thread creation / POSIX / process management

Changelog

21 January 2020 (between 12:30pm and 3:30pm lecture): added alternate version of typical pattern slide

last time

handling non-system-call exceptions

context switching in xv6

- swtch(A, B): save A's regs on A's stacks; read B's regs from B's stacks
- trick: use return address for swtch call to save/restore program counter
- trick: use swtch call to save/restore caller-saved regs
- trick: A, B represented by (kernel) stack pointers
- user part: save/restore regs on trap() entry/exit
- extra stuff to handle address space switch

first call to swtch?

one thread calls swtch and

...return from another thread's call to swtch

...using information on that thread's stack

first call to swtch?

one thread calls swtch and

...return from another thread's call to swtch

...using information on that thread's stack

what about switching to a new thread?

trick: setup stack *as if* in the middle of `swtch`

write saved registers + return address onto stack

avoids special code to swtch to new thread

(in exchange for special code to create thread)

creating a new thread

```
static struct proc*
allocproc(void)
{
  ...
  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
  ...
```

`struct proc` ≈ process
p is new `struct proc`
`p->kstack` is its new stack
(for the kernel only)

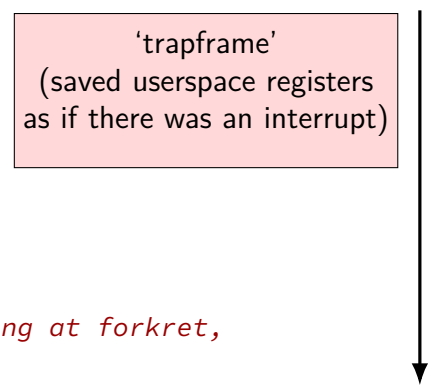
creating a new thread

new kernel stack

```
static struct proc*
allocproc(void)
{
  ...
  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
  ...
```

creating a new thread

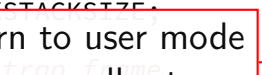
```
static struct proc*
allocproc(void)
{
  ...
  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
  ...
```

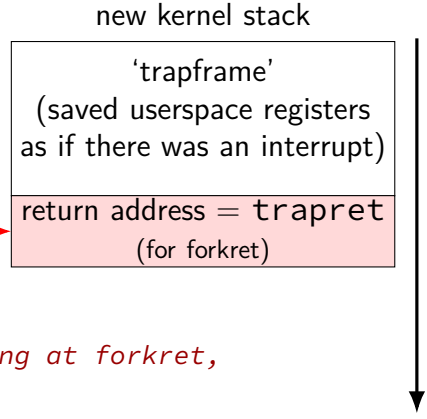


creating a new thread

```
static struct proc*
allocproc(void)
{
  ...
  sp = p->kstack + KSTACKSIZE;

  // leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
  ...
```

assembly code to return to user mode
same code as for syscall returns



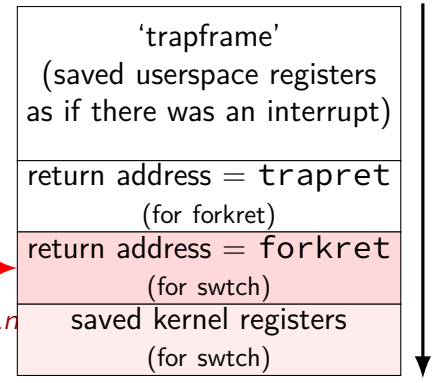


creating a new thread

```
static struct proc*
allocproc(void)
{
  ...
  sp = p->kstack + KSTACKSIZE;

  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
  ...
```

initial code to run when starting a new process

(`fork` = process creation system call)

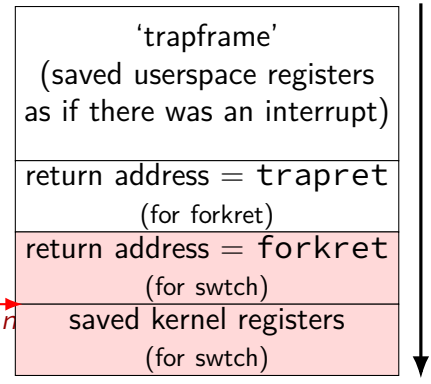


creating a new thread

```
static struct proc*
allocproc(void)
{
  ...
  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executin
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
  ...
```

saved registers (incl. return address) for `swtch` to pop off the stack

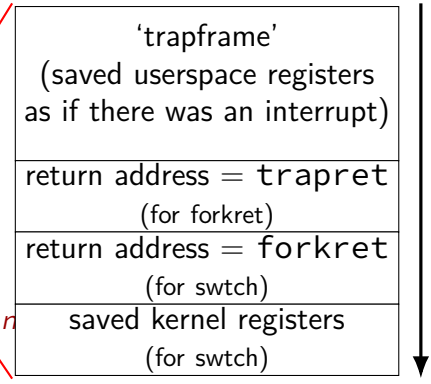
new kernel stack



creating a new thread

```
static struct proc*
allocproc(void)
{
  ...
  sp =
  // L
  sp =
  p->t
  // Set up new context to start executin
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
  ...
```

new stack says: this thread is in middle of calling `swtch` in the middle of a system call



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

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

≈ thread's state

```
struct proc
  uint sz;
  pde_t* pg
  char *kst                                                    ss
  enum proc
  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

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

