Processes

Mythili Vutukuru

CSE, IIT Bombay

Recap: OS manages hardware for users

- Creates and runs processes on a system
- Allocates memory to processes, manages virtual address space
- Schedules processes on CPU, and switches between processes to timeshare CPU
- Handles interrupts from I/O devices, and other events

The process abstraction

- Process is a running program
- When program is run, OS creates new process, allocates memory, initializes CPU context, and starts process in user mode
- User program runs on CPU normally, unless OS needs to step in for system calls, interrupts, …

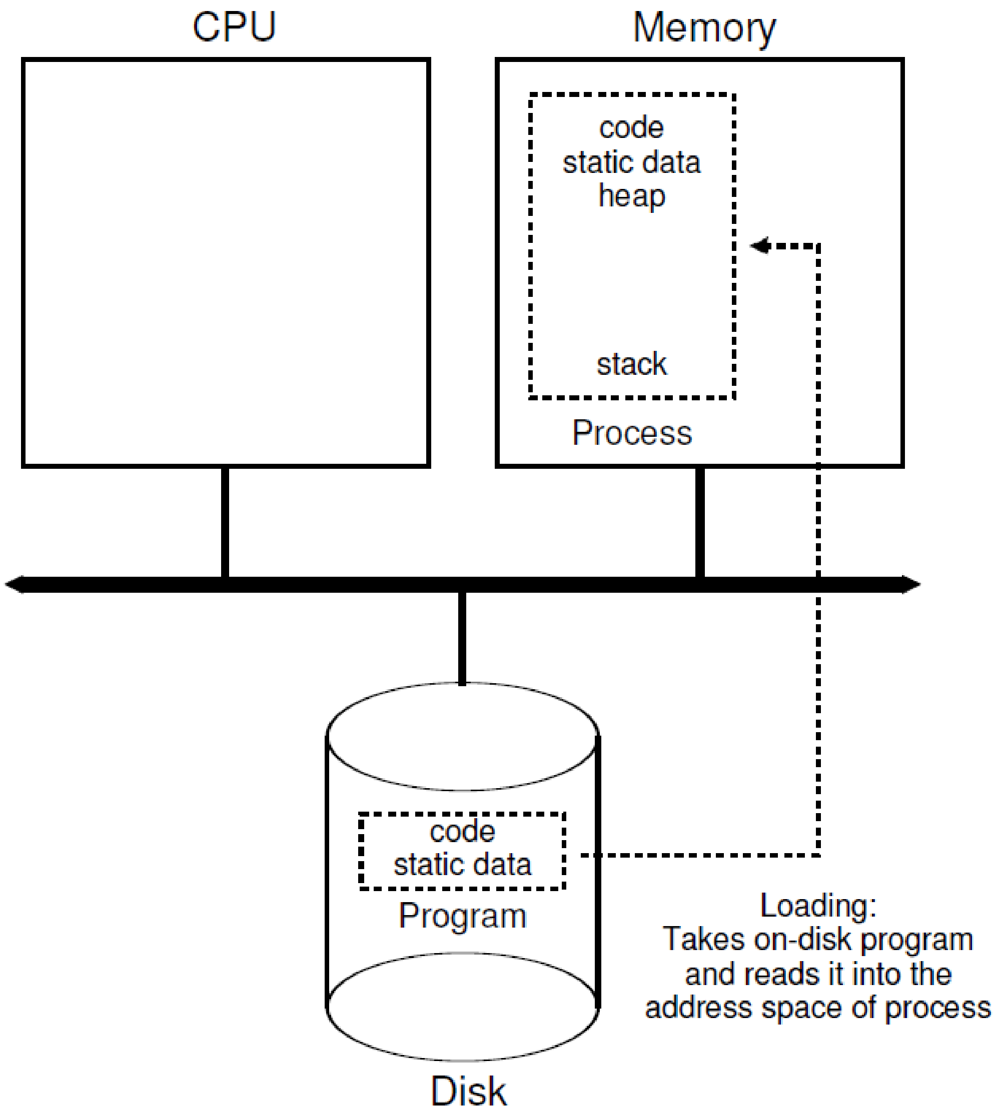

Figure 4.1: **Loading: From Program To Process**

Image credit: OSTEP

What defines a process?

