Operating Systems

The Abstraction: The Process

These slides include content from the work of:
Youjip Won, KIAT OS Lab

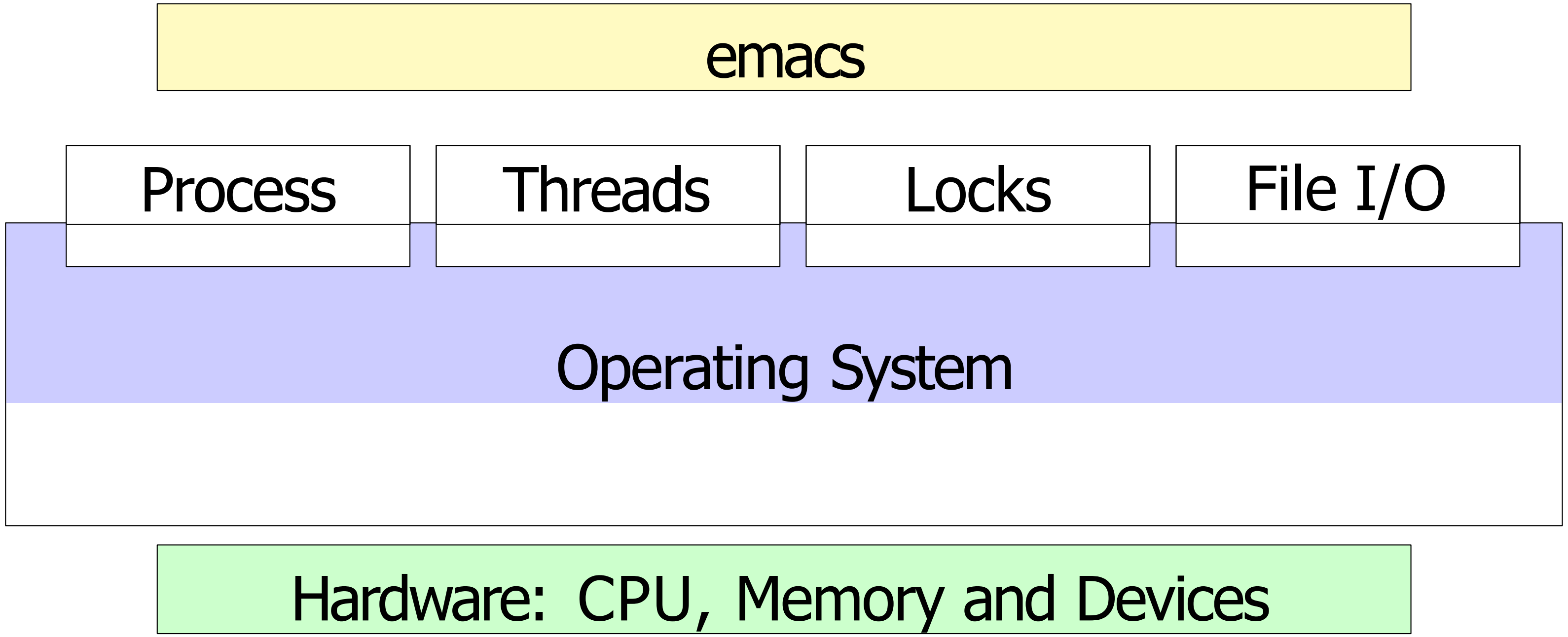
Operating System

emacs

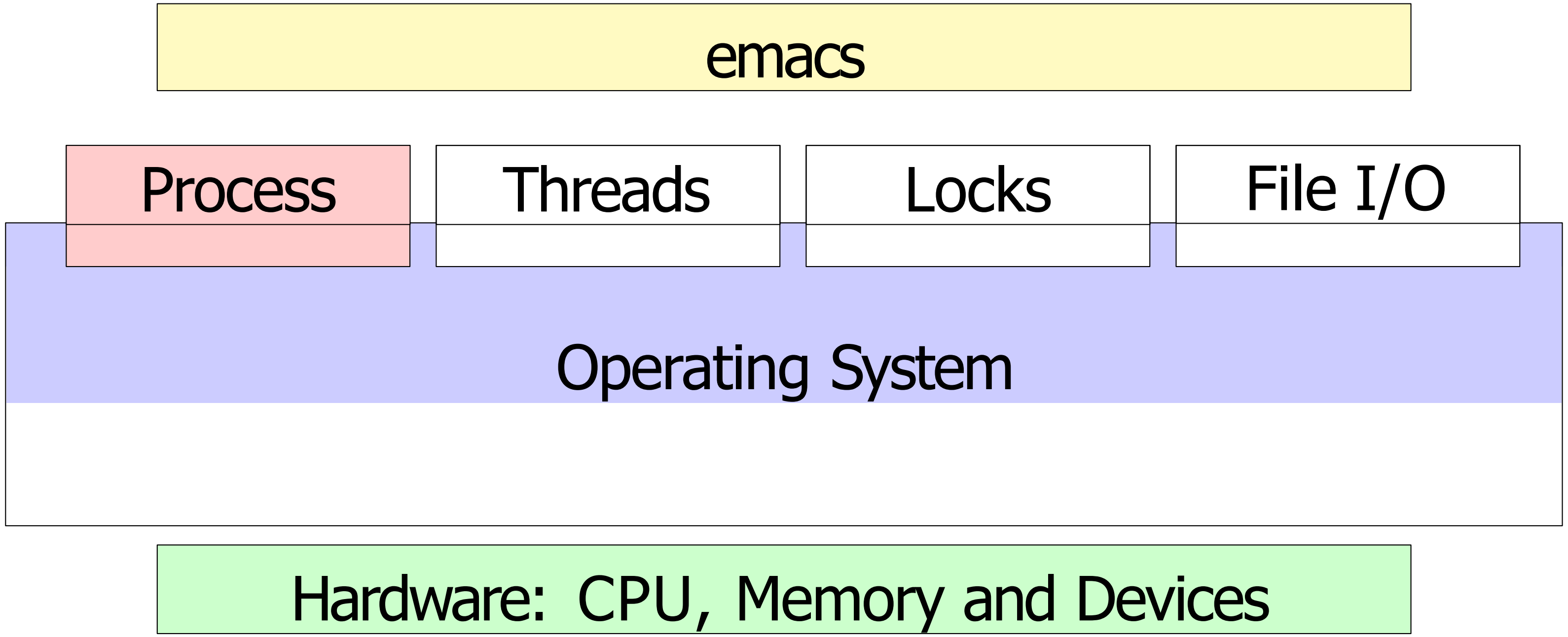
Operating System

Hardware: CPU, Memory and Devices

Operating System: Basic Abstractions and APIs



Today: Introduce the Process Abstraction



How to provide the illusion of many CPUs?

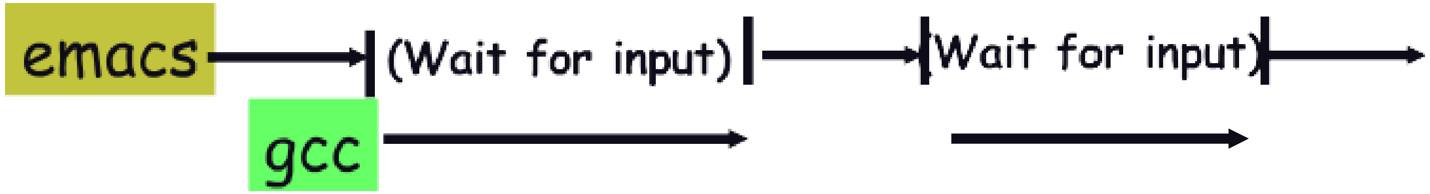
- CPU virtualizing
  - The OS can promote the illusion that many virtual CPUs exist.
  - **Time sharing**: Running one process, then stopping it and running another
    - The potential cost is performance: as each will run more slowly if the CPU(s) must be shared.

Processes

- A process is an instance of a program running
- Examples (can all run simultaneously):
  - `gcc file_A.c` – compiler running on file A
  - `gcc file_B.c` – compiler running on file B
  - `emacs` – text editor
  - `firefox` – web browser
- Non-examples (implemented as one process):
  - Multiple older versions of Firefox (still one process)
- Modern OSes run multiple processes simultaneously

