other exceptions / context switches

Changelog

16 January 2020: fix location of stack pointer indicator on swtch summary

22 January 2020: expand a little on I/O interrupts to separate them from I/O-requesting system calls

last time

logistics

kernel versus user mode

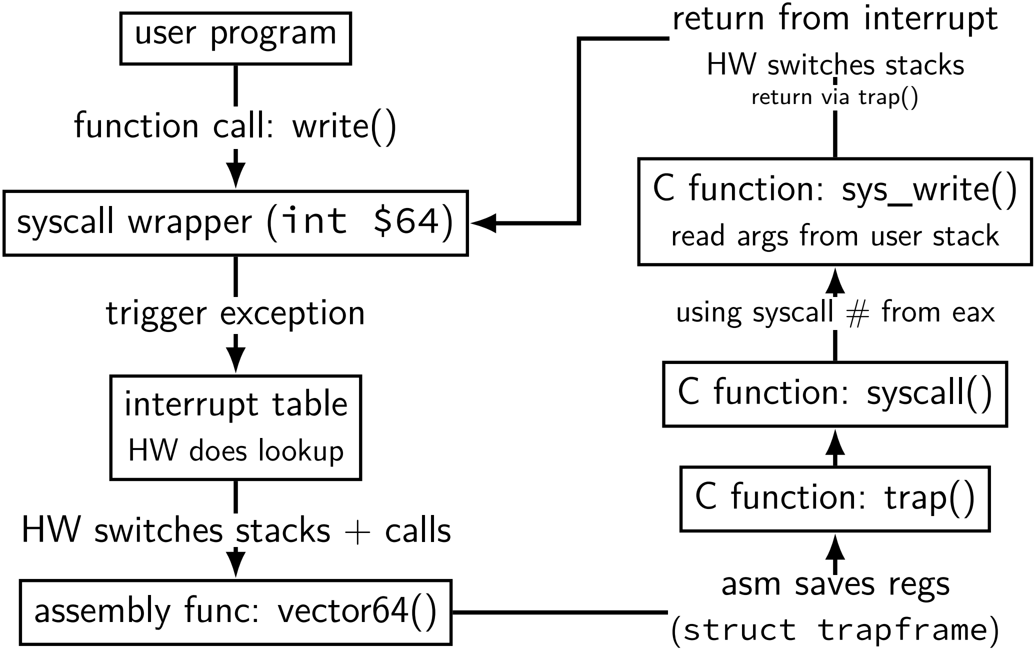
exceptions (AKA traps AKA ...): run OS when needed
- controlled mechanism for switching
- handling input
- keeping programs from running for too long

xv6 demo

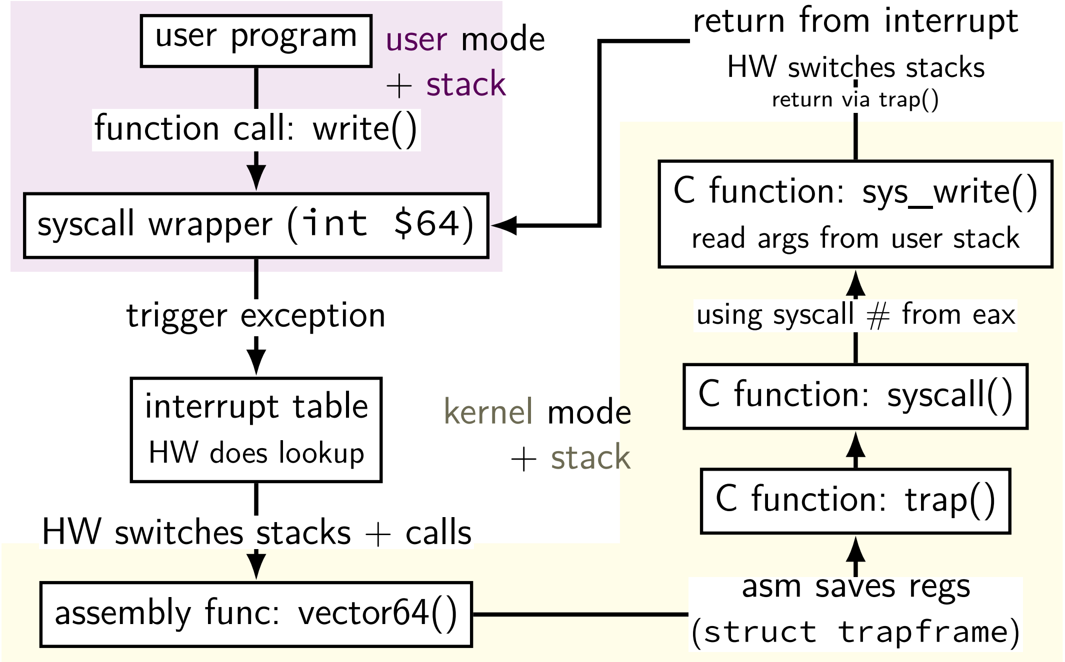
path of a system call in xv6

quiz demo

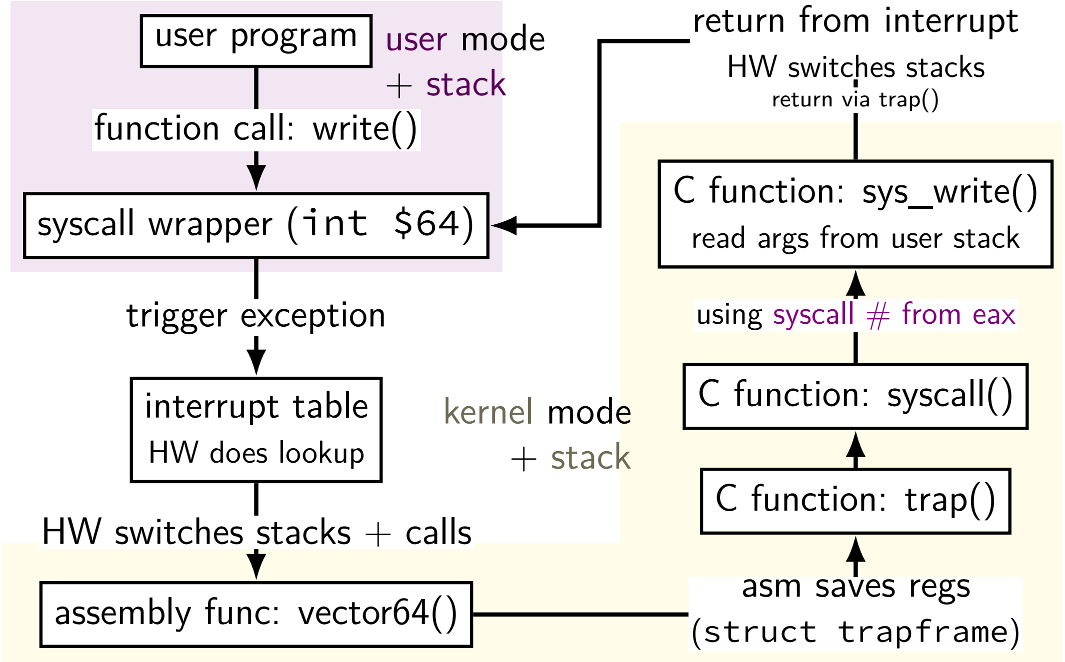
write syscall in xv6



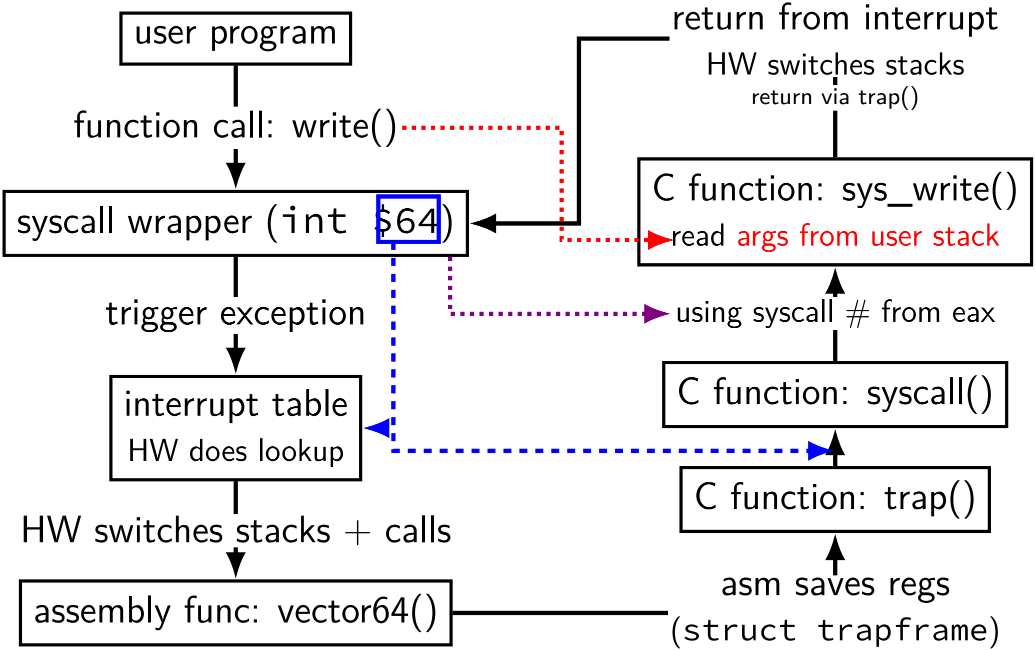
write syscall in xv6



write syscall in xv6



write syscall in xv6



xv6intro homework

get familiar with xv6 OS

add a new system call: `writecount()`

returns total number of times write call happened

add a new system call: `setwritecount(new_count)`

change the counter used by set writecount()

