Changelog

Changes made in this version not seen in first lecture:

6 September: fix stray @s on 'implementing file descriptors in xv6 slide'
6 September: typical pattern with redirection: hilite parts of code more sensibly
6 September: exec preserves open files: add slide
6 September: dup2 example: clarify comment, note overall purpose at top
6 September: read'ing one byte at a time: missing)
6 September: layering: annotate to indicate read/write are system calls, kernel buffers in layers, user buffers in layers

Unix API 2: files

last time

POSIX — standardized Unix

process control blocks

fork, exec, waitpid

post-quizzes

starting this week, post-quizzes

link off course website

same software as CS 3330

- box around question turns green: answer recorded

no time limits, due before Tuesday's class

released Friday morning

- or possibly earlier (e.g. Thursday evening)

shell

allow user (= person at keyborad) to run applications

user's wrapper around process-management functions

upcoming homework — make a simple shell

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?

some POSIX command-line features

searching for programs (not in assignment)

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

running in background (not in assignment)

```
    ./someprogram &
```

redirection:

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

pipelines:

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

some POSIX command-line features

searching for programs (not in assignment)

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

running in background (not in assignment)

```
    ./someprogram &
```

redirection:

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

pipelines:

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

searching for programs

POSIX convention: PATH environment variable

- example: `/home/cr4bd/bin:/usr/bin:/bin`
- checked in order

one way to implement: [pseudocode]

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

some POSIX command-line features

searching for programs (not in assignment)

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

running in background (not in assignment)

```
./someprogram &
```

redirection:

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

pipelines:

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

running in background

```
$ ./long_computation >tmp.txt &
[1] 4049
$ ...
[1]+    Done        ./long_computation > tmp.txt
$ cat tmp.txt
the result is ...
```

& — run a program in “background”

initially output PID (above: 4049)

print out after terminated

- one way: use `waitpid` with option saying “don’t wait”

some POSIX command-line features

searching for programs (not in assignment)

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

running in background (not in assignment)

```
    ./someprogram &
```

redirection:

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

pipelines:

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

shell redirection

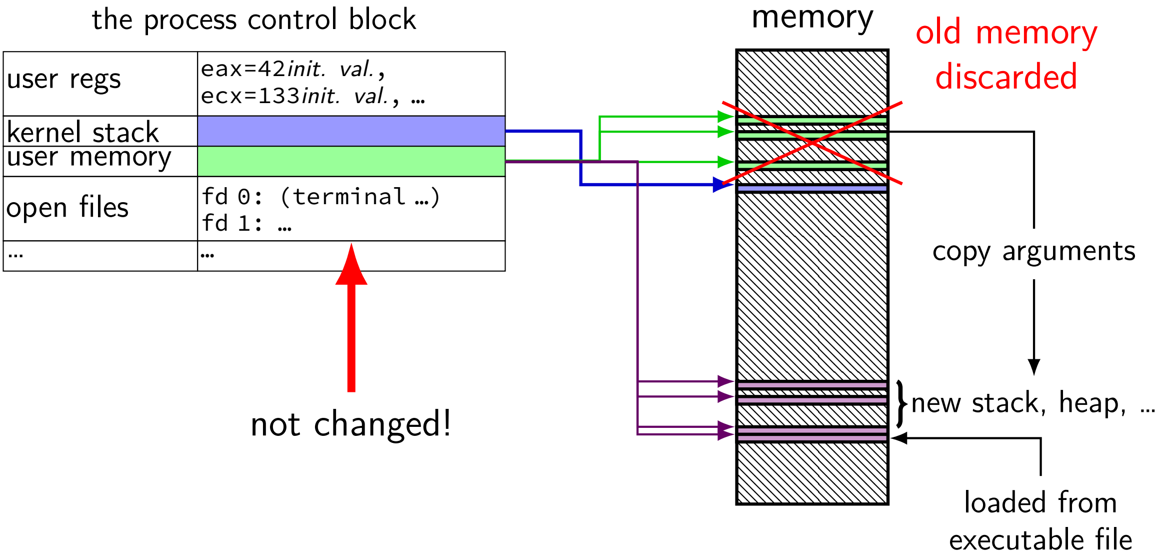
`./my_program ... <input.txt`:

