Lecture.8.Scheduling.txt

- Why Scheduling?
    - Systems have limited time and resources
        - Due to this, performance and allocation of resources are very important
    - Scheduling is used to improve computer performance
    - The operating system makes 2 related kinds of decisions about resources:
        - Allocation:
            - What process gets what resource
        - Scheduling:
            - How long can a process keep a resource
                - Even if a process hasn't completed, it needs to release the resource after some time

- Basic Concept
    - There are two types of bursts of computing operations:
        - CPU burst
        - I/O burst
            - I/O burst is slow compared to CPU burst
    - A process is executed until it must wait for the completion of some I/O request
        - While the process waits for the completion of some I/O, other processes are allowed to execute
            - CPU bursts are followed by I/O bursts
    - On a multi-core system, this concept is extended to all processing cores on the system
    - Process execution consists of a cycle of CPU execution and I/O wait

- Types Of Scheduling
    - There are four types of scheduling:
        - Long term scheduling
            - The decision to add to the pool of processes to be executed
        - Medium term scheduling
            - The decision to add to the number of processes that are partially or fully in main memory
            - Part of the swapping function
        - Short term scheduling
            - The decision as to which available process will be executed by the processor
        - I/O scheduling
            - The decision to which process' pending I/O request shall be handled by an available I/O device
    - We will mostly focus on short term scheduling
        - However, similar logic applies to other types of scheduling algorithms

- CPU Scheduler
  - Selects from among the processes in ready queue, and allocates the CPU to one of them
  - The Ready Queue may be ordered in various ways
    - i.e.
      - FIFO queue
      - Priority queue
      - Tree
      - Unordered linked list
  - Scheduling decisions take place when processes are moved from one state to another. The process may:
    1. Switch from running to waiting state
       - i.e. As a result of an I/O request
    2. Switch from running to ready state
       - i.e. When an interrupt occurs
    3. Switch from waiting to ready state
       - i.e. Upon completion of I/O
    4. Terminates
    - Note:
      - Options (1) and (4) are non-preemptive or cooperative
      - Options (2) and (3) are preemptive
  - The state diagram shows how processes move through different states
    - Scheduling may occur:
      - During an I/O or Wait event
      - After an interrupt
      - When moving from Wait to Ready
      - When a process terminates
  - Processes need to be put in an order that increases performance and resource utilization is maximized

- Dispatcher
  - The Dispatcher module gives control of the CPU to the process selected by the short term scheduler
    - This is known as context switching
      - It requires switching to user mode, and jumping to the proper location in the user program to restart that program
    - Put simply, the job of the Dispatcher is to switch from one process to another
      - This is context switching; one process is stopped, and another process starts executing
      - The state of the process must be saved before it can be switched with another process
    - i.e.

```
|---------------|
| P_0 Executing |
|---------------| ----------|
        |                   |
        |                   |
        |                   |
|---------------|           |
| Save State    |           |
| Into PCB_0    |           |
|---------------|           |
        |                   |
        |                   |====> Dispatch Latency
        |                   |
|---------------|           |
| Restore state |           |
| from PCB_0    |           |
|---------------|           |
        |                   |
        |                   |
        |                   |
|---------------| ----------|
| P_1 Executing |
|---------------|
```

  - The time it takes for the Dispatcher to stop one process and start another process is called Dispatch Latency
  - The scheduler is much smarter than the Dispatcher

- Scheduling Criteria
  - When selecting the next process to execute, the algorithm factors in the following:
    - Max CPU Utilization
      - Keep the CPU as busy as possible
    - Max Throughput
      - The number of processes that complete their execution per unit of time
    - Min Turnaround Time
      - The amount of time to execute a particular process
        - The average turnaround time is calculated by adding up all the minimum turnaround time for each process, and then dividing the result by the number of processes
    - Min Waiting Time
      - The amount of time a process has been waiting in the ready queue
    - Min Response Time
      - The amount of time it takes from when a request was submitted until the first response is produced
  - Different scheduling algorithms are used, depending on the criteria
    - i.e. The implementation requires maximum CPU utilization

- First Come First Served Scheduling
  - The simplest scheduling algorithm is 'First-Come, First-Served';
    - Abbreviated as FCFS

- i.e.

Process	Burst Time
P1	24
P2	3
P3	3

  Suppose that the processes arrive in the order: P1, P2, P3.
  The Gantt Chart for the schedule is:
  ```
  |---------------------------------------|-----|-----|
  |                   P_1                 | P_2 | P_3 |
  |---------------------------------------|-----|-----|
  0                                       24    27   30
  ```
  Waiting time for P_1 = 0; P_2 = 24; P_3 = 27;
  Average waiting time: (0 + 24 + 27) / 3 = 17

  The wait time for P_1 is 0 because P_1 does not wait at all
  The wait time for P_2 is 24 time units
  The wait time for P_3 is 27 time units

- First Come First Served Scheduling (Continued)
  - i.e.
    - Suppose that the processes arrive in the order: P2, P3, P1.
      The Gantt Chart for the schedule is:
      ```
      |-----|-----|-------------------------------------|
      | P_2 | P_3 |                 P_1                 |
      |-----|-----|-------------------------------------|
      0     3     6                                    30
      ```
      Waiting time for P_1 = 6; P_2 = 0; P_3 = 3
      Average waiting time: (0 + 3 + 6) / 3 = 3
  - The average waiting time depends on how the processes are coming/scheduled
    - In the previous example, the average waiting time is 17, and in this example the average waiting time is 3
    - When short processes are behind long processes, they need to wait much longer
      - This is the Convoy effect
  - This example demonstrates that the FCFS scheduling algorithm is not very effective
    - However, if all incoming processes had the same burst time, then FCFS would be acceptable

- Shortest Job First (SJF) Scheduling
  - The processes are executed from shortest to longest CPU burst time
  - Each process is associated with the length of its CPU burst time
    - These lengths are used to schedule the process with the shortest (CPU burst) time
  - SJF is optimal; it gives the minimum average waiting time for a given set of processes

    - The difficulty is knowing the length of the next CPU request

- Example Of SJF (Shortest Job First)
  - i.e.

