last time

page table permission bits (read/write/user-mode/...)

- checked like valid bit (except depends why PTE used)

copy-on-write

POSIX process API

fork — create new process

- copies current process
- returns twice — once in old process (parent), once in new process (child)

exec — make current process start running different program

- when successful, doesn't return

next week lab logistics

pagetable code review

next Wednesday

submission due just before

- you should have working code for code review

groups of 3

yes, feedback on:

- code organization + style

not for debugging, writing code for others, etc.

final pagetable submission shortly after

- suggest doing README, licenses, etc earlier

late policy and pagetable2/lab

pagetable2 submissions due BEFORE first lab time

normal late policy does not apply

normal late policy will apply to pagetable3 (week from Fri)

code review lab in-person only

if you can't/shouldn't be there, let me know

- can make ad-hoc alternate arrangements

anonymous feedback (1)

"I would really appreciate some sort of formula or process overview for calculating sizes of page tables. I find it hard to follow what you say in class when it is just specific examples and not a general formula."

usually, we've given the size of page tables

- as in "page tables take up one page"

I think confusion is:

- unit conversion (pages, bytes, entries)
- relating size to virtual page number part size

page table sizes and VPN sizes

N byte page table

if page table entries E bytes long...

N/E entry page table

$log_2(N/E)$ bits to index into page table

if a page table is 1 page in size (like assignment), then $N = 2^{\text{POBITS}}$

if multiple levels, need to bits to index into *each level*

anonymous feedback (2)

"Could we have some started code for the assignments? That would make it easier for us to do the coding as the write up is sometimes very vague."

intentional that we don't supply more than function prototypes
often multiple good implementation strategies
I'm not sure where the lack of clarity is that 'started code' would fix
(If something is actually vague, I want it documented in writeup + examples, not template code)

" Is it possible to not have a quiz over fall break in order for us to relax and/or catch up on other work like page tables..."

it's due on Thursday, so full day after break before it
less quizzes means quizzes worth more
trying to avoid mega-quiz covering 3 lectures of stuff

quiz Q1

64 $= 2^6$ bytes $\rightarrow$ 6 bit page offset

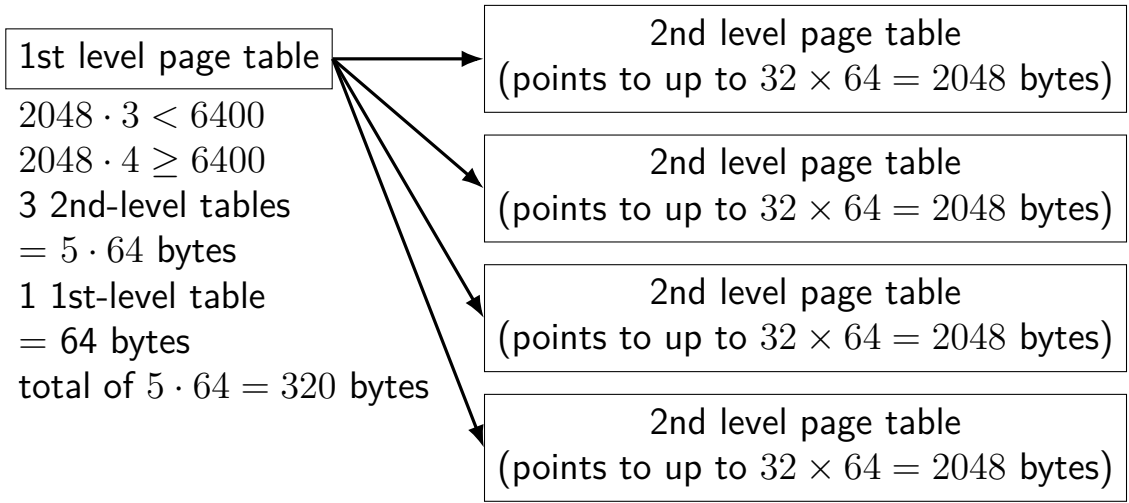
32 $= 2^5$ entries $=$ 5 bits per VPN part

0xABCD: [10101][01111][001101]

PTE address $=$ base address $+$ index $\times$ size

$=$ 0x3300 (PTBR) $+$ 10101 $\times$ 2 $=$ 0x332a

quiz Q2


1st level page table
$2048 \cdot 3 < 6400$
$2048 \cdot 4 \geq 6400$
3 2nd-level tables
$= 5 \cdot 64$ bytes
1 1st-level table
$= 64$ bytes
total of $5 \cdot 64 = 320$ bytes
2nd level page table
(points to up to $32 \times 64 = 2048$ bytes)
2nd level page table
(points to up to $32 \times 64 = 2048$ bytes)
2nd level page table
(points to up to $32 \times 64 = 2048$ bytes)
2nd level page table
(points to up to $32 \times 64 = 2048$ bytes)

quiz Q3

executable bit = executing code from that page?

(not for %rcx access)

quiz Q6

after both write — changes made so write worked

means both have independent copies

licenses.txt

part 3 of assignment

LICENSE / license.txt

- want you to understand — "free" code has conditions
- not a law class — I'm not qualified to say what conditions are legally enforceable, etc.
- understanding expectations authors have about how code should/should not be used
  - many things I would do without legal requirements

selected other part 3 things

README

deallocate

- implement something *or*
  tell us why it would be non-trivial or impractical to do so

POSIX process management

essential operations

process information: `getpid`

process creation: `fork`

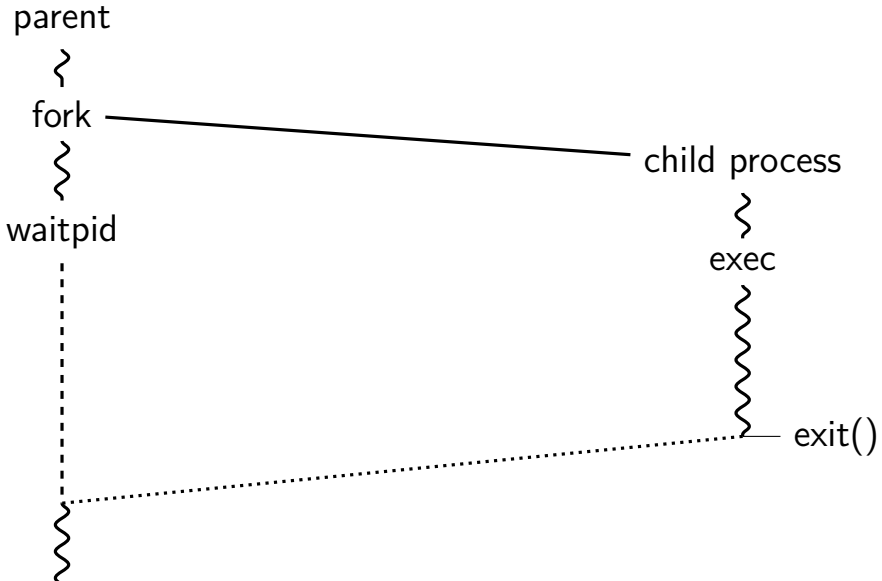
running programs: `exec*`

also `posix_spawn` (not widely supported), ...

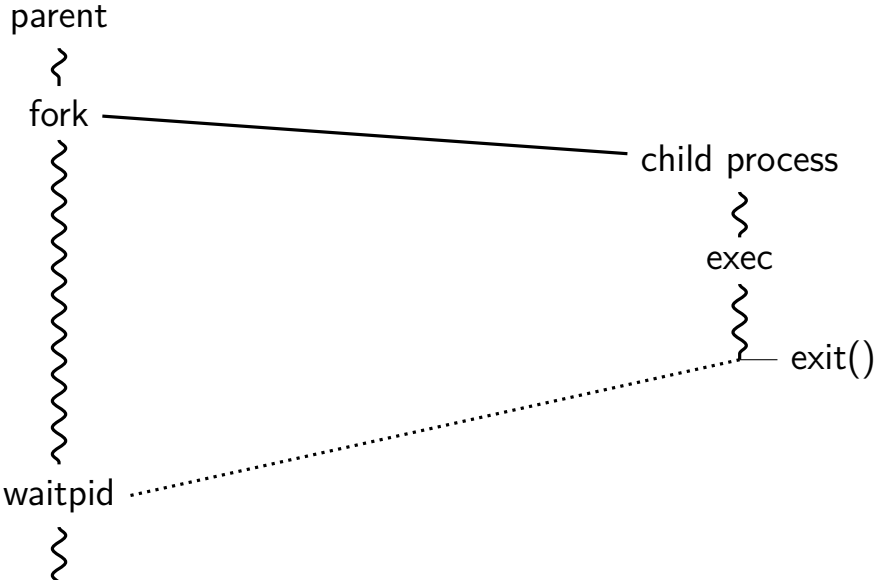
waiting for processes to finish: `waitpid` (or `wait`)

process destruction, 'signaling': `exit`, `kill`

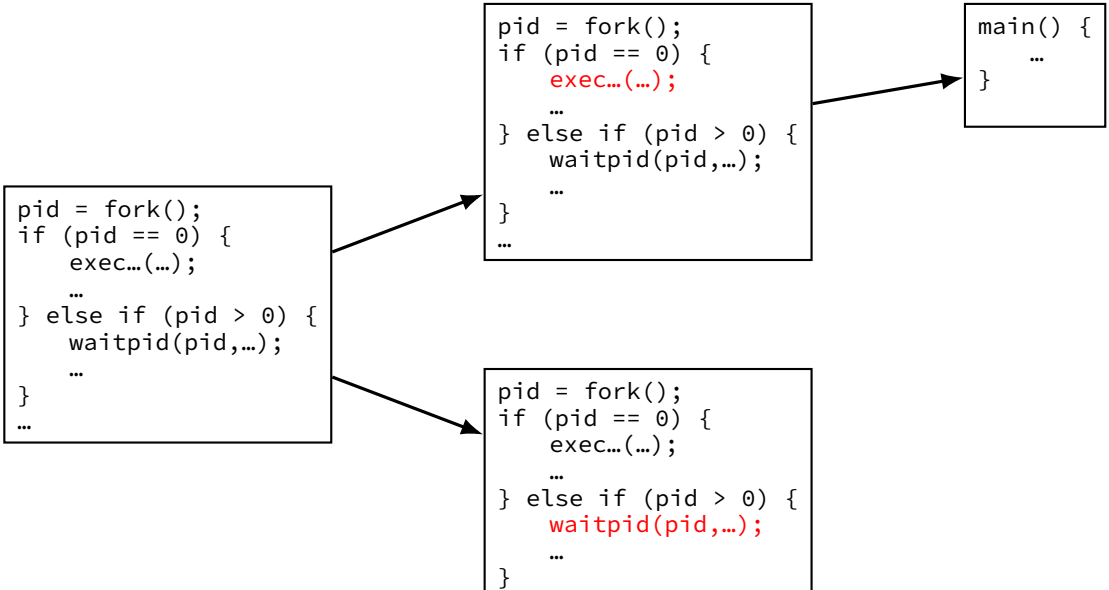
typical pattern



typical pattern (alt)



typical pattern (detail)



POSIX process management

essential operations

process information: `getpid`

process creation: `fork`

running programs: `exec*`

also `posix_spawn` (not widely supported), ...

waiting for processes to finish: `waitpid` (or `wait`)

process destruction, 'signaling': `exit`, `kill`

exercise (1)

```
int main() {
    pid_t pids[2]; const char *args[] = {"echo", "ARG", NULL};
    const char *extra[] = {"L1", "L2"};
    for (int i = 0; i < 2; ++i) {
        pids[i] = fork();
        if (pids[i] == 0) {
            args[1] = extra[i];
            execv("/bin/echo", args);
        }
    }
    for (int i = 0; i < 2; ++i) {
        waitpid(pids[i], NULL, 0);
    }
}
```

Assuming fork and execv do not fail, which are possible outputs?

A. L1 (newline) L2

B. L1 (newline) L2 (newline) L2

C. L2 (newline) L1

D. A and B

E. A and C

F. all of the above

exercise (2)

```
int main() {
    pid_t pids[2]; const char *args[] = {"echo", "0", NULL};
    for (int i = 0; i < 2; ++i) {
        pids[i] = fork();
        if (pids[i] == 0) { execv("/bin/echo", args); }
    }
    printf("1\n"); fflush(stdout);
    for (int i = 0; i < 2; ++i) {
        waitpid(pids[i], NULL, 0);
    }
    printf("2\n"); fflush(stdout);
}
```

Assuming fork and execv do not fail, which are possible outputs?

