Lecture 04: CPU Scheduling

This lecture focuses on how the operating system manages the resource of the central processing unit (CPU) so that it is available and accessible to competing jobs. The lecture proceeds under the following captions:

- Basic concepts
- Directing I/O Devices
- Ready Queue
- Scheduling Queues
- Performance Considerations
- Scheduling Algorithms
- Analytic Evaluations
- CPU Scheduling on IBM i
- CPU Scheduling on Windows
- CPU Scheduling on Traditional Unix
- CPU Scheduling on Linux
- Summary and Concluding Remarks

4.1 Basic Concepts

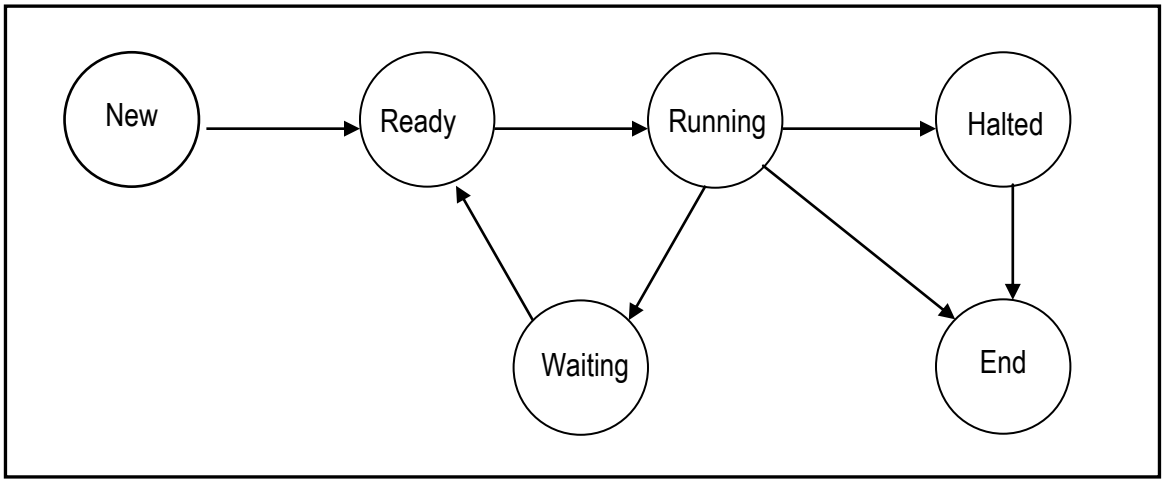
With the demand for multi-user, multitasking operating systems, CPU scheduling becomes more critical. CPU scheduling involves allocating the limited resources of a computer system (CPU time) to multiple users (jobs) simultaneously. These jobs (processes) are competing for CPU time, as the system is in operation.

Primary resources to be shared include:
- I/O devices and peripherals
- CPU itself
- Memory

CPU management has to do with the management of processes. A *process* is a program in execution. This process (also called a *job* or *task*) is created when a user signs on to the system and typically ends when the user signs off (an exception is that batch jobs could exist even after a user signs off). Processes include:
- Batch jobs
- Time-shared users (interactive jobs)
- System activities e.g. spooling
- I/O operations

Other synonyms of process, as used in literature on operating systems are *job, user, program, task,* and *activity*. A process may be in one of the following states: *New, Ready, Running, Waiting or Halted* — as illustrated in figure 4.1.

Figure 4.1: Illustrating Process States



4.1 Basic Concepts (continued)

A process is in a state of *New* when it is just created. It enters a state of *Ready* when it is admitted to the ready queue by the Long Term Scheduler (LTS). It is in a state of *Running* when it has CPU attention. It may move from *Running* to either *Waiting* or *Halted*. It is put in waiting when the CPU leaves it in order to concentrate on some other process — according to the queuing discipline. It is put in a state of *Halted* if the OS has to make a temporary stop due either to program error or system constraints (e.g. interrupts). The process eventually *ends*, i.e. leaves the system.

Each process has associated *process control block* (PCB) containing information such as:
- Process name
- Process state
- Process / Job number
- User responsible for the process
- Program counter (a pointer to the next instruction to be executed)
- Register contents
- Memory management information *(e.g. start & stop address of the process)*
- Accounting information *(e.g. time limits)*
- I/O status
- CPU scheduling information *(e.g. priority of process; scheduling parameters)*

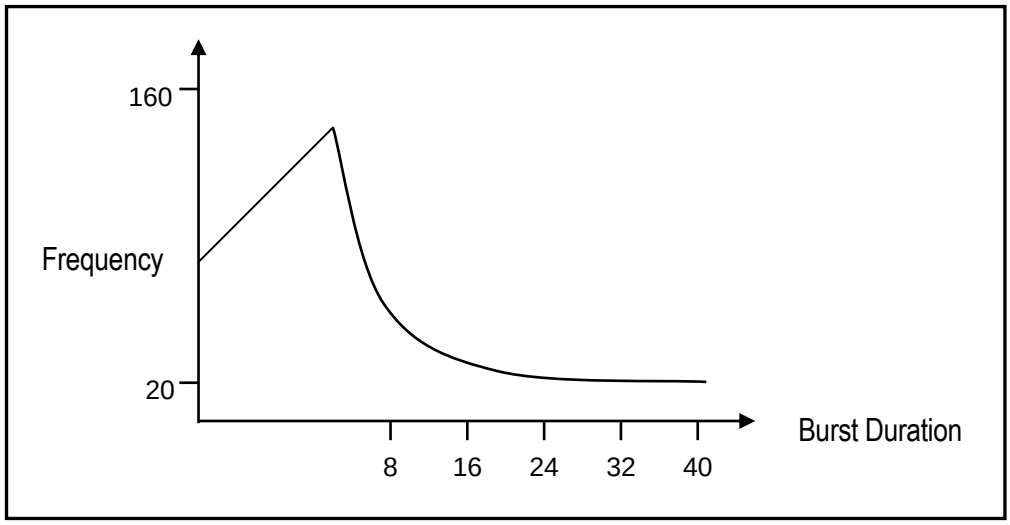
Synonymous terms for the PCB are:
- PSW — Process Status Word
- TCB —Task Control Block
- JCB — Job Control Block
- PAG — Process Access Group

Example: IBM i used the term PAG; PICK used the term PSW

Note:
- The OS ensures that user access of the PCB is non-destructive (often, access is display only).
- Process execution is a series of *CPU bursts* and *I/O waits*. The process usually starts with a CPU burst and ends with one. The I/O bursts alternate with the CPU bursts. CPU burst duration (per process) is typically described by the graph in figure 4.2.

