context switches / process management

last time

user/kernel mode

- kernel mode: for OS: I/O, access to other process's info, etc.
- user mode: those things not permitted (go through OS instead)

exceptions: switch to kernel mode, jump to OS

system call: exception triggered deliberately by program

- request OS do something on program's behalf

xv6 system calls

context switches

- save state of current thread/program
- restore state of another thread/program
- in xv6: done by OS only; need to switch to OS first (exception)
  - always switching between processes
  - can't access info about other processes in user mode
  - (but if already in kernel mode, don't need to switch again)

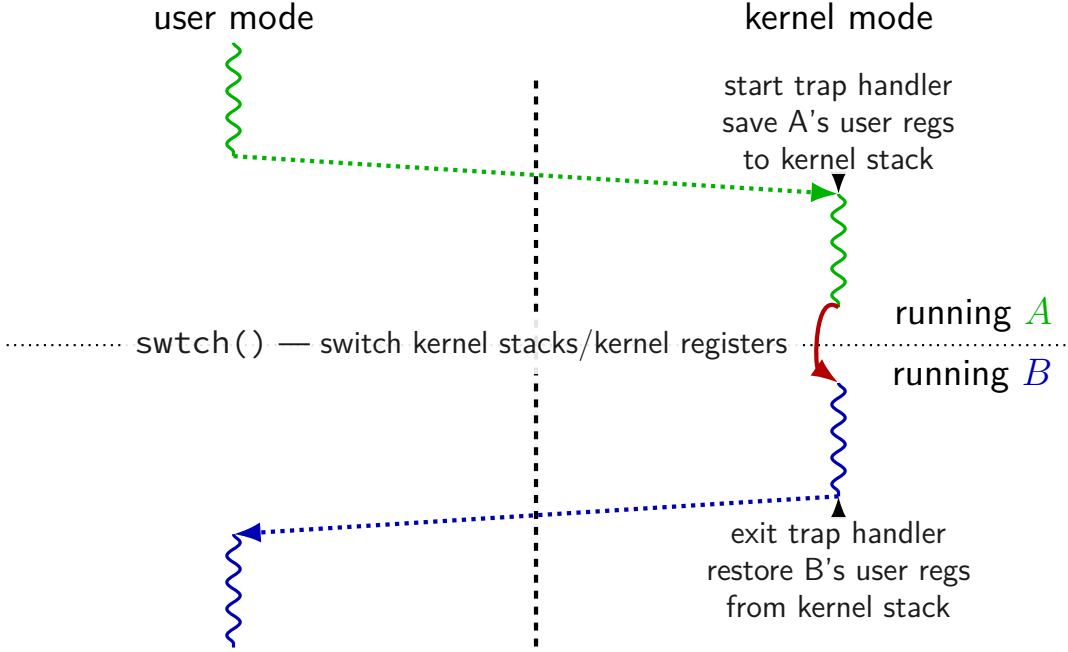
quiz reliability aside

appears the quiz site sometimes didn't give feedback when submitting logged-out of NetBadge

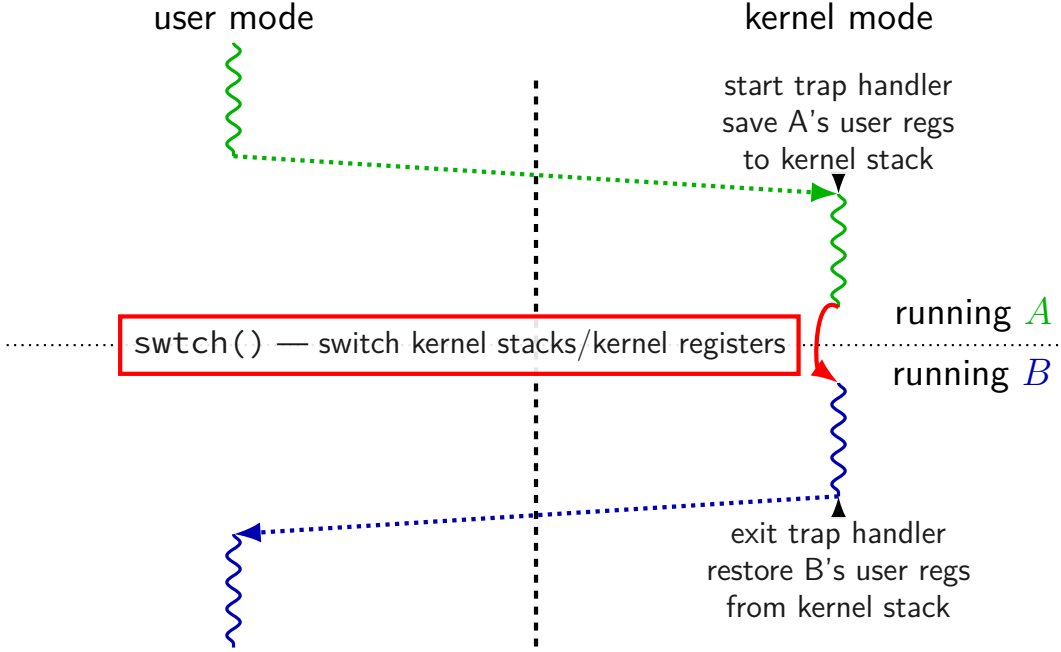
- I don't understand how, but workaround in place

