last time

makefiles — target: prereqs(newline)(tab)commands

- targets — files to generate/update
- prereqs — other files to use to do that

“phony” rules: targets that aren’t file

- e.g. “make clean” to remove generated

avoiding redundancy

- macros: CC=foo ... $(CC)
- suffix and pattern rules

anonymous feedback

“I’ve noticed some students have had their hands raised but they are not seen. Typically toward the top part of the room and the sides.”

“Please try to write more clearly, it can become difficult to read the handwriting. Thank you!”

“The C review was very helpful. I was wondering if you could go over memory allocation next class as well. I was also wondering when/ how you should allocate memory”

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?

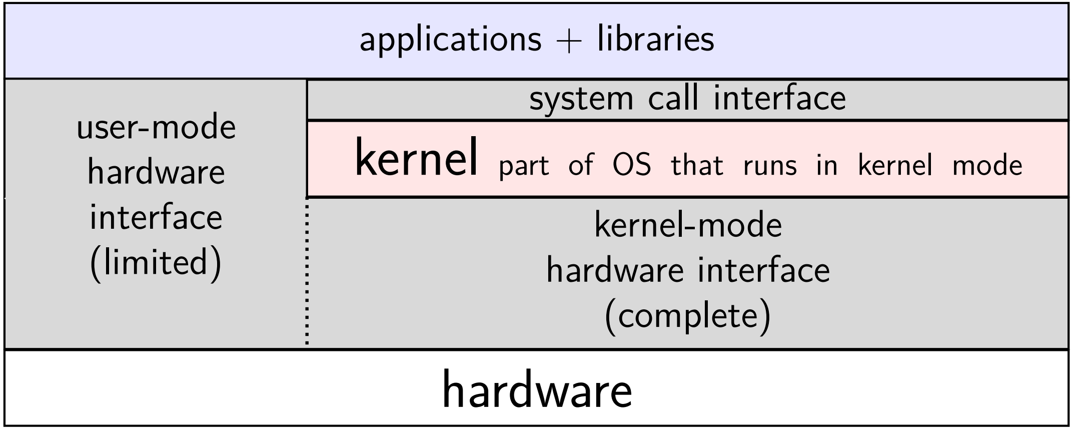
system boots in kernel mode

OS switches to user mode to run program code

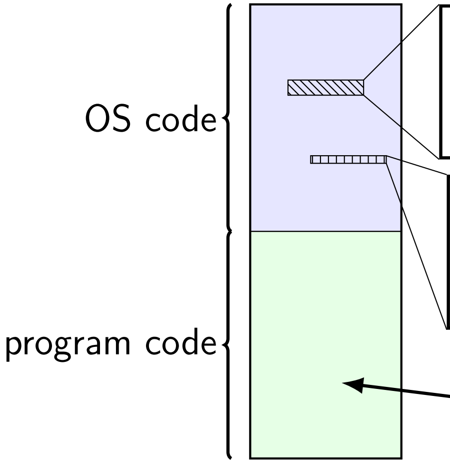
next topic: when does system switch back to kernel mode?

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

hardware + system call interface


applications + libraries
system call interface
user-mode hardware interface (limited)
kernel part of OS that runs in kernel mode
kernel-mode hardware interface (complete)
hardware

calling the OS?



```
void readFromDiskInto(int diskLocation, char *dest) {
    ...
    runPrivilegedInstruction(...);
    ...
}
```

```
void readFileSafely(const char *name, char *dest) {
    if (canCurrentProgramCanAccessFile(name)) {
        readFromDiskInto(lookupFile(name), dest)
    }
}
```

how do we let this code run `readFromSafely` in kernel mode but not `readFromDisk`?

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 privilieged instrution

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"],
        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

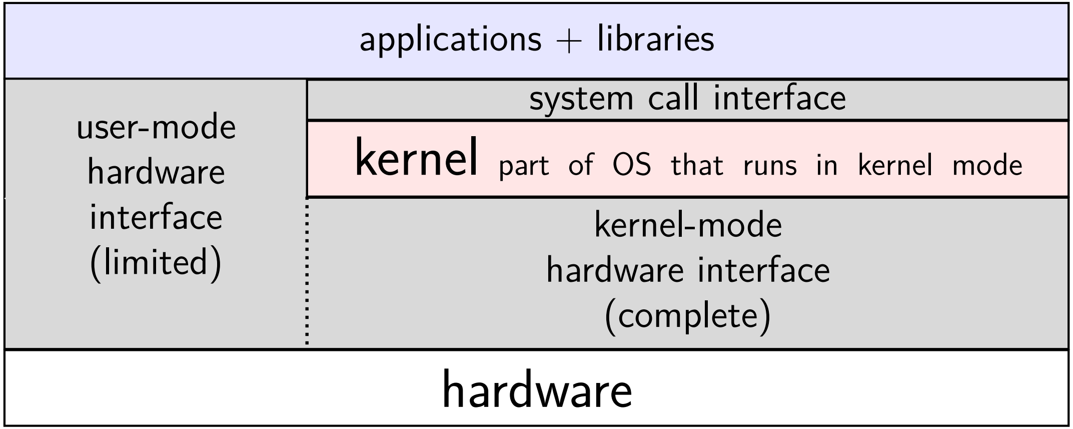
strace hello_world (2)

