context switches / process API

last time

system call implementation in xv6

- system call wrapper
- interrupt table AKA exception table initialization
- many exception handlers that all call trap()
- using saved trap type/registers to decide operation
- OS chooses system calling convention
- xv6: system calls borrow from normal calling convention

briefly, handling other exceptions in trap()

thread context switches at a high level

- context = register values + program counter + address space
- swap context between processor and OS storage

trick: trapframe (saved regs on trap) $\sim$ user-mode part of context

quiz demo

exercise: counting context switches/syscalls

two active processes:

- A: running infinite loop
- B: described below

process B asks to read from from the keyboard

after input is available, B reads from a file

then, B does a computation and writes the result to the screen

how many context switches do we expect?

how many system calls do we expect?

- your answers can be ranges

counting system calls

(no system calls from A)

B: read from keyboard

maybe more than one — lots to read?

B: read from file

maybe more than one — opening file + lots to read?

B: write to screen

maybe more than one — lots to write?

(3 or more from B)

counting context switches

B makes system call to read from keyboard

(1) **switch to A while B waits**

keyboard input: B can run

(2) **switch to B to handle input**

B makes system call to read from file

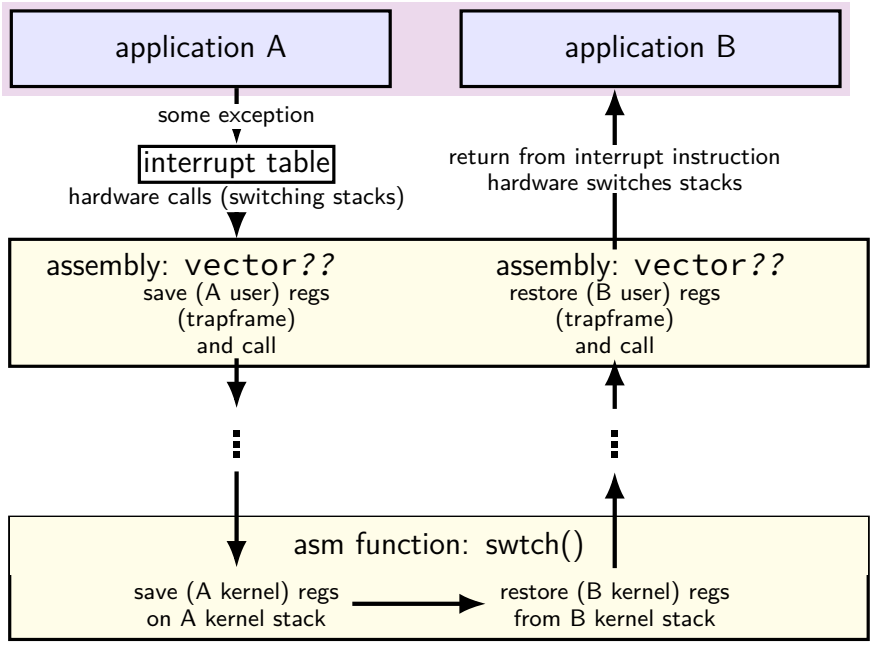
(3?) **switch to A while waiting for disk?**

if data from file not available right away

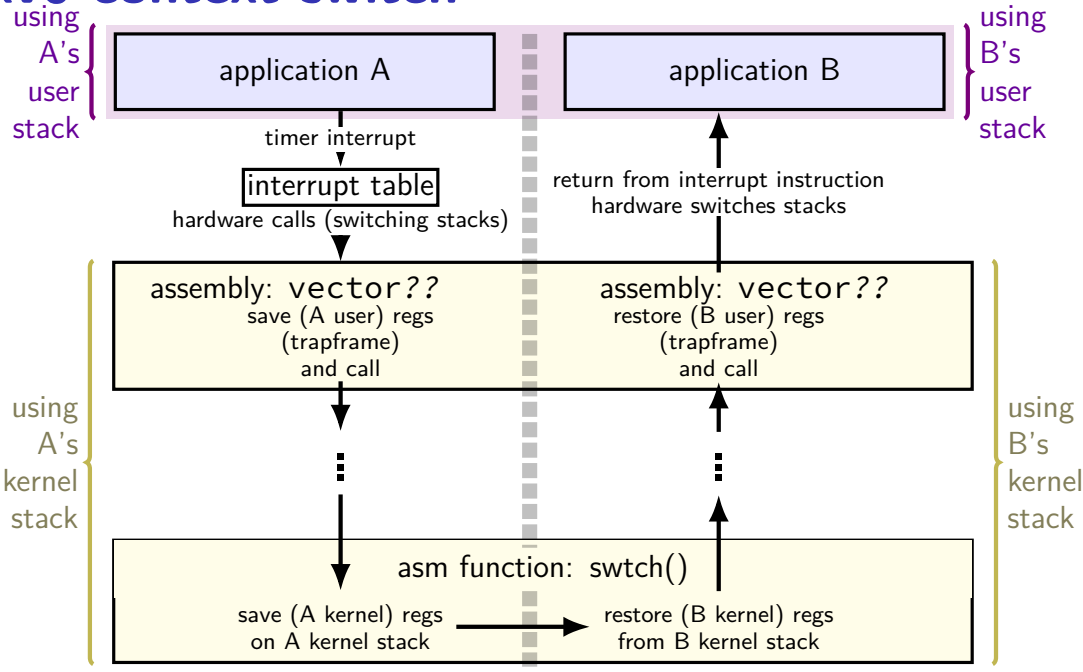
(4) **switch to B to do computation + write system call**

+ maybe switch between A + B while both are computing?

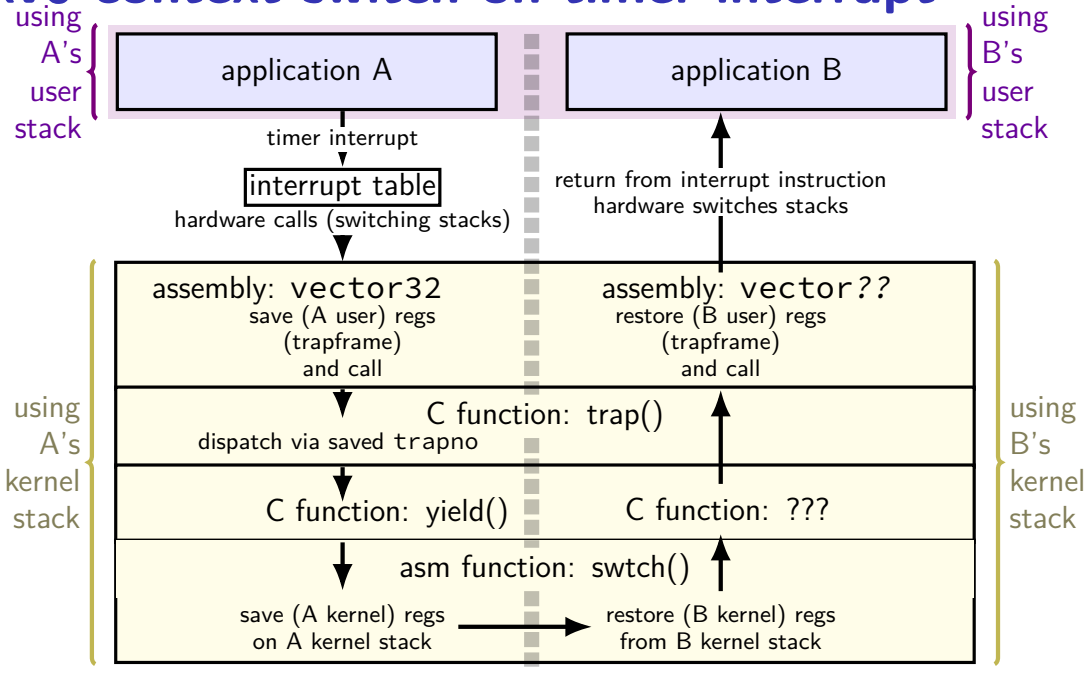
xv6 context switch



xv6 context switch



xv6 context switch on timer interrupt



preview: thread/process control block

need to have pointer to saved regs for thread

and (we'll see later) more info about threads

thread control block

term for struct/class with this information

also *process control block*s

xv6: `struct proc`

xv6: doubles as thread control block
(because each process has exactly one thread)

