last time

user versus kernel mode

system calls

- special instruction
- system call wrapper
- handler location set at boot

memory protection

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 …)

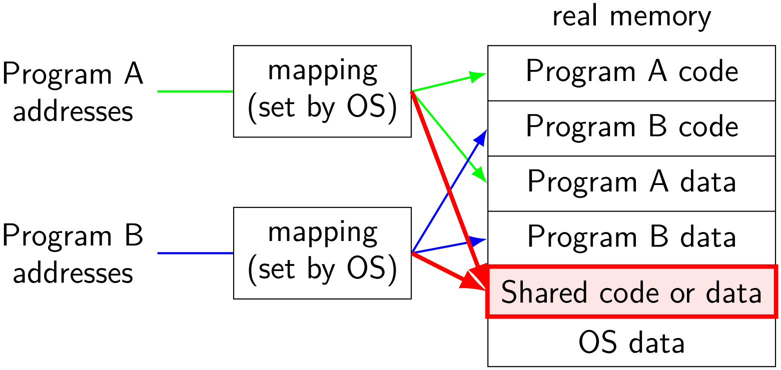
F. something else

shared memory

recall: dynamically linked libraries

would be nice not to duplicate code/data...

we can!



memory protection

modifying another program's memory?

Program A

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

result: %rax (in A) is 42

(always with 'normal' multiuser OSes)

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

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

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
- ...

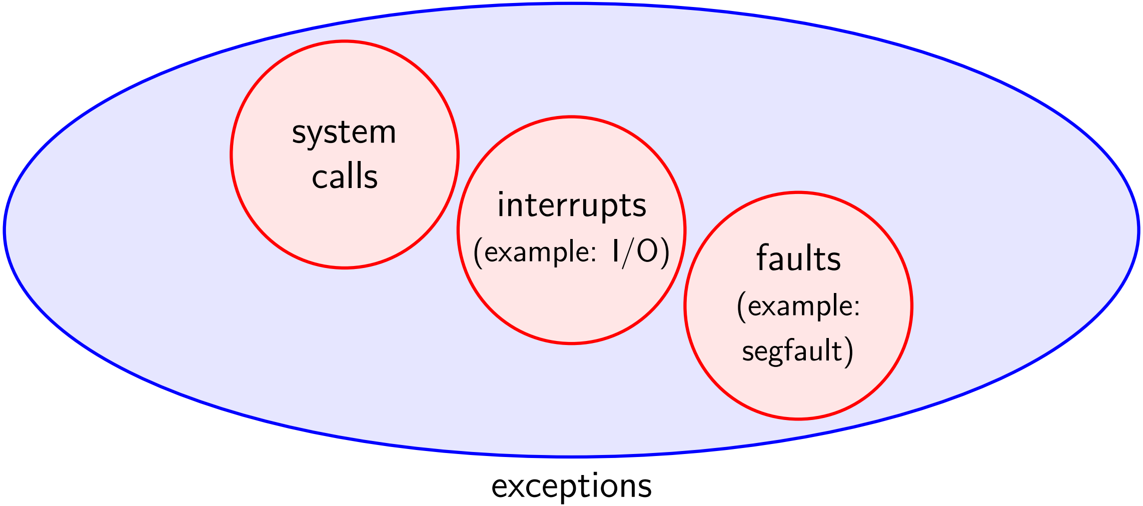
synchronous
triggered by
current program

external — I/O, etc.

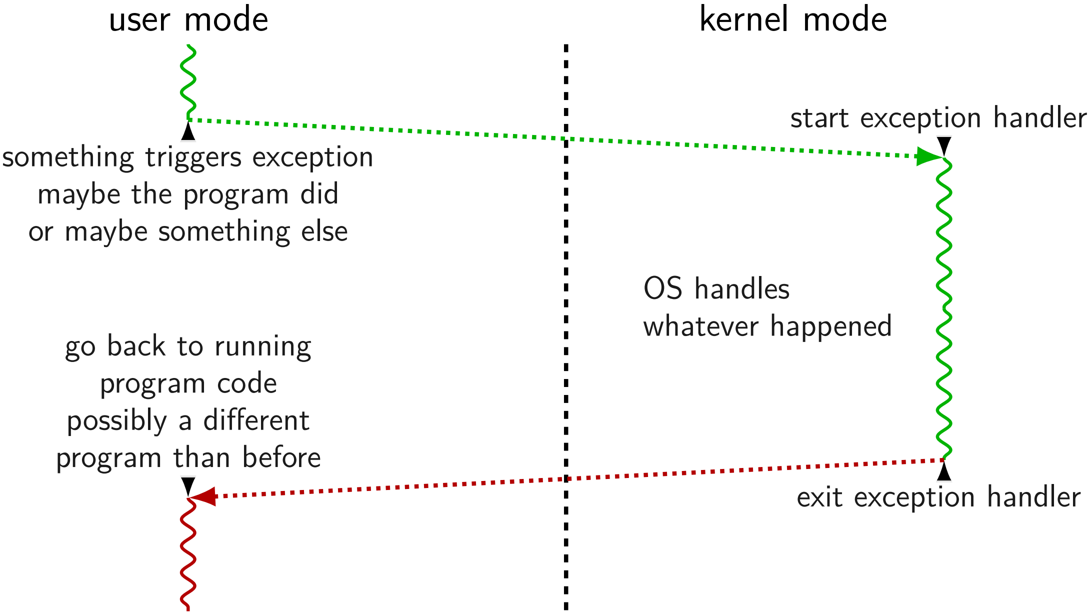
- timer — configured by OS to run OS at certain time
- I/O devices — key presses, hard drives, networks, ...
- hardware is broken (e.g. memory parity error)

asynchronous
not triggered by
running program

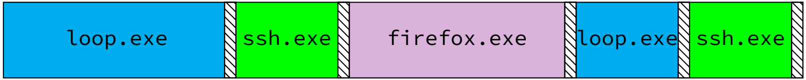
exceptions [Venn diagram]