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

    SJF Gantt Scheduling Chart:
    ```
    |-----|------------|--------------|--------------|
    | P_4 |    P_1     |      P_3     |      P_2     |
    |-----|------------|--------------|--------------|
    0     3            9              16             24
    ```
    Waiting time for P_4 = 0, P_1 = 3, P_3 = 9, P_2 = 16
    Average waiting time: (0 + 3 + 9 + 16 + 24) / 4 = 7
  - SJF is similar to a priority queue
  - The previous burst is used to predict the next burst
    - This is done on the assumption that the next burst is similar to the previous one
    - Calculations must be performed on each step
  - FCFS and SJF are theoretical implementations on what might work
    - In reality, scheduling algorithms are much more complex
  - It is not possible for a process to acquire a resource and keep it forever

- Determining Length Of Next CPU Burst
  - Exponential averaging is used to determine the length of the next CPU burst
    - It is assumed that the next CPU burst will be similar in length to the previous ones
    - The process with the shortest predicted next CPU burst time is selected for execution
  - Predicting the length of the next CPU burst is done via exponential averaging
    - The process is:
      1. t_n = Actual length of n'th CPU burst
      2. theta_n + 1 = Predicted value for the next CPU burst
      3. 'alpha', 0 <= alpha <= 1
      4. Define: theta_n+1 = alpha * t_n + (1 - alpha) * theta_n
      - Note:
        - 'alpha' is a measure of much recent history and past are taken into account
          - It is commonly set to (1/2)
          - Recent history and past history are equally weighted

- Prediction Of The Length Of The Next CPU Burst

```
    - Exponential averaging example:
      - Assume you have the following history:
          CPU Burst (t_i)  =  - 6 4 6 4 13 13 13 ...
          Guess (theta_i)  = 10 8 6 6 5 09 11 12 ...
      - i.e. t_i = 6; theta_n = 10;
          theta_n+1 = (alpha * t_n) + ((1 - alpha) * theta_n)
          theta_n+1 = (0.5 * 6)     + ((1 - 0.5) * 10)
          theta_n+1 = 3 + 5
          theta_n+1 = 8
      - i.e. t_i = 4; theta_n = 8;
          theta_n+1 = (alpha * t_n) + ((1 - alpha) * theta_n)
          theta_n+1 = (0.5 * 4)     + ((1 - 0.5) * 8)
          theta_n+1 = 2 + 4
          theta_n+1 = 6
    - The initial guess is derived from some statistical earlier
      observation
    - The predicted value gets closer to the actual value after every
      iteration
      - However, a sudden change in Burst time can affect the
        efficacy of the prediction
      - If 'alpha' is 1, then previous prediction is not included in
        the calculation
        - If 'alpha' is 0, then previous actual times are not
          included in the calculation

  - Examples Of Exponential Averaging
    - If 'alpha' is 0, then the recent history does not count
      - This is because the equation:
          theta_n+1 = (alpha * t_n) + ((1 - alpha) * theta_n)
        Simplifies to:
          theta_n+1 = (1 - alpha) * theta_n
    - If 'alpha' is 1, then only the actual last CPU burst counts
      - This is because the equation:
          theta_n+1 = (alpha * t_n) + ((1 - alpha) * theta_n)
        Simplifies to:
          theta_n+1 = alpha * t_n
    - Expanding the formula yields:
        theta_n+1 = (alpha * t_n) + ((1 - alpha) * (alpha * t_n-1)) +
                    ((1 - alpha)^2 * (alpha * t_n-2)) + ... +
                    ((1 - alpha)^(n+1)) * theta_0
      - Since both 'alpha' and (1 - alpha) are less than or equal to
        1, each successive term has less weight than its predecessor
        - This is because each successive term has a greater
          exponent
    - If the system does not have huge changes in CPU bursts, then
      predicting the length of the next CPU burst can be reliable
      - If there is a dramatic change in CPU burst, it may take
        some time for the predications to match the changes

  - Example Of Shortest Remaining Time First
```

- The concepts of varying arrival times and preemption are added to the SJF algorithm analysis
- i.e.

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

    - Process P_1 arrives at time 0 and needs 8 CPU burst time
    - Process P_2 arrives at time 1 and needs 4 CPU burst time
    - Process P_3 arrives at time 2 and needs 9 CPU burst time
    - Process P_4 arrives at time 3 and needs 5 CPU burst time

  Preemptive SJF Gantt Scheduling Chart:

```
|-----|--------|-----------|-----------|----------------|
| P_1 |  P_2   |    P_4    |    P_1    |       P_3      |
|-----|--------|-----------|-----------|----------------|
0     1        5          10          17               26
```