should continue counting number of write calls starting with new count

homework steps

system call implementation: sys_writecount

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

add writecount to several tables/lists

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

create userspace program(s) that calls writecount

- recommendation: copy from given programs

repeat, adding `setwritecount`

- see, e.g., `sys_kill` for example of retrieving argument

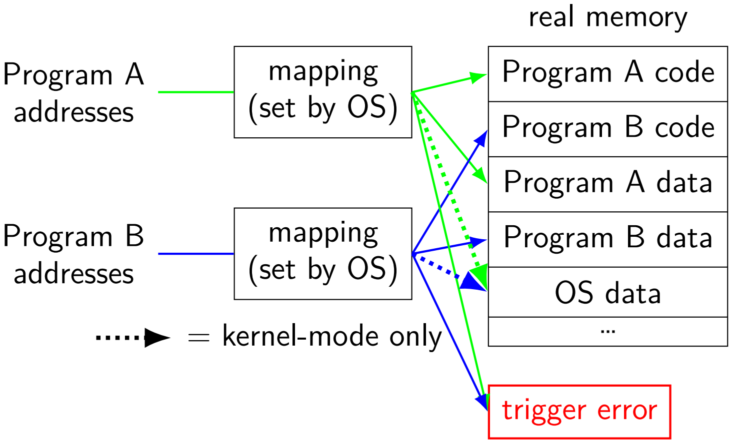
note on locks

some existing code we say to imitate uses acquire/release

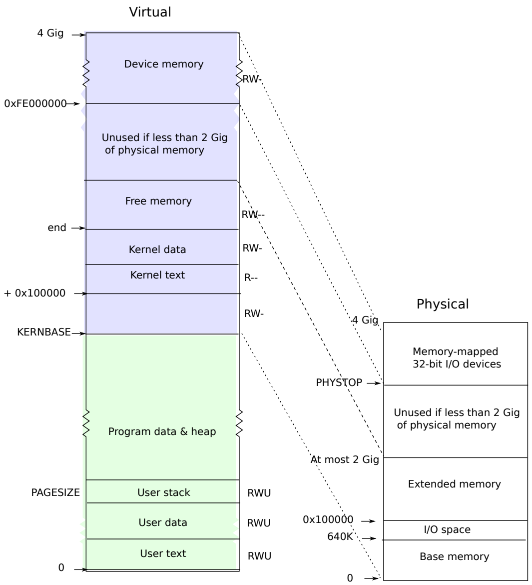
you do not have to do this

primarily to handle multiple cores

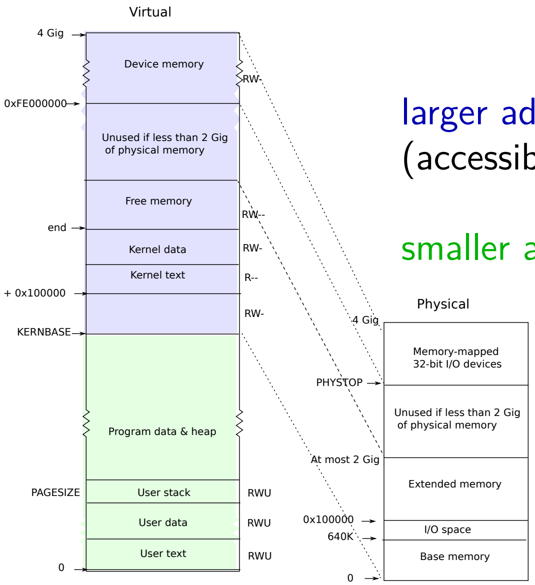
address translation



xv6 memory layout



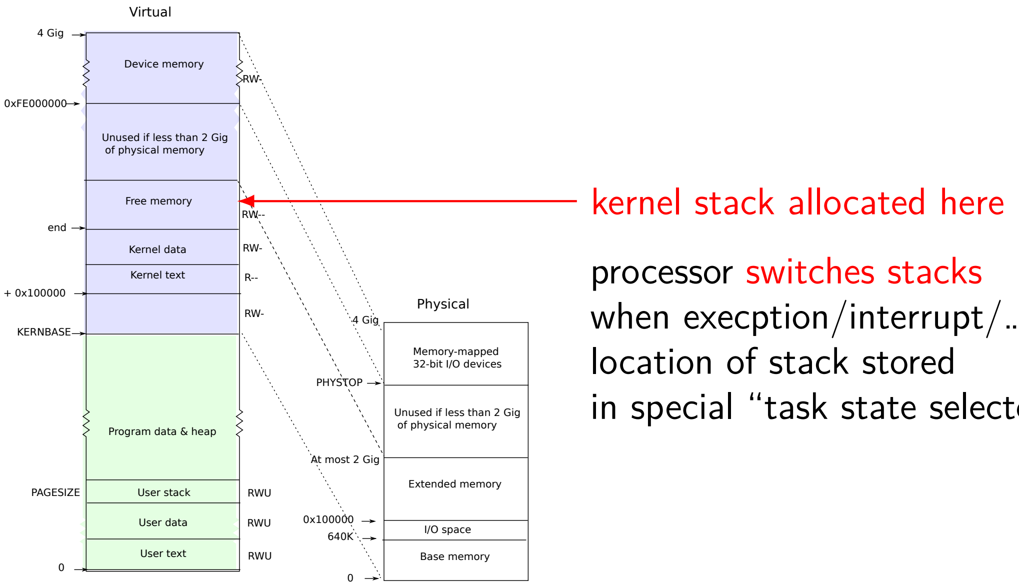
xv6 memory layout



larger addresses are for kernel (accessible in kernel mode *only*)

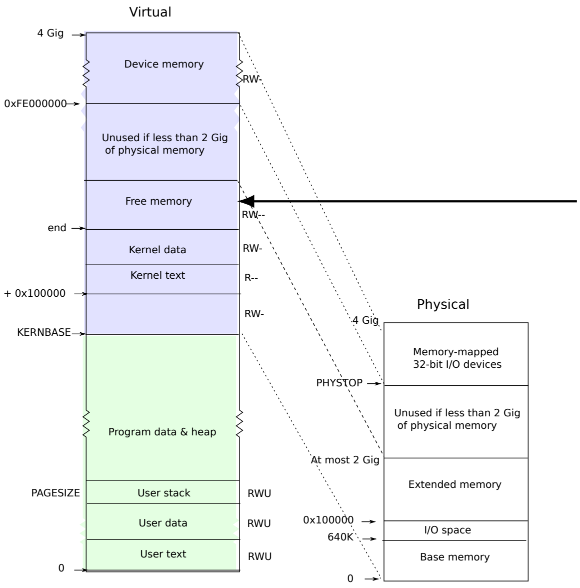
smaller addresses are for applications

xv6 memory layout



processor switches stacks
when execption/interrupt/...happens
location of stack stored
in special "task state selector"

xv6 memory layout



kernel stack**s** allocated here

one kernel stack per user thread

(plus extra stack for switching threads)

special register:
what stack for exception handler?
(stack changed by CPU (x86 feature)
along with saving old PC, etc.
xv6 sets register on thread switch)

separate stacks: design decision

many, but not all OSes use separate kernel stacks *per user thread*

makes writing system call handlers, etc. easier

- keep data on stack, even if system call involves waiting for a while
- possibly easier to figure out how big the stack should be?
- if only one kernel stack: need to save info outside stack while waiting

...but uses more space

- xv6: extra *4KB* of storage per thread/process

alternative: one kernel stack *per core*

aside: stack switching with nested exceptions

not nested: system call or other exception in user mode

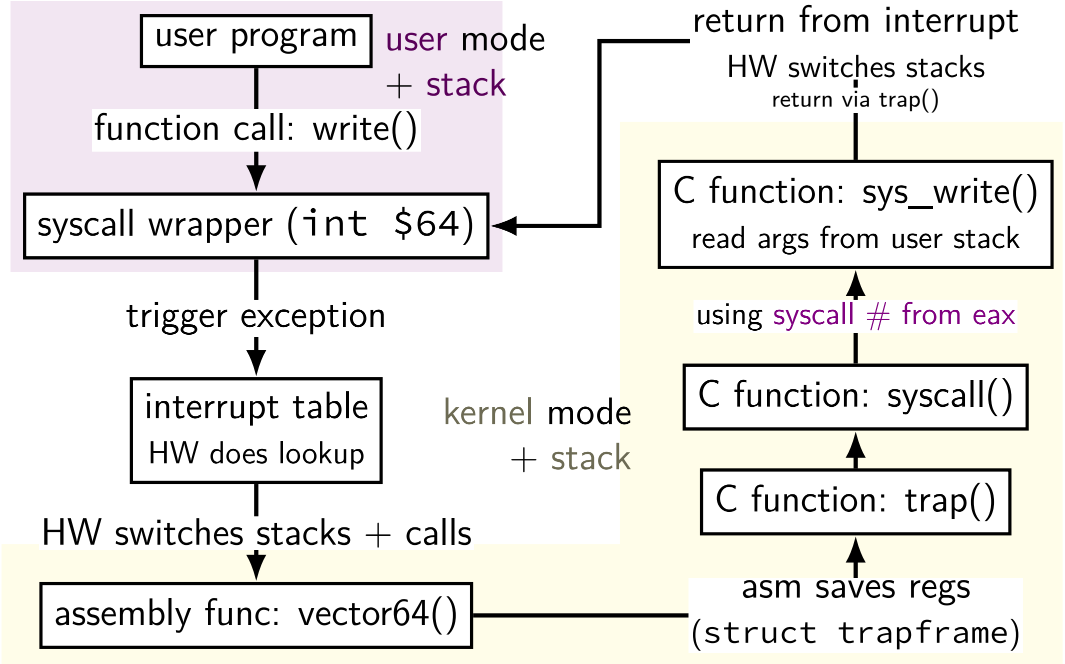
start in kernel at top of kernel stack for current thread/process

nested: exception (e.g. timer interrupt) during system call

continues using current kernel stack with same stack pointer

(processor tracks that it switched already)

write syscall in xv6



non-system call exceptions

xv6 handles many kinds of exceptions other than system calls

- recall: our orignal examples of why hardware had *exceptions*

timer interrupt — 'tick' from constantly running timer

- make sure infinite loop doesn't hog CPU
- check for programs waiting for time to pass

faults — e.g. access invalid memory, divide by zero

- xv6's action: kill the program

I/O — I/O device indicates that it requires OS action

- communicate with I/O device that now has data ready
- possibly wake up waiting programs

aside: interrupt descriptor table

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

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

shown earlier: being set for syscalls — SETGATE macro

xv6 sets all the table entries

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

- xv6 design choice: could have separate functions for each

xv6: interrupt table setup

trap.c (run on boot)

```
...
lidt(idt, sizeof(idt));
for (int i = 0; i < 256; i++)
  SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

set every entry of interrupt (descriptor) table to assembly function `vectors[i]` that saves registers, then calls `trap()`

non-system call exceptions

xv6 handles many kinds of exceptions other than system calls

- recall: our orignal examples of why hardware had *exceptions*

timer interrupt — ‘tick’ from constantly running timer

- make sure infinite loop doesn't hog CPU
- check for programs waiting for time to pass

faults — e.g. access invalid memory, divide by zero

- xv6's action: kill the program

I/O — I/O device indicates that it requires OS action

- communicate with I/O device that now has data ready
- possibly wake up waiting programs

xv6: faults

```
void
trap(struct trapframe *tf
{
  ...
  switch(tf->trapno) {
  ...
  default:
    ... // (not shown here: similar code for errors in kernel itsel
    cprintf("pid %d %s: trap %d err %d on cpu %d "
        "eip 0x%x addr 0x%x--kill proc\n",
        myproc()->pid, myproc()->name, tf->trapno,
        tf->err, cpuid(), tf->eip, rcr2());
    myproc()->killed = 1;
  }
}
```

exception not otherwise handled
(example: invalid memory access, divide-by-zer
print message and kill running program
assume it screwed up

xv6: faults

