Changelog

Changes made in this version not seen in first lecture:

- 3 September 2019: xv6: where the context is: rename from/to into A/B to avoid overloading "to" and be consistent with the preceeding context switch picture
- 3 September 2019: xv6: where the context is: make user stacks boxes labelled on top to increase consistency
- 3 September 2019: xv6: where the context is: add animation frame identifying that the saved kernel stack pointers are what are passed to swtch()
- 3 September 2019: xv6: where the context is: begin diagram with build identifying what an address space is to hopefully make it clearer
- 4 September 2019: xv6: where the context is: mark where pointers point with arrows

system calls / context switches

last time

kernel versus user mode

exceptions (AKA traps AKA …): run OS when needed

- controlled mechanism for switching (system calls — type of exception)
- handling input
- keeping programs from running for too long

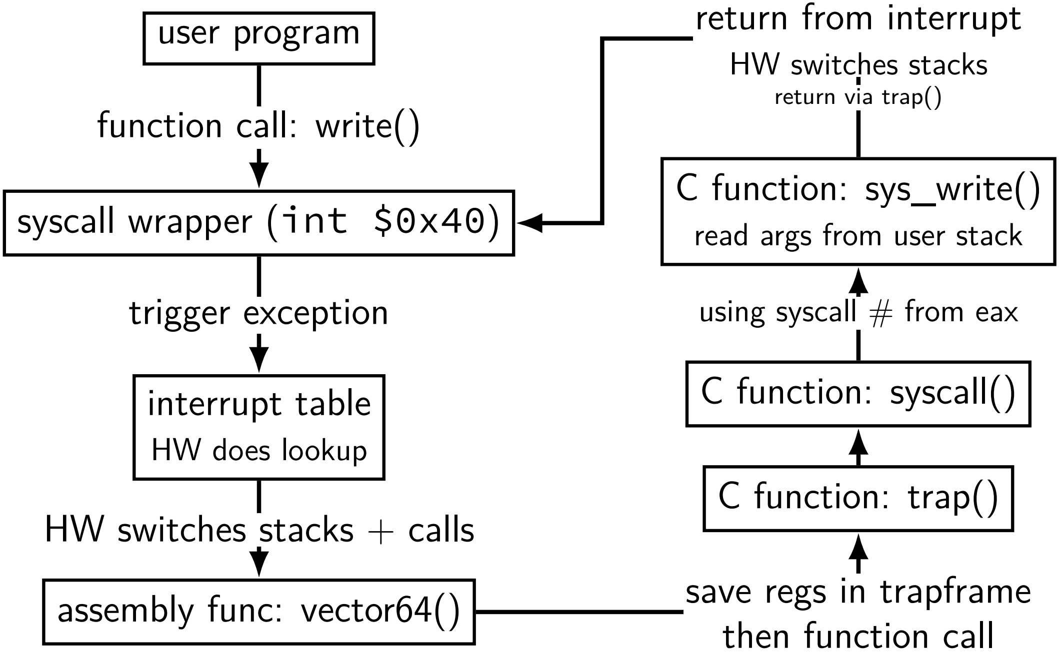
path of a system call in xv6

logistics

quiz demo

xv6 demo

write syscall in xv6



user program

function call: write()

syscall wrapper (`int $0x40`)

trigger exception

interrupt table

HW does lookup

HW switches stacks + calls

assembly func: vector64()

save regs in trapframe

then function call

C function: trap()

C function: syscall()

using syscall # from eax

C function: sys_write()

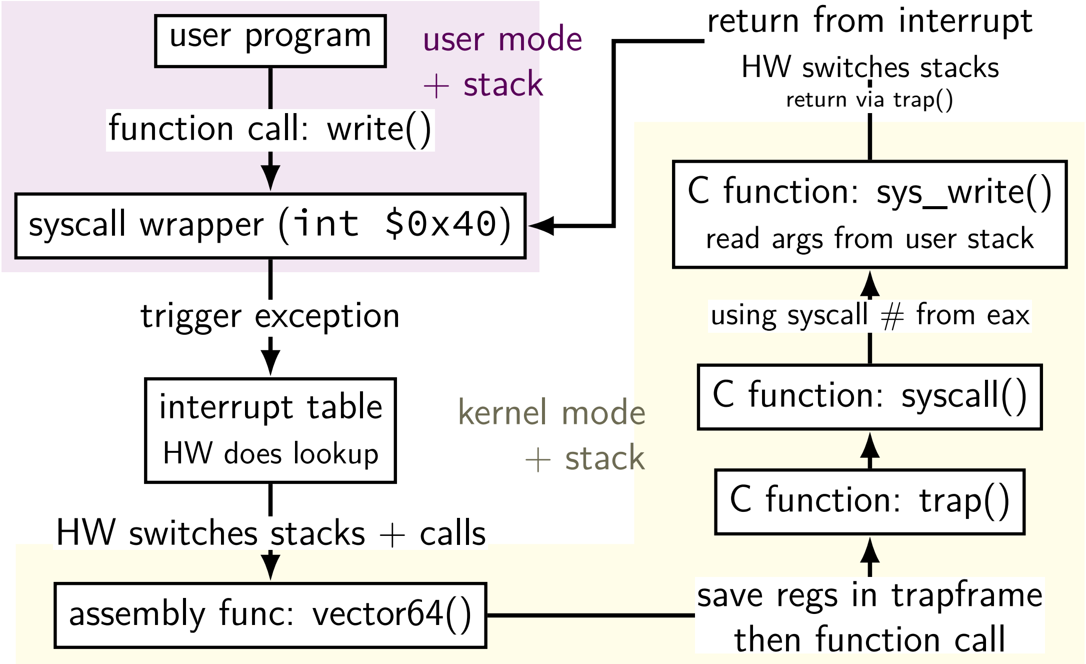
read args from user stack

return via trap()

HW switches stacks

return from interrupt

write syscall in xv6



user program

user mode
+ stack

function call: write()

syscall wrapper (`int $0x40`)

trigger exception

interrupt table
HW does lookup

kernel mode
+ stack

HW switches stacks + calls

assembly func: vector64()

save regs in trapframe
then function call

C function: trap()

C function: syscall()

using syscall # from eax

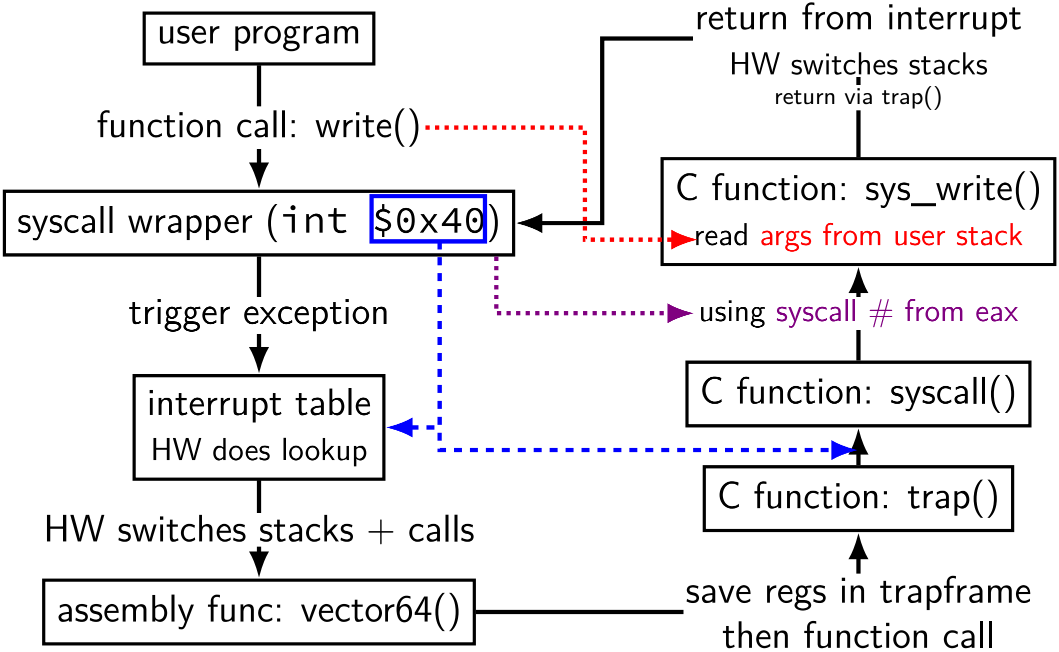
C function: sys_write()
read args from user stack

return via trap()

HW switches stacks

return from interrupt

write syscall in xv6



user program

function call: write()

syscall wrapper (`int $0x40`)

trigger exception

interrupt table

HW does lookup

HW switches stacks + calls

assembly func: vector64()

save regs in trapframe

then function call

C function: trap()

C function: syscall()

using syscall # from eax

C function: sys_write()

