system calls / context switches

changelog

Notable changes since first lecture:

- 25 Jan 2022: main write syscall diagram: add arrow for %eax
- 25 Jan 2022: write syscall and layers: remove stray arrow
- 26 Jan 2022: edit context slides to use %eXX instead of %rXX registers

last time

logistics — (quiz due next week)

OS definition ambiguity

OS roles: referee, illusionist, glue

the process virtual machine
- thread $\sim$ processor
- address space $\sim$ memory
- files $\sim$ devices

basic hardware support for OSes:
- kernel versus user mode
- address translation
- exceptions: hardware jumps to OS on certain events

[12pm] what xv6 includes / needed to run

aside: exception versus mode switch

mode switch: go from user to kernel mode or vice-versa

exception: hardware triggers OS function ("handler") to run

- will switch from user to kernel mode (if not already)
- finishing exception handler usually requires kernel to user mode switch

xv6

we will be using an teaching OS called "xv6"

- several (not all) programming assignments

based on Sixth Edition Unix

modified to be multicore and use 32-bit x86 (not PDP-11)

- (there's also a (more recent) RISC V version, but we cover x86 in CS 2150...)

xv6 setup/assignment

first assignment — adding two simple xv6 system calls

includes xv6 download instructions

and link to xv6 book

xv6 technical requirements

you will need a Linux environment

- we will supply one (VM on website), or get your own

some non-Linux environments have worked well for students, but we are limited in how much tech support we can do for them

- the Windows Subsystem for Linux
- OS X natively (needs a cross-compiler)

...with qemu installed

- qemu (for us) = emulator for 32-bit x86 system
- Ubuntu/Debian package qemu-system-i386

first assignment

released a week from Friday; due the following week

get compiled and xv6 working

...toolkit uses an emulator

- could run on real hardware or a standard VM, but a lot of details
- also, emulator lets you use GDB

xv6: what's included

Unix-like kernel

- very small set of syscalls
- some less featureful (e.g. exit without exit status)

userspace library

- very limited

userspace programs

- command line, ls, mkdir, echo, cat, etc.
- some self-testing programs

xv6: echo.c

```
#include "types.h"
#include "stat.h"
#include "user.h"

int
main(int argc, char *argv[])
{
  int i;

  for(i = 1; i < argc; i++)
    printf(1, "%s%s", argv[i], i+1 < argc ? " " : "\n");
  exit();
}
```

xv6: echo.c

```
#include "types.h"
#include "stat.h"
#include "user.h"

int
main(int argc, char *argv[])
{
  int i;

  for(i = 1; i < argc; i++)
    printf(1, "%s%s", argv[i], i+1 < argc ? " " : "\n");
  exit();
}
```

xv6: echo.c

```
#include "types.h"
#include "stat.h"
#include "user.h"

int
main(int argc, char *argv[])
{
  int i;

  for(i = 1; i < argc; i++)
    printf(1, "%s%s", argv[i], i+1 < argc ? " " : "\n");
  exit();
}
```

xv6 demo

hello.c

```c
#include "user.h"
int main(void) {
  write(1, "Hello, World!", 13);
  exit();
}
```

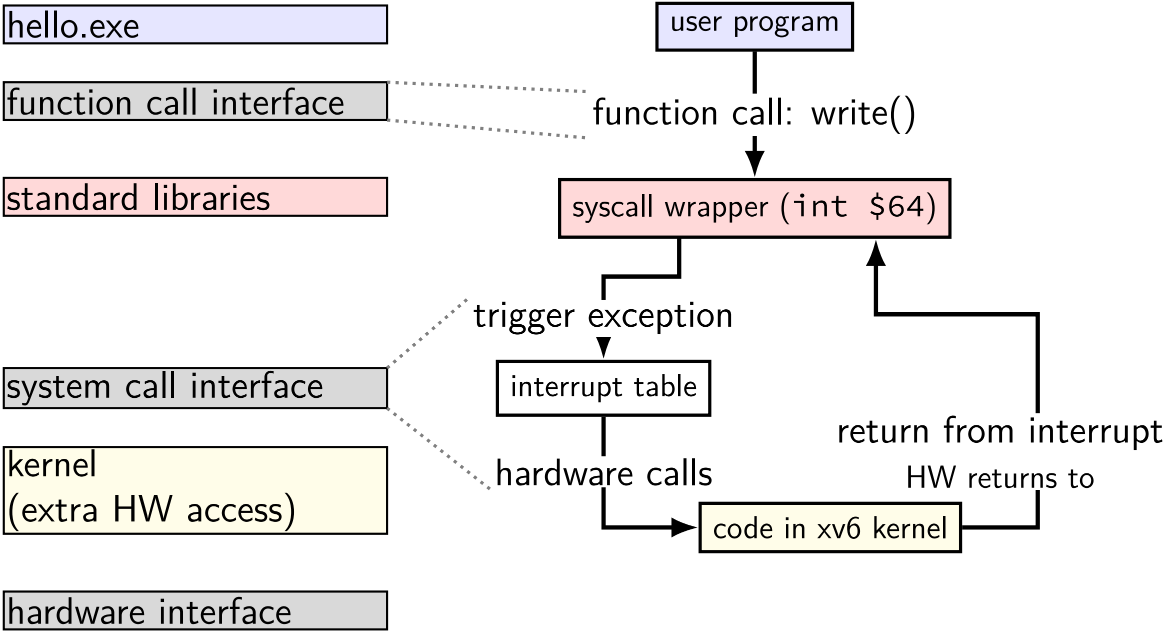
hello.exe
function call interface
standard libraries
system call interface
kernel (extra HW access)
hardware interface

hello.c

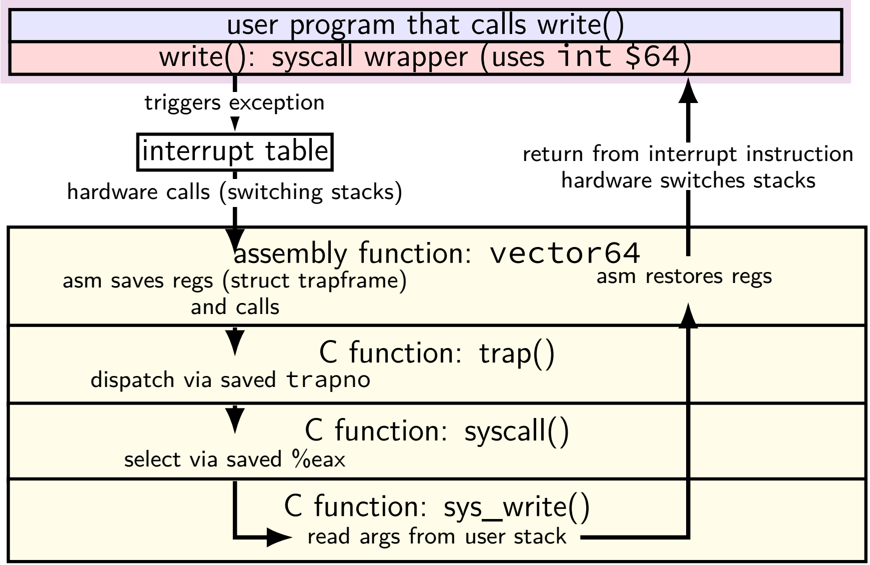
```c
#include "user.h"
int main(void) {
  write(1, "Hello, World!", 13);
  exit();
}
```

hello.exe
function call interface
standard libraries
system call interface
kernel (extra HW access)
hardware interface

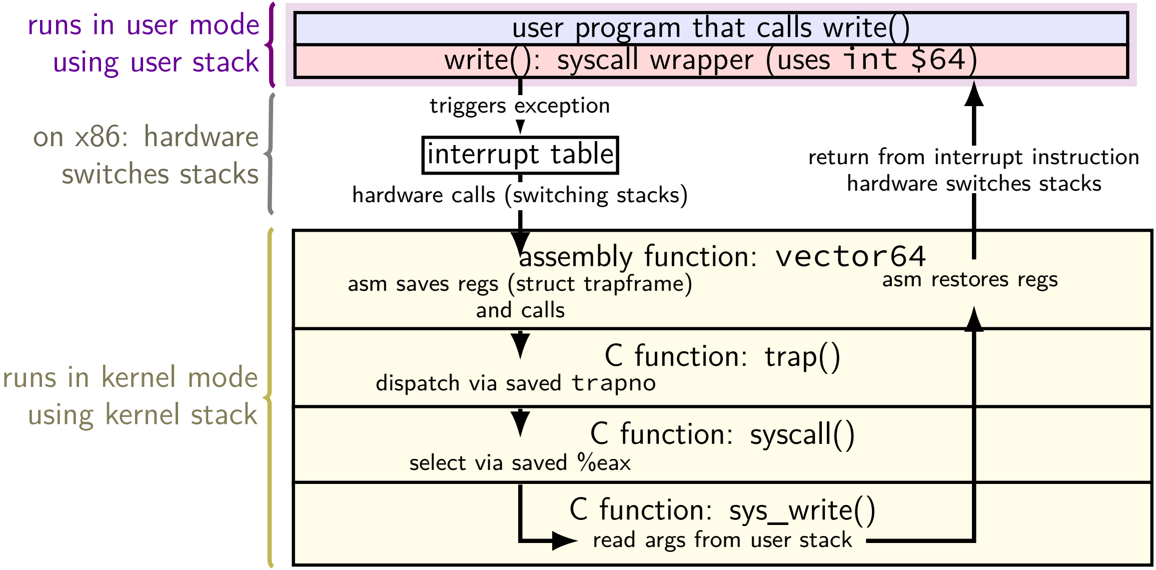
write syscall and layers



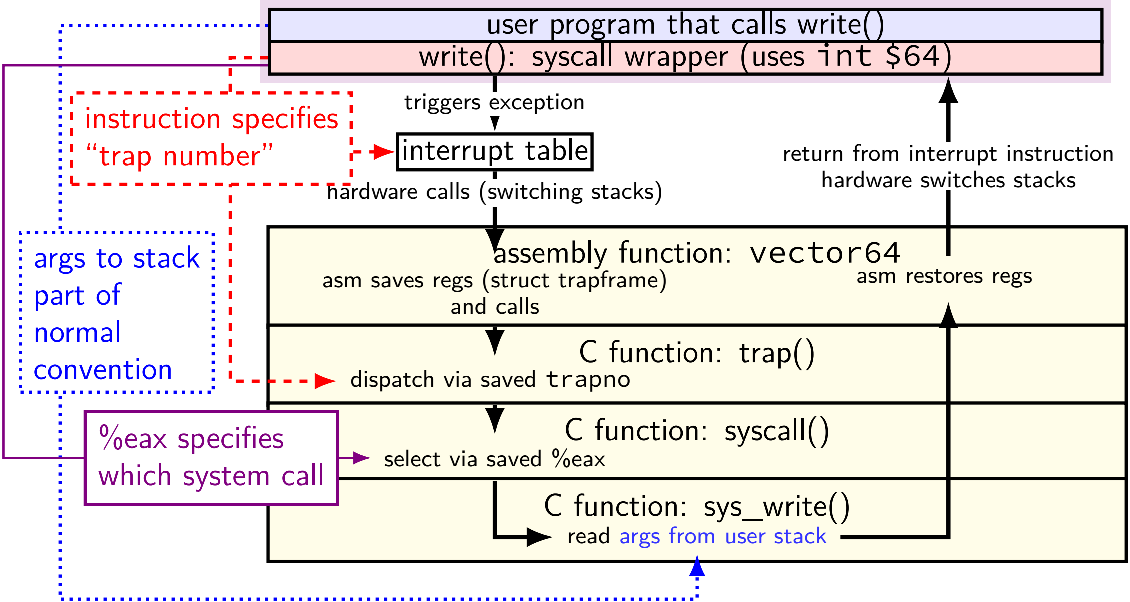
write syscall and xv6



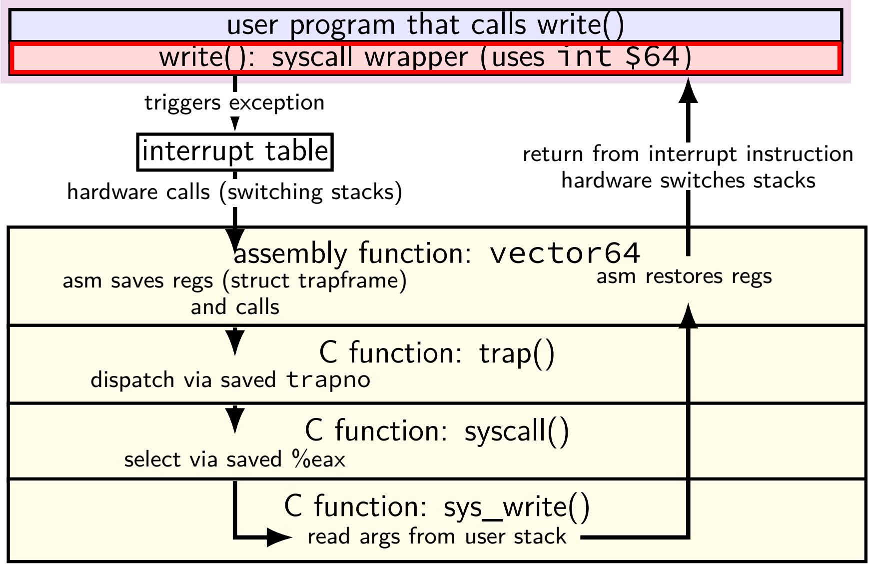
write syscall and xv6



write syscall and xv6



write syscall and xv6



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: user mode

main.c

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

usys.S

(before macro replacement:)

```
#define SYSCALL(name) \
    .global name ...
...
SYSCALL(write)
```

syscall.h / traps.h

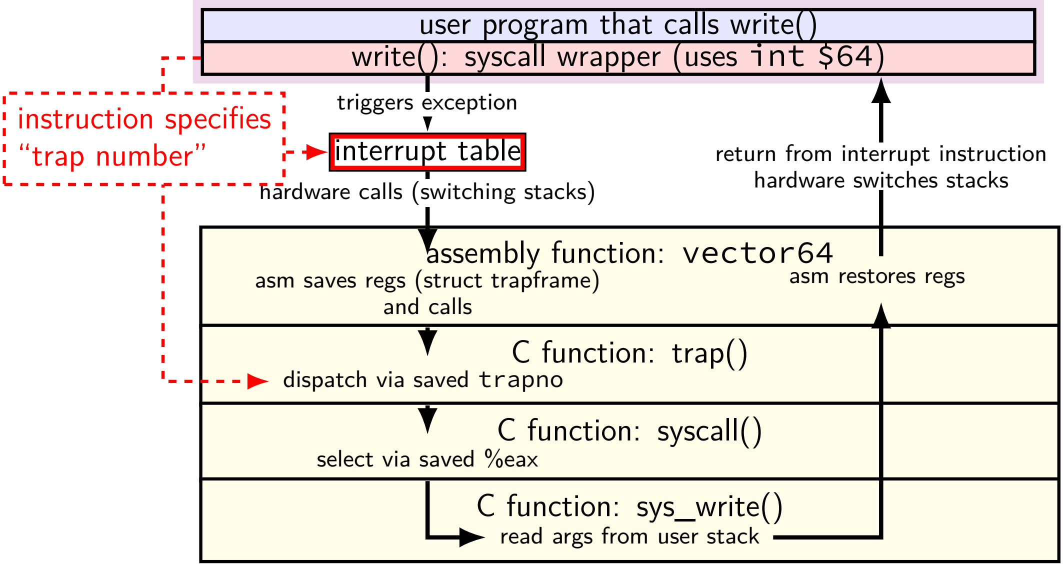
```
...
#define SYS_write   16
...
#define T_SYSCALL   64
...
```

usys.S

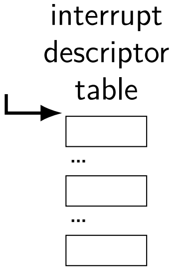
(partial, after macro replacement)