- 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



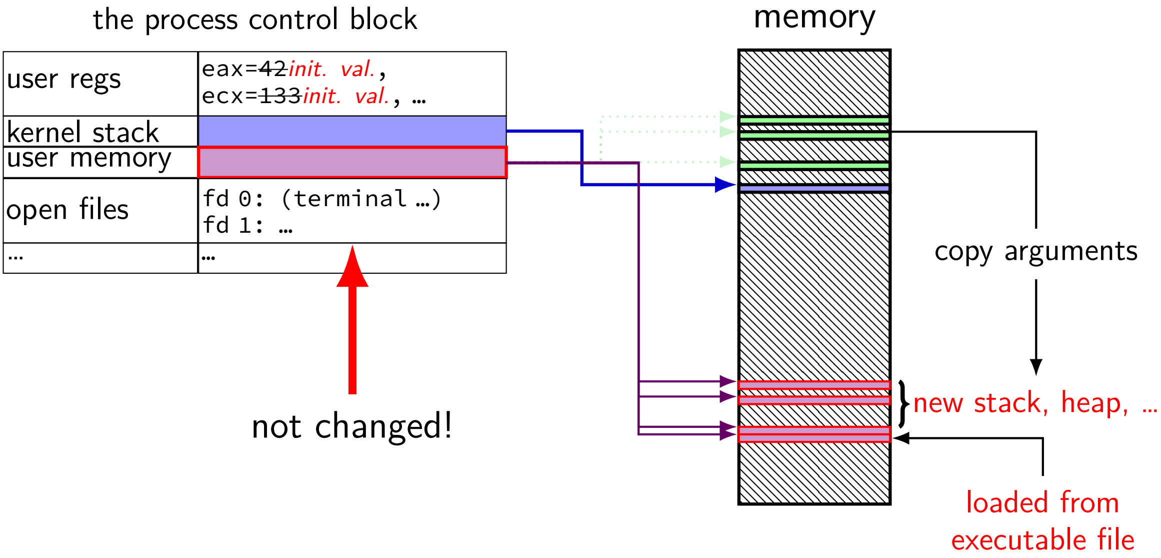
the process control block

user regs	eax=42*init. val.*,
	ecx=133*init. val.*, …

kernel stack	
user memory	
open files	fd 0: (terminal …) fd 1: …
…	…

not changed!

exec preserves open files



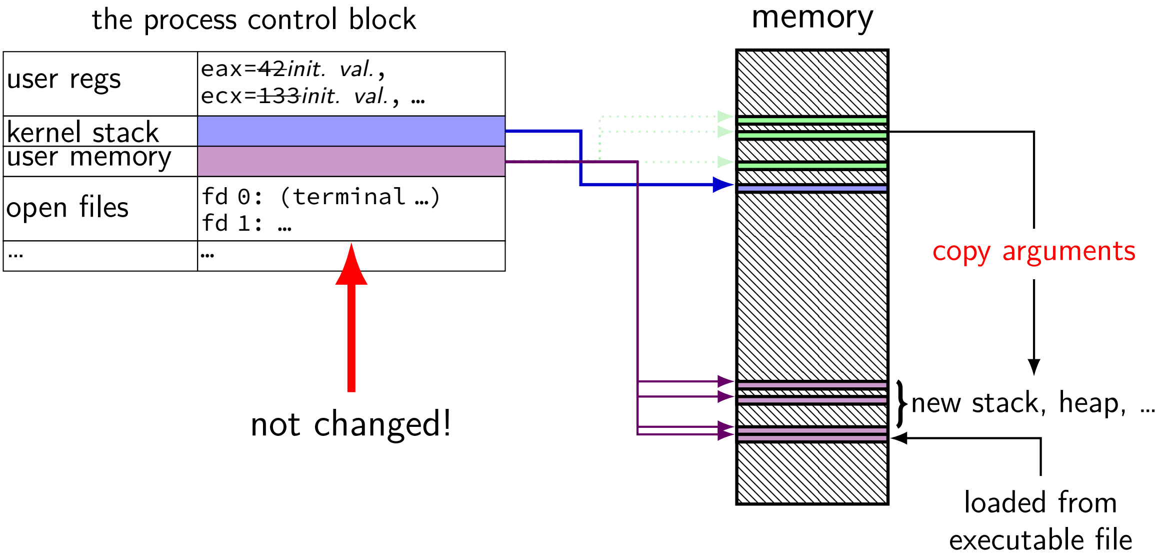
the process control block

user regs	eax=~~42~~*init. val.*,
	ecx=~~133~~*init. val.*, …

kernel stack	
user memory	
open files	fd 0: (terminal …) fd 1: …
…	…

not changed!

exec preserves open files



the process control block

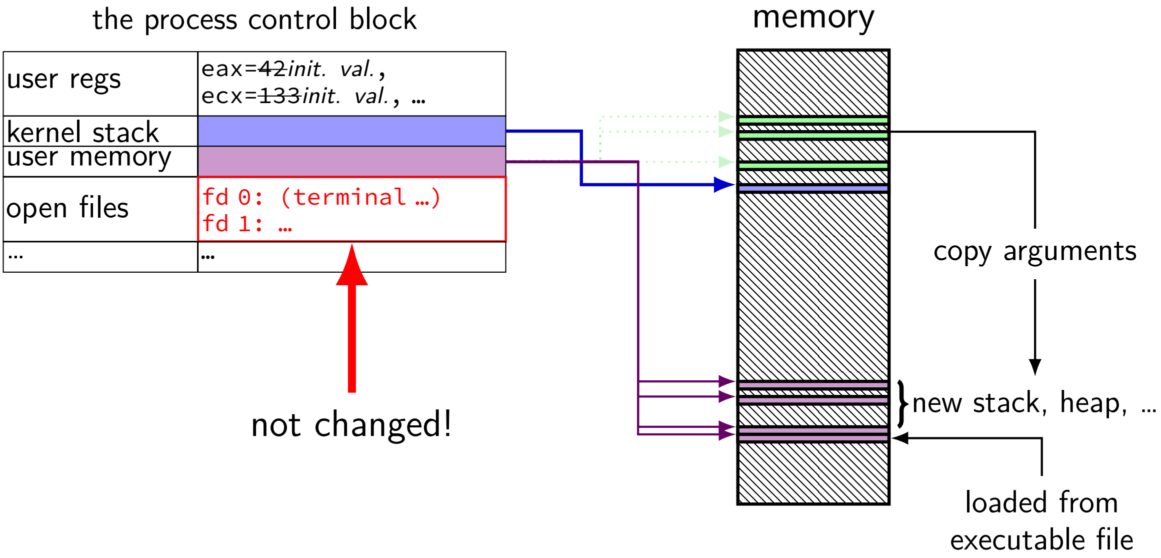
user regs	eax=~~42~~*init. val.*,
	ecx=~~133~~*init. val.*, …