```
struct proc
  uint sz;
  pde_t* pg
  char *kst
  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)
};
```

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

xv6: struct proc

process ID
to identify process in system 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

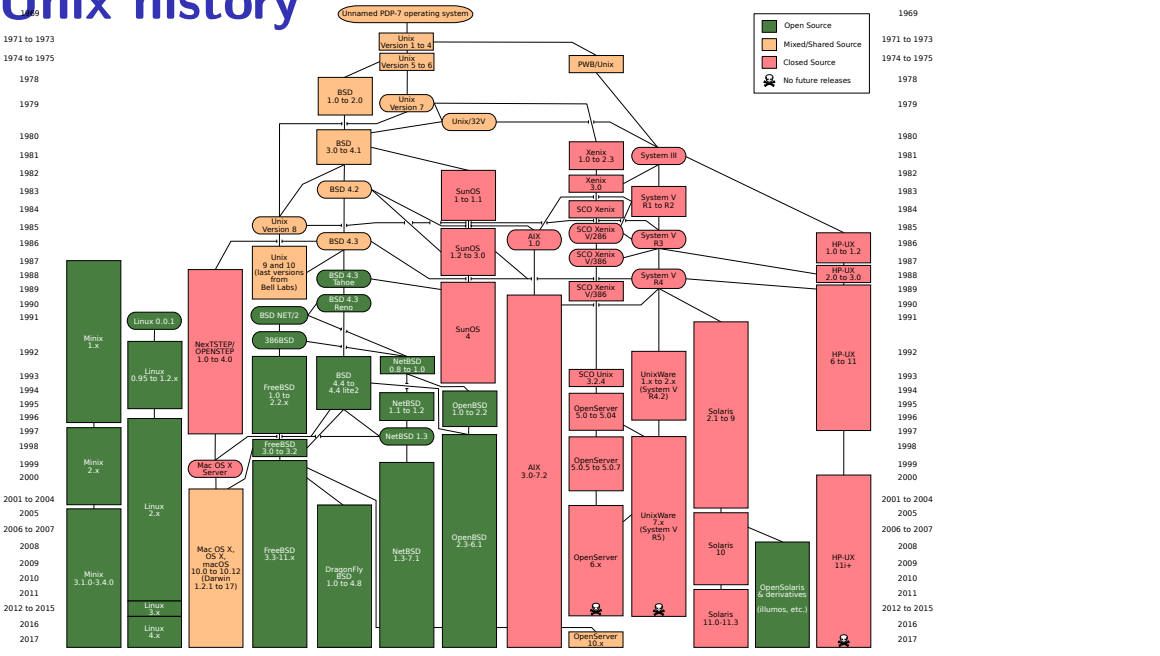
we have source code

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

...and POSIX doesn't specify everything

what POSIX defines

POSIX specifies the library and shell interface

- source code compatibility

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

doesn't specify binary formats...

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

- this was a very important goal in the 80s/90s
- at the time, 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

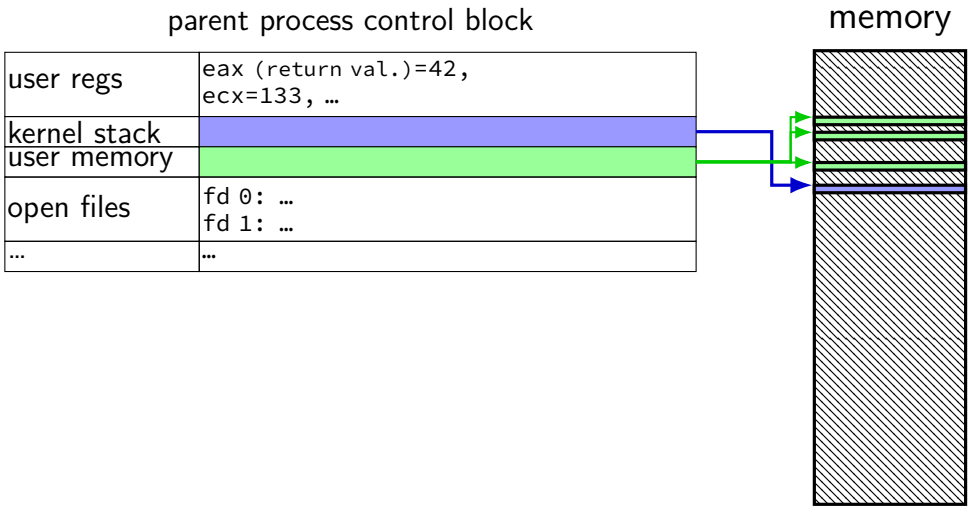
returns twice:

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

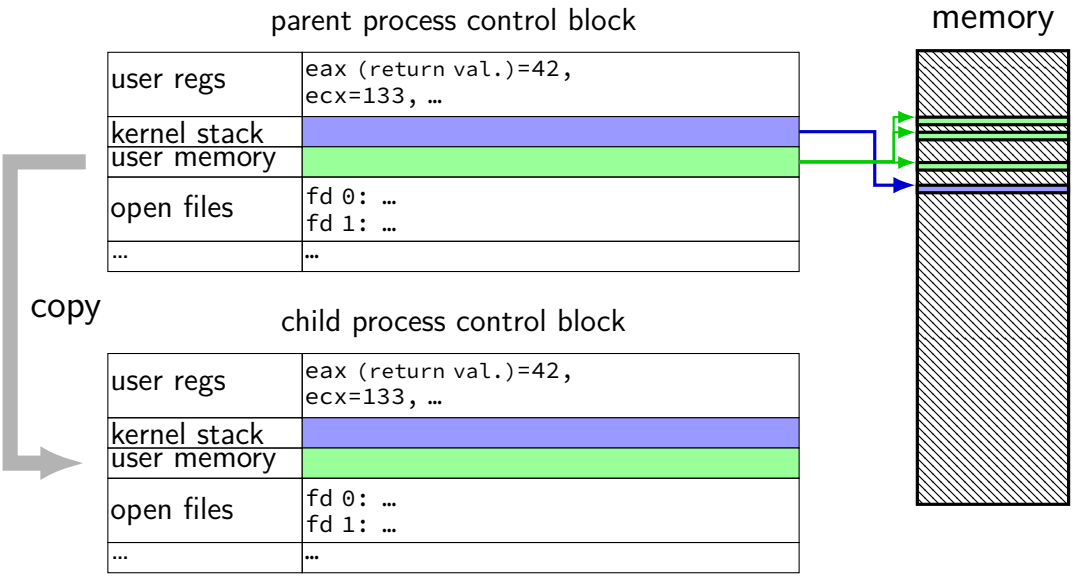
everything (but pid) duplicated in parent, child:

- memory
- file descriptors (later)
- registers

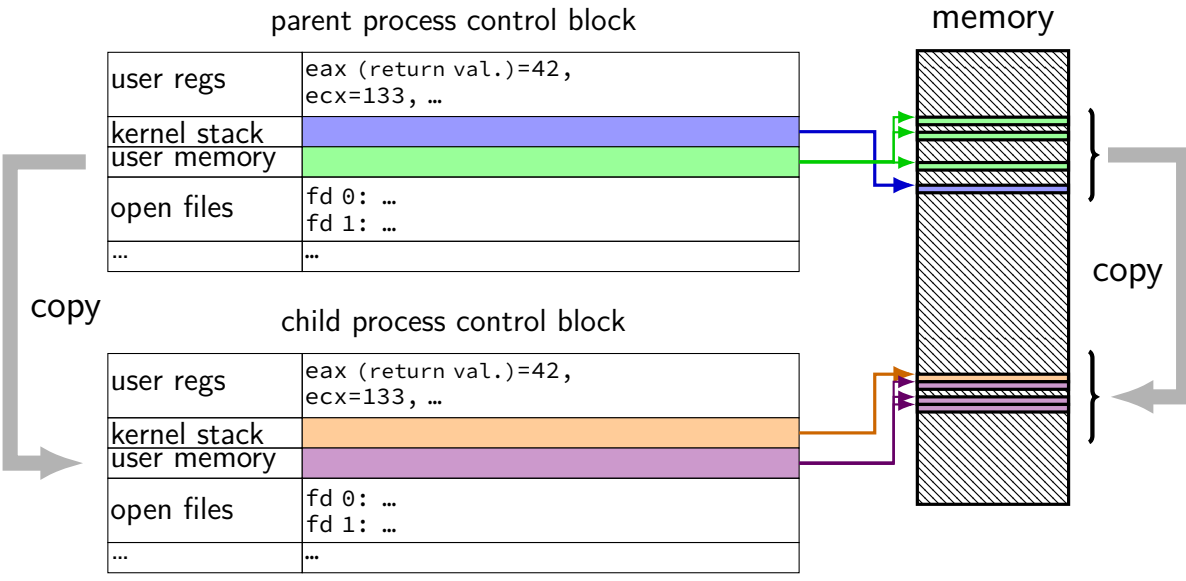
fork and PCBs



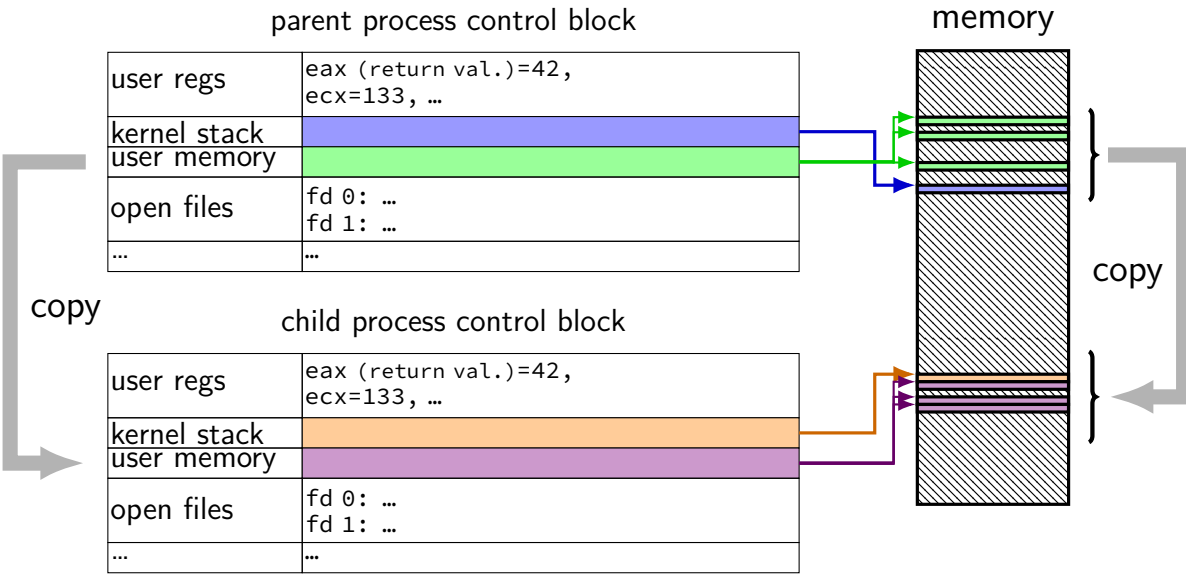
