CS 347M (Operating Systems Minor)

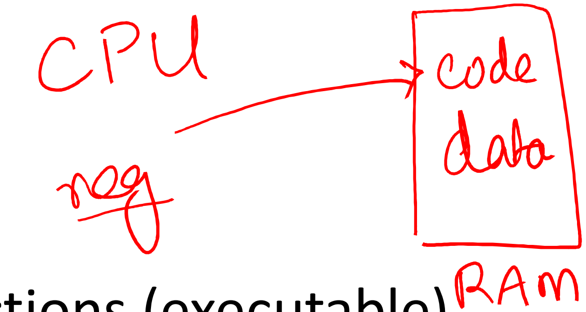
Spring 2022

Lecture 2: Processes

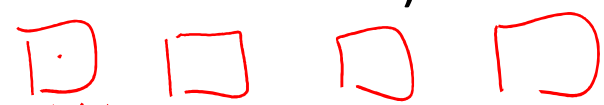
Mythili Vutukuru

CSE, IIT Bombay

Recap: System Hardware



- Users write programs, compile into set of instructions (executable)
- When a program is run, a process is created
  - Code and data of program loaded into memory
  - CPU runs instructions of the program one by one
- CPU capable of executing specific set of instructions, using registers for temporary storage of data (instruction set architecture = ISA)
- Modern CPUs have multiple CPU cores



  - Each CPU core can run one process at any time, but can concurrently handle multiple processes by timesharing CPU core across processes
  - OS schedules processes on CPU cores, switches between processes on a core

Recap: Memory hierarchy

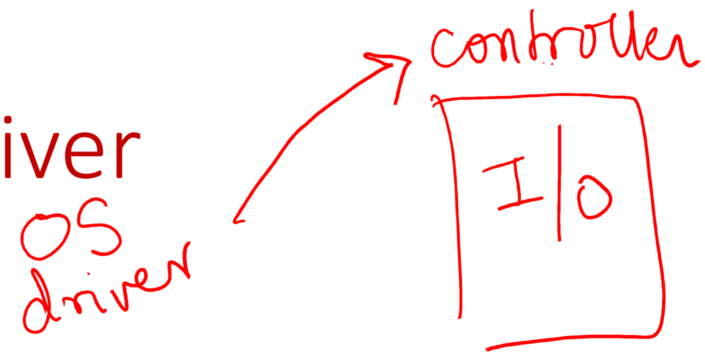
- Hierarchy of storage elements which store instructions and data
    - CPU registers (small number, <1 nanosec)
    - CPU caches (few MB, 1-10 nanosec)
    - Main memory or RAM (few GB, ~100 nanosec)
    - Hard disk (few TB, ~1 millisec)
- Hard disk is non-volatile storage, rest are volatile
    - Hard disk stores files and other data persistently
- As you go down the hierarchy, memory access technology becomes cheaper, slower, less expensive
- Code data of all active processes stored in RAM
    - Recently used code/data may be available in CPU caches

I/O devices

CPU — memory — I/O devices

- A computer system: processor (CPU), main memory, I/O devices connected by system bus
  - Some I/O devices also on special I/O buses (like USB)
  - Bus: a set of wires carrying data between components
- Examples of I/O devices
  - Hard disk: stores information in blocks persistently
  - Network Interface Card (NIC): streams information from external machines
  - Keyboard, mouse: input stream from user
  - Display monitor: stream output to user
- I/O takes a long time compared to CPU, so CPU gives command for I/O and device raises interrupt when I/O completes

