Monolithic Kernels and the Unix API

homework

xv6 introduction: due Friday

anonymous feedback

“It would be really helpful if homework directions were less vague. Things don’t have to be hard and confusing just for the sake of it.

it probably is, but I can only make (probably bad) guesses about how it’s vague

trying to maintain balance:

- giving complete homework requirements
  - (without saying “modify line X of file Y”)
- not having walls of text that no one reads
- not copying all the lecture material, etc. into homework writeup

homework steps

system call implementation: sys_writecount

- hint in writeup: imitate sys_uptime
- need a counter for number of writes

add writecount to several tables/lists

- (list of handlers, list of library functions to create, etc.)
- recommendation: imitate how other system calls are listed

create a userspace program that calls writecount

- recommendation: copy from given programs

note on locks

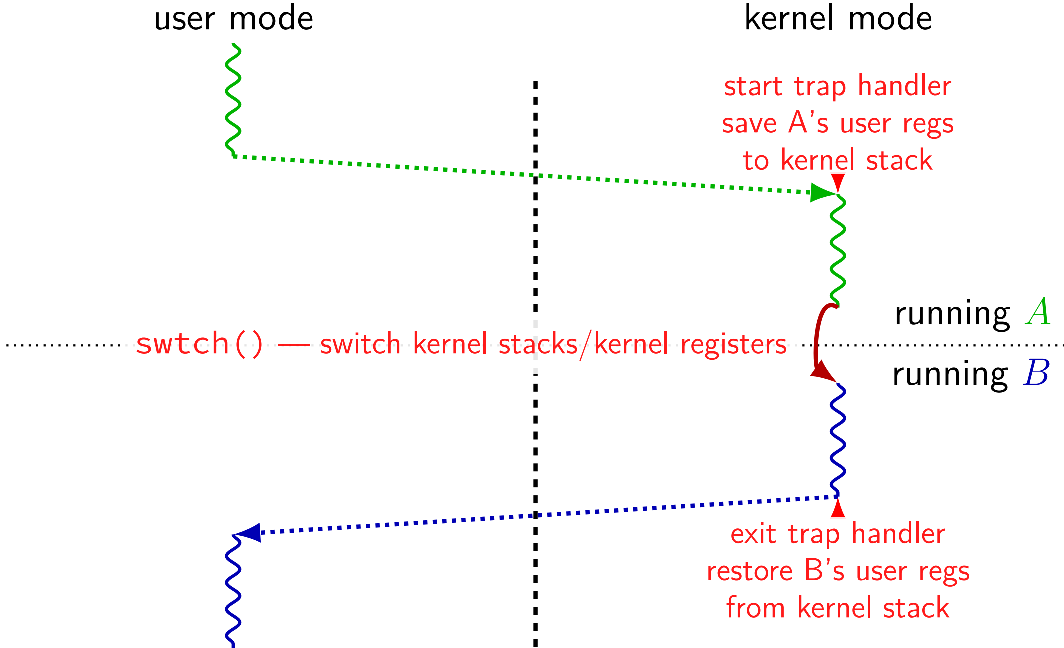
some existing code uses acquire/release

you do not have to do this

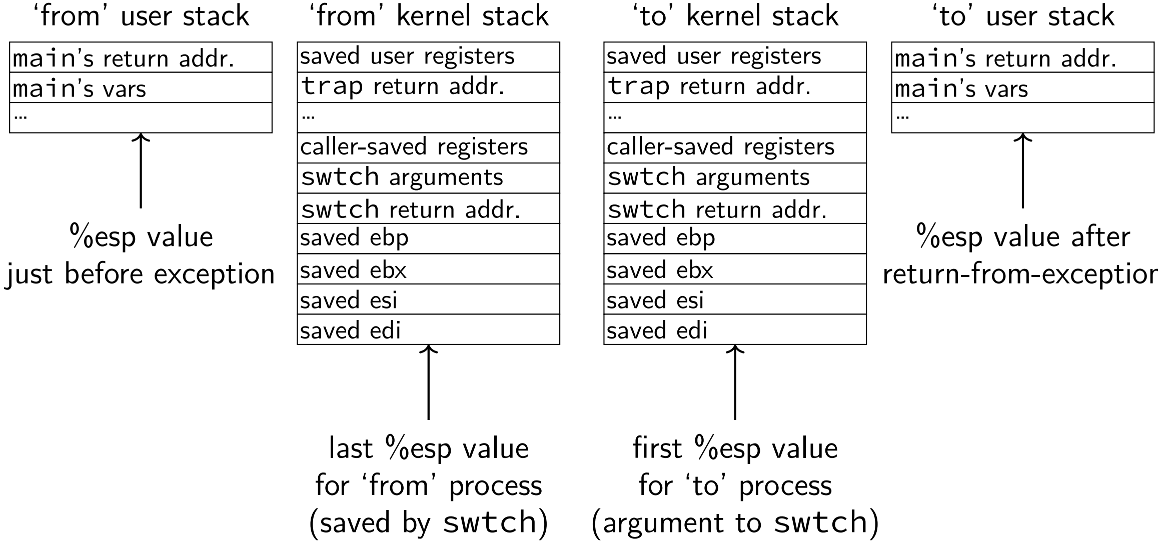
only for multiprocessor support

...but, copying what's done for ticks would be correct

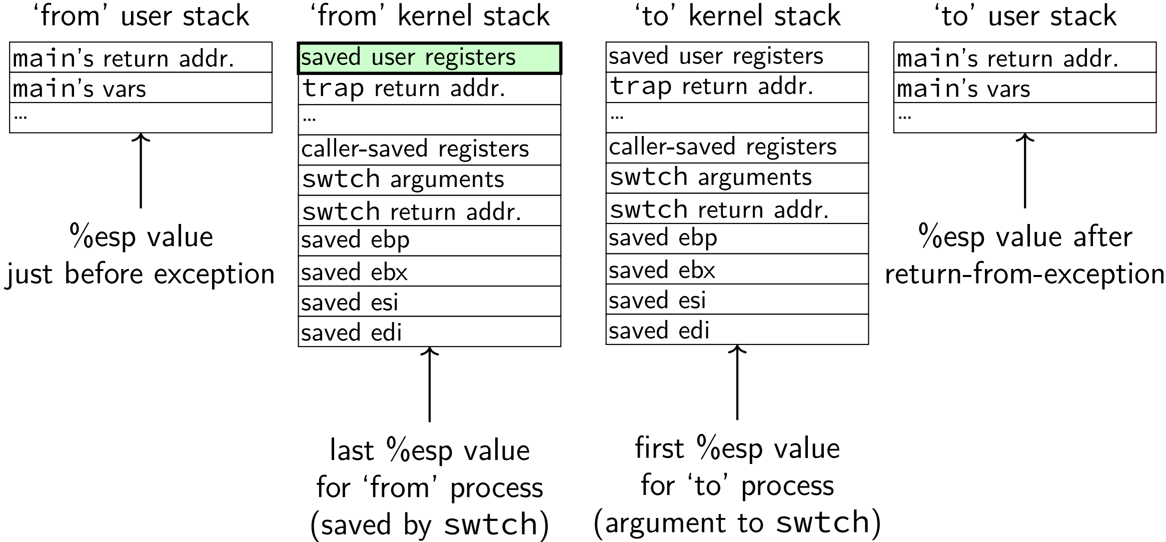
xv6 context switch and saving



where things go in context switch



where things go in context switch



write syscall in xv6: interrupt table setup

trap.c (run on boot)

```
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

`vectors[T_SYSCALL]` — OS function for processor to run
set to pointer to assembly function `vector64`

interrupt descriptor table

x86's interrupt descriptor table has an entry for each kind of exception

- segmentation fault
- timer expired ("your program ran too long")
- divide-by-zero
- system calls
- ...

xv6 sets all the table entries

...and they always call the `trap()` function

process control block

some data structure needed to represent a process

called Process Control Block

process control block

some data structure needed to represent a process

