# Process Abstraction

2023-24 COMP3230B

# Contents

- What is a “process”?
- Represent a “process”
    - Resources use by a process
    - Process states
- Important OS data structures
- Operations on processes

# Learning Outcome

- ILO 2a - explain how OS manages processes/threads and discuss the mechanisms and policies in efficiently sharing of CPU resources.

# Reading & Reference

- Required Reading
  - Operating Systems: Three Easy Pieces by Arpaci-Dusseau et. al
    - Chapter 4, Abstraction: The Process
      - http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-intro.pdf
    - Chapter 5, Interlude: Process API
      - http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf

# Process vs. Program

- Program itself is a lifeless thing
- What is a Process?
  - A process is a program in execution
    - **An instance** of a program running in a computer
    - A process is an entity that can be assigned to and executed on a CPU
  - A unit of activity characterized by
    - the execution of a sequence of instructions,
    - with an execution state, and
    - an associated set of system resources

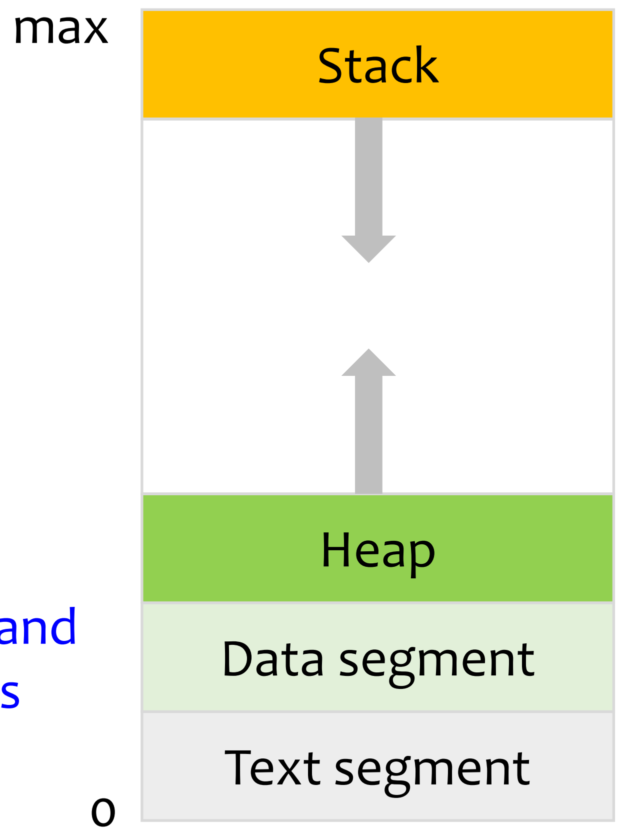
# Process – an abstraction

- To represent a running program, the OS needs to keep the following information:

| | | Examples |
| --- | --- | --- |
| The memory that the process can access (or reference) is part of the process | Memory | Program code<br>Data |
| | Resources in use | I/O in use<br>Physical memory |
| During execution, process updates/ stores data in CPU registers, e.g., program counter, stack pointer, etc. | Execution state | Register set<br>Current process state |

# Process – Address Space

- The process’s view of its memory is called the **address space**, which is a range of memory locations (or a range of memory addresses)
- Process address space consisting of a few “regions”:




Stores local variables, arguments to function, return value for active procedure calls

Where dynamically allocated memory is placed

Stores global and static variables

Stores the program code that the processor executes

# Process – Process States

- In the process execution life cycle, it moves through a series of discrete process states.

