Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
Objectives

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a cycle of CPU execution and I/O wait
  - CPU burst followed by I/O burst
  - CPU burst distribution is of main concern
Histogram of CPU-burst Times

CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state (e.g., an interrupt occurs)
  3. Switches from waiting to ready (e.g., at completion of I/O)
  4. Terminates
- Scheduling under 1 and 4 is **non-preemptive**
- All other scheduling is **preemptive**
  - Consider access to shared data
  - Consider preemption while in kernel mode => waiting for a system call to complete
  - Consider interrupts occurring during crucial OS activities
Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term 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

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time
- E.g., to guarantee that all users get good service, we may want to minimize the maximum response time.

First-Come, First-Served (FCFS) Scheduling

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst 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 Chart for the schedule is:

<table>
<thead>
<tr>
<th></th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$
**FCFS Scheduling (Cont.)**

Suppose that the processes arrive in the order: $P_2, P_3, P_1$

- The Gantt chart for the schedule is:

<table>
<thead>
<tr>
<th></th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td>30</td>
</tr>
</tbody>
</table>

- 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** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

---

**Shortest-Job-First (SJF) Scheduling**

- Associate with each process the length of its **next CPU burst**
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user
Example of SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>6</td>
</tr>
<tr>
<td>$P_2$</td>
<td>8</td>
</tr>
<tr>
<td>$P_3$</td>
<td>7</td>
</tr>
<tr>
<td>$P_4$</td>
<td>3</td>
</tr>
</tbody>
</table>

- SJF scheduling chart

<table>
<thead>
<tr>
<th></th>
<th>$P_1$</th>
<th>$P_3$</th>
<th>$P_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging
  1. $t_n = $ actual length of $n^{th}$ CPU burst
  2. $\tau_{n+1} = $ predicted value for the next CPU burst
  3. $\alpha$, $0 \leq \alpha \leq 1$
  4. Define: $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

- Commonly, $\alpha$ set to $\frac{1}{2}$
- Preemptive version called shortest-remaining-time-first
Prediction of the Length of the Next CPU Burst

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:
  $$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \ldots + (1 - \alpha)^j \alpha t_{n-j} + \ldots + (1 - \alpha)^{n+1} t_0$$
- Since both $\alpha$ and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor
Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>$P_2$</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>$P_4$</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

- Preemptive SJF Gantt Chart

- Average waiting time = $\frac{[(10-1)+(1-1)+(17-2)+5-3]}{4} = \frac{26}{4} = 6.5$ msec

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer $\equiv$ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling
- Problem $\equiv$ Starvation – low priority processes may never execute
- Solution $\equiv$ Aging – as time progresses increase the priority of the process
Example of Priority Scheduling

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>P₂</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P₃</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P₄</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>P₅</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

- Priority scheduling Gantt Chart

- Average waiting time = 8.2 msec

Round Robin (RR)

- Each process gets a small unit of CPU time (time quantum \(q\)), usually 10-100 milliseconds. 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.

- Timer interrupts every quantum to schedule next process

- Performance
  - \(q\) large \(\Rightarrow\) FCFS
  - \(q\) small \(\Rightarrow q\) must be large with respect to context switch, otherwise overhead is too high
Example of RR with Time Quantum = 4

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>24</td>
</tr>
<tr>
<td>P₂</td>
<td>3</td>
</tr>
<tr>
<td>P₃</td>
<td>3</td>
</tr>
</tbody>
</table>

- The Gantt chart is:

Typically, higher average turnaround than SJF, but better response
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time

- process time = 10
  - quantum
    - 12
  - context switches
    - 0
- process time = 10
  - quantum
    - 6
  - context switches
    - 1
- process time = 10
  - quantum
    - 1
  - context switches
    - 9
6.23 Turnaround Time Varies With The Time Quantum

80% of CPU bursts should be shorter than q

<table>
<thead>
<tr>
<th>process</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>6</td>
</tr>
<tr>
<td>P₂</td>
<td>3</td>
</tr>
<tr>
<td>P₃</td>
<td>1</td>
</tr>
<tr>
<td>P₄</td>
<td>7</td>
</tr>
</tbody>
</table>

6.24 Multilevel Queue

- Ready queue is partitioned into separate queues, e.g.,:
  - foreground (interactive)
  - background (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). **Possibility of starvation.**
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; e.g., 80% to foreground in RR
  - 20% to background in FCFS
Multilevel Queue Scheduling

A process can move between the various queues; aging can be implemented this way.