called Process Control Block

xv6: `struct proc`

xv6: struct proc

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

xv6: struct proc

current registers/PC of process (user and kernel)
stored on (pointer to) its kernel stack
(if not currently running)

≈ thread's state

```
struct proc {
  uint sz;
  pde_t* pgdir;
  char *kstack;                                                    ss
  enum procstate state
  int pid;                        // Process ID
  struct proc *parent;            // Parent process
  struct trapframe *tf;           // Trap frame for current syscall
  struct context *context;        // swtch() here to run process
  void *chan;                     // If non-zero, sleeping on chan
  int killed;                     // If non-zero, have been killed
  struct file *ofile[NOFILE];     // Open files
  struct inode *cwd;              // Current directory
  char name[16];                  // Process name (debugging)
};
```

xv6: struct proc

the kernel stack for this process
every process has one kernel stack

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

xv6: struct proc

```
struct proc
  uint sz;                                        // pr
  pde_t* pg                                       // le
  char *kst                                       // f                ss
  enum procstate state;                           // Process st
  int pid;                                        // Process ID
  struct proc *parent;                            // Parent process
  struct trapframe *tf;                           // Trap frame for current syscall
  struct context *context;                        // swtch() here to run process
  void *chan;                                     // If non-zero, sleeping on chan
  int killed;                                     // If non-zero, have been killed
  struct file *ofile[NOFILE];                     // Open files
  struct inode *cwd;                              // Current directory
  char name[16];                                  // Process name (debugging)
};
```

