last time

TLB — cache for page table entries

- lookup by virtual page number
- replace whole page table lookup — store last-level (final) result
- one page table entry per block
- divide virtual page number into tag and index

fork

- create new process by copying current process
- both processes run starting from fork() call
- different pids, return values from fork()

exec

- load different program into current process
- if successful, current process's memory discarded

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`

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 variab
            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

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

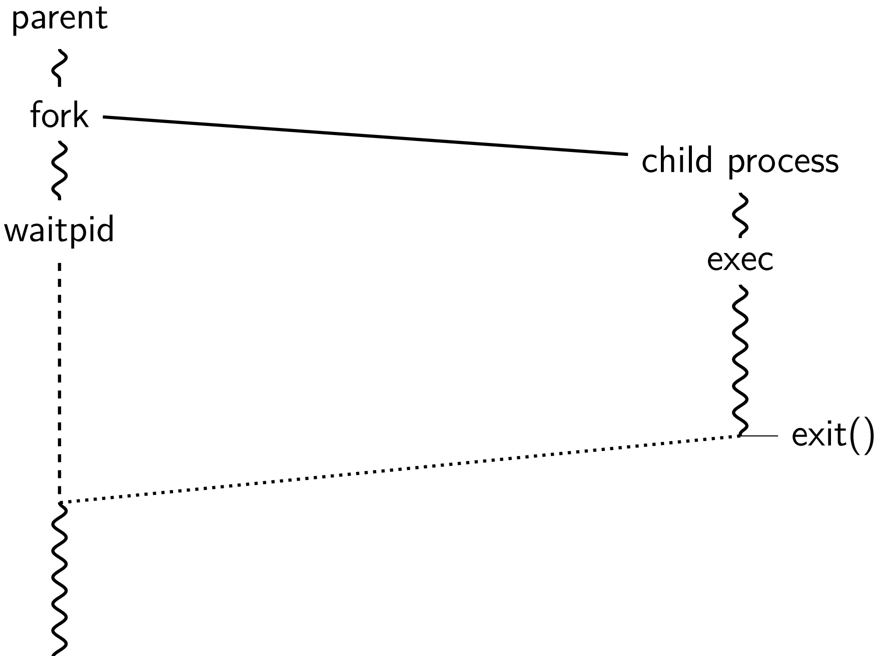
they are also how abnormal termination happens

- kernel communicating "something bad happened" → kills program by default

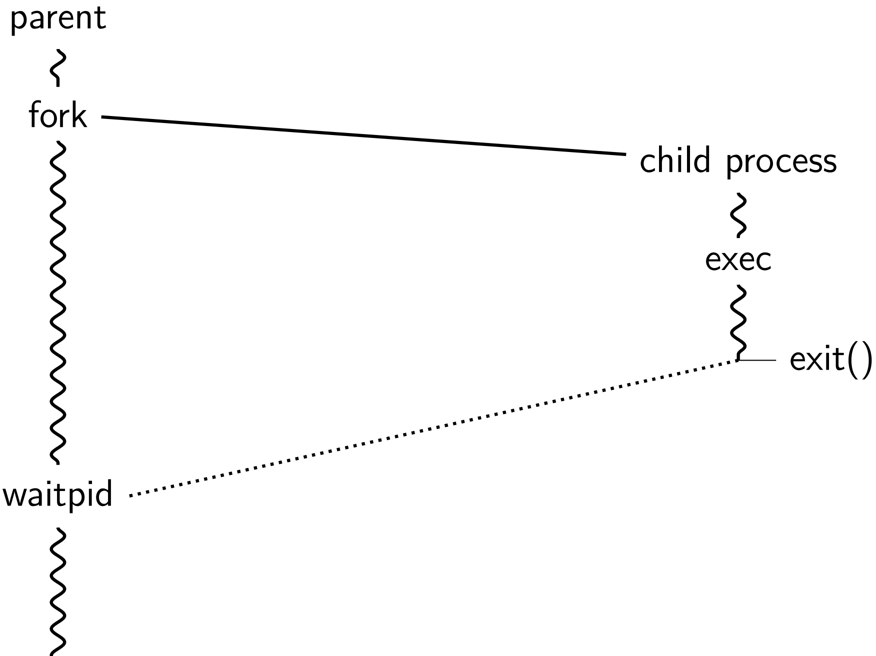
wait's status will tell you when and what signal killed a program

- constants in signal.h
- SIGINT — control-C
- SIGTERM — `kill` command (by default)
- SIGSEGV — segmentation fault
- SIGBUS — bus error
- SIGABRT — `abort()` library function
- …

typical pattern



typical pattern (alt)



typical pattern (detail)

