Process Management

- OS maintains a data structure for each process called **Process Control Block** (PCB)
- Information associated with each PCB:
  - Process state: e.g. ready, or waiting etc.
  - Program counter: address of next instruction
  - CPU registers: PC, working registers, stack pointer, condition code
  - CPU scheduling information: scheduling order and priority
  - Memory-management information: page table/segment table pointers
  - Accounting information: book keeping info e.g. amt CPU used
  - I/O status information: list of I/O devices allocated to this process

CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **non-preemptive**
- All other scheduling is **preemptive**
Dispatcher module gives control of the CPU to the process selected by the scheduler.

This involves:
- switching context
- switching to user mode
- jumping to the proper location in the user program to restart that program

Dispatch latency
- Time it takes for the dispatcher to stop one process and start another running

Issues to Consider
- Efficiency
  - If scheduler is invoked every 100 ms & it takes 10 ms to decide the order of execution of the processes then 10/(100 + 10) = 9% of the CPU is being used simply for scheduling the work

- Context switch overhead (Dispatcher latency)
  - Saving PCB on process, loading PCB of other

- Fairness
  - Give each process a fair share

- Priority
  - Allow more important processes
Scheduling Algorithm Metrics

- CPU utilization: Fraction of time CPU is utilized
- Throughput: # of processes that completed per time unit
- Turnaround time: amount of time to execute a particular process
  - Waiting in Ready Queue + Executing on CPU + doing I/O
- Response time: amount of time it takes from when a request was submitted until the first response is produced
  - Ideal for interactive systems

Note:
- For simplicity illustration and discussion only one CPU burst per process is used in examples
- Measure of comparison is done with Average Waiting Time
  - Waiting Time is the sum of the periods spent waiting in ready queue
  - Context switch time is negligible

Scheduling Scheme: FCFS

- First Come First Served (FCFS)
  - The process that requests the CPU first is allocated the CPU
  - Implemented using a FIFO queue
    - When the process enters the read queue, its PCB is linked to the tail
    - The process at the head of the queue is given to the CPU
  - FCFS is non-preemptive and hence easy to implement
  - Wait time varies substantially if the processes that comes first are CPU intensive

FCFS Example (Execution Time)

<table>
<thead>
<tr>
<th>Process</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 )</td>
<td>24</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>3</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>3</td>
</tr>
</tbody>
</table>

Suppose that the processes arrive in the order: \( P_1, P_2, P_3 \)

The Gantt/Time Chart for the schedule is:

- Waiting time for \( P_1 = 0; P_2 = 24; P_3 = 27 \)
- Average waiting time: \((0 + 24 + 27)/3 = 17\)

FCFS Example (contd..)

Suppose that the processes arrive in the order: \( P_2, P_3, P_1 \)

The time chart for the schedule is:

- Waiting time for \( P_1 = 6; P_2 = 0; P_3 = 3 \)
- Average waiting time: \((6 + 0 + 3)/3 = 3\)
  - Much better than previous case
- Convoy effect: all processes wait for the one big process to get off the CPU
Scheduling Algorithm: Shortest Job First (SJF)

- The process with the shortest execution time takes priority over others
- Associate with each process the length of its next CPU execution time
  - Achieved by guessing the run time of process for its next execution
- Non-preemptive
  - Once CPU given to the process it cannot be preempted until completes its CPU execution time is complete.
  - If 2 short jobs with same execution time then use their time of arrival to break the tie
- Preemptive
  - If a new process arrives with CPU execution time less than remaining time of current executing process, preempt

SJF with Preemption Example

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Next CPU Ex Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>P₂</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>P₃</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>P₄</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

- SJF (preemptive)

Average waiting time = (9 + 1 + 0 + 2)/4 = 3
Also known as Shortest Remaining Time First

SJF Advantages and Disadvantages

Advantage
- SJF is optimal – gives minimum average waiting time for a given set of processes

Disadvantage
- Need to have a good heuristic to guess the next CPU execution time
- Short duration processes will starve longer ones

