CS 347M (Operating Systems Minor)

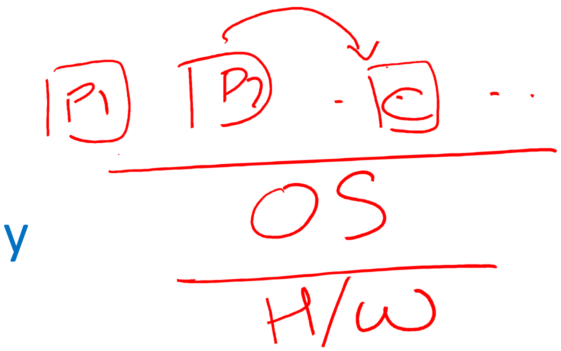
Spring 2022

Lecture 4: Kernel mode execution

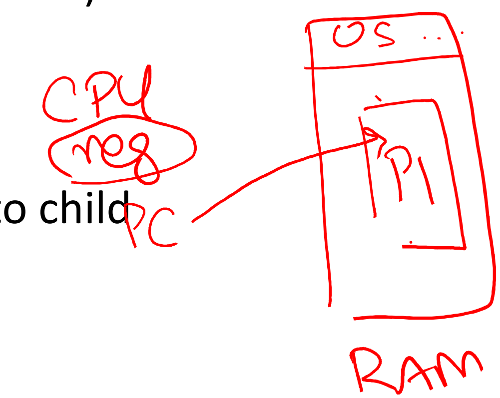
Mythili Vutukuru

CSE, IIT Bombay

Recap: OS runs processes



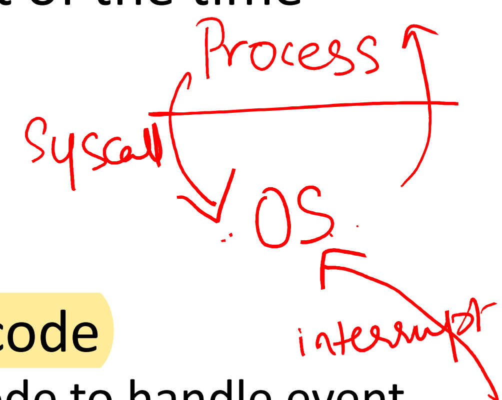
- OS manages multiple active processes concurrently
  - Information on process in PCB (process control block)
- What is a process?
  - Memory image in RAM = compiled code, data (compile-time, run-time)
  - CPU context (in CPU registers when running, else saved in PCB)
  - Other things like I/O connections, ..
- Process created by fork from parent process
  - OS adds new process PCB to list, copies memory image of parent to child