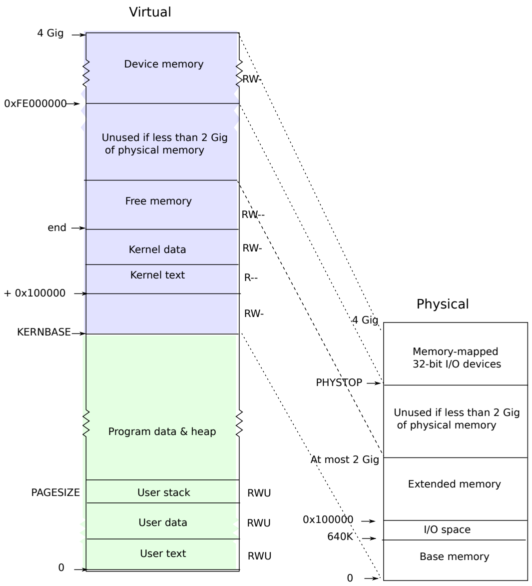
read args from user stack

return via trap()

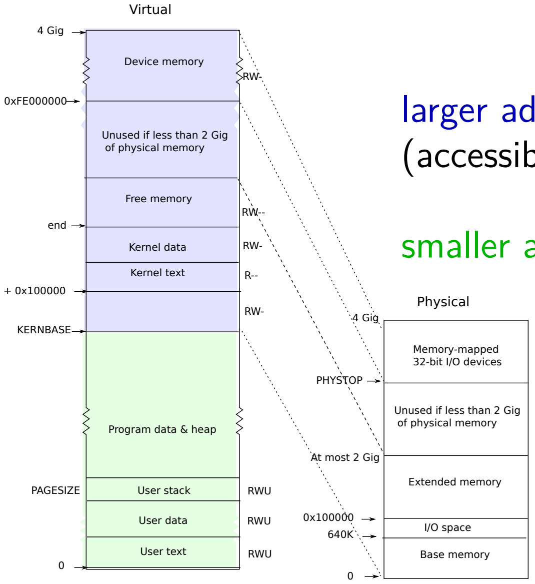
HW switches stacks

return from interrupt

xv6 memory layout



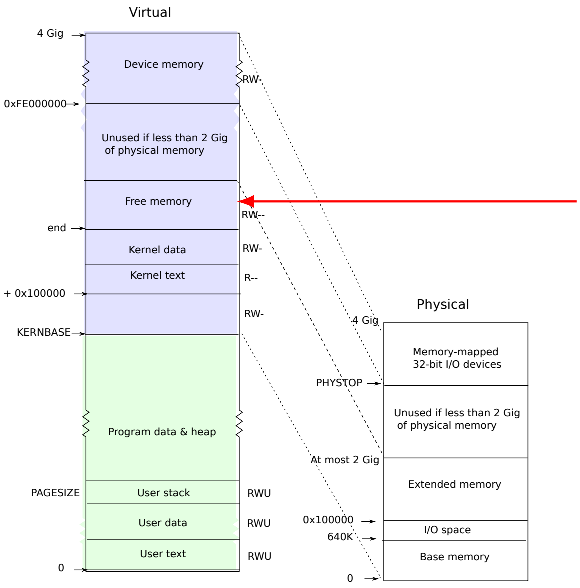
xv6 memory layout



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

smaller addresses are for applications

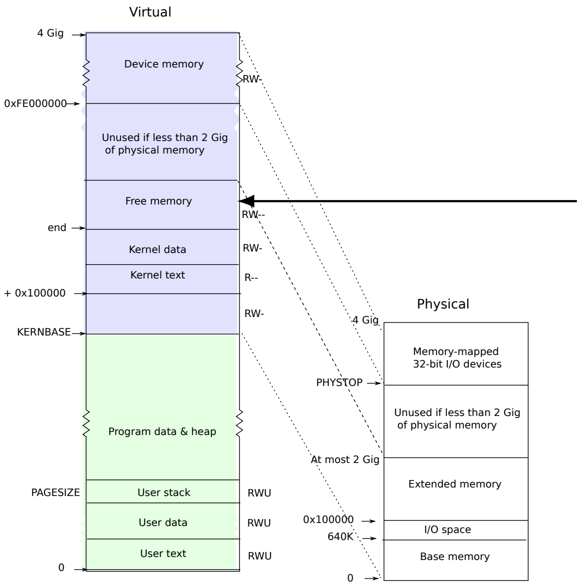
xv6 memory layout



kernel stack allocated here

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 process

change which one exceptions use as part of switching which processes is active on a processor

aside: nested exceptions

x86 switches to kernel stack on exception…

assuming it's switching to kernel mode

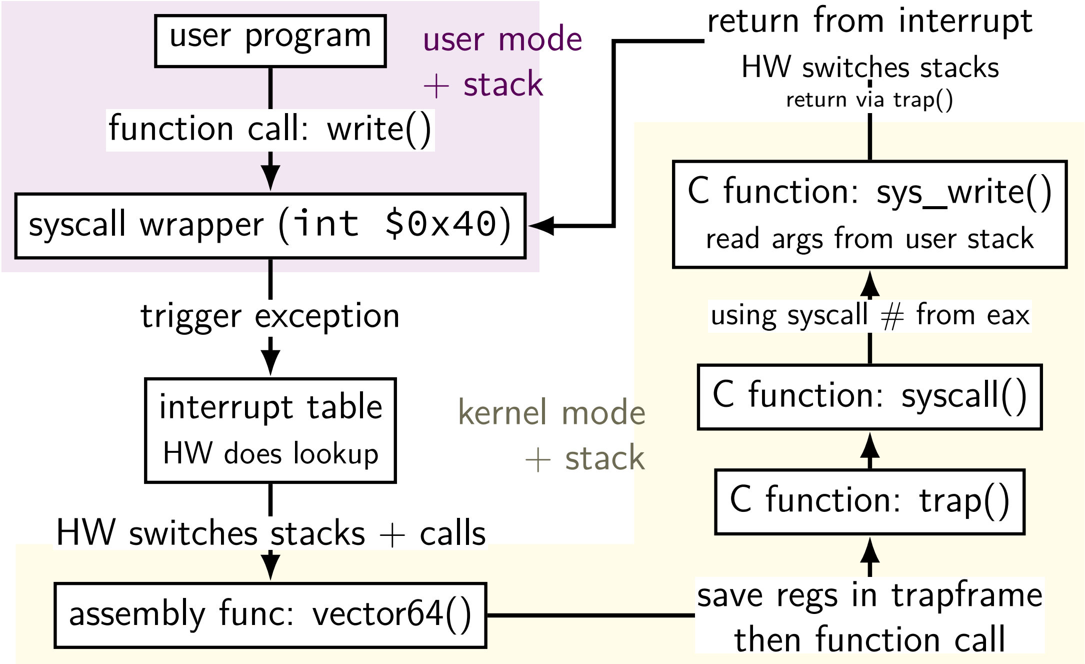
system call or timer interrupt in user mode

- start at top of kernel stack

timer interrupt during system call

- continue using current kernel stack

write syscall in xv6



user program

function call: write()

syscall wrapper (`int $0x40`)

user mode + stack

trigger exception

interrupt table
HW does lookup

HW switches stacks + calls

assembly func: vector64()

save regs in trapframe then function call

kernel mode + stack

C function: trap()

C function: syscall()

using syscall # from eax

C function: sys_write()
read args from user stack

return via trap()

