Chapter 4: The Abstraction: The Process

This chapter introduces the **process**, one of the most fundamental abstractions provided by an operating system. A process is, informally, a program in execution. The OS takes a static program on disk and transforms it into a dynamic, running entity, creating the illusion that many programs can run simultaneously.

1. The Core Challenge: CPU Virtualization

While a system may have only one or a few physical CPUs, users want to run many programs concurrently. The OS must solve this core 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?

The OS achieves this through **CPU virtualization**. By rapidly switching between processes—running one for a short time, then another—the OS creates the illusion of a vast number of virtual CPUs. This technique is known as **time sharing**.

- **Mechanisms:** These are the low-level methods that implement required functionality. An example is a **context switch**, the mechanism that allows the OS to stop one process and start another.
- **Policies:** These are the algorithms that make decisions. An example is a **scheduling policy**, which decides *which* process to run next.

> **Design Tip: Separate Policy and Mechanism**
>
> A common design principle is to separate high-level policies (the "which") from low-level mechanisms (the "how"). This modularity allows policies to be changed without redesigning the underlying machinery.

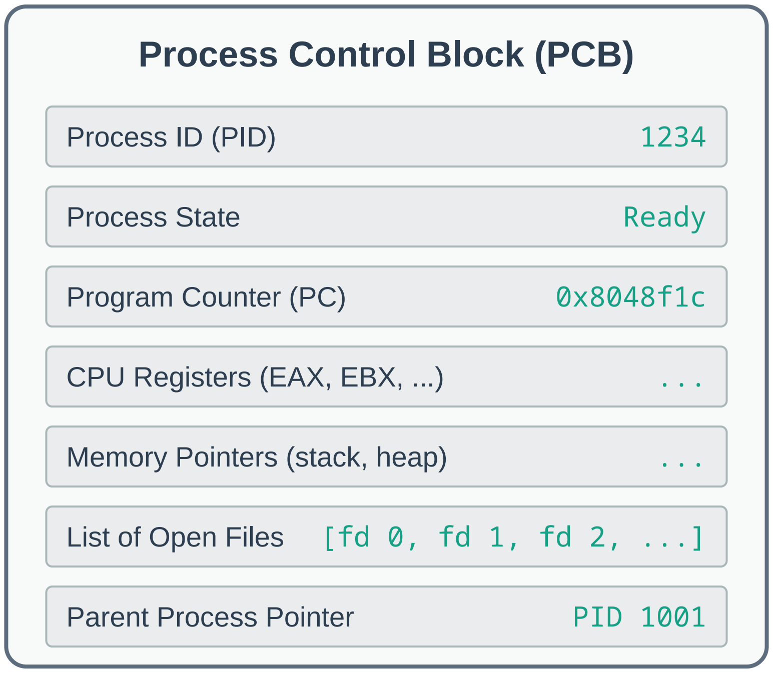
---

2. What is a Process?

A process is defined by its **machine state**—the parts of the machine it can read or update during execution. This state includes:

- **Memory (Address Space):** The memory that the process can address. This contains the program's instructions and the data it operates on.
- **CPU Registers:** These are crucial for program execution. Key registers include:
  - **Program Counter (PC) / Instruction Pointer (IP):** Points to the next instruction to be executed.

    - **Stack Pointer & Frame Pointer:** Manage the stack for local variables, function parameters, and return addresses.
- **I/O State:** Information about persistent storage, such as a list of currently open files.



Process Control Block (PCB)

Process ID (PID)	`1234`
Process State	`Ready`
Program Counter (PC)	`0x8048f1c`
CPU Registers (EAX, EBX, ...)	`...`
Memory Pointers (stack, heap)	`...`
List of Open Files	`[fd 0, fd 1, fd 2, ...]`
Parent Process Pointer	`PID 1001`

3. The Process API

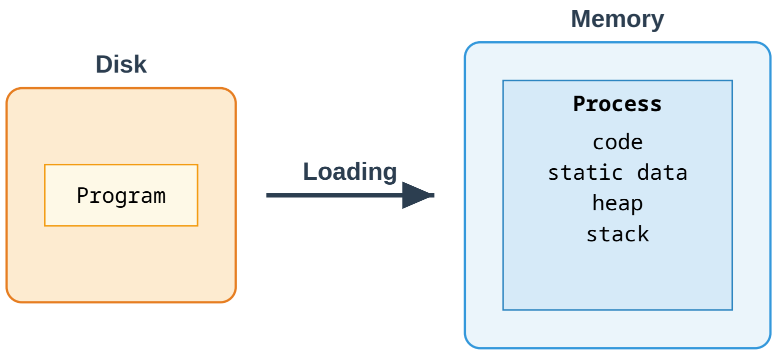
Every modern OS provides an API for managing processes. The core functionalities include:

- **Create:** A method to create a new process from a program.
- **Destroy:** An interface to forcefully terminate a process that is misbehaving or no longer needed.
- **Wait:** A mechanism to pause until a process has stopped running.
- **Miscellaneous Control:** Functions to suspend (pause) and resume a process.
- **Status:** Interfaces to get information about a process, such as its running time or current state.

