

things programs on portal shouldn't do

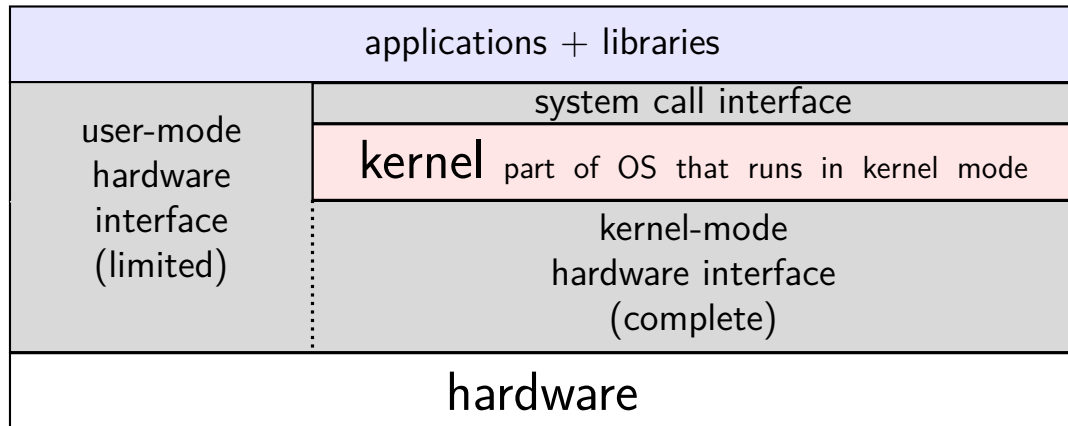
read other user's files

modify OS's memory

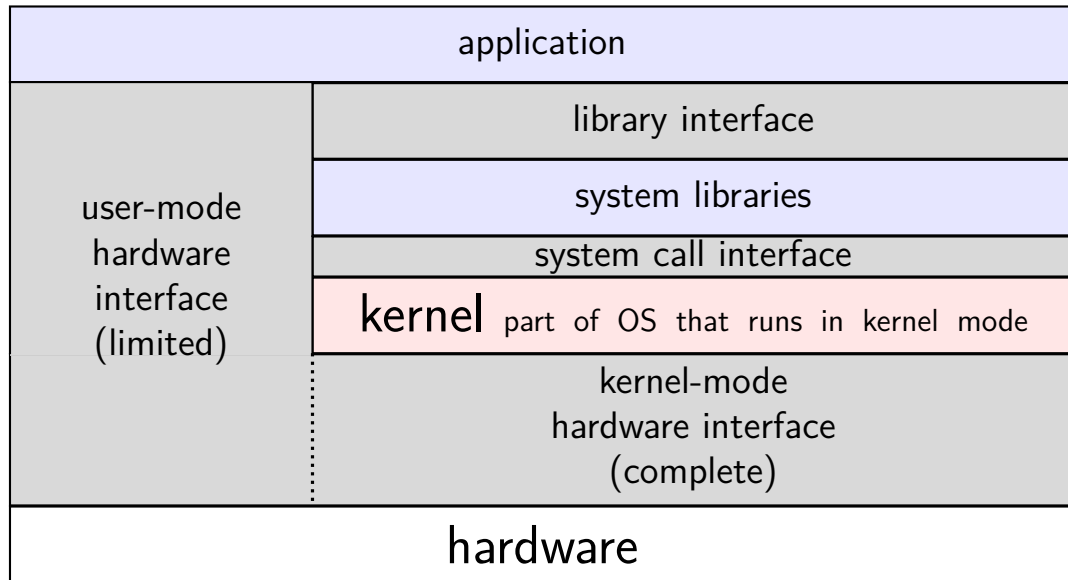
read other user's data in memory

hang the entire system

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>

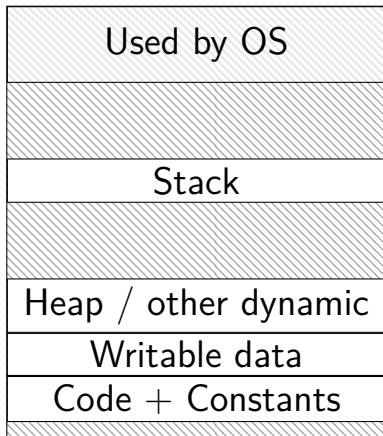
memory protection

modifying another program's memory?

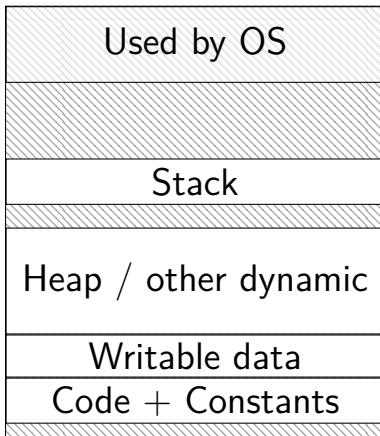
Program A	Program B
<pre>0x10000: .long 42 // ... // do work // ... movq 0x10000, %rax</pre>	<pre><i>// while A is working:</i> movq \$99, %rax movq %rax, 0x10000 ...</pre>
<p>result: %rax (in A) is ...</p> <p>A. 42 B. 99 C. 0x10000</p> <p>D. 42 or 99 (depending on timing/program layout/etc)</p> <p>E.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

program memory (two programs)

