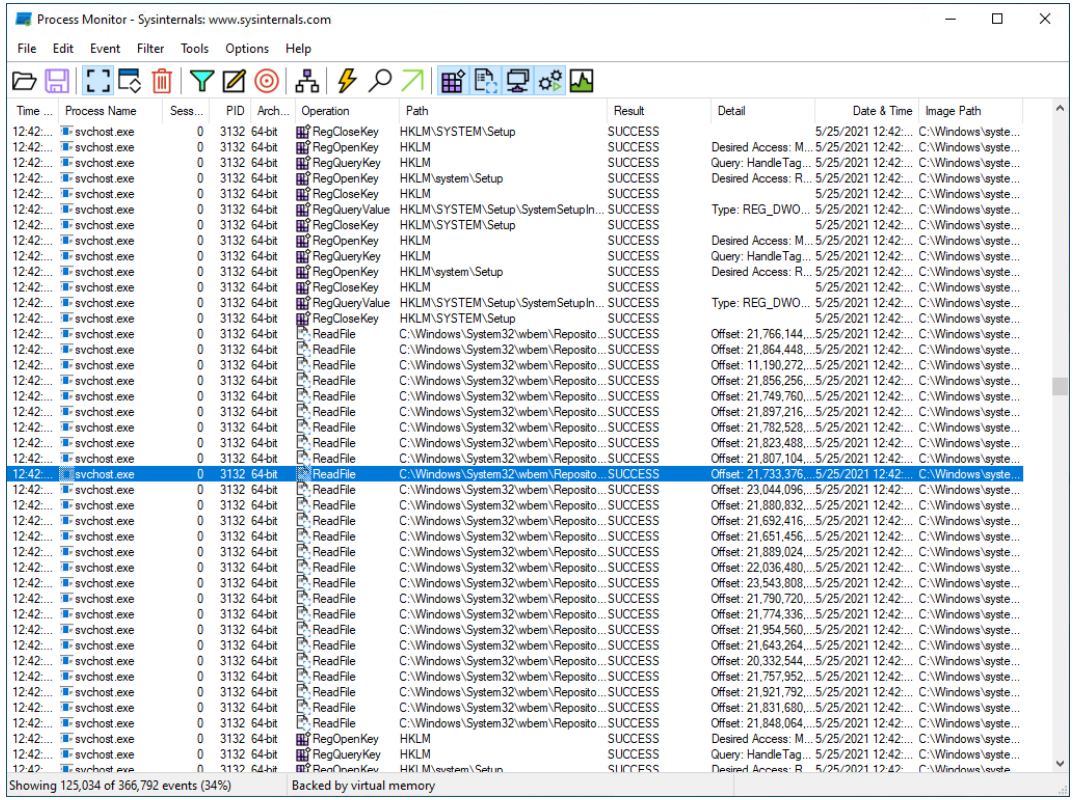
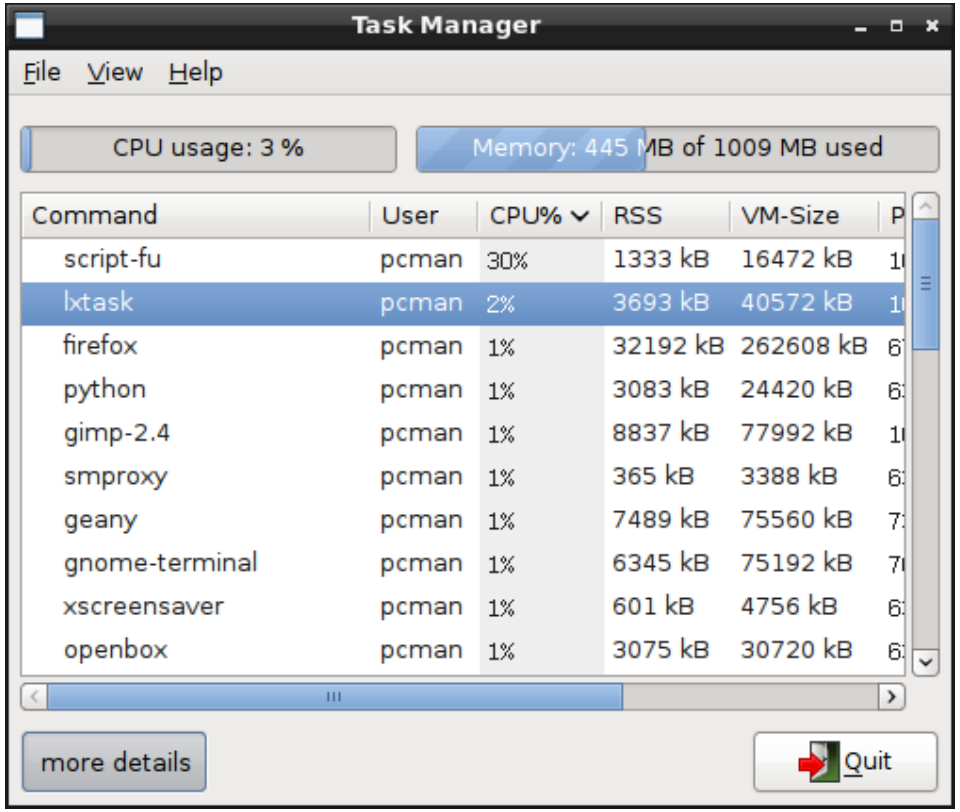
# CS3.301: Operating Systems and Networks
# Lecture 2 & 3: Process Virtualization

