Operating Systems

Isfahan University of Technology
Electrical and Computer Engineering Department
1400-1 semester

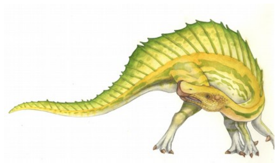
Zeinab Zali

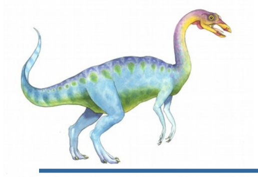
Session 5: Process Management

Reference: Operating System Concepts book slides

Process Concept

- An operating system executes a variety of programs that run as a process.
- **Process** – a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process
- Program is **passive** entity stored on disk (**executable file**); process is **active**
- Program becomes process when an executable file is loaded into memory
  - Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes
  - Consider multiple users executing the same program

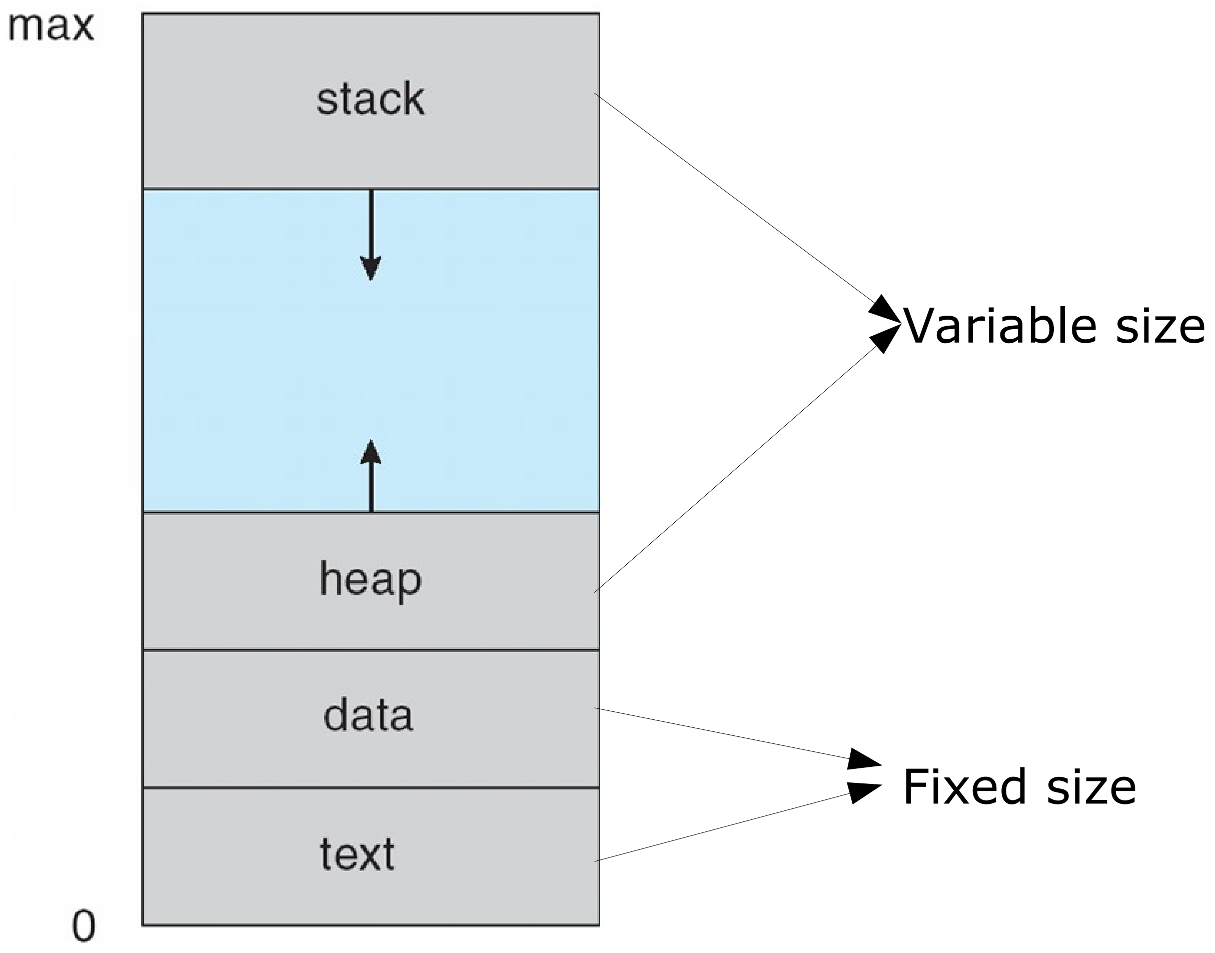




Process Parts

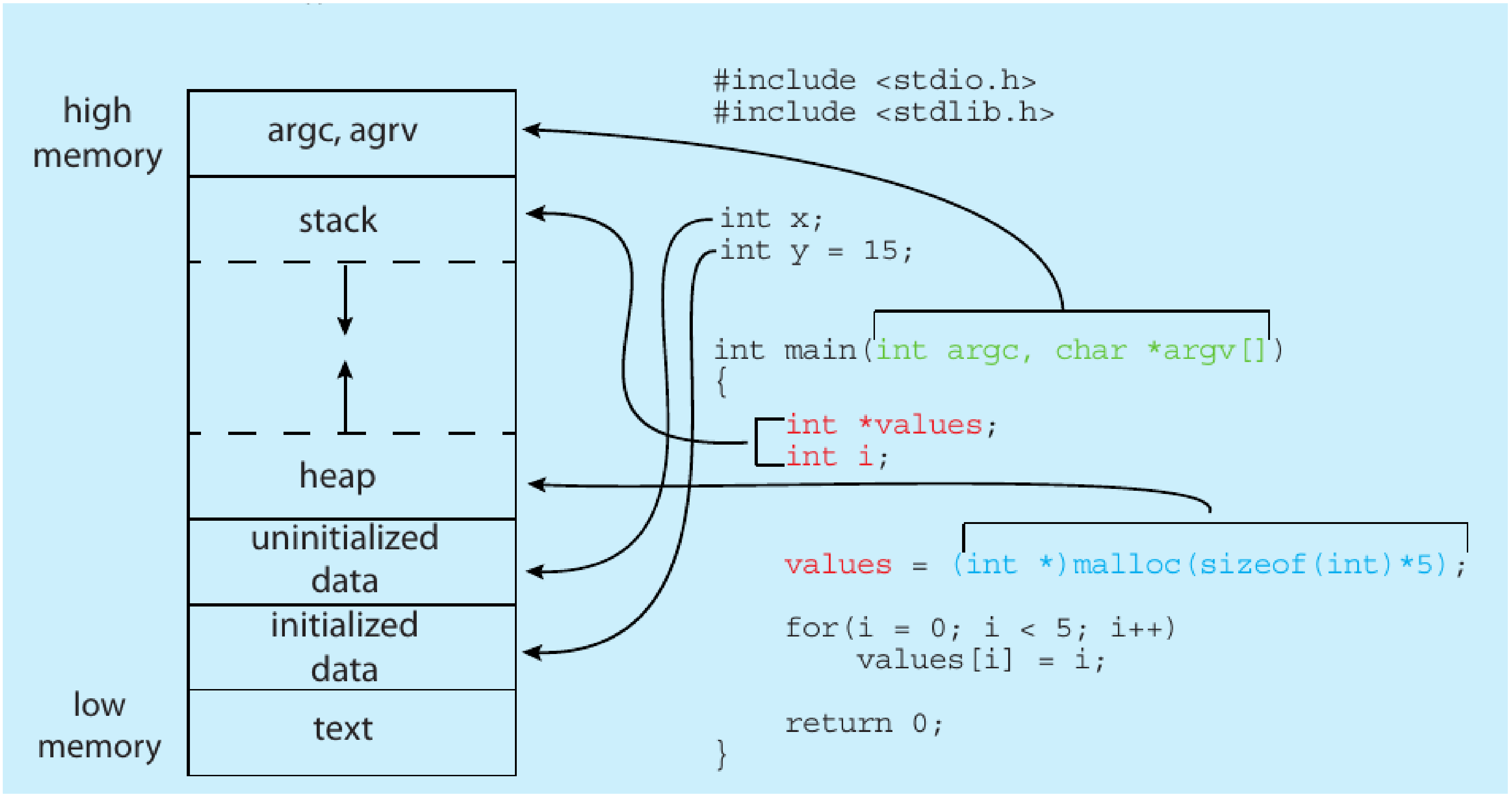
- The program code, also called **text section**
- Current activity including **program counter**, processor registers
- **Stack** containing temporary data
- Function parameters, return addresses, local variables
- **Data section** containing global variables
- **Heap** containing memory dynamically allocated during run time