general exception process

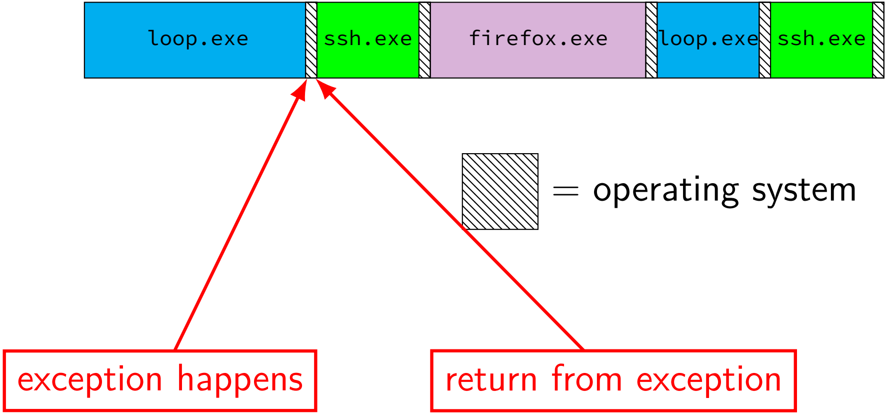


time multiplexing



▨ = operating system

time multiplexing



▨ = operating system

exception happens

return from exception

switching programs

OS starts running somehow

- some sort of exception

saves old registers + program counter + address mapping

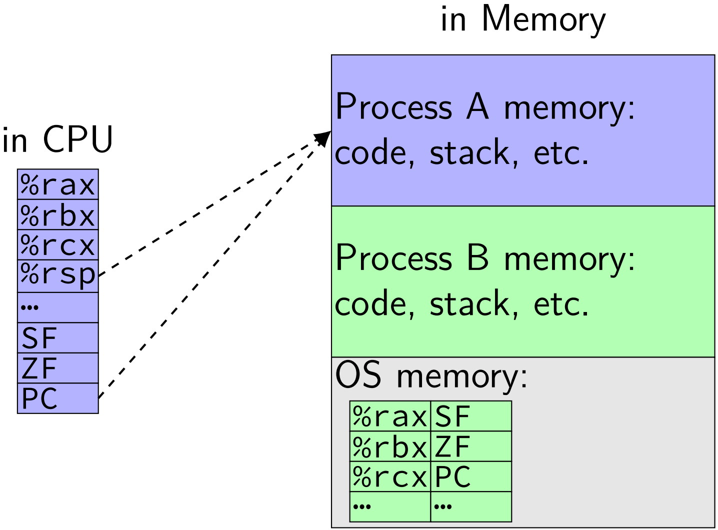
- (optimization: could omit when program crashing/exiting)

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

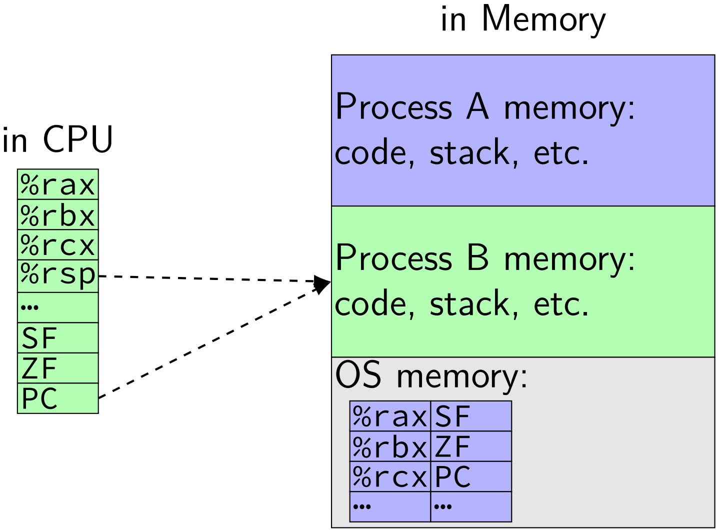
called context switch

- saved information called context

contexts (A running)



contexts (B running)



in Memory

Process A memory:
code, stack, etc.

Process B memory:
code, stack, etc.

OS memory:

%rax	SF
%rbx	ZF
%rcx	PC
…	…

in CPU

%rax
%rbx
%rcx
%rsp
…
SF
ZF
PC

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
- …

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)

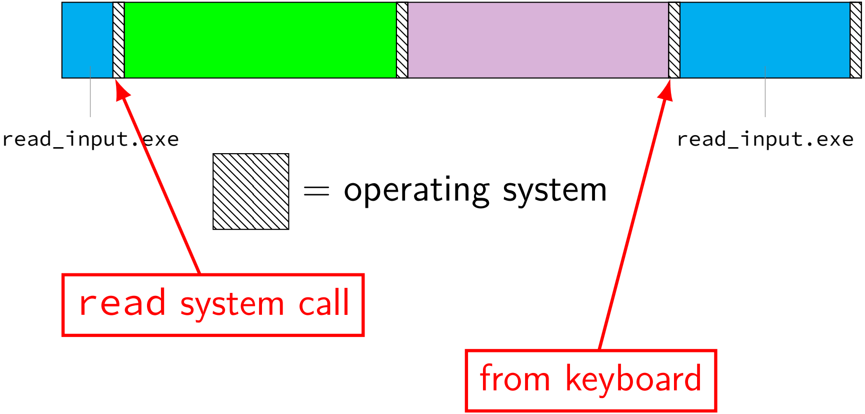
output — ready now:

- exception (syscall): program says “I want to output this’
- handler: OS sends output to 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



read_input.exe

read_input.exe

▨ = operating system

read system call

from keyboard

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
- (process could have multiple threads — with independent registers)
- 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 program counter changes	thread program counter changes

program counter = instruction to run next

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 program counter changes	thread program counter changes

program counter = instruction to run next

…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 program counter changes	thread program counter changes

program counter = instruction to run next

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 program counter changes	thread program counter changes

program counter = instruction to run next

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

```c
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);
    // SA_RESTART = if syscall interrupted,
    // complete it when handler returns
    act.sa_flags = SA_RESTART;
    sigaction(SIGINT, &act, NULL);

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

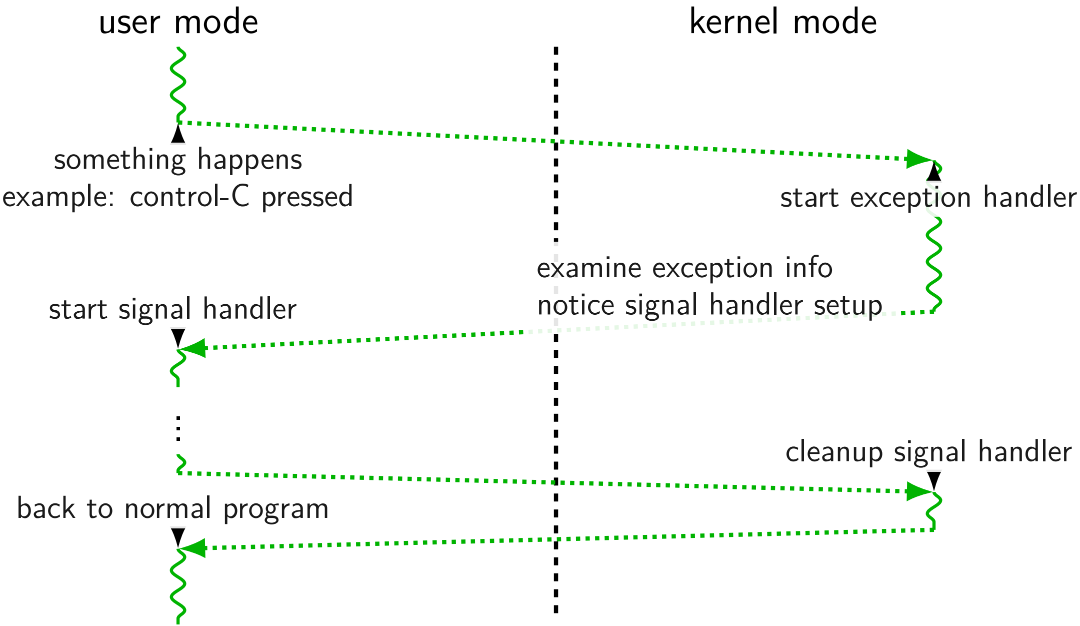
example signal program

