changelog

3 Feb 2024: The Process: move slide earlier in slide deck

3 Feb 2024: The Process: mention multiple threads idea on slide

3 Feb 2024: context switch: mention saving/restoring address mapping

3 Feb 2024: shared memory: hide mapping including OS data for now since we haven't explaind kernel-mode-only mappings yet

3 Feb 2024: exercise explanation: for A, give examples of library calls that need no system calls

last time

kernel mode versus user mode

- one-bit register: track which mode
- in kernel mode: full hardware interface
- in user mode: limited interface

normal programs run in user mode

request OS do things that require kernel mode

- (typically through library functions)

system call: make request of OS

- hardware runs *OS-specified* function in kernel mode
- OS function decodes program request (calling convention)

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

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"],
        0x7ffeedafd0a0 /* 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\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 +++
```

aside: system call wrapper versus...

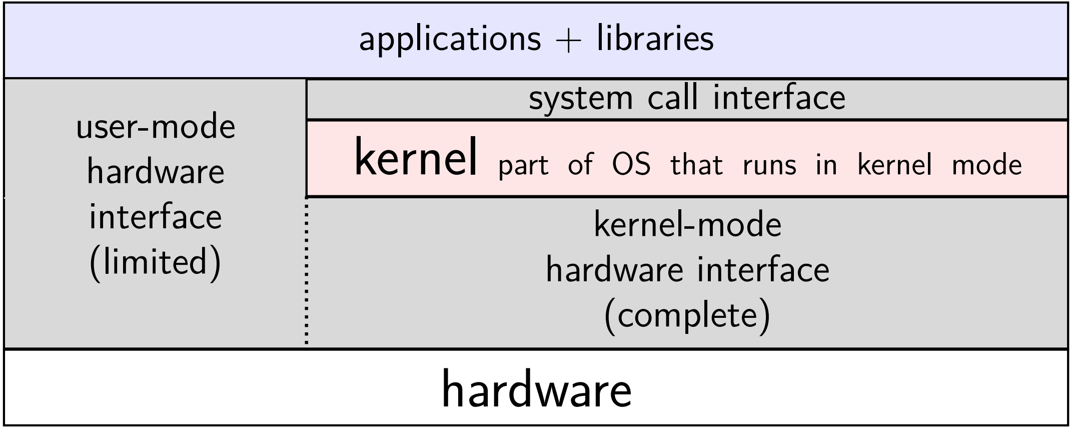
libraries provide *system call wrappers*

- examples on Linux: open(), write(),
- just convert function call to system call

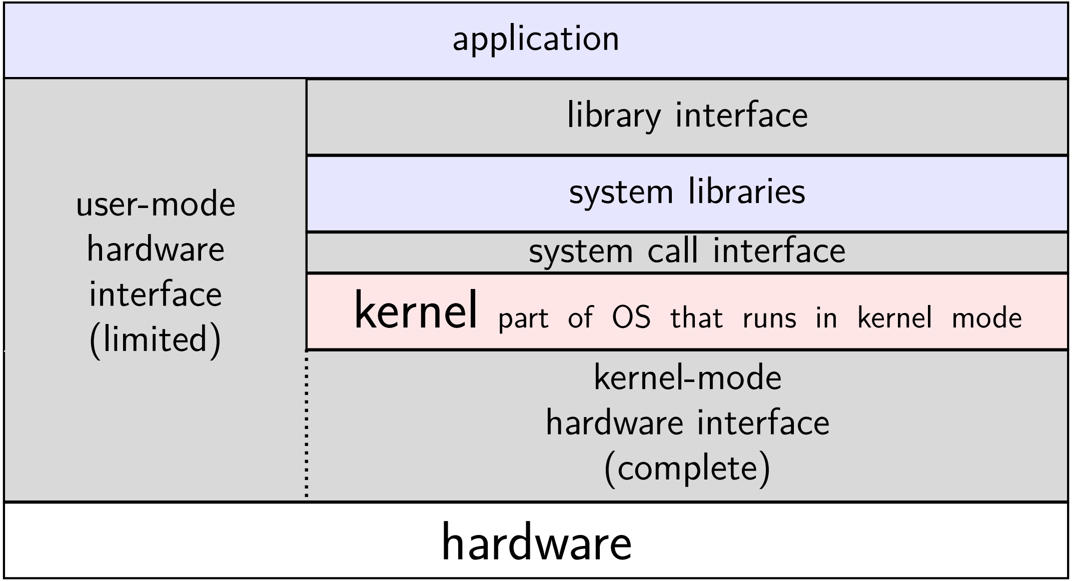
other library functions may incidentally make system calls to implement their functionality

- example: printf implemented using write-bytes system call
- example: malloc implemented using various memory management system calls

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

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

memory protection

modifying another program's memory?

Program A	Program B

`0x10000: .long 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

memory protection

modifying another program's memory?

Program A	Pro

`0x10000: .long 42`
`    // ...`
`    // do work`
`    // ...`
`    movq 0x10000, %rax`