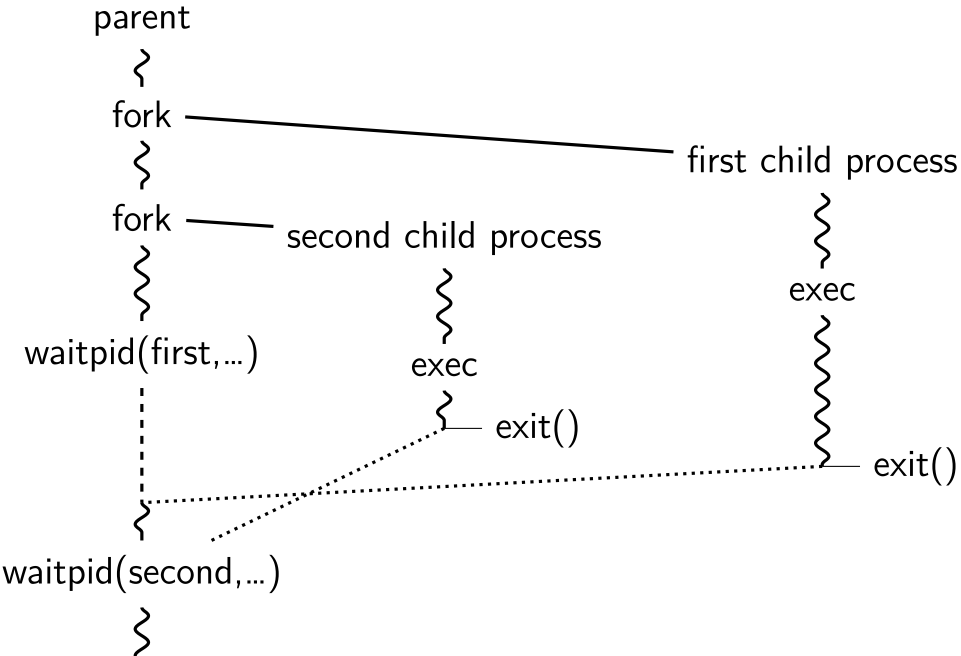
```
pid = fork();
if (pid == 0) {
    exec…(…);
    …
} else if (pid > 0) {
    waitpid(pid,…);
    …
}
…
```

```
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		**E.**	A, B, and C
B.	0	(newline)	1	(newline)	0	(newline)	2		**F.**	C and D
C.	1	(newline)	0	(newline)	0	(newline)	2		**G.**	all of the above
D.	1	(newline)	0	(newline)	2	(newline)	0		**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

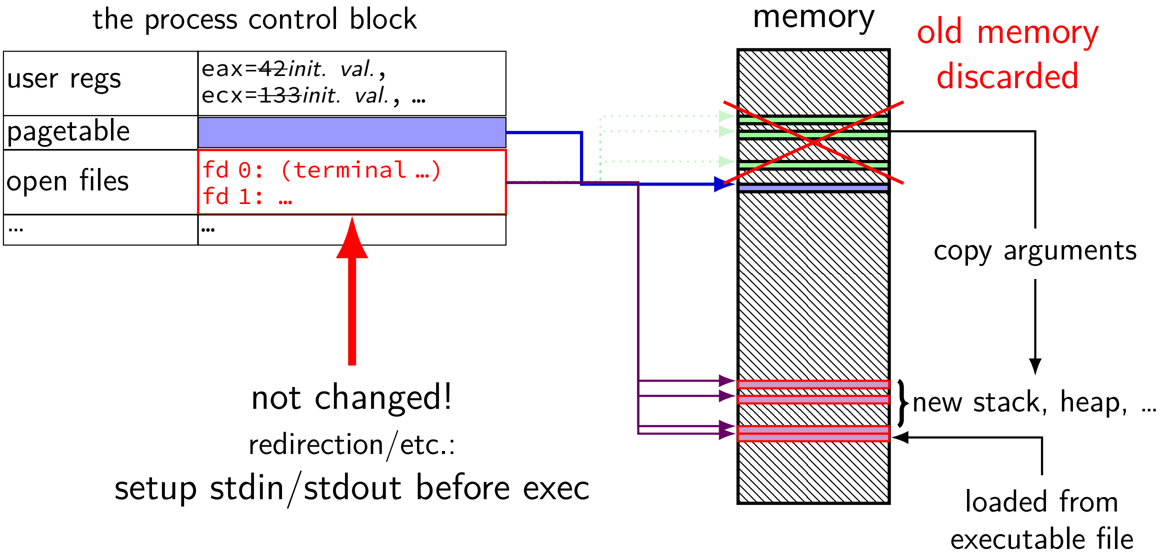
`./my_program ... < input.txt`:

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

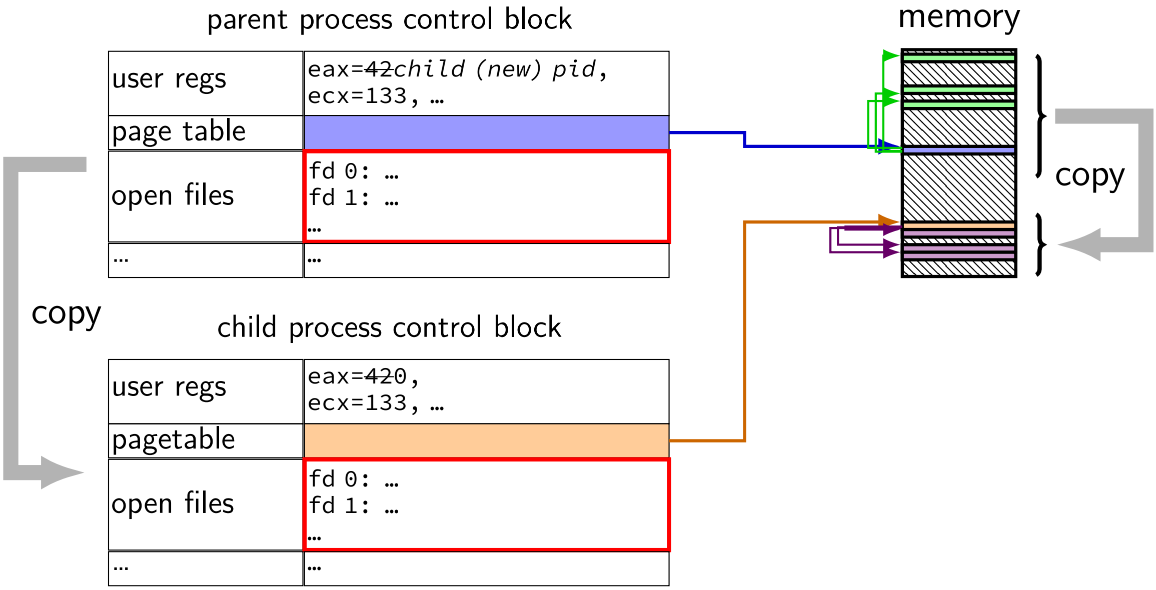
`echo foo > output.txt`:

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

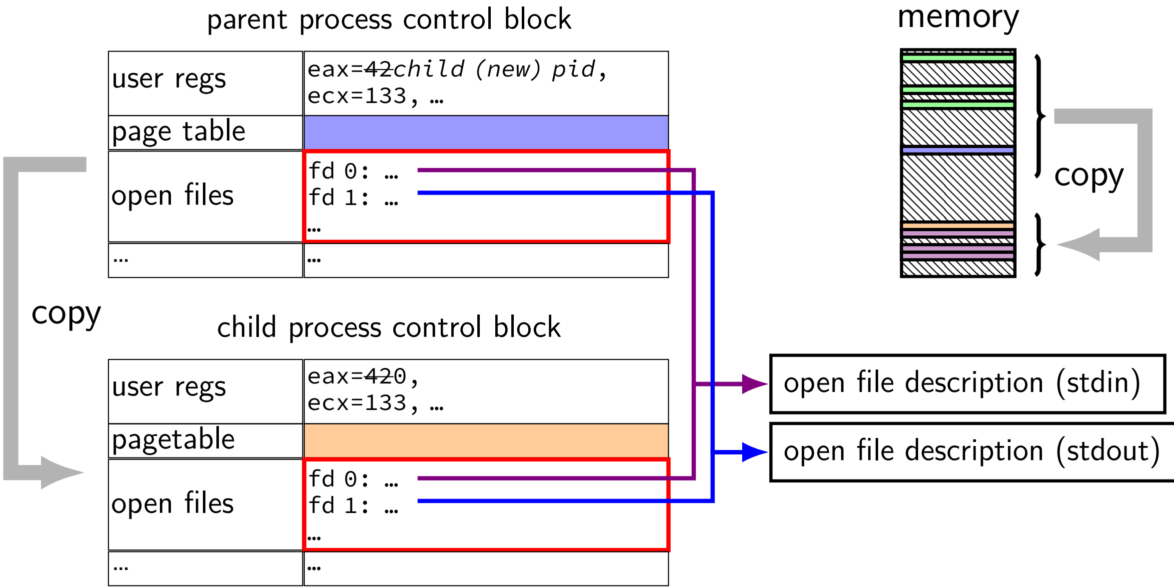
exec preserves open files



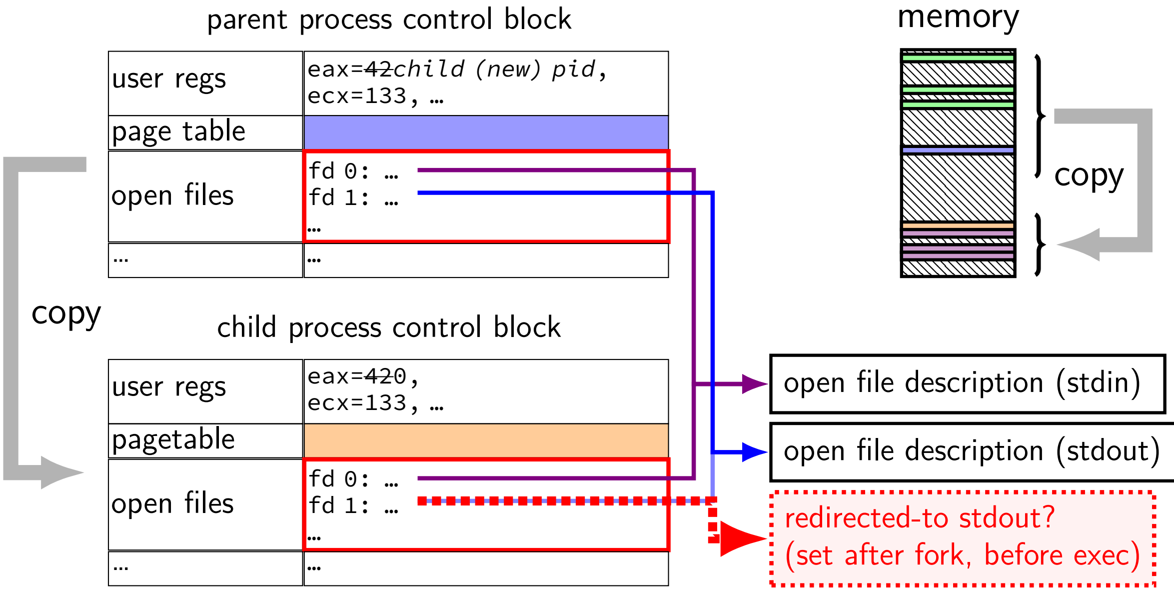
fork copies open file list



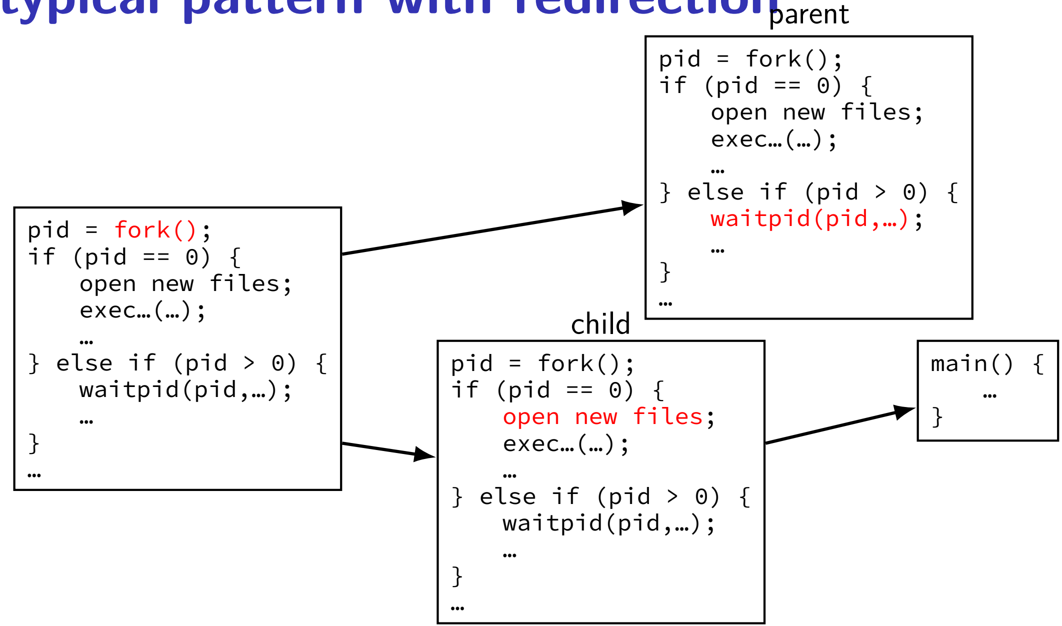
fork copies open file list



fork copies open file list



typical pattern with redirection



redirecting with exec

standard output/error/input are files

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

(probably after forking) open files to redirect

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

using `dup2()` library call

then exec, preserving new standard output/etc.

reassigning file descriptors

redirection: `./program >output.txt`

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

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

reassigning and file table

