last time

`fork` — create new process by copying current

- returns twice
- new pid in original 'parent' process
- 0 (sentinel value) in 'child' process
- both processes are running immediately afterwards (potentially at different speeds)

`exec*` — replace program in current process

- specify executable file to load
- copies arguments into new program's argv

anonymous feedback (1)

"I feel like the feedback left for this class gets disregarded. Student put in tons of time to write out issues they have with the class many of which get skimmed over in seconds with a response like "some students said the lab was too hard, I don't think so". This comes off as condescending and it feels very discouraging...."

did think lab was too long, but not going to fix this semester, so didn't talk much about it
(also needed to talk to TAs first to know what might/might not help)
I didn't think the lab was way too long based on reports of how many finished/etc.
likely supply shared memory code next semester

(also some other issues with lab size — e.g. next lab probably may be too short)

anonymous feedback (2)

"…I understand that most students may not be experiencing difficulties with the class and that it may seem like a waste to spend time addressing issues that only a small group of students have with the class but I think that it's still very important that these issues get addressed in order to help catch people having issues with the class up to speed. For me personally the class at times feels like a light general coverage of the concepts covered followed by a deep dive that focuses on particular edge cases and interactions between concepts. I wish we could spend just a little more time on the bigger picture of some of these topics before going too in depth with them."

"Would it be possible to spend a little bit longer on the importance of methods before delving into examples of their use and questions about them? I feel like we're not given enough information to solve most questions and it leaves me confused as to if I'm taking away what I'm supposed to from the lecture"

- some theme of more 'bigger picture' stuff…
- but my impression of bigger picture maybe not yours???
  - are more examples of motivational use cases bigger picture?
- specifics are usually easier for me to actually improve things

anonymous feedback (3)

"I'd like it if the quizzes opened a little sooner on Thursdays so I could work on the quizzes after class (while the lecture info is still fresh in my mind)"

probably could do earlier most weeks since quiz is mostly written in advance

but sometimes late edits and allowing about enough time for some TAs to look over

times were set based on when I could consistently have quizzes ready

on quiz grading

am aware we're behind on Q2 grading

Q2 (B+D)

signal handlers run in process, go back to same process
(via OS code to cleanup handler)

return address can't go to other process since one process can't access another's memory

kill() might not take effect immediately

if program is running on another core or not running yet

also, program can continue after kill while other sighandler runs

system records signal and acts on it later

Q3 (B)

PID reuse:

- possible after waited for a process
- (but not early so you waitpid isn't ambiguous)

Q4 (1)

should have specified other processes not sending signals to these

child gets other_pid==0 [sentinal value, not a real PID]

parent gets other_pid==(pid of child)

SIGUSR1 in child triggers SIGUSR2 in parent

SIGUSR1 in parent triggers SIGUSR2 in child

assumption: signals handled one at a time

- not ensured by given code, but could be by setting sa_mask

Q4 (2)

A: 0:SIGUSR1 0:SIGUSR2 43:SIGUSR1 43:SIGUSR2, 0

> parent SIGUSR1, then parent SIGUSR2, without child's SIGUSR1 running first

B: 0:SIGUSR1 32:SIGUSR2 32:SIGUSR1 0:SIGUSR2

> 32:SIGUSR1 printed, but parent would have exit() after printing 32:SIGUSR2

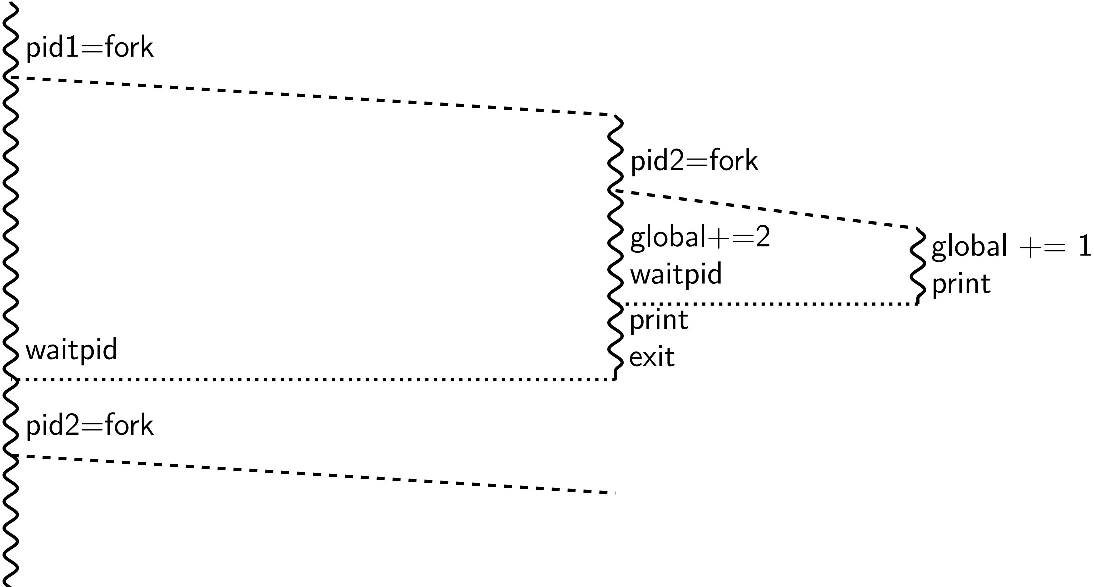
C: 0:SIGUSR1 43:SIGUSR1 44:SIGUSR2 0:SIGUSR2

> three different other_pid values

D: 0:SIGUSR1 50:SIGUSR2 0:SIGUSR2

> parent should've printed 50:SIGUSR1 before sending SIGUSR2 to child

Q6



pid1=fork
pid2=fork
global+=2
waitpid
global += 1
print
print
exit
waitpid
pid2=fork

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`

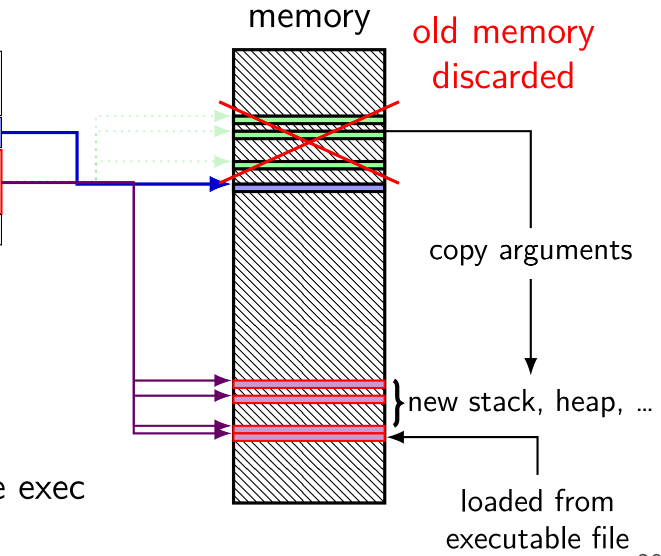
like we copied and pasted the output into that file

(as it was written)

exec preserves open files

the process control block

user regs	eax=~~42~~*init. val.*, ecx=~~133~~*init. val.*, …
pagetable	
open files	fd 0: (terminal …) fd 1: …
…	…

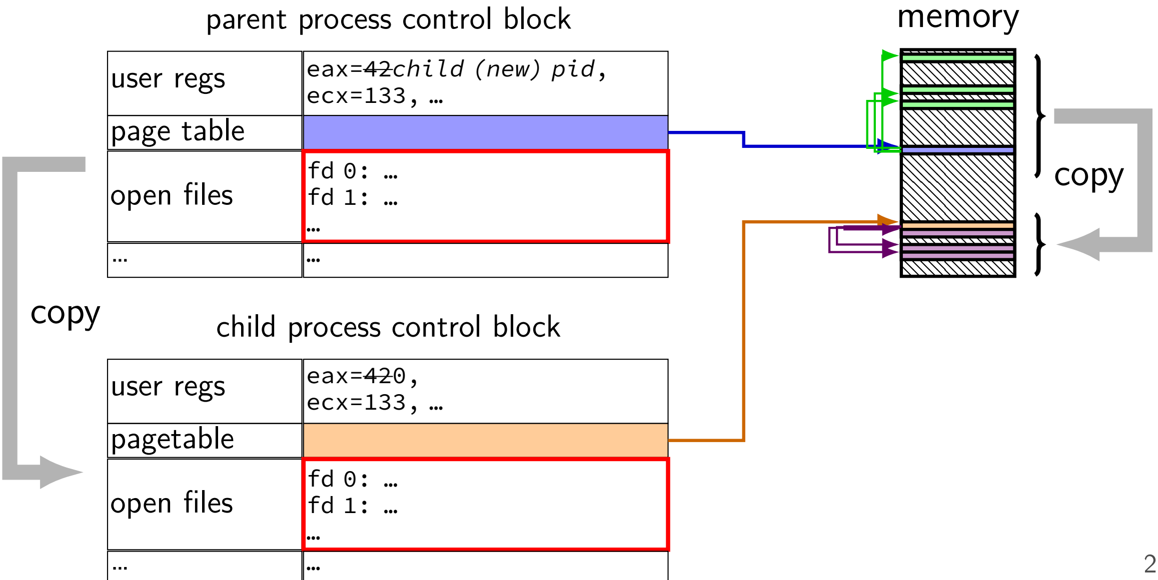


not changed!

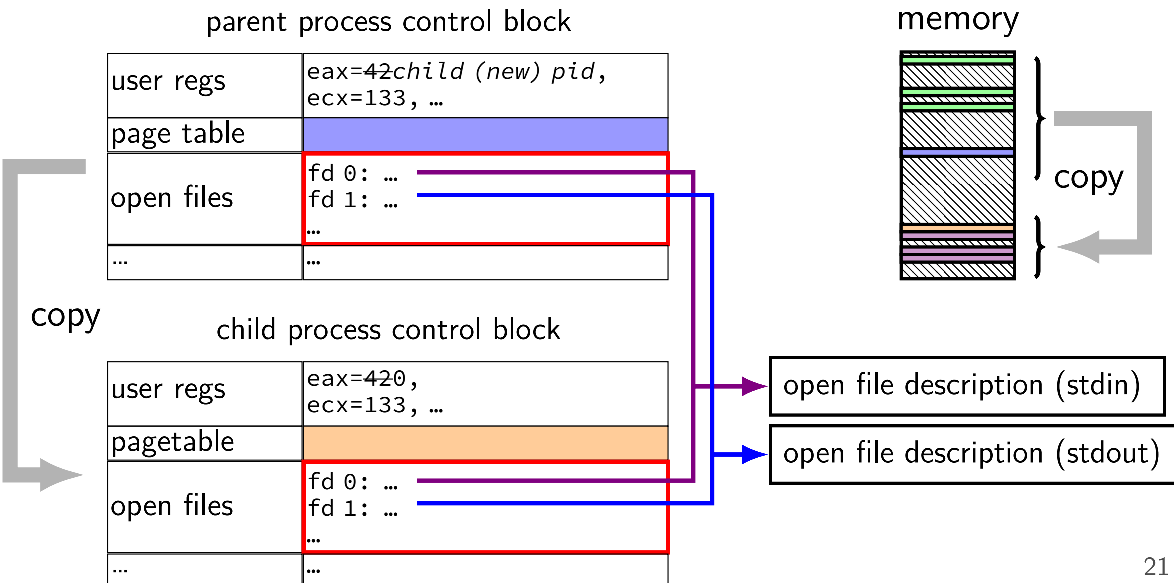
redirection/etc.:

