kernel 2 / signals

last time (1)

kernel mode
- kernel mode — "dangerous" operations allowed
- only OS code allowed to run in kernel mode

exceptions
- hardware runs OS-specified routine in kernel mode
- allows OS to help programs/hardware do something

system calls — exceptions intentionally triggered by program
- how programs ask to do something that needs kernel mode

other exceptions — things hardware needs OS help to handle
- program "errors" (divide by zero, out-of-bounds, etc.)
- I/O events (keypress, network input, etc.)
- timer

last time (2)

address translation / address spaces

- address program uses not “real” address
- OS sets mapping (function) from program to real addresses
- mapping limits what memory program can access
- mapping allows any program address OS chooses
- one mapping per running program

time multiplexing

- processor shared between multiple programs over time
- when OS runs from exception, can switch programs

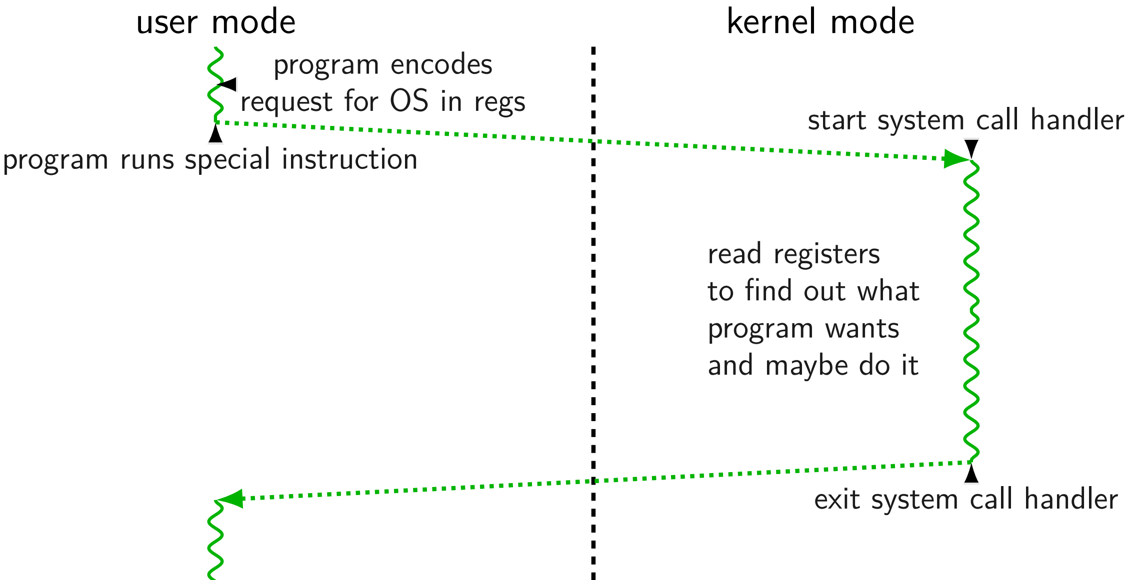
anonymous feedback

"Not a huge thing, but would it be possible to run code on the slides on a program during lecture? Seeing the text on the slides helps, but I feel it would help us better to know how to set up our code in terminal, see the results in real time, and explain errors if they arise? Seeing a lot of code on the slides is a sometimes a bit overwhelming or hard to understand in the current format."

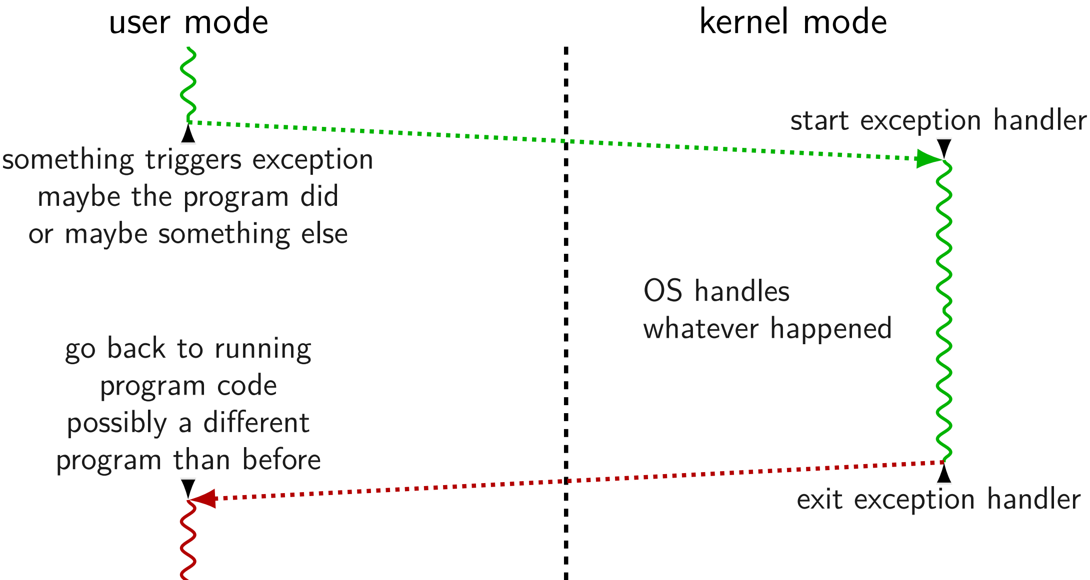
> when I do live demos, usually pretty canned/setup in advance
> so probably not helpful for what you want
> probably should spend more time explaining code on slides