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

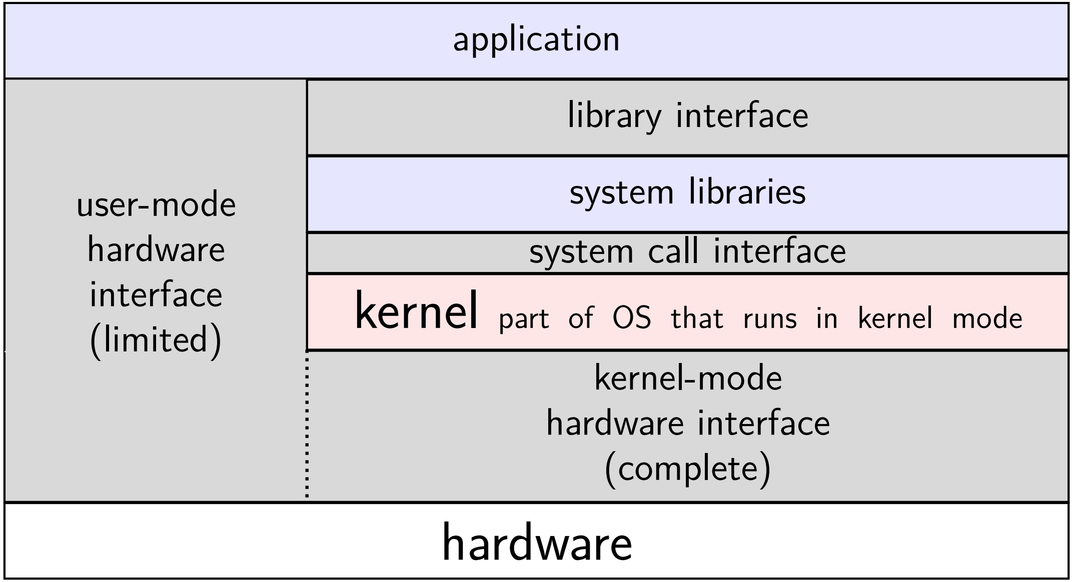
when dynamically linked:

```
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 +++
```

hardware + system call interface


applications + libraries
system call interface
user-mode hardware interface (limited)
kernel part of OS that runs in kernel mode
kernel-mode hardware interface (complete)
hardware

hardware + system call + library interface



application
library interface
system libraries
system call interface
kernel part of OS that runs in kernel mode
user-mode hardware interface (limited)
kernel-mode hardware interface (complete)
hardware

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

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

Program B

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

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

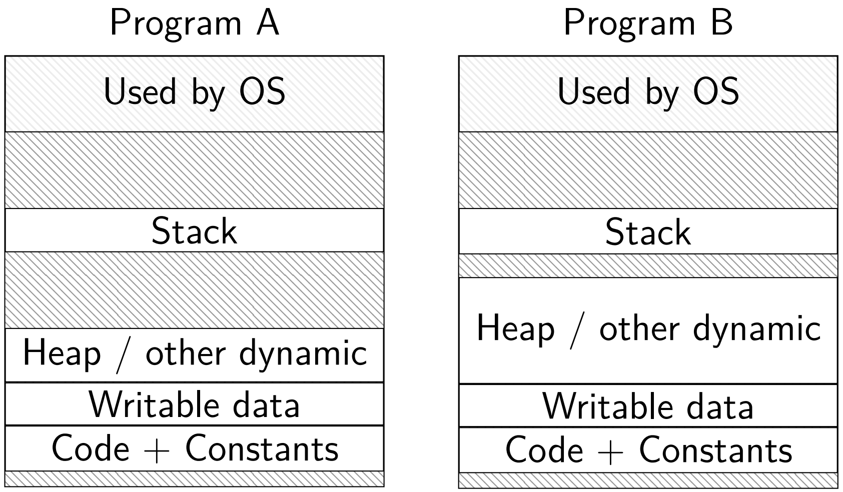
A. 42 B. 99 C. 0x10000

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

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

F. something else

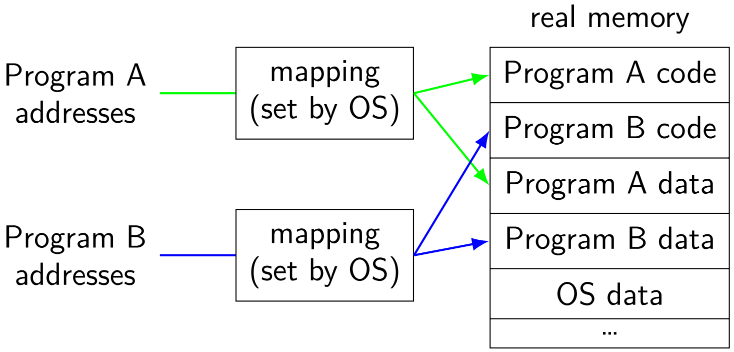
program memory (two programs)



