POSIX API 2

last time

POSIX – standard for Unix

fork: process creation via cloning

- new process called "child"; original called "parent"
- return child pid in parent; 0 in child

exec*: run different program in current process

waitpid: wait for child

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

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\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

aside: signals

signals are a way of communicating between processes

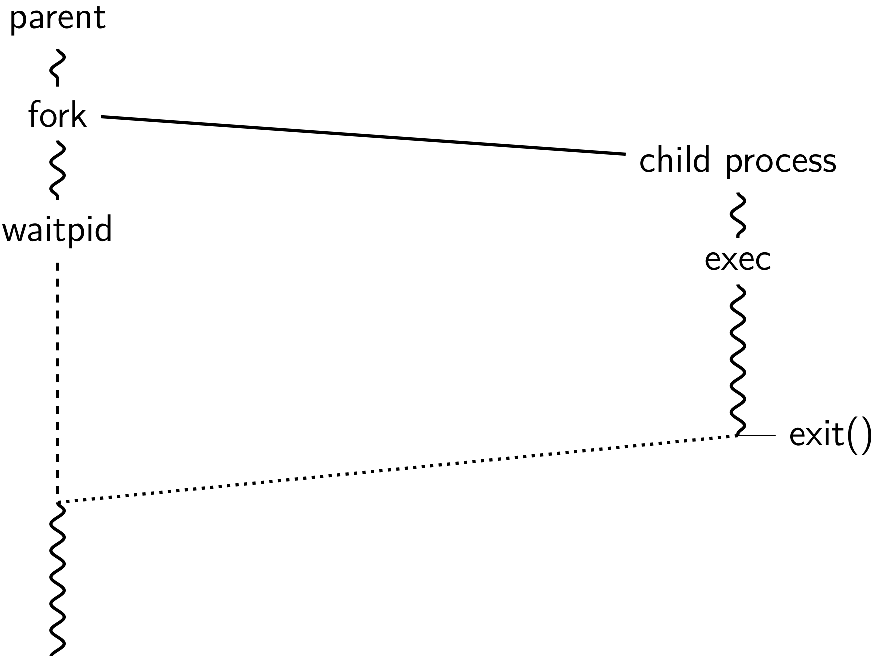
they are also how abnormal termination happens

- kernel communicating “something bad happened” → kills program by default

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
- …

typical pattern



parent

fork

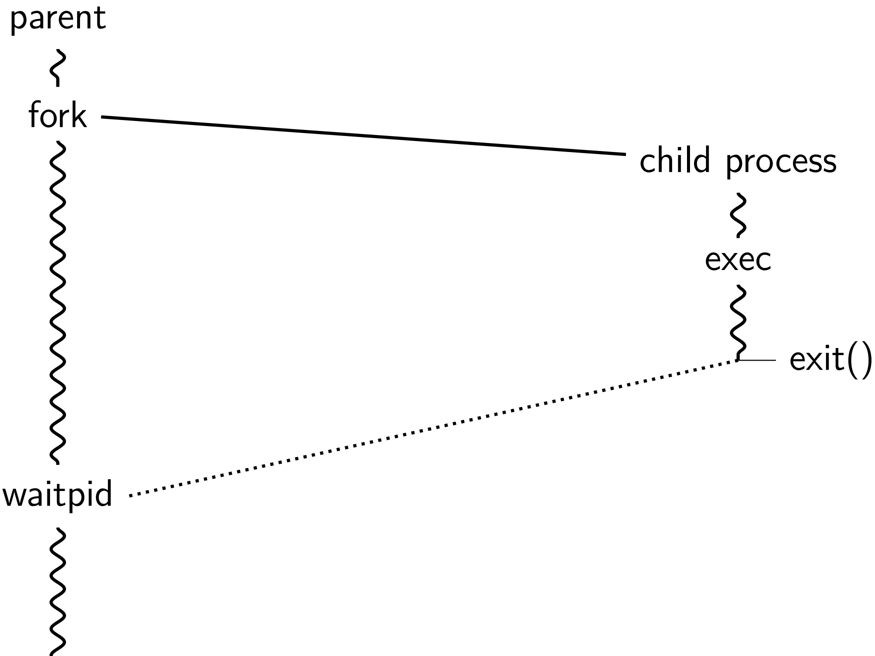
child process

exec

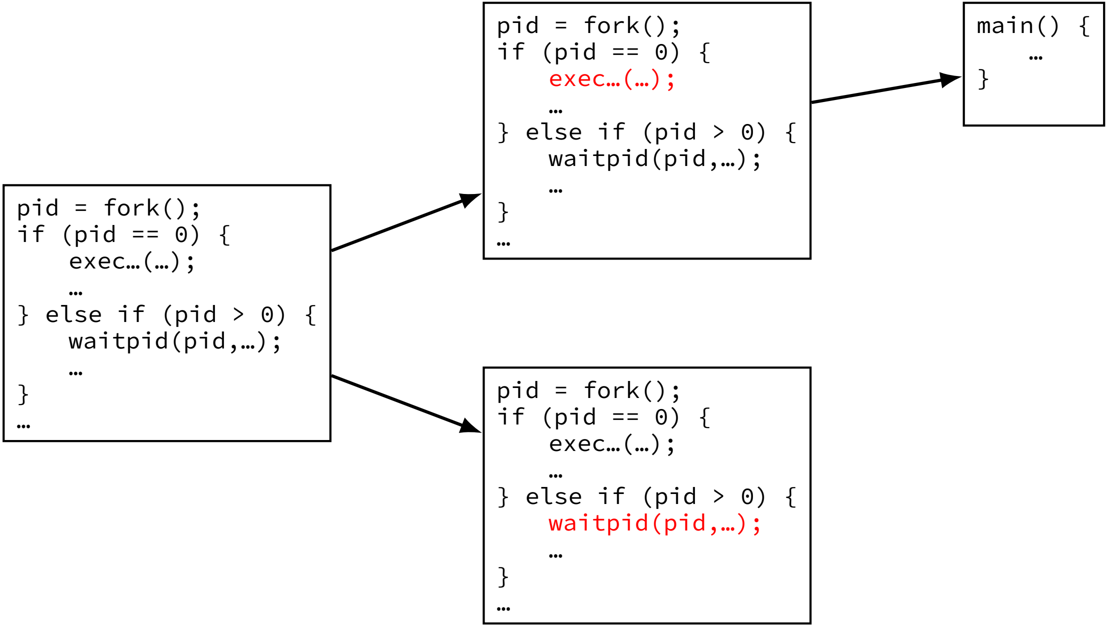
waitpid

exit()

typical pattern (alt)



typical pattern (detail)