kernel stack	
user memory	

open files	`fd 0: (terminal …)`
	`fd 1: …`

…	…

exec preserves open files



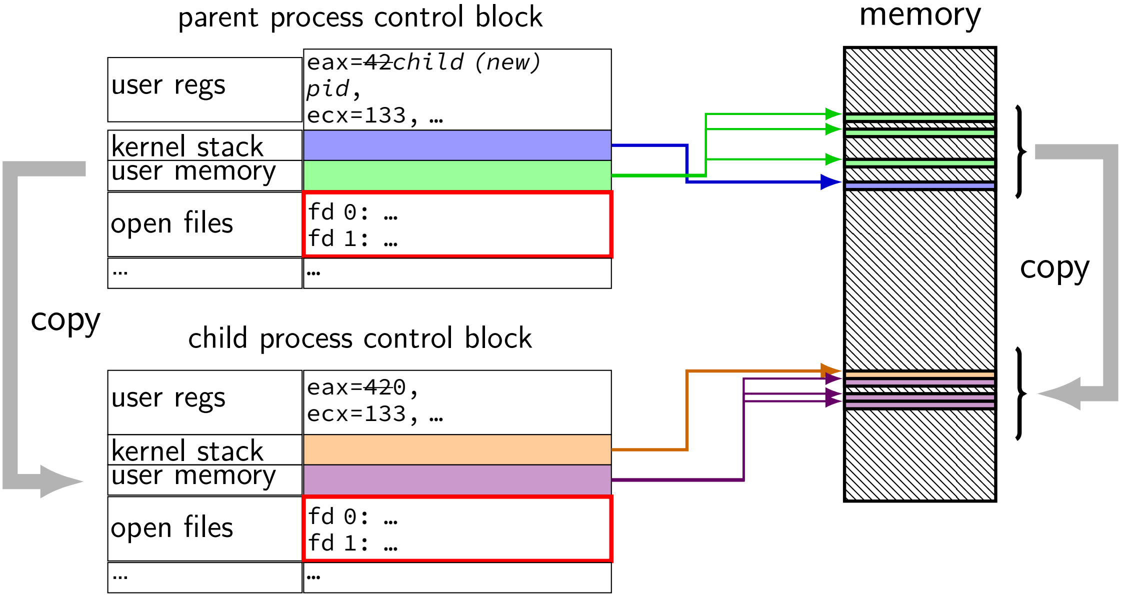
the process control block

user regs	eax=~~42~~*init. val.*,
	ecx=~~133~~*init. val.*, …

kernel stack	
user memory	
open files	fd 0: (terminal …) fd 1: …
…	…

not changed!

fork copies open files



parent process control block

user regs	eax=~~42~~*child (new) pid*, ecx=133, …
kernel stack	
user memory	
open files	fd 0: … fd 1: …
…	…