4. Process Creation: From Program to Process

The OS follows a sequence of steps to transform a static program on disk into a running process in memory.

1. **Load Code and Static Data:** The OS reads the program's executable file from disk and loads its code and any initialized static data into the process's memory address space. This can be done **eagerly** (all at once) or **lazily** (on demand).
2. **Allocate Stack Memory:** The OS allocates memory for the process's run-time stack, which is used for local variables, function parameters, and return addresses. It initializes this stack with any command-line arguments (`argc`, `argv`).
3. **Allocate Heap Memory:** The OS allocates memory for the heap, which is used for dynamically requested data (via `malloc()`). This area can grow as the program runs.
4. **Initialize I/O:** The OS performs other setup tasks, such as creating default file descriptors for standard input, standard output, and standard error.
5. **Start Execution:** The OS transfers control of the CPU to the process by jumping to its entry point (the `main()` function), and the program begins to run.



5. Process States

A process transitions between several states during its lifetime. The three fundamental states are:

- **Running:** The process is currently executing instructions on a processor.
- **Ready:** The process is ready to run but is waiting for the OS to schedule it onto a processor.
- **Blocked:** The process has performed an operation (like initiating an I/O request) that makes it unable to run until an external event occurs (e.g., the I/O completes).

These states and their transitions can be visualized as follows:

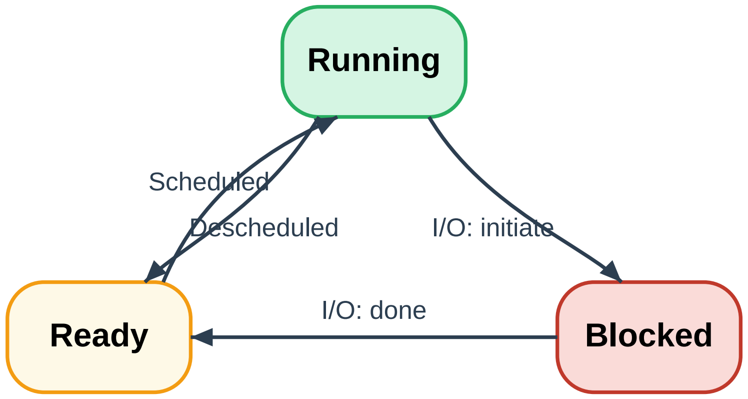


Figure 1: Process State Transitions

Process State Traces

Let's trace the states of two processes.

Scenario 1: CPU-Bound Processes Two processes that only use the CPU. The OS switches between them.

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

Scenario 2: Processes with I/O Process 0 performs an I/O operation, allowing Process 1 to run.

Time	Process 0	Process 1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process 0 initiates I/O

Time	Process 0	Process 1	Notes
4	Blocked	Running	Process 0 is blocked, so Process 1 runs
5	Blocked	Running	
6	Blocked	Running	
7	Ready	Running	I/O done, Process 0 is now ready
8	Ready	Running	Process 1 now done
9	Running	–	
10	Running	–	Process 0 now done

6. OS Data Structures for Processes

The OS needs data structures to keep track of all processes.

- **Process List:** A list containing information about all processes in the system.
- **Process Control Block (PCB):** A structure that stores all the necessary information for a single process. This is also known as a *process descriptor*.

The `xv6` kernel provides a good example of a PCB (`struct proc`):

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

// 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
};
```

- **Register Context**: Holds the saved register values when a process is stopped, allowing it to be resumed later.
- Additional States: Real OSes have more states.
  - `EMBRYO`: The state during process creation.
  - `ZOMBIE`: A state where the process has exited but has not yet been cleaned up by its parent process. This allows the parent to inspect its exit status.

7. Summary

The **process** is the OS's abstraction for a running program. The OS virtualizes the CPU to run many processes at once, manages their state transitions (Running, Ready, Blocked), and uses data structures like the Process Control Block (PCB) to track each one. This chapter lays the groundwork for understanding the mechanisms and policies the OS uses to manage processes effectively.