# CMPT 300
# Operating System I

## 3.3 –Process 3
## Chapter 3

**Dr. Hazra Imran**

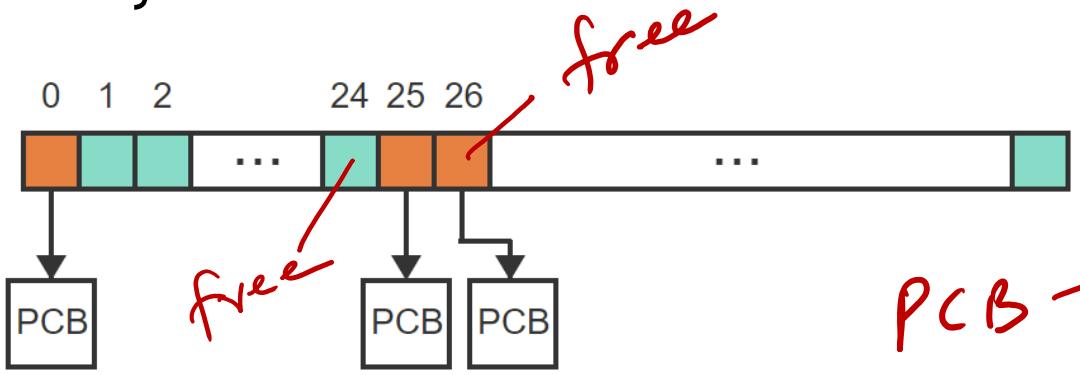
# Admin notes

- A2 available on canvas

# PCB organization

Two ways

1. An array of structure



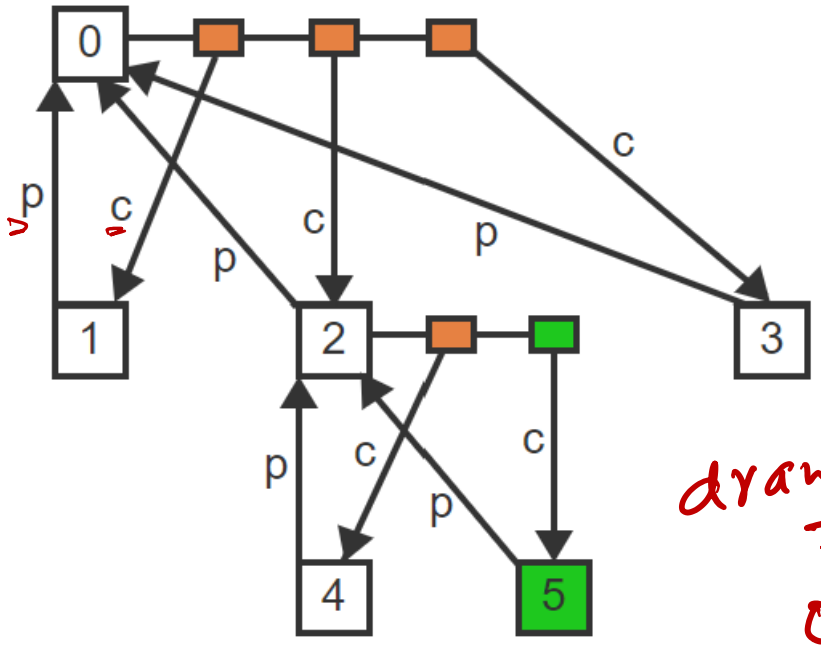

PCB – free
allocated

drawback ? wastage of memory

# PCB organization

Two ways

2. An array of pointers to dynamically allocated PCBs

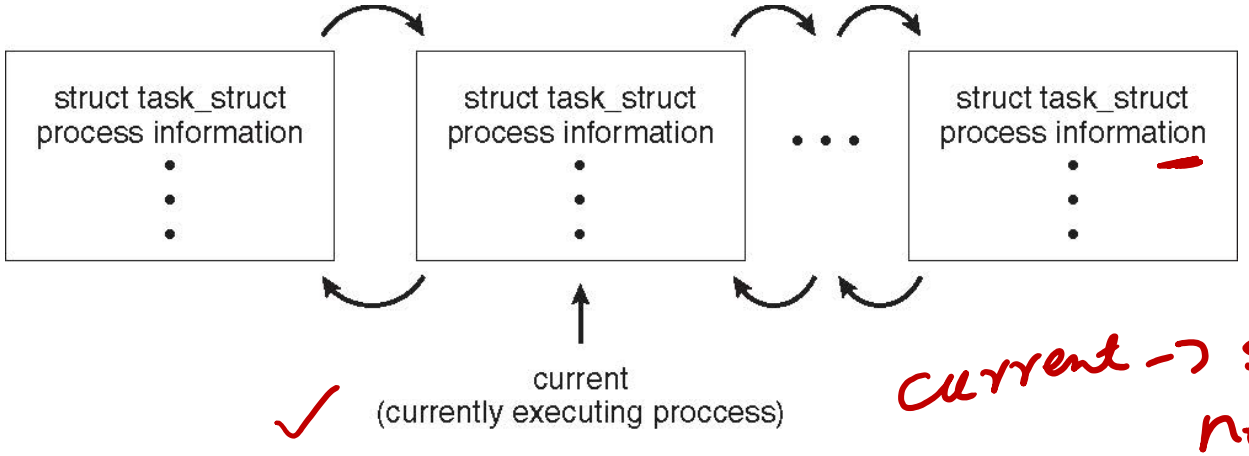



drawback
overhead.