preview: thread/process control block

need to have pointer to saved regs for thread

and (we'll see later) more info about threads

thread control block

term for struct/class with this information

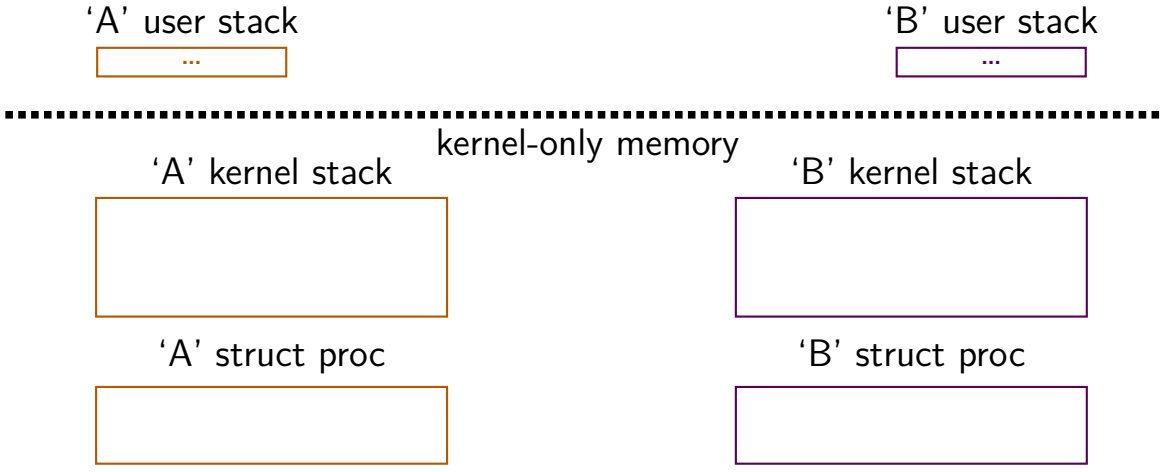
also *process control block*s

xv6: `struct proc`

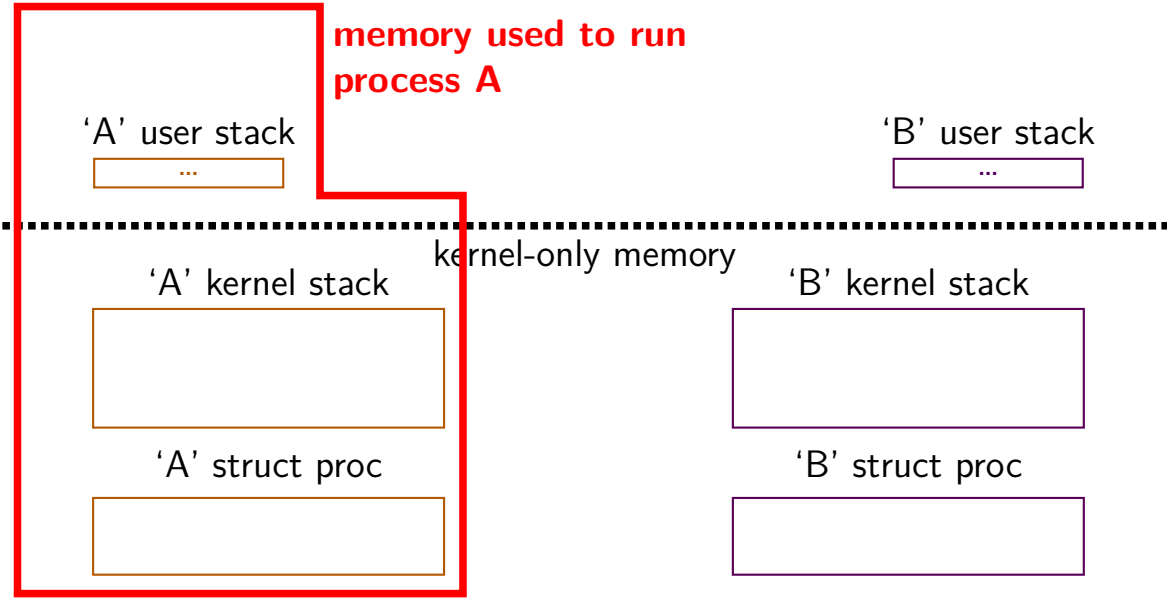
xv6: doubles as thread control block

(because each process has exactly one thread)