let me know if this affected your quiz score

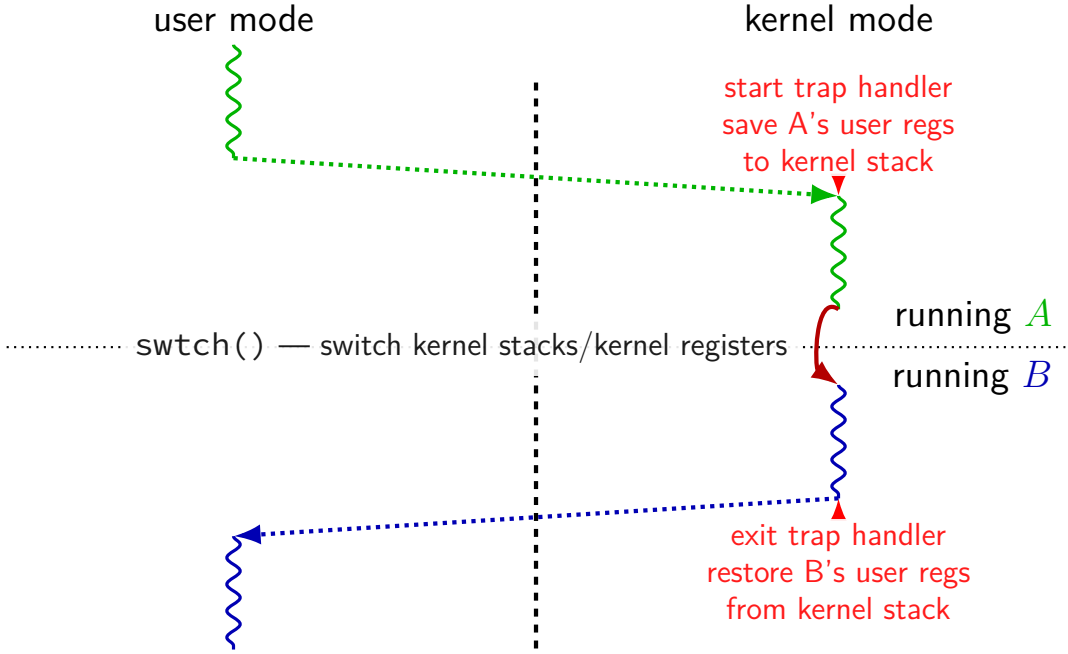
xv6 context switch and saving



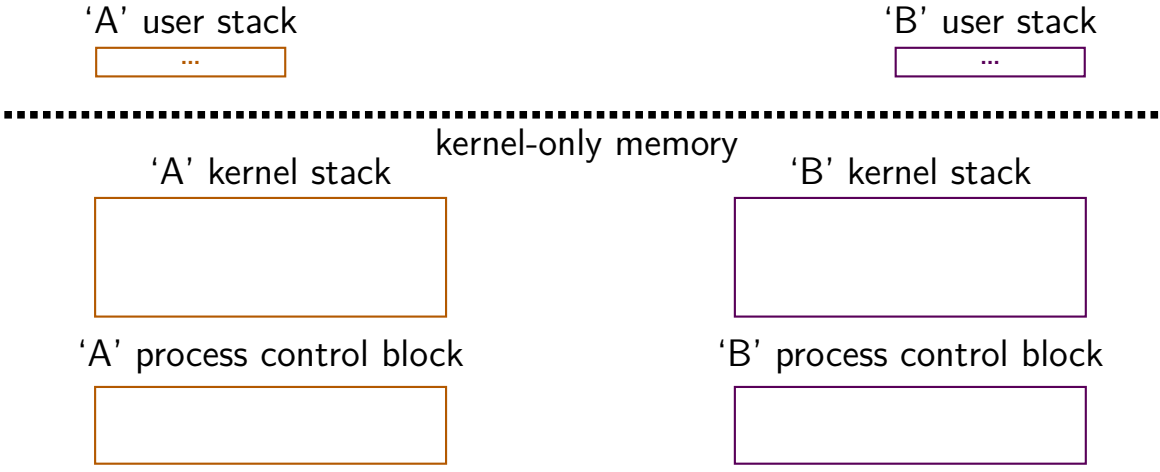
xv6 context switch and saving



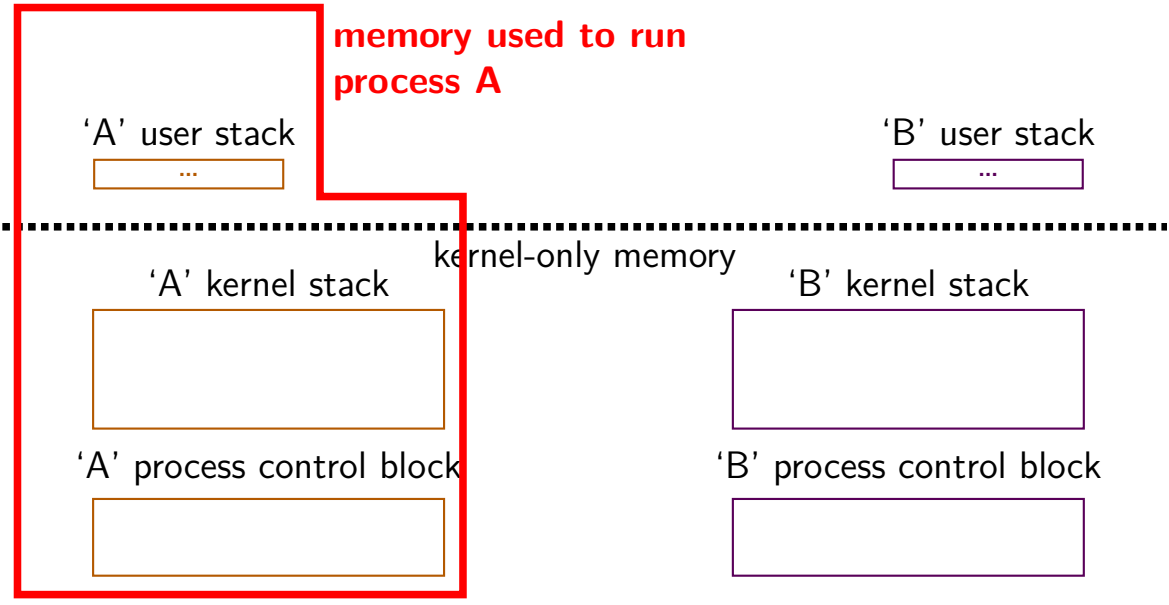
xv6 context switch and saving



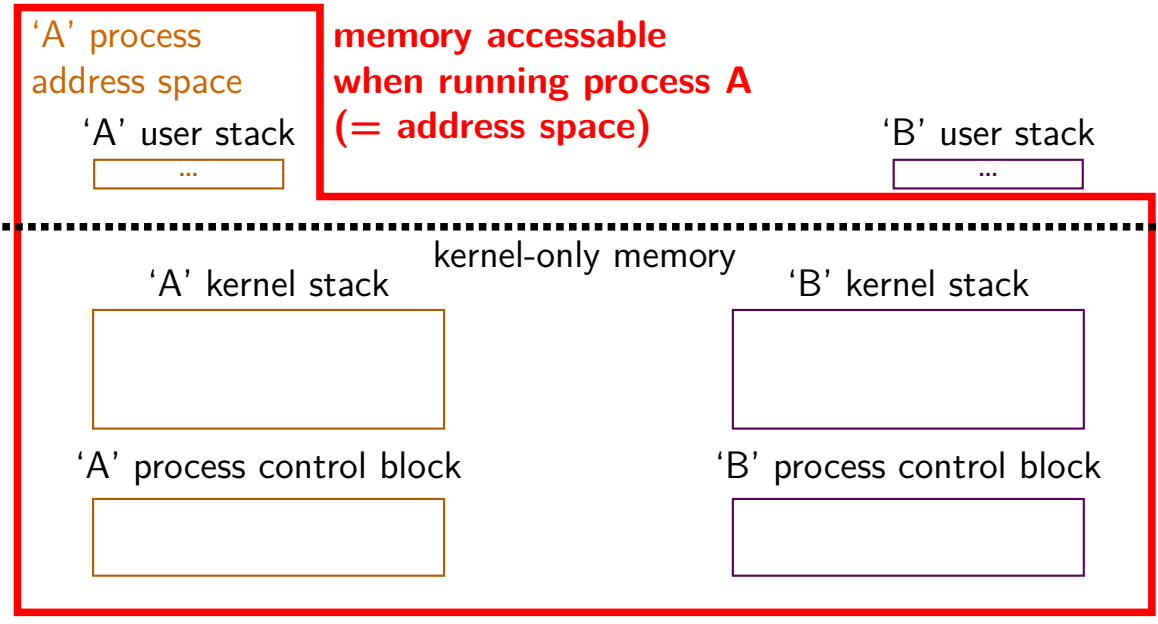
xv6: where the context is



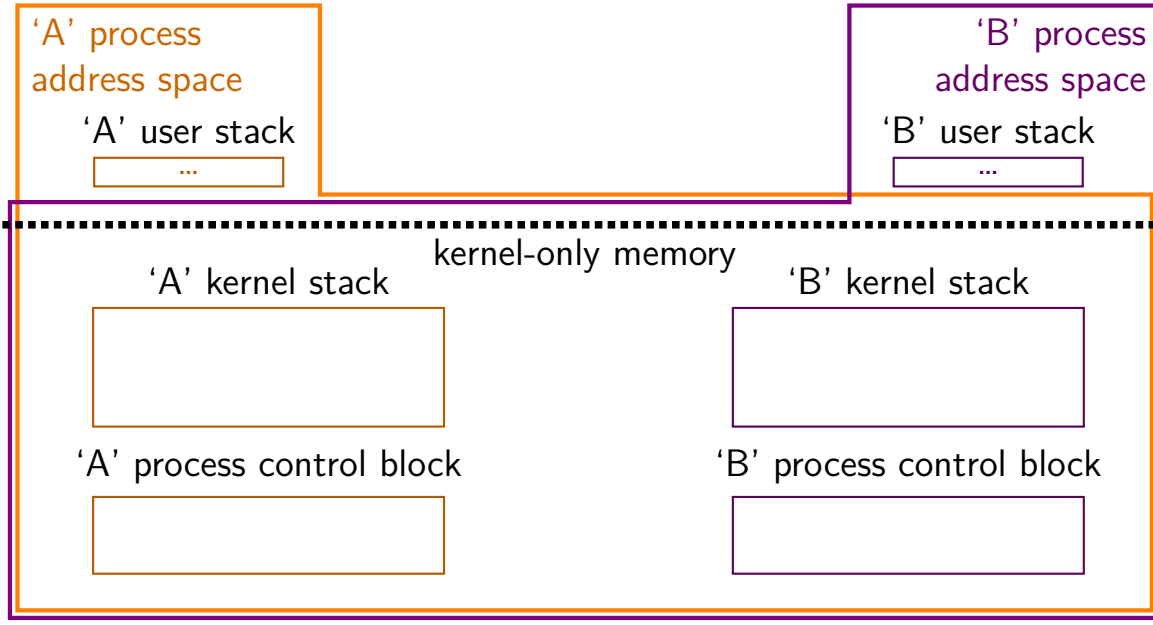
xv6: where the context is



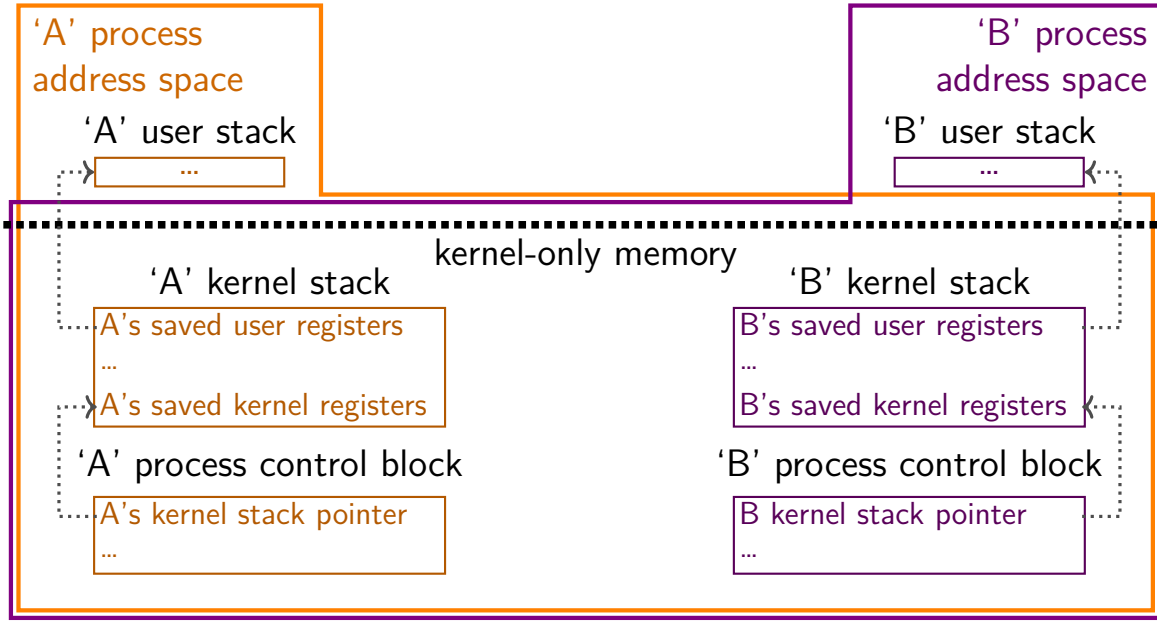
xv6: where the context is



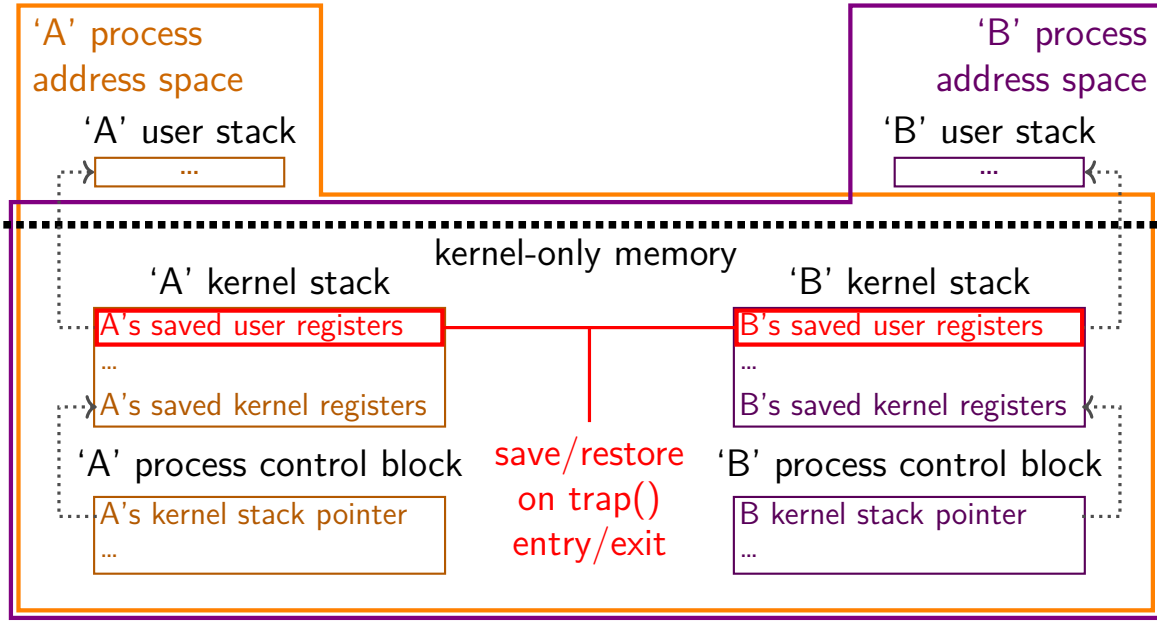
xv6: where the context is



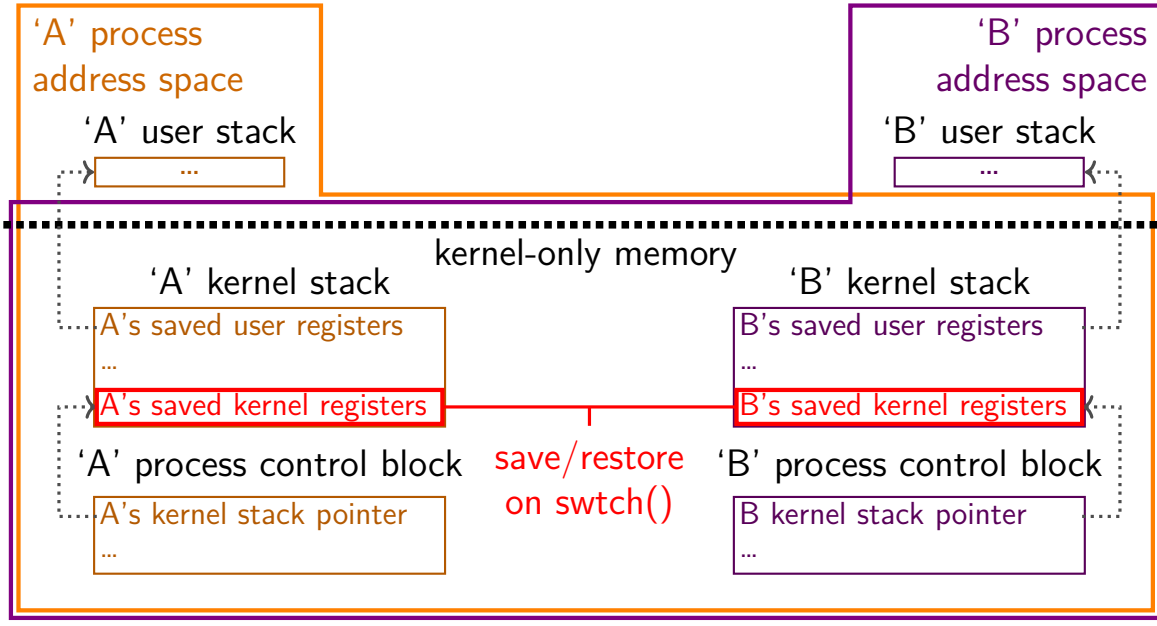
xv6: where the context is



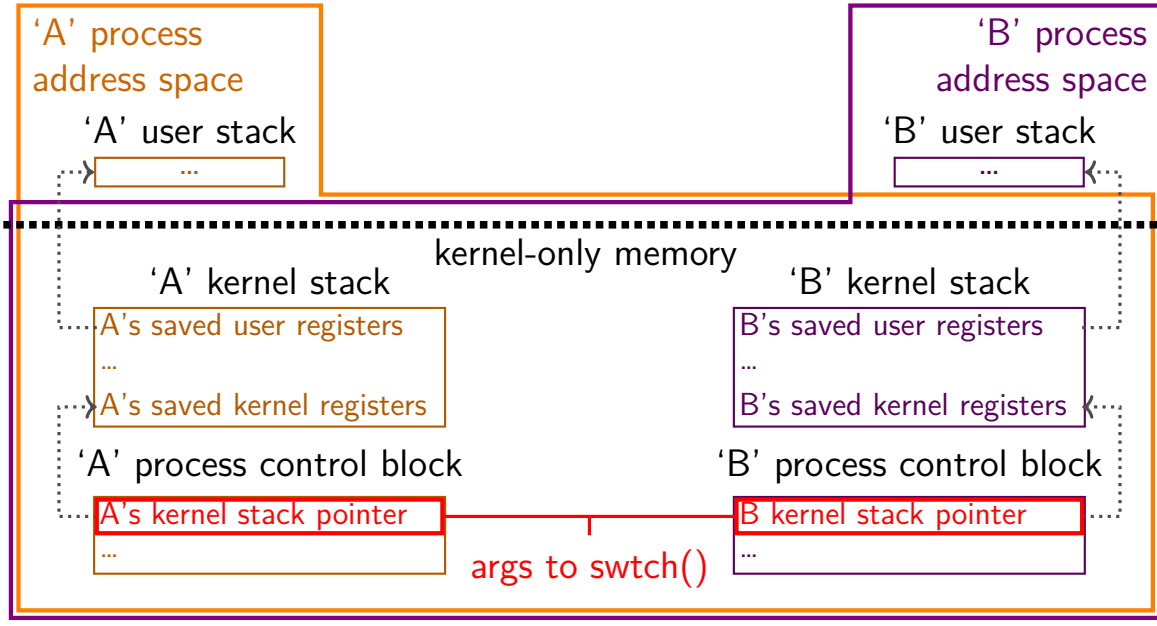
xv6: where the context is



xv6: where the context is



xv6: where the context is



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:

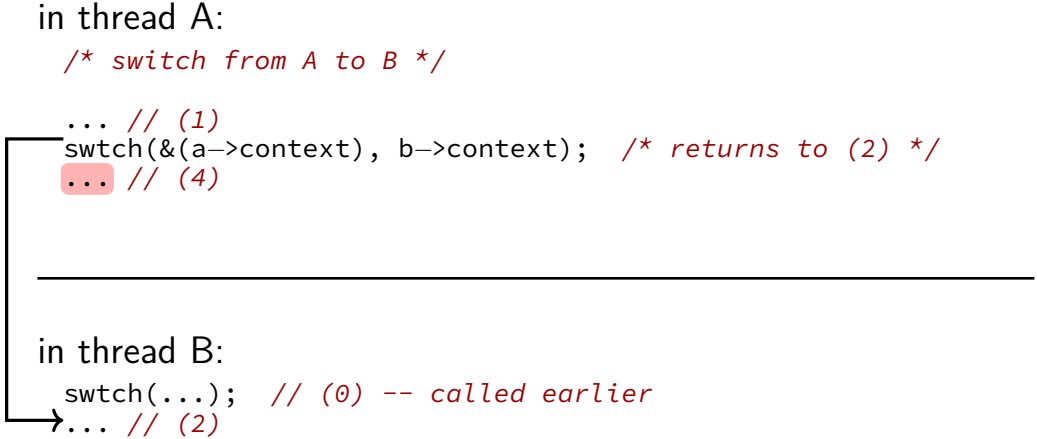
```
/* switch from A to B */

... // (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



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

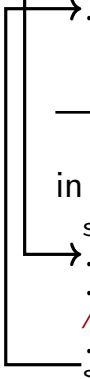
in thread A:

```
/* 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 A's caller-saved registers to stack

write swtch return address to stack

write all A's callee-saved registers to stack

save old stack pointer into arg *A*

read *B* arg as new stack pointer

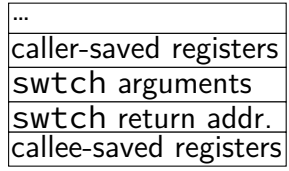
read all B's callee-saved registers from stack

read+use swtch return address from stack

restore B's caller-saved registers from stack

old (A) stack

new (B) stack



thread switching in xv6: how?

swtch(A, B) pseudocode:

save A's caller-saved registers to **stack**

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

write all A's callee-saved registers to **stack**

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

read *B* arg as new *stack* pointer

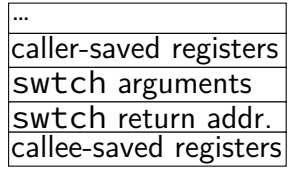
read all B's callee-saved registers from *stack*

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

restore B's caller-saved registers from *stack*

old (A) **stack**

new (B) *stack*



thread switching in xv6: how?

swtch(A, B) pseudocode:

save A's caller-saved registers to **stack**

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

write all A's callee-saved registers to **stack**

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

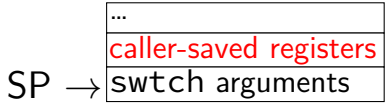
read *B* arg as new *stack* pointer

read all B's callee-saved registers from *stack*

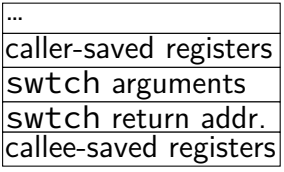