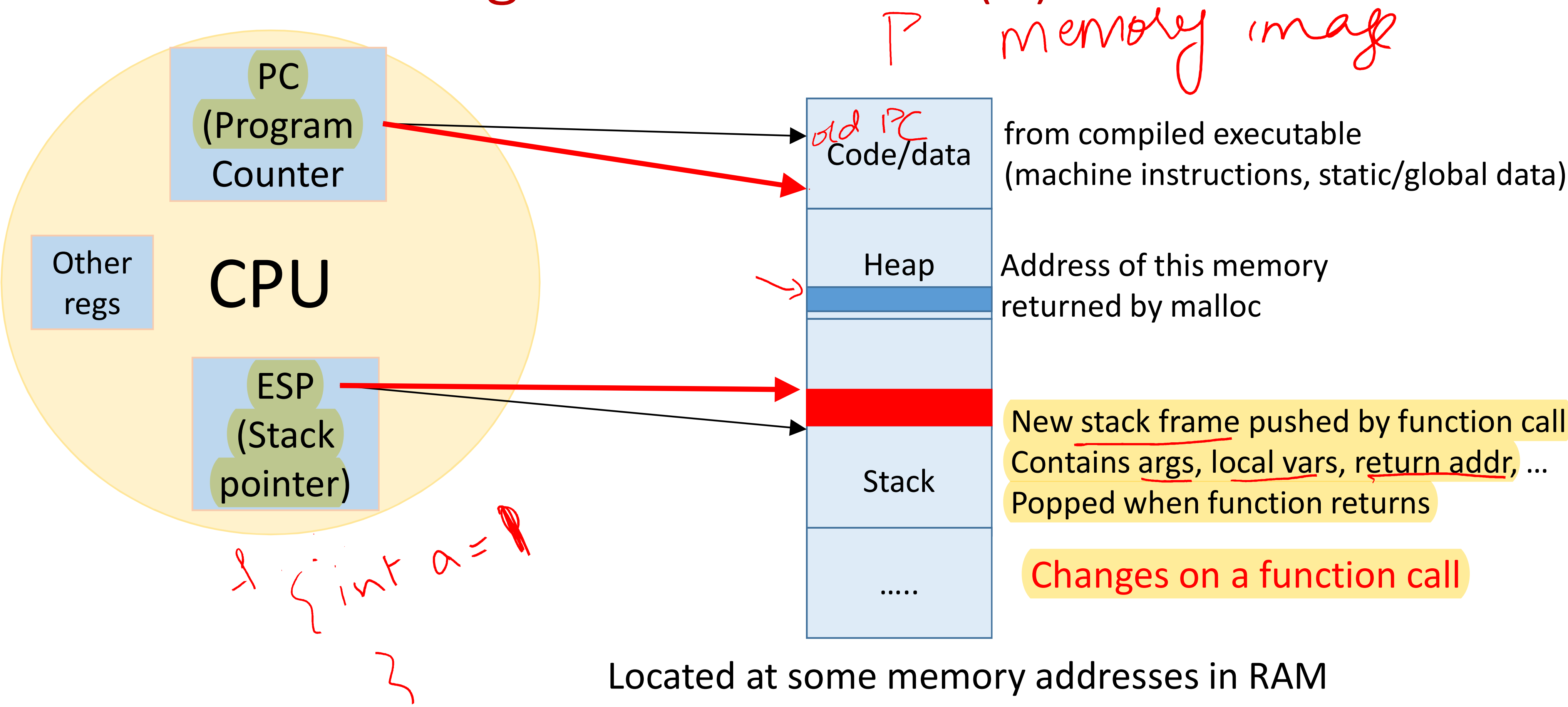
- Periodically, OS scheduler loops over ready processes
  - Finds a suitable process to run next
  - Saves context of existing process, restores context of new process
- Once process is context switched in, OS is out of picture, CPU in user mode, runs user code directly
  - When does the OS run again?

User mode vs. Kernel mode of a process

- CPU runs user code in user mode (low privilege) most of the time
- CPU switches to kernel mode execution when
  - Process makes system call, needs OS services
  - External device needs attention, raises interrupt
  - Some fault has happened during program execution
- All such events are called traps: CPU "traps" into OS code
  - CPU shifts to high privilege level (kernel mode), runs OS code to handle event
  - Later, CPU switches to low privilege level, back to user code in user mode
- Process P goes to kernel mode to run OS code, but it is still process P itself that is in running state
- OS not a separate process, runs in kernel mode of existing processes



Understanding a function call (1)



Located at some memory addresses in RAM

reg

PC 100

200 f {

Understanding a function call (2)

- What happens when a user program makes a function call?
  - Allocate memory on user stack for function arguments, local variables, ..
  - Push return address (PC where execution stopped), PC jumps to function code
  - Push register context (to resume execution when function returns)
  - Execute function
  - When returning from function, pop return address, pop register context

compiler

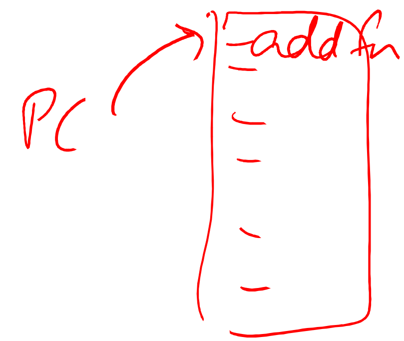
}

old PC

- System call also must
  - Use a stack to push/pop register context
  - Save old PC, change PC to point to OS code to handle system call

