

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`

fork

`pid_t fork()` — copy the current process

returns twice:

in *parent* (original process): pid of new *child* process

in *child* (new process): 0

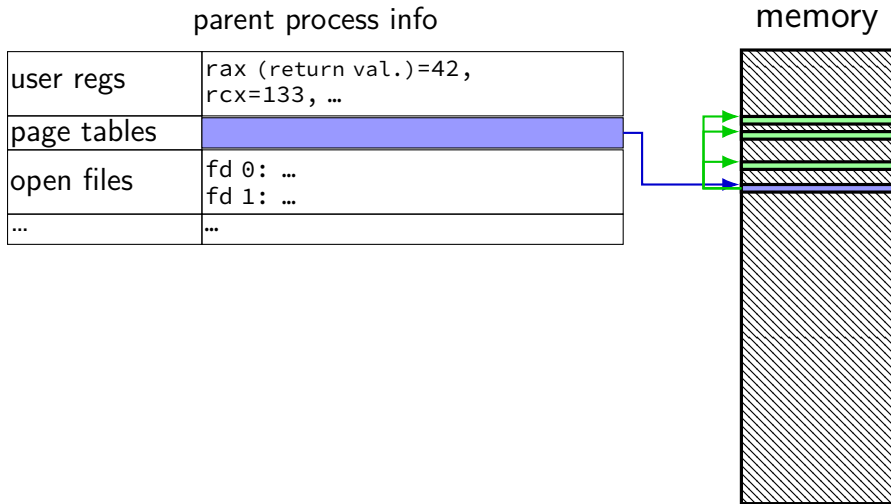
everything (but pid) duplicated in parent, child:

memory

file descriptors (later)

registers

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

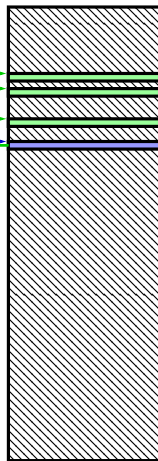


fork and process info (w/o copy-on-write)

parent process info

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

memory

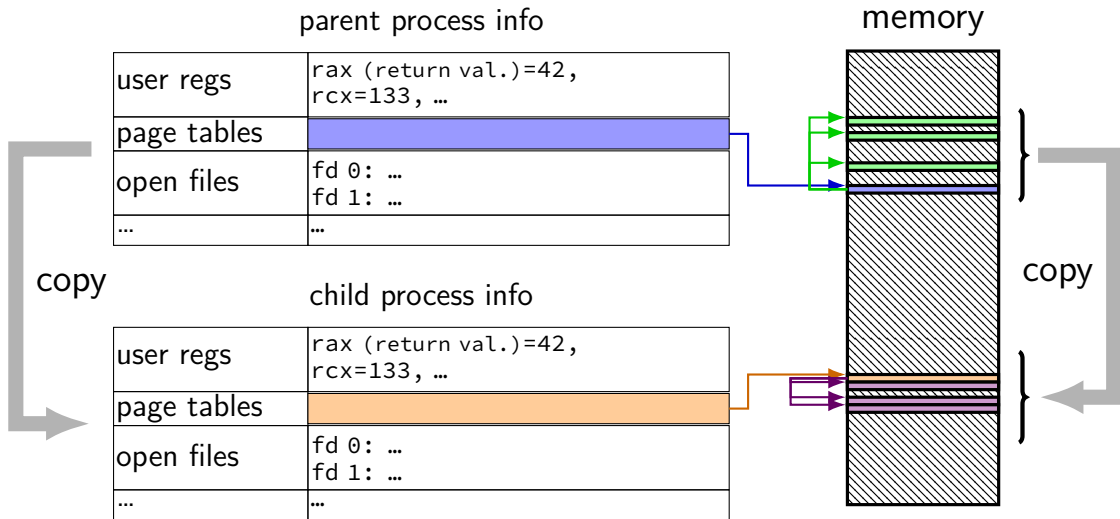


copy

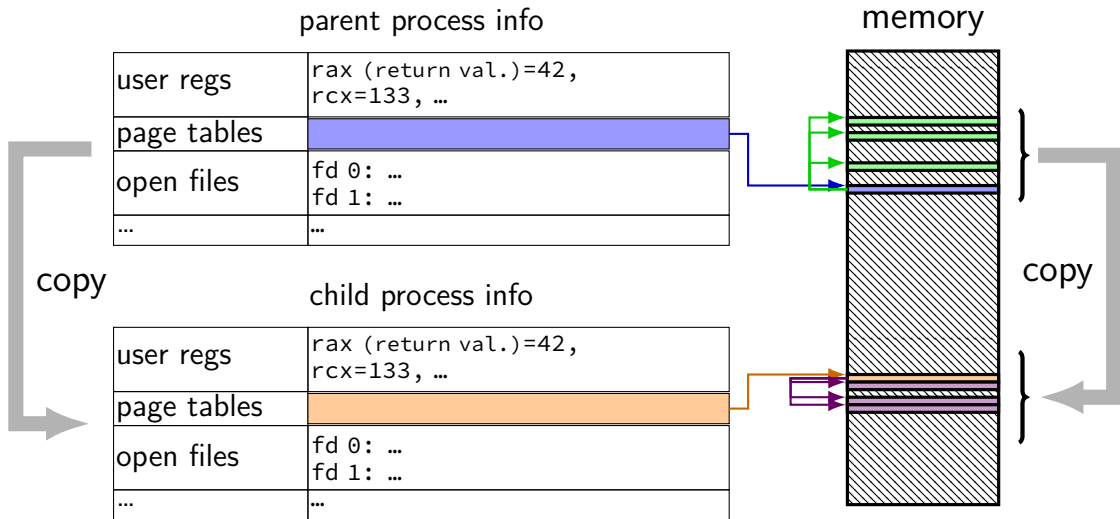
child process info

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

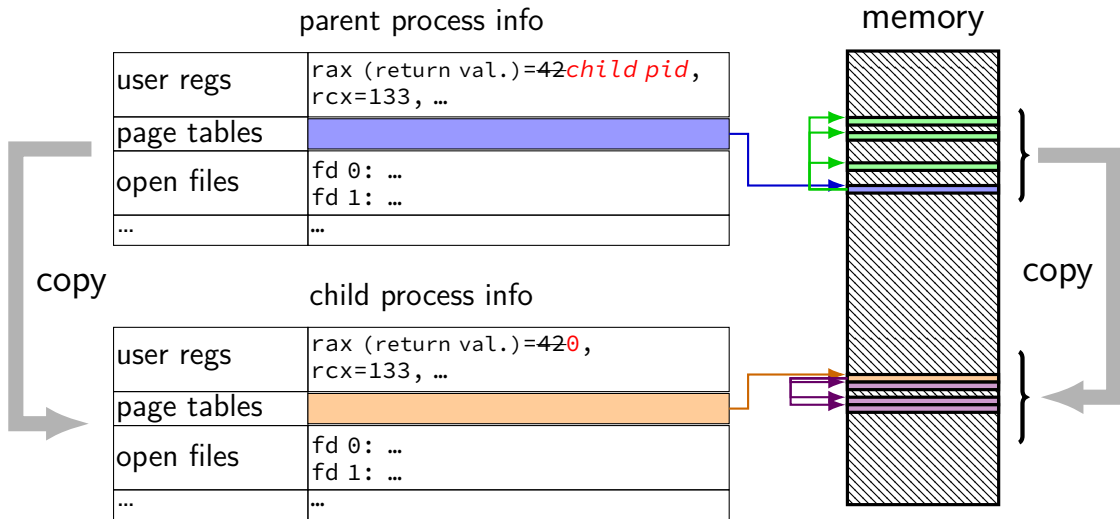
fork and process info (w/o copy-on-write)



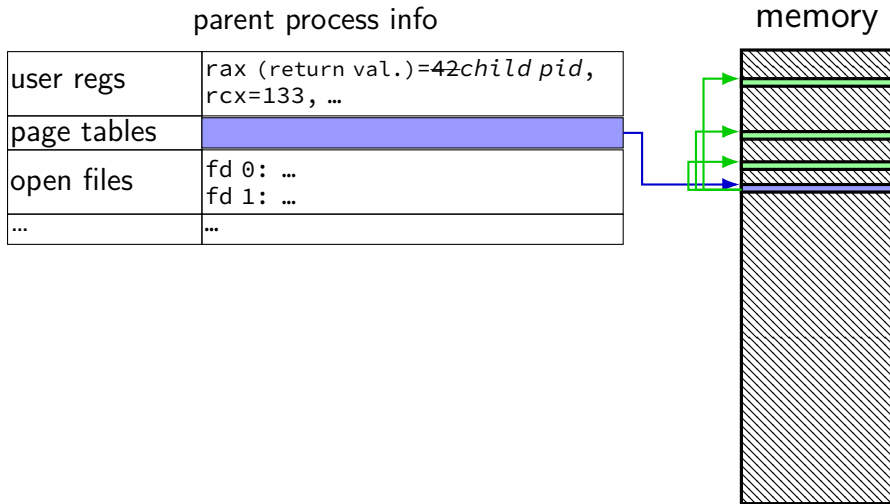
fork and process info (w/o copy-on-write)



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



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

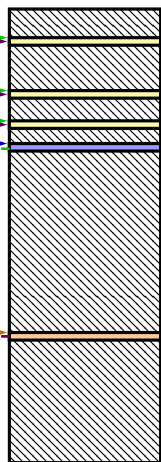


fork (w/ copy-on-write, if parent writes first)

parent process info

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

memory



shared
read-only

copy

child process info

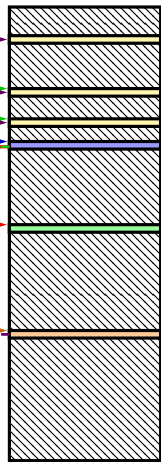
user regs	rax (return val.)=420, rcx=133, ...
page tables	
open files	fd 0: ... fd 1: ...
...	...

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

parent process info

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

memory



on parent
write
shared
read-only
copied
for
parent's
write

copy

child process info

user regs	rax (return val.)=420, rcx=133, ...
page tables	
open files	fd 0: ... fd 1: ...
...	...

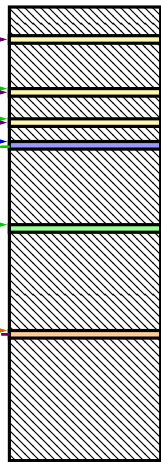


fork (w/ copy-on-write, if parent writes first)

parent process info

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

memory



no longer
shared
shared
read-only
copied
for
parent's
write

copy

child process info

user regs	rax (return val.)=420, rcx=133, ...
page tables	
open files	fd 0: ... fd 1: ...
...	...