INSTRUCTOR: DR. DEEPAK GANGADHARAN

# Many processes run at the same time!



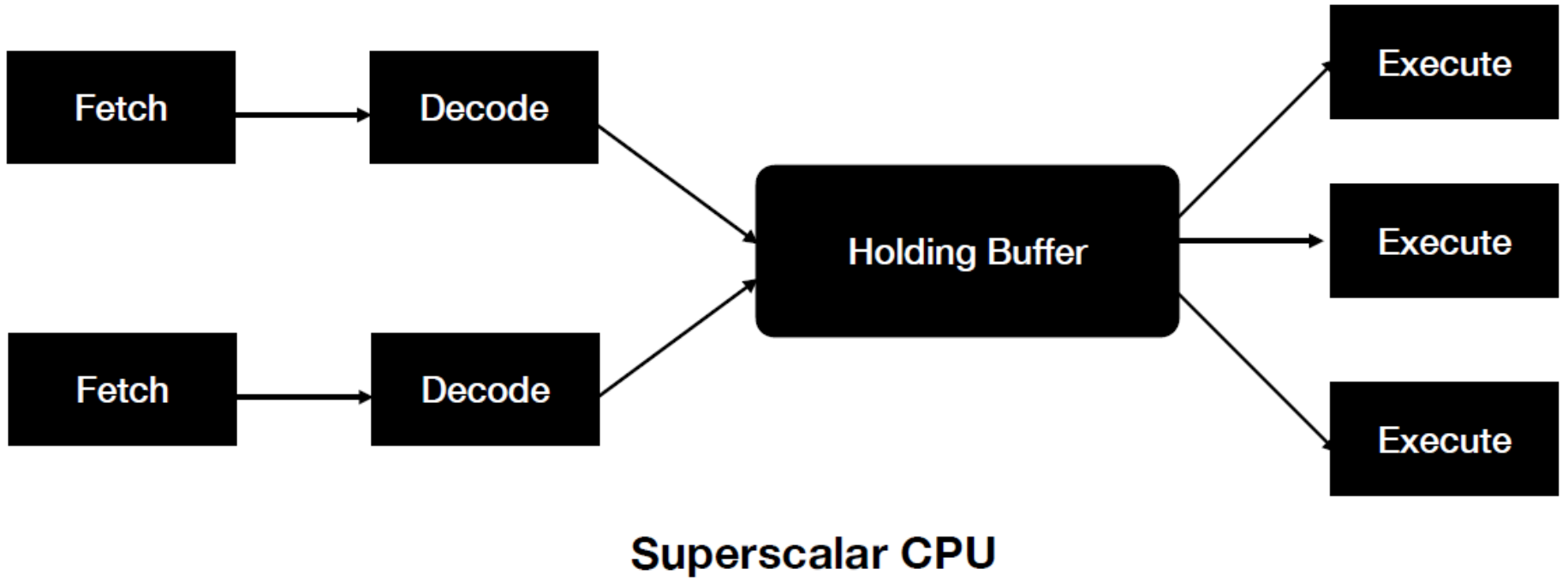
# What is a Process?