  Average waiting time =
  [(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)] / 4 = 6.5msec
- P_1 is the first process to execute because its arrival time is 0; it is the first process the algorithm is aware of
- Since P_2 has a smaller burst time than P_1, when it arrives it will preempt P_1, and the algorithm will execute P_2 instead
  - This occurs at Arrival Time 1
- At time 3, P_3 arrives, but it does not preempt P_2, because it takes more burst time than P_2
  - P_2 will continue to run
- At time 4, P_4 arrives, but it does not preempt P_2, because it takes more burst time than P_2
  - Also, P_2 has been running for some time now, and its remaining time is less than P_4 and P_3's burst time
  - P_2 coninues to run
- P_2 executes from time 1 to time 5
  - At time 5, P_2 finishes and P_4 starts to execute
    - P_4 is selected because it has the smallest burst time out of P_3 and P_1
- P_4 executes from time 5 to time 10
  - At time 10, P_4 finishes and P_1 starts to execute
    - This is because P_1 requires less burst time than P_3
    - Note: P_1 requires a CPU burst time of 7, because P_1 was able to execute from time 0 to time 1
- P_1 executes from time 10 to time 17
  - At time 17, P_1 finishes and P_3 starts to execute
- P_3 executes from time 17 to time 26
- Average time is calculated in the following manner:
  - (10 - 1) is waiting time for P_1
    - P_1 starts at 0, gets 1 burst time, and then waits until

time = 10

    - (1 - 1) is waiting time for P_2
        - P_2 started at 1, but arrival time is also 1. Hence, there is no waiting time for P_2
    - (5 - 3) is waiting time for P_4
        - P_4 arrived at time = 3, and executed at time = 5. Thus, it waited for 2 time units
    - (17 - 2) is waiting time for P_3
        - P_3 starts at time = 17, but arrives at time = 2
    - Finally, each process' waiting time is summed up, and divided by the number of processes
    - Note: This only applies to shortest "remaining time first with preemption"
- Notes:
    - Regardless of the algorithm, the waiting time starts at the same moment
    - Arrival times for all processes are different
    - Processes with small burst times are given priority, and executed via preemption

- Round Robin (RR)
    - Each process gets a small unit of CPU time
        - This is referred to as 'time quantum', `q`
        - The time is usually 10-100 milliseconds
            - The size of the time quantum is initially determined through previous data
    - After the quantum time has elapsed, the process is preempted and added to the end of the ready queue
        - Every process gets the same amount of time quantum
    - If there are 'n' processes in the ready queue, and the time quantum is 'q', then each process gets (1/n) of the CPU time in chunks of at most 'q' time units at once
        - A process can finish early and not use all of its alotted time quantum
            - i.e. If a process is given 4 quantums, but it only uses 2 quantums, then the remaining will be used by another process. The system will not sit idly for 2 quantums
        - No process waits more than (n-1) 'q' time units
    - Timer interrupts every quantum to schedule next process

- Example Of RR With Time Quantum
    - i.e.

      Time quantum is 4ms

Process	Burst Time
P_1	24
P_2	3
P_3	3

```
        |---------|------------|

        The Gantt Chart is:
        |-------|-----|-----|-------|-------|-------|-------|-------|
        |  P_1  | P_2 | P_3 |  P_1  |  P_1  |  P_1  |  P_1  |  P_1  |
        |-------|-----|-----|-------|-------|-------|-------|-------|
        0       4     7    10      14      18      22      26      30

        - Each process is given 4 quantums
            - After that, another process is executed
        - P_1 starts at time = 0, and stops at time = 4
        - P_2 starts at time = 4, and stops at time = 7
            - P_2 is given 4 quantums, but it only needs 3
        - P_3 starts at time = 7, and stops at time = 10
            - P_3 is given 4 quantums, but it only needs 3
        - By time = 10, P_2 and P_3 have finished, and the only
          process left is P_1
            - P_1 gets 4 quantums until it finishes
                - i.e. Reaches a burst time of 24
                - If there are other pending processes, then P_1
                  will be preempted
            - At the end of every 4th quantum, the system checks to
              see if there are any other processes left
        - Typically, RR with time quantum has a higher average turn-
          around than SJF, but it has a better response; response time
          is smaller/shorter
            - The response time is giving a process CPU time
        - `q` should be large compared to context switch time
            - Context switch time is in the range of micro-seconds
                - Usually < 10 micro-seconds
            - `q` is in the range of milli-seconds
                - Between 10ms to 100ms

- Time Quantum & Context Switch Time
    - The size of the time quantum dictates its turnaround and
      response time
    - i.e. Assume a process time of 10 for each figure
        |-------------------------|
        |                         |
        |-------------------------|
        0                        10
            - If the time quantum is 12, and process time is 10, then
              there is no context switching, because time quantum is
              greater than process time
                - The process is allowed to finish
        |---------------|---------|
        |               |         |
        |---------------|---------|
        0                6       10
            - If time quantum is 6, and process time is 10, then there
              is 1 context switch
        |---------------|---------|
        |               |         |
        |---------------|---------|
        0                6       10
```
- If time quantum is 1, and process time is 10, then there are 9 context switches
  - A context switch takes place after quantum

```
 |--|--|--|--|--|--|--|--|--|
 |  |  |  |  |  |  |  |  |  |
 |--|--|--|--|--|--|--|--|--|
 0  1  2  3  4  6  7  8  9 10
```
- Ideally, we want to have the time quantum to be as small as possible to a CPU burst
  - But not much bigger or smaller

- Turnaround Time Varies With The Time Quantum
  - The time quantum should be as close as possible to a CPU burst
    - Anything bigger/smaller is not ideal
  - Turnaround time is the sum of the periods spent waiting in the ready queue, executing on the CPU, and doing I/O
  - As a rule of thumb, 80% of CPU bursts should be shorter than `q`
    - In other words, the CPU burst time a process needs should be smaller than `q`
    - i.e.

      Assume q = 10

Process	Burst Time
P_0	8
P_1	9
P_2	3
P_3	2
P_4	5
P_5	12
P_6	6
P_7	4
P_8	15
P_9	2

      - There are 10 processes, and 8 of them have a burst time under 10, and 2 have a burst time greater than 10. So, 80% of processes have a burst time smaller than `q`
  - Increasing 'q' does NOT always decrease turnaround time
    - This is very important
    - The type of processes and burst time affect what the ideal `q` value can be
  - i.e.