```
void
trap(struct trapframe *
{
  ...
  switch(tf->trapno) {
  ...
  default:
    ... // (not shown here: similar code for errors in kernel itsel
    cprintf("pid %d %s: trap %d err %d on cpu %d "
        "eip 0x%x addr 0x%x--kill proc\n",
        myproc()->pid, myproc()->name, tf->trapno,
        tf->err, cpuid(), tf->eip, rcr2());
    myproc()->killed = 1;
  }
}
```

prints out trap number

can lookup in `traps.h`

more featureful OS would lookup the name for y

non-system call exceptions

xv6 handles many kinds of exceptions other than system calls

- recall: our orignal examples of why hardware had *exceptions*

timer interrupt — 'tick' from constantly running timer

- make sure infinite loop doesn't hog CPU
- check for programs waiting for time to pass

faults — e.g. access invalid memory, divide by zero

- xv6's action: kill the program

I/O — I/O device indicates that it requires OS action

- communicate with I/O device that now has data ready
- possibly wake up waiting programs

xv6: I/O

```
void
trap(struct trapframe *tf)
{
  ...
  switch(tf->trapno) {
  ...
  case T_IRQ0 + IRQ_IDE:
    ideintr();
    lapiceoi();
    break;
  ...
  case T_IRQ0 + IRQ_KBD:
    kbdintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_COM1:
    uartintr();
    lapiceoi();
    break;
```

ide = disk interface
kbd = keyboard
uart = serial port (external terminal)

exception indicates: data now ready
handlers arrange for data to be sent to appropriate application(s)

non-system call exceptions

xv6 handles many kinds of exceptions other than system calls

- recall: our orignal examples of why hardware had *exceptions*

timer interrupt — 'tick' from constantly running timer

- make sure infinite loop doesn't hog CPU
- check for programs waiting for time to pass

faults — e.g. access invalid memory, divide by zero

- xv6's action: kill the program

I/O — I/O device indicates that it requires OS action

- communicate with I/O device that now has data ready
- possibly wake up waiting programs

xv6: timer interrupt

```
void
trap(struct trapframe *tf)
{
  ...
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
  ...
  // Force process to give up CPU on clock tick.
  ...
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();
  ...
```

xv6: timer interrupt

```
void
trap(struct trapf
{
  ...
  switch(tf->trap
  case T_IRQ0 + I
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
  ...
  // Force process to give up CPU on clock tick.
  ...
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();
  ...
```

on timer interrupt
(trigger periodically by external timer):
if a process is running
yield = maybe switch to different program

xv6: timer interrupt