HW switches stacks

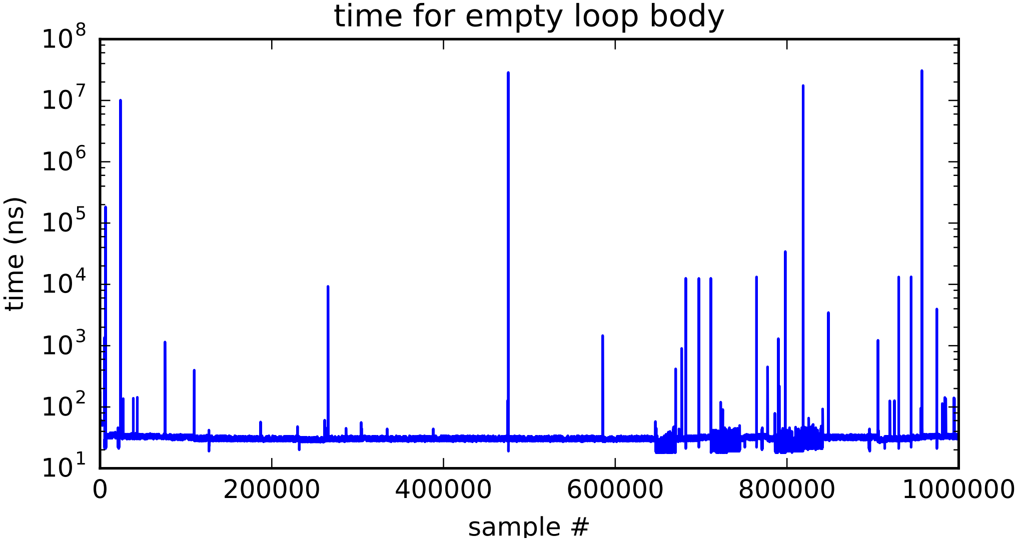
return from interrupt

timing nothing

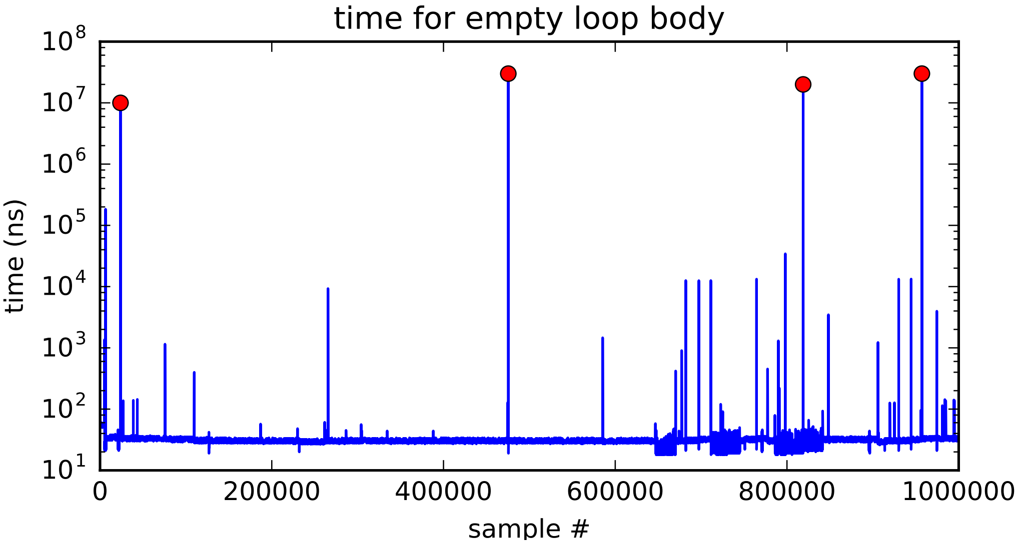
```
long times[NUM_TIMINGS];
int main(void) {
    for (int i = 0; i < N; ++i) {
        long start, end;
        start = get_time();
        /* do nothing */
        end = get_time();
        times[i] = end - start;
    }
    output_timings(times);
}
```

same instructions — same difference each time?

doing nothing on a busy system



doing nothing on a busy system



non-system call exceptions

xv6: there are traps other than system calls

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

- xv6’s action : kill the program

I/O — handle I/O

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);
...
```

`lidt` —

function (in x86.h) wrapping lidt instruction

sets the *interrupt descriptor table*

table of *handler functions* for each interrupt type

SETGATE() — set entry in that table

non-system call exceptions

xv6: there are traps other than system calls

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

- xv6’s action : kill the program

I/O — handle I/O

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() ==
      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();
  ...
}
```

non-system call exceptions

xv6: there are traps other than system calls

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
- xv6’s action : kill the program

I/O — handle I/O

xv6: faults

```
void
trap(struct tra
{
  ...
  switch(tf->trapno) {
  ...
  default:
    ...
    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;
  }
}
```

unknown exception

print message and kill running program

assume it screwed up

xv6: faults

```
void
trap(struct trapframe *tf)
{
  ...
  switch(tf->trapno) {
  ...
  default:
    ...
    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`

non-system call exceptions

xv6: there are traps other than system calls

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

- xv6’s action : kill the program

I/O — handle I/O

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)

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

CPU:

`loop.exe`

`loop.exe`

time

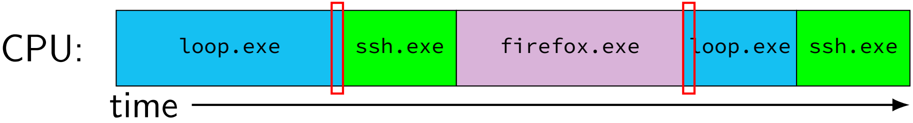
time multiplexing

