last time

xv6 context switch

- user register save/restore: via exception stuff
- kernel register save/restore: save to kernel stack
- push/pop registers using calling convention + custom asm
- swtch function: assembly for switching stacks

new xv6 threads: manual construct new kernel stack

- as if: new process in middle of swtch call

process control blocks

- files, memory, process ID, ...

last time (2)

POSIX standard

- history: many variants of Unix
- common *source-code compatible* interface

fork

- basically deep copy of process control block
- returns twice (copied saved user registers)
- different return value in *parent* (old) and *child* (copy)

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

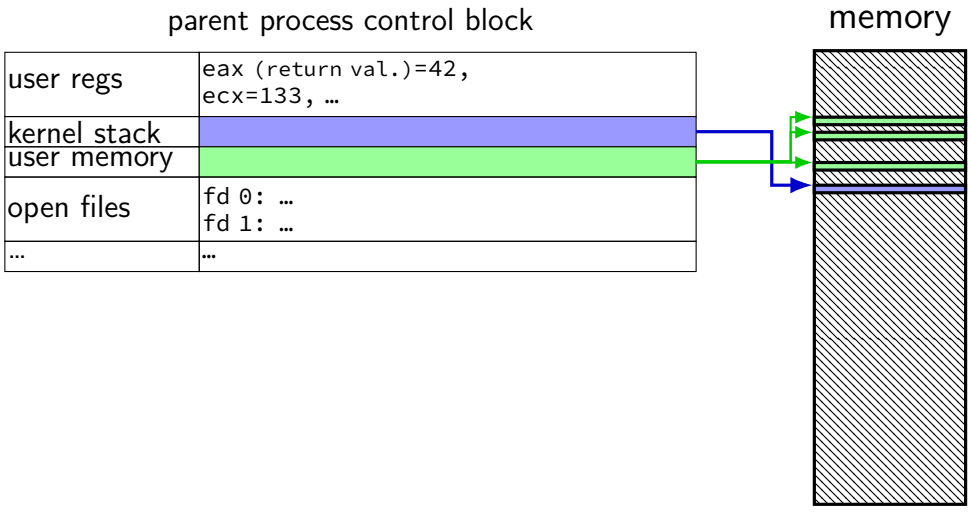
returns twice:

- in *parent* (original process): pid of new *child* process
- in *child* (new process): `0`

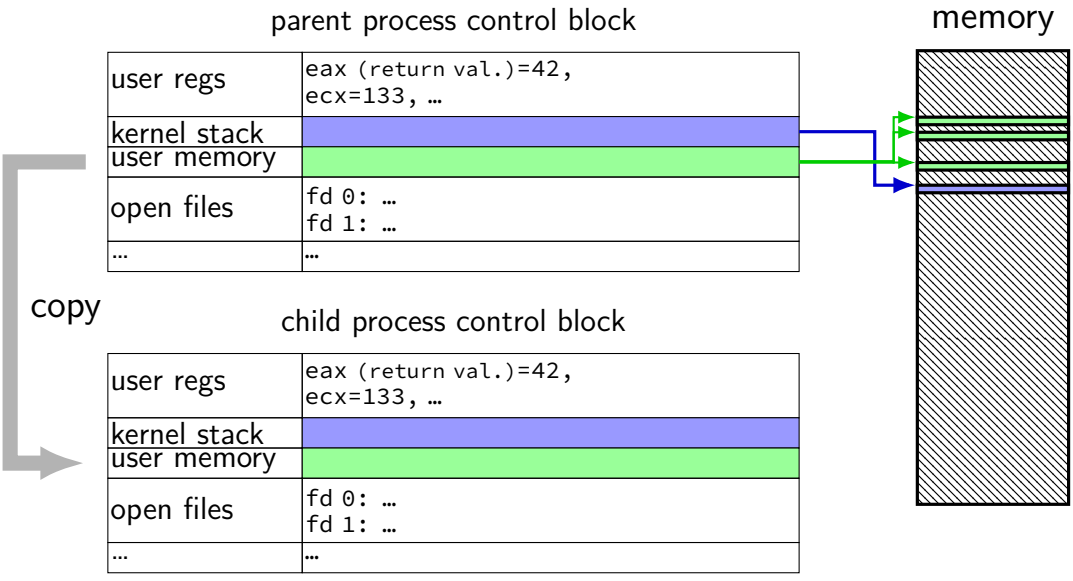
everything (but pid) duplicated in parent, child:

- memory
- file descriptors (later)
- registers

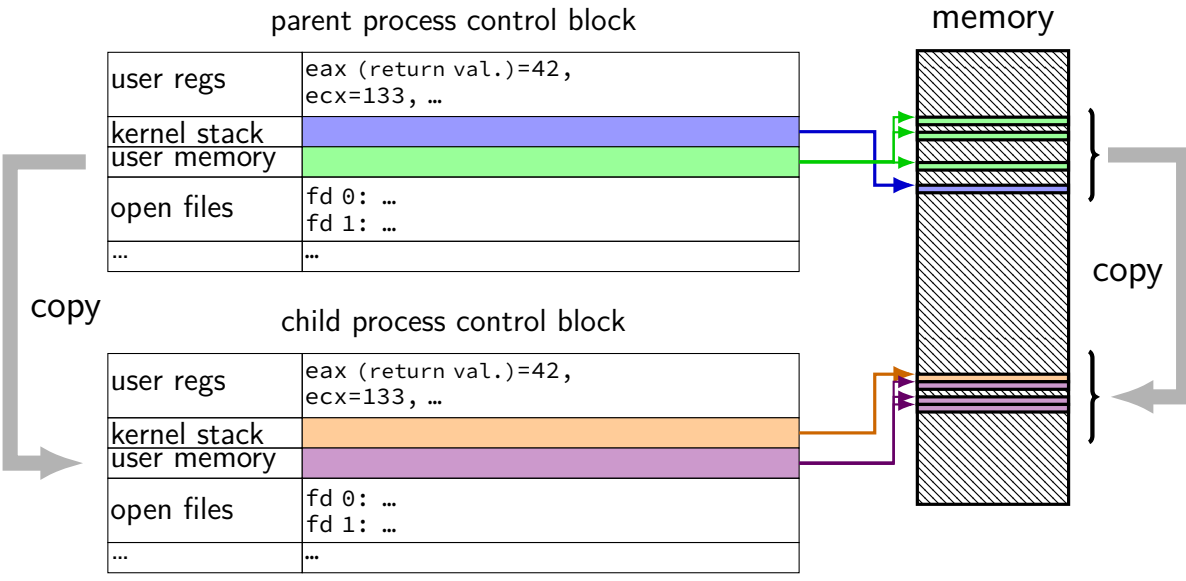
fork and PCBs



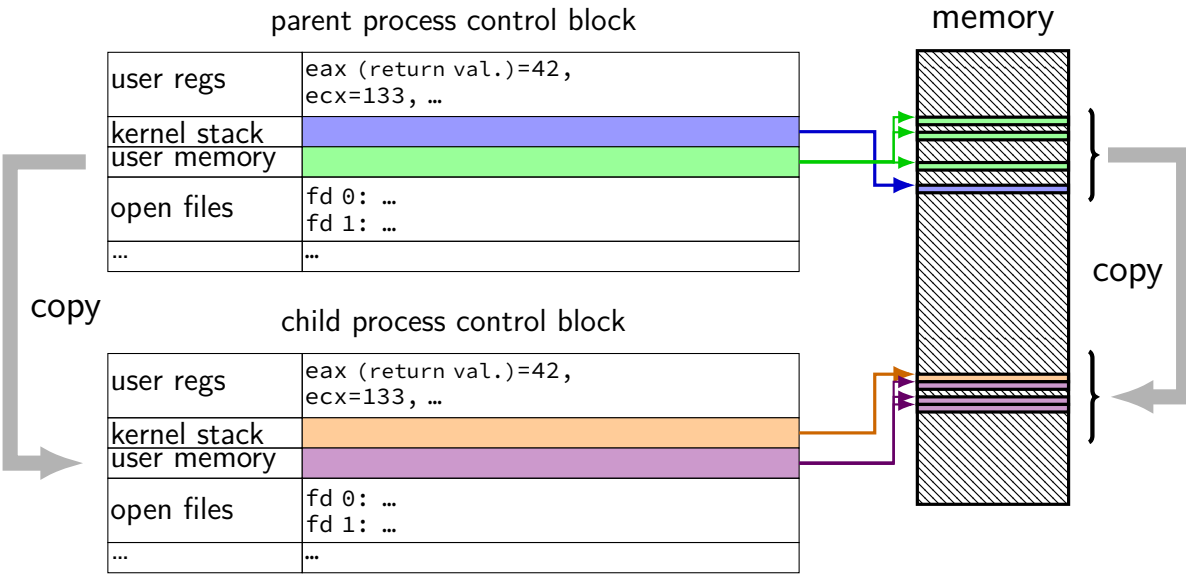
fork and PCBs



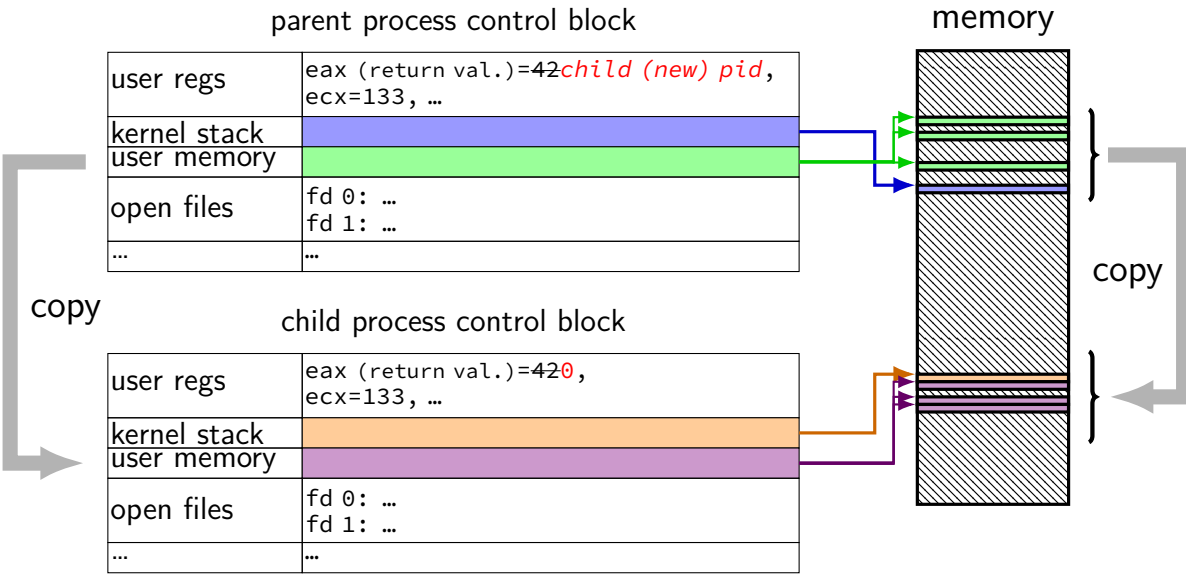
fork and PCBs



fork and PCBs



fork and PCBs



fork example

```c
#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

```c
#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 <unist
#include <sys/t
int main(int ar
    pid_t pid =
    printf("Par
    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
#include
int main(
    pid_t
    print
    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;
}
```

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

fork example

```c
#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.)