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

write syscall and xv6

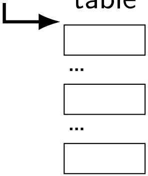


xv6: interrupt table indirection



xv6: interrupt table indirection

interrupt
descriptor
table

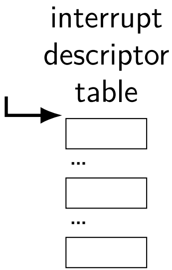


trap.c (run on boot)

```
...
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
("load interrupt descriptor table")
sets interrupt descriptor table to idt

xv6: interrupt table indirection



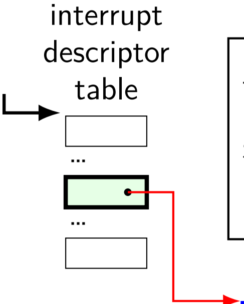
trap.c (run on boot)

```
...
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)                \
```

xv6: interrupt table indirection



trap.c (run on boot)

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

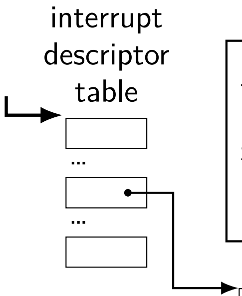
...

vectors.S

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

...

xv6: interrupt table indirection



trap.c (run on boot)

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

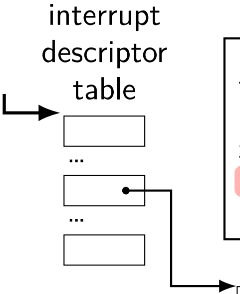
...

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

...

set table entry to
tell hardware to allow this exception
to be triggered with `int`
(otherwise: int triggers
“permission denied” exception instead)

xv6: interrupt table indirection



trap.c (run on boot)

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

...

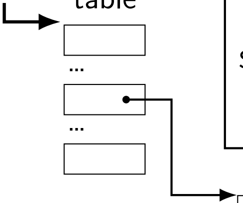
```
vector64:
    pushl $0
    pushl $64
    jmp alltraps
```

...

set table entry to
tell hardware to handle
exception in kernel mode

xv6: interrupt table indirection

interrupt descriptor table



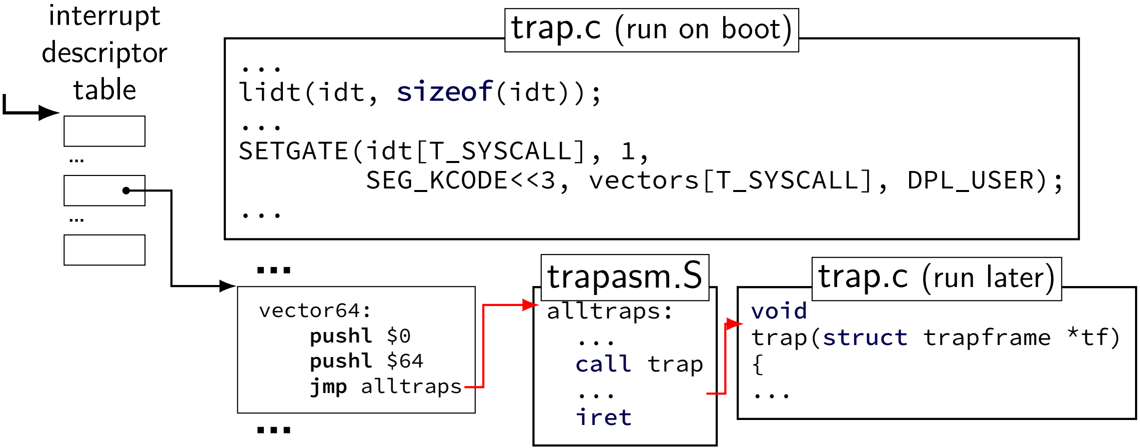
trap.c (run on boot)

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

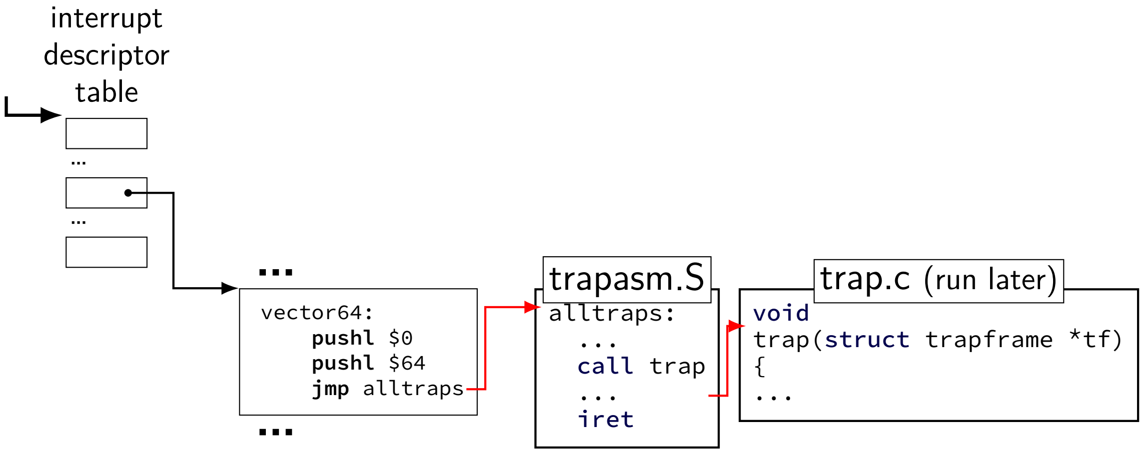
```
...
vector64:
    pushl $0
    pushl $64
    jmp alltraps
...
```

set table entry to allow `int` instructions to trigger this (otherwise: would trigger "permission denied" exception)

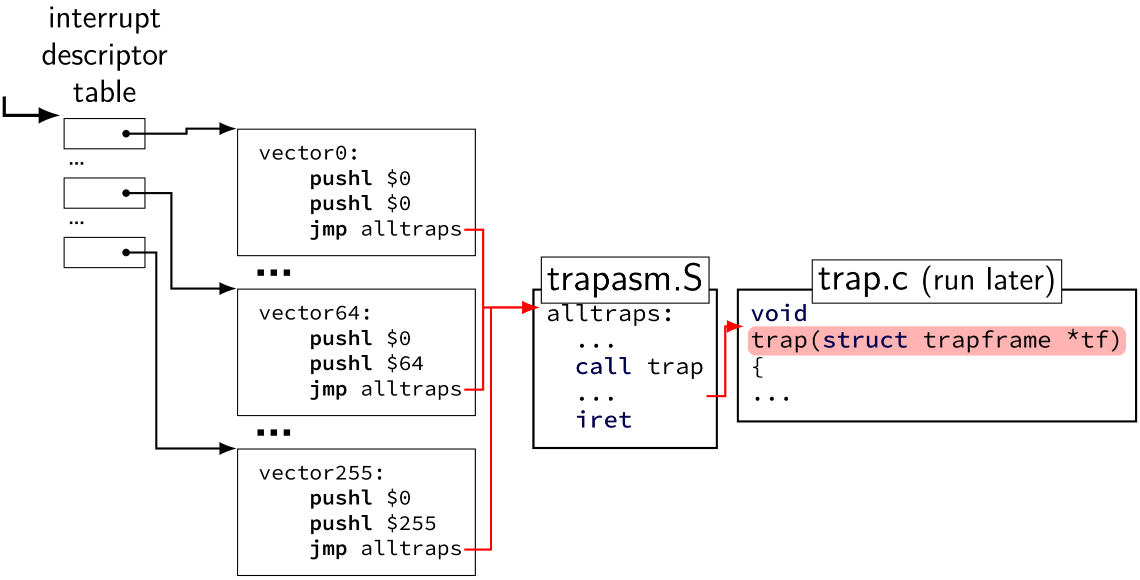
xv6: interrupt table indirection



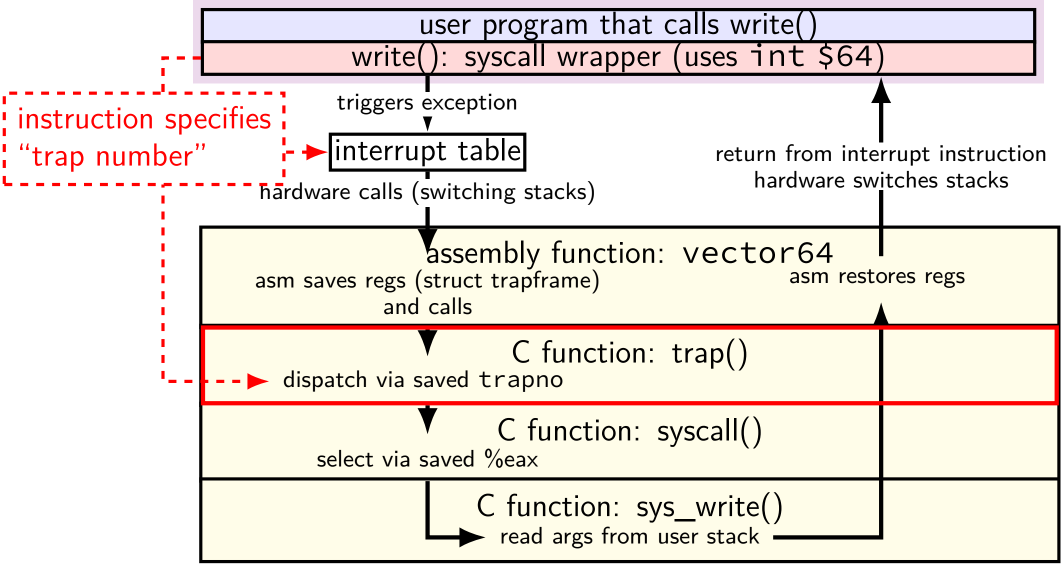
xv6: interrupt table indirection



xv6: interrupt table indirection



write syscall and xv6



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 register values,

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

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

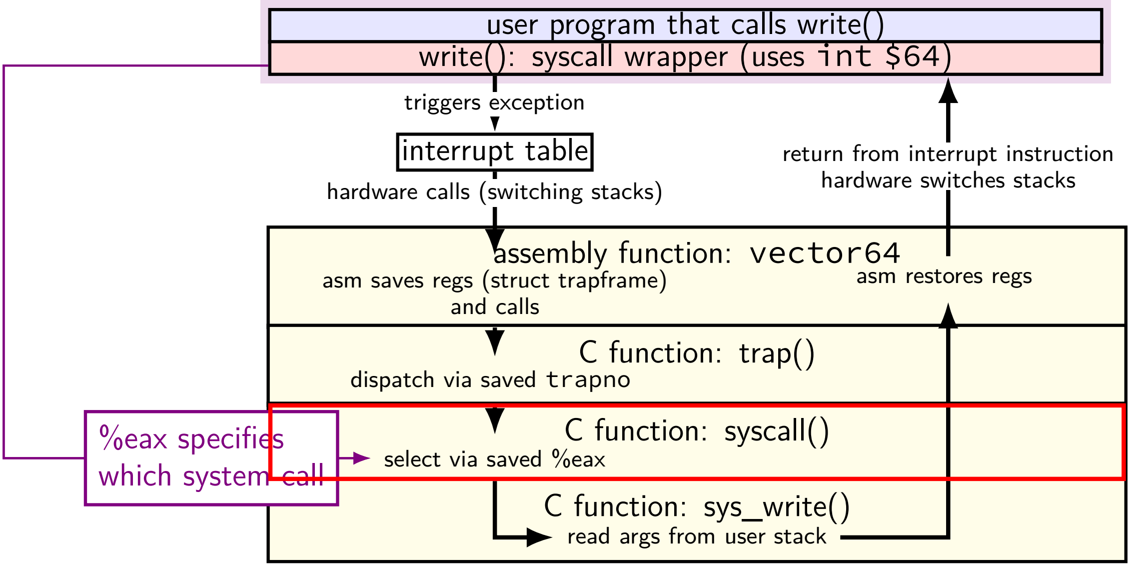
write syscall in xv6: the trap function

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 and xv6



write syscall in xv6: syscall()

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: syscall()

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: syscall()

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

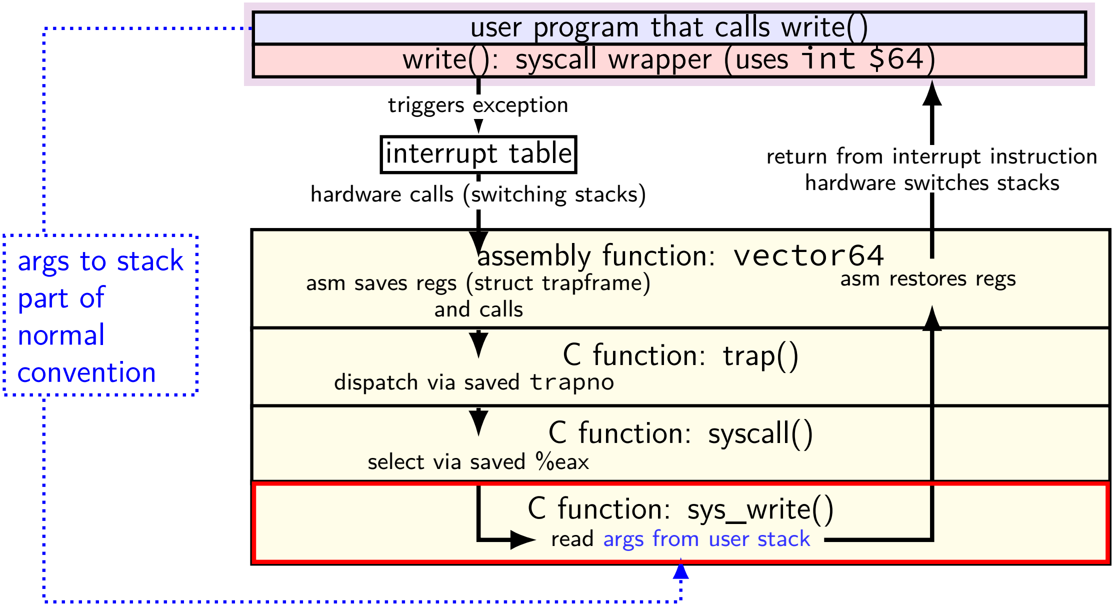
write syscall in xv6: syscall()

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 and xv6



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)

hello.c

