Changelog

Changes made in this version not seen in first lecture:

30 August: juggling stacks: add arguments to stacks

30 August: where things go in context switch: new slide

this duplicates some notional drawings made in class

30 August: creating a new thread: add slide after showing where in swtch execution starts

30 August: the userspace part?: add slide before showing where save user regs are

Multiprogramming/Dual mode operation

last time

what are OSes

- many views...
- helping out applications
- better abstractions than hardware

process = thread(s) + address space

kernel mode — privileged operations for OS only

exceptions — running OS when needed

- system calls in xv6

The Process

process = thread(s) + address space

illusion of dedicated machine:

- thread = illusion of own CPU
- address space = illusion of own memory

syscalls in xv6

fork, 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 + return value: 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);
...
```

`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`

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

```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

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

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

write syscall in xv6: the syscall function

syscall.c

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

...

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

write syscall in xv6: the syscall function

syscall.c

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

...

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

array of functions — one for syscall

‘`[number] value`’: syscalls[number] = value

write syscall in xv6: the syscall function

syscall.c

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

...

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

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

write syscall in xv6: the syscall function

syscall.c

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

...

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

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

write syscall in xv6: sys_write

sysfile.c

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

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

write syscall in xv6: sys_write

sysfile.c

```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

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: 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()`

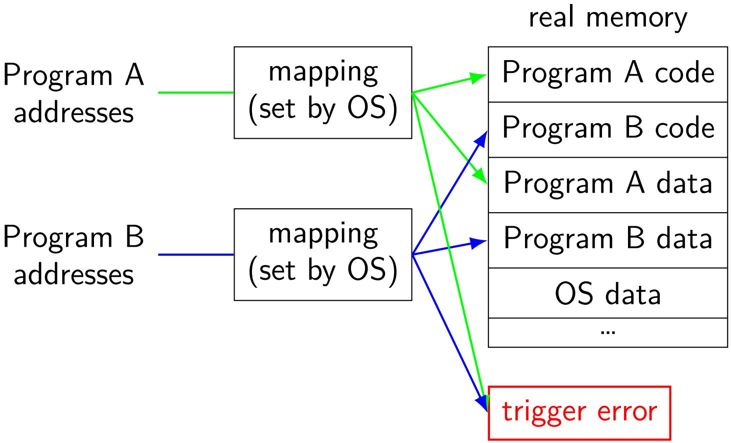
...which checks syscall number (from eax)

...and uses it to call `sys_write`

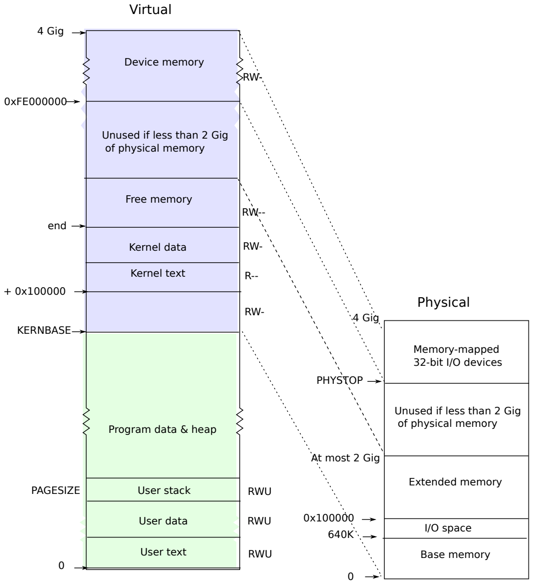
...which reads arguments from the stack and does the write

...then registers restored, return to user space

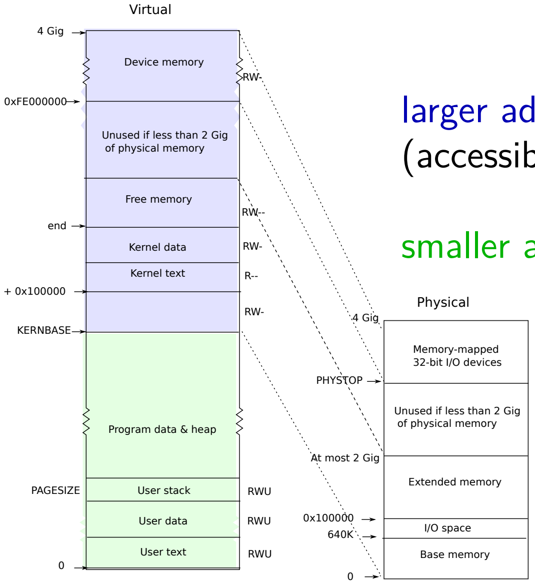
recall: address translation



xv6 memory layout



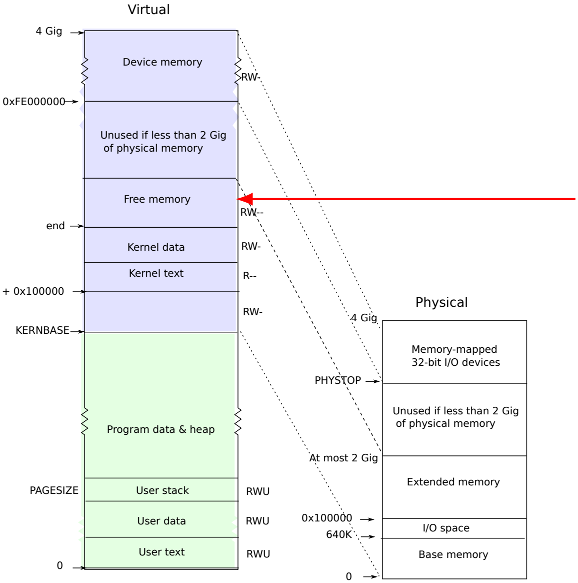
xv6 memory layout



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

smaller addresses are for applications

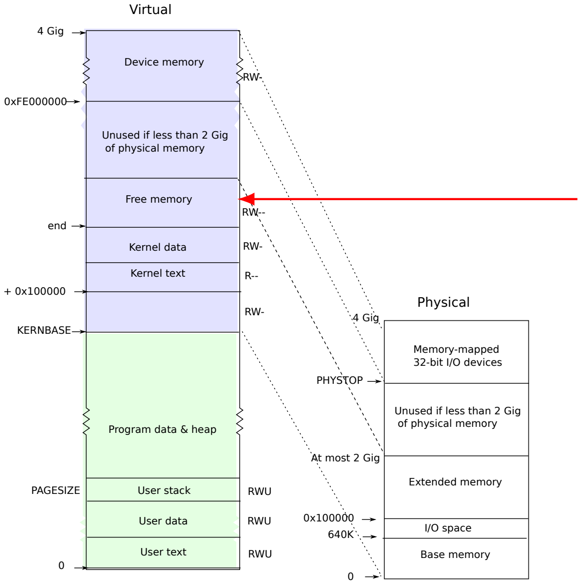
xv6 memory layout



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 allocated here

one kernel stack per process
change which one exceptions use
as part of switching which processes
is active on a processor

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

non-system call exceptions

xv6: there are traps other than system calls `trap()`

timer interrupt — every hardware “tick”

- action: schedule new process

faults — e.g. access invalid memory

I/O — handle I/O

non-system call exceptions

xv6: there are traps other than system calls `trap()`

timer interrupt — every hardware "tick"

- action: schedule new process

faults — e.g. access invalid memory

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

yield — maybe context switch

