Exceptions cont'd

rotate due tomorrow

recall: time on our testing machine

probably don't find out what that is at the last minute?

anonymous feedback (1)

“Your notes and the book contradict each other (especially this last quiz) so it’s hard to tell which to believe”

differences with the book that I don’t say are different are unintentional

specifics would really help — at least for future semesters

book:contexts

generally: contexts are what needs to change to switch threads/processes

but book includes “user stack” and “kernel stack” which is weird

short-hand for stack pointers? or just sloppy? or different definition?

book: saving PC, etc.

all CPUs save the the PC before starting exception handler

- my slides: “for example to special register”
- x86/book: special memory location

on x86: also save the stack pointer and set a new stack pointer

- new stack pointer is where CPU saves things (instead of special registers)

anonymous feedback (2)

(paraphrased) “Question 6 on the Post-quiz for week 12 should be dropped …The textbook never implies it is a function which is why no one knew that was the answer.”

would have preferred if I had it made it clearer that ‘process’ in Q was a vocab term

book: “An exception is akin to a procedure call with some important differences:”

Recall: Process

illusion of dedicated machine

thread + address space

thread = illusion of dedicated processor

address space = illusion of dedicated memory

Recall: thread

CPU:

loop.exe

loop.exe

illusion of dedicated processor

time multiplexing: operating system alternates which thread runs on the processor

- programs run concurrently on same CPU

mechanism for operating system to run: exceptions

Recall: thread

CPU:
loop.exe
loop.exe

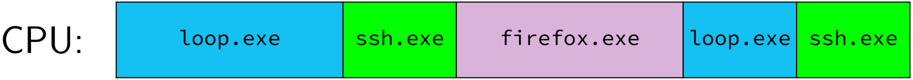
illusion of dedicated processor

time multiplexing: operating system alternates which thread runs on the processor

- programs run concurrently on same CPU

mechanism for operating system to run: exceptions

Recall: thread



CPU: loop.exe ssh.exe firefox.exe loop.exe ssh.exe

illusion of dedicated processor

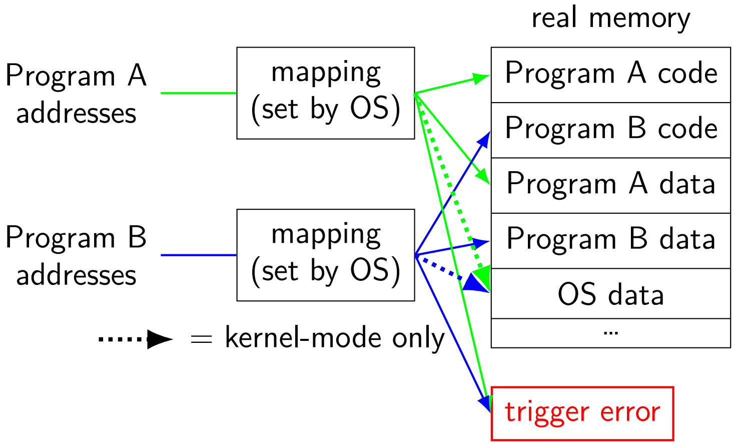
time multiplexing: operating system alternates which thread runs on the processor

- programs run concurrently on same CPU

mechanism for operating system to run: exceptions

Recall: address space

illuision of dedicated memory



Recall: protection

processes can't interfere with other processes

processes can't interfere with operating system

... except as allowed by OS

mechanism 1: kernel mode and privileged instructions

mechanism 2: address spaces

mechanism 3: exceptions for controlled access

kernel services

allocating memory? (change address space)

reading/writing to file? (communicate with hard drive)

read input? (communicate with keyborad)

all need privileged instructions!

need to run code in kernel mode

Linux x86-64 system calls

special instruction: `syscall`

triggers trap (deliberate exception)

Linux syscall calling convention

before `syscall`:

`%rax` — system call number

`%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9` — args

after `syscall`:

`%rax` — return value

on error: `%rax` contains -1 times "error number"

almost the same as normal function calls

Linux x86-64 hello world

```
.globl _start
.data
hello_str: .asciz "Hello,␣World!\n"
.text
_start:
  movq $1, %rax # 1 = "write"
  movq $1, %rdi # file descriptor 1 = stdout
  movq $hello_str, %rsi
  movq $15, %rdx # 15 = strlen("Hello, World!\n")
  syscall

  movq $60, %rax # 60 = exit
  movq $0, %rdi
  syscall
```

approx. system call handler

```
sys_call_table:
    .quad handle_read_syscall
    .quad handle_write_syscall
    // ...

handle_syscall:
    ... // save old PC, etc.
    pushq %rcx // save registers
    pushq %rdi
    ...
    call *sys_call_table(,%rax,8)
    ...
    popq %rdi
    popq %rcx
    return_from_exception
```