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.)


```
Child 100
In child
Done!
Done!
```


```
In child
Done!
Child 100
Done!
```

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

path: new program to run

argv: array of arguments, termianted by null pointer

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_p
  /* parent process
  ...
}
```

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

convention: first argument is program name

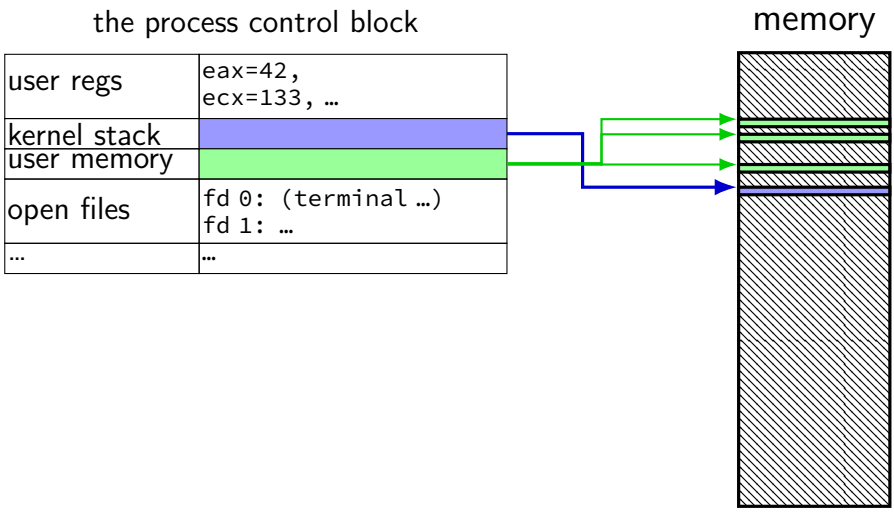
execv example