privlege

System call vs. function call



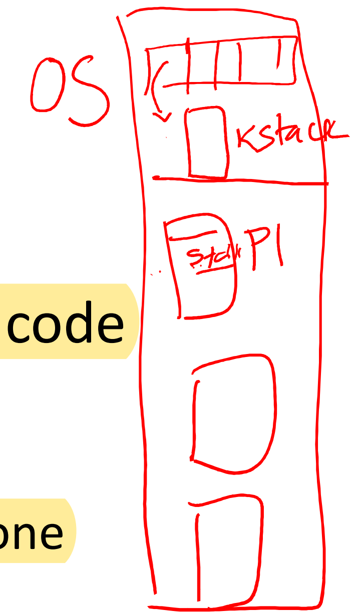
- Changing PC in function call vs. system call
  - In function call, address of function code known in executable, can jump to function code directly using a CPU instruction ("call" in x86) call add fn
  - For system call, cannot trust user to jump to correct OS code (what if user jumps to inappropriate privileged code?)
- Saving register context on stack in function call vs. system call
  - In function call, register context is saved and restored from user stack
  - For system call, OS does not wish to use user stack (what if user has setup malicious values on the stack?)
- We require: a secure stack, a secure way of jumping to OS code

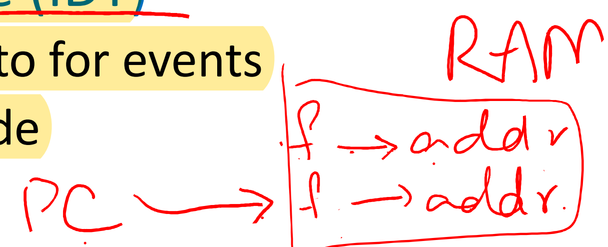
user stack

kernel stack

Kernel stack and IDT

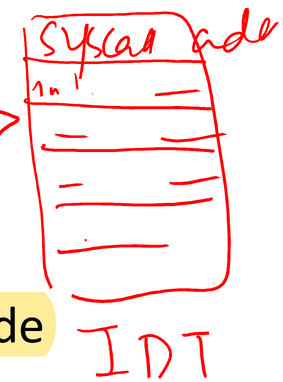
- Every process uses a separate kernel stack for running kernel code
  - Part of PCB of process, in OS memory, not accessible in user mode
  - Used like user stack, but for kernel mode execution
  - Context pushed on kernel stack during system call, popped when done
- To set PC, CPU accesses Interrupt Descriptor Table (IDT)
  - Data structure with addresses of kernel code to jump to for events
  - Setup by OS during bootup, not accessible in user mode
  - CPU uses IDT to locate address of OS code to jump to
- Together: secure way of locating OS code, secure stack for OS to run