```
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);
    // SA_RESTART = if syscall interrupted,
    // complete it when handler returns
    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);
    // SA_RESTART = if syscall interrupted,
    // complete it when handler returns
    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



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

- 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

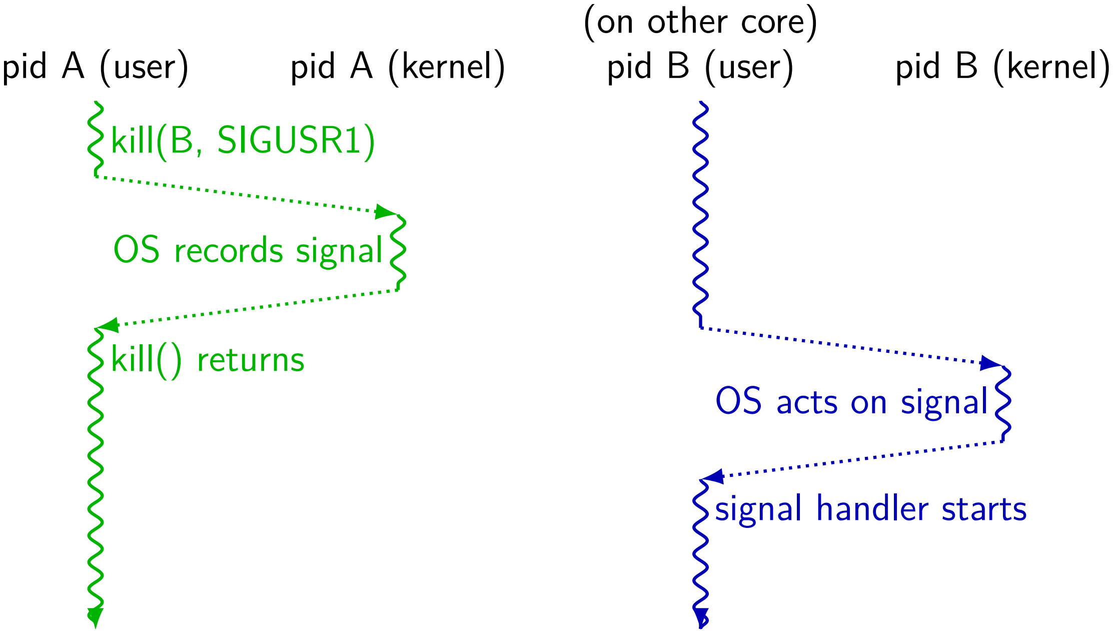
`kill 1234` — sends SIGTERM to pid 1234

in C: `kill(1234, SIGTERM)`

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

in C: `kill(1234, SIGUSR1)`

kill() not always immediate



(on other core)

pid A (user)

pid A (kernel)

pid B (user)

pid B (kernel)

kill(B, SIGUSR1)

OS records signal

kill() returns

OS acts on signal

signal handler starts

output of this?

pid 1000

```
void handle_usr1(int num) {
    write(1, "X", 1);
    kill(2000, SIGUSR1);
    _exit(0);
}
int main() {
    struct sigaction act;
    ... // initialize rest of "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;
    ... // initialize rest of "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

```c
void handle_usr1(int num) {
    write(1, "X", 1);
    kill(2000, SIGUSR1);
    _exit(0);
}
int main() {
    struct sigaction act;
    ... // initialize rest of "act"
    act.sa_handler = &handle_usr1;
    sigaction(SIGUSR1, &act);
    sleep(1);
    kill(1000, SIGUSR1);
    while (1) pause();
}
```

pid 2000

```c
void handle_usr1(int num) {
    write(1, "Y", 1);
    _exit(0);
}
int main() {
    struct sigaction act;
    ... // initialize rest of "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

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");
}
```

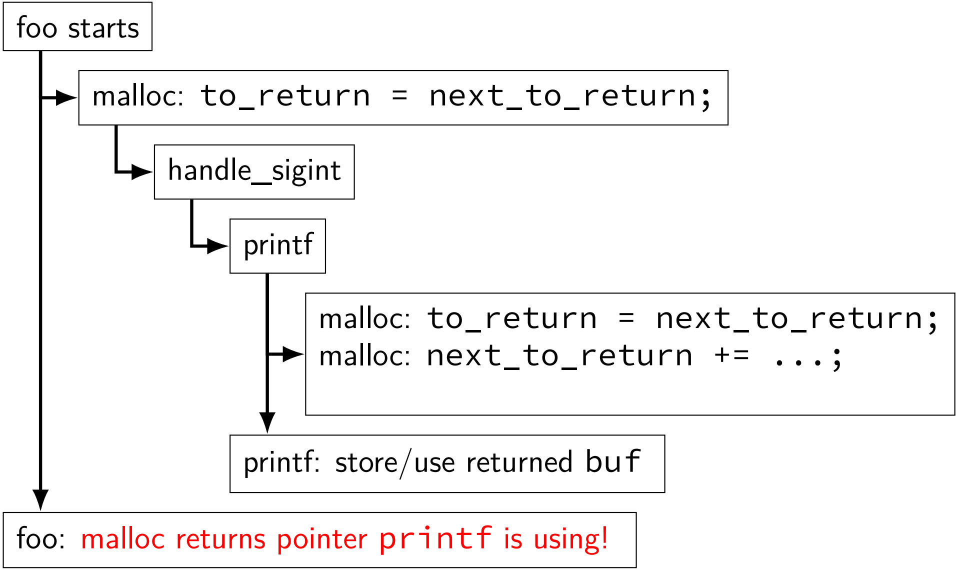
signal handler unsafety (1)