```
...
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
```

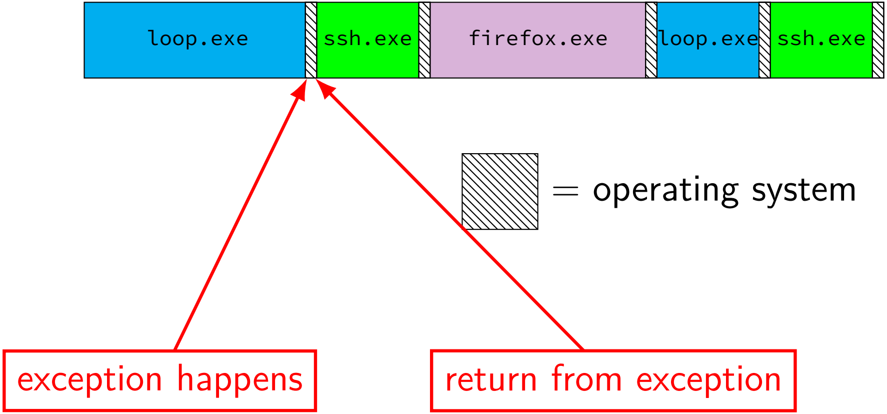
million cycle delay (from loop.exe's view)

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

time multiplexing really

▨ = operating system

time multiplexing really



loop.exe

ssh.exe

firefox.exe

loop.exe

ssh.exe

▨ = operating system

exception happens

return from exception

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

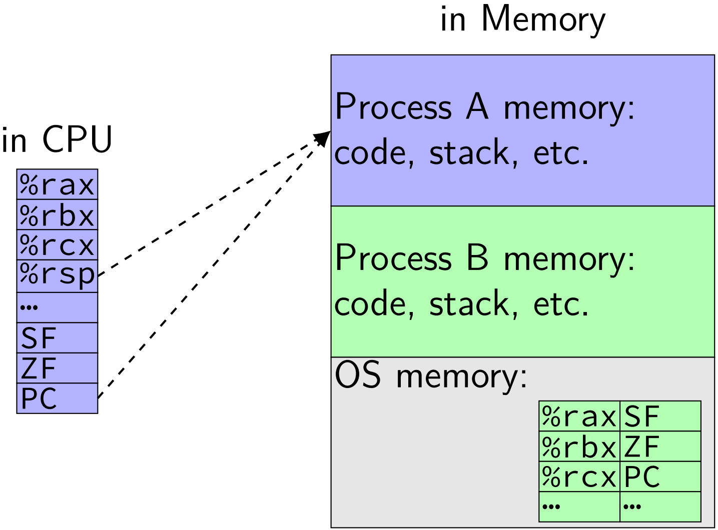
  `%rax` `%rbx`, …, `%rsp`, …

condition codes

program counter

address space = page table base pointer

contexts (A running)



in CPU

%rax
%rbx
%rcx
%rsp
...
SF
ZF
PC

in Memory

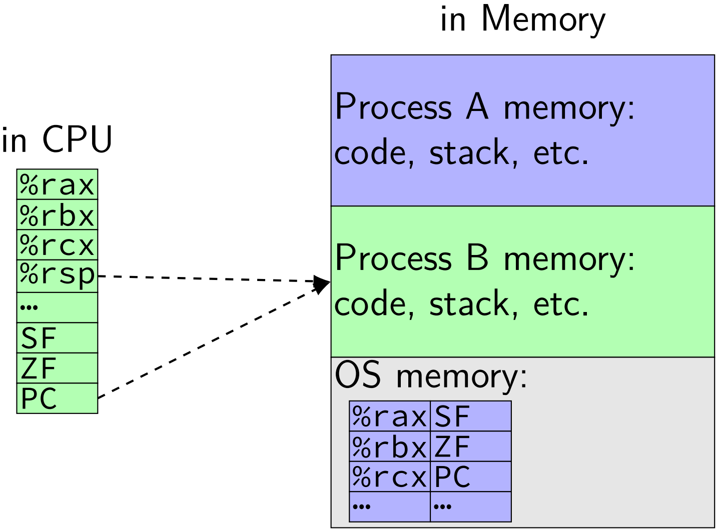
Process A memory:
code, stack, etc.

Process B memory:
code, stack, etc.

OS memory:

%rax	SF
%rbx	ZF
%rcx	PC
...	...

contexts (B running)



in CPU

%rax
%rbx
%rcx
%rsp
…
SF
ZF
PC

in Memory

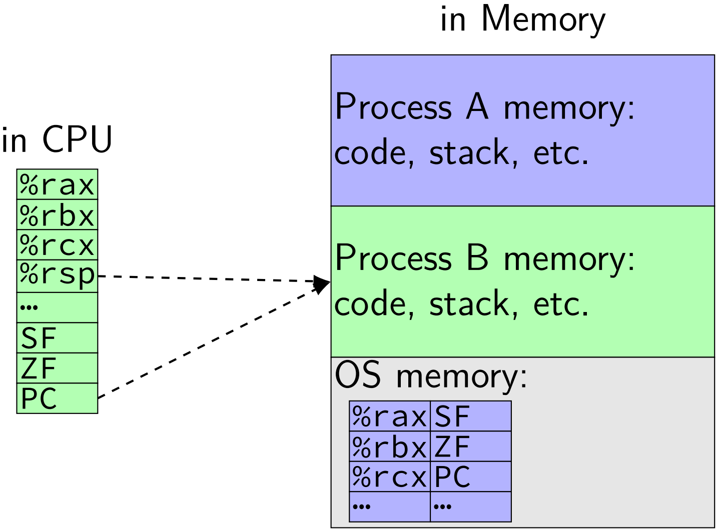
Process A memory:
code, stack, etc.

Process B memory:
code, stack, etc.

OS memory:

%rax	SF
%rbx	ZF
%rcx	PC
…	…

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**

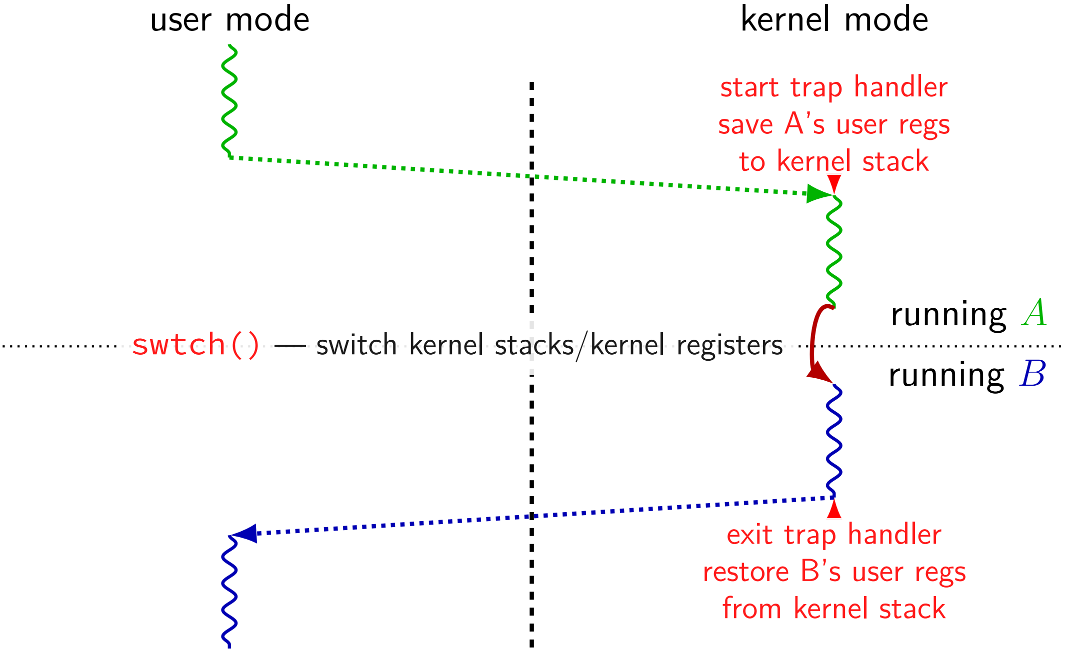
B makes system call to read from file

(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



user mode

kernel mode

start trap handler
save A's user regs
to kernel stack

running A

`swtch()` — switch kernel stacks/kernel registers

running B

exit trap handler
restore B's user regs
from kernel stack

context switch in xv6

will mostly talk about *kernel thread switch*:

xv6 function: `swtch()`

save kernel registers for A, restore for B

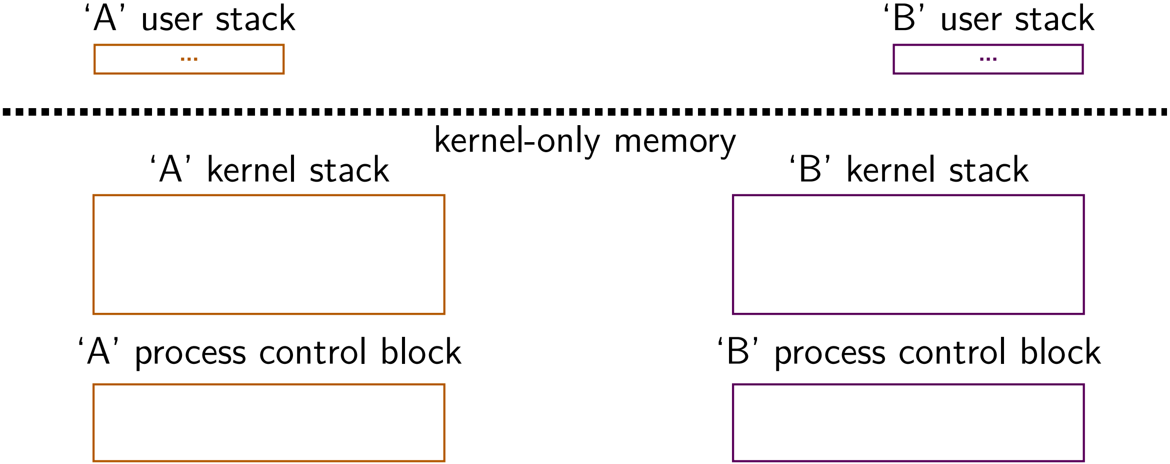
in xv6: *separate from saving/restoring user registers*

- one of many possible OS design choices

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

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

xv6: where the context is



‘A’ user stack

…

‘B’ user stack

…

kernel-only memory

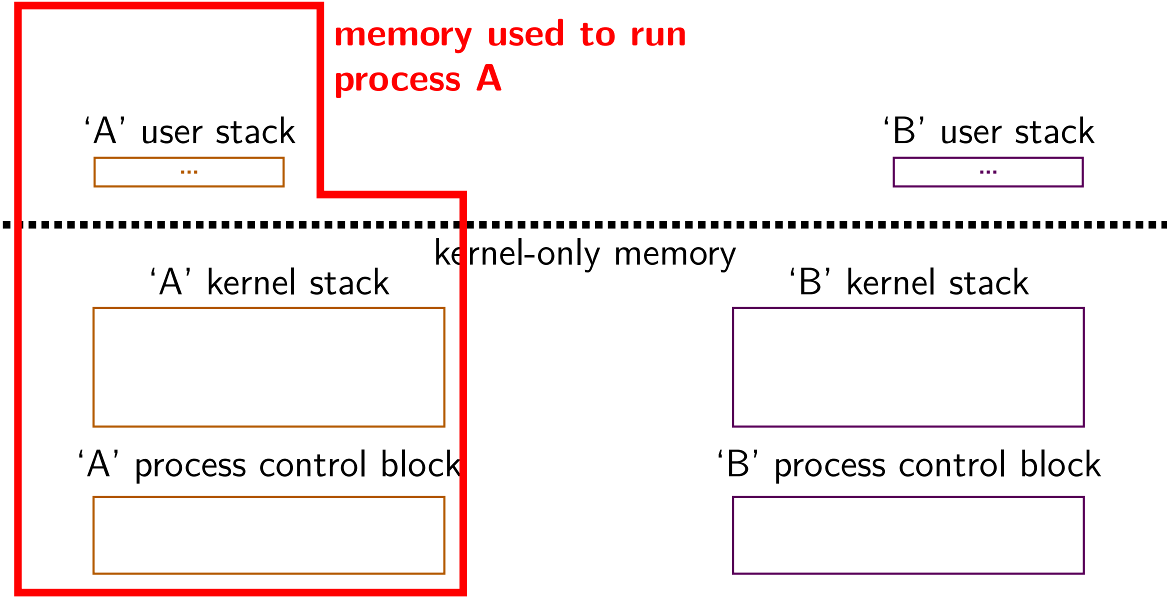
‘A’ kernel stack

‘B’ kernel stack

‘A’ process control block

‘B’ process control block