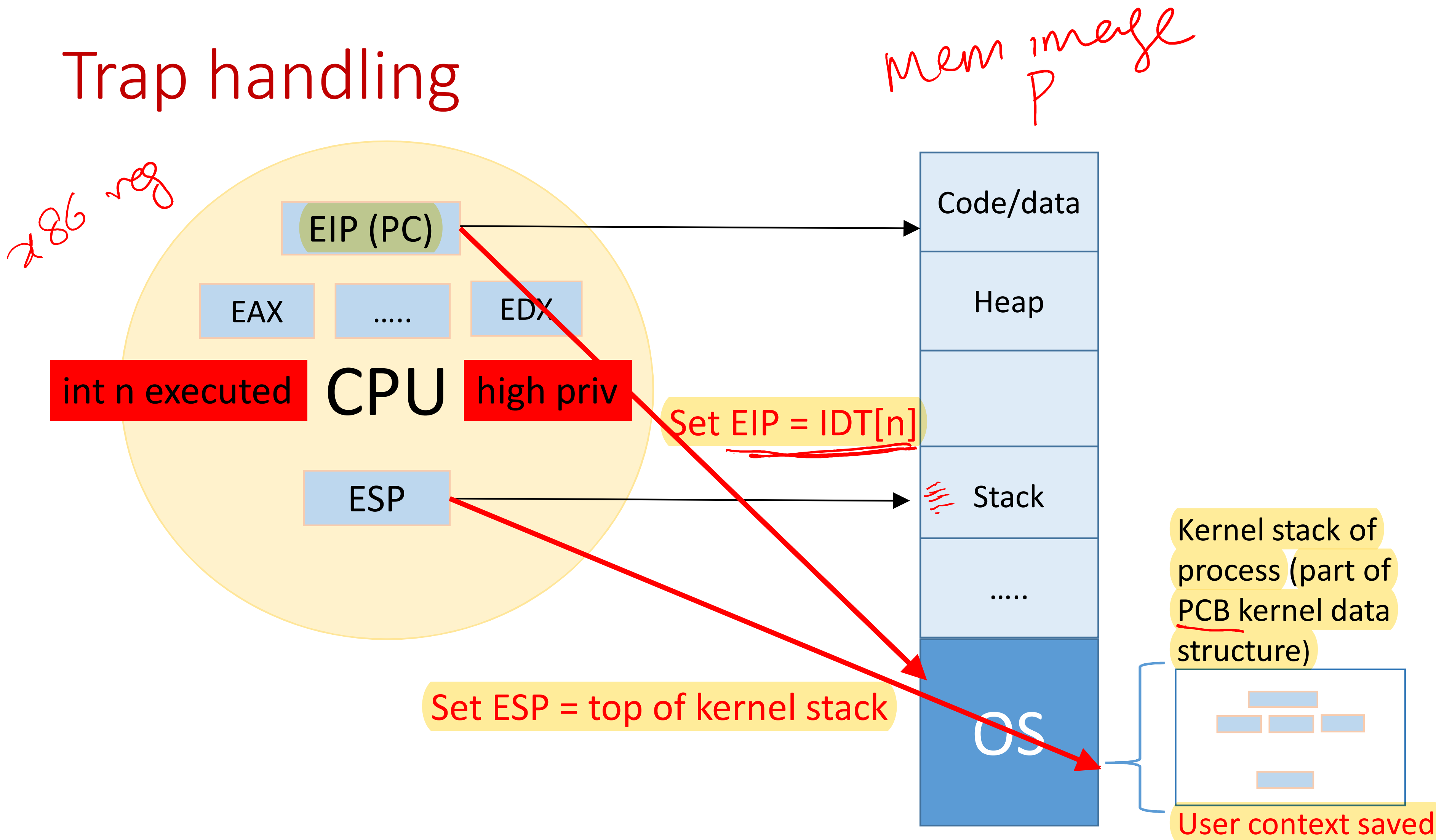
Hardware trap instruction

- When user code wants to make system call, it invokes special "trap instruction" with an argument
  - Example: "int n" in x86, argument "n" indicates type of trap (syscall, interrupt)
  - The value of "n" specifies index into IDT array, which OS function to jump to
- When CPU runs the trap instruction:
  - CPU moves to higher privilege level
  - CPU shifts stack pointer register to kernel stack of process
  - Register context is saved on kernel stack (part of PCB)
  - Address of OS code to jump to is obtained from IDT, PC points to OS code
  - OS code starts to run, on a secure stack



add
int
OS

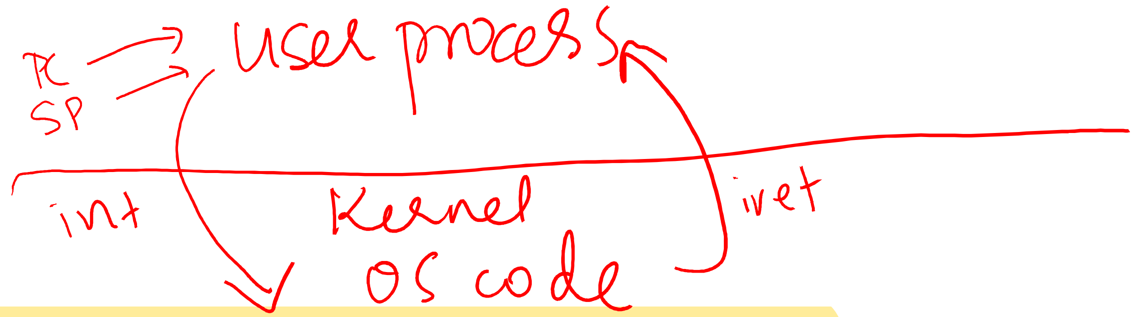
Trap handling



Why trap instruction?