Device controller and device driver

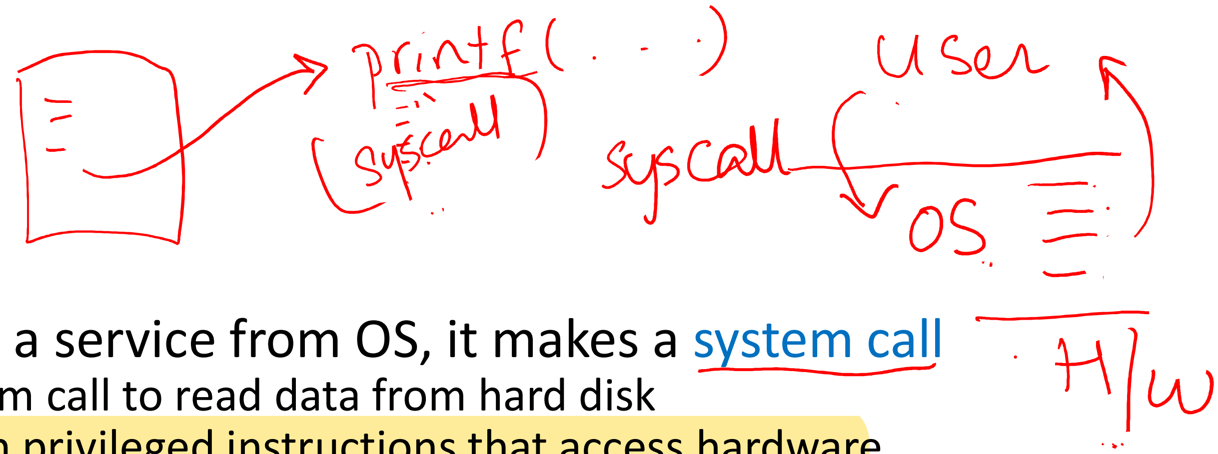


- I/O device is managed by a device controller
  - Microcontroller which communicates with CPU/memory over bus
- Device specific knowledge required to correctly communicate with device controller to handle I/O operations
  - Done by special software called device driver
  - Part of operating system code
- Functions performed by device driver
  - Initialize I/O devices
  - Start I/O operations, give commands to device (e.g., read data from hard disk)
  - Handle interrupts from device (e.g., disk raises interrupt when data is ready)

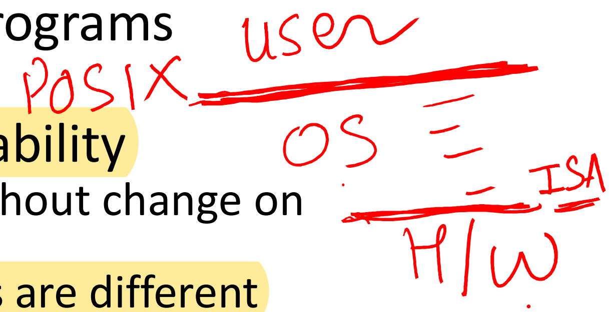
OS manages system hardware for users

- Create and run processes on a system
- Allocate memory to processes
- Schedule processes on CPU cores
- Switch between processes to timeshare CPU
- Give commands to I/O devices to perform I/O operations
- Handle interrupts from I/O devices

System calls



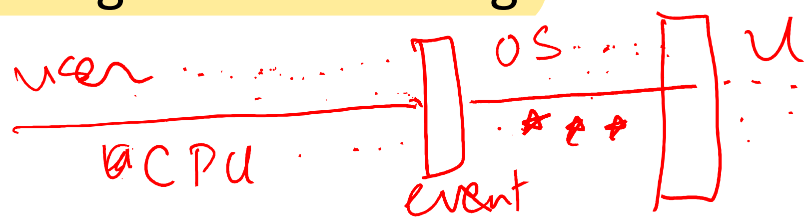
- When user program requires a service from OS, it makes a system call
  - Example: Process makes system call to read data from hard disk
  - Why? User process cannot run privileged instructions that access hardware
  - CPU jumps to OS code that implements system call, and returns back to user code
- System calls supported by an OS form the API to user programs
  - API = application programmer interface