read+use swtch return address from *stack* (x86 `ret`)

restore B's caller-saved registers from *stack*

old (A) **stack**



new (B) *stack*



thread switching in xv6: how?

swtch(A, B) pseudocode:

save A's caller-saved registers to **stack**

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

write all A's callee-saved registers to **stack**

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

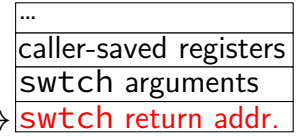
read *B* arg as new *stack* pointer

read all B's callee-saved registers from *stack*

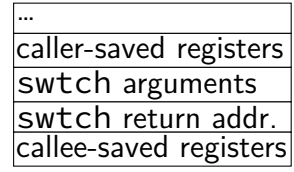
read+use swtch return address from *stack* (x86 `ret`)

restore B's caller-saved registers from *stack*

old (A) **stack**



new (B) *stack*



thread switching in xv6: how?

swtch(A, B) pseudocode:

save A's caller-saved registers to **stack**

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

write all A's callee-saved registers to **stack**

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

read *B* arg as new *stack* pointer

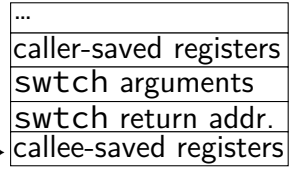
read all B's callee-saved registers from *stack*

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

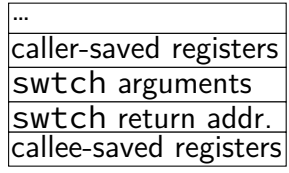
restore B's caller-saved registers from *stack*

old (A) **stack**

SP →



new (B) *stack*



thread switching in xv6: how?

swtch(A, B) pseudocode:

save A's caller-saved registers to **stack**

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

write all A's callee-saved registers to **stack**

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

read *B* arg as new *stack* pointer

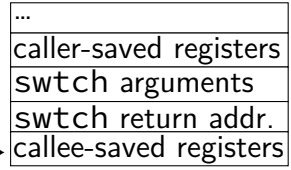
read all B's callee-saved registers from *stack*

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

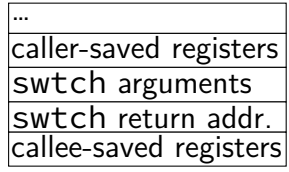
restore B's caller-saved registers from *stack*

old (A) **stack**

SP →



new (B) *stack*



thread switching in xv6: how?

swtch(A, B) pseudocode:

save A's caller-saved registers to **stack**

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

write all A's callee-saved registers to **stack**

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

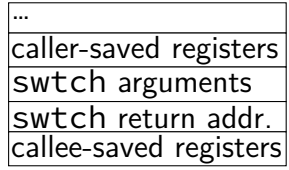
read *B* arg as new *stack* pointer

read all B's callee-saved registers from *stack*

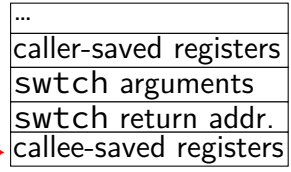
read+use swtch return address from *stack* (x86 `ret`)

restore B's caller-saved registers from *stack*

old (A) **stack**



new (B) *stack*



SP →

thread switching in xv6: how?