```
...
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 */
  ...
}
```

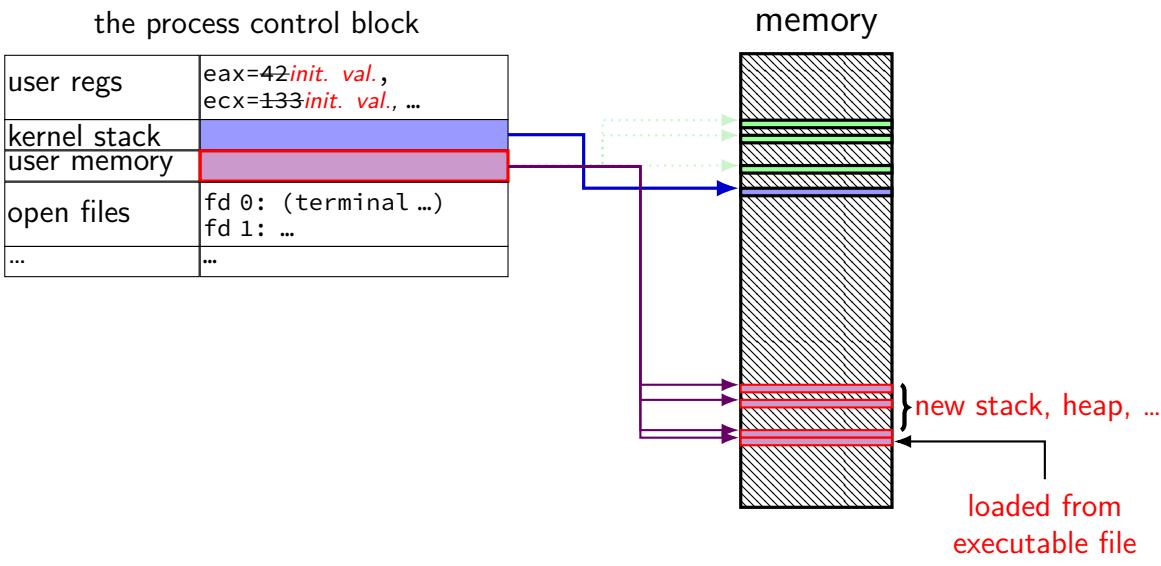
path of executable to run
need not match first argument
(but probably should match it)

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

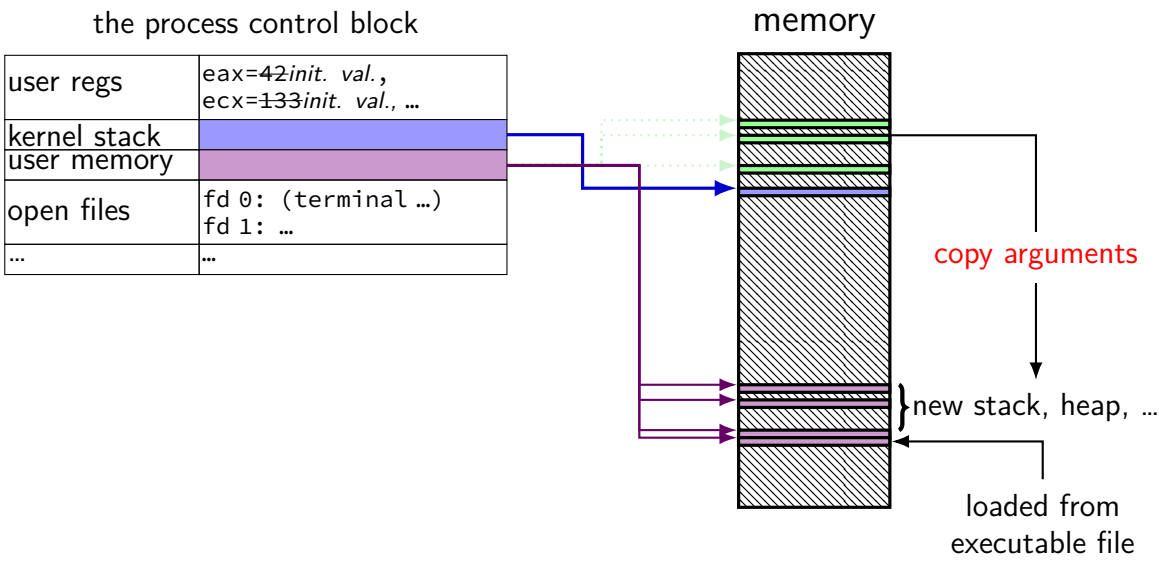
exec and PCBs



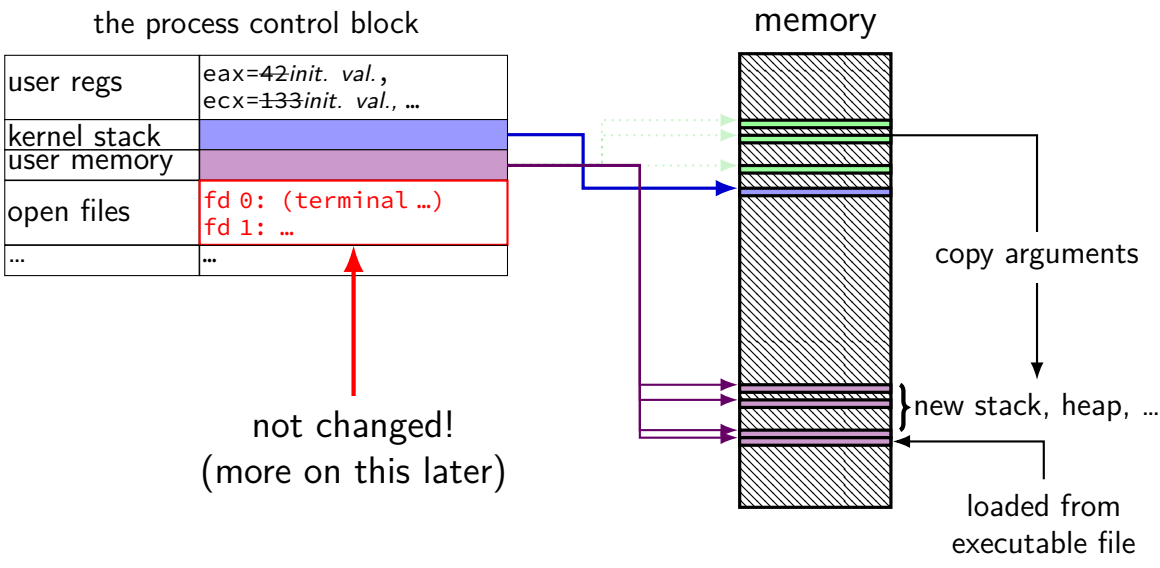
exec and PCBs



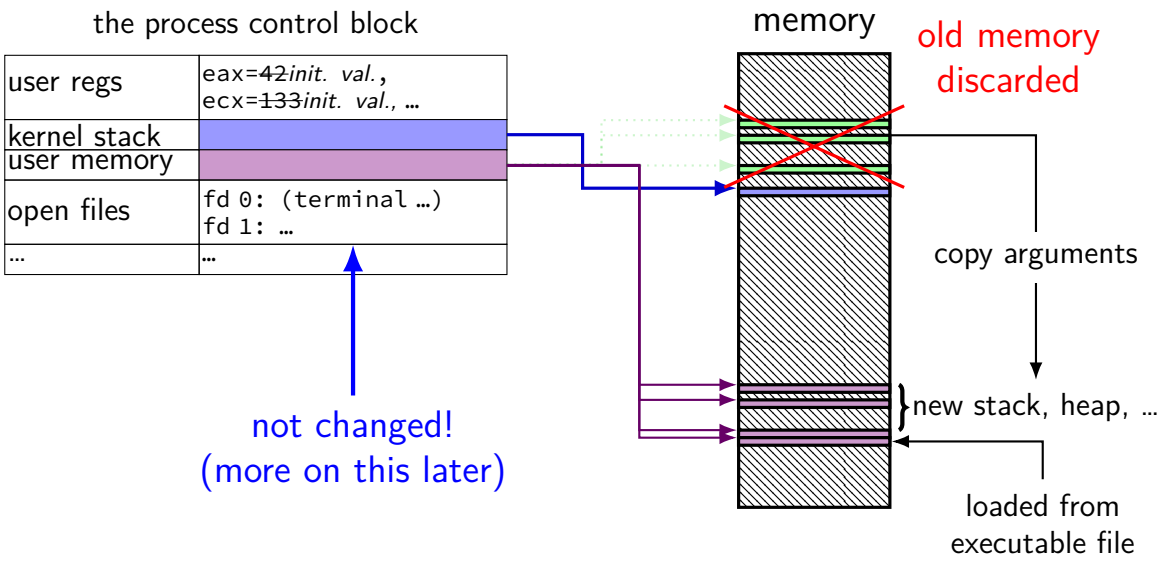
exec and PCBs



exec and PCBs



exec and PCBs



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: need function to set new program's current directory
- 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 variabl
            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

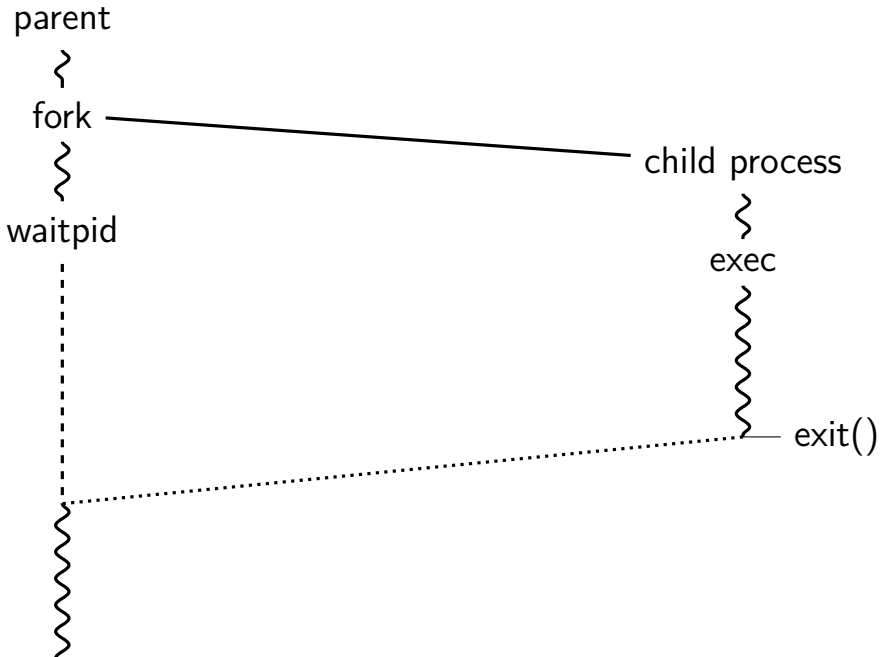
wait's status will tell you when and what signal killed a program

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

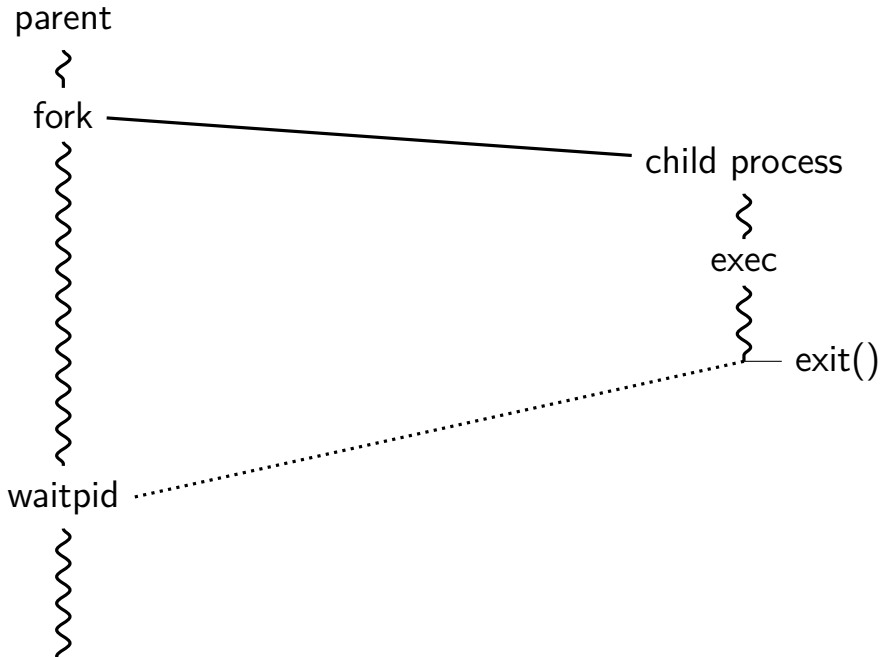
waiting for all children

