CS 423
Operating System Design: Processes and CPU Virtualization

Ram Kesavan

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

Logistics

Office Hours

Ram Kesavan
    Tue/Thu: 3:15-4pm, 1310 DCL + 3126 Siebel

Peizhe Liu
    MWF: 6-7pm Floor 0 Siebel

Gabriella Xue
    MF: 10-11am Floor 0 Siebel

AGENDA / OUTCOMES

3 pieces: Virtualization, Concurrency, and Persistence

Abstraction

What is a Process? What is its lifecycle?

Mechanism

How does process interact with the OS?

How does the OS switch between processes?

What we won't cover here, but you should read up:

Ch4+5: process-related data structures. fork() & exec().

ABSTRACTION: PROCESS

PROGRAM VS PROCESS

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

int main(int argc, char *argv[]) {
    char *str = argv[1];
    int i = 0;
    while (1) {
      printf("%s\n",str);
      i++;

    }
    return 0;
}
```

Program

Process

WHAT IS A PROCESS?

Stream of executing instructions + associated “context”

```
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

WHAT IS A PROCESS?

Stream of executing instructions + associated “context”

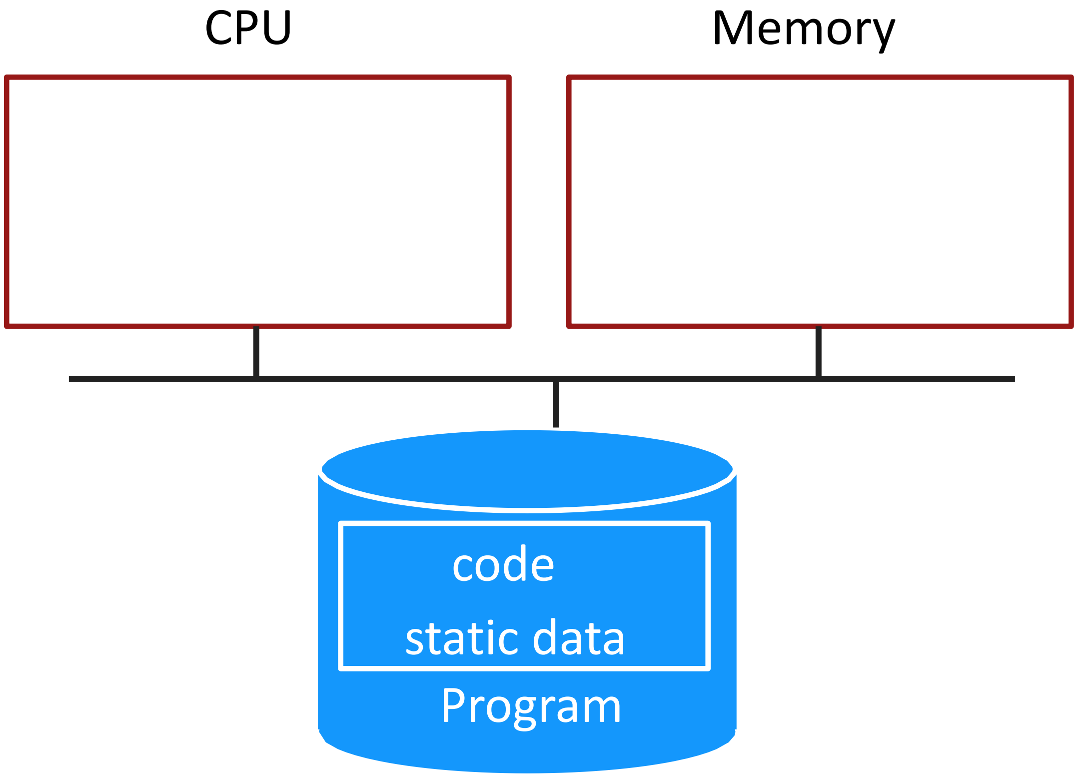
PC: program counter
aka IP
SP: stack pointer
FP: frame pointer

```
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
```

