Welcome back!

VIRTUALIZATION: CPU

Shivaram Venkataraman

CS 537, Fall 2024

ADMINISTRIVIA

gitlab repository

- Project 1 is out! Due on September 13th (Friday)
  - Check handin directory
  - Test cases → Discussion
- Signup for Piazza https://piazza.com/wisc/fall2024/cs537
- Lecture notes at pages.cs.wisc.edu/~shivaram/cs537-fa24/
- Drop? Waitlist? Email enrollment@cs.wisc.edu and cc me

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;
}
```

cpu.c

saved on disk

Program

C

C++

Java ...

Process

running

created when you run the program

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

↳ stack

↳ heap

File descriptors

↳ opened

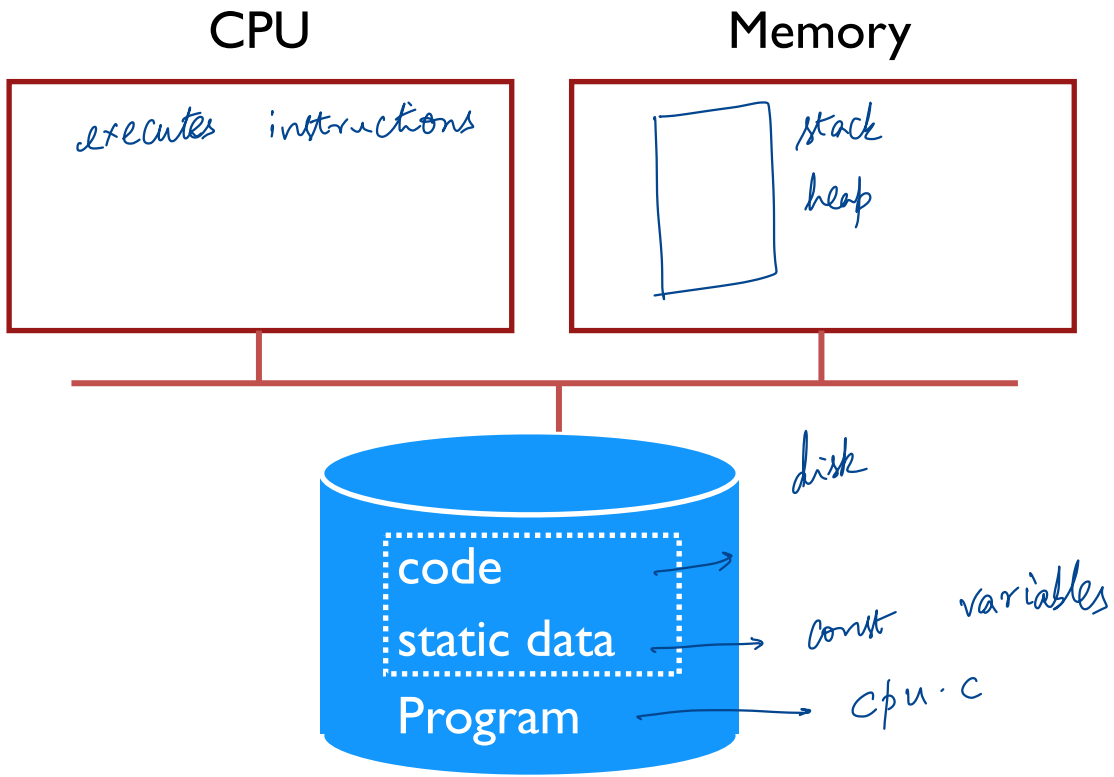
PROCESS IN XV6

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this proces
  enum procstate state;        // Process state
  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
