last time (1)

exceptions: way for hardware to run OS

- OS sets up table of *exception handlers*
- hardware jumps to exception handler
- runs exception handler in kernel mode
- typically OS returns to user mode on return
- external events (I/O, timers)
- internal events (system calls, out-of-bounds access, ...)

time multiplexing + threads

- divide up time
- when OS runs (via exception), can decide to switch
- thread = illusion of own CPU

last time (2)

context switch

- switch thread on CPU by restoring saved register/etc. values and saving current register/etc. values for later switch back
- restore registers/etc. values saved a while ago
- typically also switch address space (program → real addrs)
- typically switching stacks

process = thread(s) + address space

(start) signals: kinda like exceptions for normal programs

some anonymous feedback [edited for space]

“...It has only been two classes but we are all struggling to keep up with the pace- which we are worried about since Professor Reiss said ”he was hoping he would move faster”. it is very difficult to take notes at the pace that Professor Reiss speaks/ flips between slides. Even with doing the reading, all my attention has to go to either taking notes ...and missing out on understanding the information, or ...and having to rewatch the lecture later...I would really appreciate if the pace was slowed down slightly...”

> yes, I didn’t cover as much as expected — so some topics were dropped
>
> please ask questions/slow me down

“...We were not given guidance on what ”expected output” should be- this was really helpful for the 2130 labs...”

> for the make lab, there’s a lot of outputs that would be okay...

quiz Q1

wrong prerequisties:

```
    lookup.c: lookup.h main.c
```

prerequisite is overwritten by rule:

```
    lookup.o: lookup.c lookup.h
    (tab)python generate_lookup.py >lookup.c
    ...
```

quiz Q2/3

A asks to read from keyboard, but *no input available*

- needs OS help, explicit request — system call

B does some computation

B sends a signal to process C

- I should've dropped this (we didn't really cover signals yet)
- requires system call, since C can't access process B's stuff directly

key pressed, causing A to run

- non-system call exception (from keyboard I/O)

A acceeses invalid memory location and is terminated

- non-system call exception (from invalid memory access)

C's signal handler runs and prints message (system call)

quiz Q4

printf(..., x / 0);

local variables from printf? — no

- printf not called yet!

buffer on the stack — yes

kernel mode — yes, what processor does for exceptions

quiz Q5

adjusting stack pointer

- user: subtract instruction — no memory access

reading input from the keyboard

- kernel: don't allow programs to directly talk to potentially shared devices

converting buffer from string to integer and vice-versa

- user: just computation

returning from printf

- user: printf mostly runs in user mode (even though it makes system calls internally)

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

```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
```

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

`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

```
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

```
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?

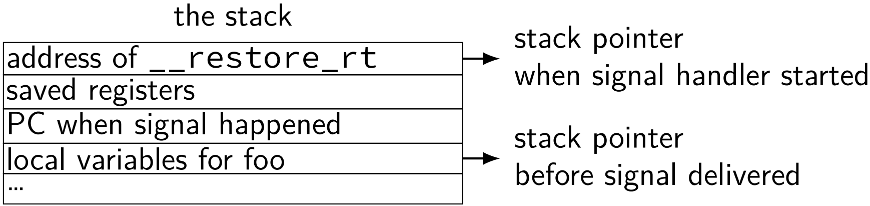
A. XY
B. X
C. Y
D. YX
E. X or XY, depending on timing
F. crash
G. (nothing)
H. something else