	`//`
	`mov`
	`mov`
	`...`

result: `%rax` (in A) is 42 (always with 'normal' multiuser OSes)

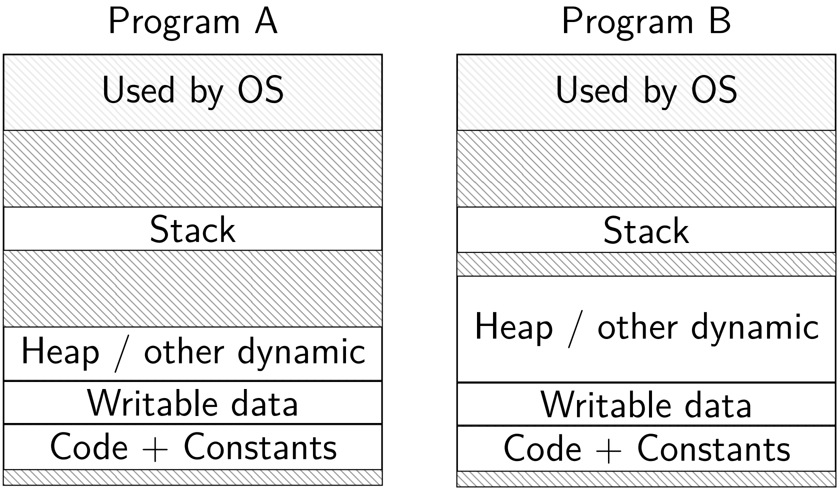
A. 42 B. 99 C. 0x10000

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

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

F. something else

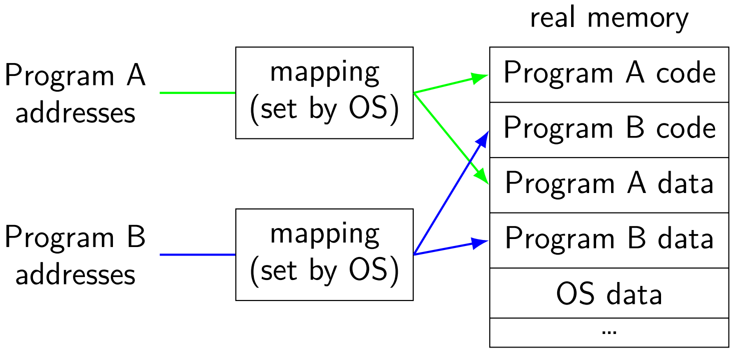
program memory (two programs)



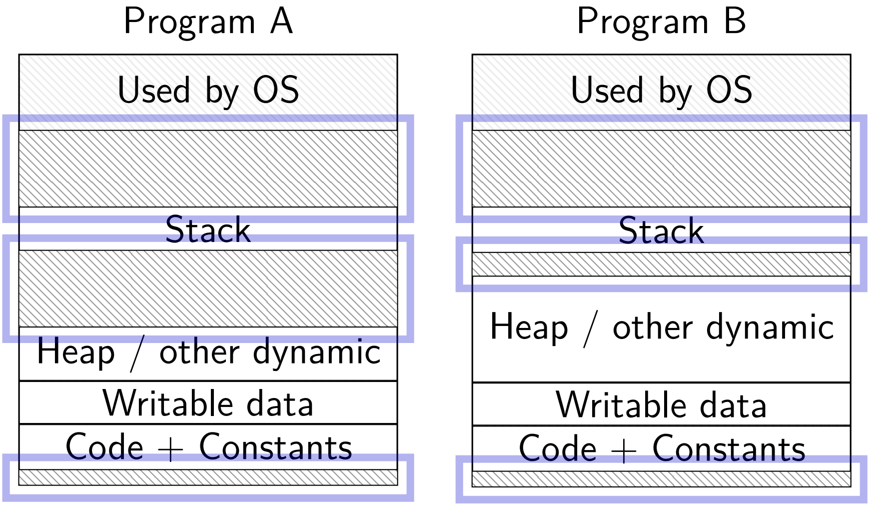
address space

programs have illusion of own memory

called a program's address space



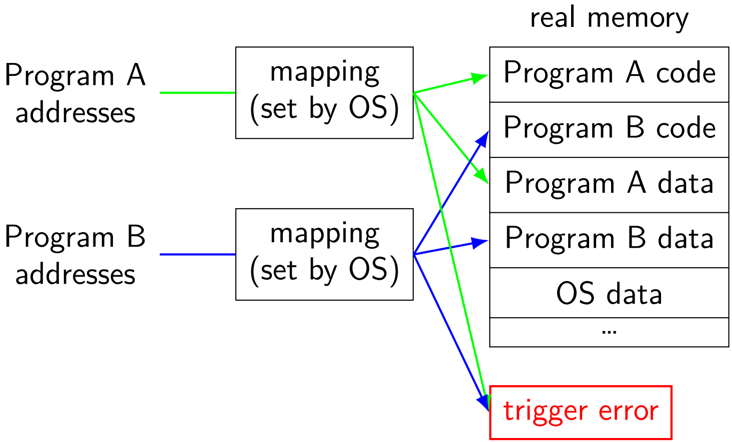
program memory (two programs)



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

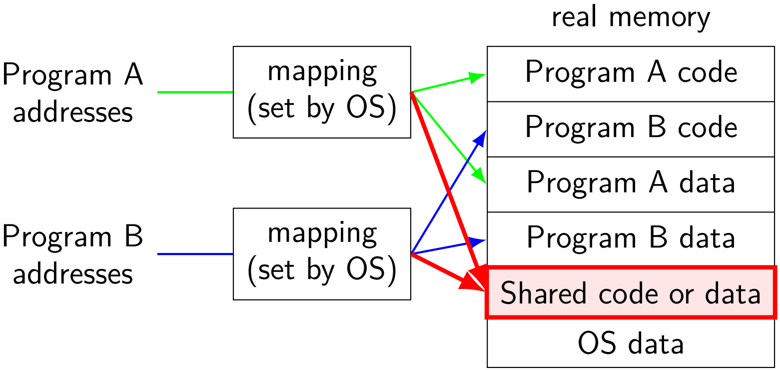
mapping part of what is changed in context switch

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

memory protection

modifying another program's memory?

Program A	Pro

`0x10000: .long 42`
`// ...`
`// do work`
`// ...`
`movq 0x10000, %rax`
	`//`
	`mov`
	`mov`
	`...`

result: `%rax` (in A) is 42 (always with 'normal' multiuser OSes)	res