- Need a secure way of jumping to OS code to handle traps
  - User code cannot be trusted to jump to correct OS code
  - Only CPU can be trusted to handover control from user to OS securely
    trust
- Who calls trap instruction?
  - System call code in a language library (printf invokes system call via int n)
  - External hardware raises interrupt, causes CPU to execute "int n"
  - Argument "n" indicates whether system call /IRQ number of hardware device
- Across all cases, the mechanism is: save context on kernel stack, switch to OS address in IDT, run OS code to handle trap

Return from trap



- When OS is done handling syscall or interrupt, it calls a special instruction return-from-trap
  - Restore context of CPU registers from kernel stack
  - Change CPU privilege from kernel mode to user mode
  - Restore PC and jump to user code after trap
- User process unaware that it was suspended, resumes execution at the point it stopped before
- Always return to the same user process from kernel mode? No
  - Before returning to user mode, OS checks if it must switch to another process

Why switch between processes?

user $\rightarrow$ kernel $\rightarrow$ user

- Sometimes when OS is in kernel mode, it cannot return back to the same process that was running in user mode before
  - Process has exited or must be terminated (e.g., segfault)
  - Process has made a blocking system call
- Sometimes, the OS does not want to return back to the same process
  - The process has run for too long
  - Must timeshare CPU with other processes
- In such cases, OS performs a context switch from one process to another