4.1 Basic Concepts (continued)

Figure 4.2: Illustrating Frequency of CPU Burst Times for a Process



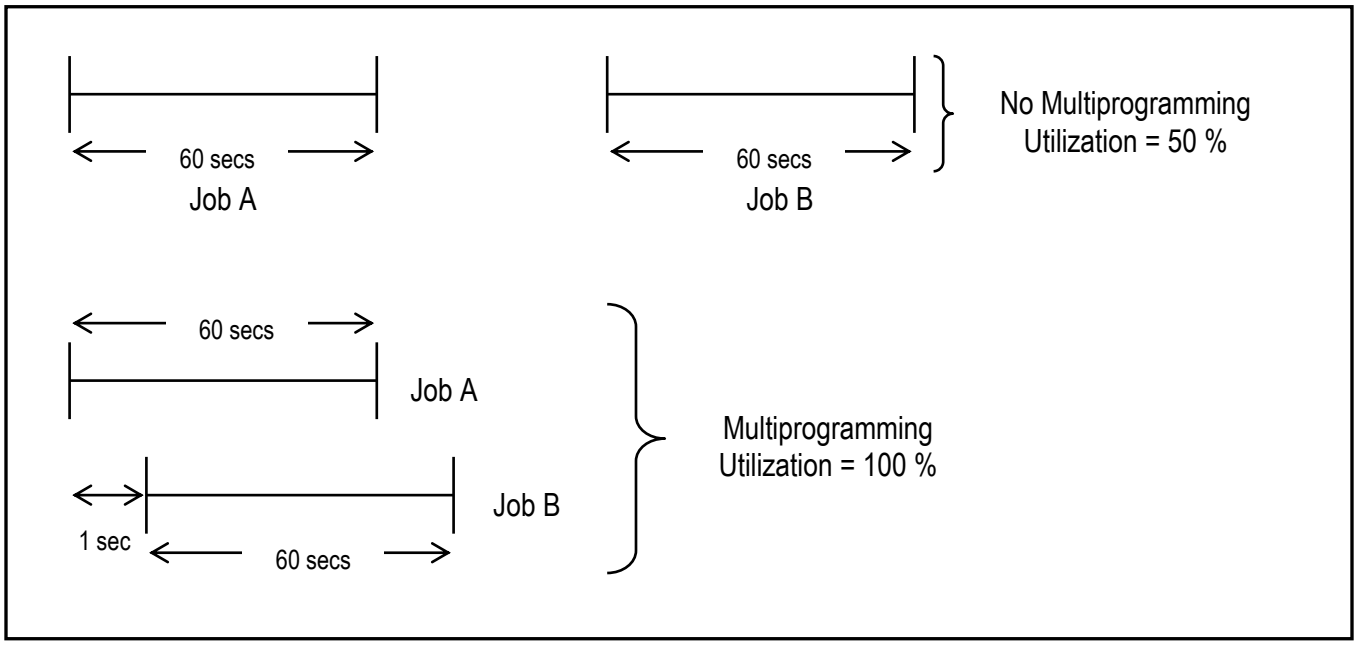
At the start of a process, the contents of the PCB are fed into the CPU; when the process stops, the current state of the CPU registers is fed into the PCB.

To illustrate the benefits of CPU scheduling, consider two jobs A, B where each job executes for one second, then waits for one second for a duration of 60 seconds.

In the absence of CPU scheduling (no multi-programming or multitasking), the jobs A, B must be serviced consecutively, and therefore take a total of 120 seconds.

With CPU scheduling (multi-programming), jobs A, B may be serviced simultaneously and take a total of 61 seconds.

Figure 4.3: Illustrating the Benefit of Multiprogramming in Enhancing CPU Utilization



4.2 Directing I/O Devices

The OS directs the use of the I/O devices according to the interconnection structure (architecture) of the computer system. The interconnection structure may be any of the following:

- I/O – CPU
- I/O – Central Switch
- I/O – Bus
- I/O – Memory

Transfer of data to/from the CPU is effected by interrupts. When a process is started its PCB is loaded into the CPU; when the process is halted, the PCB is updated from the CPU registers.

Outputs from the CPU are typically off-loaded to device queues, which are attached to the output devices. These queues are subject to the scheduling strategy chosen for the OS (more on this later).

Processes waiting for I/O devices are allowed to wait in a device queue. There may be more than one device queues per system.

Example:
On PICK, there are several output queues; an output queue may be attached to a printer or several printers.

On IBM i as well as UNIX, there are several output queues and device queues (not necessarily connected). A device queue may be connected to an output queue and thereby enable printing.

On Windows, each output device has its associated output queue.

4.3 Ready Queue

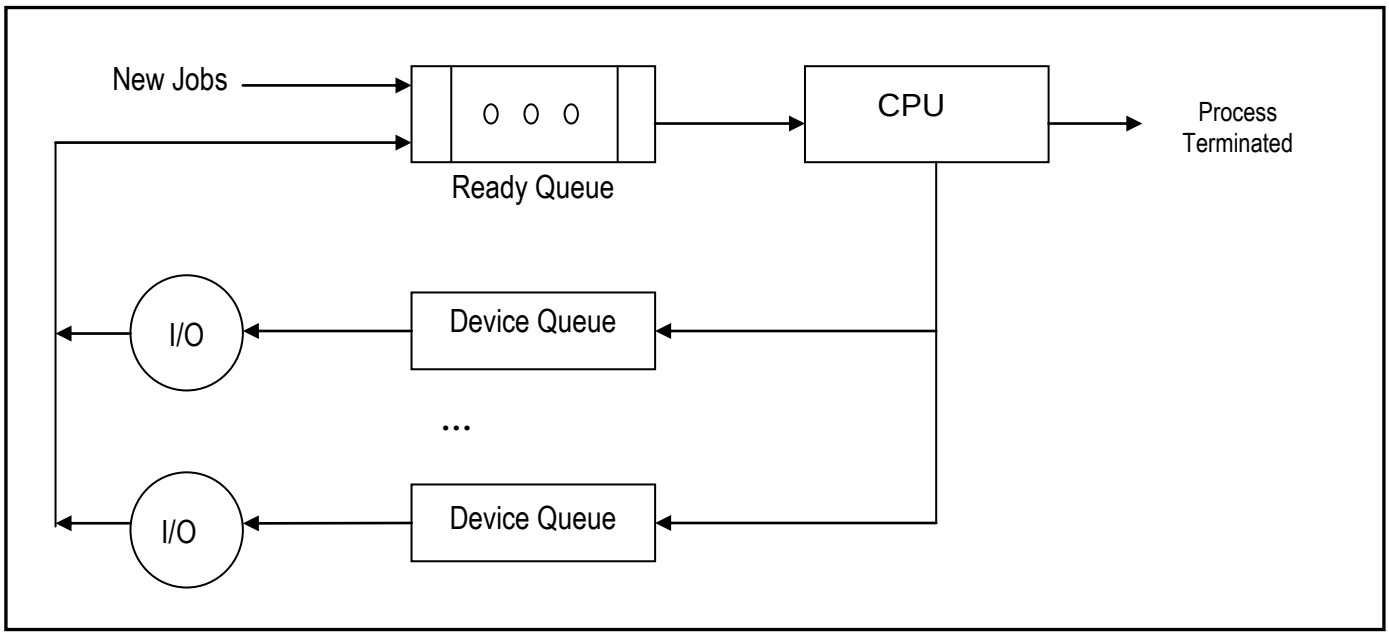
Processes ready to be executed are held in a *ready queue*. There may be several ready queues per system.

The ready queue entries are typically PCBs or pointers to PCBs. The queue is examined from time to time, according to some algorithm, and processes are selected for execution.

The queue may be implemented as a FIFO queue, a priority queue, a tree, a stack or an unsorted linked list.

