last time (1)

exceptions reviewed

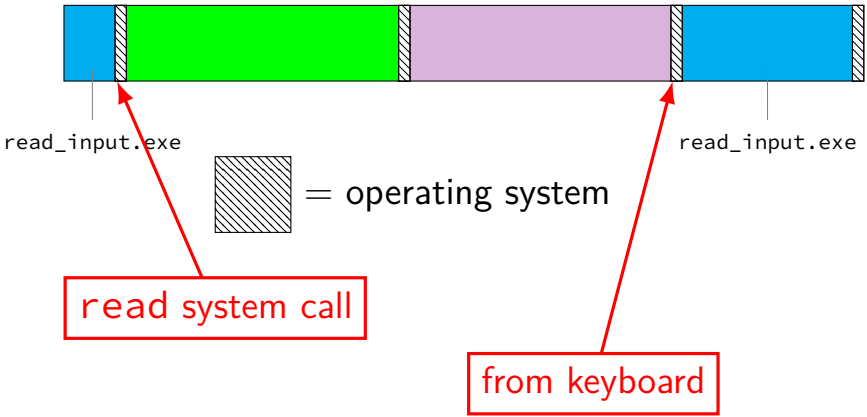
exceptions: way hardware runs OS

lots of things trigger hardware to do this

- program request (system call), I/O device,
  problematic event in program (bad memory access, etc.)

signal handling/safety/blocking

keyboard input timeline



exception patterns with I/O (1)

input — available now:

exception: device says "I have input now"
handler: OS stores input for later
exception (syscall): program says "I want to read input"
handler: OS returns that input

input — not available now:

exception (syscall): program says "I want to read input"
handler: OS runs other things (context switch)
exception: device says "I have input now"
handler: OS retrieves input
handler: (possibly) OS switches back to program that wanted it

exception patterns with I/O (2)

output — ready now:

exception (syscall): program says “I want to output this’
handler: OS sends output to deive

output — not ready now

exception (syscall): program says “I want to output”
handler: OS realizes device can’t accept output yet
(other things happen)
exception: device says “I’m ready for output now”
handler: OS sends output requested earlier

slightly more on quiz

accepted some alternate interpretations for whether "waiting" switches away from program

- imagined: ask OS to wait, it switches away
- but not crazy to think of checking something in a loop
- (especially b/c of some chat labs from CSO1 I didn't think about)

bunch of comments specifying "divide by zero" as likely

- I guess I need to rethink using the word "likely"?

anonymous feedback (1)

"In regards to the quizzes, I would respectfully request that the content better reflect what is taught in the lectures, slides, and readings. I've talked to numerous classmates and there is general consensus that what is on the quizzes often requires knowledge of deeper nuances not discussed or easily found in the previously mentioned resources. If there was a way for the grading to be more lenient or for quiz corrections to be offered as well I think that would help alleviate some anxiety that many of us feel towards this course as well as foster an environment that allows for adequate learning of concepts rather than the confusion we feel at the present moment."

"The quizzes were quite challenging, as the questions seemed a bit confusing. It wasn't clear whether we were expected to find the answers in our readings or lecture slides. Despite attending every class, I found it difficult to answer all the questions."

last time (2)

signal handling

- way OS calls program to handle "special" event
- no handler — often default exits program
- sigaction: register function to be called on signal
- kill: trigger signal in specified program

signal unsafety — libraries not written to be interrupted+called again

- malloc() → sighandler → malloc()

blocking signals

- normally: signals 'delivered' (run handler, crash program, etc.)
- while blocked, become 'pending' instead
- can check for/remove pending signals ('sigwait')
- or can unblock signal to let handler run

last time (2)

signal handling

- way OS calls program to handle "special" event
- no handler — often default exits program
- sigaction: register function to be called on signal
- kill: trigger signal in specified program

signal unsafety — libraries not written to be interrupted+called again

- malloc() → sighandler → malloc()

blocking signals

- normally: signals 'delivered' (run handler, crash program, etc.)
- while blocked, become 'pending' instead
- can check for/remove pending signals ('sigwait')
- or can unblock signal to let handler run