memory

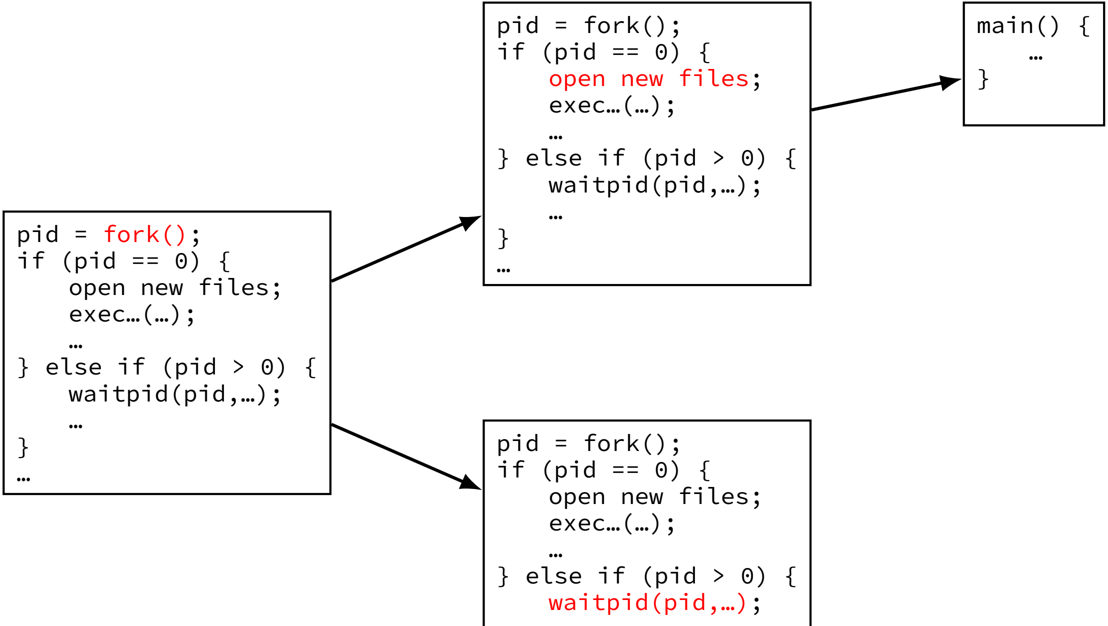
copy

copy

child process control block

user regs	eax=~~42~~0, ecx=133, …
kernel stack	
user memory	
open files	fd 0: … fd 1: …
…	…

typical pattern with redirection



```
pid = fork();
if (pid == 0) {
    open new files;
    exec…(…);
    …
} else if (pid > 0) {
    waitpid(pid,…);
    …
}
…
```

```
pid = fork();
if (pid == 0) {
    open new files;
    exec…(…);
    …
} else if (pid > 0) {
    waitpid(pid,…);
    …
}
…
```

```
main() {
    …
}
```

```
pid = fork();
if (pid == 0) {
    open new files;
    exec…(…);
    …
} else if (pid > 0) {
    waitpid(pid,…);
```

redirecting with exec

std output, std error are files

- yes, your terminal is a file
- more on this later

after forking, open files to redirect

...and make them be standard output/error

missing pieces:

- how open files becomes default output/input

some POSIX command-line features

searching for programs (not in assignment)

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

running in background (not in assignment)

```
    ./someprogram &
```

redirection:

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

pipelines:

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

shell assignment

implement a simple shell that supports redirection and pipeline

...and prints the exit code of program in the pipeline

simplified parsing: space-seperated:

okay: `/bin/ls␣-1␣>␣tmp.txt`
not okay: `/bin/ls␣-l␣>tmp.txt`
okay: `/bin/ls␣-1␣|␣/bin/grep␣foo␣>␣tmp.txt`
not okay: `/bin/ls␣-1␣|/bin/grep␣foo␣>tmp.txt`

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()

the file interface

open before use

- setup, access control happens here

byte-oriented

- real device isn't? operating system needs to hide that

explicit close

the file interface

open before use

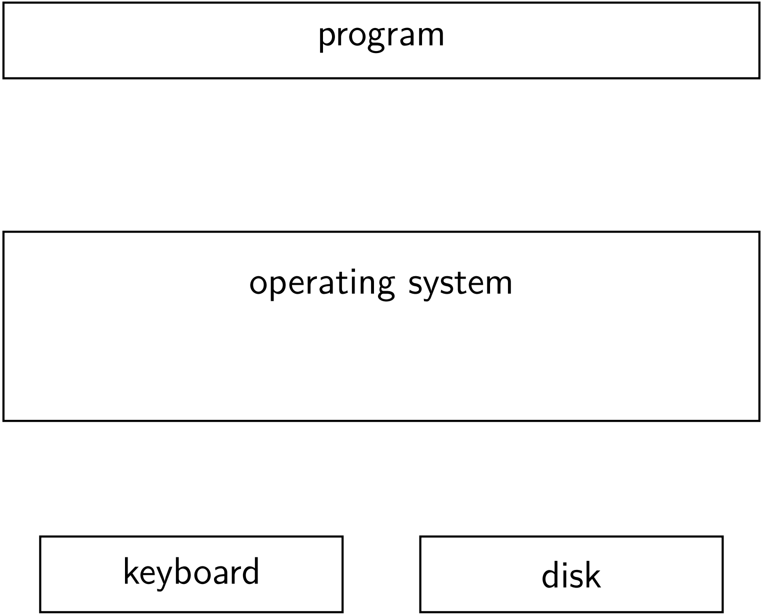
setup, access control happens here

byte-oriented

real device isn't? operating system needs to hide that

explicit close

kernel buffering (reads)



