    read others files from hard drive

## some ideas

OS tells HW 'okay' parts of hard drive before running program code

complex for hardware and for OS

## some ideas

OS tells HW 'okay' parts of hard drive before running program code

- complex for hardware and for OS

OS verifies your program's code can't do bad hard drive access

- no work for HW, but complex for OS

- may require compiling differently to allow analysis



## some ideas

OS tells HW 'okay' parts of hard drive before running program code

- complex for hardware and for OS

OS verifies your program's code can't do bad hard drive access

- no work for HW, but complex for OS

- may require compiling differently to allow analysis

OS tells HW to only allow OS-written code to access hard drive

- that code can enforce only 'good' accesses

- requires program code to call OS routines to access hard drive

- relatively simple for hardware

# kernel mode

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

other names: privileged mode, supervisor mode, ...

not in kernel mode = *user mode*

certain operations only allowed in kernel mode

*privileged instructions*

example: talking to any I/O device

# what runs in kernel mode?

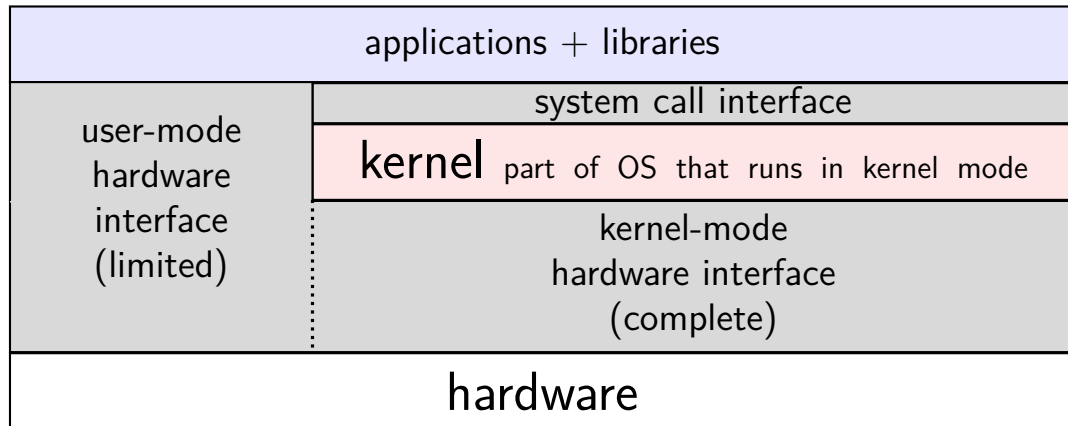
system boots in kernel mode

OS switches to user mode to run program code

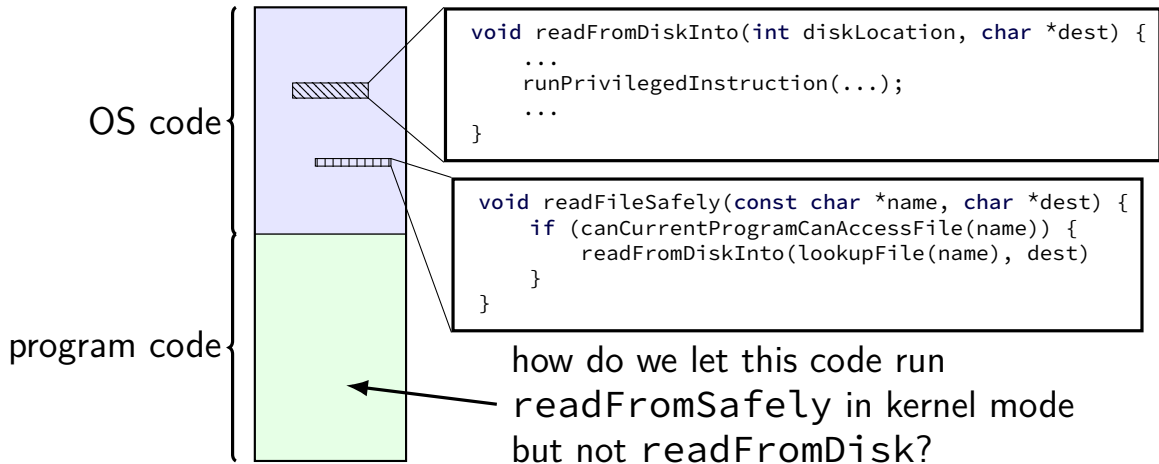
next topic: when does system switch back to kernel mode?

how does OS tell HW where the (trusted) OS code is?