```
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

wakeup — handle processes waiting a certain amount o

```
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

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();
  ...
}
```

xv6: timer interrupt

```
void
trap(struct trapf
{
  switch(tf->trap
  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();
  ...
}
```

myproc() retrieves running process
check state == RUNNING in case
process was just about to stop running

non-system call exceptions

xv6: there are traps other than system calls `trap()`

timer interrupt — every hardware “tick”

- action: schedule new process

faults — e.g. access invalid memory

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

non-system call exceptions

xv6: there are traps other than system calls `trap()`

timer interrupt — every hardware “tick”

- action: schedule new process

faults — e.g. access invalid memory

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);
  ...
}
```

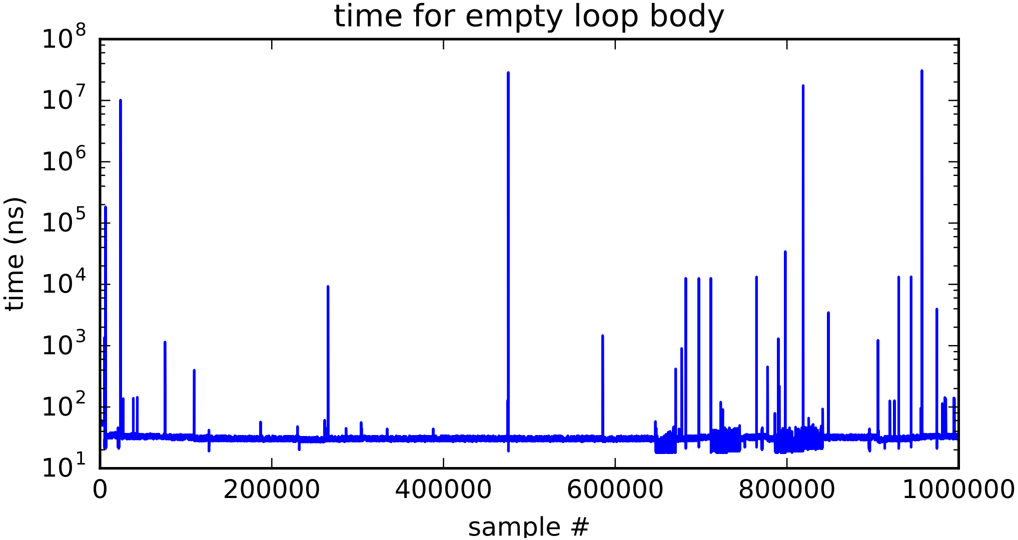
finds process waiting on consle
make it run soon

timing nothing

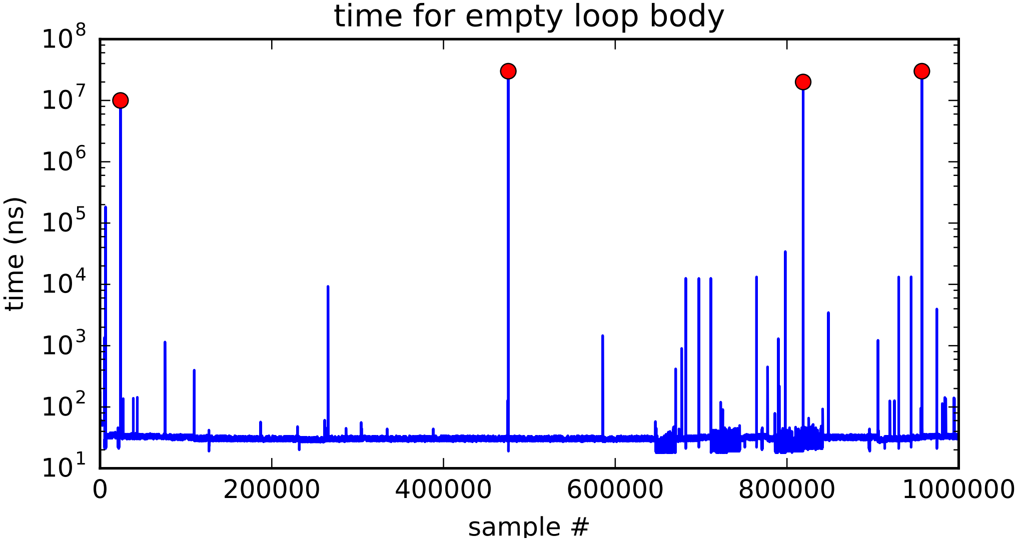
```
long times[NUM_TIMINGS];
int main(void) {
    for (int i = 0; i < N; ++i) {
        long start, end;
        start = get_time();
        /* do nothing */
        end = get_time();
        times[i] = end - start;
    }
    output_timings(times);
}
```

same instructions — same difference each time?

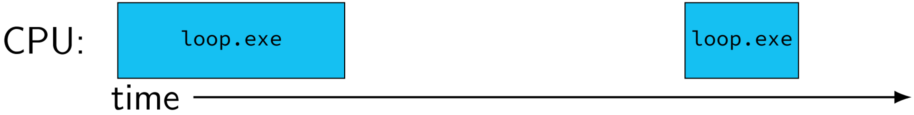
doing nothing on a busy system



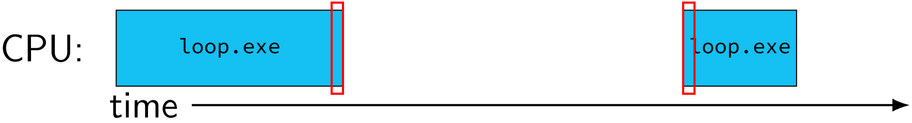
doing nothing on a busy system



time multiplexing



time multiplexing

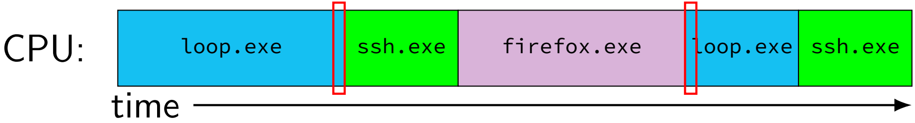