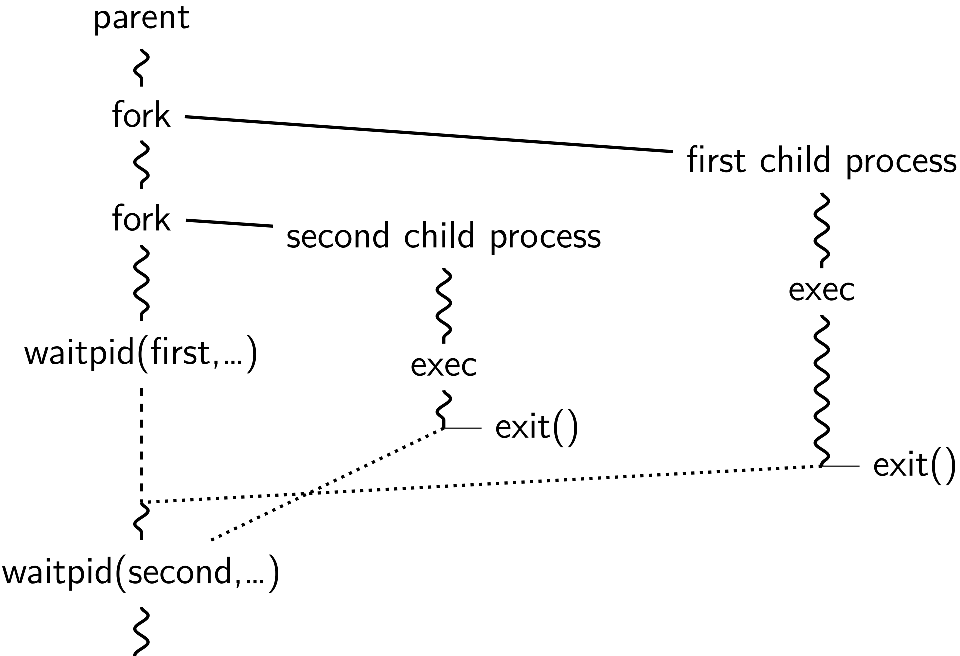
```
pid = fork();
if (pid == 0) {
    exec…(…);
    …
} else if (pid > 0) {
    waitpid(pid,…);
    …
}
…
```

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

```
main() {
    …
}
```

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

pattern with multiple?



POSIX process management

essential operations

process information: `getpid`

process creation: `fork`

running programs: `exec*`

- also `posix_spawn` (not widely supported), …

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

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

exercise (1)

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

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

A. L1 `(newline)` L2

B. L1 `(newline)` L2 `(newline)` L2

C. L2 `(newline)` L1

D. A and B

E. A and C

F. all of the above

G. something else

exercise (2)

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

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

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

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

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

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

E. A, B, and C

F. C and D

G. all of the above

H. something else

exercise (2)

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

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

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

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

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

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

E. A, B, and C

F. C and D

G. all of the above

H. something else

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 (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`
- 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);
}
```

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
```

shell assignment

implement a simple shell that supports redirection and pipeline

for Linux or another POSIX system (not xv6)

- but our supplied tests assume specifically Linux (**not OS X**)

...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

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)

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

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)
- ...and more

`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)