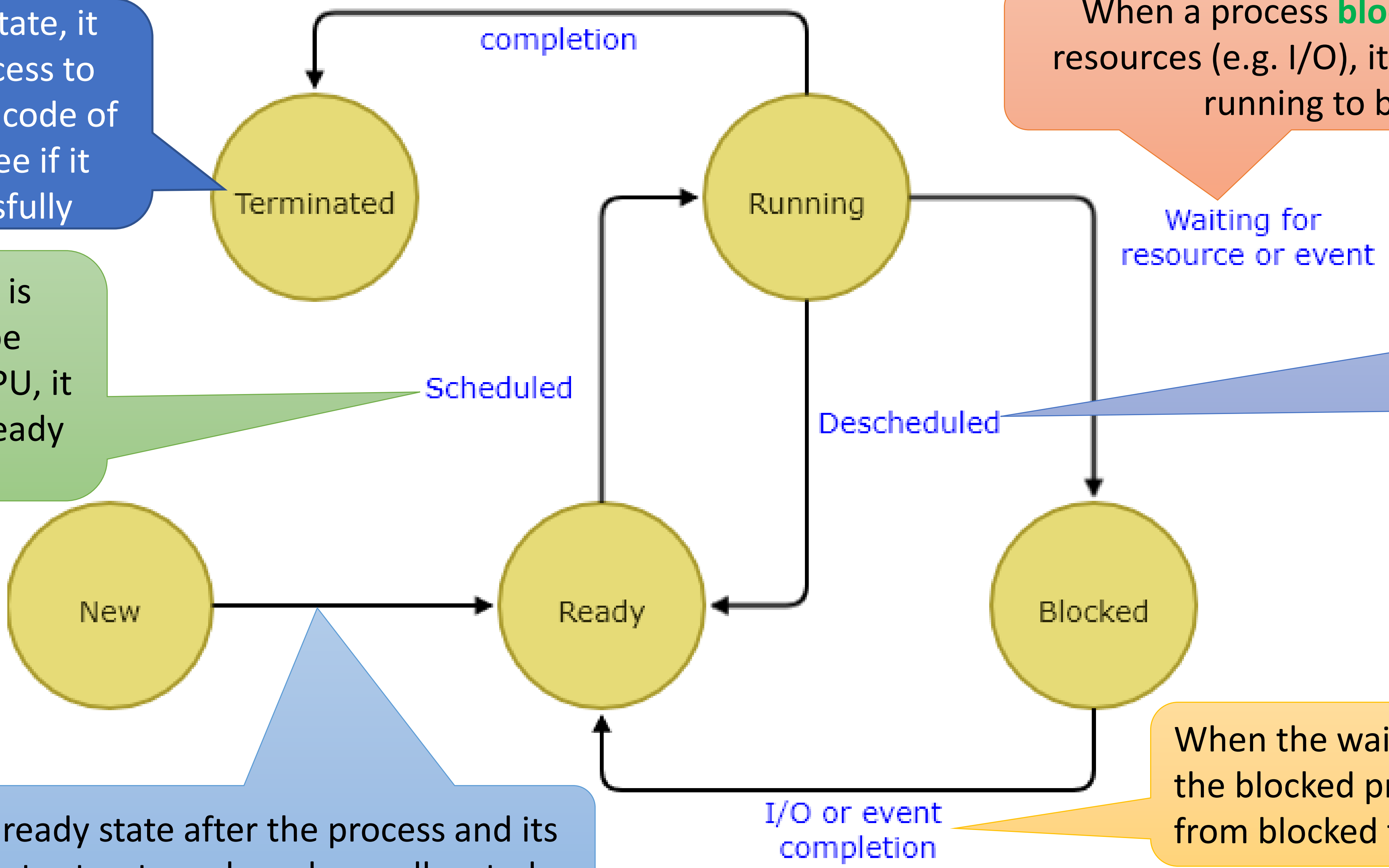
- Process state - an indicator of the nature of the current activity of a process
  - **new (initial)**: The process is just being created
  - **running**: The process is executing on a processor
  - **blocked**: The process is **waiting for** some event (e.g., I/O or communication) to happen before it can proceed
  - **ready**: The process **is ready to run** on a processor and **is waiting** to be assigned to a processor
  - **terminated (final / zombie)**: The process has finished execution but has not yet been cleaned up; *why not just discard it?*

# Life Cycle of a Process

With terminated state, it allows parent process to examine the return code of the process and see if it executed successfully

When a process is dispatched to be executed by the CPU, it transitions from ready to running

When a process **blocks waiting** for resources (e.g. I/O), it transitions from running to blocked




OS decides to switch a process from running to ready

When the waiting event occurred, the blocked process transitions from blocked to ready

Switch to the ready state after the process and its associated data structures have been allocated

# Process Control Block

- To manage a process, OS makes use of a **data structure** to maintain information about a process
  - Process Control Block (**PCB**) or Process Descriptor

- PCB typically includes
  - Process identification number (PID) - **a unique ID**
  - The current **process state**
  - The program counter - indicates the address of **next** instruction
  - **Register context** - a snapshot of the register contents in which the process was last running before it **transitioned** out of the running state
  - Scheduling information - process priority, pointers to scheduling queues, etc.

