CS 423
Operating System Design: Processes and CPU Virtualization
1/24

Ram Alagappan

Logistics

We have a second TA

Max Qian, zq2@illinois.edu

Office Hours

Ram Alagappan

W: 3:15-4pm, 1306 Everitt Laboratory

Max Qian

Tue: 1-2pm

Fri: 3:30-4:30pm

Xuhao Luo

Wed: 5-6pm

Fri: 2-3pm

AGENDA / OUTCOMES

Abstraction

- What is a Process ? What is its lifecycle ?

Mechanism

- How does process interact with the OS ?
- How does the OS switch between processes ?

ABSTRACTION: PROCESS

PROGRAM VS PROCESS

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[]) {
    char *str = argv[1];

    while (1) {
      printf("%s\n", str);
      Spin(1);
    }
    return 0;
}
```

Program

Process

Figures and slides ack: Prof. Shivaram Venkataraman (Wisconsin)

WHAT IS A PROCESS?

Stream of executing instructions and their "context"

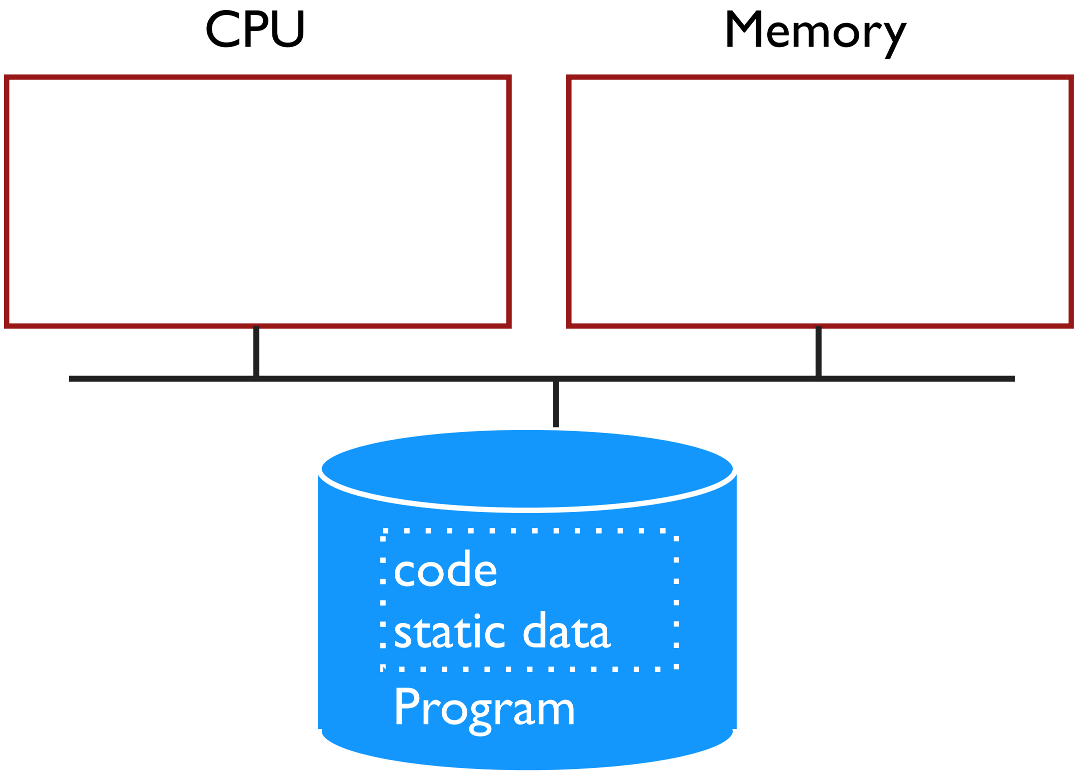
Instruction
Pointer