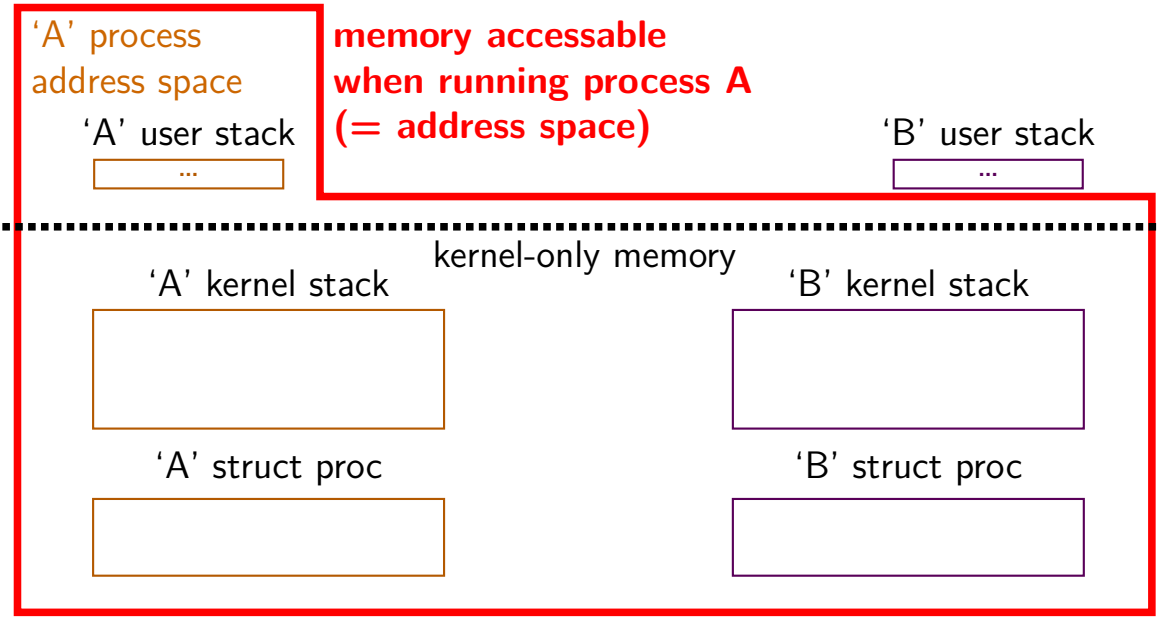
xv6: where the context is



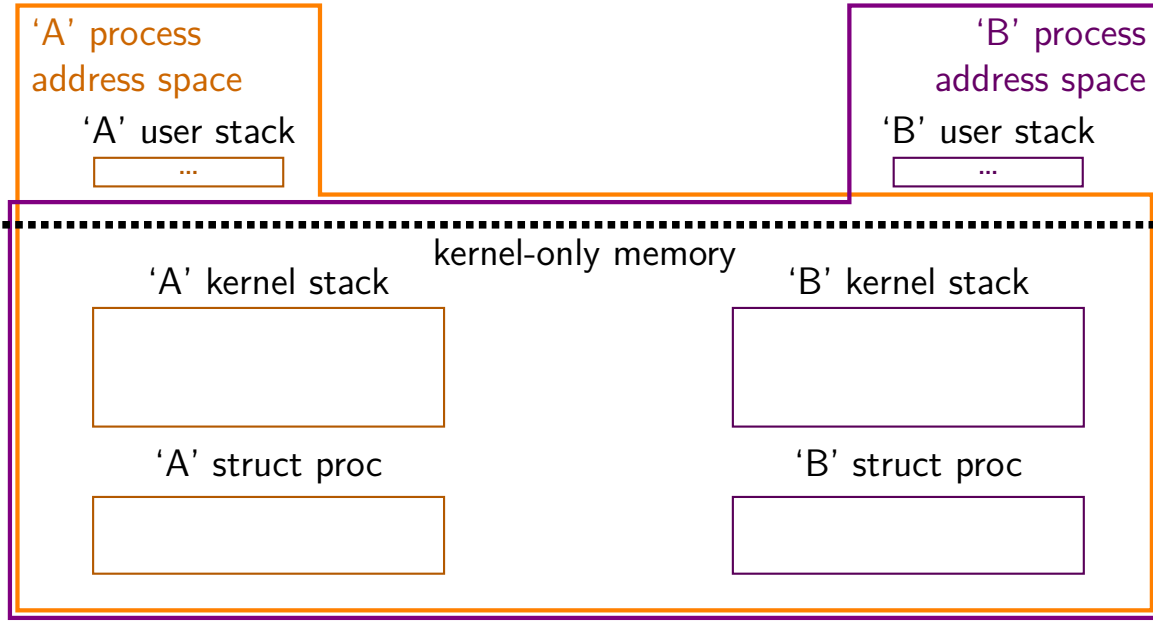
xv6: where the context is



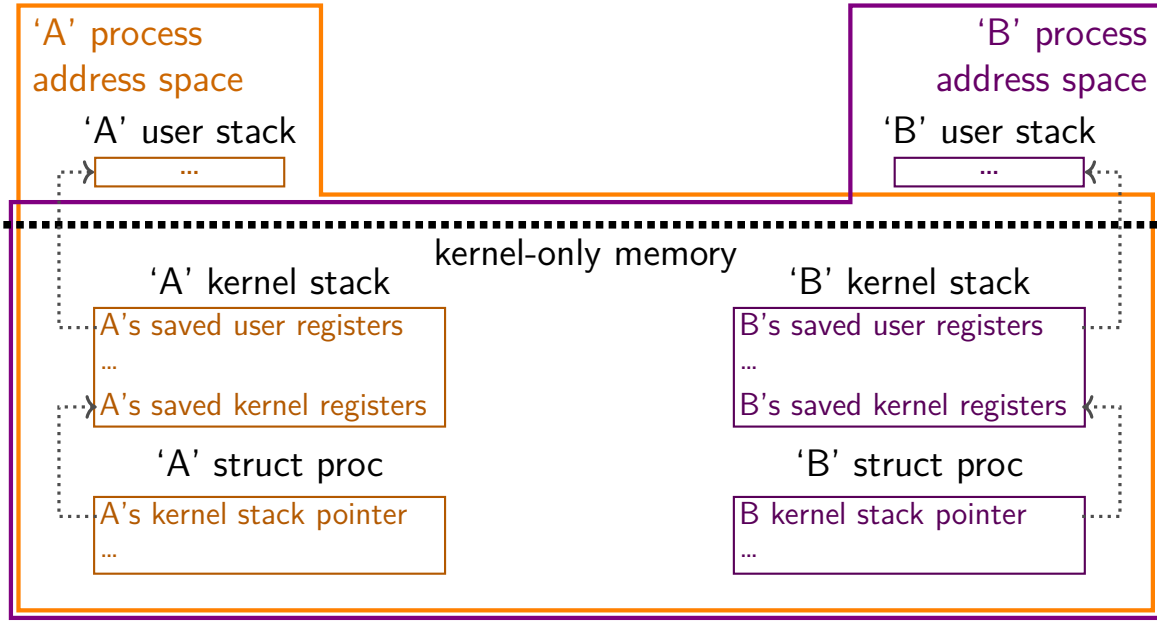
xv6: where the context is



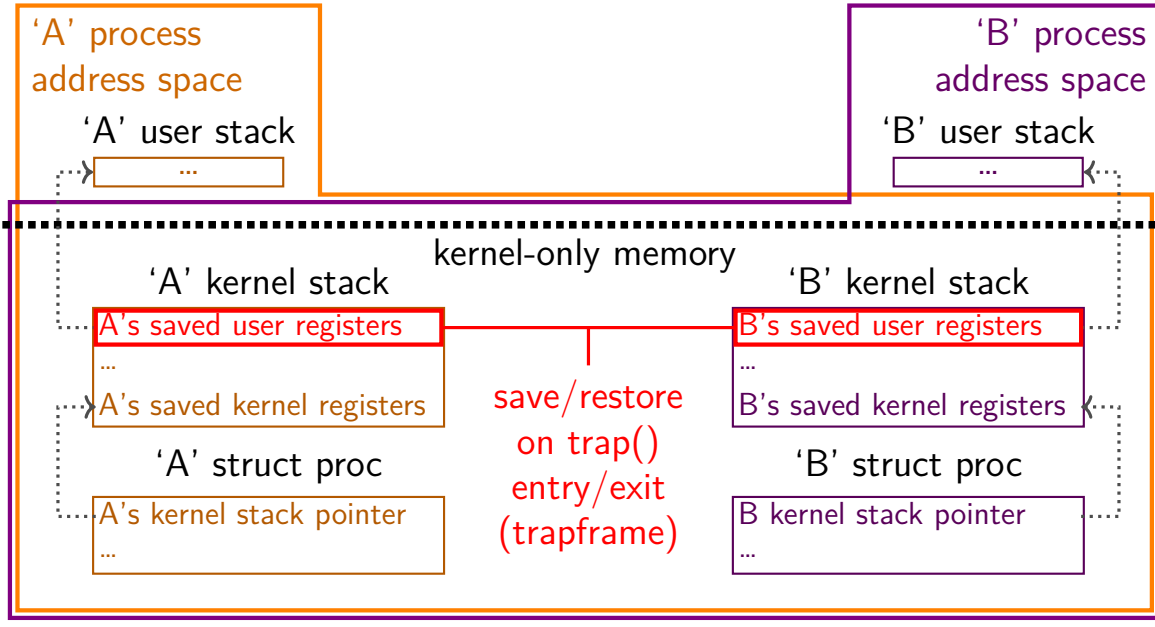
xv6: where the context is



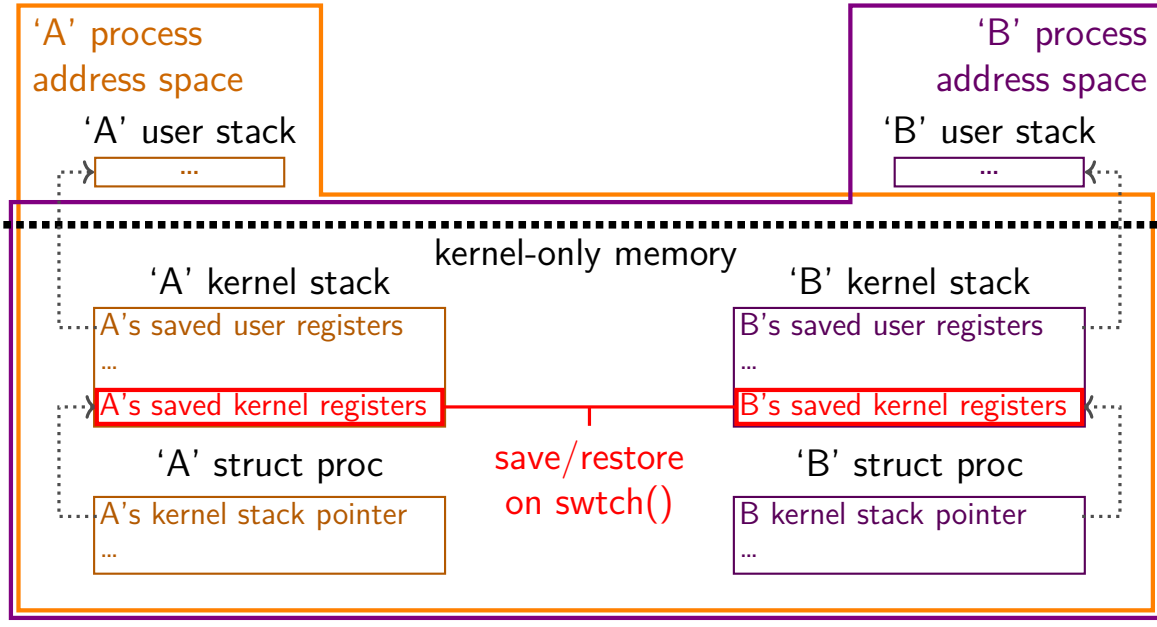
xv6: where the context is



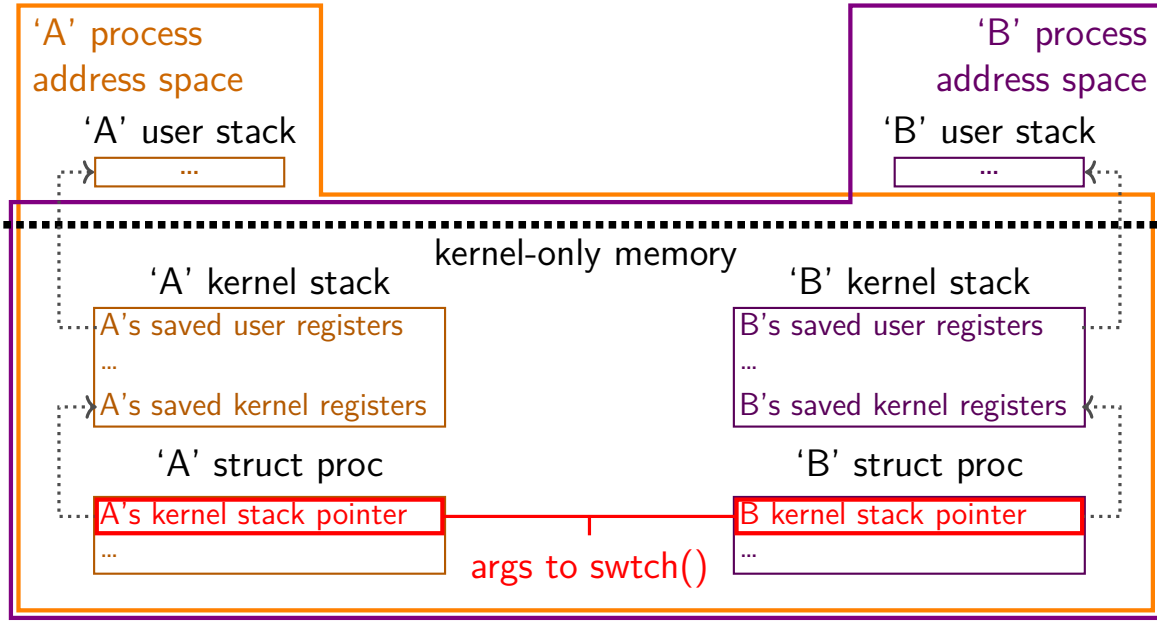
xv6: where the context is



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:

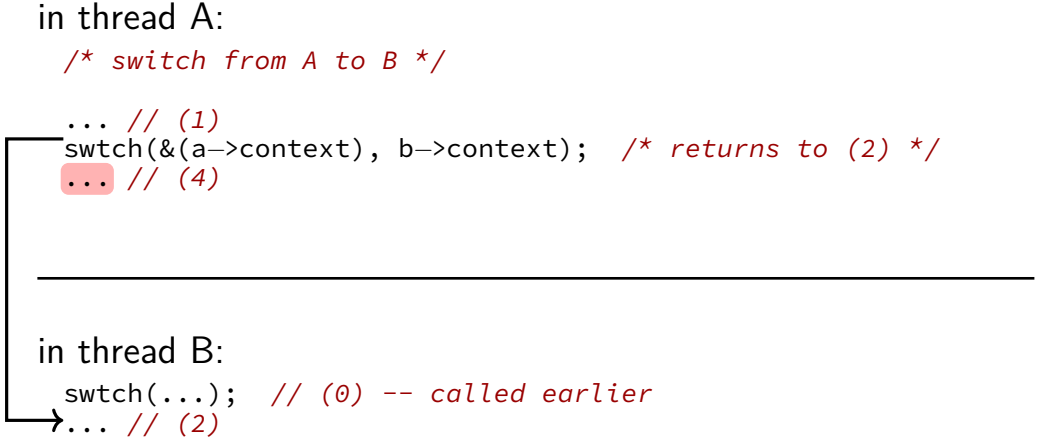
```
/* switch from A to B */

... // (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

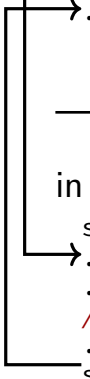
in thread A:

```
/* 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

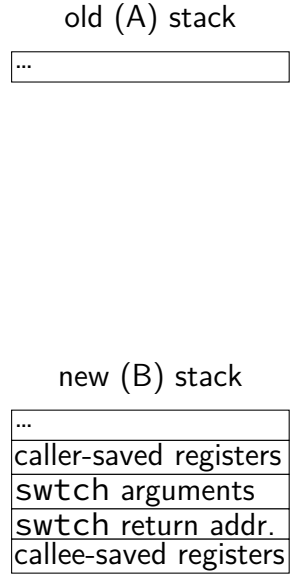
save old stack pointer into arg *A*

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



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

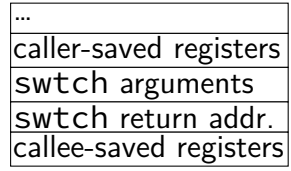
read all B's callee-saved registers from *stack*

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*

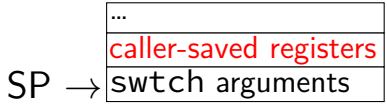
read *B* arg as new *stack* pointer

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

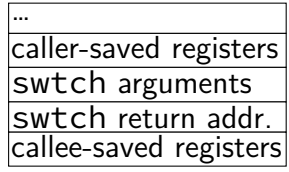
read+use swtch return address from *stack* (x86 `ret`)

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*

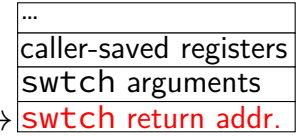
read *B* arg as new *stack* pointer

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

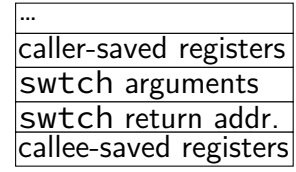
read+use swtch return address from *stack* (x86 `ret`)

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