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

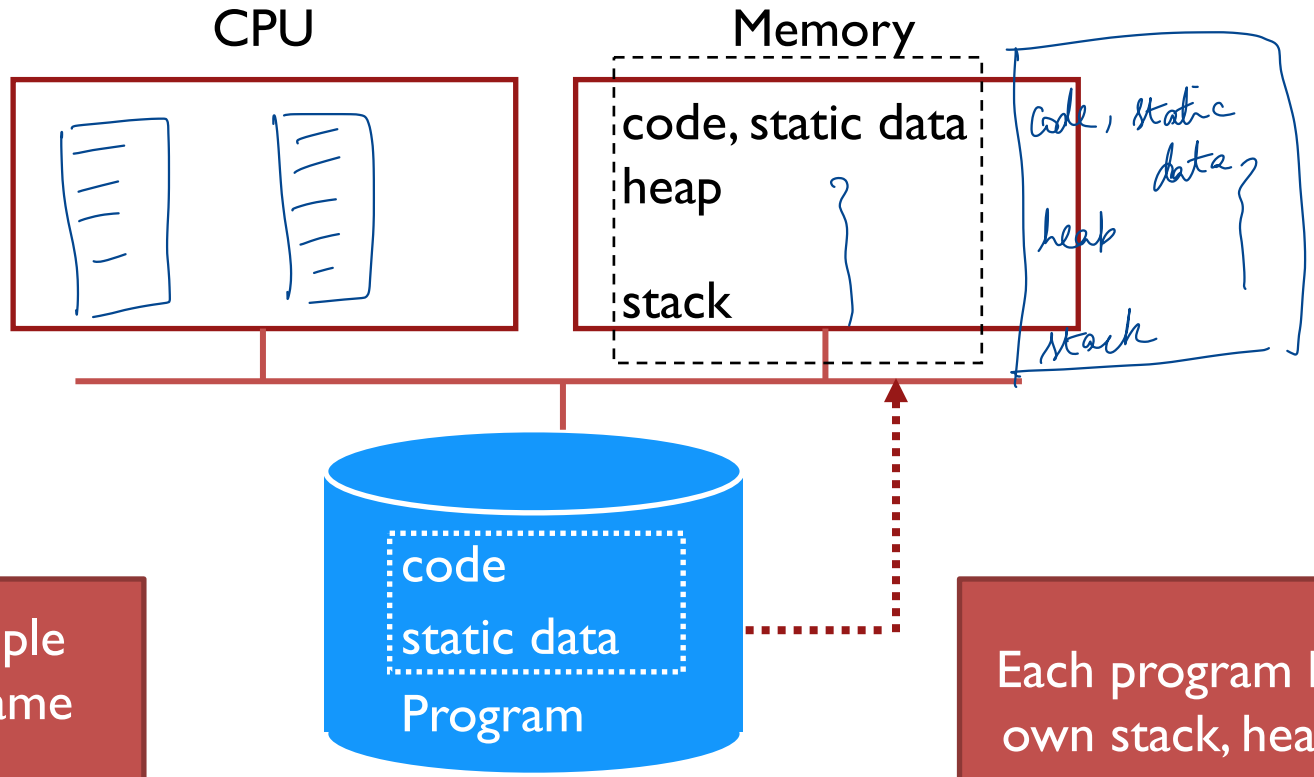
cpu

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

PROCESS CREATION



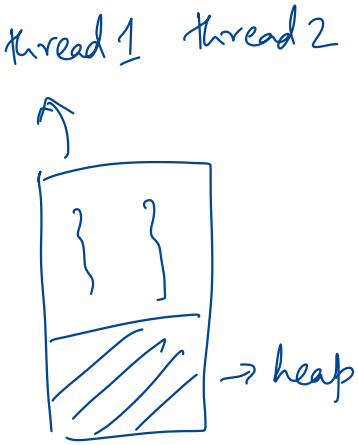
PROCESS CREATION



Can run multiple instances of same program

Each program has its own stack, heap etc.

PROCESS VS THREAD



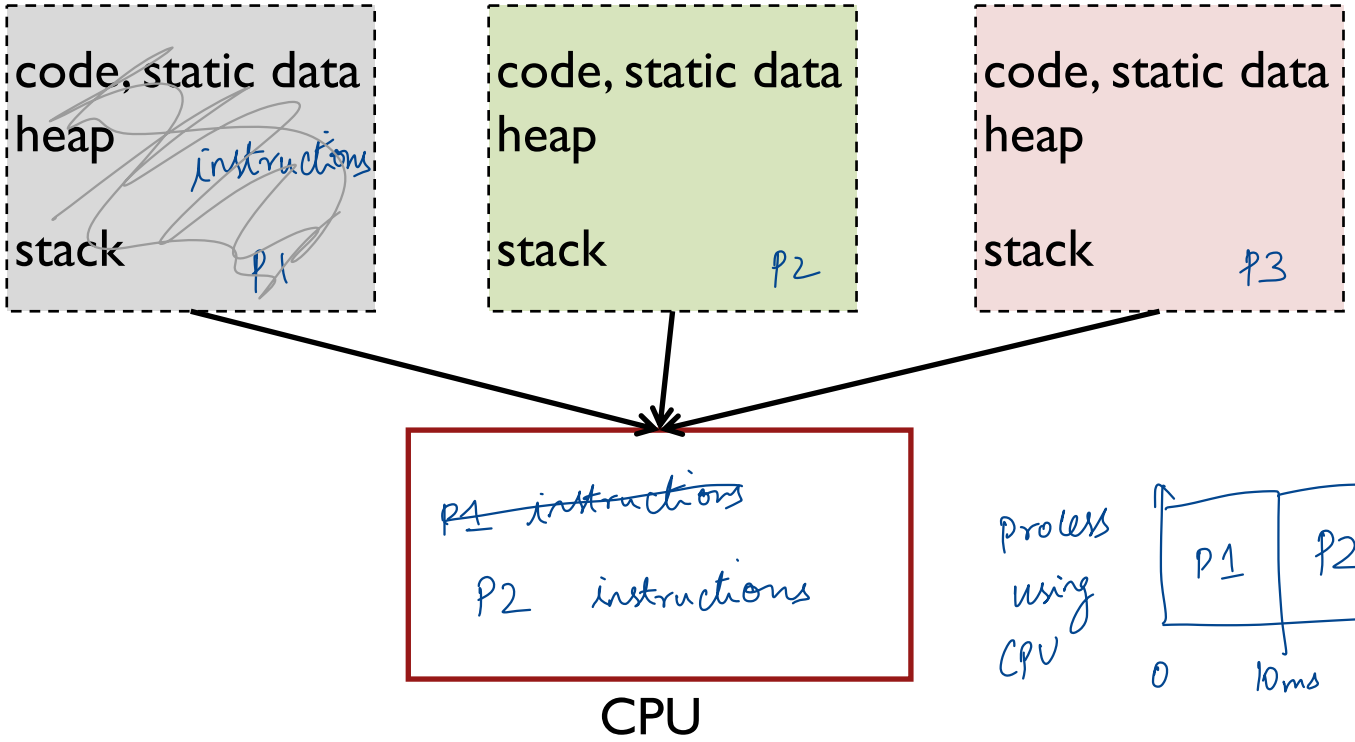
Threads: "Lightweight process"

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