```
pushq   %rbp
movq    %rsp, %rbp
subq    $32, %rsp
movl    $0, -4(%rbp)
movl    %edi, -8(%rbp)
movq    %rsi, -16(%rbp)
cmpl    $2, -8(%rbp)
je      LBB0_2
```

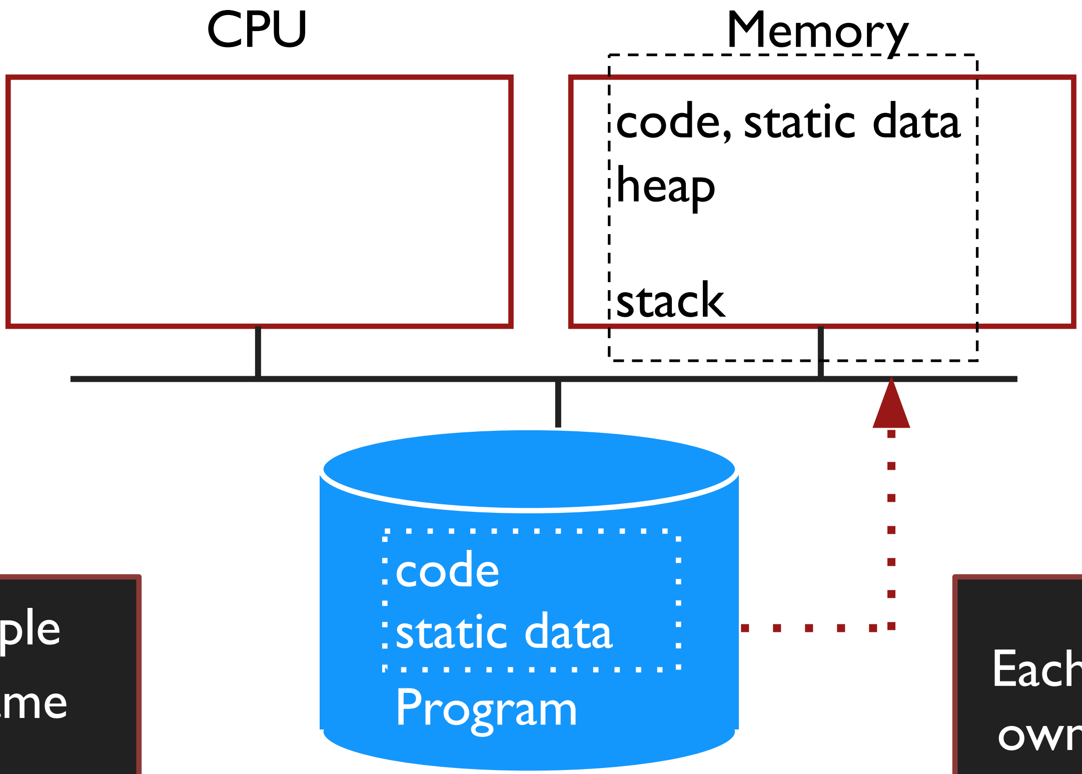
Registers
Memory addrs

File descriptors

PROCESS CREATION



PROCESS CREATION



Can run multiple instances of same program

Each program has its own stack, heap etc.

PROCESS VS THREAD

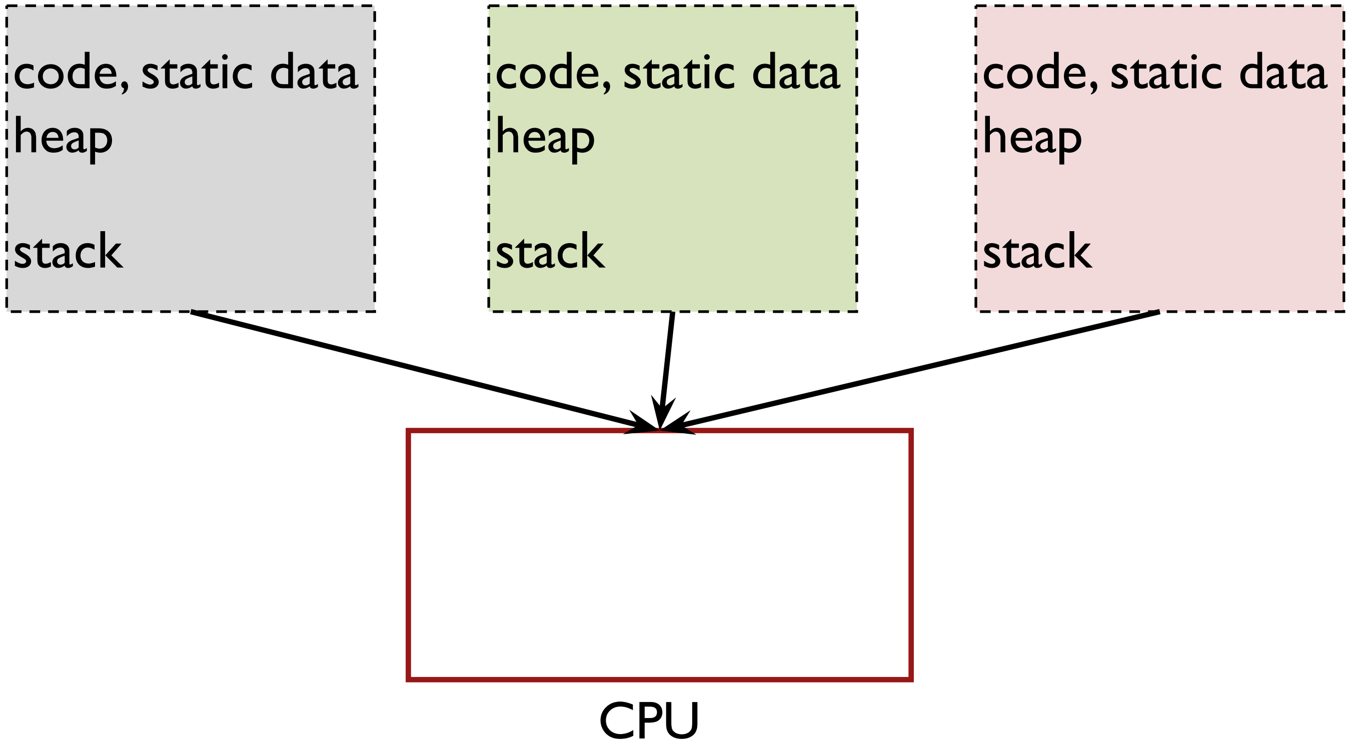
Threads: “Lightweight process”

Execution streams that share an address space
Can directly read / write memory

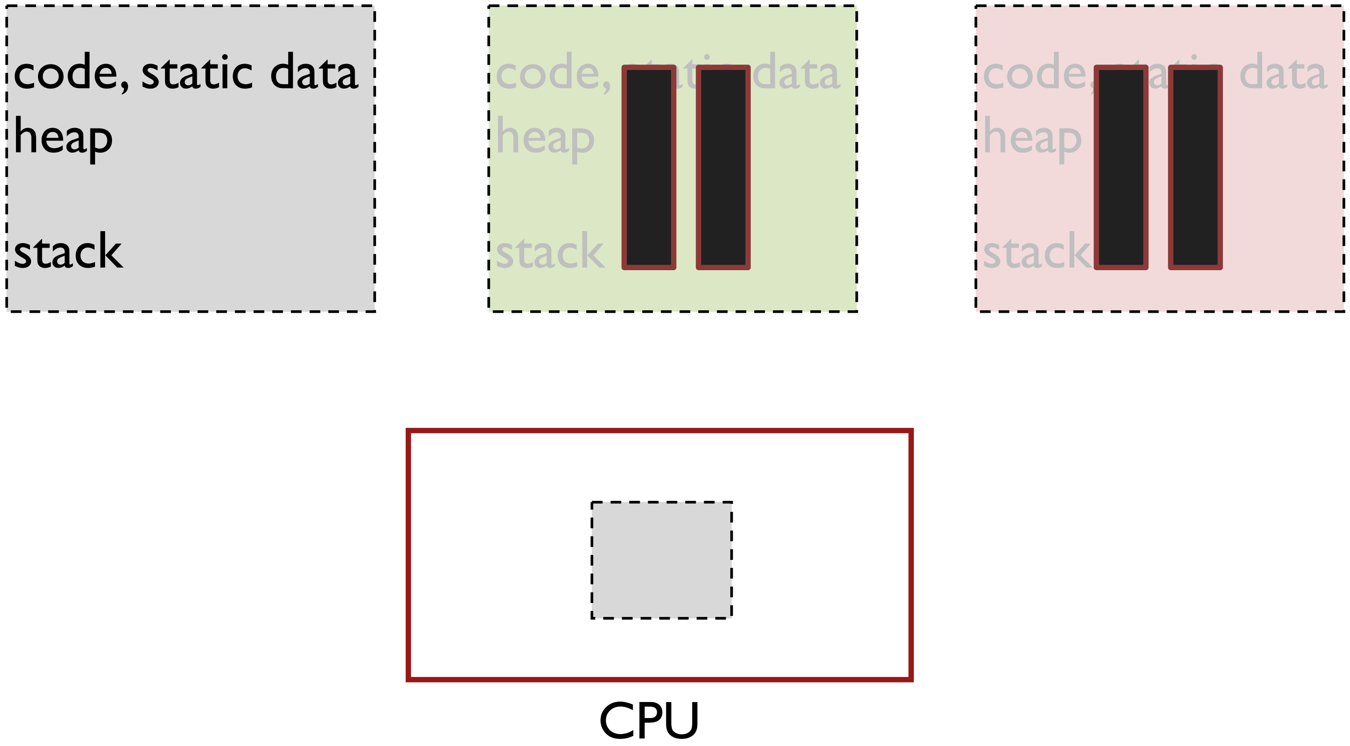
Can have multiple threads within a single process

SHARING THE CPU

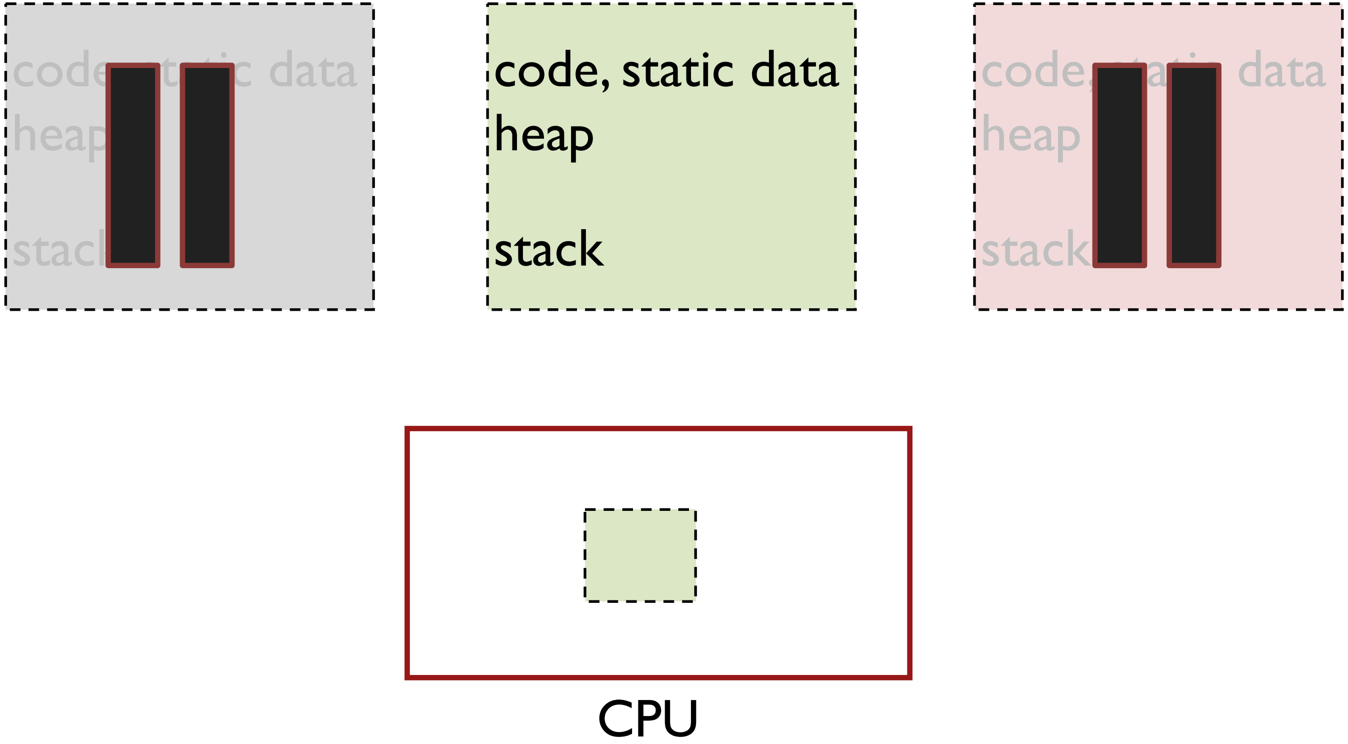
SHARING CPU


code, static data
heap
stack
code, static data
heap
stack
code, static data
heap
stack
CPU

TIME SHARING



TIME SHARING