xv6: where the context is



memory used to run process A

‘A’ user stack

...

‘B’ user stack

...

kernel-only memory

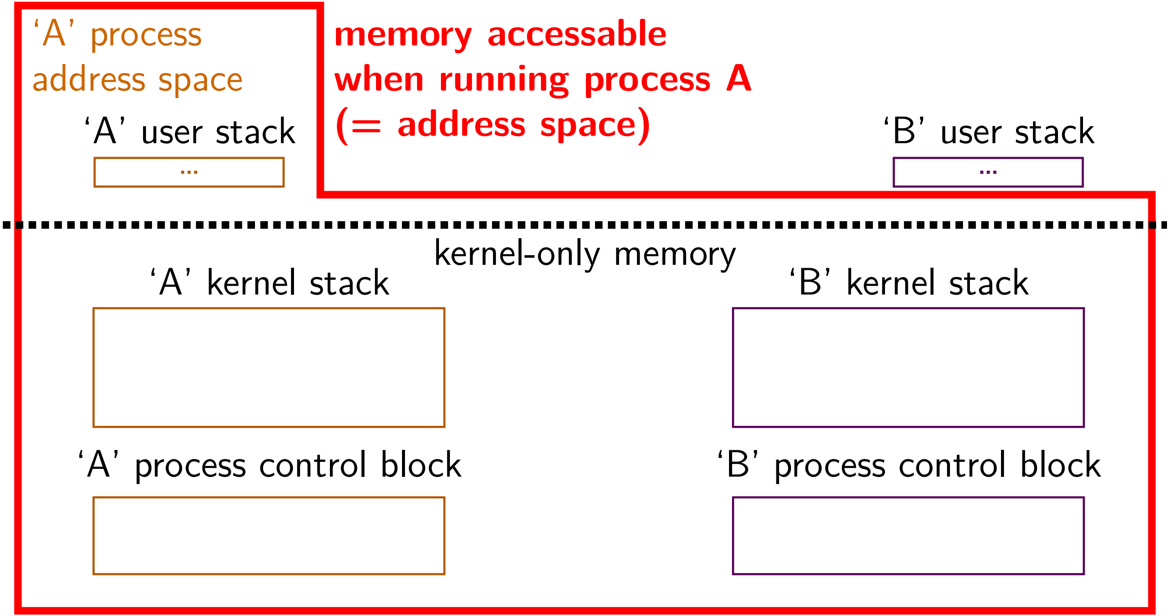
‘A’ kernel stack

‘B’ kernel stack

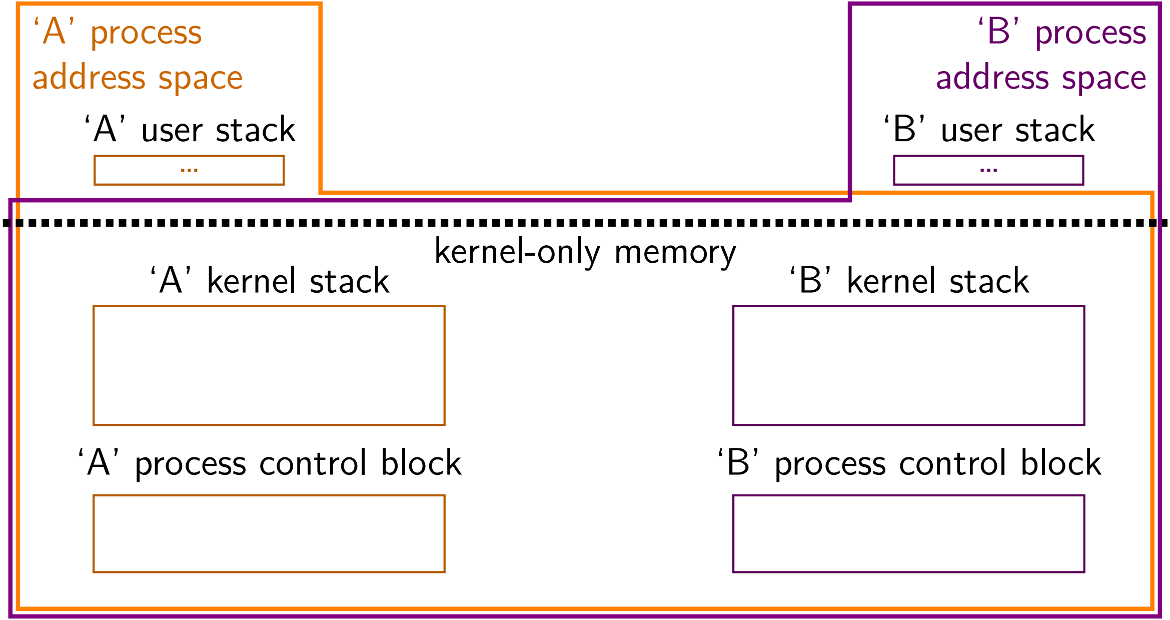
‘A’ process control block

‘B’ process control block

xv6: where the context is



xv6: where the context is



‘A’ process address space

‘A’ user stack

…

‘B’ process address space

‘B’ user stack

…

kernel-only memory

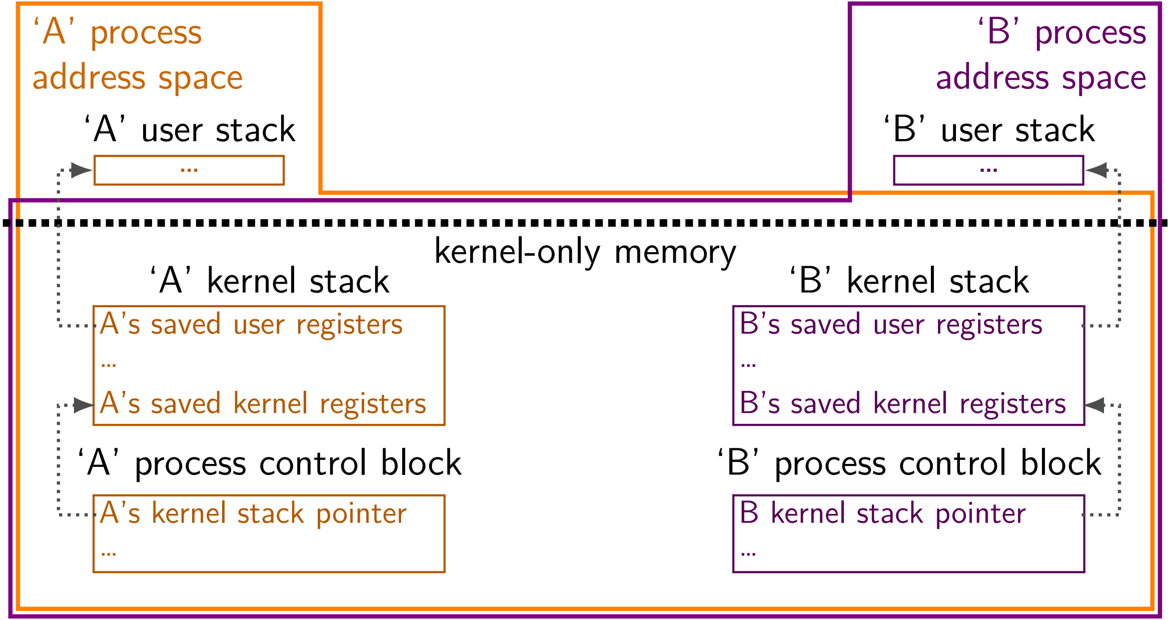
‘A’ kernel stack

‘B’ kernel stack

‘A’ process control block

‘B’ process control block

xv6: where the context is



‘A’ process address space

‘A’ user stack

…

‘B’ process address space

‘B’ user stack

…

kernel-only memory

‘A’ kernel stack

A’s saved user registers

…

A’s saved kernel registers

‘A’ process control block

A’s kernel stack pointer

…

‘B’ kernel stack

B’s saved user registers

…

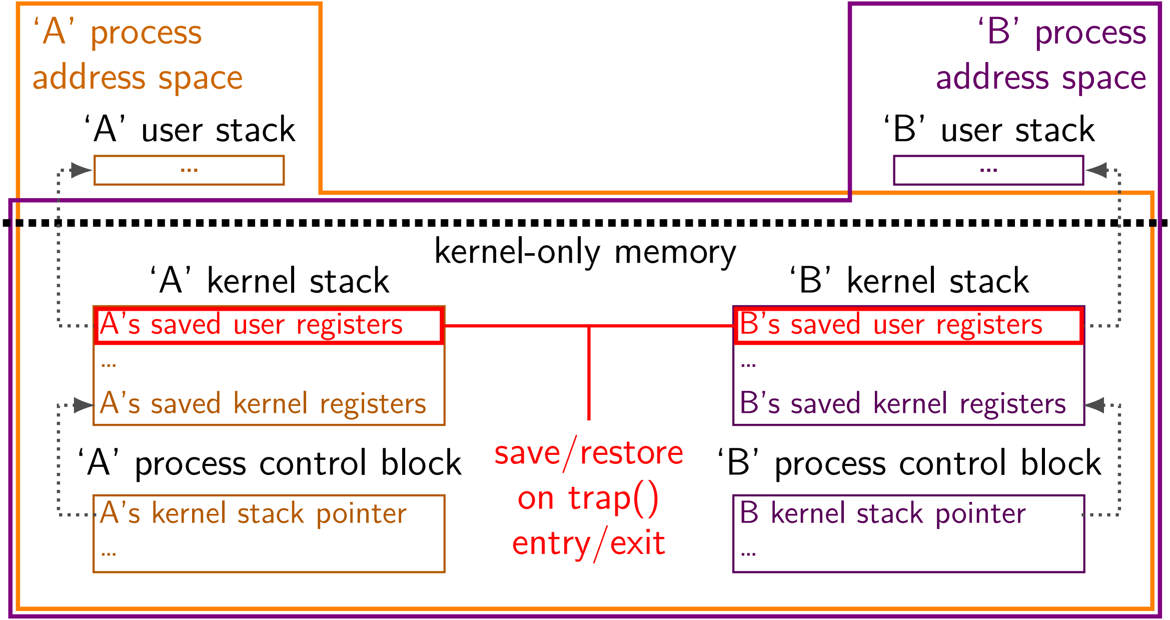
B’s saved kernel registers

‘B’ process control block

B kernel stack pointer

…

xv6: where the context is



‘A’ process address space

‘A’ user stack

...

‘B’ process address space

‘B’ user stack

...

kernel-only memory

‘A’ kernel stack

A’s saved user registers

...

A’s saved kernel registers

‘B’ kernel stack

B’s saved user registers

...

B’s saved kernel registers

save/restore on trap() entry/exit

‘A’ process control block

A’s kernel stack pointer

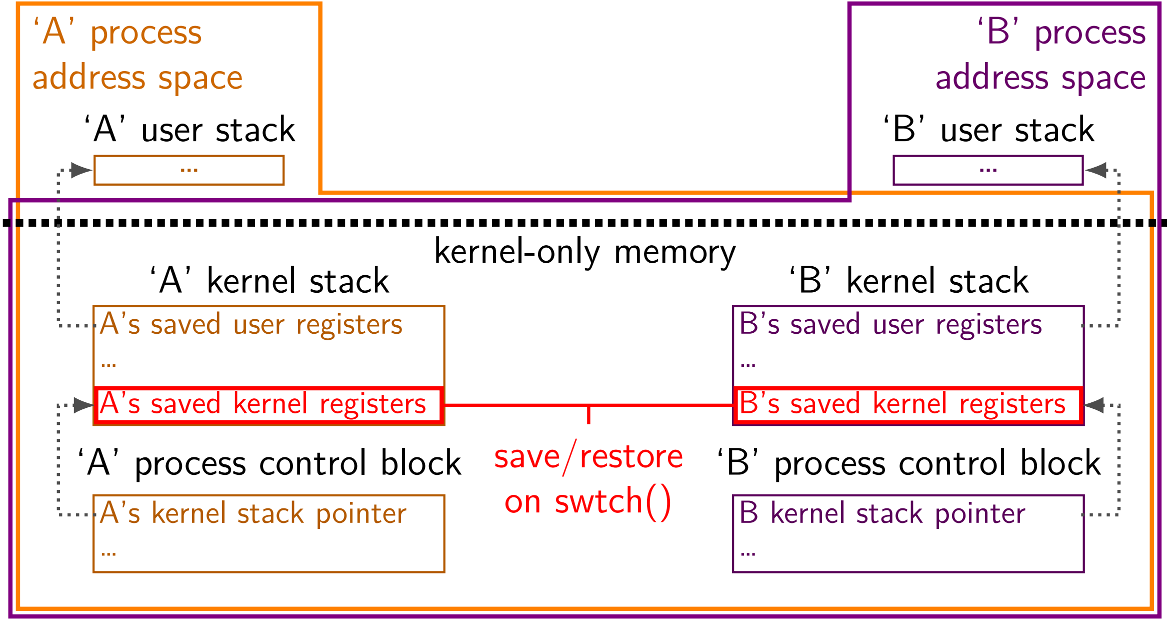
...

‘B’ process control block

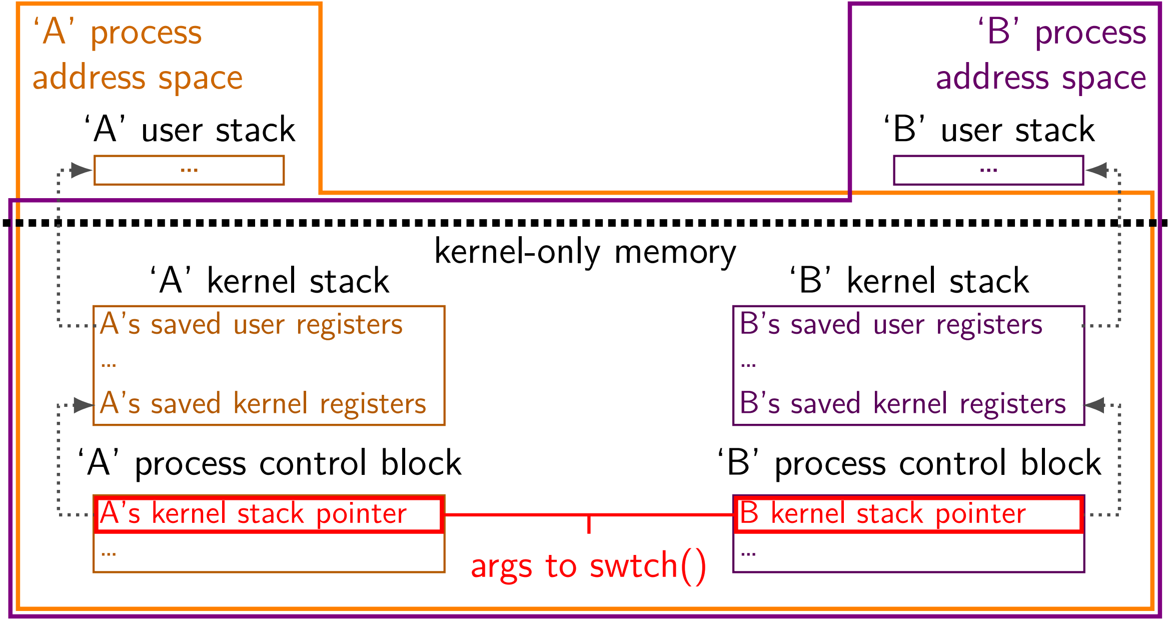
B kernel stack pointer

...

xv6: where the context is



xv6: where the context is



'A' process address space

'B' process address space

'A' user stack

…

'B' user stack

…

kernel-only memory

'A' kernel stack

A's saved user registers

…

A's saved kernel registers

'B' kernel stack

B's saved user registers

…

B's saved kernel registers

'A' process control block

A's kernel stack pointer

…

'B' process control block

B kernel stack pointer

…

args to swtch()

thread switching

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
}
```