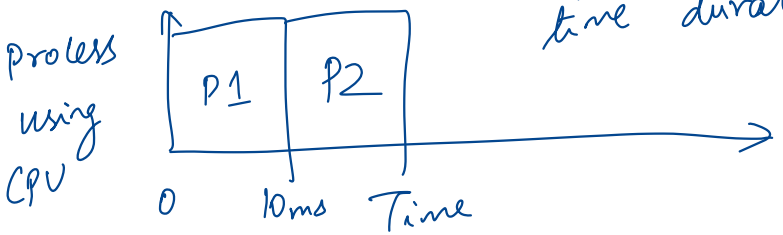
Can have multiple threads within a single process

SHARING THE CPU

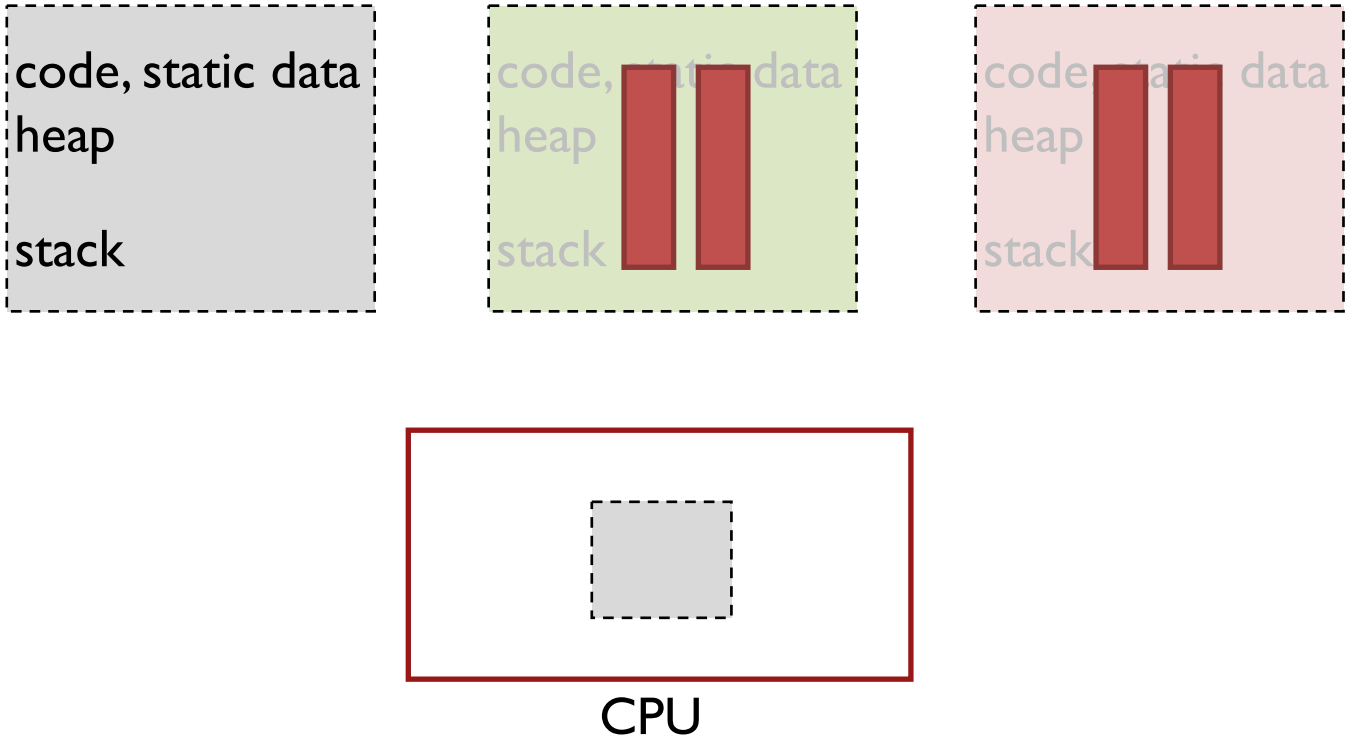
SHARING CPU



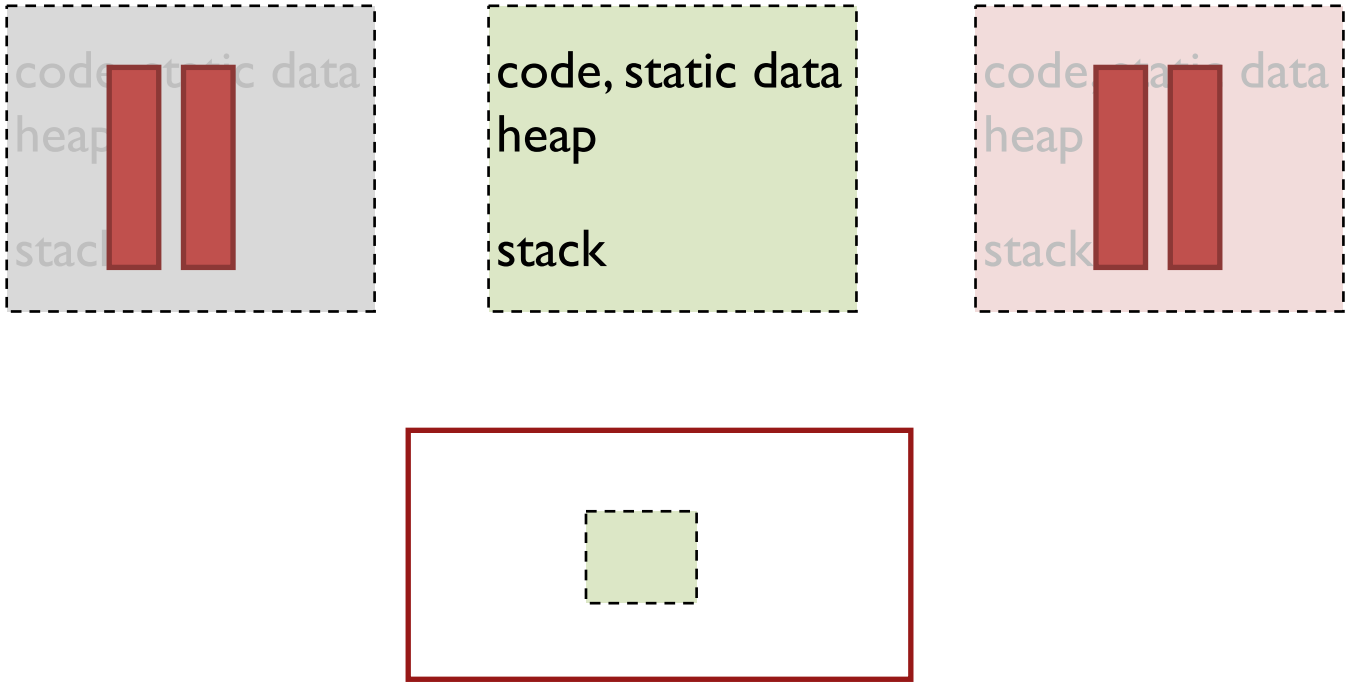
Time sharing
- one process is given exclusive access to the CPU for a fixed time duration



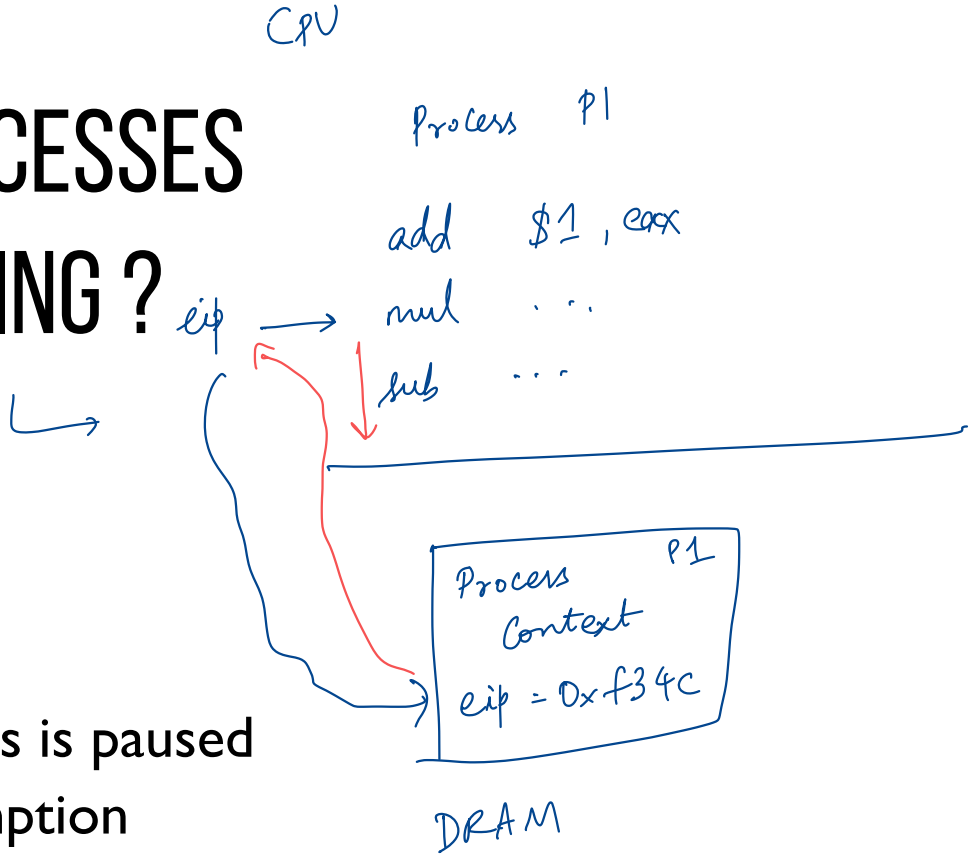
TIME SHARING



TIME SHARING



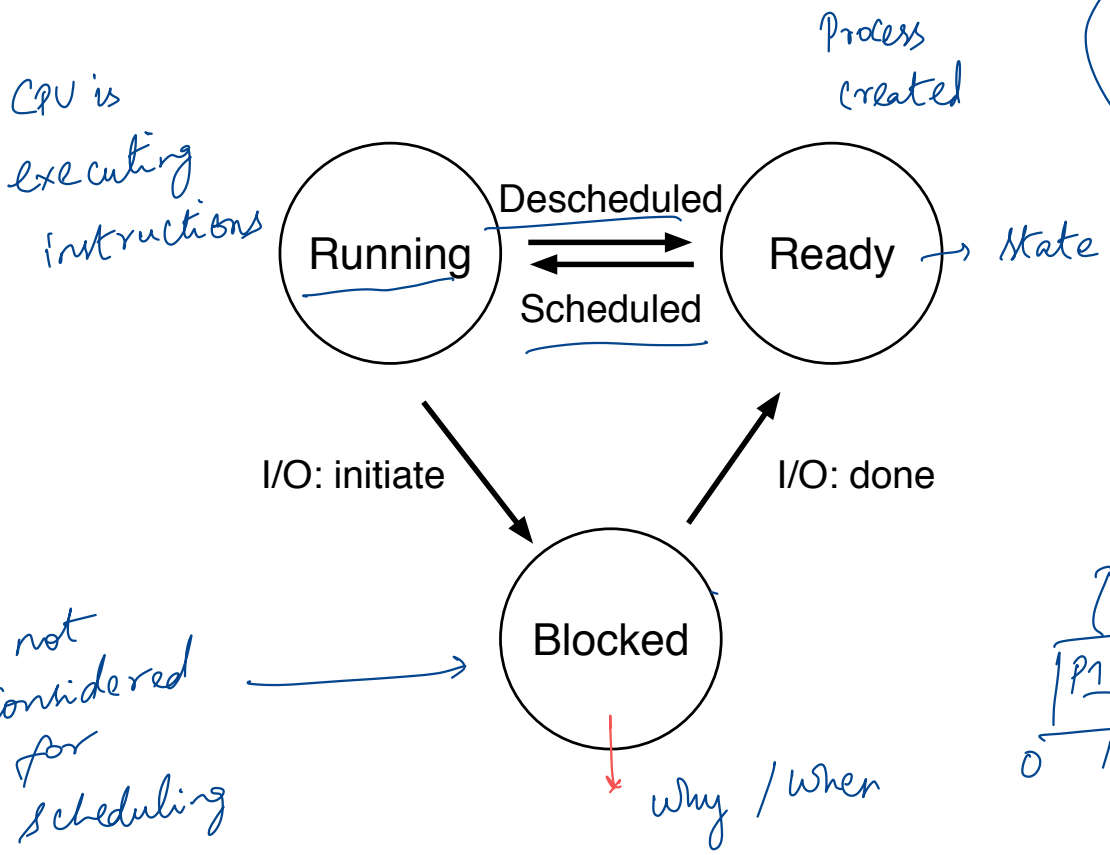
WHAT TO DO WITH PROCESSES THAT ARE NOT RUNNING ?



