# 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)
- I/O 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)

# 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

```
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);
    }
}
```

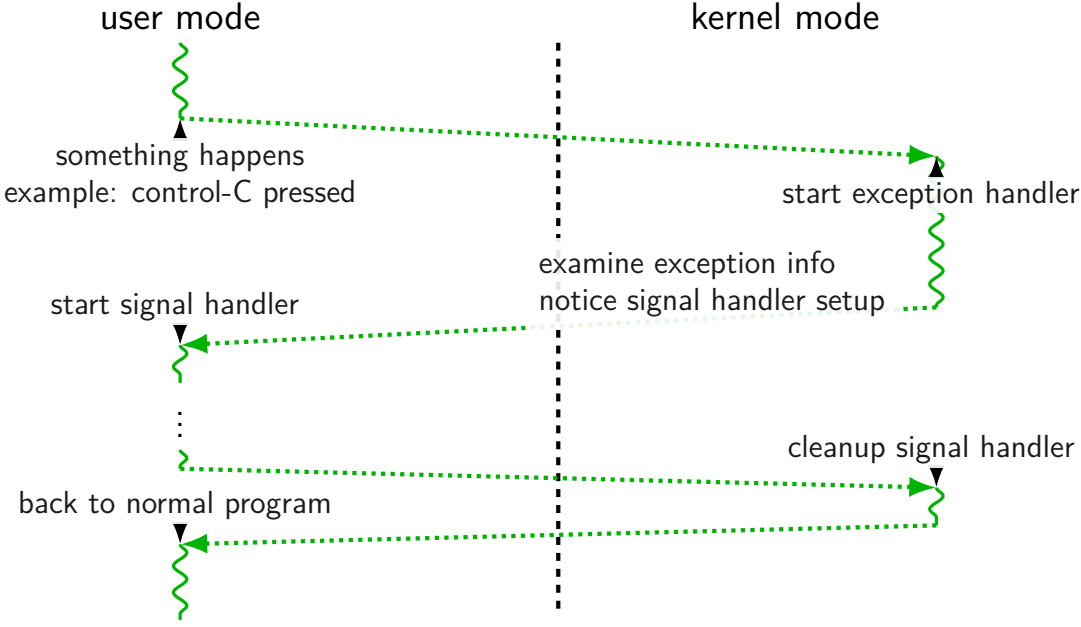
# example signal program

```
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



# 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

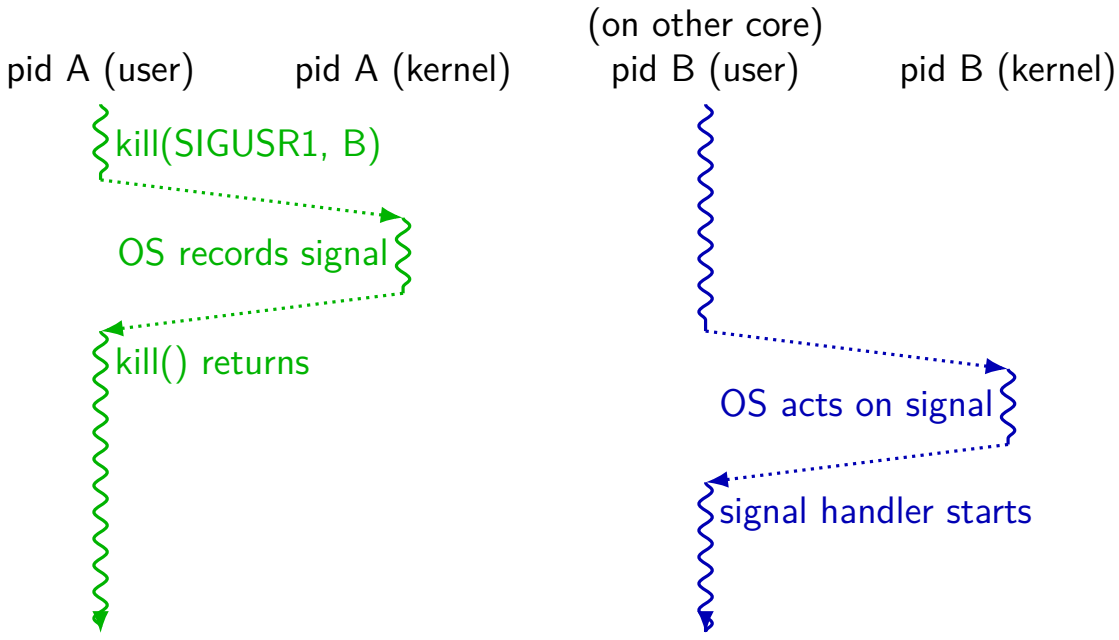
`kill 1234` — sends SIGTERM to pid 1234

in C: `kill(1234, SIGTERM)`

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

in C: `kill(1234, SIGUSR1)`

# kill() not always immediate



# 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)

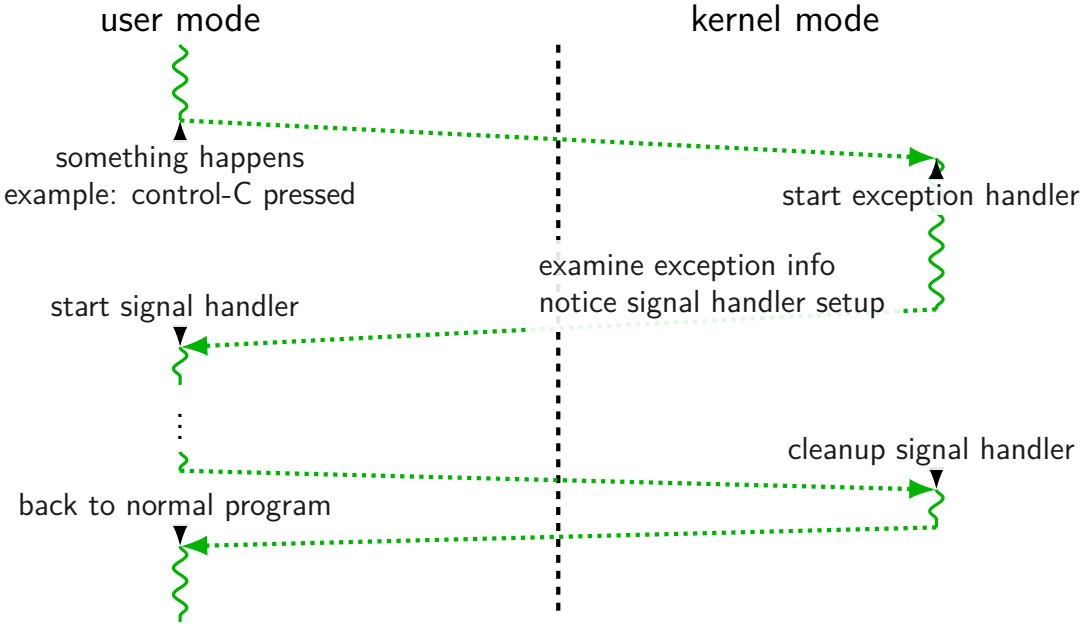
pid 1000