- Every process has a unique process identifier (PID)
- Process occupies some memory in RAM (memory image)
  - Code+data from executable
  - Stack+heap 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

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

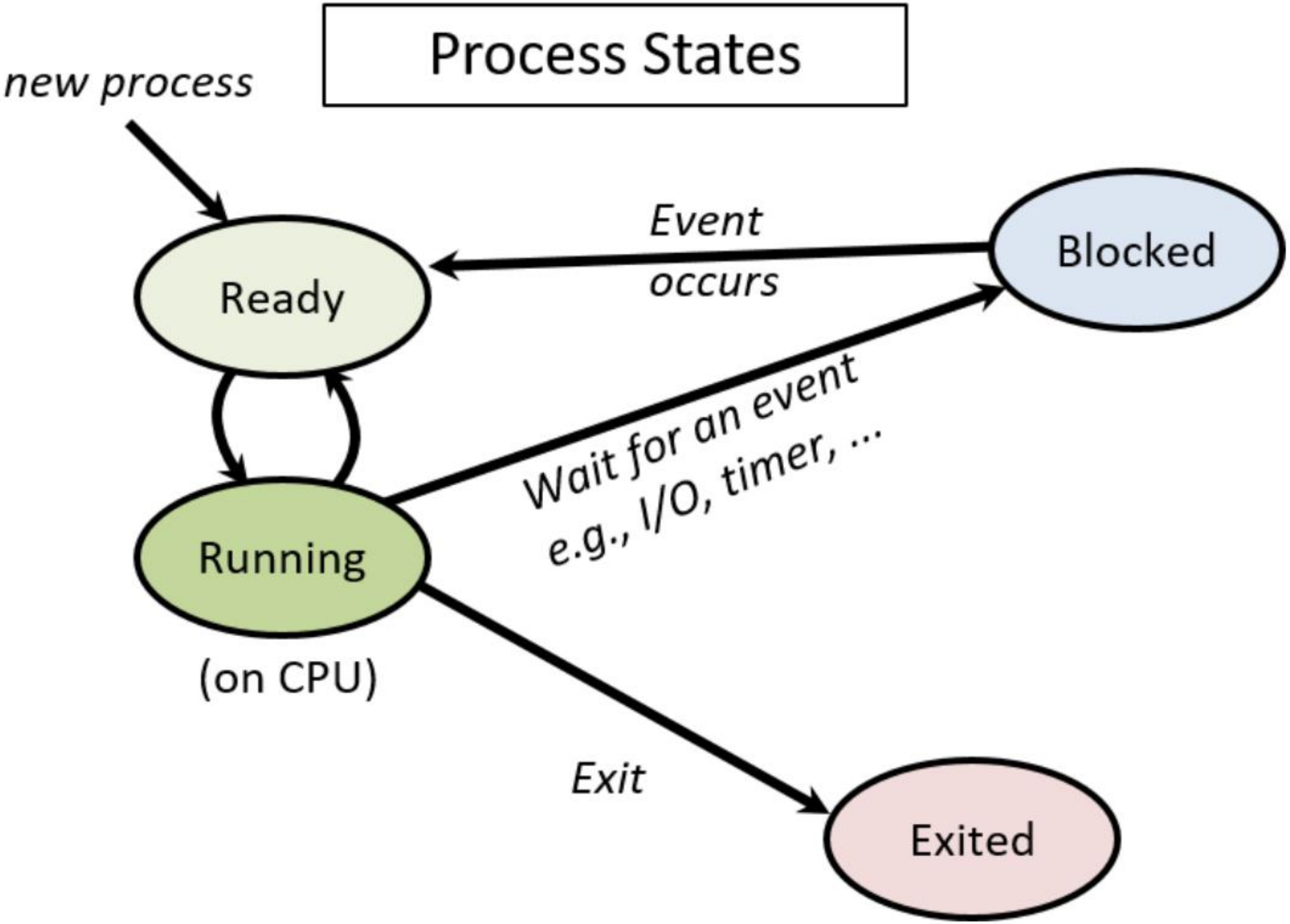

Figure 1. The states of a process during its lifetime

Image credit: Dive into Systems

Example: process state transitions

- Consider a system that has two user processes P0 and P1
  - Initially P0 is running, P1 is ready and awaiting its turn
  - P0 wants to read a file from disk via a system call
  - OS handles the system call and gives command to disk, but data is not available immediately
  - Process P0 is moved to blocked state, OS switches to process P1
  - Process P1 runs for some time, and then an interrupt occurs from disk
  - CPU jumps to OS which handles interrupt, P0 is moved to ready state
  - OS can continue to run P1 again after interrupt and OS scheduler switches to ready process P0 later on after some time