Linux system call examples

`mmap`, `brk` — allocate memory

`fork` — create new process

`execve` — run a program in the current process

`_exit` — terminate a process

`open`, `read`, `write` — access files

terminals, etc. count as files, too

system calls and protection

exceptions are only way to access kernel mode

operating system controls what proceses can do

… by writing exception handlers very carefully

careful exception handlers

```
movq $important_os_address, %rsp
```

can't trust user's stack pointer!

need to have own stack in kernel-mode-only memory

need to check all inputs really carefully

protection and sudo

programs always run in user mode

extra permissions from OS do not change this

- sudo, superuser, root, SYSTEM, …

operating system may remember extra privileges

system call wrappers

library functions to not write assembly:

```
open:
    movq $2, %rax // 2 = sys_open
    // 2 arguments happen to use same registers
    syscall
    // return value in %eax
    cmp $0, %rax
    jl has_error
    ret
has_error:
    neg %rax
    movq %rax, errno
    movq $−1, %rax
    ret
```

system call wrappers

library functions to not write assembly:

```
open:
    movq $2, %rax // 2 = sys_open
    // 2 arguments happen to use same registers
    syscall
    // return value in %eax
    cmp $0, %rax
    jl has_error
    ret
has_error:
    neg %rax
    movq %rax, errno
    movq $−1, %rax
    ret
```

system call wrapper: usage

```
/* unistd.h contains definitions of:
    O_RDONLY (integer constant), open() */
#include <unistd.h>
int main(void) {
  int file_descriptor;
  file_descriptor = open("input.txt", O_RDONLY);
  if (file_descriptor < 0) {
      printf("error:␣%s\n", strerror(errno));
      exit(1);
  }
  ...
  result = read(file_descriptor, ...);
  ...
}
```

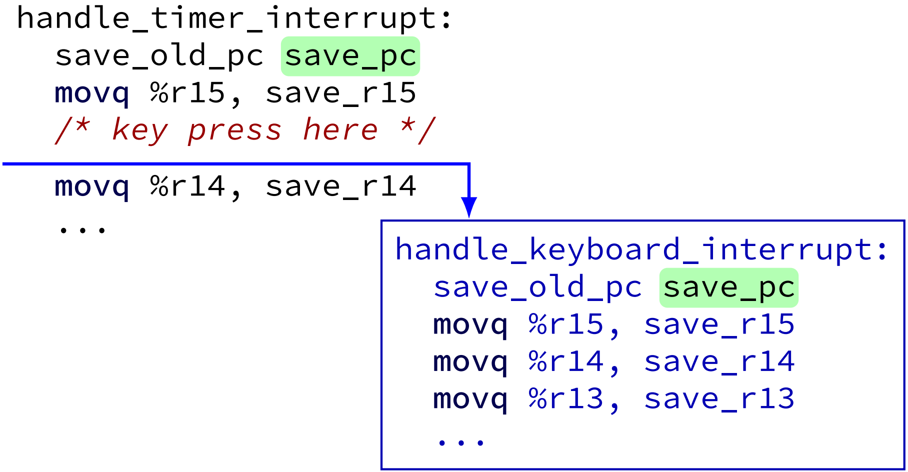
system call wrapper: usage

```
/* unistd.h contains definitions of:
    O_RDONLY (integer constant), open() */
#include <unistd.h>
int main(void) {
  int file_descriptor;
  file_descriptor = open("input.txt", O_RDONLY);
  if (file_descriptor < 0) {
      printf("error:␣%s\n", strerror(errno));
      exit(1);
  }
  ...
  result = read(file_descriptor, ...);
  ...
}
```

exceptions in exceptions

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

  movq %r14, save_r14
  ...
```

exceptions in exceptions



```
handle_timer_interrupt:
  save_old_pc save_pc
  movq %r15, save_r15
  /* key press here */
  movq %r14, save_r14
  ...
```

```
handle_keyboard_interrupt:
  save_old_pc save_pc
  movq %r15, save_r15
  movq %r14, save_r14
  movq %r13, save_r13
  ...
```

exceptions in exceptions

```
handle_timer_interrupt:
  save_old_pc save_pc
  movq %r15, save_r15
  /* key press here */
  movq %r14, save_r14
  ...
```