fork and PCBs



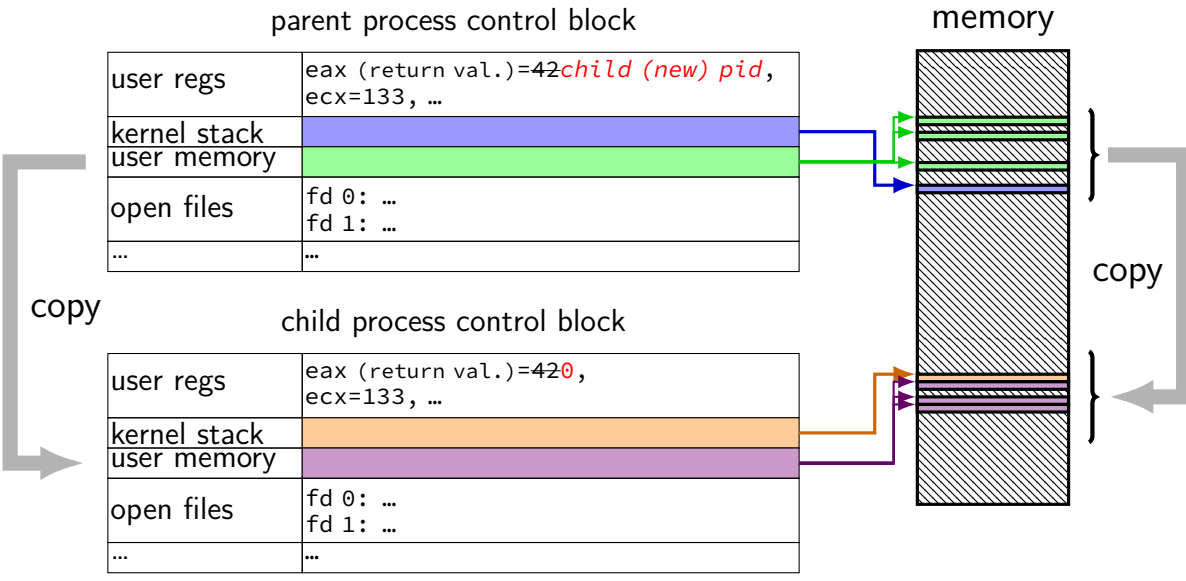
fork and PCBs



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

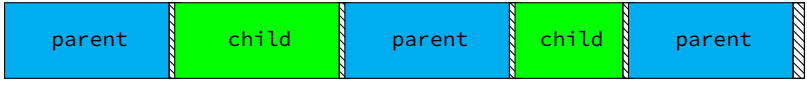
```
int main() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("In child\n");
    } else {
        printf("Child %d\n", pid);
    }
    printf("Done!\n");
}
```

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

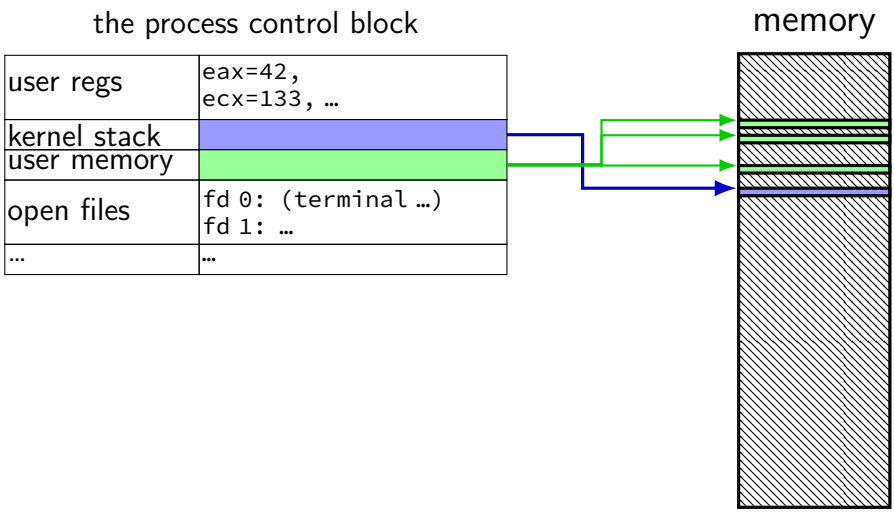
execv example

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

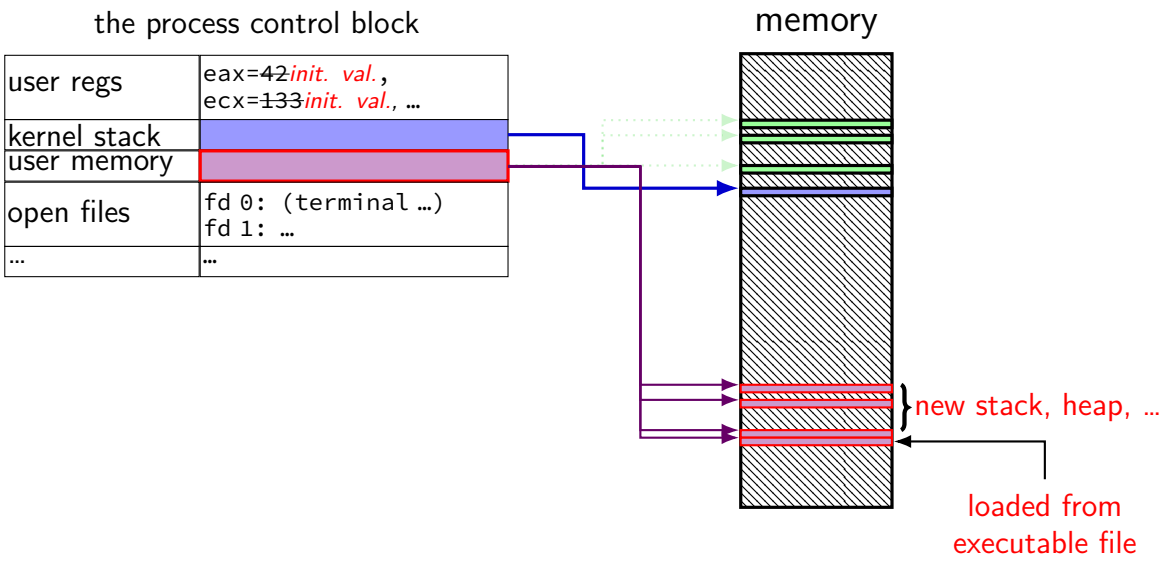
path of executable to run
need not match first argument
(but probably should match it)

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

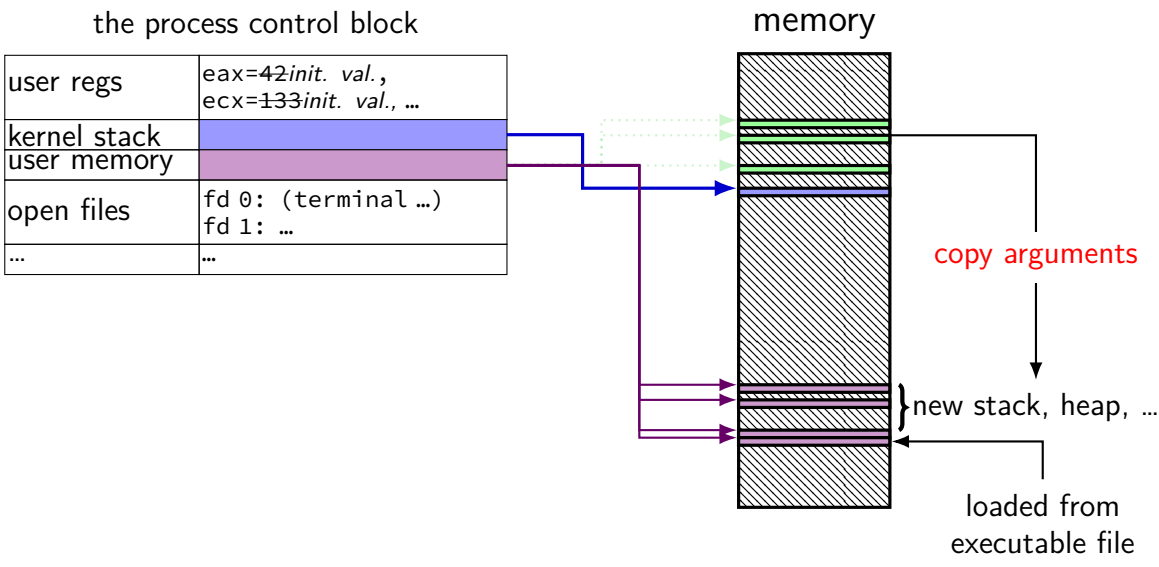
exec and PCBs



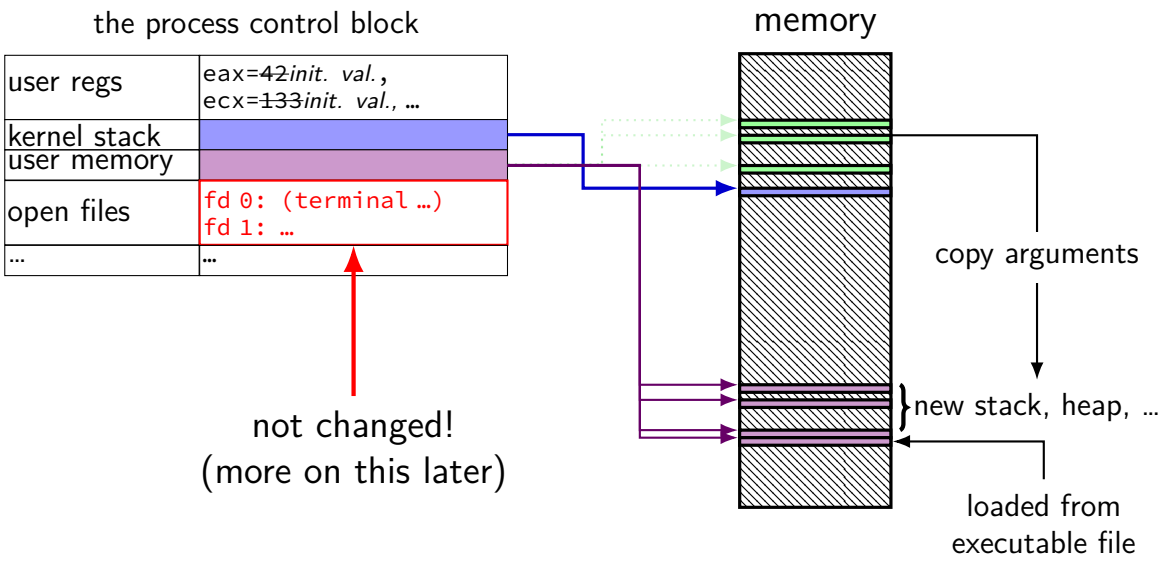
exec and PCBs



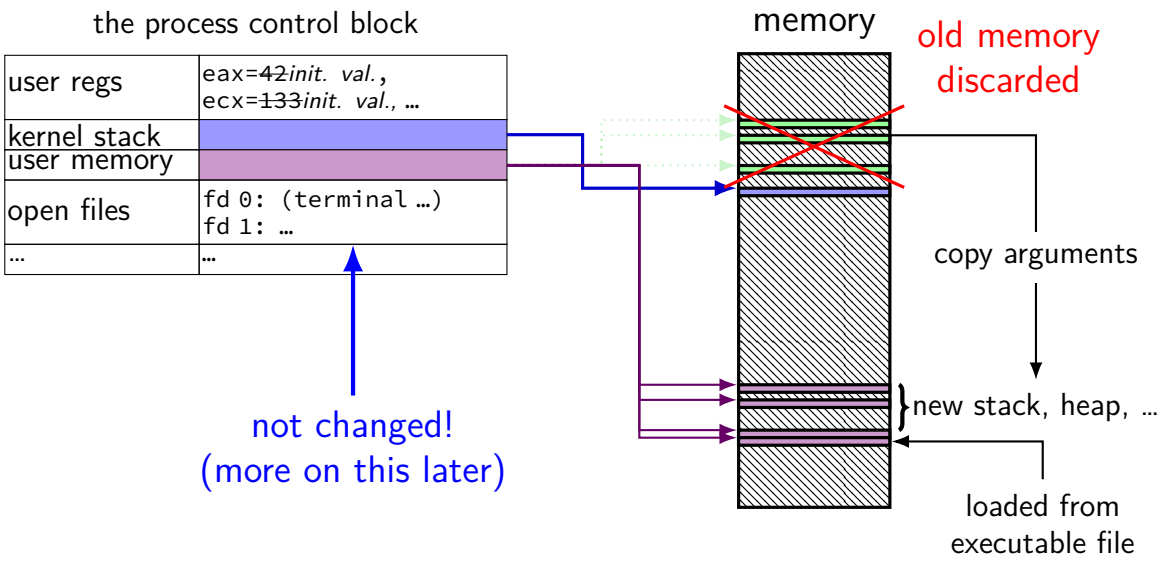
exec and PCBs



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

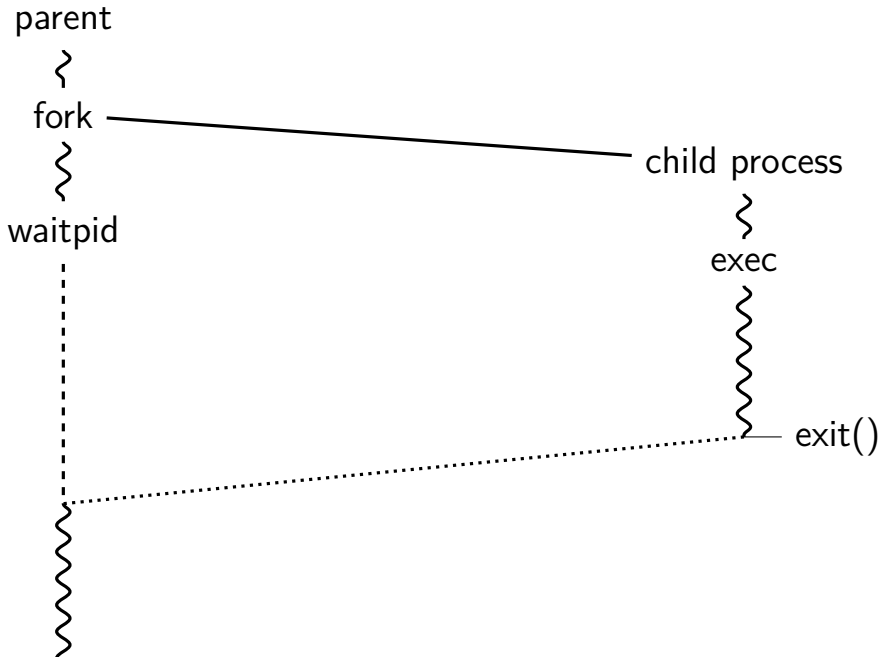
wait's status will tell you when and what signal killed a program

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

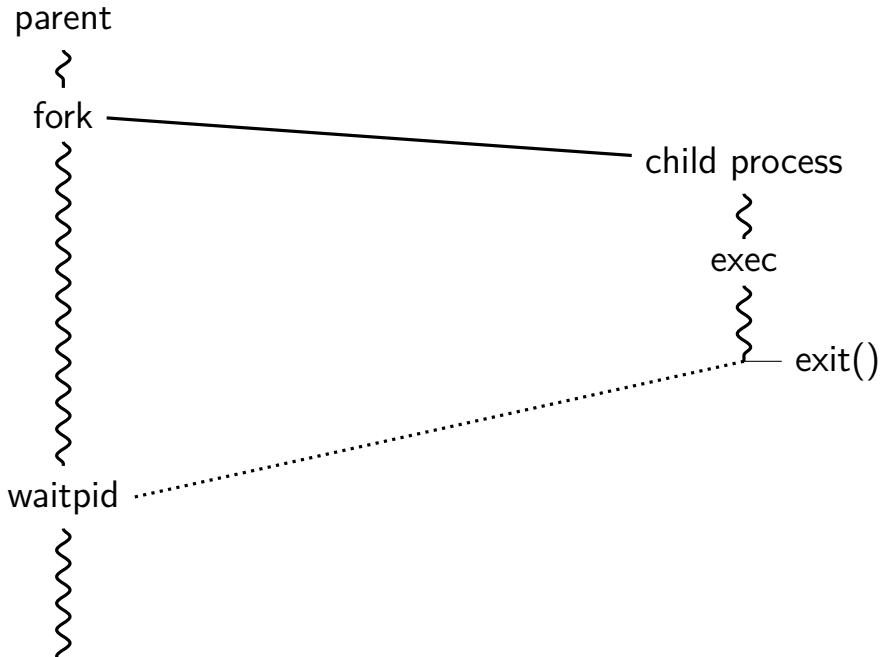
waiting for all children