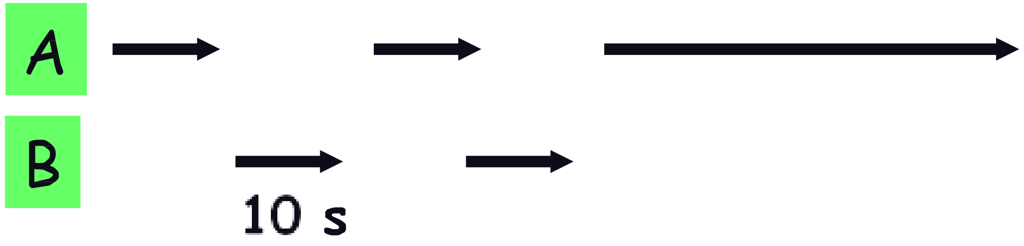
Speed

- Multiple processes can increase CPU utilization
  - Overlap one process's computation with another's wait



- Multiple processes can reduce latency
  - Running *A* then *B* requires 100 sec for *B* to complete

  - Running *A* and *B* concurrently makes *B* finish faster

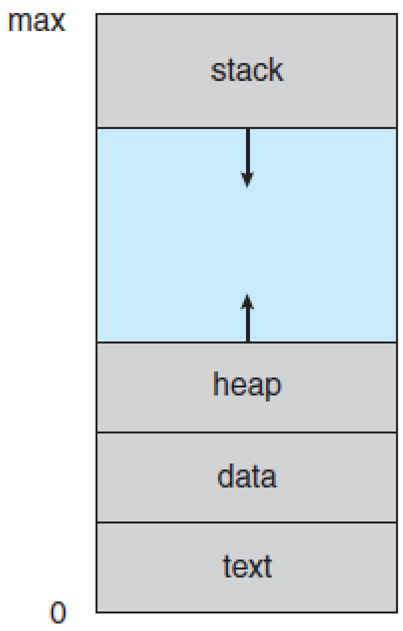


A Process

A process is a running program.

- **Process comprising of:**

1. Memory:
   - **Text Segment (Instructions):** This is the memory space where the executable code of the process is loaded.
   - **Data Segment**: This holds the global and static variables used by the process.
   - **Heap**: This is the dynamically allocated portion of memory that grows and shrinks as the process makes system calls like `malloc` and `free` in C, for instance.
   - **Stack**: This memory is used for function call management, local variables, return addresses

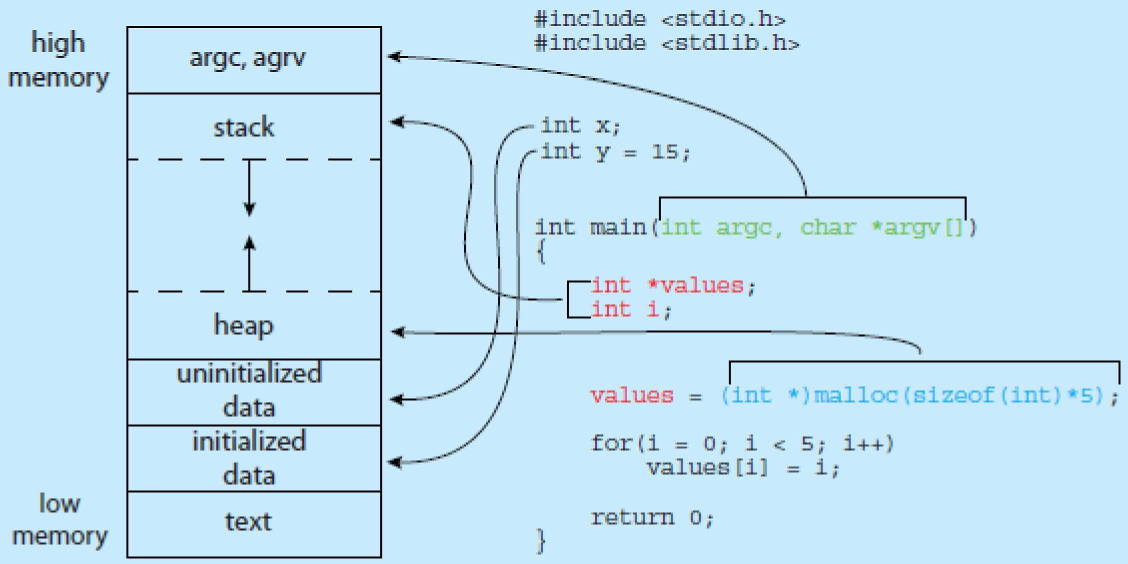


2. Process Control Block (PCB):

Memory Layout of a C Program

The figure shown below illustrates the layout of a C program in memory, highlighting how the different sections of a process relate to an actual C program. This figure is similar to the general concept of a process in memory as shown in Figure 3.1, with a few differences:

- The global data section is divided into different sections for (a) initialized data and (b) uninitialized data.
- A separate section is provided for the `argc` and `argv` parameters passed to the `main()` function.