Process	Time

P_1	6
P_2	3
P_3	1
P_4	7

        - The average turnaround time if:
            - `q` = 1 is 11.0
            - `q` = 2 is 11.5
                - See below for proof
            - `q` = 3 is 10.8
            - `q` = 4 is 11.5
            - `q` = 5 is 12.2
            - `q` = 6 is 10.5
            - `q` = 7 is 10.5
    - If the time quantum is 2, then the average turnaround time is 11.5
        - The Gantt Chart is:

```
|----|----|----|----|----|----|----|----|----|----|
| P1 | P2 | P3 | P4 | P1 | P2 | P4 | P1 | P4 | P4 |
|----|----|----|----|----|----|----|----|----|----|
0    2    4    5    7    9   10   12   14   16   17
              Fin            Fin       Fin       Fin
P1 finishes at time = 14
P2 finishes at time = 10
P3 finishes at time = 5
P4 finishes at time = 17

Average turnaround time = [(P1 fin time) +
                          (P2 fin time)  +
                          (P3 fin time)  +
                          (P4 fin time)] / 4
Average turnaround time = (14 + 10 + 5 + 17) / 4
                        = 11.5
```

    - Suppose there are 3 incoming processes on a system, and each process requires 10 CPU bursts
        - If `q` = 1, then the average turnaround time is 29
        - If `q` = 10, then the average turnaround time is 20

- Priority Scheduling
    - A priority number (integer) is associated with each process
        - This is in addition to the burst time, arrival time, etc.
        - Processes with the highest priority are executed first
    - The CPU is allocated to the process with the highest priority (smallest integer highest priority)
        - It can be:
            - Preemptive
            - Non preemptive
    - Shortest Job First (SJF) is priority scheduling where priority is the inverse of predicted next CPU burst time

        - If next CPU burst time is smaller, then priority is higher
    - A problem with priority scheduling is: Starvation
        - In starvation, a low priority process may never execute
    - The solution to starvation is: Aging
        - As time progresses, the priority of unexecuted processes is increased
            - The longer a process waits, the greater its priority becomes; eventually it will have the greatest priority

- Example Of Priority Scheduling
    - i.e.

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

- The lower the priority number, the more important it is
    - In this case, P2 has the highest priority
        - P5 has the lowest priority
- Processes are executed in order of importance
    - From smallest priority number to largest priority

Priority scheduling Gantt Chart:

```
|----|--------|------------------------------|------|----|
| P2 |   P5   |              P1              |  P3  | P4 |
|----|--------|------------------------------|------|----|
0    1        6                             16     18   19
```

Average waiting time = 8.2msec

- Priority Scheduling With Round Robin
    - May decrease average waiting time
        - Each process has to wait less for CPU time
    - i.e.

Process	Burst Time	Priority
P1	4	3
P2	5	2
P3	8	2
P4	7	1
P5	3	3

- The process with the highest priority runs first
    - P4 has the highest priority
- The processes with the same priority run Round Robin
    - P2 and P3 have the same priority

```
        Priority scheduling Gantt Chart:
        (With 2ms time quantum)
        |-------|----|----|----|----|----|-----|----|----|----|----|
        |   P4  | P2 | P3 | P2 | P3 | P2 | P3  | P1 | P5 | P1 | P5 |
        |-------|----|----|----|----|----|-----|----|----|----|----|
        0       7    9   11   13   15   16    20   22   24   26   27
               Fin                       Fin   Fin            Fin  Fin
        P1 finishes at time = 26
        P2 finishes at time = 16
        P3 finishes at time = 20
        P4 finishes at time = 7
        Average turnaround time = (26 + 16 + 20 + 7) / 4
                                = 17.25 msec
    - Every time quantum, the system checks if there's another process
      with the same priority or higher priority
    - The execution order of processes with the same priority is
      nondeterministically decided
        - i.e. If P2 and P3 have the same priority, then P2 may run
               before P3, or P3 may run before P2. Either option can
               happen
    - Processes with the same priority are kind of concurrently
      executed

- Multi Level Queue (1)
    - With priority scheduling, there are separate queues for each
      priority class
        - i.e. The queue for processes with a priority of 2 is
               different from the queue that contains processes with a
               priority of 3
        - i.e. Diagram Example
            /----|----|----|----|-----\
            | T0 | T1 | T2 | T3 | ... | priority = 0
            \----|----|----|----|-----/
            /----|----|----|----|-----\
            | T5 | T7 | T8 | T9 | ... | priority = 1
            \----|----|----|----|-----/
            /-----|-----|-----|-------\
            | T_x | T_y | T_z |  ...  | priority = n
            \-----|-----|-----|-------/
    - Works well when scheduling is combined with Round Robin (RR)

- Multi Level Queue (2)
    - Prioritization is based on process type
        - Naturally, some processes have a higher priority
            - i.e. Real time processes have the highest priority
        - i.e.
            Highest priority
                -> Real-time processes
                -> System processes
```

-> Interactive (foreground) processes
-> Batch (background) processes
Lowest priority

  - It is important that processes are calculated correctly, and by the deadline
    - If the deadline is missed then the execution of processes is not correct
      - This is why real-time processes have the highest priority
  - Different response time requires different scheduling needs
    - There is no one size fits all approach to scheduling
      - Everything depends on context
  - Separate queues might be used for foreground and background processes, and each queue might have its own scheduling algorithm