```c
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

- `sigprocmask()`, `pthread_sigmask()`

blocked = signal handled doesn't run

- signal not *delivered*

instead, signal becomes *pending*

- delivered if unblocked

blocking signals

avoid having signal handlers anywhere:

can instead block signals

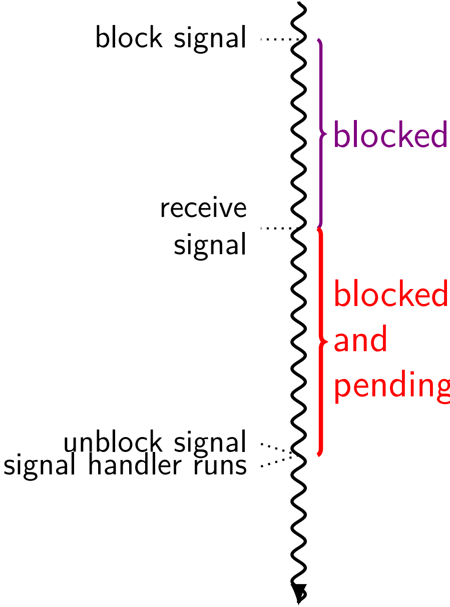
- `sigprocmask()`, `pthread_sigmask()`

blocked = signal handled doesn't run

- signal not *delivered*

instead, signal becomes *pending*

- delivered if unblocked



controlling when signals are handled

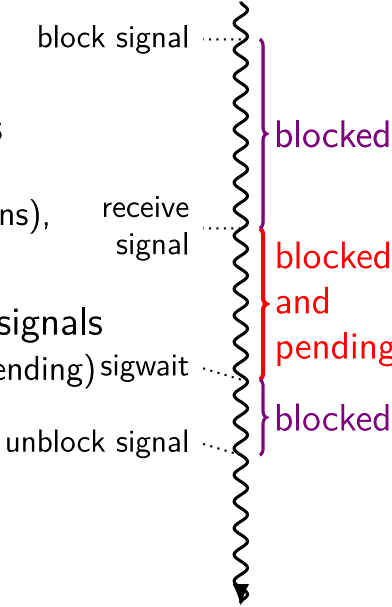
first, block a signal

then either unblock signals only at certain times

- some special functions to help:
- `sigsuspend` (unblock and wait until handler runs),
- `pselect` (unblock while checking for I/O), ...

and/or use API for checking/changing pending signals

- example: `sigwait` (wait for signal to become pending)
- typically instead of having signal handler



controlling when signals are handled

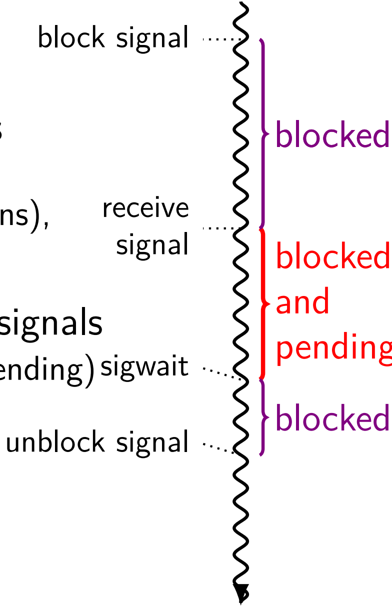
first, block a signal

then either unblock signals only at certain times

- some special functions to help:
- `sigsuspend` (unblock and wait until handler runs),
- `pselect` (unblock while checking for I/O), ...

and/or use API for checking/changing pending signals

- example: `sigwait` (wait for signal to become pending)
- typically instead of having signal handler



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

keeping permissions?

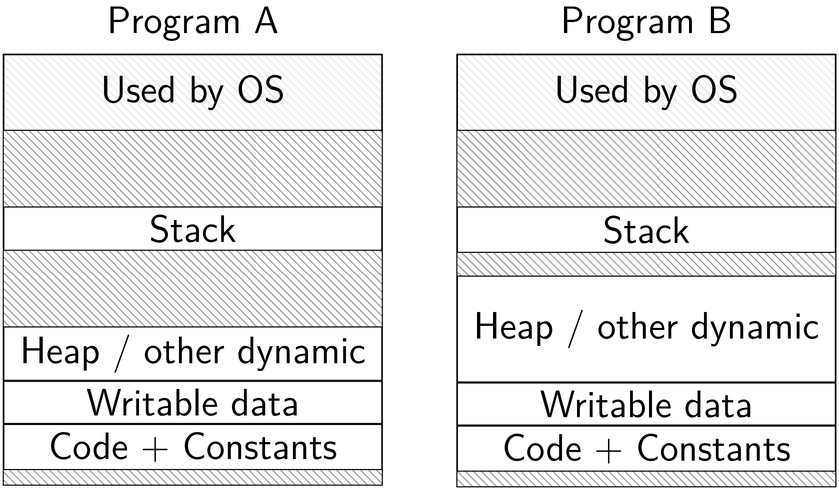
which of the following would still be secure?

A. performing authorization checks in the standard library in addition to system call handlers

B. performing authorization checks in the standard library instead of system call handlers

C. making the user ID a system call argument rather than storing it persistently in the OS's memory

program memory (two programs)