```
#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 */
  }
```

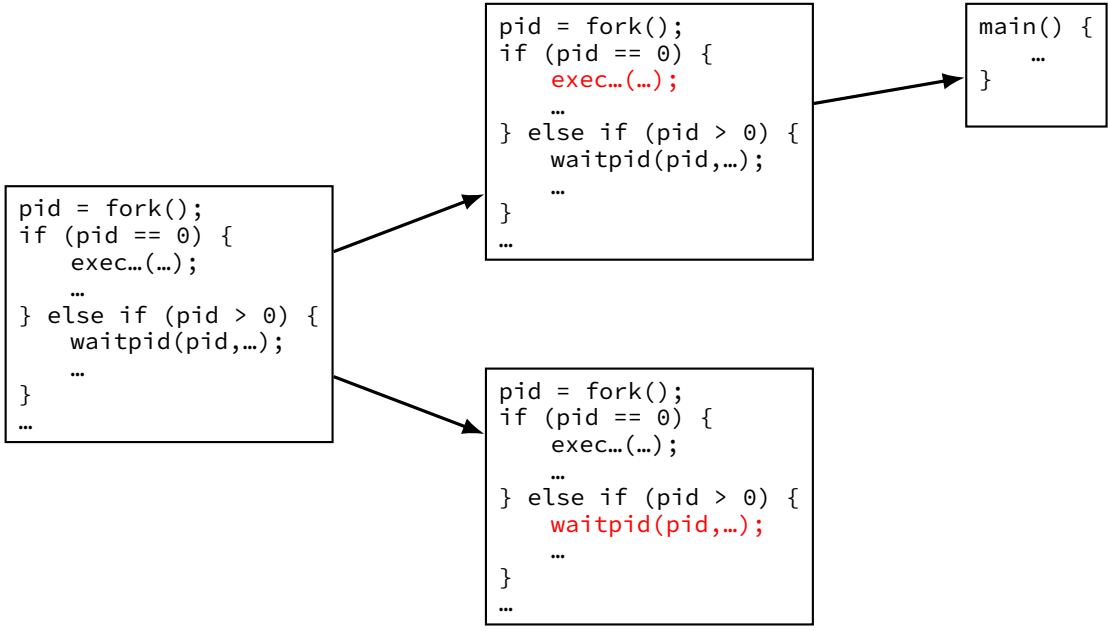
typical pattern



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);
}
```