# hardware + system call interface



# 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", 14)      = 14  
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 [in this case, a config file]?

fstat: get information about open file

exit\_group: variant of exit

## strace hello\_world (2)

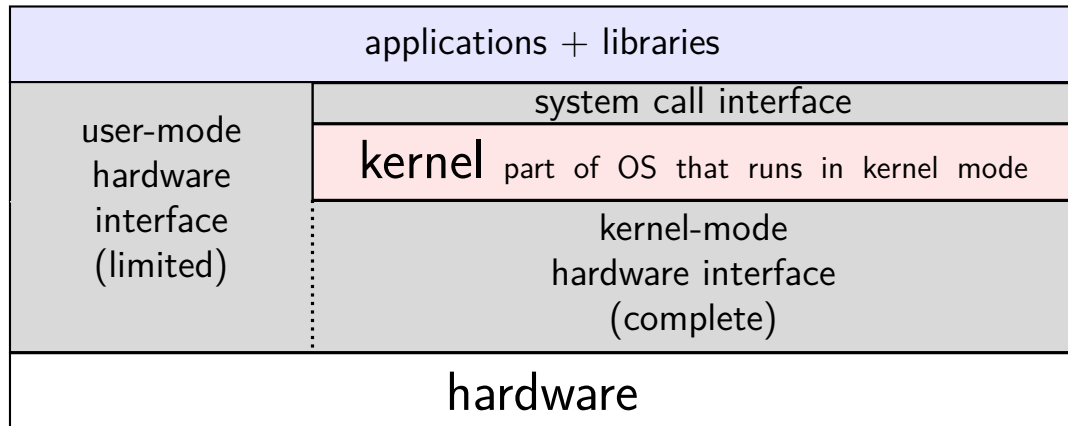
```
#include <stdio.h>
int main() { puts("Hello, World!"); }
```

---

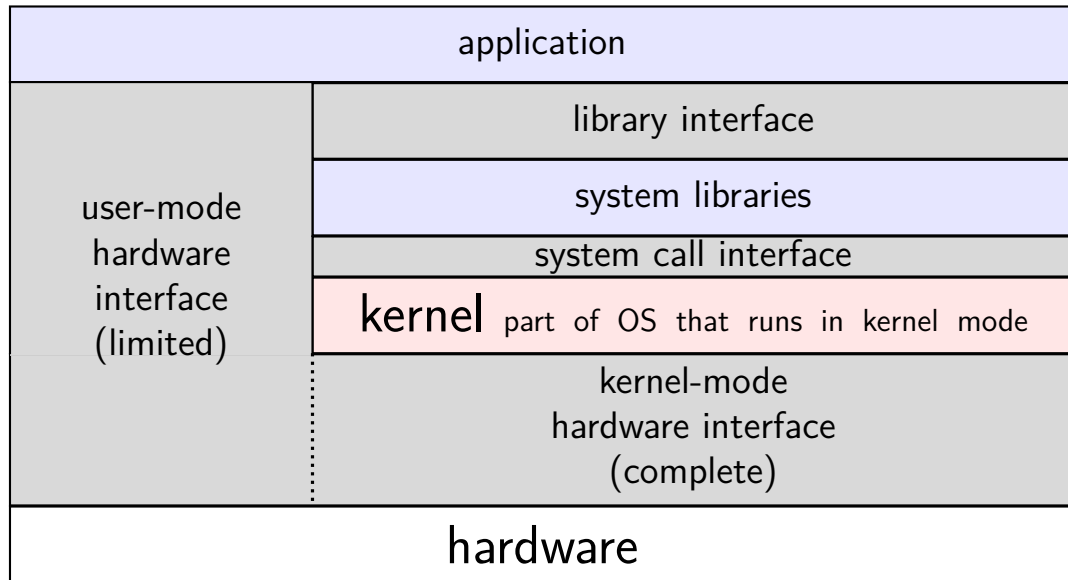
when dynamically linked:

```
execve("./hello_world", ["./hello_world"], 0x7ffcfe91d540 /* 28 vars */)
    = 0
brk(NULL)
    = 0x55d6c351b000
...
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\0\0\0\3\0>\0\1\0\0\0"..., 832) = 832
...
close(3)
    = 0
write(1, "Hello, World!\n", 14)
    = 14
exit_group(0)
    = ?
+++ exited with 0 +++
```