```c
#include "user.h"
int main(void) {
  write(1, "Hello, World!", 13);
  exit();
}
```

```asm
// following 32-bit x86
// calling convention:
pushl $13
pushl $string_address
pushl $1
call write
```

hello.exe
function call interface
standard libraries
system call interface
kernel (extra HW access)
hardware interface

hello.c

```
#include "user.h"
int main(void) {
  write(1, "Hello, World!", 13);
  exit();
}
```

```
// following 32-bit x86
// calling convention:
pushl $13
pushl $string_address
pushl $1
call write
```

hello.exe
function call interface
standard libraries
system call interface
kernel (extra HW access)
hardware interface

stack @ entry to write

	13
	ptr to “Hello, World!”
	1
%esp →	(return address of call)

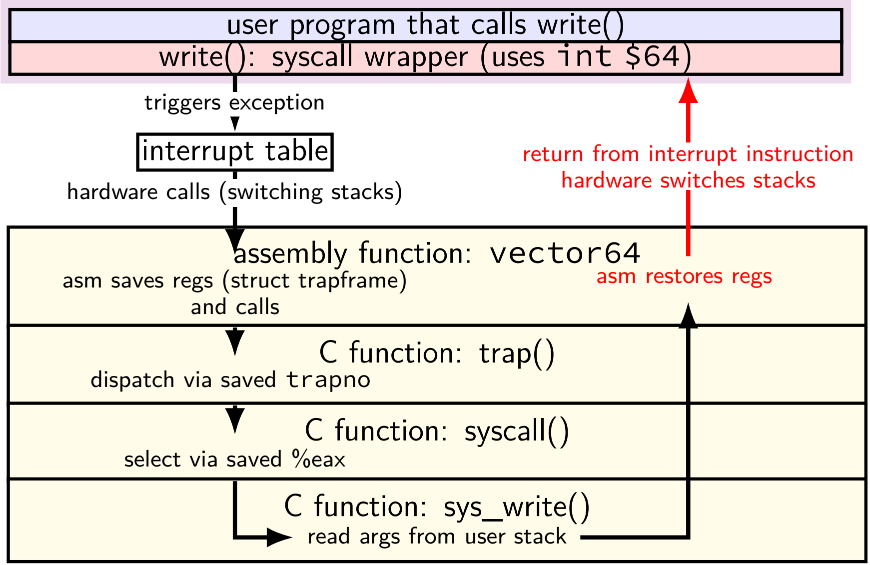
write syscall in xv6: sys_write

sysfile.c

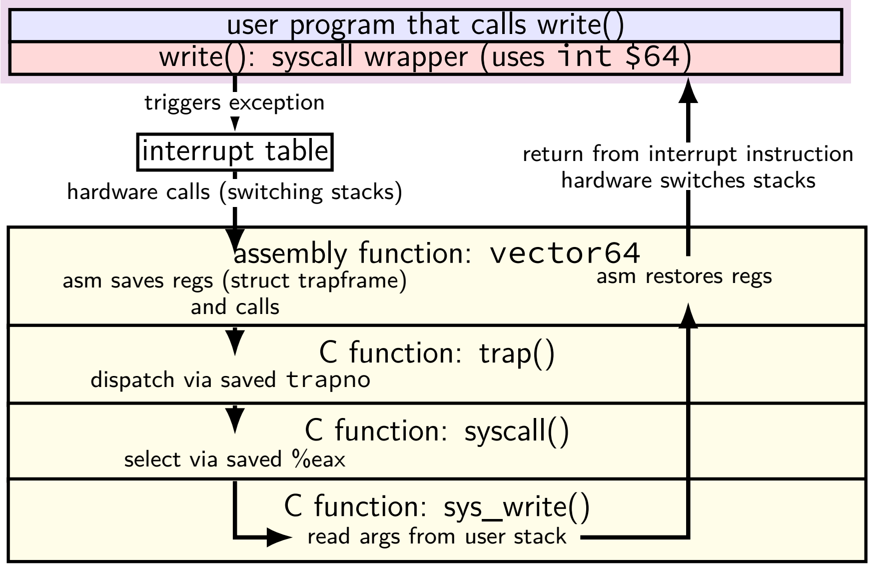
```
int
sys_write(void)
{
  struct file *f;
  int n;
  char *p;

  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)

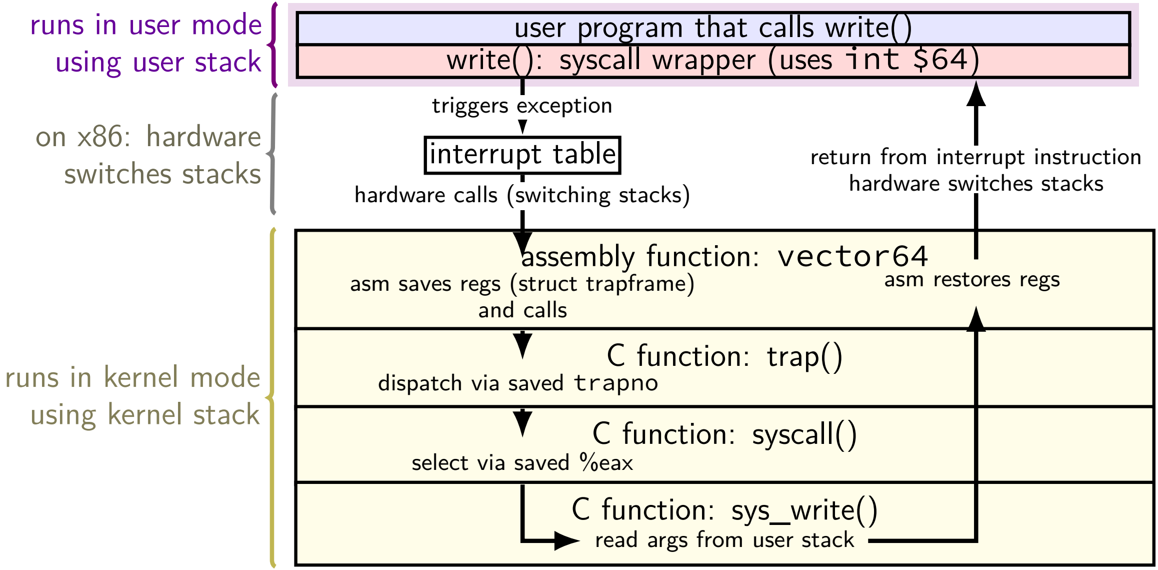
write syscall and xv6



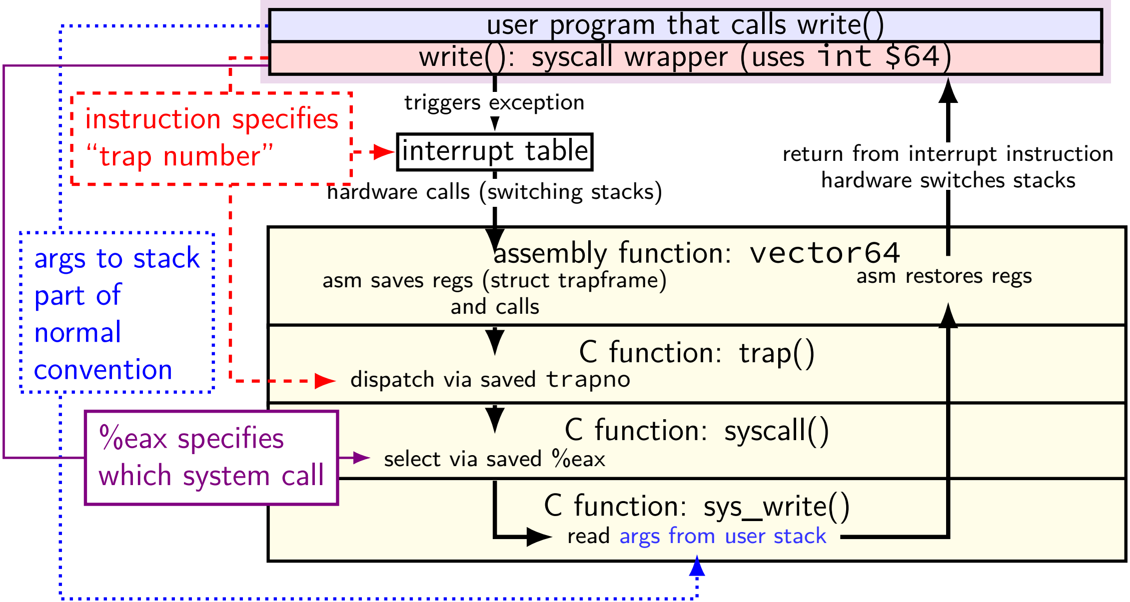
write syscall and xv6



write syscall and xv6



write syscall and 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

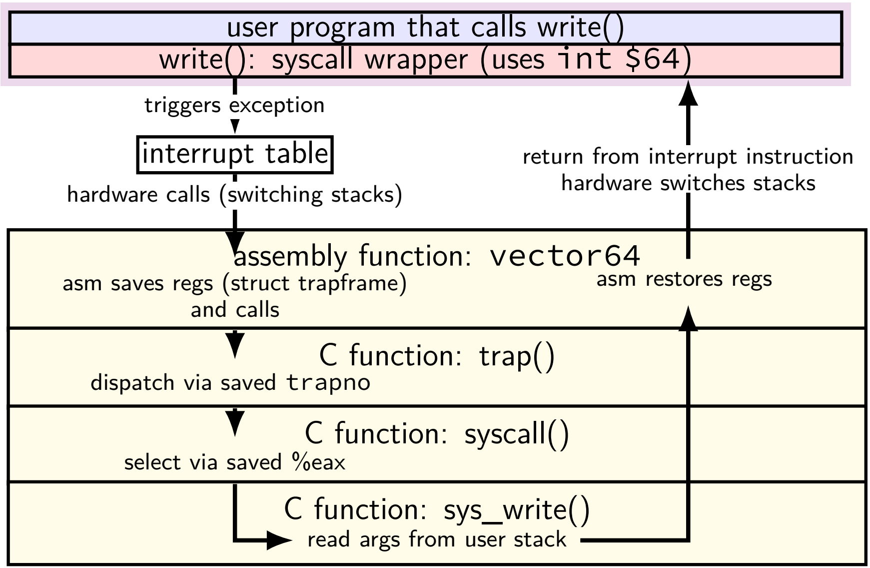
primarily to handle multiple cores

write syscall and xv6

runs in user mode
using user stack

on x86: hardware
switches stacks

runs in kernel mode
using kernel stack



exercise: where is...?

On xv6, one can use code like the following to read from stdin:

```
char c;
int result = 0;
result = read(0, &c, 1);
if (result == 1) {
    /* success */
}
```

When the read system call starts running `sys_read` in the kernel where is:

the char `c` | pointer to `c` | return address of read()