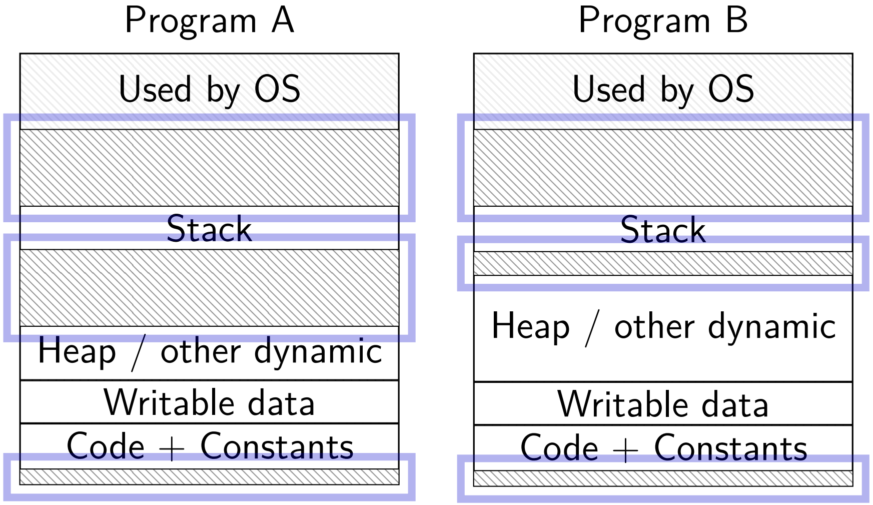
address space

programs have illusion of own memory

called a program's address space



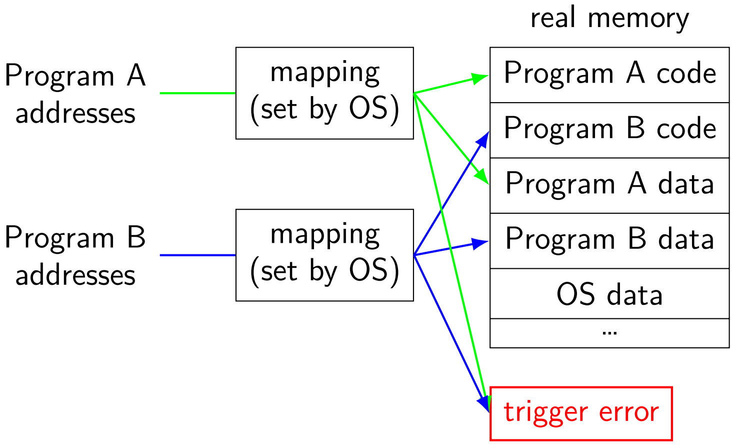
program memory (two programs)



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

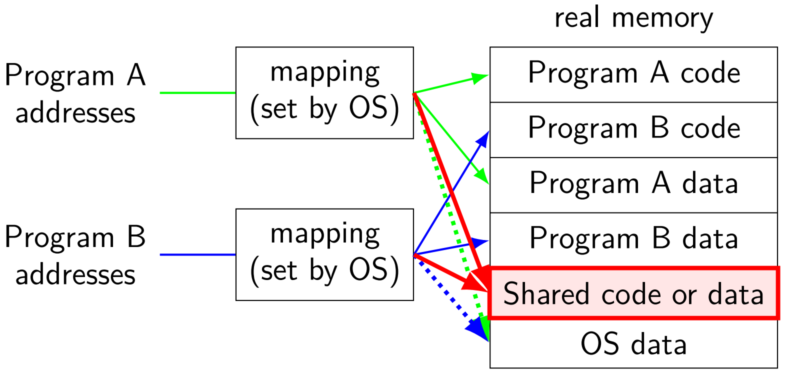
mapping part of what is changed in context switch

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

memory protection

modifying another program's memory?

Program A

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

result: `%rax` (in A) is 42 (always)

Program B

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

result: might crash

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

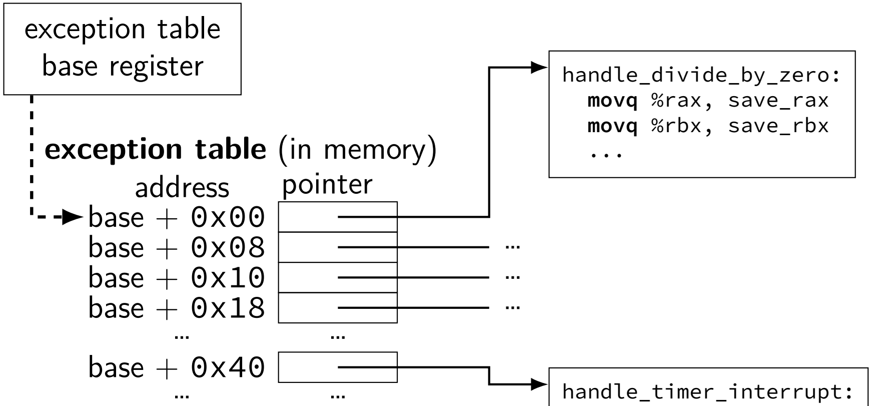
also cases where hardware asks OS for help

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)



```
handle_divide_by_zero:
  movq %rax, save_rax
  movq %rbx, save_rbx
  ...
```

```
handle_timer_interrupt:
  movq %rax, save_rax
  movq %rbx, save_rbx
  ...
```

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

$$\left.\begin{array}{l}\text{system calls}\\ \quad\text{intentional — ask OS to do something}\\ \text{errors/events in programs}\\ \quad\text{memory not in address space (“Segmentation fault”)}\\ \quad\text{privileged instruction}\\ \quad\text{divide by zero, invalid instruction}\end{array}\right\}\text{synchronous}$$

triggered by current program

- ...

(and more we'll talk about later)

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?

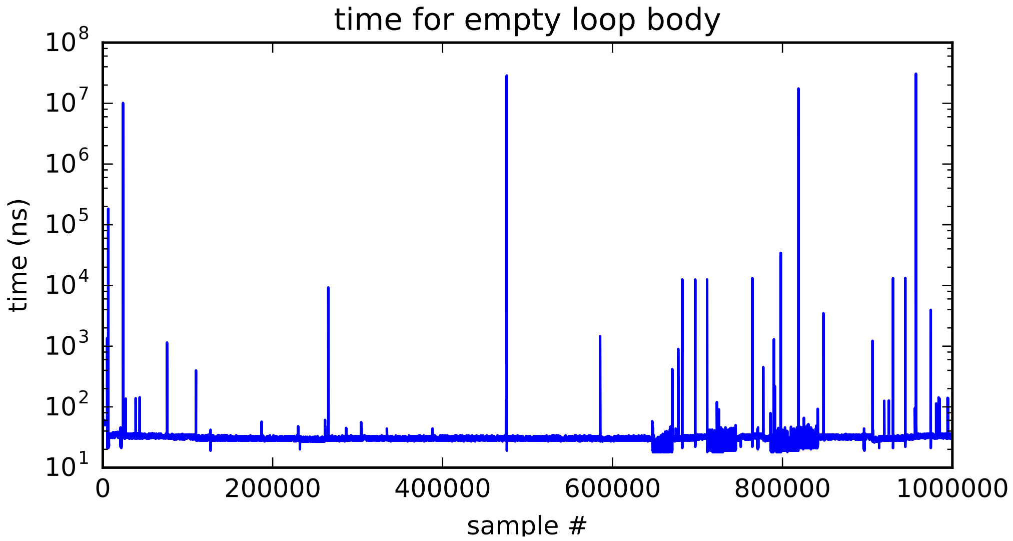
...if the machine only has one core?