# PCB 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 */
```




current -> state = new_state

# Process State Queues

- The OS maintains the PCBs of all processes in state queues.
- The OS maintains a queue for each of the states.
- PCBs of all processes in the same execution state are placed in the same queue.
- When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

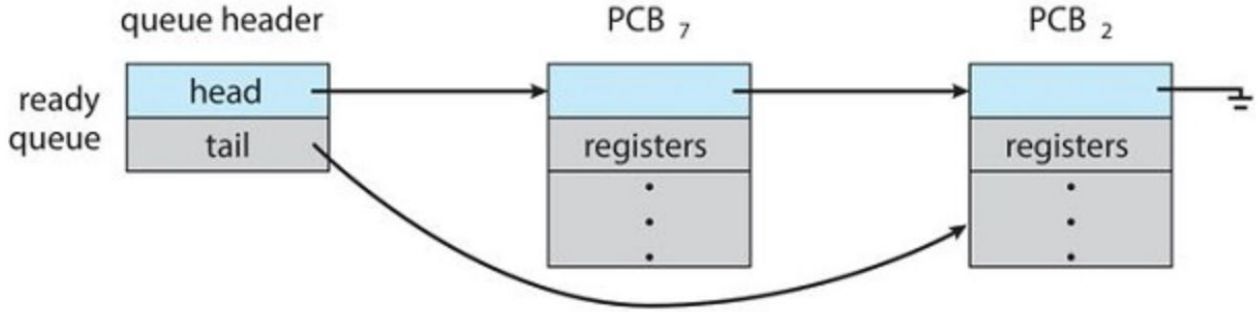
# Process Scheduling

- The goals are to:
  - Maximize CPU use
  - Give processes the CPU time that they need
- **Process scheduler** selects among available processes for next execution on CPU