first address executed in user mode after syscall runs

A. user stack
B. kernel stack (including trapframe)
C. hardware registers
D. elsewhere
E. not stored

exercise explanation

char `c`: local variable, allocated on stack in user mode

pointer to `c`: argument to read(), placed on stack in 32-bit x86

based on calling convention. usually different on 64-bit x86

return address of read(): placed on stack in 32-bit x86

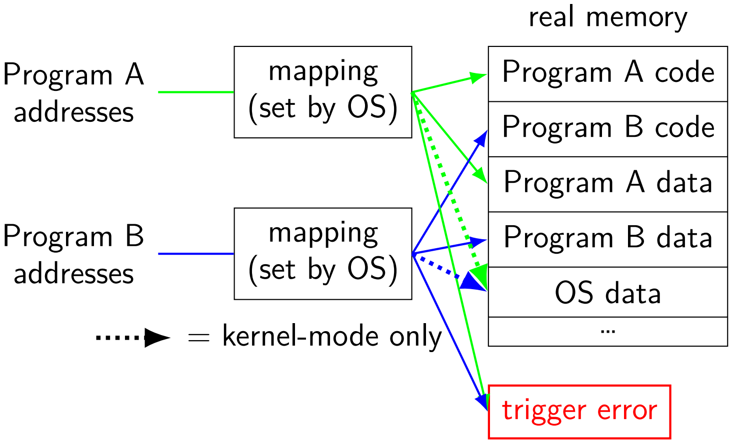
`call` instruction

first-address executed in user mode after syscall runs

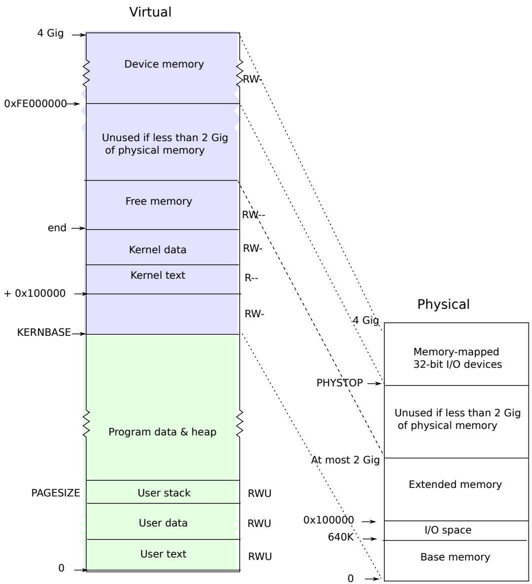
xv6: `myproc()->tf->eip`

part of trapframe; needed from return from exception instruction

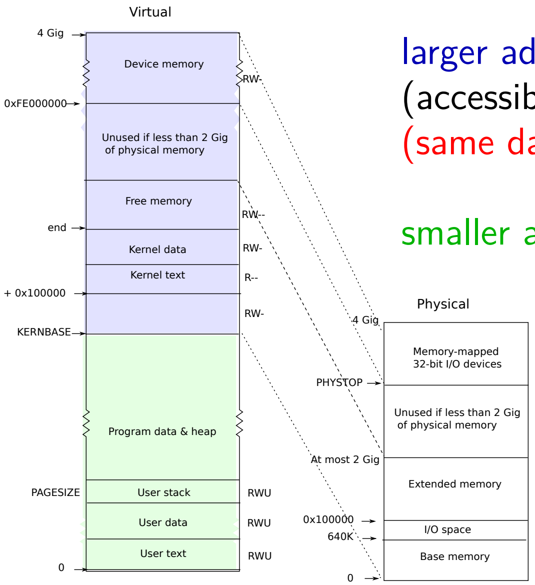
address translation



xv6 memory layout



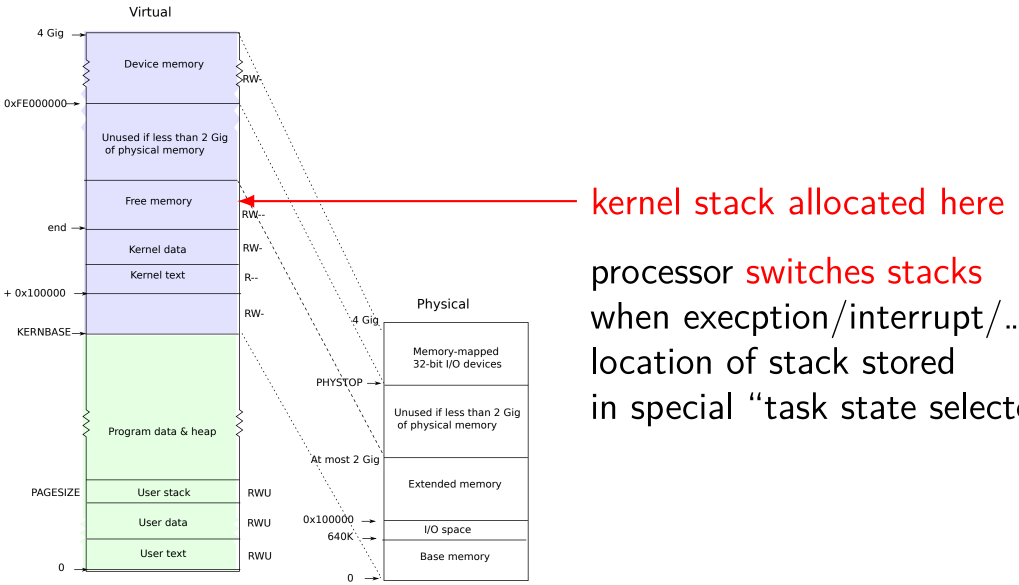
xv6 memory layout



larger addresses are for kernel
(accessible in kernel mode *only*)
(same data in all processes)

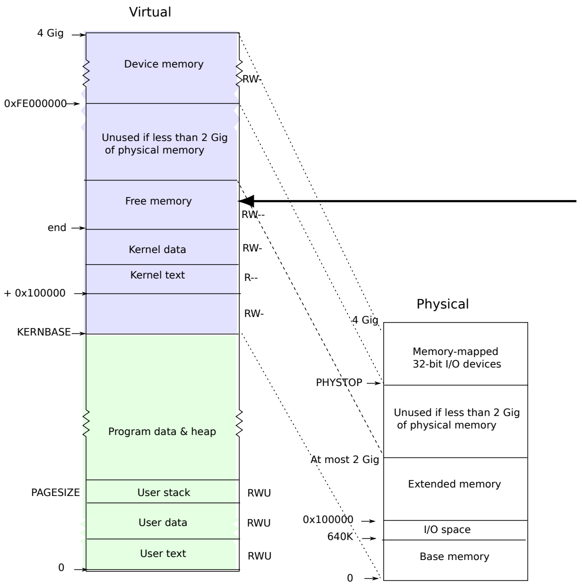
smaller addresses are for applications

xv6 memory layout



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

xv6 memory layout



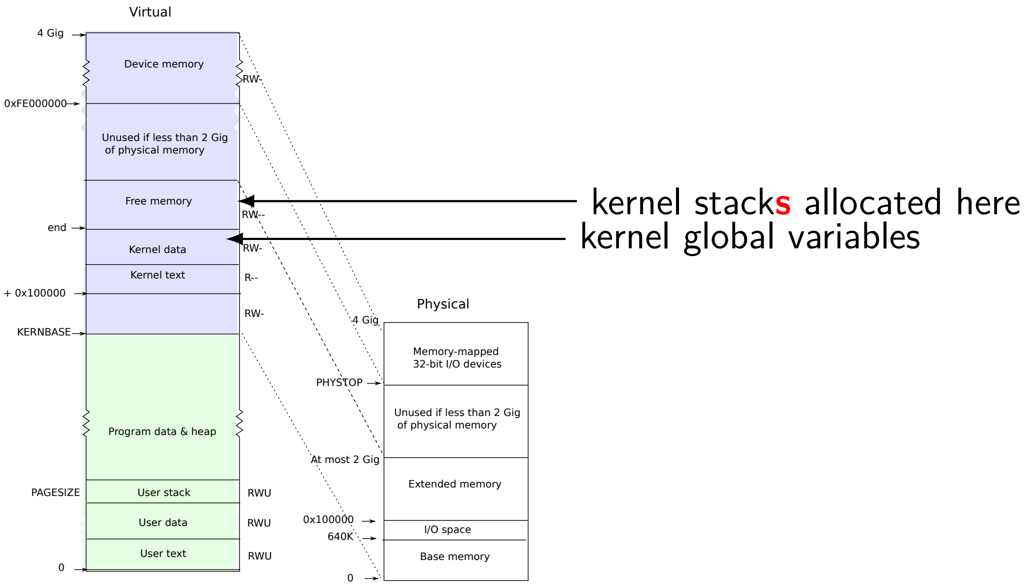
kernel stack**s** allocated here

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)

xv6 memory layout



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

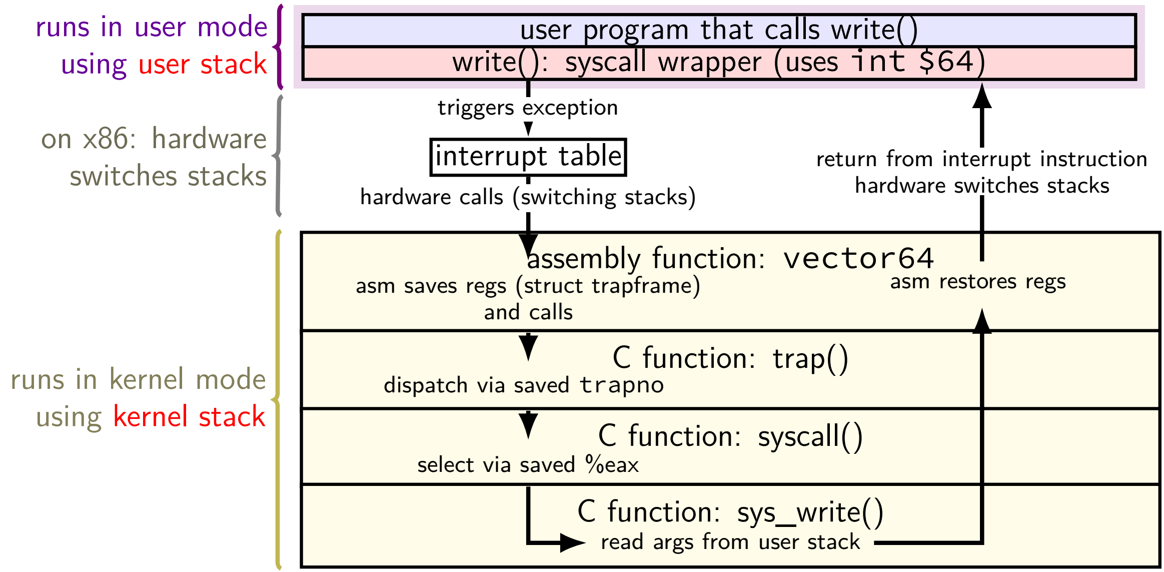
start in kernel at top of kernel stack for current thread/process

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

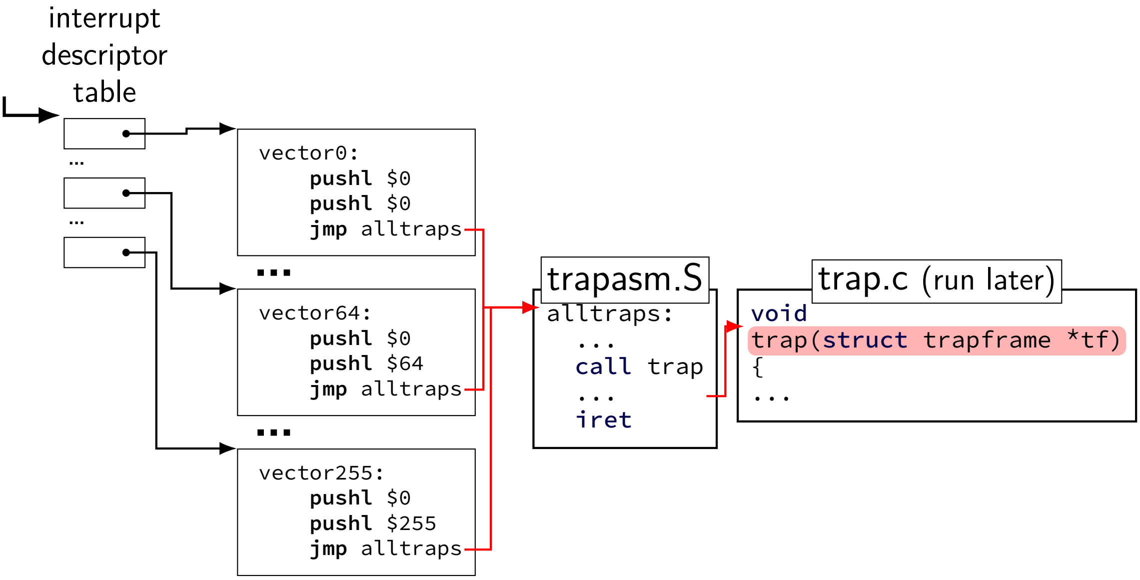
continues using current kernel stack with same stack pointer

(processor tracks that it switched already)

write syscall and xv6



xv6: interrupt table indirection



non-system call exceptions

many non-system call exceptions xv6 handles in trap():

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

non-system call exceptions

many non-system call exceptions xv6 handles in trap():

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

many non-system call exceptions xv6 handles in trap():

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)

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;
```

separate from system call

system call:

application indicates interest in I/O

these exceptions:

device indicates interest in I/O

non-system call exceptions

many non-system call exceptions xv6 handles in trap():

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)

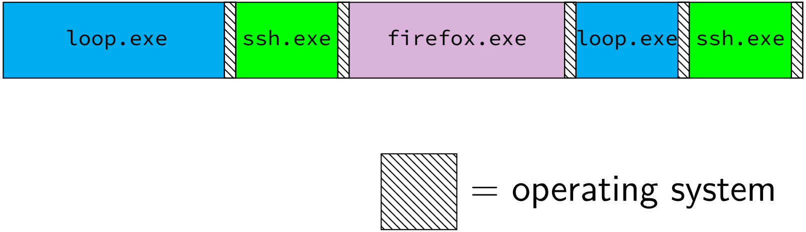
```
void
trap(struct trapf
{
  ...
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
  ...
  // Force process to give up CPU on clock tick.
  ...
  if(myproc()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