- POSIX API: standard set of system calls defined for portability
  - User program written on one POSIX-compliant OS will run without change on another POSIX-compliant OS
  - However, program may have to be recompiled if architectures are different
- Normally, user program does not call system call directly, but uses language library functions
  - Example: printf is a function in the C library, which in turn invokes the system call to write to screen

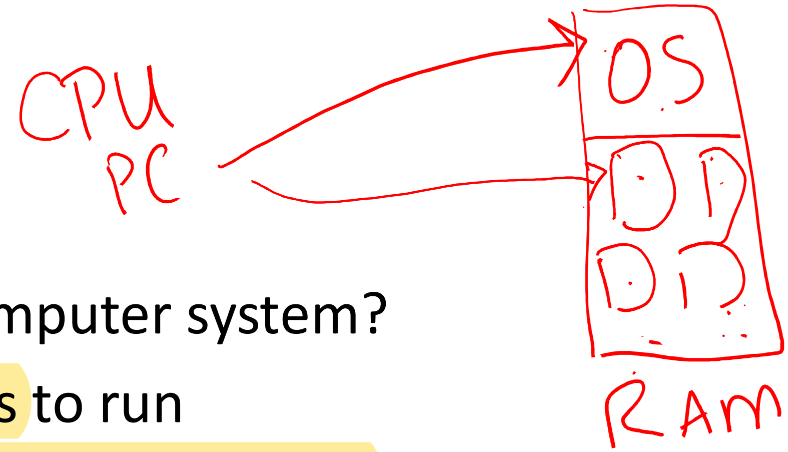
User mode and kernel mode

- Modern CPUs operate at multiple privilege levels. low high
- User programs run in unprivileged user mode of CPU
- CPU shifts to privileged kernel mode for running OS code during:
  - Interrupts: external events
  - System calls: user request for OS services
  - Program faults: errors that need OS attention



- OS code executes in kernel mode, and returns back to user code
  - CPU switches back to low privilege level to run user code
- Sometimes, can return back to user code of a different process
  - OS can context switch to another process for concurrent execution

Booting your system



- What happens when you boot up a computer system?
- Basic Input Output System (BIOS) starts to run
  - Resides in non-volatile memory, sets up all other hardware
- BIOS locates the boot loader in the boot disk (hard disk, USB, ..)
  - Simple program whose job is to locate and load the OS
- Boot loader loads OS in memory and sets it up for execution
- CPU starts executing OS code
- OS exposes a terminal / shell / other interfaces to user

$

- User runs programs, starts processes, which start more processes, …

Questions?

c = a + b
increment (c)
f. incr {
}
OS

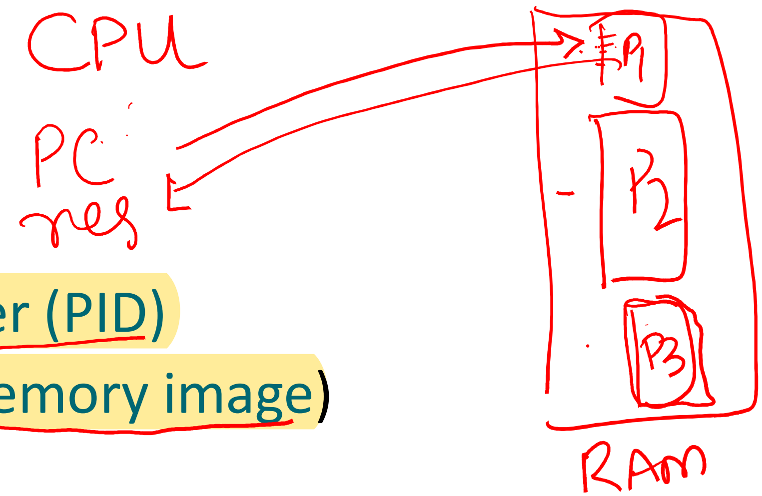
- So far: introduction to system hardware and role of OS
- Next: deep dive into processes

xv6
QEMU
emulator
CPU H/W

s/w
syscall
CPU instruction

What defines a process?