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

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

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?

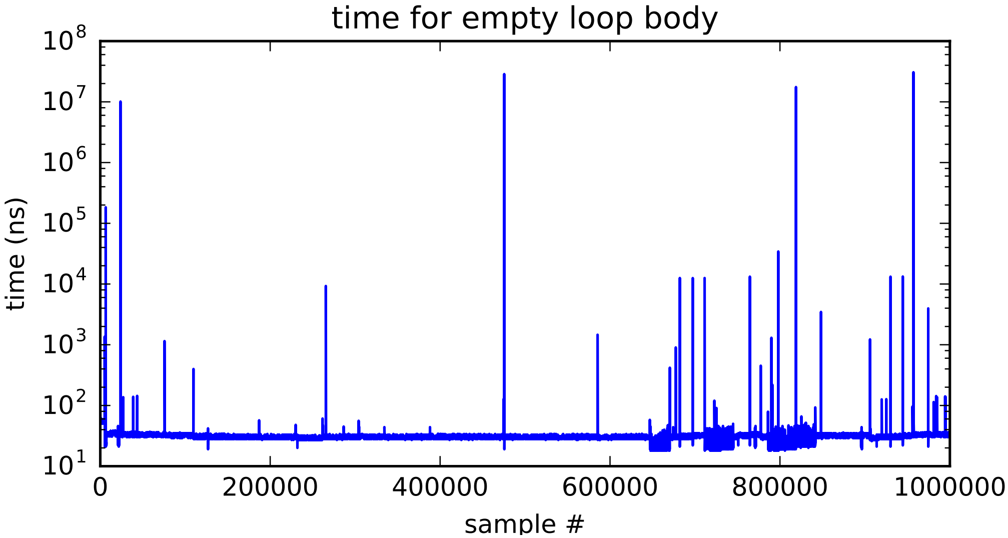
...if the machine only has one core?

timing nothing

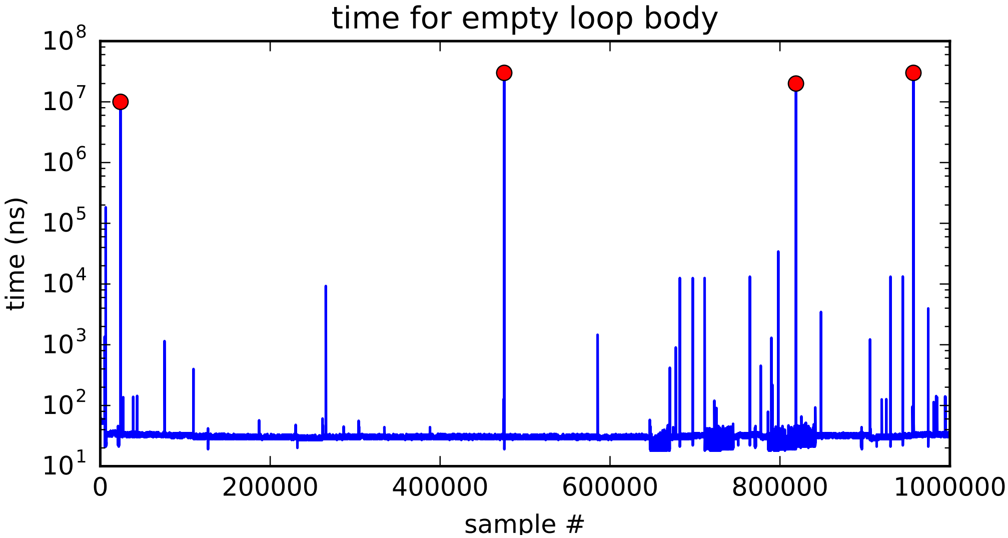
```
long times[NUM_TIMINGS];
int main(void) {
    for (int i = 0; i < N; ++i) {
        long start, end;
        start = get_time();
        /* do nothing */
        end = get_time();
        times[i] = end - start;
    }
    output_timings(times);
}
```

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

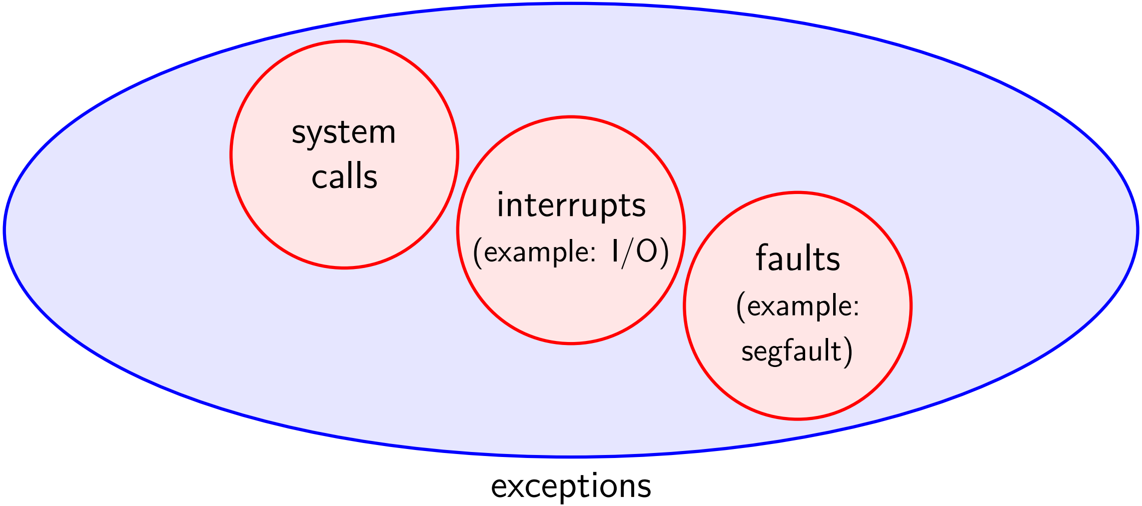
errors/events in programs
- memory not in address space ("Segmentation fault")
- privileged instruction
- divide by zero, invalid instruction
- ...

} **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

exceptions [Venn diagram]



time multiplexing

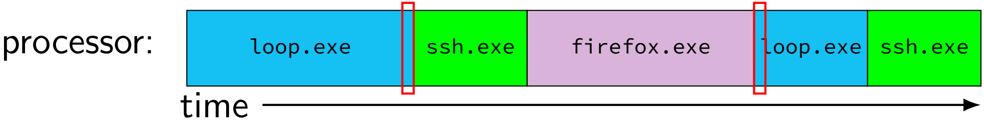
time multiplexing