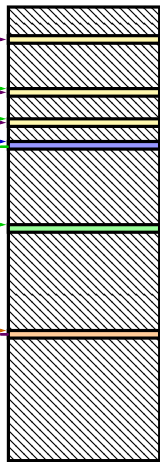


fork (w/ copy-on-write, if parent writes first)

parent process info

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

memory



} copied
for
parent's
write

child process info

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

copy



fork example

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(int argc, char *argv[]) {
    pid_t pid = getpid();
    printf("Parent pid: %d\n", (int) pid);
    pid_t child_pid = fork();
    if (child_pid > 0) {
        /* Parent Process */
        pid_t my_pid = getpid();
        printf("[%d] parent of [%d]\n", (int) my_pid, (int) child_pid);
    } else if (child_pid == 0) {
        /* Child Process */
        pid_t my_pid = getpid();
        printf("[%d] child\n", (int) my_pid);
    } else {
        perror("Fork failed");
    }
    return 0;
}
```

fork example

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    pid_t pid = getpid();
    printf("Parent pid: %d\n", (int) pid);
    pid_t child_pid = fork();
    if (child_pid > 0) {
        /* Parent Process */
        pid_t my_pid = getpid();
        printf("[%d] parent of [%d]\n", (int) my_pid, (int) child_pid);
    } else if (child_pid == 0) {
        /* Child Process */
        pid_t my_pid = getpid();
        printf("[%d] child\n", (int) my_pid);
    } else {
        perror("Fork failed");
    }
    return 0;
}
```

getpid — returns current process pid

fork example

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(int argc, char **argv) {
    pid_t pid;
    printf("Parent\n");
    pid_t child_pid = fork();
    if (child_pid > 0) {
        /* Parent Process */
        pid_t my_pid = getpid();
        printf("[%d] parent of [%d]\n", (int) my_pid, (int) child_pid);
    } else if (child_pid == 0) {
        /* Child Process */
        pid_t my_pid = getpid();
        printf("[%d] child\n", (int) my_pid);
    } else {
        perror("Fork failed");
    }
    return 0;
}
```

cast in case pid_t isn't int
POSIX doesn't specify (some systems it is, some not...)
(not necessary if you were using C++'s cout, etc.)

fork example

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
int main()
{
    pid_t child_pid;
    printf("Forking...\n");
    child_pid = fork();
    if (child_pid > 0) {
        /* Parent Process */
        pid_t my_pid = getpid();
        printf("[%d] parent of [%d]\n", (int) my_pid, (int) child_pid);
    } else if (child_pid == 0) {
        /* Child Process */
        pid_t my_pid = getpid();
        printf("[%d] child\n", (int) my_pid);
    } else {
        perror("Fork failed");
    }
    return 0;
}
```

prints out Fork failed: *error message*
(example *error message*: "Resource temporarily unavailable")
from error number stored in special global variable errno

fork example

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(int argc, char *argv[]) {
    pid_t pid = getpid();
    printf("Parent pid: %d\n", (int) pid);
    pid_t child_pid = fork();
    if (child_pid > 0) {
        /* Parent Process */
        pid_t my_pid = getpid();
        printf("[%d] parent of [%d]\n", (int) my_pid, (int) child_pid);
    } else if (child_pid == 0) {
        /* Child Process */
        pid_t my_pid = getpid();
        printf("[%d] child\n", (int) my_pid);
    } else {
        perror("Fork failed");
    }
    return 0;
}
```

Example output:

Parent pid: 100

[100] parent of [432]

[432] child

a fork question

```
int main() {  
    pid_t pid = fork();  
    if (pid == 0) {  
        printf("In child\n");  
    } else {  
        printf("Child %d\n", pid);  
    }  
    printf("Done!\n");  
}
```

Exercise: Suppose the pid of the parent process is 99 and child is 100. Give **two** possible outputs. (Assume no crashes, etc.)

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`

exec*

exec* — **replace** current program with new program

* — multiple variants

same pid, new process image

```
int execlv(const char *path, const char  
**argv)
```

path: new program to run

argv: array of arguments, terminated by null pointer

also other variants that take argv in different form and/or environment variables*

*environment variables = list of key-value pairs

execv example

```
...
child_pid = fork();
if (child_pid == 0) {
    /* child process */
    char *args[] = {"ls", "-l", NULL};
    execv("/bin/ls", args);
    /* execv doesn't return when it works.  

    So, if we got here, it failed. */
    perror("execv");
    exit(1);
} else if (child_pid > 0) {
    /* parent process */
    ...
}
```

execv example

```
...
child_pid = fork();
if (child_pid == 0) {
    /* child process */
    char *args[] = {"ls", "-l", NULL};
    execv("/bin/ls", args);
    /* execv doesn't return when it works.
        So, if we got
    perror("execv");
    exit(1);
} else if (child_pid > 0) {
    /* parent process */
    ...
}
```

used to compute argv, argc
when program's main is run

convention: first argument is program name

execv example

```
...
child_pid = fork();
if (child_pid == 0) {
    /* child process */
    char *args[] = {"ls", "-l", NULL};
    execv("/bin/ls", args);
    /* execv doesn't return when it works.
```

```
    So, if we got here,
    perror("execv");
    exit(1);
} else if (child_pid > 0) {
    /* parent process */
```

```
    ...
}
```

path of executable to run
need not match first argument
(but probably should match it)

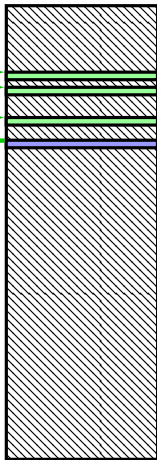
on Unix /bin is a directory
containing many common programs,
including `ls` ('list directory')

exec in the kernel

the process control block

user regs	eax=42, ecx=133, ...
pagetables	
open files	fd 0: (terminal ...) fd 1: ...
...	...

memory

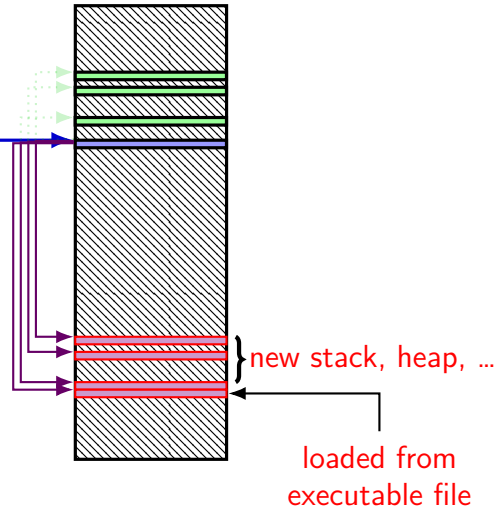


exec in the kernel

the process control block

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

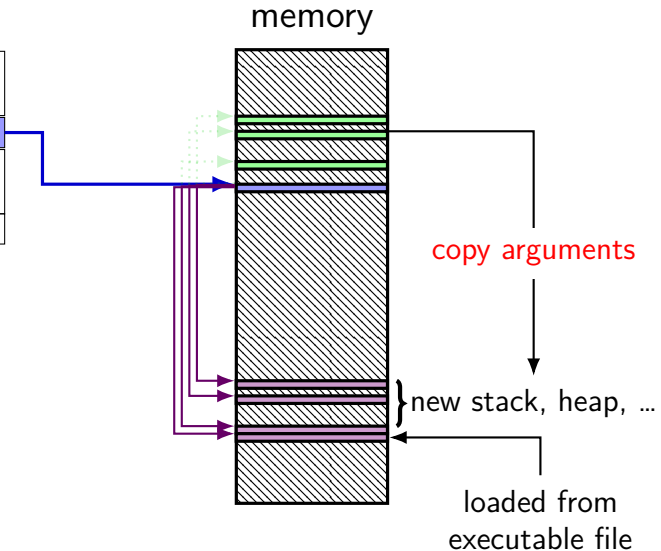
memory



exec in the kernel

the process control block

user regs	<code>eax=42</code> <i>init. val.</i> , <code>ecx=133</code> <i>init. val.</i> , ...
pagetables	
open files	fd 0: (terminal ...) fd 1: ...
...	...



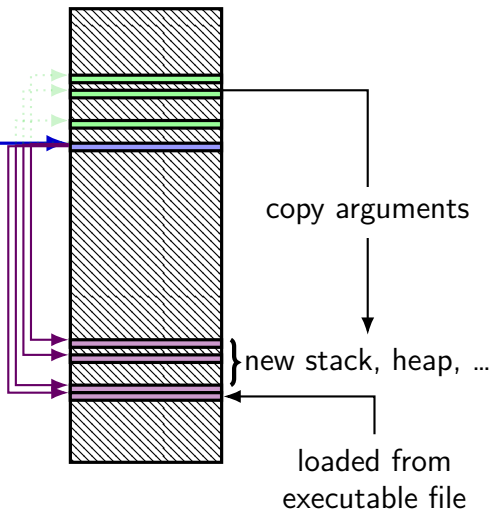
exec in the kernel

the process control block

user regs	<code>eax=42</code> <i>init. val.</i> , <code>ecx=133</code> <i>init. val.</i> , ...
pagetables	
open files	<code>fd 0: (terminal ...)</code> <code>fd 1: ...</code>
...	...

not changed!
(more on this later)

memory



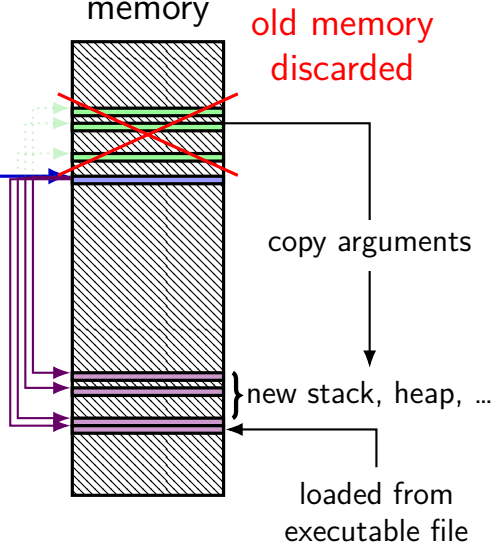
exec in the kernel

the process control block

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

not changed!
(more on this later)

memory



why fork/exec?

could just have a function to spawn a new program

Windows `CreateProcess()`; POSIX's (rarely used) `posix_spawn`

some other OSs do this (e.g. Windows)

needs to include API to set new program's state

e.g. without fork: either:

need function to set new program's current directory, *or*

need to change your directory, then start program, then change back

e.g. with fork: just change your current directory before exec

but allows OS to avoid 'copy everything' code

probably makes OS implementation easier

posix_spawn

```
pid_t new_pid;
const char argv[] = { "ls", "-l", NULL };
int error_code = posix_spawn(
    &new_pid,
    "/bin/ls",
    NULL /* null = copy current process's open files;
          if not null, do something else */,
    NULL /* null = no special settings for new process */,
    argv,
    NULL /* null = copy current process's "environment variables";
          if not null, do something else */
);
if (error_code == 0) {
    /* handle error */
}
```


some opinions (via HotOS '19)

A fork() in the road

Andrew Baumann
Microsoft Research

Jonathan Appavoo
Boston University

Orran Krieger
Boston University

Timothy Roscoe
ETH Zurich

ABSTRACT

The received wisdom suggests that Unix's unusual combination of `fork()` and `exec()` for process creation was an inspired design. In this paper, we argue that `fork` was a clever hack for machines and programs of the 1970s that has long outlived its usefulness and is now a liability. We catalog the ways in which `fork` is a terrible abstraction for the modern programmer to use, describe how it compromises OS implementations, and propose alternatives.

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

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 */
    ...
}
```

changing page tables

what happens to TLB when page table base pointer is changed?