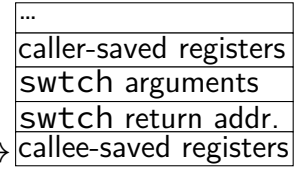
read all B's callee-saved registers from *stack*

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

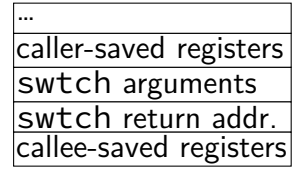
restore B's caller-saved registers from *stack*

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

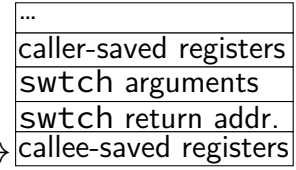
read all B's callee-saved registers from *stack*

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

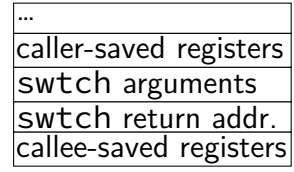
restore B's caller-saved registers from *stack*

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*

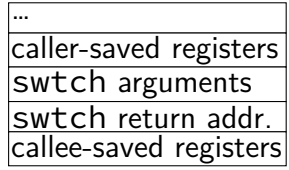
read *B* arg as new *stack* pointer

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

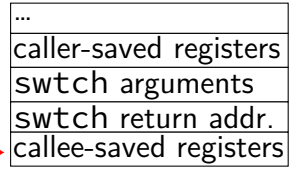
read+use swtch return address from *stack* (x86 `ret`)

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*

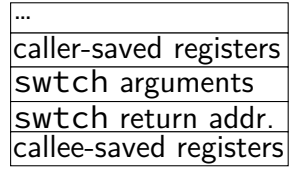
read *B* arg as new *stack* pointer

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

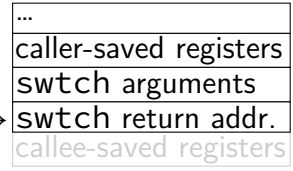
read+use swtch return address from *stack* (x86 `ret`)

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**

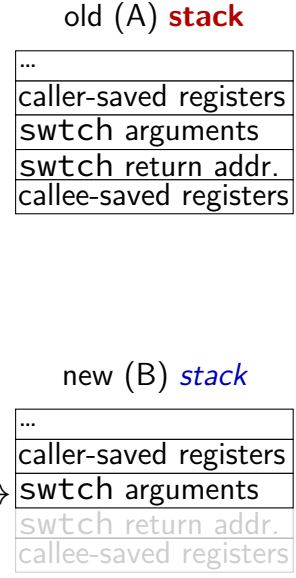
save old **stack** pointer into arg *A*

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*



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*

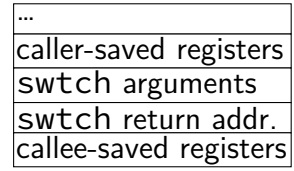
read *B* arg as new *stack* pointer

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

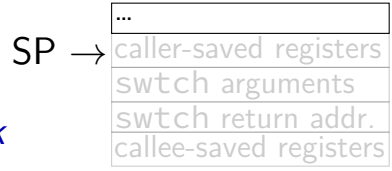
read+use swtch return address from *stack* (x86 `ret`)