4.3 Ready Queue (continued)

Figure 4.4: Illustrating the Relationship between the CPU and Ready /Device Queue



4.4 Scheduling Queues

Jobs entering and leaving a queue must do so according to some scheduling rule. Schedulers are OS programs that manage queues based on established and predetermined queuing disciplines. Three schedulers are worth mentioning:

- The *Long Term Scheduler*
- The *Short Term Scheduler*
- The *Medium Term Scheduler*

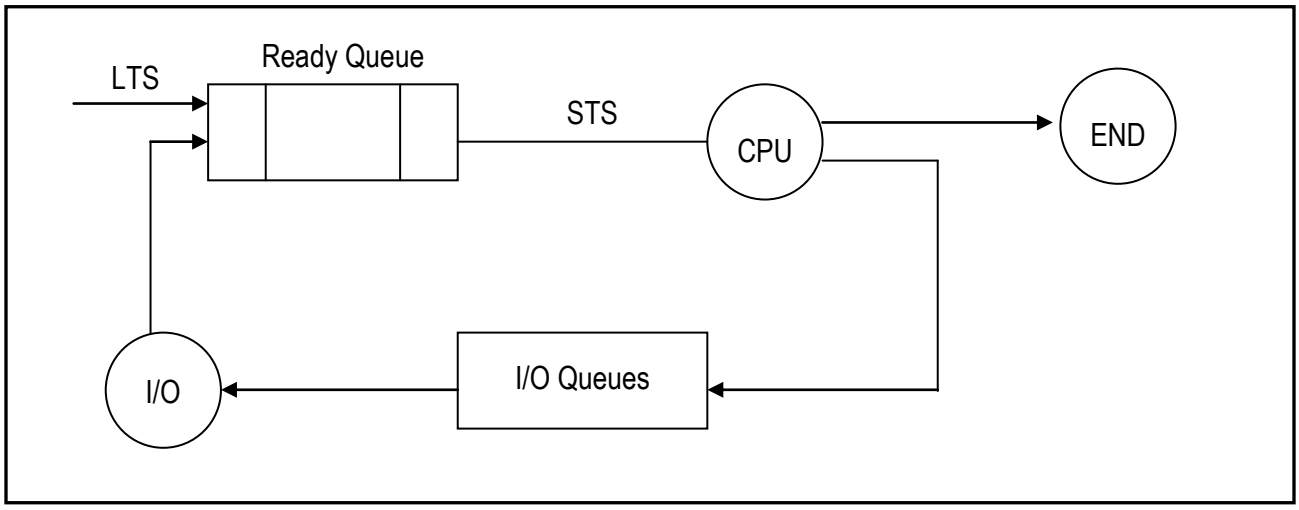
The long-term scheduler (LTS) looks at incoming jobs and decides whether to take them or let them wait. The jobs are selected from a job pool and loaded into memory (ready queue) for execution. The LTS may also need to manage the mix of jobs running -I/O bound vs. CPU bound jobs. The mix of jobs affects CPU performance. The short-term scheduler (STS) examines the ready queue and selects jobs for CPU execution.

Differences between LTS and STS:

- The STS has a high frequency of job selection relative to the LTS.
- The LTS acts on a job once; STS may act on a job more than once.
- The LTS controls the degree of multiprogramming via the number of jobs concurrently running.
- The STS ensures that multiprogramming takes place, by servicing all jobs admitted to the ready queue.

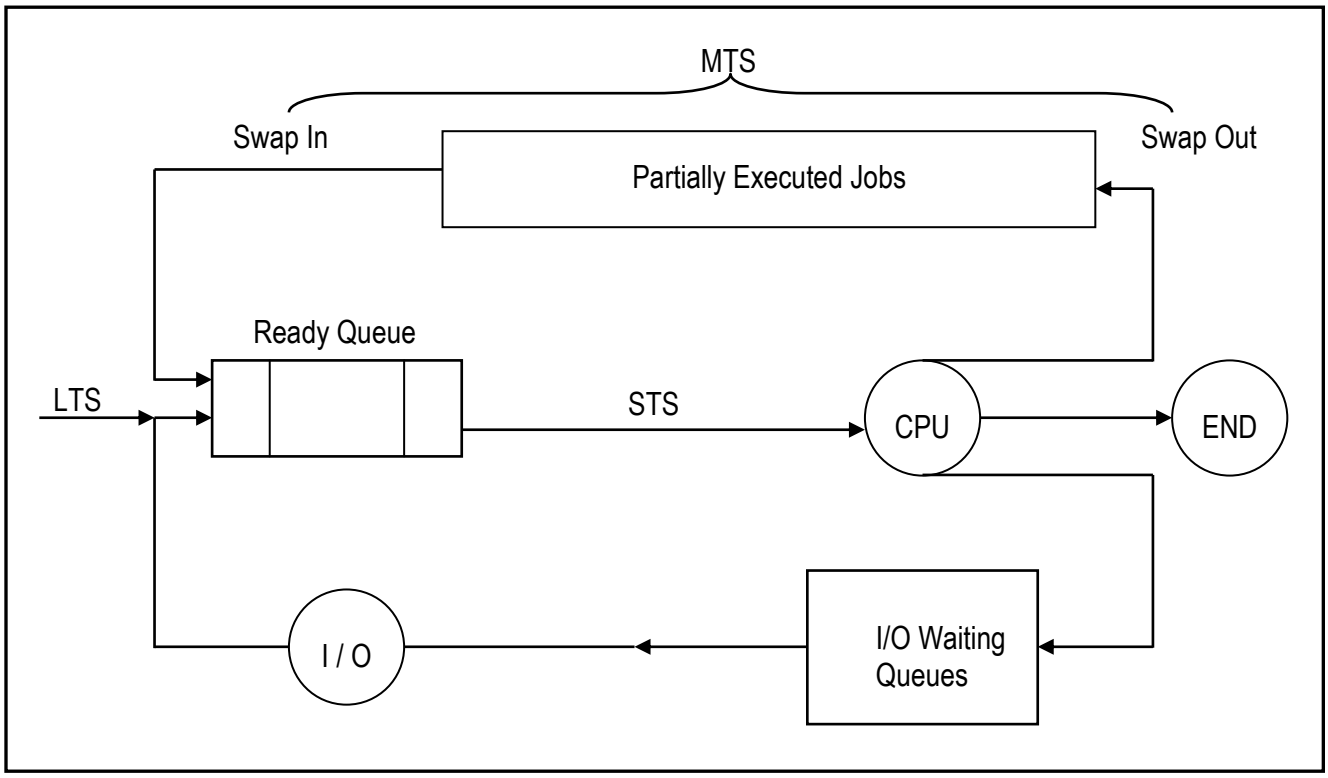
4.4 Scheduling Queues (continued)

Figure 4.5: Illustrating the Role of the LTS and the STS



The medium term schedule (MTS) allows second thought to be given to admissions of the LTS. Jobs are swapped in and out, in order to control the mix of CPU bound vs. I/O bound jobs. The process is called r*olling*. Figure 4.6 illustrates.

Figure 4.6: Illustrating the Role of the MTS



4.4 Scheduling Queues (continued)

The *dispatcher* is the program called by the STS to effect the servicing of a job from the ready queue. Activities of the dispatcher include:

- loading the relevant registers;
- setting the program counter;
- switching to user mode;
- advancing (jumping) to next instruction.

4.5 Performance Considerations

When an operating system's performance is assessed, there are certain performance criteria which are typically used. These include:

- Wait time — the time spent waiting in the ready queue
- Turnaround time — the total of the wait time, execution time on CPU, time spent waiting on memory access, time spent doing I/O
- Response time — the time taken to start responding to the process
- Throughput — the amount of work (i.e. processes) completed per unit time
- CPU utilization — level of occupation of the CPU (expressed as a percentage)

Wait time refers to the time a job waits for CPU service while in the ready queue or output queue. Wait time is further classified as:

- Average wait time
- Best wait time
- Worst wait time

Turnaround time is the time between submission of a job and the time of the last result of that job. It is the summation of:

- Wait time on LTS
- Wait time on CPU
- Wait time on STS
- Wait time on I/O
- Wait time due to overheads

Response time is the time between the user pressing enter and obtaining a response. *Throughput* is the amount of jobs that the system can completely handle in a given time.

CPU utilization is a measure of how taxed the operating system is. It is typically expressed as a percentage. Recommended utilization range is 75% – 85%. Utilization greater than 93% is considered dangerous, while a utilization less than 65% is considered low.