```
...
call get_time
    // whatever get_time does
movq %rax, %rbp
```

million cycle delay

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

time multiplexing

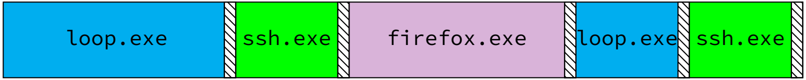


```
...
call get_time
    // whatever get_time does
movq %rax, %rbp
```

——— million cycle delay ———

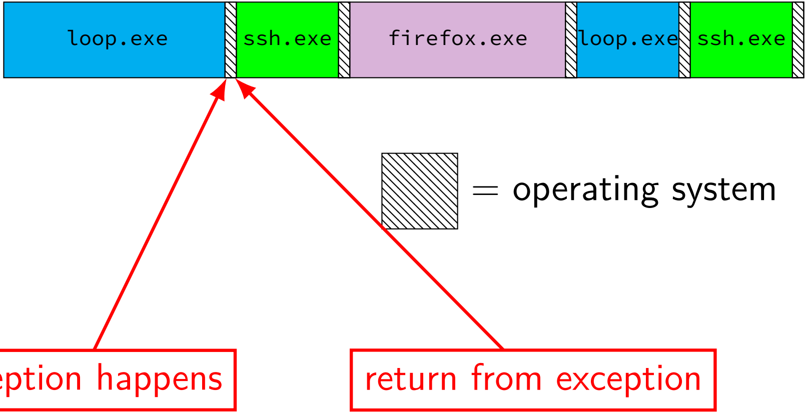
```
call get_time
    // whatever get_time does
subq %rbp, %rax
...
```

time multiplexing really



▨ = operating system

time multiplexing really



OS and time multiplexing

starts running instead of normal program

- mechanism for this: exceptions (later)

saves old program counter, registers somewhere

sets new registers, jumps to new program counter

called context switch

- saved information called context

context

all registers values

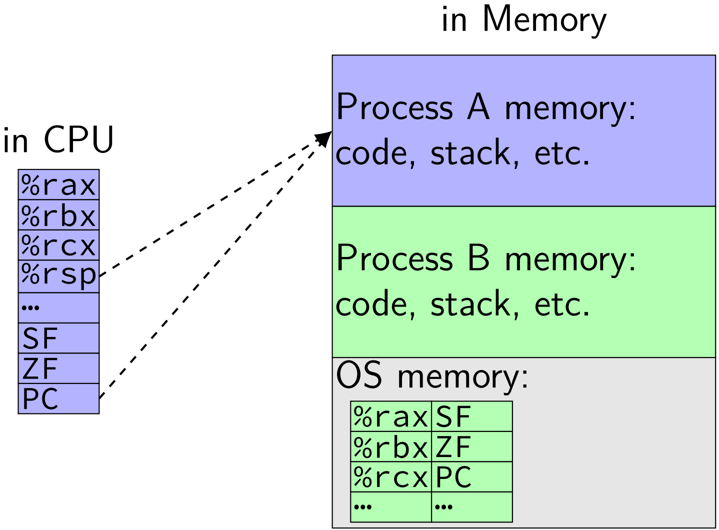
%rax %rbx, ..., %rsp, ...

condition codes

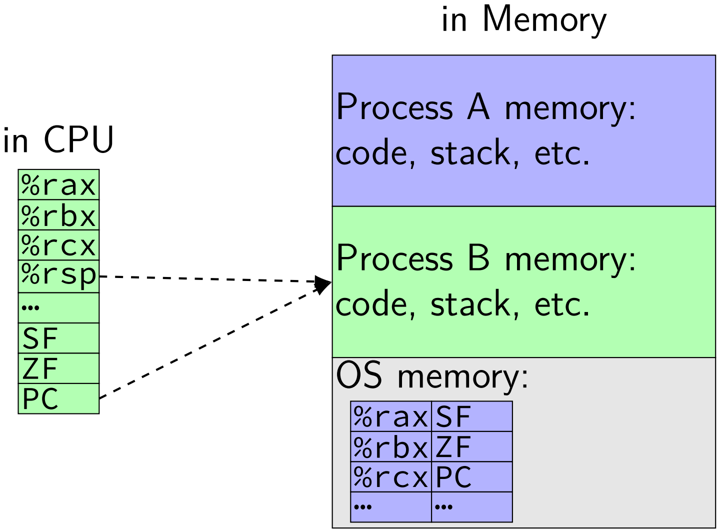
program counter

address space = page table base pointer

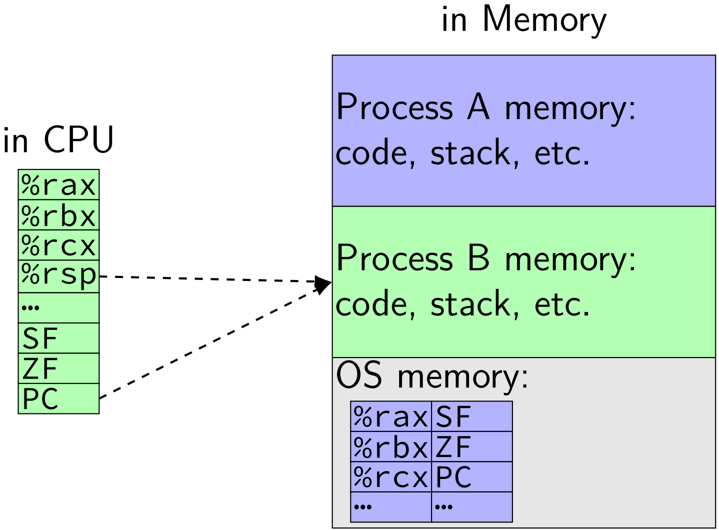
contexts (A running)



contexts (B running)



contexts (B running)



xv6: saved by exception handler into “trapframe” on B’s kernel stack

context switch in xv6

xv6 context switch has two parts

switching threads

switching user address spaces + kernel stack to use for exception

context switch in xv6

xv6 context switch has two parts

switching threads

switching user address spaces + kernel stack to use for exception

context switch in xv6

xv6 context switch has two parts

switching threads

switching user address spaces + kernel stack to use for exception

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:

```
... // (2) [just after another swtch() call?]
...
/* 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:

```
    ... // (2) [just after another swtch() call?]
    ...
    /* 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:

```

    ... // (2) [just after another swtch() call?]
    ...
    /* 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:

```
    ... // (2) [just after another swtch() call?]
    ...
    /* 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

saved: ebp, ebx, esi, edi

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

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

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

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

thread switching in xv6: assembly

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

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

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

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

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

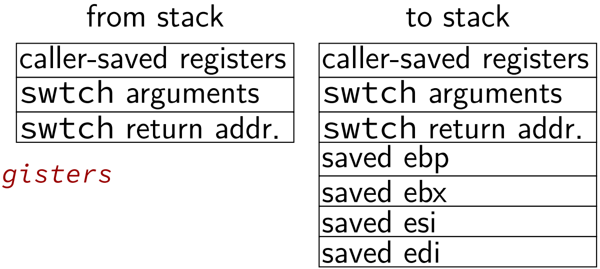
thread switching in xv6: assembly

```
.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

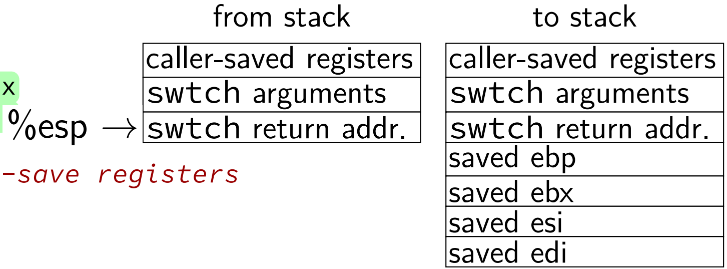
juggling stacks

```
.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
```



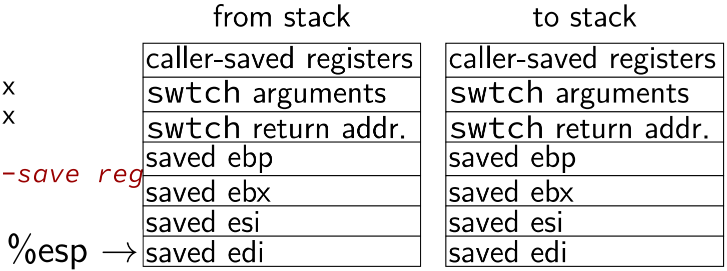
juggling stacks

```
.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
```