- Program is a code.
- Processes are running instance of the program.
- For one program, there can be multiple processes created.

# Process and CPU Virtualization

- Each process feels that it has its own CPU
- We only have one CPU in single core machines, but there are multiple processes even there!
- How is CPU handling this?
  How is the illusion created that there are many CPUs?
- OS can create this illusion by virtualizing the CPU.

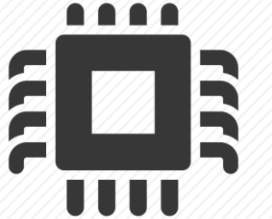
# Some prerequisites




Superscalar CPU

# CPU Virtualization

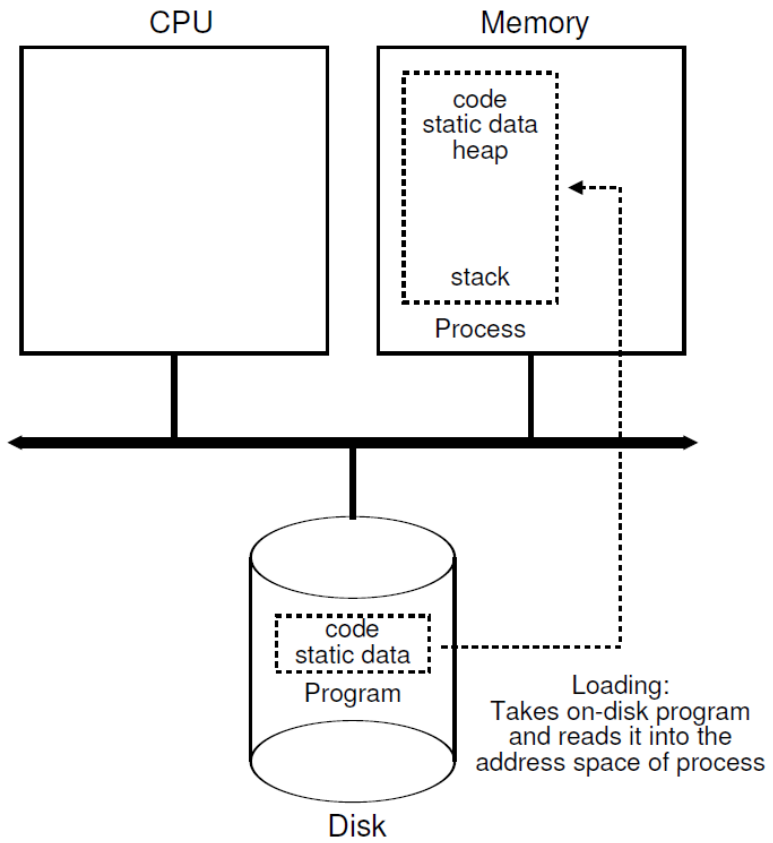
- Need support from hardware
  - Low level *mechanisms* for functionalities like *context switch*
  - Enables OS to run processes on a *time-sharing* mechanism.