```
enum procstate {
    UNUSED, EMBRYO, SLEEPING,
    RUNNABLE, RUNNING, ZOMBIE
};
```

is process running?
or waiting?
or finished?
if waiting,
waiting for what (chan)?

xv6: struct proc

process ID
to identify process in systemn calls

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

xv6: struct proc

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Proc
  struct proc *parent;         // Pare
  struct trapframe *tf;        // Trap
  struct context *context;     // swtc
  void *chan;                  // If n
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

information about address space

pgdir — used by processor

sz — used by OS only

xv6: struct proc

information about open files, etc.

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

process control blocks generally

contains process's context(s) (registers, PC, …)

- if context is not on a CPU
- (in xv6: pointers to these, actual location: process's kernel stack)

process's status — running, waiting, etc.

information for system calls, etc.

- open files
- memory allocations
- process IDs
- related processes

xv6 myproc

xv6 function: `myproc()`

retrieves pointer to currently running `struct proc`

myproc: using a global variable

```
struct cpu cpus[NCPU];
```

```
struct proc*
myproc(void) {
    struct cpu *c;
    ...
    c = mycpu();    /* finds entry of cpus array
                       using special "ID" register
                       as array index */
    p = c->proc;
    ...
    return p;
}
```

this class: focus on Unix

Unix-like OSes will be our focus

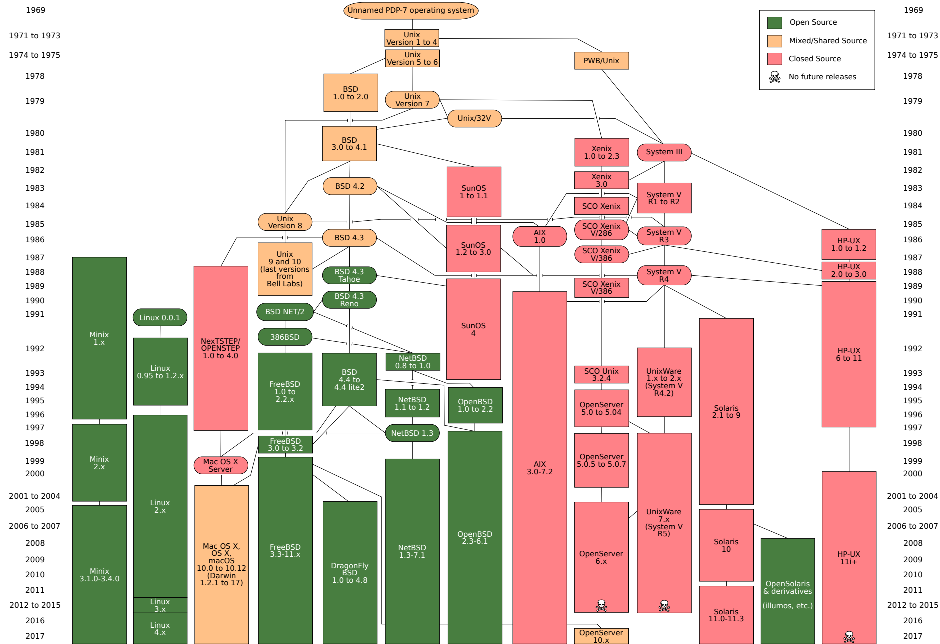
we have source code

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:
http://pubs.opengroup.org/onlinepubs/9699919799/

(almost) followed by most current Unix-like OSes

...but OSes add extra features

...but 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, 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`

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
```

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

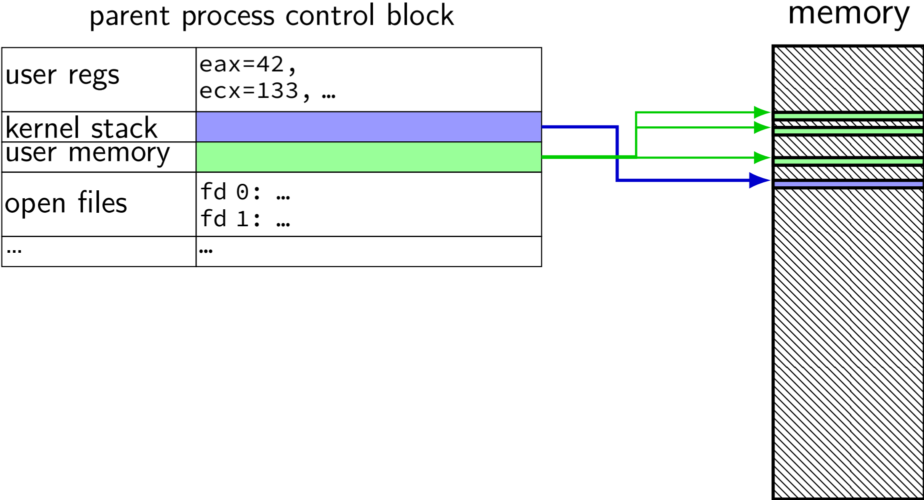
returns twice:

- 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



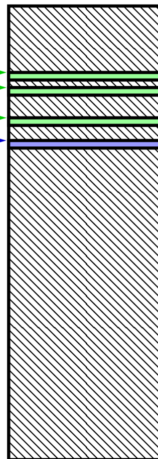
fork and PCBs

parent process control block

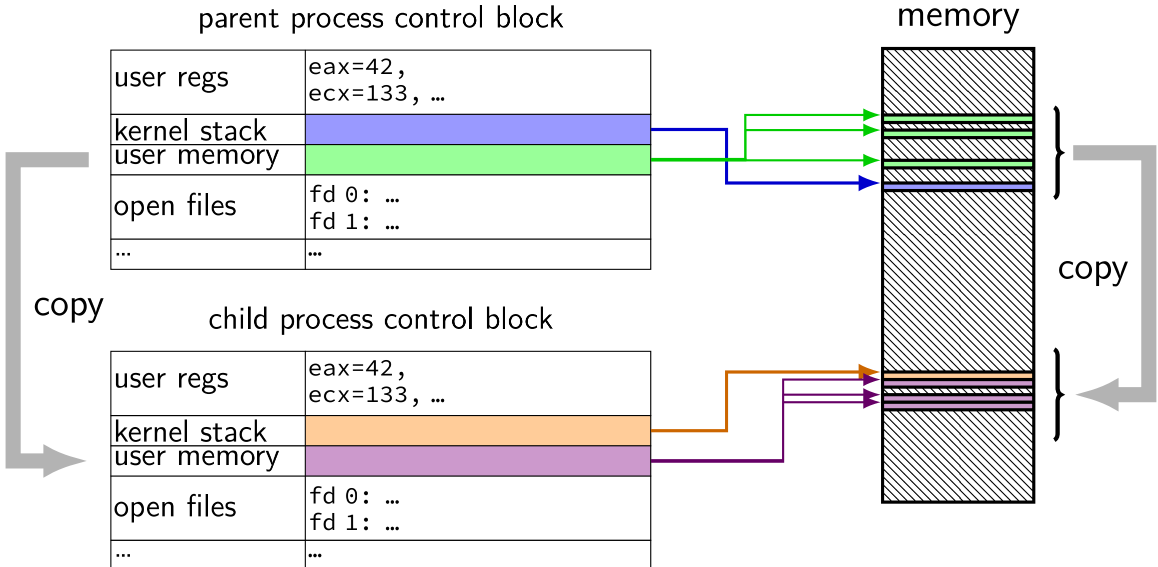
user regs	eax=42, ecx=133, …
kernel stack	
user memory	
open files	fd 0: … fd 1: …
…	…

child process control block

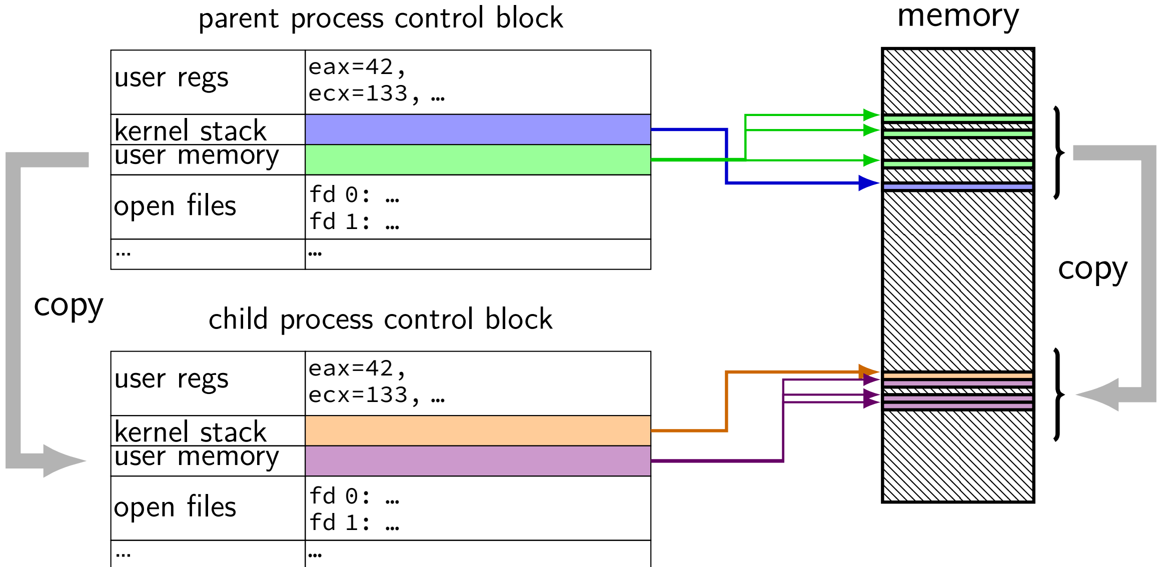
user regs	eax=42, ecx=133, …
kernel stack	
user memory	
open files	fd 0: … fd 1: …
…	…



