Chapter 4: Processes

Adam Disney

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Processes

- A process is a running program.
  - A program is just a bunch of instructions.
  - A process is a program that has state attached to it.
- We usually want to have more processes than we have CPUs.
  - How does we get this to work?

Crux of the Problem

THE CRUX OF THE PROBLEM:
HOW TO PROVIDE THE ILLUSION OF MANY CPUS?

Although there are only a few physical CPUs available, how can the OS provide the illusion of a nearly-endless supply of said CPUs?

Virtualization of the CPU

- The operating system virtualizes the CPU by running one process, stopping it and running another, and so forth.
  - This basic technique is known as time-sharing.
  - We need some mechanisms to enable this.
  - We'll learn how to implement a context switch.
- And we need policies that use the mechanisms intelligently.
  - We'll look at scheduling policies

Machine State (The Parts of a Process)

- Memory
  - Instructions lie in memory
  - The data that the process reads/writes is in memory
  - The memory that the process can address is its **address space**
- Registers
  - Many instructions read/update registers and are thus important to the machine state
  - There are some special registers that are particularly important.
    - Program Counter / Instruction Pointer (PC/IP)
      - Tells us which instruction of the program will execute next
    - Stack Pointer and Frame Pointer
      - Manages the stack for function parameters, local variables, and return addresses
- I/O Information
  - Processes interact with persistent storage devices
  - Which are open? What is our current offset? Etc.

Process API

- This generic interface is provided in some form on any modern operating system.
  - Create – Create a new process
  - Destroy – Destroy a process forcefully
  - Wait – Wait for process to stop running
  - Miscellaneous Control – Suspend/Resume, etc.
  - Status – Give info about process like running time, state, etc.

Process Creation

- Load code and static data into memory
  - Loaded into the processes address space
  - The code is on storage (disk/flash/etc) in some executable format
  - Early OSes loaded entire program into memory before executing
  - Modern OSes does this lazily
    - Loading the code/data only as needed
    - This requires paging/swapping (later topic)