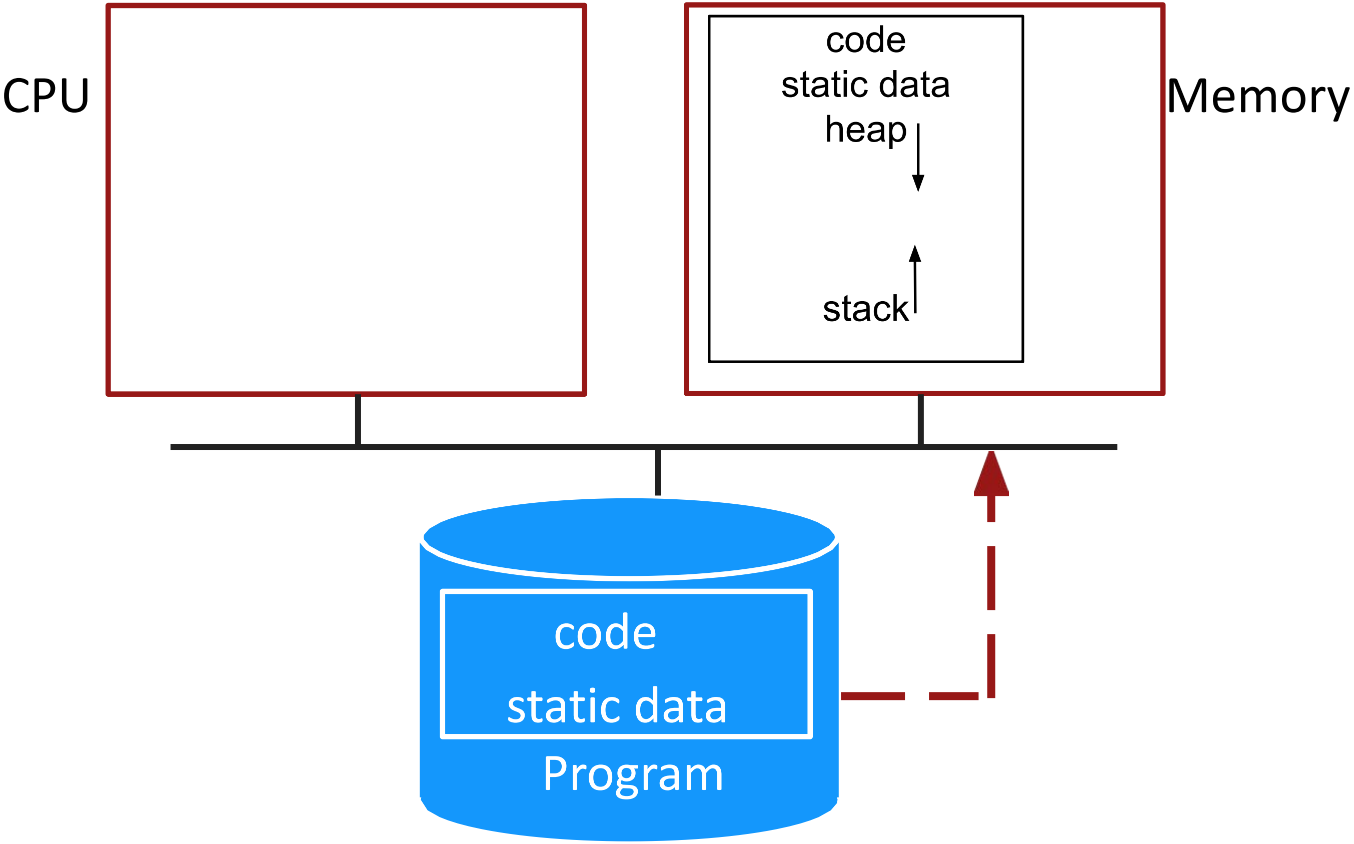
Registers
Memory addrs

File descriptors

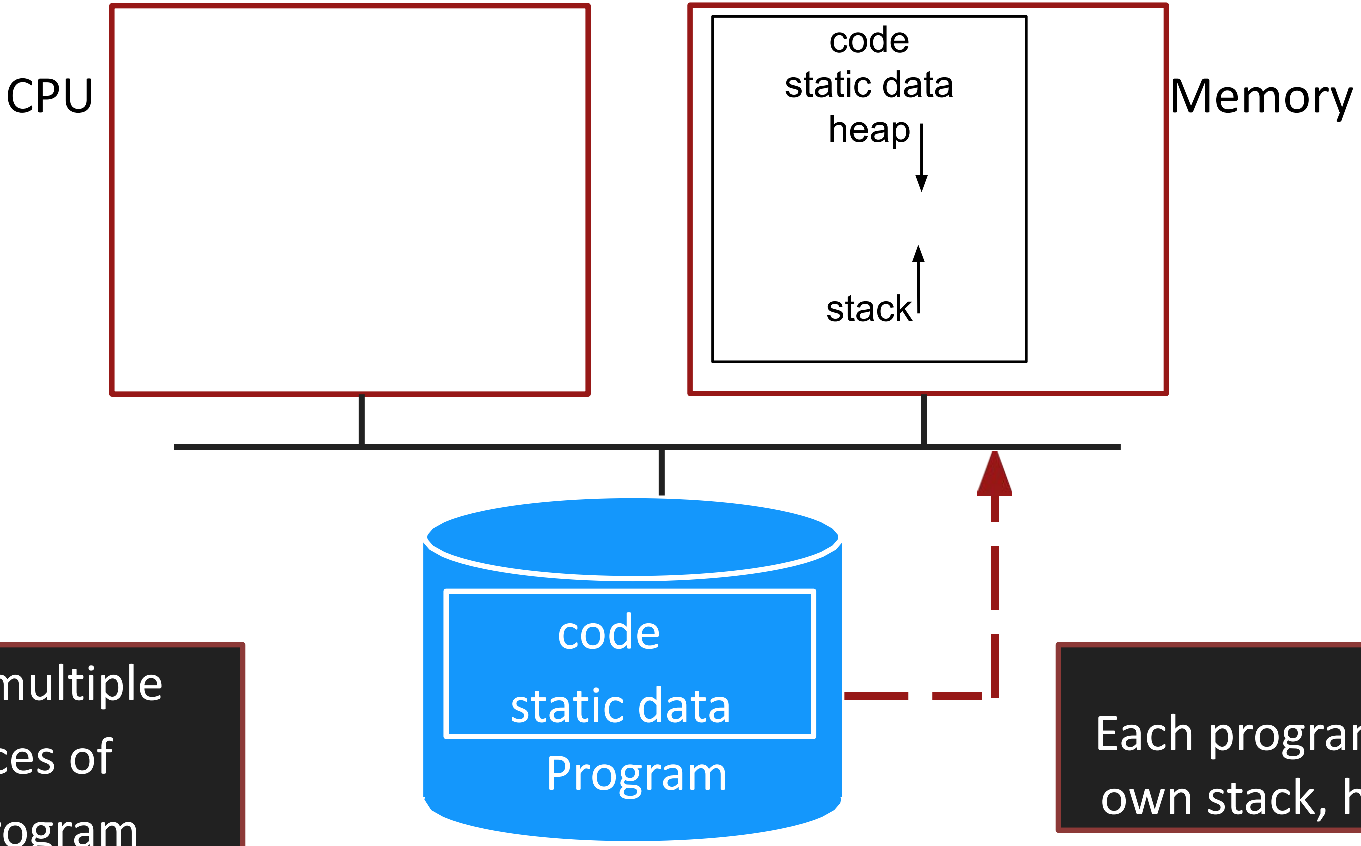
PROCESS CREATION



PROCESS CREATION



PROCESS CREATION



Can run multiple instances of same program

Each program has its own stack, heap etc.