# Process Control Block (2)

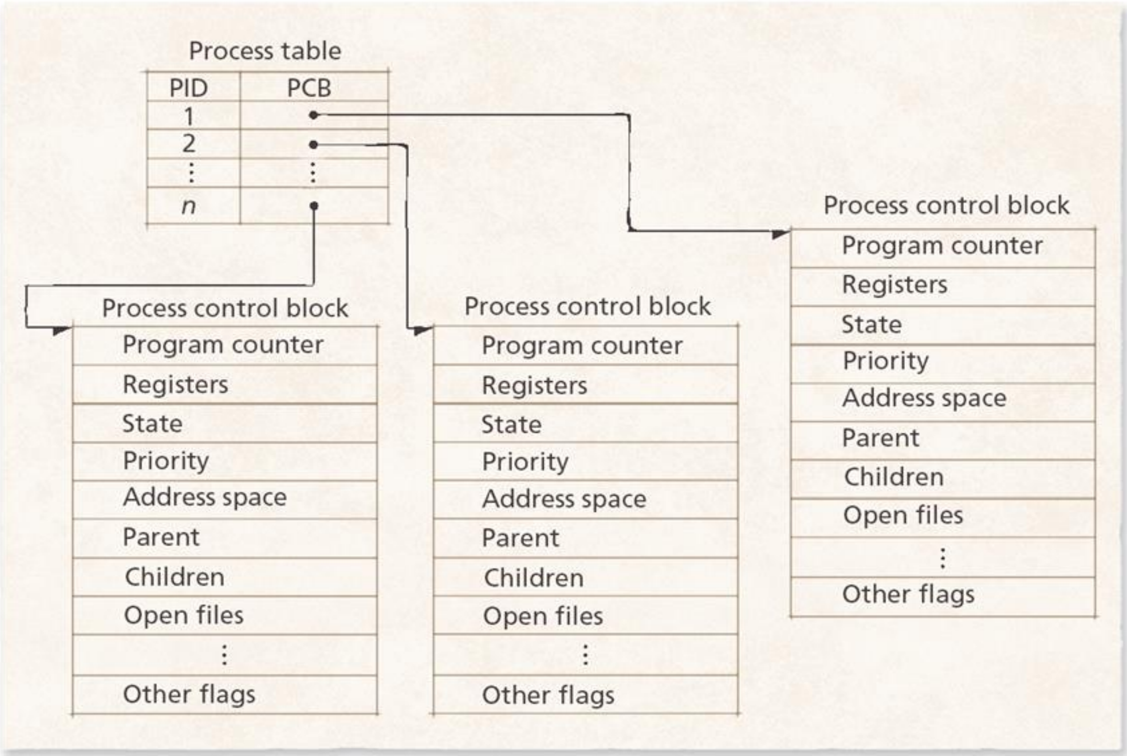
- Credentials - determines the resources this process can access
- Memory Management information - concerning memory areas allocated to the process
- Accounting information - CPU usage statistics, time limits, etc.
- A pointer to the process's parent process
- Pointers to the process's child processes
- Pointers to allocated resources

:

Example: Linux process descriptor
***struct** task_struct*
in /usr/src/linux/include/linux/sched.h
at around 800 lines of statements