# hardware + system call interface



# hardware + system call + library interface



# 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

modifying another program's memory?

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



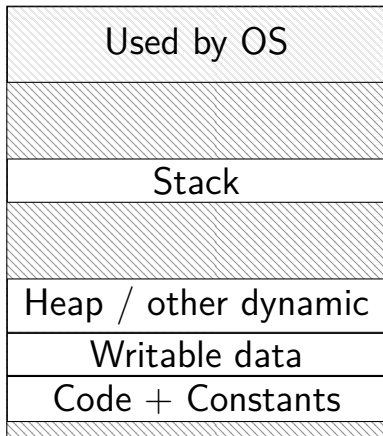
# memory protection

modifying another program's memory?

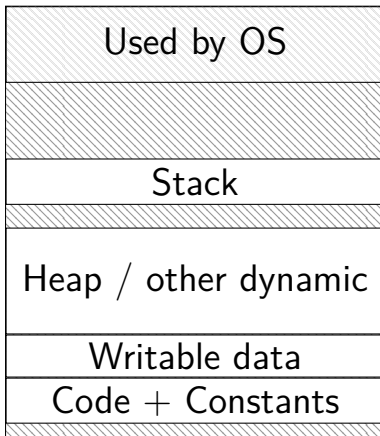
Program A	Program B
<pre>0x10000: .long 42 // ... // do work // ... movq 0x10000, %rax</pre>	<pre><i>// while A is working:</i> movq \$99, %rax movq %rax, 0x10000 ...</pre>
<p>result: %rax (in A) is ...</p> <p>A. 42      B. 99      C. 0x10000</p> <p>D. 42 or 99 (depending on timing/program layout/etc)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

# program memory (two programs)

Program A



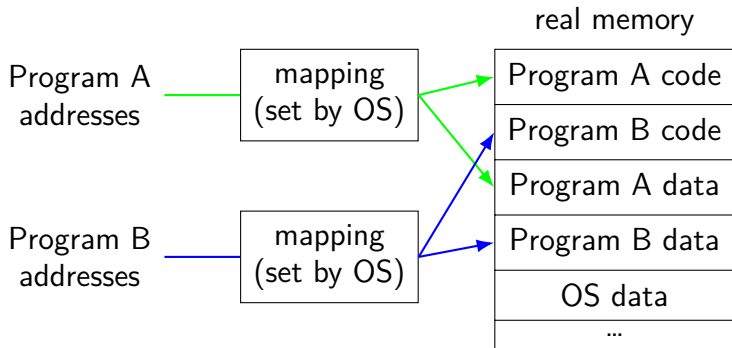
Program B



# address space

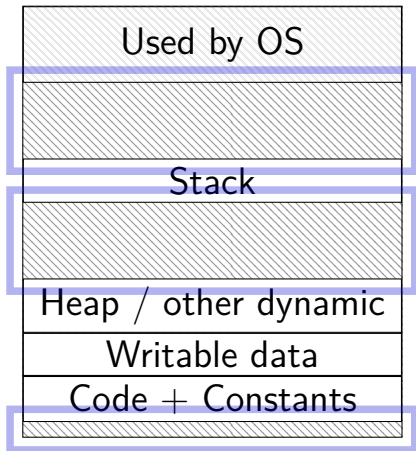
programs have **illusion of own memory**