Process Control Block (PCB)

- OS keeps data structure for each proc
  - Process Control Block (PCB)
  - Called `proc` in Unix, `task_struct` in Linux, and `struct Process` in COS
- Process ID (PID): A unique identifier for the process.
- Process State: The current state (e.g., ready, running, waiting, terminated).
- Program Counter: The address of the next instruction to execute.
- CPU Registers: The values of all CPU registers for this process.
- I/O Status Information: List of I/O devices allocated to the process, list of open files, etc.
- ….

Process state
Process ID
User id, etc.
Program counter
Registers
Address space (VM data structs)
Open files

PCB

Process Creation

1. **Load** a program code into memory, into the address space of the process.
   - Programs initially reside on disk in *executable format*.
   - In early (or simple) operating systems, the loading process is done eagerly.
     - all at once before running the program.
   - modern OSes perform the loading process lazily.
     - Loading pieces of code or data only as they are needed during program execution.

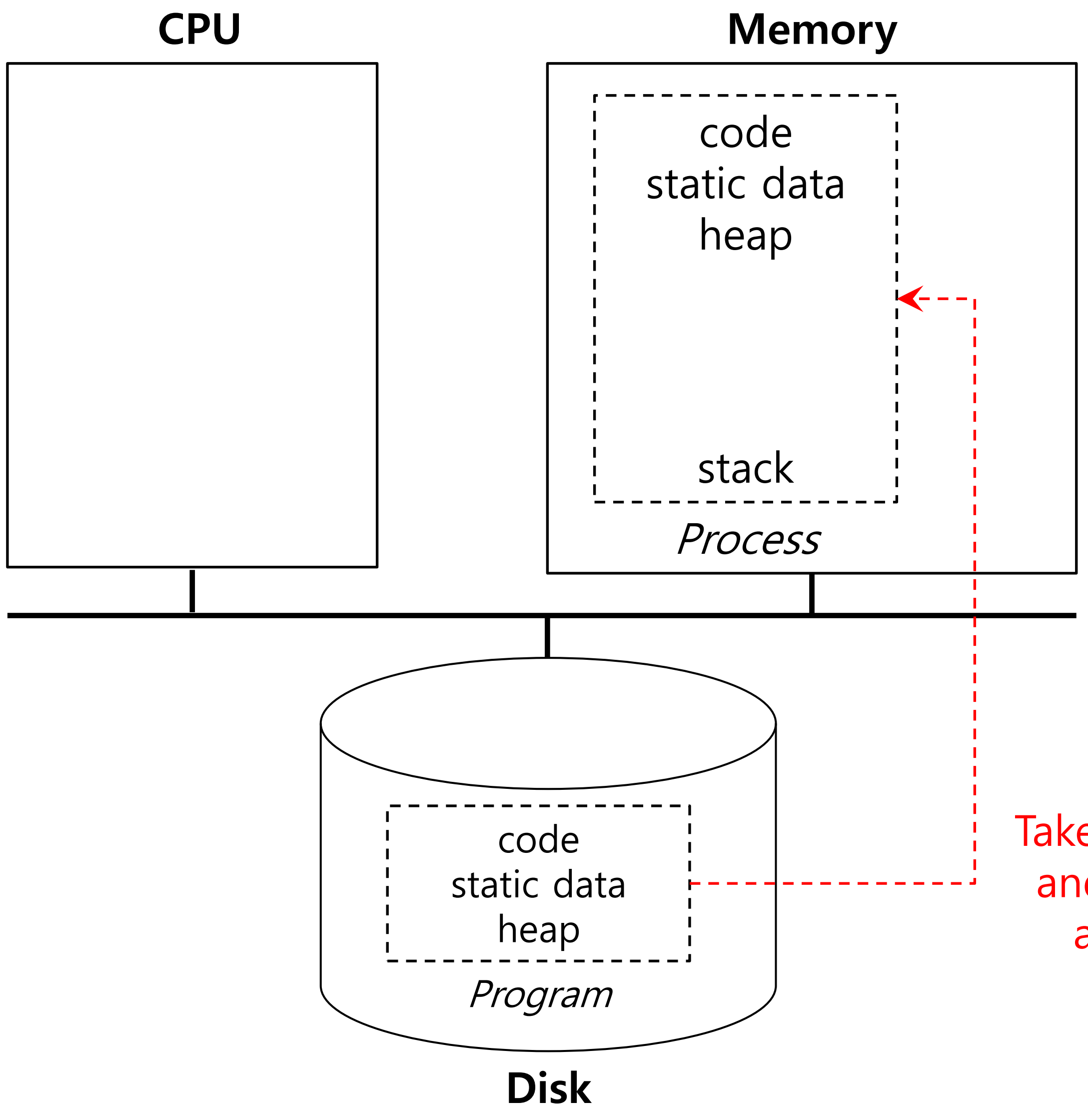
2. The program's run-time **stack** is allocated.
   - Use the stack for *local variables*, *function parameters*, and *return address*.
   - Initialize the stack with arguments → `argc` and the `argv` array of `main()` function