OS Scheduler

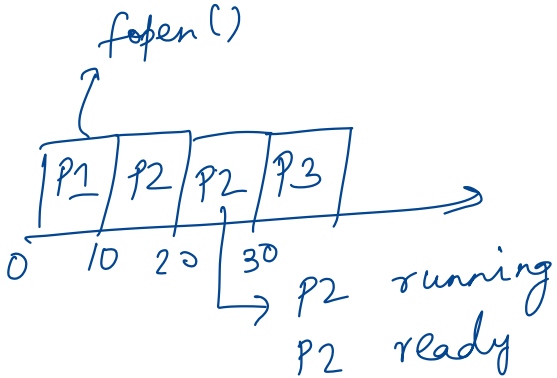
- Save context when process is paused
- Restore context on resumption

STATE TRANSITIONS



few milliseconds

```
main() {
    fopen("board.txt")
}
```



ASIDE: OSTEP HOMEWORKS!

- Optional homeworks corresponding to each chapter in book
- Little simulators to help you understand
- Can generate problems and solutions!

http://pages.cs.wisc.edu/~remzi/OSTEP/Homework/homework.html

github

PROCESS HW

Run ./process-run.py –l 2:100,2:0

QUIZ 1

https://tinyurl.com/cs537-fa24-quiz1a

≥ ./process-run.py -l 3:80,3:50

Time	PID: 0	PID: 1	CPU	IOs
1	RUN:io	READY	1	
2	BLOCKED	RUN:cpu	1	1
3	BLOCKED	RUN:io	1	1
4	BLOCKED	BLOCKED		2
5	BLOCKED	BLOCKED		2
6	BLOCKED	BLOCKED		2
7*	RUN:io_done	BLOCKED	1	1

→ Two Processes

8 RUN: CPU BLOCKED

Each IO takes 5 time units,
3 CPU slices per process

What happens at time 8?

CPU SHARING

Policy goals

- Virtualize CPU resource using processes
- Reschedule process for fairness? efficiency ?

→ next lecture

Mechanism goals

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

↓

Safety

EFFICIENT EXECUTION

Simple answer !?: Direct Execution

- Allow user process to run directly
- Create process and transfer control to main() → program you are trying to run

Challenges

- What if the process wants to do something restricted ? Access disk ?
- What if the process runs forever ? Buggy ? Malicious ?

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) → CPU

- User processes run in user mode (restricted mode)
- OS runs in kernel mode (not restricted)

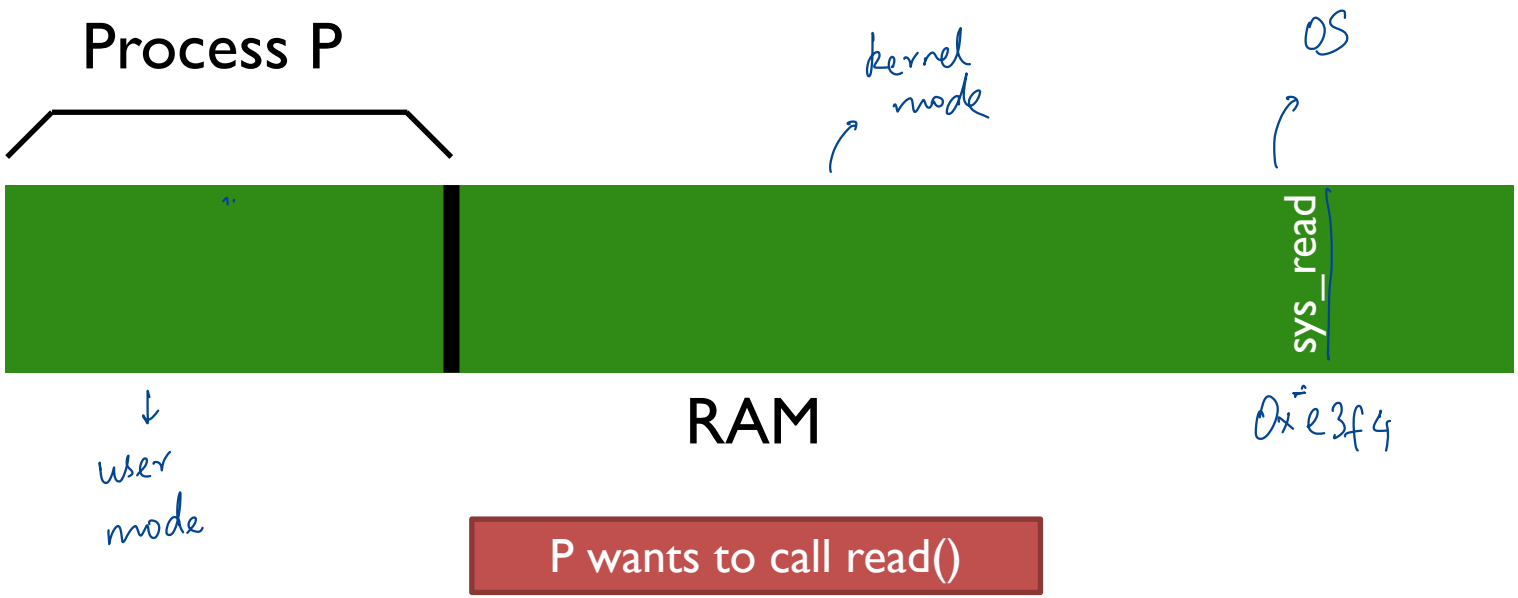
→ which memory regions
which devices
etc.