```
#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 */
  }
```

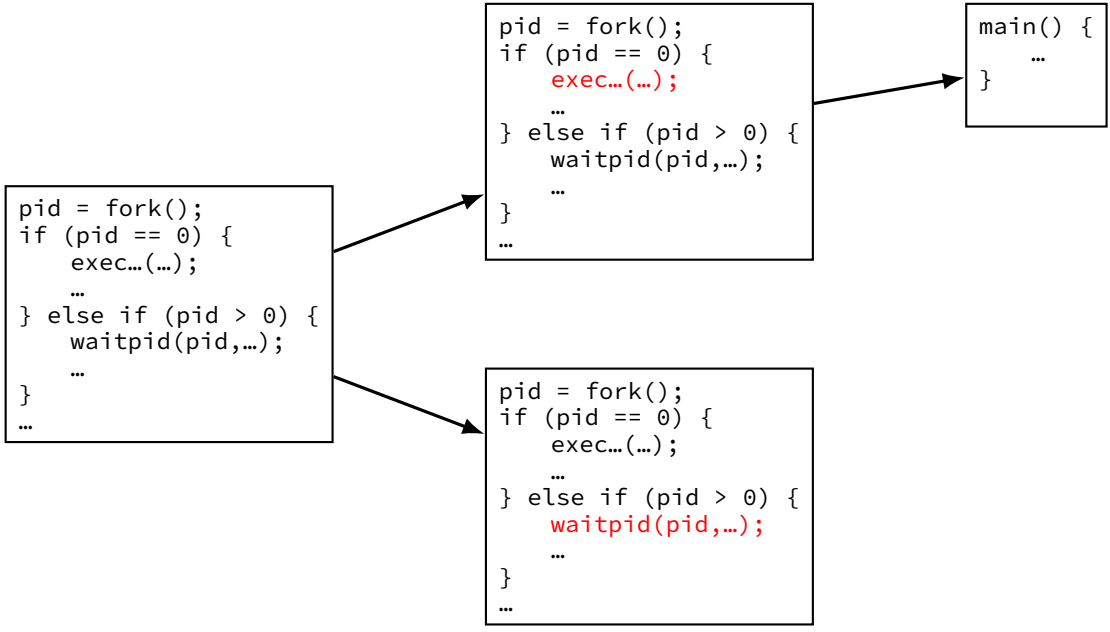
typical pattern



typical pattern (alt)



typical pattern (detail)



multiple processes?

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

/* retrieve exit statuses in order */
for (pid_t pid : pids) {
    waitpid(pid, ...);
    ...
}
```

multiple processes?

