

Changelog

Changes made in this version not seen in first lecture:

24 Jan 2019: wait: return value for WNOHANG when process not done is 0, not -1

24 Jan 2019: reading a fixed amount: add check for amount_read = 0

POSIX API 2

last time

context switch (finish)

POSIX

fork

exec

waitpid

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`

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`

wait/waitpid

```
pid_t waitpid(pid_t pid, int *status,  
              int options)
```

wait for a child process (with `pid=pid`) to finish

sets `*status` to its “status information”

`pid=-1` → wait for any child process instead

options? see manual page (command `man waitpid`)

0 — no options

WNOHANG — return 0 rather than hanging if process not yet done

wait/waitpid

```
pid_t waitpid(pid_t pid, int *status,  
              int options)
```

wait for a child process (with `pid=pid`) to finish

sets `*status` to its “status information”

`pid=-1` → wait for any child process instead

options? see manual page (command `man waitpid`)

0 — no options

WNOHANG — return 0 rather than hanging if process not yet done

exit statuses

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


waitpid example

```
#include <sys/wait.h>
...
child_pid = fork();
if (child_pid > 0) {
    /* Parent process */
    int status;
    waitpid(child_pid, &status, 0);
} else if (child_pid == 0) {
    /* Child process */
    ...
}
```

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 (control-C causes signal %d)\n",
           WTERMSIG(status), SIGINT);
} 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 (control-C causes signal %d)\n",
           WTERMSIG(status), SIGINT);
} else {
    ...
}
```

“status code” encodes both return value and if exit was abnormal
W* macros to decode it

aside: signals

signals are a way of communicating between processes

they are also how abnormal termination happens

wait's status will tell you when and what signal killed a program

constants in `signal.h`

`SIGINT` — control-C

`SIGTERM` — kill command (by default)

`SIGSEGV` — segmentation fault

`SIGBUS` — bus error

`SIGABRT` — `abort()` library function

...

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 */
}
```

'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:

```
init(1)--ModemManager(919)--{ModemManager}(972)
|
|   |--{ModemManager}(1064)
|   |--NetworkManager(1160)--dhcpcd(1755)
|   |   |--dnsmasq(1985)
|   |   |--{NetworkManager}(1180)
|   |   |--{NetworkManager}(1194)
|   |   |--{NetworkManager}(1195)
|   |--accounts-daemon(1649)--{accounts-daemon}(1757)
|   |--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)
|   |--avahi-daemon(934)--avahi-daemon(944)
|   |--bluetoothd(924)
|   |--colord(1193)--{colord}(1329)
|
|   |--nongod(1336)--{nongod}(1556)
|   |   |--{nongod}(1557)
|   |   |--{nongod}(1983)
|   |   |--{nongod}(2031)
|   |   |--{nongod}(2047)
|   |   |--{nongod}(2048)
|   |   |--{nongod}(2049)
|   |   |--{nongod}(2050)
|   |   |--{nongod}(2051)
|   |   |--{nongod}(2052)
|   |--msh-server(19090)--bash(19091)---tmux(5442)
|   |--msh-server(21996)--bash(21997)
|   |--msh-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.tnmapd(875)
|   |--rpc.statd(954)
|   |--rpcbind(884)
|   |--rsrver(1561)--{rsrver}(1786)
|   |   |--{rsrver}(1787)
|   |--rsyslogd(1090)--{rsyslogd}(1092)
|   |   |--{rsyslogd}(1093)
|   |   |--{rsyslogd}(1094)
|   |--rtkit-daemon(2565)--{rtkit-daemon}(2566)
|   |   |--{rtkit-daemon}(2567)
|   |--sd_clicero(2852)--sd_clicero(2853)
|   |   |--{sd_clicero}(2854)
|   |   |--{sd_clicero}(2855)
|   |--sd_dunny(2849)--{sd_dunny}(2850)
|   |   |--{sd_dunny}(2851)
|   |--sd_espeak(2749)--{sd_espeak}(2845)
|   |   |--{sd_espeak}(2846)
|   |   |--{sd_espeak}(2847)
|   |   |--{sd_espeak}(2848)
|   |--sd_generic(2463)--{sd_generic}(2464)
```

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()` / `wait()`s for child?