4.6 Scheduling Algorithms

The traditional approaches to queue scheduling are:
- First-In-First-Out (FIFO)
- Shortest-Job-First (SJF)
- Priority
- Preemptive strategies
- Round-Robin
- Multi-Level
- Preemptive multitasking

Variations of these fundamental approaches are common.

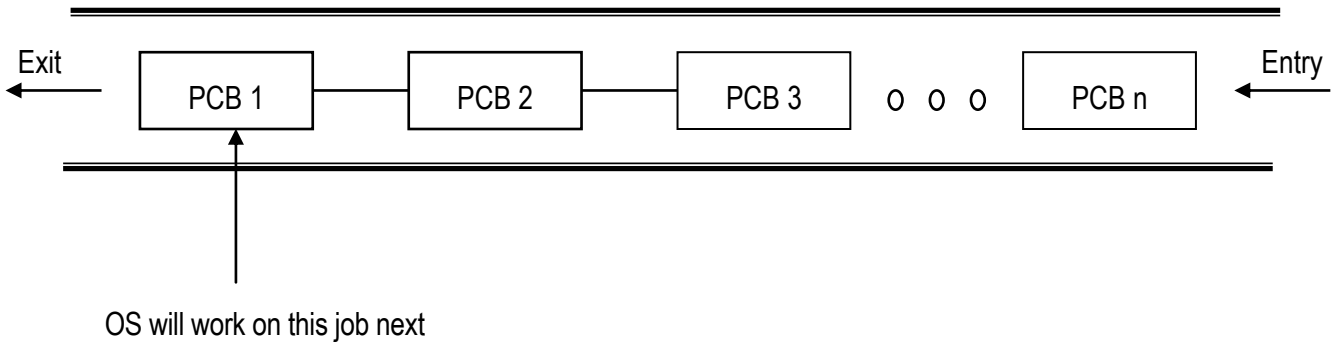
Examples:

1. The IBM i implements a multi-level queuing system with the default strategy for a queue being FIFO within priority. However, the user has the flexibility of selecting other strategies.
2. DOS used FIFO strategy.
3. Windows uses preemptive multitasking.
4. For traditional UNIX, the strategy is a multi-level feedback queuing system, with round robin within each priority queue.
5. Linux uses a preemptive strategy for regular jobs and a priority based strategy for real time jobs.

4.6.1 First-In-First-Out

First-in-first-out scheduling (FIFO) is the simplest scheduling algorithm used in CPU scheduling. It may be employed by the LTS and STS. CPU time is allocated to processes based on the order in which the requests are made. The PCB's are linked in the ready queue based on their arrival there.

Figure 4.7: Illustrating FIFO Scheduling



4.6.1 First-In-First-Out (continued)

The performance of FIFO is poor. Consider three jobs submitted consecutively in order 1, 2, 3, with (assumed known) CPU bursts as shown in the following example:

Job	CPU Burst Time	Turnaround	Avg. Wait
1	24	24	0
2	5	29	24
3	6	35	29

The average turnaround time = 29.3 and is not minimal since the order of the jobs affects the performance.
Average Wait = (0 + 24 + 29) / 3 = 17.7
Average Turnaround = (24 + 29 + 35)/3 = 29.3

FIFO scheduling also suffers from the *convoy effect*. Consider one CPU-bound job & several I/O-bound jobs in the ready queue. When the CPU-bound job is in the CPU, all the I/O-bound jobs are stacked up and waiting in the ready queue. Similarly, I/O-bound jobs will clog up the device (output) queues while the CPU is idle. The result is low CPU utilization. A second problem with FIFO is that performance is influenced by the order in which the jobs are processed. This will become clear in the next subsection.

4.6.2 Shortest-Job-First

In *shortest-job-first (SJF) scheduling*, to each job is associated the length of its next CPU burst (based on previous CPU bursts). When the CPU is ready, it is allocated to the job with the shortest (projected) next CPU burst. If the jobs have equal projected CPU bursts, FIFO is used.

If the system insists that users supply estimates of the length of time of jobs, SJS may be used by the LTS; otherwise, it applies to the STS.

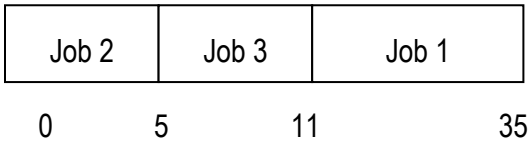
The performance of SJF is better than FIFO; SJF is said to be optimal i.e. it gives minimum average wait time for a given set of jobs (proof beyond the scope of this course).

Consider jobs 1, 2, 3 in the ready queue at an instance in time:

Job:	1	2	3
Next CPU Burst:	24	5	6

4.6.2 Shortest-Job-First (continued)

The jobs would be processed as follows:



The average turnaround time = (5 + 11 + 35) / 3 = 17
Average wait time = (0 + 5 + 11) / 3 = 5.3

Two problems associated with SJF scheduling are:
- Estimating CPU burst time
- Starvation

How do we predict the next CPU burst? To do this, we need a forecasting algorithm. One popular such algorithm is exponential forecasting. With this forecasting model, the next CPU burst is estimated based the previous predicted burst and actual burst: The formula is stated and clarified in figure 4.8.

Figure 4.8: Exponential Smoothing Forecasting

$T_{n+1} = \alpha t_n + (1 - \alpha) T_n$

Where $0 < \alpha < 1$ (α is called the smoothing parameter)
$T_{n+1} \equiv$ the predicted CPU burst duration
$T_n \equiv$ the last predicted CPU burst
$t_n \equiv$ the last actual CPU burst

The smoothing parameter (α) is typically 0.5 so that recent and past history are equally weighted. T_o can be defined as a constant or overall system average.

The second problem, *starvation*, occurs when shorter jobs consistently obtain precedence over larger job(s), thus starving the larger job(s).

SJF performs much more desirably than FIFO, as illustrated in the comparison chart (figure 4.9) for the same three jobs that were previously analyzed with FIFO. SJF may be used on its own or in conjunction with Round Robin or a Multilevel Queue or a preemptive strategy (these will be discussed shortly).

Figure 4.9: Comparison of FIFO with SJF

Performance Criteria	FIFO	SJF
Average Wait Time	17.7	5.3
Average Turnaround Time	29.3	17

4.6.3 Priority

In *priority scheduling*, a priority is assigned to each job. A job gains access to the CPU based on its priority.

Priority is typically graded [1 .. n] or [0 .. n] where **n** is a positive integer; some systems use the largest number as the highest priority (e.g. Windows), others use the lowest number as the highest priority (e.g. Unix).

In more sophisticated systems where they may be more than one job queues (and output queues), priority may also be assigned to the job queues.

Example:

The default scheduling algorithm on IBM i is *FIFO within priority*

- *User classes* are assigned priority codes.
- *User profiles and job descriptions* have user class associated with them.
- When a batch job is submitted, its priority is traced to its job description; the priority of interactive jobs are traced to their respective user profiles. That priority determines its relative position among other jobs in a given queue.
- Multiple jobs submitted by a particular user are processed FIFO.
- Queues within a subsystem are serviced based on their priority codes. Within a given queue, it is FIFO.

Modern versions of the operating system (System i) provide a more sophisticated choice or scheduling algorithms.

Priority algorithms suffer from starvation (SJF is a case of priority). *Ageing* is a strategy employed to address the problem of starvation: the priority of a long waiting job is upgraded by the system as the job's wait time increases.