Multilevel-feedback-queue scheduler defined by the following parameters:

- number of queues
- scheduling algorithms for each queue
- method used to determine when to upgrade a process
- method used to determine when to demote a process
- method used to determine which queue a process will enter when that process needs service
Example of Multilevel Feedback Queue

- Three queues:
  - $Q_0$ – RR with time quantum 8 milliseconds
  - $Q_1$ – RR time quantum 16 milliseconds
  - $Q_2$ – FCFS

- Scheduling
  - A new job enters queue $Q_0$ which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue $Q_1$
  - At $Q_1$ job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue $Q_2$

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
  - **Homogeneous processors** within a multiprocessor
  - **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
  - **Symmetric multiprocessing** (SMP) – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
    - Currently, most common
  - **Processor affinity** – process has affinity for processor on which it is currently running
    - **soft affinity**
    - **hard affinity**
    - Variations including processor sets
NUMA and CPU Scheduling

Note that memory-placement algorithms can also consider affinity.

Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – a periodic task checks the load on each processor, and if it finds an imbalance – pushes task from overloaded CPU to other (idle or less-busy) CPUs
- **Pull migration** – an idle processor pulls a waiting task from a busy processor
**Multicore Processors**

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens

**Multithreaded Multicore System**

![Diagram showing a multithreaded multicore system with time line and thread progression.](image-url)
Real-Time CPU Scheduling

- Can present obvious challenges
- **Soft real-time systems** — no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** — a task must be serviced by its deadline
- Two types of latencies affect performance
  1. **Interrupt latency** — time from arrival of interrupt to start of routine that services interrupt
  2. **Dispatch latency** — time for scheduler to take current process off CPU and switch to another

Real-Time CPU Scheduling (Cont.)

- **Conflict phase of dispatch latency:**
  1. Preemption of any process running in kernel mode
  2. Release by low-priority process of resources needed by high-priority processes
Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: periodic ones require CPU at constant intervals
  - Has processing time \( t \), deadline \( d \), period \( p \)
  - \( 0 \leq t \leq d \leq p \)
  - Rate of periodic task is \( 1/p \)

Virtualization and Scheduling

- Virtualization software schedules multiple guests onto CPU(s)
- Each guest doing its own scheduling
  - Not knowing it doesn’t own the CPUs
  - Can result in poor response time
  - Can effect time-of-day clocks in guests
- Can undo good scheduling algorithm efforts of guests
Rate Monotonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- P₁ is assigned a higher priority than P₂.

Missed Deadlines with Rate Monotonic Scheduling
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

![Diagram showing Earliest Deadline First Scheduling]

Proportional Share Scheduling

- $T$ shares are allocated among all processes in the system

- An application receives $N$ shares where $N < T$

- This ensures each application will receive $N / T$ of the total processor time

POSIX Real-Time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads
- Defines two scheduling classes for real-time threads:
  1. SCHED_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
  2. SCHED_RR - similar to SCHED_FIFO except time-slicing occurs for threads of equal priority
- Defines two functions for getting and setting scheduling policy:
  1. pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)
  2. pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)


POSIX Real-Time Scheduling API

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
    int i, policy;
    pthread_t_tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* get the current scheduling policy */
    if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
        fprintf(stderr, "Unable to get policy.\n");
    else {
        if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
        else if (policy == SCHED_RR) printf("SCHED_RR\n");
        else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
    }
```
POSIX Real-Time Scheduling API (Cont.)

/* set the scheduling policy - FIFO, RR, or OTHER */
if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
    fprintf(stderr, "Unable to set policy.\n");
/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i],&attr,runner,NULL);
/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}

End of Chapter 6