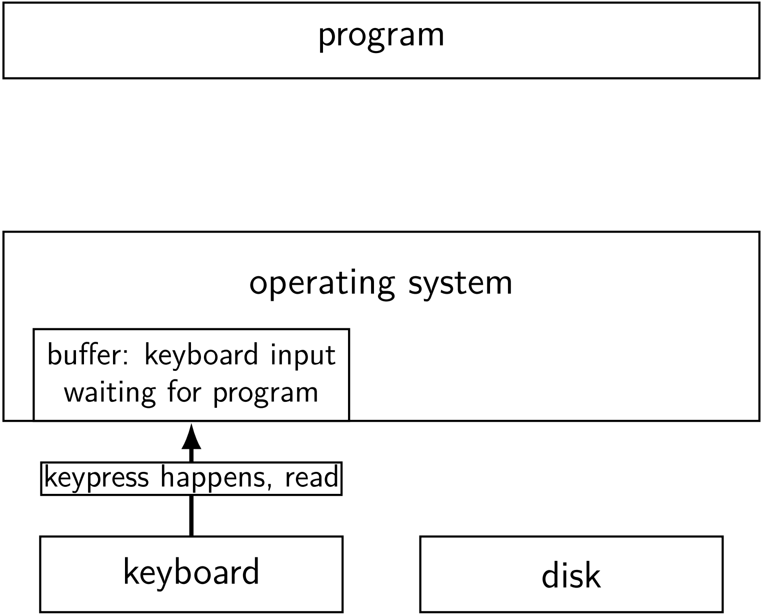
program

operating system

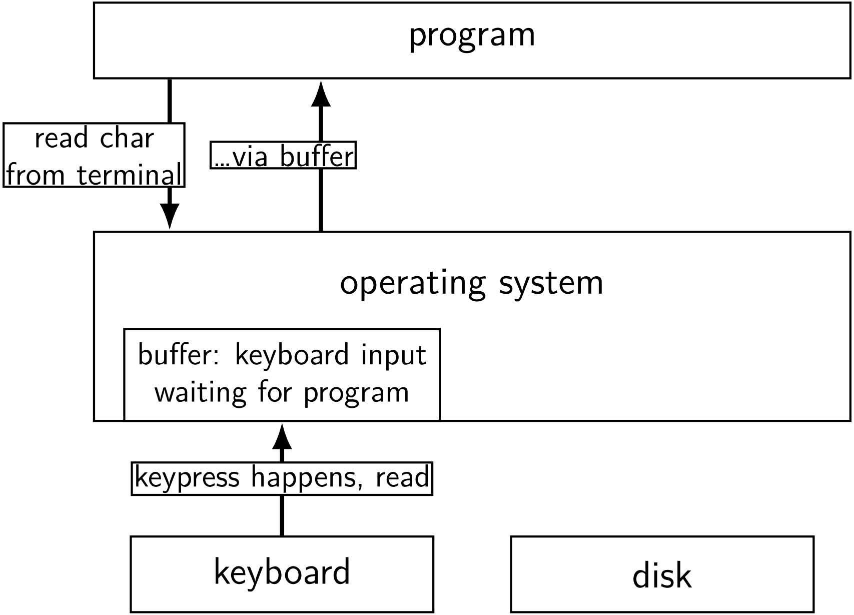
keyboard

disk

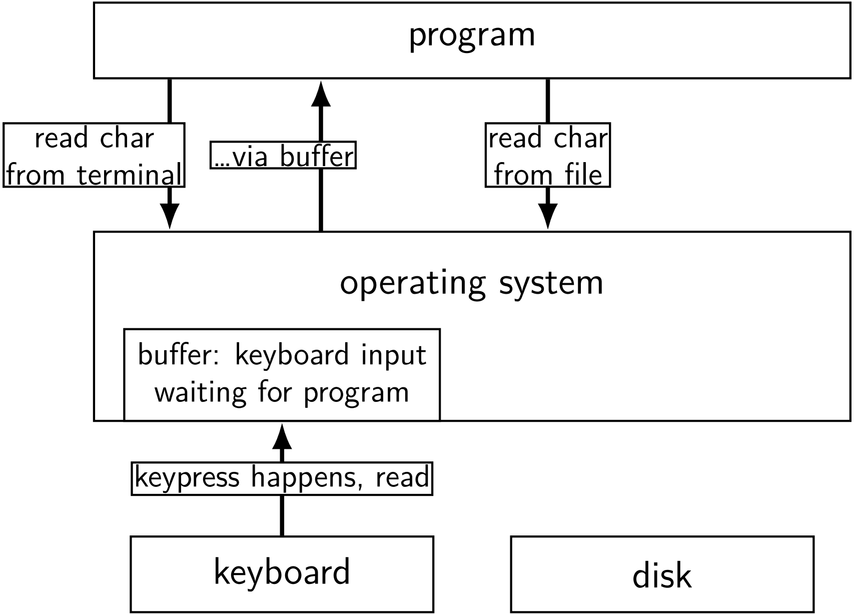
kernel buffering (reads)