```
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 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);
```

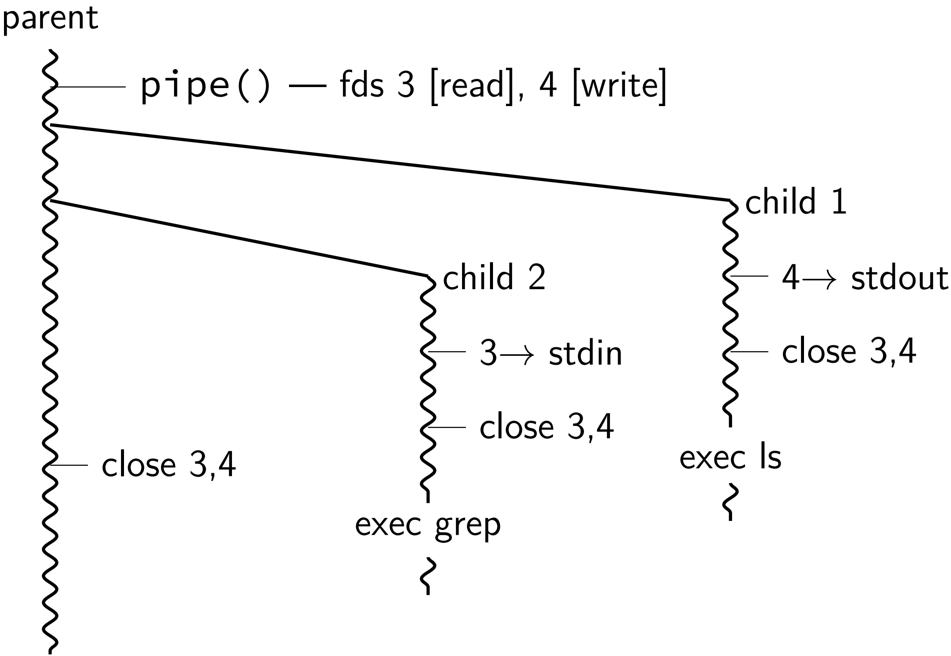
This is likely to not terminate. What's the problem?

pipe and pipelines

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

```
pipe(pipe_fd);
ls_pid = fork();
if (ls_pid == 0) {
    dup2(pipe_fd[1], STDOUT_FILENO);
    close(pipe_fd[0]); close(pipe_fd[1]);
    char *argv[] = {"ls", "-1", NULL};
    execv("/bin/ls", argv);
}
grep_pid = fork();
if (grep_pid == 0) {
    dup2(pipe_fd[0], STDIN_FILENO);
    close(pipe_fd[0]); close(pipe_fd[1]);
    char *argv[] = {"grep", "foo", NULL};
    execv("/bin/grep", argv);
}
close(pipe_fd[0]); close(pipe_fd[1]);
/* wait for processes, etc. */
```

example execution



why threads?

concurrency: different things happening at once

- one thread per user of web server?
- one thread per page in web browser?
- one thread to play audio, one to read keyboard, ...?
- ...

parallelism: do same thing with more resources

- multiple processors to speed-up simulation (life assignment)

aside: alternate threading models

we'll talk about kernel threads

OS scheduler deals **directly** with threads

alternate idea: library code handles threads

kernel doesn't know about threads w/in process

hierarchy of schedulers: one for processes, one within each process

not currently common model — awkward with multicore

thread versus process state

thread state

- registers (including stack pointer, program counter)
- ...

process state

- address space
- open files
- process id
- list of thread states
- ...

process info with threads

parent process info

thread infos	`thread 0: {PC = 0x123456, rax = 42, rbx = …}`
	`thread 1: {PC = 0x584390, rax = 32, rbx = …}`
	…

page tables	
open files	`fd 0: …` `fd 1: …`
…	…

Linux idea: task_struct

Linux model: single "task" structure = thread

pointers to address space, open file list, etc.

pointers can be shared

- e.g. shared open files: open fd 4 in one task → all sharing can use fd 4

`fork()`-like system call "clone": choose what to share

- `clone(0, ...)` — similar to `fork()`
- `clone(CLONE_FILES, ...)` — like fork(), but **sharing** open files
- `clone(CLONE_VM, new_stack_pointer, ...)` — like fork(), but **sharing** address space

Linux idea: task_struct

Linux model: single "task" structure = thread

pointers to address space, open file list, etc.

pointers can be shared

- e.g. shared open files: open fd 4 in one task → all sharing can use fd 4

`fork()`-like system call "clone": choose what to share

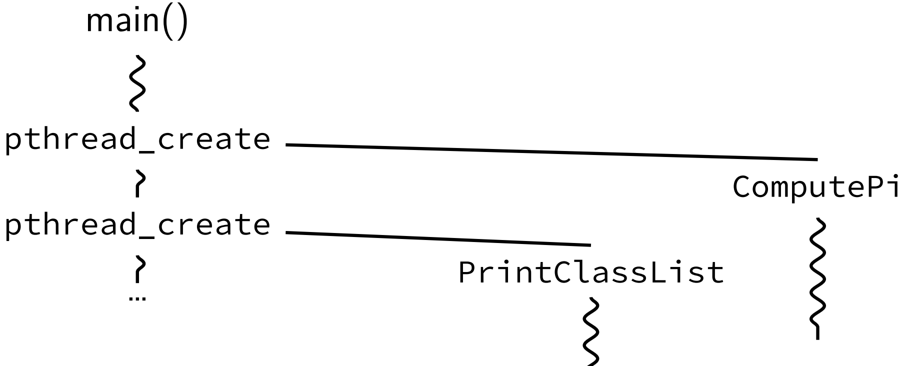
- `clone(0, ...)` — similar to `fork()`
- `clone(CLONE_FILES, ...)` — like fork(), but **sharing** open files
- `clone(CLONE_VM, new_stack_pointer, ...)` — like fork(), but **sharing** address space

advantage: no special logic for threads (mostly)

- two threads in same process = tasks sharing everything possible

pthread_create

```
void *ComputePi(void *argument) { ...  }
void *PrintClassList(void *argument) { ...  }
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```



pthread_create

```
void *ComputePi(void *argument) { ...  }
void *PrintClassList(void *argument) { ...  }
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```

pthread_create arguments:

thread identifier

function to run

thread starts here, terminates if this function returns

thread attributes (extra settings) and function argument

pthread_create

```
void *ComputePi(void *argument) { ...  }
void *PrintClassList(void *argument) { ...  }
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```

pthread_create arguments:

thread identifier

function to run

thread starts here, terminates if this function returns

thread attributes (extra settings) and function argument

pthread_create

```
void *ComputePi(void *argument) { ...  }
void *PrintClassList(void *argument) { ...  }
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```

pthread_create arguments:

thread identifier

function to run

thread starts here, terminates if this function returns

thread attributes (extra settings) and function argument

pthread_create

```
void *ComputePi(void *argument) { ...  }
void *PrintClassList(void *argument) { ...  }
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```

pthread_create arguments:

thread identifier

function to run

- thread starts here, terminates if this function returns

thread attributes (extra settings) and function argument

a threading race

```
#include <pthread.h>
#include <stdio.h>
void *print_message(void *ignored_argument) {
    printf("In the thread\n"); return NULL;
}
int main() {
    printf("About to start thread\n");
    pthread_t the_thread;
    pthread_create(&the_thread, NULL, print_message, NULL);
    printf("Done starting thread\n");
    return 0;
}
```

My machine: outputs `In the thread` about 4% of the time.