How can process access devices? → fopen

- System calls (function call implemented by OS)

SYSTEM CALL

SYSTEM CALL


Process P
kernel mode
OS
sys_read
RAM
user mode
0x e3f4
P wants to call read()

SYSTEM CALL

P can only see its own memory because of **user mode**
(other areas, including kernel, are hidden)

P wants to call read() but no way to call it directly

SYSTEM CALL

all interrupts are handled by OS

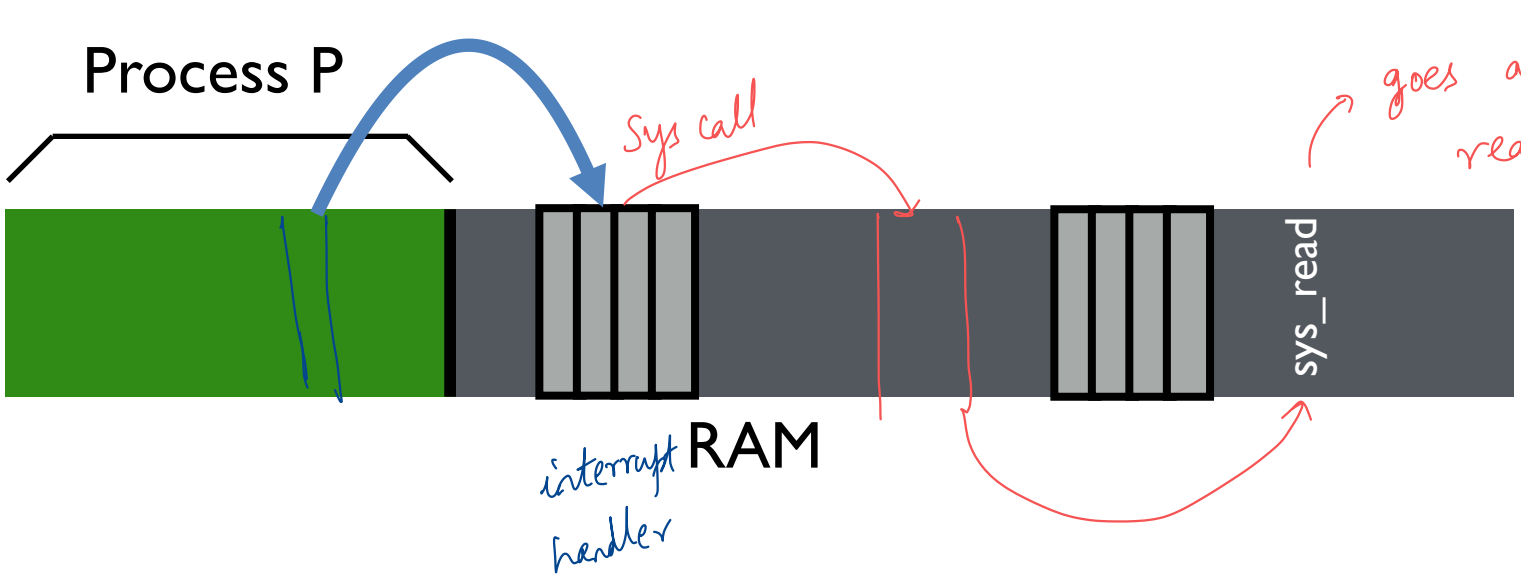
Step 1 : write down 6 in EAX

```
movl $6, %eax;    int $64
```

register

Step 2 : interrupt 64 System call interrupt

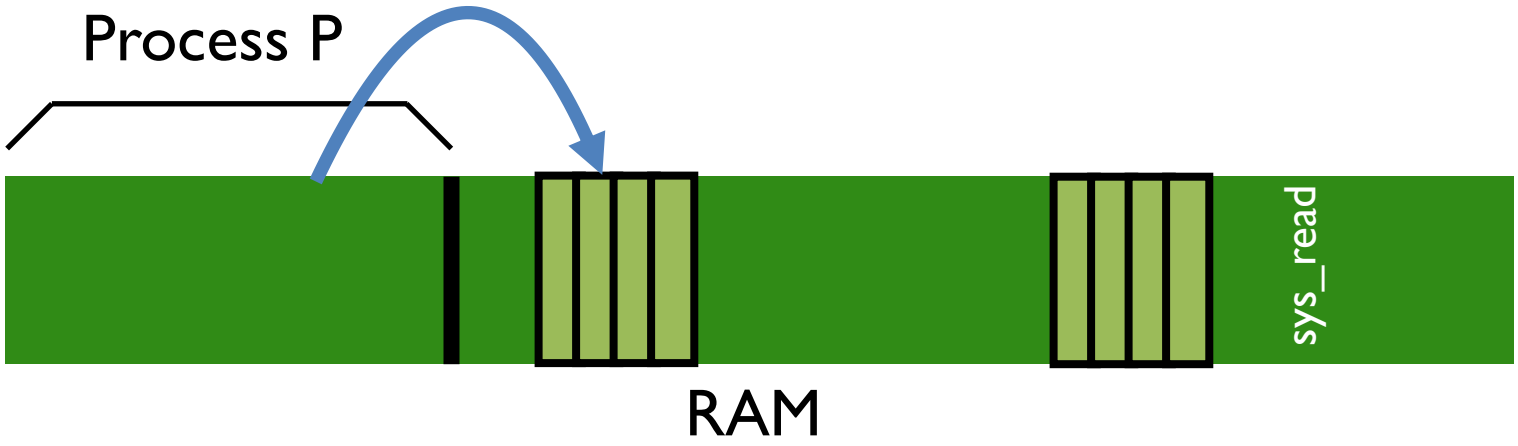
SYSTEM CALL




```