PROCESS VS THREAD

Threads: “Lightweight process”

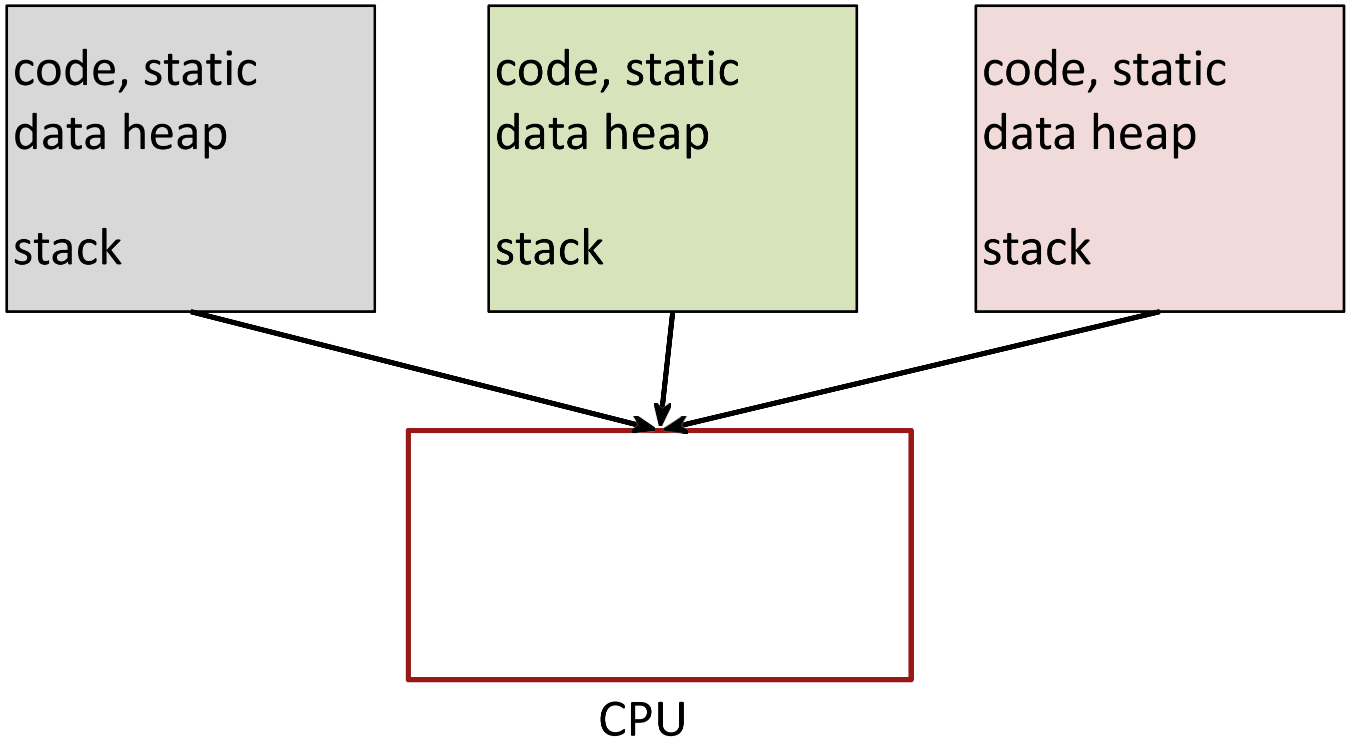
Execution streams that share the parent process’ resources: address space, files, sockets, etc.