- Multi Level Feedback Queue
  - Processes can move between queues
  - The idea is to separate processes according to the characteristics of their CPU bursts
  - A process can move between the various queues
    - Processes can be upgraded and moved to a higher priority queue, and they can be downgraded and moved to a lower priority queue
    - Aging (the solution to Starvation) can be implemented this way
  - The multi-level-feedback-queue scheduler is defined by the following parameters:
    - Number of queues
    - Scheduling algorithms for each queue
    - Method used to determine when to upgrade/promote a process
    - Method used to determine when to downgrade/demote a process
    - Method used to determine which queue a process will enter when that process needs service

- Example Of Multi Level Feedback Queue
  - Suppose there are 3 queues with different priorities:
    - Queue 0
      - Round Robin with time quantum = 8 ms
      - Priority = 1
    - Queue 1
      - Round Robin with time quantum = 16 ms
      - Priority = 2
    - Queue 2
      - FCFS (first-come, first-served) scheduler
      - Priority = 3
  - Scheduling
    - A new job enters Queue 0, which relies on FCFS scheduling
      - When it executes on the CPU, it has 8 milliseconds
      - If the job does not finish in 8 milliseconds, it is

moved to Queue 1

    - When a job from Queue 0 enters Queue 1, it is served via FCFS
        - Once it is time for the job to execute, it receives 16 milliseconds of CPU time
        - If the job does not finish in 16 milliseconds, it is preempted and moved to Queue 2
  - Dynamically moving jobs/threads/processes from one queue to another is really effective, especially in practice

- Thread Scheduling
  - Distinction between user level and kernel level threads
  - On most operating systems, kernel level threads are being scheduled
  - On a many-to-one and many-to-many model, the thread library schedules user level threads to run on LWP (Lightweight process)
    - This is known as process-contention scope (PCS)
      - Scheduling competition is within the process
    - Typicall done via priority set by programmer
  - Kernel thread scheduled onto available CPU is system-contention scope (SCS)
    - Scheduling competition is among all threads in the system
  - Systems using the one-to-one model, such as Windows and Linux, schedule threads only using SCS

- Multiple Processor Scheduling (1)
  - Before multi-core/processor systems, scheduling was simple because all processes would run on the same CPU
  - CPU scheduling is much more complex when multiple CPUs are available
    - Which process should run on which core?
  - Multiprocess may be any one of the following architectures:
    - Multicore CPUs
    - Multi threaded cores
      - i.e. Hardware threads inside the core
    - NUMA (non-uniform memory access) systems
      - The CPUs are connected by a shared system interconnect, so that all CPUs share one physical address space
        - In other words, CPUs can access local memory of other CPUs
      - A CPU accessing its own local memory is fast, but a CPU accessing another CPU's local memory is slow
        - Hence, the term: non-uniform
          - Sometimes the access to memory is fast, and other times it is slower
    - Hetergeneous multi processing
      - Some cores are faster/slower than others
        - This is mostly practiced on mobile systems
          - i.e. Phones having low power cores that are purposely clocked at a lower frequency to

preserve battery life
- Some cores may have access to other resources to speed up computation time
  - i.e. Combination of a multicore processor with a GPU

- Multiple Processor Scheduling (2)
  - There are two types of multi-processor scheduling:
    - Asymmetric multiprocessing (AMP)
      - There is only one core, called Master core
        - It is the only core that can access system data structures
        - Decides which task other cores will carry out
          - Essentially, the master core does scheduling
          - However, the master core can be a potential bottleneck because it is always busy, and other cores may be idling
    - Symmetric multiprocessing (SMP)
      - Each processor is self scheduling
      - All modern operating systems support SMP
      - There are 2 different approaches to SMP:
        1. All threads may be in a common ready queue
           - i.e. Diagram Of Common Ready Queue

```
|----|----|----|-----|-----|
| T0 | T1 | T2 | ... | T_n |
|----|----|----|-----|-----|
            /|\
           / | \
          /  |  \
         /   |   \
        /    |    \
       /     |     \
      /      |      \
  core_0   core_1   core_n
```

        2. Each processor may have its own private queue of threads
           - i.e. Diagram Of Per-Core Thread

```
|----|      |----|      |----|
| T0 |      | T0 |      | T0 |
|----|      |----|      |----|
| T1 |      | T1 |      | T1 |
|----|      |----|      |----|
| T2 |        |         | T2 |
|----|        |         |----|
| T3 |        |           |
|----|        |           |
   |          |           |
   |          |           |
   |          |           |
core_0      core_1      core_n
```

- Multicore Processors
  - Recent trend to place multiple processor cores on same physical chip
    - This makes the system faster, and consume less power
      - This is good for performance and battery life
  - Multiple threads per core is also growing
    - Takes advantage of memory stall to make progress on another thread while memory retrieve happens
      - i.e. Diagram

```
                 |---|---|---|---|---|---|   C = compute
  thread -----> | C | M | C | M | C | M |       cycle
                 |---|---|---|---|---|---|   M = memory
                 ------------------------>       stall
                            time                 cycle
```

      - Every computation has a computing cycle and a memory access cycle
        - If the data is inside CPU registers, then memory access is very fast
          - But only a limited amount of data can be inside the registers; most of the working data is in cache or RAM
        - A memory access cycle takes more time than a computing cycle
          - A memory access can be:
            - Accesing cache, ram, HDD
            - Moving memory from one storage to another
      - During a memory stall cycle, the CPU is not doing anything
        - The processor can spend up to 50% of its time waiting for data to become available from memory

- MultiThreaded MultiCore System
  - Each core has > 1 hardware thread
    - 2 or more threads are running in parallel
      - If one thread has a memory stall, the CPU will switch to another thread
      - i.e. Diagram

```
                |---|---|---|---|---|---|---|
  ------------>| C | M | C | M | C | M | C |
  thread_1      |---|---|---|---|---|---|---|
            |---|---|---|---|---|---|---|     c = compute
  --------->| C | M | C | M | C | M | C |     m = memory
  thread_0 |---|---|---|---|---|---|---|         stall
           -------------------------------->
                          time
```