e.g. context switch

most entries in TLB refer to things from **wrong process**

oops — read from the wrong process's stack?

changing page tables

what happens to TLB when page table base pointer is changed?

e.g. context switch

most entries in TLB refer to things from **wrong process**

oops — read from the wrong process's stack?

option 1: **invalidate** all TLB entries

side effect on “change page table base register” instruction

changing page tables

what happens to TLB when page table base pointer is changed?

e.g. context switch

most entries in TLB refer to things from **wrong process**

oops — read from the wrong process's stack?

option 1: **invalidate** all TLB entries

side effect on “change page table base register” instruction

option 2: TLB entries contain process ID

set by OS (special register)

checked by TLB in addition to TLB tag, valid bit

editing page tables

what happens to TLB when OS changes a page table entry?

most common choice: has to be handled **in software**

editing page tables

what happens to TLB when OS changes a page table entry?

most common choice: has to be handled **in software**

invalid to valid — nothing needed

- TLB doesn't contain invalid entries

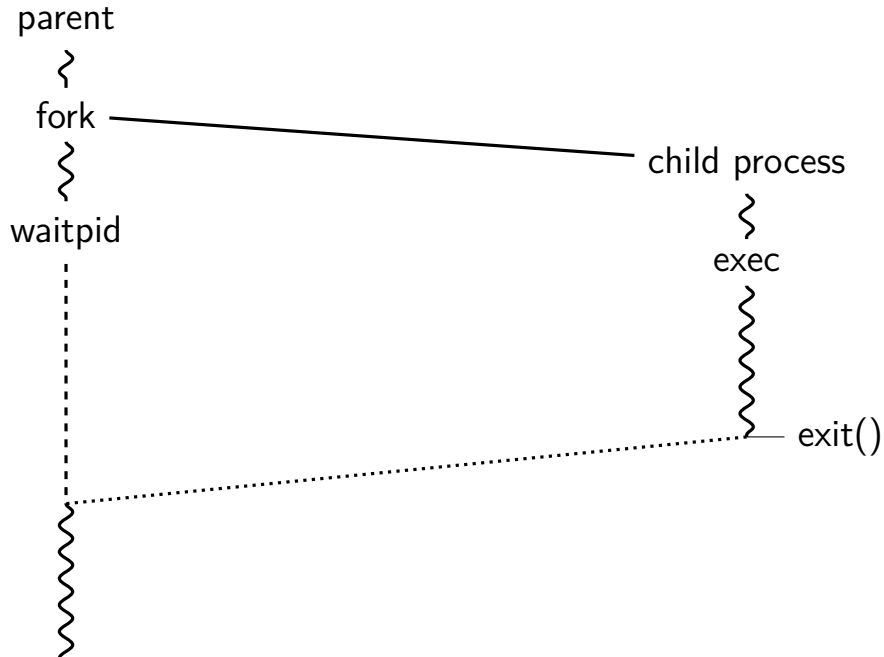
- MMU will check memory again

valid to invalid — **OS needs to tell processor** to invalidate it

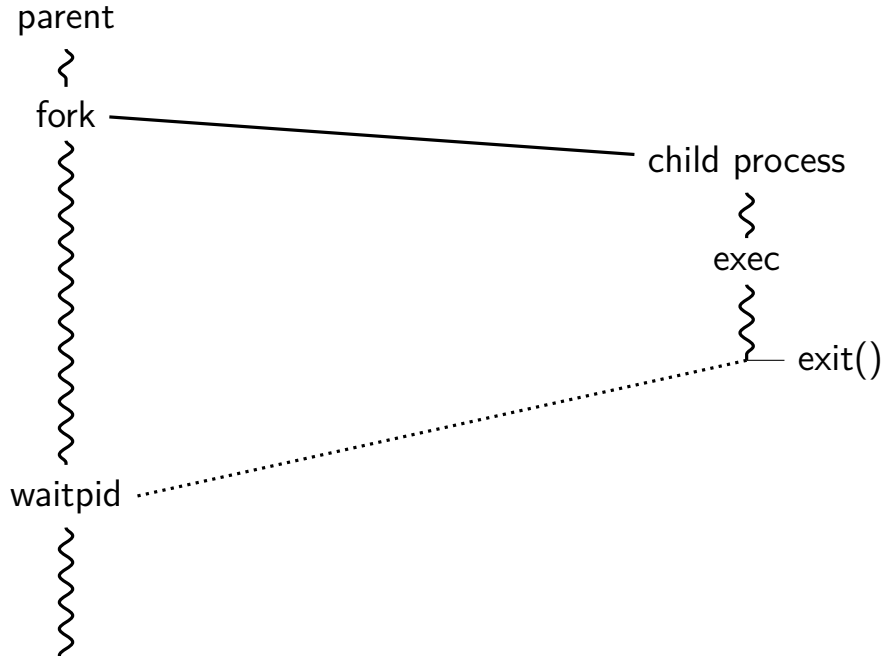
- special instruction (x86: `invlpg`)

valid to other valid — **OS needs to tell processor** to invalidate it

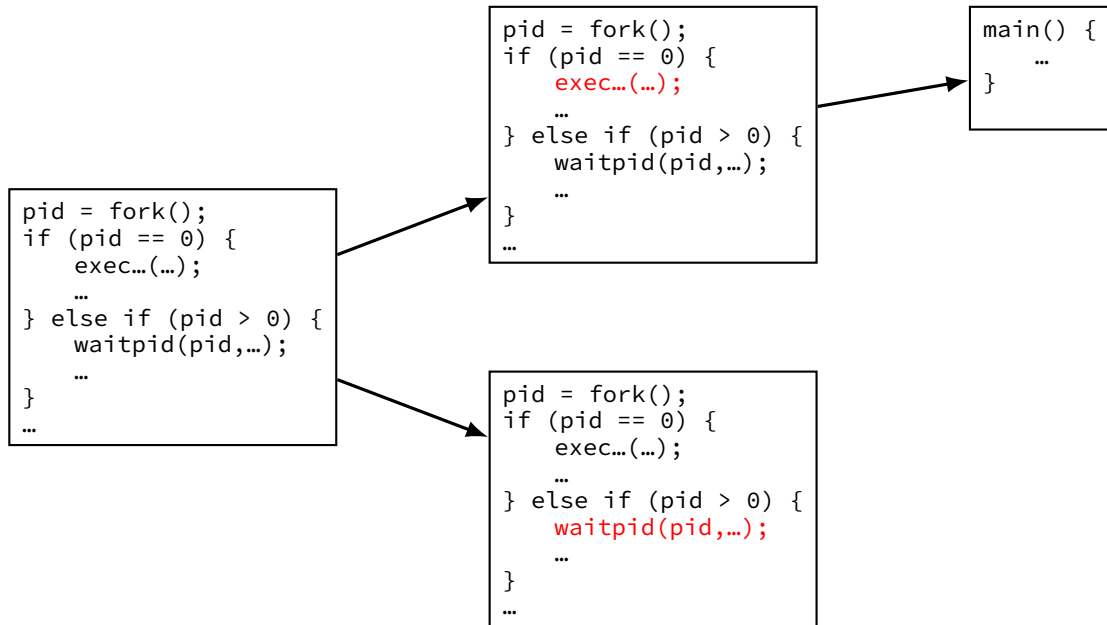
typical pattern



typical pattern (alt)



typical pattern (detail)



POSIX process management

essential operations

process information: `getpid`

process creation: `fork`

running programs: `exec*`

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

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

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

exercise (1)

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

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

A. L1 (newline) L2

D. A and B

B. L1 (newline) L2 (newline) L2

E. A and C

C. L2 (newline) L1

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

shell

allow user (= person at keyboard) to run applications

user's wrapper around process-management functions

aside: shell forms

POSIX: command line you have used before

also: graphical shells

e.g. OS X Finder, Windows explorer

other types of command lines?

completely different interfaces?

some POSIX command-line features

searching for programs

```
ls -l ≈ /bin/ls -l
```

```
make ≈ /usr/bin/make
```

running in background

```
./someprogram &
```

redirection:

```
./someprogram >output.txt
```

```
./someprogram <input.txt
```

pipelines:

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

some POSIX command-line features

searching for programs

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

running in background

```
./someprogram &
```

redirection:

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

pipelines:

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

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

```
ls -l ≈ /bin/ls -l
```

```
make ≈ /usr/bin/make
```

running in background

```
./someprogram &
```

redirection:

```
./someprogram >output.txt
```

```
./someprogram <input.txt
```

pipelines:

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

some POSIX command-line features

searching for programs

```
ls -l ≈ /bin/ls -l
```

```
make ≈ /usr/bin/make
```

running in background

```
./someprogram &
```

redirection:

```
./someprogram >output.txt
```

```
./someprogram <input.txt
```

pipelines:

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

file descriptors

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

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

“open file”: terminal · socket · regular file · pipe

file descriptor = index into array

usually what's used with system calls

stdio.h FILE*s usually have file descriptor index + buffer

special file descriptors

file descriptor 0 = standard input

file descriptor 1 = standard output

file descriptor 2 = standard error

constants in `unistd.h`

`STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO`

special file descriptors

file descriptor 0 = standard input

file descriptor 1 = standard output

file descriptor 2 = standard error

constants in `unistd.h`

`STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO`

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

more on this later

getting file descriptors

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

used internally by `fopen()`, etc.

also for files without normal filenames...:

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

close

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

close the file descriptor, deallocating that array index

does not affect other file descriptors

that refer to same “open file description”

(e.g. in `fork()`ed child or created via (later) `dup2`)

if last file descriptor for open file description, resources deallocated

returns 0 on success

returns -1 on error

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

shell redirection

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

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

`echo foo > output.txt:`

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

exec preserves open files

the process control block

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

not changed!

redirection/etc.:

setup stdin/stdout before exec

memory

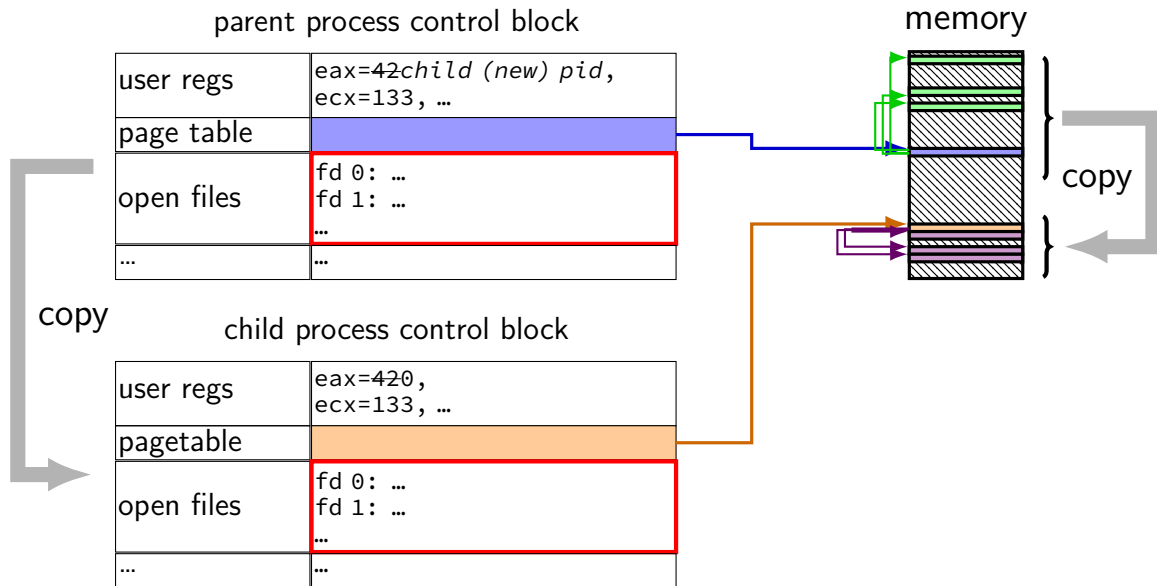
old memory
discarded

copy arguments

} new stack, heap, ...

loaded from
executable file

fork copies open file list



fork copies open file list

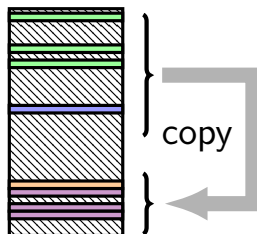
parent process control block

user regs	eax=42, child (new) pid, ecx=133, ...
page table	
open files	fd 0: ... fd 1:
...	...

child process control block

user regs	eax=420, ecx=133, ...
pagetable	
open files	fd 0: ... fd 1:
...	...