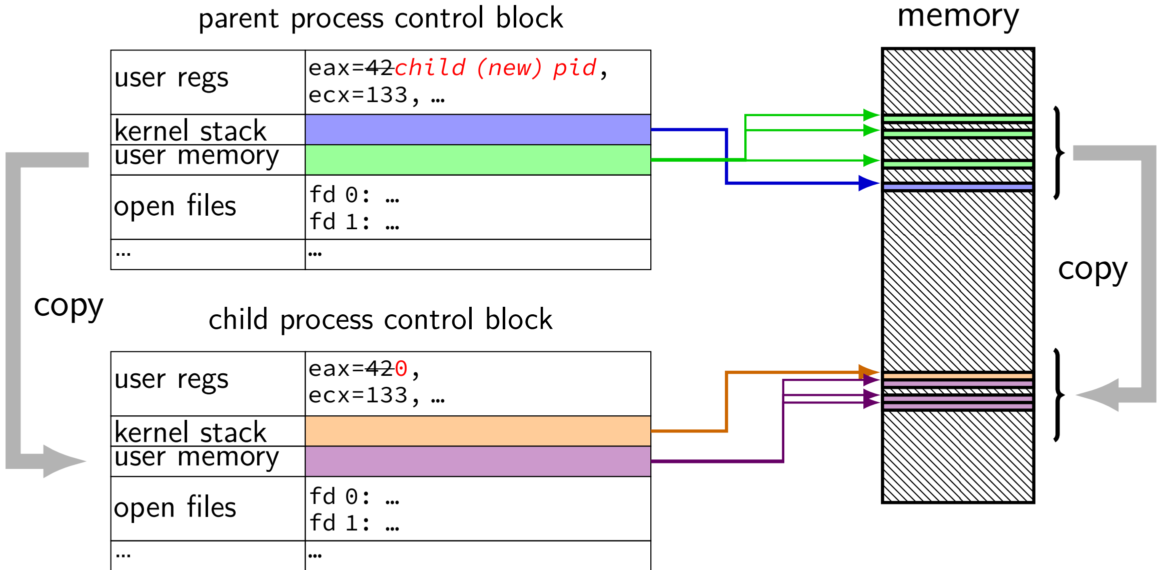
fork and PCBs



fork and PCBs



fork and PCBs



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

getpid — returns current process pid

```
#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 <stdli
#include <stdio
#include <unist
#include <sys/t
int main(int ar
    pid_t pid =
    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;
}
```

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
#include
#include
#include
int main(
    pid_t
    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;
}
```

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.)

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

used to compute argv, argc

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 */
  ...
}
```

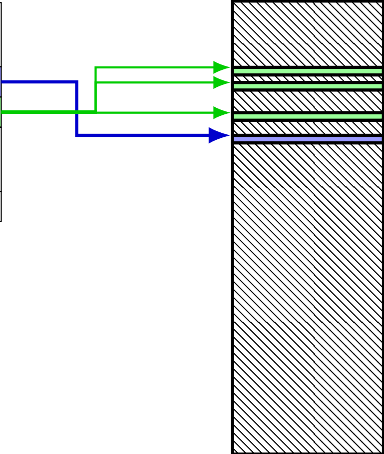
filename of program to run
need not match first argument
(but probably should match it)

exec and PCBs

the process control block

user regs	`eax=42`*init. val.*, `ecx=133`*init. val.*, …
kernel stack	
user memory	
open files	`fd 0: (terminal …)` `fd 1: …`
…	…