juggling stacks

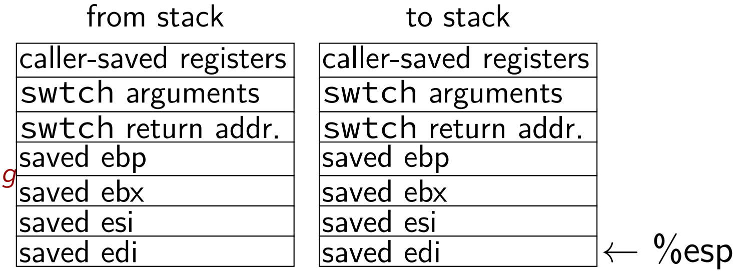
```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # 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

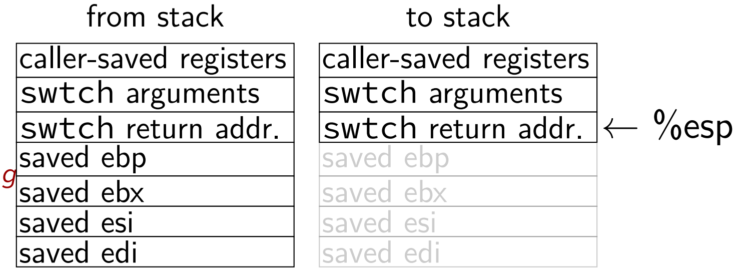
caller-saved registers
swtch arguments
swtch return addr.
saved ebp
saved ebx
saved esi
saved edi

to stack

caller-saved registers
swtch arguments
swtch return addr.
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

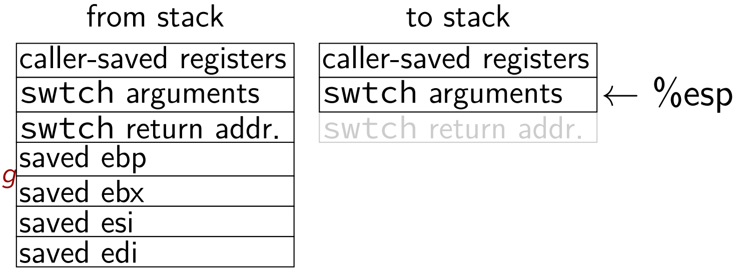
saved ebx

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
```



first call to swtch?

one thread calls swtch and

…return from another thread's call to swtch

what about switching to a new thread?

creating a new thread

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

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

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
  ...
```

`struct proc` ≈ process
`p` is new `struct proc`
`p->kstack` is its new stack
(for the kernel only)

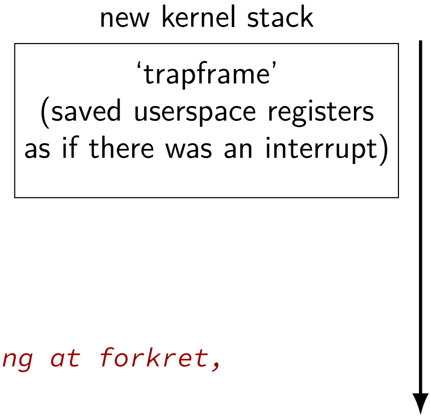
creating a new thread

new kernel stack

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

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

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
  ...
```

creating a new thread

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

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

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
  ...
```



new kernel stack

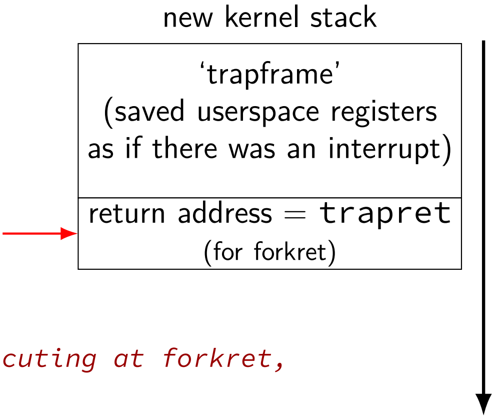
‘trapframe’
(saved userspace registers
as if there was an interrupt)

creating a new thread

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

  // Leave room for trap frame
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
  ...
```