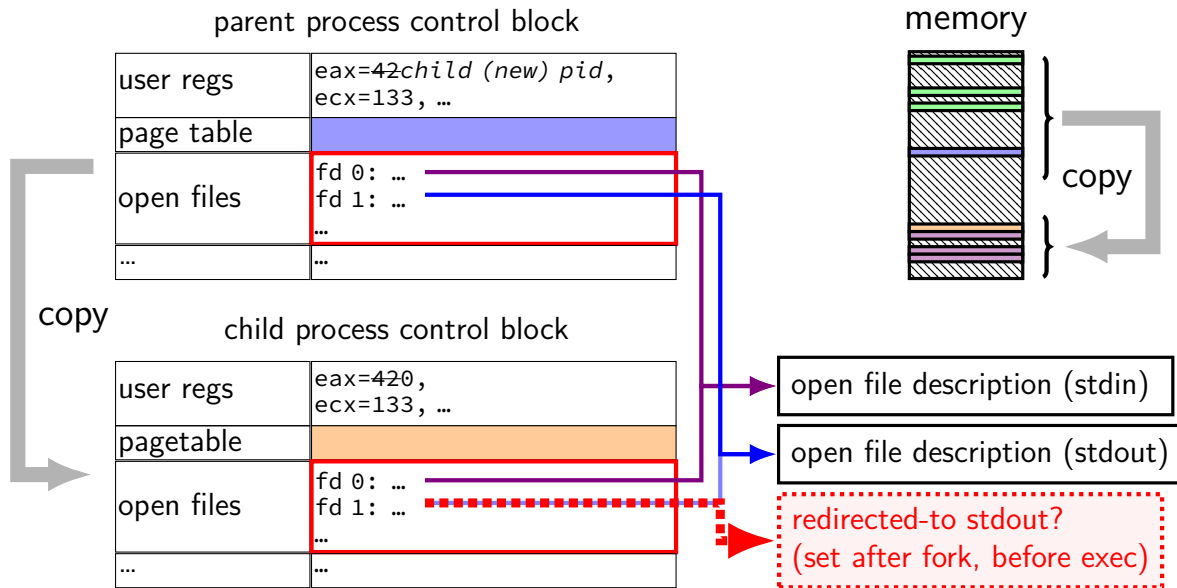
memory



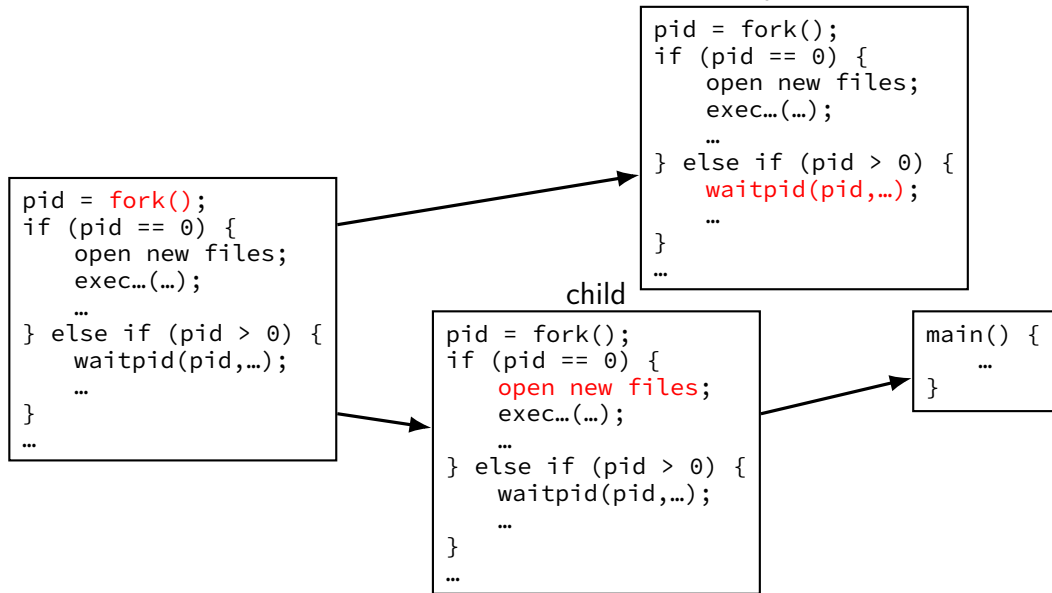
open file description (stdin)

open file description (stdout)

fork copies open file list



typical pattern with redirection



redirecting with exec

standard output/error/input are files

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

(probably after forking) open files to redirect

...and make them be standard output/error/input
using `dup2()` library call

then `exec`, preserving new standard output/etc.

reassigning file descriptors

redirection: `./program >output.txt`

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

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

reassigning and file table

```
struct process_info {  
    ...  
    struct open_file *files;  
};  
...  
process->files[STDOUT_FILENO] = process->files[opened-fd];  
syscall: dup2(opened-fd, STDOUT_FILENO);
```

reassigning file descriptors

redirection: `./program >output.txt`

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

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

tool: `int dup2(int oldfd, int newfd)`

make `newfd` refer to same open file as `oldfd`

same open file description

shares the current location in the file

(even after more reads/writes)

what if `newfd` already allocated — closed, then reused

dup2 example

redirects stdout to output to output.txt:

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

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

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

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

open/dup/close/etc. and fd array

```
struct process_info {
```

```
    ...
```

```
    struct file *files;
```

```
};
```

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

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

```
close: files[fd] = NULL;
```

```
fork:
```

```
    for (int i = ...)
```

```
        child->files[i] = parent->files[i];
```

(plus extra work to avoid leaking memory)

exercise

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

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

- A.** ABCDE **C.** ABC **E.** something else
B. ABCD **D.** ACD

pipes

special kind of file: pipes

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

created with `pipe()` library call

intended use: communicate between processes
like implementing shell pipelines

pipe()

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

then from one process...

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

and from another

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

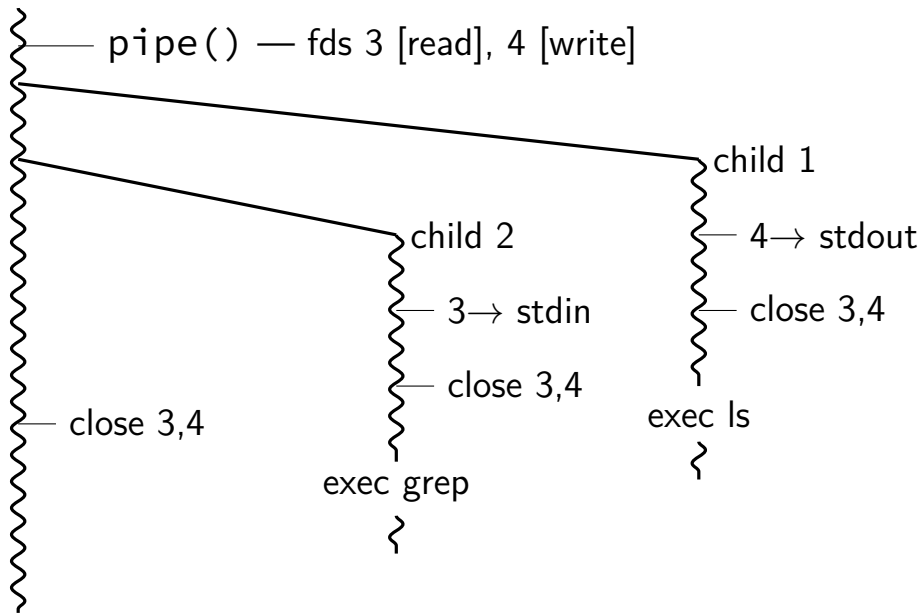
pipe and pipelines

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

example execution

parent



Unix API summary

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

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

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

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

backup slides

exit statuses

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

the status

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

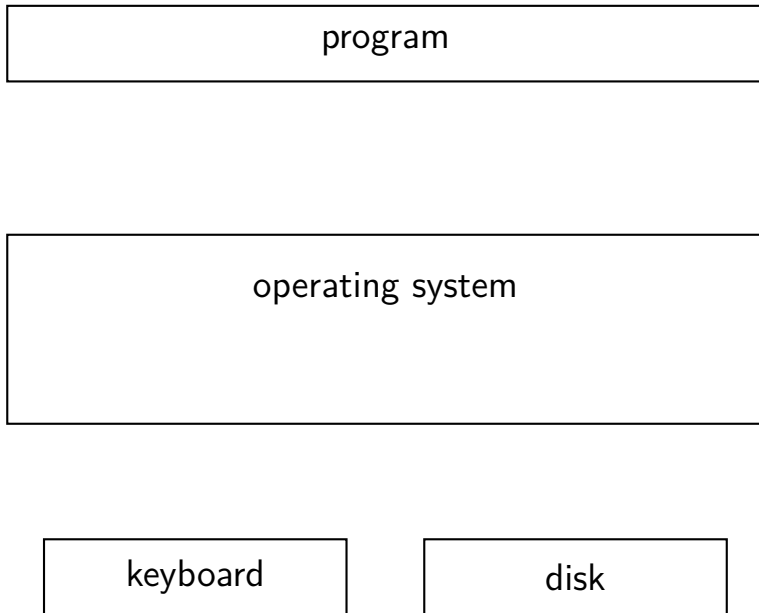
“status code” encodes both return value and if exit was abnormal
W* macros to decode it