Program A



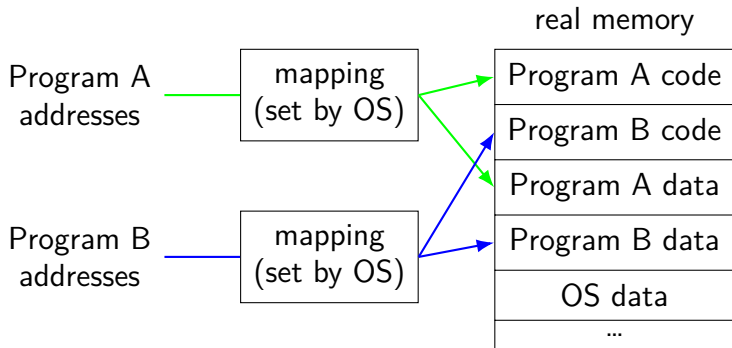
Program B



address space

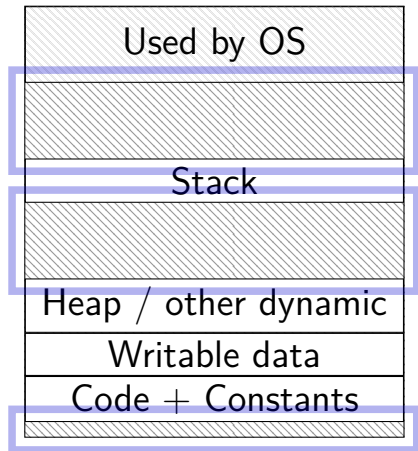
programs have **illusion of own memory**

called a program's **address space**

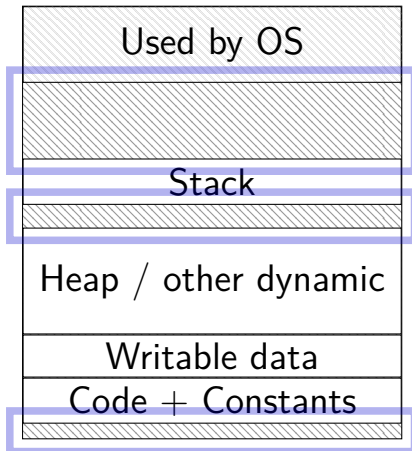


program memory (two programs)

Program A



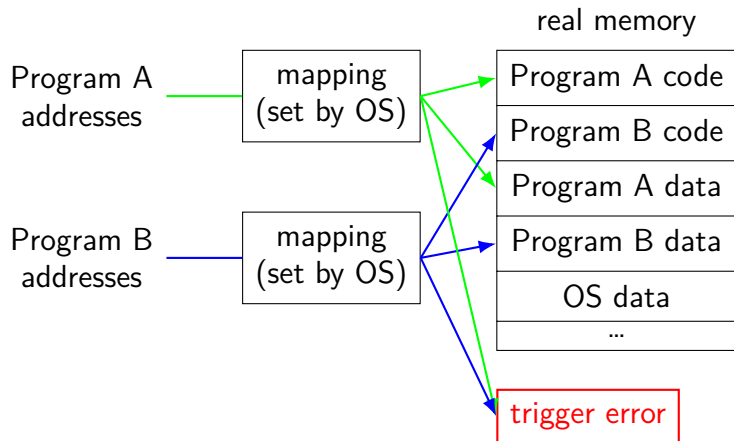
Program B



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**

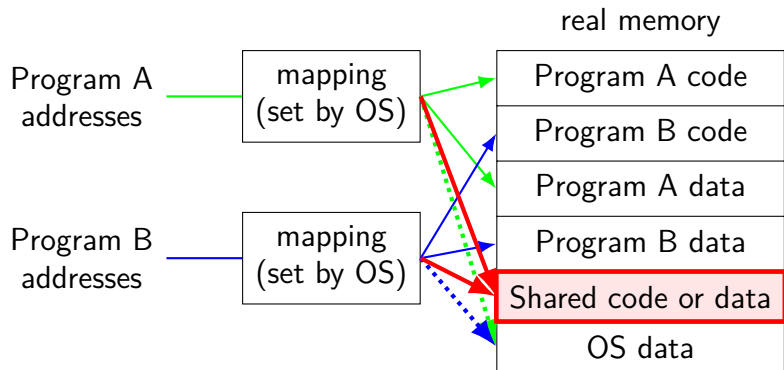
mapping part of what is changed in context switch

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) 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	result: might crash