A. 0 (newline) 0 (newline) 1 (newline) 2

B. 0 (newline) 1 (newline) 0 (newline) 2

C. 1 (newline) 0 (newline) 0 (newline) 2

D. 1 (newline) 0 (newline) 2 (newline) 0

E. A, B, and C

F. C and D

G. all of the above

H. something else

exercise (2)

```
int main() {
    pid_t pids[2]; const char *args[] = {"echo", "0", NULL};
    for (int i = 0; i < 2; ++i) {
        pids[i] = fork();
        if (pids[i] == 0) { execv("/bin/echo", args); }
    }
    printf("1\n"); fflush(stdout);
    for (int i = 0; i < 2; ++i) {
        waitpid(pids[i], NULL, 0);
    }
    printf("2\n"); fflush(stdout);
}
```

Assuming fork and execv do not fail, which are possible outputs?

A. 0 (newline) 0 (newline) 1 (newline) 2

B. 0 (newline) 1 (newline) 0 (newline) 2

C. 1 (newline) 0 (newline) 0 (newline) 2

D. 1 (newline) 0 (newline) 2 (newline) 0

E. A, B, and C

F. C and D

G. all of the above

H. something else

some POSIX command-line features

searching for programs

```
ls -l ≈ /bin/ls -l
make ≈ /usr/bin/make
```

running in background

```
./someprogram &
```

redirection:

```
./someprogram >output.txt
./someprogram <input.txt
```

pipelines:

```
./someprogram | ./somefilter
```

some POSIX command-line features

searching for programs

```
ls -l ≈ /bin/ls -l
make ≈ /usr/bin/make
```

running in background

```
./someprogram &
```

redirection:

```
./someprogram >output.txt
./someprogram <input.txt
```

pipelines:

```
./someprogram | ./somefilter
```

some POSIX command-line features

searching for programs

```
ls -l ≈ /bin/ls -l
make ≈ /usr/bin/make
```

running in background

```
./someprogram &
```

redirection:

```
./someprogram >output.txt
./someprogram <input.txt
```

pipelines:

```
./someprogram | ./somefilter
```

file descriptors

```
struct process_info {  /* <-- in the kernel somewhere */
    ...
    struct open_file_description *files[SIZE};
    ...
};
...
process->files[file_descriptor]
```

Unix: every process has
array (or similar) of *open file descriptions*

"open file": terminal · socket · regular file · pipe

file descriptor = index into array

- usually what's used with system calls
- stdio.h FILE*s usually have file descriptor + buffer

special file descriptors

file descriptor 0 = standard input

file descriptor 1 = standard output

file descriptor 2 = standard error

constants in `unistd.h`

`STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO`

special file descriptors

file descriptor 0 = standard input

file descriptor 1 = standard output

file descriptor 2 = standard error

constants in `unistd.h`

- `STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO`

but you can't choose which number `open` assigns...?

- more on this later

getting file descriptors

```
int read_fd = open("dir/file1", O_RDONLY);
int write_fd = open("/other/file2", O_WRONLY | ...);
int rdwr_fd = open("file3", O_RDWR);
```

used internally by fopen(), etc.

also for files without normal filenames…:

```
int fd = shm_open("/shared_memory", O_RDWR, 0666); // shared memory
int socket_fd = socket(AF_INET, SOCK_STREAM, 0); // TCP socket
int term_fd = posix_openpt(O_RDWR); // pseudo-terminal
int pipe_fds[2]; pipe(pipefds); // "pipes" (later)
...
```

close

```
int close(int fd);
```

close the file descriptor, deallocating that array index

- does not affect other file descriptors
- that refer to same "open file description"
- (e.g. in `fork()`ed child or created via (later) `dup2`)

if last file descriptor for open file description, resources deallocated

returns 0 on success

returns -1 on error

- e.g. ran out of disk space while finishing saving file

shell redirection

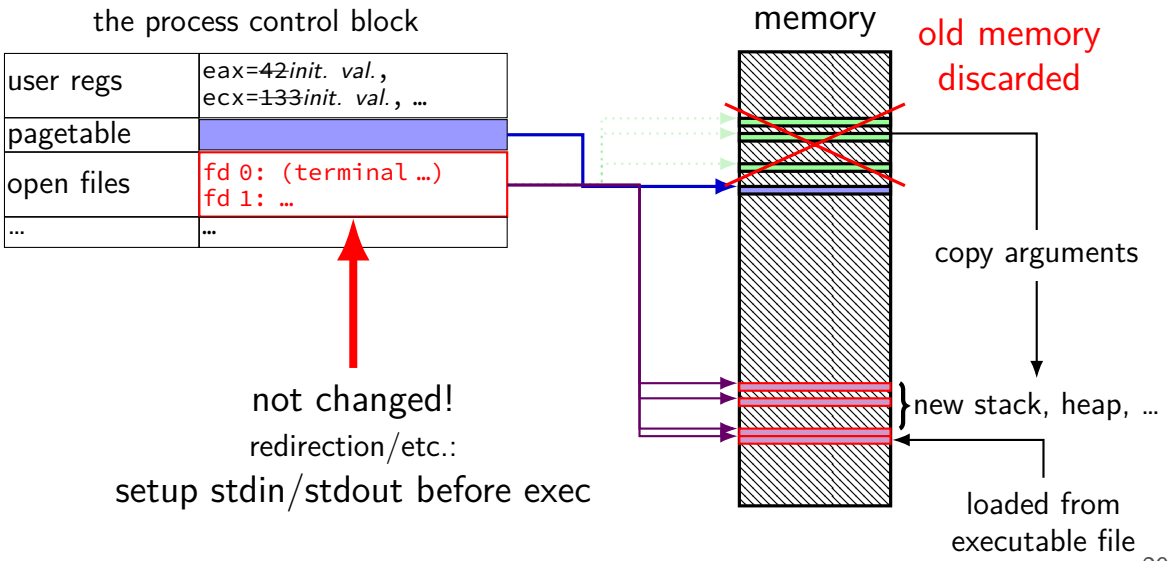
`./my_program ... < input.txt`:

run `./my_program ...` but use `input.txt` as input
like we copied and pasted the file into the terminal

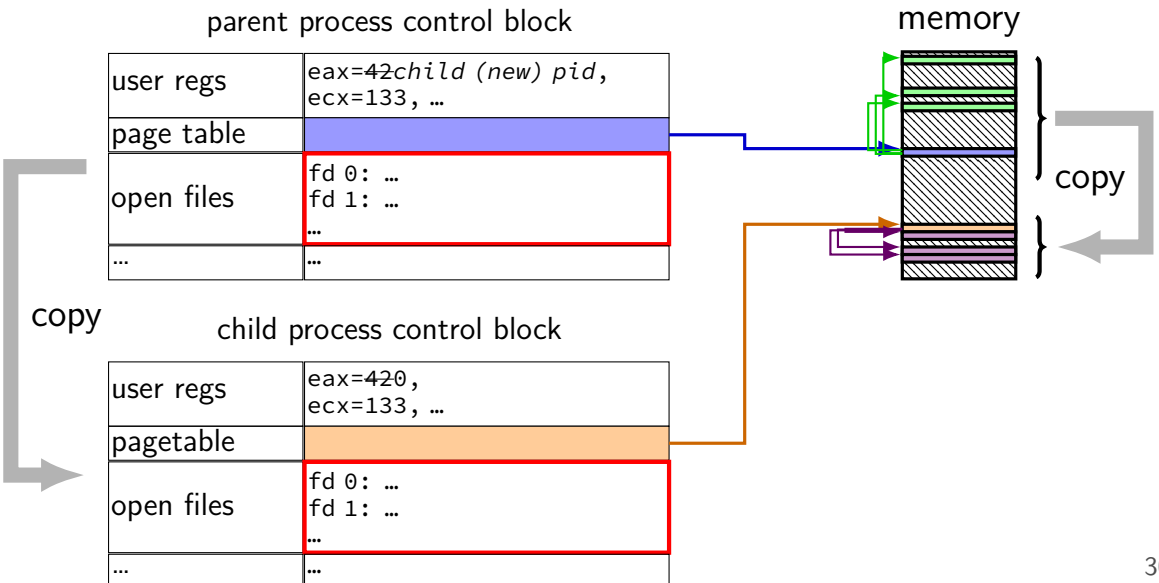
`echo foo > output.txt`:

runs `echo foo`, sends output to `output.txt`
like we copied and pasted the output into that file
(as it was written)