top ps



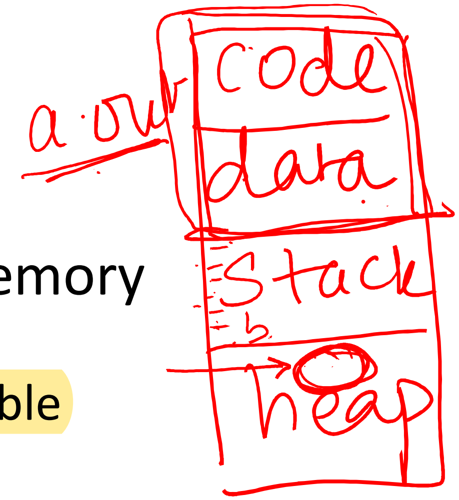
- Every process has a unique process identifier (PID)
- Process occupies some memory in RAM (memory image)
  - Code+data from executable
  - Some memory allocated for runtime use
- The execution context of the process (values of CPU registers)
  - PC has address of instruction of process, some registers have process data
  - Process context is in CPU registers when process is running on CPU
  - Context saved in memory when process is paused, restored when run again
- Ongoing communication with I/O devices
  - Information is maintained about files that are open, ongoing network connections, other active connections to I/O devices

Memory allocation in RAM

- When is memory allocated for process code/data in RAM?
- When OS creates process, memory to store compiled executable allocated in RAM a.out
  - Executable contains code (instructions) in the program, global/static variables in program
- Should we allocate memory for local variables, arguments of functions in executable?
  - No, since we do not now if/how many times the function will be called at runtime
- Similarly, malloc is for dynamic memory allocation at runtime, not compile time a.out

```
int g;

int increment(int a) {
    int b;
    b = a+1;
    return b;
}

main() {
    int x, y;
    x = 1;
    y = increment(x);
     print b X
    int *z = malloc(40);
        print g
}
```

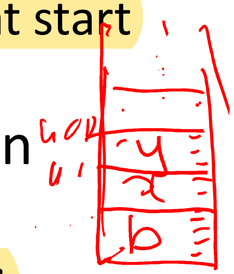
Memory image of a process



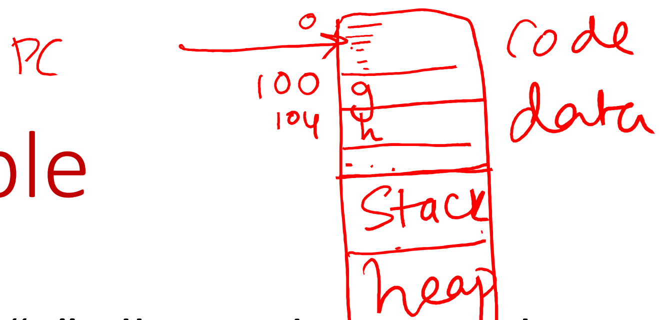
- Memory image of a process: code+data of process in memory
  - Code: CPU instructions in the program
  - Compile-time data: global/static variables in program executable
  - Stack and heap for dynamic memory allocation at runtime
- Stack: memory allocation during function call
  - Function arguments, local variables etc. pushed onto stack (allocated) at start of function, popped (freed) when function returns

new

- Heap: dynamic memory allocation by malloc and related function
  - Malloc returns address on heap of allocated memory chunk
- Heap and stack can grow/shrink as process runs, with help of OS
  - Stack pointer CPU register keeps track of top of stack