called a program's **address space**

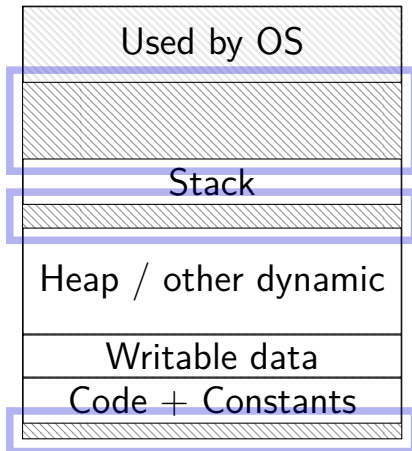


# program memory (two programs)

Program A



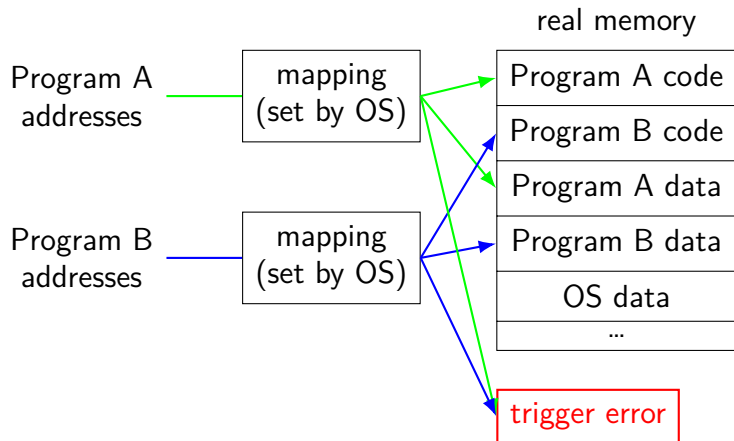
Program B



# 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**

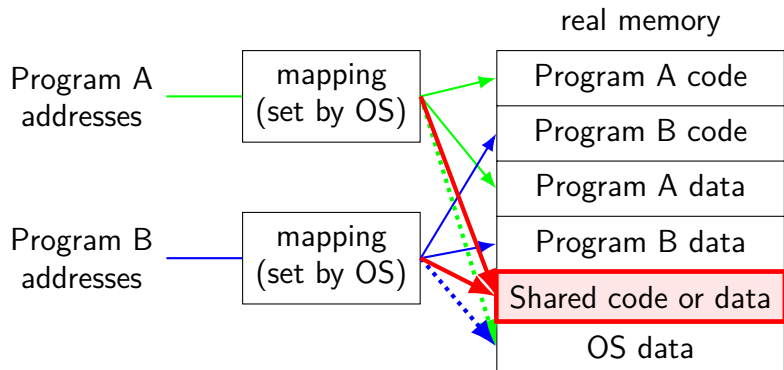
mapping part of what is changed in context switch

# 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



# memory protection

modifying another program's memory?

Program A	Program B
<pre>0x10000: .long 42 // ... // do work // ... movq 0x10000, %rax</pre>	<pre><i>// while A is working:</i> movq \$99, %rax movq %rax, 0x10000 ...</pre>
result: %rax (in A) is 42 (always)	result: <b>might crash</b>

- 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 crashing?

what happens on processor when program crashes?

other program informed of crash to display message

use processor to run some other program

# program crashing?

what happens on processor when program crashes?

other program informed of crash to display message

use processor to run some other program

how does hardware do this?

would be complicated to tell about other programs, etc.