Process in Memory



stack and heap sections grow toward one another,
the operating system must ensure they do not overlap one another

Memory Layout of a C Program



```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
   int *values;
   int i;

   values = (int *)malloc(sizeof(int)*5);

   for(i = 0; i < 5; i++)
       values[i] = i;

   return 0;
}
```

Process State

- As a process executes, it changes **state**
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process has finished execution

With single CPU, only One program is running
while many programs may be ready or waiting

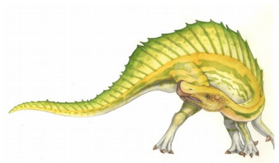
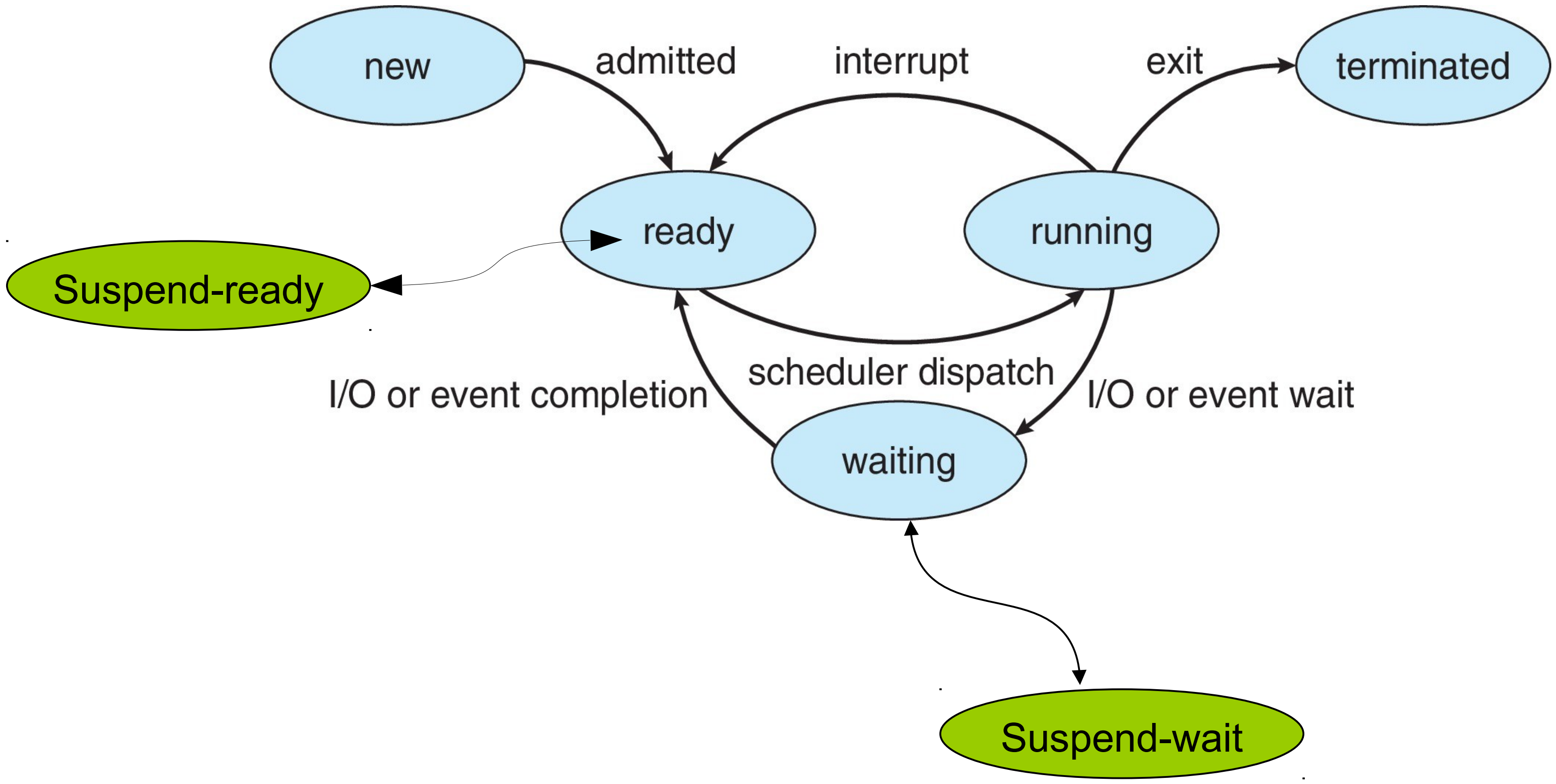