movl $6, %eax;    int $64
```

= read system call

SYSTEM CALL

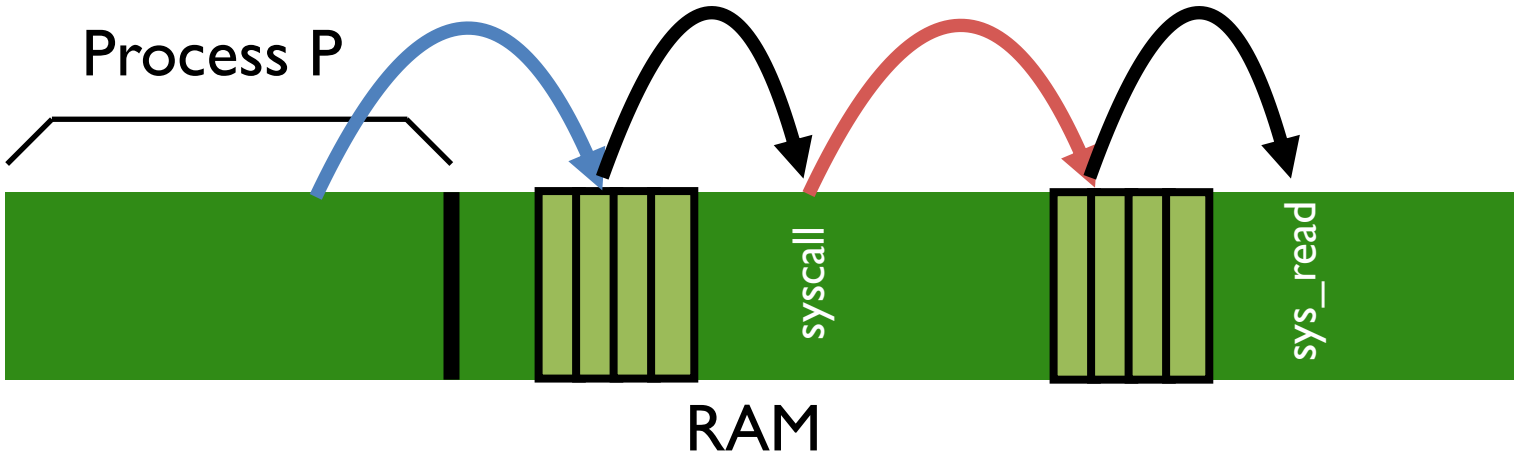


Syscall table index

```
movl $6, %eax;    int $64
```

Trap table index

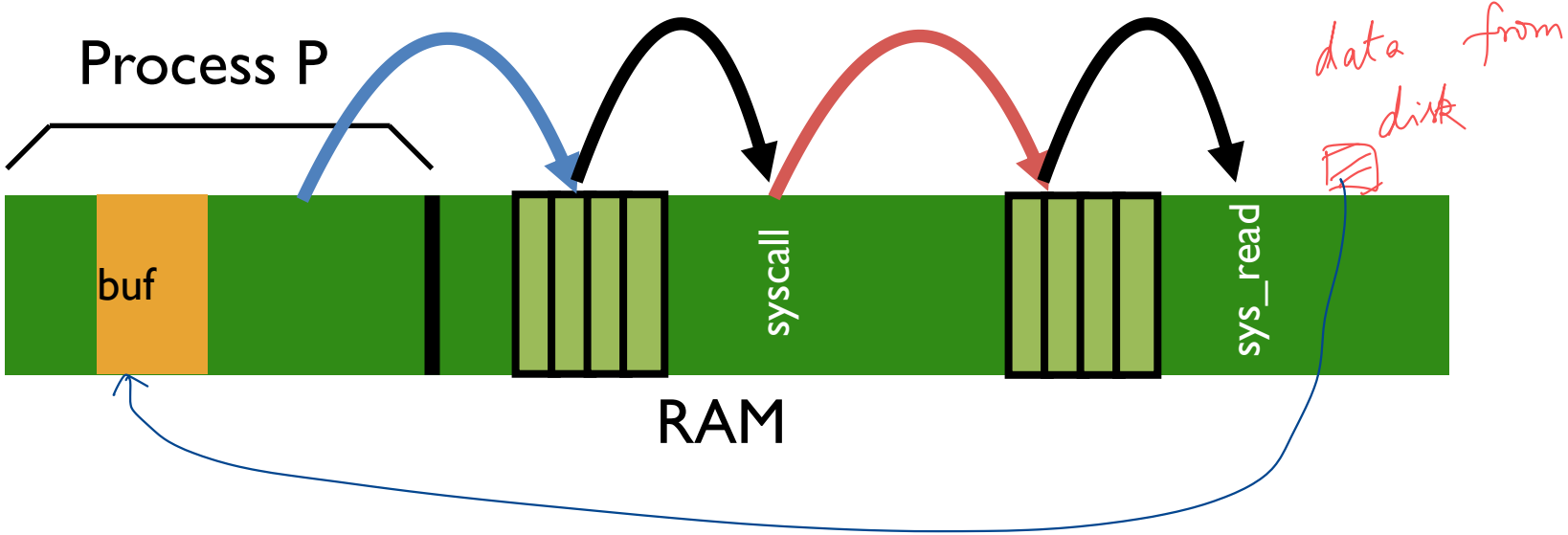
SYSTEM CALL




```
movl $6, %eax;    int $64
```

Follow entries to correct system call code

SYSTEM CALL



Kernel can access user memory to fill in user buffer
return-from-trap at end to return to Process P

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)

QUIZ 2

https://tinyurl.com/cs537-fa24-quiz1b

To call SYS_read the instructions we used were

movl $6, %eax
int $64

To call SYS_exec what will be the instructions?