setup stdin/stdout before exec

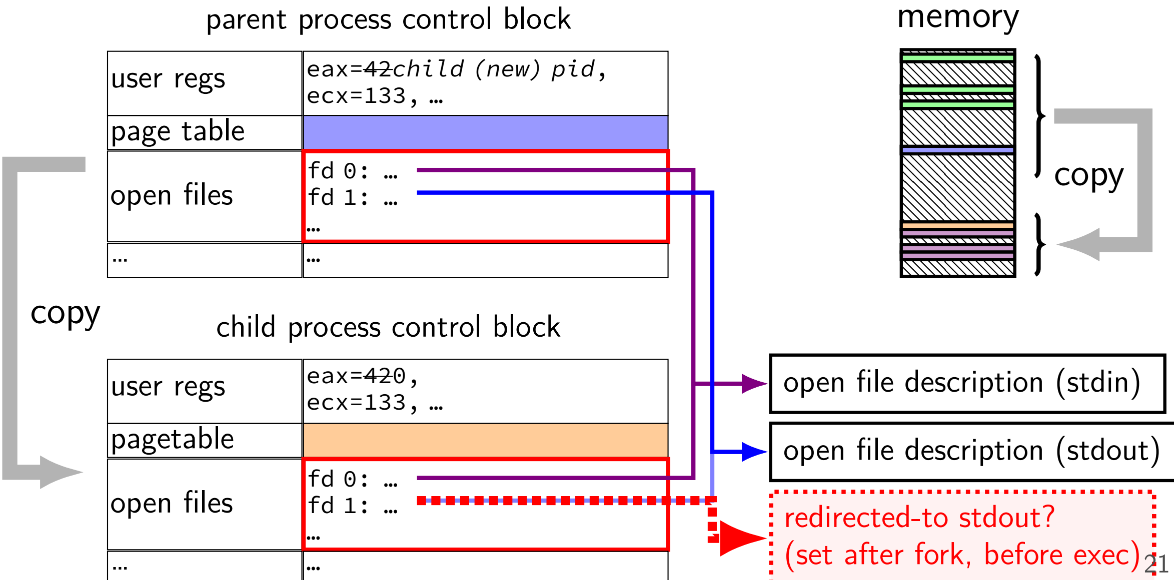
fork copies open file list



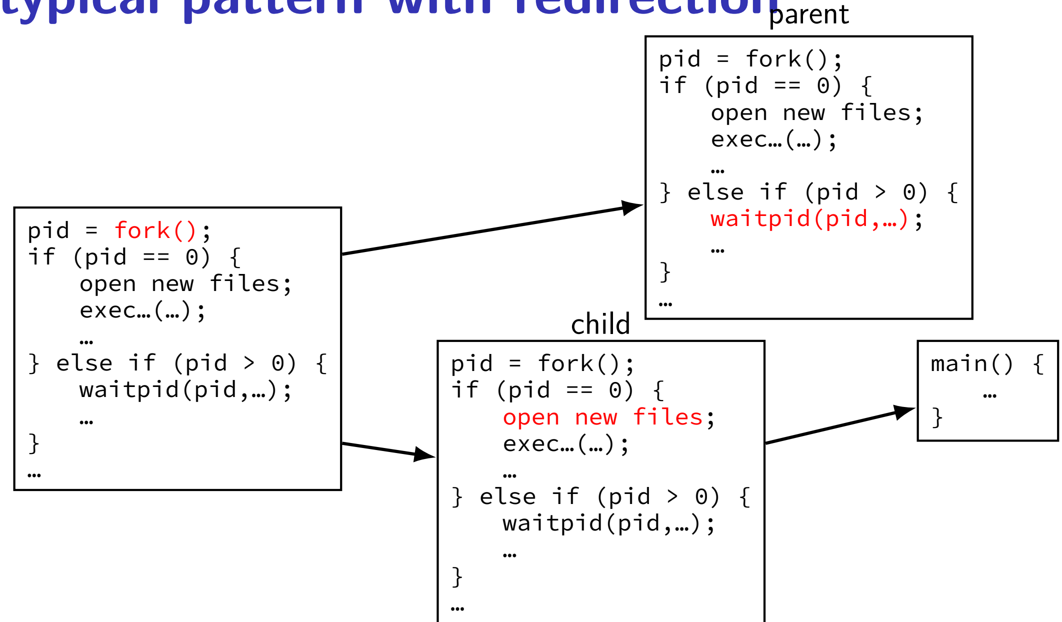
fork copies open file list



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];
```

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 example (1)

```
int pipe_fd[2];
if (pipe(pipe_fd) < 0)
    handle_error(); /* e.g. out of file descriptors */
int read_fd = pipe_fd[0];
int write_fd = pipe_fd[1];
child_pid = fork();
if (child_pid  == 0) {
    /* in child process, write to pipe */
    close(read_fd);
    write_to_pipe(write_fd); /* function not shown */
    exit(EXIT_SUCCESS);
} else if (child_pid > 0) {
    /* in parent process, read from pipe */
    close(write_fd);
    read_from_pipe(read_fd); /* function not shown */
    waitpid(child_pid, NULL, 0);
    close(read_fd);
} else { /* fork error */ }
```

pipe example (1)

'standard' pattern with fork()

```
int pipe_fd[2];
if (pipe(pipe_fd) < 0)
    handle_error(); /* e.g. out of file descriptors */
int read_fd = pipe_fd[0];
int write_fd = pipe_fd[1];
child_pid = fork();
if (child_pid  == 0) {
    /* in child process, write to pipe */
    close(read_fd);
    write_to_pipe(write_fd); /* function not shown */
    exit(EXIT_SUCCESS);
} else if (child_pid > 0) {
    /* in parent process, read from pipe */
    close(write_fd);
    read_from_pipe(read_fd); /* function not shown */
    waitpid(child_pid, NULL, 0);
    close(read_fd);
} else { /* fork error */ }
```

pipe example (1)

```
int pipe_fd[2];
if (pipe(pipe_fd) < 0)
    handle_error(); /* e.g. out of file
int read_fd = pipe_fd[0];
int write_fd = pipe_fd[1];
child_pid = fork();
if (child_pid  == 0) {
    /* in child process, write to pipe */
    close(read_fd);
    write_to_pipe(write_fd); /* function not shown */
    exit(EXIT_SUCCESS);
} else if (child_pid > 0) {
    /* in parent process, read from pipe */
    close(write_fd);
    read_from_pipe(read_fd); /* function not shown */
    waitpid(child_pid, NULL, 0);
    close(read_fd);
} else { /* fork error */ }
```

read() will not indicate end-of-file if write fd is open (any copy of it)

pipe example (1)

```
int pipe_fd[2];
if (pipe(pipe_fd) < 0)
    handle_error(); /* e.g. out of fi
int read_fd = pipe_fd[0];
int write_fd = pipe_fd[1];
child_pid = fork();
if (child_pid  == 0) {
    /* in child process, write to pipe */
    close(read_fd);
    write_to_pipe(write_fd); /* function not shown */
    exit(EXIT_SUCCESS);
} else if (child_pid > 0) {
    /* in parent process, read from pipe */
    close(write_fd);
    read_from_pipe(read_fd); /* function not shown */
    waitpid(child_pid, NULL, 0);
    close(read_fd);
} else { /* fork error */ }
```

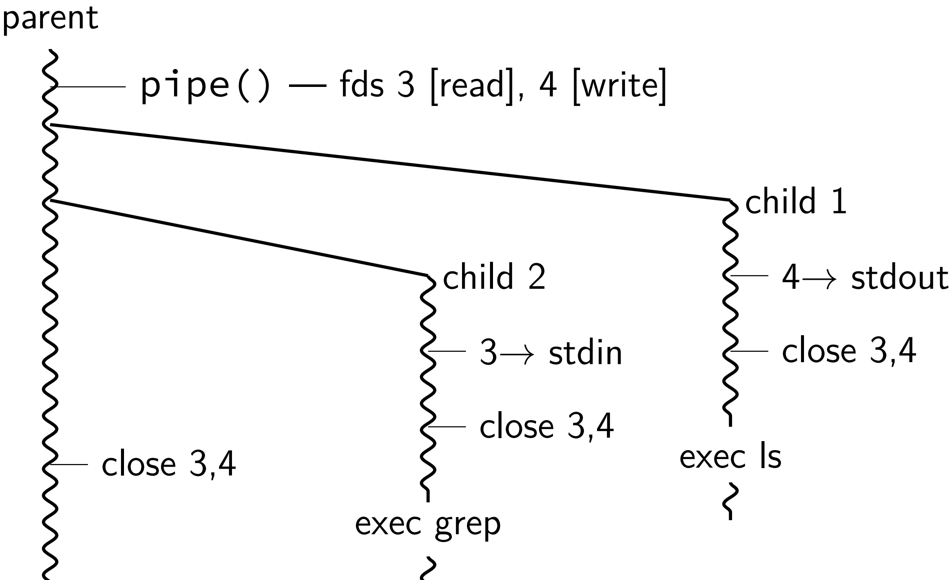
have habit of closing
to avoid 'leaking' file descriptors
you can run out

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



exercise