        - The threads are synchronized so that while one thread waits for data to be transferred from memory, the other thread can use the CPU
          - The CPU switches between the two threads
  - From an OS perspective, each hardware thread maintains its

architectural state, such as instruction pointer and register set, and thus appears as a logical CPU that is available to run a software thread
  - Even though there is 1 physical CPU, to the operating system it seems as if there are 2 logical CPUs
  - The software thread is mapped to a particular hardware thread
  - This technique is known as chip multi-threading (CMT)

- Example Of Multi Level Feedback Queue
  - Chip multi-threading (CMT) assigns each core muliple hardware threads
  - Intel refers to this as hyperthreading or simultaneous multi-threading (SMT)
  - On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors
    - 4 cores x 2 hardware threads per core = 8 logical cores
    - This greatly increases throughput

- MultiThreaded MultiCore System
  - The resources of the physical core (such as caches & pipelines) must be shared among its hardware threads. Therefore, a processing core can only execute one hardware thread at a time
  - A multi-threaded, multi-core processor requires two levels of scheduling
    1. The operating system decides which software thread runs on which logical CPU
       - The OS can use any scheduling algorithm (FCFS, SJF, RR, etc.)
       - The OS maps software threads to hardware threads
    2. Each core has to decide which hardware thread to run on the physical core
       - How is this done?
         - Usually done via Round Robin
         - Priority can also be used
           - i.e. Dynamic urgency value ranging from 0 to 7
       - This scheduler maps hardware threads to processing core
  - Example of mapping software threads to hardware threads to CPU
    - i.e. Diagram

```
|| ||   || ||
|| ||   || ||   <----- Software threads (x4)
|| ||   || ||
|| ||   || ||
 \ /     \ /
  \       /
   \     /
    \   /
     \ /
      \
      /\
     /  \
    /    \
   ||    ||
   ||    ||     <----- Hardware threads (x2)
   ||    ||            (Logical processors)
   ||    ||
    \    /
     \  /
      \/
|------------|
| Processing |  <----- Physical CPU (x1)
|    core    |
|------------|
```

- In this example, each hardware thread is mapped to 2 software threads
- 1 processing core is responsible for 2 hardware threads
- There needs to be 2 types of schedulers in a multi-threaded or multi-core system
    - The first scheduler maps software threads to hardware threads
    - The second schedular maps hardware threads to the processing core

- Multiple Processor Scheduling Load Balancing
    - If symmetric multi-processing (SMP) is incorporated in a system, it need to keep all CPUs loaded for efficiency
        - This is known as load balancing
    - If load balancing is not done correctly, then the performance of the system will be severely degraded
    - Load balancing attempts to keep workload evenly distributed
        - The system tries to avoid putting more work on a single CPU
        - Allows us to get maximum performance from the system
    - There are two general approaches to load balancing
        - Push migration
            - A periodic task checks the load on each processor, and if one processor has a higher load than the other, it will push the task(s) from the overloaded CPU to other CPUs (that are not overloaded)
        - Pull migration
            - Idle processors pulls waiting task from busy waiting
                - If a processor is idle, it will pull a task from another processor and work on it

- Multiple Processor Scheduling (Processor Affinity)
    - When CPUs use their own dedicated cache, they operate faster
    - Moving a process from one CPU to another may cause the system to lose the contents of the initial CPU's cache
    - When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread

- Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off
  - To avoid losing processor cache content, affinity is specified for the thread:
    - Soft affinity
      - The operating system attempts to keep a thread running on the same processor
        - However, it does not guarantee
    - Hard affinity
      - Allows a process to specify a set of processors it may run on
        - Since some cores share the same memory, a process is allowed to run on any of the cores that can access shared memory
- Note: None of these problems are present in a single-CPU system; they are only present in multi-processor/core systems. However, multi-processor/core systems are much more powerful than single core systems

- NUMA & CPU Scheduling
  - NUMA stands for Non Uniform Memory Access
  - A system interconnect allows all CPUs in a NUMA system to share one physical address space
    - This allows the CPUs to access each others memory
  - If the operating system is NUMA aware, it will assign memory closest to the CPU the thread is running on
    - This technique helps in improving performance
  - i.e. Two physical processor chips each with their own CPU and local memory

```
|------------------------------------------------|
|                                                |
|   |---------|                  |---------|     |
|   |  CPU_0  |-------->|        |  CPU_1  |     |
|   |---------|         |        |---------|     |
|        |              |             |          |
|        | Fast         | Slow        | Fast     |
|        | Access       | Access      | Access   |
|        |              |             |          |
|   |----------|        |        |----------|    |
|   | Memory_0 |        |------->| Memory_1 |    |
|   |----------|                 |----------|    |
|                                                |
|------------------------------------------------|
```

    - 'CPU_0' has quick access to its own local memory, 'Memory_0'
      - Same principle applies to 'CPU_1' for its own memory, 'Memory_1'
    - If 'CPU_0' or 'CPU_1' tried to access the other CPU's memory, the operation would be (relatively) slow

  - In general, if a CPU tries to access another CPU's local memory, the access time is slow
- If the OS is NUMA-aware, then it will assign memory that is closest to the CPU
  - i.e. If a process is executed on 'CPU_0', then the OS will transfer the process' memory to 'Memory_0'