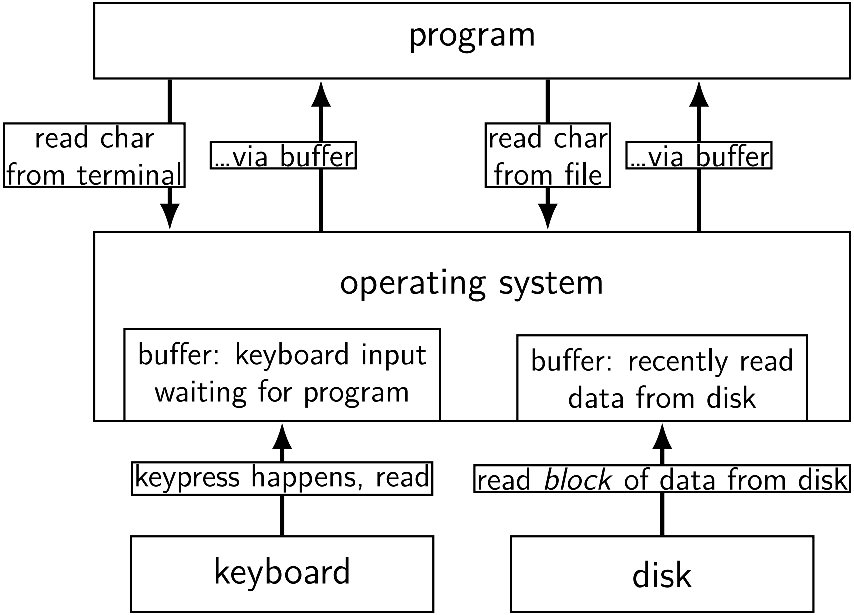
kernel buffering (reads)



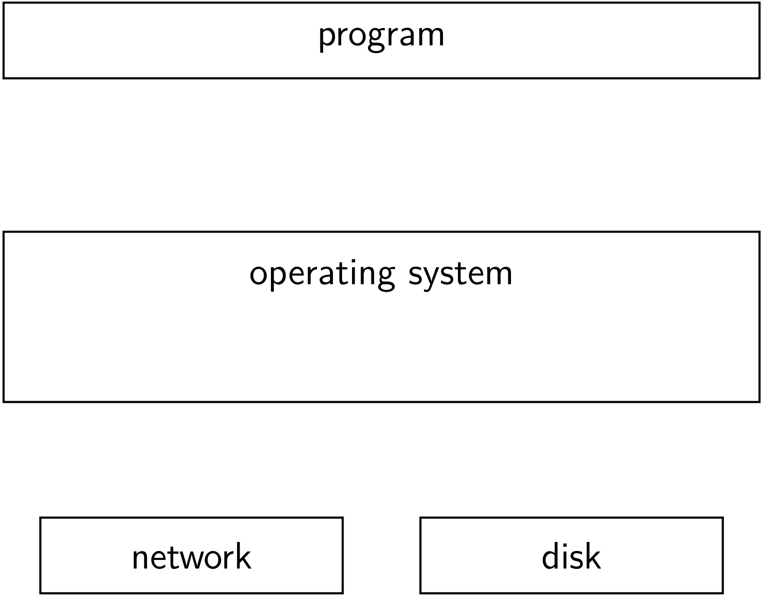
kernel buffering (reads)



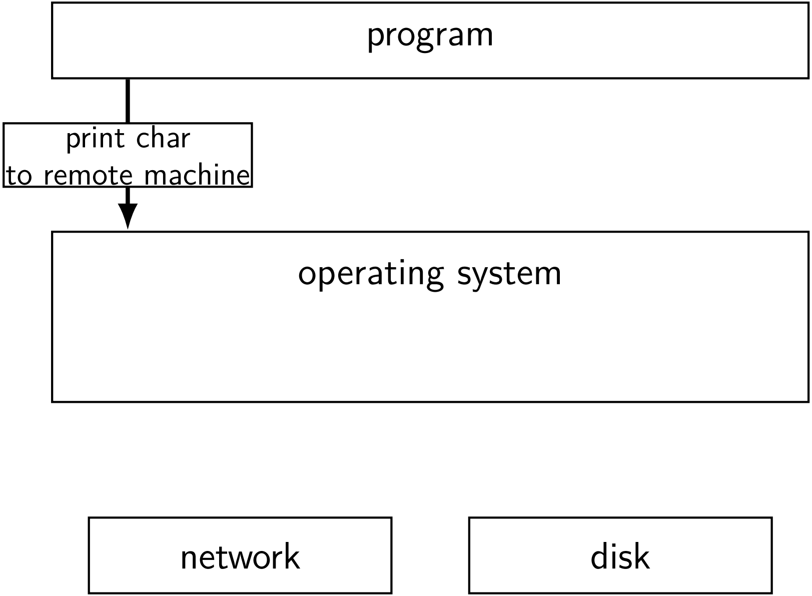
kernel buffering (reads)