4.6.4 Preemptive Algorithms

A *preemptive algorithm* allows jobs to start with no guarantee of finishing. FIFO cannot be preemptive but SJF and priority may be.

In preemptive SJF, a job may be preempted if another job arises with next CPU burst time being less. This algorithm is sometimes referred to as Shortest-Remaining-Time-First. Similarly, there may be preemptive priority systems.

By way of experiments, it has been observed that the performance of PSJF is better than that of SJF.

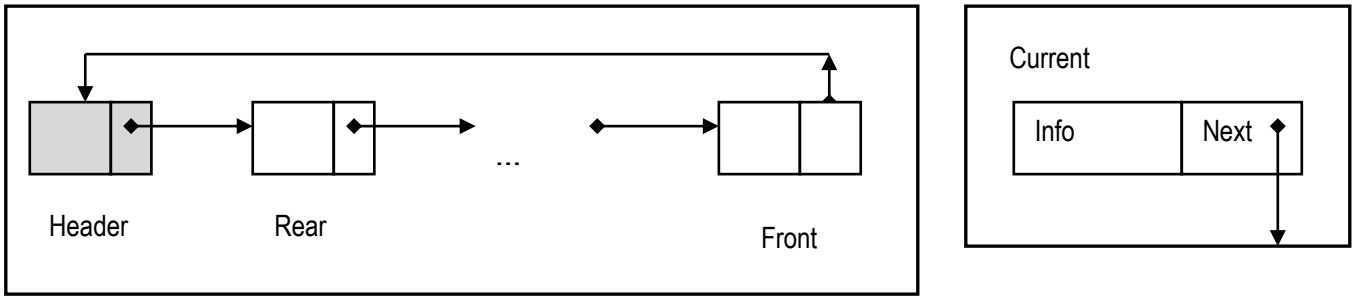
4.6.5 Round Robin

In *round robin scheduling*, a small unit of CPU time (called a *time quantum* or *time slice*) is allocated to each job in the ready queue. The ready queue is therefore treated as a circular queue.

The scheduler goes around the queue, allocating CPU time to each job. However, a time slice may be shortened by an interrupt.

The ready queue is kept as a FIFO queue of PCBs. The scheduler services the jobs one after the other. If a job completes before the end of its CPU burst, it releases the CPU for the next job in line. This gives the impression of a circular queue (figure 4.10).

Figure 4.10: Graphical Illustration of a Circular Queue



The performance of round robin depends on the time slice. If it is too large, RR reduces to FIFO; if it is sufficiently small, (relative to the average job duration), the illusion is given that each job has a dedicated processor.

The algorithm suffers from one setback: a *context switch* occurs every time the CPU changes from one job to another, thus forcing an interrupt. This is an expensive overhead.

4.6.6 Multi-Level Queues

In the *multi-level queue* (MLQ) approach, there are several ready queues and output queues. Typically, the different queues have specific purposes. The queues may have priority codes assigned to them.

Example:

On IBM i, there are three job pools — QINTER, QBATCH, QSPOOL — for interactive jobs, batch jobs and spool (system) jobs respectively. Each pool has a job queue of the same name (as the pool).

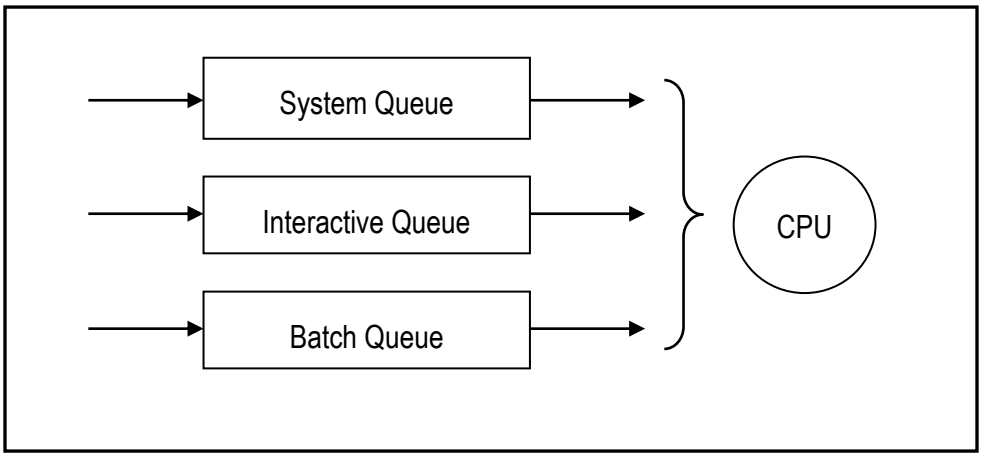
The queue QSPOOL has higher priority than QINTER which has higher priority than QBATCH

Batch jobs may be scheduled as FIFO or SJF; interactive jobs may be Round Robin or priority or SJF or some preemptive strategy.

4.6.6 Multi-Level Queues (continued)

Figure 4.11 illustrates a multi-level queuing system. There must be scheduling among the queues. This is commonly a preemptive priority scheduling (as on the IBM i). Another possibility is to time slice among the queues in a round robin fashion.

Figure 4.11: Illustrating Multi-Level Queues



One obvious drawback with multi-level queues is that programmatically, they are more difficult to implement.

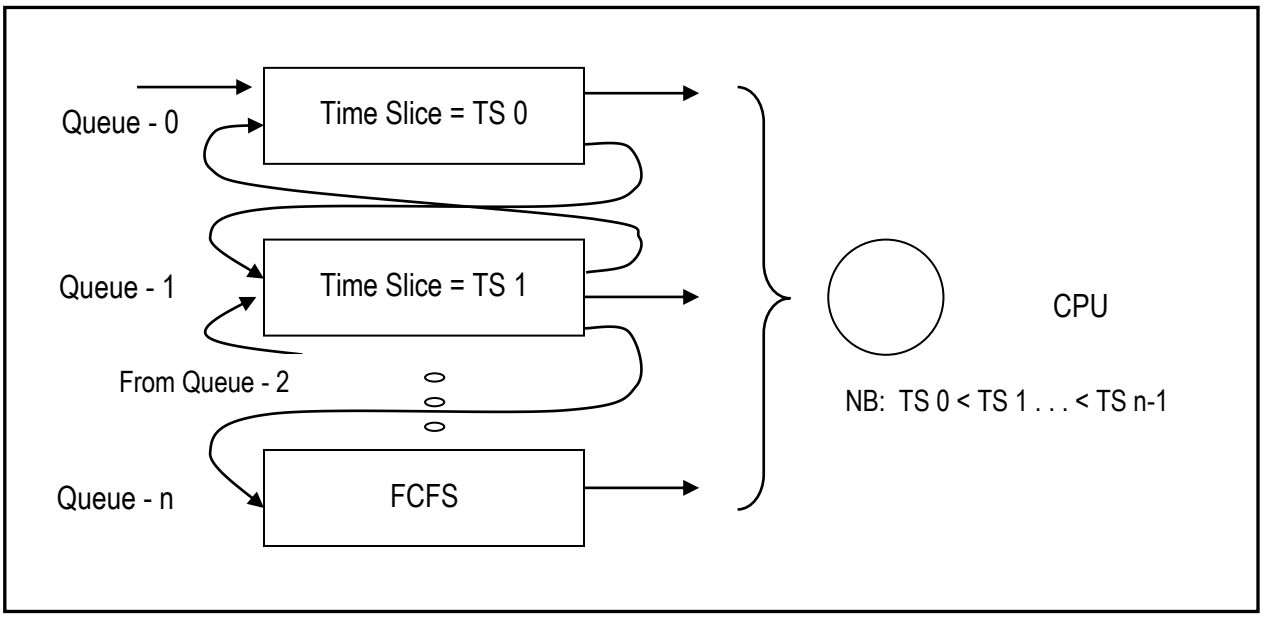
4.6.7 Multi-Level Feedback Queues

In a system that implements *multi-level feedback queues* (MLFQs), jobs are allowed to move from one queue to another, based on their CPU utilization. This is the prevalent strategy in the Unix system.