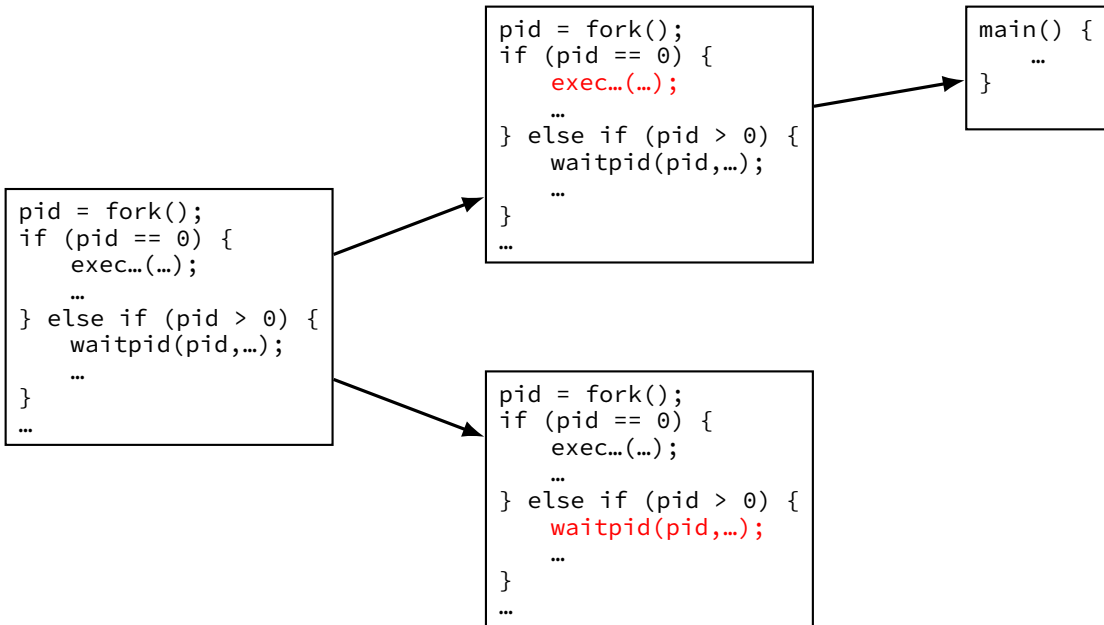
child process stays around as a "zombie"

can't reuse pid in case parent wants to use `waitpid()`

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

`waitpid` fails

typical pattern



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, ...);  
    ...  
}
```

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);  
}
```

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`

shell

allow user (= person at keyboard) 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

```
./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

```
./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

```
./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