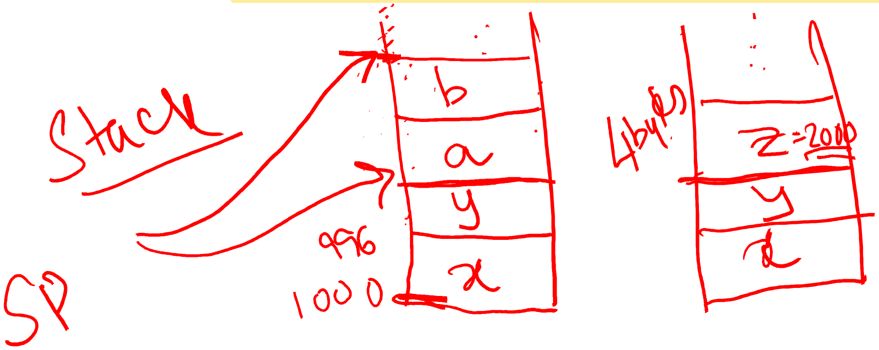


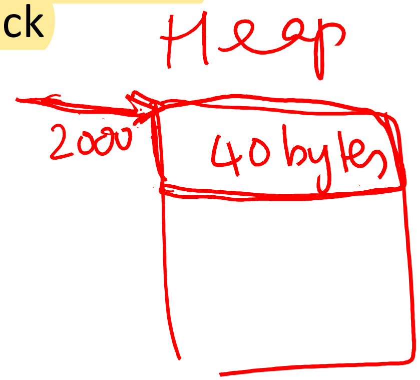
Example



- Variable “g” allocated at compile time (data)
- Main is also considered function, so variables “x”, “y” are allocated on stack when program starts
- When function “increment” is called, variables “a”, “b” are allocated on stack
- When malloc is called, 40 bytes allocated on heap
  - Memory address of 40 bytes on heap is stored in pointer variable “z” which is on the stack

```
int g;

int increment(int a) {
    int b;
    b = a+1;
    return b;
}

main() {
    int x, y;
    x = 1;
    y = increment(x);

    int *z = malloc(40);

}
```