```
void
trap(struct trapframe *tf)
{
  ...
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
  ...
  // Force process to give up CPU on clock tick.
  ...
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();
  ...
```

on timer interrupt:

wakeup — handle waiting processes

certain amount of time

(`sleep` system call)

xv6: timer interrupt

lapiceoi — tell hardware we have handled this interrupt
(needed for all interrupts from 'external' devices)

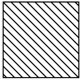
```
void
trap(struct trapf
{
  ...
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
  ...
  // Force process to give up CPU on clock tick.
  ...
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();
  ...
```

xv6: timer interrupt

acquire/release — related to synchronization (later)

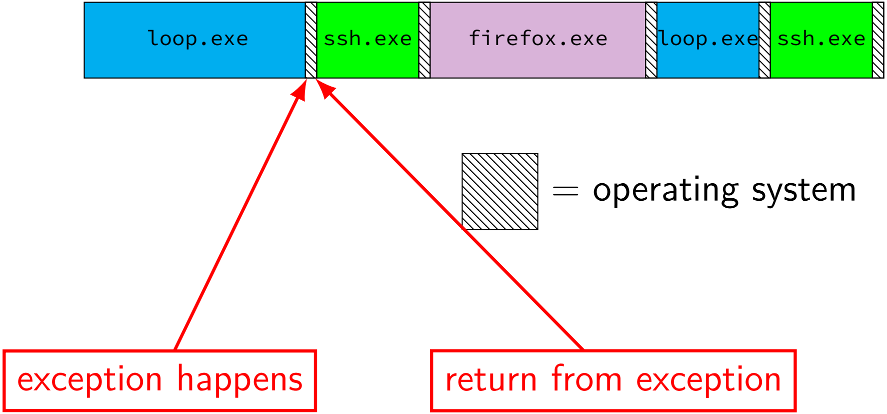
```
void
trap(struct trapf
{
  ...
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
  ...
  // Force process to give up CPU on clock tick.
  ...
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();
  ...
```

time multiplexing



= operating system

time multiplexing



OS and time multiplexing

starts running instead of normal program via exception

saves old program counter, registers somewhere

sets new registers, jumps to new program counter

called context switch

- saved information called context

context

all registers values

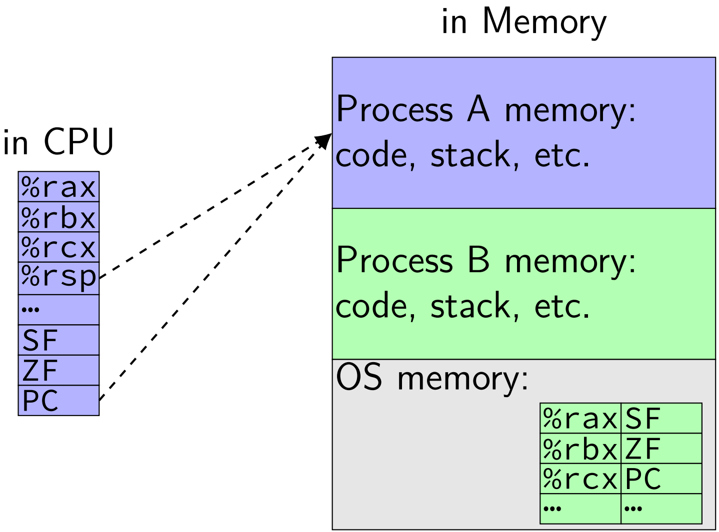
`%rax` `%rbx`, …, `%rsp`, …

condition codes

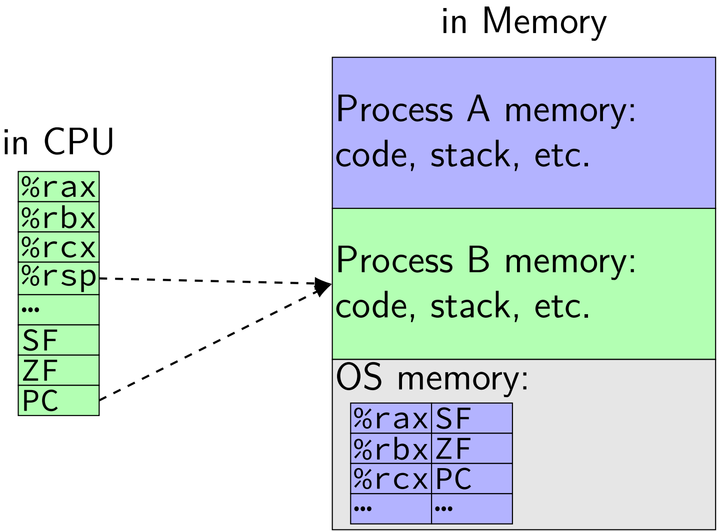
program counter

address space = page table base pointer

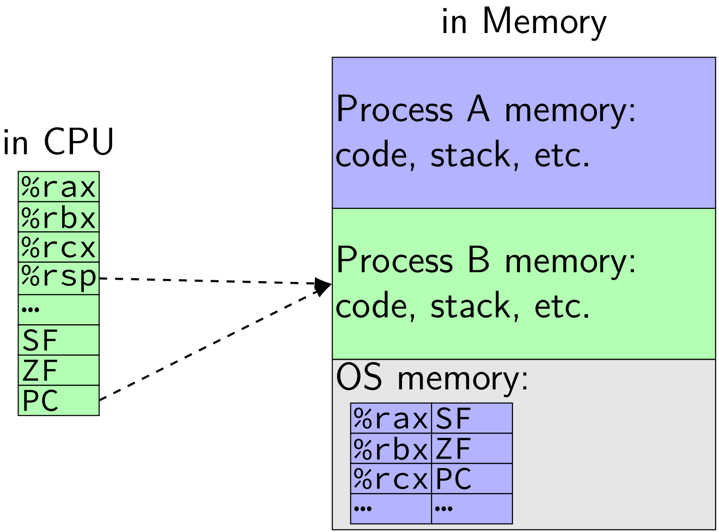
contexts (A running)



contexts (B running)



contexts (B running)



xv6: A's registers saved by exception handler into "trapframe" on A's kernel stack

exercise: counting context switches

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 system calls do we expect?

how many context switches 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

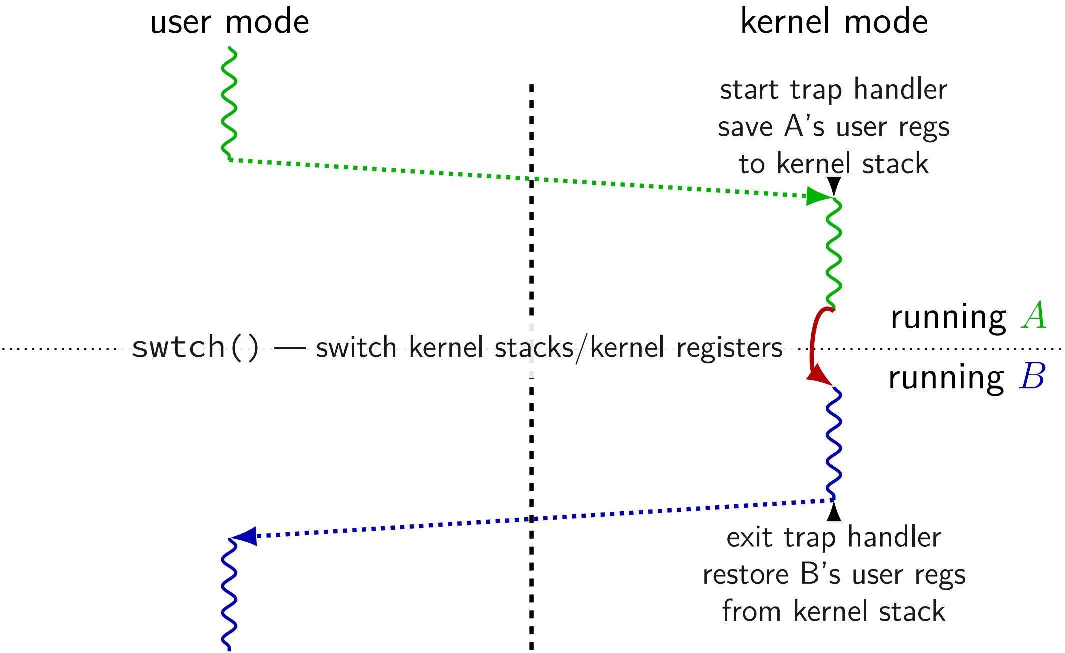
(3?) **switch to A while waiting for disk?**

if data from file not available right away

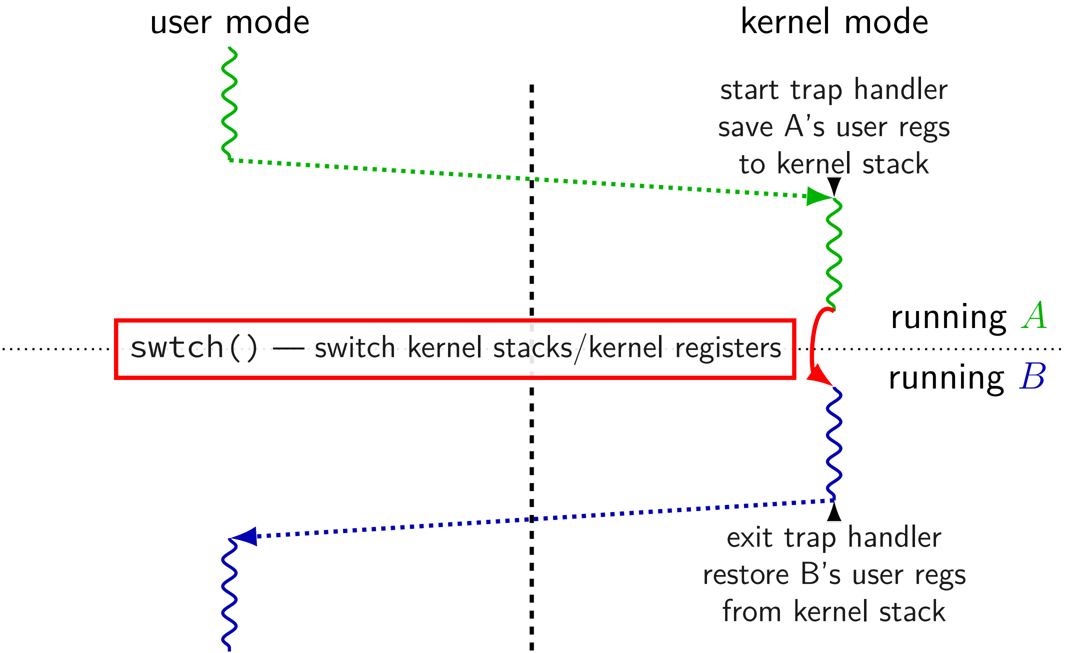
(4) **switch to B to do computation + write system call**

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

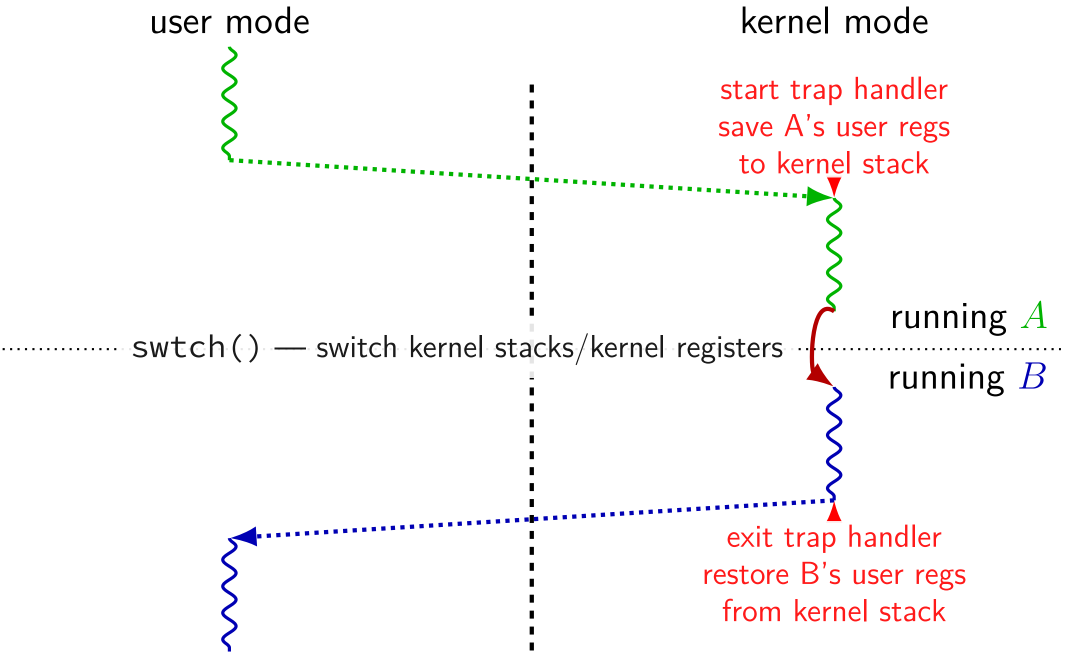
xv6 context switch and saving



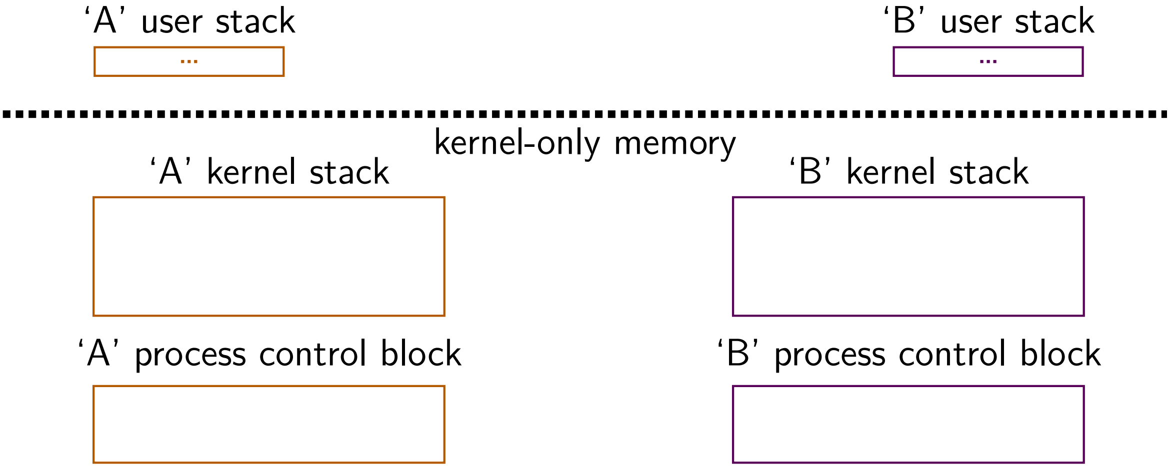
xv6 context switch and saving



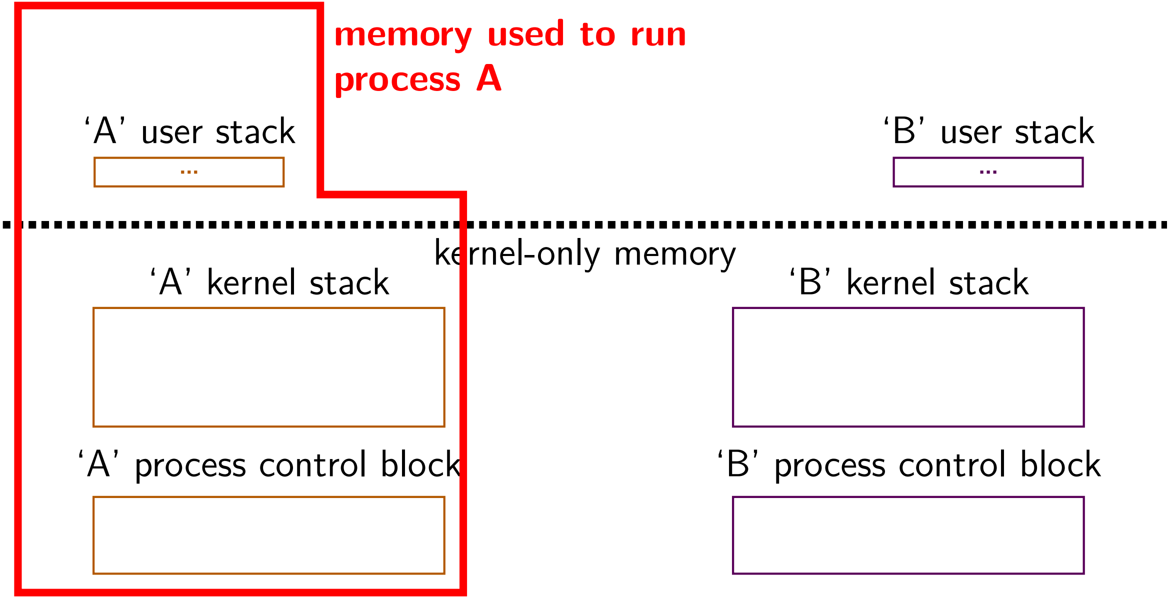
xv6 context switch and saving



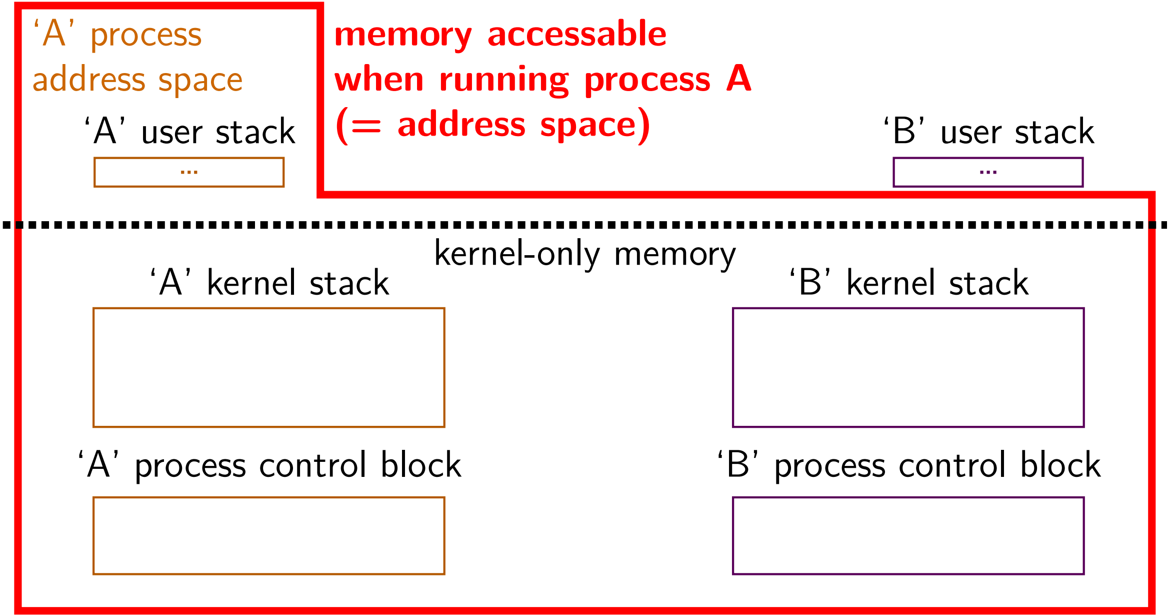
xv6: where the context is



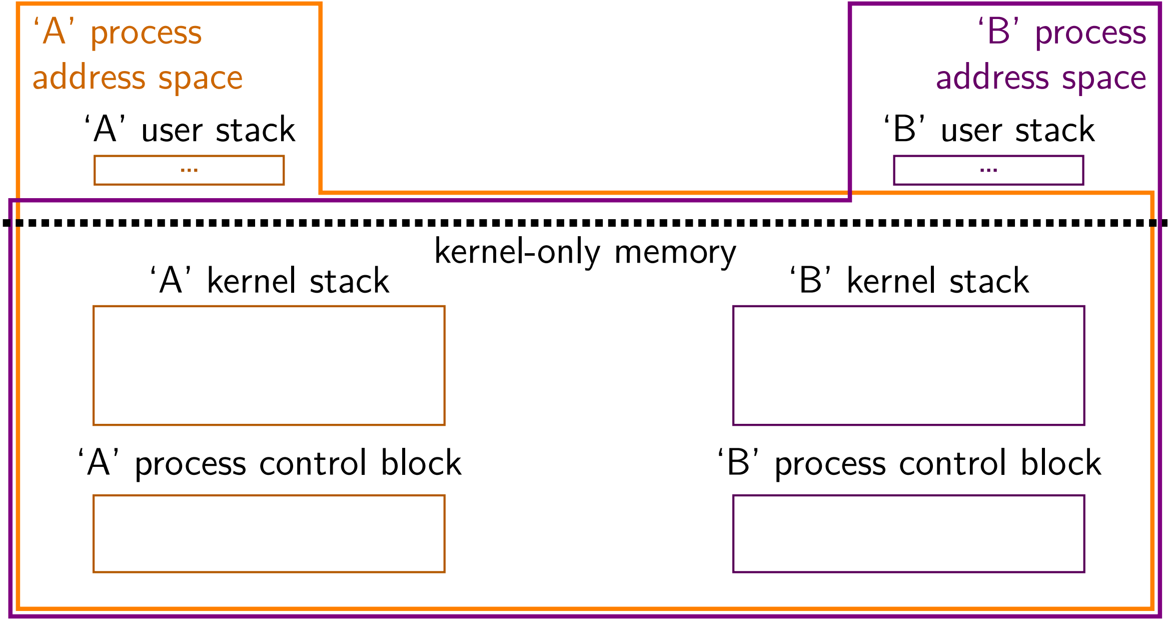
xv6: where the context is



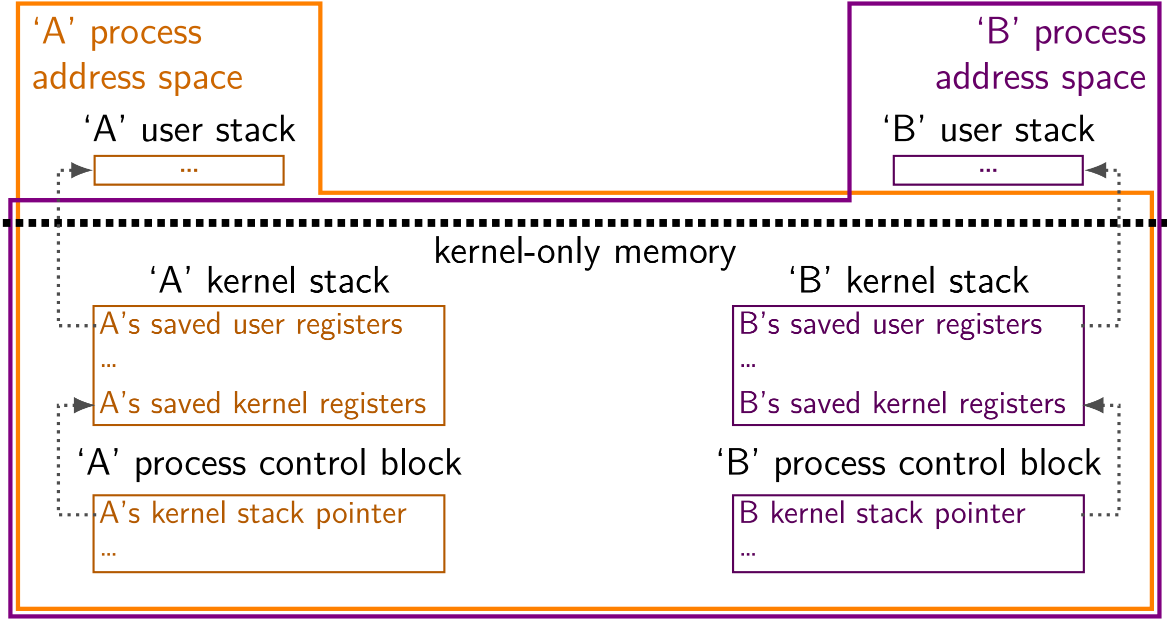
xv6: where the context is



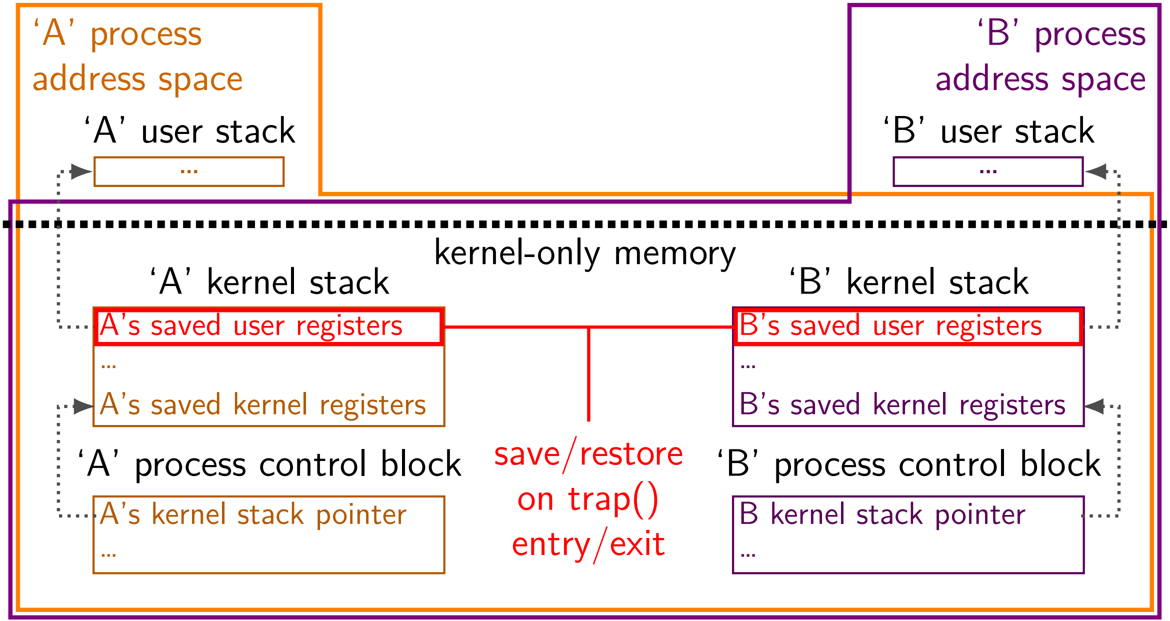
xv6: where the context is



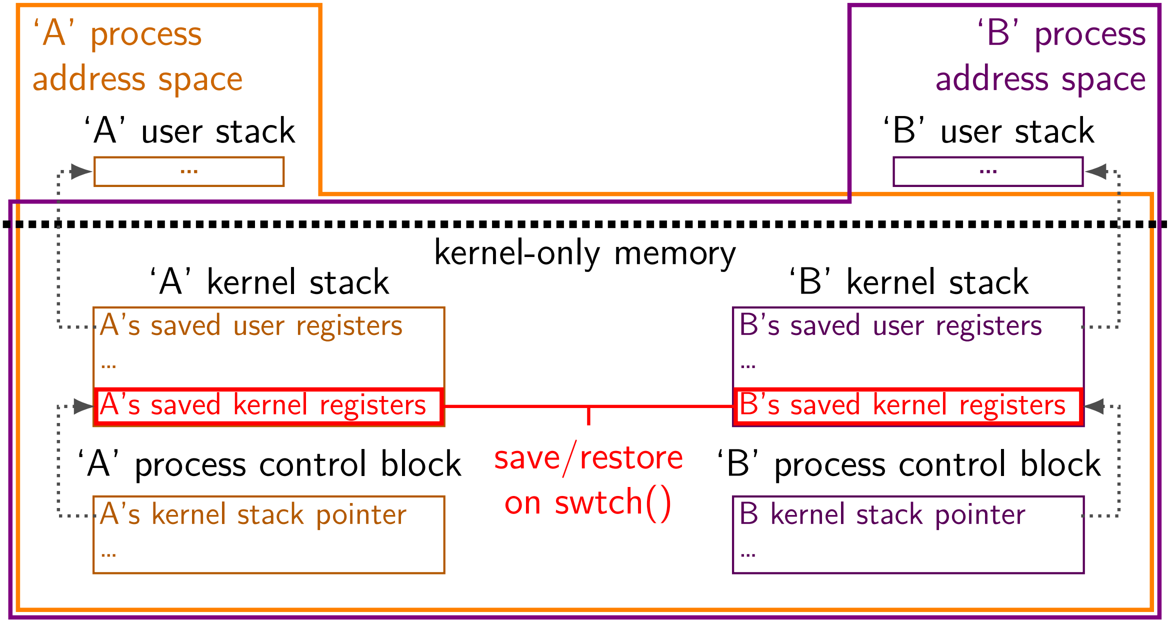
xv6: where the context is



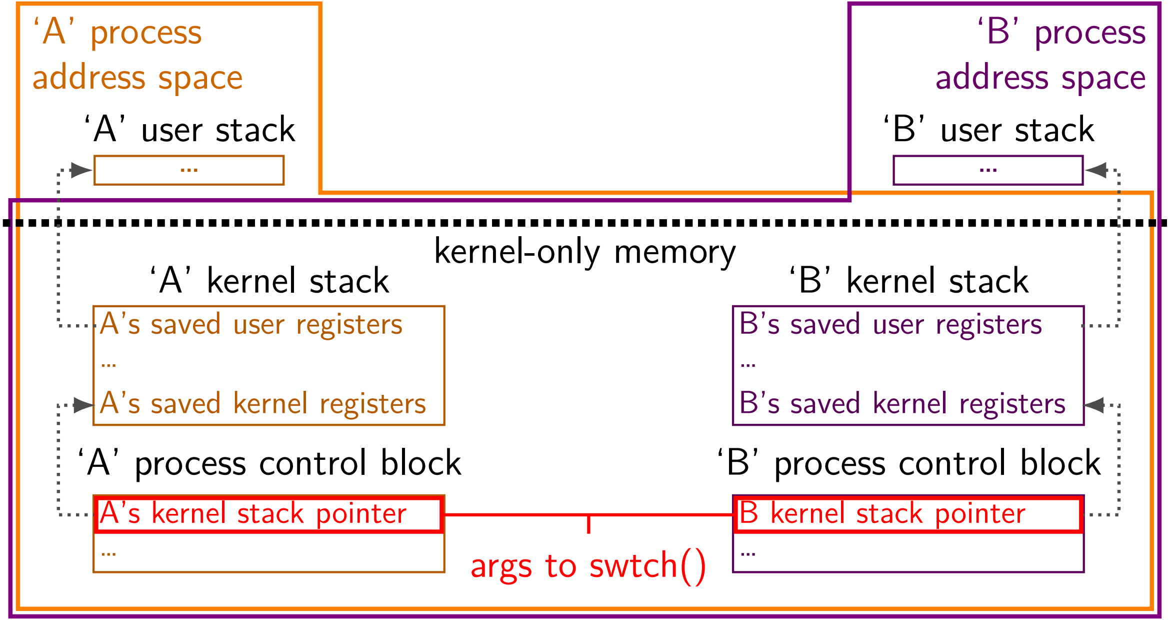
xv6: where the context is



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

in thread A:

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

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

---

in thread B:

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

thread switching in xv6: C

in thread A:

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

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

---

in thread B:

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

thread switching in xv6: C

in thread A:

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

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

---

in thread B:

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

thread switching in xv6: how?

swtch(A, B) pseudocode:

save caller-saved registers to stack

write swtch return address to stack

write all callee-saved registers to stack

save old stack pointer into arg *A*

read *B* arg as new stack pointer

read all callee-saved registers from stack

read+use swtch return address from stack

restore caller-saved registers from stack

old (A) stack

...

new (B) stack

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

thread switching in xv6: how?

swtch(A, B) pseudocode:

save caller-saved registers to **stack**

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

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

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

read *B* arg as new *stack* pointer

read all callee-saved registers from *stack*

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

restore caller-saved registers from *stack*

old (A) **stack**

…

new (B) *stack*

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

thread switching in xv6: how?

swtch(A, B) pseudocode:

save caller-saved registers to **stack**

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

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

save old **stack** pointer into arg A

read B arg as new *stack* pointer

read all callee-saved registers from *stack*

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

restore caller-saved registers from *stack*

old (A) **stack**

...
caller-saved registers
`swtch` arguments ← SP

new (B) *stack*

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

thread switching in xv6: how?

swtch(A, B) pseudocode:

save caller-saved registers to **stack**

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

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

save old **stack** pointer into arg A

read B arg as new *stack* pointer

read all callee-saved registers from *stack*

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

restore caller-saved registers from *stack*

old (A) **stack**

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

SP → `swtch` return addr.

new (B) *stack*

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

thread switching in xv6: how?

swtch(A, B) pseudocode:

save caller-saved registers to **stack**

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

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

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

read *B* arg as new *stack* pointer

read all callee-saved registers from *stack*

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

restore caller-saved registers from *stack*

old (A) **stack**

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

new (B) *stack*

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

thread switching in xv6: how?

swtch(A, B) pseudocode:

save caller-saved registers to **stack**

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

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

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

read *B* arg as new *stack* pointer

read all callee-saved registers from *stack*

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

restore caller-saved registers from *stack*

old (A) **stack**

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

new (B) *stack*

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

thread switching in xv6: how?

swtch(A, B) pseudocode:

save caller-saved registers to **stack**

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

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

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

read *B* arg as new *stack* pointer

read all callee-saved registers from *stack*

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

restore caller-saved registers from *stack*

old (A) **stack**

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

new (B) *stack*

	…
	caller-saved registers
	`swtch` arguments
	`swtch` return addr.
SP →	callee-saved registers

thread switching in xv6: how?

swtch(A, B) pseudocode:

save caller-saved registers to **stack**

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

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

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

read *B* arg as new *stack* pointer

read all callee-saved registers from *stack*

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

restore caller-saved registers from *stack*

old (A) **stack**

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

new (B) *stack*

	…
	caller-saved registers
	`swtch` arguments
SP →	`swtch` return addr.
	callee-saved registers

thread switching in xv6: how?

swtch(A, B) pseudocode:

save caller-saved registers to **stack**

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

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

save old **stack** pointer into arg A

read B arg as new *stack* pointer

read all callee-saved registers from *stack*

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

restore caller-saved registers from *stack*

old (A) **stack**

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

new (B) *stack*

	...
	caller-saved registers
SP $\rightarrow$	`swtch` arguments
	`swtch` return addr.
	callee-saved registers

thread switching in xv6: how?

swtch(A, B) pseudocode:

save caller-saved registers to **stack**

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

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

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

read *B* arg as new *stack* pointer

read all callee-saved registers from *stack*

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

restore caller-saved registers from *stack*

old (A) **stack**

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

new (B) *stack*

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

thread switching in xv6: how?

swtch(A, B) pseudocode:

save caller-saved registers to **stack**

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

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

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

read *B* arg as new *stack* pointer

read all callee-saved registers from *stack*

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

restore caller-saved registers from *stack*

old (A) **stack**
old (A) **stack**

saved user regs
…
caller-saved registers
`swtch` arguments
`swtch` return addr.
callee-saved registers

new (B) *stack*
new (B) *stack*

saved user regs
…
caller-saved registers
`swtch` arguments
`swtch` return addr.
callee-saved registers

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