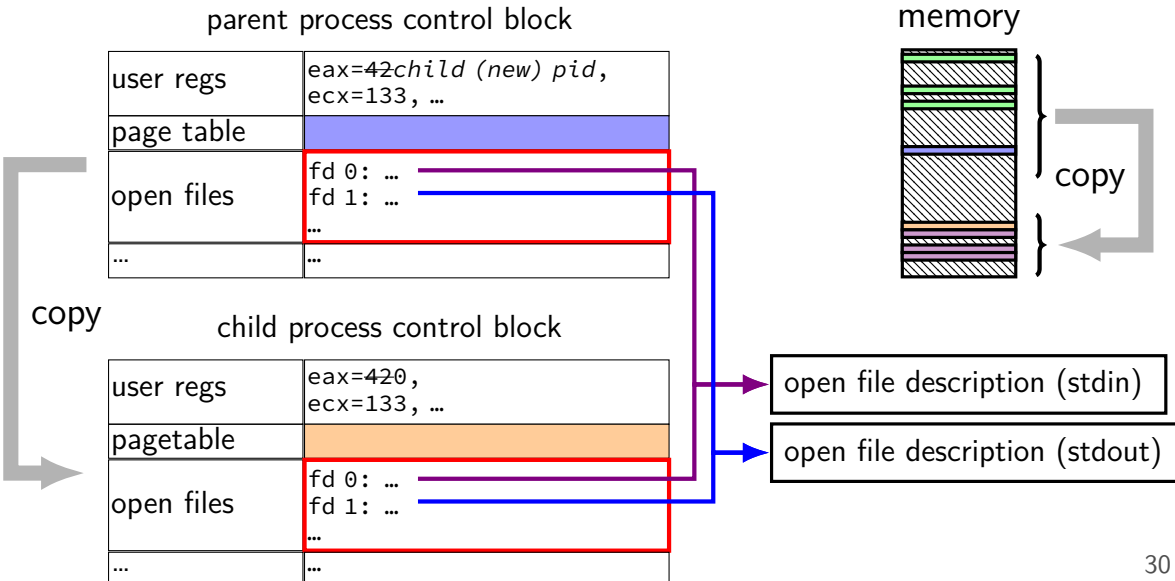
exec preserves open files



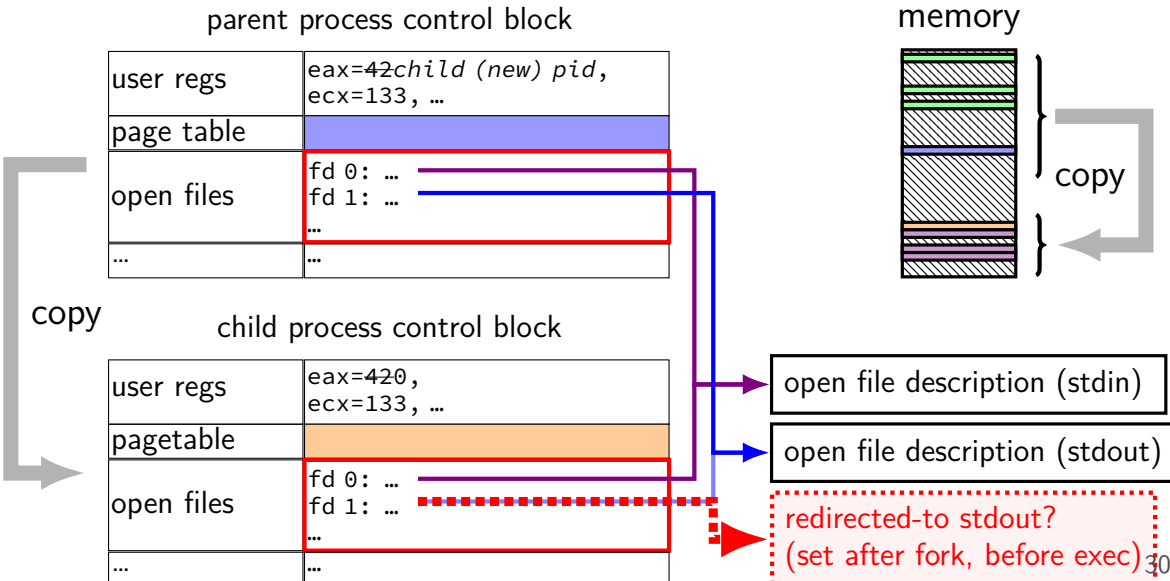
fork copies open file list



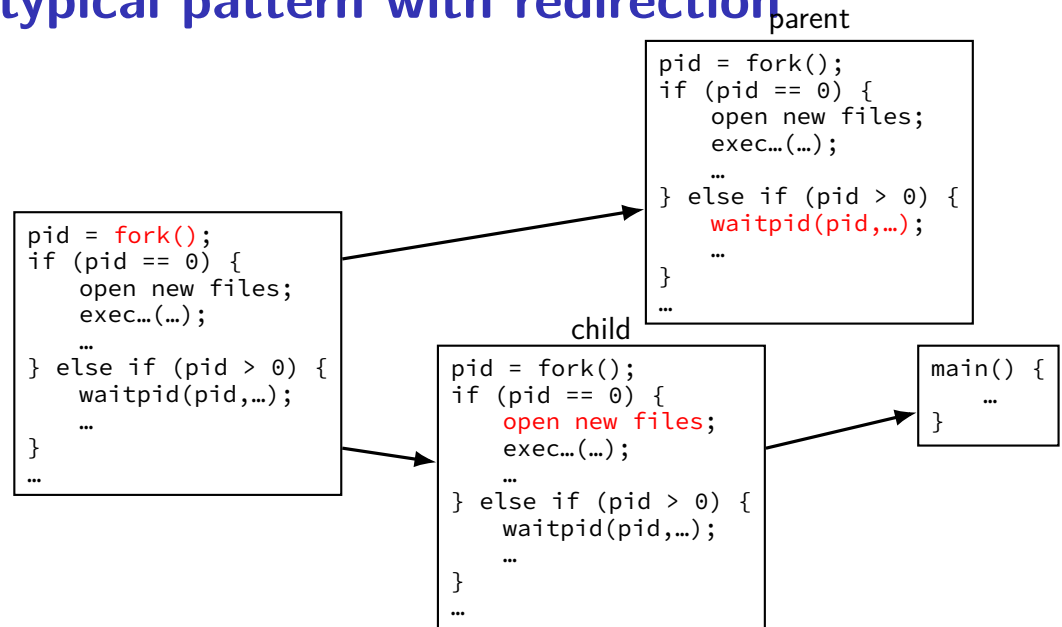
fork copies open file list



fork copies open file list



typical pattern with redirection



redirecting with exec

standard output/error/input are files

(C stdout/stderr/stdin; C++ cout/cerr/cin)

(probably after forking) open files to redirect

...and make them be standard output/error/input

using `dup2()` library call

then exec, preserving new standard output/etc.

reassigning file descriptors

redirection: `./program >output.txt`

step 1: open output.txt for writing, get new file descriptor

step 2: make that new file descriptor stdout (number 1)

reassigning and file table

```
// something like this in OS code
struct process_info {
    ...
    struct open_file_description *files[SIZE];
    ....
};
...
process->files[STDOUT_FILENO] = process->files[opened-fd];
```

syscall: dup2(*opened-fd*, STDOUT_FILENO);

reassigning file descriptors

redirection: `./program >output.txt`

step 1: open output.txt for writing, get new file descriptor

step 2: make that new file descriptor stdout (number 1)

tool: `int dup2(int oldfd, int newfd)`
make `newfd` refer to same open file as `oldfd`

- same *open file description*
- shares the current location in the file
- (even after more reads/writes)

what if newfd already allocated — closed, then reused

dup2 example

redirects stdout to output to `output.txt`:

```
fflush(stdout);  /* clear printf's buffer */
int fd = open("output.txt",
              O_WRONLY | O_CREAT | O_TRUNC);
if (fd < 0)
    do_something_about_error();

dup2(fd, STDOUT_FILENO);
/* now both write(fd, ...) and write(STDOUT_FILENO, ...)
   write to output.txt
   */

close(fd); /* only close original, copy still works! */

printf("This will be sent to output.txt.\n");
```

open/dup/close/etc. and fd array

```
// something like this in OS code
struct process_info {
  ...
  struct open_file_description *files[NUM];
};
```

open: `files[new_fd] = ...;`

dup2(from, to): `files[to] = files[from];`

close: `files[fd] = NULL;`

fork:

```
for (int i = ...)
    child->files[i] = parent->files[i];
```

exercise

```
int fd = open("output.txt", O_WRONLY|O_CREAT|O_TRUNC, 0666);
write(fd, "A", 1);
dup2(STDOUT_FILENO, 100);
dup2(fd, STDOUT_FILENO);
write(STDOUT_FILENO, "B", 1);
write(fd, "C", 1);
close(fd);
write(STDOUT_FILENO, "D", 1);
write(100, "E", 1);
```

Assume fd `100` is not what open returns. What is written to `output.txt`?

A. ABCDE **C.** ABC **E.** something else

B. ABCD **D.** ACD

pipes

special kind of file: pipes

bytes go in one end, come out the other — once

created with `pipe()` library call

intended use: communicate between processes

- like implementing shell pipelines

pipe()

```
int pipe_fd[2];
if (pipe(pipe_fd) < 0)
    handle_error();
/* normal case: */
int read_fd = pipe_fd[0];
int write_fd = pipe_fd[1];
```

then from one process...

```
write(write_fd, ...);
```

and from another

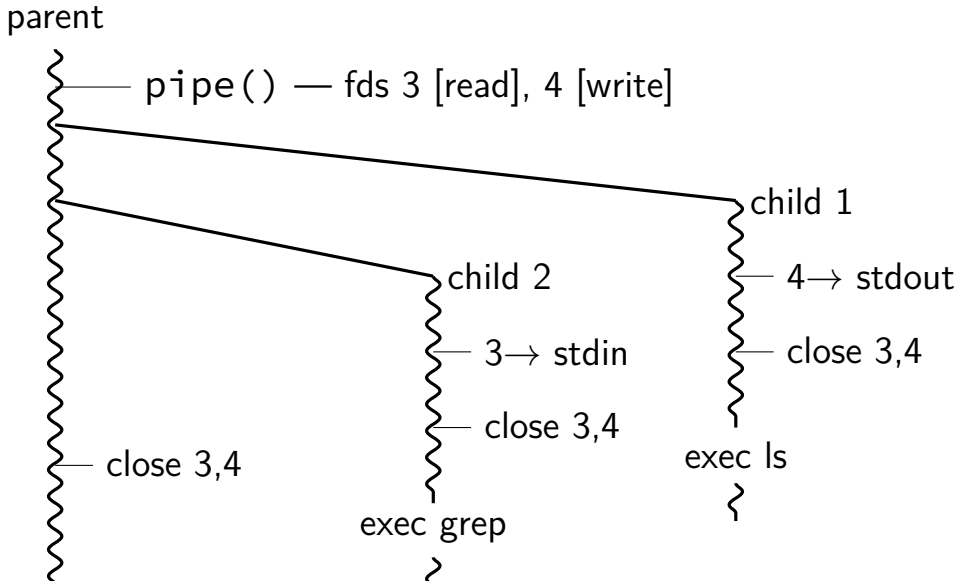
```
read(read_fd, ...);
```

pipe and pipelines

```
ls -1 | grep foo
```

```
pipe(pipe_fd);
ls_pid = fork();
if (ls_pid == 0) {
    dup2(pipe_fd[1], STDOUT_FILENO);
    close(pipe_fd[0]); close(pipe_fd[1]);
    char *argv[] = {"ls", "-1", NULL};
    execv("/bin/ls", argv);
}
grep_pid = fork();
if (grep_pid == 0) {
    dup2(pipe_fd[0], STDIN_FILENO);
    close(pipe_fd[0]); close(pipe_fd[1]);
    char *argv[] = {"grep", "foo", NULL};
    execv("/bin/grep", argv);
}
close(pipe_fd[0]); close(pipe_fd[1]);
```

example execution



Unix API summary

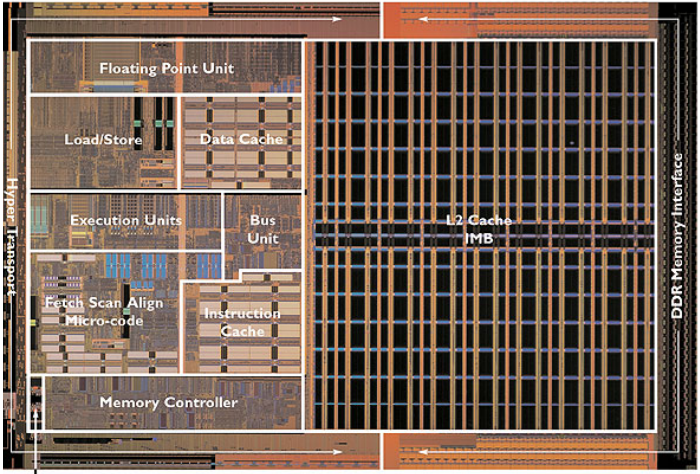
spawn and wait for program: `fork` (copy), then
- in child: setup, then `execv`, etc. (replace copy)
- in parent: `waitpid`

files: open, read and/or write, close
- one interface for regular files, pipes, network, devices, ...

file descriptors are indices into per-process array
- index 0, 1, 2 = stdin, stdout, stderr
- `dup2` — assign one index to another
- `close` — deallocate index

redirection/pipelines
- open() or pipe() to create new file descriptors
- dup2 in child to assign file descriptor to index 0, 1

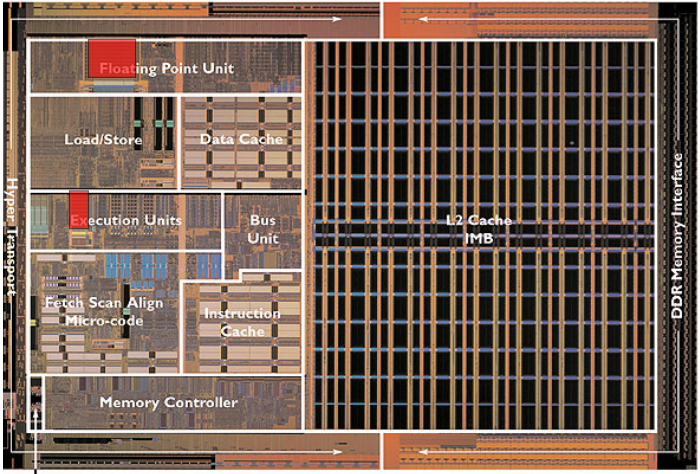
2004 CPU



AMD

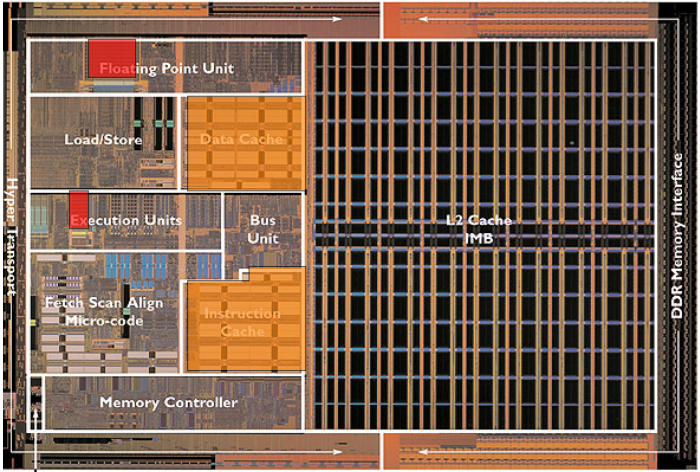
ce: approx 2004 AMD press image of Opteron die;
prox register location via chip-architect.org (Hans de Vries)

2004 CPU



Registers

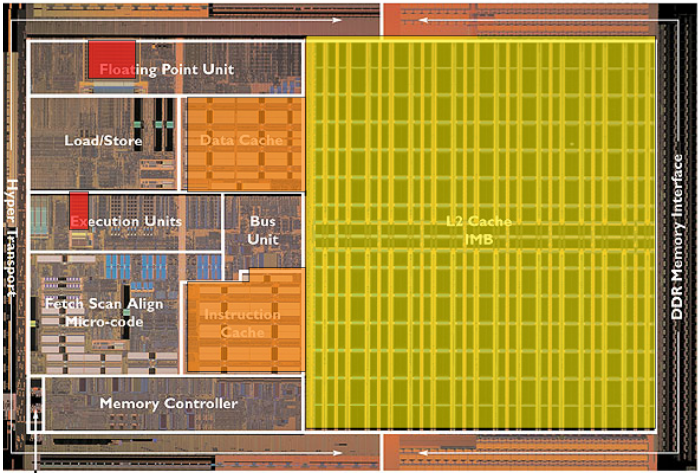
2004 CPU

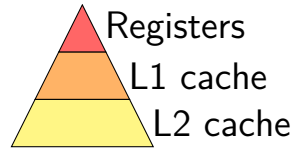


AMD

ce: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU

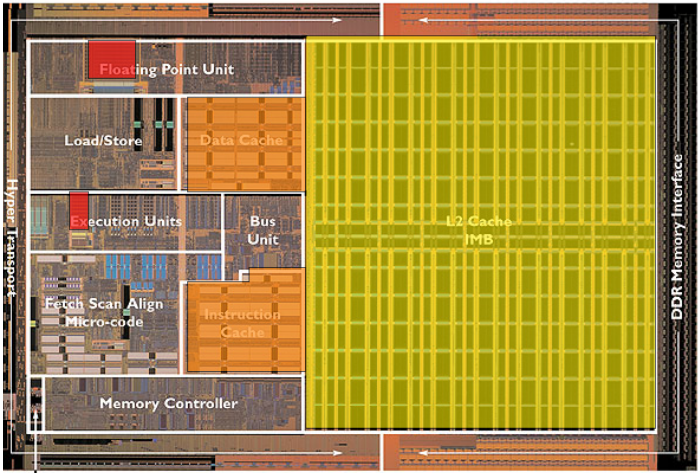


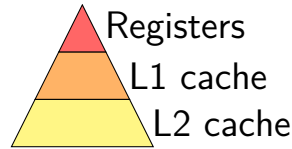


AMD

ce: approx 2004 AMD press image of Opteron die;
prox register location via chip-architect.org (Hans de Vries)