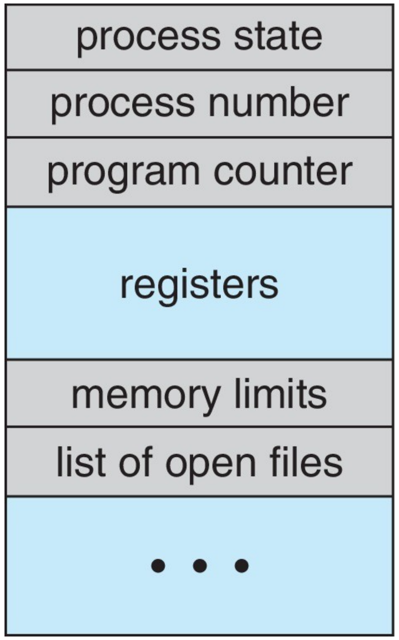
Diagram of Process State



Process Control Block (PCB)

Information associated with each process(also called **task control block**)

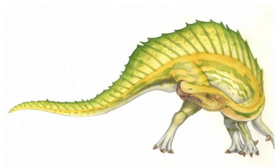
- Process state – running, waiting, etc.
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



All processor designs include a register or set of registers, often known as the **program status word** (PSW), that contains status information.

Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- Explore in detail in Chapter 4

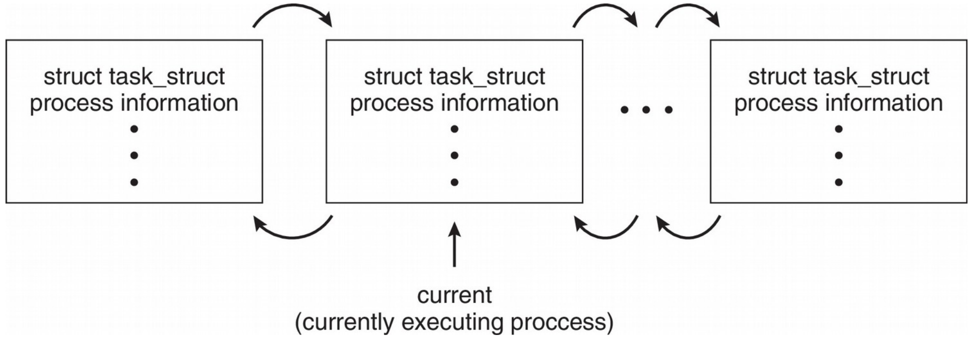


Process Representation in Linux

Represented by the C structure `task_struct`

<include/linux/sched.h>

```
pid t_pid;           /* process identifier */
long state;                /* state of the process */
unsigned int time_slice    /* scheduling information */
struct task_struct *parent;/* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files;/* list of open files */
struct mm_struct *mm;  /* address space of this process */
```