```
./someprogram &
```

redirection:

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

pipelines:

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

shell redirection

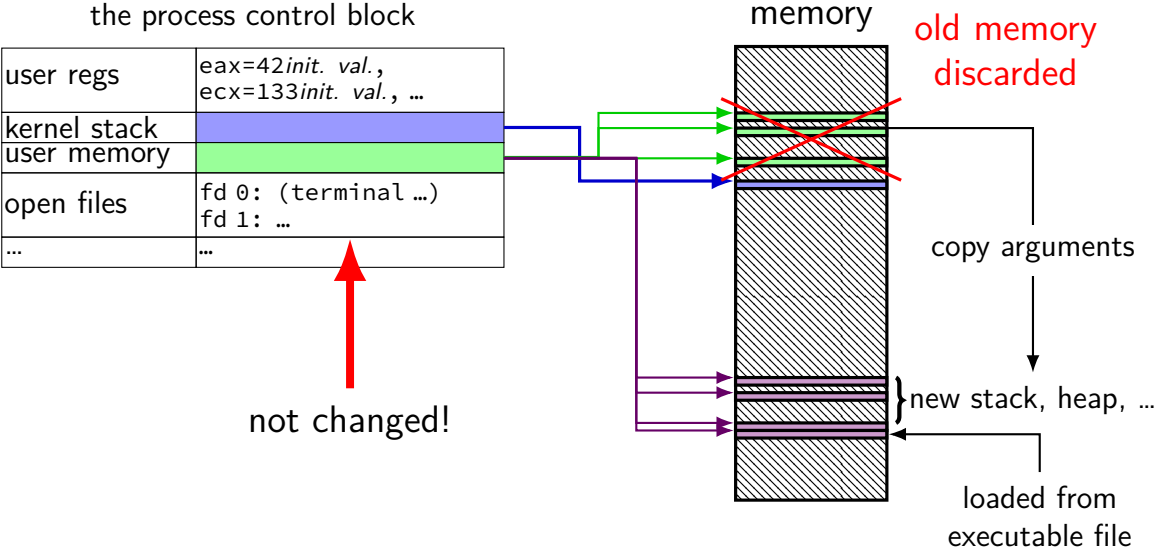
`./my_program ... <input.txt:`

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

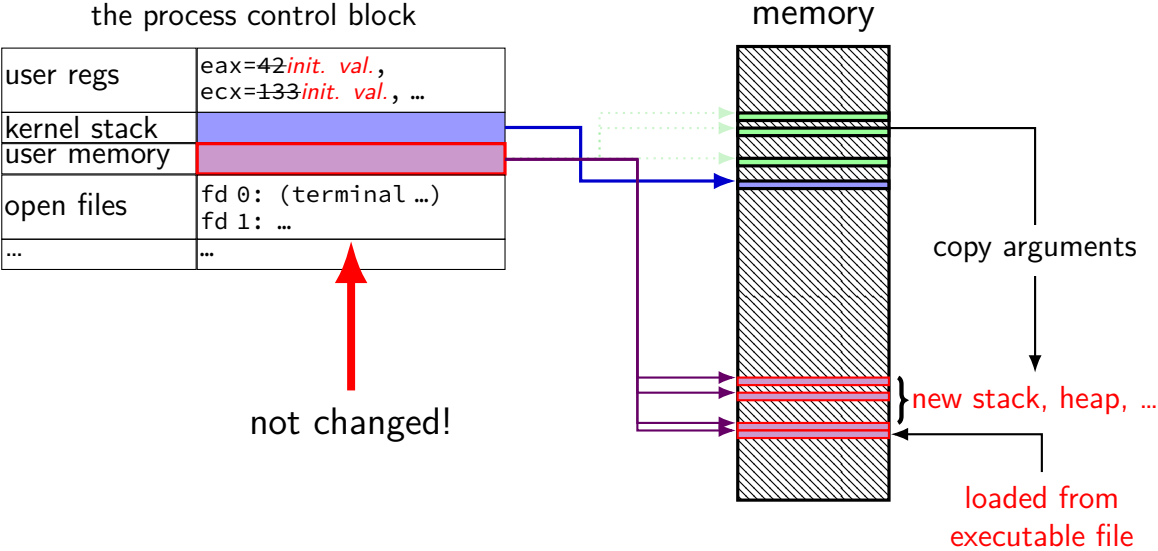
`echo foo >output.txt:`

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

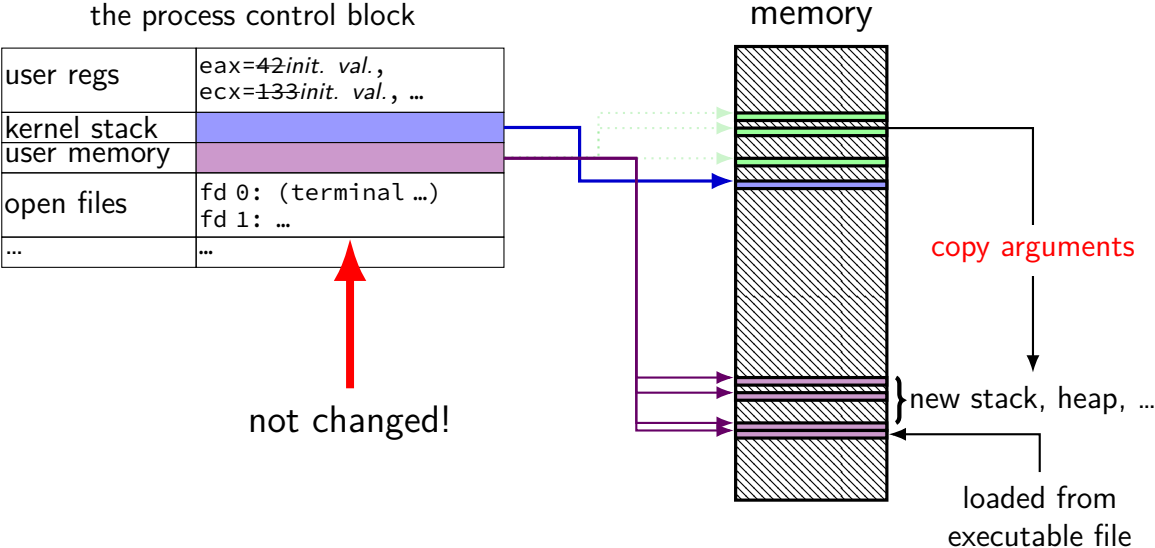
exec preserves open files



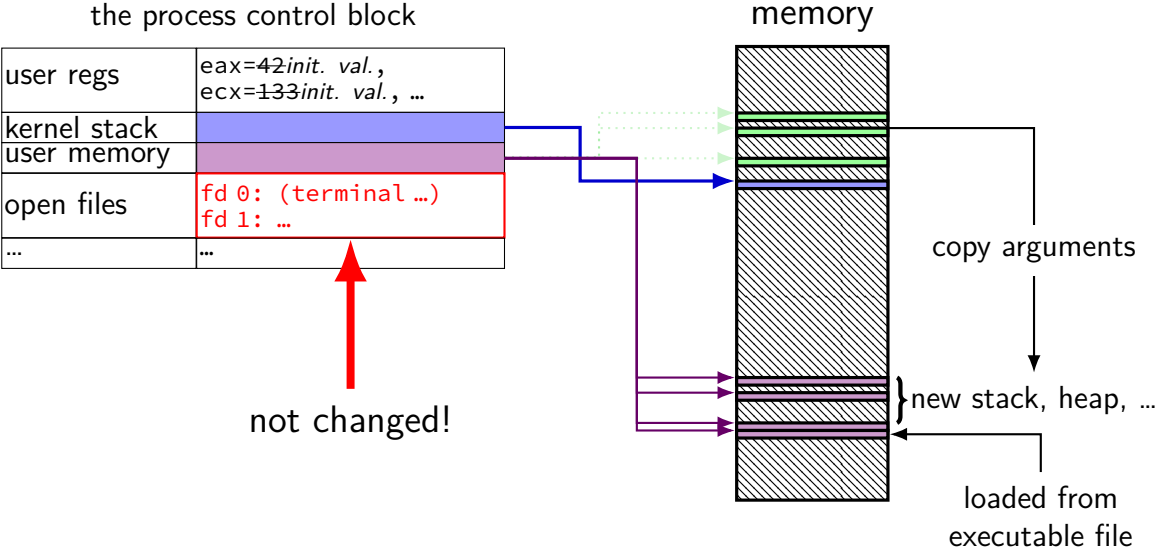
exec preserves open files



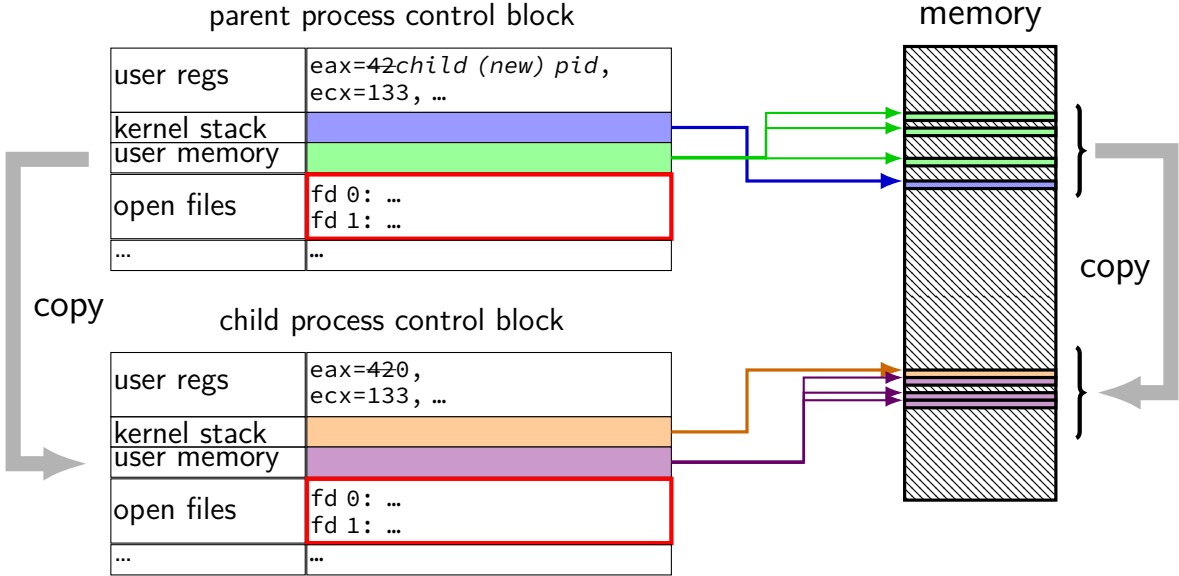
