Multiprocessor and Real-Time Scheduling

Chapter 10
Roadmap

• Multiprocessor Scheduling
• Real-Time Scheduling
• Linux Scheduling
• Unix SVR4 Scheduling
• Windows Scheduling
Classifications of Multiprocessor Systems

• Loosely coupled or distributed multiprocessor, or cluster
  – Each processor has its own memory and I/O channels

• Functionally specialized processors
  – Such as I/O processor
  – Controlled by a master processor

• Tightly coupled multiprocessing
  – Processors share main memory
  – Controlled by operating system
Independent Parallelism

- No synchronization among processes
- Separate application or job
- Example is time-sharing system
Coarse and Very Coarse-Grained Parallelism

• A set of concurrent processes running on a multiprogrammed uniprocessor
  – can be supported on a multiprocessor with little or no change to user software

• Synchronization among processes at a very gross level
Medium-Grained Parallelism

- Single application is a collection of threads
- Threads usually interact frequently
- A high degree of coordination and interaction among the threads of an application
Fine-Grained Parallelism

- Highly parallel applications
- Specialized and fragmented area
### Table 10.1 Synchronization Granularity and Processes

<table>
<thead>
<tr>
<th>Grain Size</th>
<th>Description</th>
<th>Synchronization Interval (Instructions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fine</td>
<td>Parallelism inherent in a single instruction stream.</td>
<td>&lt;20</td>
</tr>
<tr>
<td>Medium</td>
<td>Parallel processing or multitasking within a single application</td>
<td>20-200</td>
</tr>
<tr>
<td>Coarse</td>
<td>Multiprocessing of concurrent processes in a multiprogramming environment</td>
<td>200-2000</td>
</tr>
<tr>
<td>Very Coarse</td>
<td>Distributed processing across network nodes to form a single computing environment</td>
<td>2000-1M</td>
</tr>
<tr>
<td>Independent</td>
<td>Multiple unrelated processes</td>
<td>(N/A)</td>
</tr>
</tbody>
</table>
Valve Example

- Valve (half-life2 etc) found a hybrid approach works best for their games
- Some systems worked best assigned to a single processor. E.G sound mixing
- Others can be threaded so they work on single processors but greatly improve performance spread over multiple processors. E.g. scene rendering
Hybrid Threading for Rendering Module
Scheduling : Design Issues

• Assignment of processes to processors
• Use of multiprogramming on individual processors
• Actual dispatching of a process
Assignment of Processes to Processors

• Treat processors as a pooled resource and assign process to processors on demand

• Permanently assign process to a processor
  – Known as group or gang scheduling
  – Dedicate short-term queue for each processor
  – Less overhead
  – Processor could be idle while another processor has a backlog
Assignment of Processes to Processors

- Global queue
  - Schedule to any available processor

- Master/slave architecture
  - Key kernel functions always run on a particular processor
  - Master is responsible for scheduling
  - Slave sends service request to the master, e.g. peer making I/O request and waiting
  - Disadvantages
    - Failure of master brings down whole system
    - Master can become a performance bottleneck
Assignment of Processes to Processors

- Peer architecture
  - Operating system can execute on any processor
  - Each processor does self-scheduling
  - Complicates the operating system
  - Make sure two processors do not choose the same process
Process Scheduling

- Single queue for all processes
- Multiple queues are used for priorities
- All queues feed to the common pool of processors
Thread Scheduling

• Executes separate from the rest of the process
• An application can be a set of threads that cooperate and execute concurrently in the same address space
• Threads running on separate processors yields a dramatic gain in performance
Approaches for Multiprocessor Thread Scheduling

• Load sharing
  – Processes are not assigned to a particular processor

• Gang scheduling
  – A set of related threads is scheduled to run on a set of processors at the same time
Approaches for Multiprocessor Thread Scheduling

• Dedicated processor assignment
  – Threads are assigned to a specific processor

• Dynamic scheduling
  – Number of threads can be altered during course of execution
Load Sharing

- Load is distributed evenly across the processors
- No centralized scheduler required
- Use global queues
- Versions of load sharing:
  - First come first served (FCFS)
  - Smallest number of threads first
  - Preemptive smallest number of threads first
Disadvantages of Load Sharing

- Central queue needs mutual exclusion
  - May be a bottleneck when more than one processor looks for work at the same time
- Preemptive threads are unlikely resume execution on the same processor
  - Cache use is less efficient
- If all threads are in the global queue, all threads of a program will not gain access to the processors at the same time
- Despite the potential disadvantages, this is one of the most commonly used schemes in current multiprocessors.
Gang Scheduling