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

e.g. after `fork` same pointer shared in parent, child

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

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

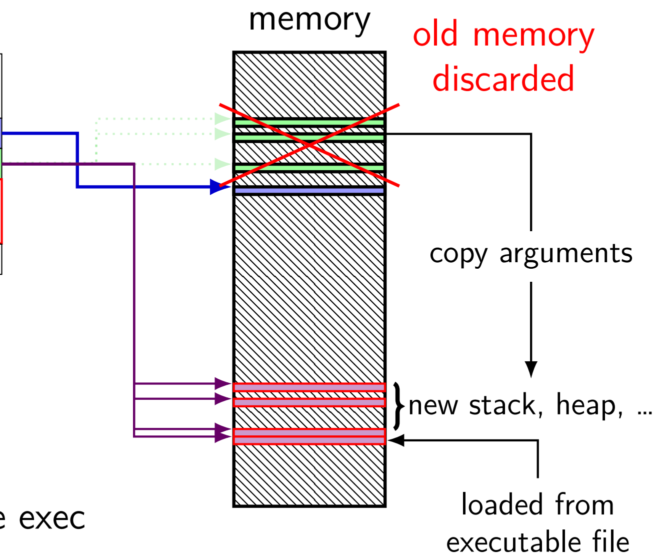
the process control block

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

kernel stack	🟦
user memory	🟩

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

…	…

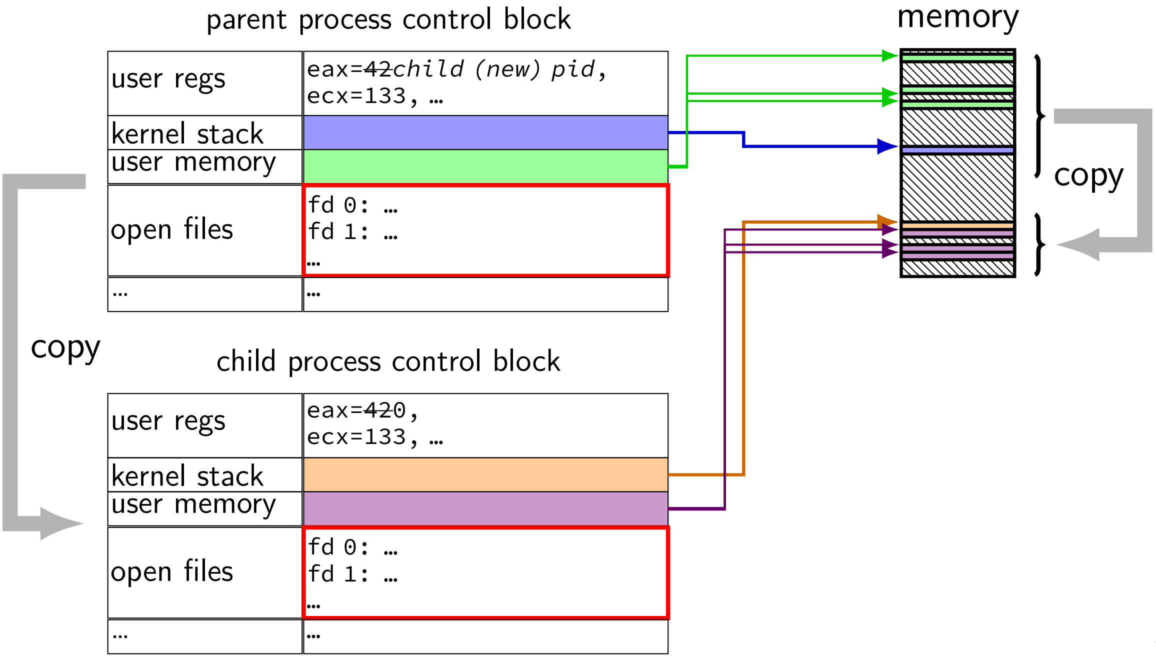


not changed!

redirection/etc.:

setup stdin/stdout before exec

fork copies open file list



parent process control block

user regs	eax=~~42~~*child (new) pid*,
	ecx=133, …

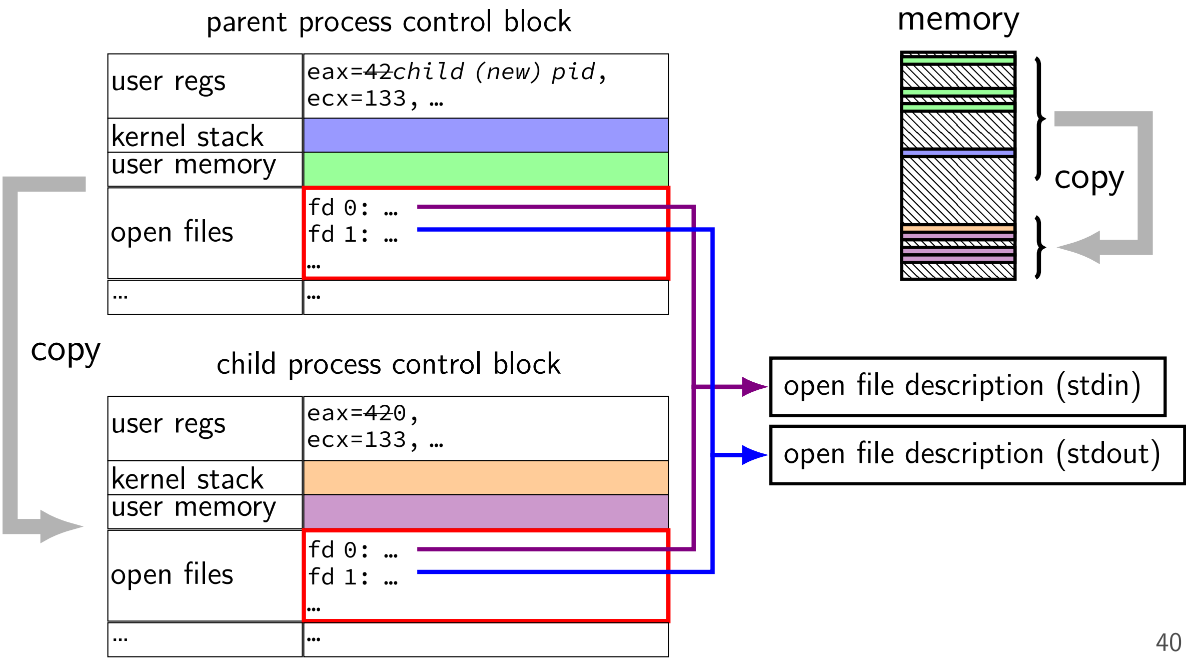
kernel stack	
user memory	
open files	fd 0: … fd 1: … …
…	…