xv6: timer interrupt

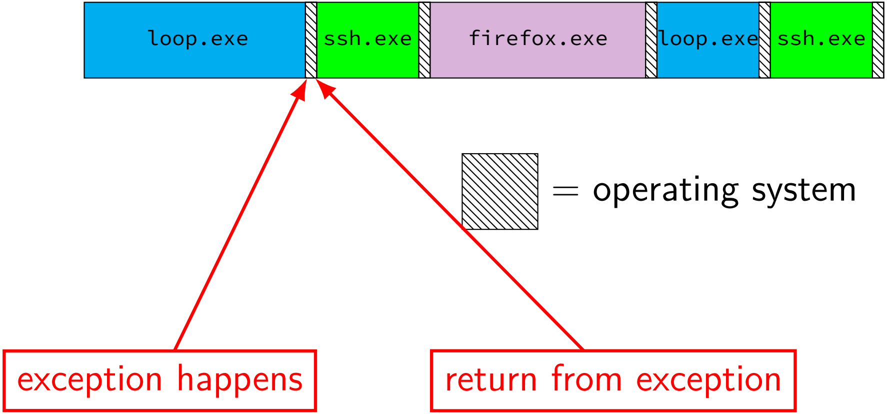
acquire/release — related to synchronization (later)

```
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



time multiplexing



OS and time multiplexing

starts running instead of normal program via exception

saves old program counter, register values somewhere

sets new register values, jumps to new program counter

called context switch

- saved information called context

context

all registers value

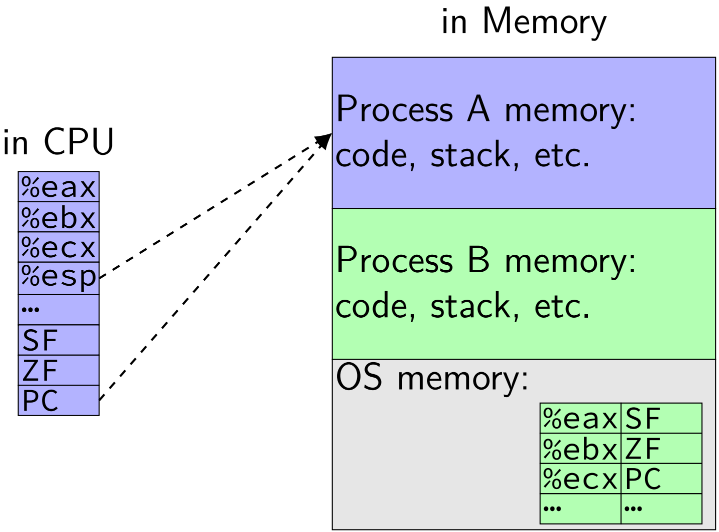
`%eax %ebx`, ..., `%esp`, ...

condition code values

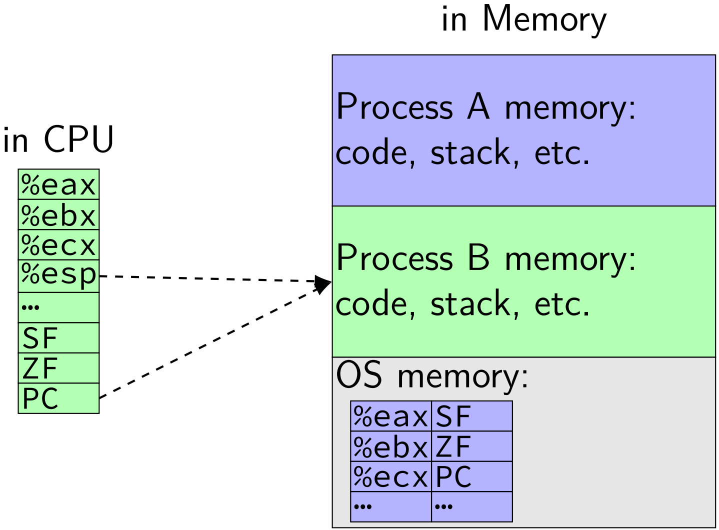
program counter value

address space = page table base pointer

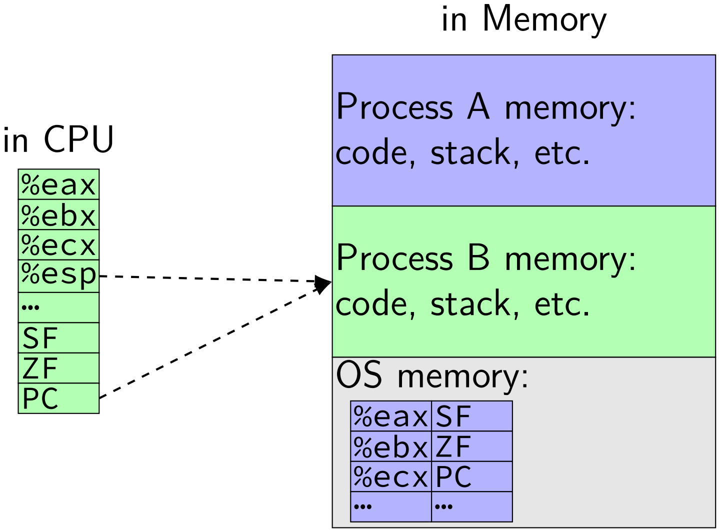
contexts (A running)



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/syscalls

two active processes:

- A: running infinite loop
- B: described below

process B asks to read from from the keyboard

after input is available, B reads from a file

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

how many context switches do we expect?

how many system calls do we expect?

- your answers can be ranges

counting system calls

(no system calls from A)

B: read from keyboard

- maybe more than one — lots to read?

B: read from file

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

B: write to screen

- maybe more than one — lots to write?

(3 or more from B)

counting context switches

B makes system call to read from keyboard

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

keyboard input: B can run

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

B makes system call to read from file

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

if data from file not available right away

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

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

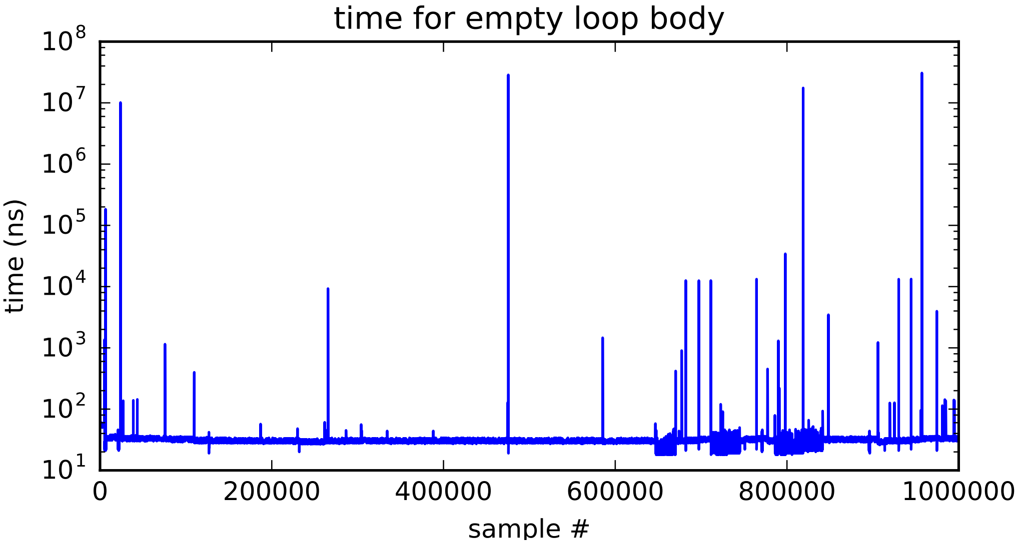
backup slides

timing nothing

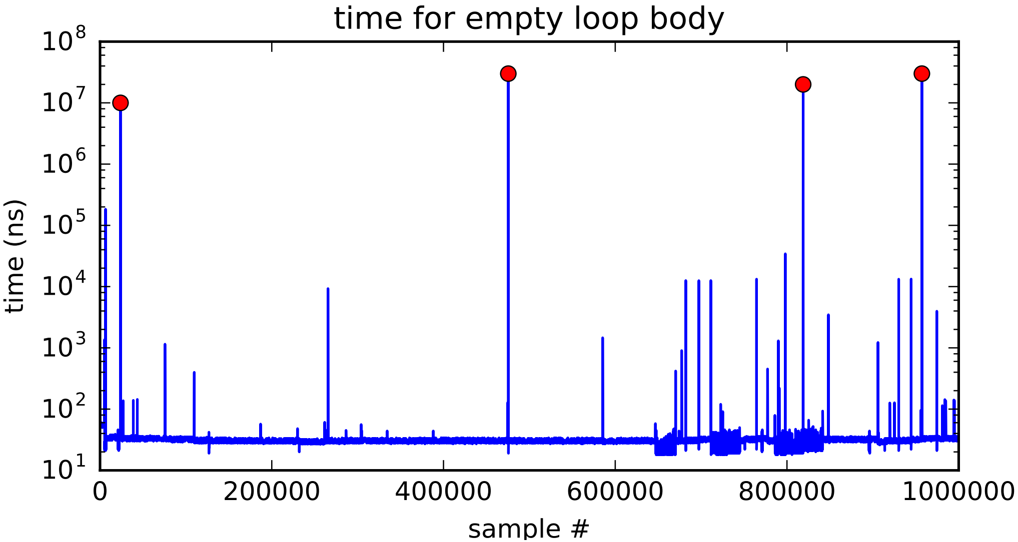
```
long times[NUM_TIMINGS];
int main(void) {
    for (int i = 0; i < N; ++i) {
        long start, end;
        start = get_time();
        /* do nothing */
        end = get_time();
        times[i] = end - start;
    }
    output_timings(times);
}
```

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

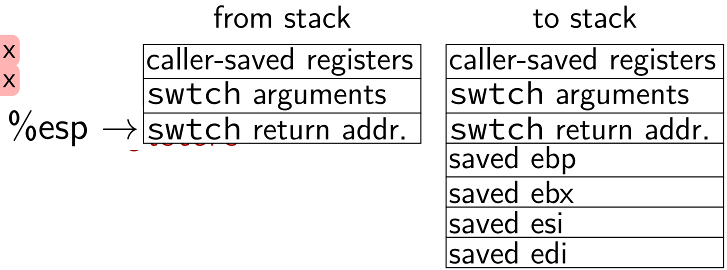
```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # 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
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

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

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



juggling stacks

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

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

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

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

	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

struct context
(saved into `from` arg)

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

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

first instruction executed by new thread

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	to stack
saved user regs	saved user regs
…	…
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

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

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

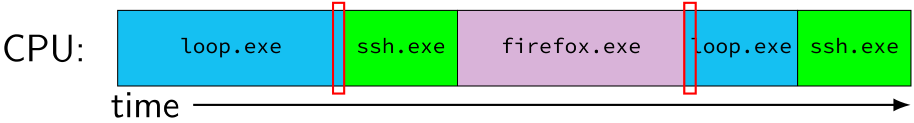
time multiplexing


```
...
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);
```