Estimating the Next CPU Execution Time

- Use previous value of the execution time
- Predict new time using Exponential Averaging
- Method:
  1. \( t_n \) = actual length of \( n^{th} \) CPU Execution Time
  2. \( t_{n+1} \) = predicted value for the next CPU Execution Time
  3. \( 0 \leq \alpha \leq 1 \)
  4. Formula: \( t_{n+1} = \alpha t_n + (1-\alpha)t_n \)
- Since both \( \alpha \) and \( (1-\alpha) \) are less than or equal to 1, each successive term has less weight than its predecessor
Example of Exponential Averaging

Actual CPU burst time $t_i$

Predicted CPU burst time $\tau_i$ with $\alpha = 1/2$ and $\tau_0 = 10$

Scheduling Scheme: Priority

- A priority number (integer) is associated with each process
- CPU is allocated to the process with the highest priority
  - If processes have same priority then schedule according to FCFS
- SJF is an example of a priority scheduling
- Problem: Starvation – low priority processes may never execute
  - Solution: Aging – as time progresses increase the priority of the process

Scheduling Scheme: Round Robin (RR)

- Each process gets a small unit of CPU time called *time quantum or slice*
  - After this time has elapsed, the process is preempted and added to the end of the ready queue
- 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
  - No process waits more than $(n-1)q$ time units
- Performance
  - If $q$ is large then RR becomes like FCFS
  - $q$ should not be so small such that it requires too many context switches

What do real systems use?

- Multi-level Feedback queues
  - N priority levels
  - Priority scheduling between levels
  - RR within a level
  - Quantum size decreases as priority level increase
  - Process in a given level not scheduled until all higher priority queues are empty
  - If process does not complete in a given quantum at a priority level it is moved to next lower priority level
  - If processes voluntarily relinquishes CPU (becomes I/O bound) then move process to a higher priority queue
### User Threads
- Implemented by a library at the user level
- Supports thread creation, scheduling, and management with no support from the kernel
- Advantage
  - Fast and easy to create (no overhead of system calls)
- Disadvantage
  - If one thread blocks then all block

### Kernel Threads
- OS having its own threads
  - Also known as LWP (Light Weight Processes)
  - A LWP can be viewed as “virtual CPUs” to which the scheduler of threads library schedules user-level threads
- Advantage
  - If one thread blocks then OS can schedule another thread of the application to execute
- Disadvantage
  - Slower to create and manage as creation and management is happening via system calls

### Thread Models: Many to One Model
- All application-level threads map to a single kernel-level scheduled entity
- Threads of a process are not visible to the os/kernel
- Threads are not kernel scheduled entities
  - Process is the only kernel scheduled entity
- Problem with this model is that it constrains concurrency
  - Since there is only one kernel scheduled entity, only one thread per process can execute at a time
  - Cannot benefit from the hardware acceleration on multi-threaded processors or multi-processor computers

### Many to One model
- User thread
  - Kernel thread
Thread Models: One to One Model
- Maps each user thread to schedulable entity in the kernel
- All user threads are visible to the kernel
- All user threads are kernel scheduled entities, and all threads can concurrently execute
- The drawback to this model is that the creation and management of the threads entails creation of corresponding kernel thread

Thread Models: Hybrid Models
- Offers the speed of library threads and the concurrency of kernel threads
- Multiplexes many user-level threads to a smaller or equal number of kernel threads
- There are two levels of thread scheduling:
  - Library manages the scheduling of library threads onto kernel scheduled entities
  - Kernel manages the scheduling of kernel scheduled entities onto processor(s)
Thread Scheduling: Contention Scope

- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
- Known as **process-contention scope (PCS)** since scheduling competition is within the process

- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)**
  - Competition among all threads in system

- In Pthreads, scope can be set with thread attribute object (pthread_attr_t) via the pthread_attr_setscope(..)
  - By default the scope used is PCS
  - .h also provided macros PTHREAD_SCOPE_PROCESS and PTHREAD_SCOPE_SYSTEM

Thread Scheduling: Scheme

- Most user-level thread libraries provide getting/setting scheduling policies

- Example Pthreads API
  - int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
  - int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);
    - policy -> SCHED_FIFO, SCHED_RR and SCED_OTHER