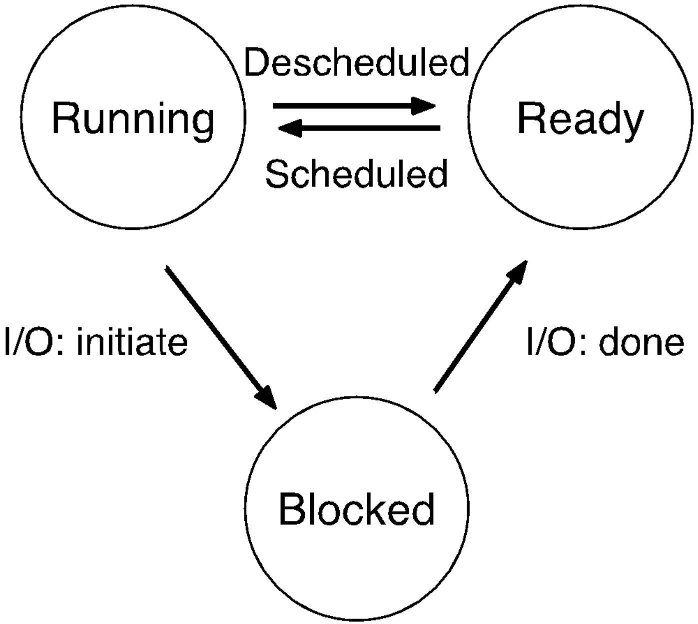


WHAT TO DO WITH PROCESSES THAT ARE NOT RUNNING ?

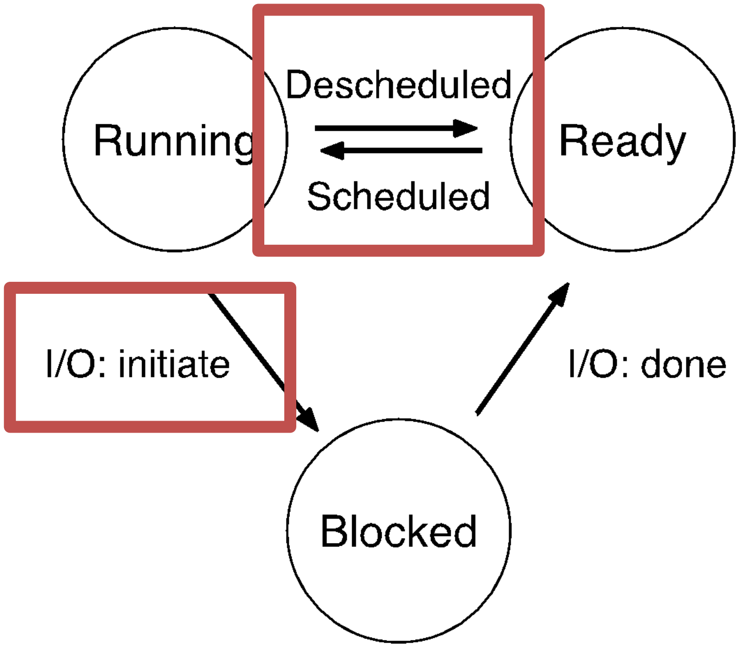
OS Scheduler

- Save context when process is paused
- Restore context on resumption

STATE TRANSITIONS



STATE TRANSITIONS



Question

Process 0
- io
- io
- cpu

Process 1
- cpu
- io
- io

Each IO takes 5 time units

Time	PID: 0	PID: 1
1	RUN:io	READY
2	WAITING	RUN:cpu
3	WAITING	RUN:io
4	WAITING	WAITING
5	WAITING	WAITING
6	RUN:io	WAITING
7	WAITING	WAITING

What happens at time 8?

CPU SHARING

Policy goals

- Virtualize CPU resource using processes
- Reschedule process for better CPU utilization? fairness?

Mechanism goals

- Efficiency: Sharing should not add overhead
- Control: OS should be able to intervene when required

EFFICIENT EXECUTION

Answer: Direct Execution