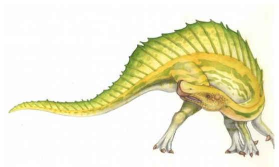
Look inside sched.h and find the corresponding data structures to this chapter

<linux/sched.h>

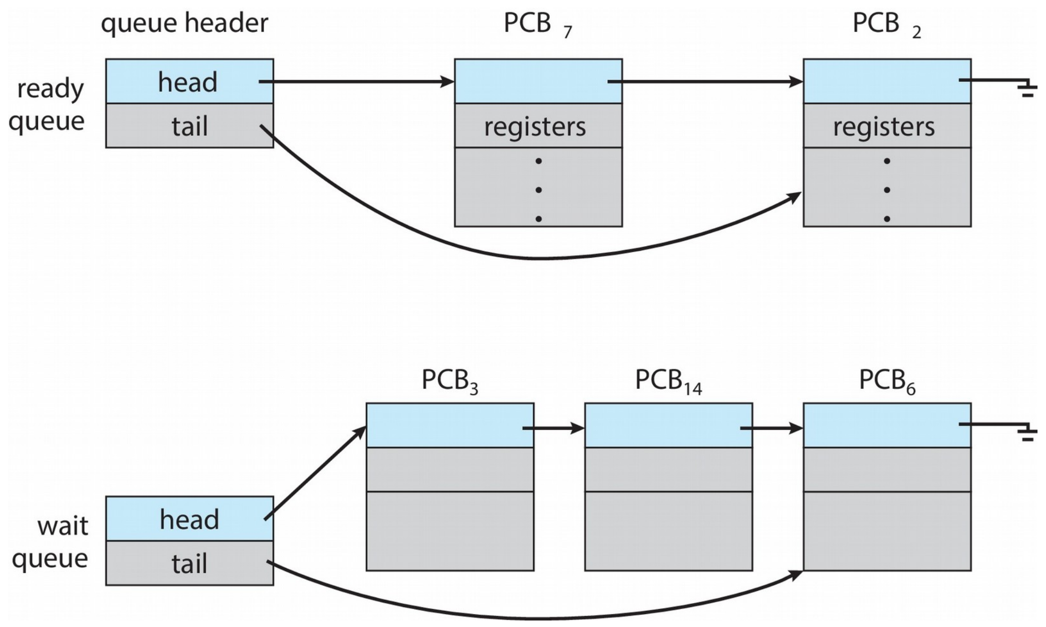
rb_node
Rq (running queue)
State (runnable, unrunnable, stopped)