- Need intelligence in software
  - Policies/Algorithms that make decision within the OS
  - Given a number of processes, which process should the OS run?
  - The policy is called *Scheduling Policy*

# What is a Process?

- Program is code sitting idle on the disk
- When OS takes an instance of the program and puts in memory for execution, it becomes a process



# What constitutes a Process?

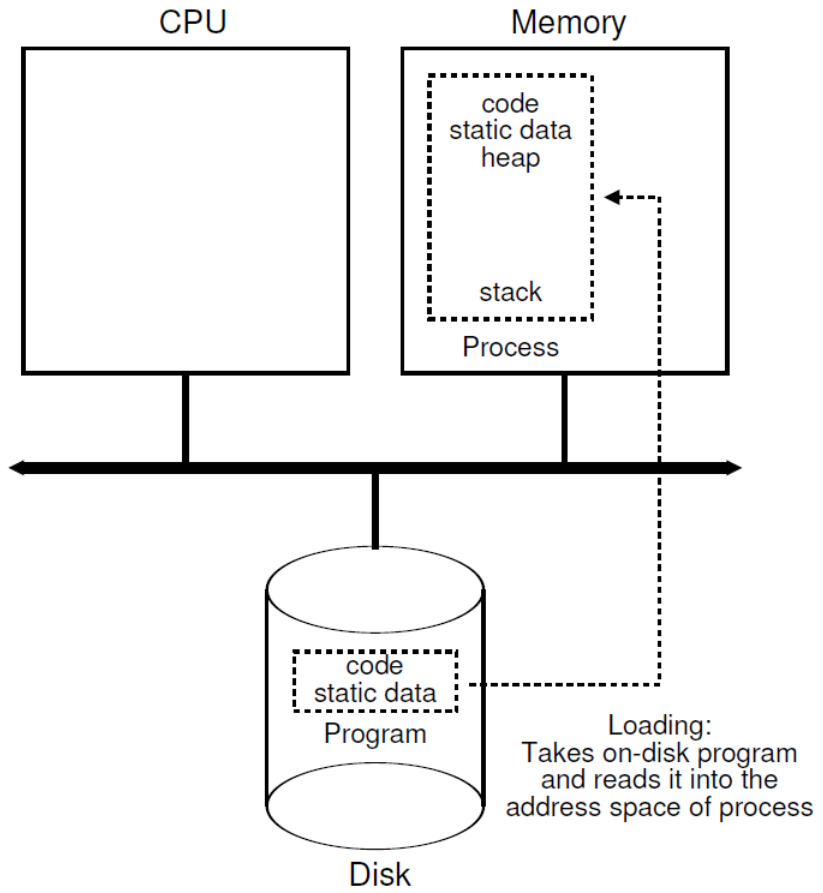
- We need to understand machine state – what the program can read or update while running.
- Can you name one component in the machine state?
- Memory (Address) space – Instructions and data for a process lie in memory
- Registers – Instructions explicitly read or update some registers
- Program counter – which instruction in the process is executed
- Stack pointer – local variables, functions and return addresses
- Persistent Storage – I/O information or list of files that are currently open

# What constitutes a Process?

- **Unique identifier (Process ID)**
- **Memory Image**
  - Code and data (static)
  - Stack and Heap (dynamic)
- **CPU context: Registers**
  - Program Counter
  - Operand registers
  - Stack pointer
- **File Descriptors**
  - Pointers to open files and devices