timing nothing

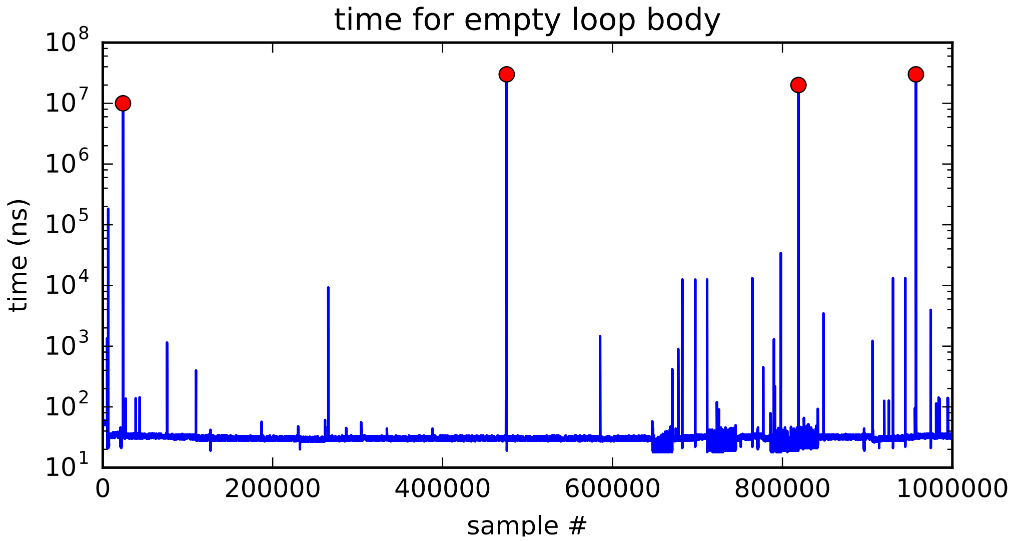
```
long times[NUM_TIMINGS];
int main(void) {
    for (int i = 0; i < N; ++i) {
        long start, end;
        start = get_time();
        /* do nothing */
        end = get_time();
        times[i] = end - start;
    }
    output_timings(times);
}
```

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
- ...

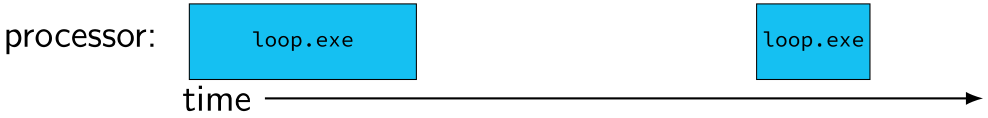
$\left.\right\}$ synchronous
triggered by current program

external — I/O, etc.

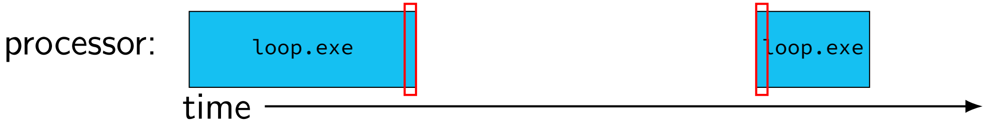
- timer — configured by OS to run OS at certain time
- I/O devices — key presses, hard drives, networks, ...
- hardware is broken (e.g. memory parity error)

$\left.\right\}$ asynchronous
not triggered by running program

time multiplexing



time multiplexing

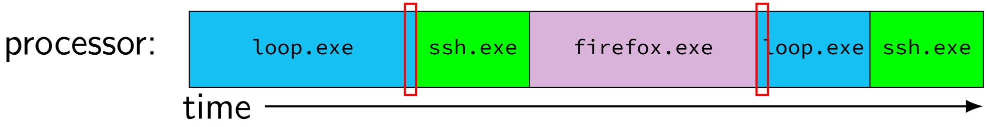


```
...
call get_time
    // whatever get_time does
movq %rax, %rbp
```

million cycle delay

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

time multiplexing

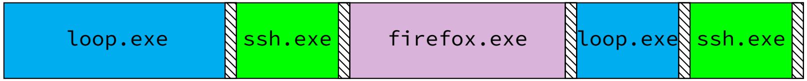


```
...
call get_time
    // whatever get_time does
movq %rax, %rbp
```

