signals

changelog

8 Feb 2024: kill() is not already immediate: correct argument order to kill() call

last time

exceptions = processor runs OS

- call handler setup at boot in kernel mode
- many causes
- system calls (program requests OS help)
- program does something unexpected (example: divide by zero)
- input/ouptut devices, timer (external event interrupts program)

process = ‘virtual’ machine

- thread = processor simulated by sharing real processor over time
- address space = memory simulated by mapping program addresses (so programs cannot interfere with each other)

Q1-3 (part 1)

(1-2) compiler waits for read from disk (system call to wait)

- I guess you could loop checking if read is done, but that's pretty inefficient

(3) simulation runs — probably switched to by handler for system call in (1)

(4) text editor runs + update screen from keypress

- I/O exception causes text editor to run
- finishes operation that was started by earlier system call exception
- text editor makes system calls for output/requesting input

Q1-3 (part 1)

(1-2) compiler waits for read from disk (system call to wait)

- I guess you could loop checking if read is done, but that's pretty inefficient

(3) simulation runs — probably switched to by handler for system call in (1)

(4) text editor runs + update screen from keypress

- I/O exception causes text editor to run
- finishes operation that was started by earlier system call exception
- text editor makes system calls for output/requesting input

Q1-3 (part 2)

(4) text editor runs + update screen from keypress

  I/O exception causes text editor to run

  finishes operation that was started by earlier system call exception

  text editor triggers system calls for output/requesting input

(5) simulation resumes runnning (part of handling text editor input system call)

(6) read from disk finishes, run compiler (I/O exception)

  part of handling compiler's system call from (1)

(7) compiler open+write file (probably at least two system calls)

(8) while waiting for write, simulation runs

  (part of handling compiler system call)

Q1-3 (part 2)

(4) text editor runs + update screen from keypress

- I/O exception causes text editor to run
- finishes operation that was started by earlier system call exception
- text editor triggers system calls for output/requesting input

(5) simulation resumes runnning (part of handling text editor input system call)

(6) read from disk finishes, run compiler (I/O exception)

- part of handling compiler's system call from (1)

(7) compiler open+write file (probably at least two system calls)

(8) while waiting for write, simulation runs

- (part of handling compiler system call)

Q1

non-system-call exception handler would complete operation requested via prior system call exception

for this purpose, most notable that exceptions can come from input and output devices

probably should avoid using plain 'system call':

- system call operation $\sim$ thing requested by program of OS using exception
- system call exception $\sim$ jumping to the OS handler that will figure out what program wants in response to special 'system call' instruction

Q4

first process: 1, yield

second process: A, yield

first process: 2, yield

second process: B, yield

first process: 3, yield

second process: C, yield

Q5

print/fflush make system calls?

normal function call to printf/fflush

implementation of printf/fflush triggers system call

- special instruction to do this part of library

system call causes code in OS (not library/main()) to run in kernel mode

Q6

x stored in %r15 when first process running

when second process running, its data is in %r15

so first process's %r15 must be saved somewhere else

will be done by OS

anonymous feedback (1)

“I feel like quiz 2 is too difficult, because we were not taught enough about the first part of the quiz. I also feel like the sequence of events, are vague. We did not learn what exceptions happen after a keypress occurs, and what exceptions happen in the stages of a keypress. Also in the notes, it just says exceptions, it does not say what type of exception. is context switching an exception? Also, I don’t like how in question 2, you say not likely, that is so vague. Also, what is the difference between a system call and a system call exception? isn’t a system call an exception? After a program ends or completes a process, is there an interrupt. If so, then every context switch has an interrupt? Is an exception just when its kernel mode? The definitions are vague”

“you should make a list of non sys call exceptions and sys call exceptions and exceptions that lead to context switches. Also the context between them, like what happens when a keypress occurs in every stage, because this was not in depth enough during lecture for us to answer the quiz” exception ∼ hardware runs the OS to do something

- yes, runs the OS in kernel mode (way to get into kernel mode from user mode)
- lots of reasons this might happen ('kinds' of exceptions)
    - external (e.g. input/output device needs attention, timer)
    - internal, unintentional (e.g. divide-by-zero, out-of-bounds)
    - internal, intentional (system calls)
- (list of more specific reasons not exhaustive because it varies…)

system call ∼ request from program for the OS to do something for it

- that is made by deliberately triggering exception

quiz avoided other exception vocabulary because I don't intend to test about it

context switches and exceptions

context switch ~ change registers values to differnet program

something the OS can do whenever it runs

only related to exceptions because OS runs due to execptions

- means if program ‘ends’, OS had to run to do it
- some some exception happened to do this — which one depends on details of how it ended

anonymous feedback (2)