The OS scheduler

P1

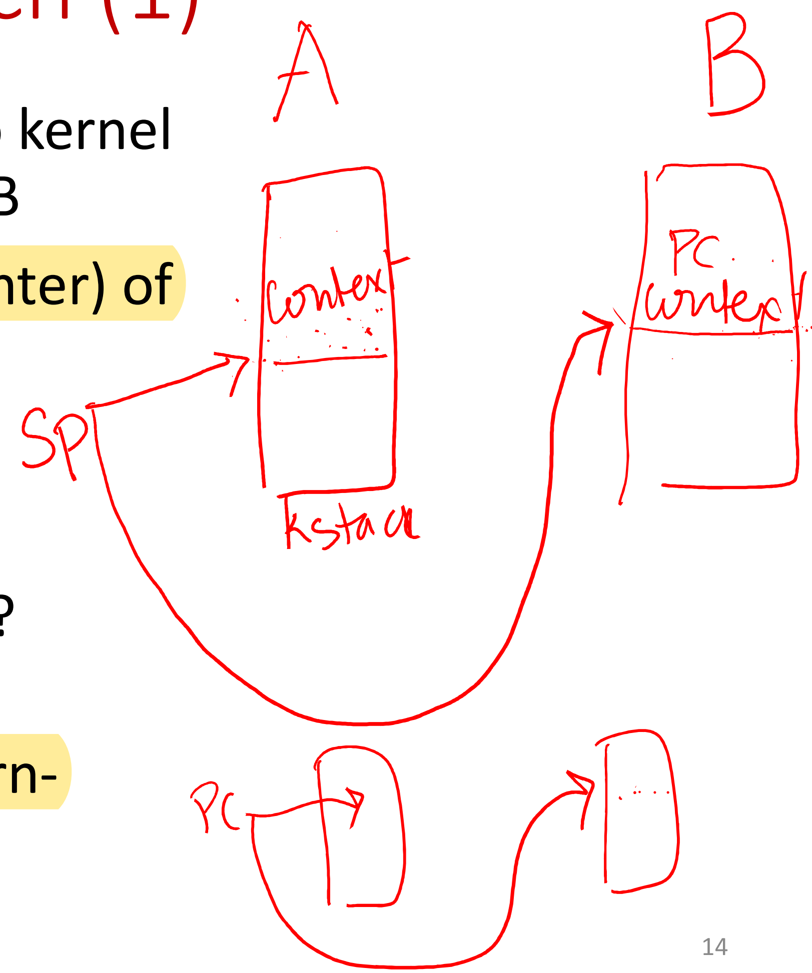
P2

kernel mode → OS scheduler

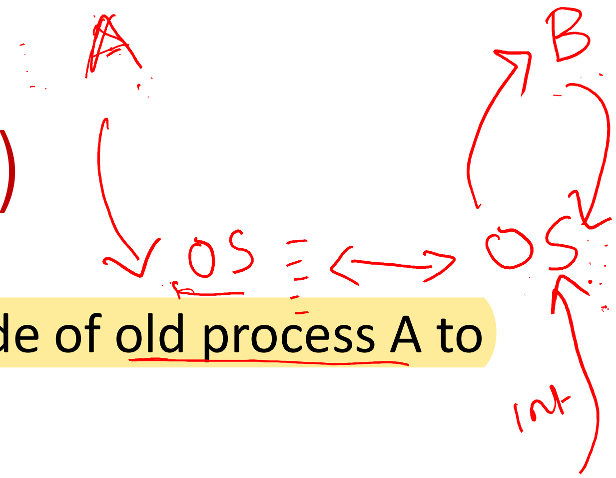
- OS scheduler has two parts
  - Policy to pick which process to run (next lecture)
  - Mechanism to switch to that process (this lecture)
- Non preemptive (cooperative) schedulers are polite
  - Switch only if process blocked or terminated
- Preemptive (non-cooperative) schedulers can switch even when process is ready to continue
  - CPU generates periodic timer interrupt
  - After servicing interrupt, OS checks if the current process has run for too long

Mechanism of context switch (1)

- Example: process A has moved from user to kernel mode, OS decides it must switch from A to B
- Save context (PC, registers, kernel stack pointer) of A on kernel stack
- Switch SP to kernel stack of B
- Restore context from B's kernel stack
- Who has saved registers on B's kernel stack?
  - OS did, when it switched out B in the past
- Now, CPU is running B in kernel mode, return-from-trap to switch to user mode of B



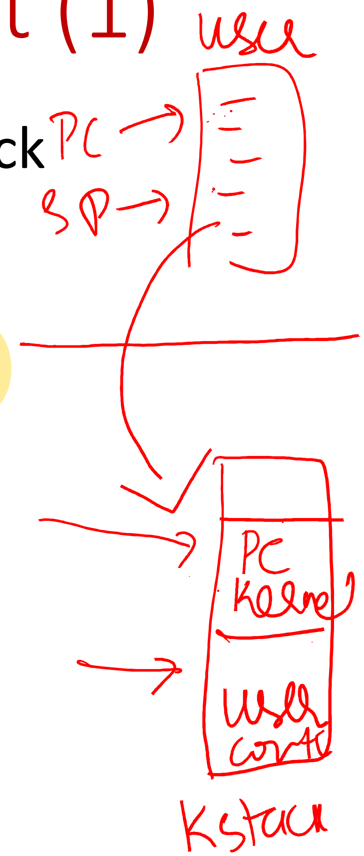
Mechanism of context switch (2)