```
pid_t p = fork();
int pipe_fds[2];
pipe(pipe_fds);
if (p == 0) { /* child */
  close(pipe_fds[0]);
  char c = 'A';
  write(pipe_fds[1], &c, 1);
  exit(0);
} else { /* parent */
  close(pipe_fds[1]);
  char c;
  int count = read(pipe_fds[0], &c, 1);
  printf("read %d bytes\n", count);
}
```

The child is trying to send the character A to the parent, but the above code outputs `read 0 bytes` instead of `read 1 bytes`. What happened?

exercise solution

pipe() is after fork — two pipes, one in child, one in parent

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

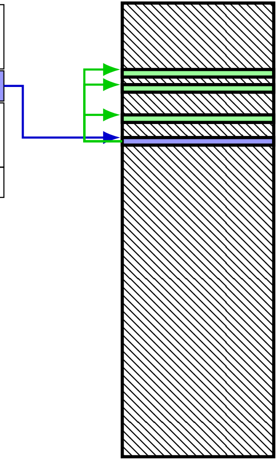
backup slides

fork and process info (w/o copy-on-write)

parent process info

user regs	`rax (return val.)=42,` `rcx=133, …`
page tables	
open files	`fd 0: …` `fd 1: …`
…	…

memory



fork and process info (w/o copy-on-write)

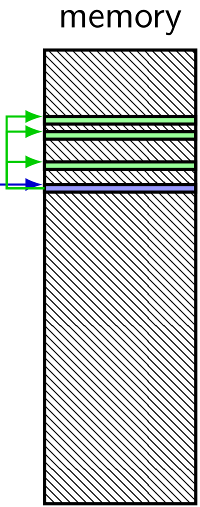
parent process info

user regs	rax (return val.)=42, rcx=133, …
page tables	
open files	fd 0: … fd 1: …
…	…

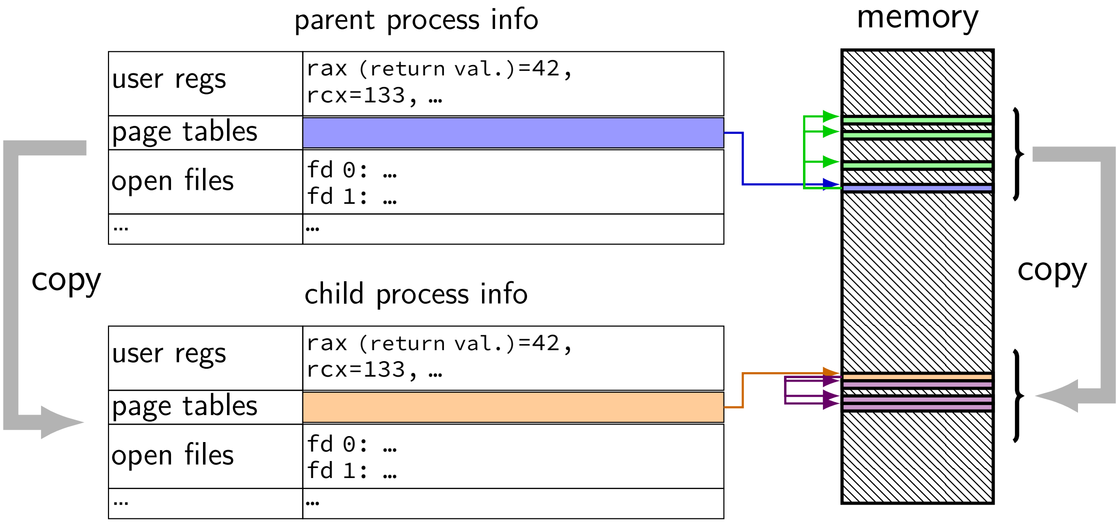
copy

child process info

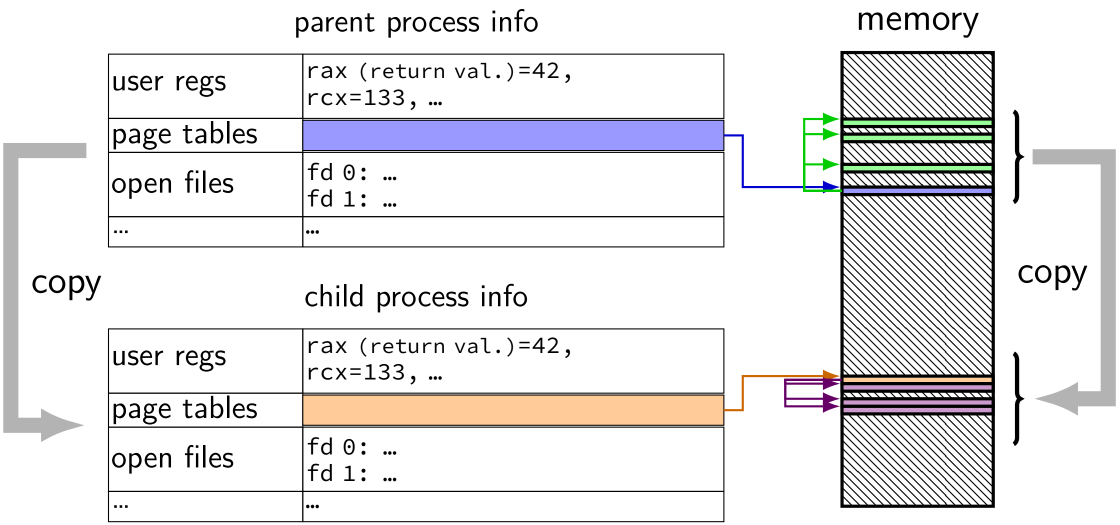
user regs	rax (return val.)=42, rcx=133, …
page tables	
open files	fd 0: … fd 1: …
…	…



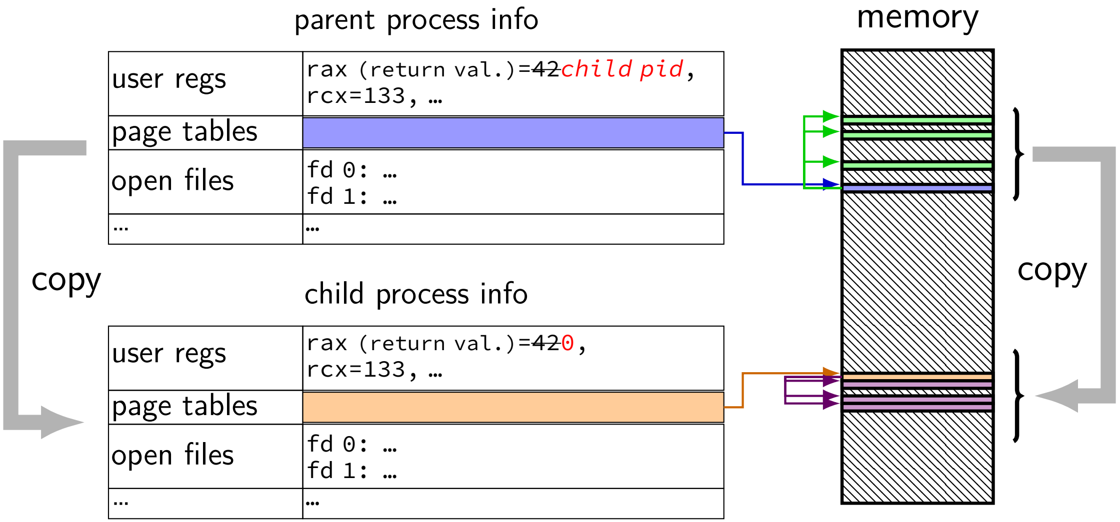
fork and process info (w/o copy-on-write)



fork and process info (w/o copy-on-write)



fork and process info (w/o copy-on-write)



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

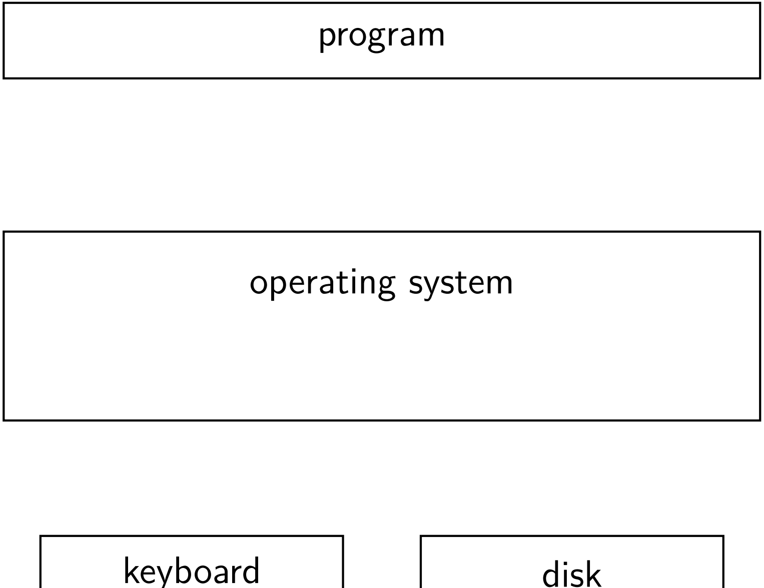
environment variables = key/value pairs stored with process

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

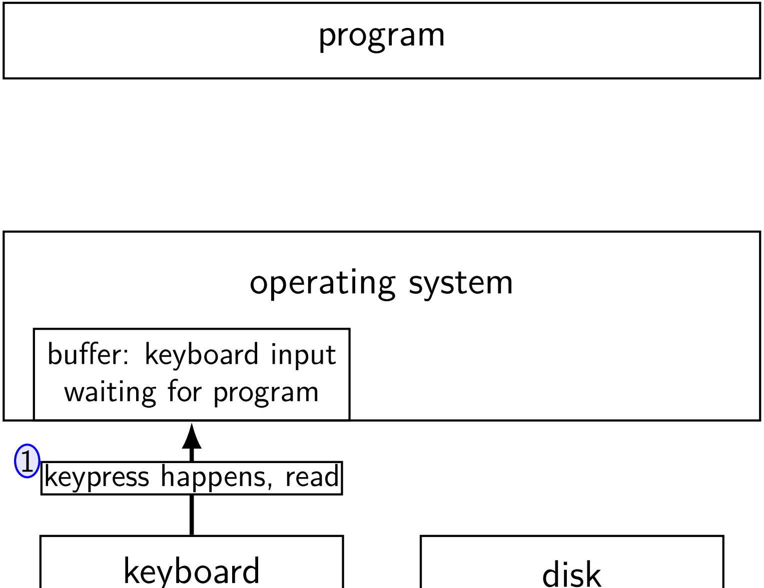
one way to implement: [pseudocode]

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

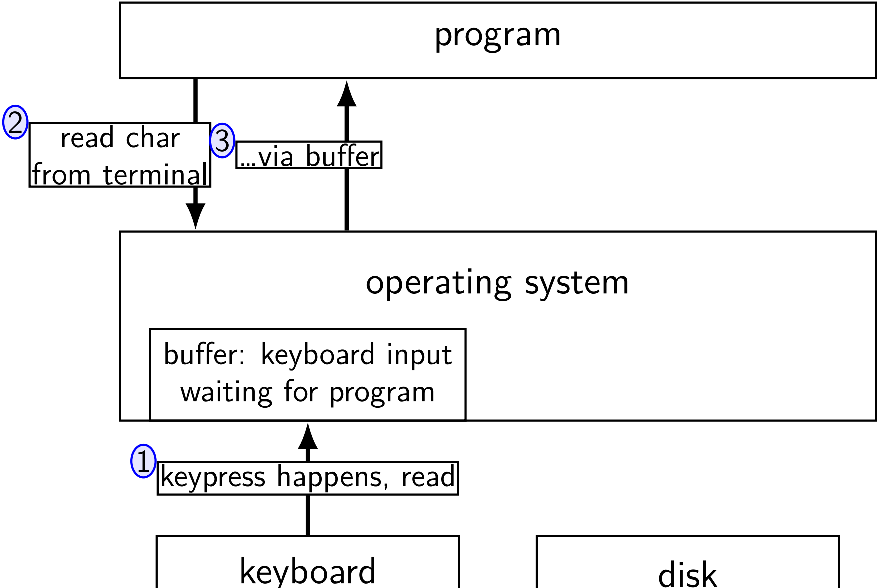
kernel buffering (reads)



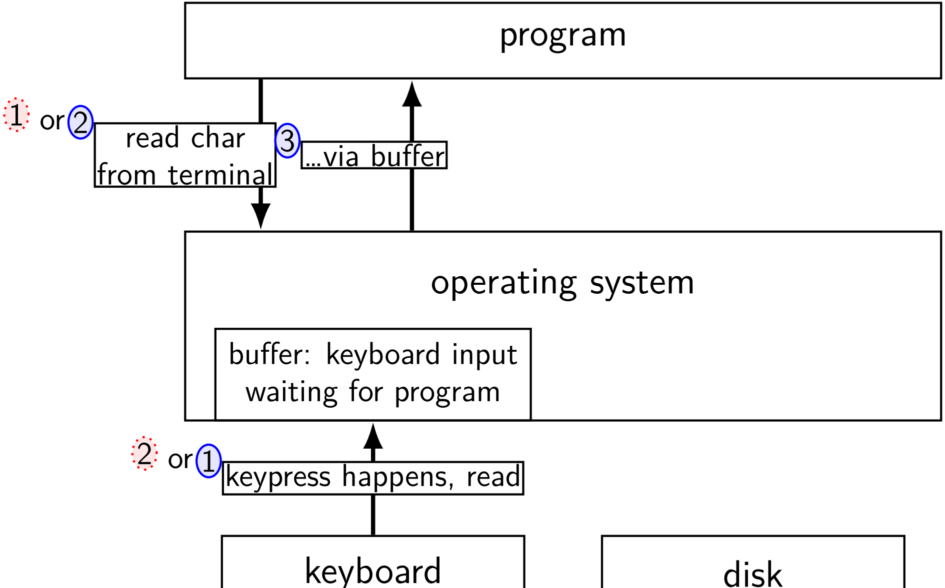
kernel buffering (reads)



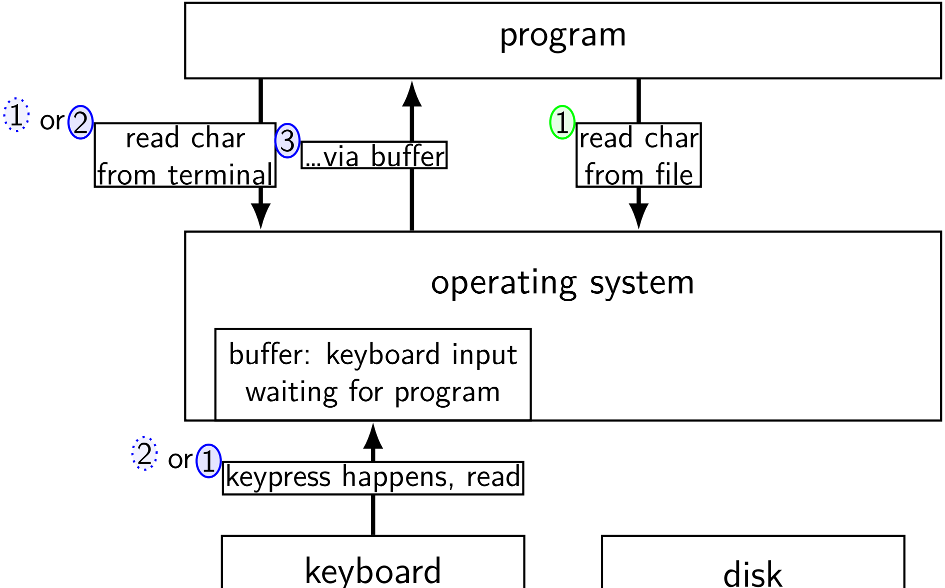
kernel buffering (reads)



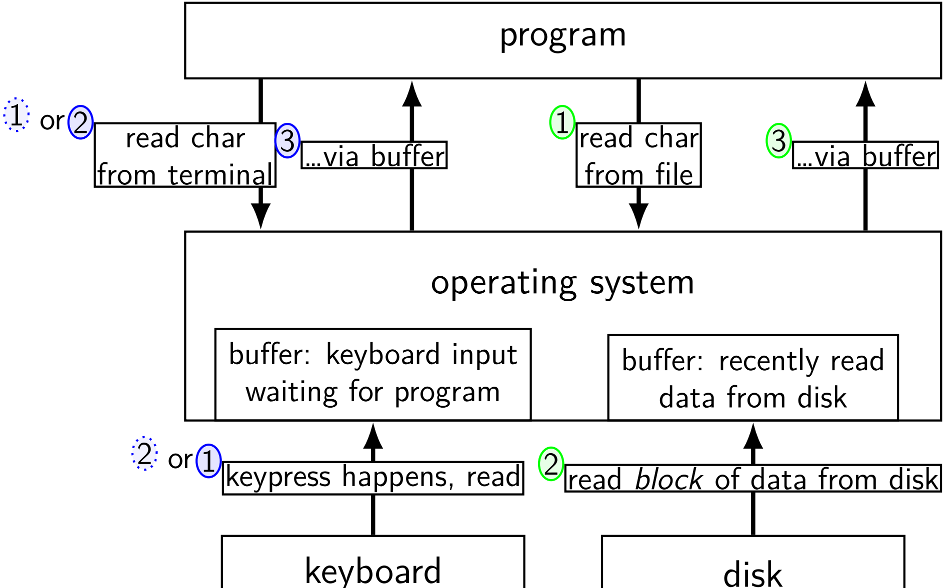
kernel buffering (reads)



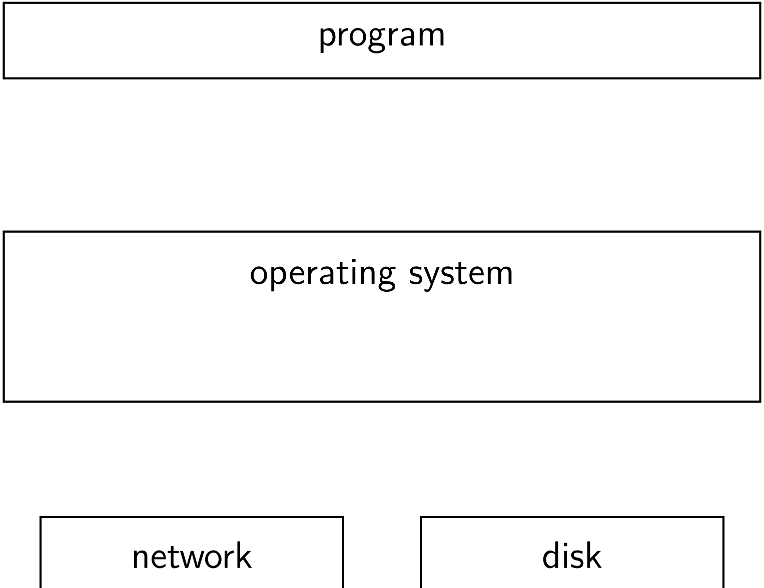
kernel buffering (reads)



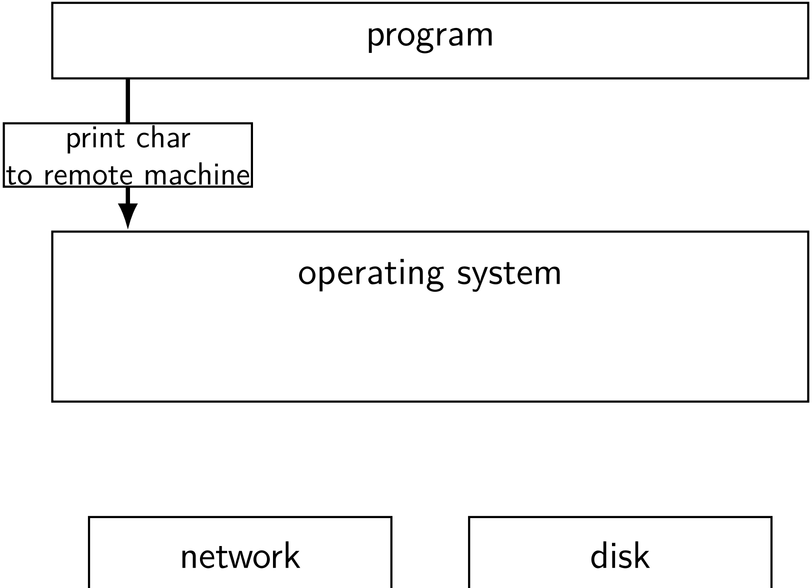
kernel buffering (reads)



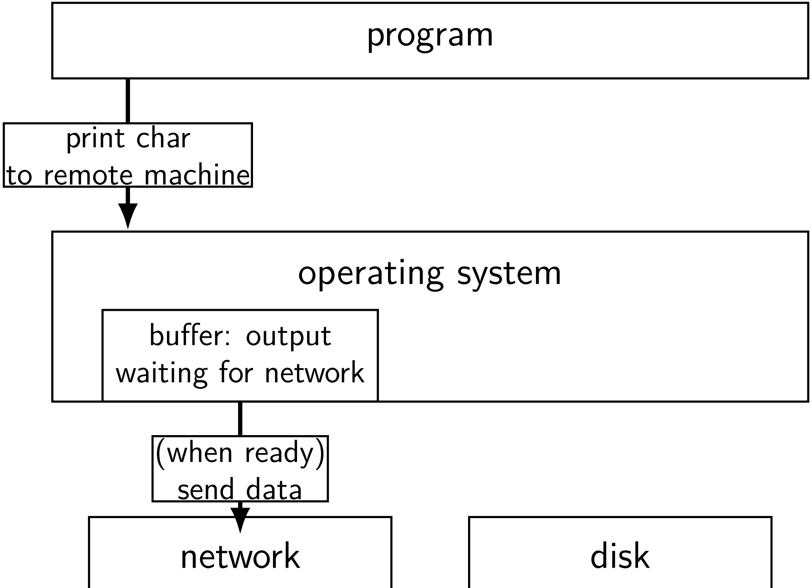
kernel buffering (writes)



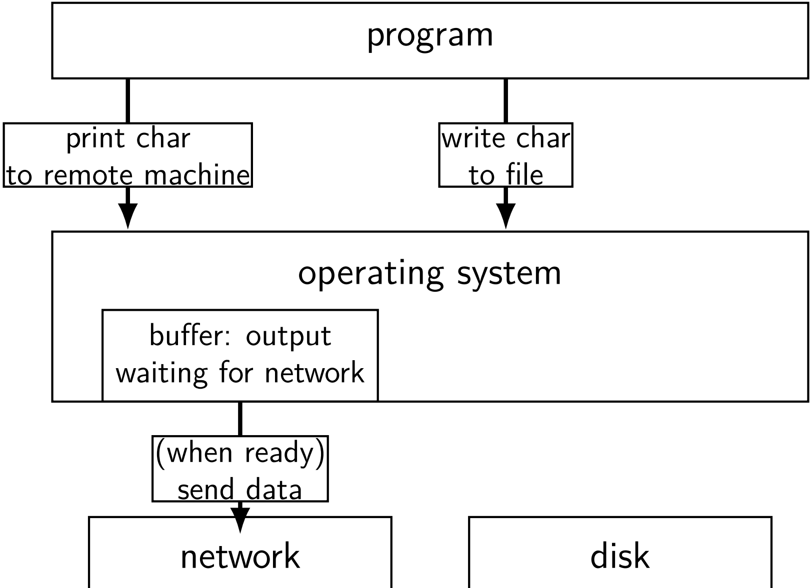
kernel buffering (writes)



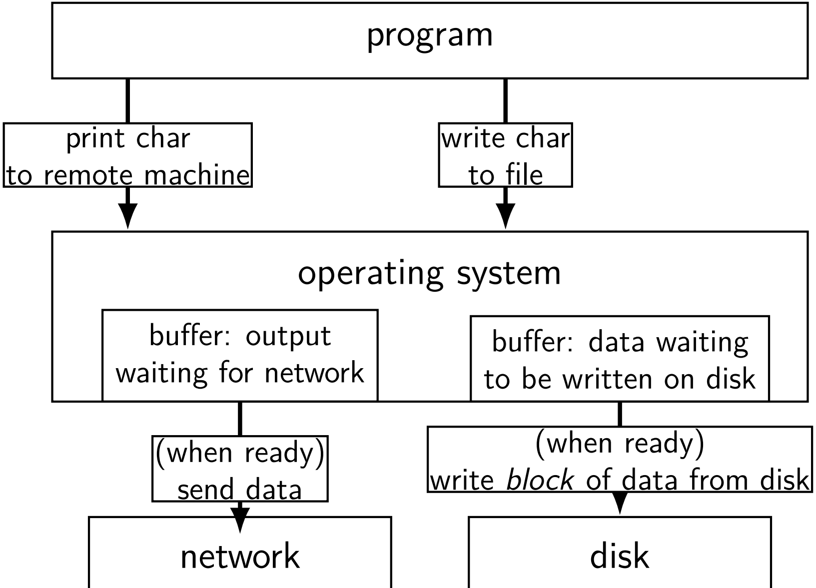
kernel buffering (writes)



kernel buffering (writes)



kernel buffering (writes)



read/write operations

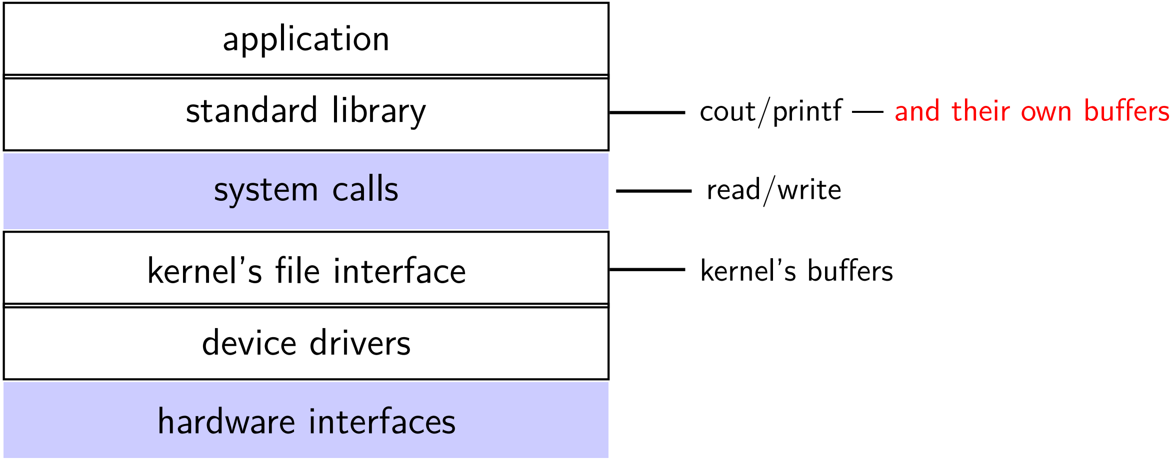
read()/write(): move data into/out of buffer

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

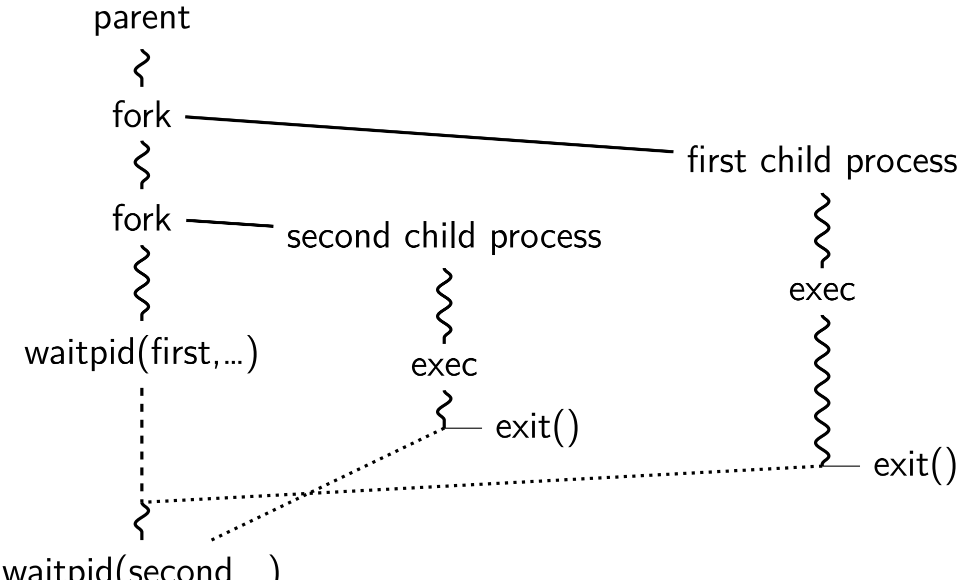
pipe() and blocking

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

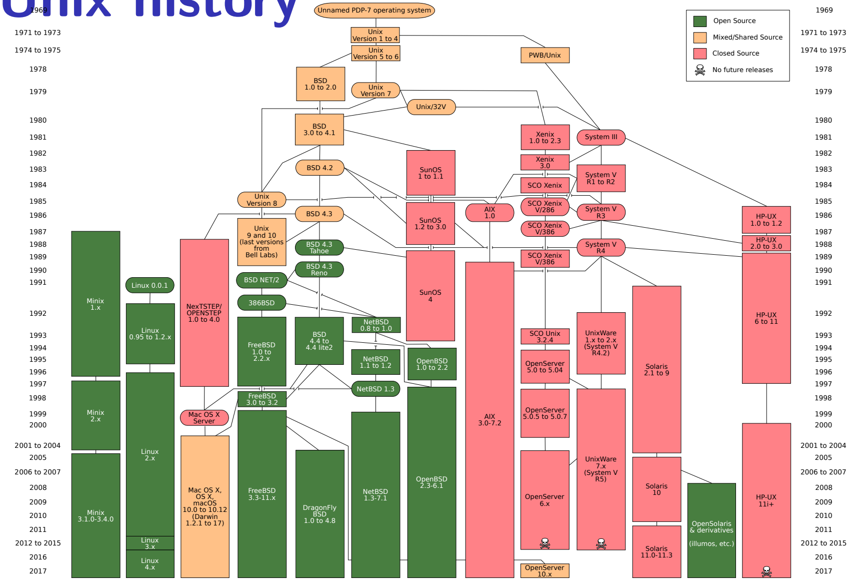


image: Wikpedia/Eraserhead1+Infinity0+Sav_vas

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):