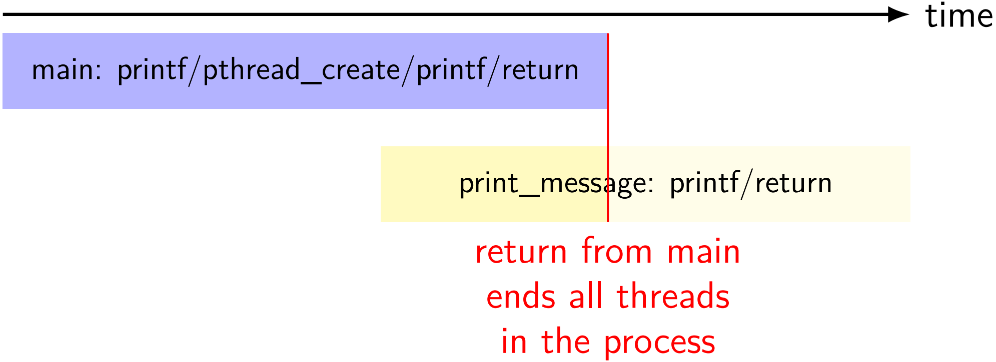
What happened?

a race

returning from main exits the entire process (all its threads)

same as calling exit; not like other threads

race: main's return 0 or print_message's printf first?



fixing the race (version 1)

```
#include <pthread.h>
#include <stdio.h>
void *print_message(void *ignored_argument) {
    printf("In the thread\n");
    return NULL;
}
int main() {
    printf("About to start thread\n");
    pthread_t the_thread;
    pthread_create(&the_thread, NULL, print_message, NULL);
    printf("Done starting thread\n");
    pthread_join(the_thread, NULL);  /* WAIT FOR THREAD */
    return 0;
}
```

fixing the race (version 2; not recommended)

```
#include <pthread.h>
#include <stdio.h>
void *print_message(void *ignored_argument) {
    printf("In the thread\n");
    return NULL;
}
int main() {
    printf("About to start thread\n");
    pthread_t the_thread;
    pthread_create(&the_thread, NULL, print_message, NULL);
    printf("Done starting thread\n");
    pthread_exit(NULL);
}
```

pthread_join, pthread_exit

`pthread_join`: wait for thread, retrieves its return value

- like `waitpid`, but for a thread
- return value is pointer to anything

`pthread_exit`: exit current thread, returning a value

- like `exit` or returning from main, but for a single thread
- same effect as returning from function passed to pthread_create

sum example (only globals)

```
int values[1024];
int results[2];
void *sum_front(void *ignored_argument) {
    int sum = 0;
    for (int i = 0; i < 512; ++i) { sum += values[i]; }
    results[0] = sum;
    return NULL;
}
void *sum_back(void *ignored_argument) {
    int sum = 0;
    for (int i = 512; i < 1024; ++i) { sum += values[i]; }
    results[1] = sum;
    return NULL;
}
int sum_all() {
    pthread_t sum_front_thread, sum_back_thread;
    pthread_create(&sum_front_thread, NULL, sum_front, NULL);
    pthread_create(&sum_back_thread, NULL, sum_back, NULL);
    pthread_join(sum_front_thread, NULL); pthread_join(sum_back_thread, NULL);
    return results[0] + results[1];
}
```

sum example (only globals)

values, results: global variables — shared

```
int values[1024];
int results[2];
void *sum_front(void *ignored_argument) {
    int sum = 0;
    for (int i = 0; i < 512; ++i) { sum += values[i]; }
    results[0] = sum;
    return NULL;
}
void *sum_back(void *ignored_argument) {
    int sum = 0;
    for (int i = 512; i < 1024; ++i) { sum += values[i]; }
    results[1] = sum;
    return NULL;
}
int sum_all() {
    pthread_t sum_front_thread, sum_back_thread;
    pthread_create(&sum_front_thread, NULL, sum_front, NULL);
    pthread_create(&sum_back_thread, NULL, sum_back, NULL);
    pthread_join(sum_front_thread, NULL); pthread_join(sum_back_thread, NULL);
    return results[0] + results[1];
}
```

sum example (only globals)

two different functions
happen to be the same except for some numbers

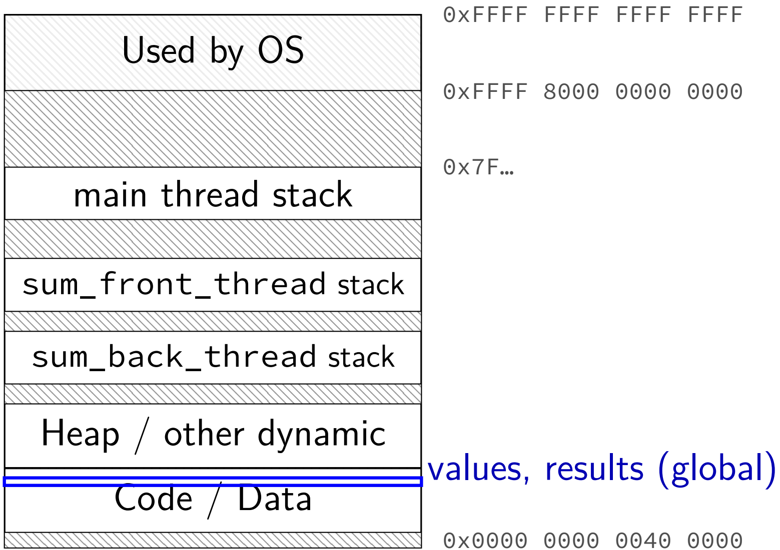
```
int values[1024];
int results[2];
void *sum_front(void *ignored_argument) {
    int sum = 0;
    for (int i = 0; i < 512; ++i) { sum += values[i]; }
    results[0] = sum;
    return NULL;
}
void *sum_back(void *ignored_argument) {
    int sum = 0;
    for (int i = 512; i < 1024; ++i) { sum += values[i]; }
    results[1] = sum;
    return NULL;
}
int sum_all() {
    pthread_t sum_front_thread, sum_back_thread;
    pthread_create(&sum_front_thread, NULL, sum_front, NULL);
    pthread_create(&sum_back_thread, NULL, sum_back, NULL);
    pthread_join(sum_front_thread, NULL); pthread_join(sum_back_thread, NULL);
    return results[0] + results[1];
}
```

sum e

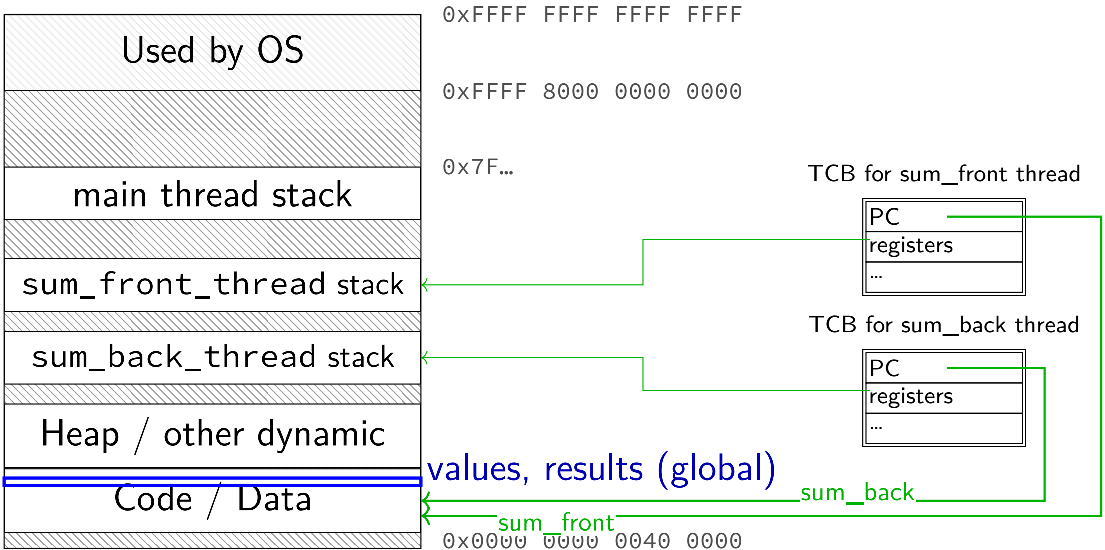
values returned from threads
via global array instead of return value
(partly to illustrate that memory is shared,
partly because this pattern works when we don't join (later))

```
int valu
int resu
void *su
    int
    for (int i = 0; i < 512; ++i) { sum += values[i]; }
    results[0] = sum;
    return NULL;
}
void *sum_back(void *ignored_argument) {
    int sum = 0;
    for (int i = 512; i < 1024; ++i) { sum += values[i]; }
    results[1] = sum;
    return NULL;
}
int sum_all() {
    pthread_t sum_front_thread, sum_back_thread;
    pthread_create(&sum_front_thread, NULL, sum_front, NULL);
    pthread_create(&sum_back_thread, NULL, sum_back, NULL);
    pthread_join(sum_front_thread, NULL); pthread_join(sum_back_thread, NULL);
    return results[0] + results[1];
}
```