swtch(A, B) pseudocode:

save A's caller-saved registers to **stack**

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

write all A's callee-saved registers to **stack**

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

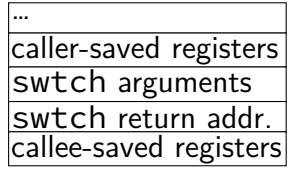
read *B* arg as new *stack* pointer

read all B's callee-saved registers from *stack*

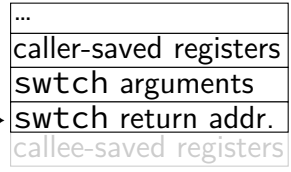
read+use swtch return address from *stack* (x86 `ret`)

restore B's caller-saved registers from *stack*

old (A) **stack**



new (B) *stack*



thread switching in xv6: how?

swtch(A, B) pseudocode:

save A's caller-saved registers to **stack**

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

write all A's callee-saved registers to **stack**

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

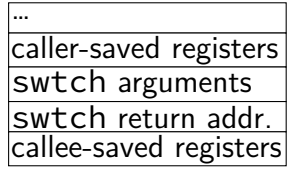
read *B* arg as new *stack* pointer

read all B's callee-saved registers from *stack*

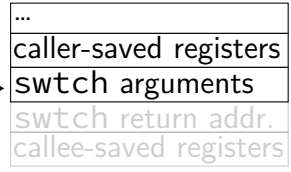
read+use swtch return address from *stack* (x86 `ret`)

restore B's caller-saved registers from *stack*

old (A) **stack**



new (B) *stack*



thread switching in xv6: how?

swtch(A, B) pseudocode:

save A's caller-saved registers to **stack**

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

write all A's callee-saved registers to **stack**

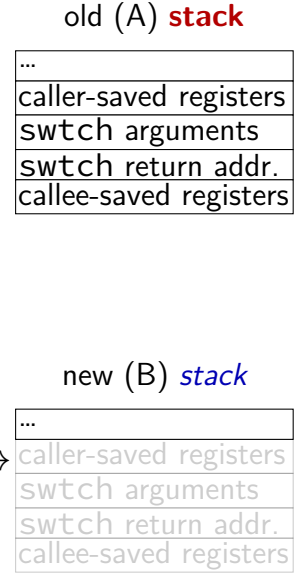
save old **stack** pointer into arg *A*

read *B* arg as new *stack* pointer

read all B's callee-saved registers from *stack*

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

restore B's caller-saved registers from *stack*



thread switching in xv6: how?

swtch(A, B) pseudocode:

save A's caller-saved registers to **stack**

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

write all A's callee-saved registers to **stack**

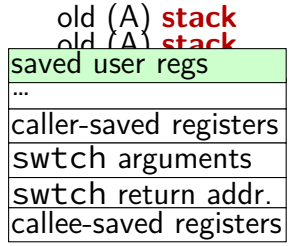
save old **stack** pointer into arg *A*

read *B* arg as new *stack* pointer

read all B's callee-saved registers from *stack*

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

restore B's caller-saved registers from *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

the userspace part?

user registers stored in ‘trapframe’ struct

- created on kernel stack when interrupt/trap happens
- restored before using `iret` to switch to user mode

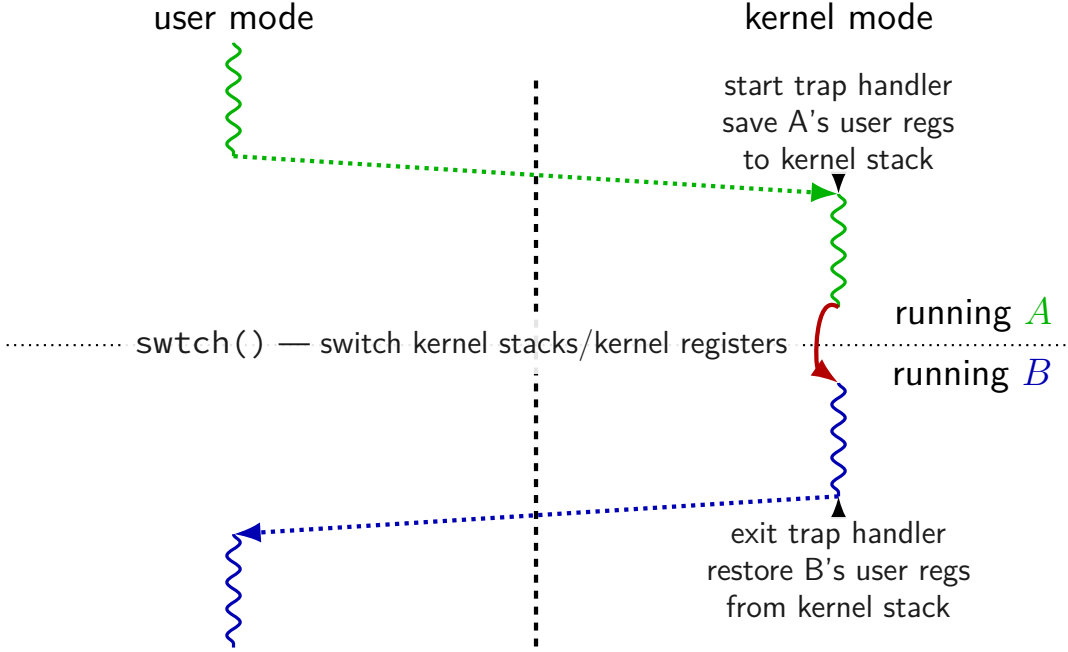
the userspace part?

user registers stored in 'trapframe' struct

- created on kernel stack when interrupt/trap happens
- restored before using `iret` to switch to user mode

other code (not shown) handles setting address space

xv6 context switch and saving



missing pieces

showed how we change kernel registers, stacks, program counter

not everything:

trap handler saving/restoring registers:

- before swtch: saving *user* registers before calling trap()
- after swtch: restoring *user* registers after returning from trap()

changing address spaces: `switchuvm`

- changes address translation mapping
- changes stack pointer for HW to use for exceptions

missing pieces

showed how we change kernel registers, stacks, program counter

not everything:

trap handler saving/restoring registers:

- before swtch: saving *user* registers before calling trap()
- after swtch: restoring *user* registers after returning from trap()

changing address spaces: `switchuvm`

- changes address translation mapping
- changes stack pointer for HW to use for exceptions

still missing: starting new thread?

exercise

suppose xv6 is running this `loop.exe`:

```
main:
    mov $0, %eax     // eax ← 0
start_loop:
    add $1, %eax     // eax ← eax + 1
    jmp start_loop   // goto start_loop
```

when xv6 switches away from this program, where is the value of loop.exe's `eax` stored?

A. loop.exe's user stack
B. loop.exe's kernel stack
C. the user stack of the program switched to
D. the kernel stack for the program switched to
E. loop.exe's heap
F. a special register
G. elsewhere

exercise (alternative)

suppose xv6 is running this `loop.exe`:

```
main:
    mov $0, %eax     // eax ← 0
start_loop:
    add $1, %eax     // eax ← eax + 1
    jmp start_loop   // goto start_loop
```

when xv6 switches away from this program, where is the value loop.exe's program counter had when it was last running in user mode stored?

A. loop.exe's user stack
B. loop.exe's kernel stack
C. the user stack of the program switched to
D. the kernel stack for the program switched to
E. loop.exe's heap
F. a special register
G. elsewhere

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)

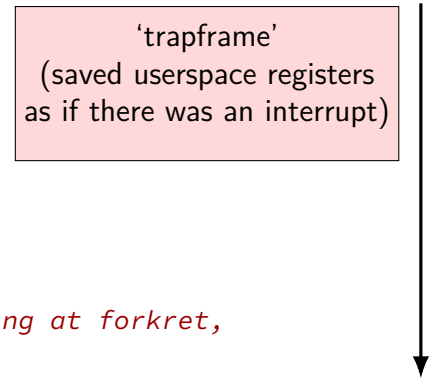
creating a new thread

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

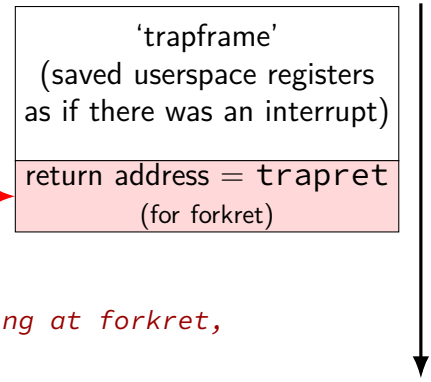
```
static struct proc*
allocproc(void)
{
  ...
  sp = p->kstack + KSTACKSIZE;

  // 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

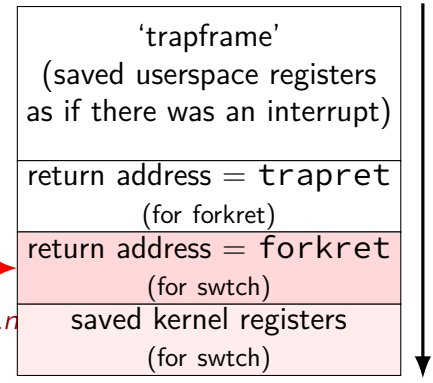


creating a new thread

```
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)