exec preserves open files



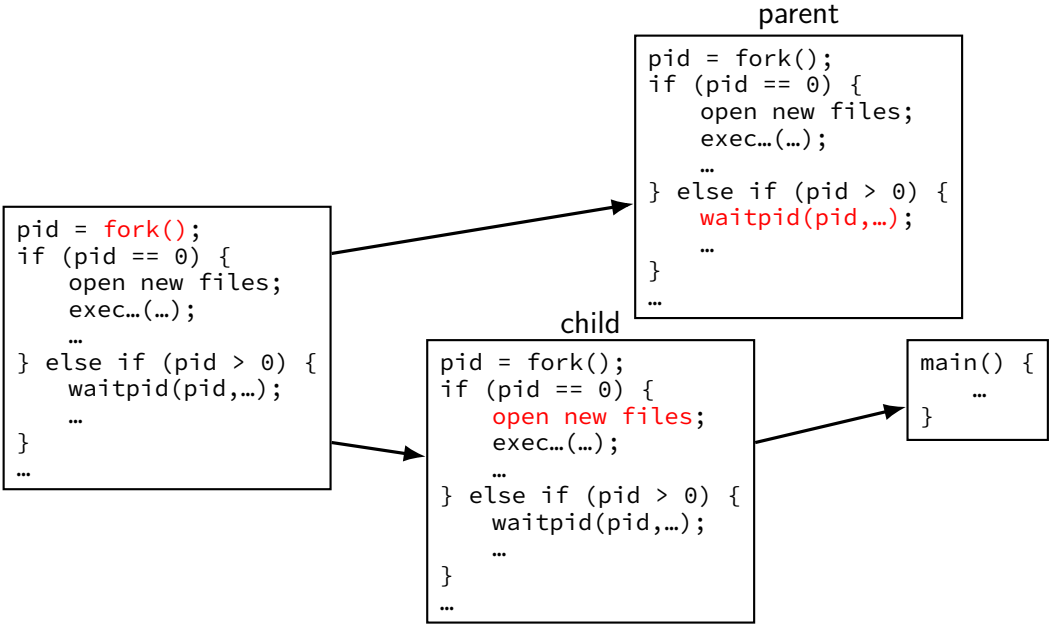
exec preserves open files



fork copies open files



typical pattern with redirection



redirecting with exec

standard output/error/input are files

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

yes, your terminal is a file

more on this later

after forking, open files to redirect

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

typically using `dup2()`

some POSIX command-line features

searching for programs (not in assignment)

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

running in background