instead: hardware runs designated OS routine

# exceptions

recall: system calls — software asks OS for help

also cases where hardware asks OS for help

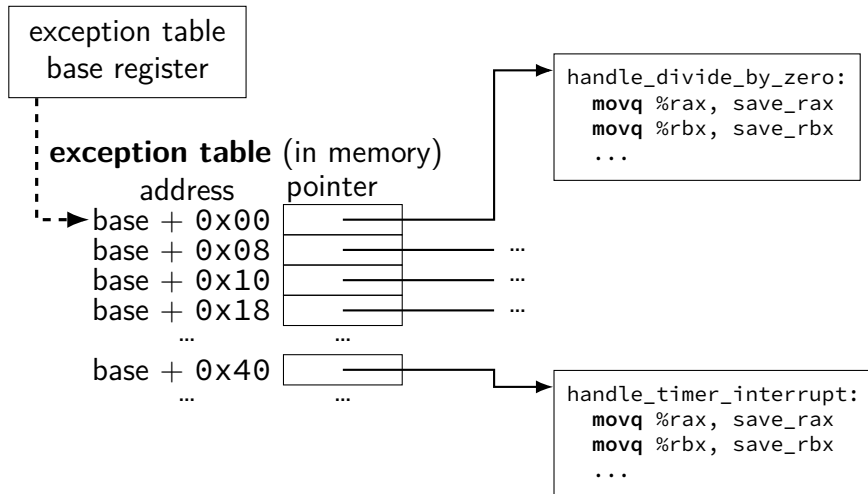
different triggers than system calls

but same mechanism as system calls:

- switch to kernel mode (if not already)

- call OS-designated function

# locating exception handlers (one strategy)



# 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.

# types of exceptions

- system calls

  - intentional — ask OS to do something

- errors/events in programs

  - memory not in address space (“Segmentation fault”)

  - privileged instruction

  - divide by zero, invalid instruction

  - ...

- (and more we'll talk about later)

# types of exceptions

## system calls

intentional — ask OS to do something

## errors/events in programs

memory not in address space (“Segmentation fault”)

privileged instruction

divide by zero, invalid instruction

...

(and more we'll talk about later)



# types of exceptions

- system calls

  - intentional — ask OS to do something

- errors/events in programs

  - memory not in address space (“Segmentation fault”)

  - privileged instruction

  - divide by zero, invalid instruction

  - ...

  - (and more we'll talk about later)

# types of exceptions

system calls

intentional — ask OS to do something

errors/events in programs

memory not in address space (“Segmentation fault”)

privileged instruction

divide by zero, invalid instruction

...

(and more we'll talk about later)