Each thread has its own stack & registers

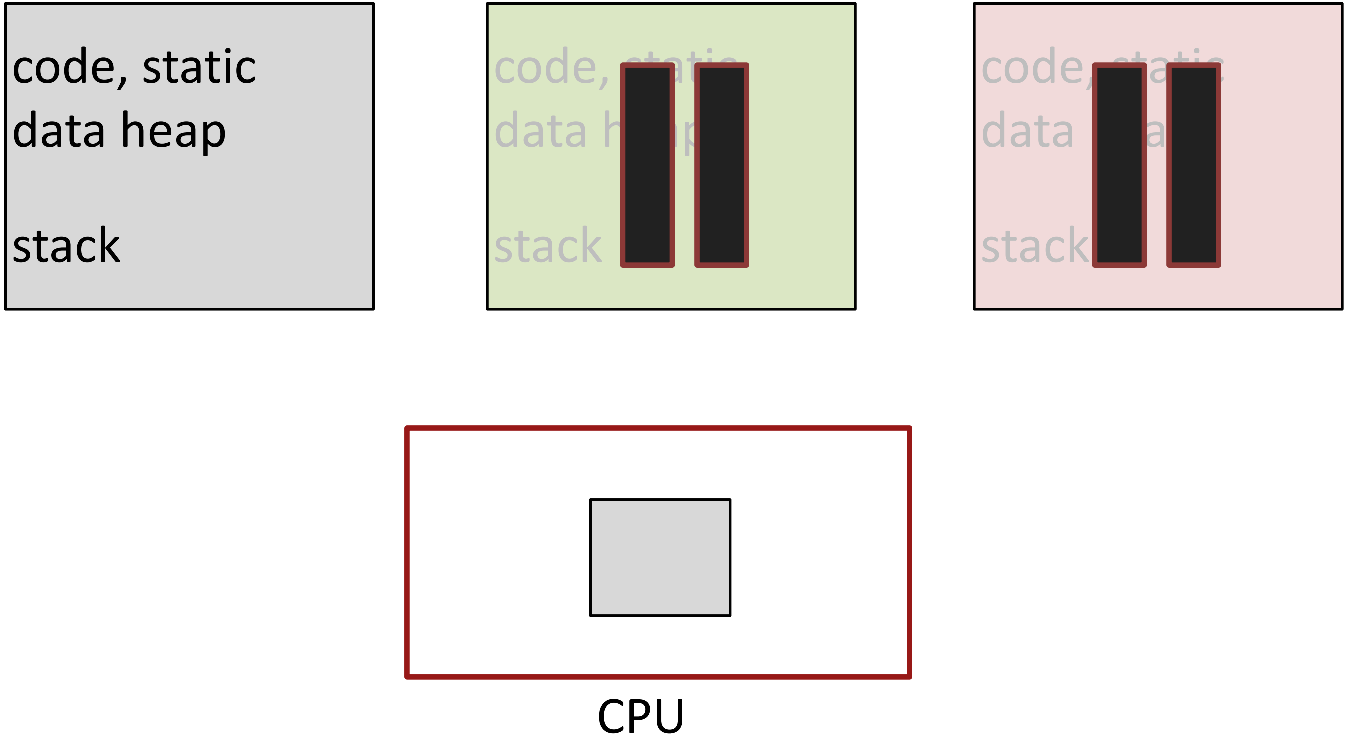
Can have multiple threads within a single process