the status

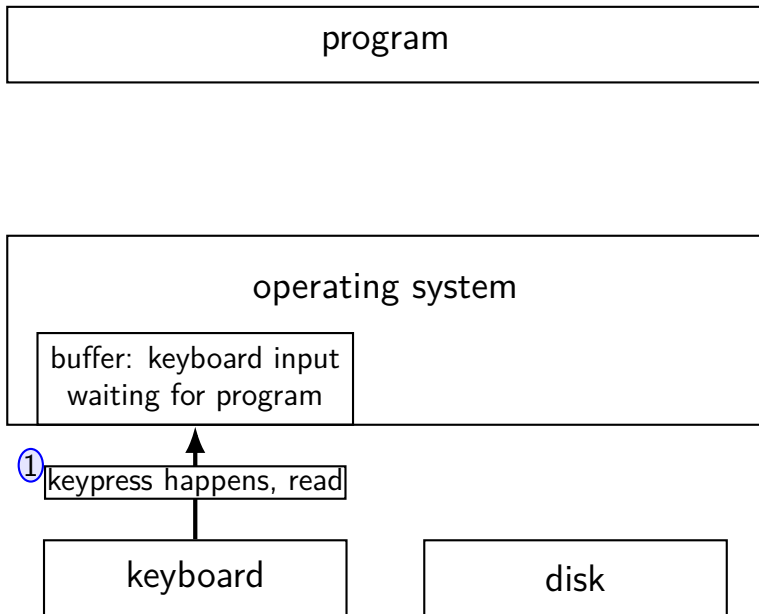
```
#include <sys/wait.h>
...
waitpid(child_pid, &status, 0);
if (WIFEXITED(status)) {
    printf("main returned or exit called with %d\n",
           WEXITSTATUS(status));
} else if (WIFSIGNALED(status)) {
    printf("killed by signal %d\n", WTERMSIG(status));
} else {
    ...
}
```

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

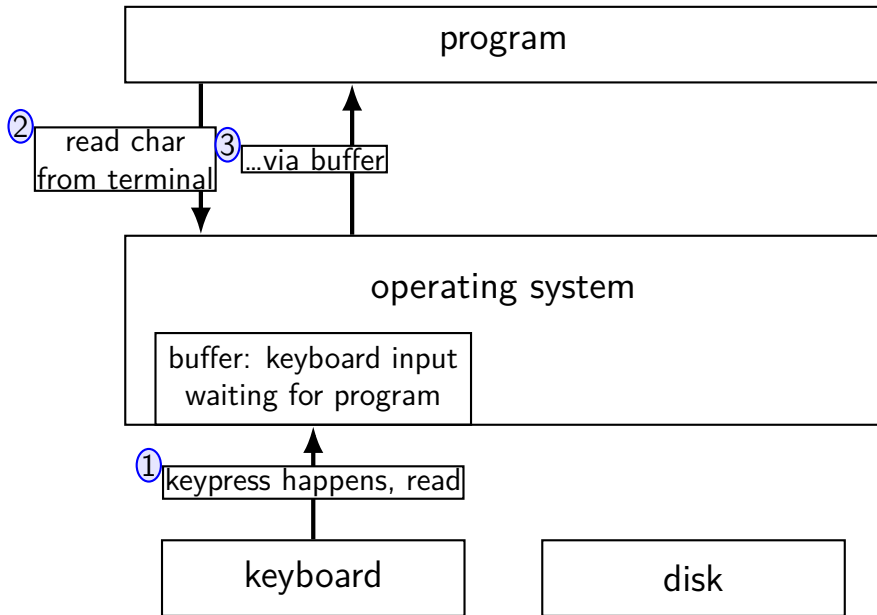
kernel buffering (reads)



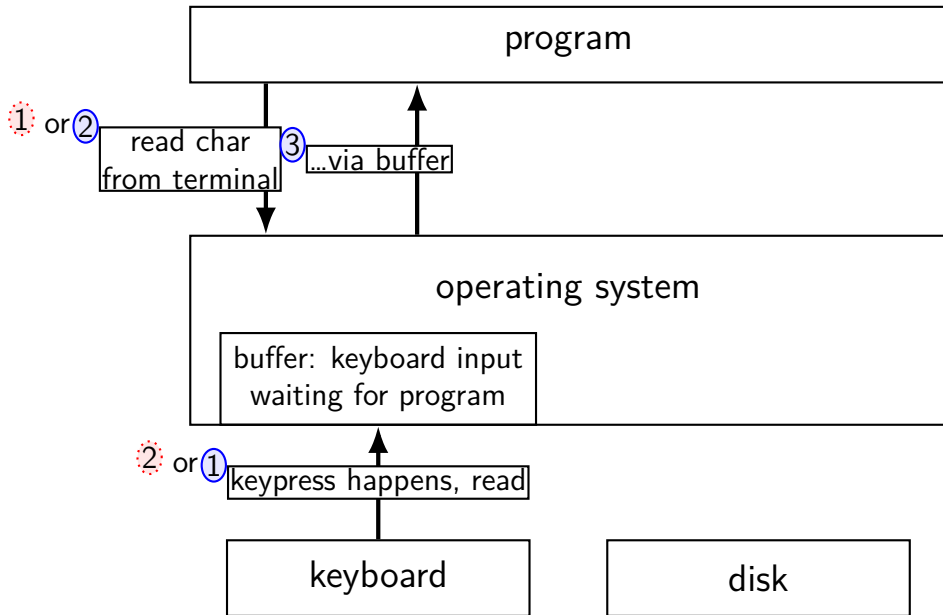
kernel buffering (reads)



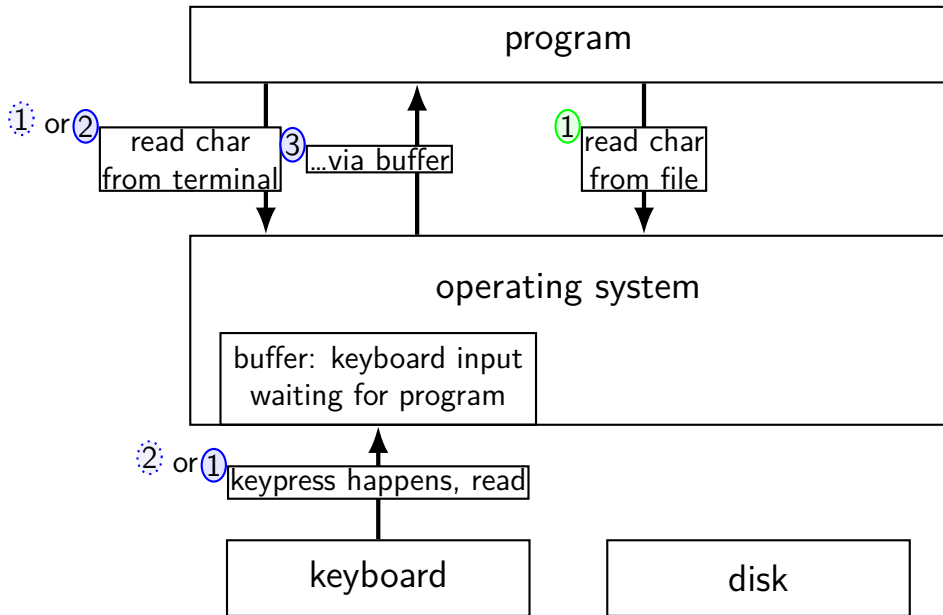
kernel buffering (reads)



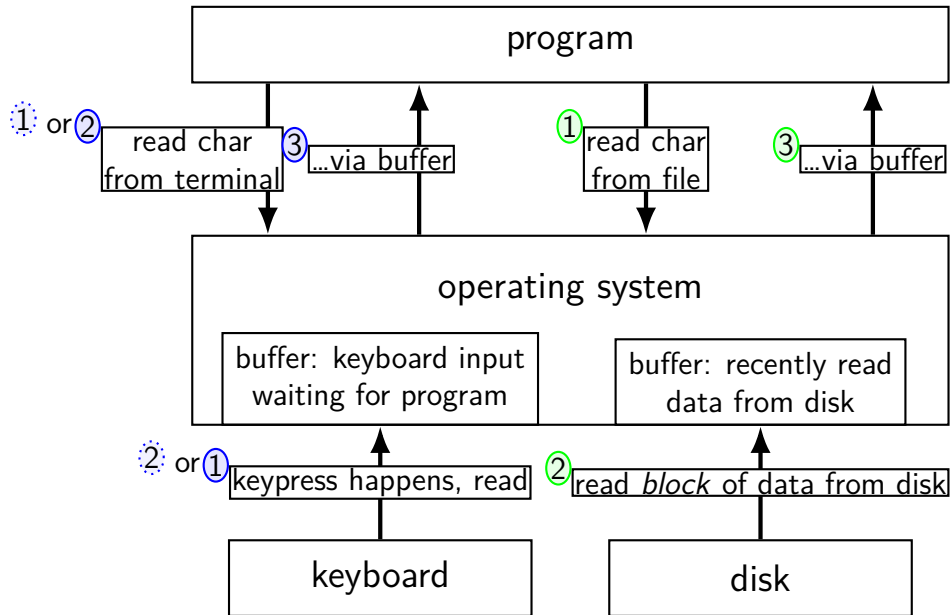
kernel buffering (reads)



kernel buffering (reads)



kernel buffering (reads)



kernel buffering (writes)