# Process Scheduler

Maintains **scheduling queues** of processes

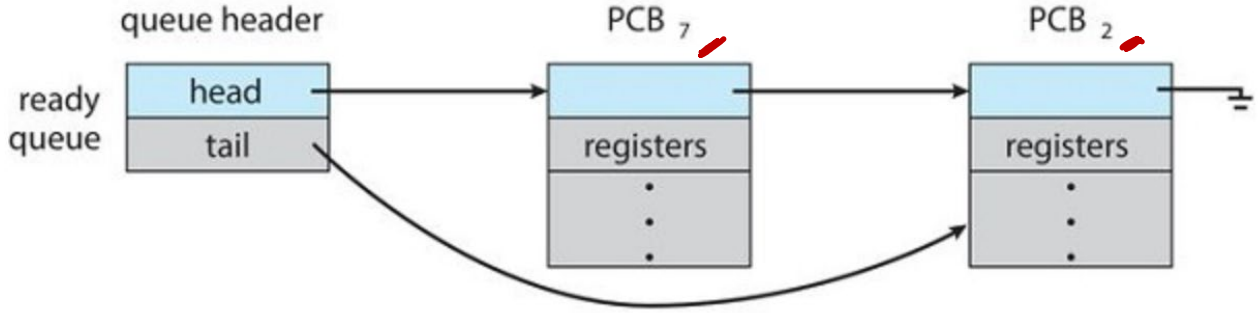
- **Ready queue:** set of all processes residing in main memory, ready and waiting to execute



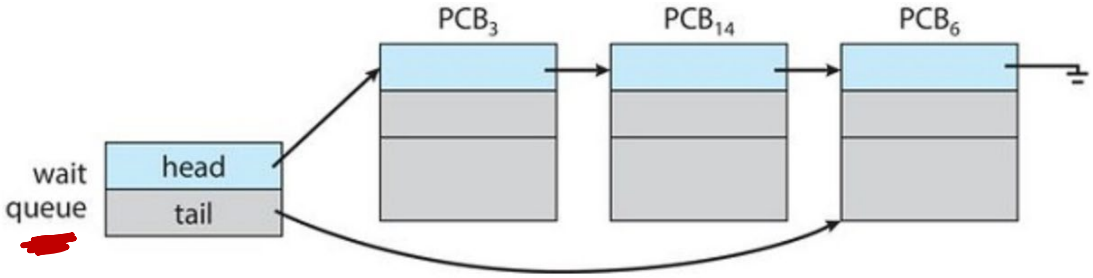
# Process Scheduler

Maintains **scheduling queues** of processes

- **Ready queue:** set of all processes residing in main memory, ready and waiting to execute




- **Wait queues**



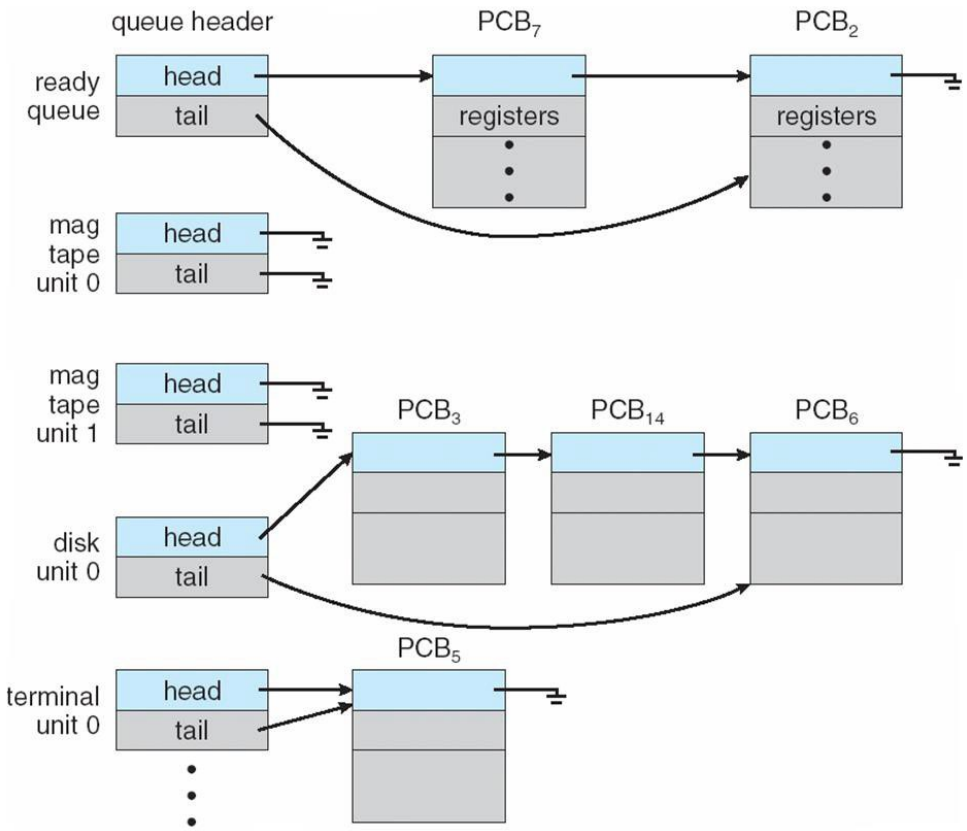

- Processes migrate among the various queues, depending (in part) on their state