SHARING THE CPU

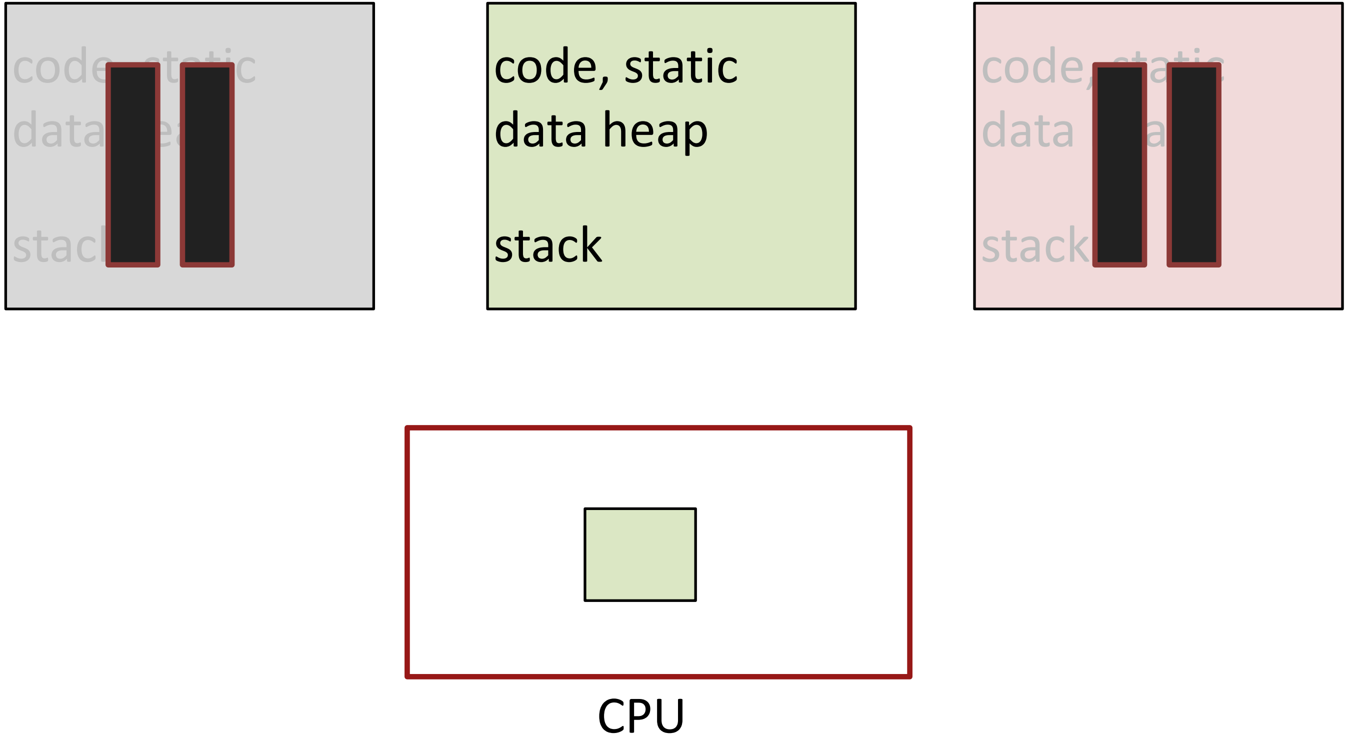
SHARING CPU


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

TIME SHARING



TIME SHARING

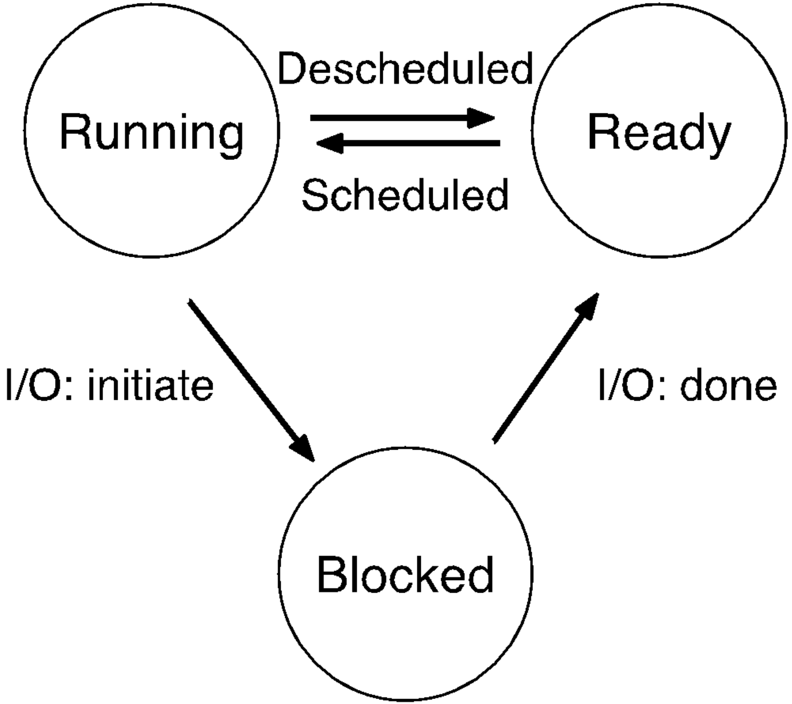


WHAT TO DO WITH PROCESSES THAT ARE NOT RUNNING ?

