Trap handling

Mythili Vutukuru

CSE, IIT Bombay

Recap: OS runs processes

- OS manages multiple active processes concurrently
- 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, ..
- Processes created by fork from parent processes
- Periodically, OS scheduler loops over ready processes
  - Find a suitable process to run, save old process context, restore new context
- 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

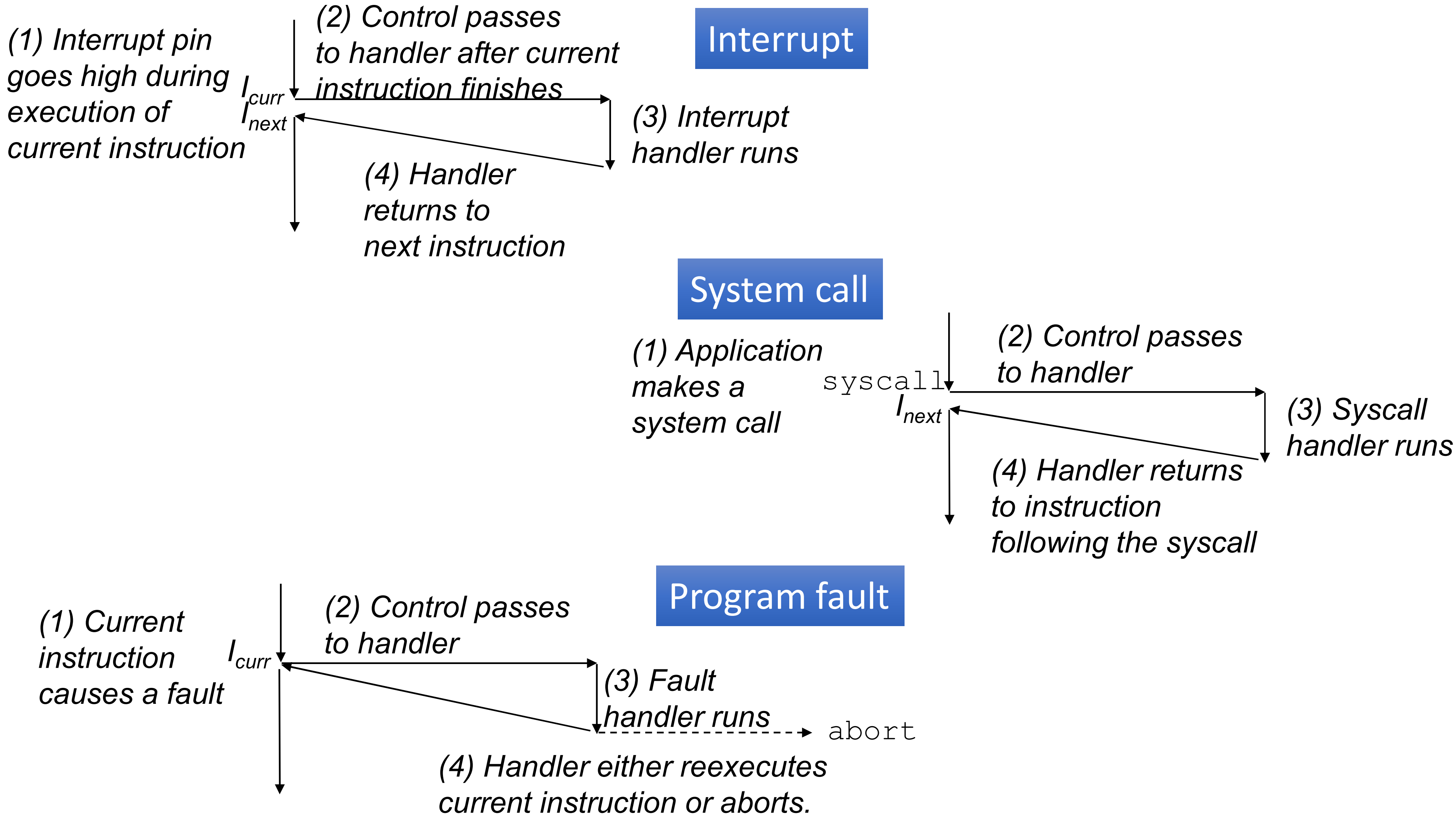
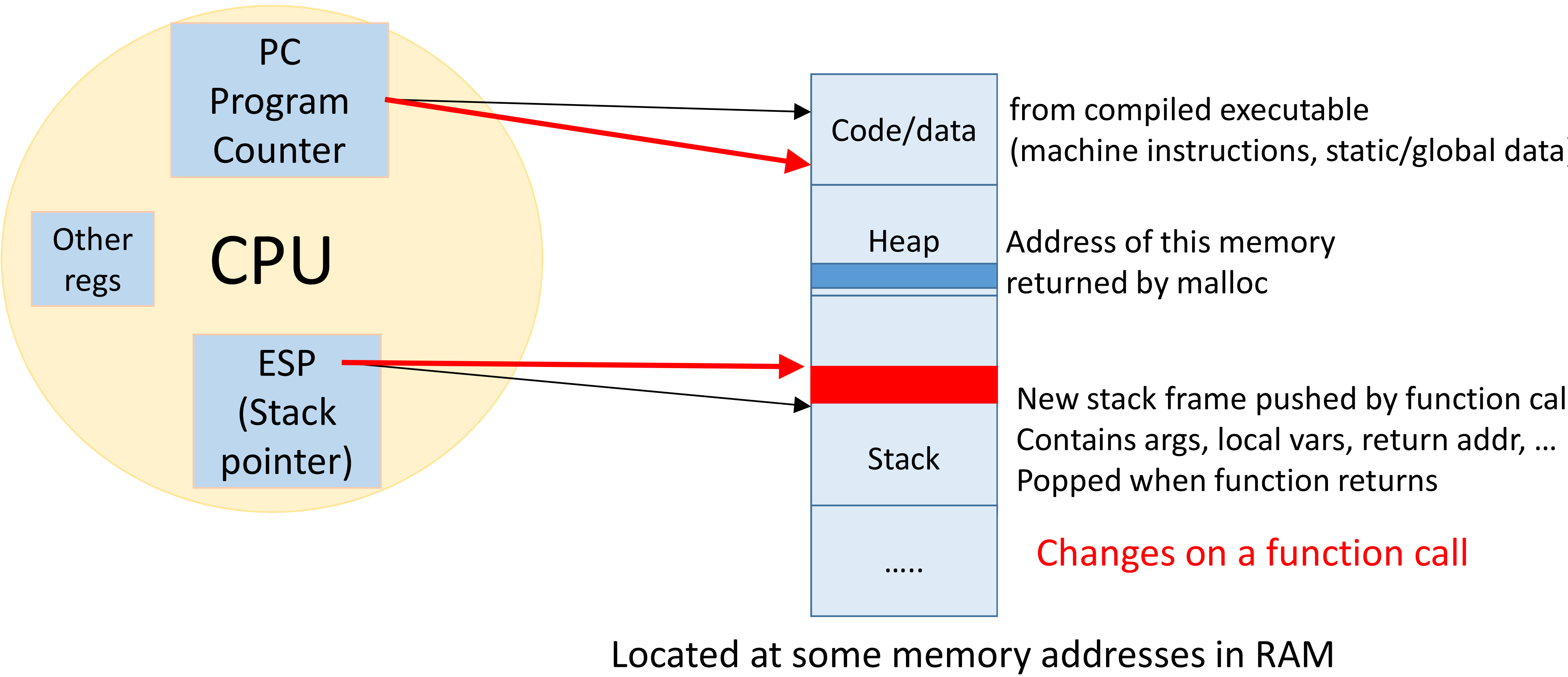