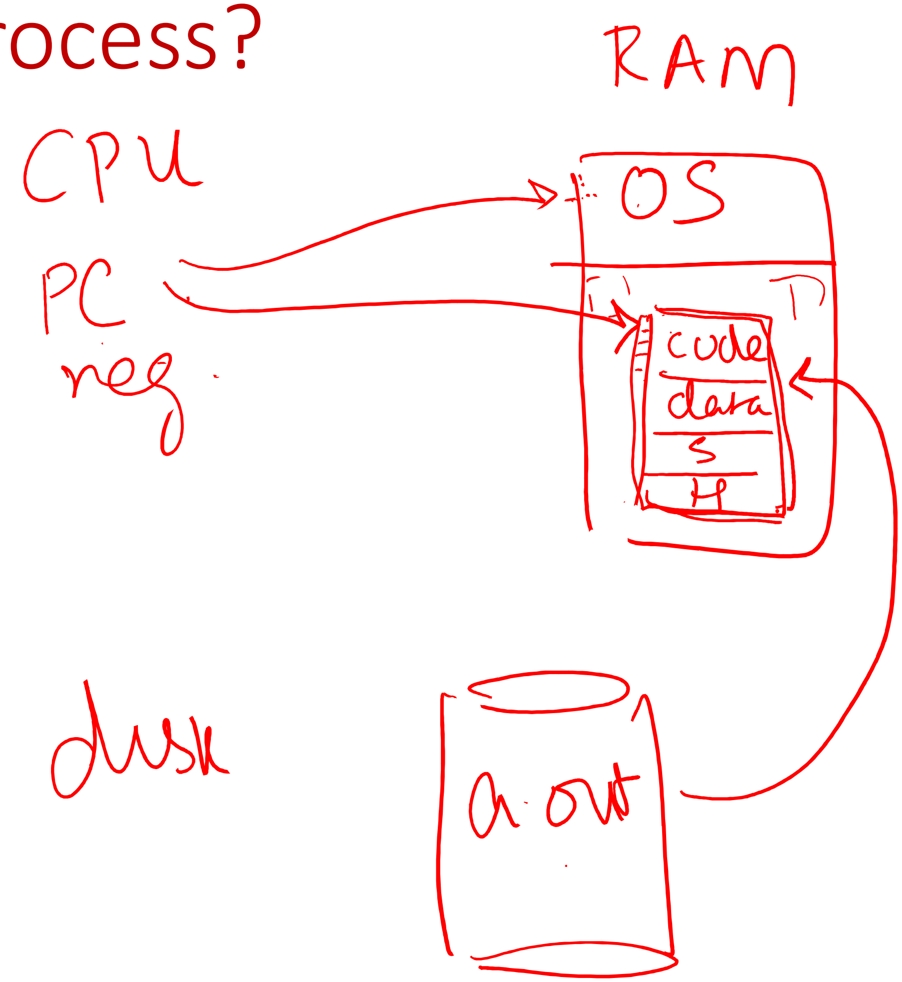


How does OS create a process?

- Allocates memory and creates memory image
  - Loads code, data from disk exe
  - Creates runtime stack, heap
- Opens basic files
  - STD IN, OUT, ERR

initialization

- Initializes CPU registers
  - PC points to first instruction
- Process starts to run
  - OS steps in as needed for interrupts, system calls, …



States of a process

- OS manages multiple active processes at the same time. An active process can be in one of the following situations.
- Running: currently executing on CPU
  - CPU registers contain context of process
- Blocked/suspended/sleeping: process cannot run for some time
  - Example: process has requested data from disk, command issued, but process cannot proceed until the data from disk is available
- Ready/runnable: ready to run but waiting for OS scheduler to switch the process in
  - Many processes can be ready but scheduler can only run one on a CPU core
- Context of blocked and ready processes is saved in memory, so that they can continue to run later on

