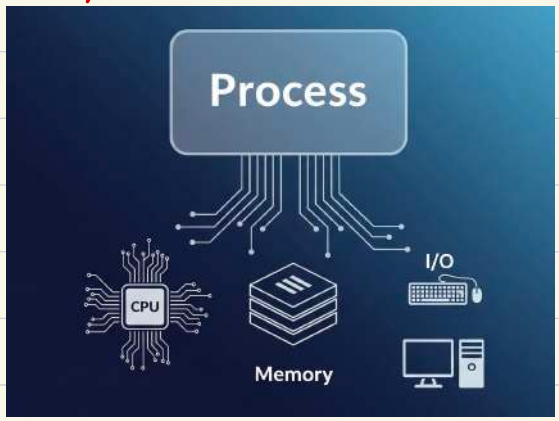
Process Abstraction

What is the difference b/w a program and process?

Program: static instructions on disk

process: running program



For an analogy, a program could be considered as a cake recipe, where as a process is like someone baking the cake, with a bowl, flour, oven, timer etc..

Goal of this chapter is to understand the notion of "process".

Question: what is the machine state of a process?
what parts of machine are important for a process?

* memory → instructions, data that programs read/write
  → address space
* registers eg:- program counter → tells what's the next instruction
  stack pointer → to manage the stack for function parameters, local variables etc..

* Storage devices for I/O.

What are the basic expectations on any interface of an operating system?

- create : type a command into the shell / double-click on an icon, the OS needs to be invoked to create a process.

- destroy : if the process doesn't exit by itself, there should be a provision to kill the process.

- wait : wait for another process to stop running.

- miscellaneous controls like suspend for sometime and then resume.

- status : information about the processes.

But, how does an OS actually creates a process?

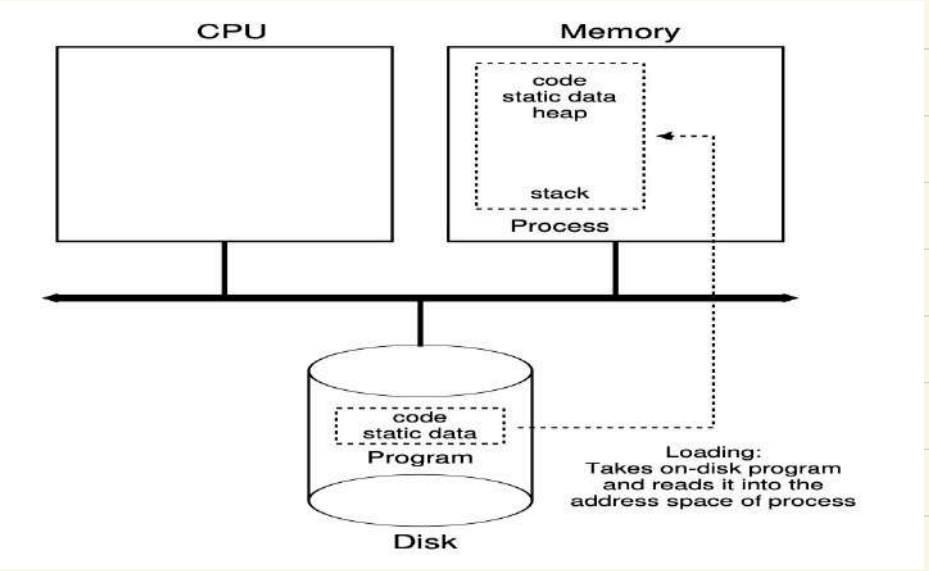
OS perform the following steps to run a program.

Step 1: Allocates memory and creates a memory image

* load its code and any static data from disk exe like, initialized variables
* Creates runtime stack, heap

stack → local variables, function parameters, return addresses, etc.

heap → for explicitly requested dynamically-allocated data, for data structures like linked lists, hash tables etc..



Step 2: Open basic files - STD IN, OUT, ERR

Step 3: Initialize CPU registers.
- PC points to first instruction.
program counter

States of a process

(currently executing on CPU) **Running**

Ready (waiting to be scheduled)