child process control block

user regs	eax=~~42~~0,
	ecx=133, …

kernel stack	
user memory	
open files	fd 0: … fd 1: … …
…	…

fork copies open file list



parent process control block

user regs	`eax=~~42~~`*`child (new) pid`*`,`
	`ecx=133, …`

kernel stack	🟦
user memory	🟩

open files	`fd 0: …`
	`fd 1: …`
	`…`

…	…

child process control block

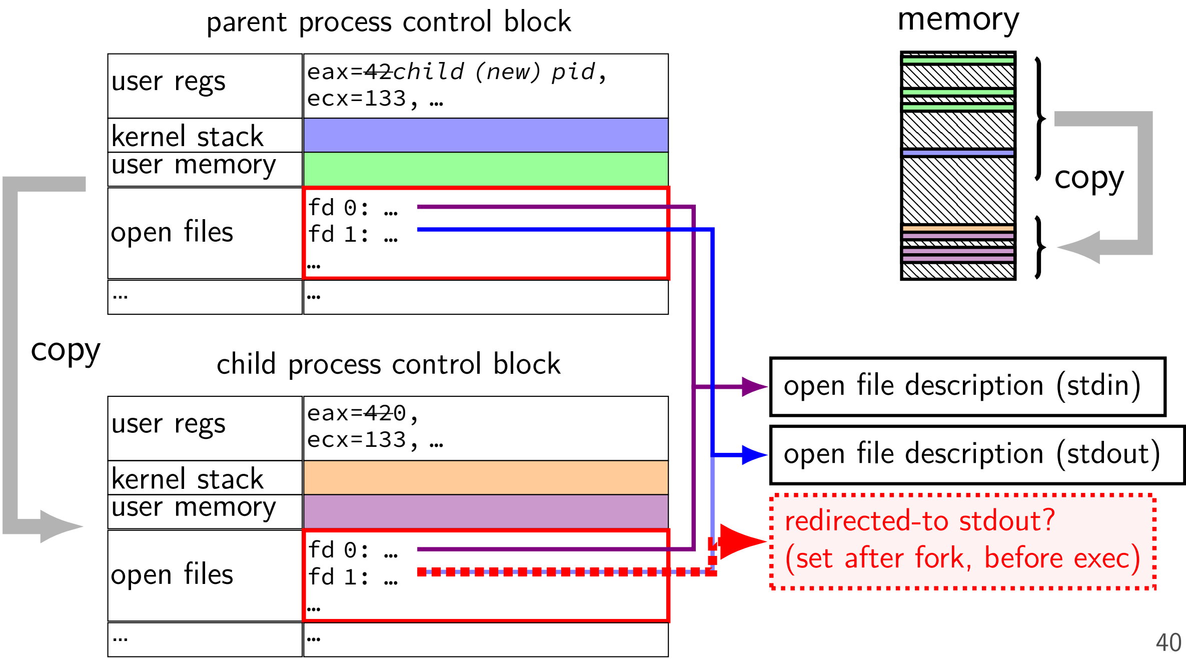
user regs	`eax=~~42~~0,`
	`ecx=133, …`

kernel stack	🟧
user memory	🟪

open files	`fd 0: …`
	`fd 1: …`
	`…`

…	…

fork copies open file list



parent process control block

user regs	eax=~~42~~*child (new) pid*,
	ecx=133, …

kernel stack	🟦
user memory	🟩
open files	fd 0: … fd 1: … …
…	…

copy

child process control block

user regs	eax=~~42~~0,
	ecx=133, …

kernel stack	🟧
user memory	🟪
open files	fd 0: … fd 1: … …
…	…

memory

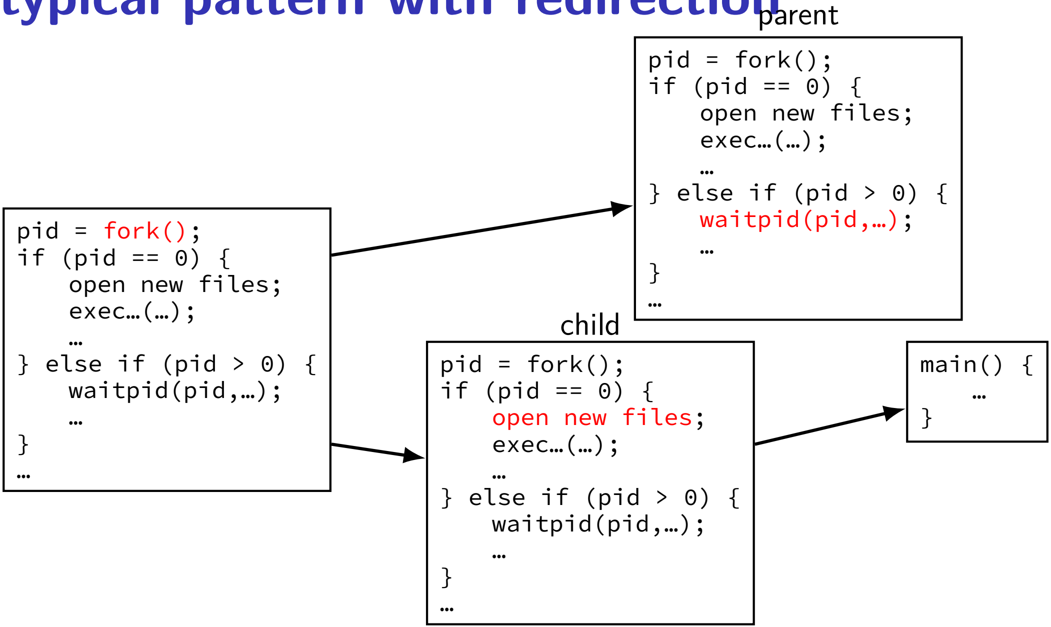
copy

open file description (stdin)

open file description (stdout)

redirected-to stdout?
(set after fork, before exec)

typical pattern with redirection



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

parent

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

child

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

```
main() {
    …
}
```

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

```
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");
```

open/dup/close/etc. and fd array

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

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

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

close: `ofile[fd] = NULL;`

fork:

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

(plus extra work to avoid leaking memory)

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);
```

exercise

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

Assume `open()` and `dup2()` *do not fail*, `write()` does not fail as long as the fd it writes to is open, fd `100` was closed and is not what open returns, and `STDOUT_FILENO` is initially open. What is written to `output.txt`?

A. ABCDE

B. ABCD

C. ABC

D. ACD

E. something else

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)

```
int pipe_fd[2];
if (pipe(pipe_fd) < 0)
    handle_error(); /* e.g. out of file d
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 fil
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 */ }
```

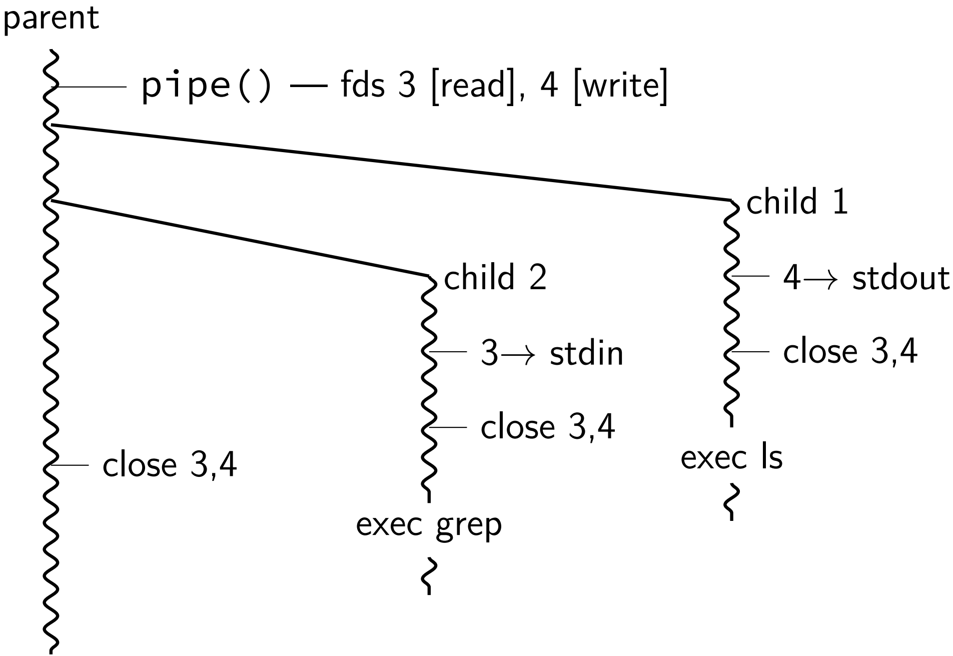
have habit of closing
to avoid ‘leaking’ file descriptors
you can run out

pipe and pipelines

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