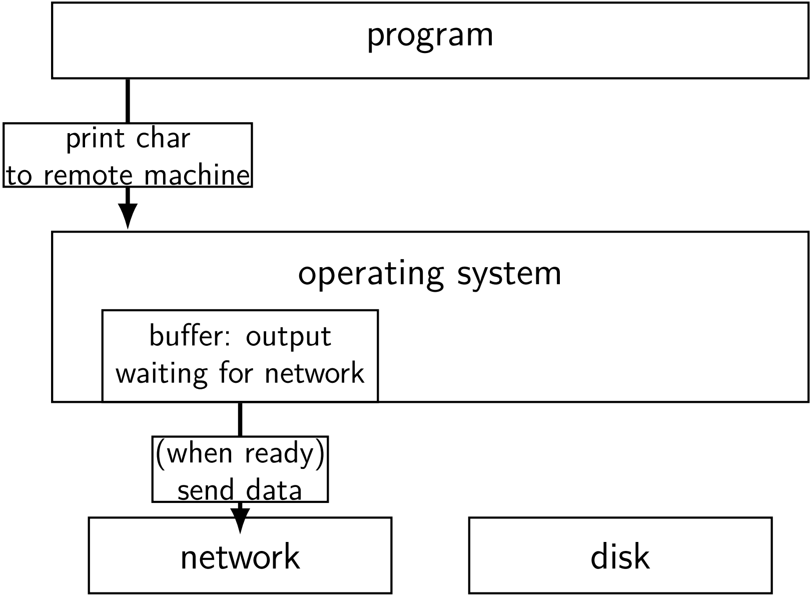
kernel buffering (writes)



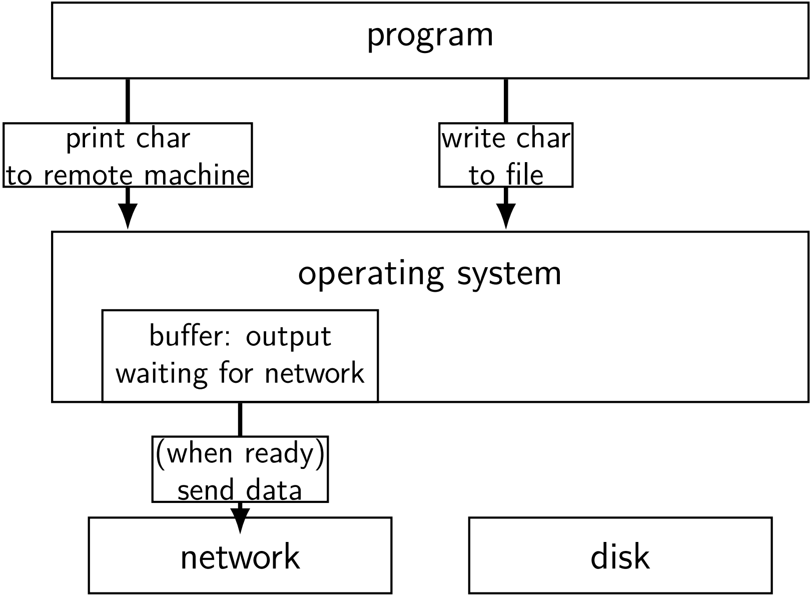
kernel buffering (writes)



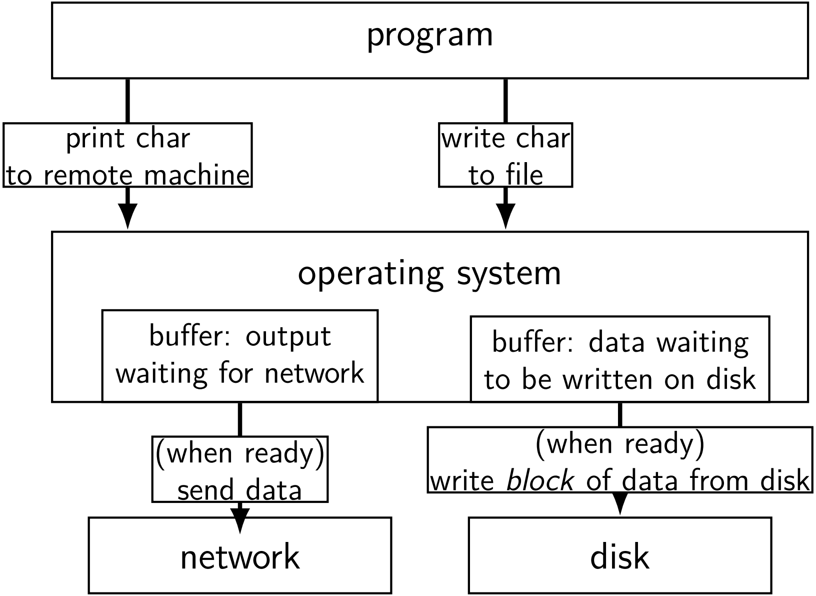
kernel buffering (writes)



kernel buffering (writes)



kernel buffering (writes)