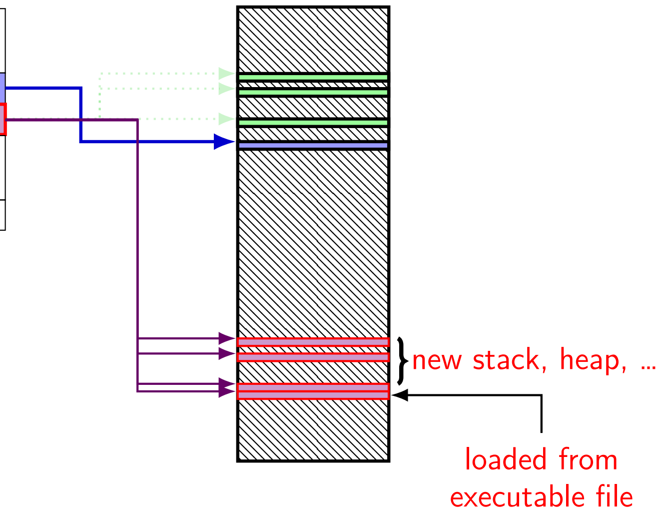
memory



exec and PCBs

the process control block

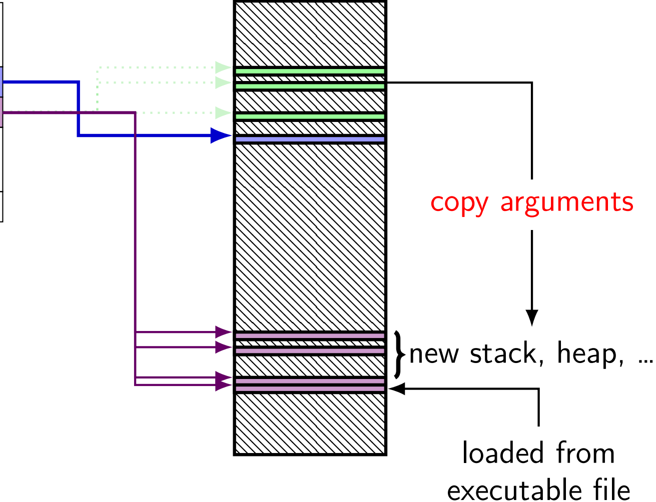
user regs	eax=~~42~~*init. val.*, ecx=~~133~~*init. val.*, …
kernel stack	
user memory	
open files	`fd 0: (terminal …)` `fd 1: …`
…	…



exec and PCBs

the process control block

user regs	eax=~~42~~*init. val.*, ecx=~~133~~*init. val.*, …
kernel stack	
user memory	
open files	fd 0: (terminal …) fd 1: …
…	…



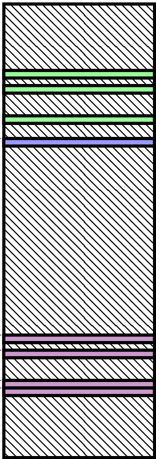
exec and PCBs

the process control block

user regs	eax=~~42~~*init. val.*, ecx=~~133~~*init. val.*, …
kernel stack	
user memory	
open files	fd 0: (terminal …) fd 1: …
…	…

not changed!
(more on this later)

memory



copy arguments

new stack, heap, …

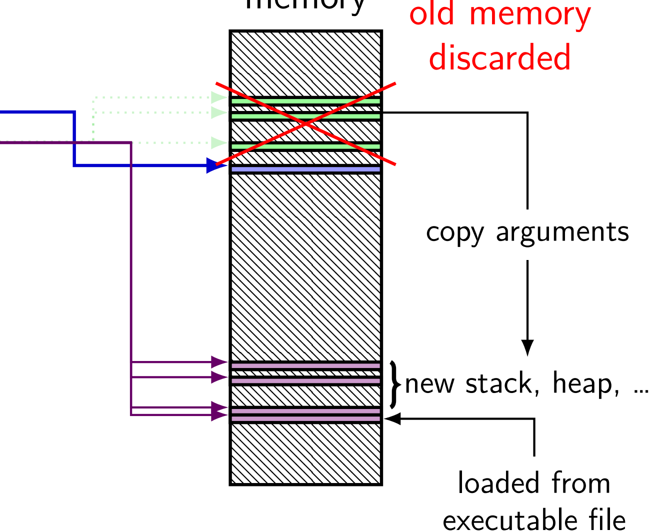
loaded from executable file

exec and PCBs

the process control block

user regs	eax=~~42~~*init. val.*, ecx=~~133~~*init. val.*, …
kernel stack	
user memory	
open files	fd 0: (terminal …) fd 1: …
…	…

not changed!
(more on this later)



aside: environment variables (1)