```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);
}
close(pipe_fd[0]); close(pipe_fd[1]);
/* wait for processes, etc. */
```

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 it has a (subtle) bug.

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

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

```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

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

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: 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.

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

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:/us
PWD=/zf14/cr4bd
LANG=en_US.UTF-8
MODULEPATH=/sw/centos/Modules/modulefiles:/sw/linux-any/Modules/modulefiles
LOADEDMODULES=
KDEDIRS=/usr
```

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 */
  }
```

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"

execv and const

```
int execv(const char *path, char *const *argv);
```

`argv` is a pointer to constant pointer to char

probably should be a pointer to constant pointer to *constant* char

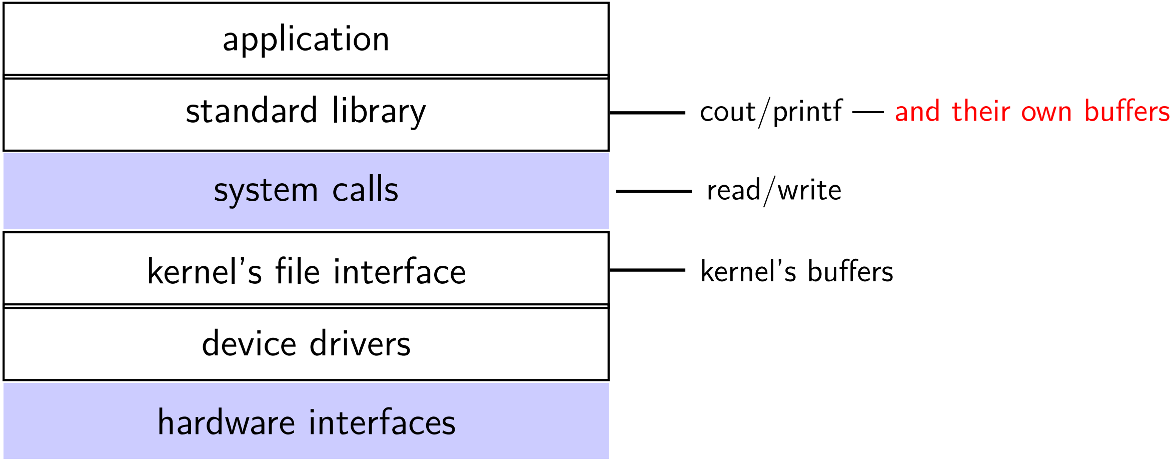
…this causes some awkwardness:

```
const char *array[] = { /* ... */ };
execv(path, array); // ERROR
```

solution: cast

```
const char *array[] = { /* ... */ };
execv(path, (char **) array); // or (char * const *)
```

layering



application

standard library — cout/printf — and their own buffers

system calls — read/write

kernel's file interface — kernel's buffers

device drivers

hardware interfaces

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

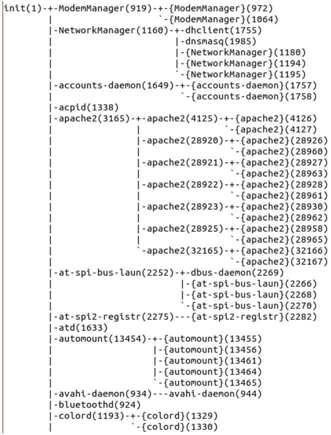
- cin, printf, etc. defined by C and C++ standards

parent and child processes

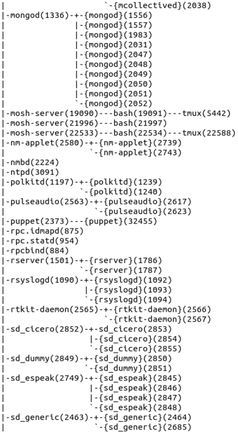
every process (but process id 1) has a *parent process* (`getppid()`)

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)
        |-avahi-daemon(934)---avahi-daemon(944)
        |-bluetoothd(924)
        |-colord(1193)-+-{colord}(1329)
        |              `-{colord}(1330)
```