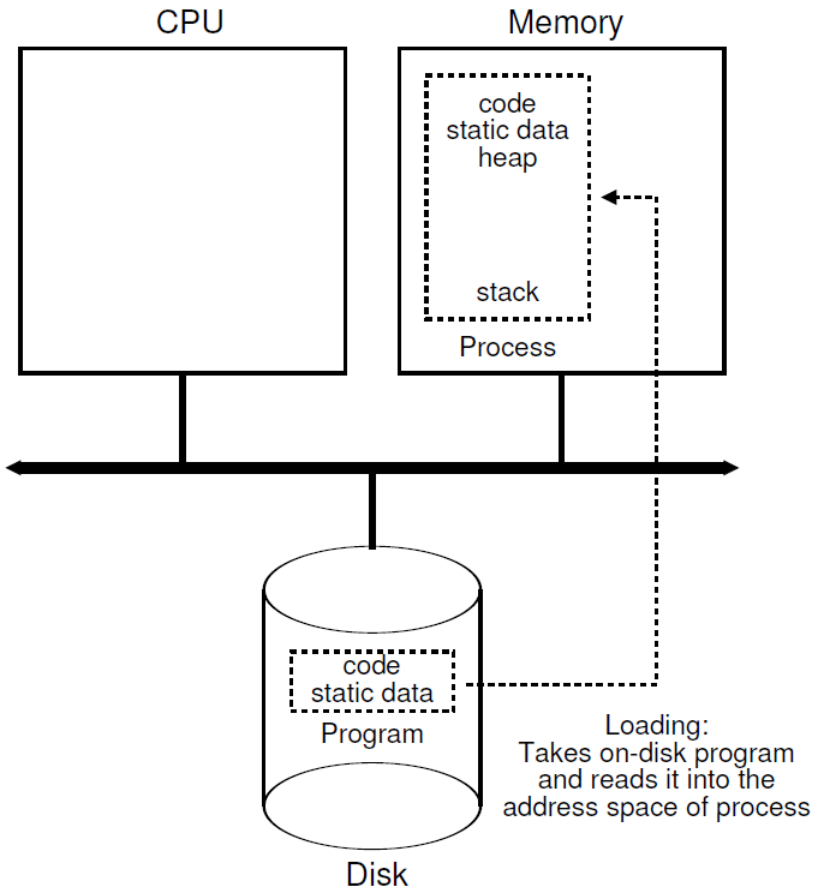
# Creation of a Process by OS

- **Load program into memory**
  - Initially program resides on the disk
  - Modern OSes do lazy loading
  - Done using paging and swapping
- **Allocated runtime stack**
  - For local variables
  - Fuction parameters and return arguments



# Creation of a Process by OS

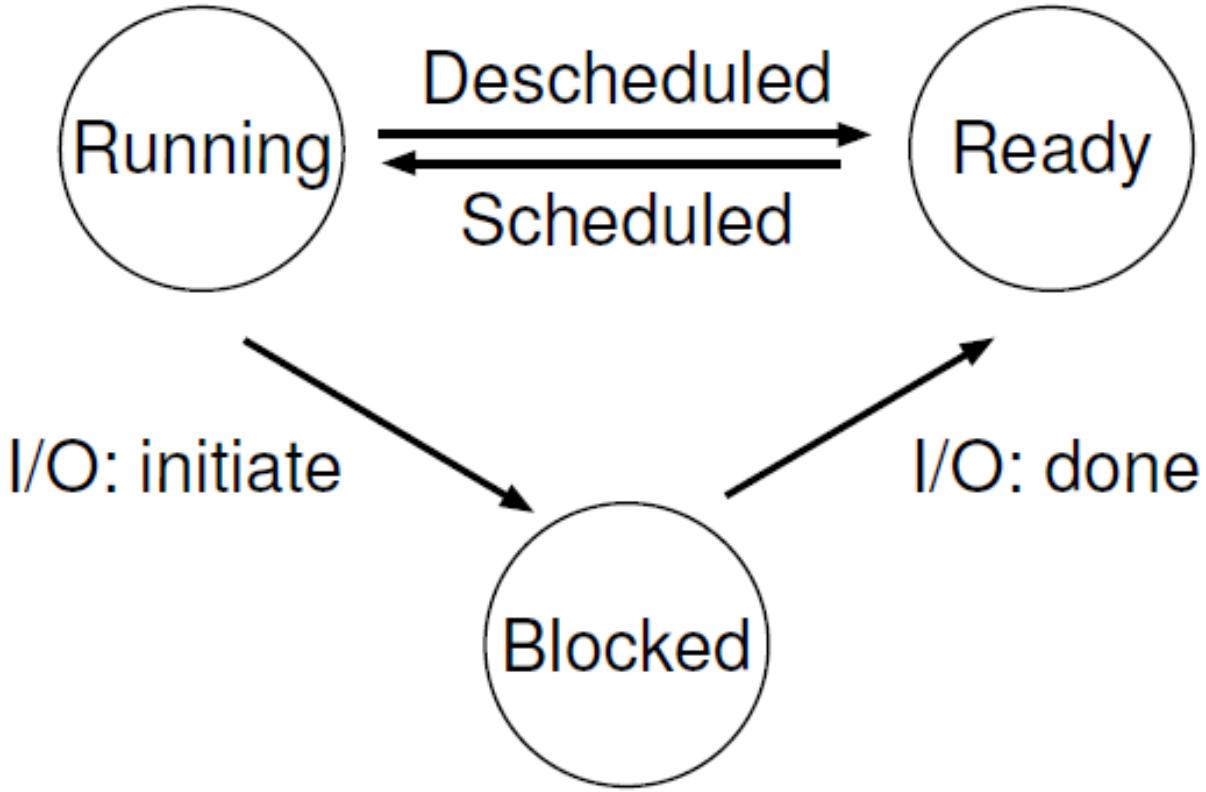
- **Creating Program heap**
  - Used for dynamically allocated data
  - malloc() and free()
- **Basic file setup**
  - STDIN, OUT, ERR
- **Initialise CPU registers**
  - PC to first instruction
- **Start the Program**