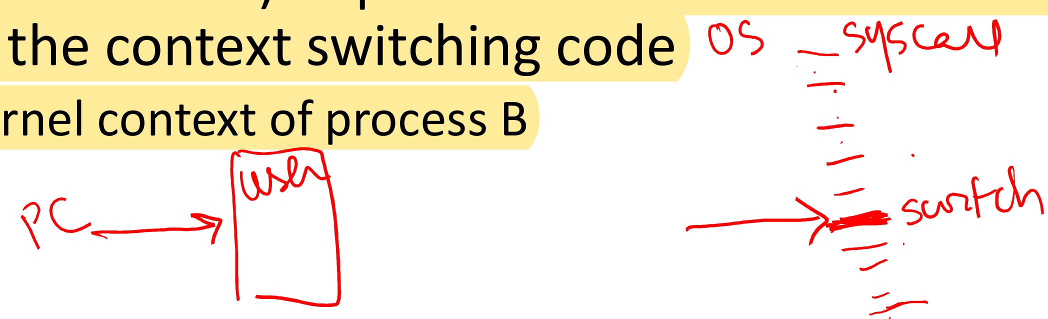


- Context switch: switch CPU context from kernel mode of old process A to kernel mode of new process B
- Before context switch
  - A entered kernel mode, OS decides not to run A anymore (e.g., blocking system call)
  - CPU registers have context of A (stack pointer is pointing kernel stack of A, PC is pointing to some OS code being run by A)
- Mechanism of context switch
  - Save CPU context of A into kernel stack/PCB of A
  - Load CPU context from kernel stack/PCB of B into CPU registers
- After context switch
  - B resumes execution in kernel mode, stack pointer points to B's kernel stack
  - Where does B begin execution? At some point in the past, B went into kernel mode, and was switched out by OS. B resumes execution in same place.

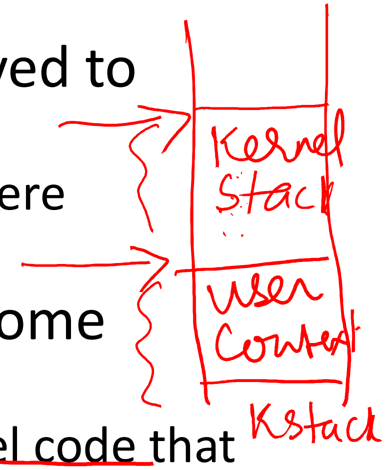
Understand saving and restoring context (1)

- Context (PC and other CPU registers) saved on the kernel stack in two different scenarios
- When going from user mode to kernel mode, user context (e.g., which instruction of user code you stopped at) is saved on kernel stack by the trap instruction
  - Restored by return-from-trap
- During a context switch, kernel context (e.g., where you stopped in the OS code) of process A is saved on the kernel stack of A by the context switching code
  - Restores kernel context of process B





Understand saving and restoring context (2)



- Suppose process A has made a blocking system call and moved to kernel mode
  - Context of user mode execution (e.g., PC pointing to user code where execution stopped) is saved on kernel stack/PCB
- After handling system call, OS decides to context switch to some other process, since A cannot continue now
  - Again, context of kernel mode execution (e.g., PC pointing to kernel code that has handled system call) is saved in kernel stack/PCB
- When A becomes ready and is run by scheduler again in future
  - Kernel context is restored from kernel stack/PCB into CPU registers, CPU resumes running in kernel mode of A
- A returns from trap into user mode
  - User context is restored, CPU resumes running user code of A

Trap handling in xv6

- The following events cause a user process to "trap" into the kernel (xv6 refers to all these events as traps)
  - System calls (requests by user for OS services)
  - Interrupts (external device wants attention)
  - Program fault (illegal action by program)
- When above events happen, CPU executes the special "int" instruction
  - Example seen in usys.S, "int" invoked to handle system calls
  - For hardware interrupts, device sends a signal to CPU, and CPU executes int
- Trap instruction has a parameter (int n), indicating type of interrupt
  - E.g., syscall has a different value of n from keyboard interrupt
  - The value of "n" is used to index into IDT, get address of kernel code to run