Process State Transitions

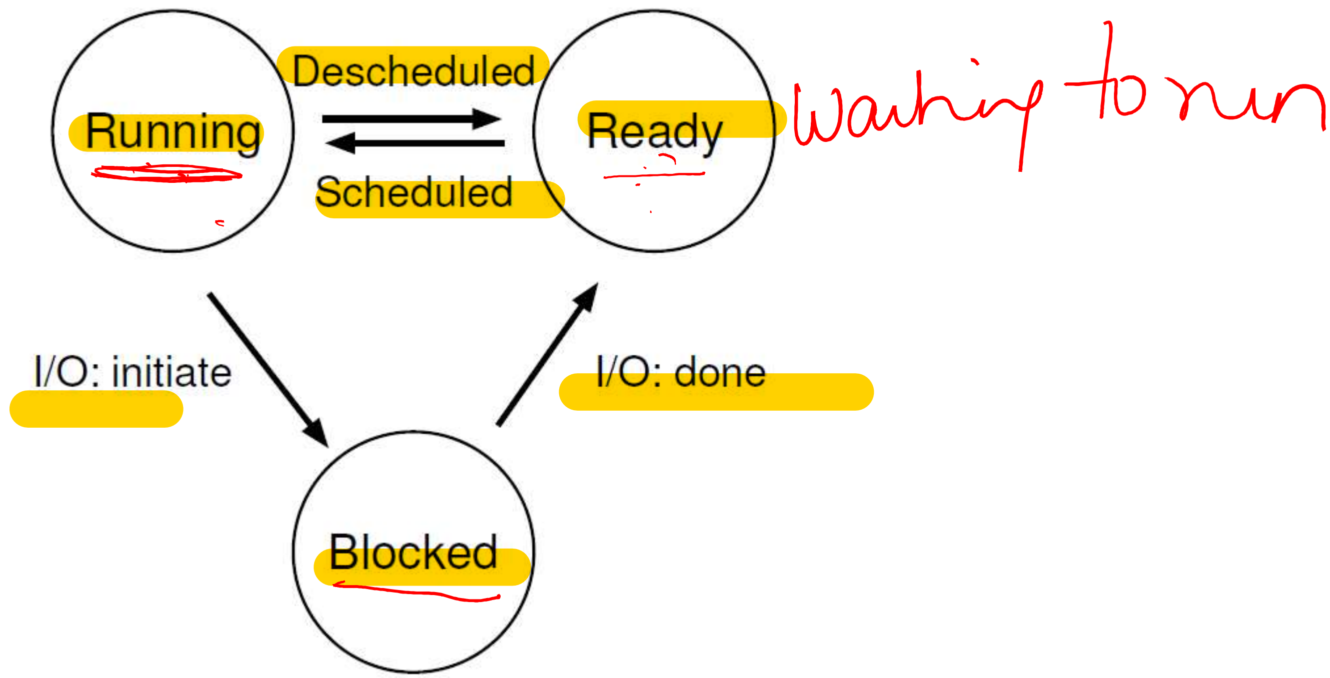


Figure 4.2: **Process: State Transitions**

Example: process state transitions

P1 P2

running ready

OS running

- Consider a system that has two user processes P1 and P2
  - Initially P1 is running, P2 is ready and awaiting its turn

    blocked
  - P1 opens a file and wants to read some bytes from disk via a system call
  - OS handles the system call and gives command to disk, but data is not available immediately

    OS

    intr
  - Process P1 is moved to blocked state, OS switches to process P2

    ready
  - Process P2 runs for some time, and then an interrupt occurs from disk
  - CPU jumps to OS which handles interrupt, P1 is moved to ready state

    running
  - OS can continue to run P2 again after interrupt and OS scheduler switches to ready process P1 later on after some time

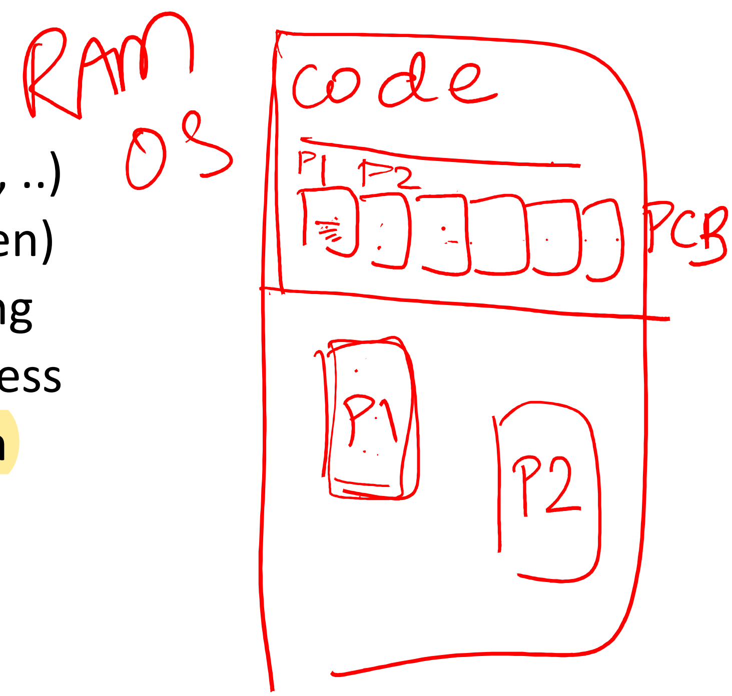
running

Process control block (PCB)

OS data structure

- All information about a process is stored in a data structure called the process control block (PCB)
  - Process identifier (PID)
  - Process state (running, ready, blocked, terminated, ..)
  - Pointers to other related processes (parent, children)
  - Saved CPU context of process when it is not running
  - Information related to memory locations of a process
  - Information related to ongoing I/O communication
  - …

list of all PCB



System call API for processes

P1

P2

- Next lecture: process-related systems calls
  - How to create processes?
  - How to terminate processes?
- `fork()` creates a new child process
  - All processes are created by forking from a parent
  - The `init` process is ancestor of all processes, which creates shell, which creates other user processes
- `exec()` makes a process execute a given executable
- `exit()` terminates a process
- `wait()` causes a parent to block until child terminates and clean it up

Questions?