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 */  
    }  
}
```

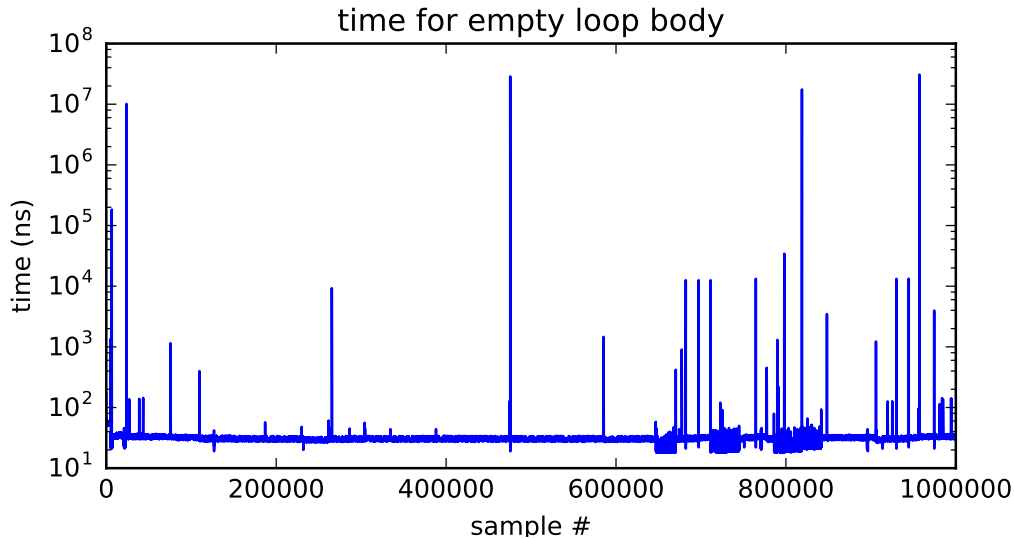
If I run this on a shared department machine, can you still use it?
...if the machine only has one core?

timing nothing

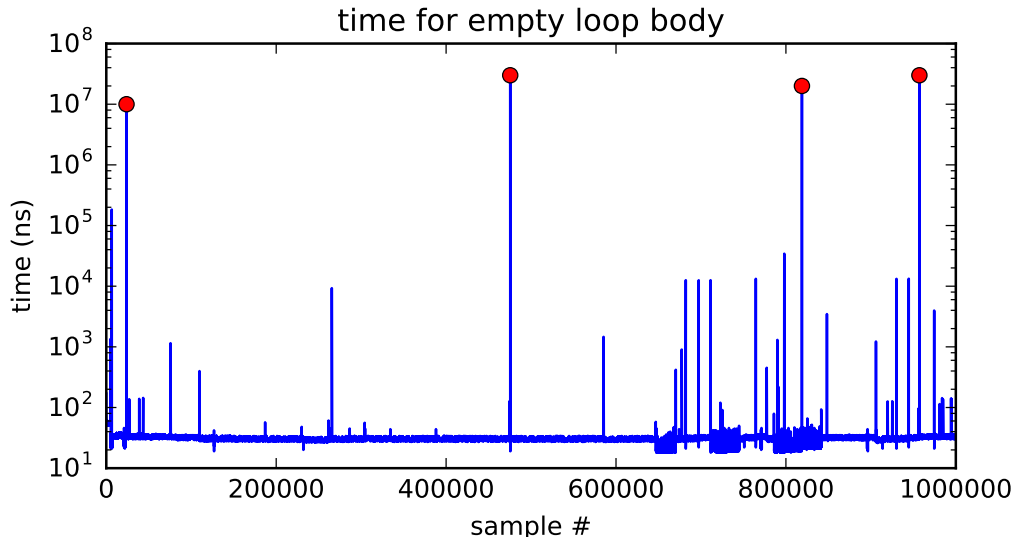
```
long times[NUM_TIMINGS];  
int main(void) {  
    for (int i = 0; i < N; ++i) {  
        long start, end;  
        start = get_time();  
        /* do nothing */  
        end = get_time();  
        times[i] = end - start;  
    }  
    output_timings(times);  
}
```

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

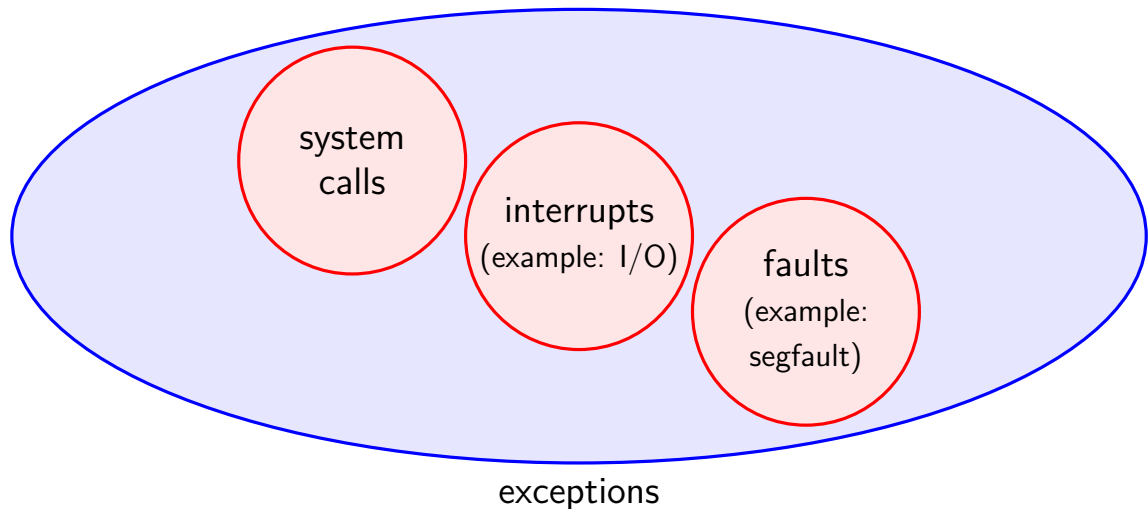
I/O devices — key presses, hard drives, networks, ...

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

asynchronous

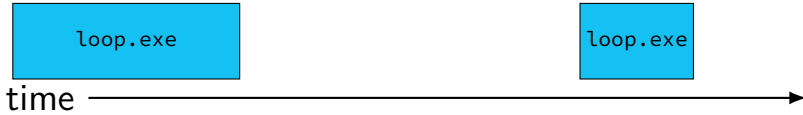
not triggered by
running program

exceptions [Venn diagram]



time multiplexing

processor:



time multiplexing



...