# Ready Queue And Various I/O Device Queues

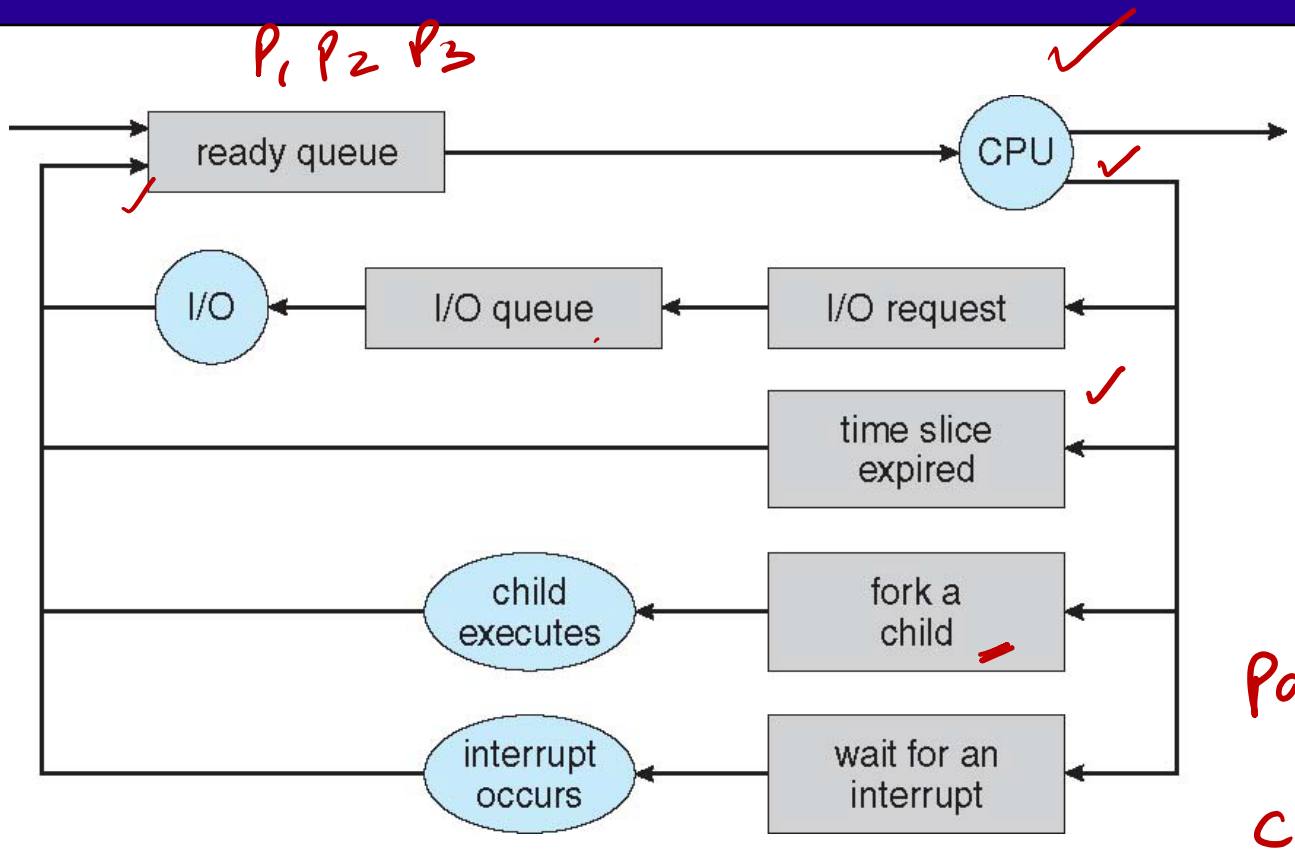
Processes are maintained in queues

**Ready queue** – set of all processes residing in main memory, ready and waiting to execute

**Device queues** – set of processes waiting for an I/O device



# Scheduling Queues




**Queueing diagram** represents queues, resources, flows

I/O bound v/s
CPU bound processes

# Scheduler Components

**Short-term scheduler** (or CPU scheduler)

- selects which process should be executed next and allocates CPU
- is invoked frequently (milliseconds), so it must be fast

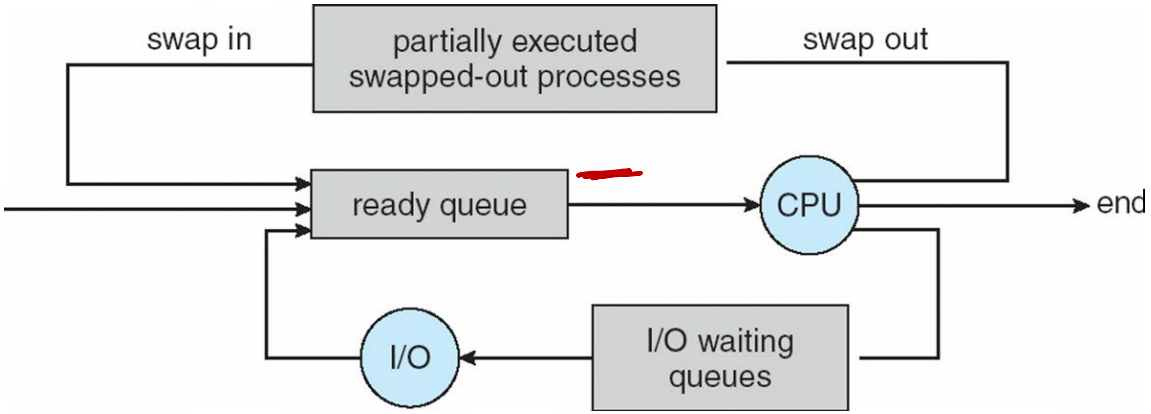