Image credit: CSAPP

Function call vs. system call

- What happens when a user program makes a function call?
  - Allocate memory on user stack for function arguments, local variables, ..
  - Push return address, PC jumps to function code
  - Push register context (to resume execution when function returns)
  - Execute function code
  - When returning from function, pop return address, pop register context
- System call also must
  - Use a stack to push register context
  - Save old PC, change PC to point to OS code to handle system call
  - Run system call, restore context back to user code

Understanding a function call



What is different for a system 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)
  - 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

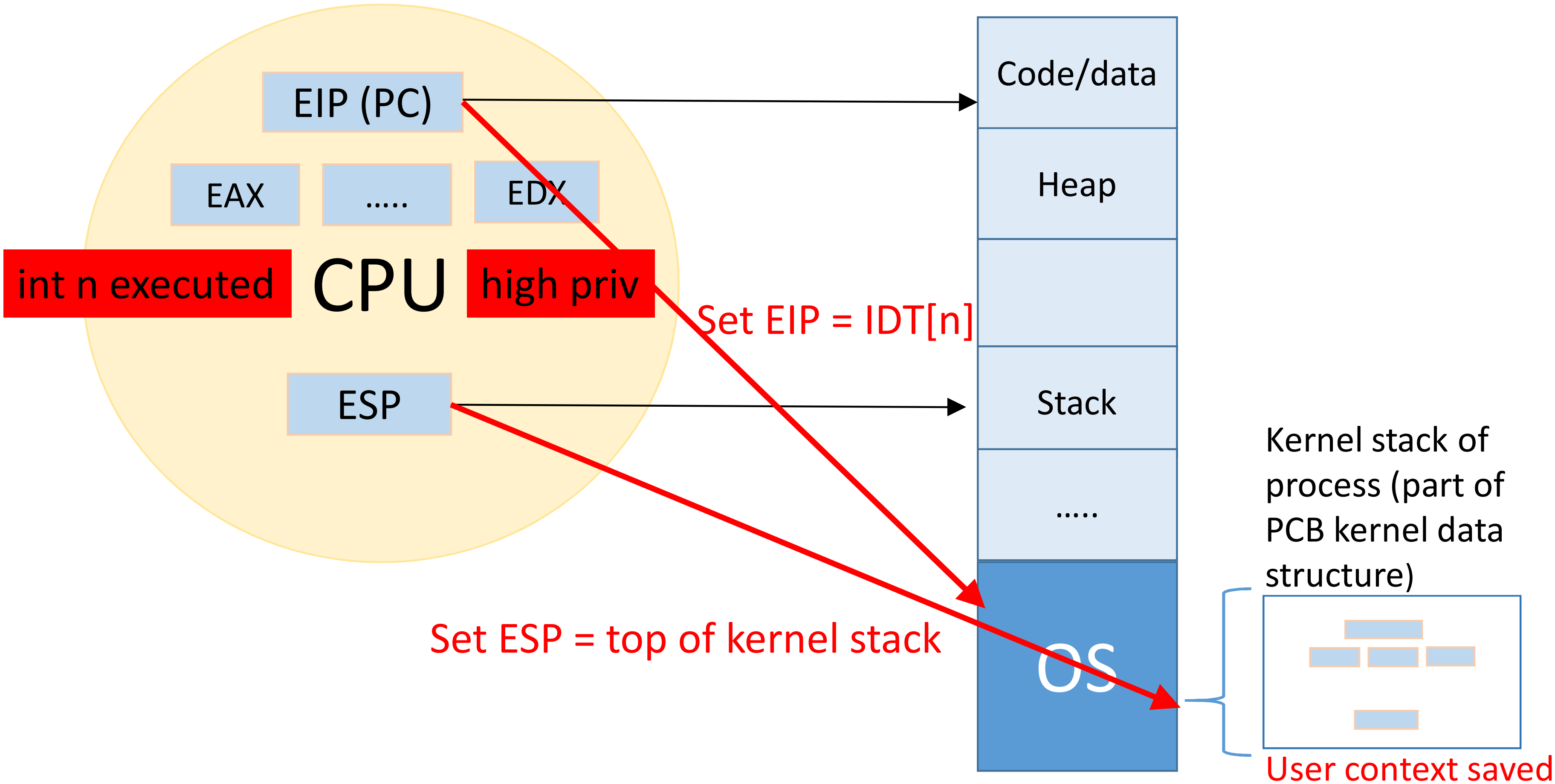
Kernel stack and IDT

- 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

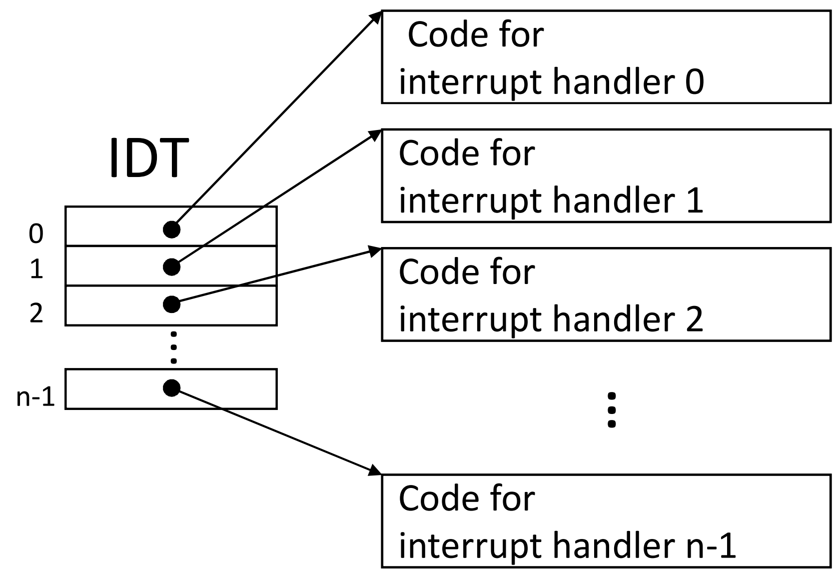
- When user code wants to make system call, it invokes special "trap instruction" with an argument
  - Example: "int n" in x86, argument "n" indicates type of trap (syscall, interrupt)
  - The value of "n" specifies index into IDT array, which OS function to jump to
- When CPU runs the trap instruction:
  - 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