```
call get_time
```

```
// whatever get_time does
```

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

———— million cycle delay ————

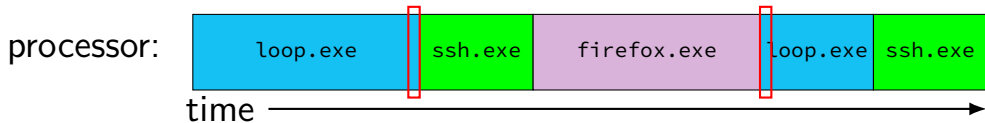
```
call get_time
```

```
// whatever get_time does
```

```
subq %rbp, %rax
```

...

time multiplexing



...

```
call get_time
```

```
// whatever get_time does
```

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

———— million cycle delay ————

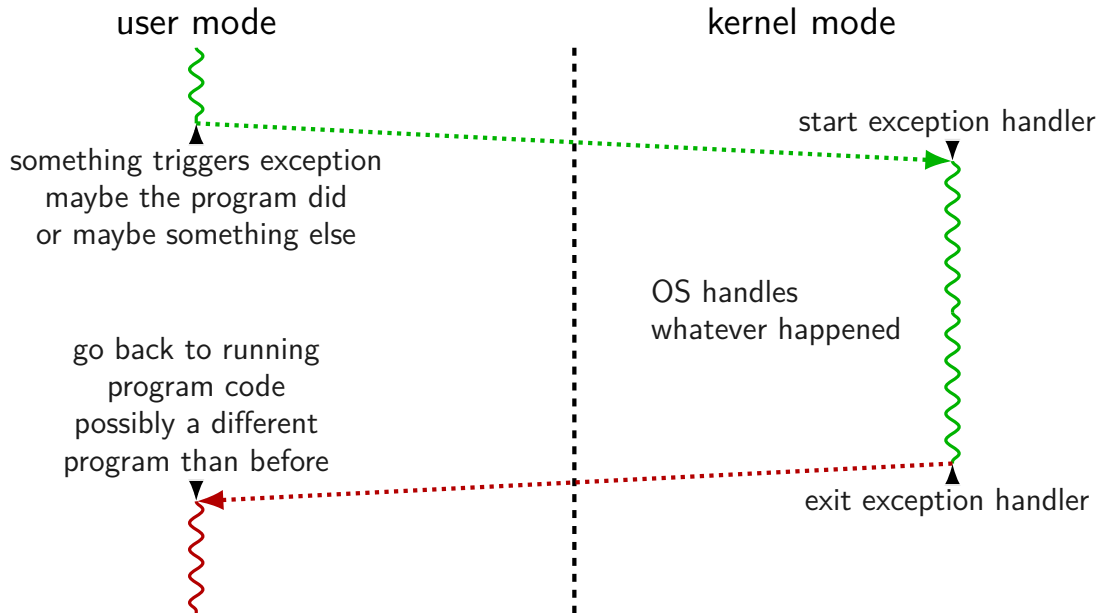
```
call get_time
```