```
...
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

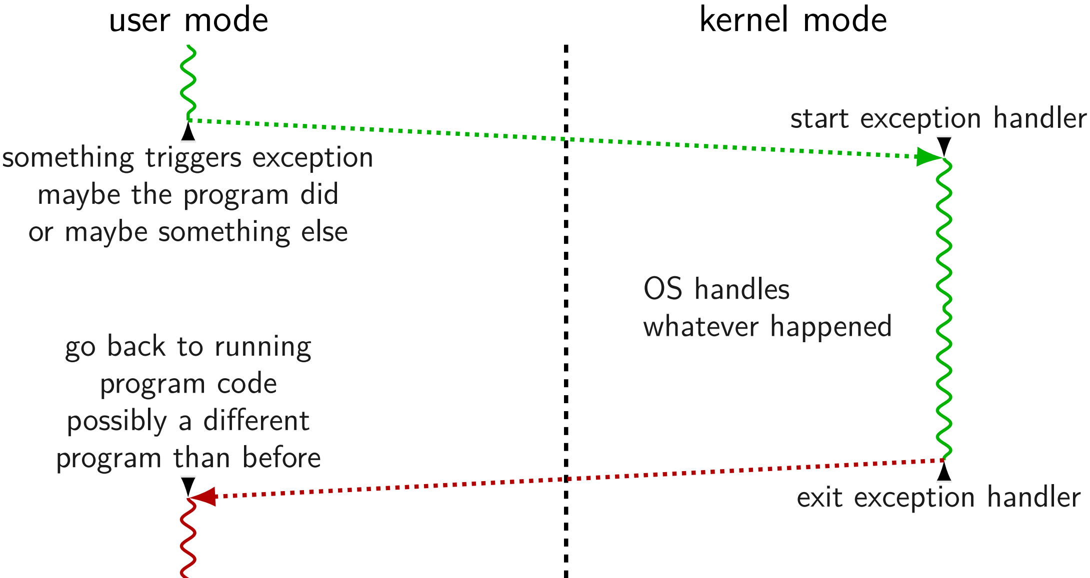



```
...
call get_time
    // whatever get_time does
movq %rax, %rbp
```