key=value pairs associated with every process:

```
$ printenv
MODULE_VERSION_STACK=3.2.10
MANPATH=:/opt/puppetlabs/puppet/share/man
XDG_SESSION_ID=754
HOSTNAME=labsrv01
SELINUX_ROLE_REQUESTED=
TERM=screen
SHELL=/bin/bash
HISTSIZE=1000
SSH_CLIENT=128.143.67.91 58432 22
SELINUX_USE_CURRENT_RANGE=
QTDIR=/usr/lib64/qt-3.3
OLDPWD=/zf14/cr4bd
QTINC=/usr/lib64/qt-3.3/include
SSH_TTY=/dev/pts/0
QT_GRAPHICSSYSTEM_CHECKED=1
USER=cr4bd
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=
MODULE_VERSION=3.2.10
MAIL=/var/spool/mail/cr4bd
PATH=/zf14/cr4bd/.cargo/bin:/zf14/cr4bd/bin:/usr/lib64/qt-3.3/bin:/usr/local/bin:/usr/bin:/us
PWD=/zf14/cr4bd
LANG=en_US.UTF-8
MODULEPATH=/sw/centos/Modules/modulefiles:/sw/linux-any/Modules/modulefiles
```

aside: environment variables (2)

environment variable library functions:

getenv("KEY") → *value*
putenv("KEY=*value*") (sets KEY to *value*)
setenv("KEY", "value") (sets KEY to *value*)

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

```
char *envp[] = { "KEY1=value1", "KEY2=value2", NULL };
char *argv[] = { "somecommand", "some arg", NULL };
execve("/path/to/somecommand", argv, envp);
```

normal exec versions — keep same environment variables

aside: environment variables (3)

interpretation up to programs, but common ones…

`PATH=/bin:/usr/bin`

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

`HOME=/zf14/cr4bd`

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

`TERM=screen-256color`

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

…

why fork/exec?

could just have a function to spawn a new program

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

needs to include API to set new program state
- open files, current directory, environment variables, ...
- with fork: just use 'normal' API

POSIX: `posix_spawn`, but rarely used, often not implemented (or implemented poorly)

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`)

- we'll use `0`

exit statuses

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

waitpid example

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

the status

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

"status code" encodes both return value and if exit was abnormal

W* macros to decode it

the status

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

“status code” encodes both return value and if exit was abnormal

W* macros to decode it

aside: signals

signals are a way of communicating between processes

they are also how abnormal termination happens

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

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

waiting for all children

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

parent and child processes

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

`getppid()`

this is the process that can wait for it

creates tree of processes:

```
init(1)-+-ModemManager(919)-+-{ModemManager}(972)
        |                   `-{ModemManager}(1064)
        |-NetworkManager(1160)-+-dhclient(1755)
        |                      |-dnsmasq(1985)
        |                      |-{NetworkManager}(1180)
        |                      |-{NetworkManager}(1194)
        |                      `-{NetworkManager}(1195)
        |-accounts-daemon(1649)-+-{accounts-daemon}(1757)
        |                       `-{accounts-daemon}(1758)
        |-acpid(1338)
        |-apache2(3165)-+-apache2(4125)-+-{apache2}(4126)
        |               |               `-{apache2}(4127)
        |               |-apache2(28920)-+-{apache2}(28926)
        |               |                `-{apache2}(28960)
        |               |-apache2(28921)-+-{apache2}(28927)
        |               |                `-{apache2}(28963)
        |               |-apache2(28922)-+-{apache2}(28928)
        |               |                `-{apache2}(28961)
        |               |-apache2(28923)-+-{apache2}(28930)
        |               |                `-{apache2}(28962)
        |               |-apache2(28925)-+-{apache2}(28958)
        |               |                `-{apache2}(28965)
        |               `-apache2(32165)-+-{apache2}(32166)
        |                                `-{apache2}(32167)
        |-at-spi-bus-laun(2252)-+-dbus-daemon(2269)
        |                       |-{at-spi-bus-laun}(2266)
        |                       |-{at-spi-bus-laun}(2268)
        |                       `-{at-spi-bus-laun}(2270)
        |-at-spi2-registr(2275)---{at-spi2-registr}(2282)
        |-atd(1633)
        |-automount(13454)-+-{automount}(13455)
        |                  |-{automount}(13456)
        |                  |-{automount}(13461)
        |                  |-{automount}(13464)
        |                  `-{automount}(13465)
```

```
|                                `-{mcollectived}(2038)
|-mongod(1336)-+-{mongod}(1556)
|              |-{mongod}(1557)
|              |-{mongod}(1983)
|              |-{mongod}(2031)
|              |-{mongod}(2047)
|              |-{mongod}(2048)
|              |-{mongod}(2049)
|              |-{mongod}(2050)
|              |-{mongod}(2051)
|              `-{mongod}(2052)
|-mosh-server(19090)---bash(19091)---tmux(5442)
|-mosh-server(21996)---bash(21997)
|-mosh-server(22533)---bash(22534)---tmux(22588)
|-nm-applet(2580)-+-{nm-applet}(2739)
|                 `-{nm-applet}(2743)
|-nmbd(2224)
|-ntpd(3091)
|-polkitd(1197)-+-{polkitd}(1239)
|               `-{polkitd}(1240)
|-pulseaudio(2563)-+-{pulseaudio}(2617)
|                  `-{pulseaudio}(2623)
|-puppet(2373)---{puppet}(32455)
|-rpc.idmapd(875)
|-rpc.statd(954)
|-rpcbind(884)
|-rserver(1501)-+-{rserver}(1786)
|               `-{rserver}(1787)
|-rsyslogd(1090)-+-{rsyslogd}(1092)
|                |-{rsyslogd}(1093)
|                `-{rsyslogd}(1094)
|-rtkit-daemon(2565)-+-{rtkit-daemon}(2566)
|                    `-{rtkit-daemon}(2567)
|-sd_cicero(2852)-+-sd_cicero(2853)
|                 |-{sd_cicero}(2854)
|                 `-{sd_cicero}(2855)
|-sd_dummy(2849)-+-{sd_dummy}(2850)
|                `-{sd_dummy}(2851)
|-sd_espeak(2749)-+-{sd_espeak}(2845)
|                 |-{sd_espeak}(2846)
```