```
init(1)-+-ModemManager(919)-+-{ModemManager}(972)
        |                   `-{ModemManager}(1064)
        |-NetworkManager(1160)-+-dhclient(1755)
        |                      |-dnsmasq(1985)
        |                      |-{NetworkManager}(1180)
        |                      |-{NetworkManager}(1194)
        |                      `-{NetworkManager}(1195)
        |-accounts-daemon(1649)-+-{accounts-daemon}(1757)
        |                       `-{accounts-daemon}(1758)
        |-acpid(1338)
        |-apache2(3165)-+-apache2(4125)-+-{apache2}(4126)
        |               |               `-{apache2}(4127)
        |               |-apache2(28920)-+-{apache2}(28926)
        |               |                `-{apache2}(28960)
        |               |-apache2(28921)-+-{apache2}(28927)
        |               |                `-{apache2}(28963)
        |               |-apache2(28922)-+-{apache2}(28928)
        |               |                `-{apache2}(28961)
        |               |-apache2(28923)-+-{apache2}(28930)
        |               |                `-{apache2}(28962)
        |               |-apache2(28925)-+-{apache2}(28958)
        |               |                `-{apache2}(28965)
        |               `-apache2(32165)-+-{apache2}(32166)
        |                                `-{apache2}(32167)
        |-at-spi-bus-laun(2252)-+-dbus-daemon(2269)
        |                       |-{at-spi-bus-laun}(2266)
        |                       |-{at-spi-bus-laun}(2268)
        |                       `-{at-spi-bus-laun}(2270)
        |-at-spi2-registr(2275)---{at-spi2-registr}(2282)
        |-atd(1633)
        |-automount(13454)-+-{automount}(13455)
        |                  |-{automount}(13456)
        |                  |-{automount}(13461)
        |                  |-{automount}(13464)
        |                  `-{automount}(13465)
```