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

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

POSIX process management

essential operations

process information: `getpid`

process creation: `fork`

running programs: `exec*`

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

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

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

exercise (1)

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

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

A. L1 (newline) L2

B. L1 (newline) L2 (newline) L2

C. L2 (newline) L1

D. A and B

E. A and C

F. all of the above

G. something else

exercise (2)

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

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

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

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

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

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

E. A, B, and C

F. C and D

G. all of the above

H. something else

shell

allow user (= person at keyboard) to run applications

user's wrapper around process-management functions

upcoming homework — make a simple shell

aside: shell forms

POSIX: command line you have used before

also: graphical shells

e.g. OS X Finder, Windows explorer

other types of command lines?

completely different interfaces?

some POSIX command-line features

searching for programs (not in assignment)

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

running in background (not in assignment)

```
./someprogram &
```

redirection:

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

pipelines:

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

some POSIX command-line features

searching for programs (not in assignment)

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

running in background (not in assignment)

```
./someprogram &
```

redirection:

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

pipelines:

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

searching for programs

POSIX convention: PATH *environment variable*

- example: `/home/cr4bd/bin:/usr/bin:/bin`
- list of directories to check in order

environment variables = key/value pairs stored with process

- by default, left unchanged on execve, fork, etc.

one way to implement: [pseudocode]

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

some POSIX command-line features

searching for programs (not in assignment)

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

running in background (not in assignment)

```
./someprogram &
```

redirection:

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

pipelines:

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

some POSIX command-line features

searching for programs (not in assignment)

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

running in background (not in assignment)

```
./someprogram &
```

redirection:

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

pipelines:

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

shell assignment

implement a simple shell that supports redirection and pipeline

(for Linux or another POSIX system — not xv6)

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

simplified parsing: space-seperated:

okay: `/bin/ls␣-1␣>␣tmp.txt`

not okay: `/bin/ls␣-l␣>tmp.txt`

okay: `/bin/ls␣-1␣|␣/bin/grep␣foo␣>␣tmp.txt`

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

POSIX: everything is a file

the file: one interface for

- devices (terminals, printers, ...)
- regular files on disk
- networking (sockets)
- local interprocess communication (pipes, sockets)

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

the file interface

open before use

- setup, access control happens here

byte-oriented

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

explicit close

the file interface

open before use

- setup, access control happens here

byte-oriented

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

explicit close

filesystem abstraction

regular files — named collection of bytes

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

directories — folders containing files and directories

- hierarchical naming: `/net/zf14/cr4bd/fall2018/cs4414`
- *mostly* contains regular files or directories

open

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

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

open

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

path = filename

e.g. `"/foo/bar/file.txt"`
`file.txt` in
directory `bar` in
directory `foo` in
"the root directory"

e.g. `"quux/other.txt`
`other.txt` in
directory `quux` in
"the current working directory" (set with `chdir()`)

open: file descriptors

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

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

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

used by system calls that deal with open files

implementing file descriptors in xv6 (1)

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

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

pointer — *can be shared between proceses*

not part of deep copy fork does

null pointers — no file open with that number

implementing file descriptors in xv6 (2)

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

implementing file descriptors in xv6 (2)

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

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

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

implementing file descriptors in xv6 (2)

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

number of pointers to this struct file
used to safely delete this struct

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

implementing file descriptors in xv6 (2)

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

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

implementing file descriptors in xv6 (2)

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

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

special file descriptors

file descriptor 0 = standard input

file descriptor 1 = standard output

file descriptor 2 = standard error

constants in `unistd.h`

`STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO`

special file descriptors

file descriptor 0 = standard input

file descriptor 1 = standard output

file descriptor 2 = standard error

constants in `unistd.h`

- `STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO`

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

- more on this later

open: flags

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

flags: bitwise or of:

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

`man 2 open`

open: mode

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

mode: permissions of newly created file

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

simple advice: always use `0666`

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

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

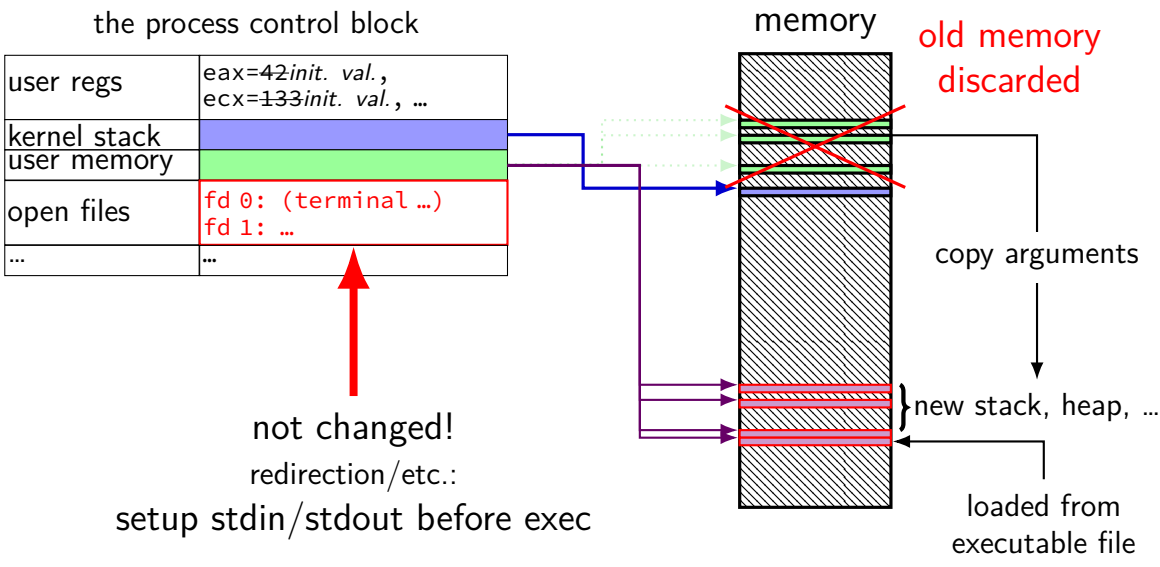
`./my_program ... < input.txt`:

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

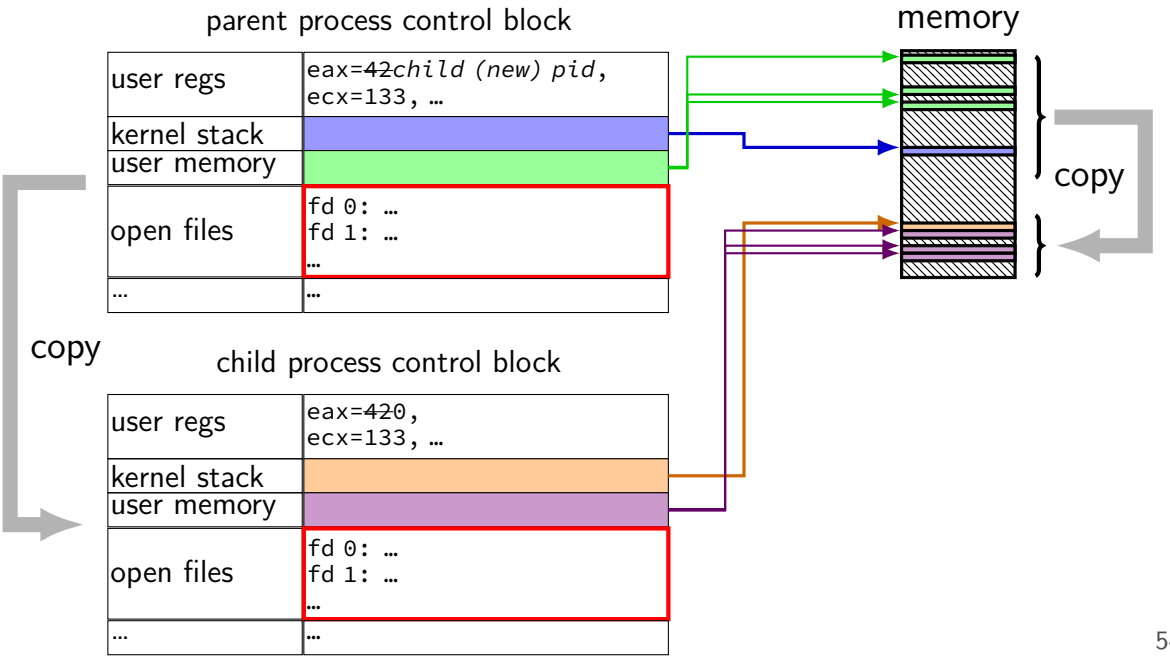
`echo foo > output.txt`:

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

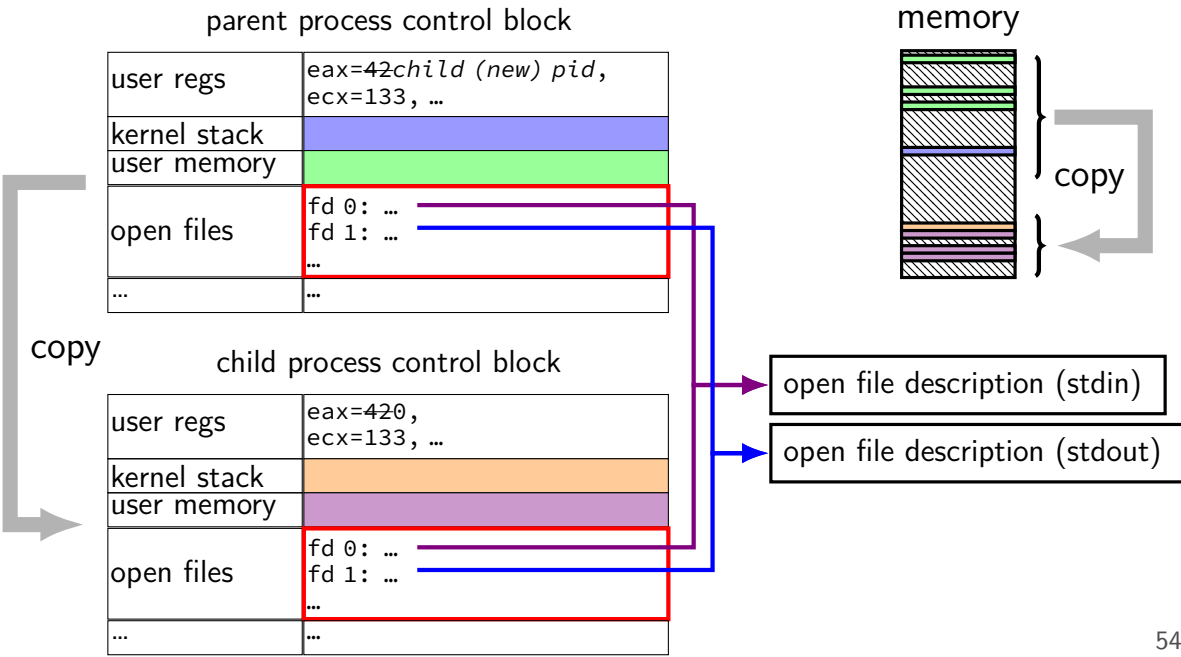
exec preserves open files



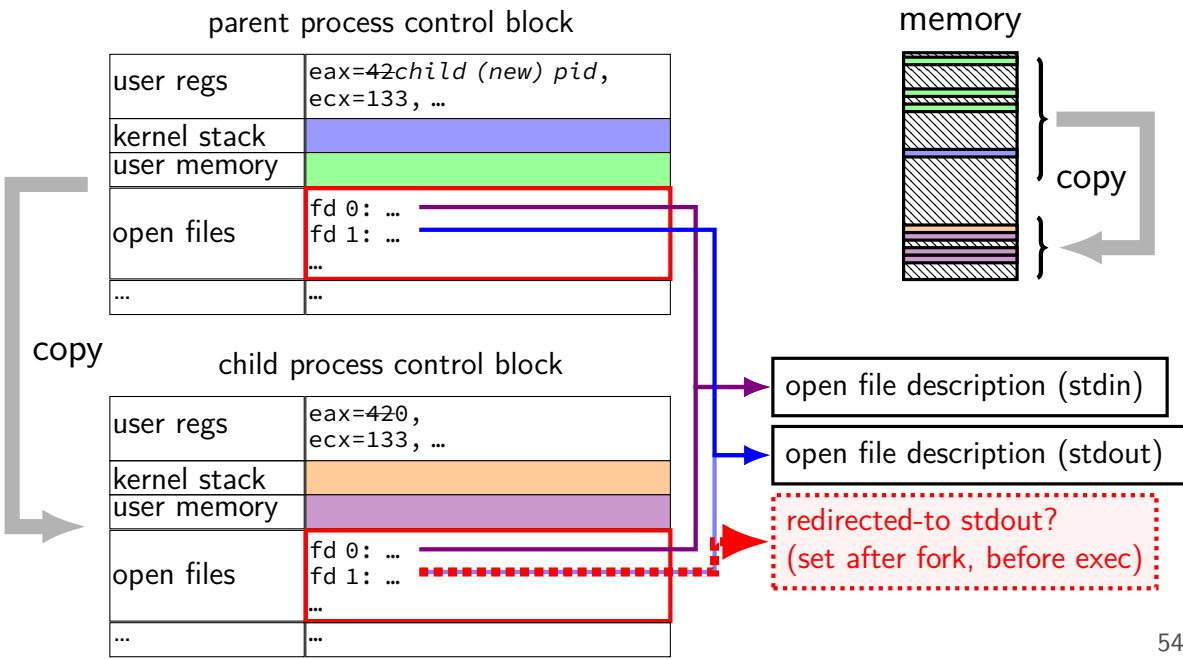
fork copies open file list



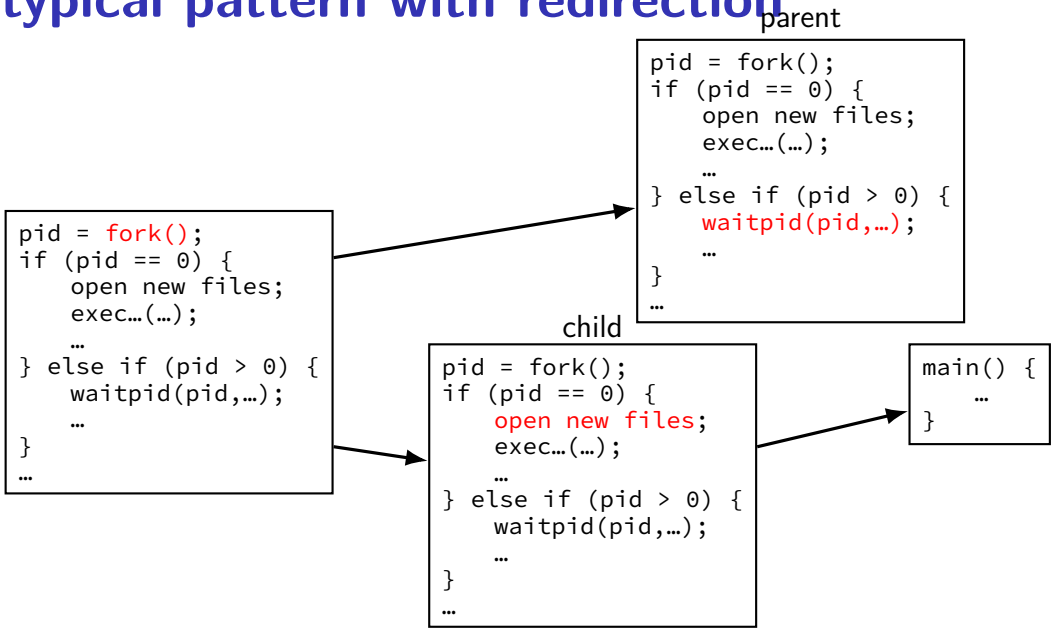
fork copies open file list



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

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

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

but can't access `ofile` from userspace

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

reassigning file descriptors

redirection: `./program >output.txt`

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

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

tool: `int dup2(int oldfd, int newfd)`
make `newfd` refer to same open file as `oldfd`

- same *open file description*
- shares the current location in the file
- (even after more reads/writes)

what if newfd already allocated — closed, then reused

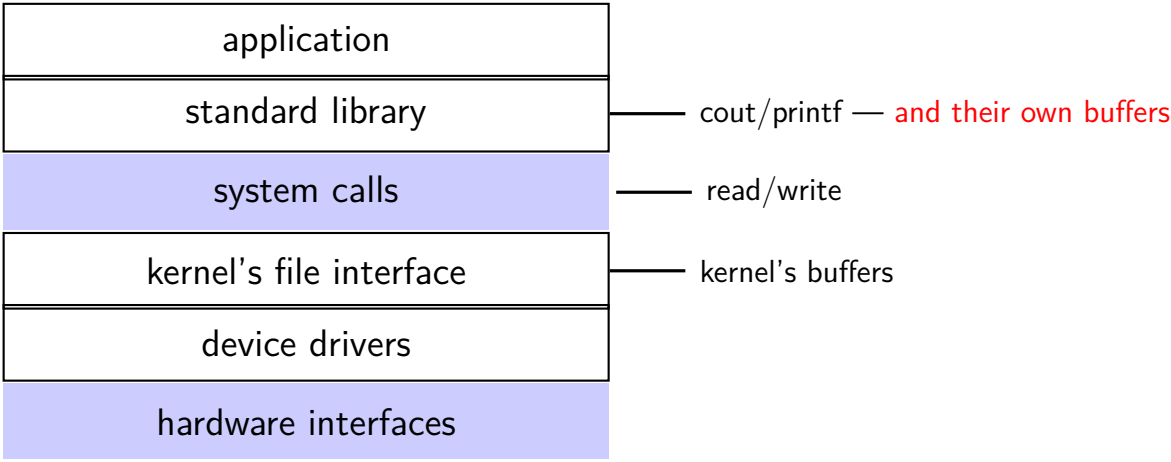
dup2 example

redirects stdout to output to `output.txt`:

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

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

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

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

backup slides

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

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):

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