```
// whatever get_time does
```

```
subq %rbp, %rax
```

...

general exception process

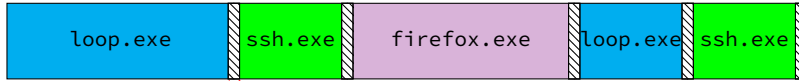


time multiplexing really



= operating system

time multiplexing really



= operating system

exception happens

return from exception

switching programs

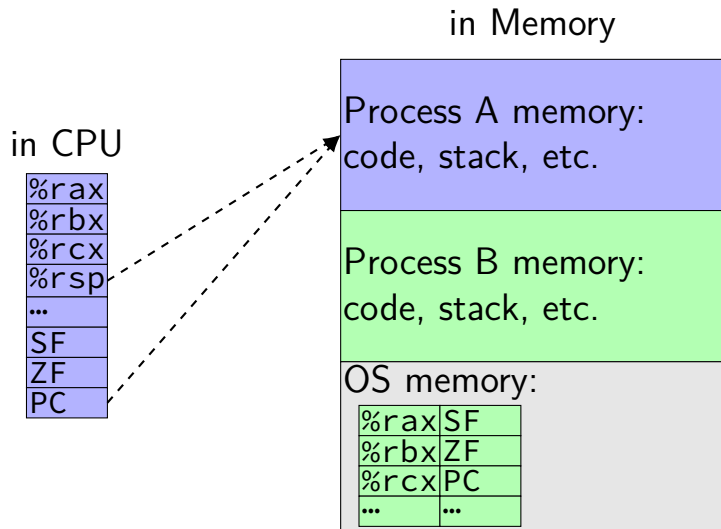
OS starts running somehow
some sort of exception

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

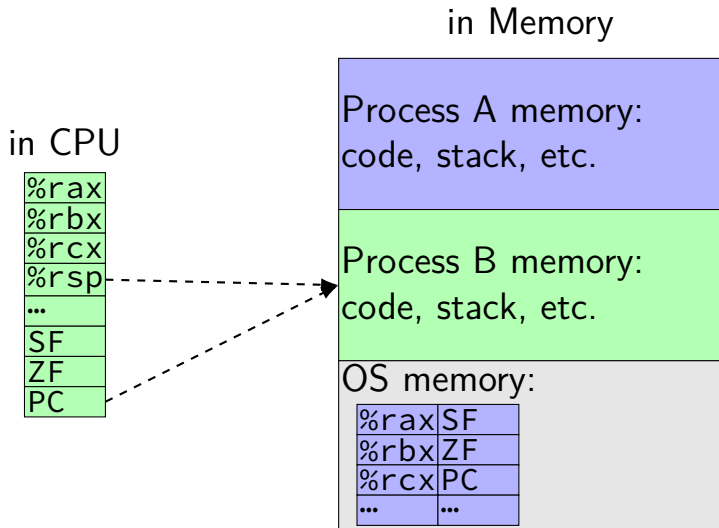
sets new registers, jumps to new program counter

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

contexts (A running)



contexts (B running)



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?

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

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 device

output — not ready now

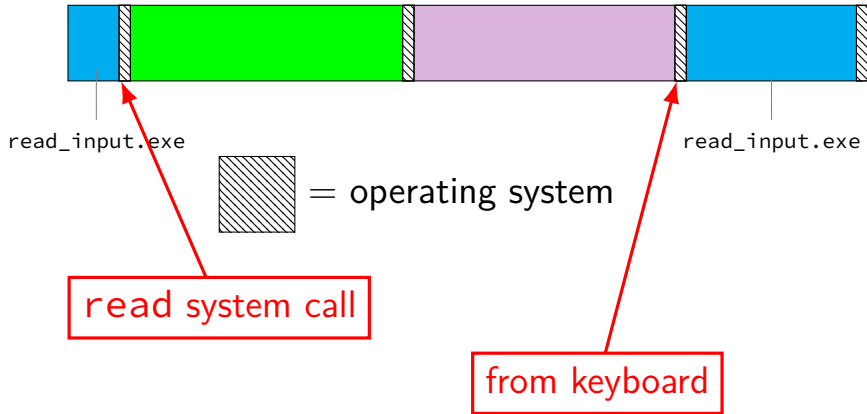
exception (syscall): program says “I want to output”

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

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

handler runs in kernel mode

hardware decides when

hardware needs to save PC

processor next instruction changes

signals

handler runs in user mode

OS decides when

OS needs to save PC + registers

thread next instruction changes

exceptions v signals

(hardware) exceptions

handler runs in kernel mode

hardware decides when

hardware needs to save PC

processor next instruction changes

signals

handler runs in user mode

OS decides when

OS needs to save PC + registers

thread next instruction changes

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

exceptions v signals

(hardware) exceptions

handler runs in kernel mode

hardware decides when

hardware needs to save PC

processor next instruction changes

signals

handler runs in user mode

OS decides when

OS needs to save PC + registers

thread next instruction changes

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

exceptions v signals

(hardware) exceptions

handler runs in kernel mode

hardware decides when

hardware needs to save PC

processor next instruction changes

signals

handler runs in user mode

OS decides when

OS needs to save PC + registers

thread next instruction changes

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

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

base program

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

base program

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

some input

read some input

more input

read more input

(control-C pressed)

(program terminates immediately)

base program

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

some input

read some input

more input

read more input

(control-C pressed)

(program terminates immediately)

new program

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

some input

read some input

more input

read more input

(control-C pressed)

Control-C pressed?!

another input **read another input**

new program

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

some input

read some input

more input

read more input

(control-C pressed)

Control-C pressed?!

another input **read another input**

new program

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

some input

read some input

more input

read more input

(control-C pressed)

Control-C pressed?!

another input **read another input**

example signal program

```
void handle_sigint(int signum) {  
    /* signum == SIGINT */  
    write(1, "Control-C pressed?!\n",  
        sizeof("Control-C pressed?!\n"));  
}  
  
int main(void) {  
    struct sigaction act;  
    act.sa_handler = &handle_sigint;  
    sigemptyset(&act.sa_mask);  
    act.sa_flags = SA_RESTART;  
    sigaction(SIGINT, &act, NULL);  
  
    char buf[1024];  
    while (fgets(buf, sizeof buf, stdin)) {  
        printf("read %s", buf);  
    }  
}
```

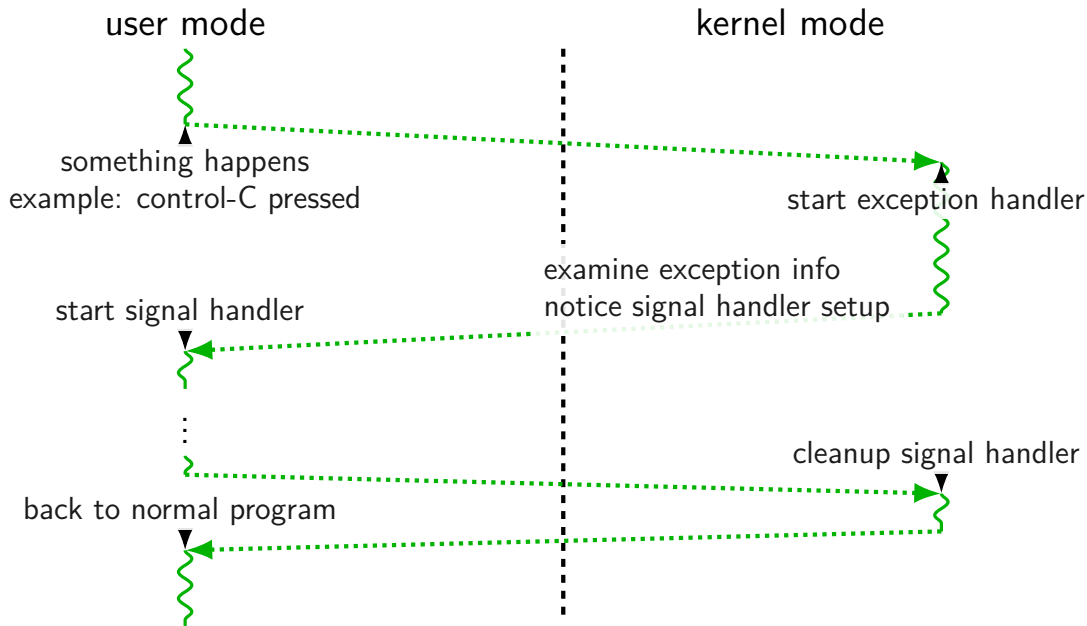
example signal program