- Simultaneous scheduling of threads that make up a single process
- Useful for medium-grained to fine-grained parallel applications where performance severely degrades when any part of the application is not running
- Threads often need to synchronize with each other
Scheduling Groups

Figure 10.2  Example of Scheduling Groups with Four and One Threads [FEIT90]
Dedicated Processor Assignment

• When application is scheduled, its threads are assigned to a processor
• Some processors may be idle
• No multiprogramming of processors
Figure 10.3 Application Speedup as a Function of Number of Processes [TUCK89]
Dynamic Scheduling

- Number of threads in a process are altered dynamically by the application
- Operating system adjust the load to improve utilization
  - Assign idle processors
  - New arrivals may be assigned to a processor that is used by a job currently using more than one processor
  - Hold request until processor is available
  - Assign processor a job in the list that currently has no processors (i.e., to all waiting new arrivals)
Dynamic Scheduling

• Upon release of one or more processors (including job departure),
  – Scan the current queue of unsatisfied requests for processors.
    Assign a single processor to each job in the list that currently has no processors (i.e., to all waiting new arrivals).
    Then scan the list again, allocating the rest of the processors on an FCFS basis
Roadmap

- Multiprocessor Scheduling
- **Real-Time Scheduling**
- Linux Scheduling
- Unix SVR4 Scheduling
- Windows Scheduling
Real-Time Systems

- Correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced.
- Tasks or processes attempt to control or react to events that take place in the outside world.
- These events occur in “real time” and tasks must be able to keep up with them.
Hard vs Soft

- “Hard “ real time task:
  - One that must meet a deadline;
  - otherwise it will cause unacceptable damage or a fatal error to the system.

- “Soft” real time task
  - Has a deadline which is desirable but not mandatory
  - it still makes sense to schedule and complete the task even if it has passed its deadline.
Periodic vs Aperiodic

• Periodic tasks
  – Are completed regularly, once per period \( T \) or \( T \) units apart

• Aperiodic tasks
  – have time constraints either for deadlines or start
Real-Time Systems

• Control of laboratory experiments
• Process control in industrial plants
• Robotics
• Air traffic control
• Telecommunications
• Military command and control systems
• Financial Analysis: Stock Exchange
Characteristics of Real-Time Operating Systems

• Deterministic
  – Operations are performed at fixed, predetermined times or within predetermined time intervals
  – Concerned with how long the operating system delays before acknowledging an interrupt and there is sufficient capacity to handle all the requests within the required time
Characteristics of Real-Time Operating Systems

Responsiveness

• How long, after acknowledgment, it takes the operating system to service the interrupt
  – Includes amount of time to begin execution of the interrupt
  – Includes the amount of time to perform the interrupt
  – Effect of interrupt nesting
Characteristics of Real-Time Operating Systems

• User control
  – User specifies priority
  – Specify paging
  – What processes must always reside in main memory
  – Disk transfer algorithms to use
  – Rights of processes
Characteristics of Real-Time Operating Systems

- **Reliability**
  - Degradation of performance may have catastrophic consequences

- **Fail-soft operation**
  - Ability of a system to fail in such a way as to preserve as much capability and data as possible
  - Stability
Features of Real-Time Operating Systems

• Fast process or thread switch
• Small size
• Ability to respond to external interrupts quickly
• Multitasking with interprocess communication tools such as semaphores, signals, and events
Features of Real-Time Operating Systems

- Use of special sequential files that can accumulate data at a fast rate
- Preemptive scheduling base on priority
- Minimization of intervals during which interrupts are disabled
- Delay tasks for fixed amount of time
- Special alarms and timeouts
Round Robin scheduling unacceptable

- A real-time task would be added to the ready queue to await its next time slice.
- The scheduling time will generally be unacceptable for real-time applications.
Priority driven unacceptable

- Priority scheduling mechanism, giving real-time tasks higher priority. A real-time task would be scheduled as soon as the current process blocks or runs to completion.
- This could lead to a delay of several seconds if a slow, low-priority task were executing at a critical time.