```
|                    `-{mcollectived}(2038)
|-mongod(1336)-+-{mongod}(1556)
|              |-{mongod}(1557)
|              |-{mongod}(1983)
|              |-{mongod}(2031)
|              |-{mongod}(2047)
|              |-{mongod}(2048)
|              |-{mongod}(2049)
|              |-{mongod}(2050)
|              |-{mongod}(2051)
|              `-{mongod}(2052)
|-mosh-server(19090)---bash(19091)---tmux(5442)
|-mosh-server(21996)---bash(21997)
|-mosh-server(22533)---bash(22534)---tmux(22588)
|-nm-applet(2580)-+-{nm-applet}(2739)
|                 `-{nm-applet}(2743)
|-nmbd(2224)
|-ntpd(3091)
|-polkitd(1197)-+-{polkitd}(1239)
|               `-{polkitd}(1240)
|-pulseaudio(2563)-+-{pulseaudio}(2617)
|                  `-{pulseaudio}(2623)
|-puppet(2373)---{puppet}(32455)
|-rpc.idmapd(875)
|-rpc.statd(954)
|-rpcbind(884)
|-rserver(1501)-+-{rserver}(1786)
|               `-{rserver}(1787)
|-rsyslogd(1090)-+-{rsyslogd}(1092)
|                |-{rsyslogd}(1093)
|                `-{rsyslogd}(1094)
|-rtkit-daemon(2565)-+-{rtkit-daemon}(2566)
|                    `-{rtkit-daemon}(2567)
|-sd_cicero(2852)-+-sd_cicero(2853)
|                 |-{sd_cicero}(2854)
|                 `-{sd_cicero}(2855)
|-sd_dummy(2849)-+-{sd_dummy}(2850)
|                `-{sd_dummy}(2851)
|-sd_espeak(2749)-+-{sd_espeak}(2845)
|                 |-{sd_espeak}(2846)
```

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;
    }
} while (offset != amount_to_read && amount_read != 0)
```