```
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);  
    }  
}
```

'forwarding' exception as signal



SIGxxx

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  
+ SIGSEGV
```

signal API

`sigaction` — register handler for signal

`kill` — send signal to process

uses **process ID** (integer, retrieve from `getpid()`)

`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

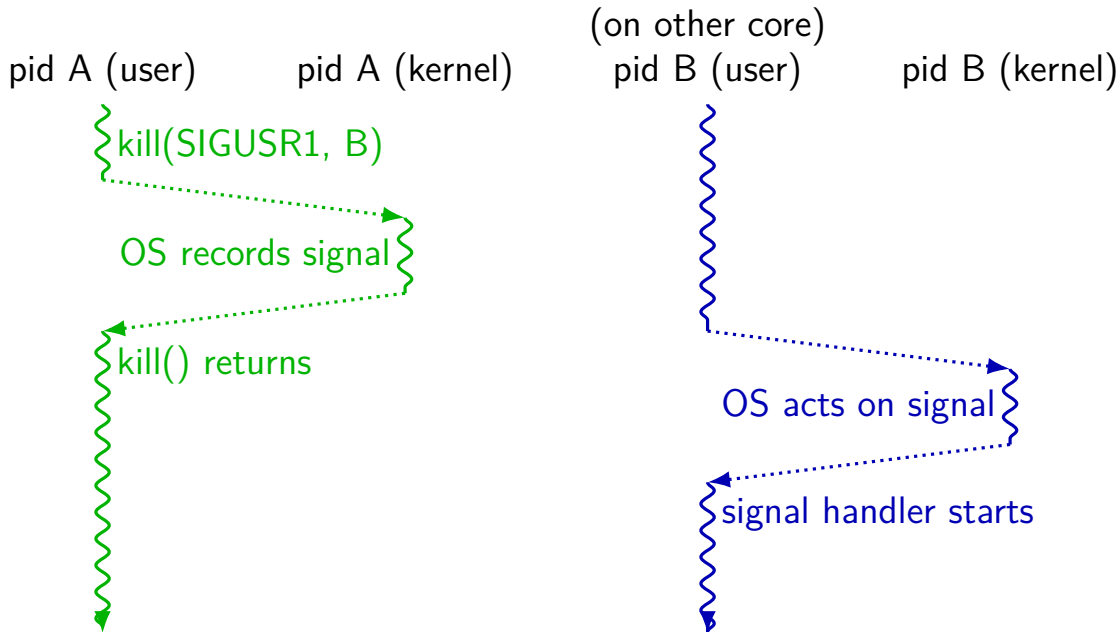
`kill 1234` — sends SIGTERM to pid 1234

in C: `kill(1234, SIGTERM)`

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

in C: `kill(1234, SIGUSR1)`

kill() not always immediate



SA_RESTART

```
struct sigaction sa; ...  
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_usr1(int num) {
    write(1, "X", 1);
    kill(2000, SIGUSR1);
    _exit(0);
}

int main() {
    struct sigaction act;
    act.sa_handler = &handle_usr1;
    sigaction(SIGUSR1, &act, NULL);
    kill(1000, SIGUSR1);
}
```

pid 2000

```
void handle_usr1(int num) {
    write(1, "Y", 1);
    _exit(0);
}

int main() {
    struct sigaction act;
    act.sa_handler = &handle_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_usr1(int num) {
    write(1, "X", 1);
    kill(2000, SIGUSR1);
    _exit(0);
}

int main() {
    struct sigaction act;
    act.sa_handler = &handle_usr1;
    sigaction(SIGUSR1, &act);
    kill(1000, SIGUSR1);
    while (1) pause();
}
```

pid 2000

```
void handle_usr1(int num) {
    write(1, "Y", 1);
    _exit(0);
}

int main() {
    struct sigaction act;
    act.sa_handler = &handle_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

sending signals (1)

pid 1000

```
void handle_usr1(int num) {
    write(1, "Y", 1);
    kill(2000, SIGUSR2);
}

int main() {
    struct sigaction act;
    ... // initialize act
    act.sa_handler = &handle_usr1;
    sigaction(SIGUSR1, &act, NULL);
    sleep(60); // wait for pid 2000 to start
    kill(2000, SIGUSR1);
    while (1) pause();
}
```

sending signals (2)

pid 1000

```
void handle_usr1(int num) {
    write(1, "Y", 1);
    kill(2000, SIGUSR2);
}

int main() {
    struct sigaction act;
    ... // initialize act
    act.sa_handler = &handle_usr1;
    sigaction(SIGUSR1, &act, NULL);
    sleep(60); // wait for pid 2000 to start
    kill(2000, SIGUSR1);
    while (1) pause();
}
```

pid 2000