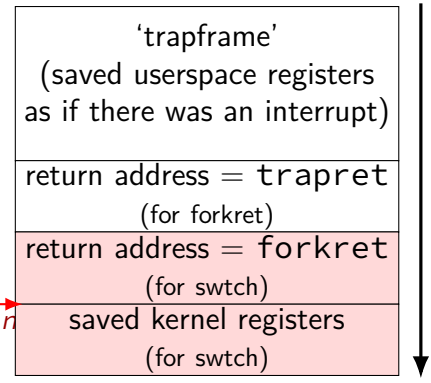


creating a new thread

```
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

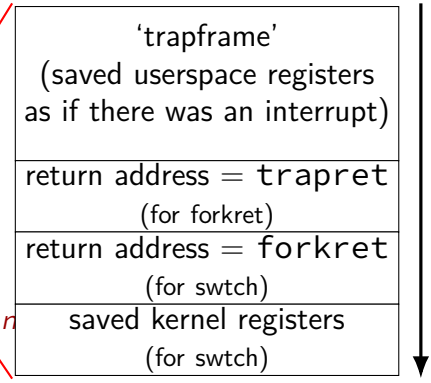
new kernel stack



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

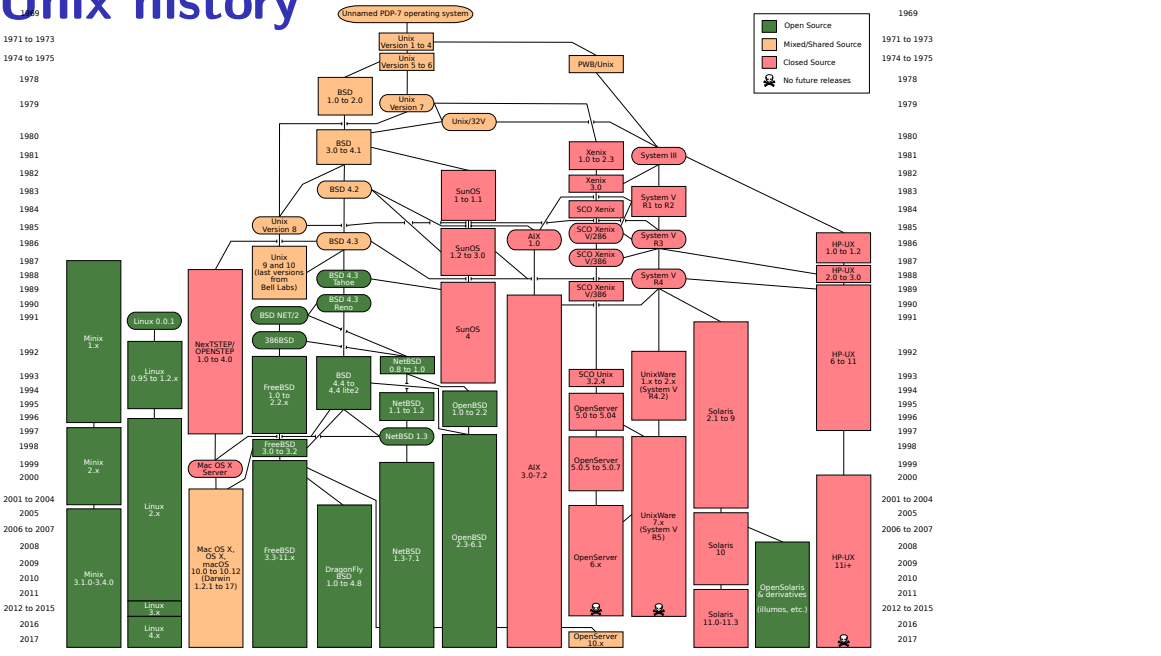
we have source code

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

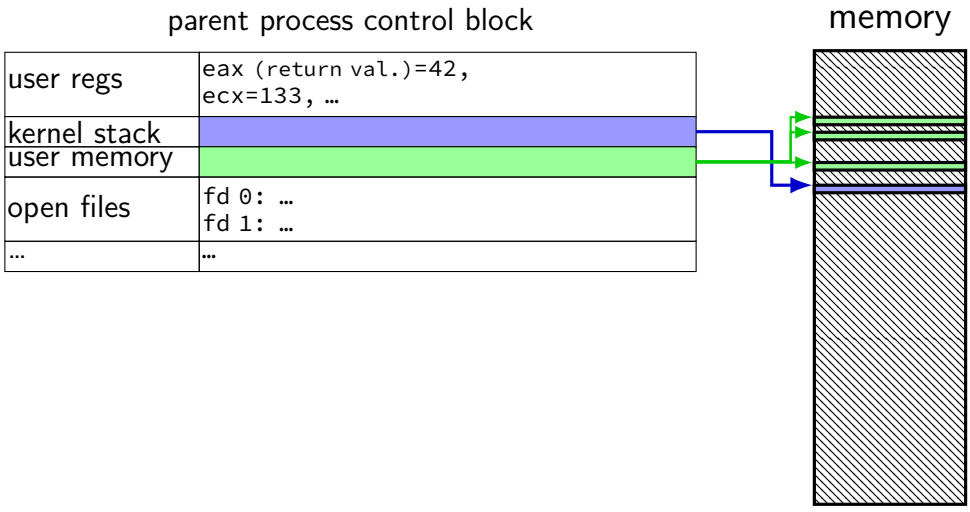
returns twice:

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

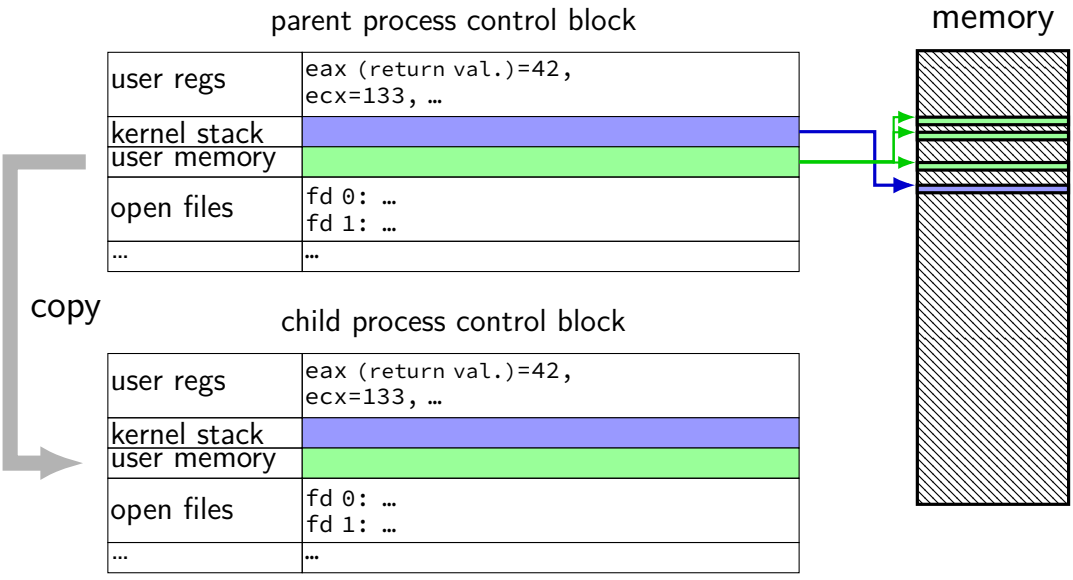
everything (but pid) duplicated in parent, child:

- memory
- file descriptors (later)
- registers

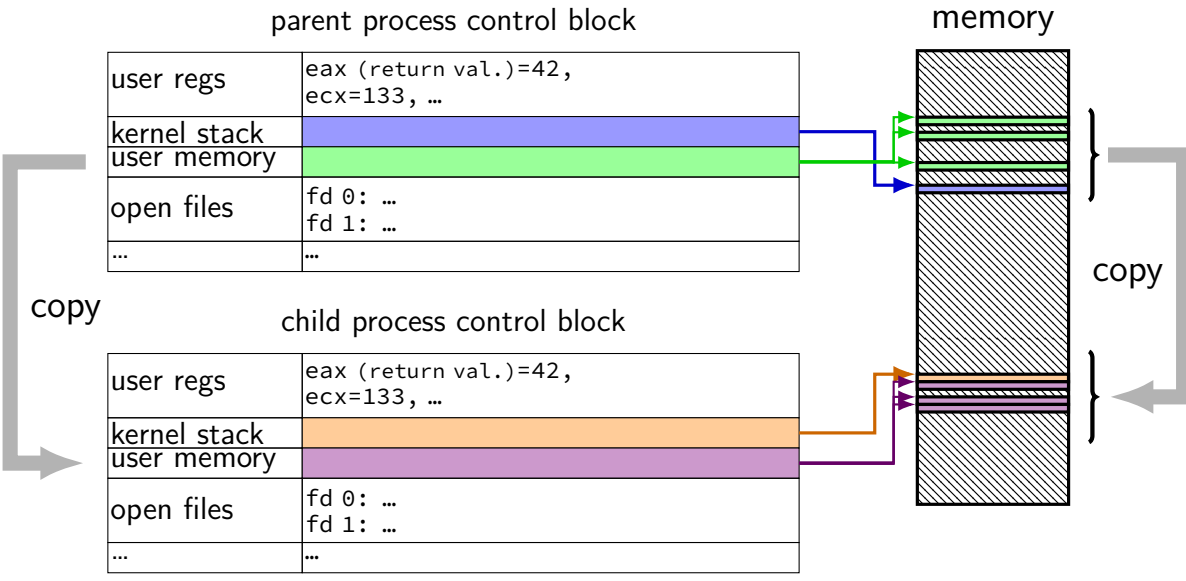
fork and PCBs



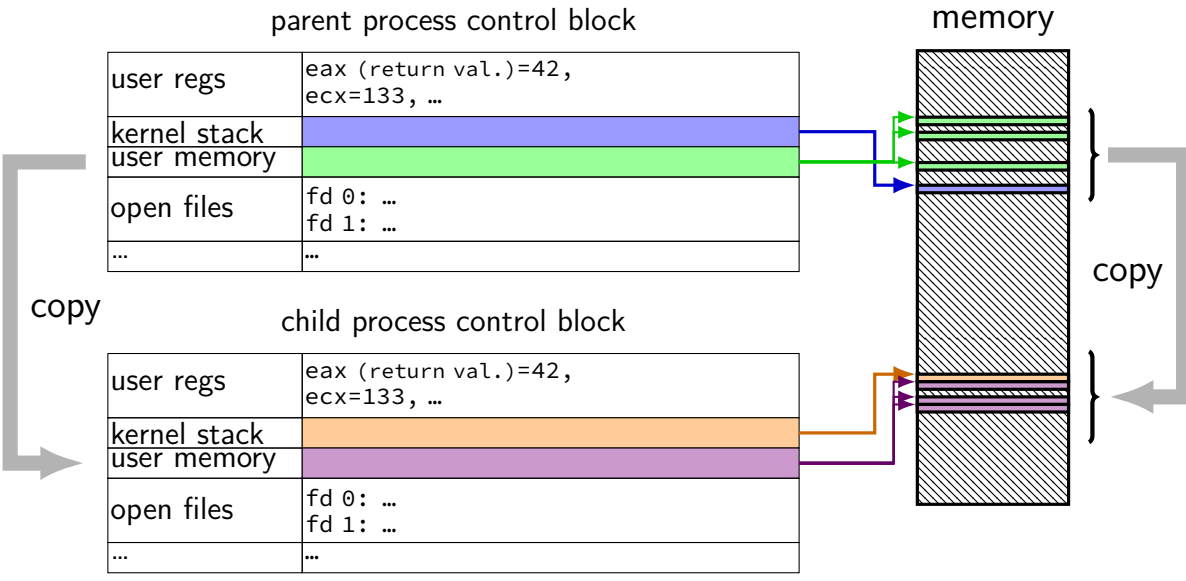
fork and PCBs



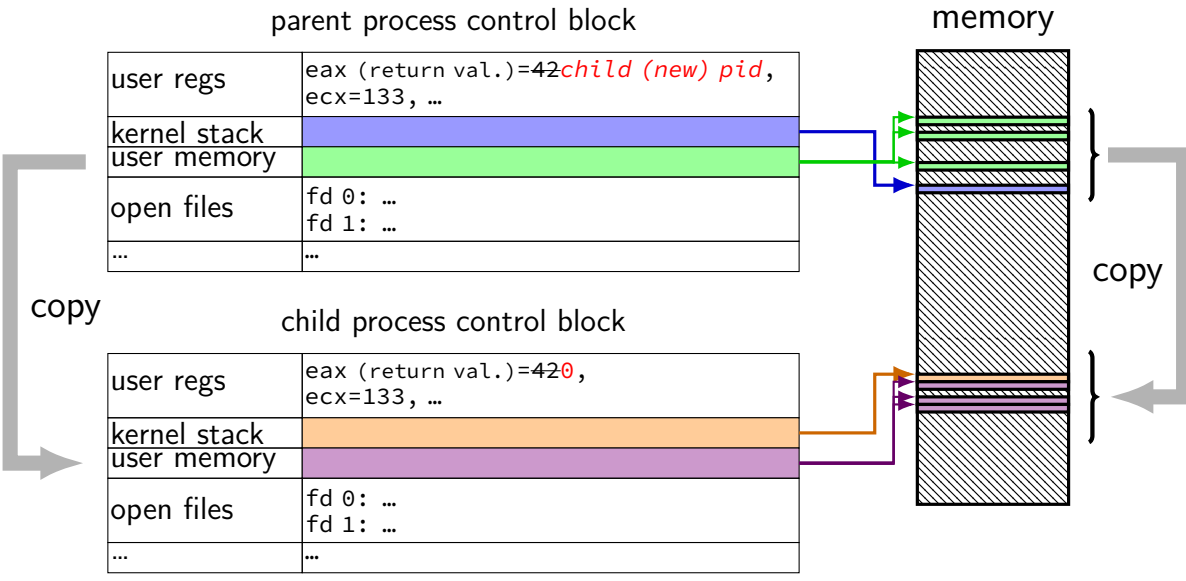
fork and PCBs



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

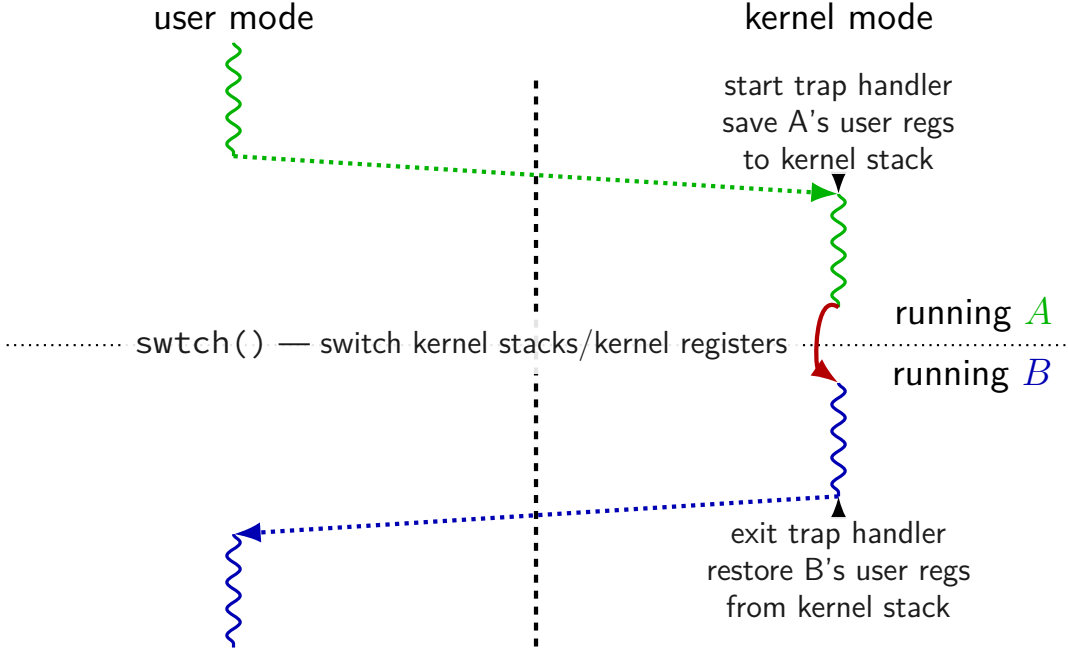
```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(int argc, char *argv[]) {
    pid_t pid = getpid();
    printf("Parent pid: %d\n", (int) pid);
    pid_t child_pid = fork();
    if (child_pid > 0) {
        /* Parent Process */
        pid_t my_pid = getpid();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