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

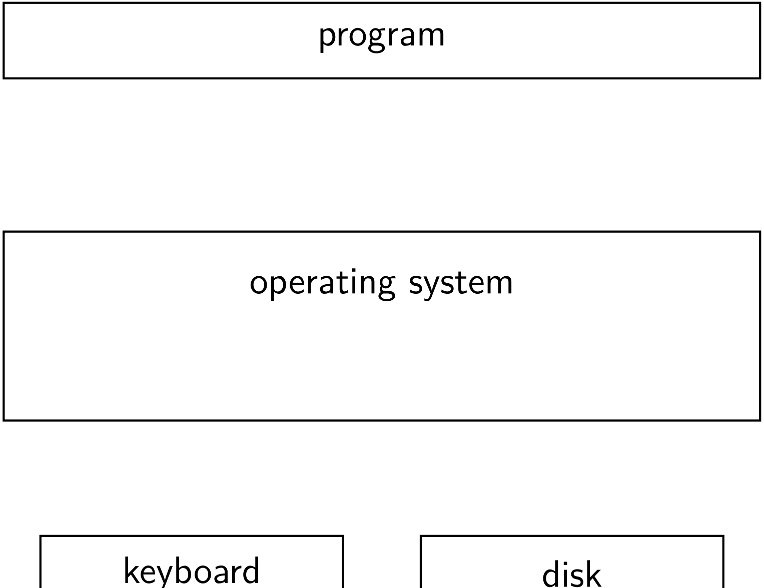
usually only happen on error or interruption

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

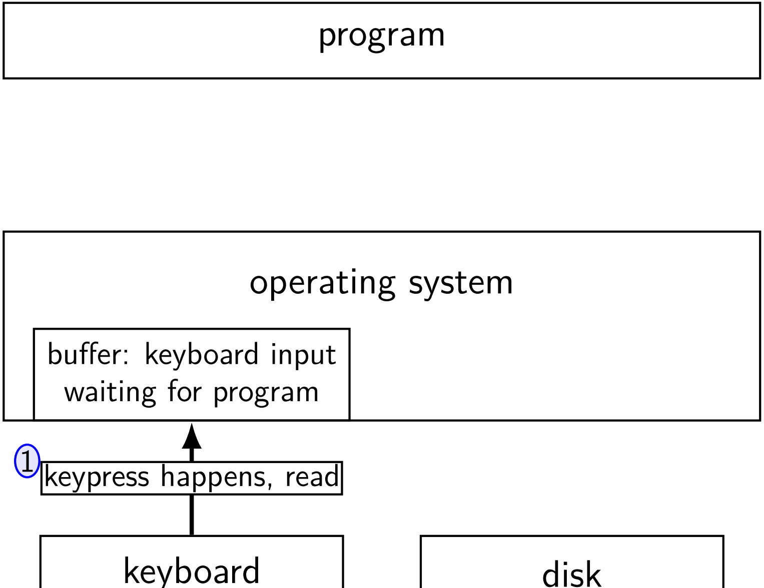
usually: write waits until it completes

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

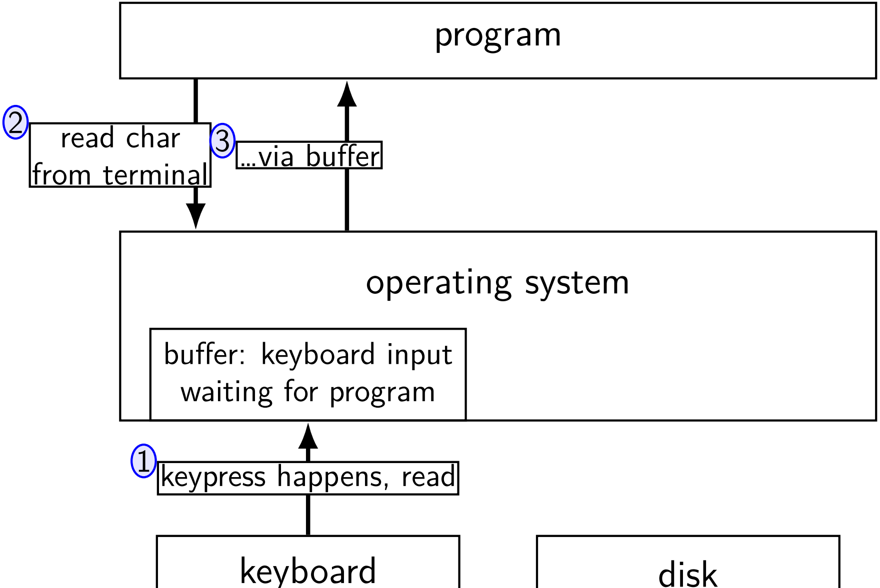
kernel buffering (reads)



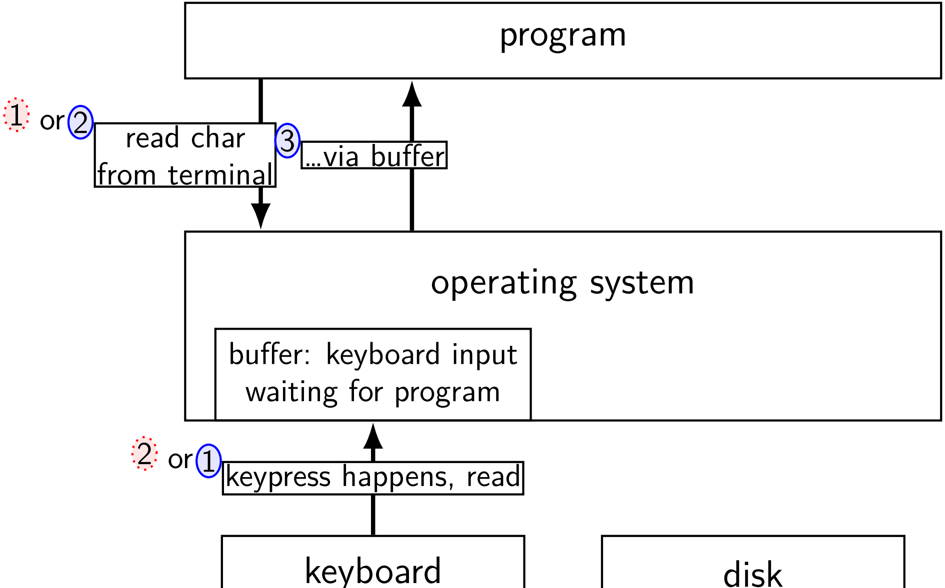
kernel buffering (reads)



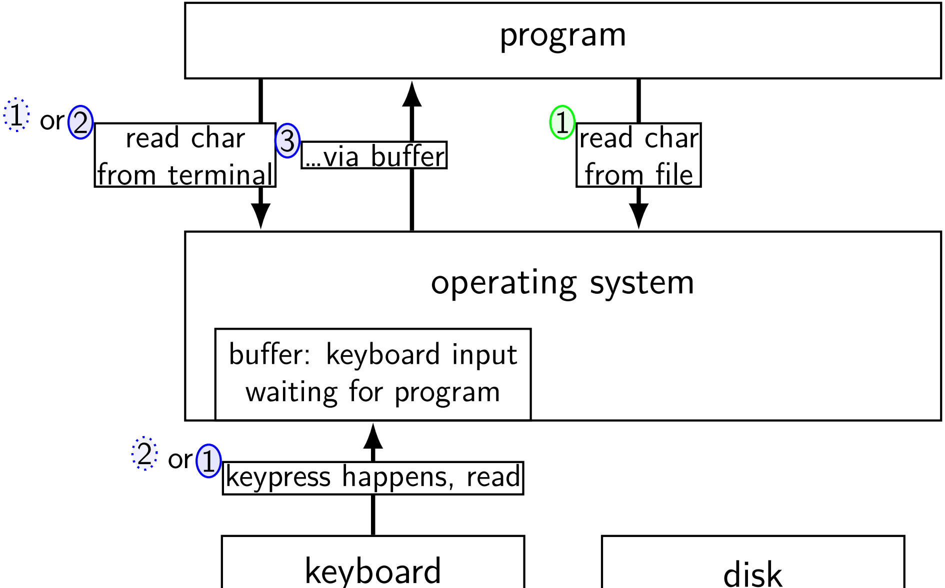
kernel buffering (reads)



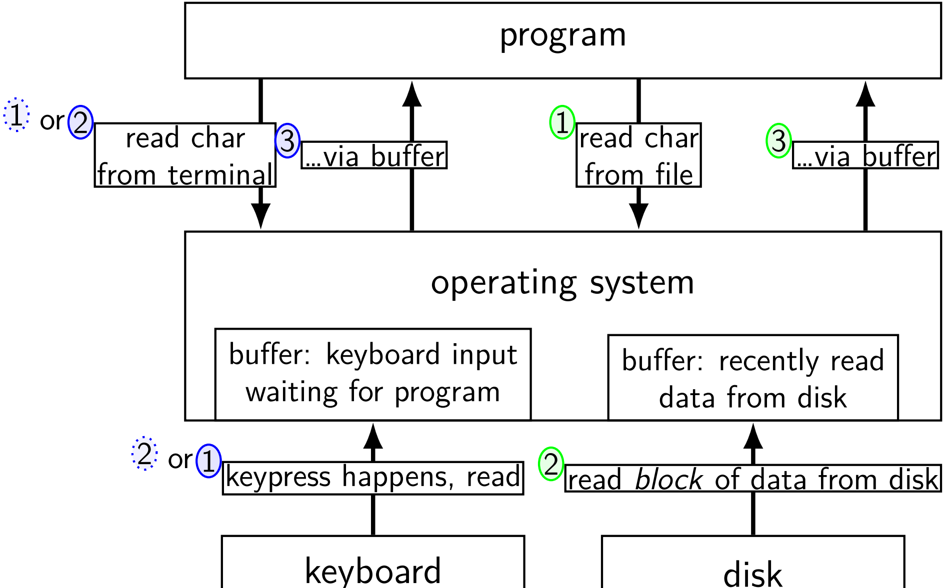
kernel buffering (reads)



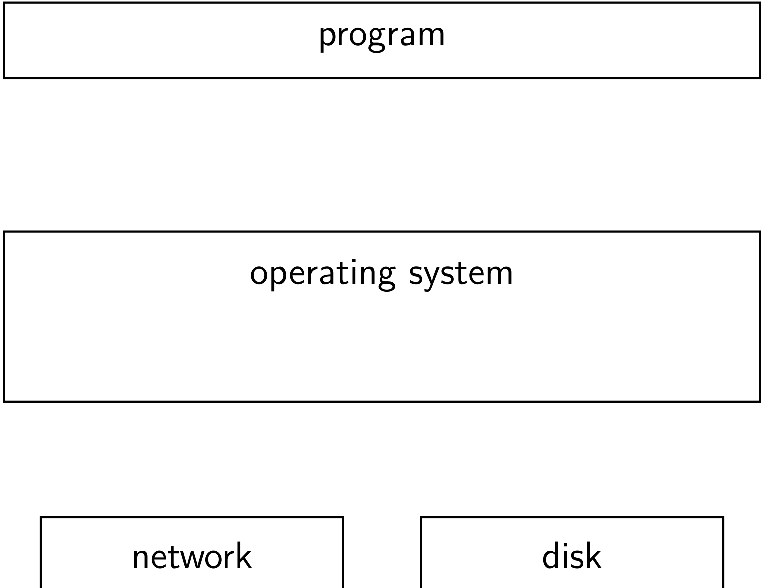
kernel buffering (reads)