2004 CPU

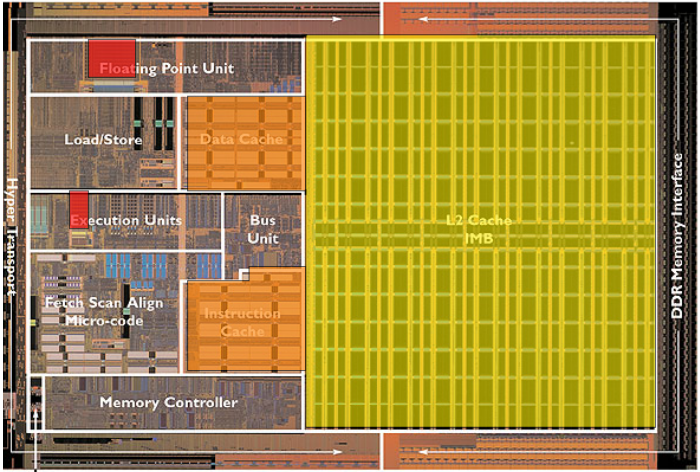


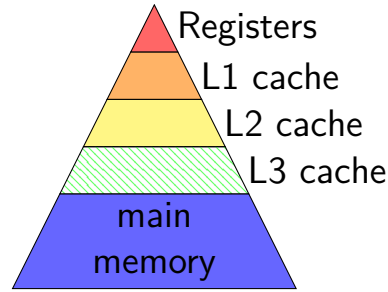


AMD

ce: approx 2004 AMD press image of Opteron die;
prox register location via chip-architect.org (Hans de Vries)

2004 CPU

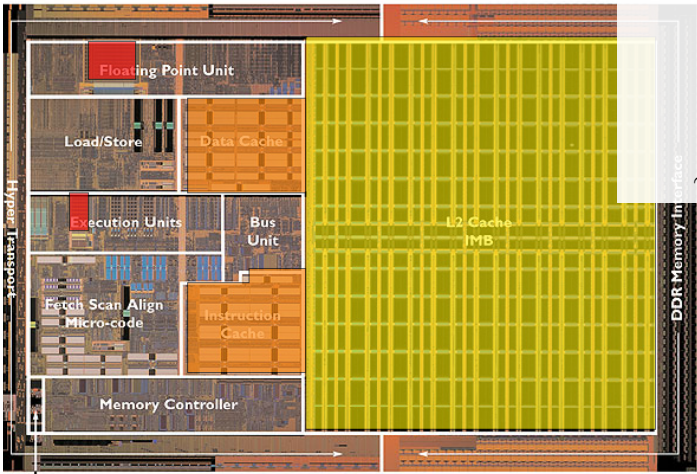


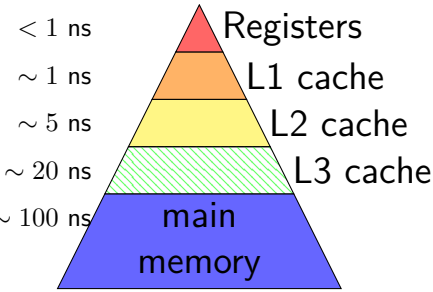


AMD

ce: approx 2004 AMD press image of Opteron die;
prox register location via chip-architect.org (Hans de Vries)

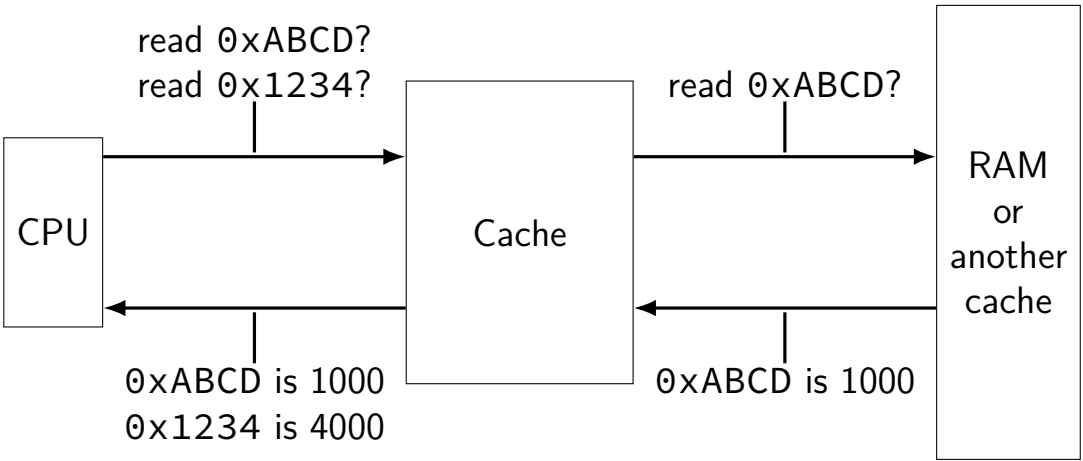
2004 CPU





ce: approx 2004 AMD press image of Opteron die;
prox register location via chip-architect.org (Hans de Vries)

the place of cache



memory hierarchy goals

performance of the fastest (smallest) memory

- hide 100x latency difference? 99+% hit (= value found in cache) rate

capacity of the largest (slowest) memory

backup slides

exit statuses

```
int main() {
    return 0;  /* or exit(0);  */
}
```

the status

```
#include <sys/wait.h>
...
  waitpid(child_pid, &status, 0);
  if (WIFEXITED(status)) {
    printf("main returned or exit called with %d\n",
           WEXITSTATUS(status));
  } else if (WIFSIGNALED(status)) {
    printf("killed by signal %d\n", WTERMSIG(status));
  } else {
      ...
  }
```

"status code" encodes both return value and if exit was abnormal

W* macros to decode it

the status

```
#include <sys/wait.h>
...
  waitpid(child_pid, &status, 0);
  if (WIFEXITED(status)) {
    printf("main returned or exit called with %d\n",
           WEXITSTATUS(status));
  } else if (WIFSIGNALED(status)) {
    printf("killed by signal %d\n", WTERMSIG(status));
  } else {
      ...
  }
```

"status code" encodes both return value and if exit was abnormal

W* macros to decode it

shell

allow user (= person at keyboard) to run applications

user's wrapper around process-management functions

aside: shell forms

POSIX: command line you have used before

also: graphical shells

e.g. OS X Finder, Windows explorer

other types of command lines?

completely different interfaces?

searching for programs

POSIX convention: PATH *environment variable*

- example: `/home/cr4bd/bin:/usr/bin:/bin`
- list of directories to check in order

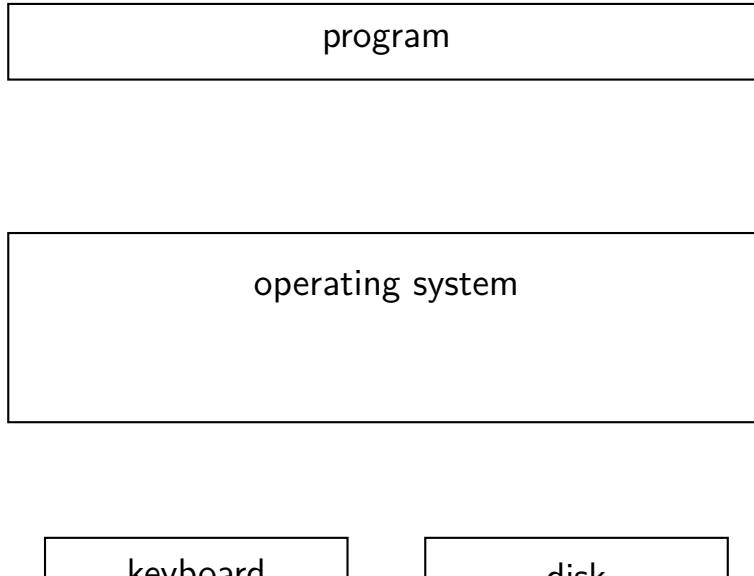
environment variables = key/value pairs stored with process

- by default, left unchanged on execve, fork, etc.

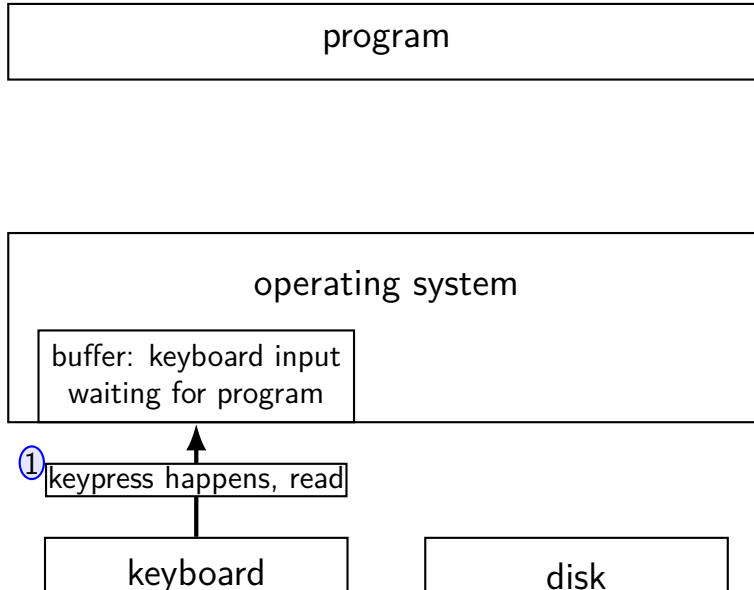
one way to implement: [pseudocode]

```
for (directory in path) {
    execv(directory + "/" + program_name, argv);
}
```

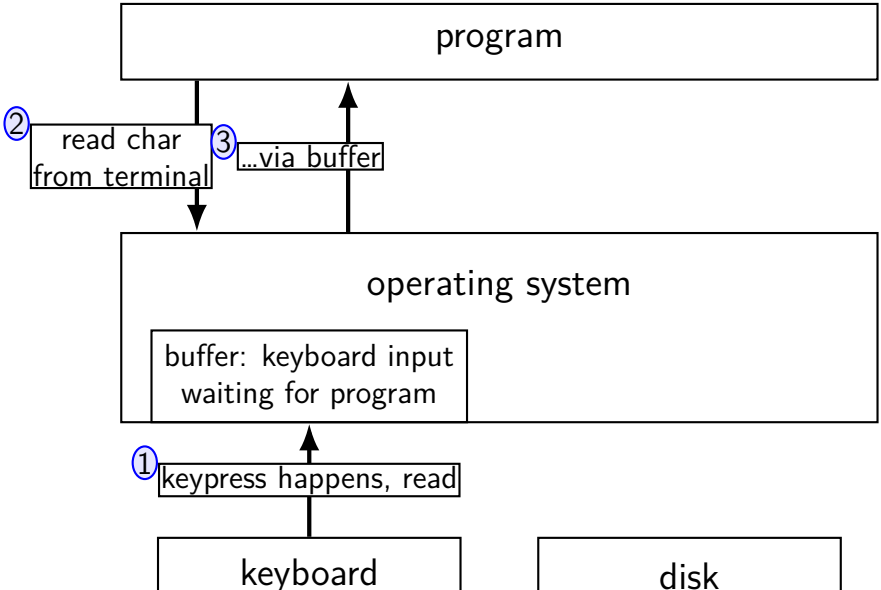
kernel buffering (reads)