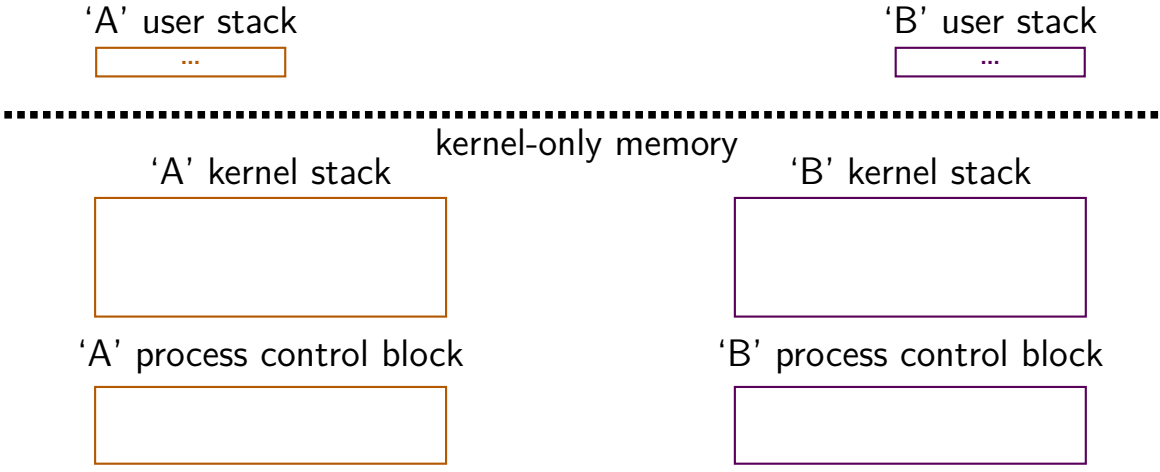
```
Parent pid: 100
[100] parent of [432]
[432] child
```