- Allow user process to run directly on the CPU (no OS intervention)
- Create process and transfer control to main()

What’s the Problem with DE?

EFFICIENT EXECUTION

Problems with DE:

Restricted ops: What if the process wants to do something restricted like allocating more resources, access IO devices, etc?

Switching b/w processes: What if the process runs forever? Buggy? Malicious?

General solution: Limited Direct Execution (LDE)

PROBLEM 1: RESTRICTED OPS

How can we ensure user process can't harm others?

Solution: privilege levels supported by hardware (bit of status)

User processes run in user mode (restricted mode)

OS runs in kernel mode (not restricted)

How can process access devices?

System calls (function call implemented by OS)

SYSTEM CALL

Syscall

Trap instruction :

Jumps into the kernel, changes the processor mode (to kernel)

What is it in x86?

Ret-from-trap instruction:

Return from the kernel, change to user mode

What is it in x86?

Libraries usually hide these instructions and give a nicer interface like read()/write()

Syscall

Must save callee registers and instruction pointer to resume after syscall

Where are these saved?

Kernel stack: every process has its own kernel stack

Operating System | Hardware | Program

Process A

Run main() ...
Call system call
trap into OS

save regs(A) to k-stack(A)
move to kernel mode
jump to trap handler

Handle the trap
Do work of syscall
return-from-trap

Restore regs (from kstack)
move to user mode
jump to PC after trap

Syscall

How does the hardware know where to jump (i.e., trap handler location) !?

Solution: trap table and system call table

64 tells that this is syscall (other numbers for other exceptional events)

6 tells that this is a sys_read

During boot time, OS “configures” the hardware to say where the trap handlers are located…

On trap, hardware simply jumps to this location

OS then knows this is a syscall, uses the syscall number to decide which particular syscall to invoke

SYSCALL SUMMMARY

Separate user-mode from kernel mode for security

Syscall: call kernel mode functions

- Transfer from user-mode to kernel-mode (trap)
- Return from kernel-mode to user-mode (return-from-trap)

To call SYS_read the instructions we used were

movl $6, %eax
int $64

To call SYS_exec what will be the instructions?

movl ____ %eax
int ____

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_write   5
#define SYS_read    6
#define SYS_close   7
#define SYS_kill    8
#define SYS_exec    9
#define SYS_open   10
```