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*

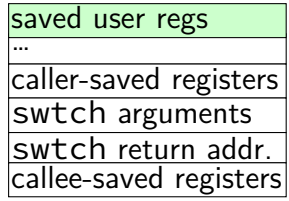
read *B* arg as new *stack* pointer

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

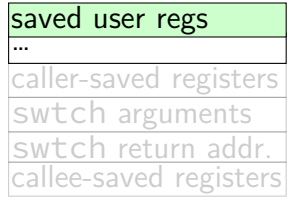
read+use swtch return address from *stack* (x86 `ret`)

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

old (A) **stack**



new (B) *stack*



thread switching in xv6: how?

swtch(A, B) pseudocode:

most work done by compiler — part of function call

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**

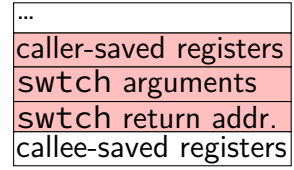
save old **stack** pointer into arg *A*

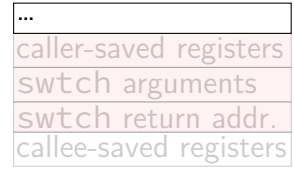
read *B* arg as new *stack* pointer

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 // eax ← M[esp+4]
  movl 8(%esp), %edx // edx ← M[esp+8]

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

  # Switch stacks
  movl %esp, (%eax) // M[eax] ← esp
  movl %edx, %esp   // esp ← edx

  # 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 // eax ← M[esp+4]
  movl 8(%esp), %edx // edx ← M[esp+8]

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

  # Switch stacks
  movl %esp, (%eax) // M[eax] ← esp
  movl %edx, %esp   // esp ← edx

  # 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

```
.globl swtch
swtch:
  movl 4(%esp), %eax // eax ← M[esp+4]
  movl 8(%esp), %edx // edx ← M[esp+8]

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

  # Switch stacks
  movl %esp, (%eax) // M[eax] ← esp
  movl %edx, %esp   // esp ← edx

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

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

thread switching in xv6: assembly

```
.globl swtch
swtch:
  movl 4(%esp), %eax // eax ← M[esp+4]
  movl 8(%esp), %edx // edx ← M[esp+8]

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

  # Switch stacks
  movl %esp, (%eax) // M[eax] ← esp
  movl %edx, %esp   // esp ← edx

  # 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 // eax ← M[esp+4]
  movl 8(%esp), %edx // edx ← M[esp+8]

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

  # Switch stacks
  movl %esp, (%eax) // M[eax] ← esp
  movl %edx, %esp   // esp ← edx

  # 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 argumen

thread switching in xv6: assembly

```
.globl swtch
swtch:
  movl 4(%esp), %eax // eax ← M[esp+4]
  movl 8(%esp), %edx // edx ← M[esp+8]

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

  # Switch stacks
  movl %esp, (%eax) // M[eax] ← esp
  movl %edx, %esp   // esp ← edx

  # 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

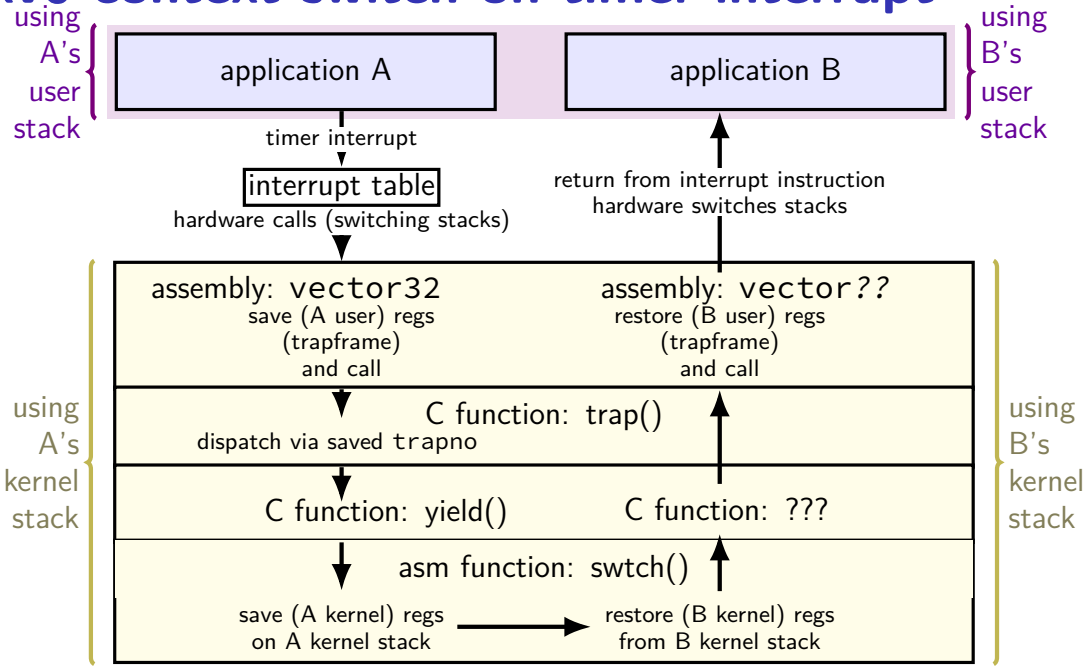
the userspace part?

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 on timer interrupt



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)

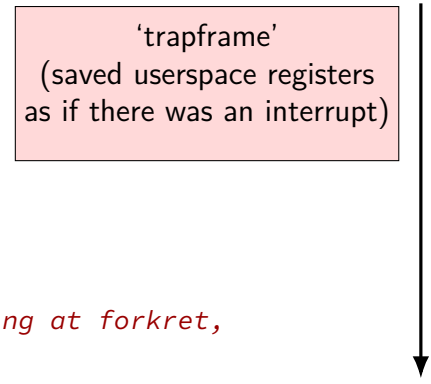
creating a new thread

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

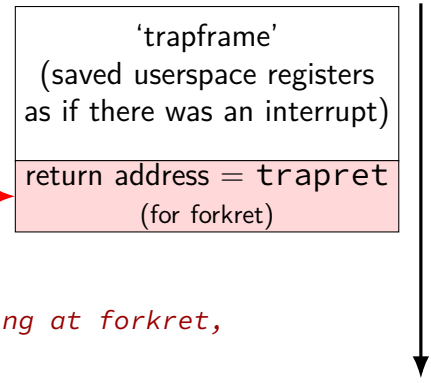
```
static struct proc*
allocproc(void)
{
  ...
  sp = p->kstack + KSTACKSIZE;

  // 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

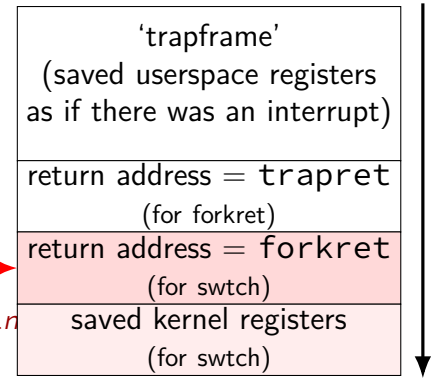


creating a new thread

```
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)

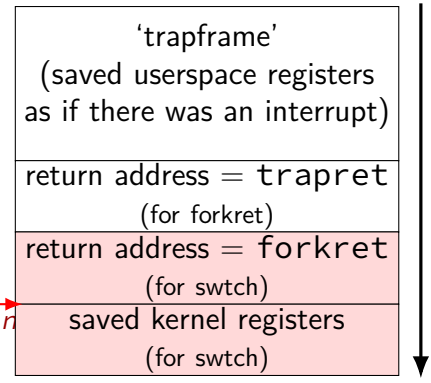


creating a new thread