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

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

backup slides

context switch in xv6

will mostly talk about *kernel thread switch*:

xv6 function: `swtch()`

save kernel registers for A, restore for B

in xv6: *separate from saving/restoring user registers*

- one of many possible OS design choices

additional process switch pieces: (*switchuvm()*)

- changing address space (page tables)
- telling processor new stack pointer for exceptions

swtch prototype

```
void swtch(struct context **old, struct context *new);
```

save current context into `*old`

start running context from `new`

swtch prototype

```
void swtch(struct context **old, struct context *new);
```

save current context into `*old`

start running context from `new`

trick: `struct context*` = thread's stack pointer

top of stack contains saved registers, etc.

thread switching in xv6: C

in thread A:

```
/* switch from A to B */

... // (1)
swtch(&(a->context), b->context);  /* returns to (2) */
... // (4)
```

---

in thread B:

```
swtch(...);  // (0) -- called earlier
... // (2)
...
/* later on switch back to A */
... // (3)
swtch(&(b->context), a->context)  /* returns to (4) */
...
```

thread switching in xv6: C

in thread A:

```
/* switch from A to B */

... // (1)
swtch(&(a->context), b->context);  /* returns to (2) */
... // (4)
```

---

in thread B:

```
swtch(...);  // (0) -- called earlier
... // (2)
...
/* later on switch back to A */
... // (3)
swtch(&(b->context), a->context)  /* returns to (4) */
...
```

thread switching in xv6: C

in thread A:

```
/* switch from A to B */

... // (1)
swtch(&(a->context), b->context);  /* returns to (2) */
... // (4)
```

---

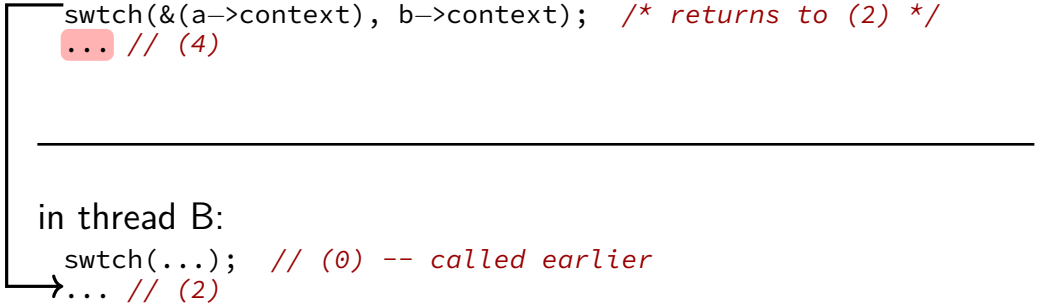
in thread B:

```
swtch(...);  // (0) -- called earlier
... // (2)
...
/* later on switch back to A */
... // (3)
swtch(&(b->context), a->context)  /* returns to (4) */
...
```

thread switching in xv6: C

in thread A:

```
/* switch from A to B */

... // (1)
swtch(&(a->context), b->context);  /* returns to (2) */
... // (4)
```



in thread B:

```
swtch(...);  // (0) -- called earlier
... // (2)
...
/* later on switch back to A */
... // (3)
swtch(&(b->context), a->context)  /* returns to (4) */
...
```

thread switching in xv6: C

in thread A:

```
/* switch from A to B */

... // (1)
swtch(&(a->context), b->context);  /* returns to (2) */
... // (4)
```

---

in thread B:

```
swtch(...);  // (0) -- called earlier
... // (2)
...
/* later on switch back to A */
... // (3)
swtch(&(b->context), a->context)  /* returns to (4) */
...
```

thread switching in xv6: C

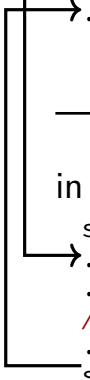
in thread A:

```
/* switch from A to B */

... // (1)
swtch(&(a->context), b->context);  /* returns to (2) */
... // (4)
```

---

in thread B:

```
swtch(...);  // (0) -- called earlier
... // (2)
...
/* later on switch back to A */
... // (3)
swtch(&(b->context), a->context)  /* returns to (4) */
...
```



thread switching in xv6: how?

swtch(A, B) pseudocode:

save caller-saved registers to stack

write swtch return address to stack

write all callee-saved registers to stack

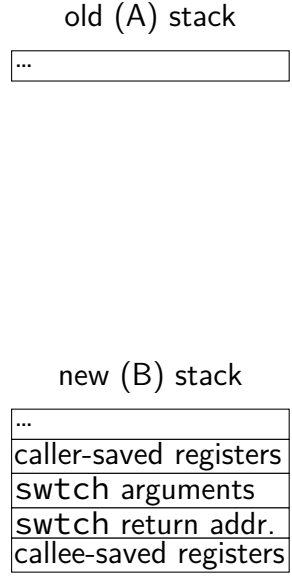
save old stack pointer into arg *A*

read *B* arg as new stack pointer

read all callee-saved registers from stack

read+use swtch return address from stack

restore caller-saved registers from stack



thread switching in xv6: how?