```
./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-separated:

okay: `/bin/ls -l > tmp.txt`

not okay: `/bin/ls -l >tmp.txt`

okay: `/bin/ls -l | /bin/grep foo > tmp.txt`

not okay: `/bin/ls -l | /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)

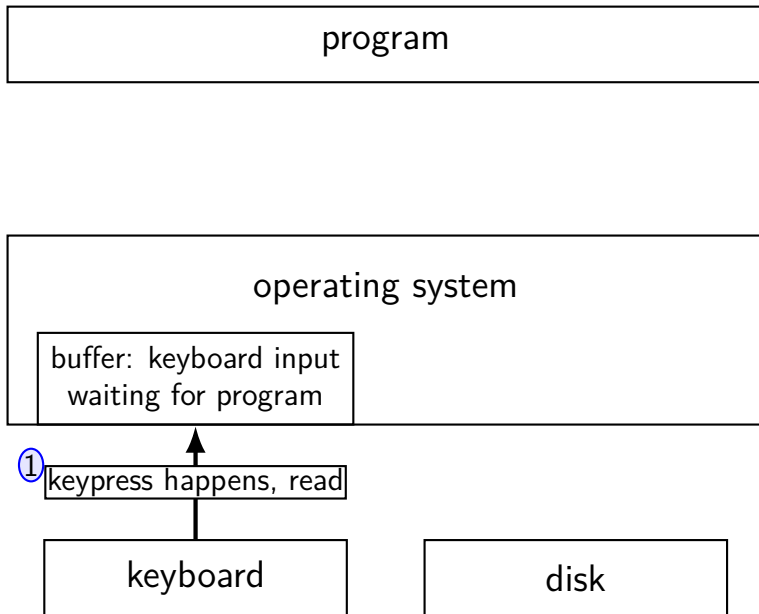