Process Creation (Cont.)

3. The program's **heap** is created.
   - Used for explicitly requested dynamically allocated data.
   - Program request such space by calling `malloc()` and free it by calling `free()`.

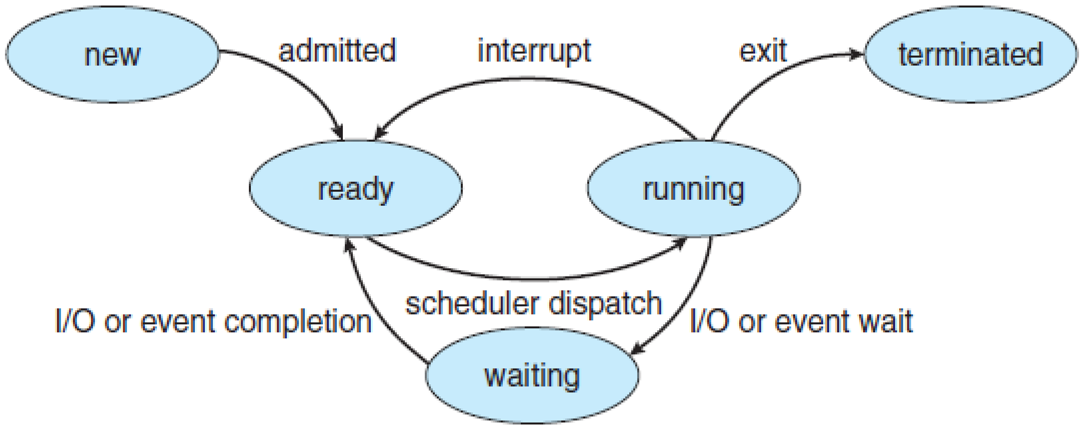
4. The OS do some other initialization tasks.
   - input/output (I/O) setup
5. **Start the program** running at the entry point, namely `main()`.
   - The OS *transfers control* of the CPU to the newly-created process.

Loading: From Program To Process



Loading:
Takes on-disk program and reads it into the address space of process

Process State Transition



- Process can be in one of several states
  - new & terminated at beginning & end of life
  - running – currently executing (or will execute on kernel return)
  - ready – can run, but kernel has chosen different process to run
  - waiting – needs async event (e.g., disk operation) to proceed