"Can you explain system calls/ time multiplexing again/ clarify it. It was confusing during lecture/ felt rushed. And could you further explain the diagram with kernel/ system call more clearly"

system call process



general exception process



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

an infinite loop

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

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

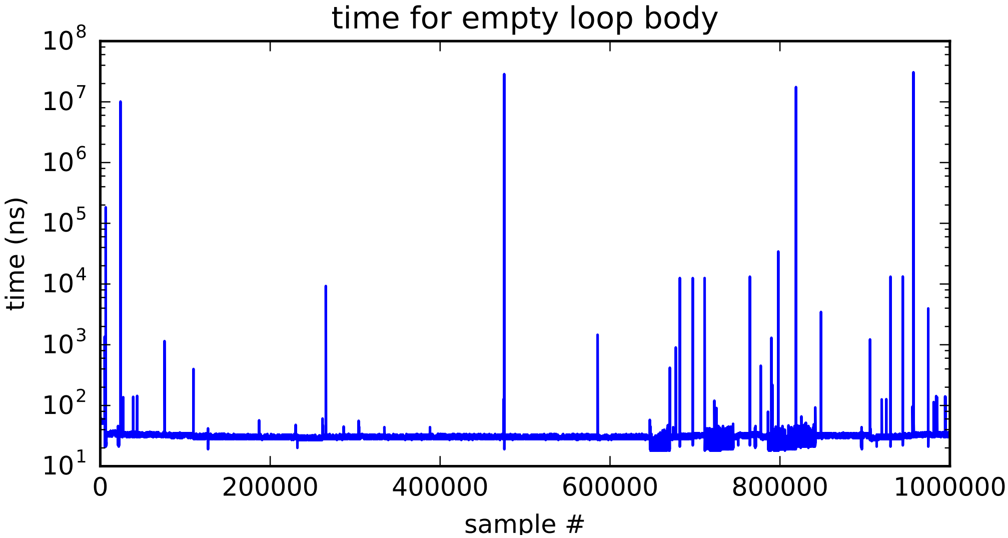
...if the machine only has one core?

timing nothing

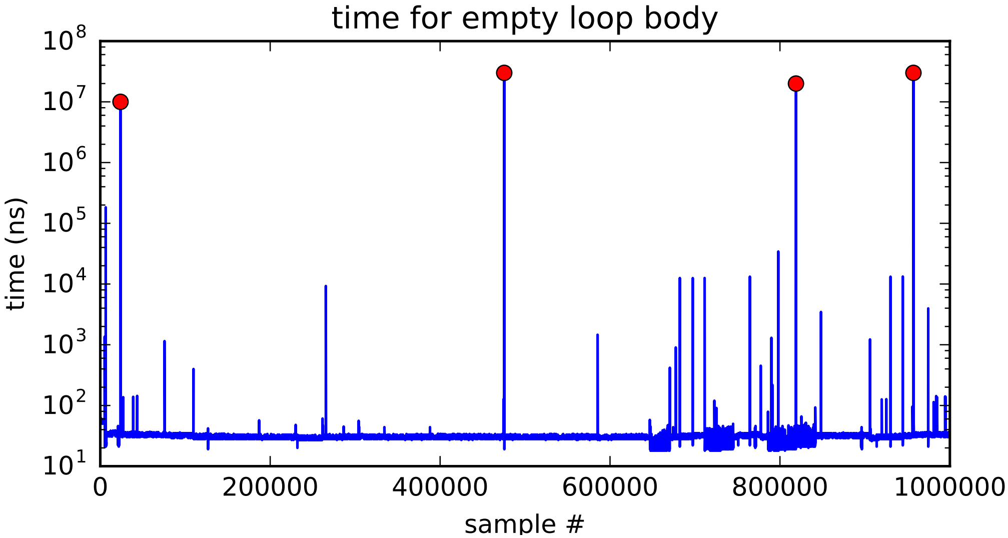
```
long times[NUM_TIMINGS];
int main(void) {
    for (int i = 0; i < N; ++i) {
        long start, end;
        start = get_time();
        /* do nothing */
        end = get_time();
        times[i] = end - start;
    }
    output_timings(times);
}
```

same instructions — same difference each time?