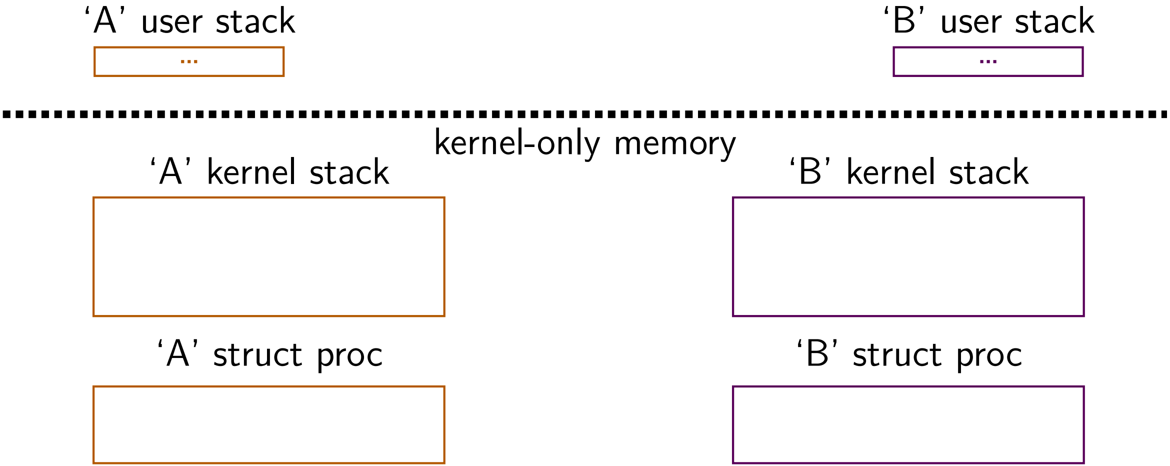
struct context

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

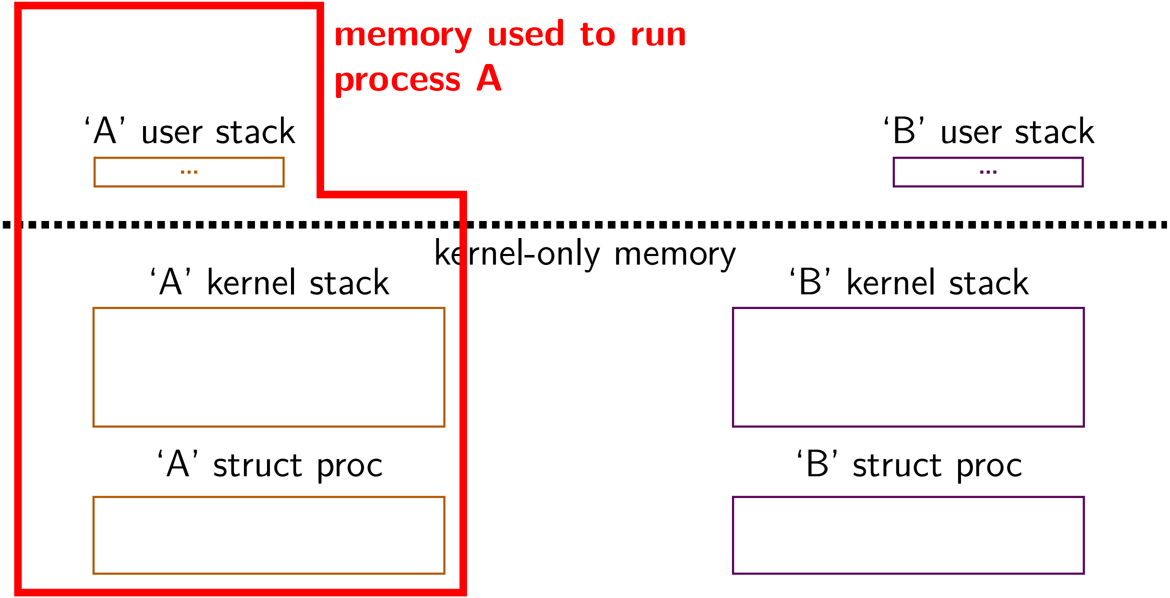
```
struct context {
  uint edi;            /* <-- top of stack of this thread */
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;          /* <-- return address of swtch() */
  /* not in struct but stored on stack thread after eip:
        arguments to current call to swtch
        caller-saved registers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

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

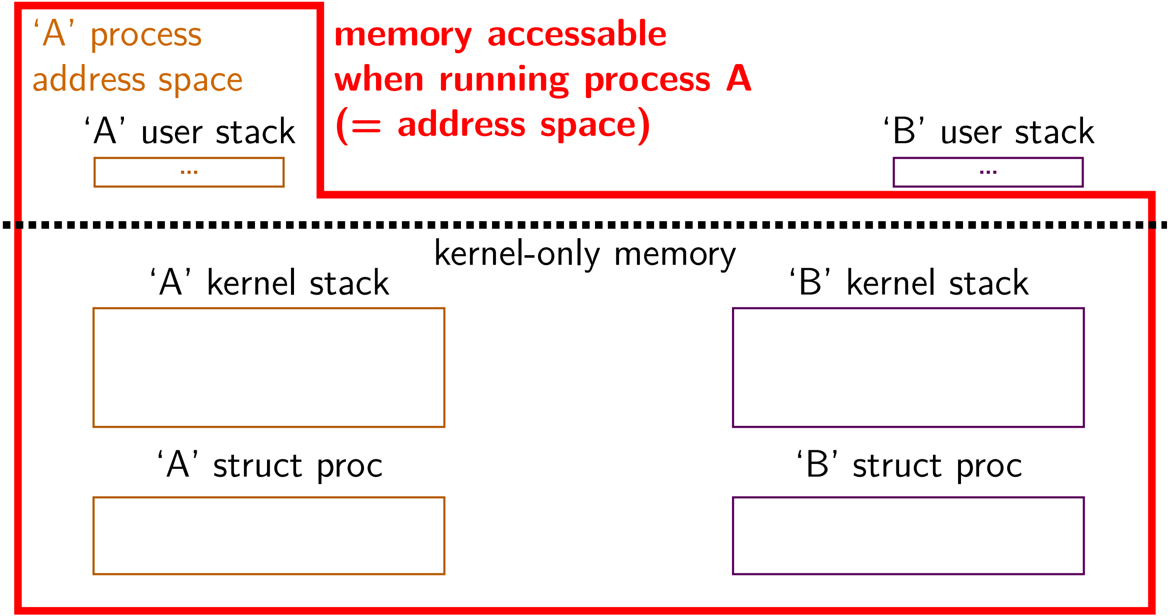
xv6: where the context is



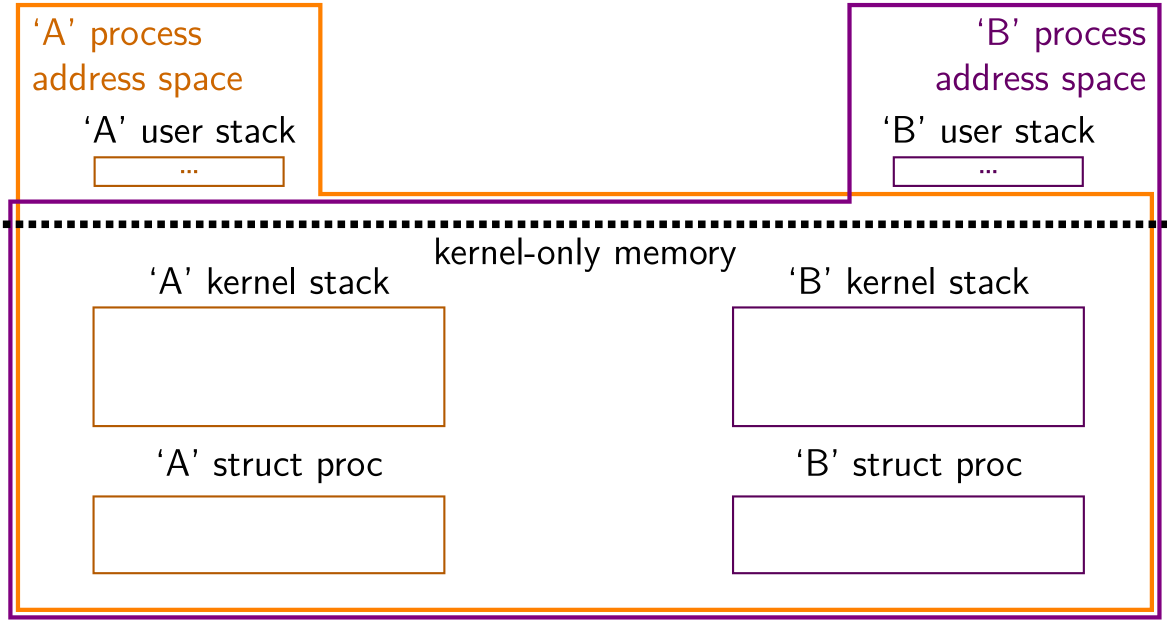
xv6: where the context is



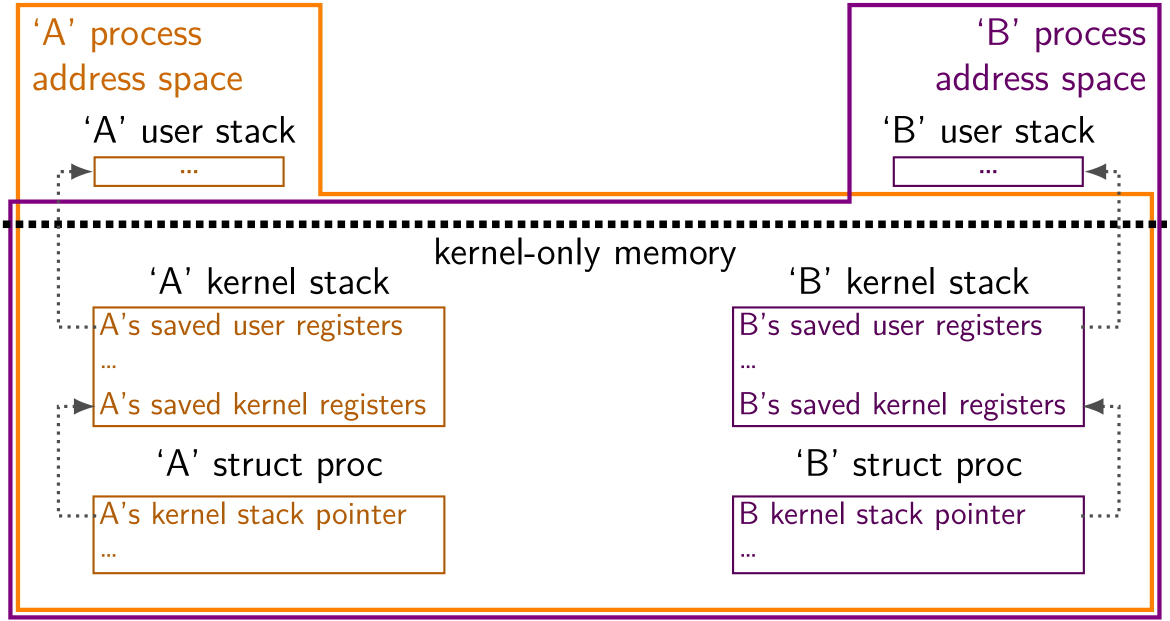
xv6: where the context is



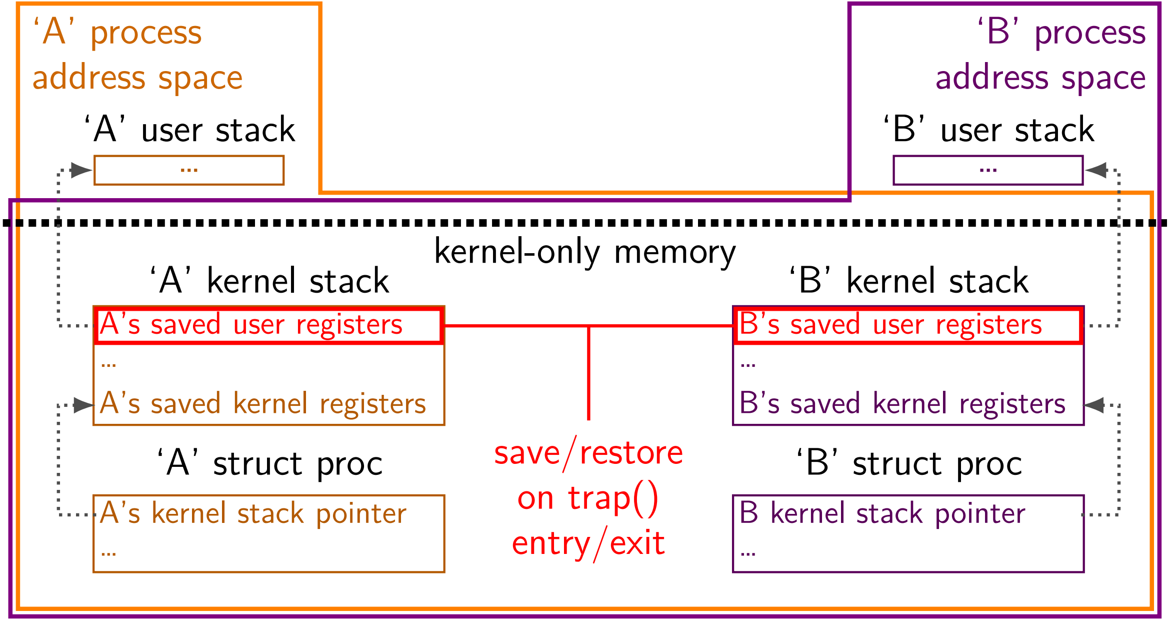
xv6: where the context is



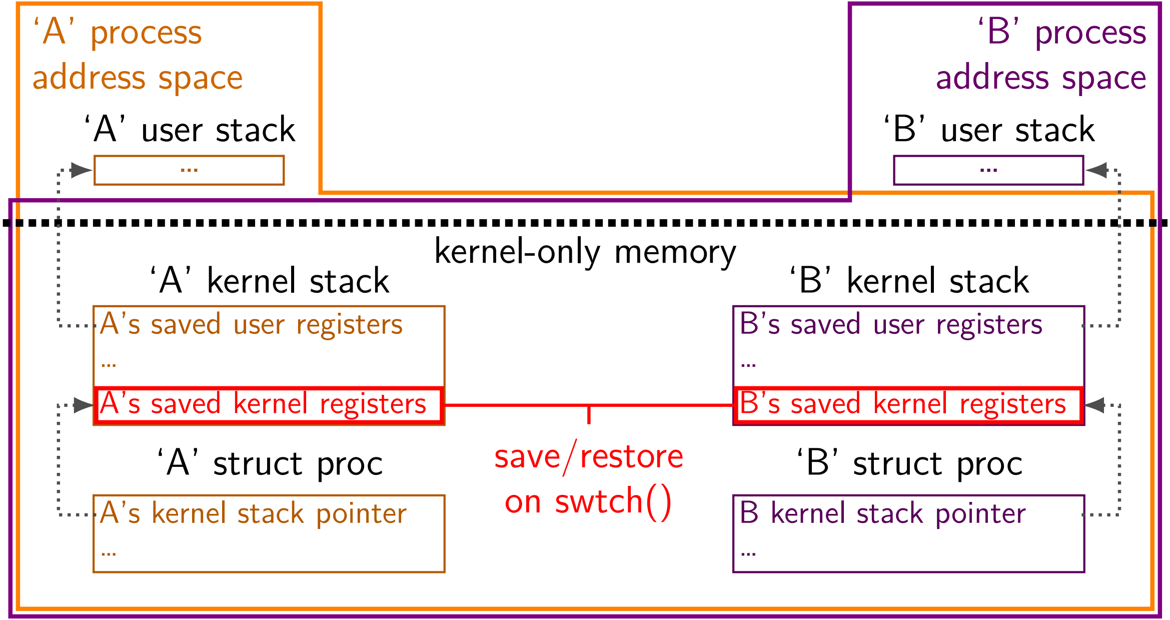
xv6: where the context is



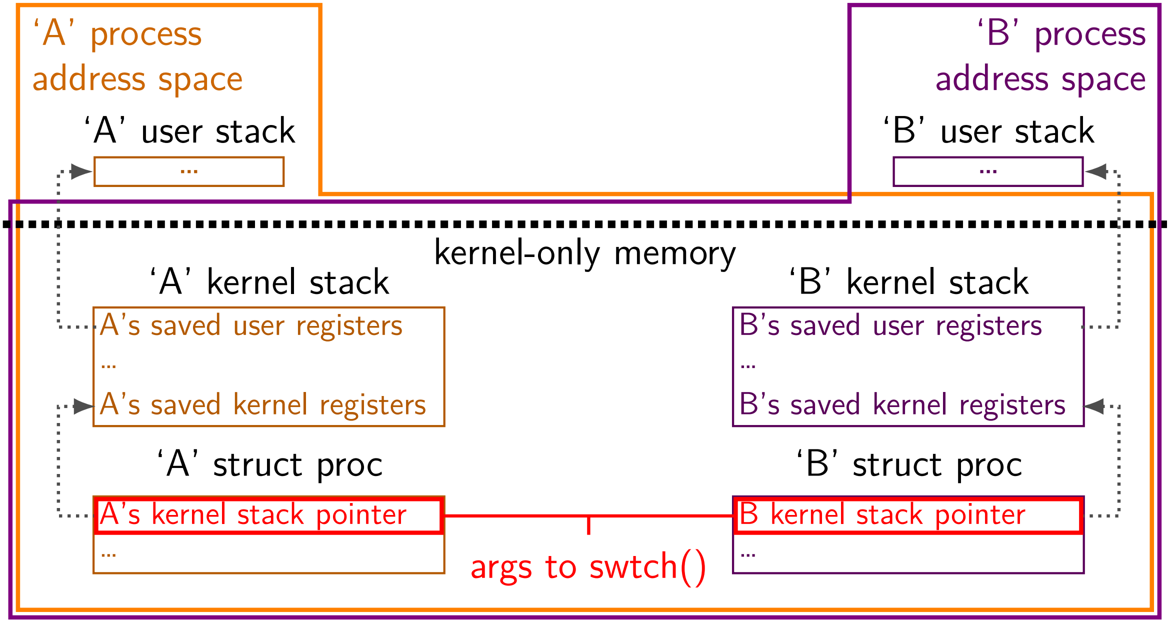
xv6: where the context is



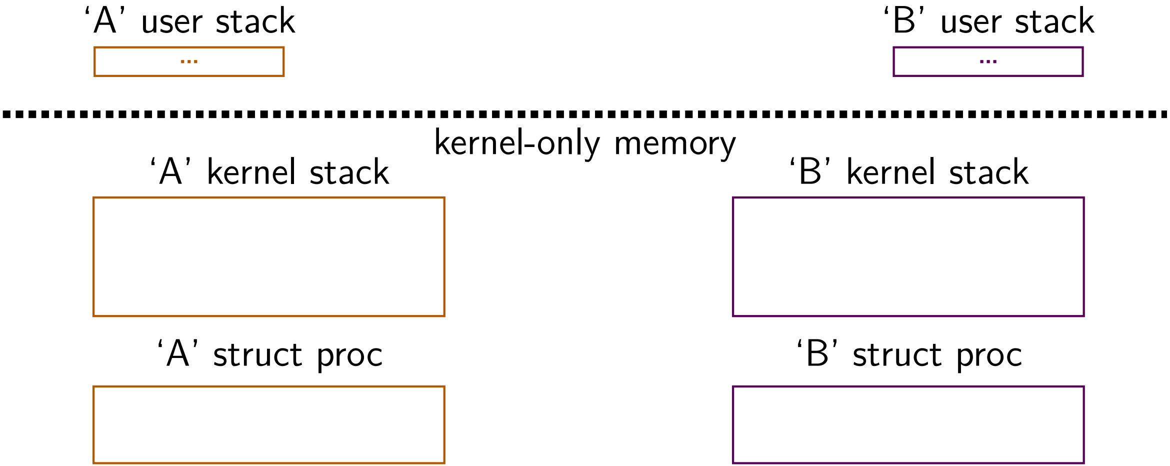
xv6: where the context is



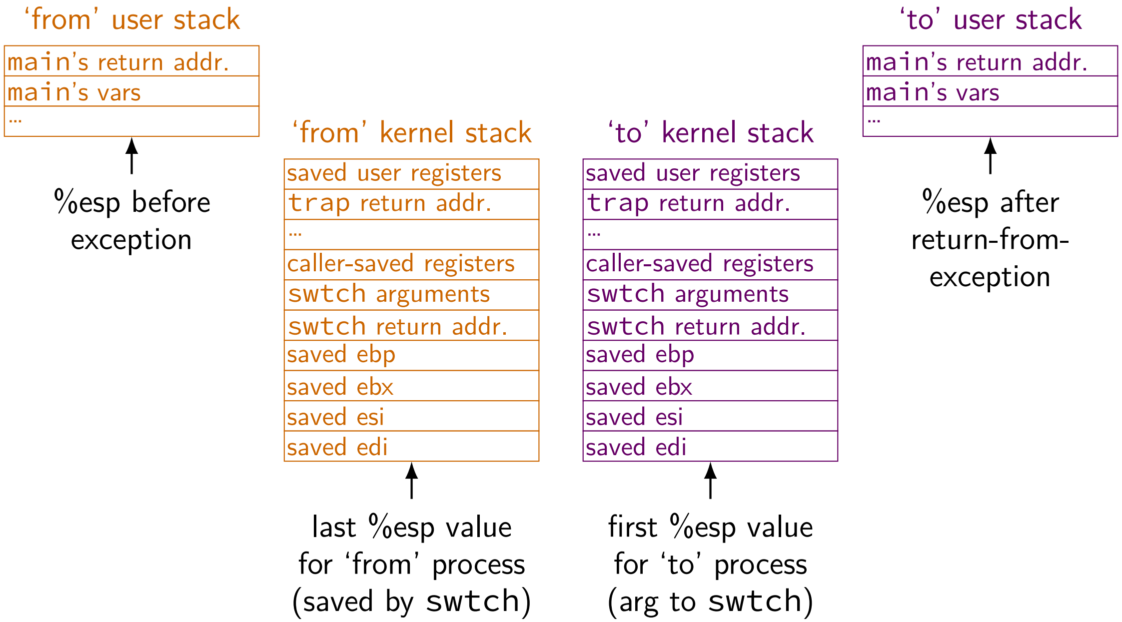
xv6: where the context is



xv6: where the context is



xv6: where the context is (detail)



xv6: where the context is (detail)

‘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

saved in ‘from’ `struct proc`

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`)

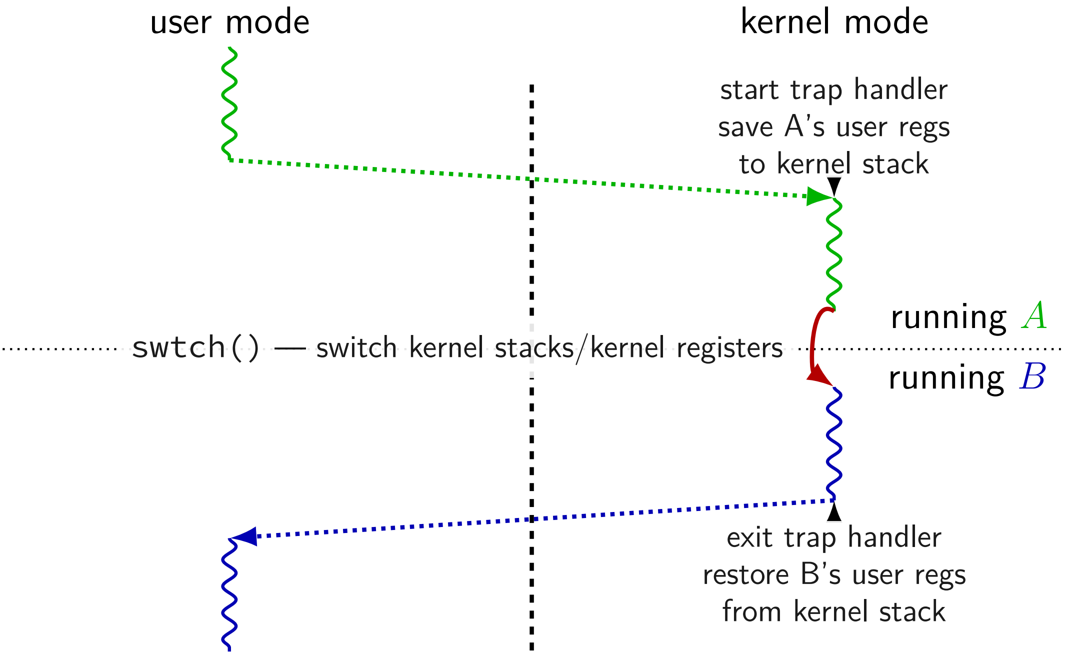
retrieved via ‘to’ `struct proc`

‘to’ user stack

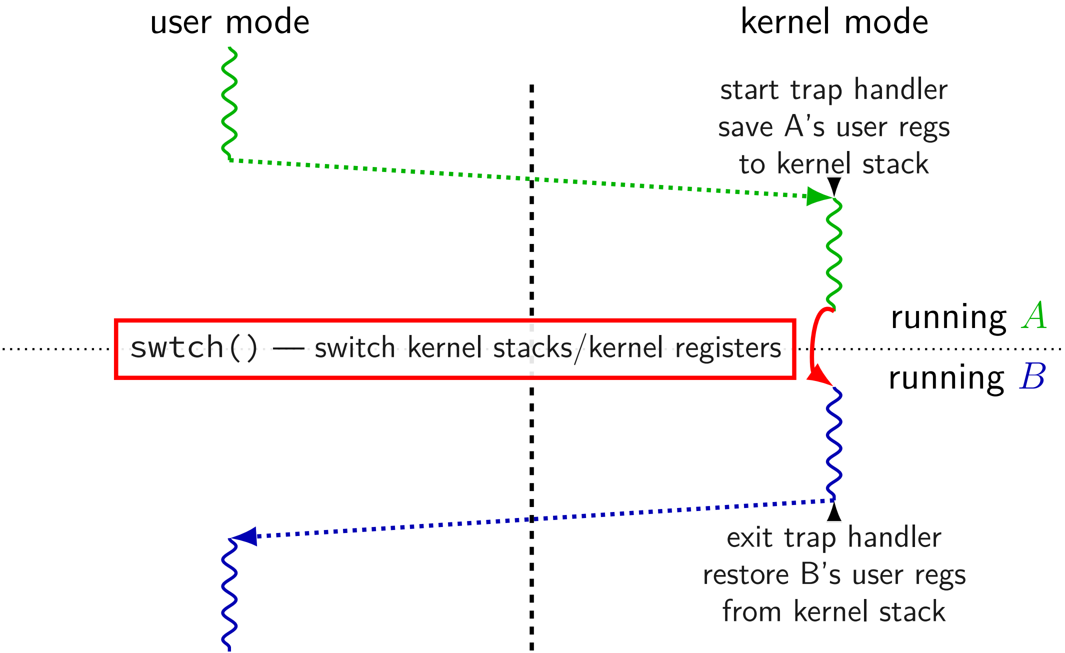
`main`’s return addr.
`main`’s vars
…

%esp after return-from-exception

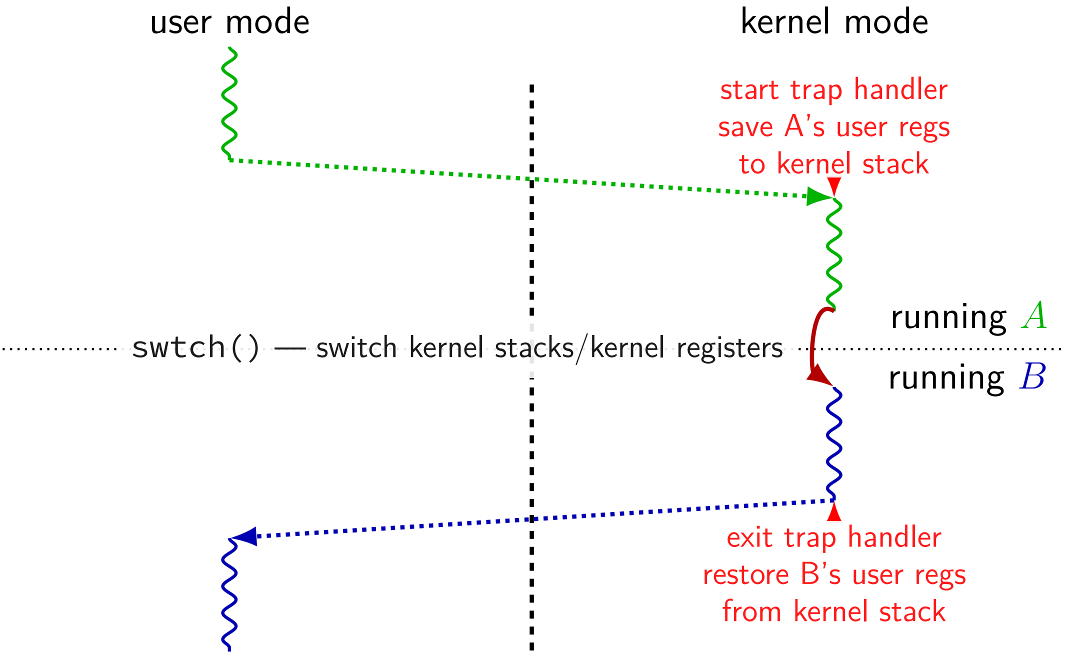
xv6 context switch and saving



xv6 context switch and saving



xv6 context switch and saving



system call timeline (xv6)

in user mode
(= limited hardware access)
(the standard library)