Process Scheduling

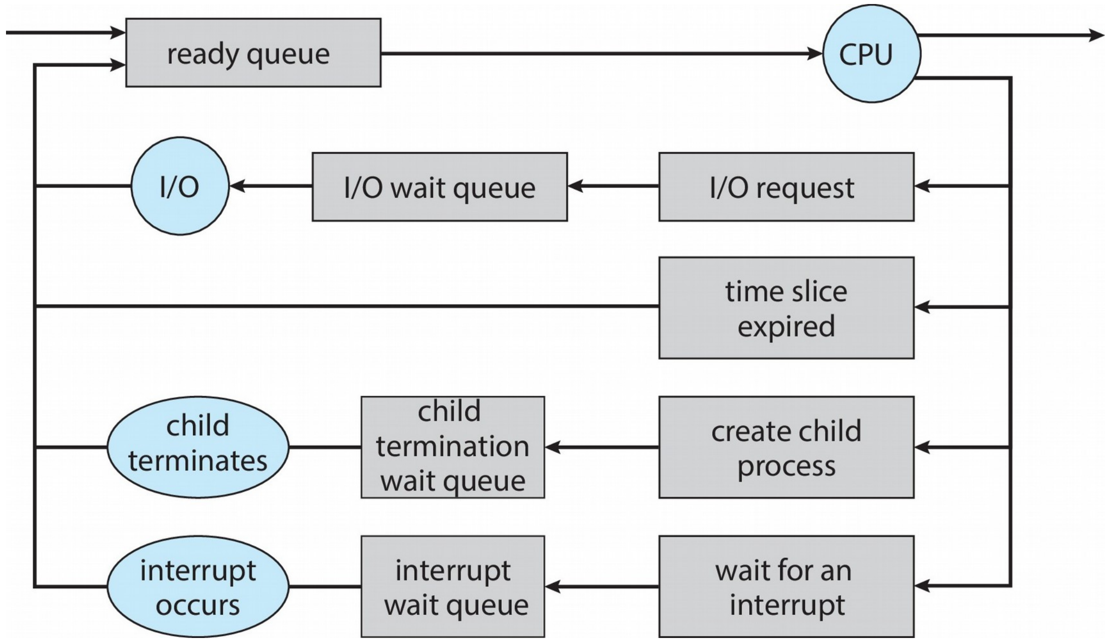
- **Process scheduler** selects among available processes for next execution on CPU core
- Goal -- Maximize CPU use, quickly switch processes onto CPU core
- Maintains **scheduling queues** of processes
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Wait queues** – set of processes waiting for an event (i.e., I/O)
  - Processes migrate among the various queues