```
|                   `-{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)
|                 |-{sd_espeak}(2847)
|                 `-{sd_espeak}(2848)
|-sd_generic(2463)-+-{sd_generic}(2464)
|                  `-{sd_generic}(2685)
```

parent and child questions...

what if parent process exits before child?

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

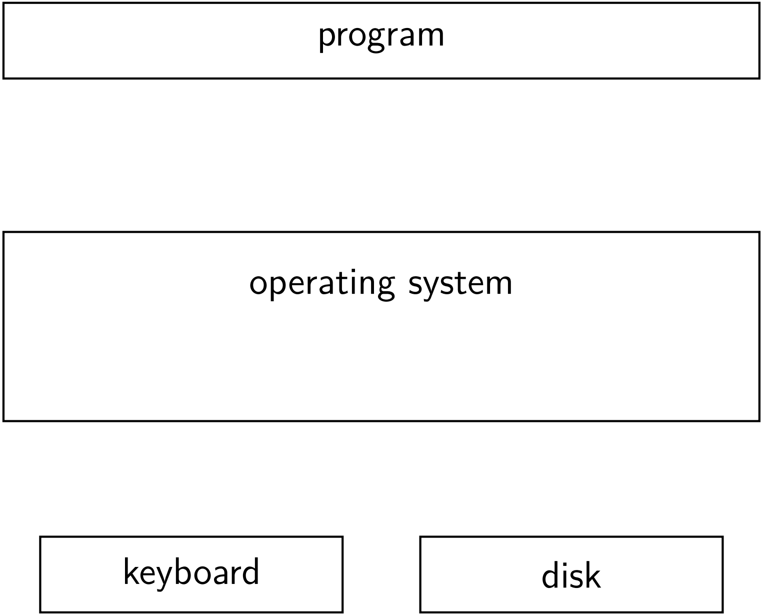
what if parent process never `waitpid()`s (or equivalent) for child?

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

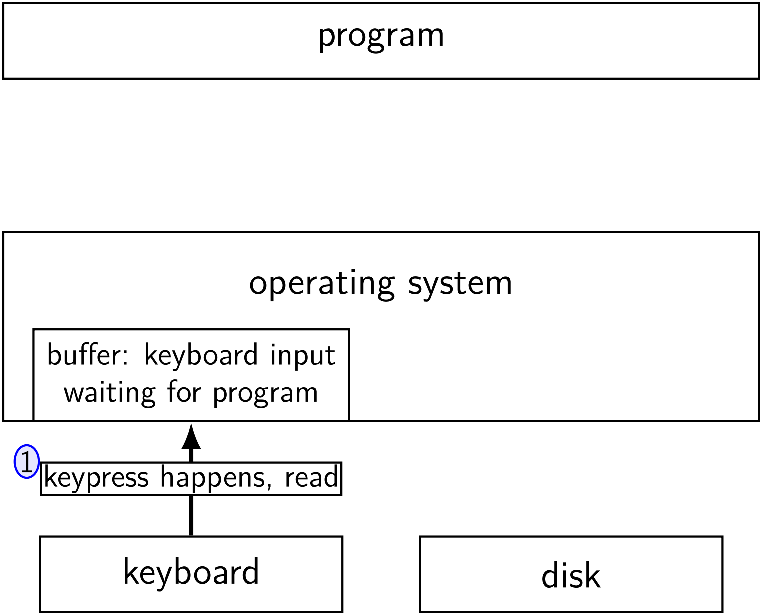