Program A

Used by OS

Stack

Heap / other dynamic

Writable data

Code + Constants

Program B

Used by OS

Stack

Heap / other dynamic

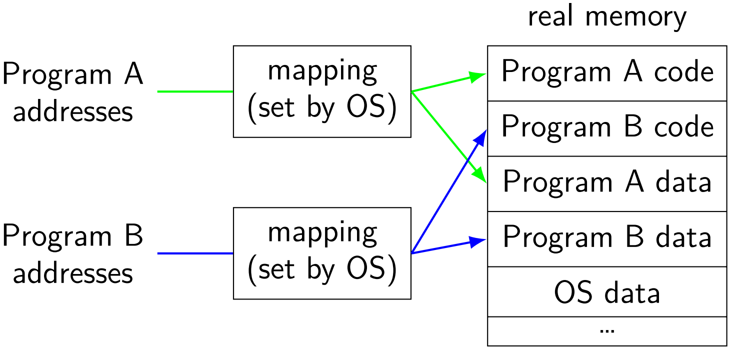
Writable data

Code + Constants

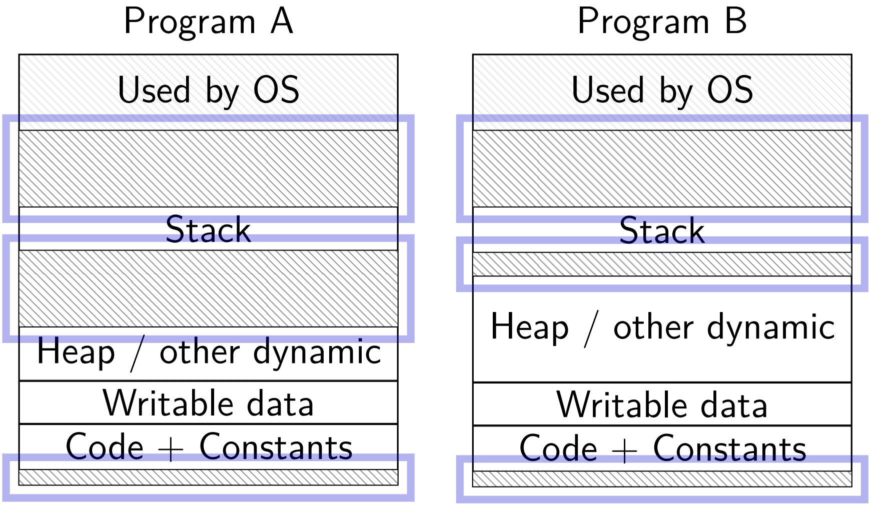
address space

programs have illusion of own memory

called a program's address space



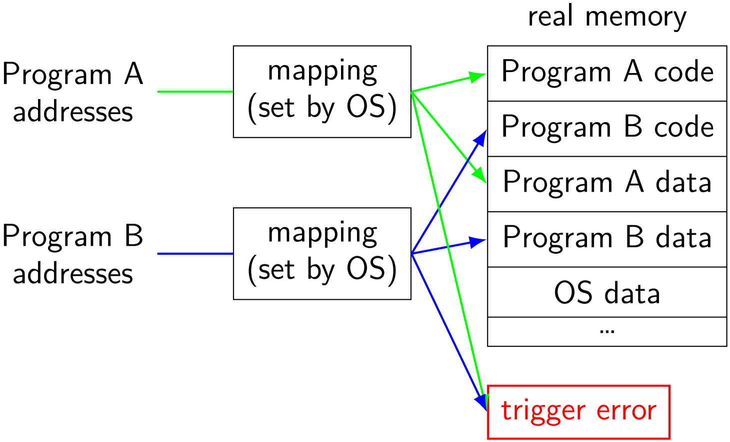
program memory (two programs)



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

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

an infinite loop

```
int main(void) {
    while (1) {
        /* waste CPU time */
    }
}
```

If I run this on a shared department machine, can you still use it?

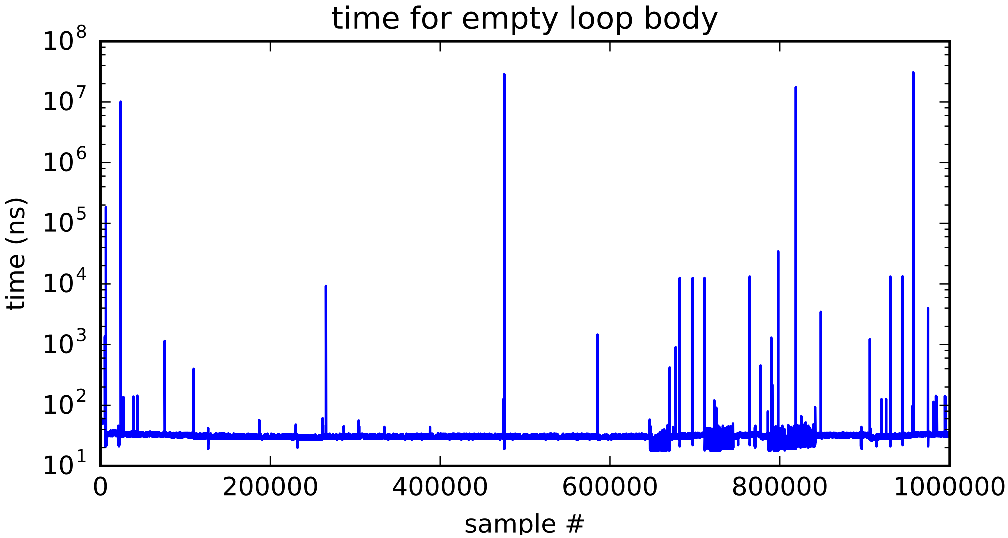
...if the machine only has one core?

timing nothing

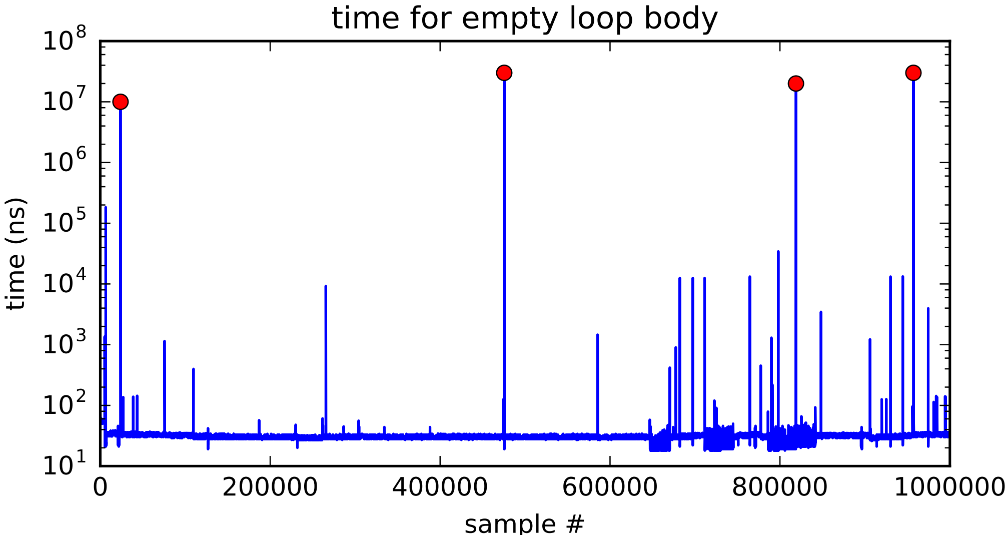
```
long times[NUM_TIMINGS];
int main(void) {
    for (int i = 0; i < N; ++i) {
        long start, end;
        start = get_time();
        /* do nothing */
        end = get_time();
        times[i] = end - start;
    }
    output_timings(times);
}
```

same instructions — same difference each time?