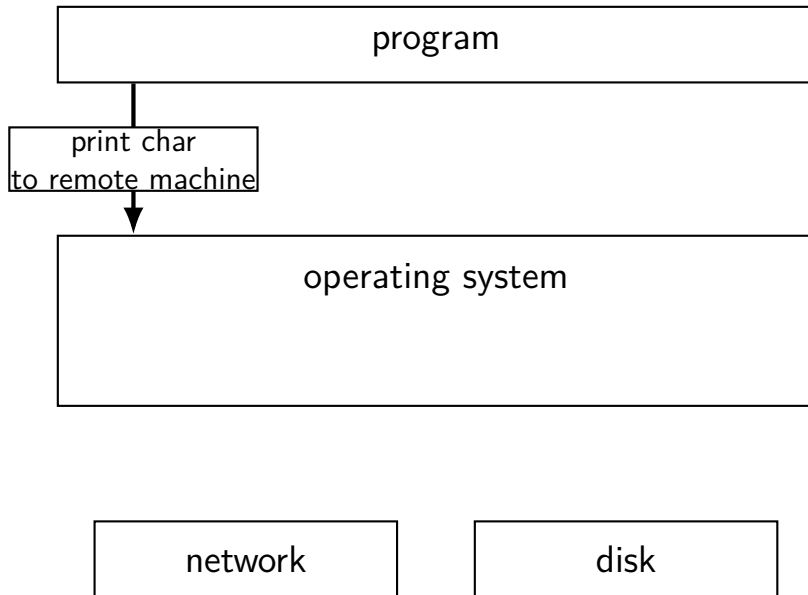
program

operating system

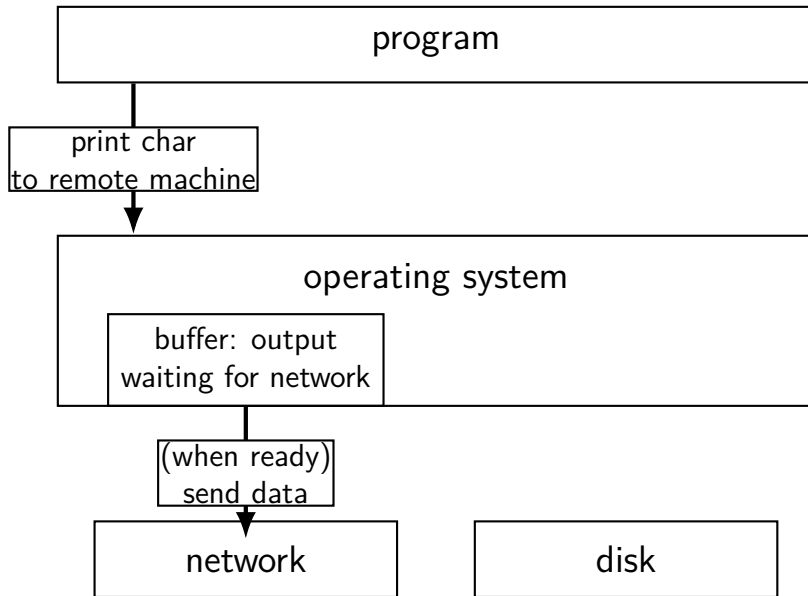
network

disk

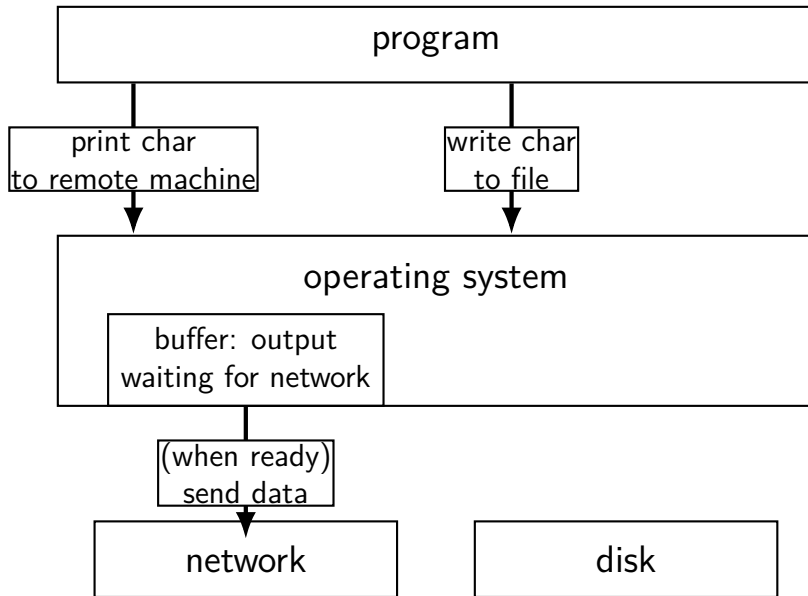
kernel buffering (writes)



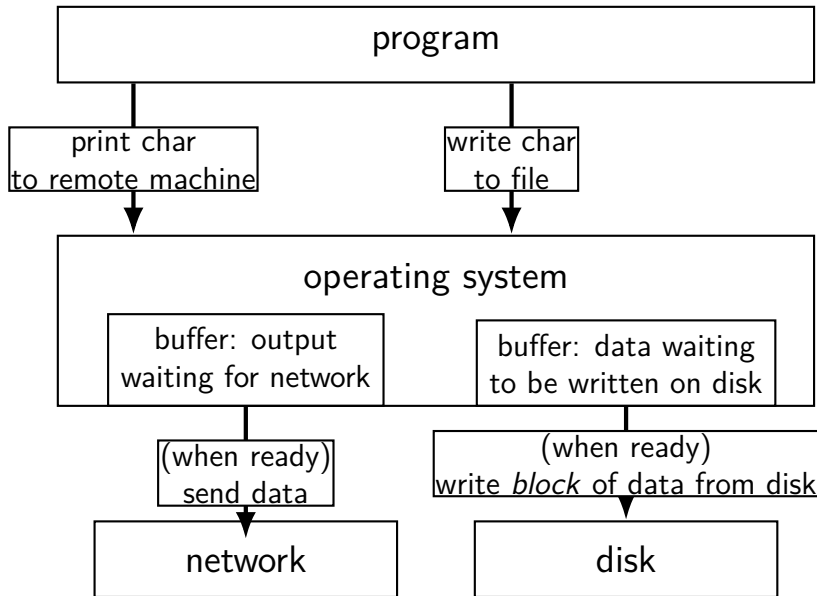
kernel buffering (writes)



kernel buffering (writes)



kernel buffering (writes)



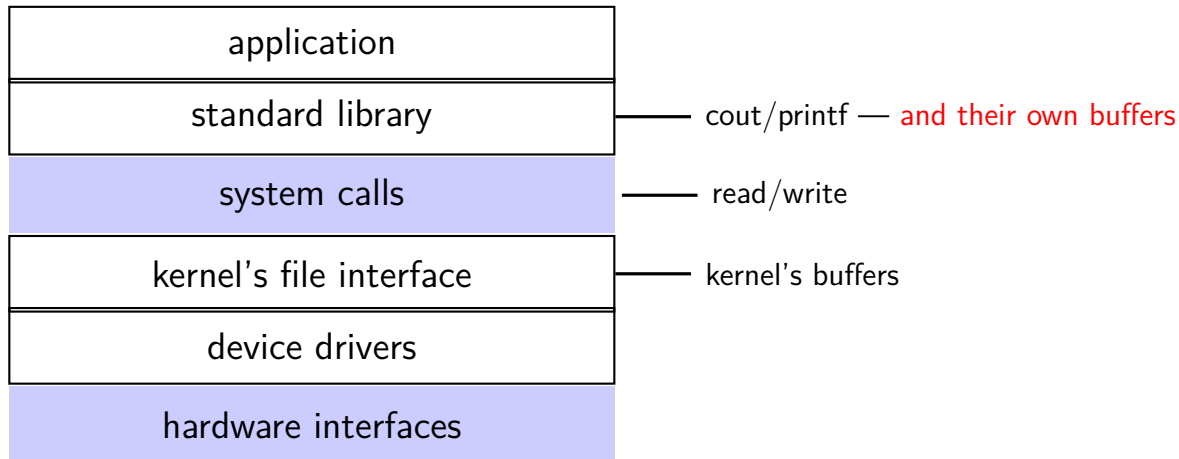
read/write operations

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

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

actual I/O operations — wait for device to be ready
trigger process to stop waiting if needed

layering



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

exercise

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

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

exercise solution

pipe example (1)

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

pipe example (1)

'standard' pattern with fork()

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

pipe example (1)

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

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

pipe example (1)

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

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

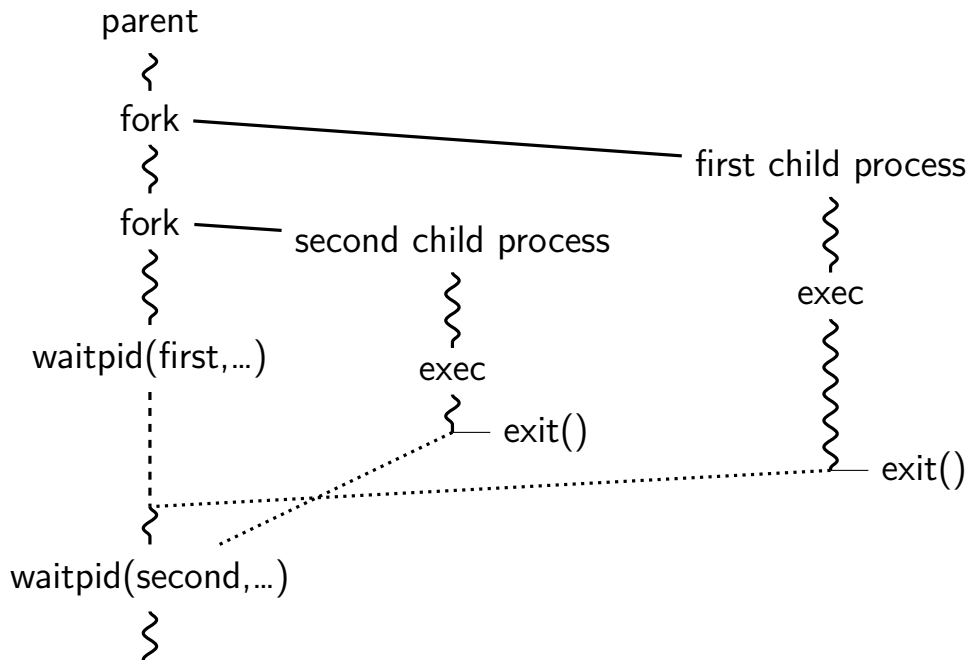
pipe() and blocking

BROKEN example:

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

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

pattern with multiple?



this class: focus on Unix

Unix-like OSes will be our focus

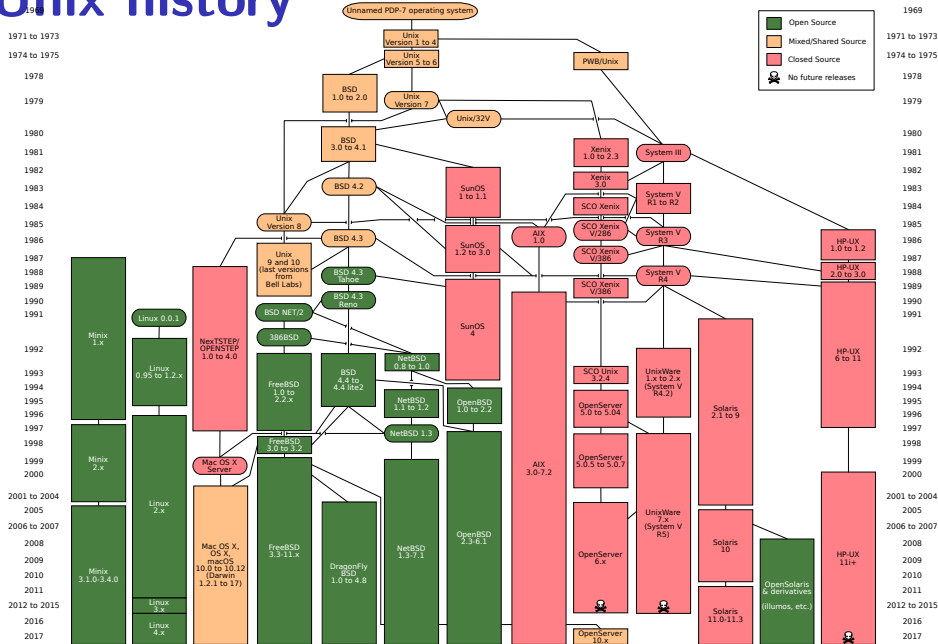
we have source code

used to from 2150, etc.?

have been around for a while

xv6 imitates Unix

Unix history



POSIX: standardized Unix

Portable Operating System Interface (POSIX)

“standard for Unix”

current version online:

<https://pubs.opengroup.org/onlinepubs/9699919799/>

(almost) followed by most current Unix-like OSes

...but OSes add extra features

...and POSIX doesn't specify everything

what POSIX defines

POSIX specifies the **library and shell interface**
source code compatibility

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

doesn't specify binary formats...