```
.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
```

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

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

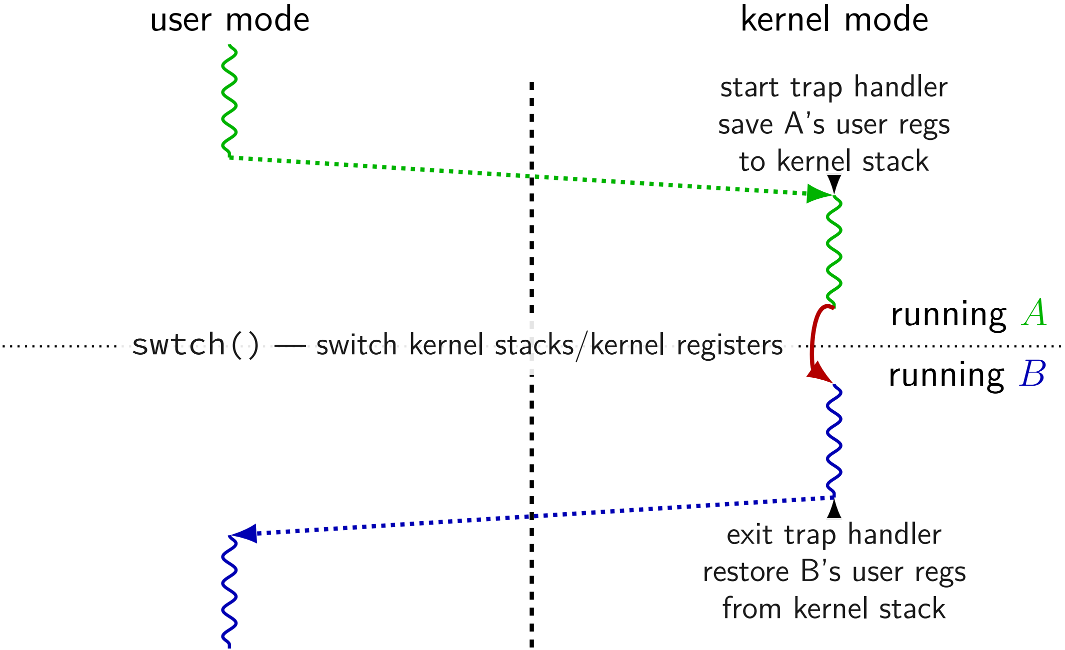
the userspace part?

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)

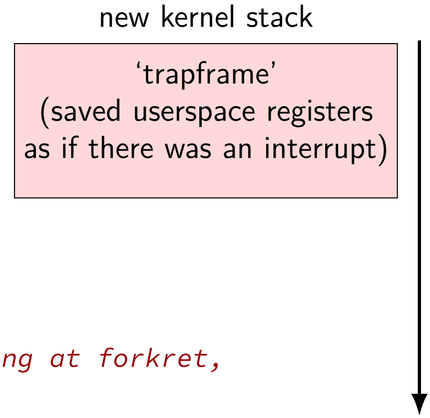
creating a new thread

new kernel stack