program

operating system

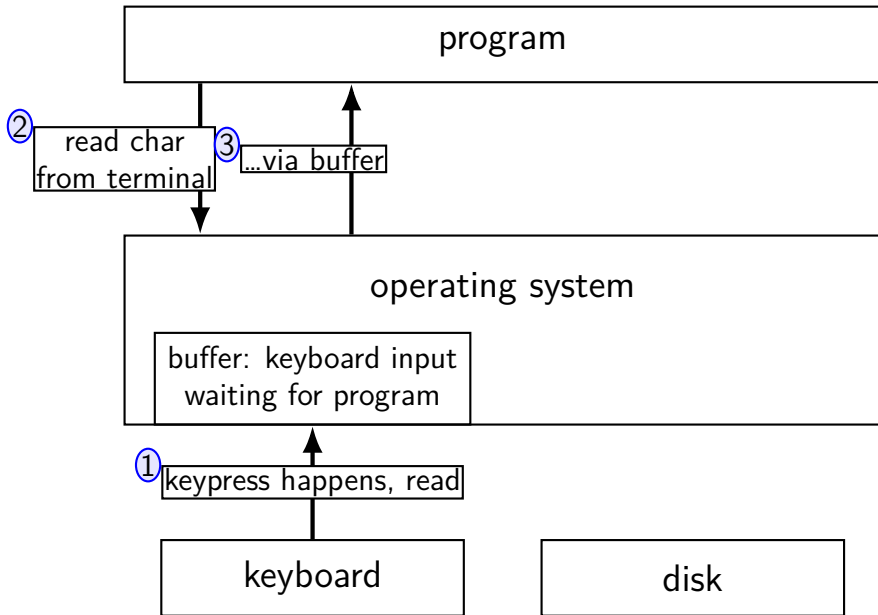
keyboard

disk

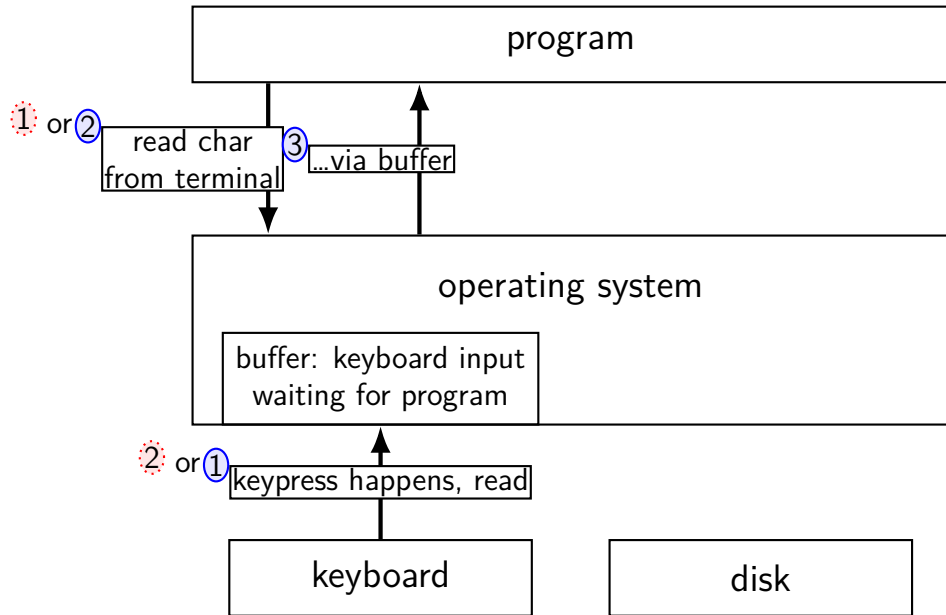
kernel buffering (reads)



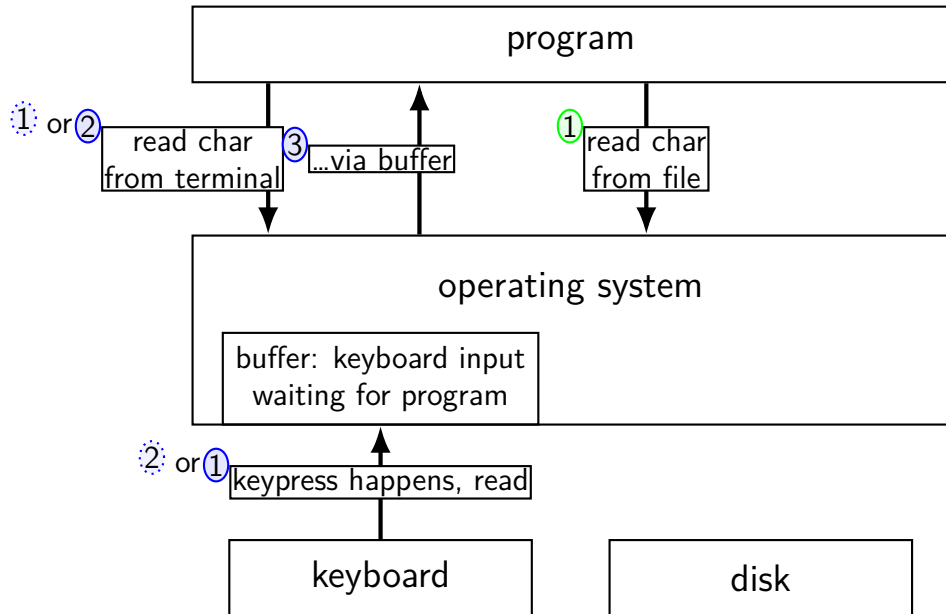
kernel buffering (reads)



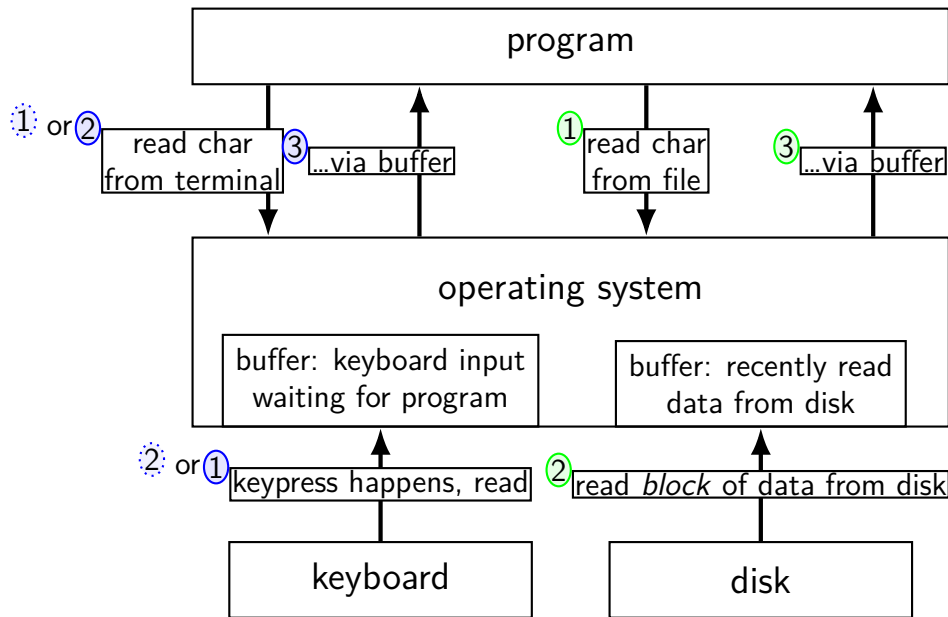
kernel buffering (reads)