- So far: the concept of a process
- Next: processes in xv6

PCB in xv6: struct proc

- PCB is known by different names in different OS
  - struct proc in xv6
  - task_struct in Linux

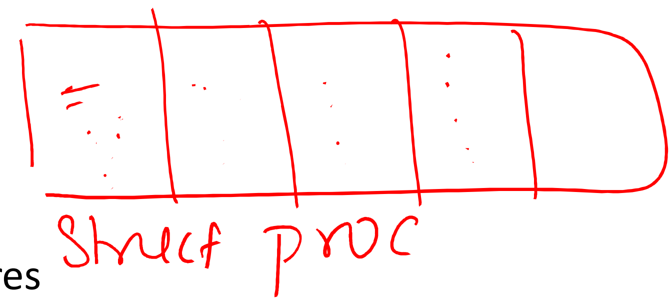
- Page 23, process structure and process states

ready

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// 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 process
  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)
};
```

Process table (ptable) in xv6

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```



- ptable: Fixed-size array of all processes
  - Real kernels have dynamic-sized data structures
- CPU scheduler in the OS loops over all runnable processes, picks one, and sets it running on the CPU

```
    // 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;
```

Process state transition examples in xv6

- A process that needs to sleep (e.g., for disk I/O) will set its state to SLEEPING and invoke scheduler
- A process that has run for its fair share will set itself to RUNNABLE (from RUNNING) and invoke scheduler
- Scheduler will once again find another RUNNABLE process and set it to RUNNING

run for too long

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

ready

read

```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  if(p == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");

  // Must acquire ptable.lock in order to
  // change p->state and then call sched.
  // Once we hold ptable.lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup runs with ptable.lock locked),
  // so it's okay to release lk.
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;

  sched();
```

for (. . . . . .)

OS code in C or assembly?

- OS is also like any other program run by CPU, but it is the most important program that manages other programs
  - OS code mostly written in a high-level language like C, compiled into executable, loaded at boot time
- But some parts of OS are written directly in assembly language or CPU instructions that the hardware can understand?
  - Why not write everything in C? Not possible to express certain low level actions performed by OS in high level language
- Basic understanding of x86 assembly code required for understanding xv6 OS code in this course

Reading xv6 for this course

- In this course, only a high level understanding of xv6 is expected
  - For theory as well as labs
- Read C code, comments in code, lecture videos and slides
- No need to read and understand assembly, explanations will be provided for assembly code
- However, it will be useful to understand simple assembly

Reference: x86 registers

- General purpose registers: store data during computations (eax, ebx, ecx, edx, esi, edi)
- Pointers to stack locations: base of stack (ebp) and top of stack (esp)
- Program counter or instruction pointer (eip): next instruction to execute
- Control registers: hold control information or metadata of a process (e.g., cr3 has information related to memory of process)
- Segment registers (cs, ds, es, fs, gs, ss): information about segments (related to memory of process)

Reference: x86 instructions

- Load/store: *mov src, dst*
  - *mov %eax, %ebx* (copy contents of eax to ebx)
  - *mov (%eax), %ebx* (copy contents at the address in eax into ebx)
  - *mov 4(%eax), %ebx* (copy contents stored at offset of 4 bytes from address stored at eax into ebx)
- Push/pop on stack: changes esp
  - *push %eax* (push contents of eax onto stack, update esp)
  - *pop %eax* (pop top of stack onto eax, update esp)
- *jmp* sets eip to specified address
- *call* to invoke a function, *ret* to return from a function
- Variants of above (*movw, pushl*) for different register sizes