The idea is to separate jobs based on their CPU burst characteristics: a job that uses too much CPU time will move to a queue with a relatively lower priority.

Figure 4.12 illustrates. Jobs are first placed in queue 0. The time slice for queue-0 jobs is set at TS0. If after time TS0, a job in queue 0 is not finished, it is preempted and moved to the tail of queue-1. The cycle continues.

Figure 4.12: Illustrating Multi-Level Feedback Queues



Again, there must be a strategy to service all the queues. This may be via time slice, or a preemptive priority.
The scheduler must concern itself with the following:

- The number of queues
- Scheduling algorithm for each queue
- Method of upgrading a job to a queue of higher priority.

The main advantages of this approach are minimization of starvation as well as the convoy effect.

The main drawback associated with this approach is that it is very complex to program.

4.7 Analytic Evaluation

Analysis of the efficiency of different scheduling algorithms may be done via a deterministic model (if parameters are known), or queuing models or simulation.

Deterministic analysis is unreal and confined only to the given set of values analyzed. For example, we could set up an experiment to compare SJF, FCFS, and Round Robin on a set of jobs with projected CPU bursts and a time slice (for round robin) established.

Queuing models are mathematical models that allow queues to be analyzed (and compared). *Little's formula* is commonly used here:

N = R * W

N is the average queue length; W is the average wait time;
R is the average arrival rate for new processes.

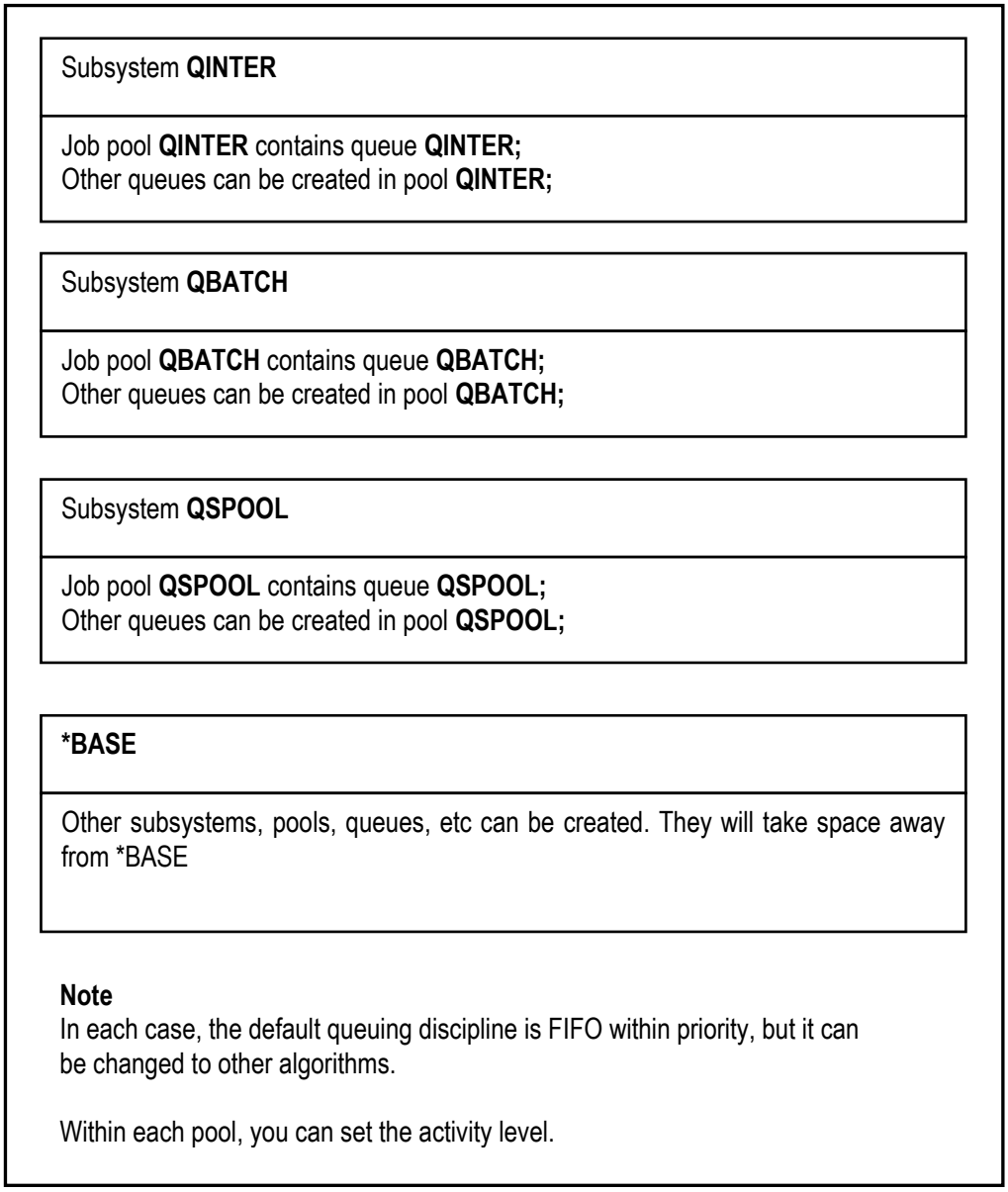
Simulation is the most accurate method of analysis and comparison.

4.8 CPU Scheduling on IBM i

As a special case study of an operating system, IBM i will be discussed in a subsequent lecture (lecture 13). Figure 4.13 provides an overview of CPU scheduling on the IBM i. The information on CPU scheduling is summarized here for clarity:

- The PCB is implemented as the Process Access Group (PAG).
- The system uses a multi-level queue strategy, however
- Users can create additional queues, thus making it more flexible than the approach discussed in section 4.6.6.
- Also, different queues have different purposes (example interactive, batch, spool).
- Actually, job queues are associated with (and therefore operate in) job pools which belong to subsystems. By default, the system is partitioned into major subsystems — QINTER, QBATCH, QSPOOL, QSYS. These subsystems have job pools and job queues of the same respective names. Additionally, the user can create additional subsystems, pools and queues.
- The default queue discipline is *FIFO* within priority, but users may choose other queue disciplines.
- Additionally, each job pool has an activity level (which can be set or modified by the user), which determines the level of multiprogramming allowed. A job queue operates within a job pool (but several job queues may be attached to a job pool).
- Output queues are usually operated with an activity level of 1.

Figure 4.13: CPU Scheduling on the IBM i



4.9 CPU Scheduling on Windows NT

Windows NT [the base operating system for many versions of the Windows platform] employs preemptive multitasking — a kind of preemptive scheduling on top of round robin. The operating system reserves the right to start and inhibit processes in the interest of smooth operation.

Multithreading is also supported by the operating system — processes can be split up into threads that run concurrently on different processors. To take advantage of this feature, the application must also support multithreading, e.g. MS Word instant spellchecker.

The Windows Scheduler ensures that the highest priority threads always run. A thread selected to run by the dispatcher will run until one of the following occurs:

- It is preempted by a higher priority thread
- It terminates
- Its time quantum ends
- It makes a blocking system call, such as I/O request

The dispatcher uses a 32-level priority scheme to determine the order of thread execution. Priorities are divided into the following categories:
0 : Memory management
1 – 15: Variable Class
16 – 31: Real-time Class

The dispatcher uses a queue for each scheduling priority; it traverses these levels from highest to lowest until it finds a thread that is ready to execute. If no thread is found, it executes a special *idle thread*.