Ready and Wait Queues

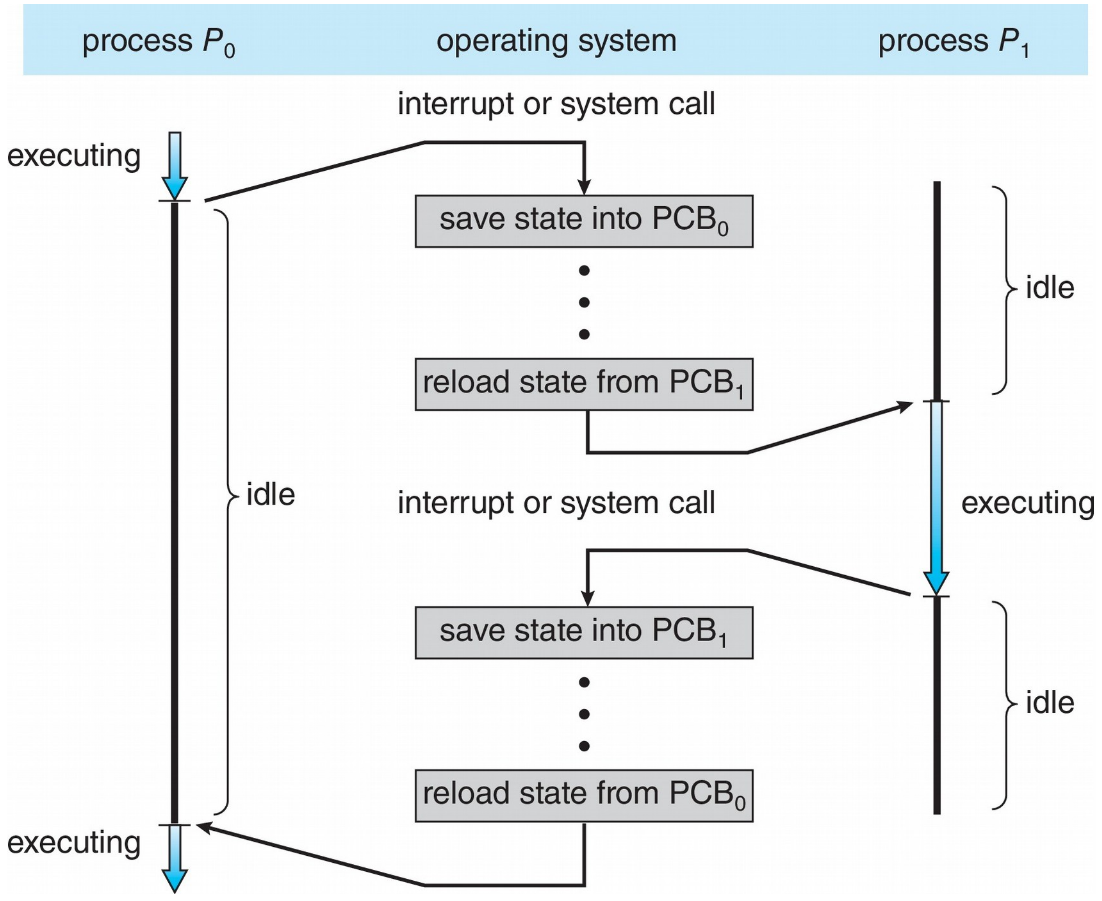


Representation of Process Scheduling



CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another.



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**
- **Context** of a process represented in the PCB
- Context-switch time is pure overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ⇒ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ⇒ multiple contexts loaded at once

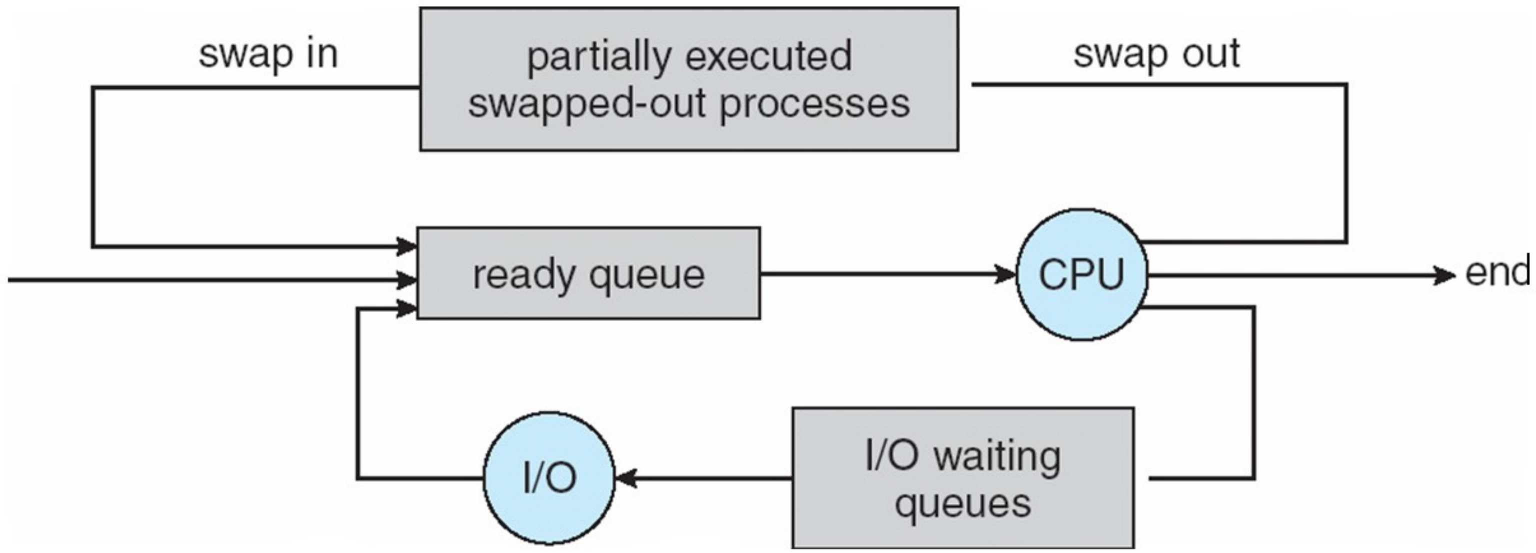
What is the difference between Interrupt handling and context switch?

Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Load balancing of I/O and CPU bound processes
  - Long-term scheduler strives for good ***process mix***

Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**


swap in
partially executed swapped-out processes
swap out
ready queue
CPU
end
I/O
I/O waiting queues