PROBLEM2: HOW TO TAKE CPU AWAY

Policy

- To decide which process to schedule when
- Decision-maker to optimize some workload performance metric

Mechanism

- To switch between processes
- Low-level code that implements the decision

Separation of policy and mechanism: Recurring theme in OS

DISPATCH MECHANISM

OS runs dispatch loop

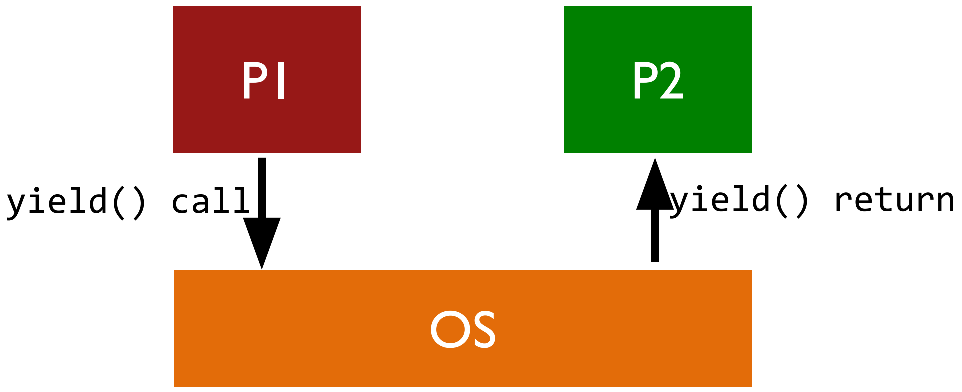
```
while (1) {
     run process A for some time-slice
     stop process A and save its context
     load context of another process B
}
```

Question 1: How does dispatcher gain control?
Question 2: What must be saved and restored?

HOW DOES DISPATCHER GET CONTROL?

Option 1: Cooperative Multi-tasking: Trust process to relinquish CPU through traps

- Examples: System call, page fault (access page not in main memory), or error (illegal instruction or divide by zero)
- Provide special `yield()` system call



PROBLEMS WITH COOPERATIVE ?

Disadvantages: Processes can misbehave

By avoiding all traps and performing no I/O, can take over entire machine

Only solution: Reboot!

Not performed in modern operating systems

TIMER-BASED INTERRUPTS

Option 2: Timer-based Multi-tasking

Guarantee OS can obtain control periodically

Enter OS by enabling periodic alarm clock

- Hardware generates timer interrupt (CPU or separate chip)
- Example: Every 10ms
- Can user code turn off timer?

Operating System

Hardware

Program

Process A

timer interrupt
save regs(A) to k-stack(A)
move to kernel mode
jump to trap handler

Operating System

Hardware

Program

Process A

timer interrupt
save regs(A) to k-stack(A)
move to kernel mode
jump to trap handler

Handle the trap
Call switch() routine
 save kernel regs(A) to proc-struct(A)
 restore kernel regs(B) from proc-struct(B)
 switch to k-stack(B)
 return-from-trap (into B)

Note: difference b/w caller vs. callee-saved registers

Operating System | Hardware | Program
Process A

timer interrupt
save regs(A) to k-stack(A)
move to kernel mode
jump to trap handler

Handle the trap
Call switch() routine
 save kernel regs(A) to proc-struct(A)
 restore kernel regs(B) from proc-struct(B)
 switch to k-stack(B)
 return-from-trap (into B)

restore regs(B) from k-stack(B)
move to user mode
jump to B's IP

Operating System	Hardware	Program
		Process A
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call switch() routine save kernel regs(A) to proc-struct(A) restore kernel regs(B) from proc-struct(B) switch to k-stack(B) —> this is the key point return-from-trap (into B)		
	restore regs(B) from k-stack(B) move to user mode jump to B's IP	
		Process B

xv6 Example

```c
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

- Scheduler switches to user process in “scheduler” function
- User process switches to scheduler thread in the “sched” function