Additionally, the following priority classes and their respective base priorities apply:

- Real-Time-Priority-Class (24)
- High-Priority-Class (13)
- Above-Normal-Priority-Class (10)
- Normal-Priority-Class (8)
- Below-Normal-Priority-Class (6)
- Idle-Priority-Class (4)

} Variable Class priorities

Within each priority class are relative possibilities as follows:

- Time-Critical
- Highest
- Above-Normal
- Normal
- Below-Normal
- Lowest
- Idle

Figure 4.14 shows how these apply to the priority classes.

Figure 4.14: Windows Priorities

	Real-Time	High	Above Normal	Normal	Below Normal	Idle Priority
Time-Critical	31	15	15	15	15	15
Highest	26	15	12	10	8	6
Above Normal	25	14	11	9	7	5
Normal	24	13	10	8	6	4
Below Normal	23	12	9	7	5	3
Lowest	22	11	8	6	4	2
Idle	16	1	1	1	1	1

At creation, a thread takes the base priority of its parent class. If a thread's time quantum runs out and it is not finished, its priority is lowered. Additionally, the operating system dynamically gives higher priority to *foreground processes* (currently selected) as opposed to *background processes* (not currently selected).

4.10 CPU Scheduling on Traditional UNIX

Traditional UNIX scheduling employed a multilevel feedback queues with round robin within each of the priority queues.

The system employed a 1-second preemption: if a running process did not block or complete within 1 second, it was pre-empted. Priority was based on process type and execution history. Figure 4.15 provides the formulas that were used.

Figure 4.15: Unix CPU Scheduling Formulas

$$P[j][i] = Base[j] + \{CPU[j][i-1]\}/2 + Nice[j]$$

$$CPU[j][i] = \{CPU[j][i-1]\}/2 + \{U[j][i]\}/2$$

Where
U[j][i] is the measure of processor utilization by process j through interval i.
CPU[j][i] is the exponentially weighted average processor utilization by process j through interval i.
P[j][i] is the priority of process j at beginning of interval i (lower values represent higher priorities.
Base[j] is the base priority of process j.
Nice[j] is the user-controlled adjustment factor.

4.10 CPU Scheduling on Traditional UNIX (continued)

The priority of each process was recomputed once per second, at which point, a new scheduling decision was made. The base priority and nice factor ensured that processes remained within a priority band. Established priority bands (in decreasing order of priority) included:

- Swapper
- Block I/O device control
- File manipulation
- Character I/O device control
- User processes

4.11 CPU Scheduling on Linux

Early versions of Linux were similar to the Unix MLFQ model. Two process scheduling algorithms were implemented:

- A time-sharing preemptive strategy for typical user jobs
- A priority based strategy for real-time jobs

The operating system allowed only processes in user mode to be preempted. Processes running in kernel mode were not preempted, even if a real-time process with a higher priority was available.

Each executing process had an associated scheduling class, based on the type of scheduling that was employed. For time-sharing processes, the operating system used a prioritized, credit-based algorithm:

- Each process had a certain number of scheduling credits.
- When a new task was to be chosen for CPU attention, the process with the most credit was selected.
- Every time a process experienced a timer interrupt, it lost 1 credit.
- When a process's credit reached zero, it was suspended and another process selected.
- If none of the ready processes had credits, the operating system performed a re-crediting exercise according to the following rule:

```
Credits[j] = LastCredits[j]/2 + Priority[j]

Where
   LastCredits[j]  =  The amount of credits held by the process j at the last re-crediting
     exercise.
```

Note:

1. Processes that were CPU bound tended to lose credits rapidly.
2. Processes that spent much of their time suspended (e.g. I/O bound jobs) were able to accumulate credits over multiple re-crediting and thereby end up with a higher credit count after a re-credit.
3. This strategy therefore gave high priority to interactive or I/O bound processes for which a rapid response time was important.
4. Background batch jobs were given lower priority than interactive jobs and therefore had lower credit counts.

4.11 CPU Scheduling on Linux (continued)

For real-time problems, Linux implemented two real-time scheduling classes: FIFO and Round-Robin (RR). In either case, each process had a priority in addition to its scheduling class.

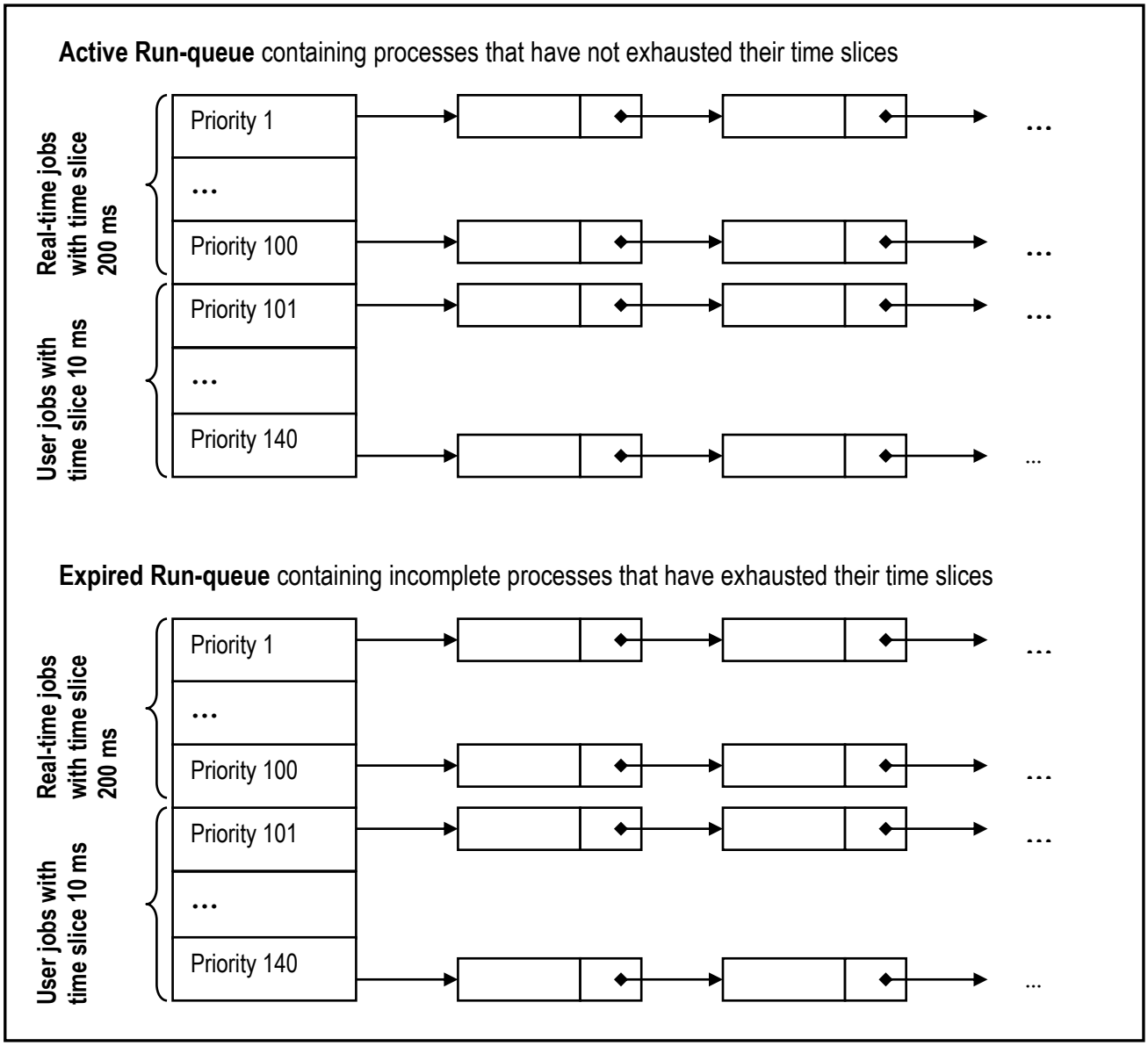
In recent years, Linux designers have sought to overhaul the CPU scheduling of the operating system. One approach that has been tried is a preemptive priority-based algorithm, contingent on the round-robin strategy. Following are some of the salient features of the algorithm (see [Silberschatz, et. al., 2009]):