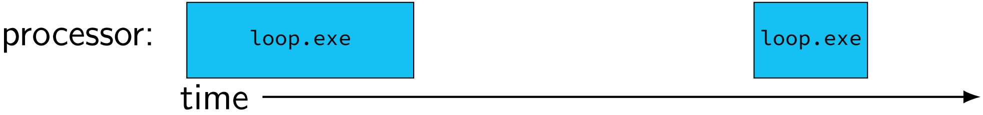
doing nothing on a busy system



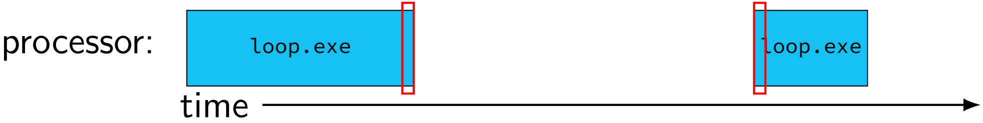
doing nothing on a busy system



time multiplexing



time multiplexing

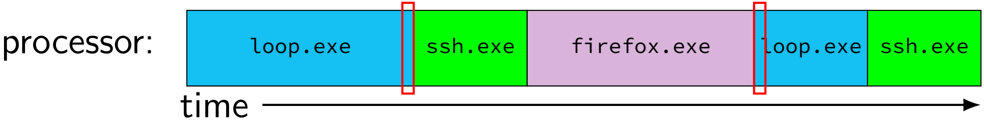


```
...
loop: ...
      ...
      jmp loop
loop: ...
      ...
```

million cycle delay

```
      ...
      jmp loop
loop: ...
      ...
```

time multiplexing

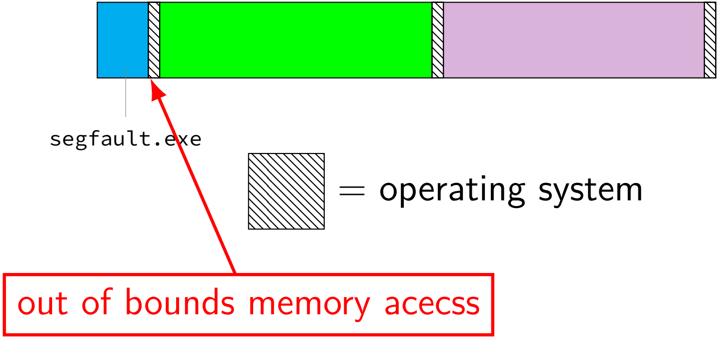


```
...
loop: ...
    ...
    jmp loop
loop: ...
    ...
```

million cycle delay

```
    ...
    jmp loop
loop: ...
    ...
```

crash timeline timeline

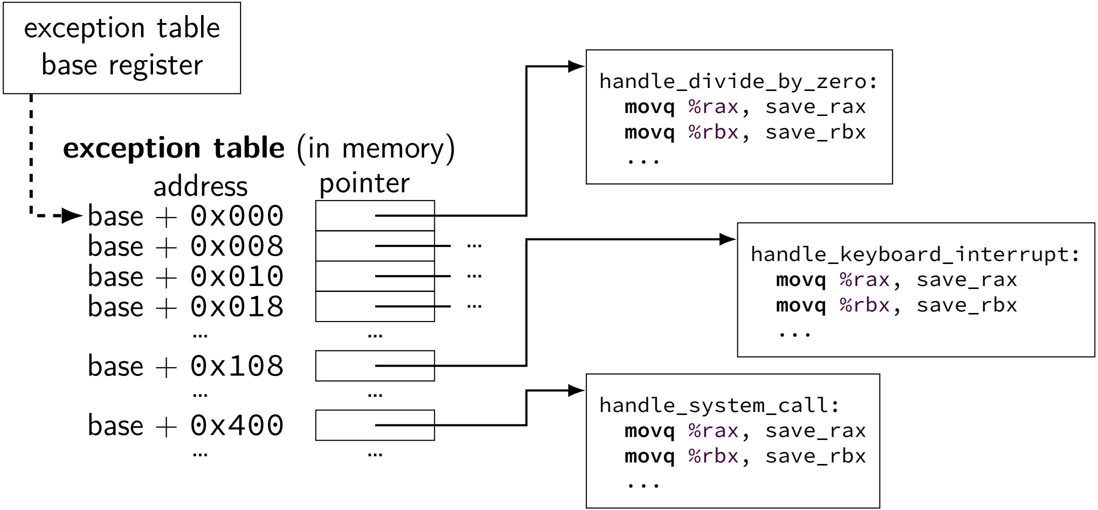


segfault.exe

▨ = operating system

out of bounds memory acecss

locating exception handlers (one strategy)



exception table base register

exception table (in memory)

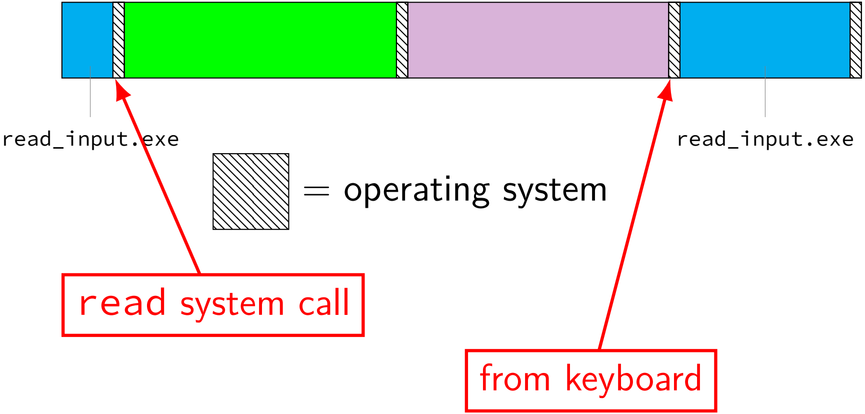
address	pointer
base + 0x000	
base + 0x008	…
base + 0x010	…
base + 0x018	…
…	…
base + 0x108	
…	…
base + 0x400	
…	…

