2. When an operating system wants to run multiple applications at once, the act of deciding which one to run is called **scheduling**. Tell me what you know about it, with regards to both single-core processors and multi-core processors.

- **General scheduling**
  - **Criteria**
    1. to maximize CPU utilization
    2. to maximize throughput (average process finish in a time unit)
    3. to minimize avg. process’s waiting time
    4. to minimize avg. process’s response time
    5. to minimize avg. process’s turnaround time

- **Timeline**

- **Algorithm & Effects**
  1. First-come, First-served
     - easy but may starve later coming processes if first coming ones take too long
  2. Shortest Job First
     - execute shortest burst time’s tasks first
     - hard to get a right estimate of burst time and tasks take too long
     - may fail to respond to urgent processes
  3. Round Robin
     - feasible but sometimes it fails to respond to vital processes
     - and the size time slices if not fairly designed may waste too much
  4. Multi-level queue
     - consist of multi-level queues where each represents a priority
     - processes may change between queues,
     - rather fair, but there is never a best solution

(see backwards for multi-core stuffs)
Multi-core scheduling

- Need to judge sequential tasks and parallelized tasks
- More a load balancing stuff and estimate burst time accurately
- Need to consider synchronization and cache coherence

Generally use more mutex locks or semaphores to protect

Some detailed one that when multiple

Push migration:
When some cores have too much loads, it push tasks to other not-so-busy cores

Pull migration:
When some cores are idle or not so busy, it pulls some tasks from other busy cores.
The communication through threads and processes are generally called inter process communication (IPC).

**IPC**

**IPC Implementation by theory**

1. Could be implemented via shared memory like mailboxes
2. Could also be done via message queue where a sender push some messages queue resides in the receiver's space
3. Could be implemented via interrupt, so when a message sent, the other thread got interrupted and check it in interrupt handler
4. Could be done via polling, aka. receiver checks it periodically

**Some features**

1. Could be unidirectional or bidirectional
2. A link could be established within more than 2 threads
3. More than one linke could be established between 2 threads

**Some unix implementation examples**

pipe, file, send/recv (socket)

**hardware involved**

- need to offer some shared memory and interrupt mechanism
- may offer some register as pointer of message queue if it has already offered many registers for context switching use (like Sun Sparc)

**Synchronization**

It is vital to prevent race condition in multi-thread tasks, i.e. other thread won't affect current thread to generate a different result.

A famous example could be read & write problems.

To prevent from race condition, we can

- don't share data
  - sounds stupid but it works
- disable interrupts to ensure one thing at a time won't work if multi cores involved
- use test and set mechanism
  - need some cache-coherence to help if we want it work as well in multi-core
- use special designed lock like:

  DisableInterrupt();
  DoStuffAboutCriticalSection();
  EnableInterrupt();

**Hardware involving**
- Test-and-set / compare-and-swap mechanism needs hardware offer some places of memory to help.
- Other software mechanism like mutex lock / semaphore needs hardware supports cache-coherence if using in multi-core environment.