**synchronous**

triggered by  
current program

# 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 */  
    }  
}
```

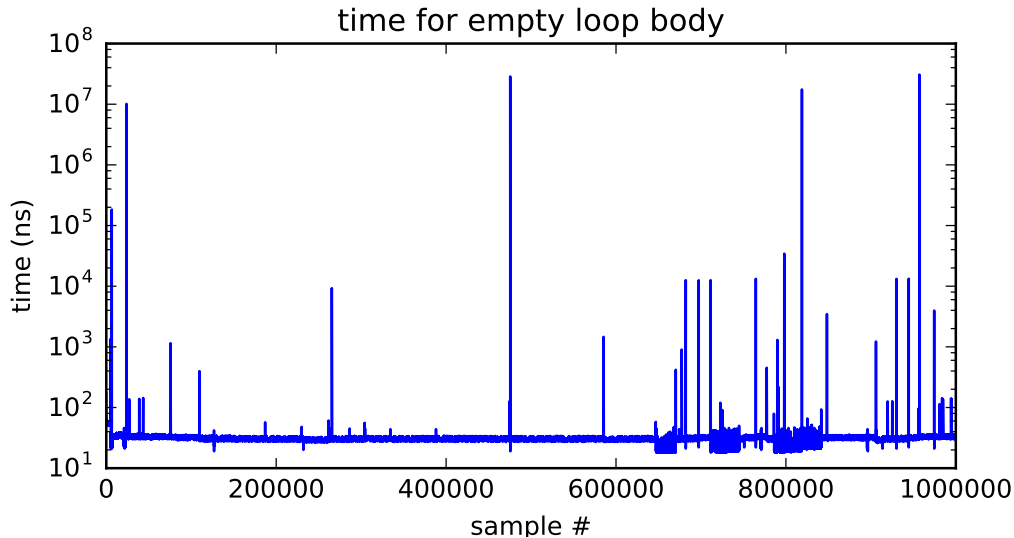
If I run this on a shared department machine, can you still use it?  
...if the machine only has one core?

# timing nothing

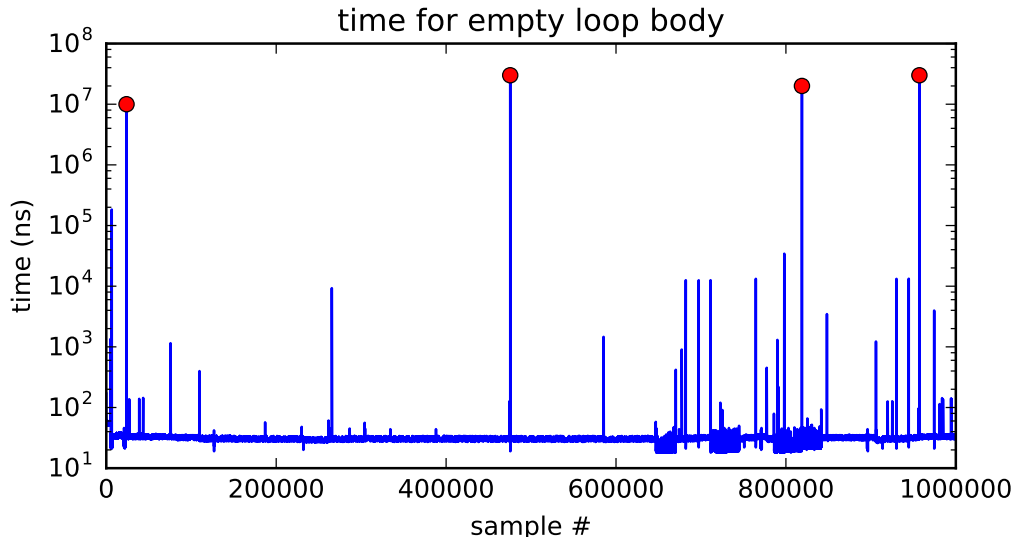
```
long times[NUM_TIMINGS];  
int main(void) {  
    for (int i = 0; i < N; ++i) {  
        long start, end;  
        start = get_time();  
        /* do nothing */  
        end = get_time();  
        times[i] = end - start;  
    }  
    output_timings(times);  
}
```

same instructions — same difference each time?

# doing nothing on a busy system



# doing nothing on a busy system



# types of exceptions

system calls

intentional — 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

**external — I/O, etc.**

timer — configured by OS to run OS at certain time

I/O devices — key presses, hard drives, networks, ...

hardware is broken (e.g. memory parity error)

**asynchronous**

not triggered by  
running program



# time multiplexing

processor:



# time multiplexing



...

```
call get_time
```

```
// whatever get_time does
```

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

———— million cycle delay ————

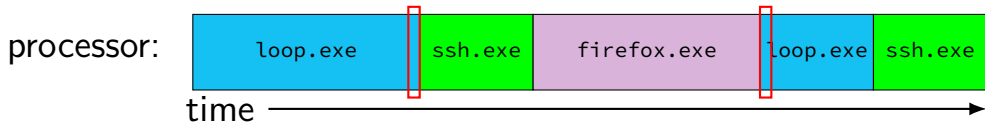
```
call get_time
```