- Two priority ranges supported are the **real-time** range of {0 .. 99} and a **nice** range of {100 .. 140}, where the lower number indicates a higher priority.
- The algorithm supports symmetric multiprocessing (SMP).
- Higher priority jobs are assigned a larger time slice than lower priority jobs, the range of time slice values being {10 milliseconds .. 200 milliseconds}.
- Real-time processes are assigned static priorities, while other jobs are assigned dynamic priorities based on their **nice** values and an adjustment factor (of ±5).
- The kernel keeps track of all processes to be executed in a *run-queue* (similar to a ready queue). In a multi-processor environment, each processor has its own run-queue.
- Each run-queue has up to 140 priority lists (one per priority level). Jobs in each priority list are processed in FIFO order.
- Each run-queue is implemented as an **active run-queue** containing all processes that have not exhausted their allocated time slices, and an **expired run-queue** containing processes that have exhausted their time slices.
- At each iteration, the scheduler selects the **active-run-queue** process for CPU attention based on
- their assigned priorities (highest priority first). Processes that complete are removed from the system. After each iteration, incomplete processes that have exhausted their allotted time slices are identified. For each such process, its priority is re-calculated (if it is not a real-time job), and it is moved to the **expired run-queue**.
- When the **active run-queue** becomes empty, it is switched with the **expired run-queue**, and the scheduling starts over.

Figure 4.16 provides a basic overview of the scheduling mechanism. Note that by maintaining a priority list at each priority level, and by periodically recalculating the priority for incomplete processes that have exhausted their time slices, the operating system indirectly implements a multi-level feedback queue.

The effect is to keep similar jobs together at each priority level. This results in improved performance. The scheduler is said to be an O(1), meaning that the scheduling time is fixed and deterministic, irrespective of the number of active jobs. Moreover, to further improve performance, Linux implements its memory references via *red-black trees*; a red-black is a special height-balanced binary search tree (see Bovet and Cesati, 2003]).

Figure 4.16: Linux CPU Scheduling Mechanism



4.12 Summary and Concluding Remarks

Here is a summary of what has been covered in this lecture:

- CPU scheduling involves allocating the limited resources of the CPU to multiple users (jobs) simultaneously.
- A process is created when a user logs on. Other synonymous terms include *job* and *task*. The process may go through any of the following states: *New, Ready, Running, Waiting or Halted.*
- Each process has a process control block (PCB), which stores essential information about the process.
- When processes enter the system, they are admitted to the ready queue by the long term scheduler (LTS). They gain CPU access through the short term scheduler (STS). Sometimes a medium term scheduler works on maintaining a healthy balance of CPU-bound jobs and I/O bound jobs.
- CPU scheduling algorithms are often evaluated in terms of criteria such as wait time, turnaround time, response time, throughout, CPU utilization.
- Among the common CPU scheduling algorithms are FIFO (first-in-first-out), SJF (shortest-job-first), priority, preemptive strategy, (RR) round-robin , multi-level queue, and multi-level feedback queue.
- FIFO scheduling involves processing the job in order of arrival sequence.
- In SJF scheduling, the jobs are processed in ascending order of anticipated CPU burst times.
- In priority scheduling, the jobs are processed in descending order of assigned priorities. SJF is a special kind of priority scheduling that is based on the anticipated CPU burst time.
- In a preemptive scheduling algorithm, the operating system reserves the right to preempt a process prior to its completion, based on some other overriding criteria.
- Round-robin scheduling algorithm involves allowing each job to obtain time slices of CPU time (typically in FIFO order, or FIFO within priority order), until the jobs eventually complete.
- In a multi-level queuing (MLQ) system, the operation system places jobs in different queues based on predetermined characteristics of the jobs. Each queue contains jobs that are similar.
- A multi-level feedback queuing (MLFQ) system is an MLQ system in which the operating system reserves the right to move jobs among the various queues based on certain dynamic characteristics.

Contemporary operating systems tend to implement various combinations of the common scheduling algorithm.

- IBM i implements an MLQ system that provides the user with the flexibility of participating in the categorization of jobs, management of the level of multiprogramming, and choosing what scheduling algorithm they want to use.
- Windows implements a preemptive multitasking algorithm that is based on a matrix of priority bands. Jobs are classified as real-time or interactive, and their priorities are assigned based on certain criteria as outlined in the priority matrix.
- Unix implements an MLFQ based on priority. The operating system iteratively uses a special formula to calculate priority to be assigned to each job. Based on the determined priority, the jobs are placed in specific queues that are serviced by the operating system.
- Early versions of Linux implemented a kind of MLFQ based on a special crediting system. This has been recently upgraded to a more elaborate and sophisticated MLFQ system based on priority in collaboration with round-robin.

Intense and fascinating as this may appear, the CPU is not the only resource that the operating system must manage. The next two lectures discuss how the operating system manages memory.

4.13 Recommended Readings

[Aas 2005] Aas, Josh. 2005. *Understanding the Linux 2.6.8.1 CPU Scheduler*. N.p.: Silicon Graphics, Inc. http://www.angelfire.com/folk/citeseer/linux_scheduler.pdf (accessed February 2010).

[Bacon & Harris 2003] Bacon, Jean S. and Tim Harris. 2003. *Operating Systems: Concurrent and Distributed Software Design*. Boston, MA: Addison-Wesley. See chapter 4.

[Bovet & Cesati 2003] Bovet, Daniel and Marco Cesati. 2003. *Understanding the Linux Kernel* 2nd ed. Sebastopol, CA: O'Reilly & Associates.

[Flynn & McHoes 1997] Flynn, Ida M. and Ann McIver McHoes. 1997. *Understanding Operating Systems* 2nd ed. Boston, MA: PWS Publishing. See chapter 4.

[Nutt 2004] Nutt, Gary. 2004. *Operating Systems* 3rd ed. Boston, MA: Addison-Wesley. See chapters 6 & 7.

[Singhal & Shivaratri 1994] Singhal, Mukesh, and Niranjan G. Shivaratri. 1994. *Advanced Concepts in Operating Systems*. New York: McGraw-Hill Inc. See chapter 2.

[Silberschatz 2012] Silberschatz, Abraham, Peter B. Galvin, & Greg Gagne. 2012. *Operating Systems Concepts*, 9th Ed. Update. New York: John Wiley & Sons. See chapters 3 & 5.

[Stallings 2005] Stallings, William. 2005. *Operating Systems* 5th ed. Upper Saddle River, New Jersey: Prentice Hall. See chapters 3, 9, & 10.

[Tanenbaum 2008] Tanenbaum, Andrew S. 2008. *Modern Operating Systems* 3rd ed. Upper Saddle River, NJ: Prentice Hall. See chapter 2.