swtch(A, B) pseudocode:

save caller-saved registers to **stack**

write swtch return address to **stack** (x86 `call`)

write all callee-saved registers to **stack**

save old **stack** pointer into arg *A*

read *B* arg as new *stack* pointer

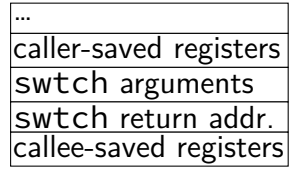
read all callee-saved registers from *stack*

read+use swtch return address from *stack* (x86 `ret`)

restore caller-saved registers from *stack*

old (A) **stack**

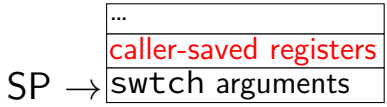
new (B) *stack*



thread switching in xv6: how?

swtch(A, B) pseudocode:

old (A) **stack**



save caller-saved registers to **stack**

write swtch return address to **stack** (x86 `call`)

write all callee-saved registers to **stack**

save old **stack** pointer into arg *A*

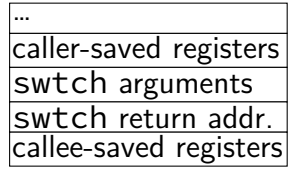
read *B* arg as new *stack* pointer

read all callee-saved registers from *stack*

read+use swtch return address from *stack* (x86 `ret`)

restore caller-saved registers from *stack*

new (B) *stack*



thread switching in xv6: how?

swtch(A, B) pseudocode:

save caller-saved registers to **stack**

write swtch return address to **stack** (x86 `call`)

write all callee-saved registers to **stack**

save old **stack** pointer into arg *A*

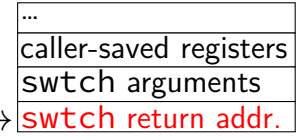
read *B* arg as new *stack* pointer

read all callee-saved registers from *stack*

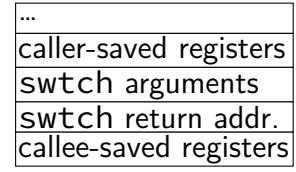
read+use swtch return address from *stack* (x86 `ret`)

restore caller-saved registers from *stack*

old (A) **stack**



new (B) *stack*



thread switching in xv6: how?

swtch(A, B) pseudocode:

save caller-saved registers to **stack**

write swtch return address to **stack** (x86 `call`)

write all callee-saved registers to **stack**

save old **stack** pointer into arg *A*

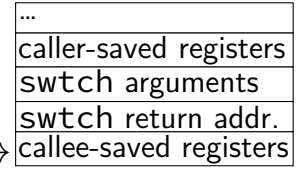
read *B* arg as new *stack* pointer

read all callee-saved registers from *stack*

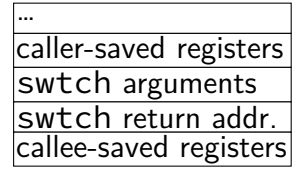
read+use swtch return address from *stack* (x86 `ret`)

restore caller-saved registers from *stack*

old (A) **stack**



new (B) *stack*



thread switching in xv6: how?

swtch(A, B) pseudocode:

save caller-saved registers to **stack**

write swtch return address to **stack** (x86 `call`)

write all callee-saved registers to **stack**

save old **stack** pointer into arg *A*

read *B* arg as new *stack* pointer

read all callee-saved registers from *stack*

read+use swtch return address from *stack* (x86 `ret`)

restore caller-saved registers from *stack*

old (A) **stack**

SP →

...
caller-saved registers
`swtch` arguments
`swtch` return addr.
callee-saved registers

new (B) *stack*

...
caller-saved registers
`swtch` arguments
`swtch` return addr.
callee-saved registers

thread switching in xv6: how?

swtch(A, B) pseudocode:

save caller-saved registers to **stack**

write swtch return address to **stack** (x86 `call`)

write all callee-saved registers to **stack**

save old **stack** pointer into arg *A*

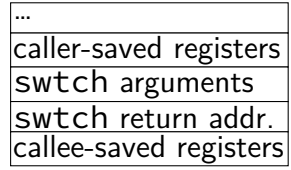
read *B* arg as new *stack* pointer

read all callee-saved registers from *stack*

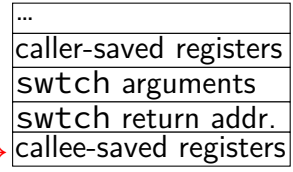
read+use swtch return address from *stack* (x86 `ret`)

restore caller-saved registers from *stack*

old (A) **stack**



new (B) *stack*



thread switching in xv6: how?

swtch(A, B) pseudocode:

save caller-saved registers to **stack**

write swtch return address to **stack** (x86 `call`)

write all callee-saved registers to **stack**

save old **stack** pointer into arg *A*

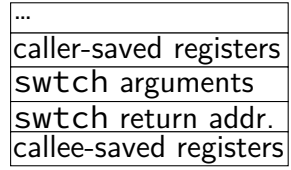
read *B* arg as new *stack* pointer

read all callee-saved registers from *stack*

read+use swtch return address from *stack* (x86 `ret`)

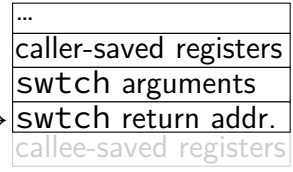
restore caller-saved registers from *stack*

old (A) **stack**



new (B) *stack*

SP →



thread switching in xv6: how?

swtch(A, B) pseudocode:

save caller-saved registers to **stack**

write swtch return address to **stack** (x86 `call`)

write all callee-saved registers to **stack**

save old **stack** pointer into arg *A*

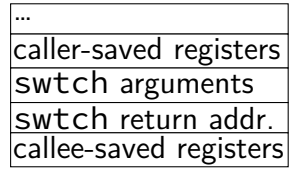
read *B* arg as new *stack* pointer

read all callee-saved registers from *stack*

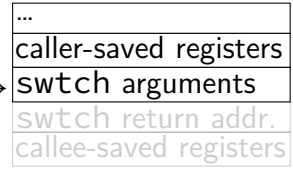
read+use swtch return address from *stack* (x86 `ret`)

restore caller-saved registers from *stack*

old (A) **stack**



new (B) *stack*



SP →

thread switching in xv6: how?

swtch(A, B) pseudocode:

save caller-saved registers to **stack**

write swtch return address to **stack** (x86 `call`)

write all callee-saved registers to **stack**

save old **stack** pointer into arg *A*

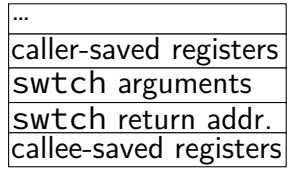
read *B* arg as new *stack* pointer

read all callee-saved registers from *stack*

read+use swtch return address from *stack* (x86 `ret`)

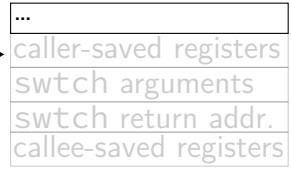
restore caller-saved registers from *stack*

old (A) **stack**



new (B) *stack*

SP →



thread switching in xv6: how?

swtch(A, B) pseudocode:

save caller-saved registers to **stack**

write swtch return address to **stack** (x86 `call`)

write all callee-saved registers to **stack**

save old **stack** pointer into arg *A*

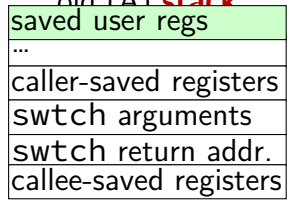
read *B* arg as new *stack* pointer

read all callee-saved registers from *stack*

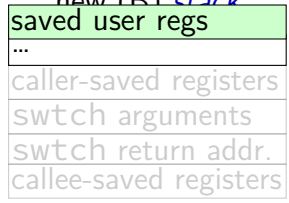
read+use swtch return address from *stack* (x86 `ret`)

restore caller-saved registers from *stack*

old (A) **stack**



new (B) *stack*



thread switching in xv6: assembly

```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

thread switching in xv6: assembly

```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

two arguments:
`struct context **from_context`
= where to save current context
`struct context *to_context`
= where to find new context

context stored on thread's stack
context address = top of stack

thread switching in xv6: assembly

callee-saved registers: ebp, ebx, esi, edi

```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

thread switching in xv6: assembly

```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

other parts of context?
eax, ecx, ...: saved by swtch's caller
esp: same as address of context
program counter: saved by `call` of swtch