million cycle delay

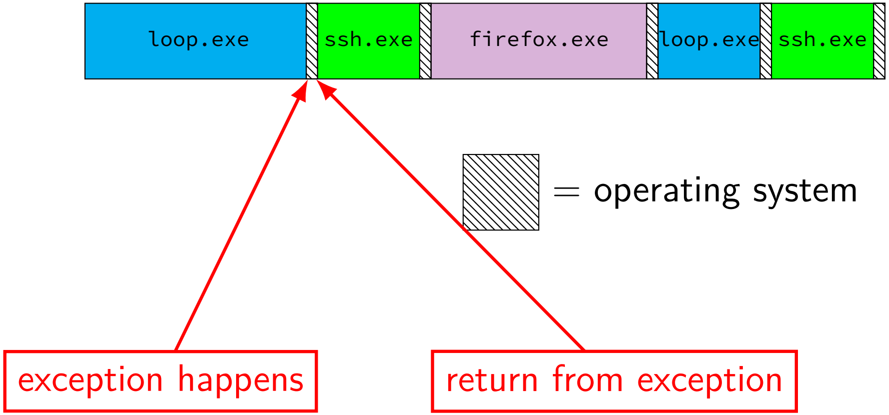
```
call get_time
    // whatever get_time does
subq %rbp, %rax
...
```

time multiplexing really



▨ = operating system

time multiplexing really



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
- ...

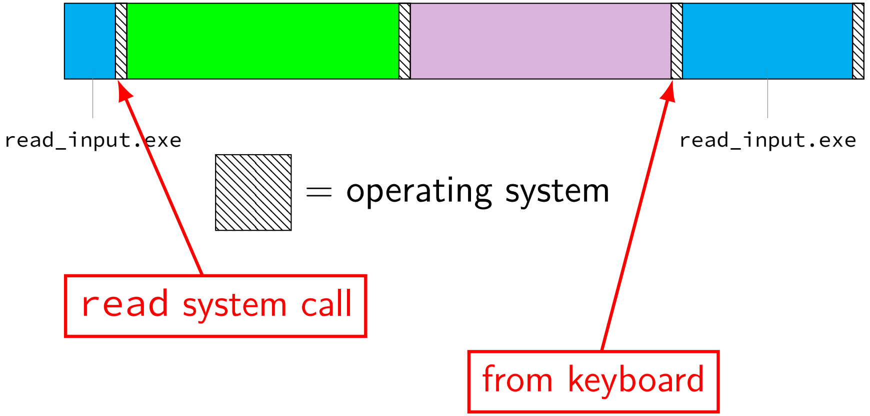
synchronous
triggered by
current program

external — I/O, etc.

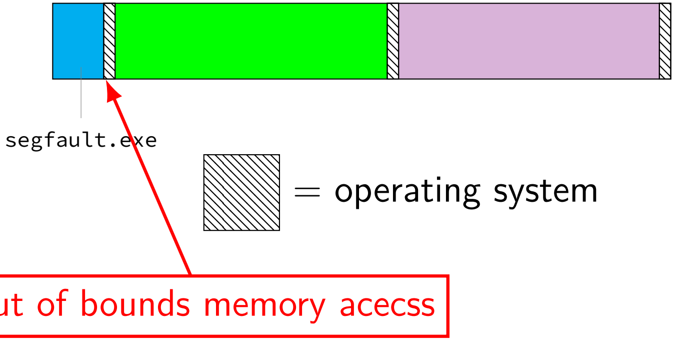
- timer — configured by OS to run OS at certain time
- I/O devices — key presses, hard drives, networks, ...
- hardware is broken (e.g. memory parity error)

asynchronous
not triggered by
running program

keyboard input timeline



crash timeline timeline



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

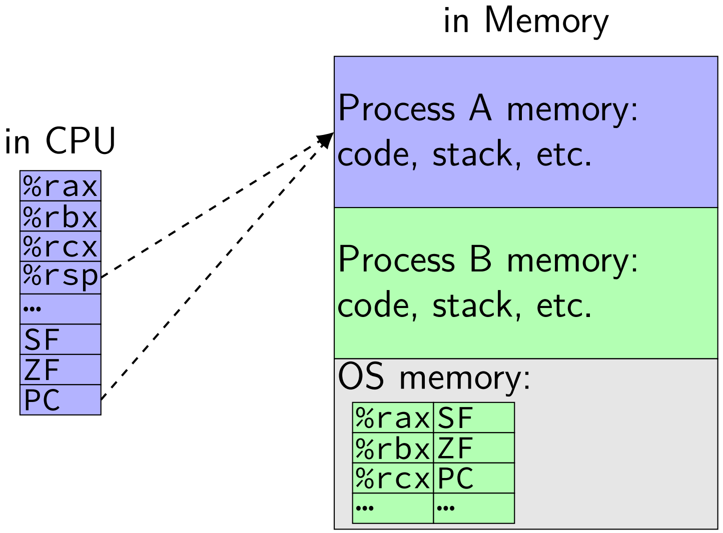
OS starts running somehow
- some sort of exception

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

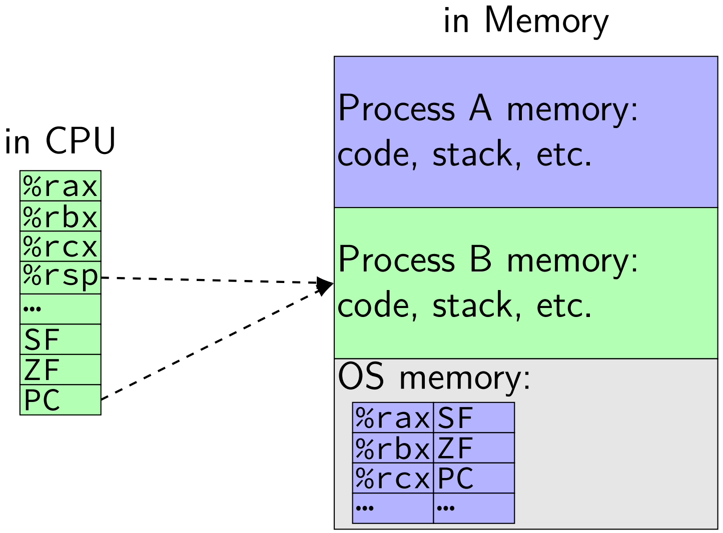
sets new registers, jumps to new program counter