PC → user

int

IDT → OS code

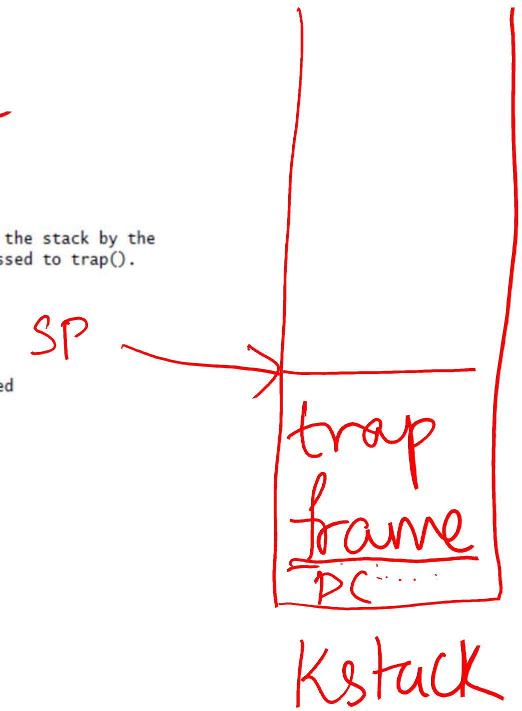
Trap frame on kernel stack

- Trap frame: state is pushed on kernel stack during trap handling
  - CPU context of where execution stopped is saved, so that it can be resumed after trap
  - Some extra information needed by trap handler is also saved
- The "int n" instruction pushes a few entries (old PC, old SP etc.) and jumps to kernel code to handle trap
- The kernel code that is run next will push remaining registers on kernel stack, and then proceed to handle the trap

```
// Layout of the trap frame built on the stack by the
// hardware and by trapasm.S, and passed to trap().
struct trapframe {
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  ushort gs;
  ushort padding1;
  ushort fs;
  ushort padding2;
  ushort es;
  ushort padding3;
  ushort ds;
  ushort padding4;
  uint trapno;

  // below here defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;
  ushort padding5;
  uint eflags;

  // below here only when crossing rings, such as from user to kernel
  uint esp;
  ushort ss;
  ushort padding6;
};
```

PC

SP



Trap handler function in xv6 (1)

syscall
Kbd
disk

- C trap handler performs different actions based on kind of trap
- If system call, "int n" is invoked with "n" equal to a value T_SYSCALL (in usys.S), indicating this trap is a system call
- Trap handler invokes common system call function
  - Looks at system call number stored in eax (whether fork or exec or ….) and calls the corresponding function
  - Return value of syscall stored in eax

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

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

Trap handler function in xv6 (2)

- If interrupt from a device, corresponding device-related code is called
  - The trap number (value of “n” in “int n”) is different for different devices
- Timer is special hardware interrupt, and is generated periodically to trap to kernel

```
switch(tf->trapno){
case T_IRQ0 + IRQ_TIMER:
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;
    wakeup(&ticks);
    release(&tickslock);
  }
  lapiceoi();
  break;
case T_IRQ0 + IRQ_IDE:
  ideintr();
  lapiceoi();
  break;
case T_IRQ0 + IRQ_IDE+1:
  // Bochs generates spurious IDE1 interrupts.
  break;
case T_IRQ0 + IRQ_KBD:
  kbdintr();
  lapiceoi();
  break;
```

Trap handler function in xv6 (3)

- On timer interrupt, a process “yields” CPU to scheduler
  - Ensures a process does not run for too long

```
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();

```

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

A → Scheduler → B

Context switching in xv6 (1)

- Every CPU has a scheduler thread (special process that runs scheduler code)
- Scheduler goes over list of processes and switches to one of the runnable ones
- The special function "swtch" performs the actual context switch
  - Save context on kernel stack of old process
  - Restore context from kernel stack of new process

switch kstack

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&ptable.lock);

  }
}
```

P2 Scheduler P1

Context switching in xv6 (2)

- After running for some time, the process switches back to the scheduler thread, when:
  - Process has terminated (exit system call)
  - Process needs to sleep (e.g., blocking read system call)
  - Process yields after running for long (timer interrupt)
- Process calls "sched" which calls "swtch" to switch to scheduler thread again
- Scheduler thread runs its loop and picks next process to run, and the story repeats

```
  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

```
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;

  sched();

```

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

sched() swtch

Context structure vs. trap frame in xv6

P → C

kstack

- Struct proc stores two different structures on kernel stack
  - Trapframe is saved when CPU switches to kernel mode (e.g., PC in trapframe is PC where syscall was made in user code)
  - Context structure is saved when process switches to another process (e.g., PC value when context switch is performed)
  - Both reside on kernel stack, struct proc has pointers to both
  - Example: Process has timer interrupt, saves trapframe on kstack, then context switch, saves context structure on kstack

kstack

context

trapframe

PCB

struct proc

```
int pid;                    // Process ID
struct proc *parent;        // Parent process
struct trapframe *tf;       // Trap frame for current syscall
struct context *context;    // swtch() here to run process
```

u→k