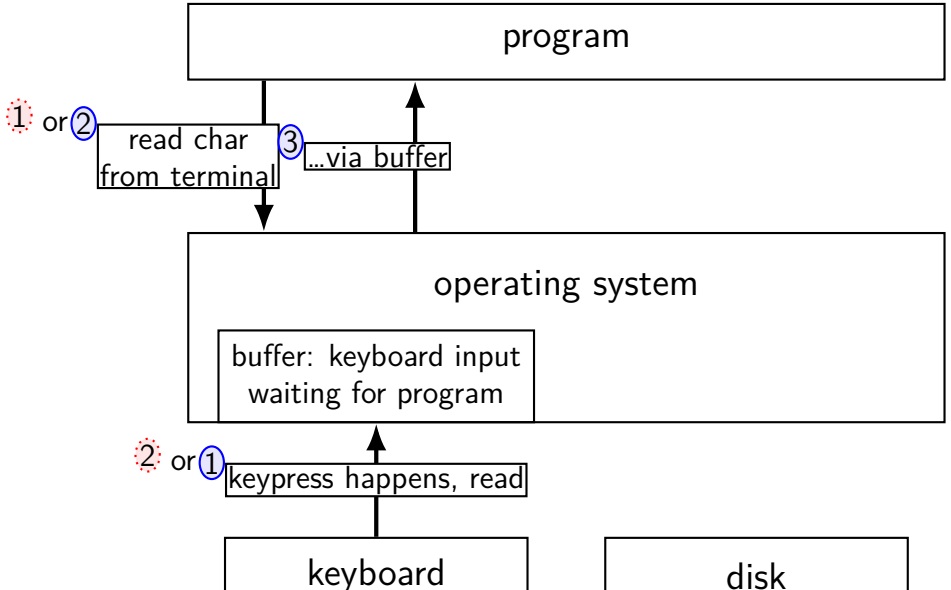
kernel buffering (reads)



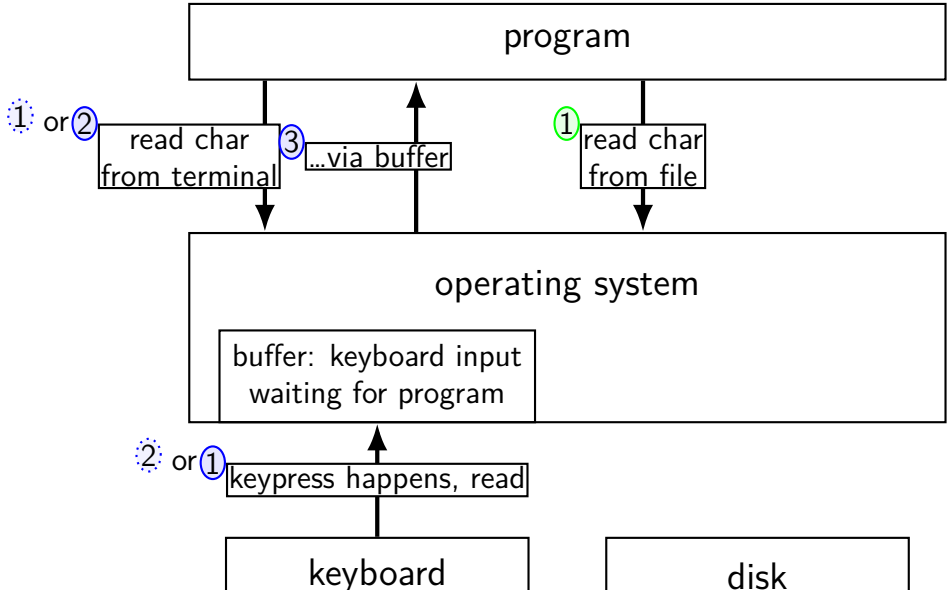
kernel buffering (reads)



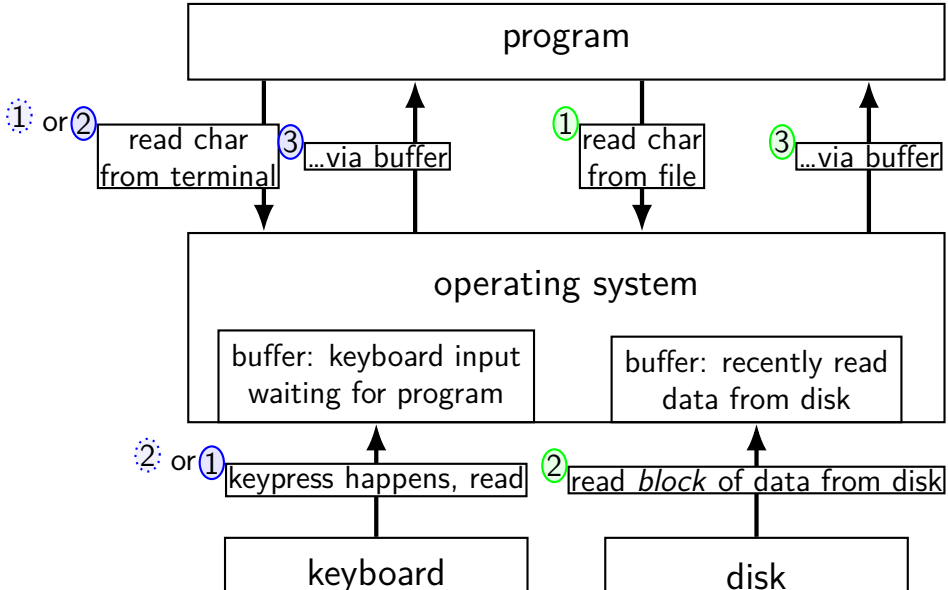
kernel buffering (reads)



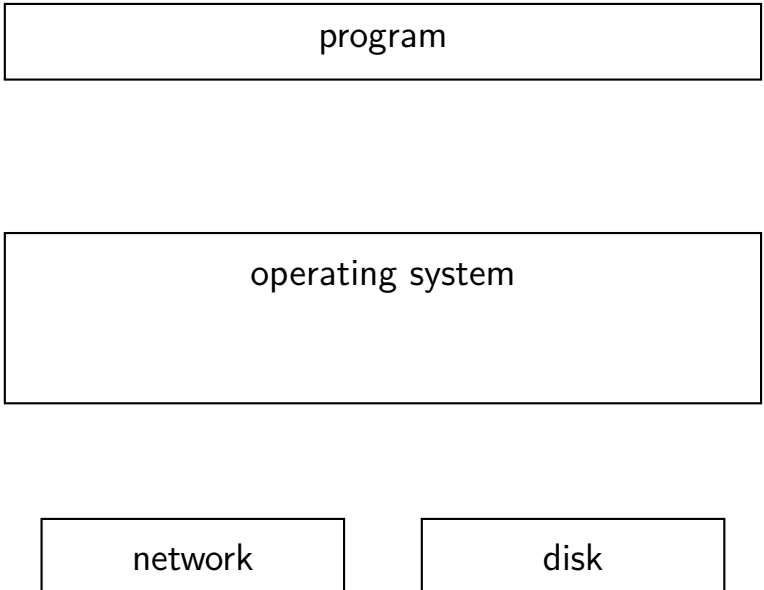
kernel buffering (reads)



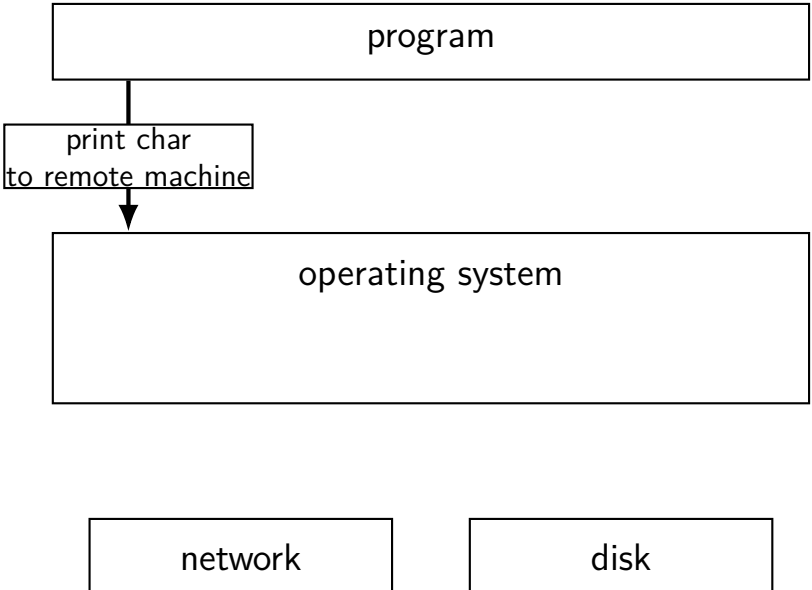
kernel buffering (reads)



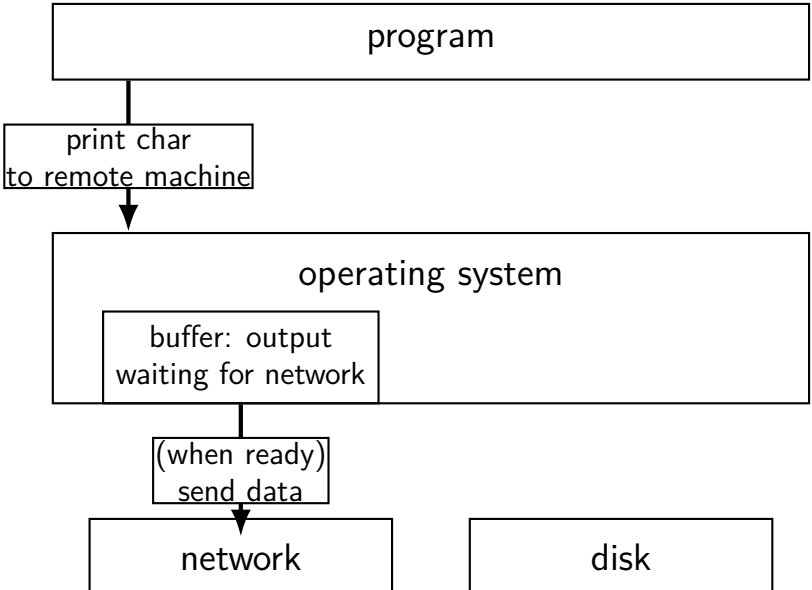
kernel buffering (writes)



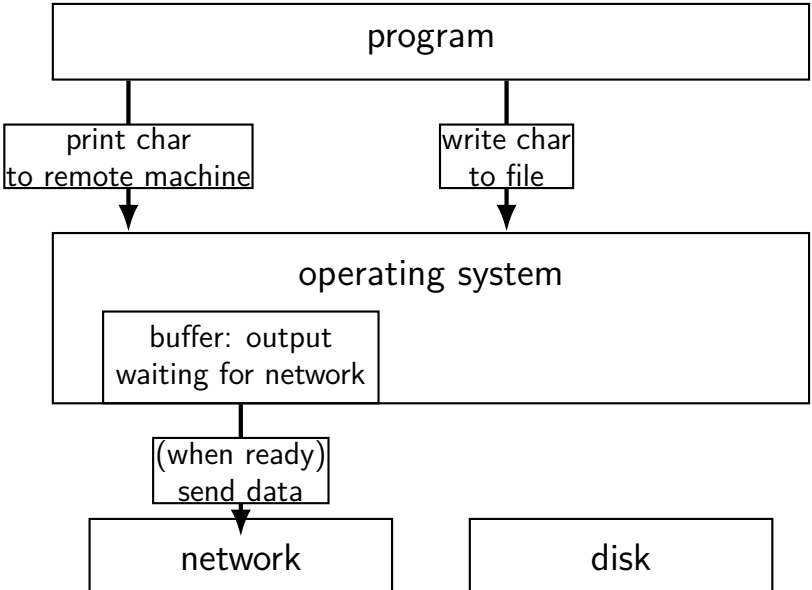
kernel buffering (writes)