```c
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);

  }
}
```

Swtch() function

- What is on the k-stack of A when a process A has just invoked the swtch?
- What does swtch do?
- What will swtch find on new kernel stack?
- Where does it return to?

Swtch() function

•What is on the k-stack when a process A has just invoked the swtch?

Just the caller save registers of A, return address (eip) – where is this exactly?

• What does swtch do?

Push remaining registers on old kernel stack (i.e., callee save registers or kernel registers of A)

Save pointer to this context into context structure pointer of old process (A)

Switch esp from old kernel stack (A k-stack) to new kernel stack (B k-stack)

ESP now points to saved context of new process (B k-stack)

Pop callee-save registers from new stack

Return (pops return address, caller save registers – hardware does this)

• What will swtch find on new kernel stack?

Whatever was pushed when the new process gave up its CPU in the past

•Where does it return to?

We switched kernel stacks from old process to new process, CPU is now executing new process code, resuming where the process gave up its CPU by calling swtch in the past

Swtch impl in xv6

```
# void swtch(struct context *old, struct context *new);
#
# Save current register context in old
# and then load register context from new.
.globl swtch
swtch:
  # Save old registers
  movl 4(%esp), %eax  # put old ptr into eax
  popl 0(%eax)        # save the old IP
  movl %esp, 4(%eax)  # and stack
  movl %ebx, 8(%eax)  # and other registers
  movl %ecx, 12(%eax)
  movl %edx, 16(%eax)
  movl %esi, 20(%eax)
  movl %edi, 24(%eax)
  movl %ebp, 28(%eax)

  # Load new registers
  movl 4(%esp), %eax  # put new ptr into eax
  movl 28(%eax), %ebp # restore other registers
  movl 24(%eax), %edi
  movl 20(%eax), %esi
  movl 16(%eax), %edx
  movl 12(%eax), %ecx
  movl 8(%eax), %ebx
  movl 4(%eax), %esp  # stack is switched here
  pushl 0(%eax)       # return addr put in place
  ret                 # finally return into new ctxt
```

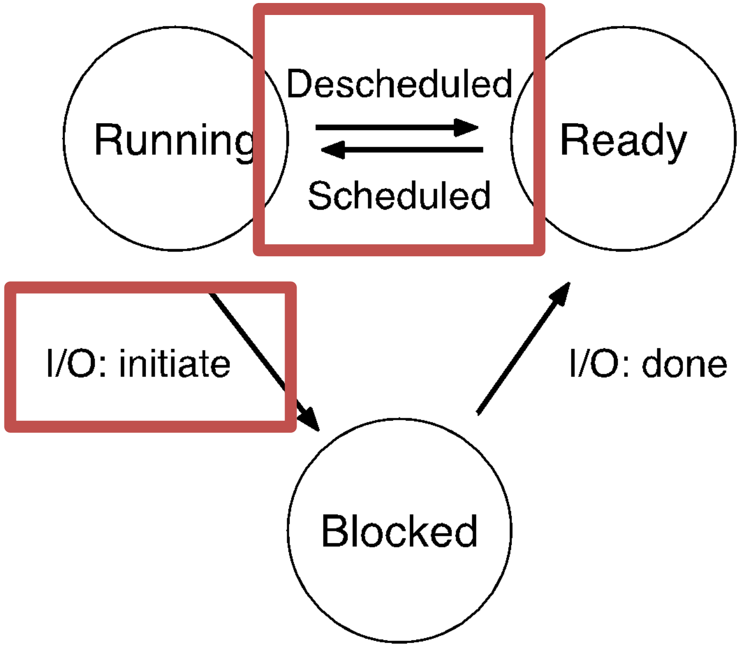
SUMMARY

Process: Abstraction to virtualize CPU

Use time-sharing in OS to switch between processes

Key aspects

- Use system calls to run access devices etc. from user mode
- Context-switch using interrupts for multi-tasking



POLICY ?
Next lecture!