# Process Table

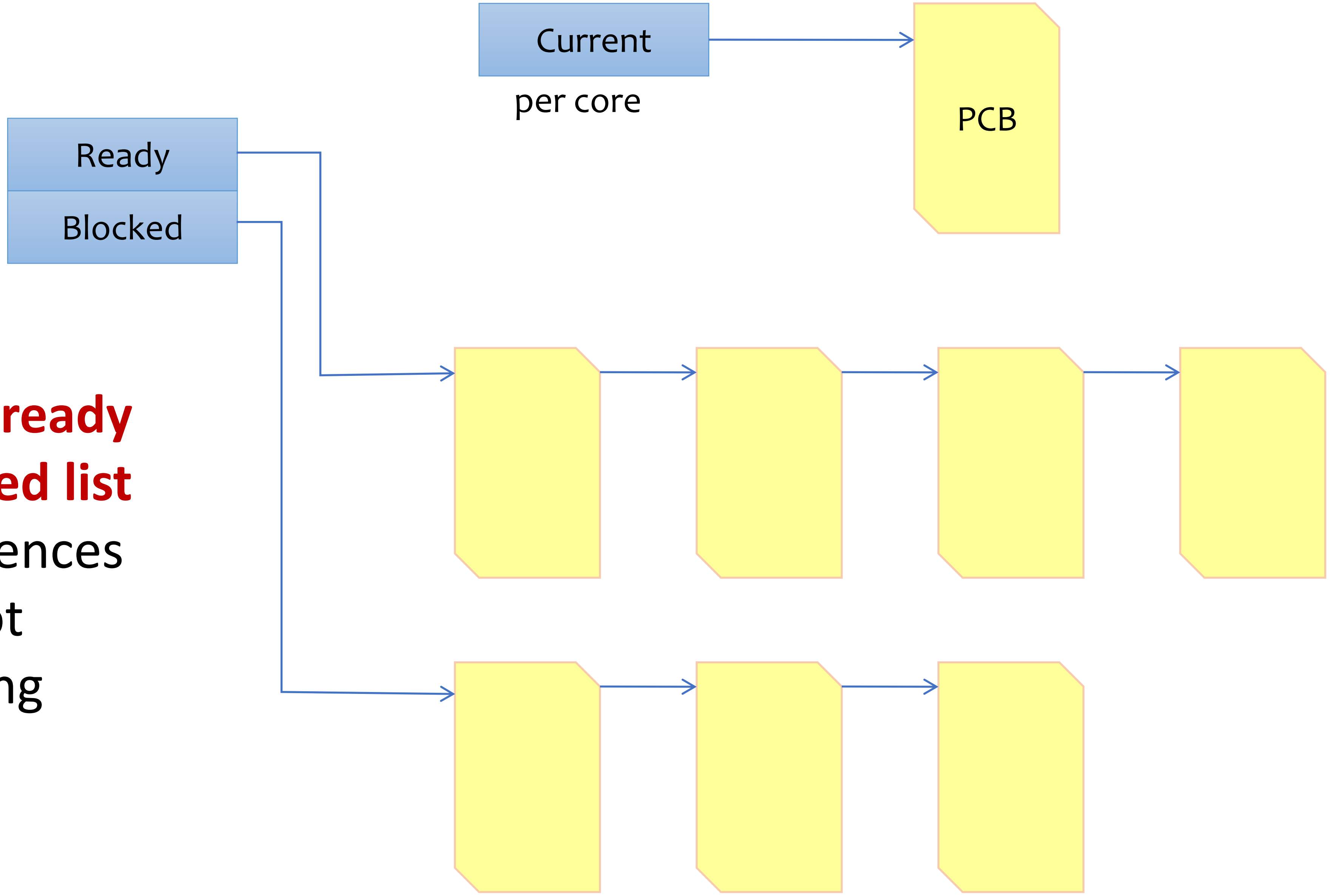
- To manage many processes, OS needs a way to **quickly access** a process's PCB




- OS keeps pointers to each process's PCB in a table
  - Example - Linux "process table" is organized in a form of hashed table

- When a process is "completely" terminated, OS removes the process from the process table and frees all of the process's resources

# Process List Structures




OS maintains a **ready list** and a **blocked list** that store references to processes not currently running

# ps - Show Processes information

- You can list the processes' details in Linux/Mac OS X by using the ***ps*** command

- Usage: ps [option]
  - When executed without any options, only processes that are associated with the current terminal are shown.