**Long-term scheduler** (or job scheduler)

- selects which processes should be brought into the ready queue
- is invoked infrequently (seconds, minutes), so it may respond slowly

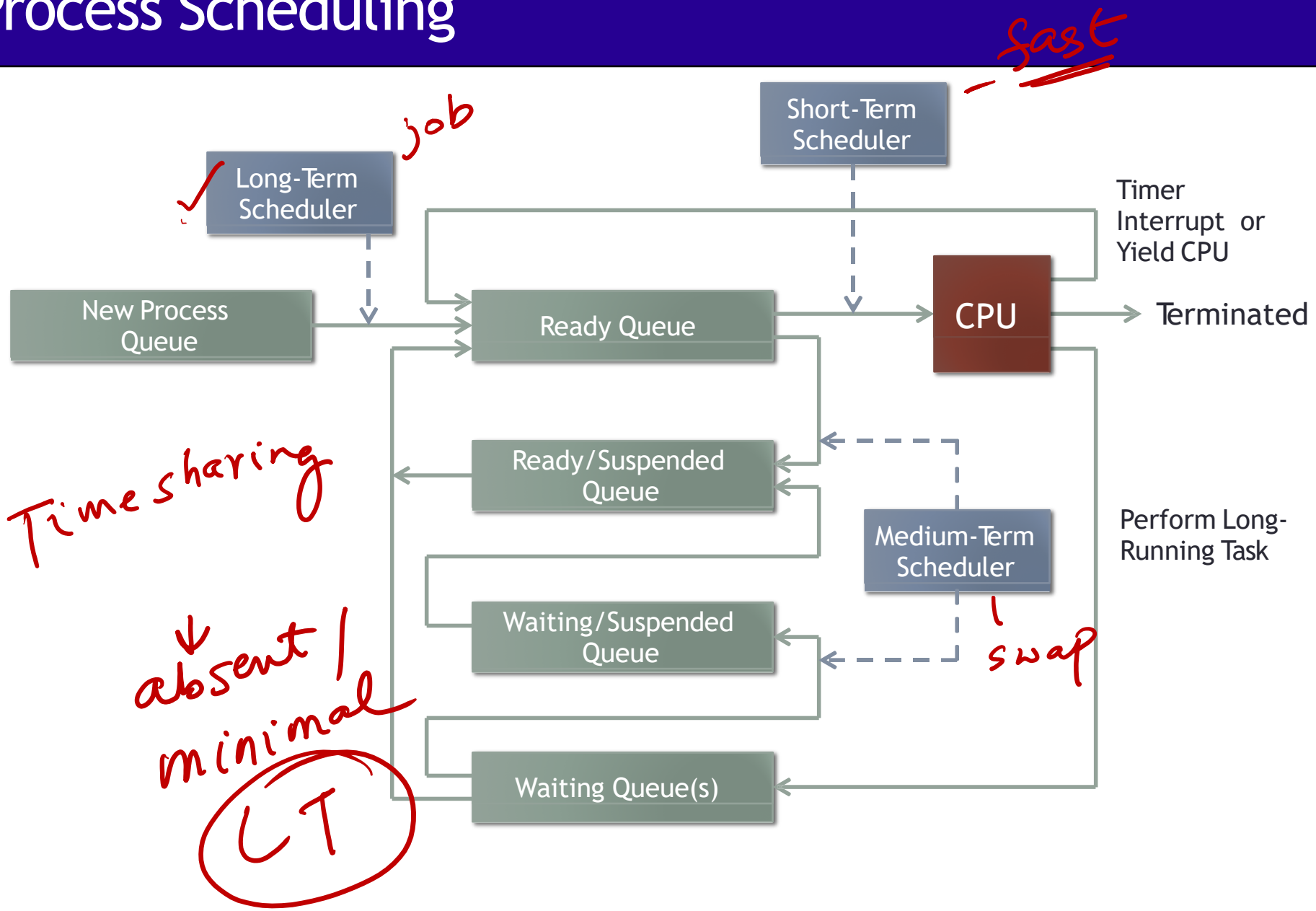
# Medium Term Scheduling

**Medium-term scheduler** can be added if degree of multiple programming needs to decrease

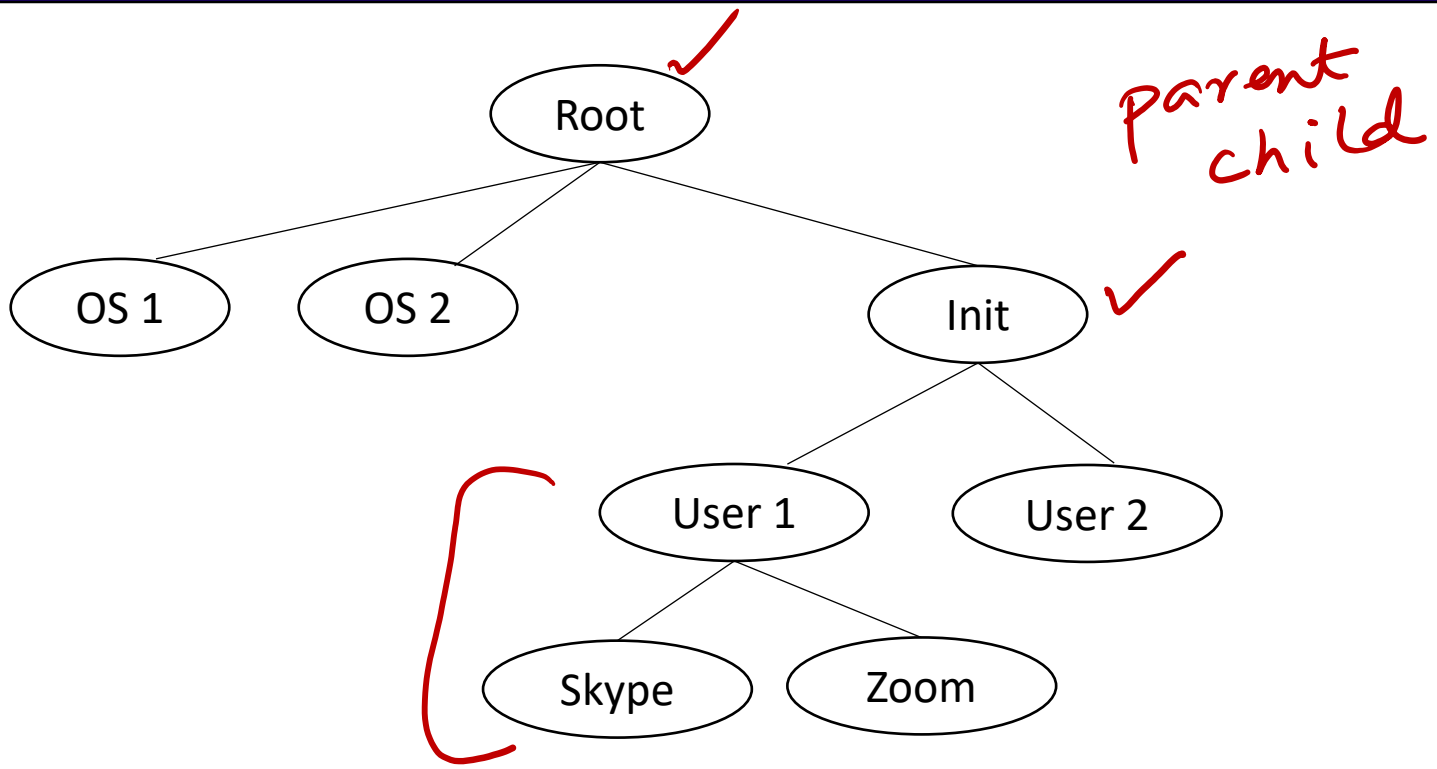
Remove process temporarily from memory, store on disk, bring back in from disk to continue execution: **swapping**



# Process Scheduling



# Process Creation Hierarchy



# Operation on processes

ps -el

active