doing nothing on a busy system



doing nothing on a busy system



time multiplexing

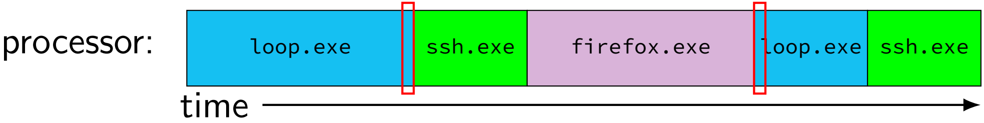
time multiplexing


```
...
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

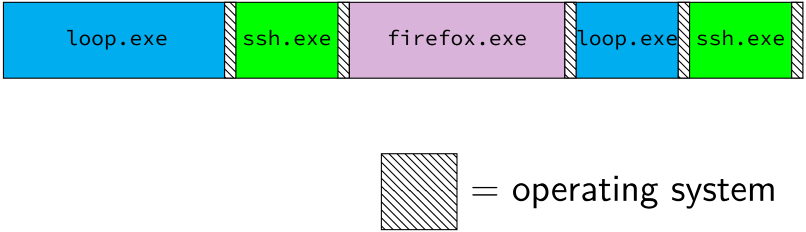



```
...
call get_time
    // whatever get_time does
movq %rax, %rbp
```

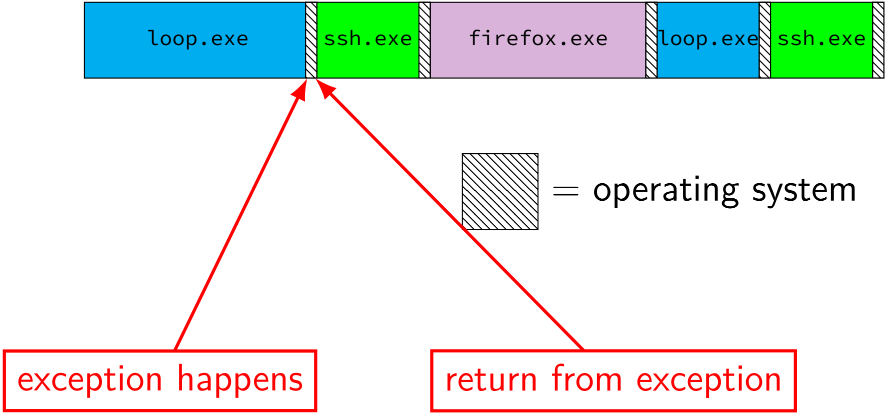
million cycle delay

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

time multiplexing really



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
- ...

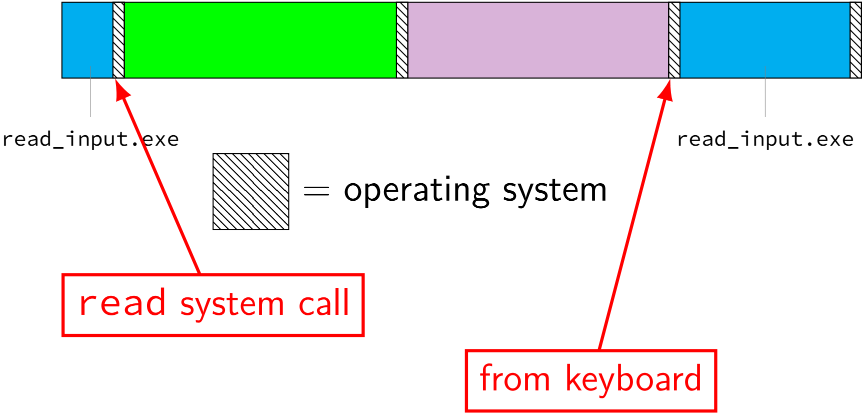
synchronous
triggered by current program

external — I/O, etc.

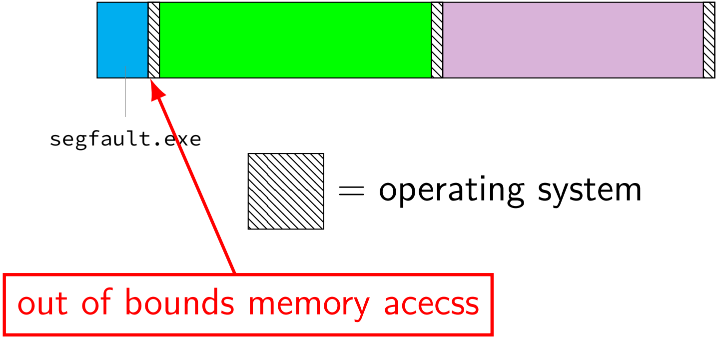
- timer — configured by OS to run OS at certain time
- I/O devices — key presses, hard drives, networks, ...
- hardware is broken (e.g. memory parity error)

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

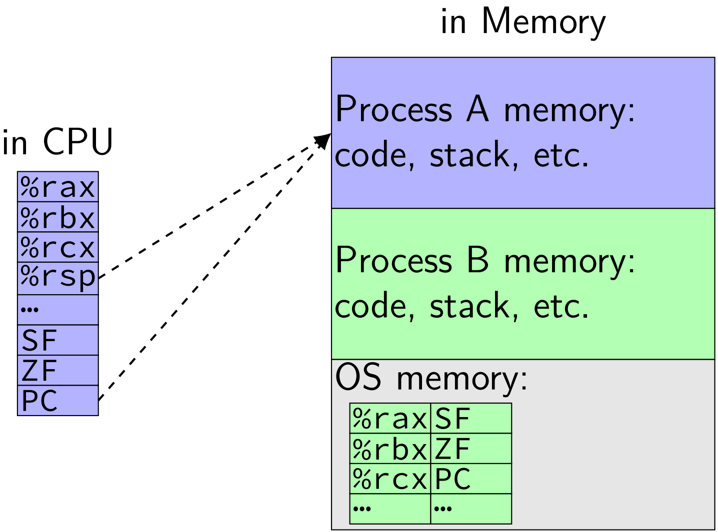
- (optimization: could omit when program crashing/exiting)

sets new registers, jumps to new program counter

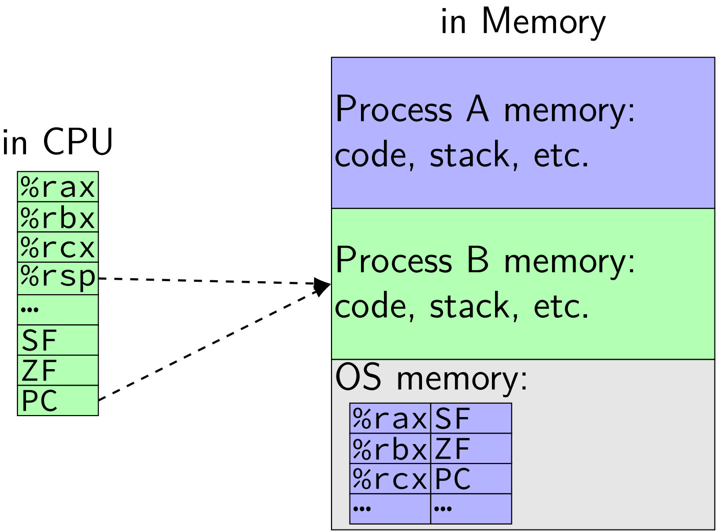
called context switch

- 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

which require exceptions [answers] (1)

A. program calls a function in the standard library

  no (same as other functions in program; some standard library functions might make system calls, but if so, that'll be part of what happens after they're called and before they return)

B. program writes a file to disk

  yes (requires kernel mode only operations)

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

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

which require exceptions [answer] (2)

D. program exits

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

E. program returns from one function to another function

no

F. program pops a value from the stack

no

which require context switches [answer]

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

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

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

yes: D. program exits

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

no: F. program pops a value from the stack

terms for exceptions

terms for exceptions aren't standardized

our readings use one set of terms

interrupts = externally-triggered
faults = error/event in program
trap = intentionally triggered

all these terms appear differently elsewhere

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

```
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

```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**

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
```

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

```
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_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

```c
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

```c
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

the stack

address of `__restore_rt`
saved registers
PC when signal happened
local variables for foo
...

→ stack pointer when signal handler started

→ stack pointer before signal delivered

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