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

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

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`

getpid

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

process ids in ps

```
cr4bd@machine:~$ ps
```

PID	TTY	TIME	CMD
14777	pts/3	00:00:00	bash
14798	pts/3	00:00:00	ps

read/write

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

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

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

ssize_t is a signed integer type

 error code in *errno*

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

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

read'ing one byte at a time

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


write example

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

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.

aside: environment variables (1)

key=value pairs associated with every process:

```
$ printenv
```

```
MODULE_VERSION_STACK=3.2.10
```

```
MANPATH=:/opt/puppetlabs/puppet/share/man
```

```
XDG_SESSION_ID=754
```

```
HOSTNAME=labsrv01
```

```
SELINUX_ROLE_REQUESTED=
```

```
TERM=screen
```

```
SHELL=/bin/bash
```

```
HISTSIZE=1000
```

```
SSH_CLIENT=128.143.67.91 58432 22
```

```
SELINUX_USE_CURRENT_RANGE=
```

```
QTDIR=/usr/lib64/qt-3.3
```

```
OLDPWD=/zf14/cr4bd
```

```
QTINC=/usr/lib64/qt-3.3/include
```

```
SSH_TTY=/dev/pts/0
```

```
QT_GRAPHICSSYSTEM_CHECKED=1
```

```
USER=cr4bd
```

```
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or
```

```
MODULE_VERSION=3.2.10
```

```
MAIL=/var/spool/mail/cr4bd
```

```
PATH=/zf14/cr4bd/.cargo/bin:/zf14/cr4bd/bin:/usr/lib64/qt-3.3/bin:/usr/local/bin:/usr/bin:/u
```

```
PWD=/zf14/cr4bd
```

```
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")` \rightarrow *value*

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

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

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

```
char *envp[] = { "KEY1=value1", "KEY2=value2", NULL };
```

```
char *argv[] = { "somecommand", "some arg", NULL };
```

```
execve("/path/to/somecommand", argv, envp);
```

normal exec versions — keep same environment variables

aside: environment variables (3)

interpretation up to programs, but common ones...

`PATH=/bin:/usr/bin`

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

`HOME=/zf14/cr4bd`

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

`TERM=screen-256color`

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

...

multiple processes?

```
while (...) {  
    pid = fork();  
    if (pid == 0) {  
        exec ...  
    } else if (pid > 0) {  
        pids.push_back(pid);  
    }  
}  
  
/* retrieve exit statuses in order */  
for (pid_t pid : pids) {  
    waitpid(pid, ...);  
    ...  
}
```

waiting for all children

```
#include <sys/wait.h>

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

multiple processes?

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

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

'waiting' without waiting

```
#include <sys/wait.h>
```

```
...
```

```
pid_t return_value = waitpid(child_pid, &status, WNOHANG);
```

```
if (return_value == (pid_t) 0) {
```

```
    /* child process not done yet */
```

```
} else if (child_pid == (pid_t) -1) {
```

```
    /* error */
```

```
} else {
```

```
    /* handle child_pid exiting */
```

```
}
```