- Use **man-page** to learn how to use

- Some Useful options (Linux)
  - -e: Select all processes
  - -f: Show in full format
  - w: Wide output
  - f: ASCII-art process hierarchy (forest)

# pstree - Show the relations between processes

- A Linux system program similar to ps
  - Processes are organized in hierarchy
    - As every process has a parent process, their relationship can be viewed by using pstree

- Usage: pstree [option]
  - Use man-page to learn how to use
  - Example:

```
atctam@workbench2:~$ ps
    PID TTY          TIME CMD
3120691 pts/1    00:00:00 bash
3123097 pts/1    00:00:00 ps
atctam@workbench2:~$ pstree -s 3120691
systemd───sshd───sshd───sshd───bash───pstree
atctam@workbench2:~$ pstree -sp 3120691
systemd(1)───sshd(205)───sshd(3120553)───sshd(3120626)───bash(3120691)───pstree(3123102)
atctam@workbench2:~$
```

# Operations on Processes

- Operating systems provide fundamental services to processes including:
  - Creating processes
  - Destroying processes
  - Suspending processes
  - Resuming processes
  - Changing process’s priority (for scheduling)
  - Waiting for a process (parent process waits for the child process)
  - Check process’s status
  - Interprocess communication (IPC)

  :

# How to create a new process?

- A process spawns a new process
  - The process that creates a new process is now called the **parent** process
  - The newly created process is called the **child** process
    - The child can also create other processes, thus forming a tree of processes

  - When the parent process is destroyed/terminated, modern operating system typically
    - Keeps the child processes and allows them to proceed independently of the terminated parent process

# Creating process

- Allocate memory for the process
  - Space for PCB must be allocated

- Initialize the process control block
  - Assign a unique process ID
  - Save the process ID, parent ID
  - Set program counter and stack pointer to appropriate values

- Create other data structures
  - Memory, files, accounting

- Set the process state to Ready and put it to the Ready queue

# How to Create a Process

- Unix
  - fork() system function
    - A system call that **creates** a new process by **duplicating** the calling process
  - exec () family of functions
    - OS replaces the current program image with a new program image

- Windows API
  - CreateProcess() function
    - Creates a new process and its primary thread and loads program for execution
    - http://msdn.microsoft.com/en-us/library/ms682512(VS.85).aspx
  - CreateProcess() is similar to fork() + exec ()

# Process Termination

- Process executes last statement and asks the operating system to delete itself
  - exit() or return from main()

- A process may terminate **involuntarily**
  - Parent may terminate execution of children processes (by sending a termination **signal**)
  - A number of **error and fault** conditions can lead to termination of process

- Return termination status from child to parent
  - Parent can obtain this info by calling wait(), wait4(), waitid() or waitpid()
  - The terminated process' resources are **de-allocated** by OS **afterward**