```
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

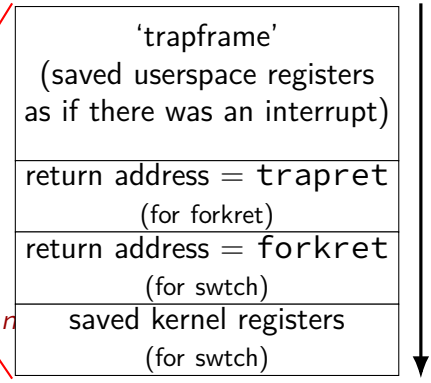
new kernel stack



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;
}
```

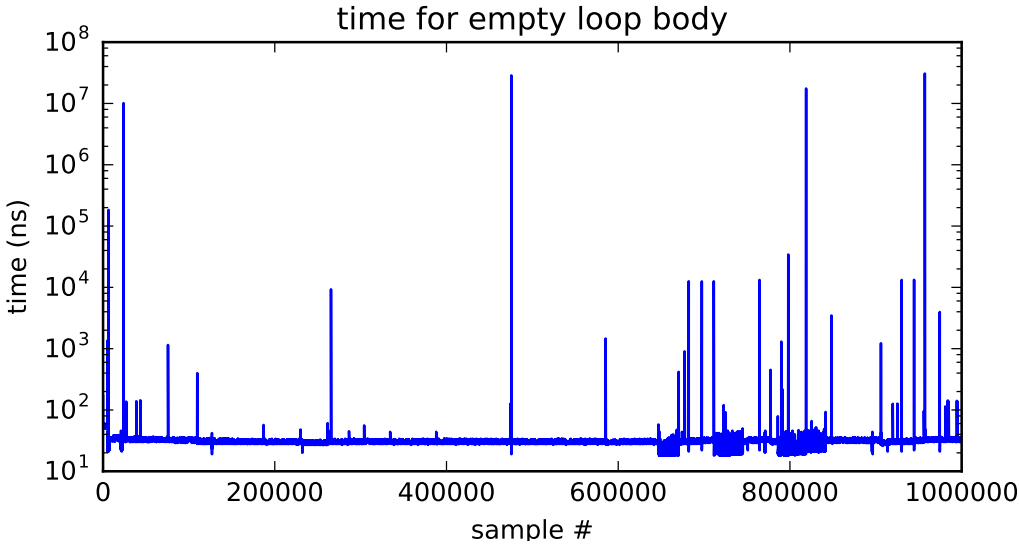
backup slides

timing nothing

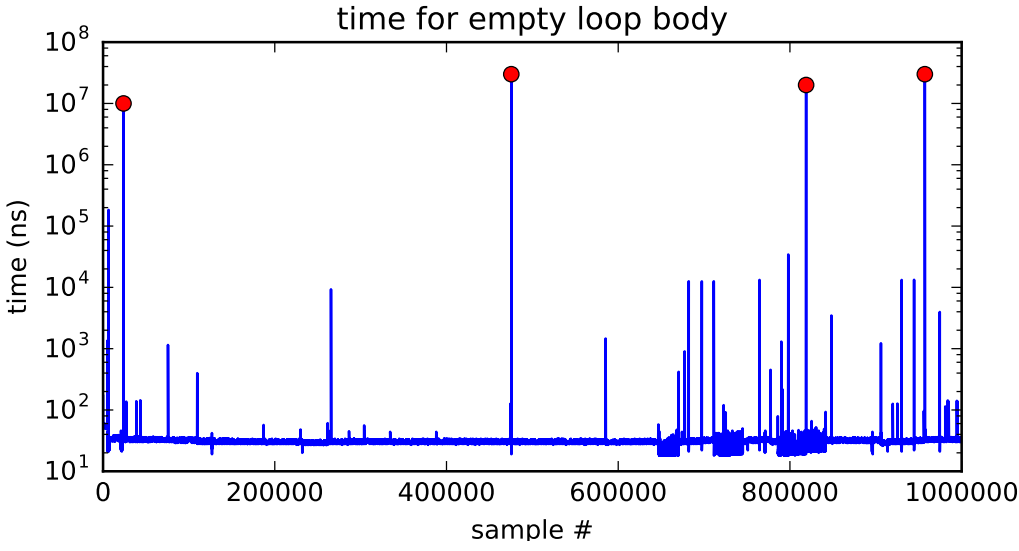
```
long times[NUM_TIMINGS];
int main(void) {
    for (int i = 0; i < N; ++i) {
        long start, end;
        start = get_time();
        /* do nothing */
        end = get_time();
        times[i] = end - start;
    }
    output_timings(times);
}
```

same instructions — same difference each time?

doing nothing on a busy system



doing nothing on a busy system



write syscall in xv6: summary

`write` function — syscall wrapper uses `int $64`

interrupt table entry setup points to assembly function `vector64`
(and switches to kernel stack)

…which calls `trap()` with trap number set to 64 (`T_SYSCALL`)
(after saving all registers into `struct trapframe`)

…which checks trap number, then calls `syscall()`

…which checks syscall number (from eax)

…and uses it to call `sys_write`

…which reads arguments from the stack and does the write

…then registers restored, return to user space

write syscall in xv6: summary

`write` function — syscall wrapper uses `int $64`

interrupt table entry setup points to assembly function `vector64`
(and switches to kernel stack)

…which calls `trap()` with trap number set to 64 (`T_SYSCALL`)
(after saving all registers into `struct trapframe`)

…which checks trap number, then calls `syscall()`

…which checks syscall number (from eax)

…and uses it to call `sys_write`

…which reads arguments from the stack and does the write

…then registers restored, return to user space

write syscall in xv6: summary

`write` function — syscall wrapper uses `int $64`

interrupt table entry setup points to assembly function `vector64`
(and switches to kernel stack)

…which calls `trap()` with trap number set to 64 (`T_SYSCALL`)
(after saving all registers into `struct trapframe`)

…which checks trap number, then calls `syscall()`

…which checks syscall number (from eax)

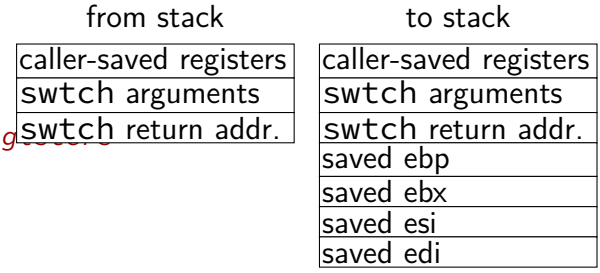
…and uses it to call `sys_write`

…which reads arguments from the stack and does the write

…then registers restored, return to user space

juggling stacks

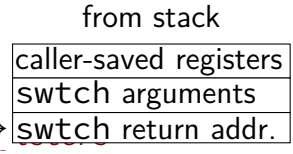
```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # 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
```

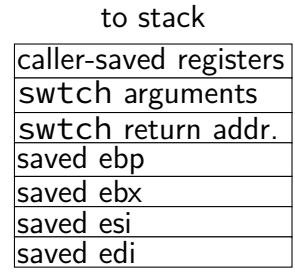


juggling stacks

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

  # Save old callee
  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
```

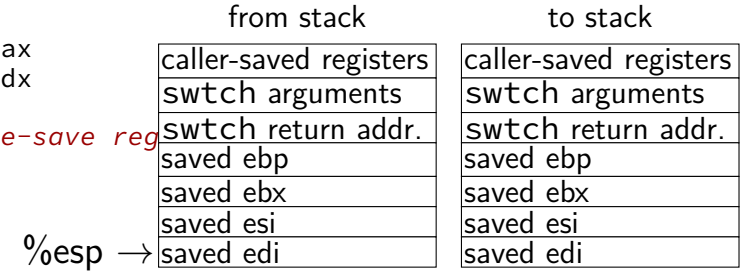
%esp →





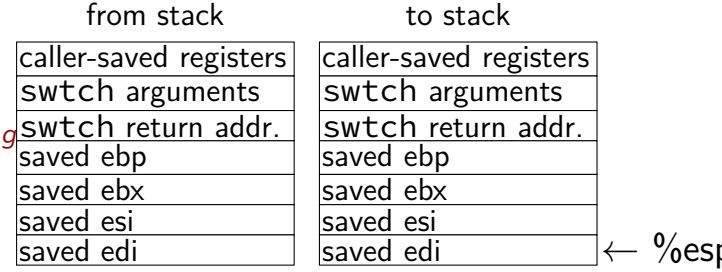
juggling stacks

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

  # Save old callee-save reg
  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
```



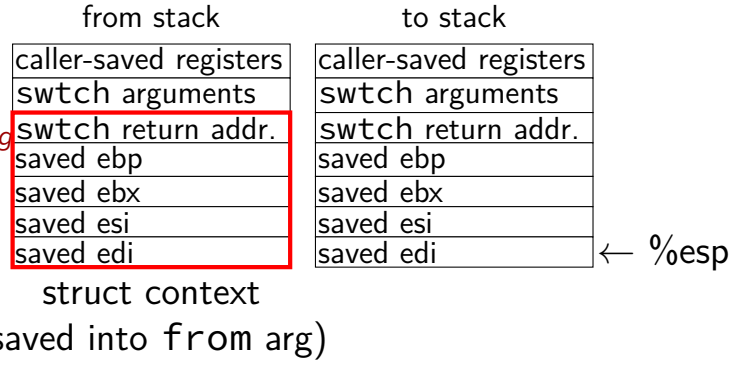
juggling stacks

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

  # Save old callee-save reg
  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
```