backup slides

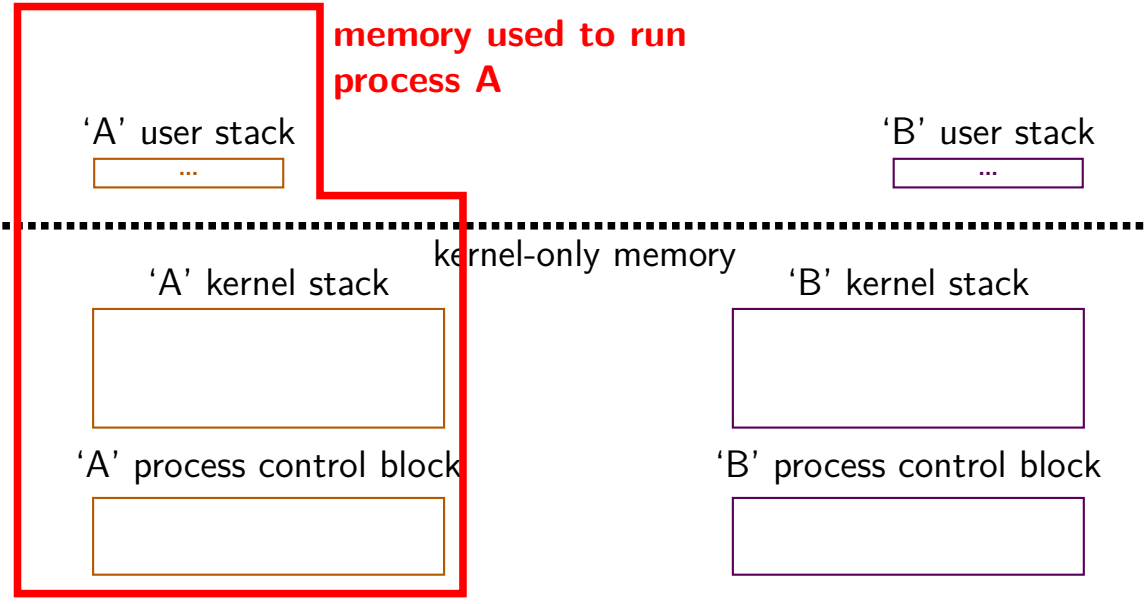
xv6 context switch and saving



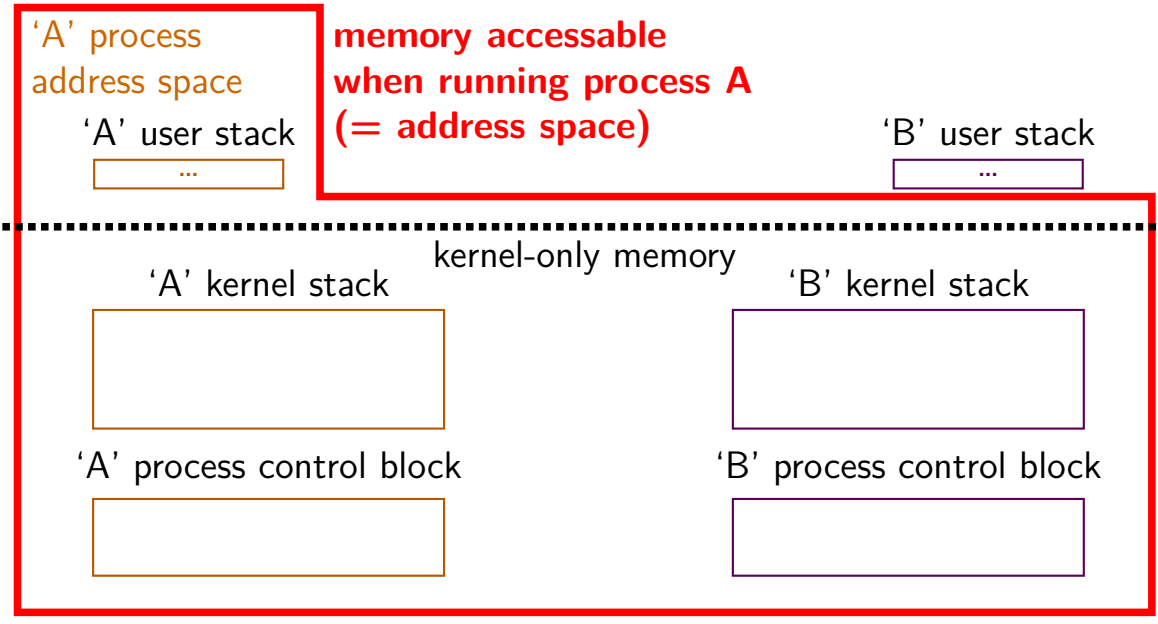
xv6: where the context is



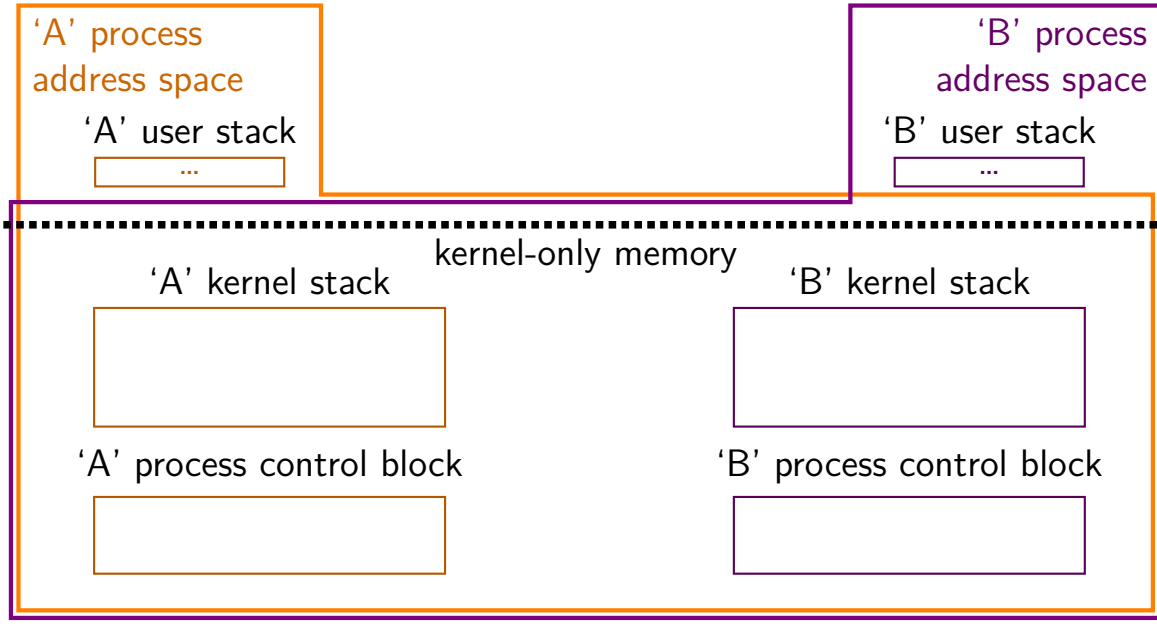
xv6: where the context is



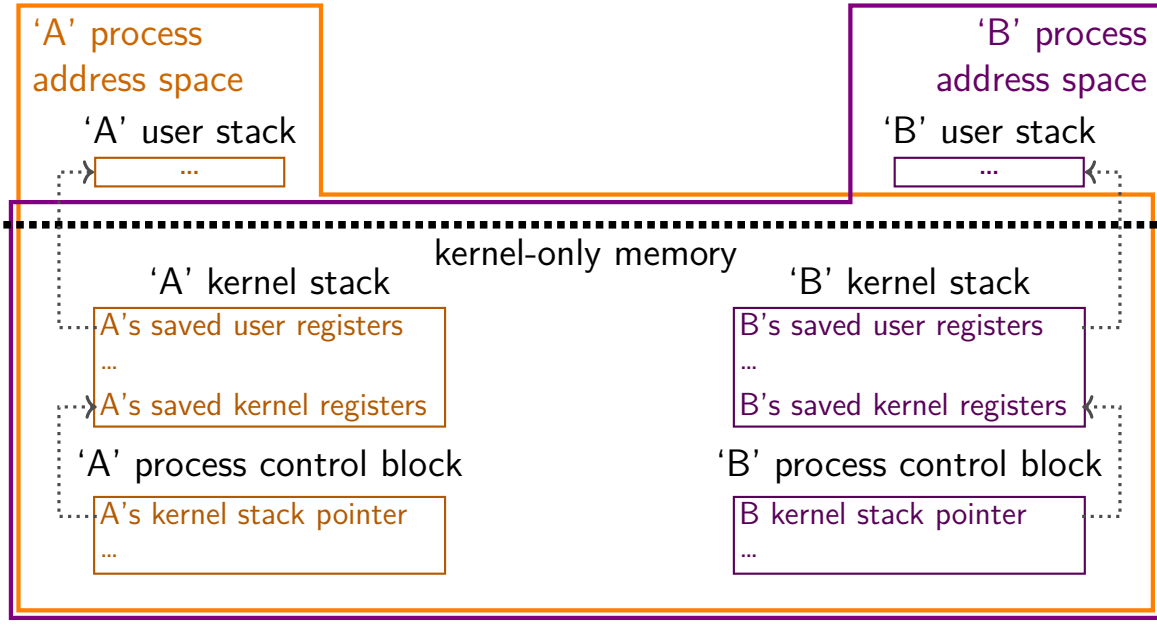
xv6: where the context is



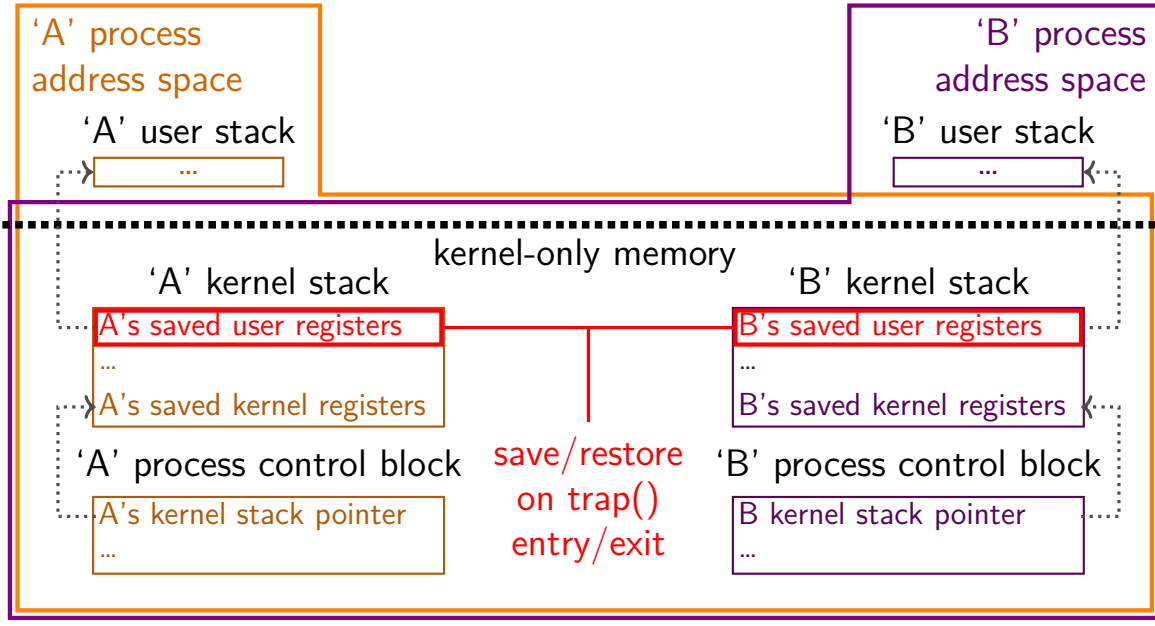
xv6: where the context is



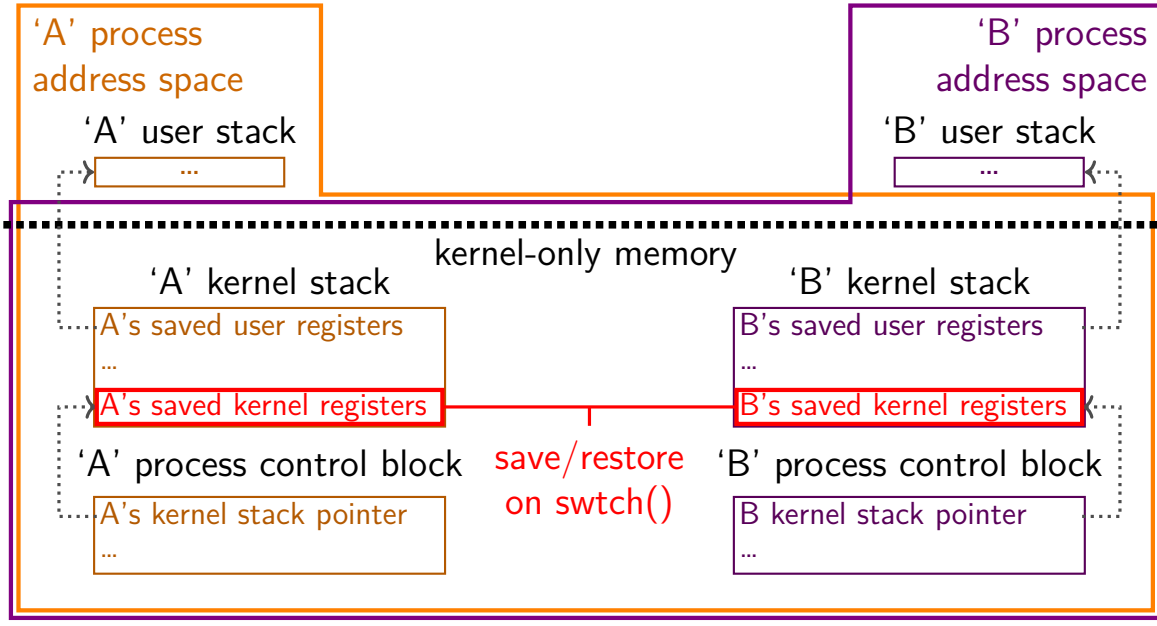
xv6: where the context is