```
void handle_usr1(int num) {
    write(1, "X", 1);
    kill(1000, SIGUSR1);
}

void handle_usr2(int num) {
    write(1, "Z", 1);
    kill(1000, SIGTERM);
    _exit(0);
}

int main() {
    struct sigaction act;
    ... // initialize act
    act.sa_handler = &handle_usr1;
    sigaction(SIGUSR1, &act, NULL);
    act.sa_handler = &handle_usr2;
    sigaction(SIGUSR2, &act, NULL);
    while (1) pause();
}
```

sending signals (2)

pid 1000

```
void handle_usr1(int num) {
    write(1, "Y", 1);
    kill(2000, SIGUSR2);
}

int main() {
    struct sigaction act;
    ... // initialize act
    act.sa_handler = &handle_usr1;
    sigaction(SIGUSR1, &act, NULL);
    sleep(60); // wait for pid 2000 to start
    kill(2000, SIGUSR1);
    while (1) pause();
}
```

pid 2000

```
void handle_usr1(int num) {
    write(1, "X", 1);
    kill(1000, SIGUSR1);
}

void handle_usr2(int num) {
    write(1, "Z", 1);
    kill(1000, SIGTERM);
    _exit(0);
}

int main() {
    struct sigaction act;
    ... // initialize act
    act.sa_handler = &handle_usr1;
    sigaction(SIGUSR1, &act, NULL);
    act.sa_handler = &handle_usr2;
    sigaction(SIGUSR2, &act, NULL);
    while (1) pause();
}
```

sending signals (2)

pid 1000

```
void handle_usr1(int num) {
    write(1, "Y", 1);
    kill(2000, SIGUSR2);
}

int main() {
    struct sigaction act;
    ... // initialize act
    act.sa_handler = &handle_usr1;
    sigaction(SIGUSR1, &act, NULL);
    sleep(60); // wait for pid 2000 to start
    kill(2000, SIGUSR1);
    while (1) pause();
}
```

pid 2000

```
void handle_usr1(int num) {
    write(1, "X", 1);
    kill(1000, SIGUSR1);
}

void handle_usr2(int num) {
    write(1, "Z", 1);
    kill(1000, SIGTERM);
    _exit(0);
}

int main() {
    struct sigaction act;
    ... // initialize act
    act.sa_handler = &handle_usr1;
    sigaction(SIGUSR1, &act, NULL);
    act.sa_handler = &handle_usr2;
    sigaction(SIGUSR2, &act, NULL);
    while (1) pause();
}
```

sending signals (2)

pid 1000

```
void handle_usr1(int num) {
    write(1, "Y", 1);
    kill(2000, SIGUSR2);
}

int main() {
    struct sigaction act;
    ... // initialize act
    act.sa_handler = &handle_usr1;
    sigaction(SIGUSR1, &act, NULL);
    sleep(60); // wait for pid 2000 to start
    kill(2000, SIGUSR1);
    while (1) pause();
}
```

pid 2000

```
void handle_usr1(int num) {
    write(1, "X", 1);
    kill(1000, SIGUSR1);
}

void handle_usr2(int num) {
    write(1, "Z", 1);
    kill(1000, SIGTERM);
    _exit(0);
}

int main() {
    struct sigaction act;
    ... // initialize act
    act.sa_handler = &handle_usr1;
    sigaction(SIGUSR1, &act, NULL);
    act.sa_handler = &handle_usr2;
    sigaction(SIGUSR2, &act, NULL);
    while (1) pause();
}
```

sending signals (2)

pid 1000

```
void handle_usr1(int num) {
    write(1, "Y", 1);
    kill(2000, SIGUSR2);
}

int main() {
    struct sigaction act;
    ... // initialize act
    act.sa_handler = &handle_usr1;
    sigaction(SIGUSR1, &act, NULL);
    sleep(60); // wait for pid 2000 to start
    kill(2000, SIGUSR1);
    while (1) pause();
}
```

pid 2000

```
void handle_usr1(int num) {
    write(1, "X", 1);
    kill(1000, SIGUSR1);
}

void handle_usr2(int num) {
    write(1, "Z", 1);
    kill(1000, SIGTERM);
    _exit(0);
}

int main() {
    struct sigaction act;
    ... // initialize act
    act.sa_handler = &handle_usr1;
    sigaction(SIGUSR1, &act, NULL);
    act.sa_handler = &handle_usr2;
    sigaction(SIGUSR2, &act, NULL);
    while (1) pause();
}
```


sending signals (2)

pid 1000