“ your lectures go over things big picture, but your quizzes are in depth. Even after reading the readings and slides, I still feel we did not learn enough to answer the quizzes. Since its our first time learning this material, I think we need it to be spelled out more. I also don’t like how when I try to find other resources on the topics, like different types of exceptions, I have a hard time finding it out because everyone seems to define it differently, which means it is even more important that we get the information from you. I think it might be helpful if we had a glossary or terms, with exact definitions all in one page, and maybe a flow chart for how things relate to each other? Or maybe some links to textbook pages. I looked at the textbook linked, but it didn’t have enough regarding exceptions since I am not confused about what they are, I am confused regarding your definition of them. I get why your reviews say you expect too much, it is because you don’t give us enough”

I agree the readings for the kernel stuff probably should have a glossary

when I point to textbooks in the ‘further resources’ for kernel, I should note what terms they are using versus our reading/lecture to make those references more useful

> e.g. I like ‘Dive Into Systems’ explanation, but they never actually use the word ‘exception’ (just interrupt (external exception) and ‘trap’ (exception triggered by trying to run something)

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

```c
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);
    }
}
```

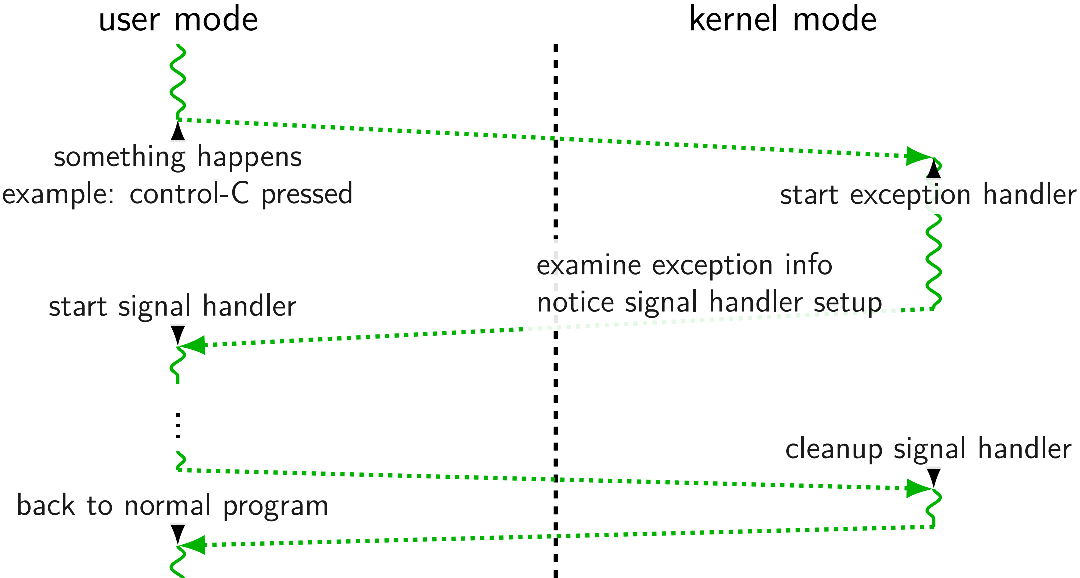
example signal program

```
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

```c
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



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

- 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

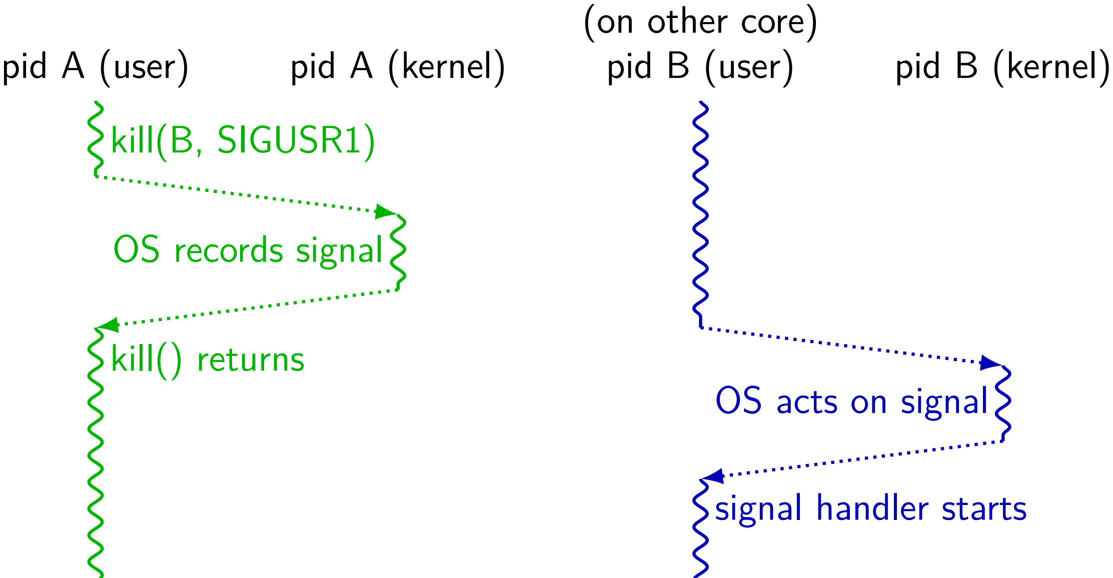
`kill 1234` — sends SIGTERM to pid 1234

  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

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 (detect by checking `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

```c
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