```c
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

```c
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

new kernel stack

'trapframe'
(saved userspace registers as if there was an interrupt)

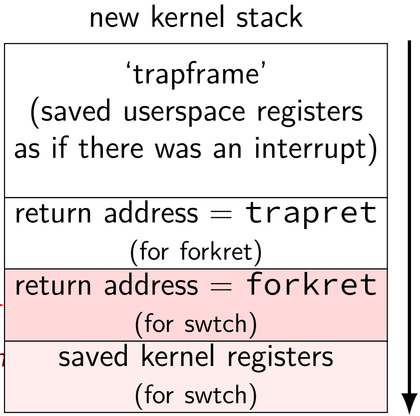
return address = `trapret`
(for forkret)

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;
  ...
```

initial code to run
when starting a new process

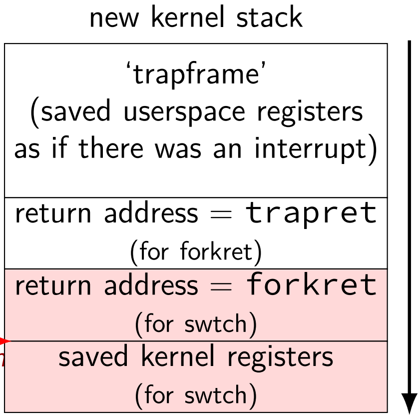
(fork = process creation system call)



creating a new thread

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

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

  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;
  ...
```

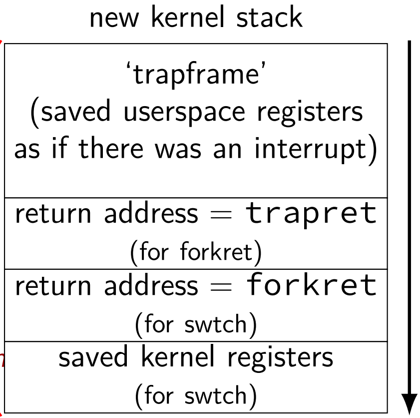
saved registers (incl. return address) for swtch to pop off the stack



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;
  ...
```