- Real Time CPU Scheduling (1)
  - A real time system can be divided into 2 main groups:
    1. Soft real time system
       - The task may be serviced by its deadline, but the system makes no guarantees
         - If the task is not completed by its deadline, the system may still be considered 'OK'
       - i.e.
         - Online shipping may take a few more/less days, and it is still acceptable
         - Loading a webpage during busy hours will take a few more seconds, and this is acceptable
    2. Hard real time system
       - Also known as firm real time system
       - The task must be serviced by its deadline
         - If a task misses its deadline, the system is considered to be in failure
       - i.e.
         - Automatic braking system
           - The system should react in 50ms; anything more is very bad and may cause fatal damage
         - Automatic laser cutting machine
  - Each real time system requires a deadline

- Real Time CPU Scheduling (2)
  - Event latency is the amount of time that elapses from when an event occurs to when it is serviced
    - If an event occurs, it needs to be serviced by the interrupt service routine (ISR)
    - The interrupt needs to be processed, then the execution of the service routine needs to be set
  - There are two types of latencies in real time CPU scheduling that affects performance
    1. Interrupt latency
       - Time from arrival of interrupt to start of routine that services interrupt
    2. Dispatch latency
       - Time for schedule to take current process off CPU and switch to another
  - There is always latency when responding to an event

- Interrupt Latency
  - Defined as the period of time between the arrival of an

interrupt at the CPU to the start of the routine that services the interrupt
    - Has two components to determine interrupt type, and perform a context switch
        - After this, the interrupt service routine (ISR) takes over
            - Dispatch latency decides when the ISR is executed
    - When an interrupt occurs:
        - The current instruction that is executed is completed, and the type of interrupt that just occurred is determined
        - The state of the current process is saved, and then the interrupt is serviced using interrupt service routine (ISR) that was determined in the previous step

- Dispatch Latency
    - Dispatch latency takes place after interrupt latency
    - Define as the amount of time required for the scheduling dispatcher to stop one process and start another
    - Has two major components:
        - Conflict phase
        - Dispatch phase
    - The conflict phase of dispatch latency has 2 steps:
        1. Preemption of any process running in kernel mode
        2. Releases resource(s) held by low priority processes, allowing high priority processes to acquire the resource(s)
    - The dispatch phase schedules the high priority process onto an available CPU
        - Occurs for all types of systems

- Priority Based Scheduling
    - For real time scheduling, scheduler must support preemptive, priority based scheduling
        - But only guarantees soft real time scheduling
    - For hard real time scheduling, the scheduler must also provide the ability to meet deadlines
        - This must be guaranteed
    - Processes have new characteristics; periodic ones require CPU at constant intervals
        - Has 3 components:
            - Processing time, t
            - Deadline, d
            - Period, p
        - $0 <= t <= d <= p$
            - The correlation between these 3 components is:
                - Processing time is always less or equal to than deadline
                - Deadline is less than or equal to the period
        - Rate of periodic task is 1/p
            - Scheduling can be done on periodic task
                - i.e. Tasks with shorter periods have higher priority

- No Rate Monotonic Scheduling
    - Assume there are two processes: P_1, P_2;
        - Periods of each process are: p1 = 50, p2 = 100; respectively
        - Processing times: t1 = 20, t2 = 35; respectively
    - The question is: Is it possible to schedule these tasks so that each meets its deadlines?
        - P_1 has a deadline at 50, and another at 100
        - P_2 has a deadline at 100
    - If P_2 executes first and then P_1, P_1 will miss its deadline
    - CPU utilization for P1 = t1/p1 = 20/50 = 0.4 = 40%
    - CPU utilization for P2 = t2/p2 = 35/100 = 0.35 = 35%
    - Suppose P2 has a higher priority than P1
        - P1 will miss the deadline
            - This is bad, and we are trying to avoid this
            - Missing the deadline will invalidate the system or cause it to fail

- Rate Monotonic Scheduling
    - Solves the problem from before; no rate monotonic scheduling
    - A priority is assigned based on the inverse of its period
        - Shorter periods = Higher priority
        - Longer periods = Lower priority
    - Since P1 has a shorter period than P2, P1 is assigned a higher priority than P2, and is executed first
    - Both P1 and P2 meets the deadline
        - This is desirable
    - The gaps in the diagram indicate that no process is running

- Missed Deadlines With Rate Monotonic Scheduling
    - Even with rate monotonic scheduling, a process may miss its deadlines
    - Assume there are 2 processes: P1 and P2
        - The periods of each process are: p1 = 50, p2 = 80
        - The processing times of each process are: t1 = 25, t2 = 35
        - The deadline for P1 is 50, and 100
        - The deadline for P2 is 80
    - The total CPU utilization of the two processes is: (25/50) + (35/80) = 0.94%
        - If CPU utilization is high, it is harder to schedule processes and meet their deadlines
    - If P1 runs first then P2 will miss its deadline at 80, and if P2 runs first then P1 will miss its deadline at 50
    - Despite being optimal, rate-monotonic scheduling has a limitation
        - For N processes, the worst case CPU utilization: N(2^(1/N) - 1)
        - High CPU utilization results in processes missing deadlines

- Earliest Deadline First Scheduling (EDF)
    - Priorities are assigned according to deadlines

    - The earlier the deadline -> The higher the priority
    - The later the deadline -> The lower the priority
  - Assume there are 2 processes: P1 and P2
    - The period for each process is: p1 = 50, p2 = 80
    - The processing time for each process is: t1 = 25, t2 = 35
    - At the beginning of its next period at time 50, EDF scheduling allows process P2 to continue running
      - In the previous scheduling algorithms, P2 was preempted, and P1 was executed instead
      - P2 is allowed to continue executing because its deadline is at 80
    - P2 now has a higher priority than P1, because its next deadline at time 80 is earlier than that of P1 at time 100