```
handle_keyboard_interrupt:
  save_old_pc save_pc
  movq %r15, save_r15
  movq %r14, save_r14
  movq %r13, save_r13
  ...
```

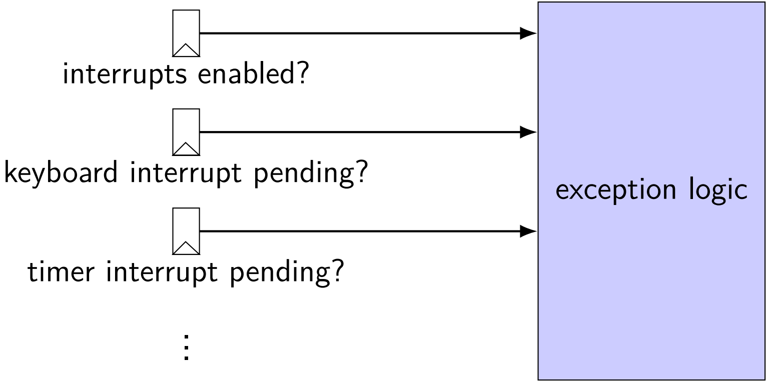
solution: disallow this!

interrupt disabling

CPU supports disabling (most) interrupts

interrupts will wait until it is reenabled

CPU has extra state:



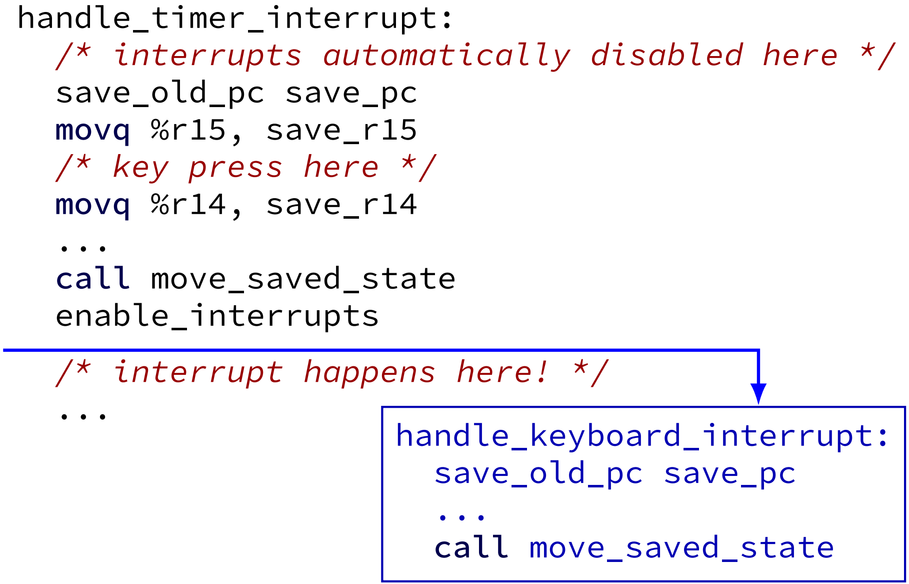
exceptions in exceptions

```
handle_timer_interrupt:
  /* interrupts automatically disabled here */
  save_old_pc save_pc
  movq %r15, save_r15
  /* key press here */
  movq %r14, save_r14
  ...
  call move_saved_state
  enable_interrupts

  /* interrupt happens here! */
  ...
```

exceptions in exceptions

```
handle_timer_interrupt:
  /* interrupts automatically disabled here */
  save_old_pc save_pc
  movq %r15, save_r15
  /* key press here */
  movq %r14, save_r14
  ...
  call move_saved_state
  enable_interrupts

  /* interrupt happens here! */
  ...
```

exceptions in exceptions



```
handle_timer_interrupt:
  /* interrupts automatically disabled here */
  save_old_pc save_pc
  movq %r15, save_r15
  /* key press here */
  movq %r14, save_r14
  ...
  call move_saved_state
  enable_interrupts
  /* interrupt happens here! */
  ...
```

```
handle_keyboard_interrupt:
  save_old_pc save_pc
  ...
  call move_saved_state
```

disabling interrupts

automatically disabled when exception handler starts

also done with privileged instruction:

```
change_keyboard_parameters:
  disable_interrupts
  ...
  /* change things used by
     handle_keyboard_interrupt here */
  ...
  enable_interrupts
```

a note on terminology (1)

real world: inconsistent terms for exceptions

we will follow textbook's terms in this course

the real world won't

you might see:

- 'interrupt' meaning what we call 'exception' (x86)
- 'exception' meaning what we call 'fault'
- 'hard fault' meaning what we call 'abort'
- 'trap' meaning what we call 'fault'
- ... and more

a note on terminology (2)

we use the term “kernel mode”

some additional terms:

- supervisor mode
- privileged mode
- ring `0`

some systems have multiple levels of privilege

- different sets of priviliged operations work

on virtual machines

process can be called a ‘virtual machine’

programmed like a complete computer…

on virtual machines

process can be called a ‘virtual machine’

programmed like a complete computer…

but weird interface for I/O, memory — system calls

can we make that closer to the real machine?

trap-and-emulate

privileged instructions trigger a protection fault

we assume operating system crashes

what if OS pretends the privileged instruction works?

trap-and-emulate: write-to-screen

```
struct Process {
    AddressSpace address_space;
    SavedRegisters registers;
};

void handle_protection_fault(Process *process) {
    // normal: would crash
    if (was_write_to_screen()) {
        do_write_system_call(process);
        process->registers->pc +=
            WRITE_TO_SCREEN_LENGTH;
    } else {
        ...
    }
}
```

trap-and-emulate: write-to-screen

```
struct Process {
    AddressSpace address_space;
    SavedRegisters registers;
};

void handle_protection_fault(Process *process) {
    // normal: would crash
    if (was_write_to_screen()) {
        do_write_system_call(process);
        process->registers->pc +=
            WRITE_TO_SCREEN_LENGTH;
    } else {
        ...
    }
}
```

was_write_to_screen()

how does OS know what caused protection fault?

option 1: hardware “type” register

option 2: check instruction:

```
int opcode = (*process->registers->pc & 0xF0) >> 4;
if (opcode == WRITE_TO_SCREEN_OPCODE)
    ...
```

trap-and-emulate: write-to-screen

```
struct Process {
    AddressSpace address_space;
    SavedRegisters registers;
};

void handle_protection_fault(Process *process) {
    // normal: would crash
    if (was_write_to_screen()) {
        do_write_system_call(process);
        process->registers->pc +=
            WRITE_TO_SCREEN_LENGTH;
    } else {
        ...
    }
}
```

trap-and-emulate: write-to-screen

```
struct Process {
    AddressSpace address_space;
    SavedRegisters registers;
};

void handle_protection_fault(Process *process) {
    // normal: would crash
    if (was_write_to_screen()) {
        do_write_system_call(process);
        process->registers->pc +=
            WRITE_TO_SCREEN_LENGTH;
    } else {
        ...
    }
}
```

system virtual machines

turn faults into system calls

emulate machine that looks more like 'real' machine

what software like VirtualBox, VMWare, etc. does

more complicated than this:

- on x86, some privileged instructions don't cause faults
- dealing with address spaces is a lot of extra work

process VM versus system VM

Linux process feature	real machine feature
files, sockets	I/O devices
threads	CPU cores
mmap/brk (used by malloc)	???
signals	exceptions

signals

Unix-like operating system feature

like interrupts for processes:

can be triggered by external process

- kill command/system call

can be triggered by special events

- pressing control-C
- faults

can invoke signal handler (like exception handler)

signal API

`sigaction` — register handler for signal

`kill` — send signal to process

`pause` — put process to sleep until signal received

`sigprocmask` — temporarily block some signals from being received

… and much more

example signal program

```
void handle_sigint(int signum) {
    write(1, "Got␣signal!\n", sizeof("Got␣signal!\n"));
    _exit(0);
}

int main(void) {
    struct sigaction act;
    act.sa_handler = &handle_sigint;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    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) {
    write(1, "Got␣signal!\n", sizeof("Got␣signal!\n"));
    _exit(0);
}

int main(void) {
    struct sigaction act;
    act.sa_handler = &handle_sigint;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGINT, &act, NULL);