```
user1@user1-VirtualBox:~$ ps -el
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0       1       0  2  80   0 - 41885 -      ?        00:00:01 systemd
1 S     0       2       0  0  80   0 -     0 -      ?        00:00:00 kthreadd
1 I     0       3       2  0  60 -20 -     0 -      ?        00:00:00 rcu_gp
1 I     0       4       2  0  60 -20 -     0 -      ?        00:00:00 rcu_par_g
1 I     0       5       2  0  80   0 -     0 -      ?        00:00:00 kworker/0
1 I     0       6       2  0  60 -20 -     0 -      ?        00:00:00 kworker/0
1 I     0       7       2  0  80   0 -     0 -      ?        00:00:00 kworker/0
1 I     0       8       2  0  80   0 -     0 -      ?        00:00:00 kworker/u
1 I     0       9       2  0  60 -20 -     0 -      ?        00:00:00 mm_percpu
1 S     0      10       2  0  80   0 -     0 -      ?        00:00:00 ksoftirqd
1 I     0      11       2  0  80   0 -     0 -      ?        00:00:00 rcu_sched
1 S     0      12       2  0 -40   - -     0 -      ?        00:00:00 migration
5 S     0      13       2  0   9   - -     0 -      ?        00:00:00 idle_inje
1 S     0      14       2  0  80   0 -     0 -      ?        00:00:00 cpuhp/0
5 S     0      15       2  0  80   0 -     0 -      ?        00:00:00 kdevtmpfs
1 I     0      16       2  0  60 -20 -     0 -      ?        00:00:00 netns
1 S     0      17       2  0  80   0 -     0 -      ?        00:00:00 rcu_tasks
1 S     0      18       2  0  80   0 -     0 -      ?        00:00:00 rcu_tasks
1 S     0      19       2  0  80   0 -     0 -      ?        00:00:00 rcu_tasks
1 S     0      20       2  0  80   0 -     0 -      ?        00:00:00 kauditd
1 S     0      21       2  0  80   0 -     0 -      ?        00:00:00 khungtask
1 S     0      22       2  0  80   0 -     0 -      ?        00:00:00 oom_reape
```

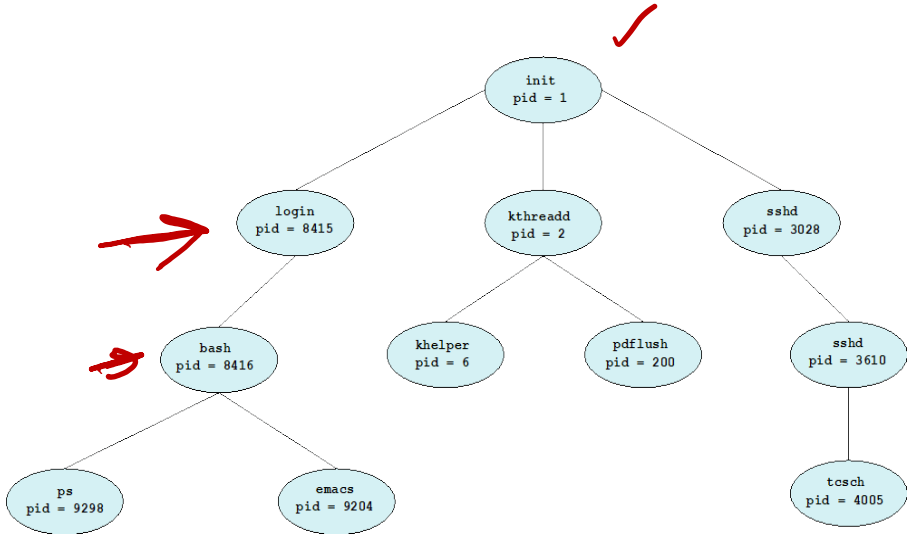
```
user1@user1-VirtualBox:~$ pstree
systemd─┬─ModemManager───2*[{ModemManager}]
        ├─NetworkManager───2*[{NetworkManager}]
        ├─accounts-daemon───2*[{accounts-daemon}]
        ├─acpid
        ├─avahi-daemon───avahi-daemon
        ├─colord───2*[{colord}]
        ├─cron
        ├─cups-browsed───2*[{cups-browsed}]
        ├─cupsd───dbus
        ├─dbus-daemon
        ├─gdm3─┬─gdm-session-wor─┬─gdm-x-session─┬─Xorg───{Xorg}
        │      │                 │               ├─gnome-session-b─┬─ssh-agent
        │      │                 │               │                 └─2*[{gnome+
        │      │                 │               └─2*[{gdm-x-session}]
        │      │                 └─2*[{gdm-session-wor}]
        │      └─2*[{gdm3}]
        ├─gnome-keyring-d───3*[{gnome-keyring-d}]