```
void handle_usr1(int num) {
    write(1, "Y", 1);
    kill(2000, SIGUSR2);
}

int main() {
    struct sigaction act;
    ... // initialize act
    act.sa_handler = &handle_usr1;
    sigaction(SIGUSR1, &act, NULL);
    sleep(60); // wait for pid 2000 to start
    kill(2000, SIGUSR1);
    while (1) pause();
}
```

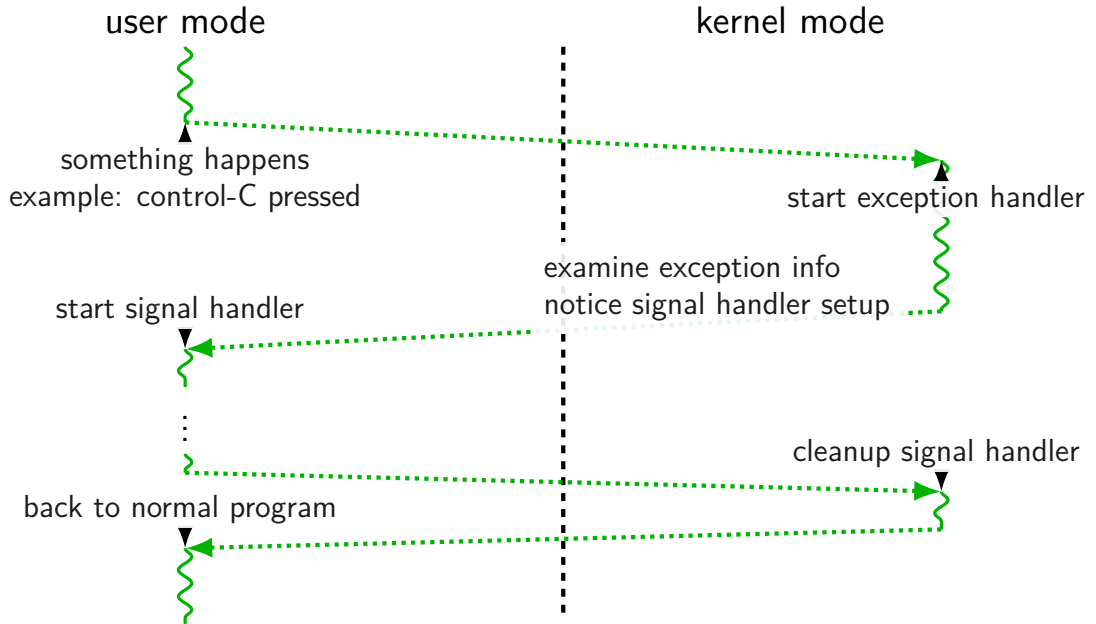
pid 2000

```
void handle_usr1(int num) {
    write(1, "X", 1);
    kill(1000, SIGUSR1);
}

void handle_usr2(int num) {
    write(1, "Z", 1);
    kill(1000, SIGTERM);
    _exit(0);
}

int main() {
    struct sigaction act;
    ... // initialize act
    act.sa_handler = &handle_usr1;
    sigaction(SIGUSR1, &act, NULL);
    act.sa_handler = &handle_usr2;
    sigaction(SIGUSR2, &act, NULL);
    while (1) pause();
}
```

'forwarding' exception as signal



x86-64 Linux signal delivery (1)

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

should stop in the middle of `foo()`

do signal handler

go back to `foo()` without...

changing local variables (possibly in registers)

(and `foo()` doesn't have code to do that)

x86-64 Linux signal delivery (1)

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

should stop in the middle of `foo()`

do signal handler

go back to `foo()` **without...**

changing local variables (possibly in registers)

(and `foo()` doesn't have code to do that)

x86-64 Linux signal delivery (2)

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

OS saves registers **to user stack**

OS modifies user registers, PC to call signal handler

the stack

address of <code>__restore_rt</code>
saved registers
PC when signal happened
local variables for <code>foo</code>
...

→ stack pointer
when signal handler started

→ stack pointer
before signal delivered

x86-64 Linux signal delivery (3)

```
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");  
}
```

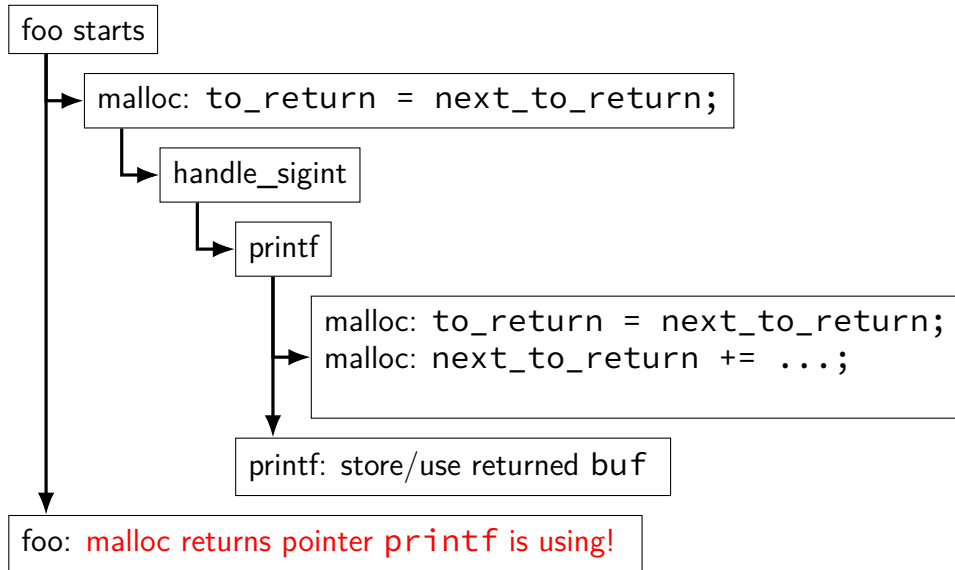

signal handler unsafety (1)

```
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



signal handler unsafety (3)

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

signal handler unsafety (3)

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

signal handler safety

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

these must work correctly no matter what they interrupt

...and no matter how they are interrupted

includes: `write`, `_exit`

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

blocking signals

avoid having signal handlers anywhere:

can instead **block signals**

`sigprocmask()`, `pthread_sigmask()`

blocked = signal handled doesn't run

signal not *delivered*

instead, signal becomes *pending*

controlling when signals are handled

first, block a signal

then use API for inspecting pending signals

example: `sigwait`

typically **instead of having signal handler**

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");  
    int num;  
    if (sigwait(&set, &num) != 0) {  
        printf("sigwait failed!\n");  
    }  
    if (num == SIGINT);  
        printf("Got SIGINT\n");  
    }  
}
```

backup slides