(b) Priority-Driven Nonpreemptive Scheduler
Combine priorities with clock-based interrupts

(c) Priority-Driven Preemptive Scheduler on Preemption Points
Combine priorities with clock-based interrupts

• A more promising approach is to combine priorities with clock-based interrupts.
• Preemption points occur at regular intervals.
• When a preemption point occurs, the currently running task is preempted if a higher-priority task is waiting.
• This would include the preemption of tasks that are part of the operating system kernel.
• Such a delay may be on the order of several milliseconds.
• While this last approach may be adequate for some real-time applications, it will not suffice for more demanding applications.
Immediate Preemption

Request from a real-time process

Real-time process preempts current process and executes immediately

Current process

Real-time process

Scheduling time

(d) Immediate Preemptive Scheduler
Immediate Preemption

• For more demanding applications, the approach that has been taken is sometimes referred to as **immediate preemption**.

• In this case, the operating system responds to an interrupt almost immediately, unless the system is in a critical-code lockout section.

• Scheduling delays for a real-time task can then be reduced to 100 milliseconds or less.
Classes of Real-Time Scheduling Algorithms

• Static table-driven
  – Input to the analysis: periodic arrival time, execution time, periodic ending deadline, and relative priority of each task
  – It performs a static analysis of feasible schedules of dispatching.
  – The result of the analysis is a schedule that determines, at run time, when a task must begin execution.

• Static priority-driven preemptive
  – static analysis is performed, but no schedule is drawn up.
  – Priority assignment is related to the time constraints associated with each task
  – The analysis is used to assign priorities to tasks, so that a traditional priority-driven preemptive scheduler can be used.
Classes of Real-Time Scheduling Algorithms

• Dynamic planning-based
  – Feasibility determined at run time
  – An arriving task is accepted for execution only if it is feasible to meet its time constraints.
  – One of the results of the feasibility analysis is a schedule or plan that is used to decide when to dispatch this task.
Classes of Real-Time Scheduling Algorithms

• Dynamic best effort
  – used by many real-time systems
  – No feasibility analysis is performed
  – System assigns a priority based on the characteristics of the task
  – The system tries to meet all deadlines and aborts any started process whose deadline is missed.
Deadline Scheduling

• Real-time applications are not concerned with speed but with completing tasks
  – despite dynamic resource demands and conflicts, processing overloads, and hardware or software faults.

• “Priorities” are a crude tool and may not capture the time-critical element of the tasks
Deadline Scheduling

- Information used
  - Ready time
  - Starting deadline
  - Completion deadline
  - Processing time
  - Resource requirements
  - Priority
  - Subtask scheduler

A task may be decomposed into a mandatory subtask and an optional subtask. Only the mandatory subtask possesses a hard deadline.
Preemption

• Other critical design issue is of preemption.
• When starting deadlines are specified, then a nonpreemptive scheduler makes sense.
• E.G. if task X is running and task Y is ready, there may be circumstances in which the only way to allow both X and Y to meet their completion deadlines is to preempt X, execute Y to completion, and then resume X to completion.
## Two Tasks

**Table 10.2 Execution Profile of Two Periodic Tasks**

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Execution Time</th>
<th>Ending Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(1)</td>
<td>0</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>A(2)</td>
<td>20</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>A(3)</td>
<td>40</td>
<td>10</td>
<td>60</td>
</tr>
<tr>
<td>A(4)</td>
<td>60</td>
<td>10</td>
<td>80</td>
</tr>
<tr>
<td>A(5)</td>
<td>80</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>B(1)</td>
<td>0</td>
<td>25</td>
<td>50</td>
</tr>
<tr>
<td>B(2)</td>
<td>50</td>
<td>25</td>
<td>100</td>
</tr>
</tbody>
</table>


Animation of Periodic with Completion Deadline: 
http://gaia.ecs.csus.edu/%7ezhangd/oscal/pdeadlineschedulingperiodic.html

- As an example of scheduling periodic tasks with completion deadlines, consider a system that collects and processes data from two sensors, A and B.

- The computer is capable of making a scheduling decision every 10 ms

- The deadline for collecting data from
  - sensor A must be met every 20 ms,
  - for B every 50 ms.

It takes 10 ms, including operating system overhead, to process each sample of data from A
- and 25 ms to process each sample of data from B.