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();
}
```

on the lab

probably should supply pid-reading code next year

- adding paragraph about scanf issue in PID section of lab was not enough
  - issue: scanf a nubmer from "12(newline)" leaves "(newline)" so fgets works right away, sends before everyone ready

wasn't clear enough that munmap ('detach') was cleanup-code stuff

many issues re: segfaults, some options in general:

- `-g -fsanitize=address` (get line number of bug, usually)
- use debugger (automatic breakpoint at segfault+all other signals)

timing assignment

time how long each of these take:

- empty function call
- simple system call
- running another program
- starting signal handler
- sending signal and back

a note on kill() timing

kill(THIS PROCESS, ...) — signal handler runs before kill() returns

kill(OTHER PROCESS, ...) — signal handler may/may not start before kill() returns

anonymous feedback (2)

"The link to the Spring 2023 quizzes on the study materials tab of the course site is forbidden. Are we supposed to have access to this?"

> Yes, should be working now.

"I submitted the anonymous feedback about all but 3 TAs leaving, and I was talking about the 3:30 lab."

> will investigate what's going on (and think about trying to make TA help better organized in labs)

"Hi, I know you said that live examples aren't practical, but they really help me and others learn where we might make mistakes in our code. Also, as you explain them, I think we might understand the structure and how things interact in our code better. Looking at examples in reading only is pretty difficult to grasp."

> if I "just" do live examples, they'd probably be scripted so few mistakes probably some sort of pacing issue

"Please go slower in lecture! I know we have limited time, but so many people are still writing/typing as you go to the next slide."

anonymous feedback (3)

"I would appreciate it if you could not assume we know everything and explain the concepts to us. Assuming that we know a lot of this material is making us lost in lectures and assignments because as you dive into complex topics, we are falling behind on the basics. If you could be more explicit in your explanations and questions it would be highly appreciated."

"Charles, I just wanted to let you know that I don't think a single person I've talked to that's enrolled in this class feels confident about what they know so far. I think you're a very smart professor, and you're good at conveying information, but I feel like we are moving really fast for the amount of new content being introduced. I really feel like reviewing some of the content from the previous lecture would go a long way, maybe just a question or two at the beginning of class. I've been ahead on the readings since the class started and there are still points during lecture where I feel totally lost, followed by assignments that sometimes cover things I feel like weren't explained in depth in either the readings or the lecture. I really appreciate the time you give for students to ask questions, and I think you're fairly good at answering them, but I feel like a lot of us end up so confused we don't even know what questions to ask."

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?

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

POSIX groups

```
gid_t getegid(void);
    // process's"effective" group ID

int getgroups(int size, gid_t list[]);
    // process's extra group IDs
```

POSIX also has *group IDs*

like user IDs: kernel only knows numbers

- standard library+databases for mapping to names

also process has some other group IDs — we'll talk later

id

```
cr4bd@power4
: /net/zf14/cr4bd ; id
uid=858182(cr4bd) gid=21(csfaculty)
        groups=21(csfaculty),325(instructors),90027(cs4414)
```

id command displays uid, gid, group list

names looked up in database

- kernel doesn't know about this database
- code in the C standard library

groups that don't correspond to users

example: video group for access to monitor

put process in video group when logged in directly

don't do it when SSH'd in

groups that don't correspond to users

example: video group for access to monitor

put process in video group when logged in directly

don't do it when SSH'd in

...but: user can keep program running with video group in the background after logout?

POSIX file permissions

POSIX files have a very restricted access control list

one user ID + read/write/execute bits for user

"owner" — also can change permissions

one group ID + read/write/execute bits for group

default setting — read/write/execute

on directories, 'execute' means 'search' instead

permissions encoding

permissions encoded as 9-bit number, can write as octal: XYZ

octal divides into three 3-bit parts:

user permissions (X), group permissions (Y), other permission (Z)

each 3-bit part has a bit for 'read' (4), 'write' (2), 'execute' (1)

`700` — user read+write+execute; group none; other none

`451` — user read; group read+execute; other none

chmod — exact permissions

```
chmod 700 file
chmod u=rwx,og= file
```

user read write execute; group/others no accesss

```
chmod 451 file
chmod u=r,g=rx,o= file
```

user read; group read/execute; others no access

chmod — adjusting permissions

```
chmod u+rx foo
```

add user read and execute permissions
leave other settings unchanged