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

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

```c
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 = SA_RESTART;
    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 = SA_RESTART;
    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 = SA_RESTART;
    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 = SA_RESTART;
    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 = SA_RESTART;
    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/unblock some signals from being received

> signal will still be *pending*, received if unblocked

… and much more

kill command

kill command-line command : calls the kill() function

`kill 1234` — sends SIGTERM to pid 1234

`kill -USR1 1234` — sends SIGUSR1 to pid 1234

SA_RESTART

`sa.sa_flags = SA_RESTART;`

- general version:
  `sa.sa_flags = SA_NAME | SA_NAME | SA_NAME;` (or `0`)

if SA_RESTART included:

- after signal handler runs, attempt to restart interrupted operations (e.g. reading from keyboard)

if SA_RESTART not included:

- after signal handler runs, interrupted operations return typically an error (errno == EINTR)

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, NULL);
    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, NULL);
}
```

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

output of this? (v2)

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);
    while (1) pause();
}
```

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);
    while (1) pause();
}
```

If these run at same time, expected output?

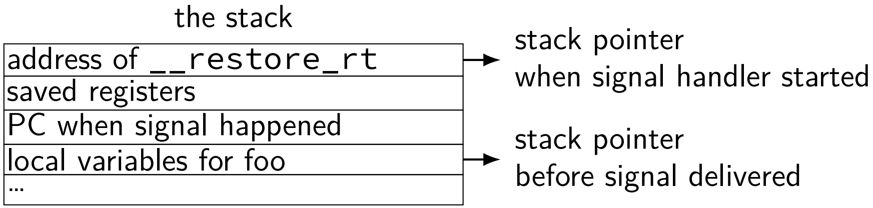
A. XY
B. X
C. Y
D. YX
E. X or XY, depending on timing
F. crash
G. (nothing)
H. something else

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
- if SA_RESTART set, restarts interrupted operation
- also handles caller-saved registers
- also might change which signals blocked (depending how sigaction was called)

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");
}
```

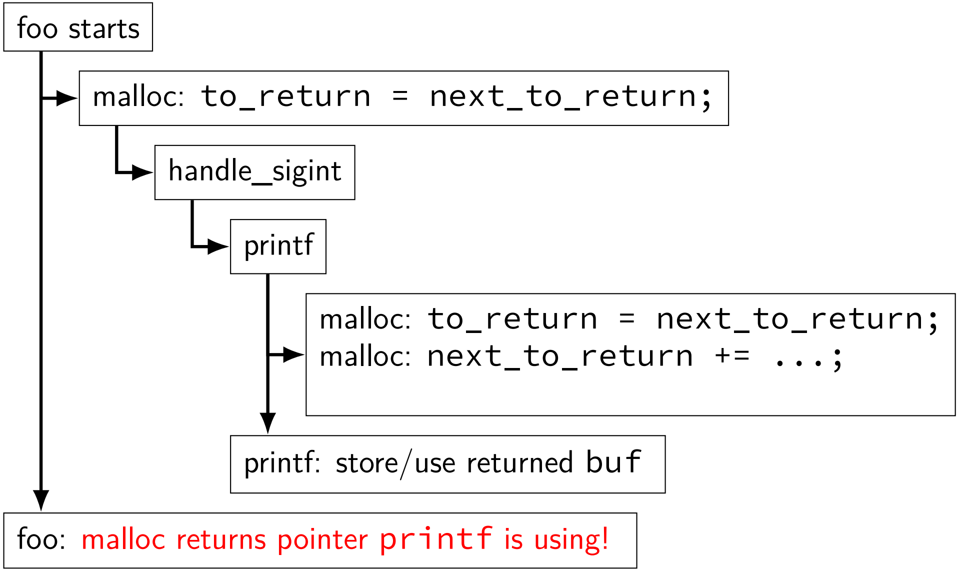
signal handler unsafety (1)

```
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


foo starts
malloc: `to_return = next_to_return;`
handle_sigint
printf
malloc: `to_return = next_to_return;`
malloc: `next_to_return += ...;`
printf: store/use returned `buf`
foo: malloc returns pointer `printf` is using!

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

can be done with `sigprocmask` or `pthread_sigmask`

signal will become "pending" instead

OS will not deliver unless unblocked

- similar mechanism provided by CPU for interrupts ("disabling interrupts")

controlling when signals are handled

first, block a signal

then use system calls to inspect pending signals
- example: `sigwait`

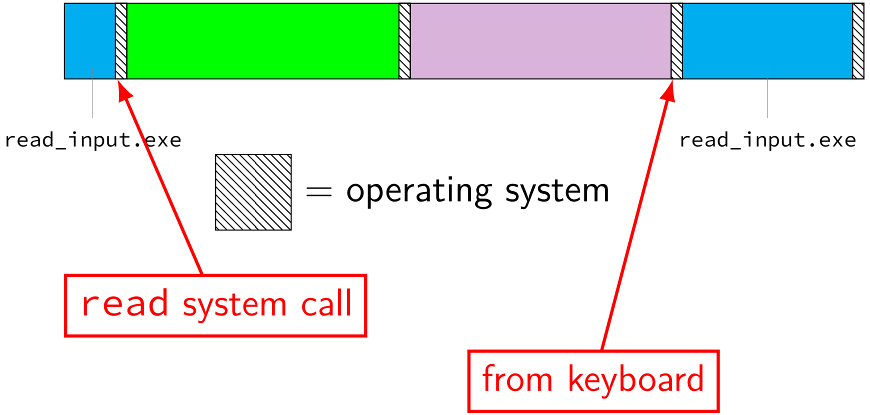
and/or unblock signals only at certain times
- some special functions to help:
  `sigsuspend` (unblock until handler runs),
  `pselect` (unblock while checking for I/O), …