thread_sum memory layout



thread_sum memory layout



sum example (to global, with thread IDs)

```
int values[1024];
int results[2];
void *sum_thread(void *argument) {
    int id = (int) argument;
    int sum = 0;
    for (int i = id * 512; i < (id + 1) * 512; ++i) {
        sum += values[i];
    }
    results[id] = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        pthread_create(&threads[i], NULL, sum_thread, (void *) i);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return results[0] + results[1];
}
```

sum example (to global, with thread IDs)

```
int values[1024];
int results[2];
void *sum_thread(void *argument) {
    int id = (int) argument;
    int sum = 0;
    for (int i = id * 512; i < (id + 1) * 512; ++i) {
        sum += values[i];
    }
    results[id] = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        pthread_create(&threads[i], NULL, sum_thread, (void *) i);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return results[0] + results[1];
}
```

values, results: global variables — shared

sum example (info struct)

```
int values[1024];
struct ThreadInfo {
    int start, end, result;
};
void *sum_thread(void *argument) {
    ThreadInfo *my_info =  (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) { sum += values[i]; }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i) { pthread_join(threads[i], NULL); }
    return info[0].result + info[1].result;
}
```

sum example (info struct)

```
int values[1024];
struct ThreadInfo
    int start, end, result;
};
void *sum_thread(void *argument) {
    ThreadInfo *my_info =  (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) { sum += values[i]; }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i) { pthread_join(threads[i], NULL); }
    return info[0].result + info[1].result;
}
```

values: global variable — shared

sum example (info struct)

```
int values[1024];
struct ThreadInfo {
    int start, end, result;
};
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->                                  lues[i]; }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i) { pthread_join(threads[i], NULL); }
    return info[0].result + info[1].result;
}
```

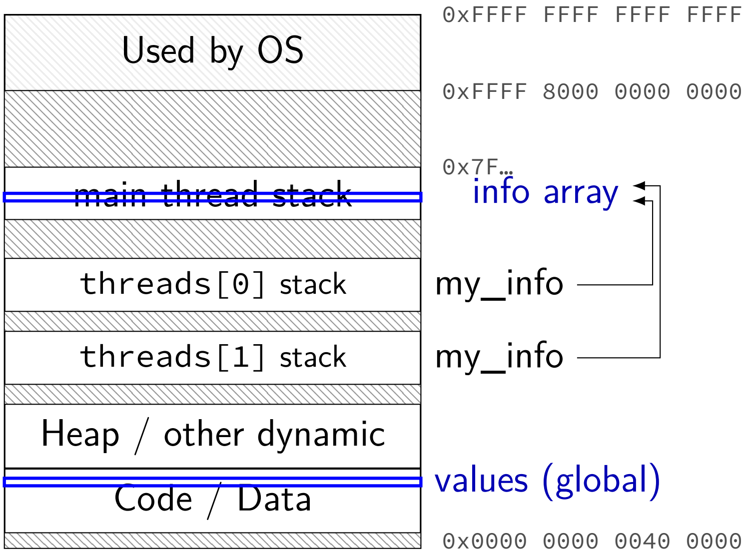
my_info: pointer to sum_all's stack

only okay because sum_all waits!

sum example (info struct)

```
int values[1024];
struct ThreadInfo {
    int start, end, result;
};
void *sum_thread(void *argument) {
    ThreadInfo *my_info =  (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) { sum += values[i]; }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i) { pthread_join(threads[i], NULL); }
    return info[0].result + info[1].result;
}
```

thread_sum memory layout (info struct)



sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

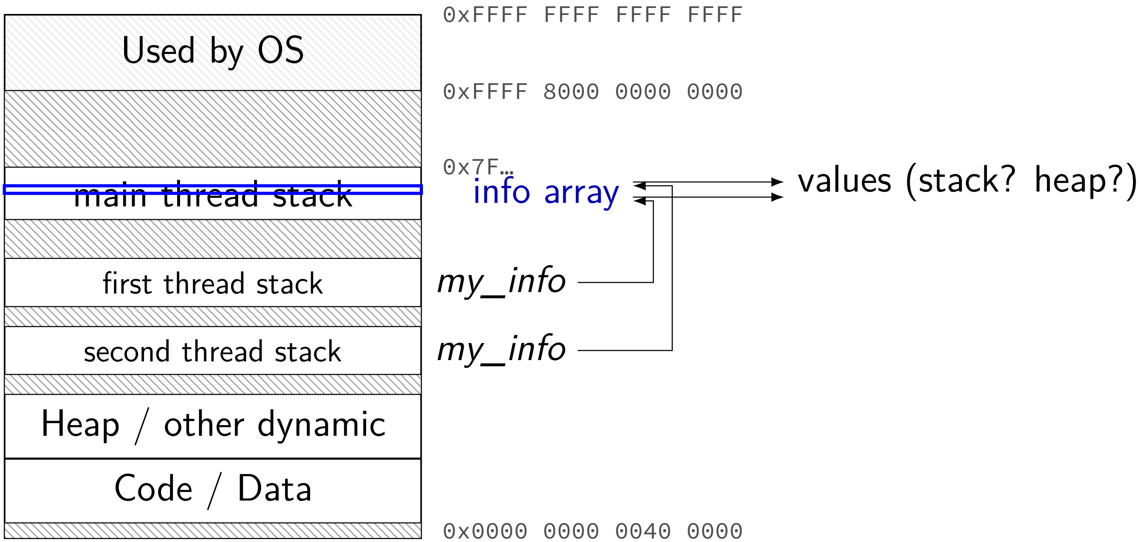
sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

program memory (to main stack)



sum example (on heap)

```
struct ThreadInfo { pthread_t thread; int *values; int start; int end; int result
void *sum_thread(void *argument) {
    ...
}

ThreadInfo *start_sum_all(int *values) {
    ThreadInfo *info = new ThreadInfo[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&info[i].thread, NULL, sum_thread, (void *) &info[i]);
    }
    return info;
}

int finish_sum_all(ThreadInfo *info) {
    for (int i = 0; i < 2; ++i)
        pthread_join(info[i].thread, NULL);
    int result = info[0].result + info[1].result;
    delete[] info;
    return result;
}
```

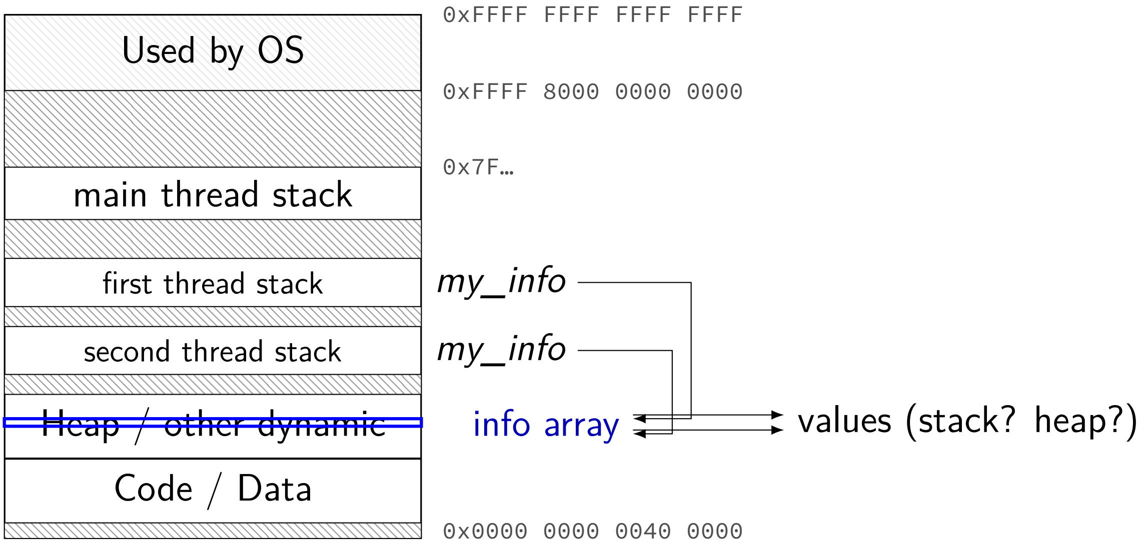
sum example (on heap)

```
struct ThreadInfo { pthread_t thread; int *values; int start; int end; int result
void *sum_thread(void *argument) {
    ...
}

ThreadInfo *start_sum_all(int *values) {
    ThreadInfo *info = new ThreadInfo[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&info[i].thread, NULL, sum_thread, (void *) &info[i]);
    }
    return info;
}

int finish_sum_all(ThreadInfo *info) {
    for (int i = 0; i < 2; ++i)
        pthread_join(info[i].thread, NULL);
    int result = info[0].result + info[1].result;
    delete[] info;
    return result;
}
```

sum example (on heap)

```
struct ThreadInfo { pthread_t thread; int *values; int start; int end; int result
void *sum_thread(void *argument) {
    ...
}

ThreadInfo *start_sum_all(int *values) {
    ThreadInfo *info = new ThreadInfo[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&info[i].thread, NULL, sum_thread, (void *) &info[i]);
    }
    return info;
}

int finish_sum_all(ThreadInfo *info) {
    for (int i = 0; i < 2; ++i)
        pthread_join(info[i].thread, NULL);
    int result = info[0].result + info[1].result;
    delete[] info;
    return result;
}
```