read/write operations

read/write: move data into/out of buffer

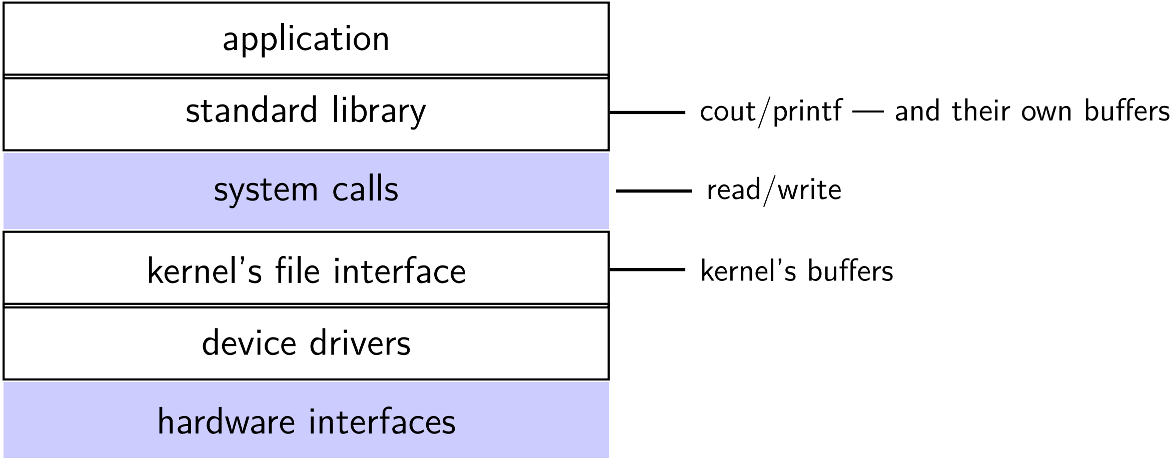
block (make process wait) if buffer is empty (read)/full (write)

- (default behavior, possibly changeable)

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

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

implementing file descriptors in xv6 (1)

```
struct proc {
  ...
  struct file *ofile[NOFILE];  // Open files
};
```

`ofile[0]` = file descriptor `0`

pointer — *can be shared between proceses*

  not part of deep copy fork does

null pointers — no file open with that number

implementing file descriptors in xv6 (2)

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

implementing file descriptors in xv6 (2)

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

FD_PIPE = to talk to other process
FD_INODE = other kind of file

alternate designs:

- class + subclass per type
- pointer to list of functions (Linux soln.)

implementing file descriptors in xv6 (2)

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe
  struct inode *ip;
  uint off;
};
```

number of pointers to this struct file

used to safely delete this struct

needs kept up-to-date (example: on `fork`)

implementing file descriptors in xv6 (2)

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

should read/write be allowed?
based on flags to open

implementing file descriptors in xv6 (2)

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

`off` = location in file
(not meaningful for all files)

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

open: flags

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

flags: bitwise or of:

- `O_RDWR`, `O_RDONLY`, or `O_WRONLY`
  - read/write, read-only, write-only
- `O_APPEND`
  - append to end of file
- `O_TRUNC`
  - truncate (set length to 0) file if it already exists
- `O_CREAT`
  - create a new file if one doesn't exist
  - (default: file must already exist)
- `O_EXCL`
  - fail if file already exists (be first to create it)

```
man 2 open
```

open: mode

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

mode: permissions of newly created file

- like numbers provided to `chmod` command
- filtered by a “umask”

simple advice: always use `0666`

- = readable/writeable by everyone, except where umask prohibits
- (typical umask: prohibit other/group writing)

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;
while ((amount_read = read(STDIN_FILENO, &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 */
}
```

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 a fixed amount

```c
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);
```

partial reads

on regular file: read reads what you request

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

partial reads

on regular file: read reads what you request

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

reading from network — what's been received

reading from keyboard — what's been typed

write example

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

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

- or if used another call to request “non-blocking”
- (interruption: via *signal*)

more typical: write waits until it completes

- until remaining part fits in buffer in kernel?

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)

returns 0 on success, -1 on error (e.g. ran out of disk space while trying to save file)

stdio and iostreams

what about `cout`, `printf`, etc.?

...implemented in terms of `read`, `write`, `open`, `close`

adds buffering in the process — faster

- read/write typically system calls
- running system call for approx. each character is slow!
- *in addition* to buffering that occurs in the kernel

more convenient

- formatted I/O, partial reads/writes handled by library, etc.

more portable

- stdio.h and iostreams defined by the C and C++ standards

mixing stdio/iostream and raw read/write

don't do it (unless you're very careful)

`cin`/`scanf` read some extra characters into a buffer?

- you call `read` — they disappear!

`cout`/`printf` has output waiting in a buffer?

- you call `write` — out-of-order output!

(if you need to: some stdio calls specify that they clear out buffers)

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

```
struct proc {
  ...
  struct file *ofile[NOFILE];  // Open files
};
```

redirect stdout: want: `ofile[1] = ofile[`*opened-fd*`];`

(plus increment reference count, so nothing is deleted early)

but can't access `ofile` from userspace

so syscall: `dup2(`*opened-fd*`, 1);`

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");
```

dup

`int dup(int oldfd)`
copy oldfd to a newly chosen file descriptor

almost same as `dup2(oldfd,` *`new-fd-number`*`)`

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 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?

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)

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

```
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)

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

```
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 and pipelines

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

```c
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);
}
/* wait for processes, etc. */
```

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

- 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