movl $9 %eax
int $64 → system call handler

```
// 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
}
```

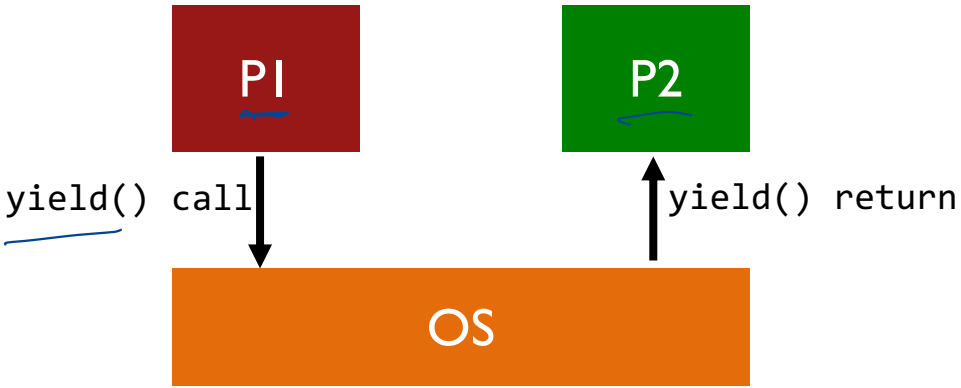
Question 1: How does dispatcher gain control?

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

timer interrupt every 10 ms

Enter OS by enabling periodic alarm clock

- Hardware generates timer interrupt (CPU or separate chip) Example: Every 10ms
- User must not be able to mask timer interrupt

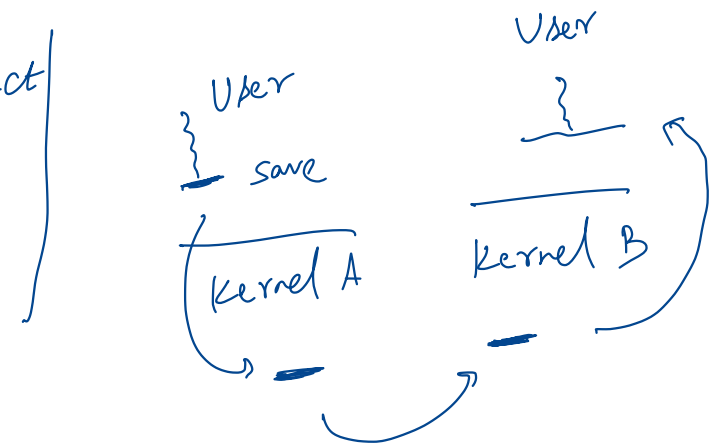
Operating System

Hardware

Program
Process A

timer interrupt
saves regs (A) to kernel stack
jumps to interrupt handler

Trap handler
Scheduler() → select next Process
save kernel regs (A) Proc Struct
restores kernel regs (B)
switch k-stack (B)



Process B

Operating System

Hardware

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

Program
Process A

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

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

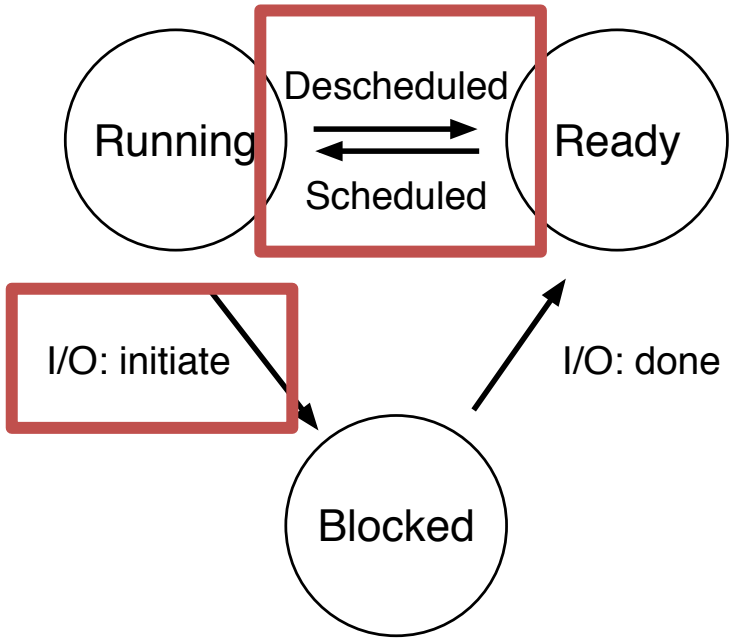
SUMMARY

Process: Abstraction to virtualize CPU

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 CLASS!

NEXT STEPS

Project 1: Due Friday, Sept 13th

Project 2: Out Friday, Sept 13th

Waitlist? Email enrollment@cs.wisc and cc me (will finalize by Thursday)