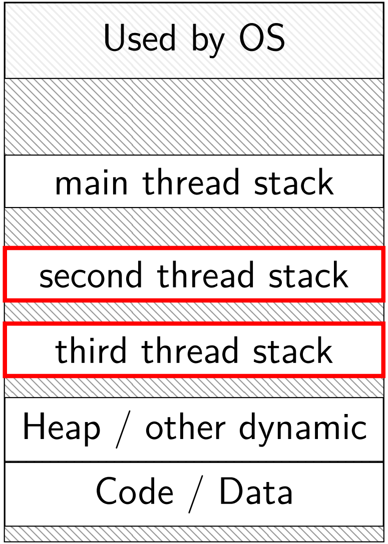
thread_sum memory (heap version)



what’s wrong with this?

```
/* omitted: headers */
#include <string>
using std::string;
void *create_string(void *ignored_argument) {
  string result;
  result = ComputeString();
  return &result;
}
int main() {
  pthread_t the_thread;
  pthread_create(&the_thread, NULL, create_string, NULL);
  string *string_ptr;
  pthread_join(the_thread, (void*) &string_ptr);
  cout << "string is " << *string_ptr;
}
```

program memory



0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

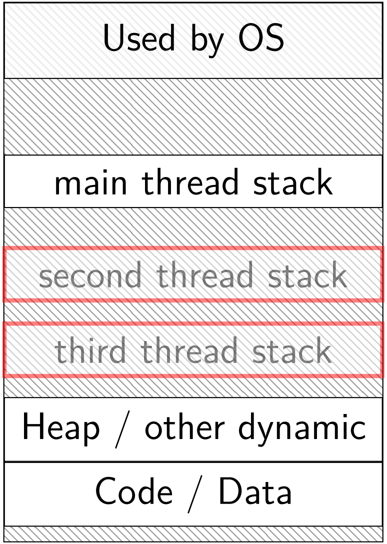
0x7F...

dynamically allocated stacks
string result allocated here
string_ptr pointed to here

...stacks deallocated when threads exit/are joined

0x0000 0000 0040 0000

program memory



`0xFFFF FFFF FFFF FFFF`

`0xFFFF 8000 0000 0000`

`0x7F`…

dynamically allocated stacks
`string result` allocated here
`string_ptr` pointed to here

…stacks deallocated when threads exit/are joined

`0x0000 0000 0040 0000`

thread resources

to create a thread, allocate:

new stack (how big???)

thread control block

deallocated when …

thread resources

to create a thread, allocate:

new stack (how big???)

thread control block

deallocated when ...

can deallocate stack when thread exits

but need to allow collecting return value

same problem as for processes and waitpid

pthread_detach

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_create(&show_progress_thread, NULL,
                   show_progress, NULL);

    /* instead of keeping pthread_t around to join thread later: */
    pthread_detach(show_progress_thread);
}

int main() {
    spawn_show_progress_thread();
    do_other_stuff();
    ...
}
```

detach = don't care about return value, etc.
system will deallocate when thread terminates

starting threads detached

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_attr_t attrs;
    pthread_attr_init(&attrs);
    pthread_attr_setdetachstate(&attrs, PTHREAD_CREATE_DETACHED);
    pthread_create(&show_progress_thread, attrs,
                   show_progress, NULL);
    pthread_attr_destroy(&attrs);
}
```