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

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

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

keyboard input timeline



exceptions in exceptions

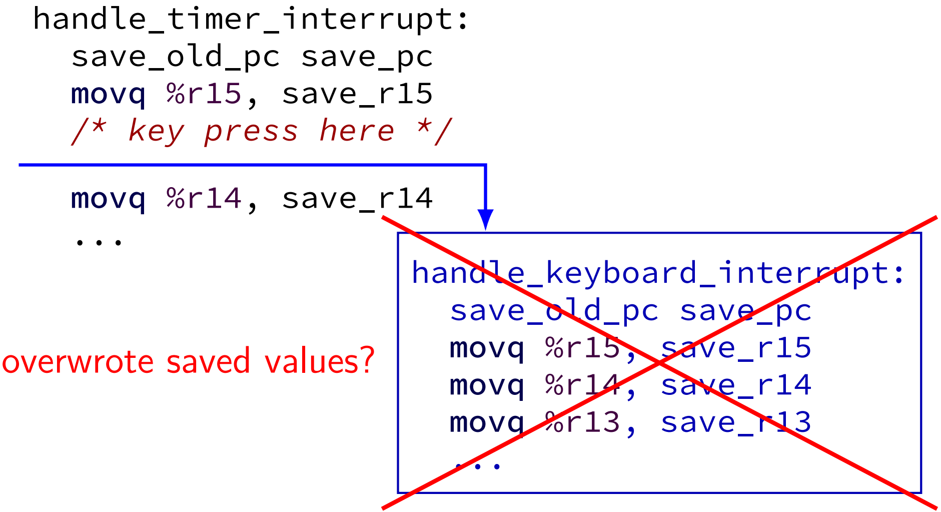
```
handle_timer_interrupt:
  save_old_pc save_pc
  movq %r15, save_r15
  /* key press here */

  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
```

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)

context

all registers values

    %rax %rbx, …, %rsp, …

condition codes

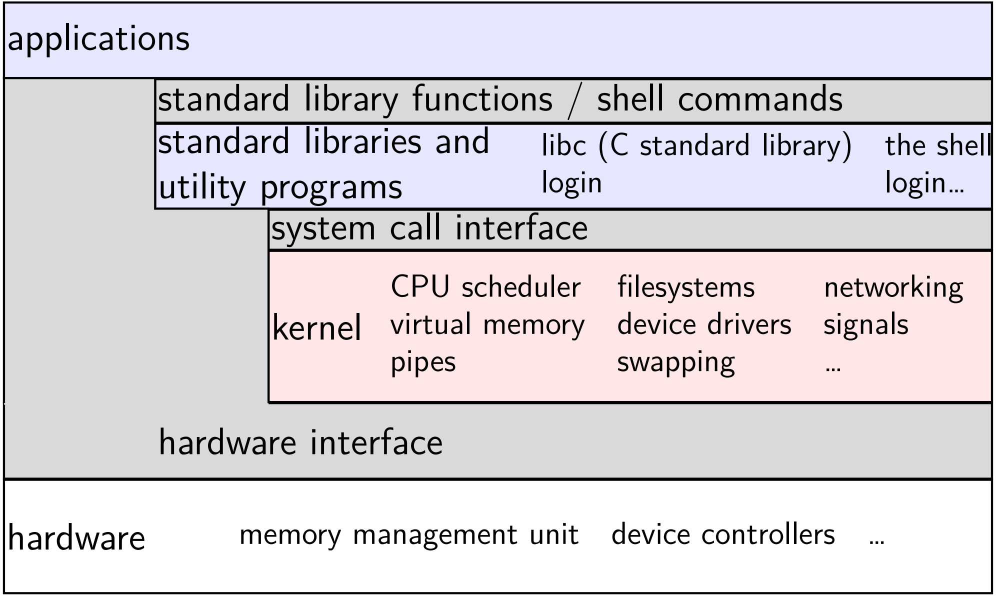
program counter

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



applications

standard library functions / shell commands

standard libraries and utility programs

libc (C standard library)

login

the shell

login...

system call interface

kernel

CPU scheduler

virtual memory

pipes

filesystems

device drivers

swapping

networking

signals

...