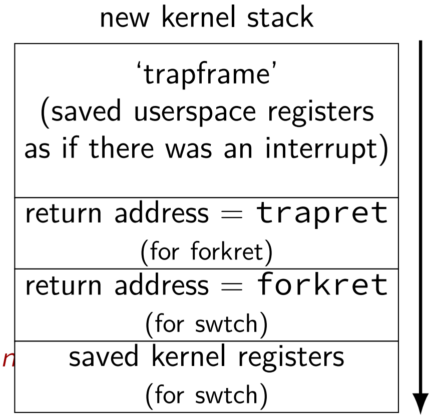
assembly code to return to user mode

same code as for syscall returns



creating a new thread

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

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

  // Set up new context to start executin
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
  ...
```



new kernel stack

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

return address = trapret
(for forkret)

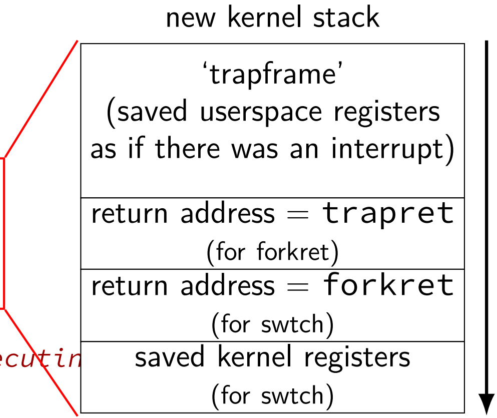
return address = forkret
(for swtch)

saved kernel registers
(for swtch)

creating a new thread

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

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

  // Set up new context to start executin
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
  ...
```

new stack says: this thread is in middle of calling `swtch` in the middle of a system call


new kernel stack

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

return address = `trapret`
(for forkret)

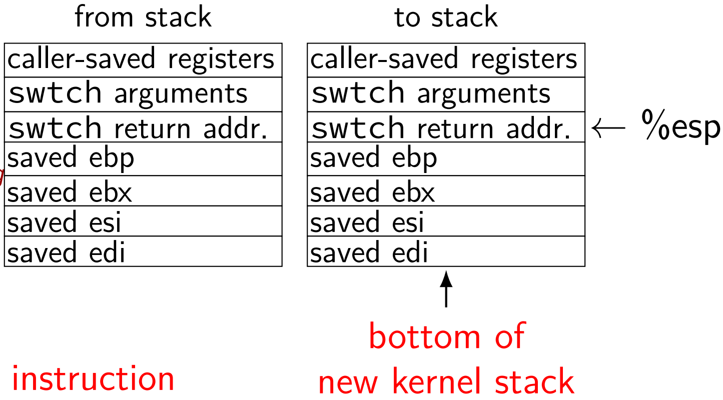
return address = `forkret`
(for swtch)

saved kernel registers
(for swtch)

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. ← %esp
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	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