setting stack sizes

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_attr_t attrs;
    pthread_attr_init(&attrs);
    pthread_attr_setstacksize(&attrs, 32 * 1024 /* bytes */);
    pthread_create(&show_progress_thread, attrs,
                   show_progress, NULL);
}
```

a note on error checking

from pthread_create manpage:

ERRORS

EAGAIN Insufficient resources to create another thread, or a system-imposed limit on the number of threads was encountered. The latter case may occur in two ways: the **RLIMIT_NPROC** soft resource limit (set via **setrlimit**(2)), which limits the number of process for a real user ID, was reached; or the kernel's system-wide limit on the number of threads, /proc/sys/kernel/threads-max, was reached.

EINVAL Invalid settings in attr.

EPERM No permission to set the scheduling policy and parameters specified in attr.

special constants for *return value*

same pattern for many other pthreads functions

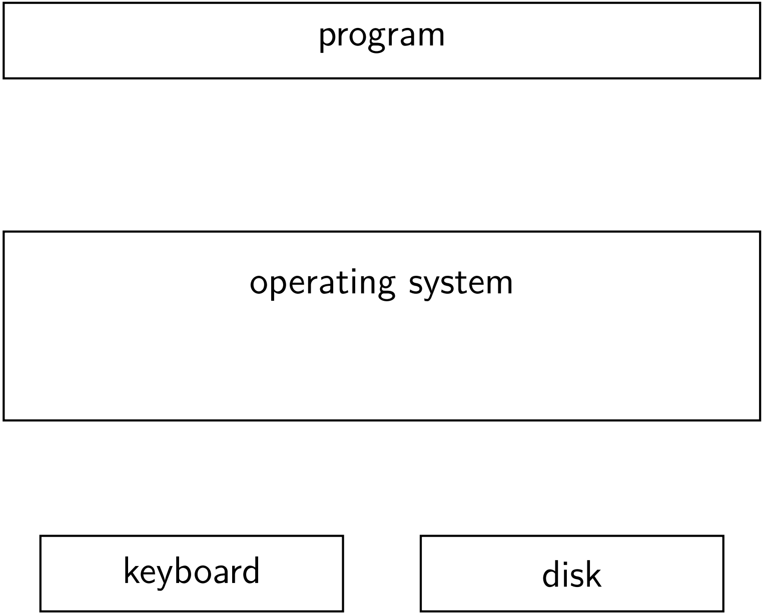
will often omit error checking in slides for brevity

error checking pthread_create

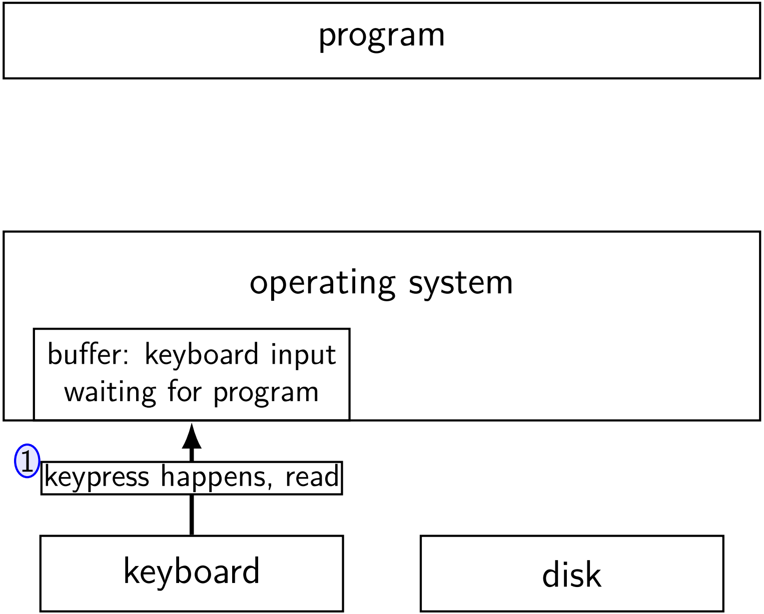
```
int error = pthread_create(...);
if (error != 0) {
    /* print some error message */
}
```

backup slides

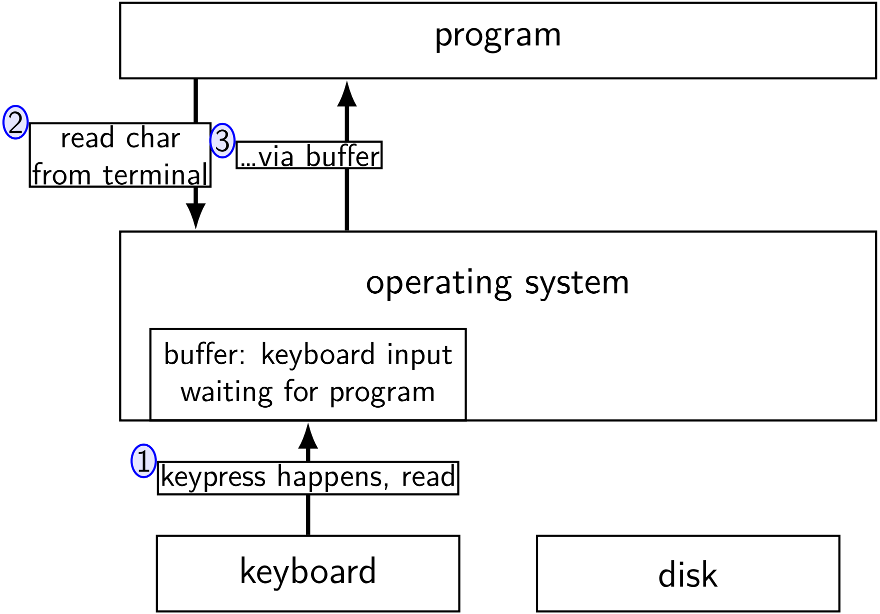
kernel buffering (reads)



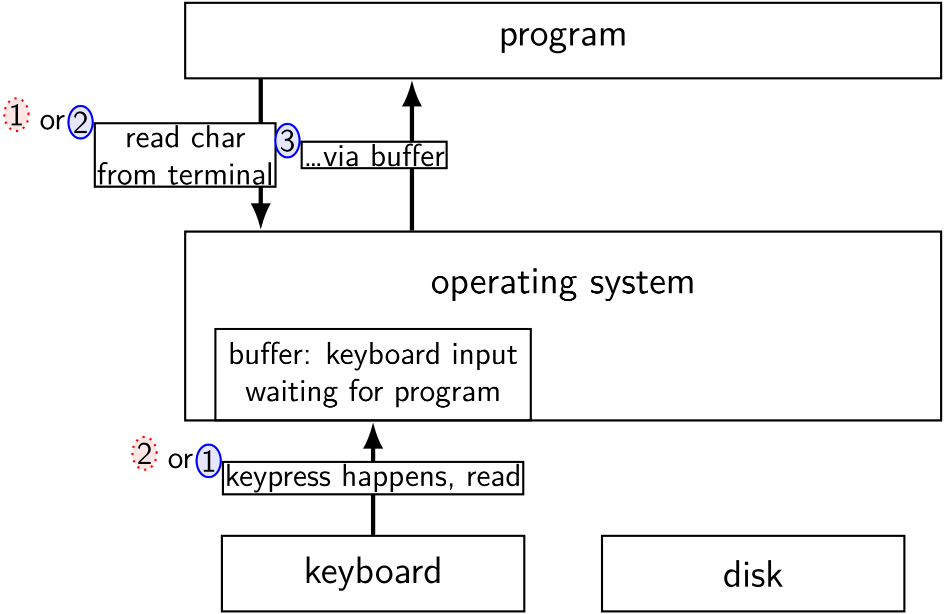
kernel buffering (reads)



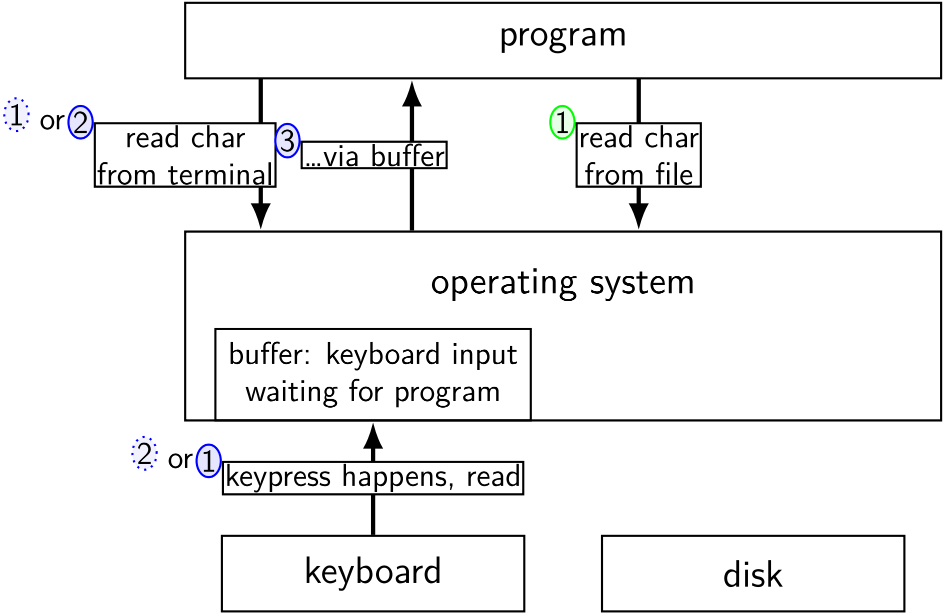
kernel buffering (reads)



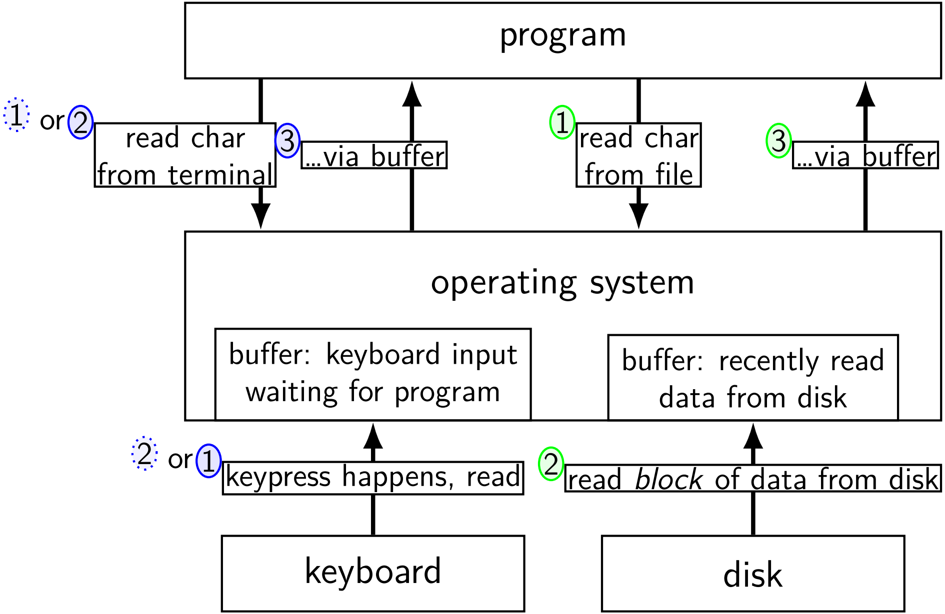