what if non-parent tries to `waitpid()` for child?

- waitpid fails

kernel buffering (reads)



kernel buffering (reads)



program

operating system

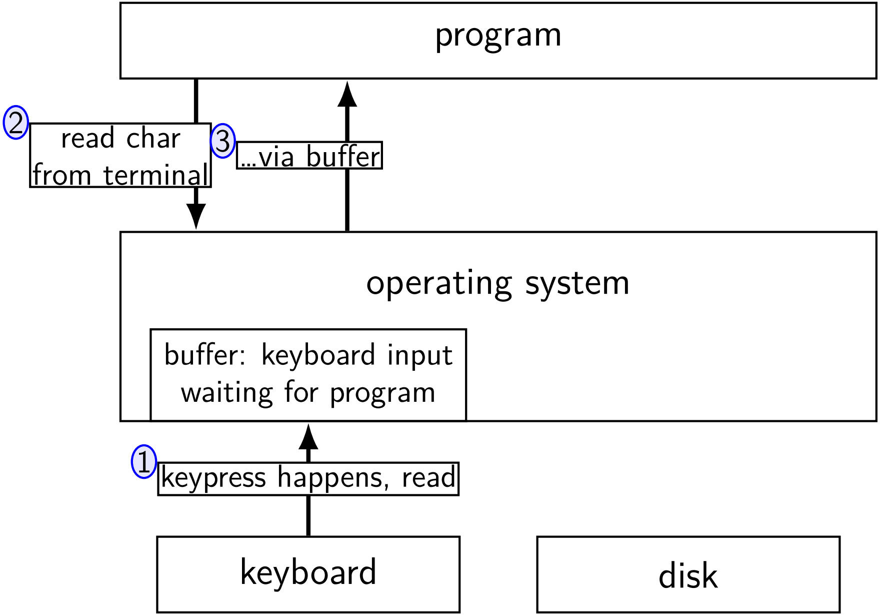
buffer: keyboard input waiting for program

1 keypress happens, read

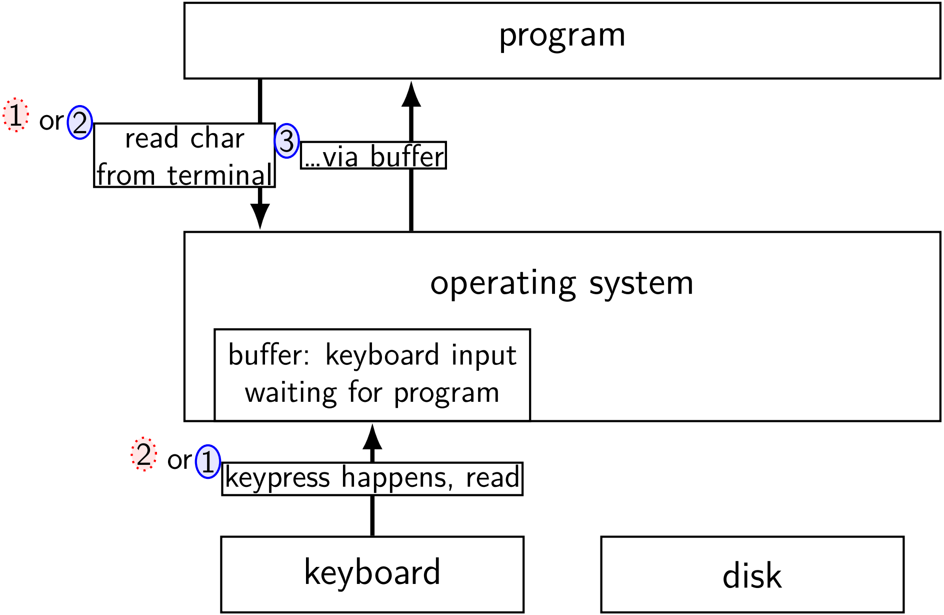
keyboard

disk

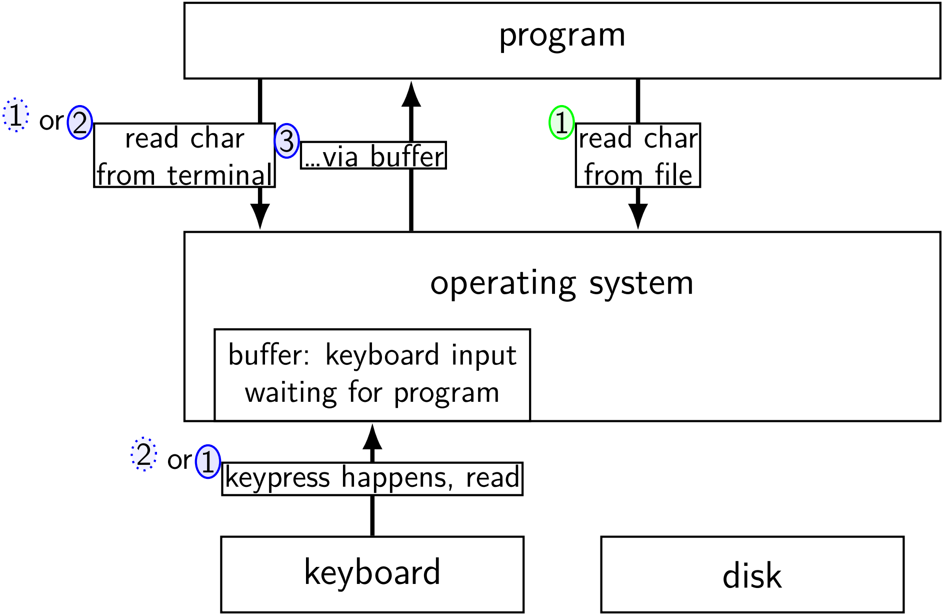
kernel buffering (reads)



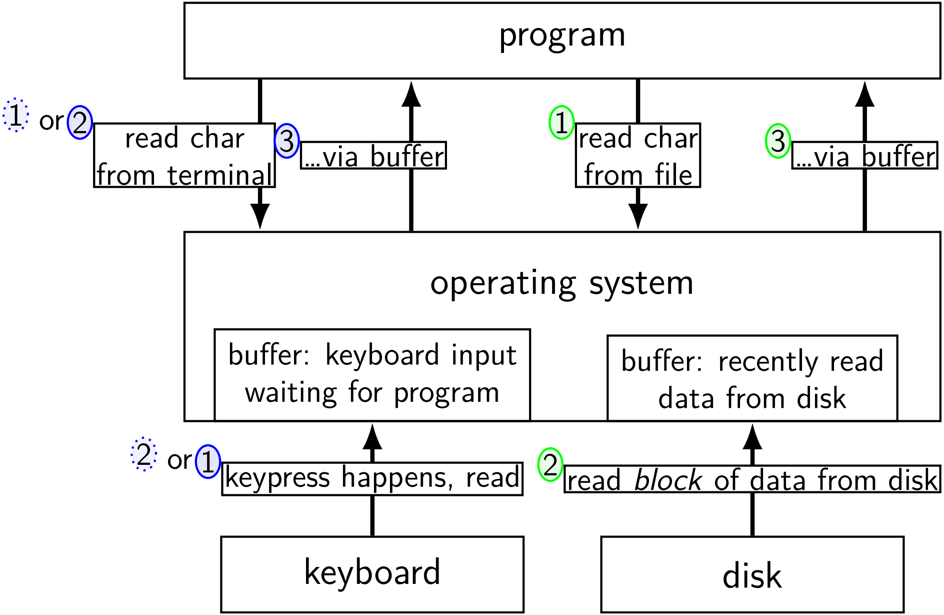
kernel buffering (reads)



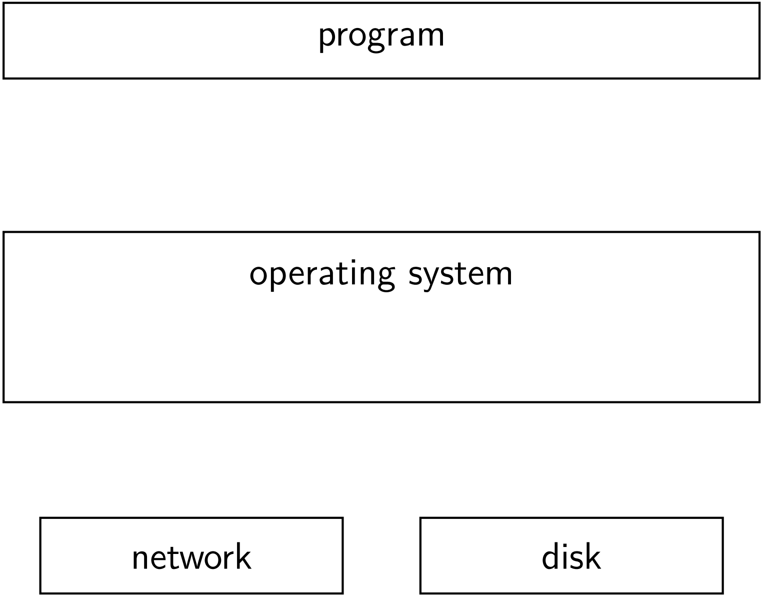
kernel buffering (reads)



kernel buffering (reads)



kernel buffering (writes)



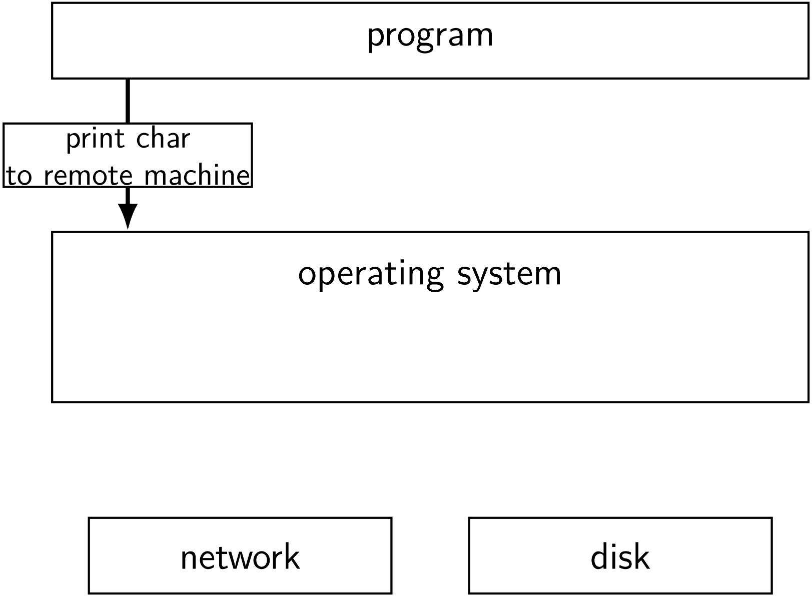