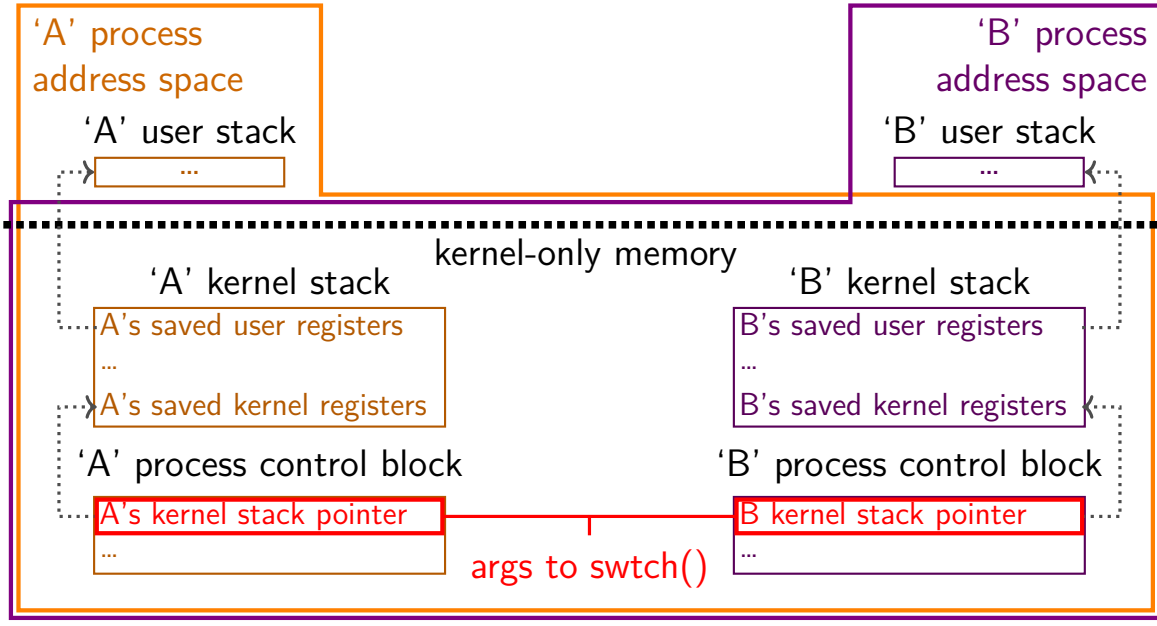
xv6: where the context is



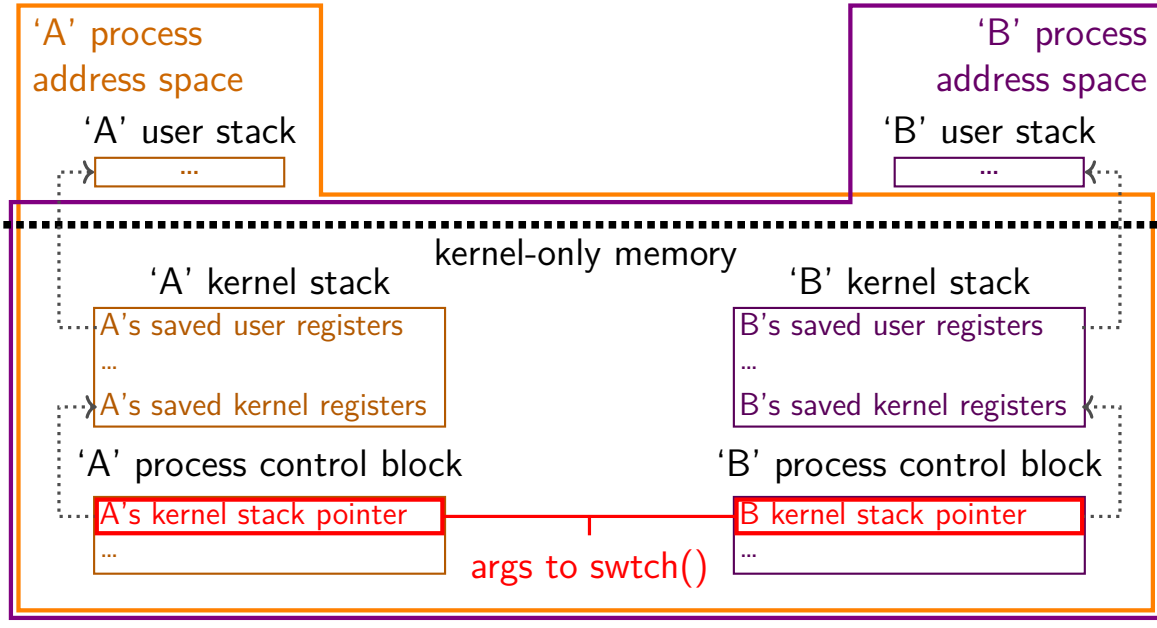
xv6: where the context is



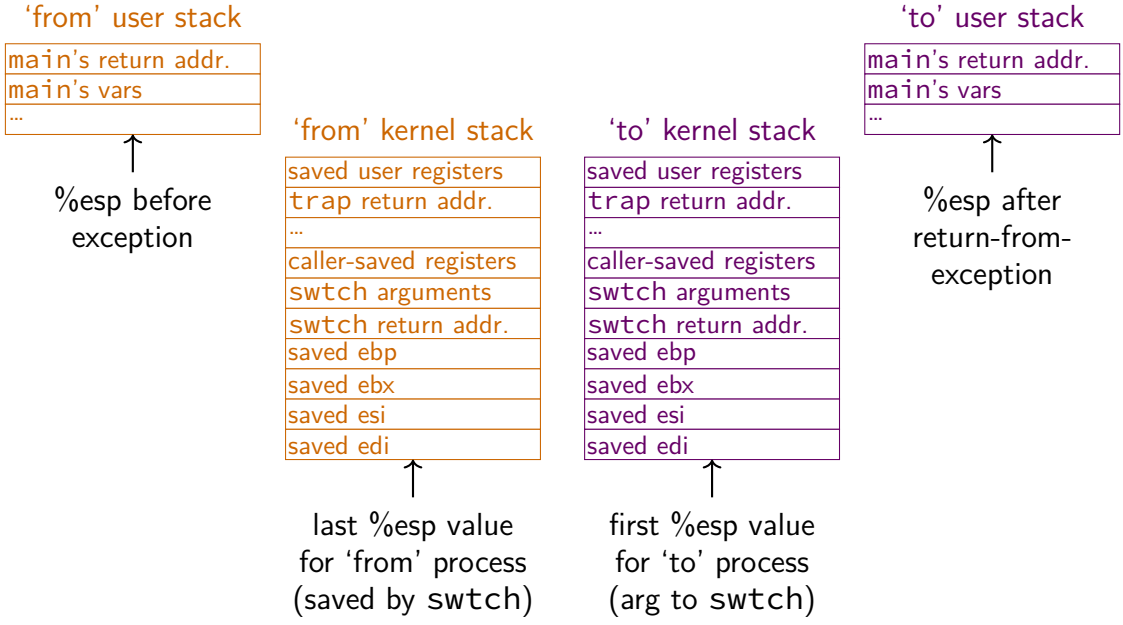
xv6: where the context is



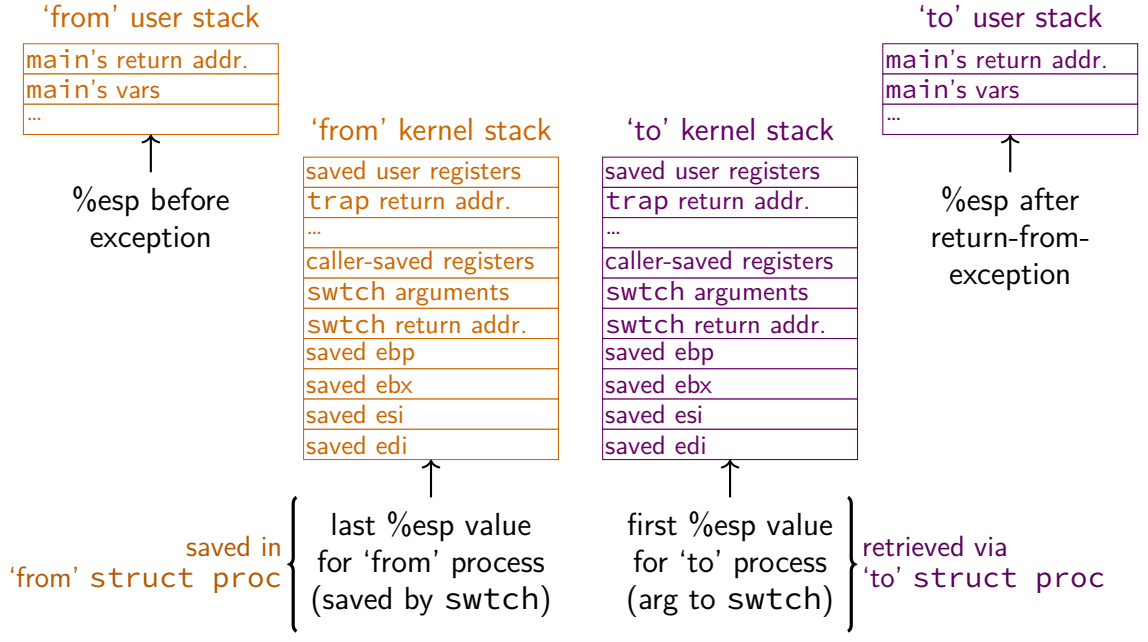
xv6: where the context is



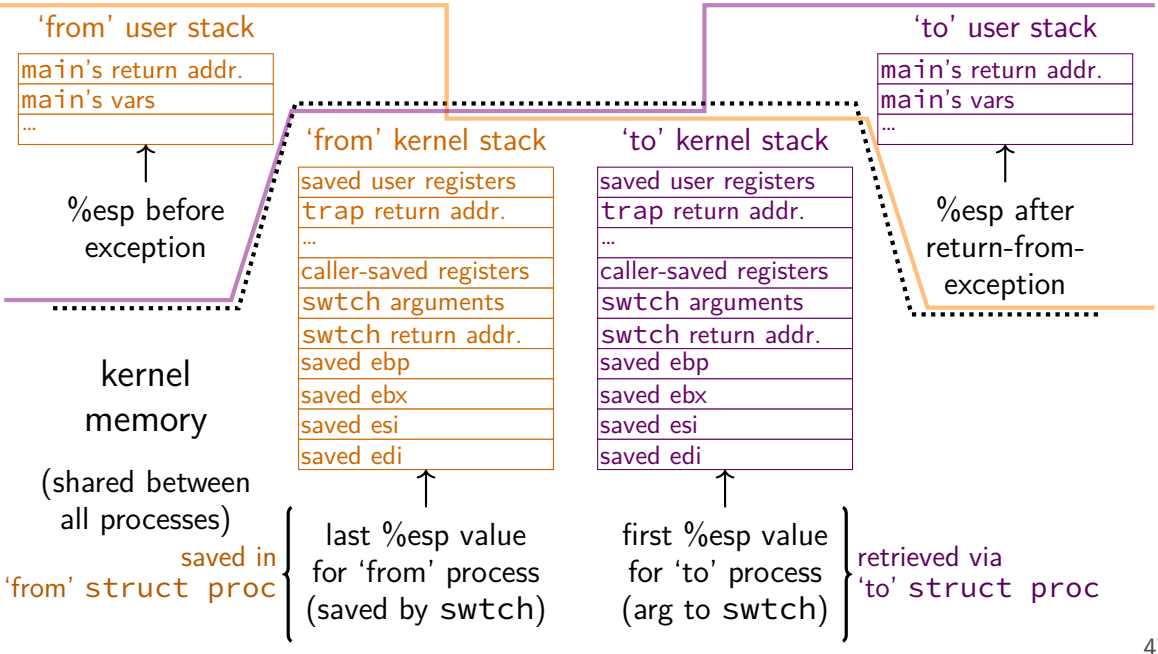
xv6: where the context is (detail)



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 *)
```