parent and child questions...

what if parent process exits before child?

- child's parent process becomes process id 1

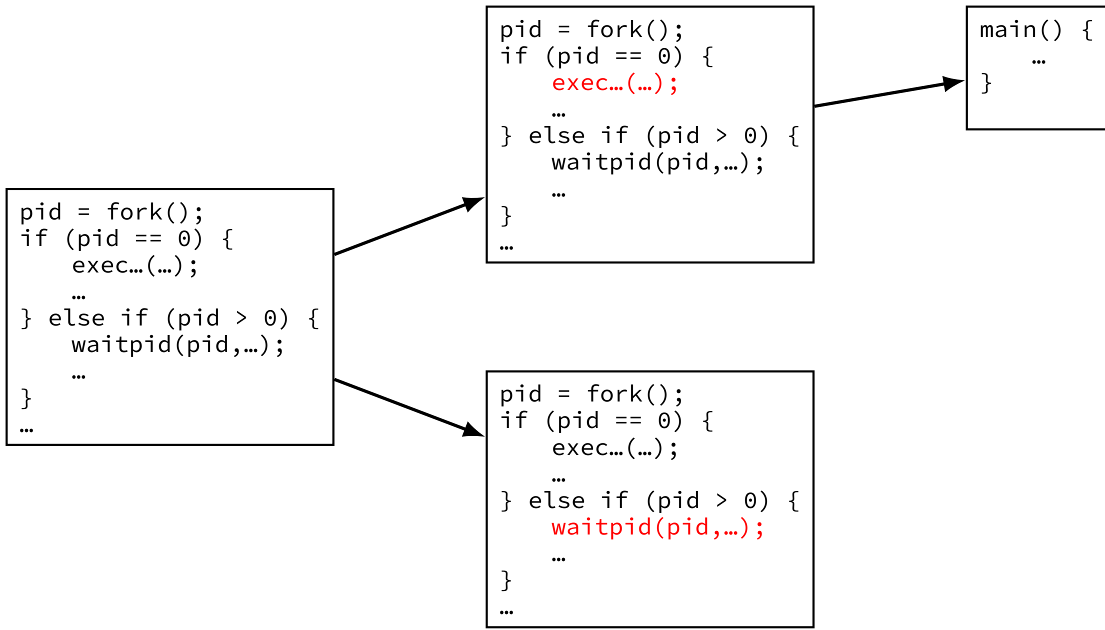
what if parent process never `waitpid()`/`wait()`s for child?

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

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

- waitpid fails

typical pattern



multiple processes?

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

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

multiple processes?

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

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

POSIX process management

essential operations

process information: `getpid`

process creation: `fork`

running programs: `exec*`

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

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

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

shell

allow user (= person at keyborad) 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 (mostly 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 (mostly 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`

checked in order

one way to implement: [pseudocode]

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

some POSIX command-line features

searching for programs (mostly 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
```

running in background

```
$ ./long_computation >tmp.txt &
[1] 4049
$ ...
[1]+    Done            ./long_computation > tmp.txt
$ cat tmp.txt
the result is ...
```

& — run a program in “background”

initially output PID (above: 4049)

print out after terminated

- one way: use `waitpid` with flag to say “don’t wait”