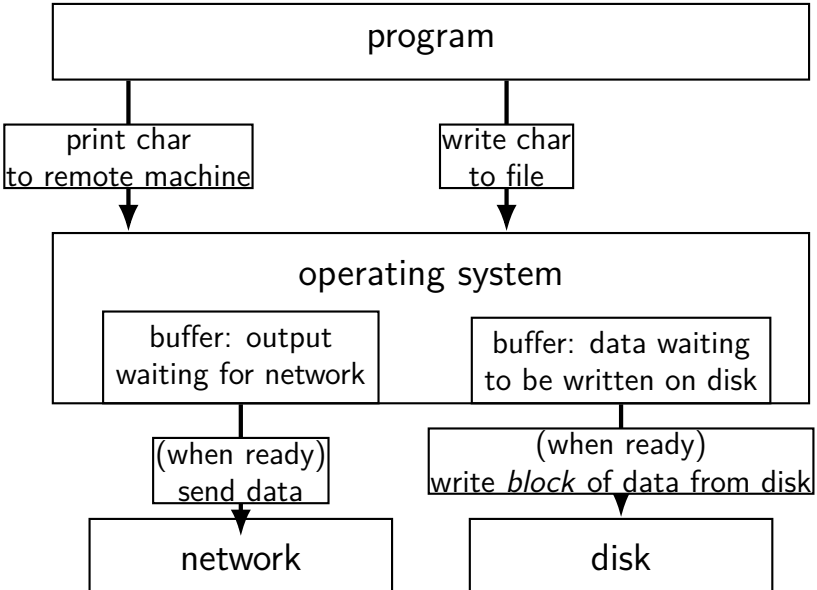
kernel buffering (writes)



kernel buffering (writes)



kernel buffering (writes)



read/write operations

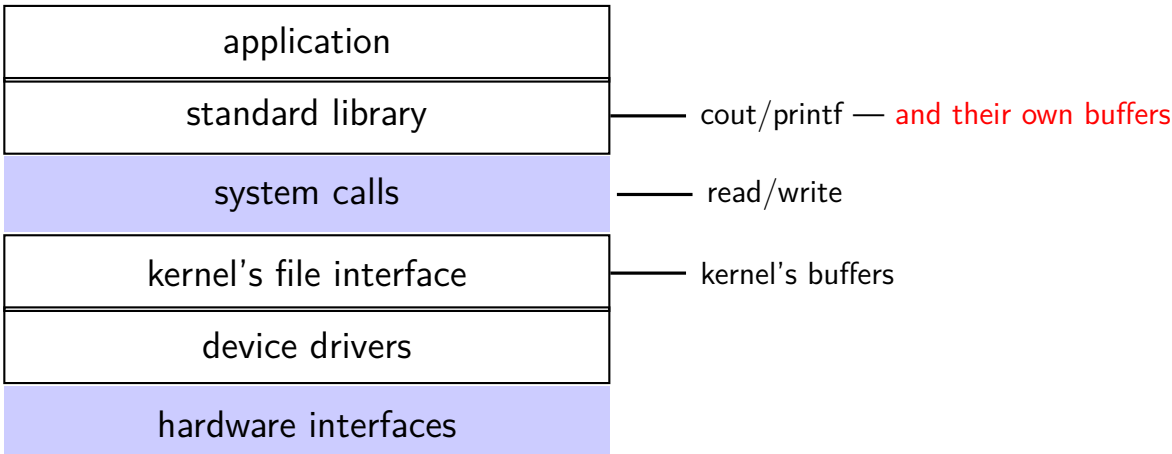
read()/write(): move data into/out of buffer

possibly wait if buffer is empty (read)/full (write)

actual I/O operations — wait for device to be ready

- trigger process to stop waiting if needed

layering


application
standard library
system calls
kernel's file interface
device drivers
hardware interfaces
cout/printf — and their own buffers
read/write
kernel's buffers

why the extra layer

better (but more complex to implement) interface:

- read line
- formatted input (scanf, cin into integer, etc.)
- formatted output

less system calls (bigger reads/writes) sometimes faster

- buffering can combine multiple in/out library calls into one system call

more portable interface

- cin, printf, etc. defined by C and C++ standards

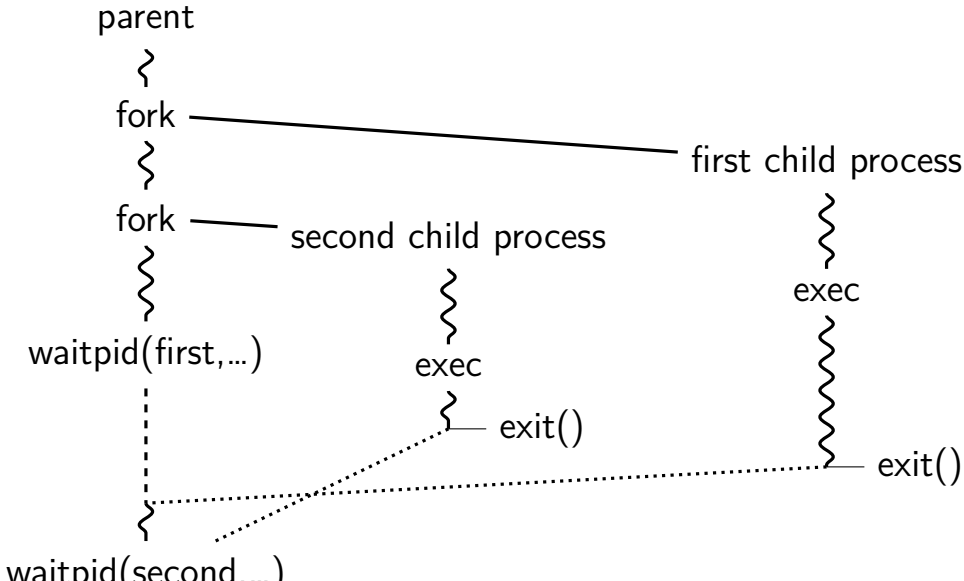
pipe() and blocking

BROKEN example:

```
int pipe_fd[2];
if (pipe(pipe_fd) < 0)
    handle_error();
int read_fd = pipe_fd[0];
int write_fd = pipe_fd[1];
write(write_fd, some_buffer, some_big_size);
read(read_fd, some_buffer, some_big_size);
```

This is likely to not terminate. What's the problem?

pattern with multiple?



this class: focus on Unix

Unix-like OSes will be our focus

we have source code

used to from 2150, etc.?

have been around for a while

xv6 imitates Unix

Unix history

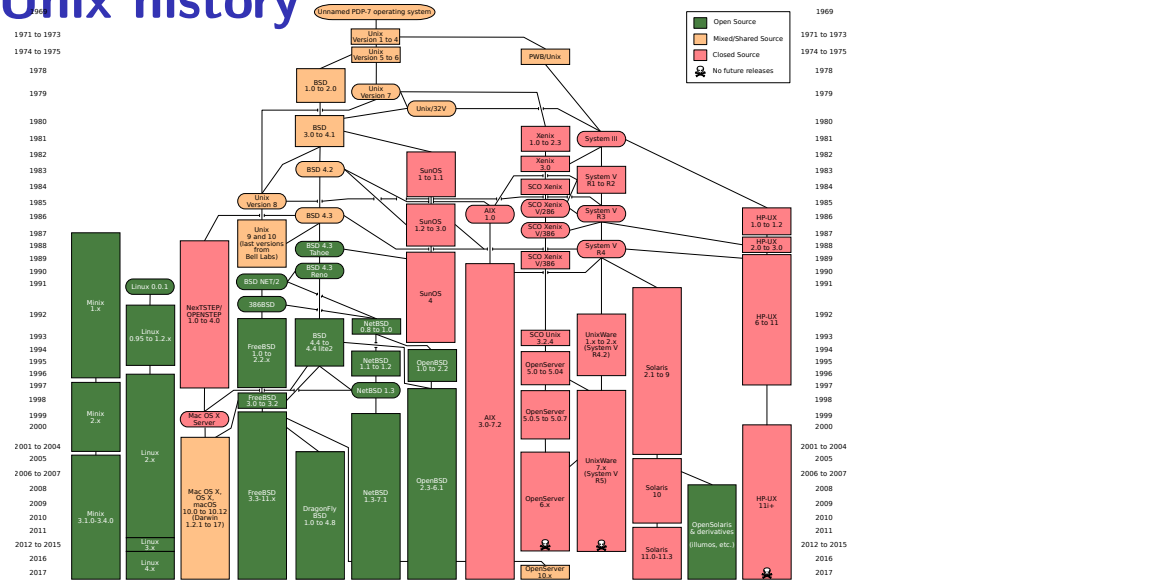


image: Wikipedia/Eraserhead1+Infinity0+Sav_vas

POSIX: standardized Unix

Portable Operating System Interface (POSIX)

> "standard for Unix"

current version online:
https://pubs.opengroup.org/onlinepubs/9699919799/

(almost) followed by most current Unix-like OSes

...but OSes add extra features

...and POSIX doesn't specify everything

what POSIX defines

POSIX specifies the library and shell interface

- source code compatibility

doesn't care what is/is not a system call...

doesn't specify binary formats...

idea: write applications for POSIX, recompile and run on all implementations

- this was a very important goal in the 80s/90s
- at the time, no dominant Unix-like OS (Linux was very immature)

getpid

```
pid_t my_pid = getpid();
printf("my pid is %ld\n", (long) my_pid);
```

process ids in ps

```
cr4bd@machine:~$ ps
  PID TTY          TIME CMD
14777 pts/3    00:00:00 bash
14798 pts/3    00:00:00 ps
```

read/write

```
ssize_t read(int fd, void *buffer, size_t count);
ssize_t write(int fd, void *buffer, size_t count);
```

read/write up to *count* bytes to/from *buffer*

returns number of bytes read/written or -1 on error

- ssize_t is a signed integer type
- error code in `errno`

read returning 0 means end-of-file (*not an error*)

- can read/write less than requested (end of file, broken I/O device, ...)

read'ing one byte at a time

```
string s;
ssize_t amount_read;
char c;
/* cast to void * not needed in C */
while ((amount_read = read(STDIN_FILENO, (void*) &c, 1)) > 0)
    /* amount_read must be exactly 1 */
    s += c;
}
if (amount_read == -1) {
    /* some error happened */
    perror("read"); /* print out a message about it */
} else if (amount_read == 0) {
    /* reached end of file */
}
```

write example

```
/* cast to void * optional in C */
write(STDOUT_FILENO, (void *) "Hello, World!\n", 14);
```

aside: environment variables (1)

key=value pairs associated with every process:

```
$ printenv
MODULE_VERSION_STACK=3.2.10
MANPATH=:/opt/puppetlabs/puppet/share/man
XDG_SESSION_ID=754
HOSTNAME=labsrv01
SELINUX_ROLE_REQUESTED=
TERM=screen
SHELL=/bin/bash
HISTSIZE=1000
SSH_CLIENT=128.143.67.91 58432 22
SELINUX_USE_CURRENT_RANGE=
QTDIR=/usr/lib64/qt-3.3
OLDPWD=/zf14/cr4bd
QTINC=/usr/lib64/qt-3.3/include
SSH_TTY=/dev/pts/0
QT_GRAPHICSSYSTEM_CHECKED=1
USER=cr4bd
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or
MODULE_VERSION=3.2.10
MAIL=/var/spool/mail/cr4bd
PATH=/zf14/cr4bd/.cargo/bin:/zf14/cr4bd/bin:/usr/lib64/qt-3.3/bin:/usr/local/bin:/usr/bin:/u
PWD=/zf14/cr4bd
```

aside: environment variables (2)

environment variable library functions:

`getenv("KEY")` → *value*

`putenv("KEY=value")` (sets KEY to *value*)

`setenv("KEY", "value")` (sets KEY to *value*)

```
int execve(char *path, char **argv, char **envp)
```

```
char *envp[] = { "KEY1=value1", "KEY2=value2", NULL };
char *argv[] = { "somecommand", "some arg", NULL };
execve("/path/to/somecommand", argv, envp);
```

normal exec versions — keep same environment variables

aside: environment variables (3)

interpretation up to programs, but common ones...

`PATH=/bin:/usr/bin`

to run a program 'foo', look for an executable in `/bin/foo`, then `/usr/bin/foo`

`HOME=/zf14/cr4bd`

current user's home directory is '/zf14/cr4bd'

`TERM=screen-256color`

your output goes to a 'screen-256color'-style terminal

...

multiple processes?

```
while (...) {
    pid = fork();
    if (pid == 0) {
        exec ...
    } else if (pid > 0) {
        pids.push_back(pid);
    }
}

/* retrieve exit statuses in order */
for (pid_t pid : pids) {
    waitpid(pid, ...);
    ...
}
```

waiting for all children

```
#include <sys/wait.h>
...
  while (true) {
    pid_t child_pid = waitpid(-1, &status, 0);
    if (child_pid == (pid_t) -1) {
      if (errno == ECHILD) {
        /* no child process to wait for */
        break;
      } else {
        /* some other error */
      }
    }
    /* handle child_pid exiting */
  }
```

multiple processes?

```
while (...) {
    pid = fork();
    if (pid == 0) {
        exec ...
    } else if (pid > 0) {
        pids.push_back(pid);
    }
}

/* retrieve exit statuses as processes finish */
while ((pid = waitpid(-1, ...)) != -1) {
    handleProcessFinishing(pid);
}
```

‘waiting’ without waiting

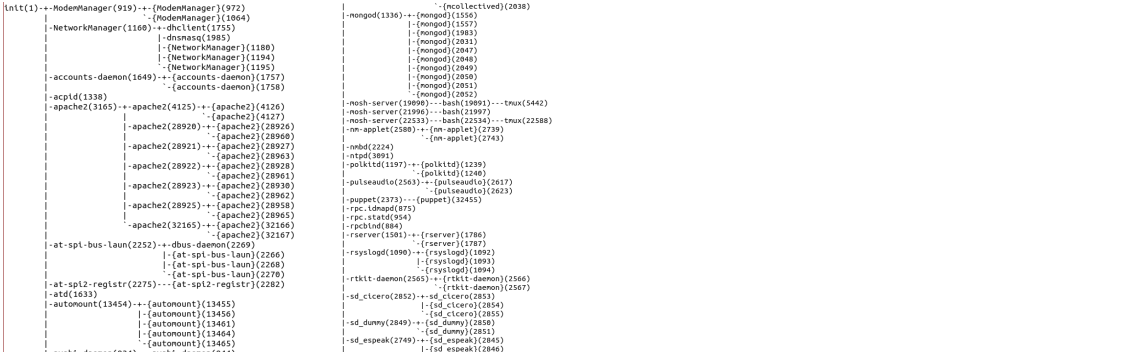
```
#include <sys/wait.h>
...
  pid_t return_value = waitpid(child_pid, &status, WNOHANG);
  if (return_value == (pid_t) 0) {
    /* child process not done yet */
  } else if (child_pid == (pid_t) -1) {
    /* error */
  } else {
    /* handle child_pid exiting */
  }
```

parent and child processes

every process (but process id 1) has a *parent process* (`getppid()`)

this is the process that can wait for it

creates tree of processes (Linux `pstree` command):



parent and child questions...

what if parent process exits before child?