- Proportional Share Scheduling
  - Each process is assigned a share of the CPU
    - For instance, if 'T' shares are allocated among all processes in the system, then each application recieves 'N' shares, where `N < T`
      - This ensures each application will recieve `N/T` of the total processor time
  - Example: T = 100 shares are to be divided among three processors: A, B, and C.
    The breakdown is: A = 50 shares, B = 15, C = 20.
    - This scheme ensures that 'A' will have 50% of total processor time, 'B' will have 15%, and 'C' will have 20%
    - What happens if a new process 'D' requests 30 shares?
      - Since 85 shares have already been allocated, and only 15 shares remain, process 'D' will have to wait for a process to terminate, so it can acquire more shares

- Algorithm Evaluation
  - Question: Out of all the available CPU scheduling algorithms for an OS, which one is the best one and how can it be identified?
    - Sometimes Shortest Job First (SJF) is the best, sometimes First Come First Served (FCFS) is the best, and other times Round Robin (RR) is the best
  - Answer:
    - Determine a criteria to evaluate the algorithms
      - Can be done via deterministic modelling:
        - This is a type of analytic evaluation
        - Takes a particular predetermined workload, and defines the performance of each algorithm for that workload
  - Example: Consider 5 processes arriving at time 0:

Process	Burst Time
P1	10

P2	29
P3	3
P4	7
P5	12

    - Evaluation on next slide

- Deterministic Evaluation
  - To determine to best algorithm for the processes, calculate minimum average waiting times
    - This is simple and fast, but it requires exact numbers for input, and the result only applies those inputs
  - Example: Consider 5 processes arriving at time 0:

Process	Burst Time
P1	10
P2	29
P3	3
P4	7
P5	12

    - FCFS is 28ms
      - i.e. (0 + 10 + 39 + 42 + 49 + 61) / 5 = 28ms
    - Non preemptive SJF is 13ms
      - i.e. (0 + 3 + 10 + 20 + 32 + 61) / 5 = 13ms
    - RR is 23ms
      - i.e. (0 + 10 + 20 + 23 + 30 + 40 + 50 + 52 + 61) / 5 = 23ms
    - For this particular example, SJF is the best CPU scheduling algorithm

- Queuing Models
  - Another approach to determine the best CPU scheduling algorithm
    - Is a mathematical description of the system, that may help us determine some of the basic parameters of the system
  - Describes the arrival of processes, and CPU and I/O bursts probabilistically
    - Commonly exponential, and described by mean
    - Computes average throughput, utilization, waiting time, etc.
  - Computer system described as network of services, each with queue of waiting processes
    - Knowing arrival rates and service rates
    - Computes utilization, average queue length, average wait time, etc.
  - When discussing the scheduling problem, or any problem, the problem may be approached/evaluated from 4 different ways:
    - Deterministic Evaluation
    - Mathematical Model

- Simulation
- Build Prototypes

- Little's Formula
  - There are three parameters
    - n = average queue length
    - W = average waiting time in queue
    - 'lambda' = average
  - May give average waiting time if 'n' and 'lambda' are known
  - Little's law: In steady state, processes leaving queue must equal processes arriving, thus: n = 'lambda' * W
    - Valid for any scheduling algorithm and arrival distribution
      - Very important
      - It does not matter if the arrival distribution is exponential, natural, or uniform, the formula is always valid
  - Example: If on average `lambda = 7` processes arrive per second, and normally `n = 14` processes in queue. Then average wait time per process: W = (n / 'lambda') = 2s
    - Answer:
      W = n / 'lambda'
      W = 14 processes in queue / 7 processes per second
      W = (14 / 7) ((processes) * (second / processes))
      W = 2 seconds

- Simulations
  - Are another way to evaluate scheduling algorithms, or any other project/problem
  - Queueing models are limited
    - i.e. Cannot see in real time how things are changing in the system
    - i.e. Cannot stop the system and peek inside the internals to see the state of the system
  - Simulations are more accurate, and allow us to see the state of the system at any point in time and observe the change
    - Programmed model of a computer system
    - Clock is a variable
    - Gather statistics indicating algorithm performance
    - Data to drive simulation gathered via:
      - Random number generator according to probabilities
        - Helps simulate a real environment
      - Distributions defined mathematically or empirically
      - Trace tapes record sequences of real events in real systems
        - Tells us how particular events work, in order of execution
          - i.e. CPU executes, then IO takes over, and then CPU executes once again, etc.

- Evaluation Of CPU Schedulers By Simulation

- A trace tape based on the actualy process execution is created
    - The trace tape is put through different simulation algorithms
        - i.e. FCFS, SJF, and RR scheduling algorithms are tried on the trace tape. The performance statistics for each algorithm are compared, and the best one is selected
- Distribution does not indicate order of event instances
    - To correct this problem, we can use trace files
- Simulations are relatively good and may help us a lot to optimize the system and pick the best scheduling algorithm for a particular environment
    - However, all simulations and modelling approaches have limitations
        - They are not 100% correct
- Simulations are usually ran before a scheduling algorithm is applied, and the best scheduler is selected based on the output
    - Even if the simulation tells us that a scheduler is the best algorithm, it may not be accurate

- Implementation
    - The final approach to testing CPU scheduling algorithms is to implement them into real systems; requires lots of testing
    - Even simulations have limited accuracy
    - Implement new scheduler and test in real systems
        - May have high cost/risk
        - Results depend on environment
        - Provides the most accurate results
    - Most flexible schedulers can be modified per-site or per-system in order to accommodate different environments
    - Researchers combine all 4 approaches: deterministic, queueing models, simulation, and implementation, when testing CPU scheduling algorithms

- End
    - Operating Systems are among the most complex pieces of software ever developed!