thread switching in xv6: assembly

```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

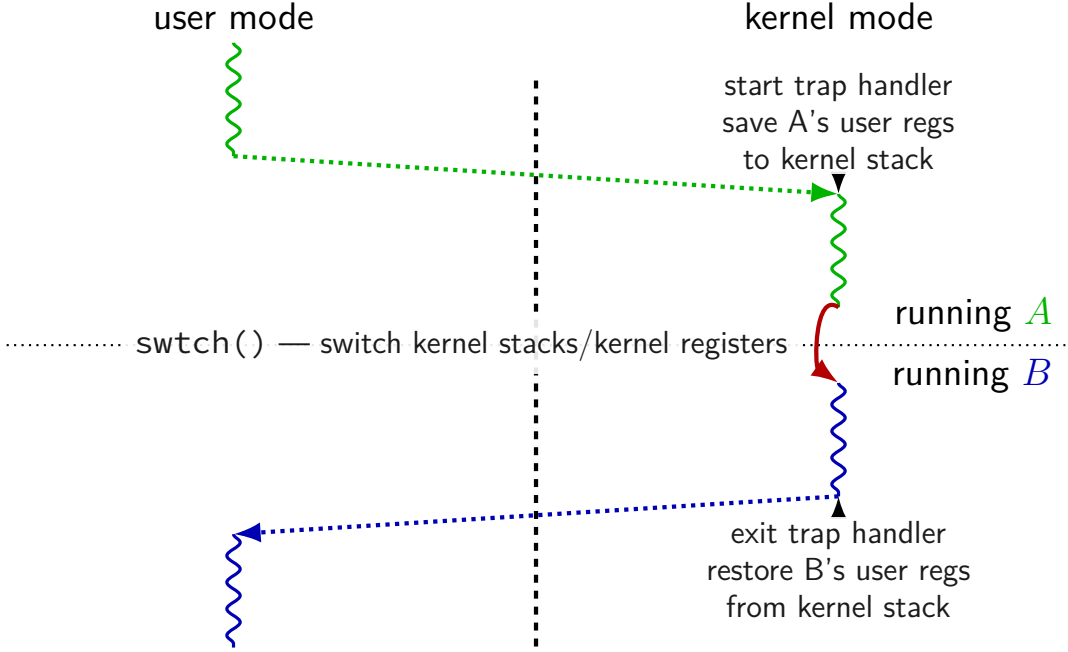
save stack pointer to first argument
(stack pointer now has all info)
restore stack pointer from second argument

thread switching in xv6: assembly

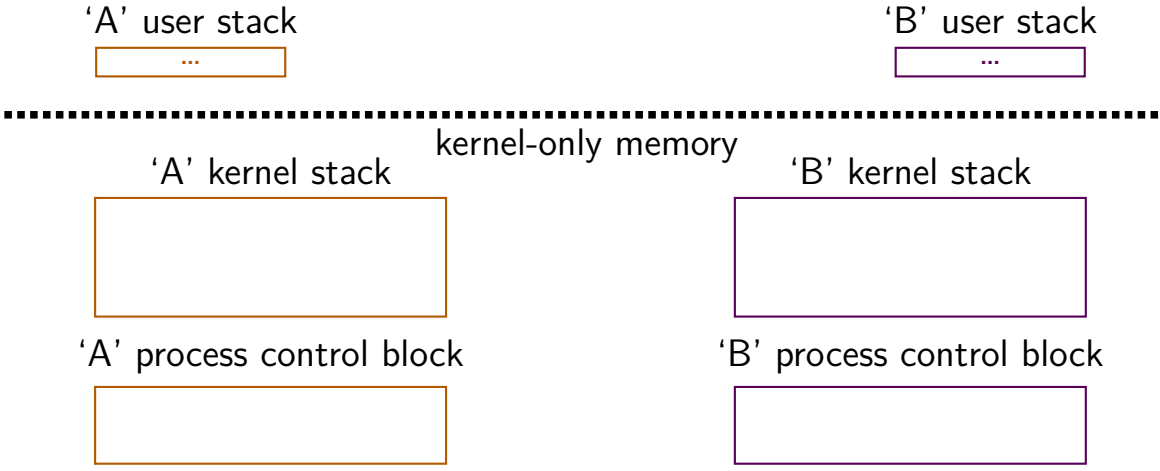
```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