```
chmod o-rwx,u=rx foo
```

remove other read/write/execute permissions
set user permissions to read/execute
leave group settings unchanged

POSIX/NTFS ACLs

more flexible access control lists

list of (user or group, read or write or execute or ...)

supported by NTFS (Windows)

a version standardized by POSIX, but usually not supported

POSIX ACL syntax

```
# group students have read+execute permissions
group:students:r-x
# group faculty has read/write/execute permissions
group:faculty:rwx
# user mst3k has read/write/execute permissions
user:mst3k:rwx
# user tj1a has no permissions
user:tj1a:---

# POSIX acl rule:
    # user take precedence over group entries
```

POSIX ACLs on command line

```
getfacl file
```

```
setfacl -m 'user:tj1a:---' file
```

add line to ACL

```
setfacl -x 'user:tj1a' file
```

REMOVE line from acl

```
setfacl -M acl.txt file
```

add to acl, but read what to add from a file

```
setfacl -X acl.txt file
```

remove from acl, but read what to remove from a file

authorization checking on Unix

checked on system call entry

- no relying on libraries, etc. to do checks

files (open, rename, ...) — file/directory permissions

processes (kill, ...) — process UID = user UID

...

keeping permissions?

which of the following would still be secure?

A. performing authorization checks in the standard library in addition to system call handlers

B. performing authorization checks in the standard library instead of system call handlers

C. making the user ID a system call argument rather than storing it persistently in the OS's memory

superuser

user ID `0` is special

superuser or *root*

(non-Unix) or Administrator or SYSTEM or ...

some system calls: only work for uid `0`

shutdown, mount new file systems, etc.

automatically passes all (or almost all) permission checks

superuser v kernel mode

superuser : OS :: kernel mode : hardware

programs running as superuser still in user mode

- just change in how OS acts on system calls, etc.

how does login work?

```
somemachine login: jo
password: ********

jo@somemachine$ ls
...
```

this is a program which...

checks if the password is correct, and

changes user IDs, and

runs a shell

how does login work?

```
somemachine login: jo
password: ********

jo@somemachine$ ls
...
```

this is a program which...

checks if the password is correct, and

changes user IDs, and

runs a shell

Unix password storage

typical single-user system: `/etc/shadow`

- only readable by root/superuser

department machines: network service

- Kerberos / Active Directory:
- server takes (encrypted) passwords
- server gives tokens: "yes, really this user"
- can cryptographically verify tokens come from server

aside: beyond passwords

/bin/login entirely user-space code

only thing special about it: when it's run

could use any criteria to decide, not just passwords

- physical tokens
- biometrics
- ...

how does login work?

```
somemachine login: jo
password: ********

jo@somemachine$ ls
...
```

this is a program which...

checks if the password is correct, and

changes user IDs, and

runs a shell

changing user IDs

```
int setuid(uid_t uid);
```

if superuser: sets effective user ID to arbitrary value

- and a "real user ID" and a "saved set-user-ID" (we'll talk later)

system starts in/login programs run as superuser

- voluntarily restrict own access before running shell, etc.

sudo

```
tj1a@somemachine$ sudo restart
Password: *********
```

sudo: run command with superuser permissions

started by non-superuser

recall: inherits non-superuser UID

can't just call `setuid(0)`

set-user-ID sudo

extra metadata bit on *executables*: set-user-ID

if set: `exec()` syscall changes effective user ID to owner's ID

`sudo` program: owned by root, marked set-user-ID

marking setuid: `chmod u+s`

uses for setuid programs

mount USB stick

- setuid program controls option to kernel mount syscall
- make sure user can't replace sensitive directories
- make sure user can't mess up filesystems on normal hard disks
- make sure user can't mount new setuid root files

control access to device — printer, monitor, etc.

- setuid program talks to device + decides who can

write to secure log file

- setuid program ensures that log is append-only for normal users

bind to a particular port number < 1024

- setuid program creates socket, then becomes not root

set-user ID programs are very hard to write

what if stdin, stdout, stderr start closed?

what if signals setup weirldy?

what if the PATH env. var. set to directory of malicious programs?

what if `argc == 0`?

what if dynamic linker env. vars are set?

what if some bug allows memory corruption?

...

privilege escalation

privilege escalation — vulnerabilities that allow more privileges

code execution/corruption in utilities that run with high privilege

- e.g. buffer overflow, command injection
- login, sudo, system services, ...
- bugs in system call implementations

logic errors in checking delegated operations