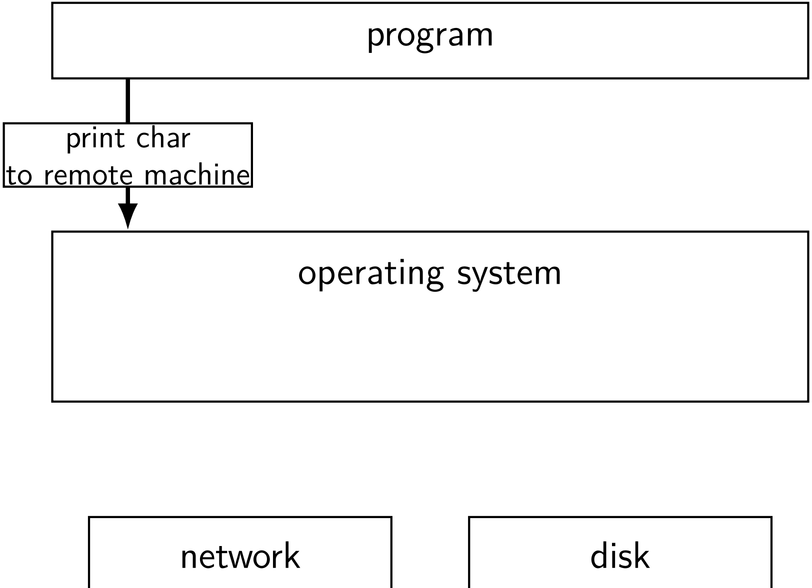
kernel buffering (reads)



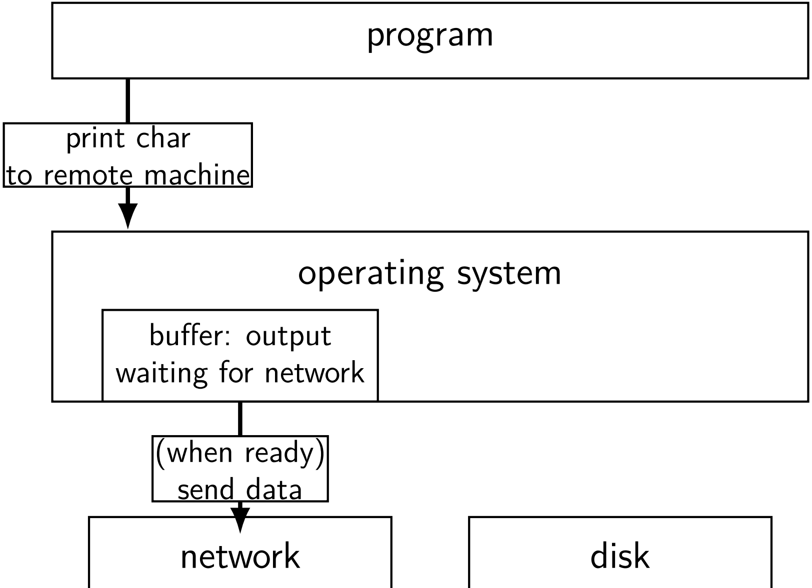
kernel buffering (writes)



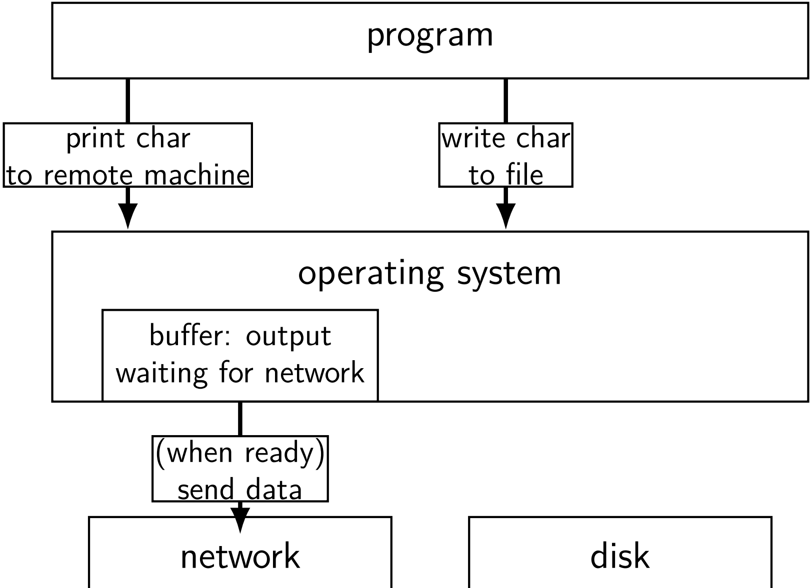
kernel buffering (writes)



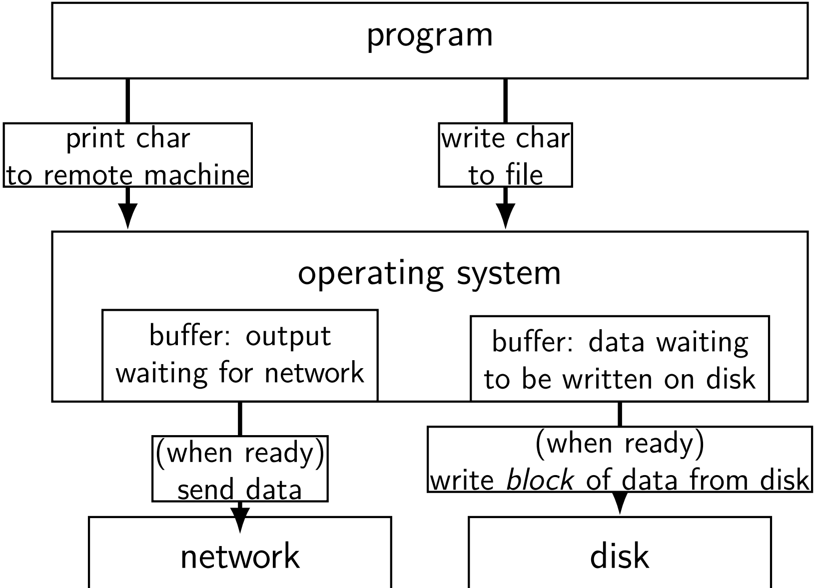
kernel buffering (writes)



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

```c
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

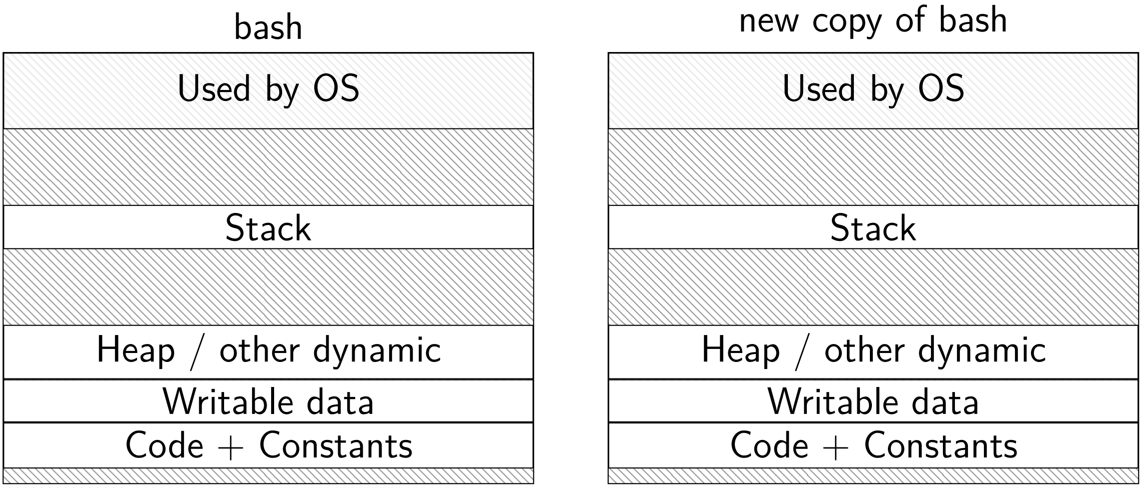
recall: `dup2(old_fd, new_fd)`