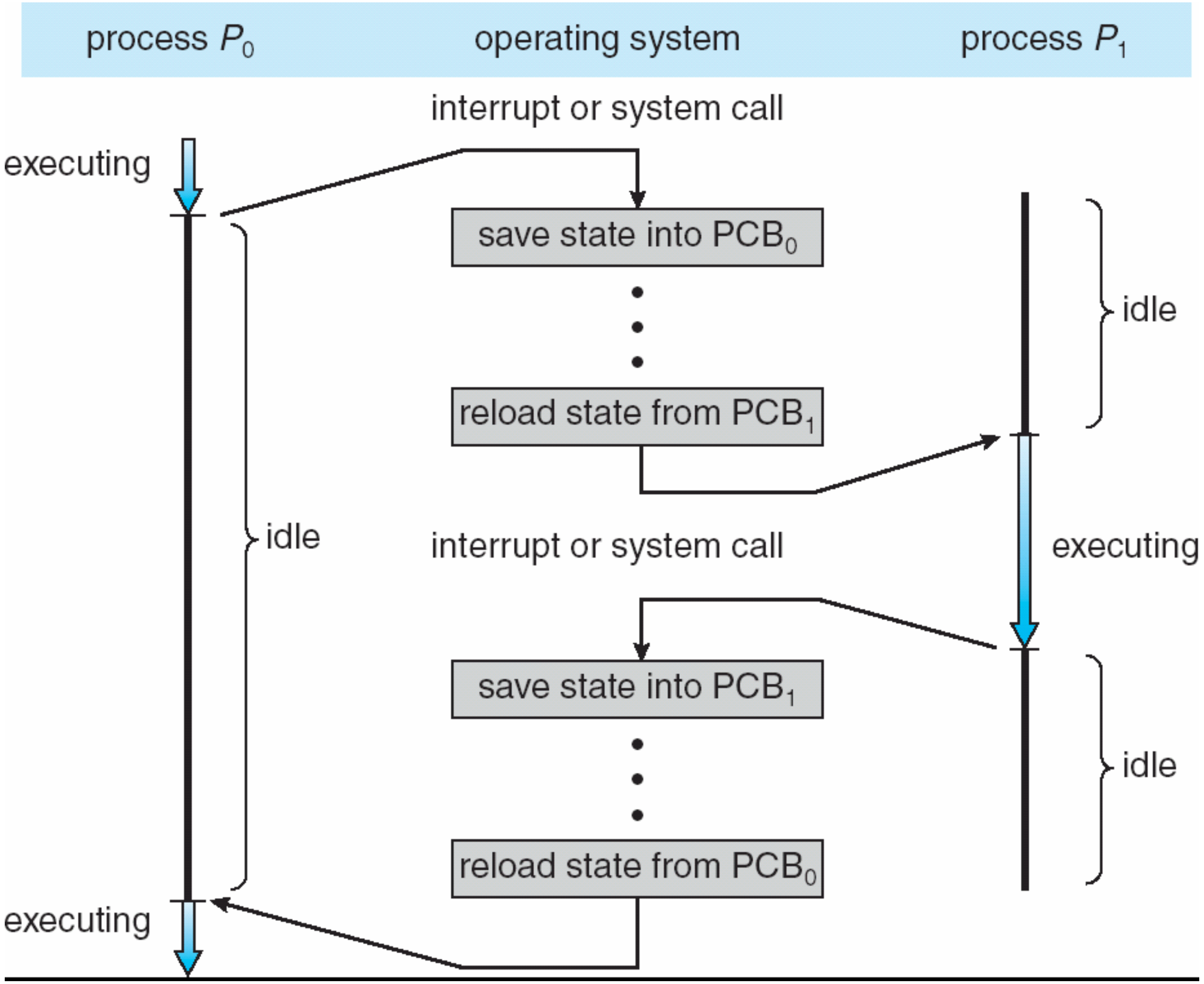
Scheduling

- An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations.
- A CPU-bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations.
- OS maintains scheduling queues of processes
  - Job queue – set of all processes in the system
  - Ready queue – set of processes ready and waiting to execute
  - Device queues – set of processes waiting for an I/O device
- Processes migrate among the various queues

Context switch

➢ When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a ***context switch***

➢ An ***interrupt*** is a signal to the CPU that tells it to stop its current execution and immediately handle an event. Examples: A key is pressed, system calls, CPU clock tick forces scheduler to switch processes



Process Life Cycle

1. Program Request
   - User clicks program / enters command.
   - OS gets request to start execution.
2. 2. Process Creation (New → Ready)
   - OS creates a PCB (Process Control Block) with PID & metadata.
   - Virtual memory allocated:
     - Code segment (program instructions)
     - Data segment (globals, statics)
     - Heap (dynamic allocation)
     - Stack (function calls, locals, return addresses)
   - Arguments/environment prepared on the stack.
   - Process placed in the Ready queue.

Process Life Cycle

3. Scheduling & Dispatch (Ready → Running)
   - CPU scheduler picks a process from Ready queue.
   - Dispatcher loads registers, program counter, etc. from PCB (**Context Switch** happens).
   - Process starts execution at its entry point (*main*).
4. Execution & System Calls (Running ↔ Waiting)
   - While running, the process may:
     - Execute user instructions in user mode.
     - Make system calls (switch to kernel mode).
     - Perform I/O → if waiting, moved to Blocked state.
   - When blocked process waits; CPU is given to another Ready process (**Context Switch** happens).
5. Process Completion (Running → Terminated)
   - When finished, process calls exit.
   - OS releases memory (heap, stack, code/data) and OCB removed.