changelog

5 Sep 2023: ‘output of this?’: fix code to consistently use handle_usr1 instead of multiple names for signal handler

last time

context switch — save current/restore old context

- context = state on processor
- (registers, program counter, ...)

thread = illusion of own processor

- has own PC, registers, etc.
- typically implemented by potentially sharing processors

process = thread(s) + address space (illusion of "own machine")

- as if separated from other programs

hardware:Unix OS::exceptions::signals

- signal handlers called by OS interrupting thread
- way for OS to ask program for help; often "forward" exception

anonymous feedback (1)

"All but 3 TAs left within the first 30 minutes of lab."

- think this is 6:30pm lab
- only 3 TAs assigned to that lab (others staying late for 5pm)

"Rice 442 can get quite confusing during office hours. There are a lot of students getting help from a lot of different classes and it is loud/tight space. Is there any chance there be more options for discord oh or a different room in rice specifically for CSO2?"

- I'm hoping sign up (on whiteboard wall or using online queue) makes this not too bad
- we could do other room in Rice Friday afternoon if needed...

anonymous feedback (2a)

"Is it possible to have access to some more practice with lecture topics? The exercises we do in class are helpful, but there are so few of them that getting to the quizzes, even after reviewing slides/readings, feels like a huge jump. I feel like I'm not getting enough exposure to the topics (examples, questions, etc) before getting to quizzes - and because they are so high stakes (30% of overall grade) making mistakes doesn't feel like it is supporting my learning."

"The quizzes feel very tricky. I do the readings and pay attention in class, is there anything else I can do to help prepare? Any suggestions for improving understanding for the type of questions asked on the quizzes?"

"This quiz feels extremely hard, given what we learned in class. I also do the reading, but it just feels very hard."

anonymous feedback (2b)

"Can you provide more practice questions and class examples that are similar in difficulty to the quiz questions? I understand that we should be applying what we learn in the quiz but the first two have gone into much more detail than is provided in lectures or readings. I attend class and do all of the readings but I often find that the quiz content is still extremely difficult and not covered in class"

"I have really struggled with this second quiz despite attending lectures, reading the suggested readings, and some additional readings linked in the suggested readings. Are there any resources that you suggest we utilize moving forward?"

anonymous feedback (3)

"I was hoping you could be more consistent with the readings and the slides. There is a lot of discrepancy between information on the notes and the slides, as well as a lack of explanation for key concepts as they are at a high level, whereas assignments go into a much deeper level."

Kinda intentional that readings + slides present things differently

I can make guesses as to what's unclear/seems contradictory, but... hard to do much with few specifics

on quiz review generally

seems people weren't as comfortable re: exceptions as I thought

- also some questions had corner cases/other interpretations I didn't anticipate

more generally:

- have past quizzes as additional examples ('study materials' on website)
- longer-term I should add more examples to readings
- follow-up Qs on Piazza/office hours/etc. good ideas

quiz Q1

A: updating implementation requires modifying fewer files

- syscall: one file to update — compiled copy of printf code in OS kernel
  - (yes, need a reboot to do this, probably)
- dynamic library: one file to update — C library .so file
- static library: relink every program that uses printf

B: cannot read args from stack

- can still access user stack in kernel mode

D: display to screen without kernel mode

- usually accessing I/O only happens in kernel mode
- (yes, exceptions, but not very common)

quiz Q2 (context switches)

SSH client running

long computation running, +1 context switch

terminal running, +1 context switch

SSH running, +1 context switch

anonymous feedback (3)

"Can you be a little more clear about system calls and non system calls + examples because there seems to be a lot of overlap"

key difference: why did OS start running

- what OS does doesn't tell you
- (but could be hint)

quiz Q3 (non-syscall except)

usually no on outputting data

- need to get to kernel mode,
- but usually HW doesn't tell you when to output
- some exceptions, e.g., if need to wait for network/disk to be ready

usually for getting external input

- need to HW to say there is input

not for keypress getting from terminal to SSH client

- OS handles sending data, don't need processor help

quiz Q4 (syscall started)

Y: output data to I/O device/other program (1, 5, 6, 8)

Y: ask to wait to receive data (2)

N: for switching to other program after starting to wait (3)
system call happened earlier, being finished (not started)

N: for receiving data that was asked for earlier (4):
system call happened earlier, being finished (not started)

quiz Q5

out-of-bounds access triggers exception to run OS

Linux-like OS might decide to run signal handler

(but hardware doesn't know how to do that)

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)`

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	

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)

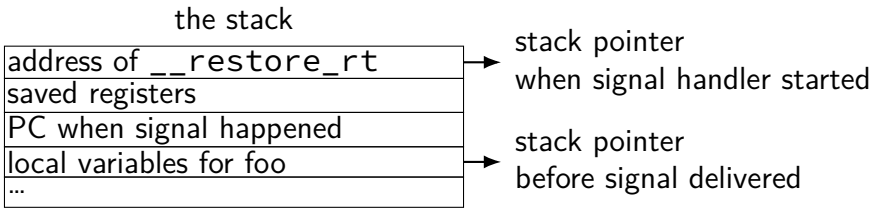
(and foo() doesn't have code to do that)

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");
}
```

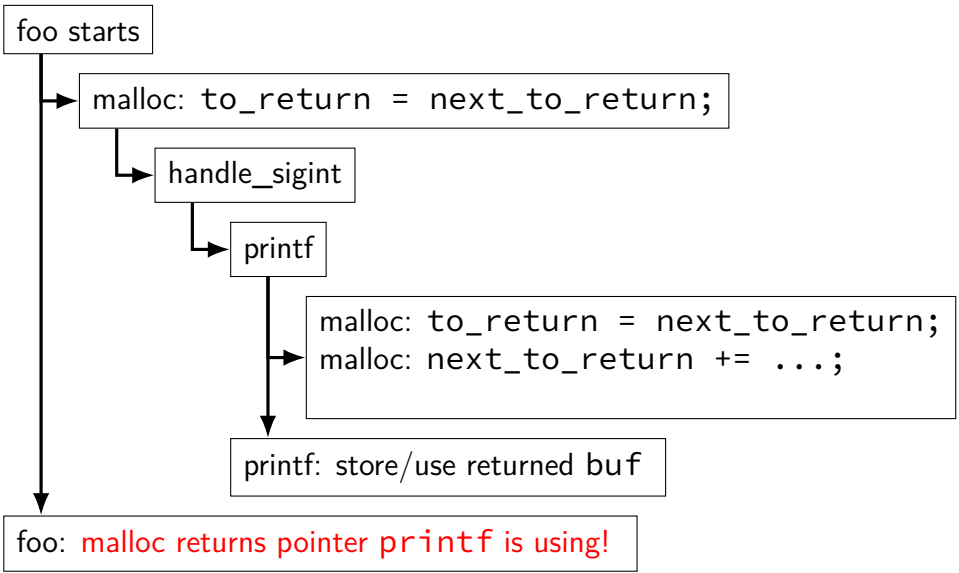
signal handler unsafety (1)

```
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");
    if (sigwait(&set, NULL) == SIGINT) {
        printf("Got SIGINT\n");
    }
}
```

backup slides

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);
    }
}
```

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!)
...	...