- child's parent process becomes process id 1 (typically called *init*)

what if parent process never `waitpid()`s (or equivalent) for child?

- child process stays around as a "zombie"
- can't reuse pid in case parent wants to use `waitpid()`

what if non-parent tries to `waitpid()` for child?

- waitpid fails

read'ing a fixed amount

```
ssize_t offset = 0;
const ssize_t amount_to_read = 1024;
char result[amount_to_read];
do {
    /* cast to void * optional in C */
    ssize_t amount_read =
        read(STDIN_FILENO,
             (void *) (result + offset),
             amount_to_read - offset);
    if (amount_read < 0) {
        perror("read"); /* print error message */
        ... /* abort??? */
    } else {
        offset += amount_read;
    }
```

partial reads

on regular file: read reads what you request

but otherwise: usually gives you what's known to be available

after waiting for something to be available

partial reads

on regular file: read reads what you request

but otherwise: usually gives you what's known to be available

after waiting for something to be available

reading from network — what's been received

reading from keyboard — what's been typed

write example (with error checking)

```
const char *ptr = "Hello, World!\n";
ssize_t remaining = 14;
while (remaining > 0) {
    /* cast to void * optional in C */
    ssize_t amount_written = write(STDOUT_FILENO,
                                   ptr,
                                   remaining);
    if (amount_written < 0) {
        perror("write"); /* print error message */
        ... /* abort??? */
    } else {
        remaining -= amount_written;
        ptr += amount_written;
    }
}
```

partial writes

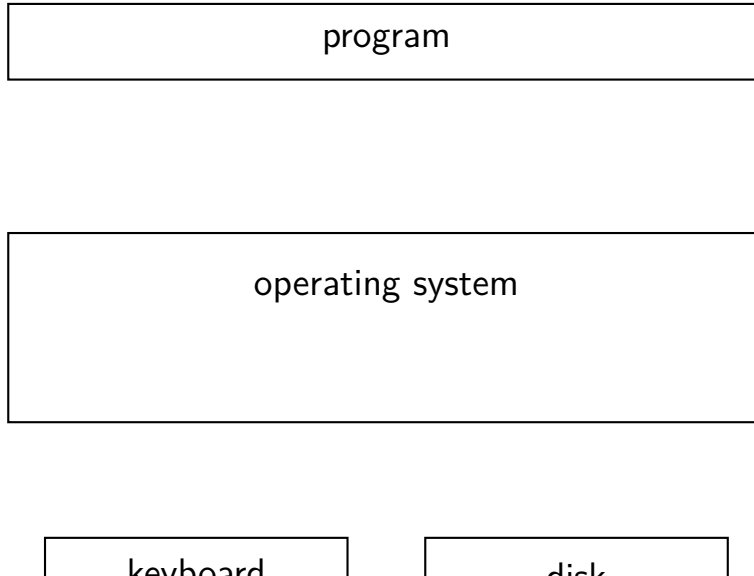
usually only happen on error or interruption

- but can request "non-blocking"
- (interruption: via *signal*)

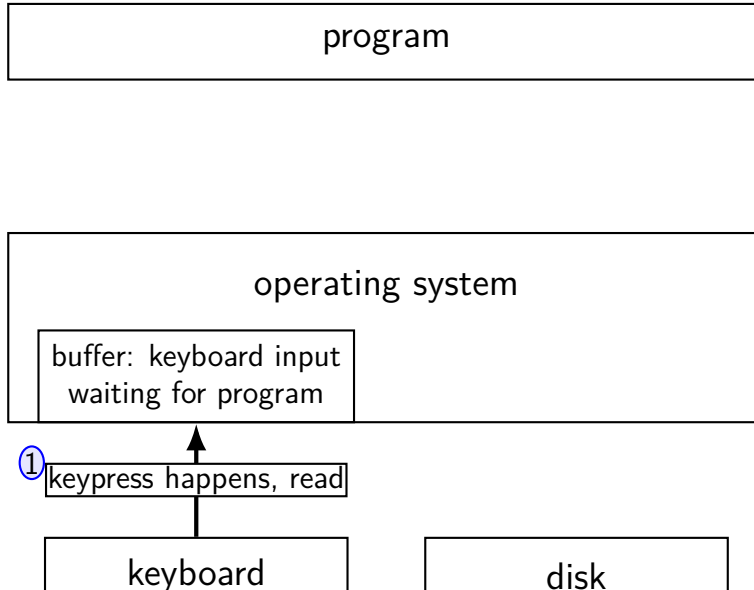
usually: write waits until it completes

- = until remaining part fits in buffer in kernel
- does not mean data was sent on network, shown to user yet, etc.

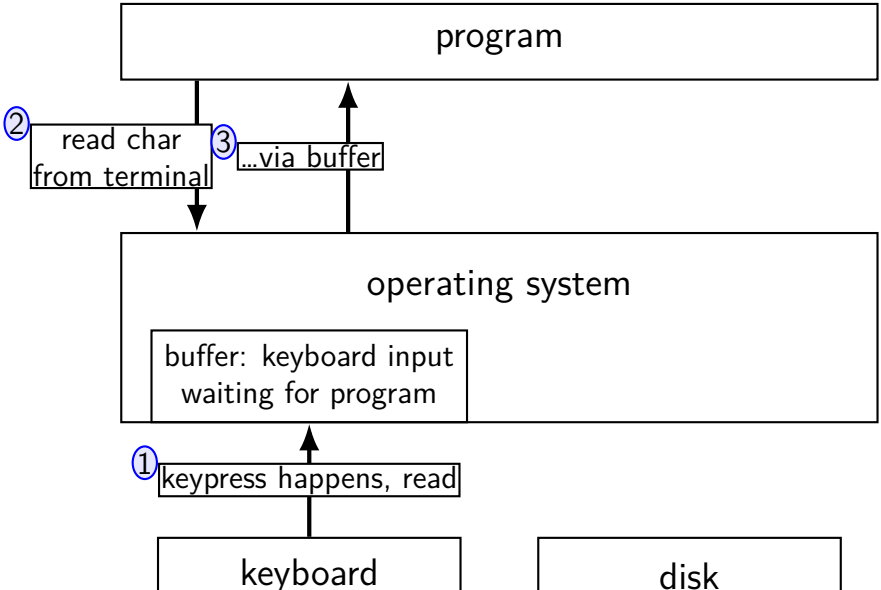
kernel buffering (reads)



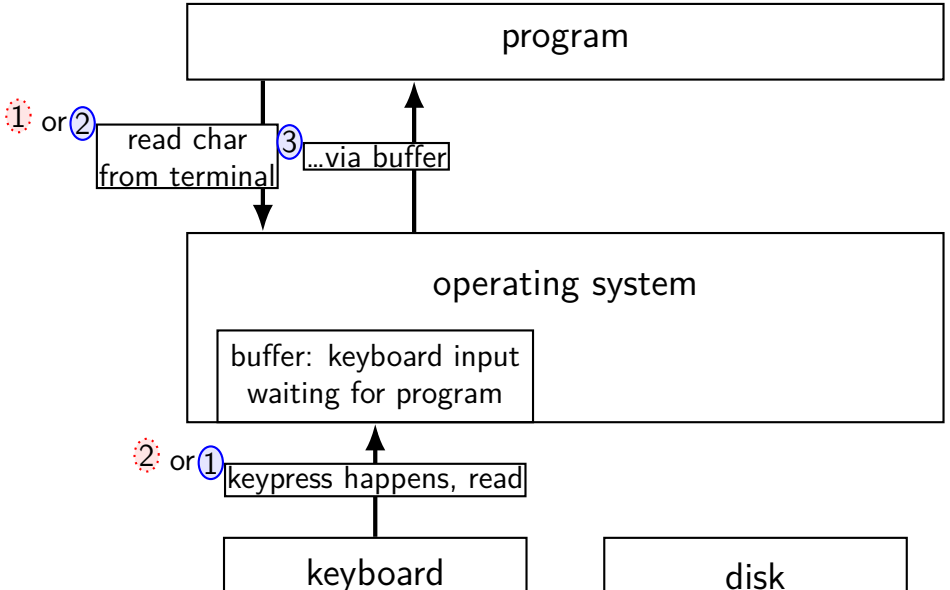
kernel buffering (reads)



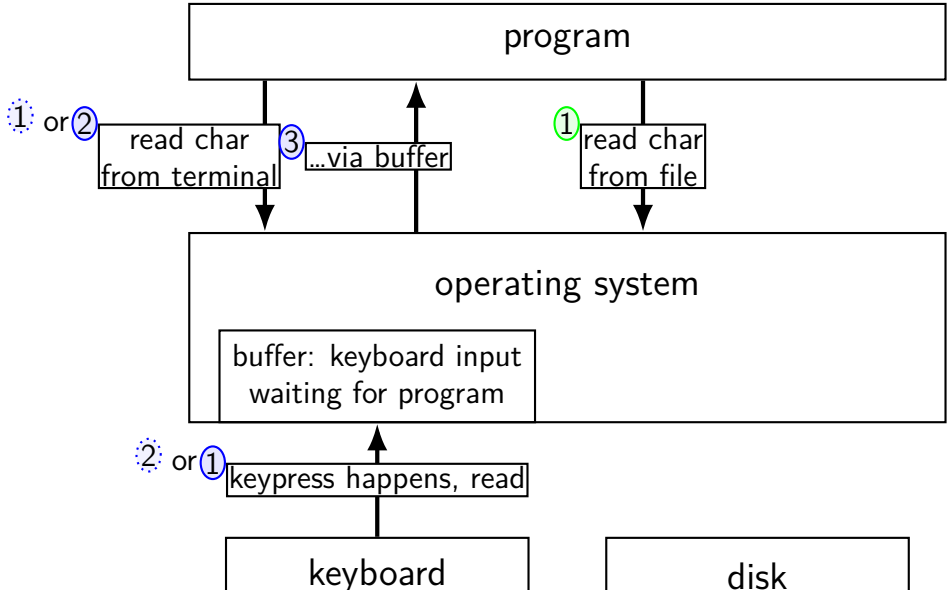
kernel buffering (reads)



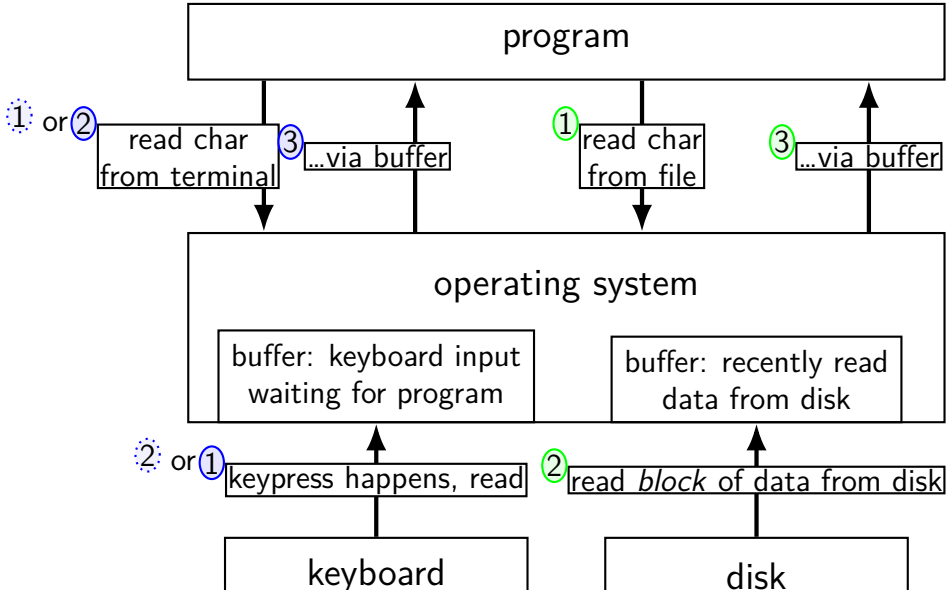
kernel buffering (reads)