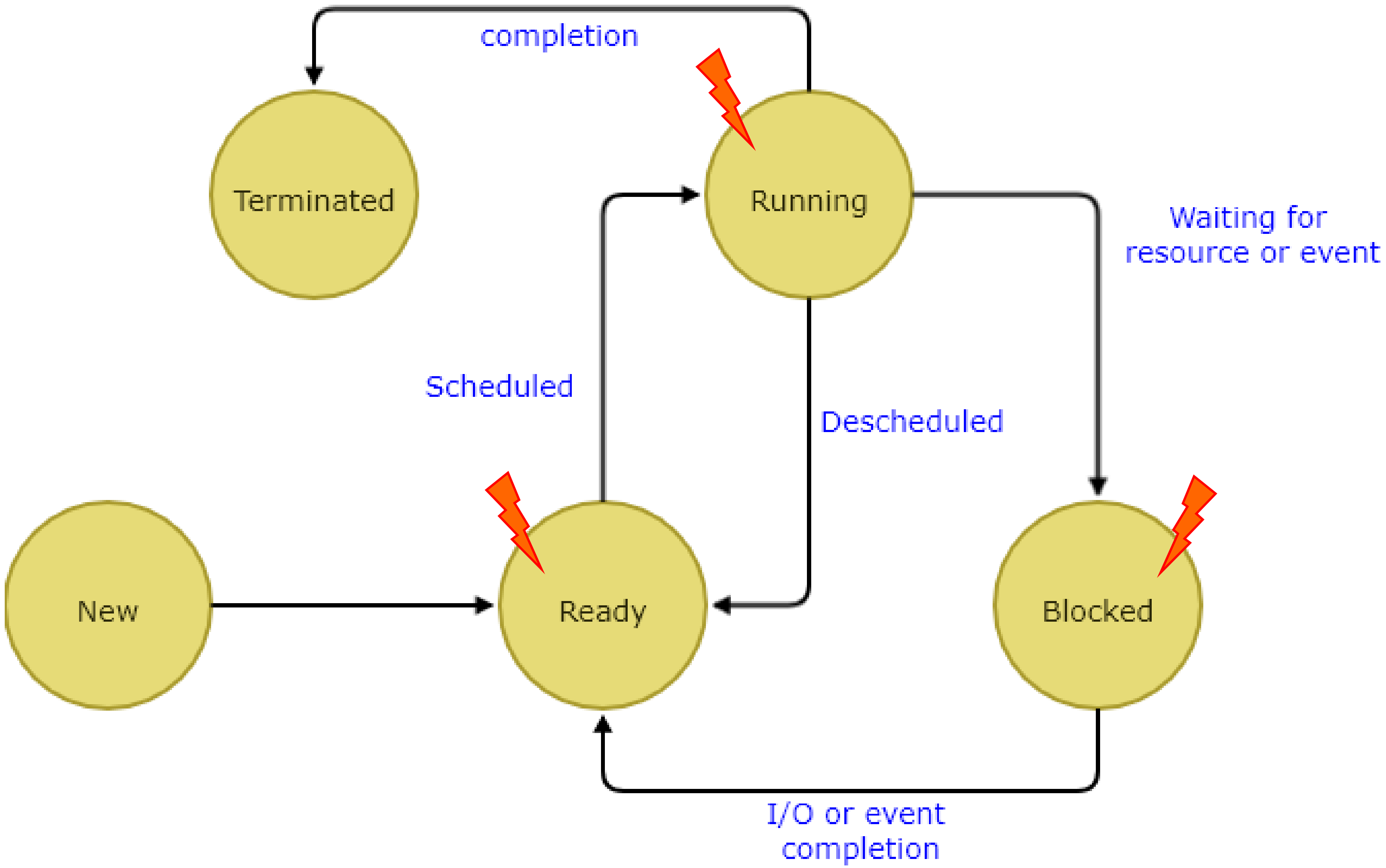
# Zombie Process (UNIX)

- When a process is in the terminated state, we call it a "zombie" process
  - When a process exits, OS still **keeps** the PCB of the process, so that the **parent can get** the information (if necessary)
    - exit status
    - resource usage
  - Parent uses waitpid() (and others) to inform OS to completely remove the zombie child process