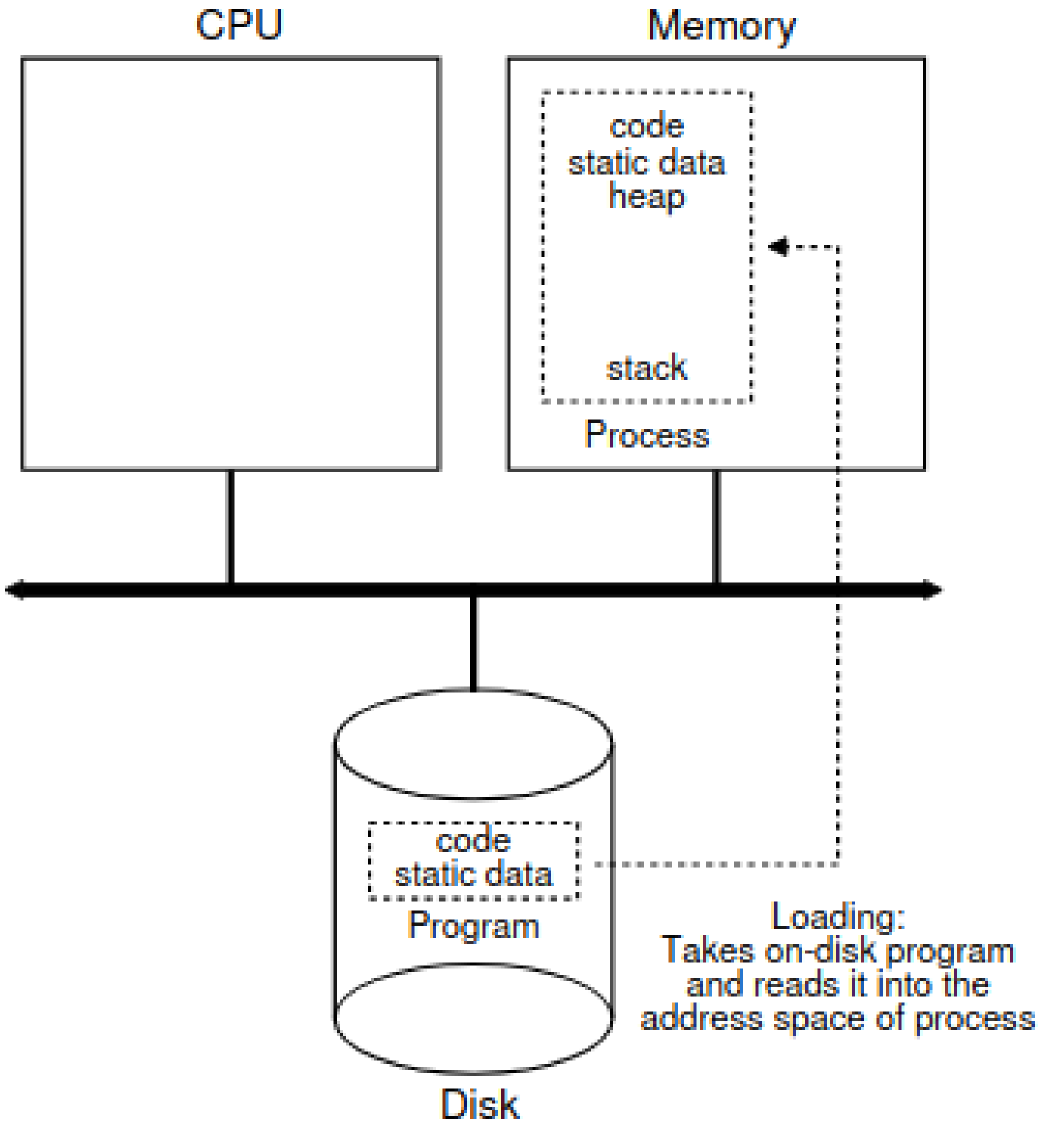

Figure 4.1: **Loading: From Program To Process**

Process Creation

- Allocate memory for the program's stack and heap
- Initialize some I/O
  - Default file descriptors stdout, stderr, stdin
- Start execution at main() and transferring control of the CPU to the new process