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

thread switching

structure to save context in
yes, it looks like we're missing
some registers we need...

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
}
```

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

thread switching

eip = saved program counter

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
}
```

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

thread switching

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;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
}
```

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

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: 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
```

saved: 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
```

what about other parts of context?

eax, ecx, …: saved by swtch's caller

esp: same as address of context

program counter: set 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 new context

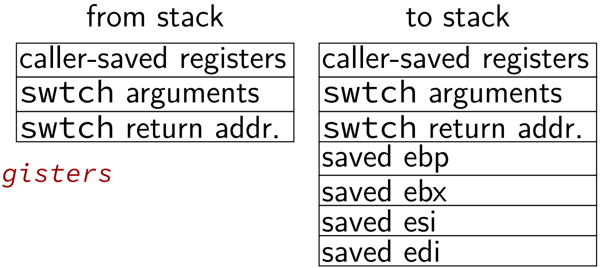
juggling stacks

```
.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

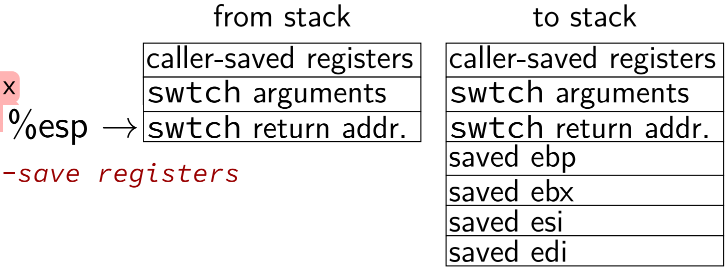
juggling stacks

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

  # 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 → swtch return addr.

to stack

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

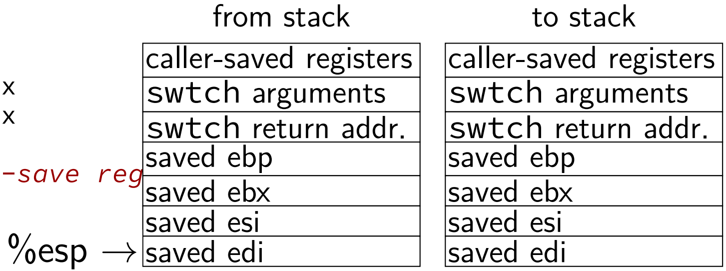
juggling stacks

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

  # Save old callee-save 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

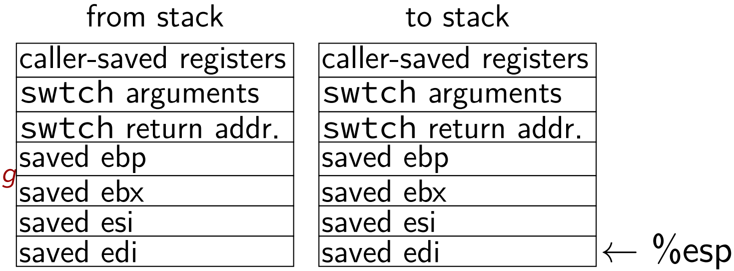
juggling stacks

```
.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

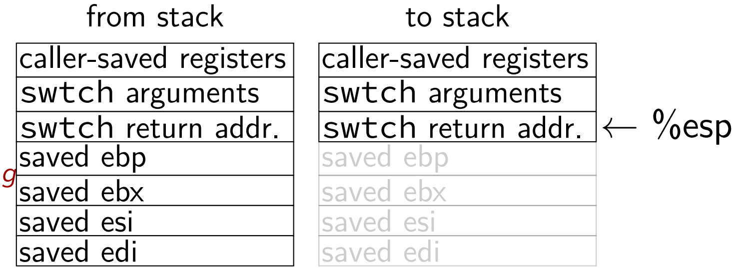
juggling stacks