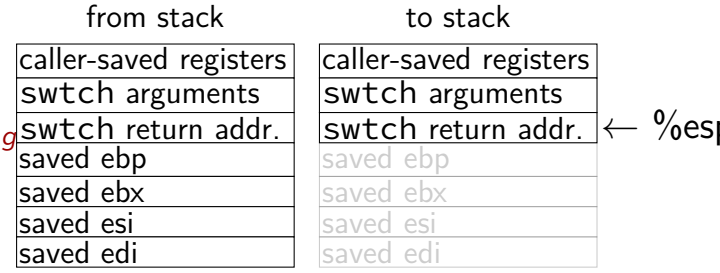
juggling stacks

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

  # Save old callee-save reg
  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
```



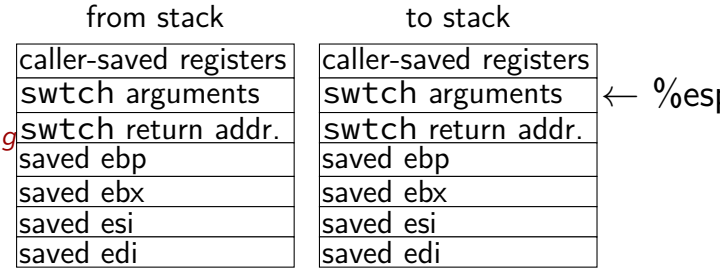
juggling stacks

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

  # Save old callee-save reg
  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
```



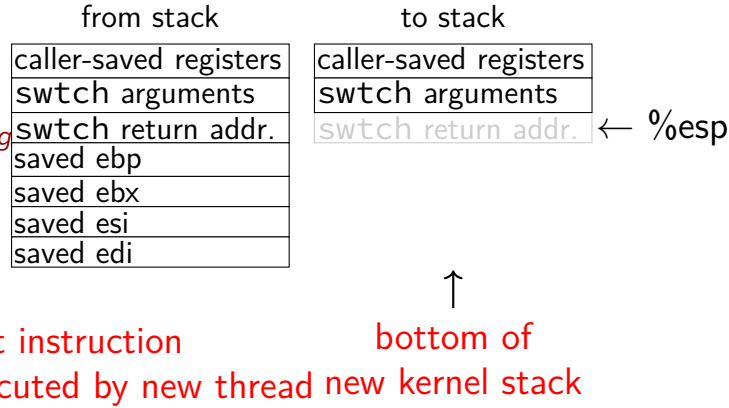
juggling stacks

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

  # Save old callee-save reg
  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
```



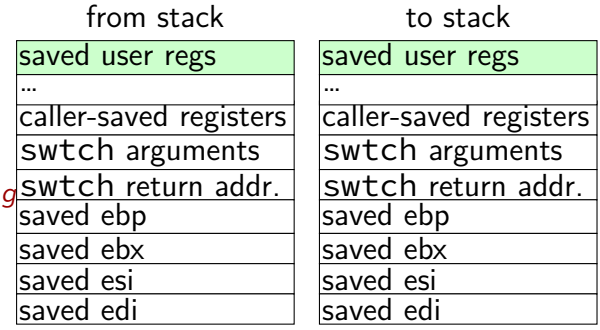
juggling stacks

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

  # Save old callee-save reg
  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
```



juggling stacks

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

  # Save old callee-save reg
  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
```



kernel-space context switch summary

swtch function

- saves registers on current kernel stack
- switches to new kernel stack and restores its registers

(later) initial setup — manually construct stack values

struct context

```
struct context {
  uint edi;              /* <-- top of stack of this thread */
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;          /* <-- return address of swtch() */
  /* not in struct but stored on stack thread after eip:
        arguments to current call to swtch
        caller-saved registers
        call stack include call to trap() function
        user registers
    */
}
```

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

struct context

structure to save context in
only includes callee-saved registers
rest is saved on stack before swtch involved

```
struct context {
  uint edi;              /* <-- top of stack of this thread */
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;          /* <-- return address of swtch() */
  /* not in struct but stored on stack thread after eip:
        arguments to current call to swtch
        caller-saved registers
        call stack include call to trap() function
        user registers
    */
}
```

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

struct context

eip = saved program counter

```
struct context {
  uint edi;              /* <-- top of stack of this thread */
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;            /* <-- return address of swtch() */
  /* not in struct but stored on stack thread after eip:
        arguments to current call to swtch
        caller-saved registers
        call stack include call to trap() function
        user registers
    */
}
```

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

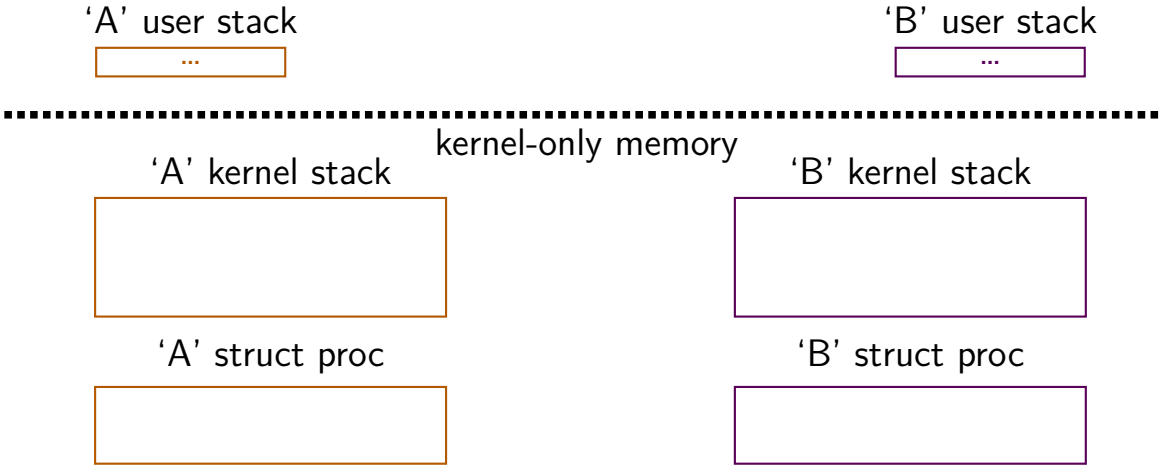
struct context

