last time

pointers in C

- pointer arithmetic and arrays
- conversion to/from pointer

libraries

- static (in executable)
- dynamic/shared (load from other file on program start)S

makefiles

- rules with dependencies

exercise: what will run?

```
W: X Y
►   buildW
X: Q
►   buildX
Y: X Z
►   buildY
```

W	modified 1 minute ago
X	modified 3 hours ago
Y	does not exist
Z	modified 1 hour ago
Q	modified 2 hours ago

exercise: "make W" will run what commands?

A. none
B. `buildY` only
C. `buildW` then `buildY`
D. `buildY` then `buildW`
E. `buildX` then `buildY` then `buildW`
F. `buildX` then `buildW`
G. something else

‘phony’ targets (1)

common to have Makefile targets that aren’t files

```
all: program1 program2 libfoo.a
```

“`make all`” effectively shorthand for “`make program1 program2 libfoo.a`”

no actual file called “all”

‘phony’ targets (2)

sometimes want targets that don’t actually build file

example: “`make clean`” to remove generated files

```
clean:
▶	rm --force main.o extra.o
```

but what if I create...

```
clean:
▶	rm --force main.o extra.o

all: program1 program2 libfoo.a
```

Q: if I make a file called “all” and then “make all” what happens?

Q: same with “clean” and “make clean”?

marking phony targets

```
clean:
▶       rm --force main.o extra.o

all: program1 program2 libfoo.a

.PHONY: all clean
```

special .PHONY rule says " 'all' and 'clean' not real files"

(not required by POSIX, but in every make version I know)

conventional targets

common convention:

target name	purpose
(default), `all`	build everything
`install`	install to standard location
`test`	run tests
`clean`	remove generated files

redundancy (1)

```
program: main.o extra.o
▶       clang -Wall -o program main.o extra.o

extra.o: extra.c extra.h
▶       clang -Wall -o extra.o -c extra.c
main.o: main.c main.h extra.h

▶       clang -o main.o -c main.c
```

what if I want to run `clang` with `-fsanitize=address` instead of `-Wall`?

what if I want to change `clang`to `gcc`?

variables/macros (1)

```
CC = gcc
CFLAGS = -Wall -pedantic -std=c11 -fsanitize=address
LDFLAGS = -Wall -pedantic -fsanitize=address
LDLIBS = -lm

program: main.o extra.o
▶       $(CC) $(LDFLAGS) -o program main.o extra.o $(LDLIBS)

extra.o: extra.c extra.h
▶       $(CC) $(CFLAGS) -o extra.o -c extra.c

main.o: main.c main.h extra.h
▶       $(CC) $(CFLAGS) -o main.o -c main.c
```

aside: conventional names

chose names `CC`, `CFLAGS`, `LDFALGS`, etc.

not required, but conventional names (incomplete list follows)

`CC`	C compiler
`CFLAGS`	C compiler options
`LDFLAGS`	linking options
`LIBS` or `LDLIBS`	libraries

variables/macros (2)

```
CC = gcc
CFLAGS = -Wall
LDFLAGS = -Wall
LDLIBS = -lm

program: main.o extra.o
▶       $(CC) $(LDFLAGS) -o $@ $^ $(LDLIBS)

extra.o: extra.c extra.h
▶       $(CC) $(CFLAGS) -o $@ -c $<

main.o: main.c main.h extra.h
▶       $(CC) $(CFLAGS) -o $@ -c $<
```

$@: target
$<: first dependency
$^: all dependencies

aside: $^ works on GNU make (usual on Linux), but not portable.

aside: make versions

multiple implementations of `make`

for stuff we've talked about so far, no differences

most common on Linux: GNU make

will talk about 'pattern rules', which aren't supported by some other make versions

- older, portable, (in my opinion less intuitive) alternative: suffix rules

pattern rules

```
CC = gcc
CFLAGS = -Wall
LDFLAGS = -Wall
LDLIBS = -lm

program: main.o extra.o
▶       $(CC) $(LDFLAGS) -o $@ $^ $(LDLIBS)

%.o: %.c
▶       $(CC) $(CFLAGS) -o $@ -c $<

extra.o: extra.c extra.h
main.o: main.c main.h extra.h
```

aside: these rules work on GNU make (usual on Linux), but less portable than suffix rules.

built-in rules

'make' has the 'make .o from .c' rule built-in already, so:

```
CC = gcc
CFLAGS = -Wall
LDFLAGS = -Wall
LDLIBS = -lm

program: main.o extra.o
▶       $(CC) $(LDFLAGS) -o $@ $^ $(LDLIBS)

extra.o: extra.c extra.h
main.o: main.c main.h extra.h
```

(don't actually need to write supplied rule!)

built-in rules

'make' has the 'make .o from .c' rule built-in already, so:

```
CC = gcc
CFLAGS = -Wall
LDFLAGS = -Wall
LDLIBS = -lm

program: main
▶       $(CC) $(LDFLAGS) -o $@ $^ $(LDLIBS)

extra.o: extra.c extra.h
main.o: main.c main.h extra.h
```

(don't actually need to write supplied rule!)

note: built-in rules not allowed on the make lab

writing Makefiles?

error-prone to write all .h dependencies

`-MM` (and related) options to `gcc` or `clang`

- outputs make rule
- ways of having make run this + use output

Makefile generators

- other programs that write Makefiles

other build systems

alternatives to writing Makefiles:

other make-ish build systems

- ninja, scons, bazel, maven, xcodebuild, msbuild, ...

tools that generate inputs for make-ish build systems

- cmake, autotools, qmake, ...