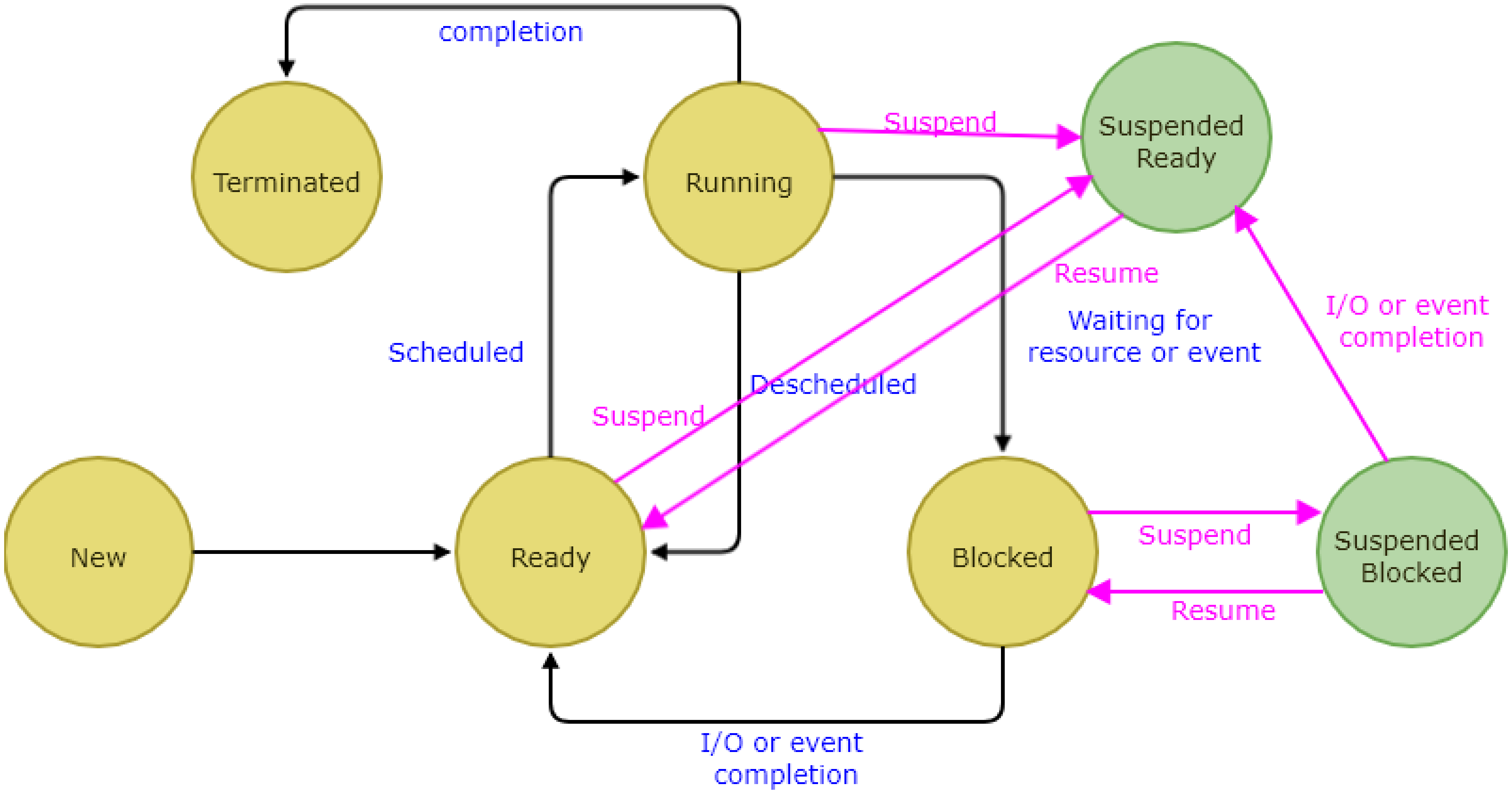
# Suspending a Process

- Temporarily deactivate the process, such that it is not being considered for processor scheduling
- Why doing so:
  - Upon user request
  - Request by the parent process
  - OS may decide to suspend a **blocked** process so as to free up the memory occupied by that process for another ready process
- A suspended process **must be** resumed by another process
- Difference between suspension and blocked
  - blocking is triggered by internal activity of the process, while suspend is coming from **external**

# Suspending a Process



# Suspending a Process



# Signals

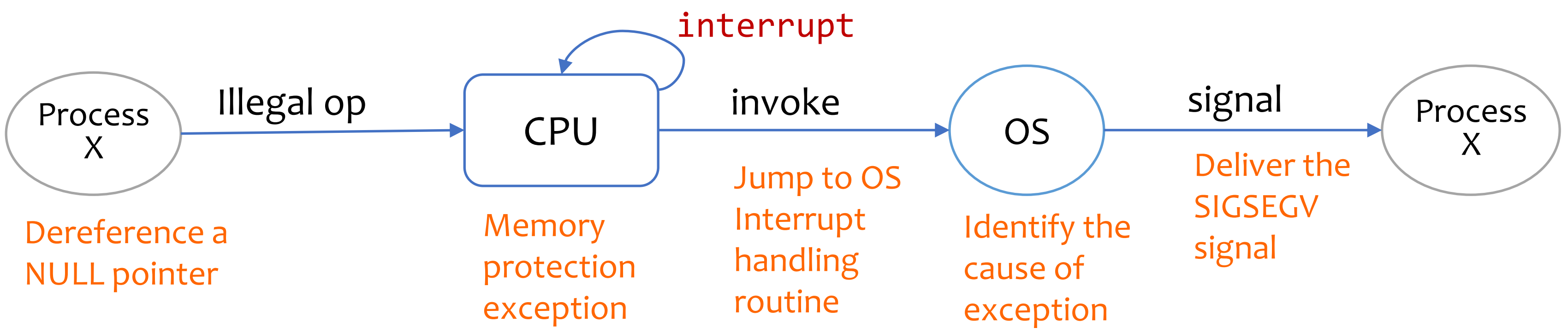
- Signals are used in UNIX systems to **notify** a process that an **event** has occurred
  - Sometimes being referred as "***Software Interrupt***"