Example: process state transitions

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done

Figure 4.4: **Tracing Process State: CPU and I/O**

Image credit: OSTEP

Process State Transitions

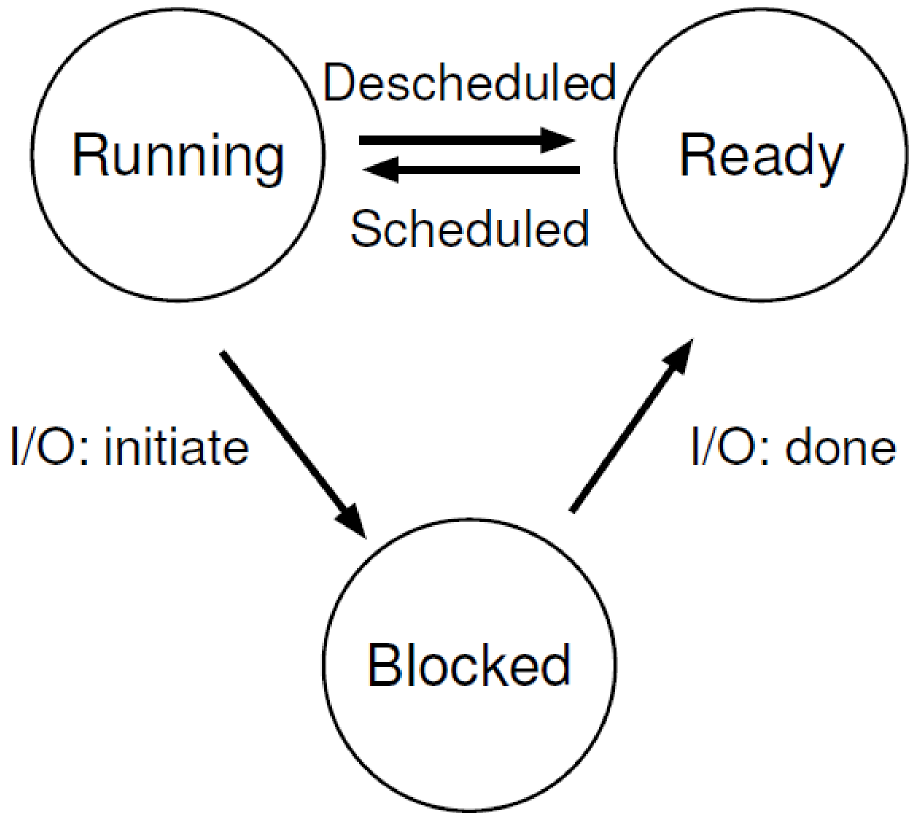

Figure 4.2: **Process: State Transitions**

Image credit: OSTEP

Process control block (PCB)

- All information about a process is stored in a kernel 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
  - …
- PCB is known by different names in different OS
  - struct proc in xv6
  - task_struct in Linux

PCB in xv6: struct proc

```c
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)
};

```

struct proc: page table

- Every instruction or data item in the memory image of process (code/data, stack, heap, etc.) has an address
  - Virtual addresses, starting from 0
  - Actual physical addresses in memory can be different (all processes cannot store their first instruction at address 0)
- Page table of a process maintains a mapping between the virtual addresses and physical addresses
- Page table used to translate virtual addresses to physical addresses

struct proc: kernel stack

- Stack to store CPU context when process jumps to kernel mode from user mode, or when process is context switched out
  - Why separate stack? OS does not trust user stack
  - Separate area of memory in the kernel, not accessible by regular user code
  - Linked from struct proc of a process

struct proc: list of open files

- Array of pointers to open files
  - When user opens a file, a new entry is created in this array, and the index of that entry is passed as a file descriptor to user
  - Subsequent read/write calls on a file use this file descriptor to refer to the file
  - First 3 files (array indices 0,1,2) open by default for every process: standard input, output and error
  - Subsequent files opened by a process will occupy later entries in the array

Process table (ptable) in xv6

- Ptable in xv6 is a fixed-size array of all processes
- Real kernels have dynamic-sized data structures

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

CPU scheduler in xv6

- The OS loops over all runnable processes in ptable, 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: example in xv6

- A process that needs to sleep (e.g., for disk I/O) will set its state to SLEEPING and invoke scheduler
- Scheduler will run its loop and find another ready process to run

```c
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();
```

Booting

- 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
    - Present in the first sector of the boot disk
    - Combination of assembly and C code
- Boot loader sets up CPU registers suitably, loads kernel image from disk to memory, transfers control to kernel
- OS code starts to run, exposes terminal to user, user starts programs

Booting real systems

- Bootloader must fit into 512 bytes (first sector of boot disk) to be found easily by BIOS
- Bootloaders for simple/old OS could fit into one sector, but no longer the case for modern OS
- Real life bootloaders are complex, need to read a large kernel image from disk and network, do not fit into 512 bytes
- Real life booting is two step process: BIOS loads simple bootloader, which loads a more complex bootloader, which then loads the OS

Summary

Three ready processes
Three ready processes
See how they run
See how they run
They all sat in the scheduler's queue
The scheduler sent them out to execute
They made an I/O request and went to sleep
Became ready to run after interrupts beeped
Did you ever see such a sight in your life
As three ready processes

(To be sung in the tune of the nursery rhyme “Three Blind Mice”)