```c
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

```c
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

```c
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

```c
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

```c
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

```c
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();
}
```

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");
}
```

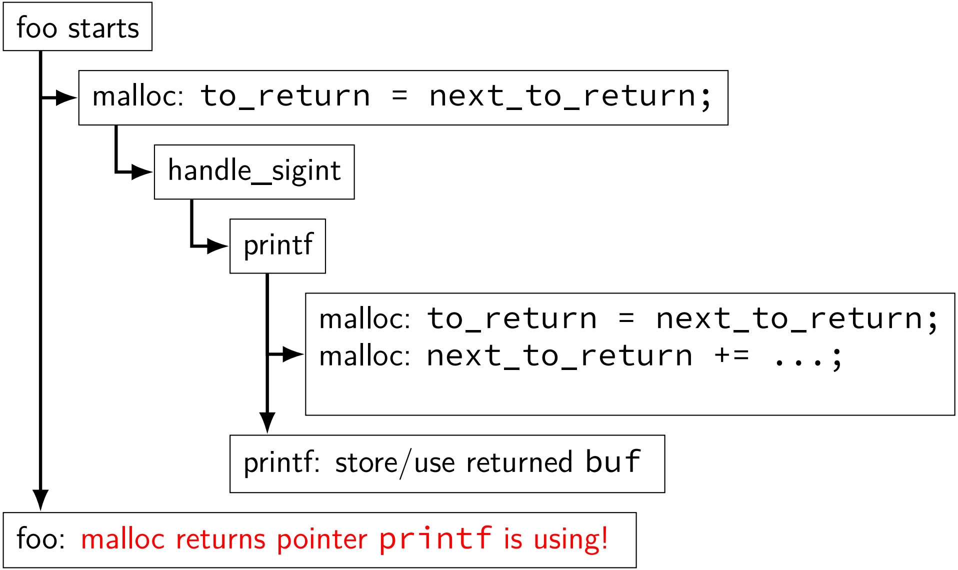
signal handler unsafety (1)

```
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

- `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

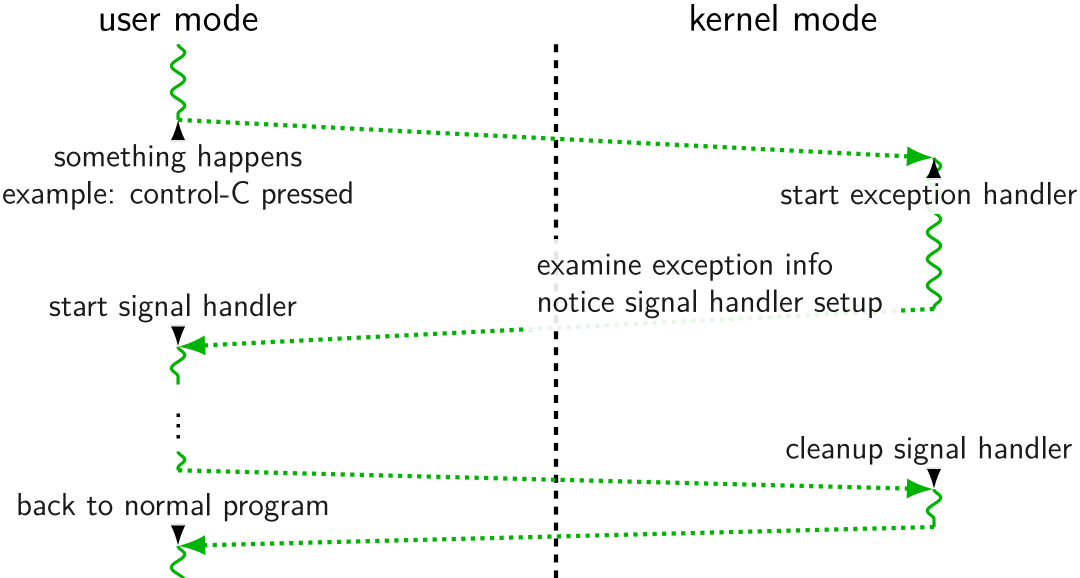
- some special functions to help:
- `sigsuspend` (unblock until handler runs),
- `pselect` (unblock while checking for I/O), …

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

‘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)

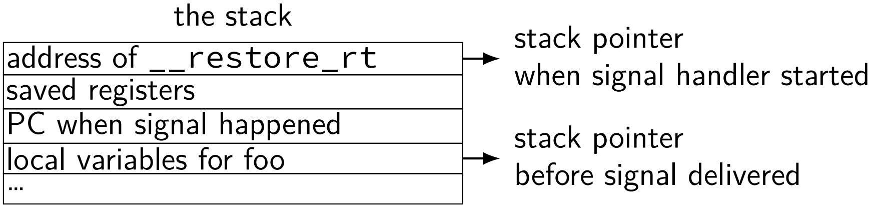
(and foo() doesn't have code to do that)

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 `__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 (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)