synchronous signal handling

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

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

backup slides

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
```

context

all registers values

    `%rax` `%rbx`, …, `%rsp`, …

condition codes

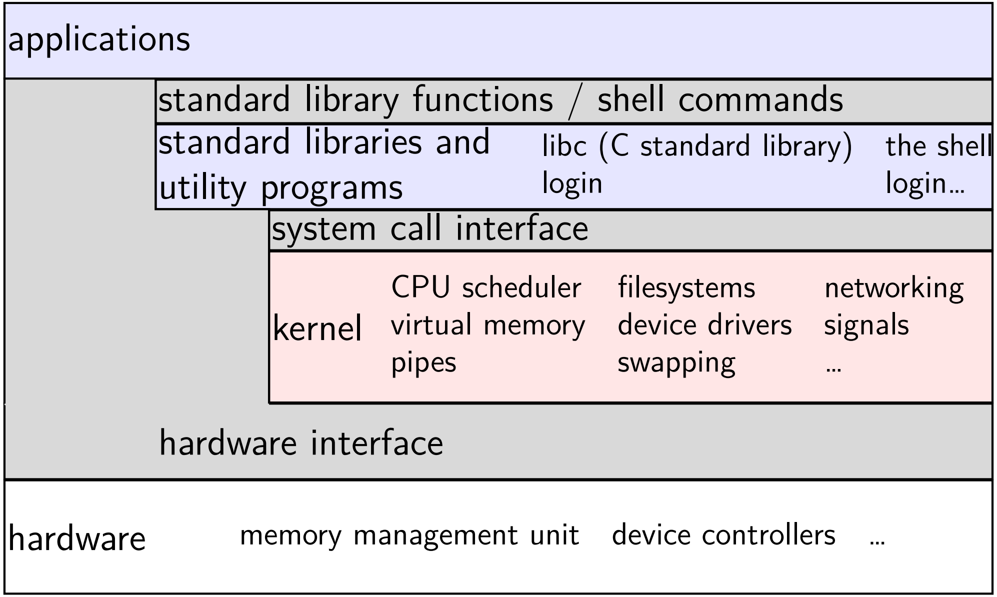
program counter

address space (map from program to real addresses)

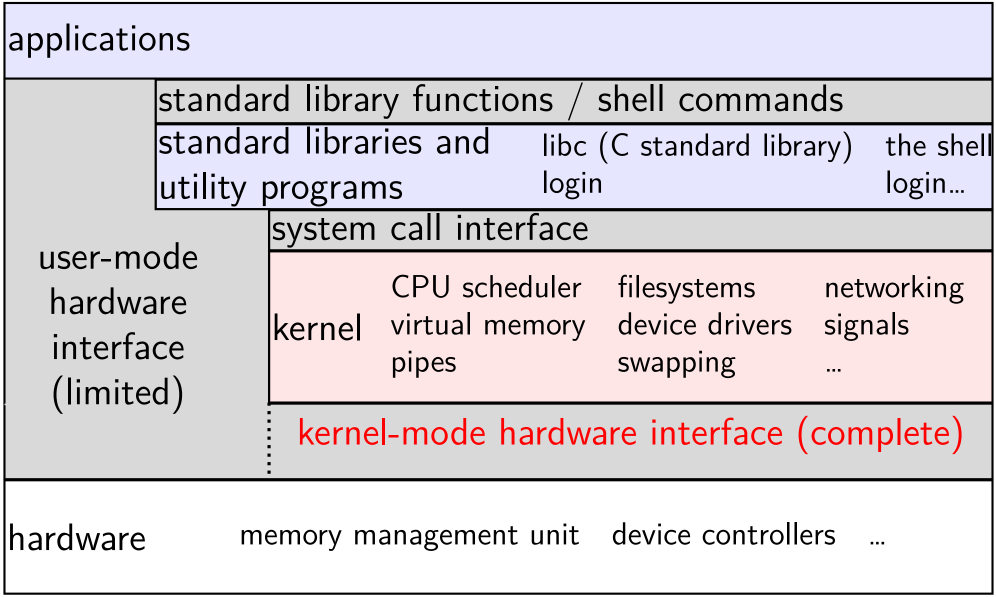
context switch pseudocode