        ├─ibus-daemon─┬─ibus-dconf───3*[{ibus-dconf}]
        │             ├─ibus-engine-sim───2*[{ibus-engine-sim}]
        │             ├─ibus-extension-───3*[{ibus-extension-}]
        │             ├─ibus-ui-gtk3───3*[{ibus-ui-gtk3}]
        │             └─2*[{ibus-daemon}]
        ├─ibus-x11───2*[{ibus-x11}]
        ├─2*[kerneloops]
        ├─networkd-dispat
        ├─packagekitd───2*[{packagekitd}]
```

pstree

# Operation on processes

A process can **create** other processes to do work.

- The creator is called the parent and the new process is the child




How to share the resources?

1. Parent & child share all resources
2. no sharing
3. child can share subset of resources

How to execute?

1. concurrently (parent, children P1 C1)
2. Parent <wait> child

terminate

# Process Creation

- Two ways to create a process
  - Build a new empty process from scratch
  - Copy an existing process and change it appropriately

(+) no wasted work

(−) setup (complex)

# Process Creation

- Option 1: New process from scratch
  - Steps
    - Load specified code and data into memory; Create empty call stack
    - Create and initialize PCB (make look like context-switch)
    - Put process on ready list

Advantages:

Disadvantages:

repetition

# Process Creation

- Option 2: Clone existing process and change
  - Example: fork() and exec()
    - fork() system call creates new process
    - exec() system call used after a fork() to replace the process' memory space with a new program
  - Advantages:
  - Disadvantages:

# Process related system calls

- **fork()** creates a new child process
  - processes are created by forking from a parent
  - init process is ancestor of all processes
- **exec()** makes a process execute a given executable
- **exit()** terminates a process
- **wait()** causes a parent to block until child terminates

# fork()

- Creates and initializes a new PCB. Creates a new address space.
- Initializes the address space with a copy of the entire contents of the address space of the parent.
- Initializes the kernel resources to point to the resources used by parent (e.g., open files)
- Places the PCB on the ready queue.
- The only difference between the child and the parent is the value returned by fork.
- Fork returns twice
  - Returns the child's PID to the parent, "0" to the child

# Creating a process : An example

- When you log in to a machine running Linux, you create a shell process.
- Every command you type into the shell is a child of your shell process and is an implicit fork and exec pair.
- For example, you type a command, the OS “forks” a new process and then “exec” (executes) that command.