```
/* place arguments somewhere*/
movl $SYS_write, %eax
pushl $BUFFER_LEN     // argument 3
pushl $buffer         // argument 2
pushl $FILENO_stdout // argument 1
pushl $0              // ignored
/* trigger exception */
int $0x40 // trigger exception
```

in kernel mode
(= extra hardware acces
(the "kernel")

```
handle_syscall:
  /* ... save registers */
  /* ... use %eax to figure o
          is needed
     ... actually do write an
          and set return value
  /* go back to "user" code
```

system call timeline (xv6)

in user mode
(= limited hardware access)
(the standard library)

in kernel mode
(= extra hardware acces
(the "kernel")

```
/* place arguments somewhere*/
movl $SYS_write, %eax
pushl $BUFFER_LEN     // argument 3
pushl $buffer         // argument 2
pushl $FILENO_stdout  // argument 1
pushl $0              // ignored
/* trigger exception */
int $0x40 // trigger exception
```

hardware knows to go here
because of pointer set during bo

```
handle_syscall:
  /* ... save registers */
  /* ... use %eax to figure o
          is needed
       ... actually do write an
          and set return value
  /* go back to "user" code
```

system call timeline (xv6)

in user mode
(= limited hardware access)
(the standard library)

```
/* place arguments somewhere*/
movl $SYS_write, %eax
pushl $BUFFER_LEN      // argument 3
pushl $buffer          // argument 2
pushl $FILENO_stdout   // argument 1
pushl $0               // ignored
/* trigger exception */
int $0x40 // trigger exception
```

'priviliged' operations
prohibited

in kernel mode
(= extra hardware acces

(the "kernel")