```
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
```

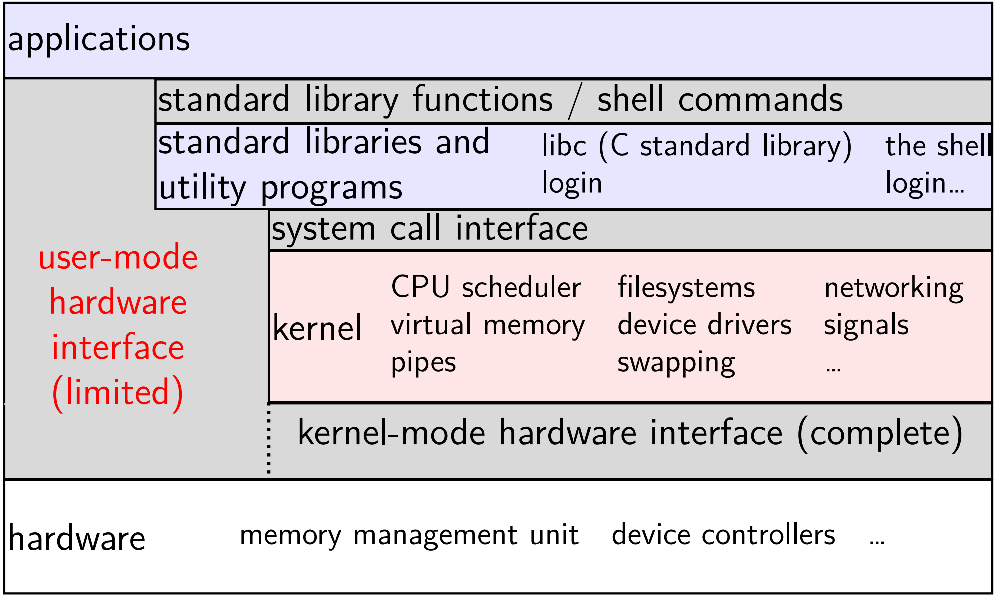
the classic Unix design



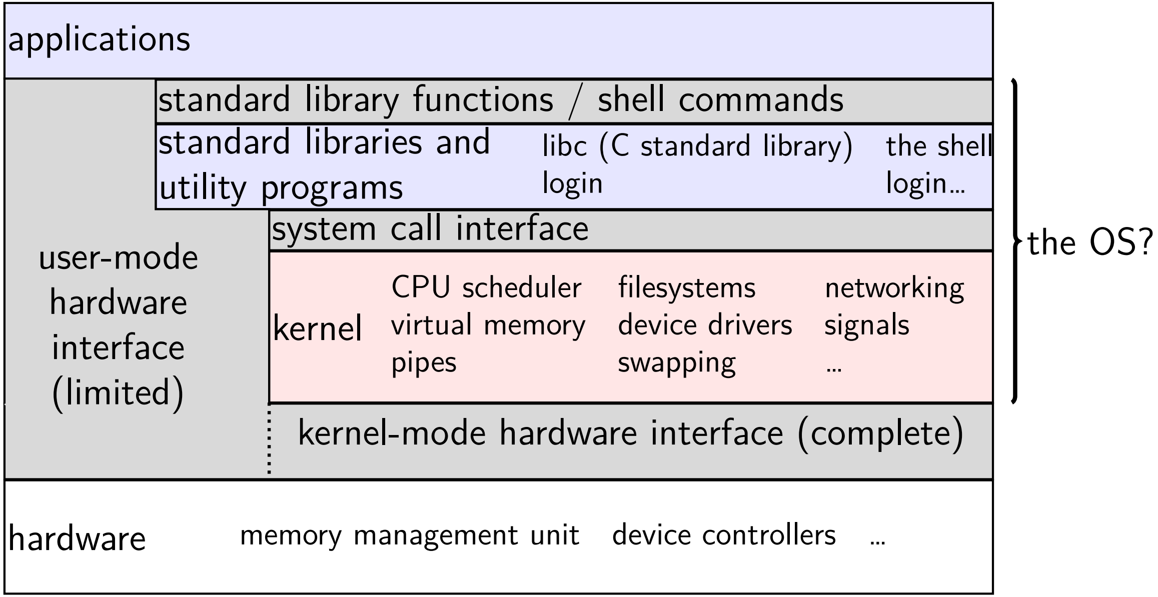
the classic Unix design



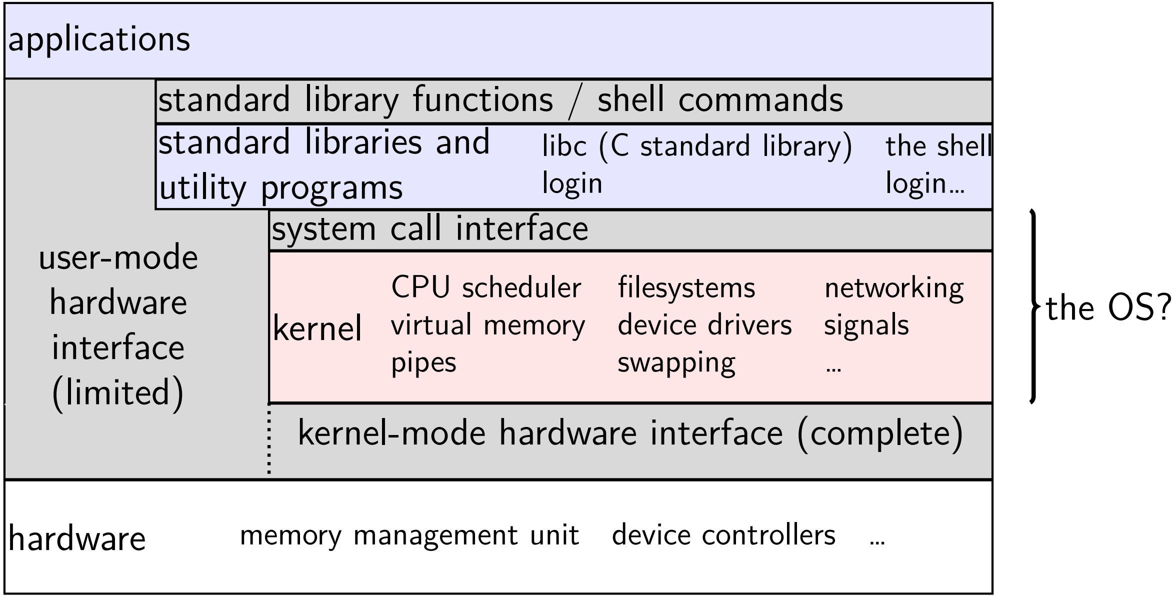
the classic Unix design



the classic Unix design



the classic Unix design



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…

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)

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.