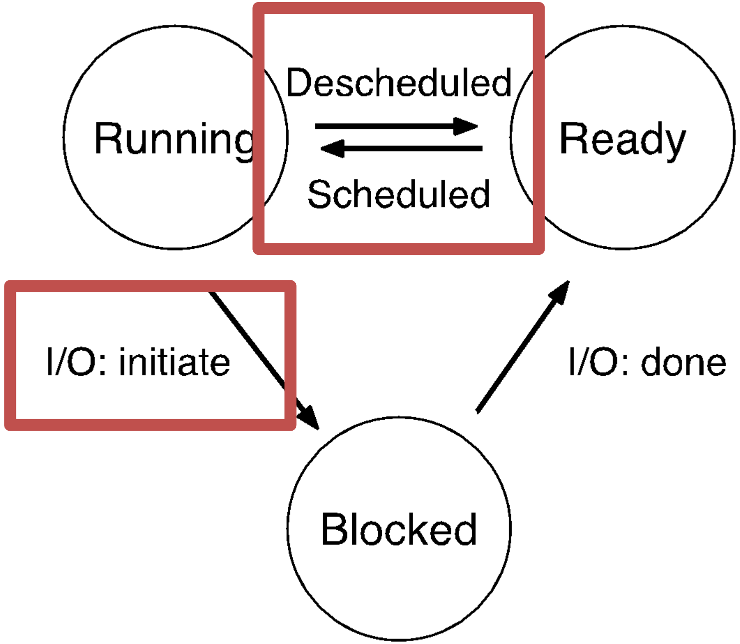
OS Scheduler

- Save context (aka state) when pausing process
- Restore context on resumption

STATE TRANSITIONS



STATE TRANSITIONS



Question

Process 0
io
io
cpu (1 unit)

Process 1
cpu (4 units)
io
io

Each IO takes 5 time units

Time	PID: 0	PID: 1
1	RUNNING	READY
2		
3		
4		
5		
6		
7		
8		

Question

Process 0
io
io
cpu (1 unit)

Process 1
cpu (4 units)
io
io

Each IO takes 5 time units

Time	PID: 0	PID: 1
1	RUNNING io	READY
2	BLOCKED	RUNNING cpu
3	BLOCKED	RUNNING cpu
4	BLOCKED	RUNNING cpu
5	BLOCKED	RUNNING cpu
6	READY	RUNNING io
7	RUNNING io	BLOCKED
8	BLOCKED	BLOCKED

CPU SHARING

Policy goals

- Virtualize CPU resource using processes
- Higher CPU utilization? Fairness?

Mechanism goals

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

Today, we're focused on only mechanism

EFFICIENT EXECUTION

Answer: Direct Execution

- User process runs directly on the CPU (no OS interposition)
- Create process and transfer control to main()

What does “run directly on the CPU” mean?

Problems with DE?

Problems with DE?

Problems with DE:

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

How to switch processes: What if the process runs “forever”?

General solution: Limited Direct Execution (LDE)

PROBLEM1: RESTRICTED OPS

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

Solution: privilege levels supported by hardware

- CPU: has a mode bit
- User process runs in user mode (restricted mode)
- OS runs in kernel mode (unrestricted)

How can a process access restricted ops?