```
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)

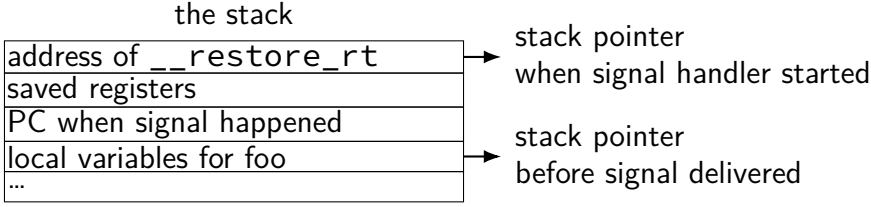
(and foo() doesn't have code to do that)

# 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



# 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");
}
```

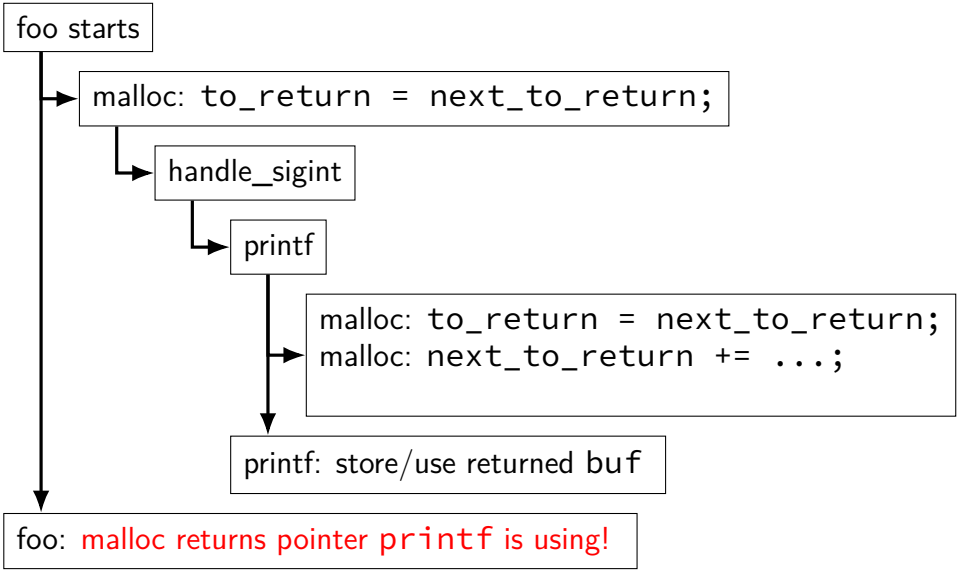
# signal handler unsafety (1)

```
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