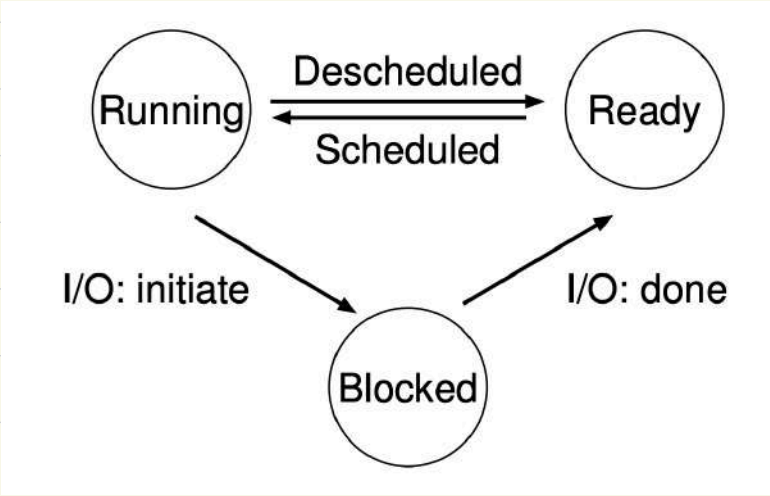
Blocked (suspended, not ready to run)

Why? waiting for some event, eg:- process issues a read from the disk.

When will it get unblocked? Disk issues an interrupt when the data is ready.

some extra states
- * New (being created, yet to run)
- * Dead (terminated)

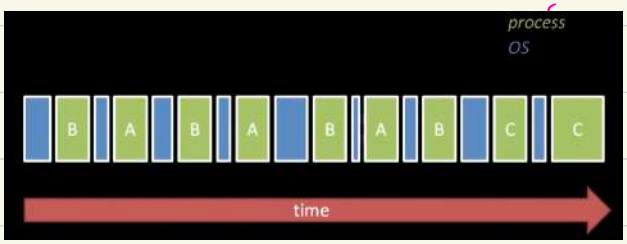
state transition Diagram →



So far, we were concered only about a single process.
But we wish to run many process in a single system.

How to provide the illusion of many CPUs such that tens or hundreds of processes can be run at the same time?

→ by virtualizing the CPU



using time sharing, allows users to run as many concurrent process as they like.

Example - 1 :
Two processes running, each of which only use the CPU (no I/O)

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

Example-2: Process₀ initiates I/O.

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

→ could be reading/writing on a disk or waiting for a packet from a network

Concerned questions:

① which process to run at a particular point of time (policy)

② How to switch b/w processes? (mechanism)

policy
(high-level intelligence)

↓

scheduling algorithms

eg:- which program to run?
how long to run this program?
etc..

mechanisms
(low-level machinery)

↓

protocols that needs to be implemented for switching

eg:- how to do a context switch?

scheduling → will learn later ← switch?

Question: what to keep in mind while switching?

Suspended game analogy:

Imagine you are playing several board games (say, chess, monopoly, scrabble etc.) with friends, but you only have one table. So you play one game at a time, and when you pause a game to play another, you carefully saves its entire state, so

you can return to it later without losing track of anything.

what all things you need to save?

Suspended board game components	Operating systems equivalent
* Game ID or name	* a unique identifier (PID)
	Memory image
* Game rules/settings player roles, cards in hands, dice, etc.	* code data (static) * stack and heap (dynamic)
	CPU context: registers
* whose turn it is?, position of all game pieces, etc..	* program counter → tells which instruction of the program will execute next * current operands * stack pointer → manages the stack of function parameters,
	file descriptors
* cards withdrawn	* pointers to opened files and local devices etc.. variables etc..

i.e., when you "suspend" a game, you need to carefully record the positions, scores, and whose turn it is, box up all the necessary accessories, just like OS saving the machine state of a process during a context switch.

and when you "resume" a game, you bring out the game, set the board up exactly it was, and continue playing seamlessly, just like OS restoring the machine state, loading all registers, stacks, program counter etc. so the process resumes execution right where it left off.

What are the data structures that OS needs to track process information?

- process list → maintains a list of processes
- process control block (PCB) → information that OS keeps to track each process.

(individual structure to each process)

In order to return back the game, you need a well-organized game box that stores all the context information resume the game later.

This box is the PCB → PCB fields in OS

* process ID (PID) - unique ID
* program counter - next instruction to execute
* Register values - snapshot of CPU registers
* stack, heap, data segment info - memory pointers
* process state - ready, waiting, running etc.

* accounting info - CPU time used, execution stats
* I/O and file descriptors - pointers to opened files, devices used
* parent/child relationship - process hierarchy info

The PCB is a self-contained storage box that ensures the process can be resumed exactly as it was left.

Now, imagine a storage shelf where you keep all your boxed games when you are not playing them. Each box is clearly labeled and carefully placed in a slot.
This shelf is the Process List.

Structure and Purpose

* It's a list/array/linked list maintained by the OS.
* each entry is a pointer to a PCB
* The OS consults this list to know what processes exist, their current state, and where their PCB is stored.

Reference: Chapter 4 of OSTEP.