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
};
```

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

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

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

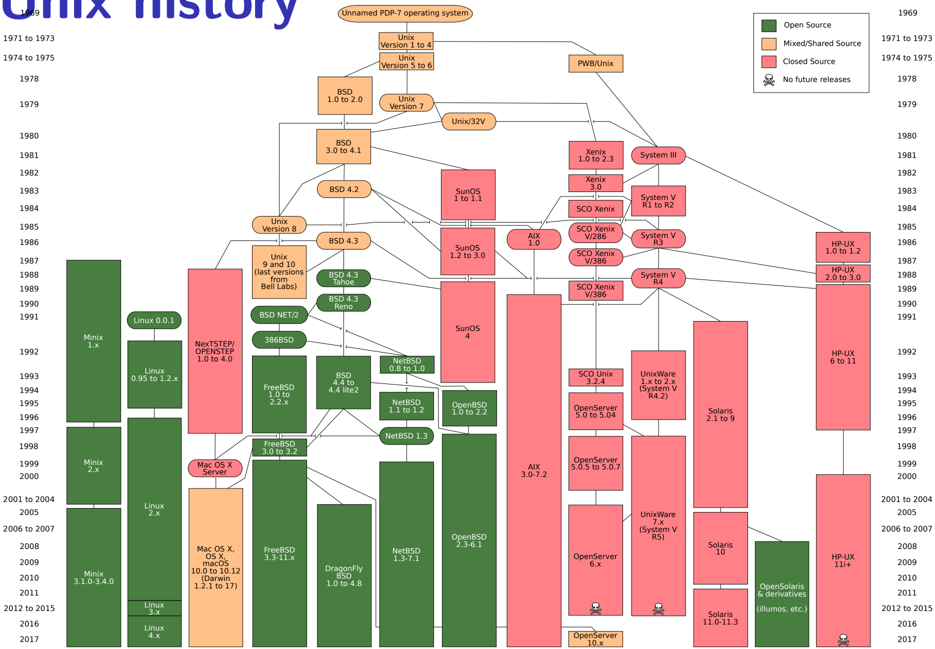
we have source code

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

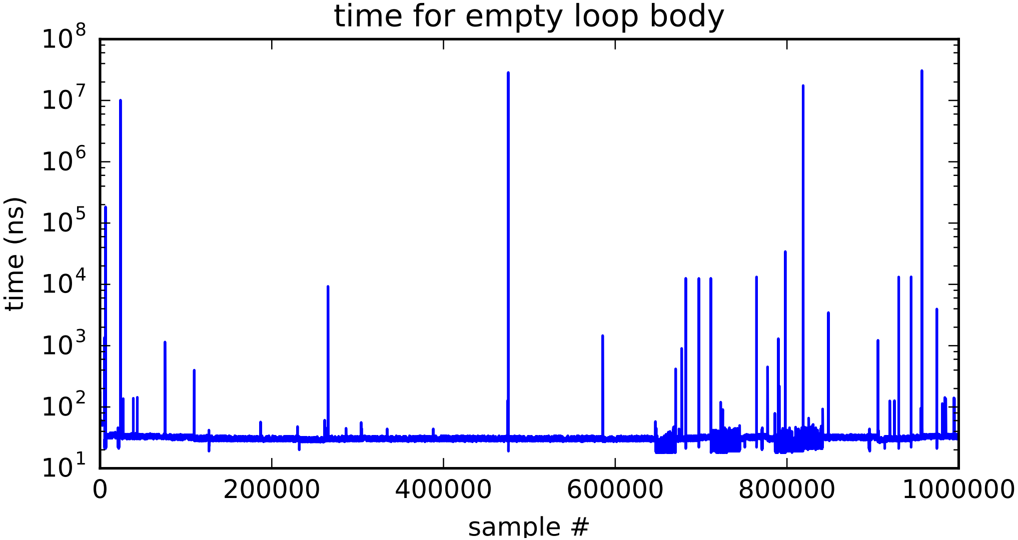
backup slides

timing nothing

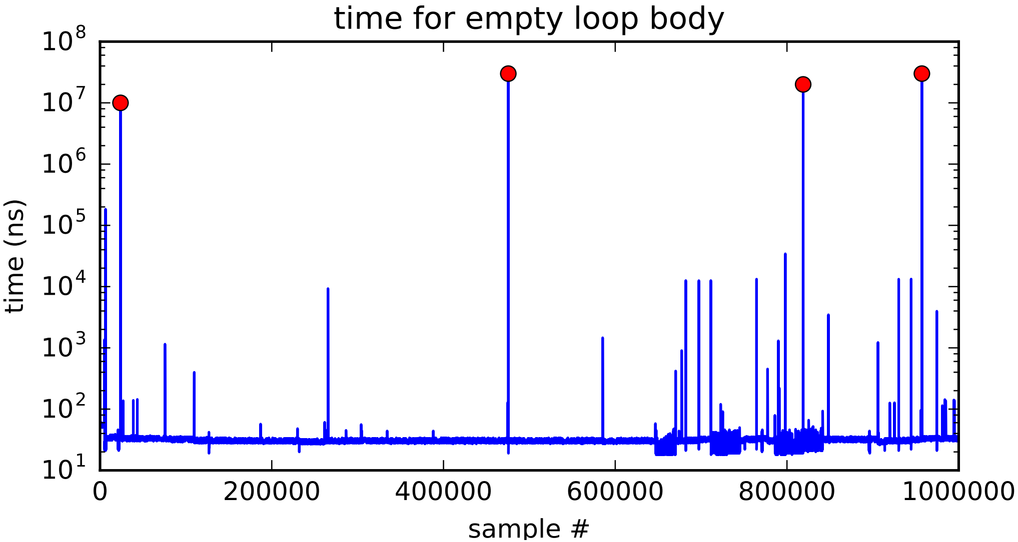
```
long times[NUM_TIMINGS];
int main(void) {
    for (int i = 0; i < N; ++i) {
        long start, end;
        start = get_time();
        /* do nothing */
        end = get_time();
        times[i] = end - start;
    }
    output_timings(times);
}
```

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
```

from stack
caller-saved registers
swtch arguments
swtch return addr.

to stack
caller-saved registers
swtch arguments
swtch return addr.
saved ebp
saved ebx
saved esi
saved edi

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
```

from stack

caller-saved registers
swtch arguments
swtch return addr.

%esp →

to stack

caller-saved registers
swtch arguments
swtch return addr.
saved ebp
saved ebx
saved esi
saved edi

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
```

	from stack	to stack
	caller-saved registers	caller-saved registers
	swtch arguments	swtch arguments
	swtch return addr.	swtch return addr.
	saved ebp	saved ebp
	saved ebx	saved ebx
	saved esi	saved esi
%esp →	saved edi	saved edi

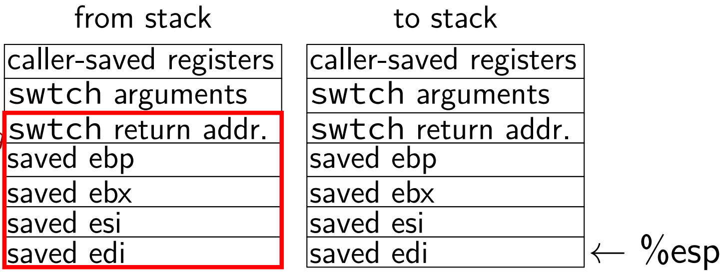
juggling stacks

```
.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
```

from stack	to stack	
caller-saved registers	caller-saved registers	
swtch arguments	swtch arguments	
swtch return addr.	swtch return addr.	
saved ebp	saved ebp	
saved ebx	saved ebx	
saved esi	saved esi	
saved edi	saved edi	← %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
```



struct context
(saved into from arg)

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
```

from stack
caller-saved registers
swtch arguments
swtch return addr.
saved ebp
saved ebx
saved esi
saved edi

to stack	
caller-saved registers	
swtch arguments	
swtch return addr.	← %esp
saved ebp	
saved ebx	
saved esi	
saved edi	

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
```

from stack	to stack	
caller-saved registers	caller-saved registers	
swtch arguments	swtch arguments	← %esp
swtch return addr.	swtch return addr.	
saved ebp	saved ebp	
saved ebx	saved ebx	
saved esi	saved esi	
saved edi	saved edi	

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
```

← first instruction executed by new thread

from stack
caller-saved registers
swtch arguments
swtch return addr.
saved ebp
saved ebx
saved esi
saved edi

to stack	
caller-saved registers	
swtch arguments	
swtch return addr.	← %esp

↑ bottom of new kernel stack

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
```

from stack
saved user regs
…
caller-saved registers
swtch arguments
swtch return addr.
saved ebp
saved ebx
saved esi
saved edi

to stack
saved user regs
…
caller-saved registers
swtch arguments
swtch return addr.
saved ebp
saved ebx
saved esi
saved edi

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

write syscall in xv6: user mode

main.c

```
...
write(1,
      "Hello, World!\n",
      14);
...
```

syscall.h / traps.h

```
...
#define SYS_write   16
...
#define T_SYSCALL   64
...
```

usys.S

(partial, after macro replacement)

```
.globl write
write:
    movl $SYS_write, %eax
    int $T_SYSCALL
    ret
```

write syscall in xv6: user mode

main.c

```
...
write(1,
      "Hello, World!\n",
      14);
...
```

syscall.h / traps.h

```
...
#define SYS_write   16
...
#define T_SYSCALL   64
...
```

usys.S

(partial, after macro replacement)

```
.globl write
write:
    movl $SYS_write, %eax
    int $T_SYSCALL
    ret
```

interrupt — trigger an exception similar to a keypress

parameter (64 in this case) — type of exception

write syscall in xv6: user mode

main.c

```
...
write(1,
      "Hello, World!\n",
      14);
...
```

syscall.h / traps.h

```
...
#define SYS_write    16
...
#define T_SYSCALL    64
...
```

usys.S

(partial, after macro replacement)

```
.globl write
write:
    movl $SYS_write, %eax
    int $T_SYSCALL
    ret
```

xv6 syscall calling convention:
eax = syscall number
otherwise: same as 32-bit x86 calling convention (arguments *on stack*)

write syscall in xv6: interrupt table setup

trap.c (run on boot)

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

write syscall in xv6: interrupt table setup

trap.c (run on boot)

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

`lidt` —
function (in x86.h) wrapping `lidt` instruction

sets the *interrupt descriptor table* to *idt*
idt = array of pointers to *handler functions* for each exception type
(plus a few bits of information about those handler functions)

write syscall in xv6: interrupt table setup

trap.c (run on boot)

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

(from mmu.h):