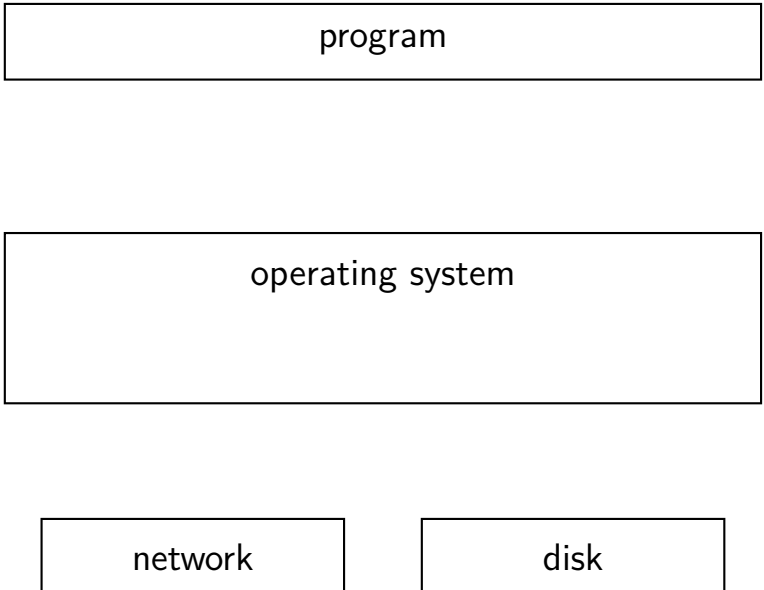
kernel buffering (reads)



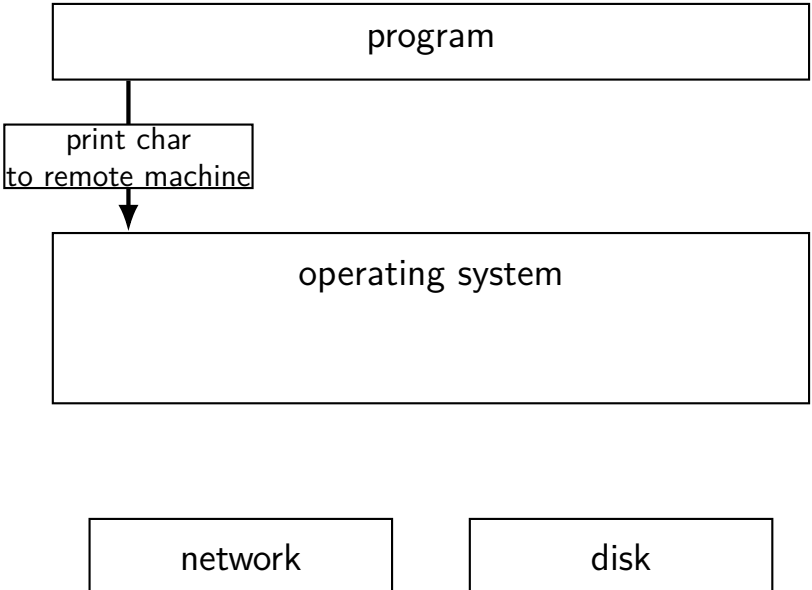
kernel buffering (reads)



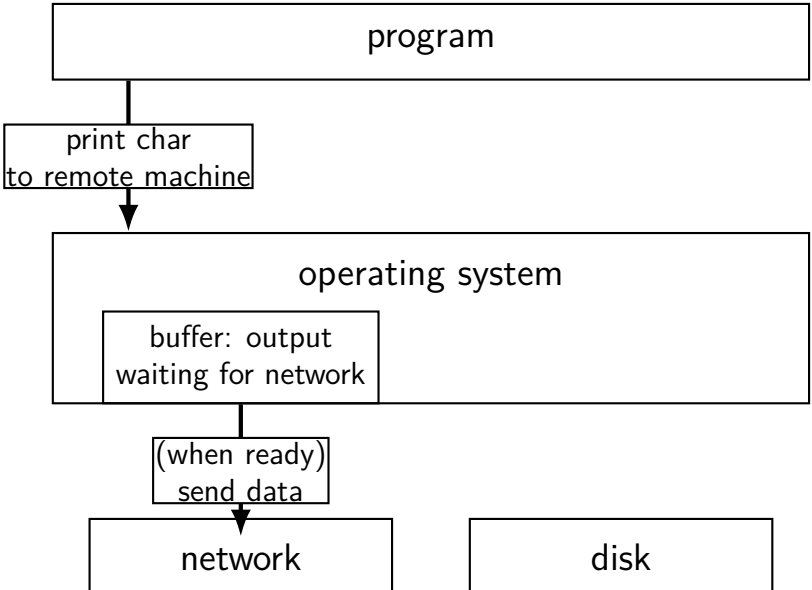
kernel buffering (writes)



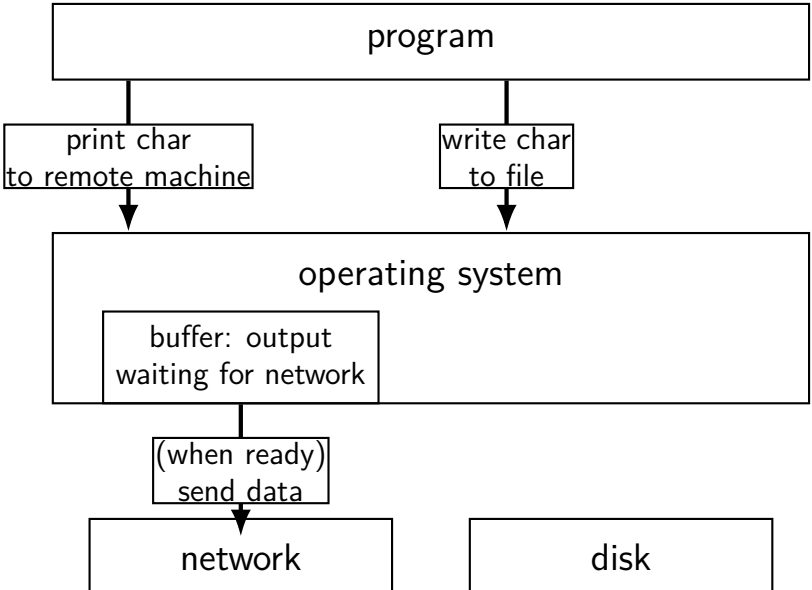
kernel buffering (writes)



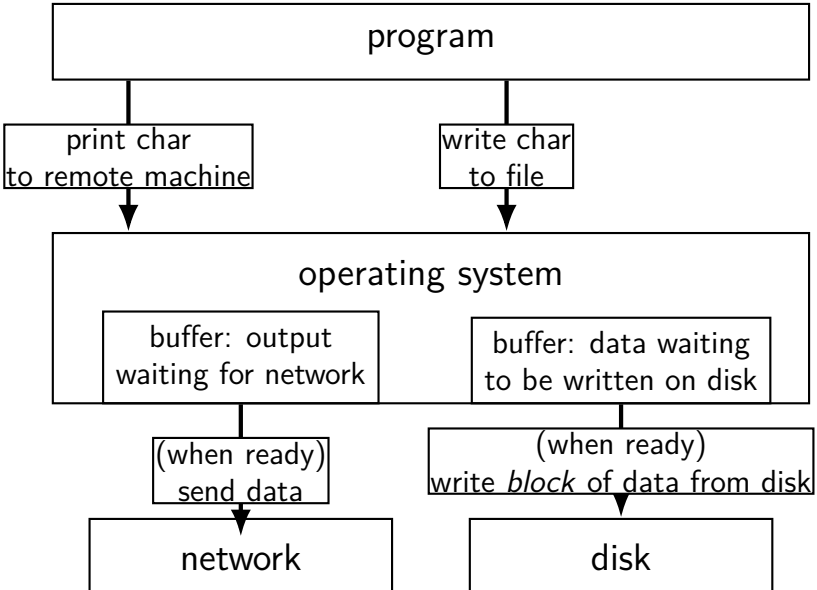
kernel buffering (writes)



kernel buffering (writes)



kernel buffering (writes)



read/write operations

read()/write(): move data into/out of buffer

possibly wait if buffer is empty (read)/full (write)

actual I/O operations — wait for device to be ready

- trigger process to stop waiting if needed

filesystem abstraction

regular files — named collection of bytes

- also: size, modification time, owner, access control info, ...

directories — folders containing files and directories

- hierarchical naming: `/net/zf14/cr4bd/fall2018/cs4414`
- *mostly* contains regular files or directories

open

```
int open(const char *path, int flags);
int open(const char *path, int flags, int mode);
...

int read_fd = open("dir/file1", O_RDONLY);
int write_fd = open("/other/file2",
        O_WRONLY | O_CREAT | O_TRUNC, 0666);
int rdwr_fd = open("file3", O_RDWR);
```

open

```
int open(const char *path, int flags);
int open(const char *path, int flags, int mode);
```

path = filename

e.g. `"/foo/bar/file.txt"`
`file.txt` in
directory `bar` in
directory `foo` in
"the root directory"

e.g. `"quux/other.txt`
`other.txt` in
directory `quux` in
"the current working directory" (set with `chdir()`)

open: file descriptors

```
int open(const char *path, int flags);
int open(const char *path, int flags, int mode);
```

return value = file descriptor (or -1 on error)

index into table of *open file descriptions* for each process

used by system calls that deal with open files

POSIX: everything is a file

the file: one interface for

- devices (terminals, printers, ...)
- regular files on disk
- networking (sockets)
- local interprocess communication (pipes, sockets)

basic operations: open(), read(), write(), close()

exercise

```
int pipe_fds[2]; pipe(pipe_fds);
pid_t p = fork();
if (p == 0) {
  close(pipe_fds[0]);
  for (int i = 0; i < 10; ++i) {
    char c = '0' + i;
    write(pipe_fds[1], &c, 1);
  }
  exit(0);
}
close(pipe_fds[1]);
char buffer[10];
ssize_t count = read(pipe_fds[0], buffer, 10);
for (int i = 0; i < count; ++i) {
  printf("%c", buffer[i]);
}
```

Which of these are possible outputs (if pipe, read, write, fork don't fail)?

A. `0123456789` B. `0` C. (nothing)

D. A and B E. A and C F. A, B, and C

exercise

```
int pipe_fds[2]; pipe(pipe_fds);
pid_t p = fork();
if (p == 0) {
  close(pipe_fds[0]);
  for (int i = 0; i < 10; ++i) {
    char c = '0' + i;
    write(pipe_fds[1], &c, 1);
  }
  exit(0);
}
close(pipe_fds[1]);
char buffer[10];
ssize_t count = read(pipe_fds[0], buffer, 10);
for (int i = 0; i < count; ++i) {
  printf("%c", buffer[i]);
}
```

Which of these are possible outputs (if pipe, read, write, fork don't fail)?

A. `0123456789` B. `0` C. (nothing)

D. A and B E. A and C F. A, B, and C

empirical evidence

```
  8 0
374 01
210 012
 30 0123
 12 01234
  3 012345
  1 0123456
  2 01234567
  1 012345678
359 0123456789
```

partial reads

read returning 0 always means end-of-file

- by default, read always waits *if no input available yet*
- but can set read to return *error* instead of waiting

read can return less than requested if not available

- e.g. child hasn't gotten far enough

pipe: closing?

if all write ends of pipe are closed

- can get end-of-file (read() returning 0) on read end
- exit()ing closes them

→ close write end when not using

generally: limited number of file descriptors per process

→ good habit to close file descriptors not being used

(but probably didn't matter for read end of pipes in example)

dup2 exercise

recall: dup2(old_fd, new_fd)

```
int fd = open("output.txt", O_WRONLY | O_CREAT, 0666);
write(STDOUT_FILENO, "A", 1);
dup2(fd, STDOUT_FILENO);
pid_t pid = fork();
if (pid == 0) { /* child: */
    dup2(STDOUT_FILENO, fd); write(fd, "B", 1);
} else {
    write(STDOUT_FILENO, "C", 1);
}
```

Which outputs are possible?

A. stdout: ABC ; output.txt: empty
B. stdout: AC ; output.txt: B
C. stdout: A ; output.txt: CB
D. stdout: A ; output.txt: BC
E. more?