hardware interface

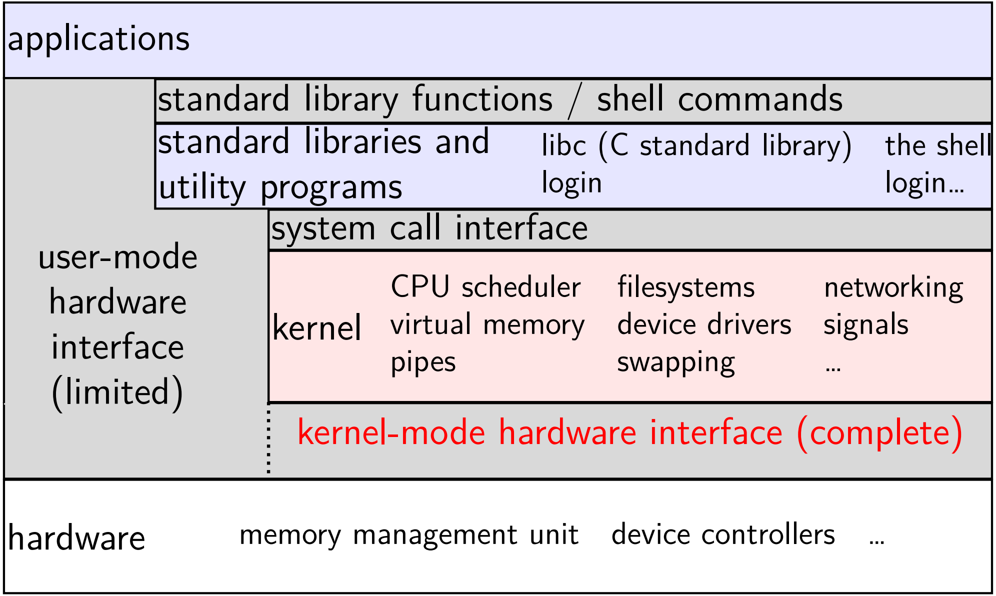
hardware

memory management unit

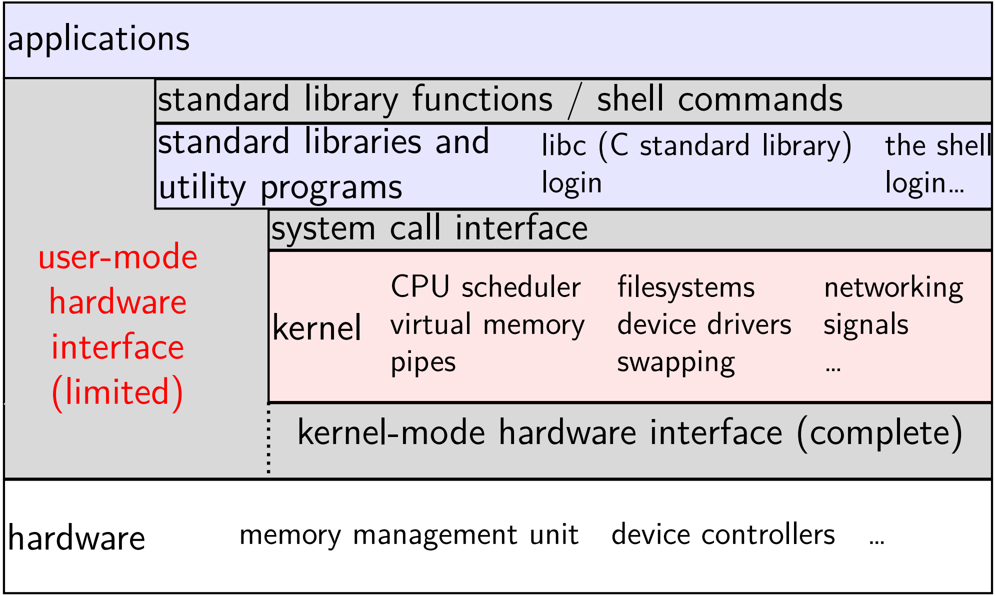
device controllers

...

the classic Unix design



the classic Unix design



applications

standard library functions / shell commands

standard libraries and utility programs

libc (C standard library)
login

the shell
login...

system call interface

kernel

CPU scheduler
virtual memory
pipes

filesystems
device drivers
swapping

networking
signals
...

user-mode hardware interface (limited)

kernel-mode hardware interface (complete)

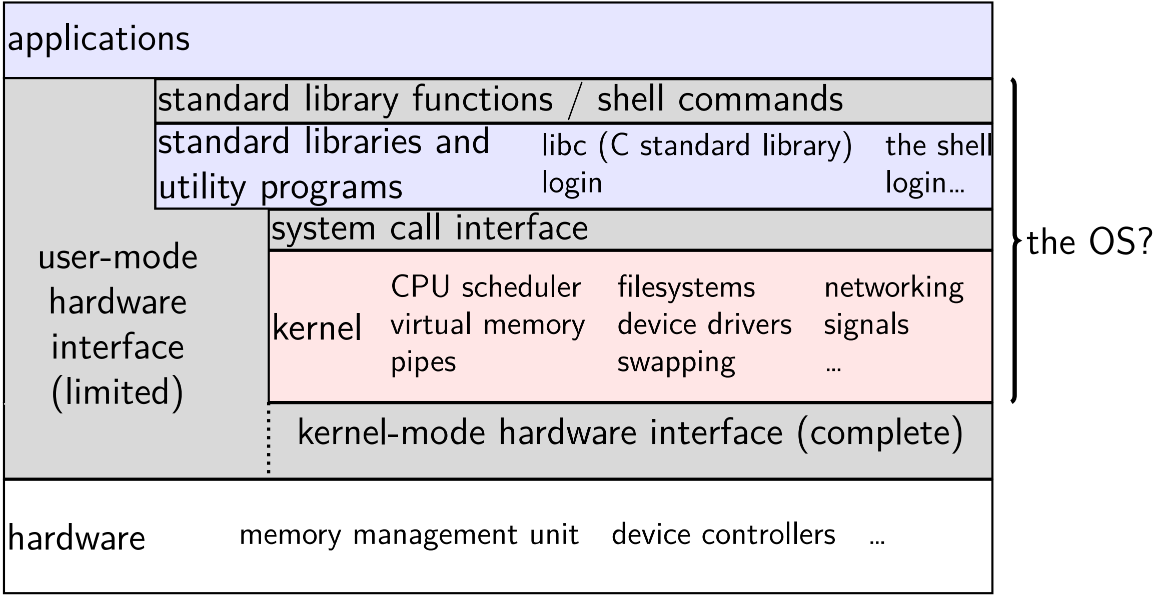
hardware

memory management unit

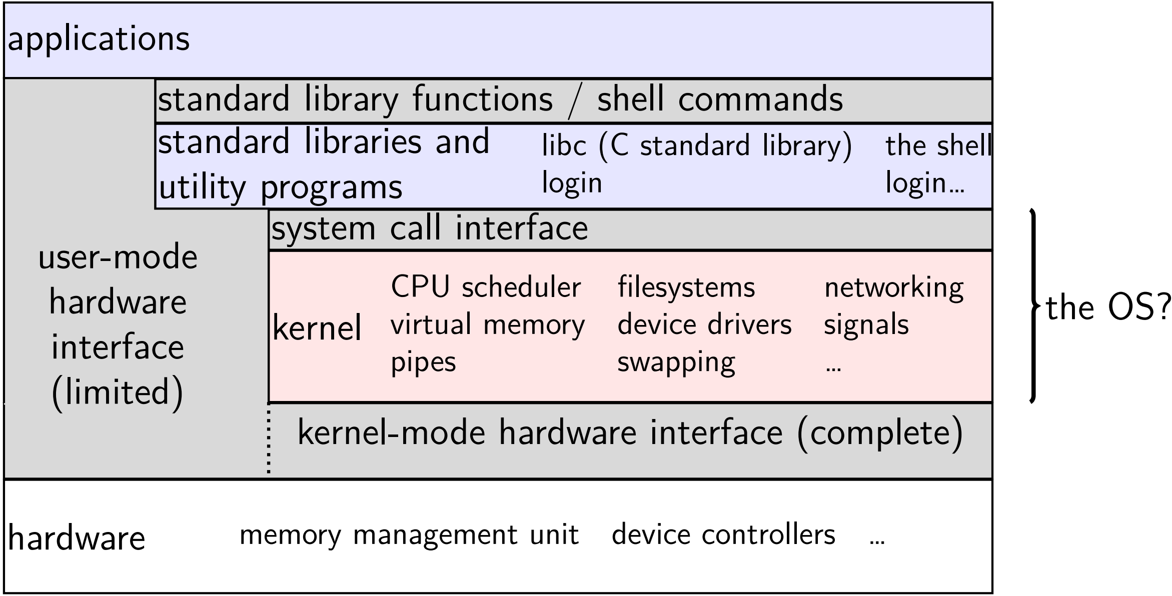
device controllers

...

the classic Unix design



the classic Unix design



applications

standard library functions / shell commands

standard libraries and utility programs

libc (C standard library)
login

the shell
login...

system call interface

kernel

CPU scheduler
virtual memory
pipes

filesystems
device drivers
swapping

networking
signals
...

the OS?

user-mode hardware interface (limited)

kernel-mode hardware interface (complete)

hardware

memory management unit

device controllers

...

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.