```
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);
}
```

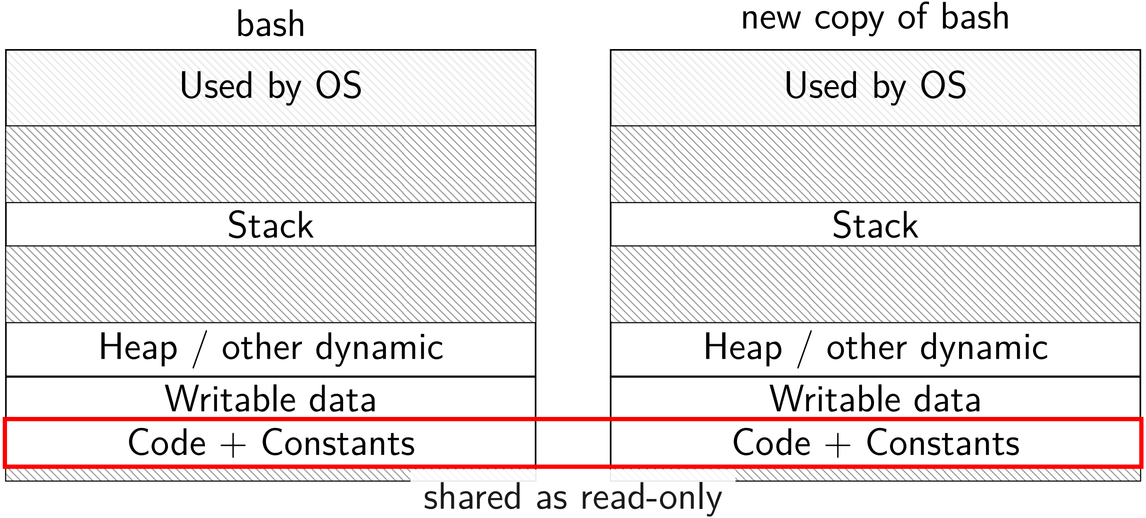
Which outputs are possible?

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?

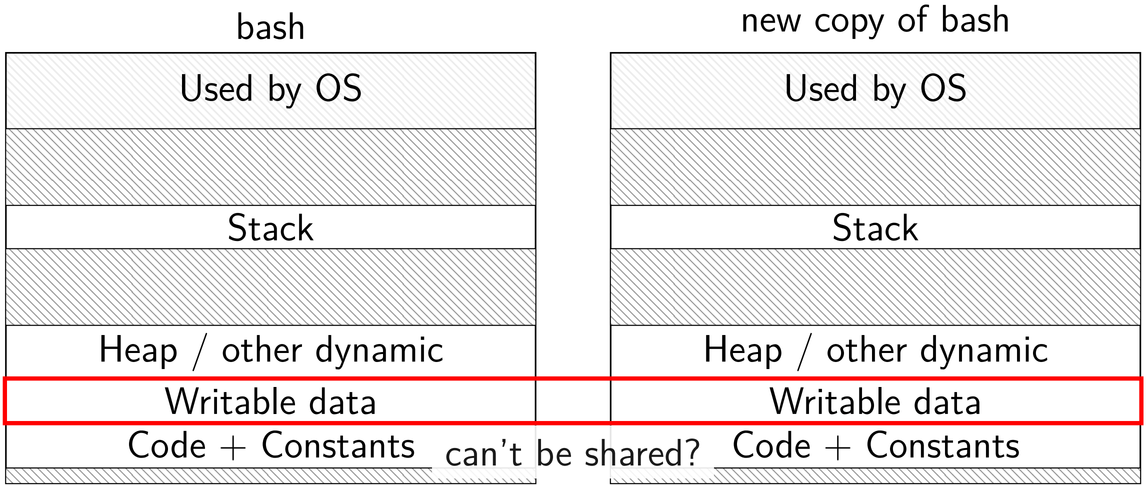
do we really need a complete copy?



do we really need a complete copy?



do we really need a complete copy?



trick for extra sharing

sharing writeable data is fine — until either process modifies it

- example: default value of global variables
- might typically not change
- (or OS might have preloaded executable's data anyways)

can we detect modifications?

trick for extra sharing

sharing writeable data is fine — until either process modifies it

- example: default value of global variables
- might typically not change
- (or OS might have preloaded executable's data anyways)

can we detect modifications?

trick: tell CPU (via page table) shared part is read-only

processor will trigger a fault when it's written

copy-on-write and page tables

VPN	valid?	write?	physical page
…	…	…	…
0x00601	1	1	0x12345
0x00602	1	1	0x12347
0x00603	1	1	0x12340
0x00604	1	1	0x200DF
0x00605	1	1	0x200AF
…	…	…	…

copy-on-write and page tables

VPN	valid?	write?	physical page
…	…	…	…
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	0	0x200AF
…	…	…	…

VPN	valid?	write?	physical page
…	…	…	…
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	0	0x200AF
…	…	…	…

copy operation actually duplicates page table
both processes share all physical pages
but marks pages in both copies as read-only

copy-on-write and page tables

VPN	valid?	write?	physical page
…	…	…	…
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	0	0x200AF
…	…	…	…

VPN	valid?	write?	physical page
…	…	…	…
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	0	0x200AF
…	…	…	…

when either process tries to write read-only page
triggers a fault — OS actually copies the page

copy-on-write and page tables

VPN	valid?	write?	physical page
…	…	…	…
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	0	0x200AF
…	…	…	…

VPN	valid?	write?	physical page
…	…	…	…
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	1	0x300FD
…	…	…	…

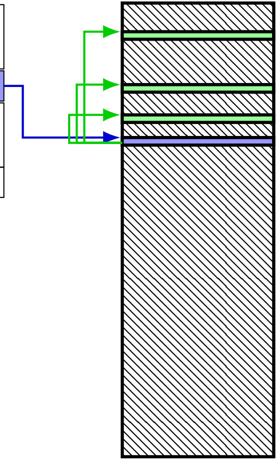
after allocating a copy, OS reruns the write instruction

fork (w/ copy-on-write, if parent writes first)

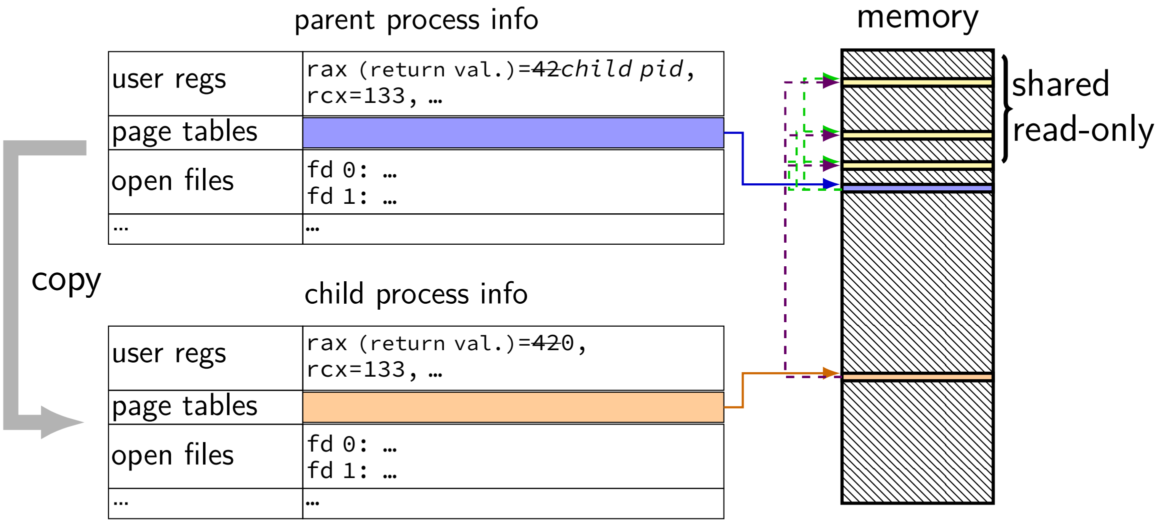
parent process info

user regs	`rax (return val.)=`~~42~~*child pid*, `rcx=133`, …
page tables	
open files	`fd 0:` … `fd 1:` …
…	…

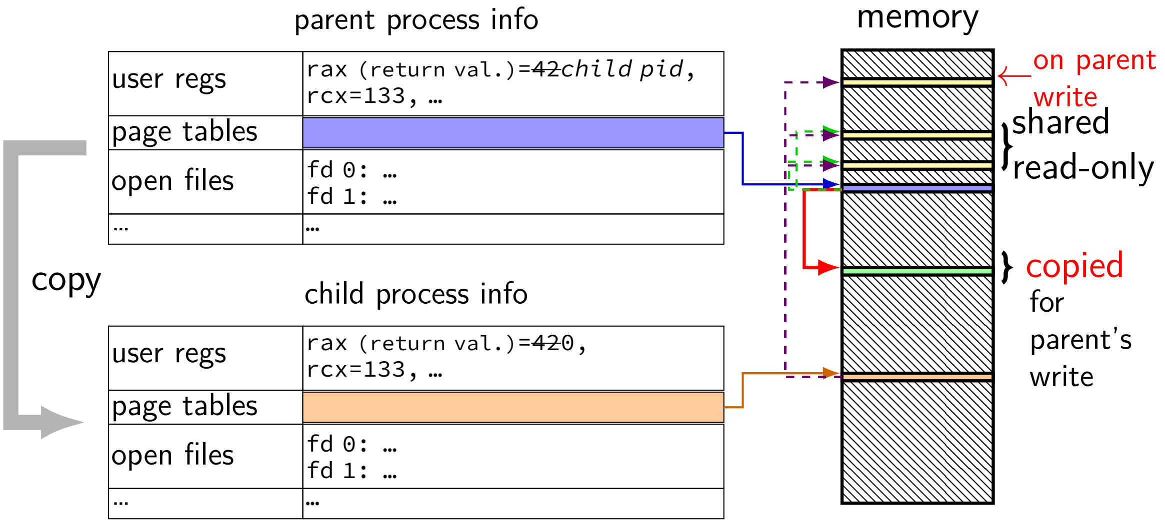
memory



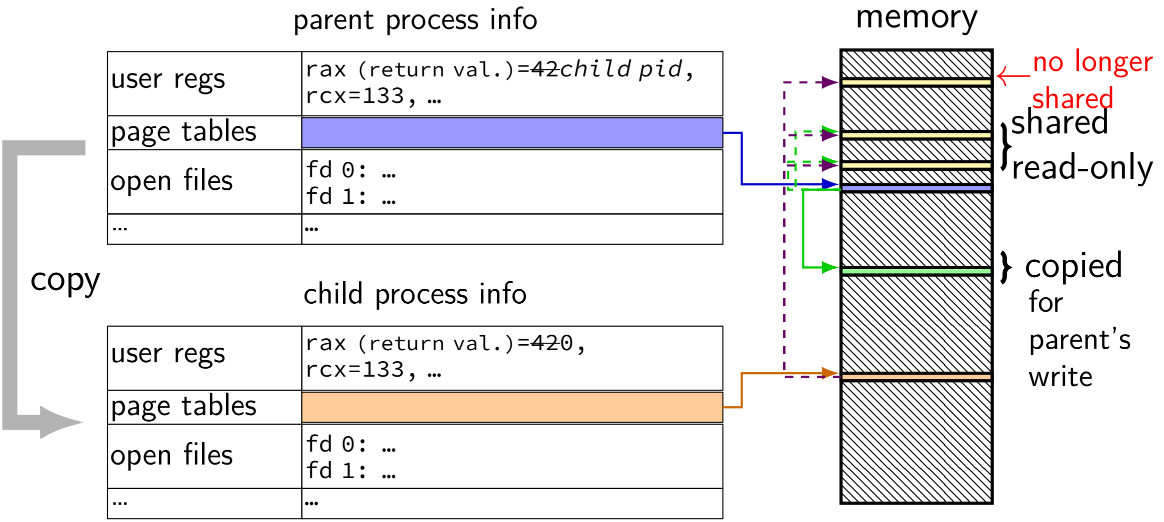
fork (w/ copy-on-write, if parent writes first)



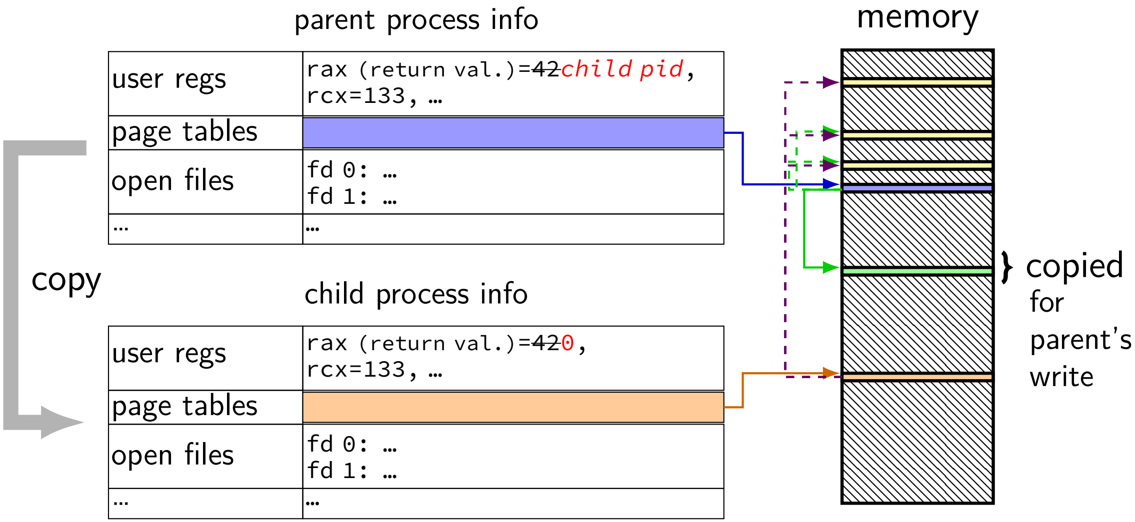
fork (w/ copy-on-write, if parent writes first)



fork (w/ copy-on-write, if parent writes first)



fork (w/ copy-on-write, if parent writes first)



fork and process info (w/o copy-on-write)