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");
}
```

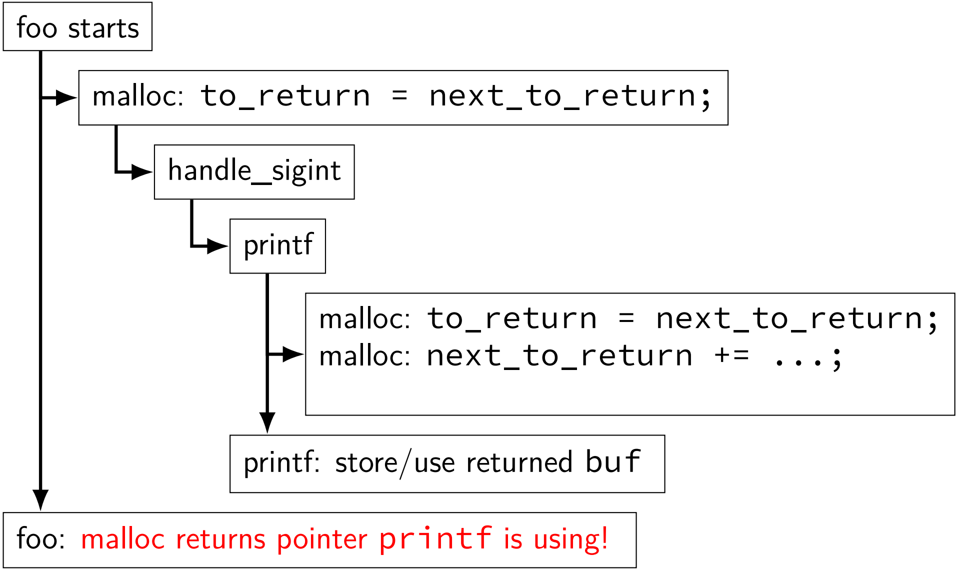
signal handler unsafety (1)

```
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");
    }
}
```

lab

program-to-program chat with shared memory + signals

has to be on *one machine* and with *same user*

timing HW

individual homework

time a bunch of things

- function call
- system call
- starting signal handler
- running command in the shell
- sending signal to process and waiting for it to send signal back

---

don't expect this to be really autograded

I think the length is appropriate (since signals lab will help with two of the items)

...but hasn't been done before

opening a file?

```
open("/u/creiss/private.txt", O_RDONLY)
```

say, private file on portal

on Linux: makes *system call*

kernel needs to decide if this should work or not

how does OS decide this?

argument: needs extra metadata

what would be wrong using...

system call arguments?

where the code calling open came from?

authorization v authentication

authentication — who is who

authorization v authentication

authentication — who is who

authorization — who can do what

- probably need authentication first...

authentication

password

hardware token

...

user IDs

most common way OSes identify what *domain* process belongs to:

(unspecified for now) procedure sets user IDs

every process has a user ID

user ID used to decide what process is authorized to do

POSIX user IDs

```
uid_t geteuid(); // get current process's "effective" user ID
```

process's user identified with unique number

kernel typically only knows about number

effective user ID is used for all permission checks

also some other user IDs — we'll talk later

POSIX user IDs

```
uid_t geteuid(); // get current process's "effective" user ID
```

process's user identified with unique number

kernel typically only knows about number

effective user ID is used for all permission checks

also some other user IDs — we'll talk later

standard programs/library maintain number to name mapping

- `/etc/passwd` on typical single-user systems
- network database on department machines

backup slides

setjmp/longjmp

```
jmp_buf env;

main() {
  if (setjmp(env) == 0) { // like try {
    ...
    read_file()
    ...
  } else { // like catch
    printf("some error happened\n");
  }
}

read_file() {
  ...
  if (open failed) {
      longjmp(env, 1) // like throw
  }
  ...
}
```

implementing setjmp/longjmp

setjmp:

- copy all registers to `jmp_buf`
- … including stack pointer

longjmp

- copy registers from `jmp_buf`
- … but change `%rax` (return value)

setjmp psuedocode

setjmp: looks like first half of context switch

```
setjmp:
  movq %rcx, env—>rcx
  movq %rdx, env—>rdx
  movq %rsp + 8, env—>rsp // +8: skip return value
  ...
  save_condition_codes env—>ccs
  movq 0(%rsp), env—>pc
  movq $0, %rax // always return 0
  ret
```

longjmp psuedocode

longjmp: looks like second half of context switch

```
longjmp:
  movq %rdi, %rax // return a different value
  movq env->rcx, %rcx
  movq env->rdx, %rdx
  ...
  restore_condition_codes env->ccs
  movq env->rsp, %rsp
  jmp env->pc
```

setjmp weirdness — local variables

Undefined behavior:

```
int x = 0;
if (setjmp(env) == 0) {
    ...
    x += 1;
    longjmp(env, 1);
} else {
    printf("%d\n", x);
}
```

setjmp weirdness — fix

Defined behavior:

```
volatile int x = 0;
if (setjmp(env) == 0) {
    ...
    x += 1;
    longjmp(env, 1);
} else {
    printf("%d\n", x);
}
```

on implementing try/catch

could do something like setjmp()/longjmp()

but setjmp is slow

setjmp exercise

```
jmp_buf env; int counter = 0;
void bar() {
    putchar('Z');
    ++counter;
    if (counter < 2) {
        longjmp(env, 1);
    }
}
int main() {
    while (setjmp(env) == 1) {
        putchar('X');
    }
    putchar('Y');
    bar();
}
```

Expected output?

A. YZ B. XYZ C. YZYZ D. XYZXYZ

setjmp exercise soln

```
jmp_buf env; int counter = 0;
void bar() {
    putchar('Z');               //           3 Z                12 Z
    ++counter;                  //           4                  13
    if (counter < 2) {          //           5 (1<2)            14 (2<2)
        longjmp(env, 1);        //           6*
    }                           //                              15
}
int main() {
    while (setjmp(env) == 1) {  // 0 (ret 0) 7*(ret 1) 9 (ret 0)
        putchar('X');           //           8 X
    }
    putchar('Y');               // 1 Y                 10 Y
    bar();                      // 2                   11
}                               //                              16
```

on implementing try/catch

could do something like setjmp()/longjmp()

but setjmp is slow

low-overhead try/catch (1)

```
main() {
  printf("about to read file\n");
  try {
    read_file();
  } catch(...) {
    printf("some error happened\n");
  }
}
read_file() {
  ...
  if (open failed) {
      throw IOException();
  }
  ...
}
```

low-overhead try/catch (2)

```
main:
  ...
  call printf
start_try:
  call read_file
end_try:
  ret
```

```
main_catch:
  movq $str, %rdi
  call printf
  jmp end_try
```

```
read_file:
  pushq %r12
  ...
  call do_throw
  ...
end_read:
  popq %r12
  ret
```

lookup table

program counter range	**action**	**recurse?**
`start_try` to `end_try`	`jmp main_catch`	no
`read_file` to `end_read`	`popq %r12, ret`	yes
anything else	error	—

low-overhead try/catch (2)

```
main:
  ...
  call printf
start_try:
  call read_file
end_try:
  ret
```

```
main_catch:
  movq $str, %rdi
  call printf
  jmp end_try
```

```
read_file:
  pushq %r12
  ...
  call do_throw
  ...
end_read:
  popq %r12
  ret
```

lookup table

program counter range	**action**	**recurse?**
`start_try` to `end_try`	`jmp main_catch`	no
`read_file` to `end_read`	`popq %r12, ret`	yes
anything else	error	—

low-overhead try/catch (2)

```
main:
  ...
  call printf
start_try:
  call read_file
end_try:
  ret
```

```
main_catch:
  movq $str, %rdi
  call printf
  jmp end_try
```

```
read_file:
  pushq %r12
  ...
  call do_throw
  ...
end_read:
  popq %r12
  ret
```

lookup table

program counter range	**action**	**recurse?**
`start_try` to `end_try`	`jmp main_catch`	no
`read_file` to `end_read`	`popq %r12, ret`	yes
anything else	error	—

low-overhead try/catch (2)

```
main:
   ...
   call printf
start_try:
   call read_file
end_try:
   ret
```

```
main_catch:
   movq $str, %rdi
   call printf
   jmp end_try
```

```
read_file:
   pushq %r12
   ...
   call do_throw
   ...
```

not actual x86 code to run
track a “virtual PC” while looking for catch block

lookup table

program counter range	**action**	**recurse?**
`start_try` to `end_try`	`jmp main_catch`	no
`read_file` to `end_read`	`popq %r12, ret`	yes
anything else	error	—

lookup table tradeoffs

no overhead if throw not used

handles local variables on registers/stack, but…

larger executables (probably)

extra complexity for compiler