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

What is a system call?

(Will discuss in more detail in next lecture)

Briefly - syscall *instruction* takes system call number as argument

Other arguments are placed in registers or on the stack

This instruction switches hardware into privileged mode and uses syscall number to find correct handler function

This function decides if the "open" call is allowed to proceed

Only implemented syscalls are supported - can't invoke fake syscall

Handlers written carefully (we hope) to ensure safe implementation and correct permission checking

how does OS decide whether syscall should proceed?

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 "who" process belongs to:

process = instance of running program (w/ own registers+memory)
(we'll be more specific about processes later)

(unspecified for now) procedure sets user IDs

every process has a user ID

user ID used to decide what process is authorized to do

user IDs

most common way OSes identify "who" process belongs to:

process = instance of running program (w/ own registers+memory)

(we'll be more specific about processes later)

(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

core part of OS only knows number (not name!)

- core, always loaded part of OS = "kernel"
- the part of the OS with extra privs with hardware
- the part of the OS that enforces program restrictions

effective user ID is used for all permission checks

also some other user IDs

POSIX user IDs

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

process's user identified with unique number

core part of OS only knows number (not name!)

- core, always loaded part of OS = "kernel"
- the part of the OS with extra privs with hardware
- the part of the OS that enforces program restrictions

effective user ID is used for all permission checks

also some other user IDs

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 (= core part of OS) 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 execute

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=x file
```

user read; group read/execute; others execute

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

request made to core part of OS = system call

handler for system calls checks permissions

- no relying on libraries, etc. to do checks

files (open, rename, ...) — file/directory permissions include UID or GID

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

...

superuser

user ID `0` is special

superuser or *root*

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

some OS funtionality: only work for uid `0`

shutdown, mount new file systems, etc.

automatically passes all (or almost all) permission checks

superuser v kernel mode

processor has two modes

- kernel mode (what core part of OS uses)
- user mode (every thing else)

programs running as superuser still in user mode

- just change in how OS acts when program asks for things

superuser : OS :: kernel mode : hardware

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

- "extra" user IDs track what original user was

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

- sudo's code: if (original user allowed) ...; else print error

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

backup slides

make

`make` — Unix program for "making" things...

...by running commands based on what's changed

what commands? based on *rules* in *makefile*

(text file called `makefile` or `Makefile` (no extension))

make rules

```
main.o: main.c main.h extra.h
▶           clang -Wall -c main.c
```

before colon: target(s) (file(s) generated/updated)

after colon: prerequisite(s) (also known as dependencies)

following lines prefixed by a tab character: command(s) to run

make runs commands if any prereq modified date after target

make rules

```
main.o: main.c main.h extra.h
▶           clang -Wall -c main.c
```

before colon: target(s) (file(s) generated/updated)

after colon: prerequisite(s) (also known as dependencies)

following lines prefixed by a tab character: command(s) to run

make runs commands if any prereq modified date after target

make rules

```
main.o: main.c main.h extra.h
▶           clang -Wall -c main.c
```

before colon: target(s) (file(s) generated/updated)

after colon: prerequisite(s) (also known as dependencies)

following lines prefixed by a tab character: command(s) to run

make runs commands if any prereq modified date after target

make rules

```
main.o: main.c main.h extra.h
▸           clang -Wall -c main.c
```

before colon: target(s) (file(s) generated/updated)

after colon: prerequisite(s) (also known as dependencies)

following lines prefixed by a tab character: command(s) to run

make runs commands if any prereq modified date after target

make rules

```
main.o: main.c main.h extra.h
▶	clang -Wall -c main.c
```

before colon: target(s) (file(s) generated/updated)

after colon: prerequisite(s) (also known as dependencies)

following lines prefixed by a tab character: command(s) to run

`make` runs commands if any prereq modified date after target

make rules

```
main.o: main.c main.h extra.h
▶           clang -Wall -c main.c
```

before colon: target(s) (file(s) generated/updated)

after colon: prerequisite(s) (also known as dependencies)

following lines prefixed by a tab character: command(s) to run

make runs commands if any prereq modified date after target

make rules

```
main.o: main.c main.h extra.h
▶           clang -Wall -c main.c
```

before colon: target(s) (file(s) generated/updated)

after colon: prerequisite(s) (also known as dependencies)

following lines prefixed by a tab character: command(s) to run

make runs commands if any prereq modified date after target

...after making sure prerequisites up to date

make rule chains

```
program: main.o extra.o
▶           clang -Wall -o program main.o extra.o

extra.o: extra.c extra.h
▶           clang -Wall -c extra.c

main.o: main.c main.h extra.h
▶           clang -Wall -c main.c
```

to *make* `program`, first...

update main.o and extra.o if they aren't

running make

"`make` *`target`*"

look in `Makefile` in current directory for rules
check if *`target`* is up-to-date
if not, rebuild it (and prerequisites, if needed) so it is

"`make` *`target1 target2`*"

check if both target1 and target2 are up-to-date
if not, rebuild it as needed so they are

"`make`"

if "*`firstTarget`*" is the first rule in Makefile,
same as '`make` *`firstTarget`*"