```
.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
```



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

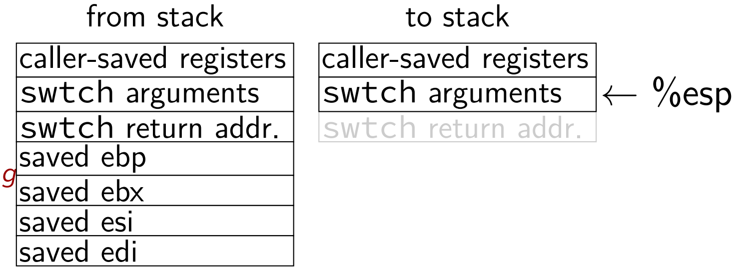
juggling stacks

```
.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

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

to stack

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

juggling stacks

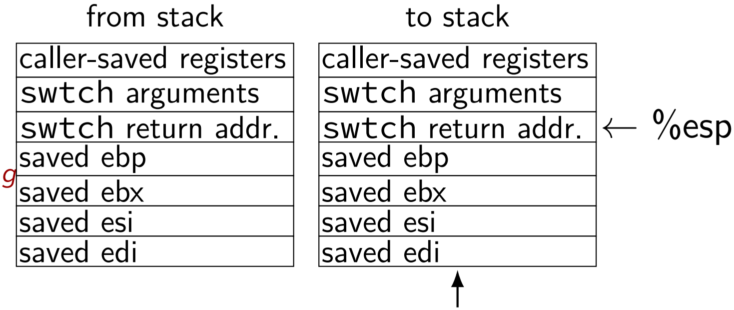
```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # 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	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

bottom of new kernel stack

kernel-space context switch summary

swtch function

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

initial setup — manually construct stack values

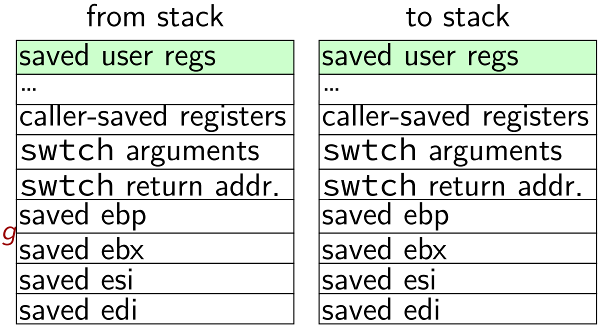
juggling stacks

```
.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

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

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

initial user registers created manually on stack

- (as if saved by system call)

the userspace part?

user registers stored in ‘trapframe’ struct

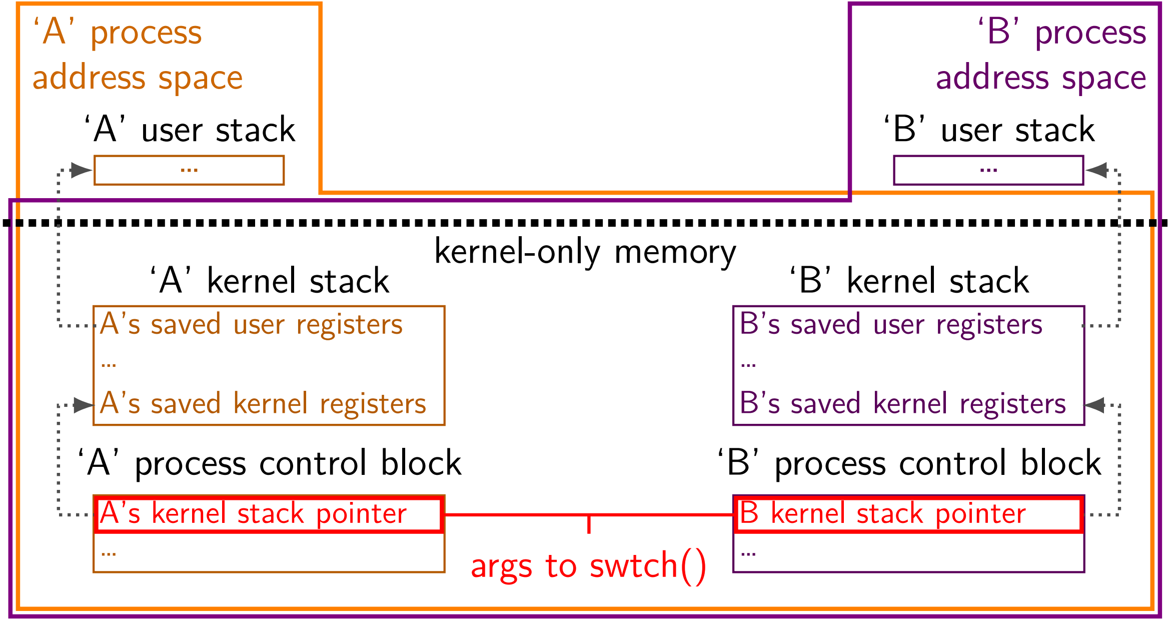
- created on kernel stack when interrupt/trap happens
- restored before using `iret` to switch to user mode

initial user registers created manually on stack

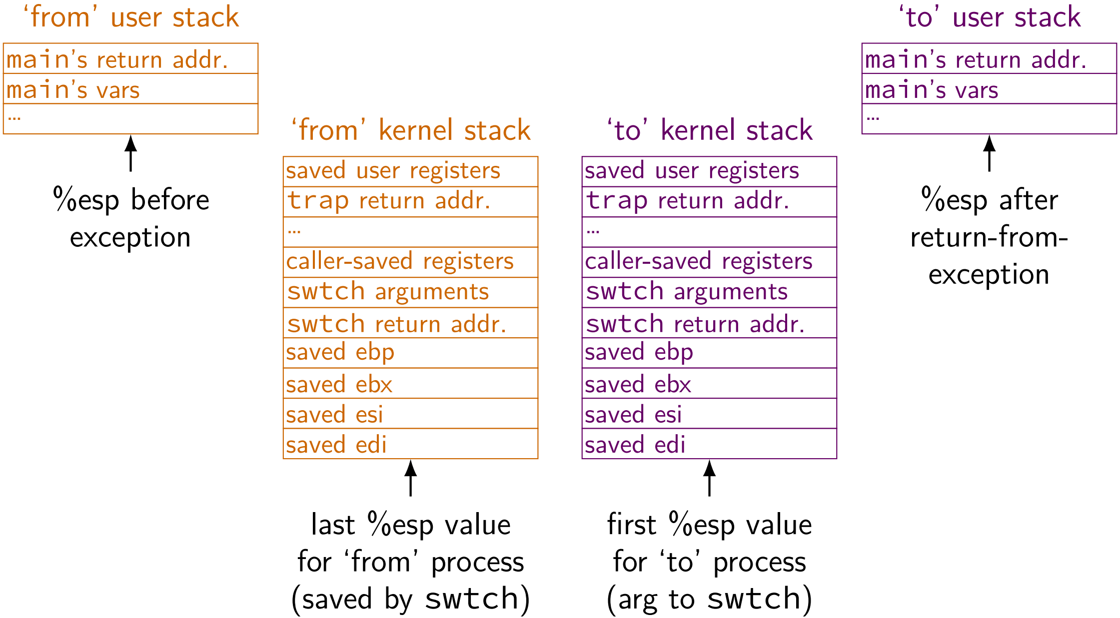
- (as if saved by system call)

other code (not shown) handles setting address space

xv6: where the context is



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`)