```
handle_syscall:
  /* ... save registers */
  /* ... use %eax to figure o
         is needed
     ... actually do write an
         and set return value
  /* go back to "user" code
```

system call timeline (xv6)

in user mode
(= limited hardware access)
(the standard library)

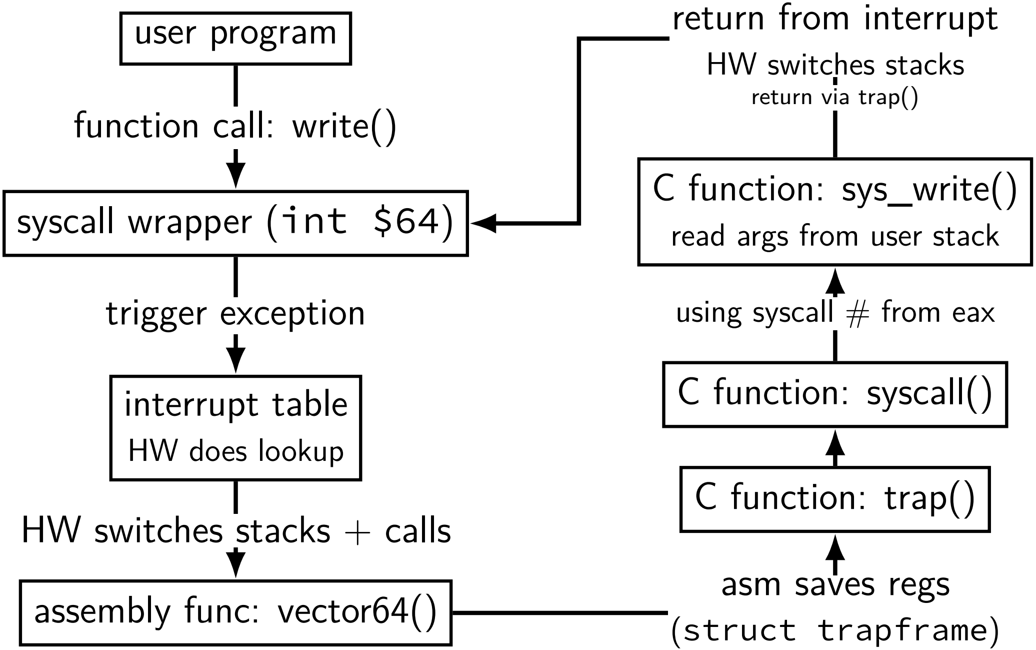
```
/* place arguments somewhere*/
movl $SYS_write, %eax
pushl $BUFFER_LEN     // argument 3
pushl $buffer         // argument 2
pushl $FILENO_stdout // argument 1
pushl $0              // ignored
/* trigger exception */
int $0x40 // trigger exception
```

in kernel mode
(= extra hardware acces
(the "kernel")

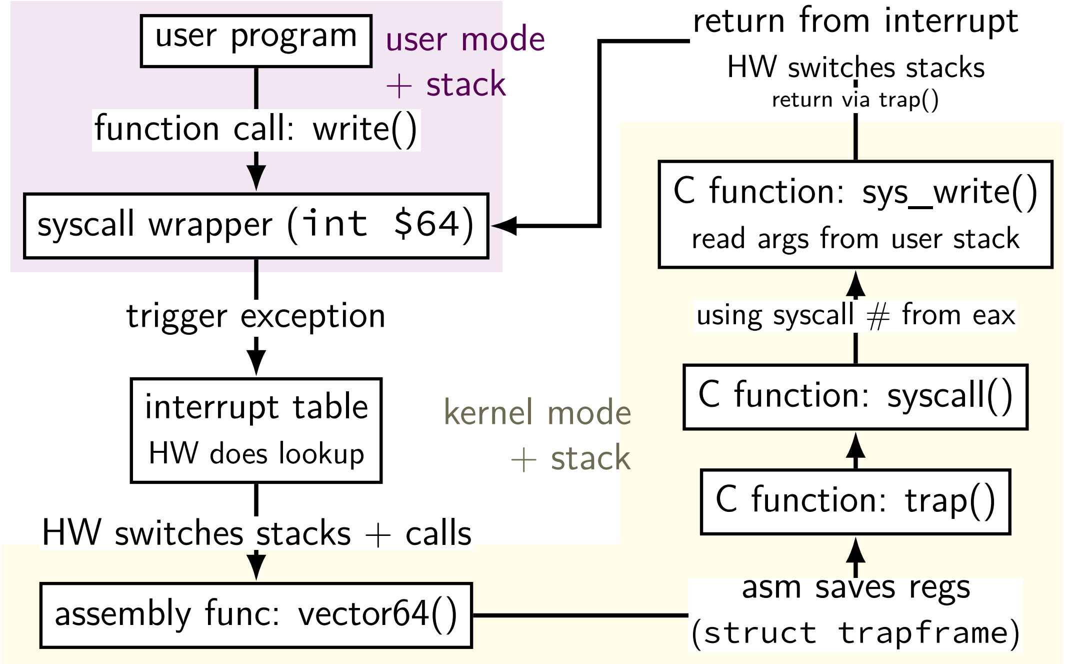
'priviliged' operations
allowed
(change memory layout, I/O, exc

```
handle_syscall:
  /* ... save registers */
  /* ... use %eax to figure o
         is needed
     ... actually do write an
         and set return value
  /* go back to "user" code
```

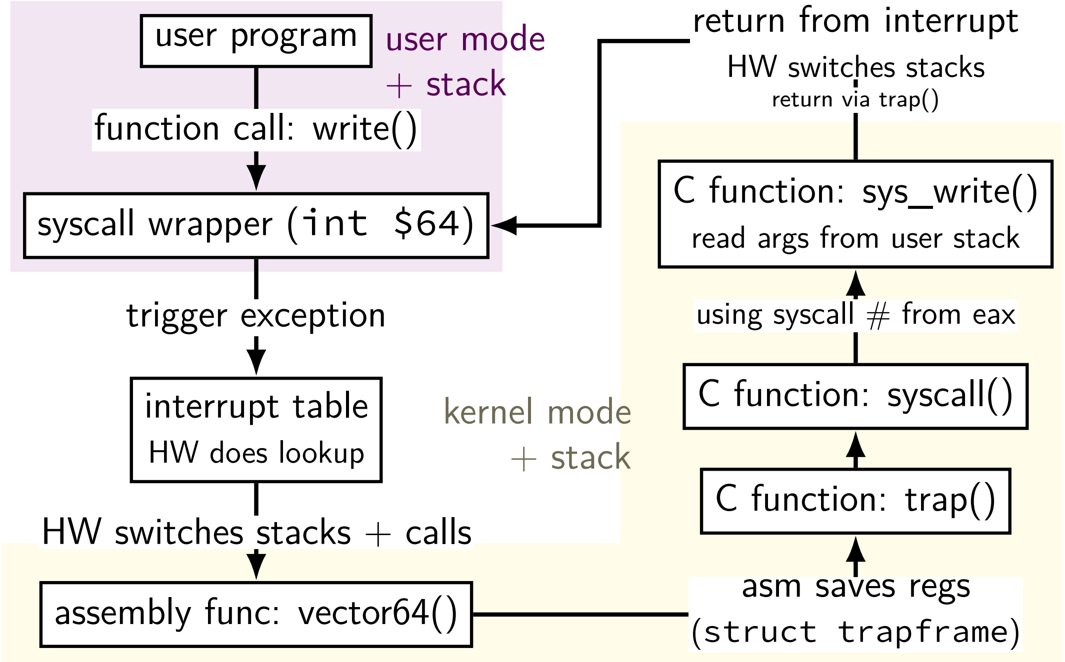
write syscall in xv6 (old)



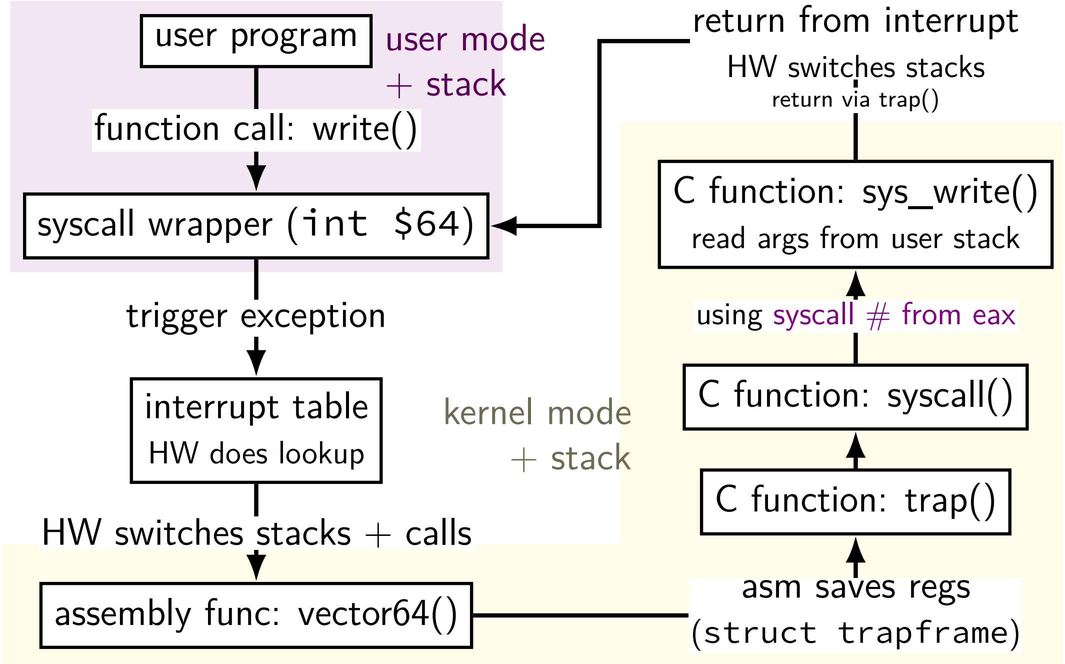
write syscall in xv6 (old)



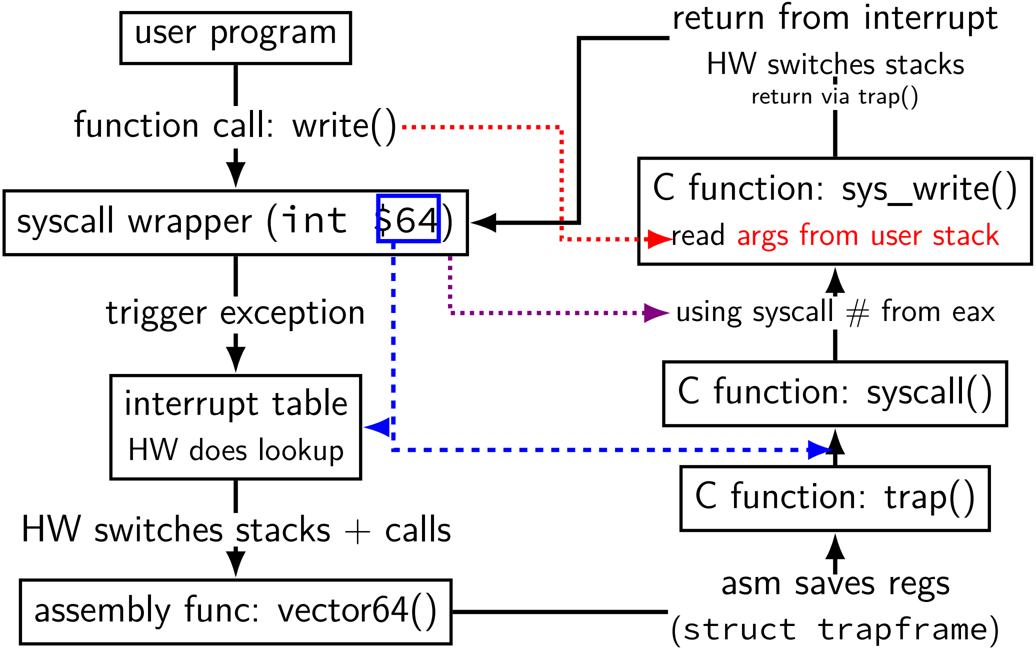
write syscall in xv6 (old)



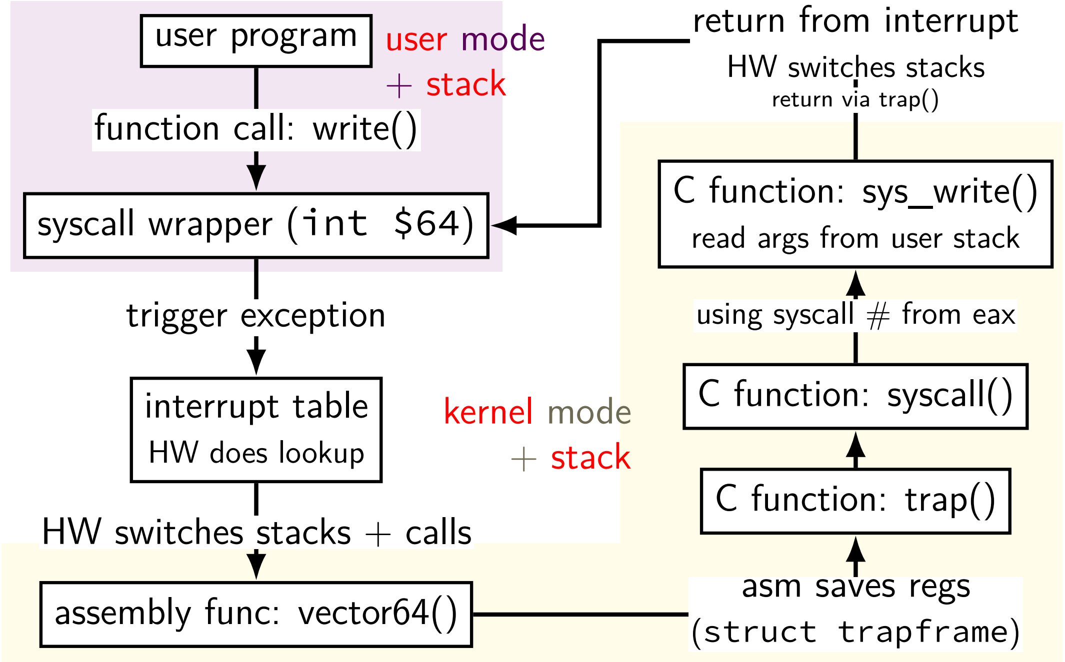
write syscall in xv6 (old)



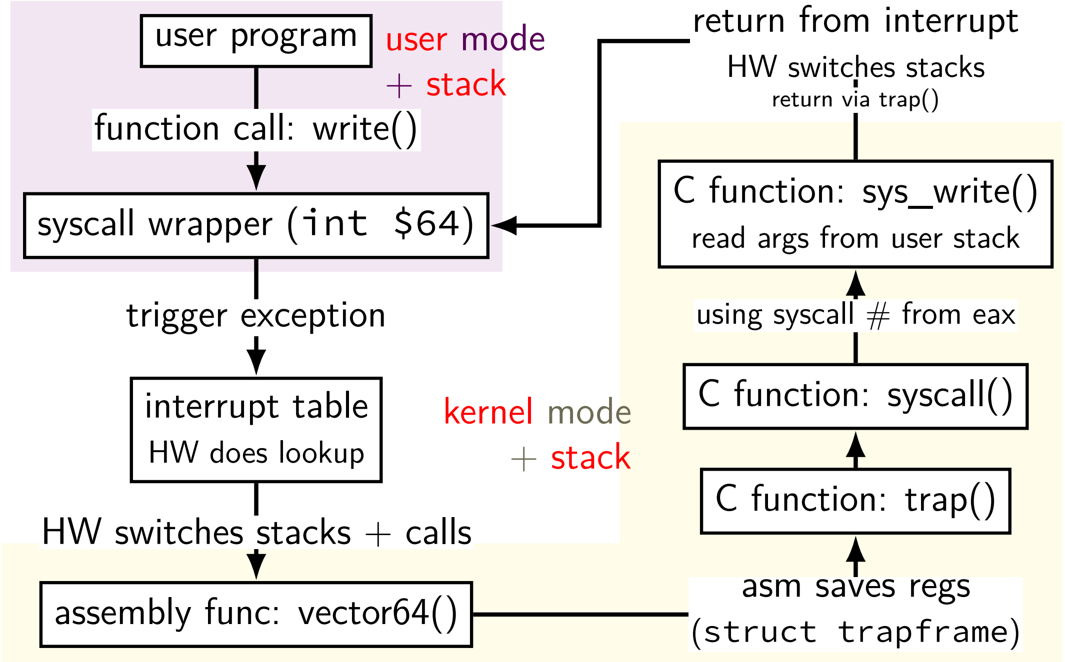
write syscall in xv6 (old)



write syscall in xv6 (old)



write syscall in xv6 (old)



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)
{
...
```

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()`