- system call: function call implemented by OS

SYSTEM CALL

Syscall

Trap instruction :

Changes to unrestricted or kernel mode

What is it in x86? INT, SYSCALL, SYSENTER

Ret-from-trap instruction:

Return from kernel to user mode

What is it in x86? IRET, SYSRET, SYSEXIT

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

Syscall

Must save caller’s 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 past trap instruction	

Syscall

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

Solution: trap table & system call table
Syscall instruction tells trap handler to consult syscall table (syscall#)

At boot OS “configures” hardware with trap handler locations

On trap, hardware simply jumps to this location

OS knows this is a syscall, uses syscall number to invoke particular syscall

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)

```
function write()

; write(1, message, msg_len)
mov 5, %rax              ; 5 => SYS_write
mov  1, %rdi             ; file descriptor
lea [message], %rsi
mov msg_len, %rdx

syscall                  ; invoke trap
```

```c
// 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
- More next lecture

Mechanism

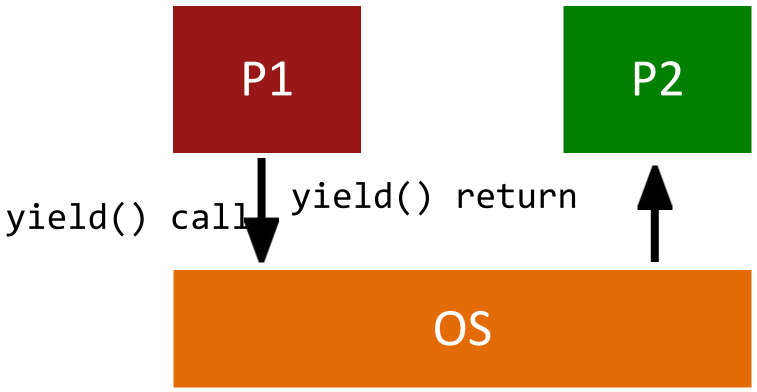
- Fast switch between processes
- Low-level code that implements the switch

Separation of policy and mechanism: Recurring theme in OS

HOW CAN OS GET CONTROL?

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

- 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 ?

Disadvantage: 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
Guarantees OS control within a deterministic time period

Enter OS by using a periodic "alarm clock"

- Hardware generates timer interrupt (CPU or separate chip)
- Example: Every 10ms

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)		

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

Process B

SUMMARY

Process: Abstraction to virtualize CPU resource

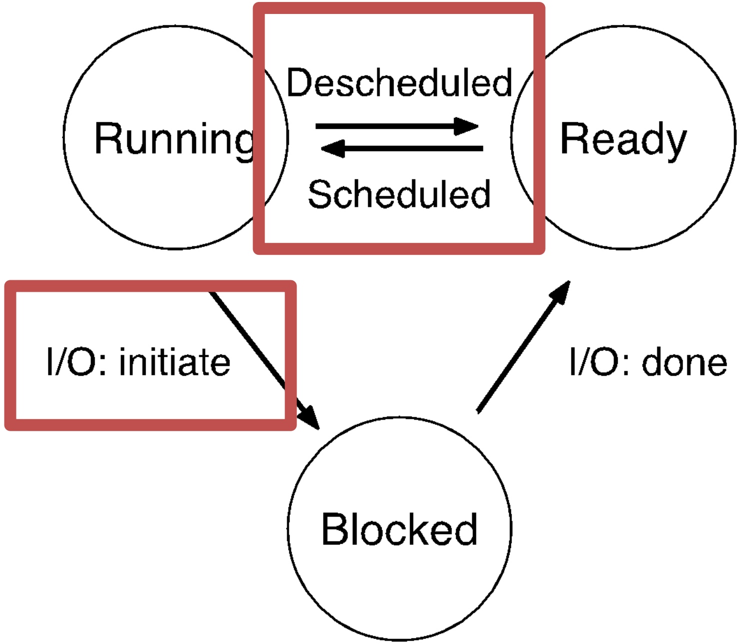
Time-sharing in OS to switch between processes

Key aspects of Mechanism

- System calls to access restricted ops
- Time-sharing: context-switch via timer interrupt

What we didn't cover here, but you should read up:

- Ch4+5: process-related data structures. fork() & exec().



POLICY ?

Next lecture!