# States of the Process

- Process can be in one of the following states
  - Running – Process is running on the processor
  - Ready – Process is ready to run but OS has chosen not to run at the moment
  - Blocked – Not ready to run as it has performed some kind of operation and is waiting for some event
  - What can put a process in blocked state?

# Process State Transition



# An Example: CPU only

| Time | Process 0 | Process 1 | Status |
|---|---|---|---|
| 1 | Running | Ready | |
| 2 | Running | Ready | |
| 3 | Running | Ready | Process 0 now done |
| 4 | - | Running | |
| 5 | - | Running | Process 1 now done |

# An Example: CPU and I/O

| Time | Process 0 | Process 1 | Status |
|---|---|---|---|
| 1 | Running | Ready | |
| 2 | Running | Ready | |
| 3 | Running | Ready | Process 0 initiates I/O |
| 4 | Blocked | Running | Process 0 blocked → Process 1 runs |
| 5 | Blocked | Running | |
| 6 | Blocked | Running | |
| 7 | Ready | Running | I/O done |
| 8 | Ready | Running | Process 1 now done |
| 9 | Running | - | |
| 10 | Running | - | Process 0 now done |

# How to store Process state?

- OS uses data structures.
- To track each process state, OS keeps a **process list** for all ready processes.
- Additional info to track running process and blocked processes.
- Individual structure for a process called Process Control Block (PCB).

# Xv6: Unix-like teaching OS developed by MIT

https://pdos.csail.mit.edu/6.828/2012/xv6.html

# Processor Structure in xv6

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
    int eip;    // Index pointer register
    int esp;    // Stack pointer register
    int ebx;    // Called the base register
    int ecx;    // Called the counter register
    int edx;    // Called the data register
    int esi;    // Source index register
    int edi;    // Destination index register
    int ebp;    // Stack base pointer register
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };
```

# Process Structure in xv6

```
// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem;                  // Start of process memory
    uint sz;                    // Size of process memory
    char *kstack;               // Bottom of kernel stack
                                // for this process
    enum proc_state state;      // Process state
    int pid;                    // Process ID
    struct proc *parent;        // Parent 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
    struct context context;     // Switch here to run process
    struct trapframe *tf;       // Trap frame for the
                                // current interrupt
};
```

# Thank You!!