kernel buffering (reads)



kernel buffering (reads)



kernel buffering (writes)

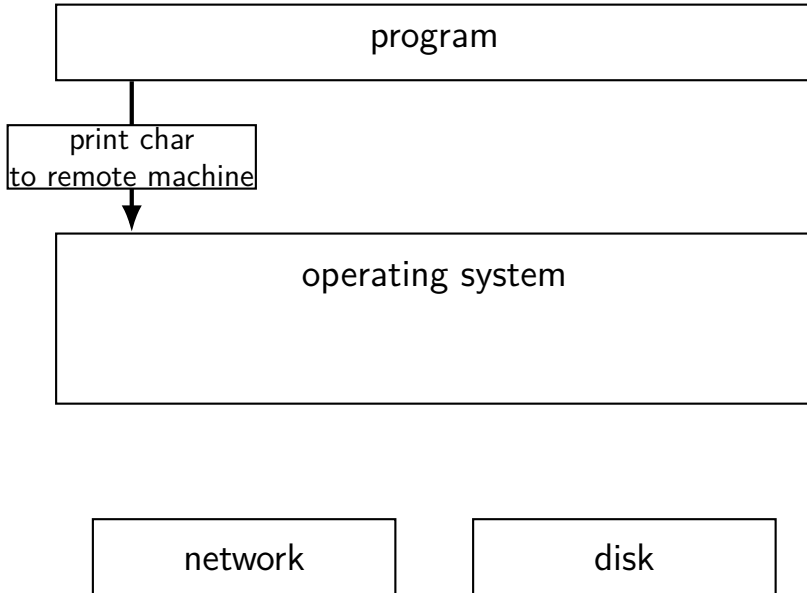
program

operating system

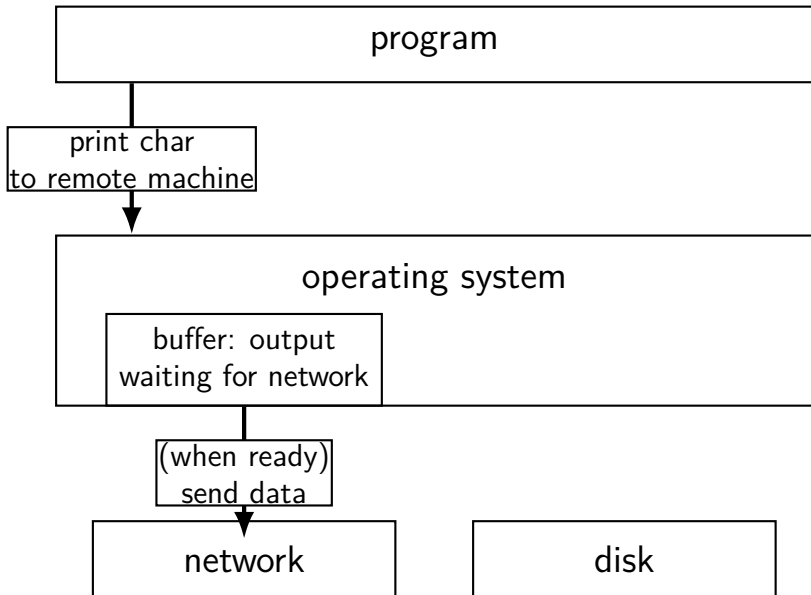
network

disk

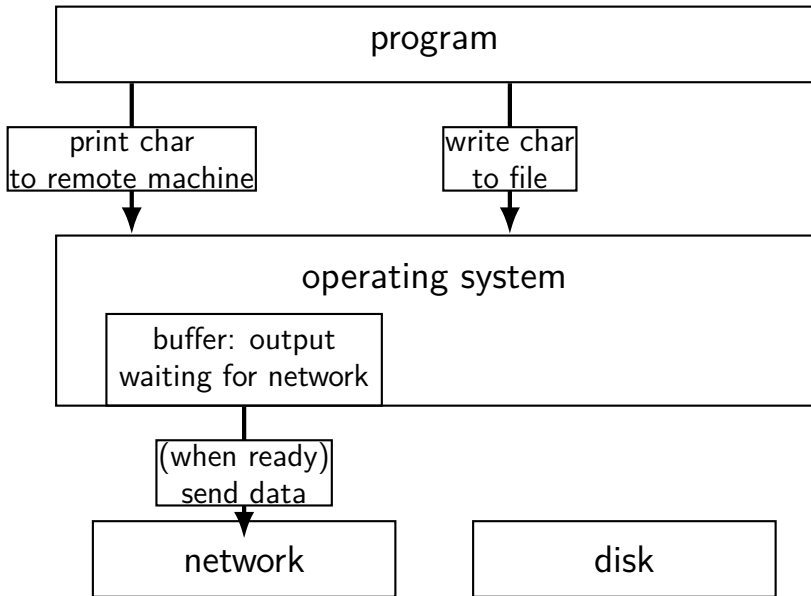
kernel buffering (writes)



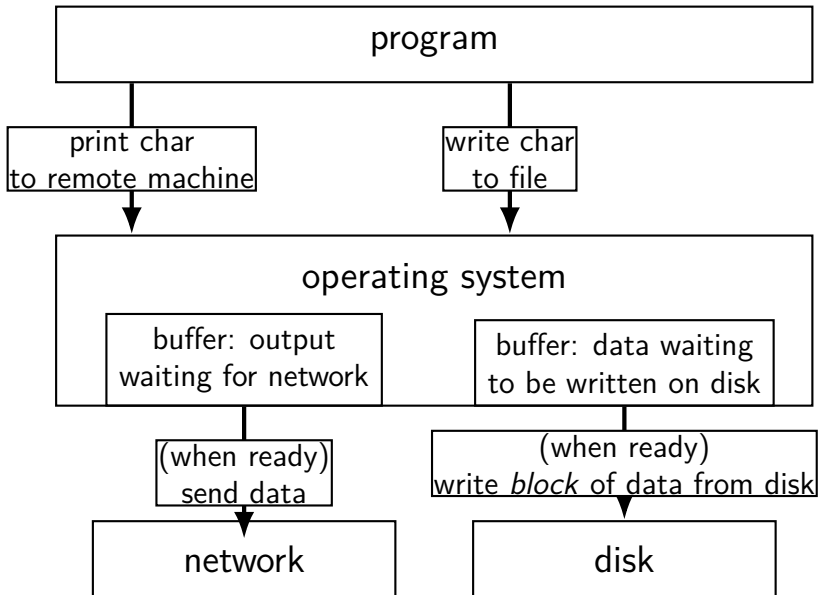
kernel buffering (writes)



kernel buffering (writes)



kernel buffering (writes)



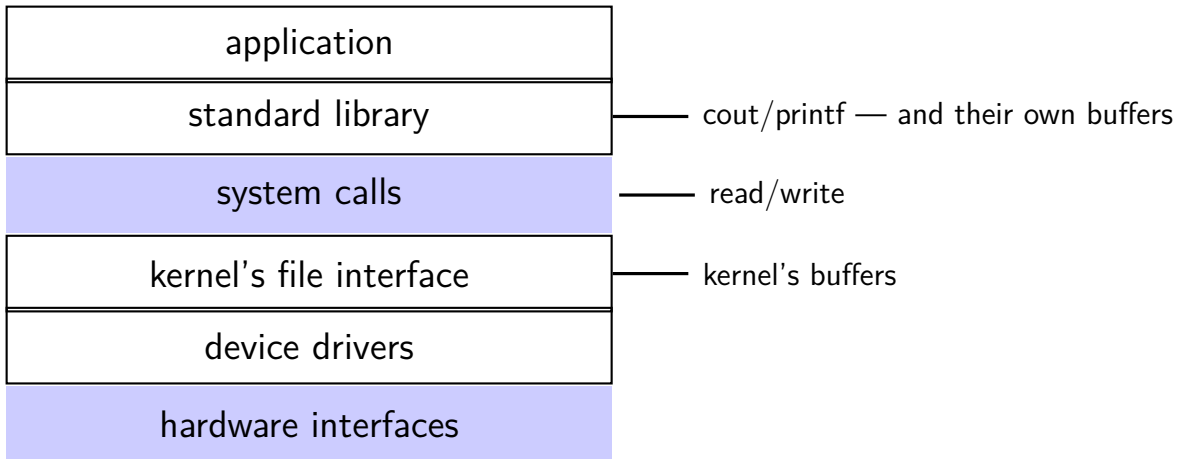
read/write operations

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



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/writable by everyone, except where umask prohibits
(typical umask: prohibit other/group writing)

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)

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*)

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

```
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: 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?

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)

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 -l | 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", "-l", 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. */
```