- Basically implemented as system calls such as kill(), signal(), sigaction(), raise(), pause(), sigsuspend(), etc.

- Each signal is represented by a value/symbolic name
  - SIGINT: value = 2, generated when Ctrl-c is pressed
  - SIGCHLD: value = 17, generated when child process finishes execution or is terminated

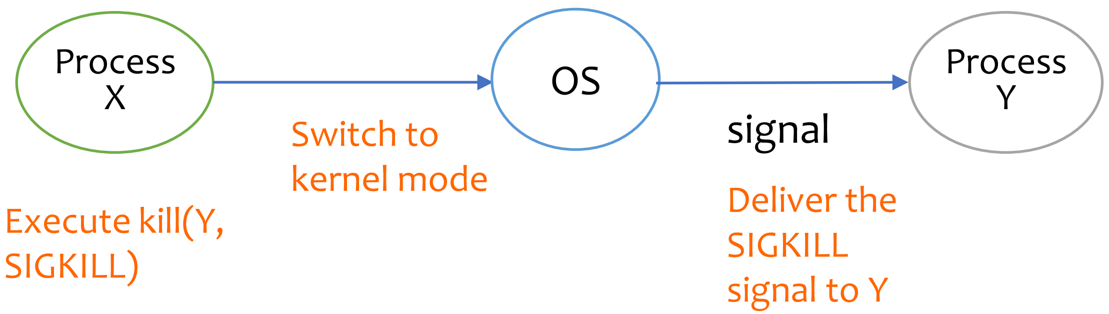
# Signals (2)

- A signal is generated by one software entity (in the occurrence of an event) to a target software entity
  - **Synchronous signal** – is triggered by the **current instruction** of the current running process itself and **is delivered** to **that** process by the OS immediately
    - e.g., illegal memory access, division by zero



# Signals (3)

- **Asynchronous signals** – are generated by external events / activities, which are **not triggered by the current activity** / action of the target process at the time of receiving the signal
  - i.e., arrive at unpredictable times during execution of the program
  - e.g., by the timer alarm
  - e.g., parent process using kill() system call to kill the child process



# Signals (4)

- A process can decide whether it wants to **catch**, **ignore** or **mask** a signal
  - Catching a signal involves specifying a routine (signal handler) in advance so that the OS will invoke that handler when the process receives that signal
    - the signal() or sigaction() system calls can be used by the program to specify the signal handler routine to the OS
  - Catching – Using OS's default action to handle the signal
  - Ignore – Inform OS that it does not want to handle that signal
  - Masking a signal is to instruct the OS not to deliver signals of that type until the process clears the signal mask
- SIGKILL and SIGSTOP cannot be caught, blocked or ignored

# Signals (5)

- How OS determines what a process will respond to a particular signal
  - A process's PCB contains a pointer to **a vector of signal handlers** (logically order by the signal number)
  - Each entry corresponds to the handler function for that entry (signal)

- A child process inherits the setting from its parent

- However, if use exec…() function to load a new program image, any signals that have the custom-made handlers **will be reset** to default setting

# Summary

- To manage and control a running process (application), OS needs some mechanisms to keep track on the current status of the process
  - Various data structures are needed
    - Process control block (PCB)
    - Process table and lists

- OS provides a set of operations for us to work with processes

- Signals were introduced in Unix systems to allow interactions between User Mode processes; the kernel also uses them to notify processes of system events.