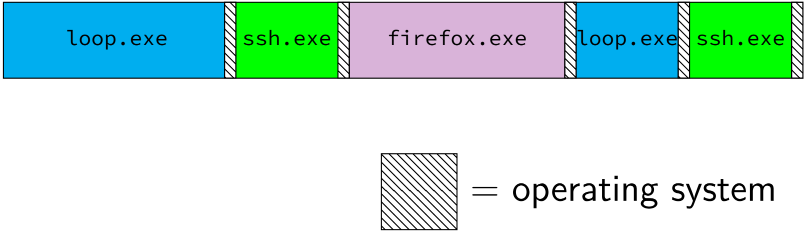
million cycle delay

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

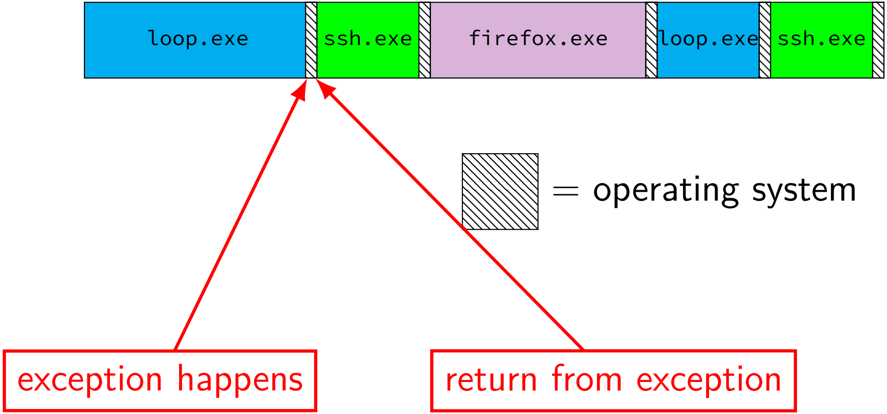
general exception process



time multiplexing really



time multiplexing really



switching programs

OS starts running somehow

- some sort of exception

saves old registers + program counter + address mapping

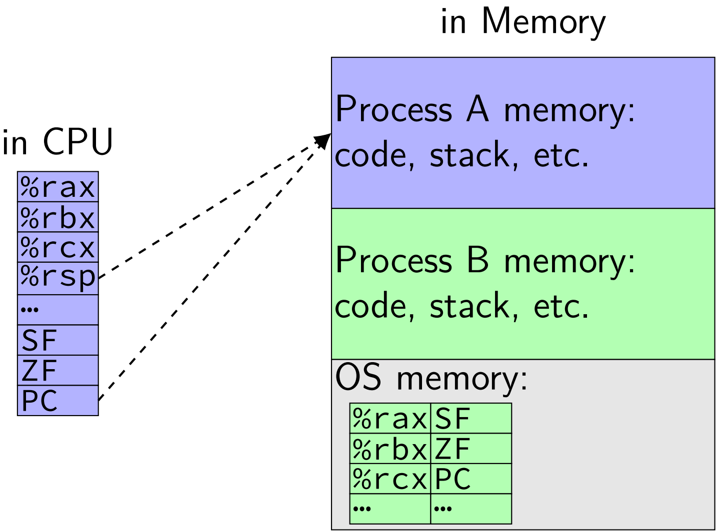
- (optimization: could omit when program crashing/exiting)