```
// Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap gate, 0 for an interrupt gate.
//   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level -
//        the privilege level required for software to invoke
//        this interrupt/trap gate explicitly using an int instruction.
#define SETGATE(gate, istrap, sel, off, d)                \
```

write syscall in xv6: interrupt table setup

trap.c (run on boot)

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

`vectors[T_SYSCALL]` — OS function for processor to run

set to pointer to assembly function `vector64`

eventually calls C function `trap`

write syscall in xv6: interrupt table setup

trap.c (run on boot)

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

set the T_SYSCALL interrupt to
be callable from user mode via **int** instruction
(otherwise: triggers fault like privileged instruction)

write syscall in xv6: interrupt table setup

trap.c (run on boot)

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

set it to use the kernel "code segment"

meaning: run in kernel mode

(yes, code segments specifies more than that — nothing we care about)

write syscall in xv6: interrupt table setup

trap.c (run on boot)

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

1: do not disable interrupts during syscalls
e.g. keypress/timer handling can interrupt slow syscall

write syscall in xv6: interrupt table setup

trap.c (run on boot)

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

1: do not disable interrupts during syscalls

e.g. keypress/timer handling can interrupt slow syscall

con: makes writing system calls safely more complicated

(what if keypress handler runs during system call?)

pro: slow system calls don't stop timers, keypresses, etc. from working

non-system call exceptions: interrupts disabled

write syscall in xv6: interrupt table setup

trap.c (run on boot)

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

`vectors[T_SYSCALL]` — OS function for processor to run

set to pointer to assembly function `vector64`

eventually calls C function `trap`

hardware jumps here

vectors.S

```
vector64:
  pushl $0
  pushl $64
  jmp alltraps
...
```

trapasm.S

```
alltraps:
  ...
  call trap
  ...
  iret
```

trap.c

```
void
trap(struct trapframe *tf)
{
...
```

write syscall in xv6: the trap function

trap.c

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
  ...
}
```

write syscall in xv6: the trap function

trap.c

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
  ...
}
```

struct trapframe — set by assembly

interrupt type, application registers, ...

example: `tf->eax` = old value of eax

write syscall in xv6: the trap function

trap.c

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
  ...
}
```

`myproc()` — pseudo-global variable

represents currently running process

much more on this later in semester

write syscall in xv6: the trap function

trap.c

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
  ...
}
```

syscall() — actual implementations uses `myproc()->tf` to determine what operation to do for program

write syscall in xv6: the syscall function

syscall.c

```
static int (*syscalls[])(void) = {
...
[SYS_write]  sys_write,
...
};

...

void
syscall(void)
{
...
  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
...
```

write syscall in xv6: the syscall function

syscall.c

```
static int (*syscalls[])(void) = {
...
[SYS_write]  sys_write,
...
};

...

void
syscall(void)
{
...
  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
...
```

array of functions — one for syscall

'`[number] value`': syscalls[number] = value

write syscall in xv6: the syscall function

syscall.c

```
static int (*syscalls[])(void) = {
...
[SYS_write]  sys_write,
...
};

...

void
syscall(void)
{
...
  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
...
```

(if system call number in range)
call sys_...function from table
store result in user's `eax` register

write syscall in xv6: the syscall function

syscall.c

```
static int (*syscalls[])(void) = {
...
[SYS_write]  sys_write,
...
};

...

void
syscall(void)
{
...
  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
...
```

result assigned to eax (assembly code this returns to copies `tf->eax` into `%eax`)

write syscall in xv6: sys_write

sysfile.c

```
int
sys_write(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return filewrite(f, p, n);
}
```

write syscall in xv6: sys_write

sysfile.c

```
int
sys_write(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return filewrite(f, p, n);
}
```

utility functions that read arguments from user's stack
returns -1 on error (e.g. stack pointer invalid)
(more on this later)
(note: 32-bit x86 calling convention puts all args on stack)

write syscall in xv6: sys_write

sysfile.c

```
int
sys_write(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return filewrite(f, p, n);
}
```

actual internal function that implements writing to a file
(the terminal counts as a file)

write syscall in xv6: interrupt table setup

trap.c (run on boot)

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

`trap` returns to `alltraps`

alltraps restores registers from `tf`, then returns to user-mode

hardware jumps here

vectors.S

```
vector64:
  pushl $0
  pushl $64
  jmp alltraps
...
```

trapasm.S

```
alltraps:
  ...
  call trap
  ...
  iret
```

trap.c

```
void
trap(struct trapframe *tf)
{
...
```

xv6: keyboard I/O

```
void
kbdintr(void)
{
  consoleintr(kbdgetc);
}
...
void consoleintr(...)
{
  ...
      wakeup(&input.r);
  ...
}
```

xv6: keyboard I/O

```
void
kbdintr(void)
{
  consoleintr(kbdgetc);
}
...
void consoleintr(...)
{
  ...
      wakeup(&input.r);
  ...
}
```

finds process waiting on console
make it run soon
(xv6 choice: usually not immediately)

time multiplexing

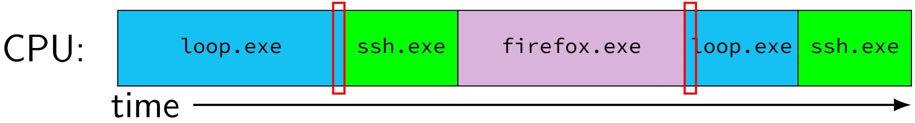
time multiplexing


```
...
call get_time
    // whatever get_time does
movq %rax, %rbp
```

million cycle delay (from loop.exe's view)

```
call get_time
    // whatever get_time does
subq %rbp, %rax
...
```

time multiplexing




```
    ...
    call get_time
        // whatever get_time does
    movq %rax, %rbp
```

million cycle delay (from loop.exe's view)

```
    call get_time
        // whatever get_time does
    subq %rbp, %rax
    ...
```

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);
```

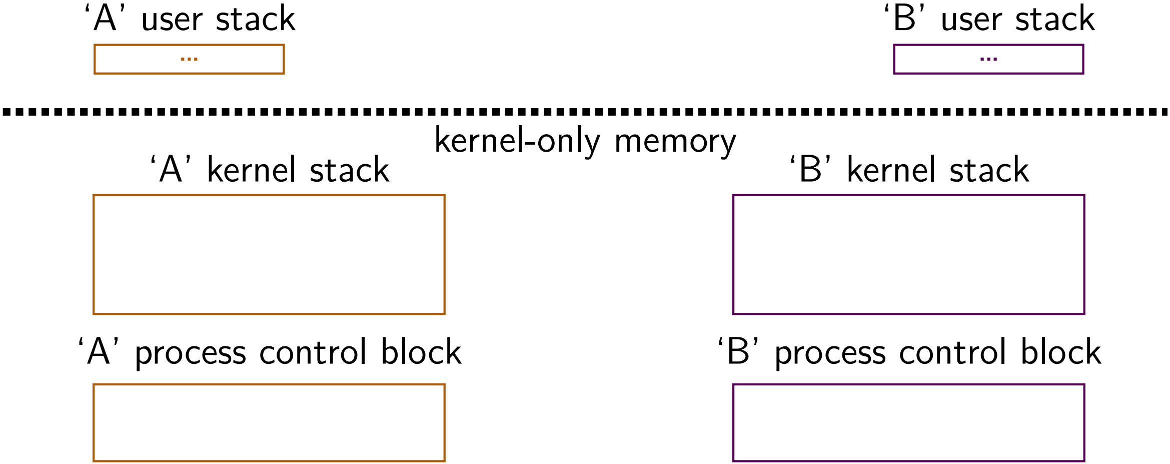
struct context

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

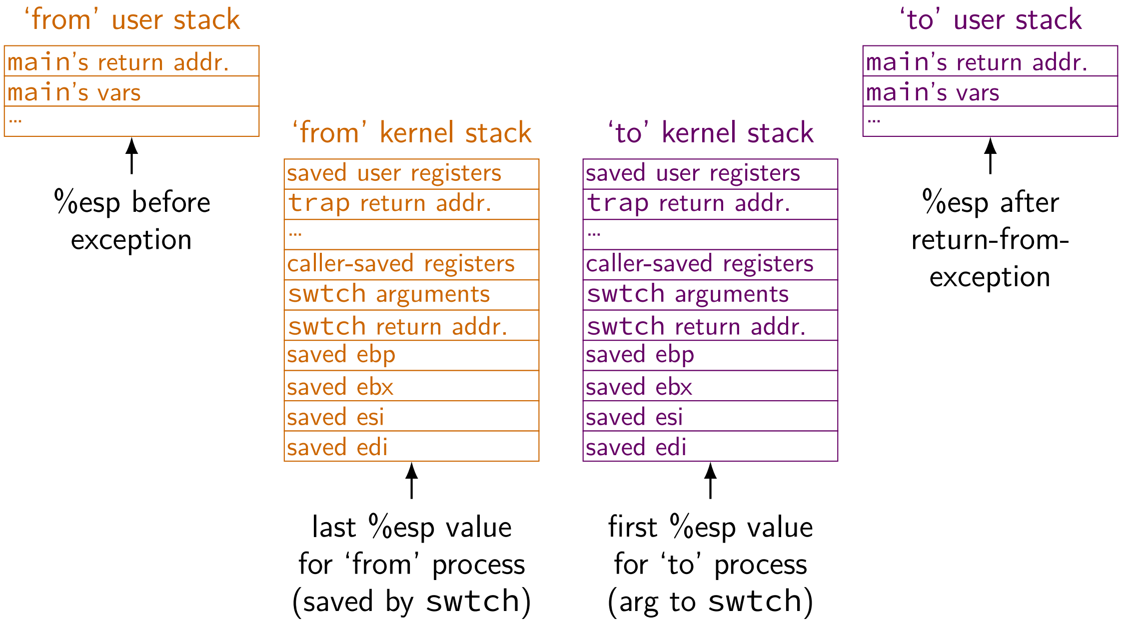
```
struct context {
  uint edi;            /* <-- top of stack of this thread */
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;          /* <-- return address of swtch() */
  /* not in struct but stored on stack thread after eip:
        arguments to current call to swtch
        caller-saved registers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

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

xv6: where the context is



xv6: where the context is (detail)



xv6: where the context is (detail)

'from' user stack

`main`'s return addr.
`main`'s vars
…

↑
%esp before exception

'from' kernel stack

saved user registers
`trap` return addr.
…
caller-saved registers
`swtch` arguments
`swtch` return addr.
saved ebp
saved ebx
saved esi
saved edi

↑
last %esp value for 'from' process (saved by `swtch`)

saved in 'from' `struct proc`

'to' kernel stack

saved user registers
`trap` return addr.
…
caller-saved registers
`swtch` arguments
`swtch` return addr.
saved ebp
saved ebx
saved esi
saved edi

↑
first %esp value for 'to' process (arg to `swtch`)

retrieved via 'to' `struct proc`

'to' user stack

`main`'s return addr.
`main`'s vars
…

↑
%esp after return-from-exception

xv6: where the context is (detail)