```
// 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
```

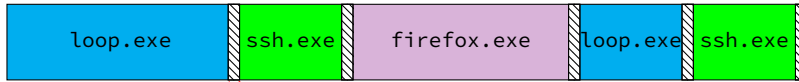
...

# time multiplexing really



= operating system

# time multiplexing really



= operating system

exception happens

return from exception

# types of exceptions

system calls

intentional — ask OS to do something

errors/events in programs

memory not in address space (“Segmentation fault”)

privileged instruction

divide by zero, invalid instruction

...

external — I/O, etc.

**timer** — configured by OS to run OS at certain time

I/O devices — key presses, hard drives, networks, ...

hardware is broken (e.g. memory parity error)

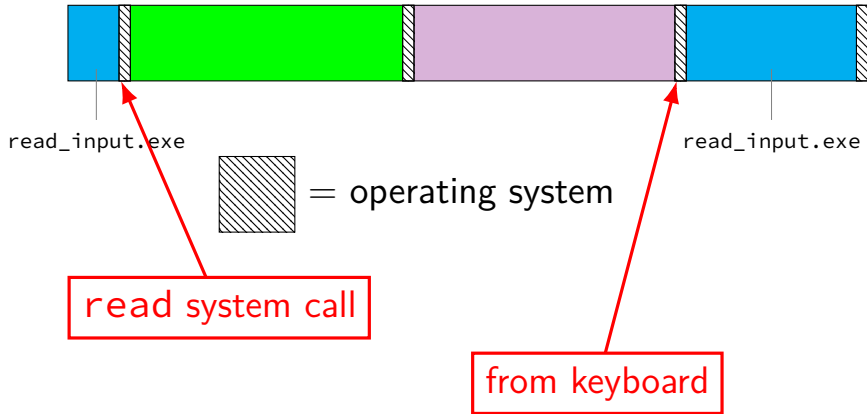
**synchronous**

triggered by  
current program

**asynchronous**

not triggered by  
running program

# keyboard input timeline



# crash timeline timeline



= operating system

out of bounds memory access



# 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?

# switching programs

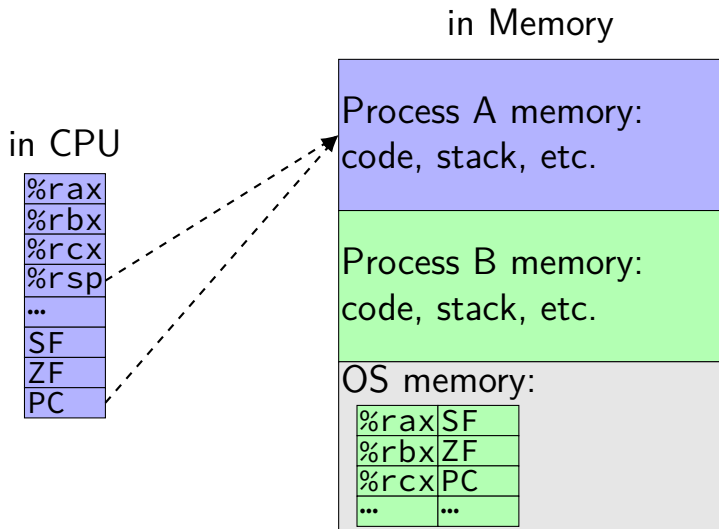
OS starts running somehow  
some sort of exception

saves old registers + program counter  
(optimization: could omit when program crashing/exiting)

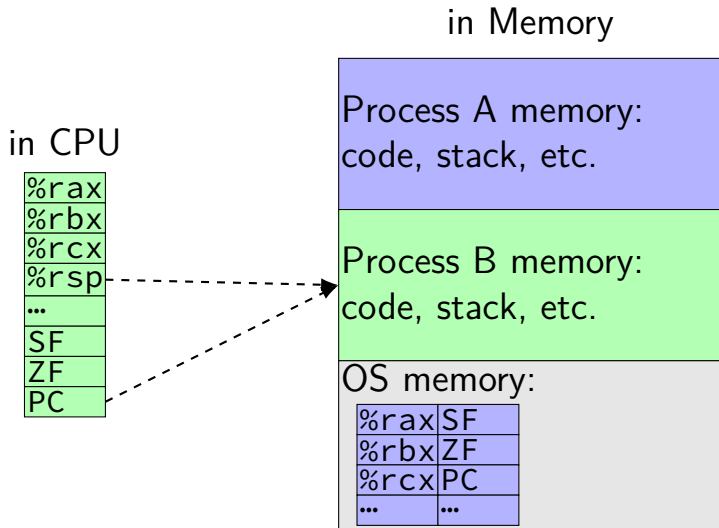
sets new registers, jumps to new program counter

called **context switch**  
saved information called **context**

# contexts (A running)



# contexts (B running)



## review: definitions

exception: hardware calls OS specified routine

- many possible reasons

- system calls: type of exception

context switch: OS switches to another thread

- by saving old register values + loading new ones

- part of OS routine run by exception

## 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

# 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



# The Process

**process** = thread(s) + address space

illusion of **dedicated machine**:

thread = illusion of own CPU

address space = illusion of own memory