‘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

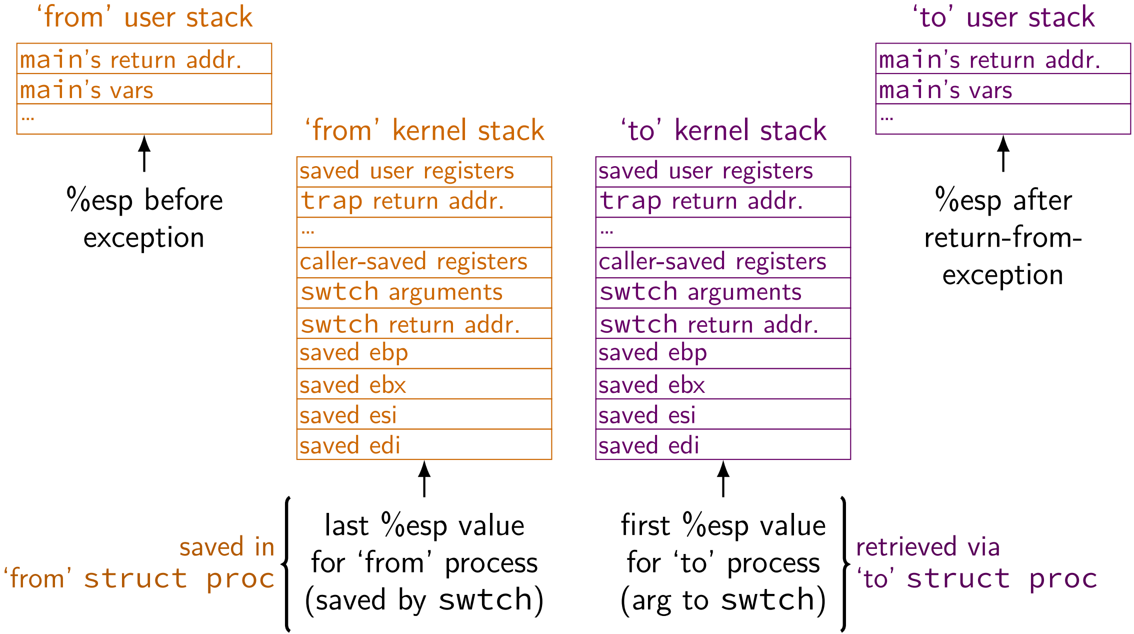
first %esp value for ‘to’ process (arg to `swtch`)

‘to’ user stack

- `main`’s return addr.
- `main`’s vars
- …

%esp after return-from-exception

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`)

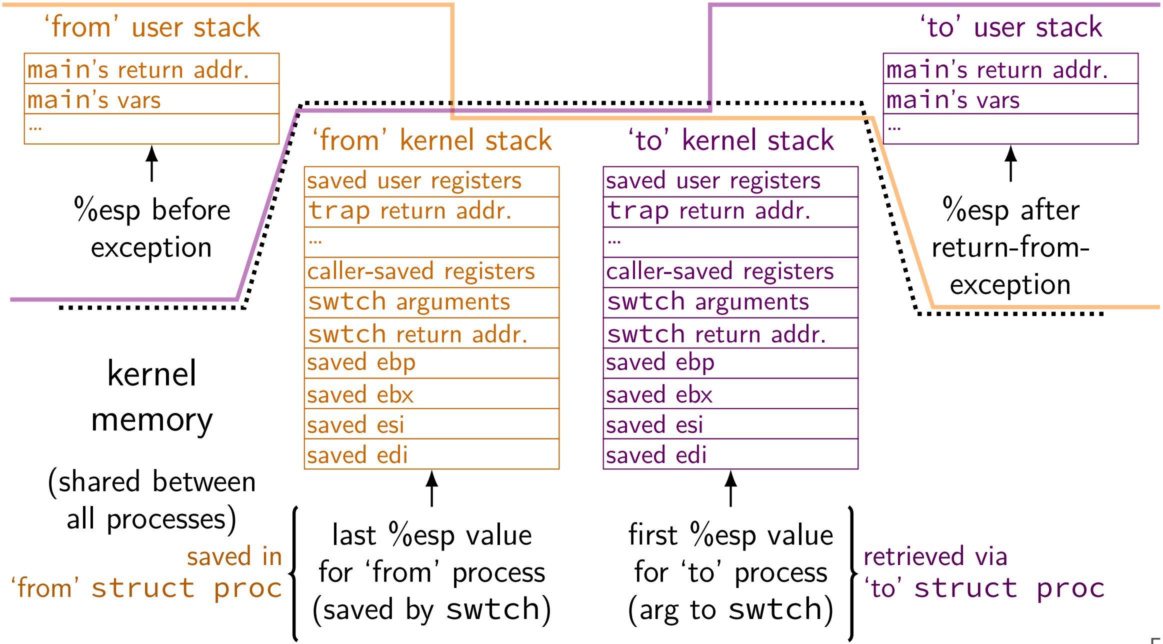
retrieved via ‘to’ `struct proc`

‘to’ user stack

- `main`’s return addr.
- `main`’s vars
- …

%esp after return-from-exception

xv6: where the context is (detail)



‘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

‘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

‘to’ user stack

main’s return addr.

main’s vars

…

%esp after return-from-exception

kernel memory

(shared between all processes)

saved in ‘from’ `struct proc`

last %esp value for ‘from’ process (saved by `swtch`)

first %esp value for ‘to’ process (arg to `swtch`)

retrieved via ‘to’ `struct proc`

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

backup slides

backup slides

write syscall in xv6: summary

`write` function — syscall wrapper uses `int $0x40`

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 $0x40`

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 $0x40`

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

xv6intro homework

get familiar with xv6 OS

add a new system call: `writecount()`

returns total number of times write call happened

homework steps

system call implementation: sys_writecount

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

add writecount to several tables/lists

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

create a userspace program that calls writecount

- recommendation: copy from given programs

note on locks

some existing code uses acquire/release

you do not have to do this

only for multiprocessor support

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

syscalls in xv6

fork, exec, exit, wait, kill, getpid — process control

open, read, write, close, fstat, dup — file operations

mknod, unlink, link, chdir — directory operations

…

write syscall in xv6: user mode

syscall.h

```
...
#define SYS_write   16
...
```

main.c

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

usys.S

(after macro replacement)

```
#include "syscall.h"
// ...
.globl write
write:
    /* 16 = SYS_write */
    movl $16, %eax
    /* 0x40 = T_SYSCALL */
    int $0x40
    ret
```

write syscall in xv6: user mode

syscall.h

```
...
#define SYS_write   16
...
```

main.c

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

usys.S

(after macro replacement)

```
#include "syscall.h"
// ...
.globl write
write:
      /* 16 = SYS_write */
      movl $16, %eax
      /* 0x40 = T_SYSCALL */
      int $0x40
      ret
```

`int`errupt — trigger an exception similar to a keypress

parameter (`0x40` in this case) — type of exception

write syscall in xv6: user mode

syscall.h

```
...
#define SYS_write   16
...
```

main.c

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

usys.S

(after macro replacement)

```
#include "syscall.h"
// ...
.globl write
write:
    /* 16 = SYS_write */
    movl $16, %eax
    /* 0x40 = T_SYSCALL */
    int $0x40
    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*

table of *handler functions* for each interrupt type

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);
...
```

set the `T_SYSCALL` (= `0x40`) 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 handling can interrupt slow syscall
con: makes writing system calls safely more complicated
pro: slow system calls don't stop timers, keypresses, etc. from working

xv6 choice: interrupts *are* disabled during non-syscall exception handling
(e.g. don't worry about keypress being handled while timer being handlec

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

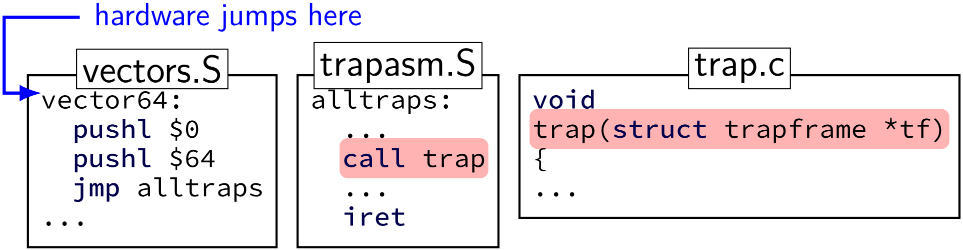
set to pointer to assembly function `vector64`

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`



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

```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
...
```

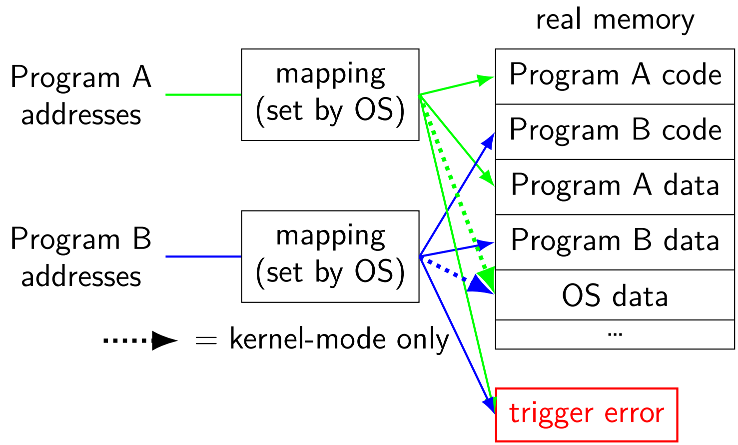
trapasm.S

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

trap.c

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

recall: address translation



real memory

Program A addresses

Program B addresses

mapping (set by OS)

mapping (set by OS)

Program A code

Program B code

Program A data

Program B data

OS data

...

trigger error

·····► = kernel-mode only