restore program counter
(and other saved registers)
from stack of new thread

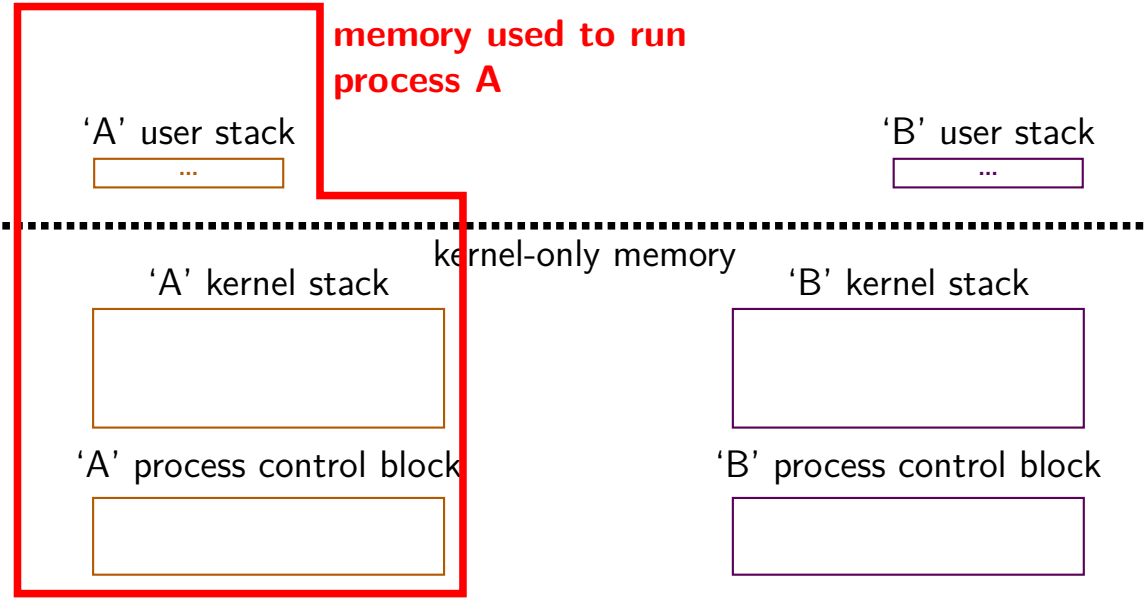
xv6 context switch and saving



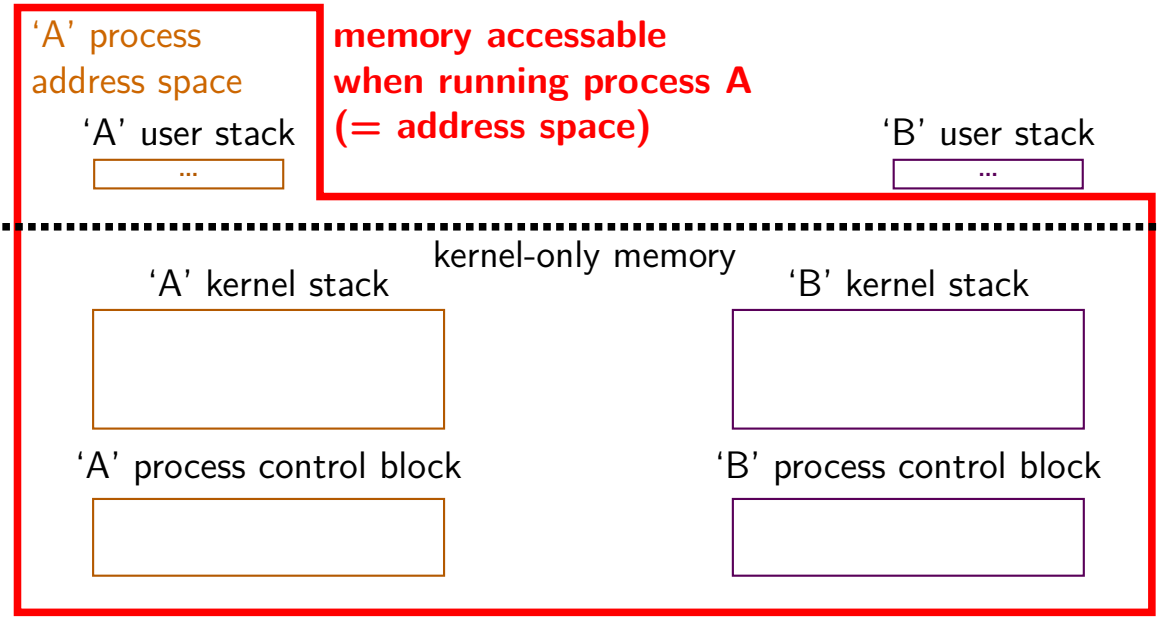
xv6: where the context is



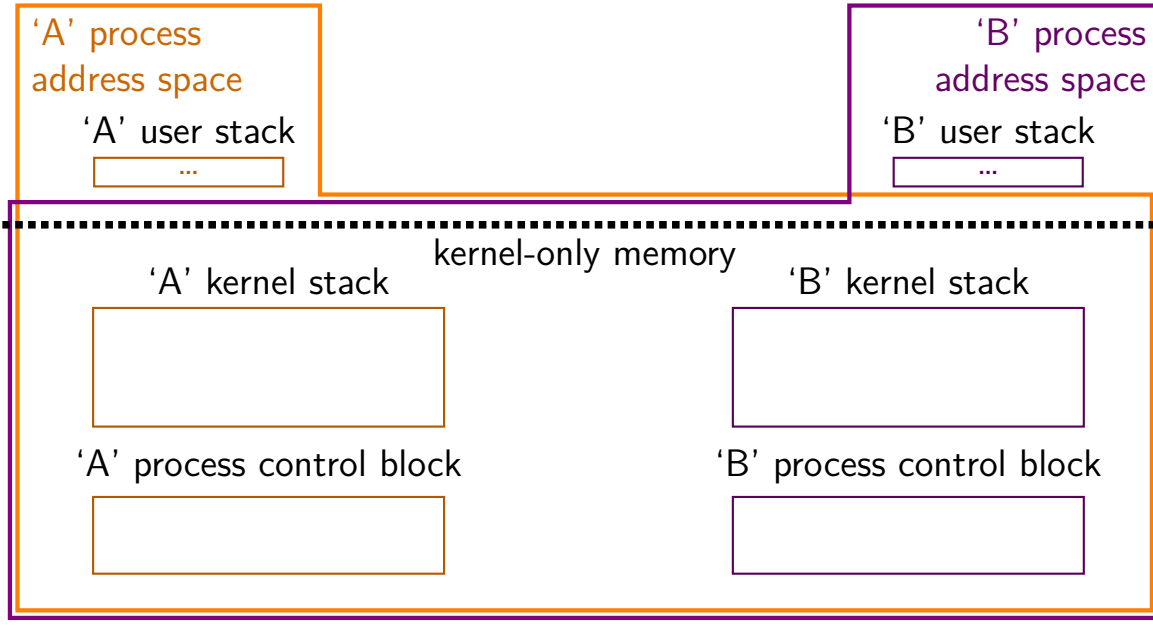
xv6: where the context is



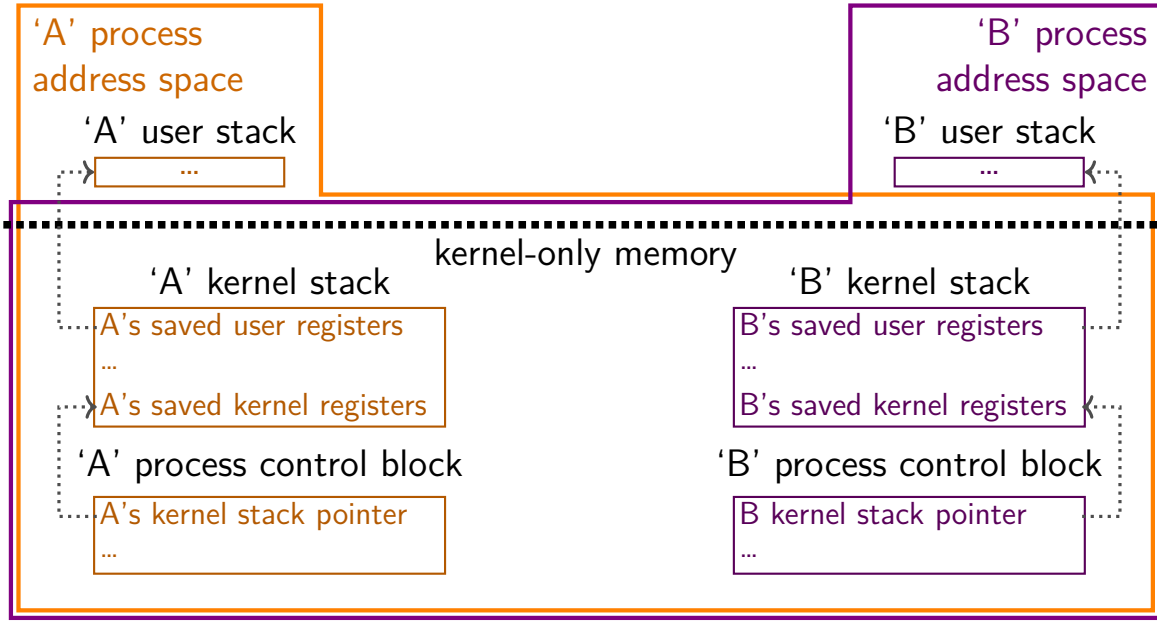
xv6: where the context is



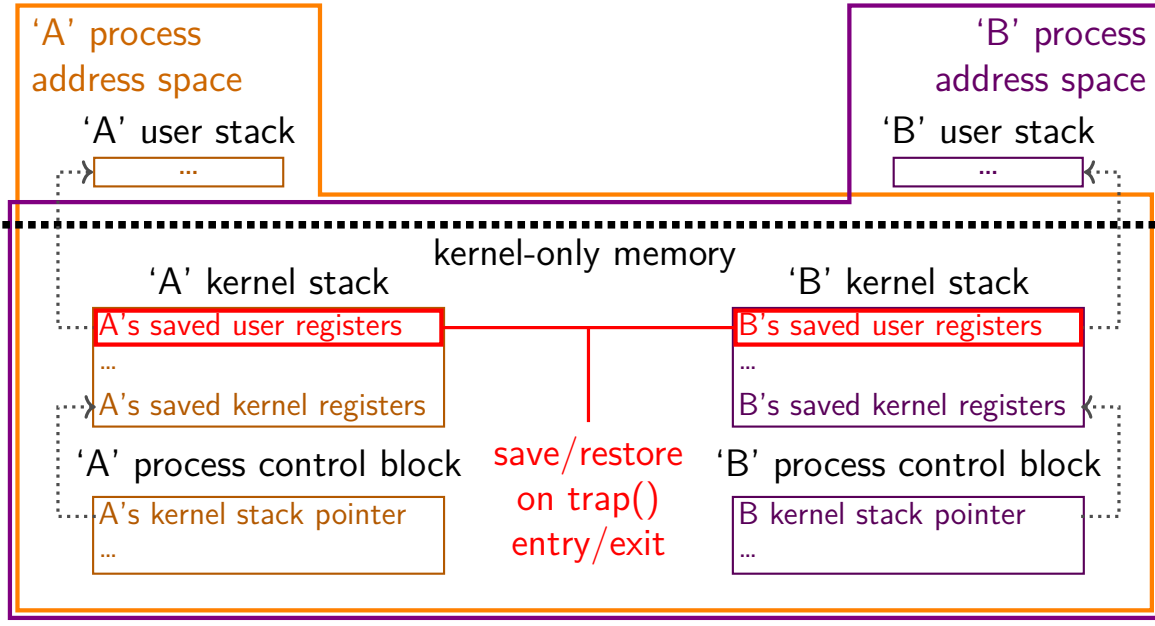
xv6: where the context is



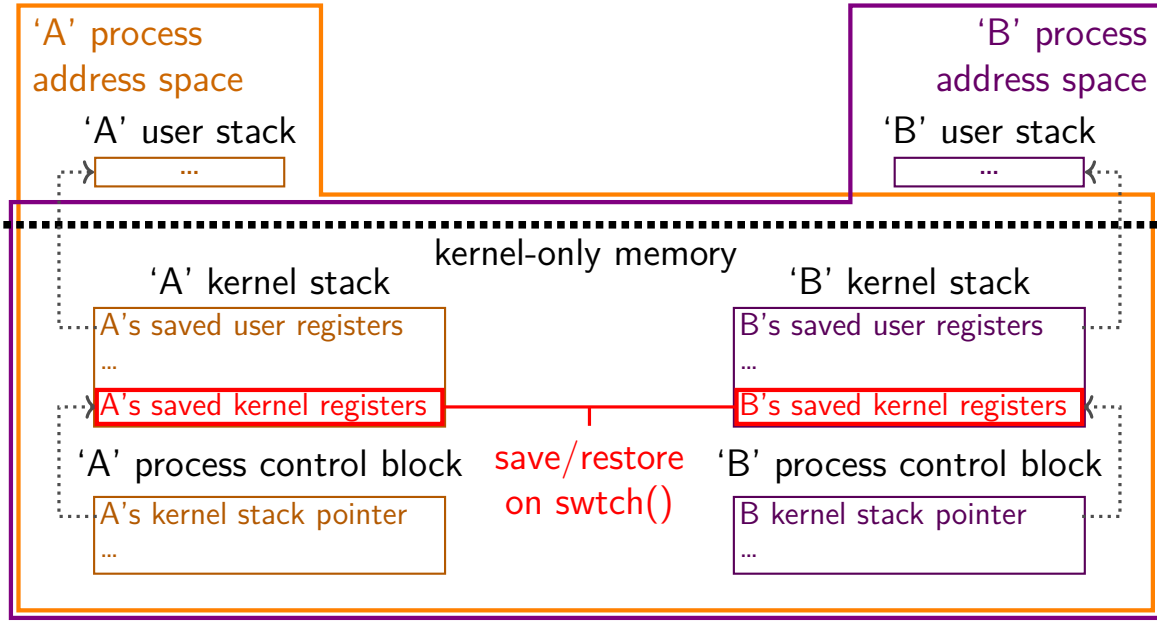
xv6: where the context is



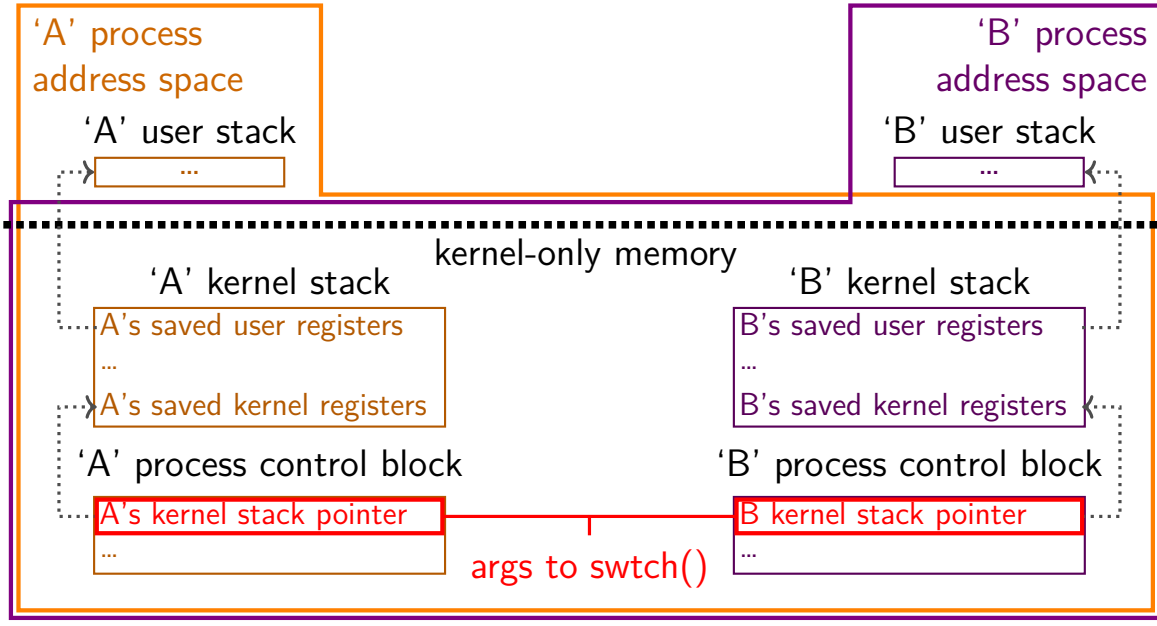
xv6: where the context is



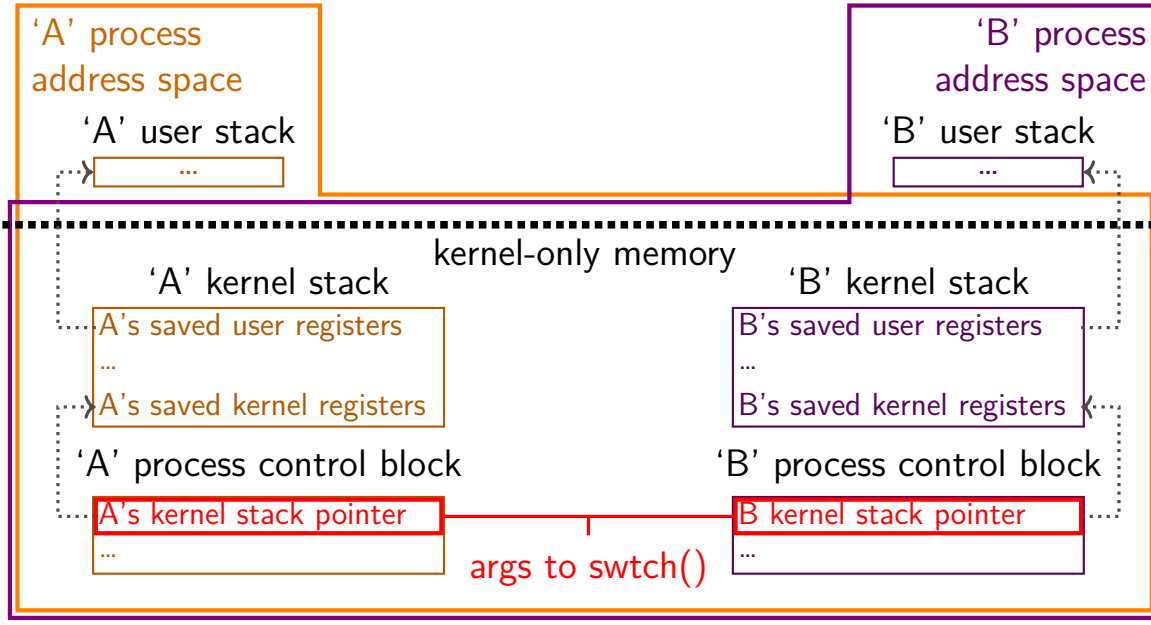
xv6: where the context is



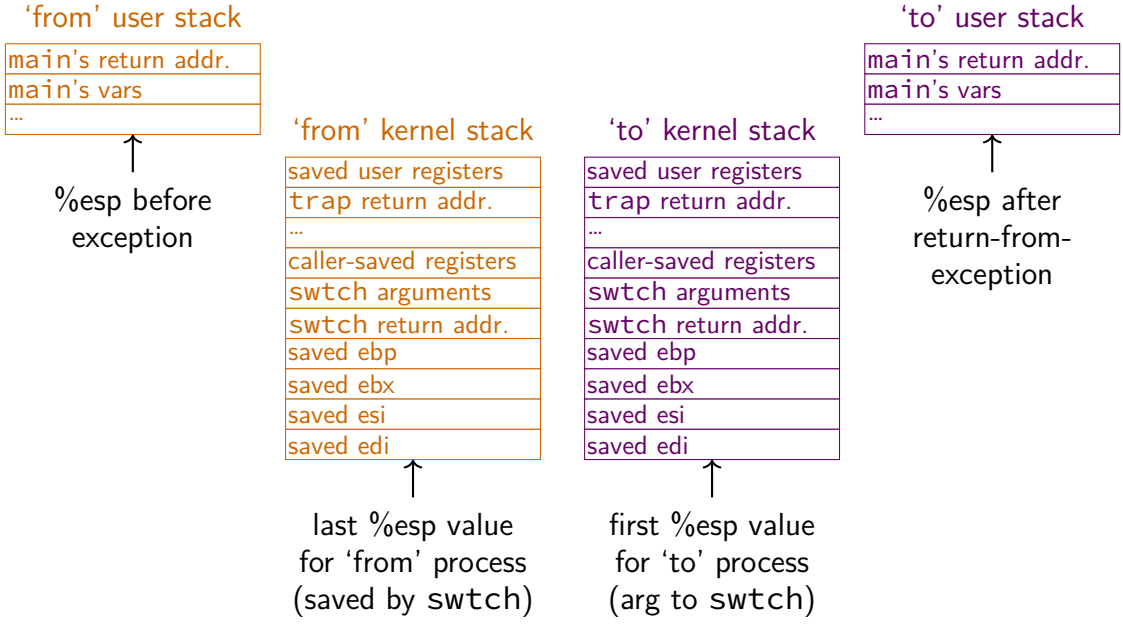
xv6: where the context is



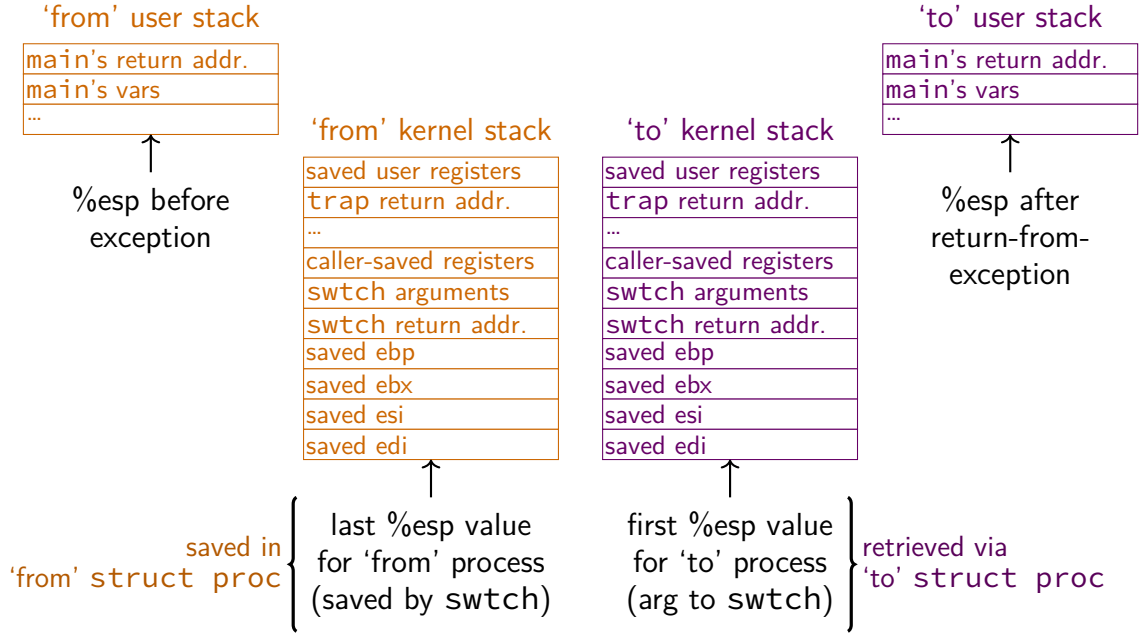
xv6: where the context is



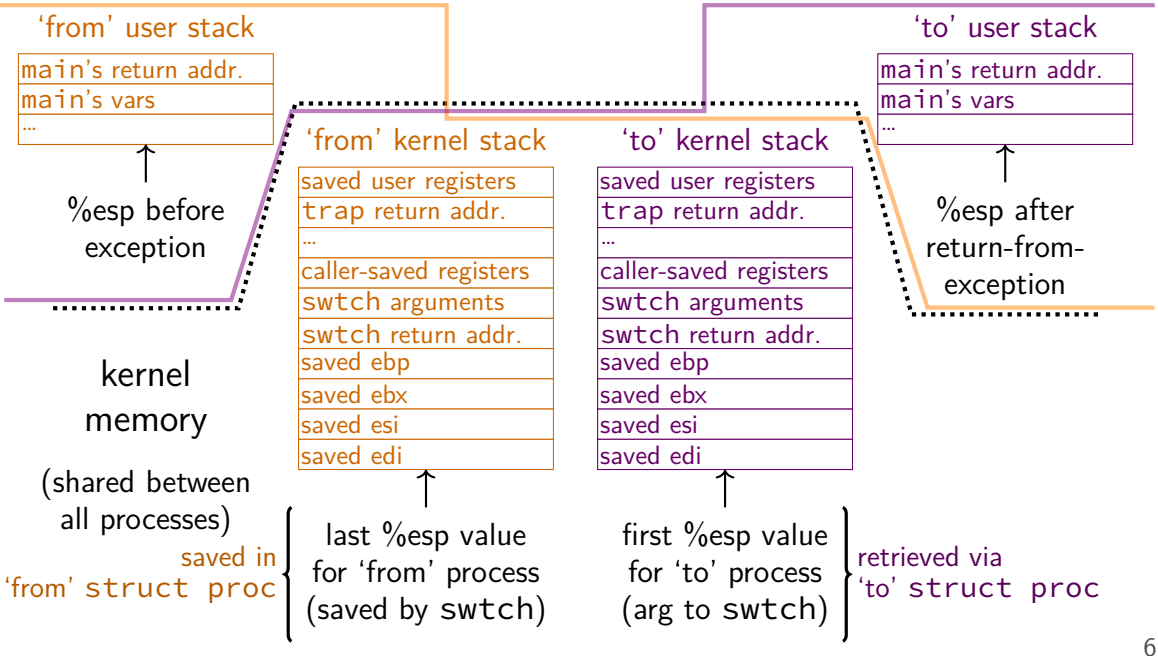
xv6: where the context is (detail)



xv6: where the context is (detail)



xv6: where the context is (detail)



aside: environment variables (1)

key=value pairs associated with every process:

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

aside: environment variables (2)

environment variable library functions:

`getenv("KEY")` → *value*

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

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

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

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

normal exec versions — keep same environment variables

aside: environment variables (3)

interpretation up to programs, but common ones...

`PATH=/bin:/usr/bin`

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

`HOME=/zf14/cr4bd`

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

`TERM=screen-256color`

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

...

‘waiting’ without waiting

```
#include <sys/wait.h>
...
  pid_t return_value = waitpid(child_pid, &status, WNOHANG);
  if (return_value == (pid_t) 0) {
    /* child process not done yet */
  } else if (child_pid == (pid_t) -1) {
    /* error */
  } else {
    /* handle child_pid exiting */
  }
```

running in background

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

& — run a program in "background"

initially output PID (above: 4049)

print out after terminated

- one way: use `waitpid` with option saying "don't wait"

execv and const

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

`argv` is a pointer to constant pointer to char

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

…this causes some awkwardness:

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

solution: cast

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