parent and child processes

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

this is the process that can wait for it

creates tree of processes (Linux pstree command):

```
init(1)-+-ModemManager(919)-+-{ModemManager}(972)
|   +-{ModemManager}(1064)
|   +-NetworkManager(1160)-+-dhcpcd(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)
|   +-mongodb(1336)-+-{mongodb}(1556)
|   |   +-{mongodb}(1557)
|   |   +-{mongodb}(1983)
|   |   +-{mongodb}(2031)
|   |   +-{mongodb}(2047)
|   |   +-{mongodb}(2048)
|   |   +-{mongodb}(2049)
|   |   +-{mongodb}(2050)
|   |   +-{mongodb}(2051)
|   |   +-{mongodb}(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.tnmapd(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_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()`s (or equivalent) for child?

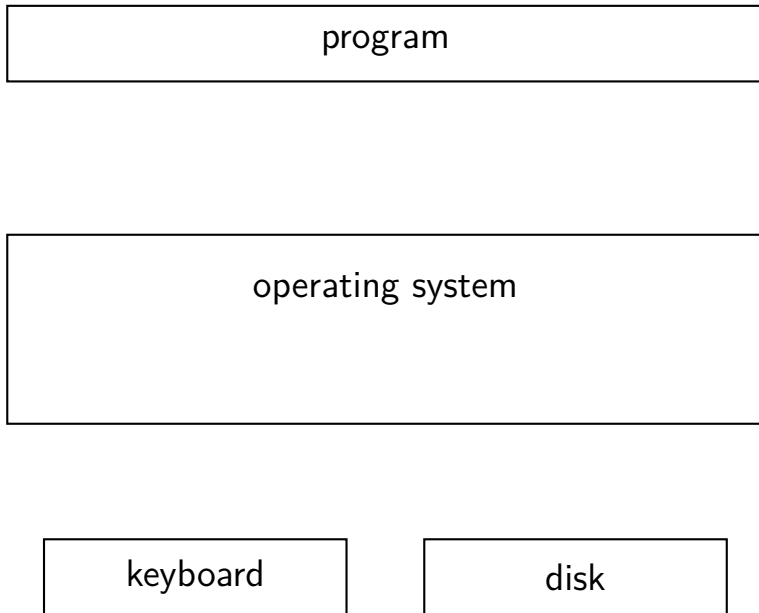
child process stays around as a “zombie”

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

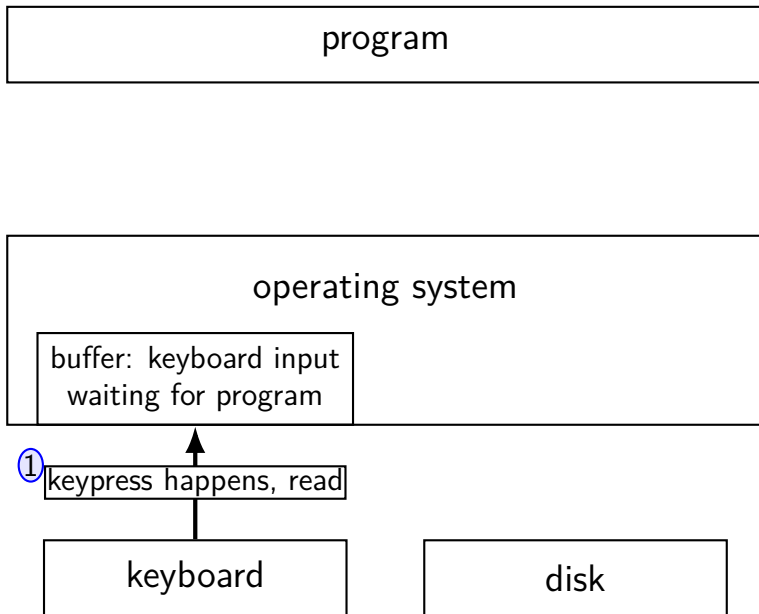
what if non-parent tries to `waitpid()` for child?

`waitpid` fails

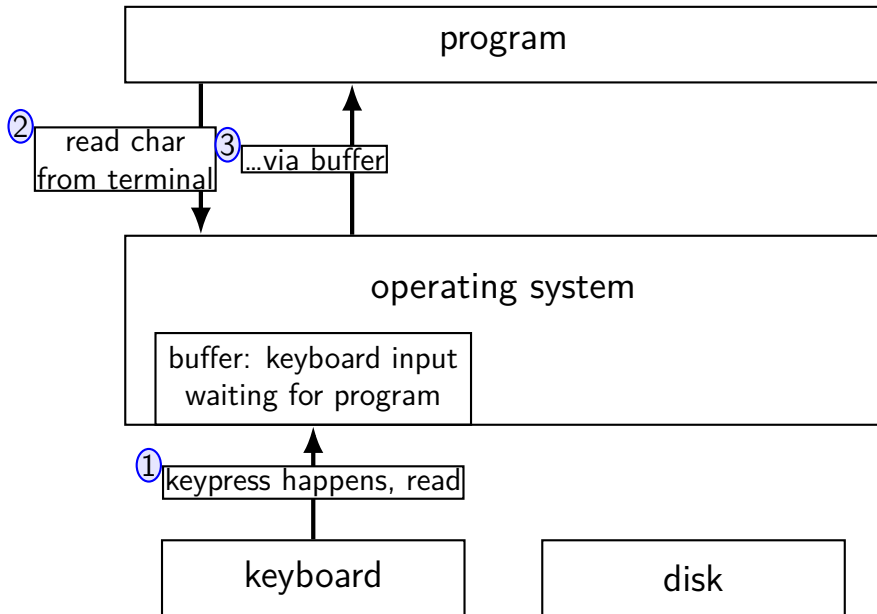
kernel buffering (reads)



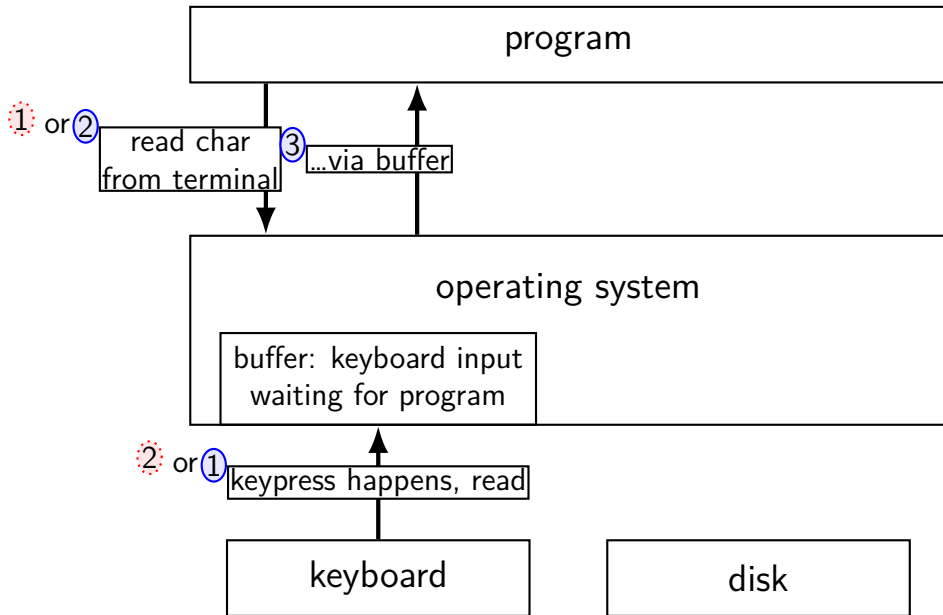
kernel buffering (reads)



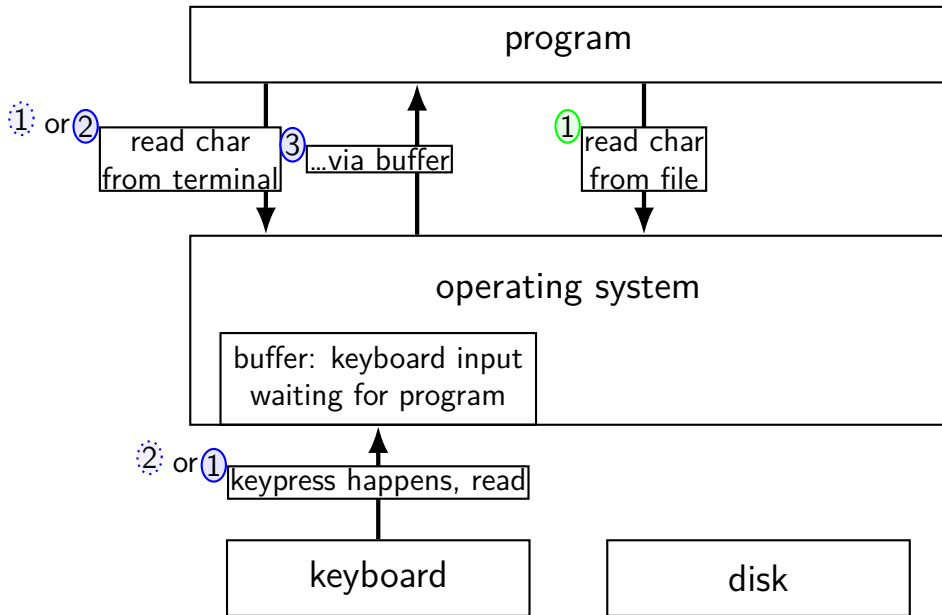
kernel buffering (reads)



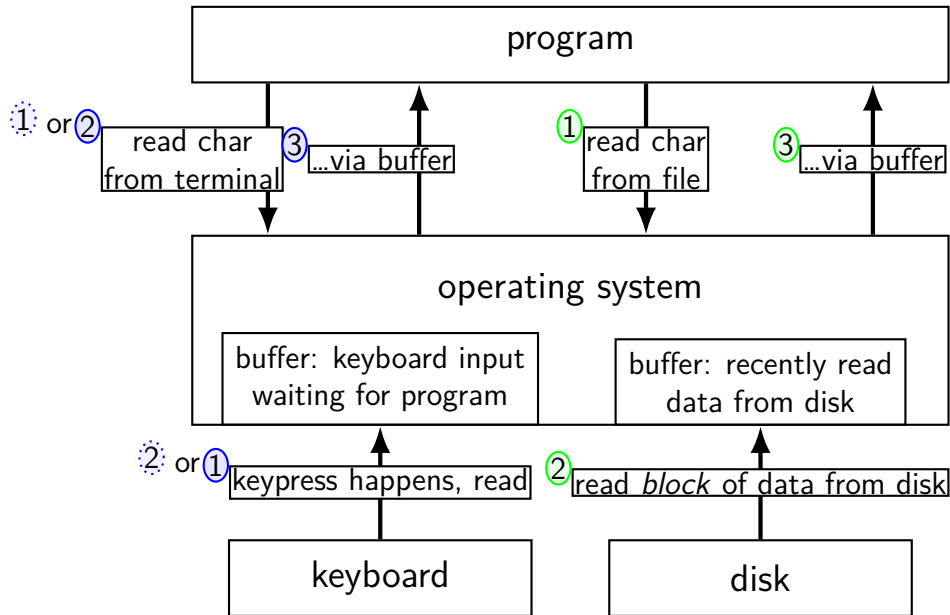
kernel buffering (reads)



kernel buffering (reads)



kernel buffering (reads)



kernel buffering (writes)

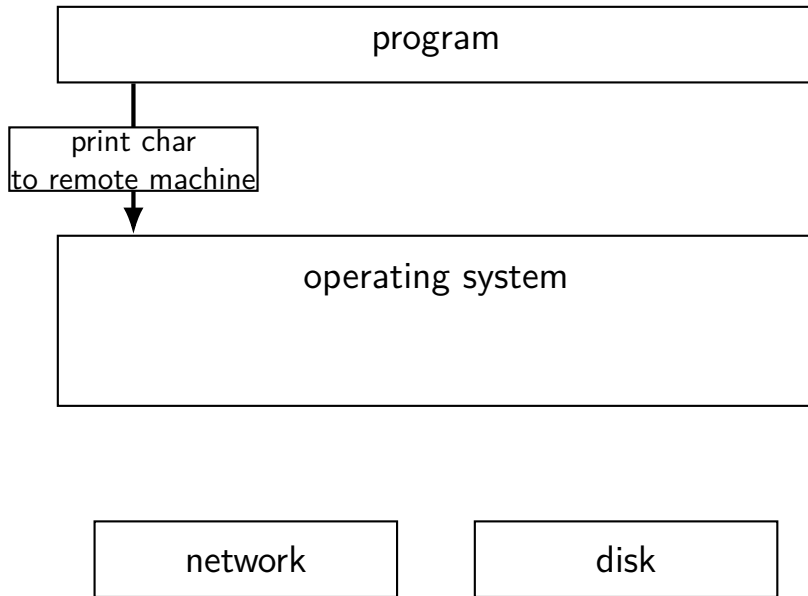
program

operating system

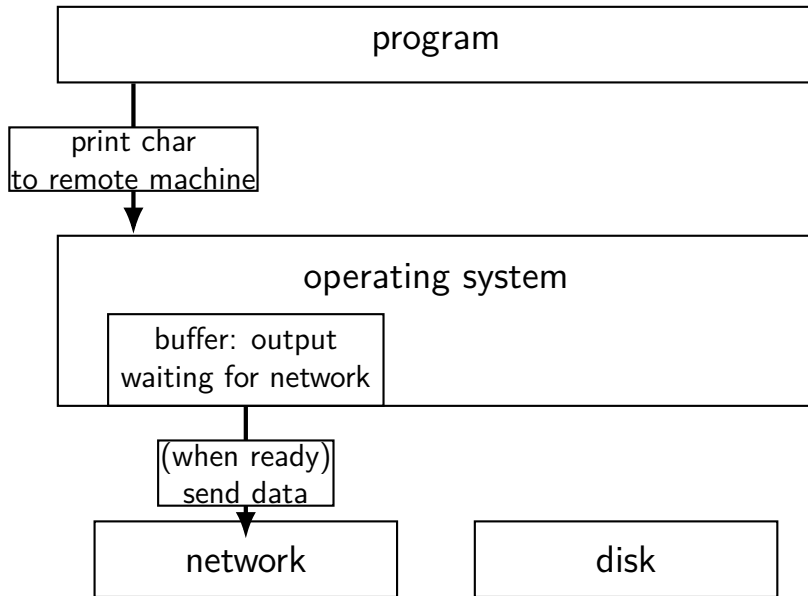
network

disk

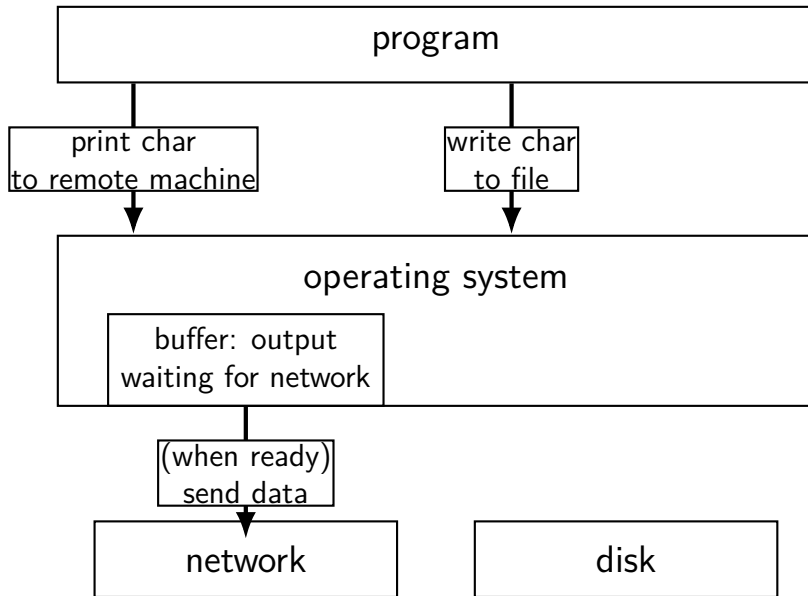
kernel buffering (writes)



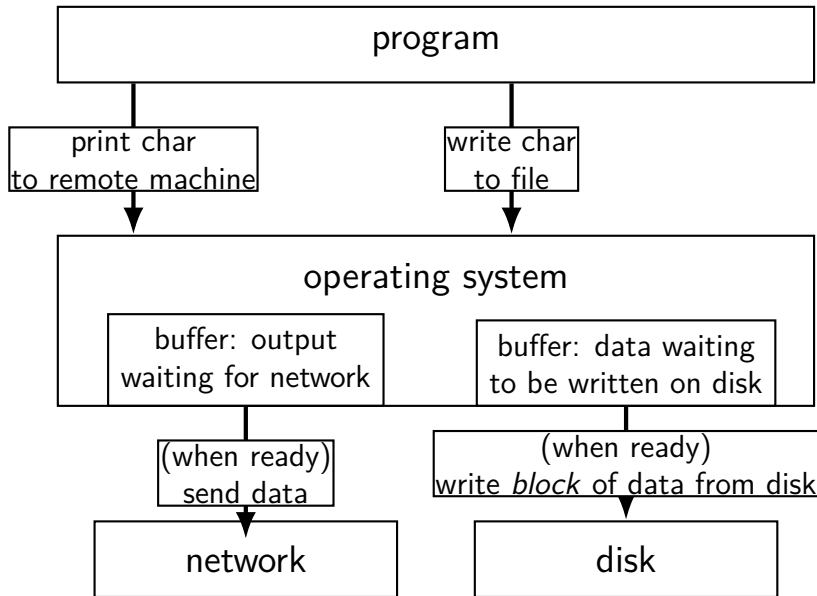
kernel buffering (writes)



kernel buffering (writes)



kernel buffering (writes)



read/write operations

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

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

actual I/O operations — wait for device to be ready
trigger process to stop waiting if needed

filesystem abstraction

regular files — named collection of bytes

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

directories — folders containing files and directories

hierarchical naming: `/net/zf14/cr4bd/fall2018/cs4414`

mostly contains regular files or directories

open

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

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

open

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

path = filename

e.g. `"/foo/bar/file.txt"`

file.txt in

directory bar in

directory foo in

"the root directory"

e.g. `"quux/other.txt"`

other.txt in

directory quux in

"the current working directory" (set with `chdir()`)

open: file descriptors

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

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

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

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

used by system calls that deal with open files

POSIX: everything is a file

the file: one interface for

- devices (terminals, printers, ...)

- regular files on disk

- networking (sockets)

- local interprocess communication (pipes, sockets)

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

exercise

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

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

- A. 0123456789 B. 0 C. (nothing)
D. A and B E. A and C F. A, B, and C

partial reads

read returning 0 always means end-of-file

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

read can return less than requested if not available
e.g. child hasn't gotten far enough

pipe: closing?

if all write ends of pipe are closed

can get end-of-file (`read()` returning 0) on read end

`exit()`ing closes them

→ close write end when not using

generally: limited number of file descriptors per process

→ good habit to close file descriptors not being used

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