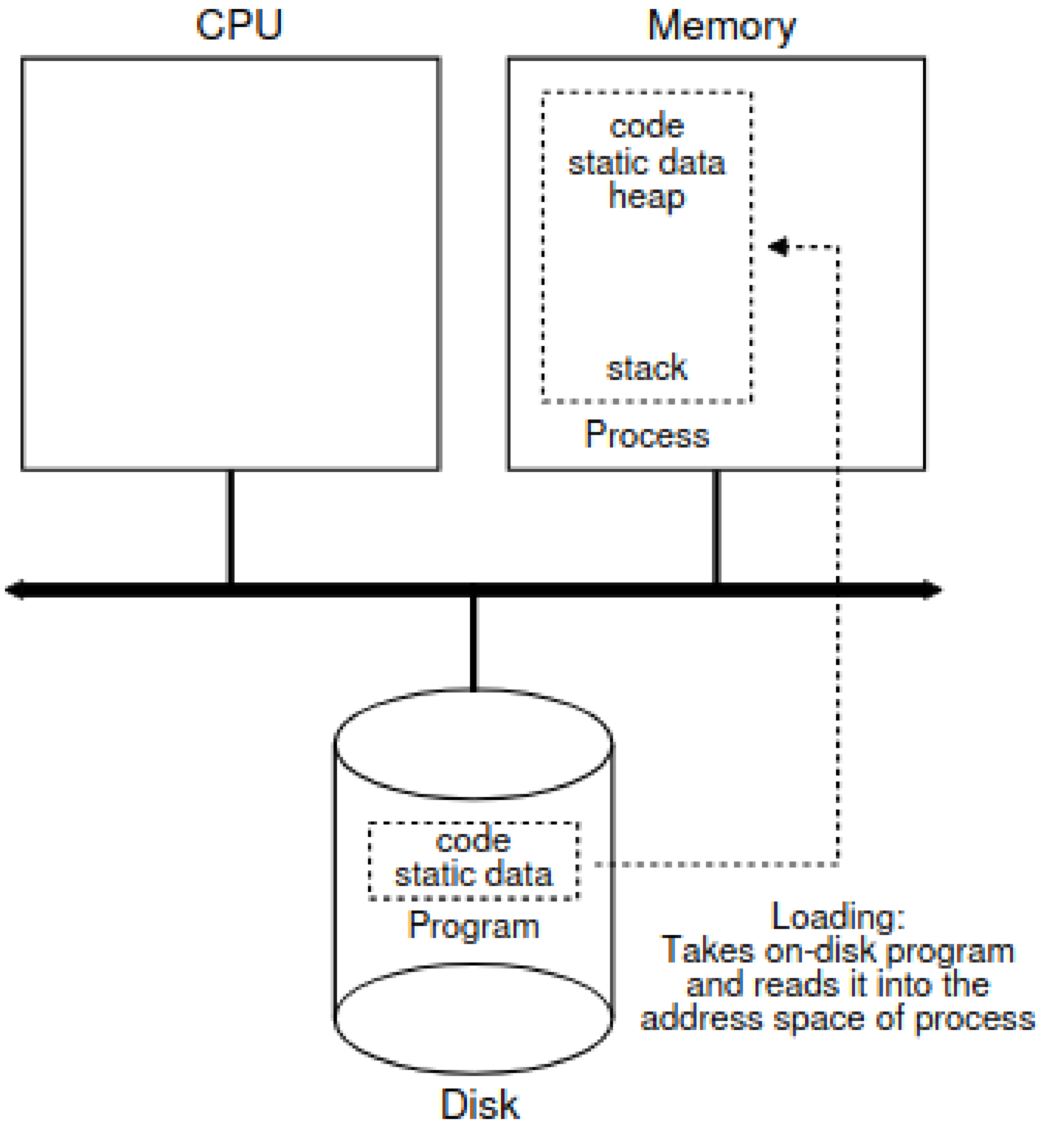

Figure 4.1: **Loading: From Program To Process**

Process States

- Running – Executing instructions
- Ready – Ready to run but is currently not running
- Blocked – Waiting for an event to happen (commonly I/O request)

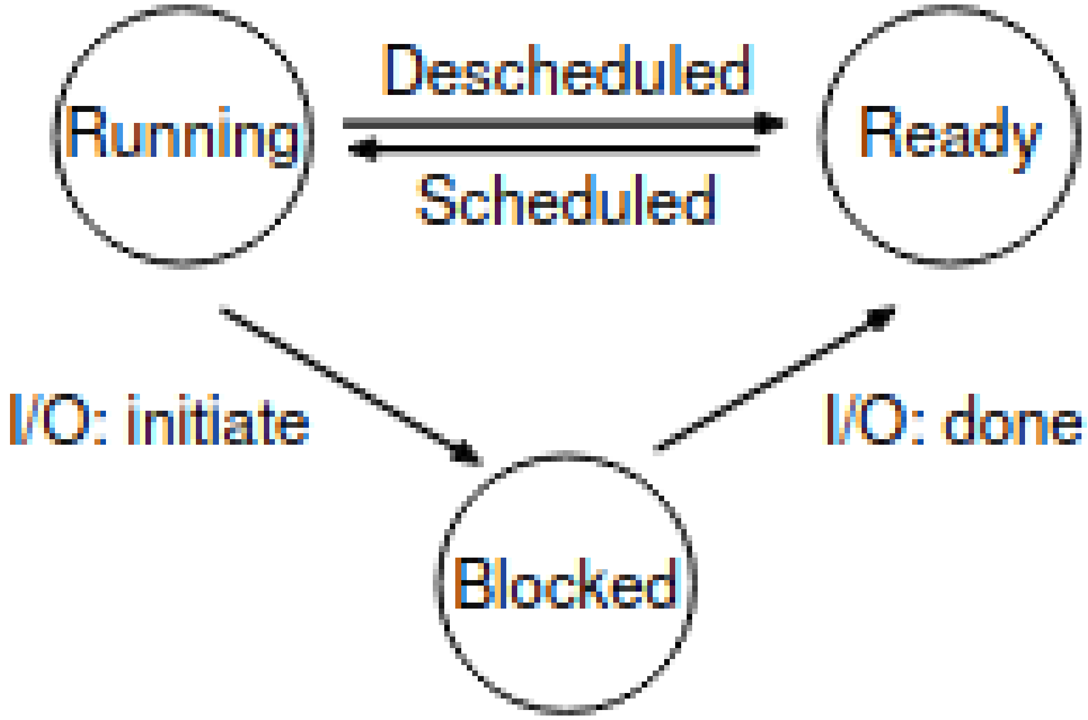

Figure 4.2: Process: State Transitions

Process State Examples

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	$Process_0$ now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	$Process_1$ now done

Figure 4.3: **Tracing Process State: CPU Only**

Process State Examples

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**

Process State Examples

- Was it a good idea to not resume P0 after its I/O completed?
- Was it a good idea to run P1 while P0 waited?
- This is where the scheduler comes into play.

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**

Process Structure Example

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

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

// 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 !zero, sleeping on chan
  int killed;                // If !zero, has 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
};
```

Figure 4.5: **The xv6 Proc Structure**