```
user1@user1-VirtualBox:~$ pwd
/home/user1
```

```
user1@user1-VirtualBox:~$ pwd & date
```

```
user1@user1-VirtualBox:~$ date
Sun 24 Jan 2021 10:16:50 PM PST
```

# Try!

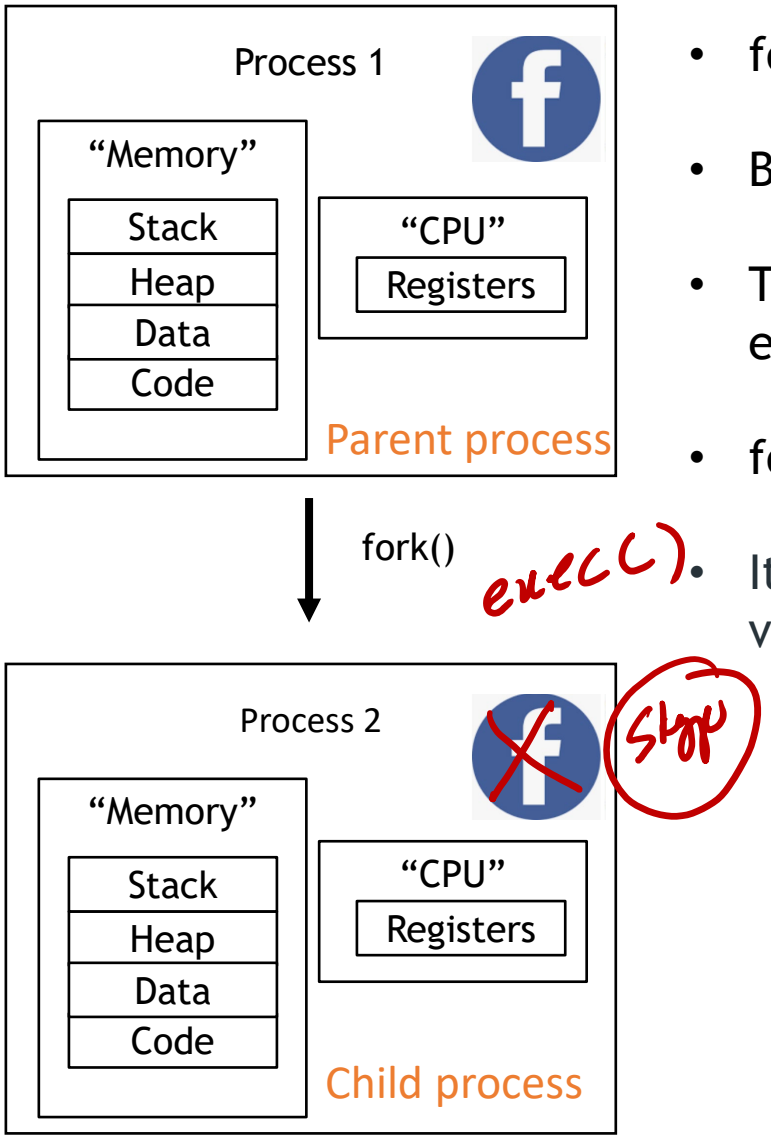
Open the terminal and type following

- pwd4 & date?

- pwd4 && date?

pwd && date

# fork()




- fork () creates a new child process
- Both processes run the code after the fork()
- The child is an almost exact copy of the parent except for PIDs
- fork() is called once, but returns twice
- It takes no parameters and returns an integer value.
  - ***Negative Value***: if creation of a child process was unsuccessful.
  - ***Zero***: Returned to the newly created child process.
  - ***Positive value***: Returned to parent process. The value contains process ID of newly created child process.