program

operating system

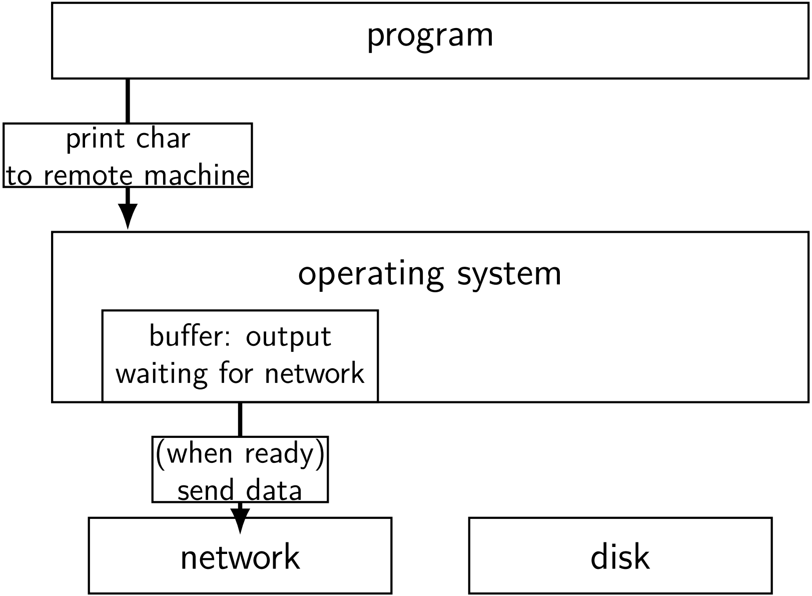
network

disk

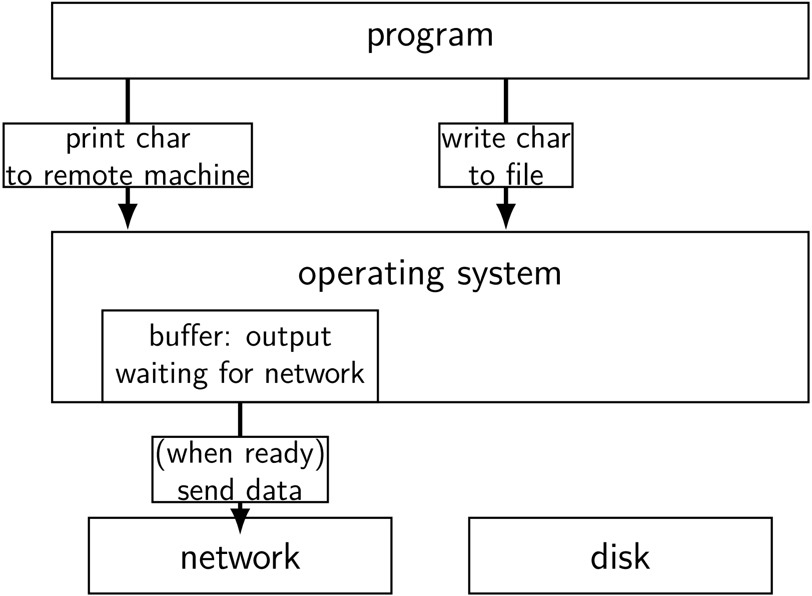
kernel buffering (writes)



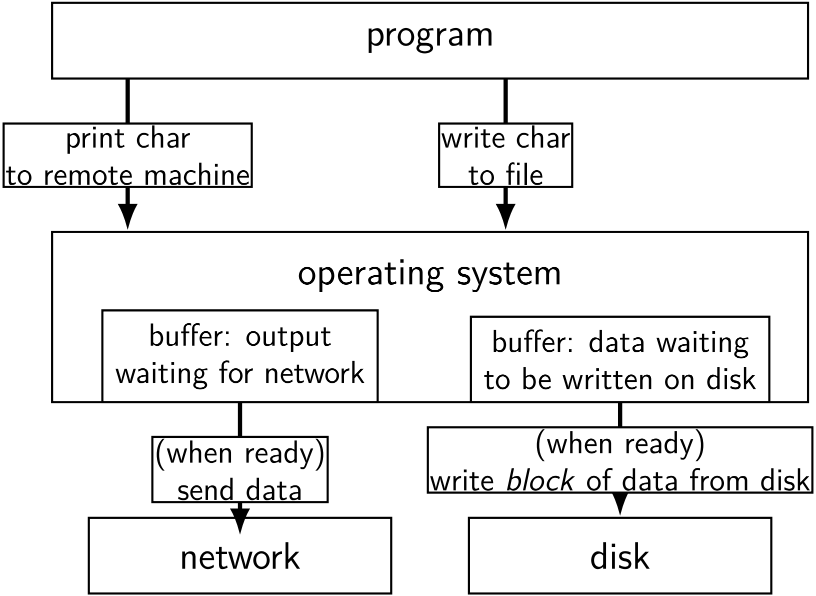
kernel buffering (writes)



kernel buffering (writes)



kernel buffering (writes)



program

print char to remote machine

write char to file

operating system

buffer: output waiting for network

buffer: data waiting to be written on disk

(when ready) send data

(when ready) write *block* of data from disk

network

disk

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

backup slides