Trap handling



IDT lookup

- IDT configured by OS
- Base address of IDT stored in CPU register
- Upon trap, CPU looks up IDT to find address of interrupt handler



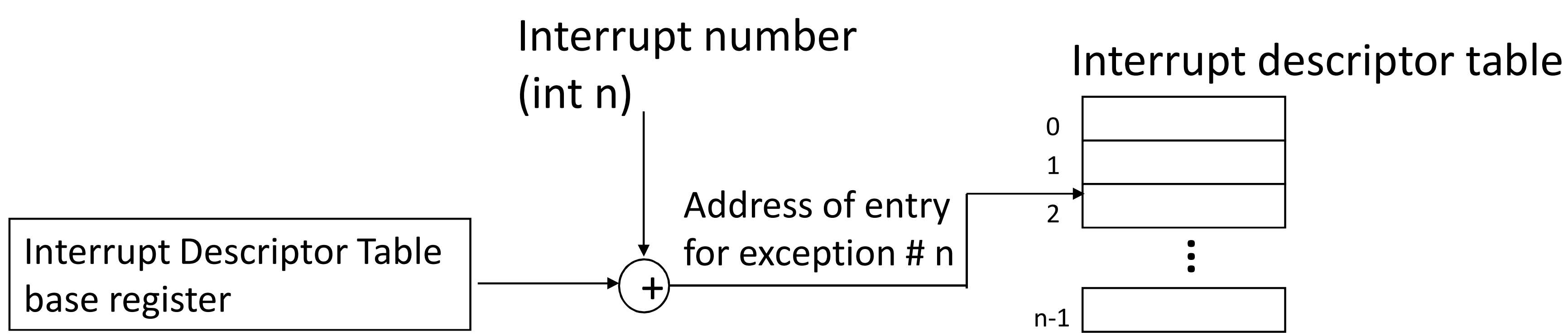


Image credit: CSAPP

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
- 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

Trap handling in xv6

- The following events in xv6 cause a user process to “trap” into the kernel
  - 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
- xv6 trap handling code saves register context, handles trap, returns

xv6 system calls

- In xv6, system calls available to user programs are defined in user library header “user.h”
  - Equivalent to C library headers (xv6 doesn’t use standard C library)
- These system call functions invoked in user programs after including “user.h”
- The actual invoking of system call is done in usys.S

```c
struct stat;
struct rtcdate;

// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
```

xv6 system calls

- The user library makes the actual system call to invoke OS code
- User library invokes trap instruction to make system call, code seen in usys.S
  - Defined using a macro
  - Move system call number to eax
  - Invoke int n where n is T_SYSCALL
- The trap (int) instruction causes a jump to kernel code that handles the system call

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
```

Trap frame in xv6

- Trap frame is the structure pushed on kernel stack before trap handling, popped when returning from trap
- Contains various registers that are saved on kernel stack before trap handling
- The “int n” instruction pushes a few registers (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
- Think: why are EIP, ESP pushed by hardware and not by kernel code?

```c
// 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;
};
```

xv6 kernel trap handler

- IDT entries for all interrupts will set eip to point to the kernel trap handler “alltraps”
- Alltraps assembly code pushes remaining registers to complete trapframe on kernel stack
- Invokes C trap handling function named “trap”
  - Push pointer to trapframe (current top of stack, esp) as argument to the C function

```
#include "mmu.h"

  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

C trap handler function in xv6

- C trap handler performs different actions based on kind of trap
- Different types of traps identified using value of “n” in “int n”
- For system call, “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 and calls the corresponding function (fork, exec, …)
  - Return value of syscall stored in eax

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

```c
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;
  }
}
```

- If interrupt from a device, corresponding driver code called
- Timer is special hardware interrupt, 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;
```

```
  // 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);
}
```

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

Return from trap

- Assembly code “trapret”
- Pop all state from kernel stack
- Return-from-trap instruction “iret” does the opposite of int
  - Pop values pushed by “int”
  - Change back privilege level
- Execution of pre-trap code can resume

```
#include "mmu.h"

  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

xv6 trap handling: the complete story

- System calls, program faults, or hardware interrupts cause CPU to run “int n” instruction and “trap” to OS
- The trap instruction (int n) causes CPU to switch ESP to kernel stack, EIP to kernel trap handling code “alltraps”
- Pre-trap CPU state is saved on kernel stack in the trap frame by int instruction + alltraps code
- Alltraps assembly code calls C trap handling function
- C trap handler handles trap suitably and returns to trapret code
- Trapret pops register context and runs “iret” instruction to return from trap to user mode of process