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

example signal program

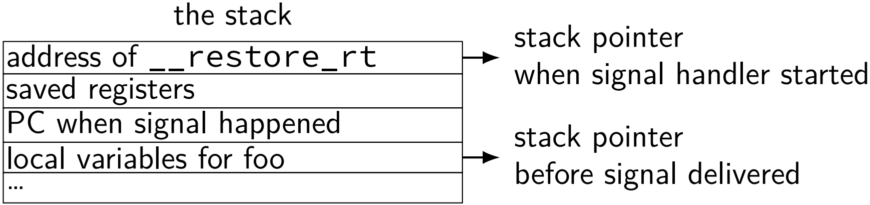
```
void handle_sigint(int signum) {
    write(1, "Got␣signal!\n", sizeof("Got␣signal!\n"));
    _exit(0);
}

int main(void) {
    struct sigaction act;
    act.sa_handler = &handle_sigint;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGINT, &act, NULL);

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

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

- needed to handle caller-saved registers
- also might unblock signals (like un-disabling interrupts)

example signals

signal	default action	description
SIGINT	terminate	control-C
SIGHUP	terminate	terminal closed
SIGTERM	terminate	request termination
SIGTSTP	stop	control-Z
SIGSEGV	terminate	Segmentation fault
SIGILL	terminate	Illegal instruction

example signals

signal	default action	description
SIGINT	terminate	control-C
SIGHUP	terminate	terminal closed
SIGTERM	terminate	request termination
SIGTSTP	stop	control-Z
SIGSEGV	terminate	Segmentation fault
SIGILL	terminate	Illegal instruction

example signals

signal	default action	description
SIGINT	terminate	control-C
SIGHUP	terminate	terminal closed
SIGTERM	terminate	request termination
SIGTSTP	stop	control-Z
SIGSEGV	terminate	Segmentation fault
SIGILL	terminate	Illegal instruction

reflecting exceptions

Linux turns faults into signals

allows process's signal handler to try running, e.g.:

save a debug log when crashing

emulate a missing instruction

special signals

SIGKILL — always terminates a process

SIGSTOP — always stops a process

both cannot have a signal handler

- might register one, but will never be called

blocking signals

avoid having signal handlers anywhere:

can instead block signals

`sigprocmask` system call

signal will become "pending" instead

OS will not deliver unless unblocked

analagous to disabling interrupts

alternatives to signal handlers

first, block a signal

then use system calls to inspect pending signals

- example: `sigwait`

or unblock signals only when waiting for I/O

- example: `pselect` system call

synchronous signal handling

```
int main(void) {
    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    sigprocmask(SIG_BLOCK, SIGINT);

    printf("Waiting␣for␣SIGINT␣(control-C)\n");
    if (sigwait(&set, NULL) == 0) {
        printf("Got␣SIGINT\n");
    }
}
```

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");
}
```

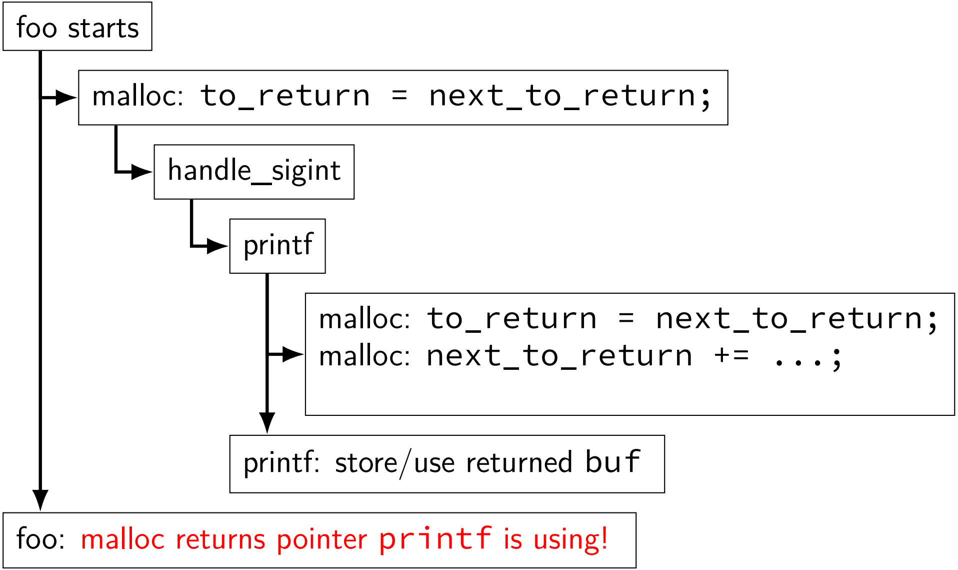
signal handler unsafety (1)

```
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 in signal handlers no matter what they interrupt

includes: `write`, `_exit`

does not include: `printf`, `malloc`, `exit`

summary

exceptions — mechanism to for OS to run

- to help out user programs
- in response to external events
- in repsonse to errors

process — “virtual machine” illusion

- thread + address space

signals — process analogy to exceptions

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

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