sets new registers + address mapping, jumps to new program counter

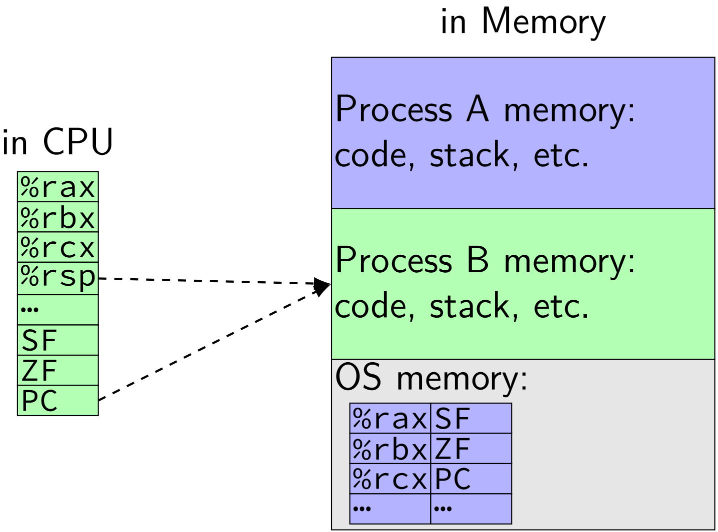
called context switch

- saved information called context

contexts (A running)



contexts (B running)



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?

The Process

process = thread(s) + address space

illusion of dedicated machine:

- thread = illusion of own CPU
- (process could have multiple threads — with independent registers)
- address space = illusion of own memory

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

exception patterns with I/O (1)

input — available now:

- exception: device says “I have input now”
- handler: OS stores input for later
- exception (syscall): program says “I want to read input”
- handler: OS returns that input

input — not available now:

- exception (syscall): program says “I want to read input”
- handler: OS runs other things (context switch)
- exception: device says “I have input now”
- handler: OS retrieves input
- handler: (possibly) OS switches back to program that wanted it

exception patterns with I/O (2)

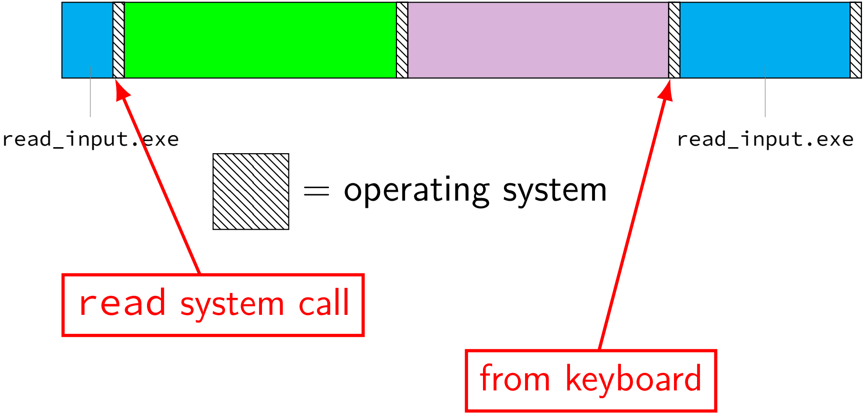
output — ready now:

- exception (syscall): program says "I want to output this'
- handler: OS sends output to deive

output — not ready now

- exception (syscall): program says "I want to output"
- handler: OS realizes device can't accept output yet
- (other things happen)
- exception: device says "I'm ready for output now"
- handler: OS sends output requested earlier

keyboard input timeline



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

which require exceptions [answers] (1)

A. program calls a function in the standard library

no (same as other functions in program; many standard library functions make no system calls (and do not otherwise trigger exceptions — for example `strlen`, `pow`; also if we consider the calling of a function just the `call` instruction, then the library functions that do make system calls won't do so until later)

B. program writes a file to disk

yes (requires kernel mode only operations)

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

yes (kernel mode usually required to change the address space to acess program B's memory)

which require exceptions [answer] (2)

D. program exits

yes (requires switching to another program, which requires accessing OS data + other program's memory)

E. program returns from one function to another function

no

F. program pops a value from the stack

no

which require context switches [answer]

no: A. program calls a function in the standard library

no: B. program writes a file to disk
(but might be done if program needs to wait for disk and other things could be run while it does)

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

yes: D. program exits

no: E. program returns from one function to another function

no: 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

backup slides