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

exercise

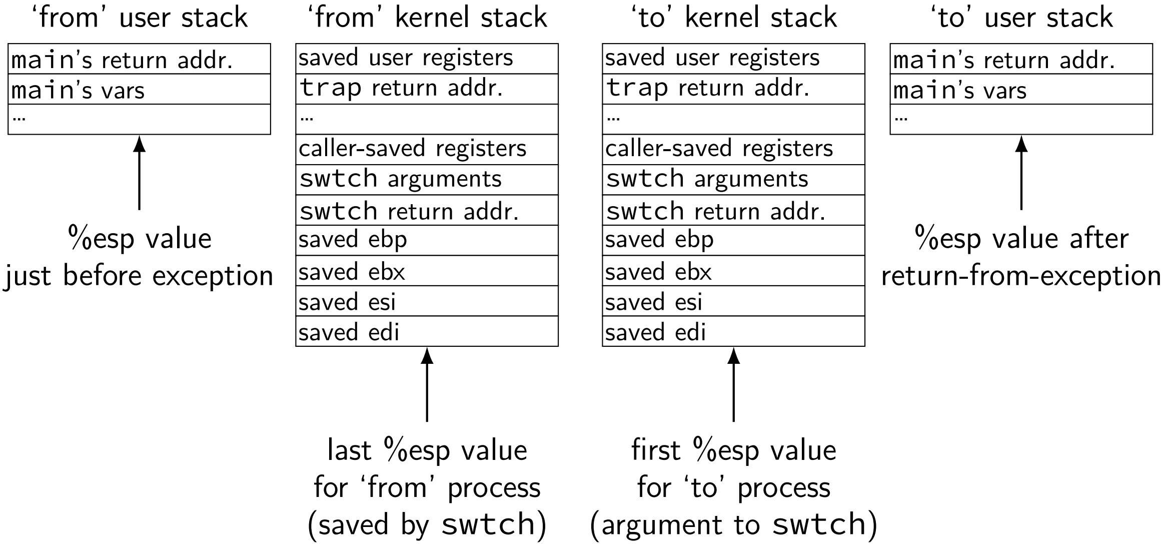
suppose xv6 is running this `loop.exe`:

```
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

where things go in context switch



‘from’ user stack

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

%esp value
just 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

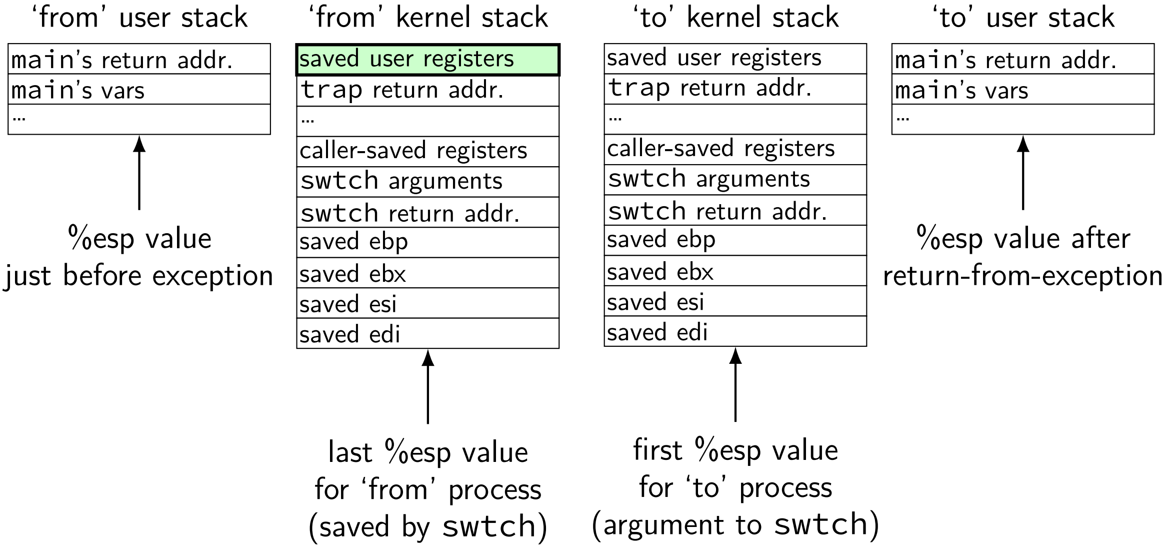
first %esp value
for ‘to’ process
(argument to swtch)

‘to’ user stack

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

%esp value after
return-from-exception

where things go in context switch



‘from’ user stack

main’s return addr.
main’s vars
…

%esp value
just 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
(argument to swtch)

‘to’ user stack

main’s return addr.
main’s vars
…

%esp value after
return-from-exception

exceptions in exceptions

```
alltraps:
  ... /* save registers
         ON KERNEL STACK*/
  pushl %esp
  call trap
      /* in trap(): */
      movl ..., %eax

  ...
  ret
```

current kernel stack

eax from user program
ecx from user program
...

exceptions in exceptions

```
alltraps:
   ... /* save registers
          ON KERNEL STACK*/
   pushl %esp
   call trap
       /* in trap(): */
       movl ..., %eax
```

```
   ...
   ret
```

current kernel stack

~~eax from user program~~ eax from `trap()`
~~ecx from user program~~ ecx from `trap()`
...

```
alltraps:  /* run a second time?? */
   ... /* setup registers on
          SAME KERNEL STACK */
   pushl %esp
   call trap
```

exceptions in exceptions

```
alltraps:
  ... /* save registers
         ON KERNEL STACK*/
  pushl %esp
  call trap
    /* in trap(): */
    movl ..., %eax
```

```
  ...
  ret
```

current kernel stack

~~eax from user program~~ eax from trap()
~~ecx from user program~~ ecx from trap()
…

```
alltraps:  /* run a second time?? */
  ... /* setup registers on
         SAME KERNEL STACK */
  pushl %esp
  call trap
```

solution: disallow this!

interrupt disabling

CPU supports disabling (most) interrupts

interrupts will wait until it is reenabled

CPU has extra state: are interrupts enabled?

xv6 interrupt disabling

xv6 policy: interrupts are usually disabled when kernel

xv6 interrupt disabling

xv6 policy: interrupts are usually disabled when kernel

this policy makes xv6 easier to code...

disadvantages?

xv6 interrupt disabling

xv6 policy: interrupts are usually disabled when kernel

this policy makes xv6 easier to code...

disadvantages?

- slow kernel code makes system hang?
- gaurenteeing minimum reaction time to keypress?