```
struct context {
  uint edi;              /* <-- top of stack of this thread */
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;          /* <-- return address of swtch() */
  /* not in struct but stored on stack thread after eip:
        arguments to current call to swtch
        caller-saved registers
        call stack include call to trap() function
        user registers
    */
}
```

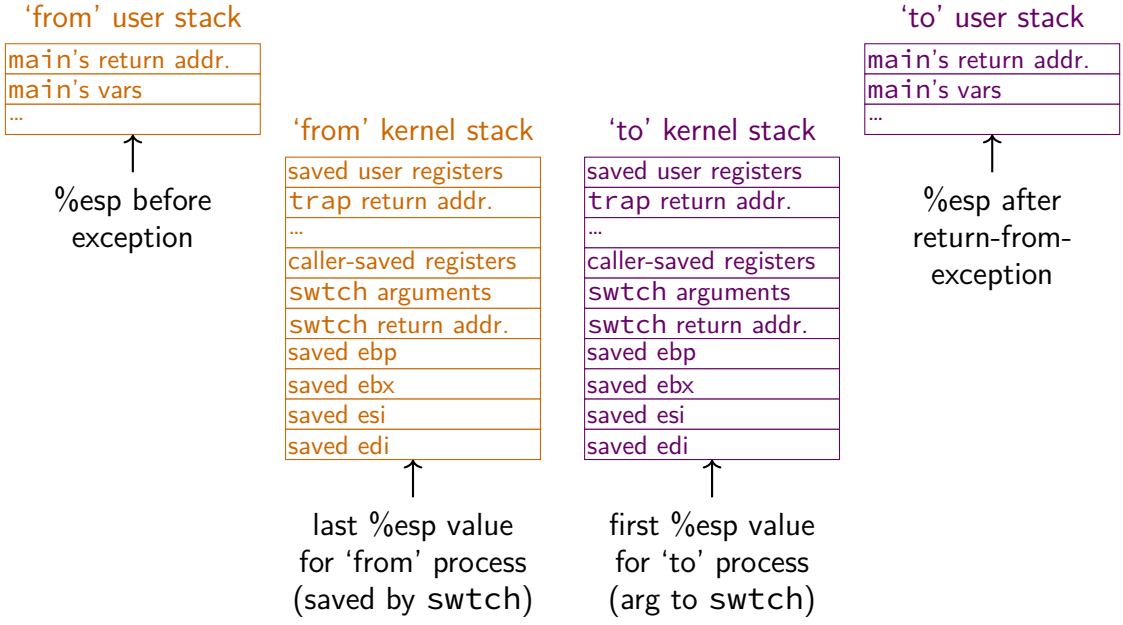
function to switch contexts
allocate space for context on top of stack
set `old` to point to it
switch to context `new`

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

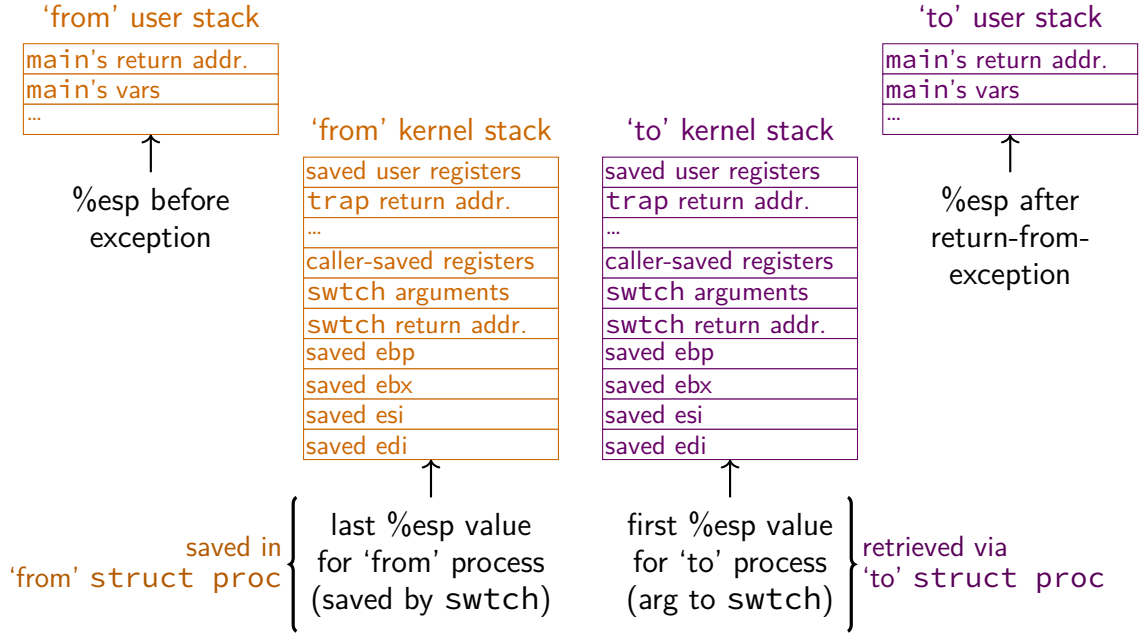
xv6: where the context is



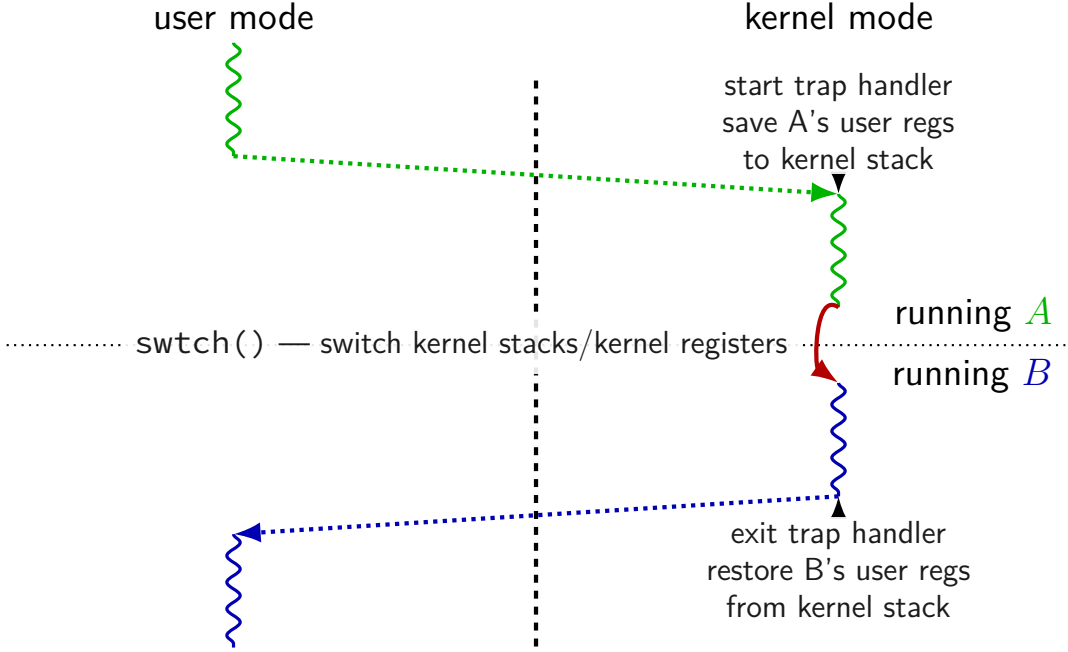
xv6: where the context is (detail)



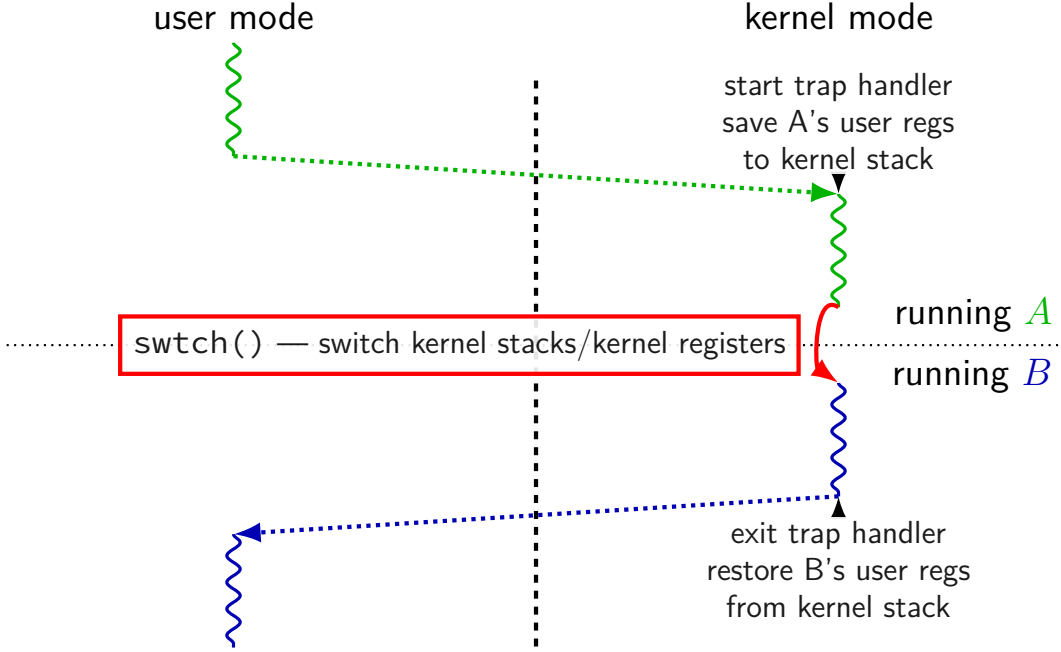
xv6: where the context is (detail)



xv6 context switch and saving



xv6 context switch and saving



xv6 context switch and saving