kernel buffering (reads)



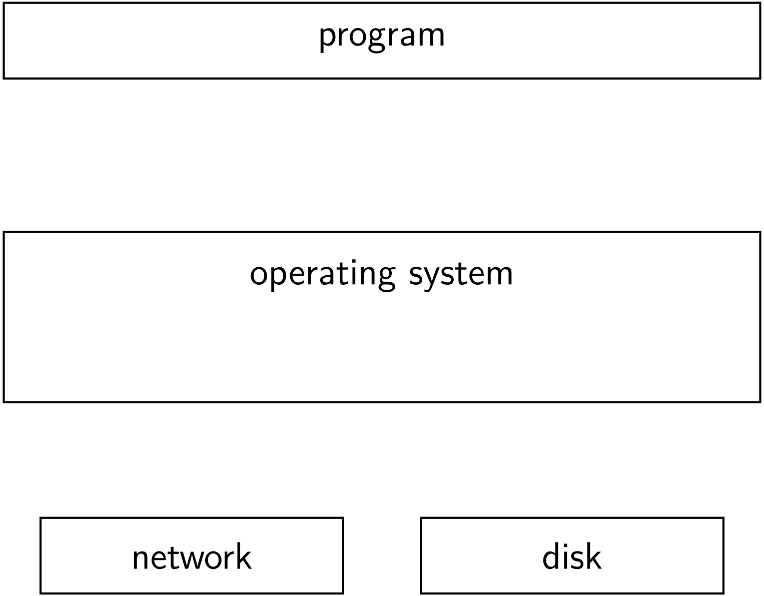
kernel buffering (reads)



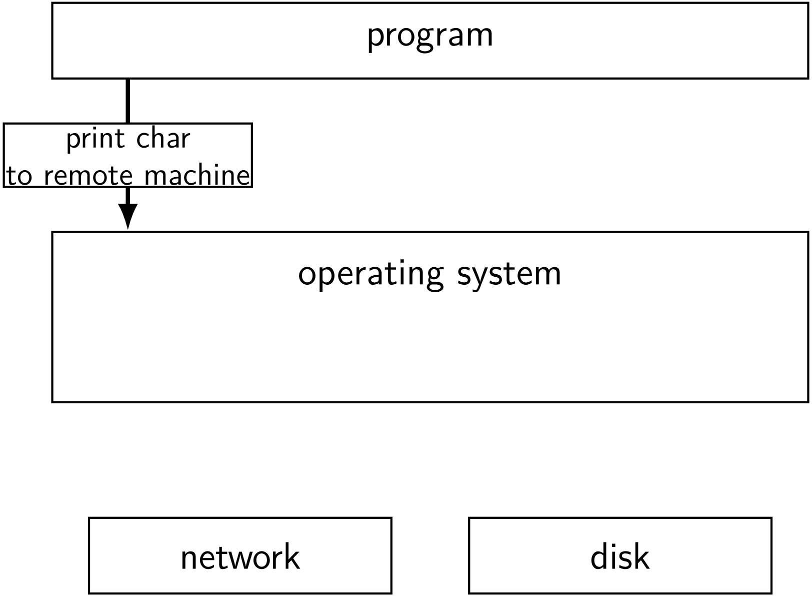
kernel buffering (reads)



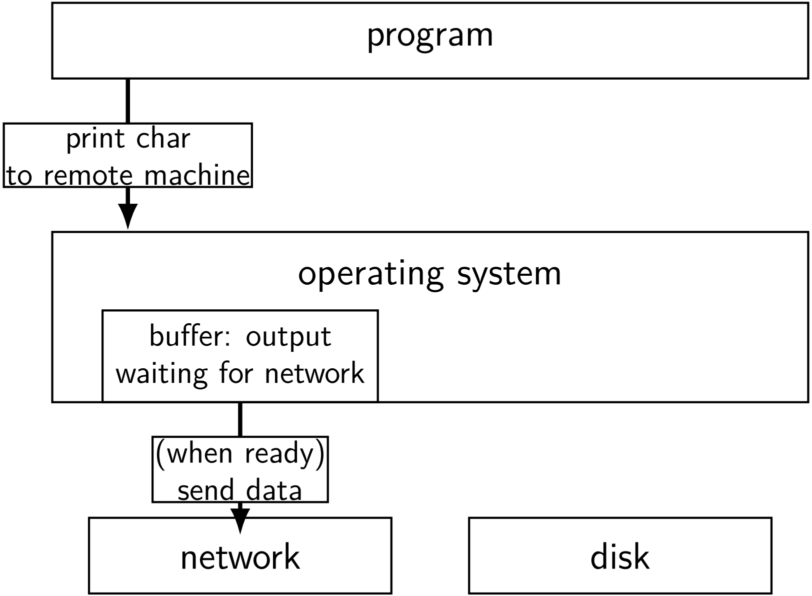
kernel buffering (writes)



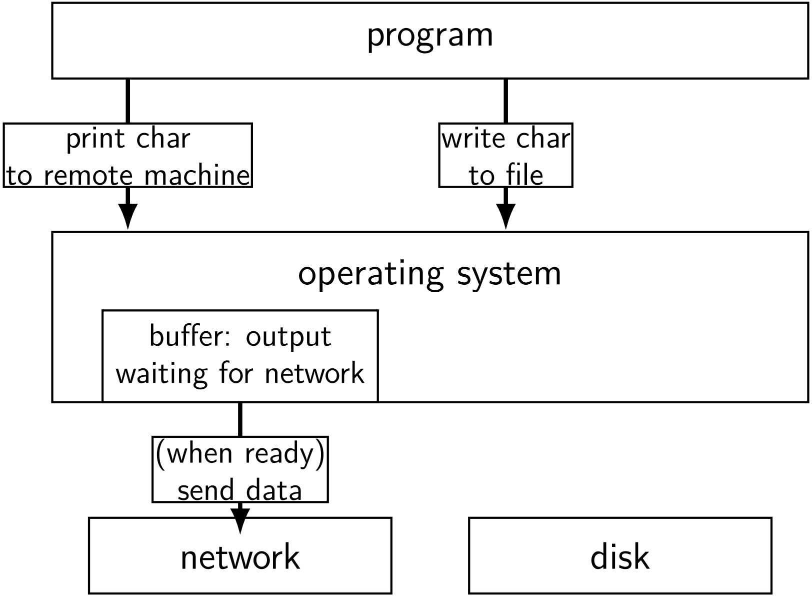
kernel buffering (writes)



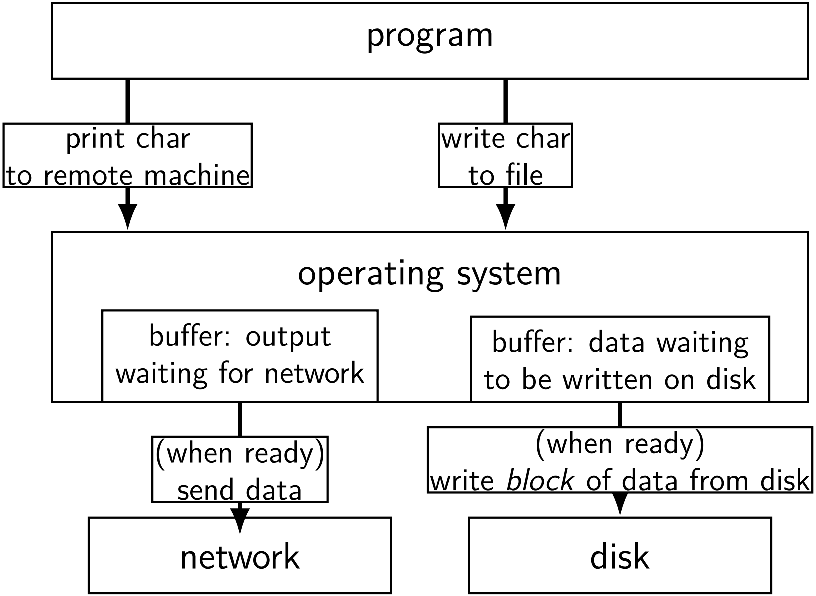
kernel buffering (writes)



kernel buffering (writes)



kernel buffering (writes)



read/write operations

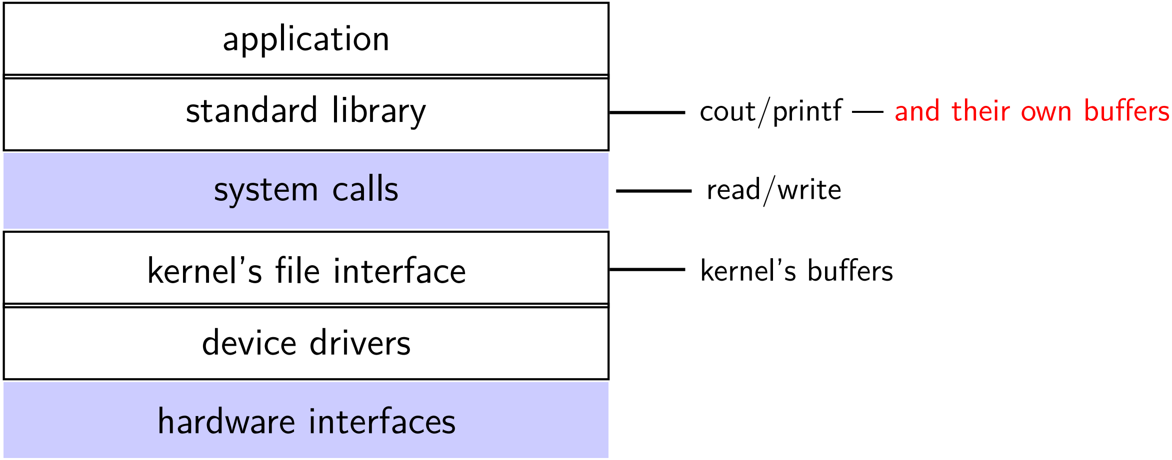
read()/write(): move data into/out of buffer

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

actual I/O operations — wait for device to be ready

- trigger process to stop waiting if needed

layering


application
standard library
system calls
kernel's file interface
device drivers
hardware interfaces
cout/printf — and their own buffers
read/write
kernel's buffers

why the extra layer

better (but more complex to implement) interface:

- read line
- formatted input (scanf, cin into integer, etc.)
- formatted output

less system calls (bigger reads/writes) sometimes faster

- buffering can combine multiple in/out library calls into one system call

more portable interface

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

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)

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

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

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?