Table 10.2 summarizes the execution profile of the two tasks.
Figure 10.6 Scheduling of Periodic Real-time Tasks with Completion Deadlines (based on Table 10.2)
## Execution Profile

### Table 10.3  Execution Profile of Five Aperiodic Tasks

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Execution Time</th>
<th>Starting Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>20</td>
<td>110</td>
</tr>
<tr>
<td>B</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>C</td>
<td>40</td>
<td>20</td>
<td>50</td>
</tr>
<tr>
<td>D</td>
<td>50</td>
<td>20</td>
<td>90</td>
</tr>
<tr>
<td>E</td>
<td>60</td>
<td>20</td>
<td>70</td>
</tr>
</tbody>
</table>
Aperiodic Scheduling

• Straightforward scheme: to schedule the ready task with the earliest deadline and let that task run to completion.
  – Although task B requires immediate service, the service is denied.
• There the risk in dealing with aperiodic tasks, especially with starting deadlines.
• A refinement of the policy will improve performance
  – if deadlines can be known in advance of the time that a task is ready.
• This policy, referred to as earliest deadline with unforced idle times, operates as follows:
  – Always schedule the eligible task with the earliest deadline and let that task run to completion.
Aperiodic Scheduling

- An eligible task may not be ready, and this may result in the processor remaining idle even though there are ready tasks.

- System refrains from scheduling task A even though that is the only ready task.
  - the processor is not used to maximum efficiency, all scheduling requirements are met.

- Finally, for comparison, the FCFS policy is shown. In this case, tasks B and E do not meet their deadlines.
Aperiodic Scheduling

Figure 10.7 Scheduling of Aperiodic Real-time Tasks with Starting Deadlines
Rate Monotonic Scheduling

- Assigns priorities to tasks on the basis of their periods.
- The task’s period, $T$, is the amount of time between the arrival of one instance of the task and the arrival of the next instance of the task.
- A task’s rate (in Hertz) is simply the inverse of its period (in seconds). For example, a task with a period of 50 ms occurs at a rate of 20 Hz.
- Highest-priority task is the one with the shortest period.
- Experiment:
  http://gaia.ecs.csus.edu/~zhangd/oscal/pschedulingrms.html
Rate Monotonic Scheduling

- The execution (or computation) time, \( C \), is the amount of processing time required for each occurrence of the task.
- On uniprocessor system, the execution time must not be greater than the period (must have \( C \leq T \)).
- For a period of 80 ms and an execution time of 55 ms, its processor utilization is \( 55/80 = 0.6875 \).
Rate Monotonic Scheduling

- One measure of the effectiveness of a periodic scheduling algorithm is whether or not it guarantees that all hard deadlines are met.
- We have $n$ tasks, each with a fixed period and execution time. To meet all deadlines, the following inequality must hold:
  \[
  \frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_n}{T_n} \leq 1
  \]
- The sum of the processor utilizations of the individual tasks must not exceed a value of 1, which corresponds to total utilization of the processor for a perfect scheduling algorithm.
Rate Monotonic Scheduling

• For any particular algorithm, the bound may be lower.

• For RMS, it can be shown that the following inequality holds:

\[
\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1)
\]
Rate Monotonic Scheduling

- As the number of tasks increases, the scheduling bound ($\ln 2$) converges to 0.693.

Table 10.4 Value of the RMS Upper Bound

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n(2^{1/n} - 1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>0.828</td>
</tr>
<tr>
<td>3</td>
<td>0.779</td>
</tr>
<tr>
<td>4</td>
<td>0.756</td>
</tr>
<tr>
<td>5</td>
<td>0.743</td>
</tr>
<tr>
<td>6</td>
<td>0.734</td>
</tr>
</tbody>
</table>

$\infty$ $\ln 2 = 0.693$
Figure 10.8  A Task Set with RMS [WARR91]
Figure 10.9 Periodic Task Timing Diagram
Priority Inversion

- Can occur in any priority-based preemptive scheduling scheme
- Occurs when circumstances within the system force a higher priority task to wait for a lower priority task
Unbounded Priority Inversion

- Duration of a priority inversion depends on unpredictable actions of other unrelated tasks
Priority Inheritance

- Lower-priority task inherits the priority of any higher priority task pending on a resource they share.
Linux Scheduling

• Scheduling classes
  – SCHED_FIFO: First-in-first-out real-time threads
  – SCHED_RR: Round-robin real-time threads
  – SCHED_OTHER: Other, non-real-time threads

• Within each class multiple priorities may be used
Roadmap

• Multiprocessor Scheduling
• Real-Time Scheduling
• Linux Scheduling
• Unix SVR4 Scheduling
• Windows Scheduling
Linux Real Time Scheduling Classes

- **SCHED_FIFO**: First-in-first-out real-time threads
- **SCHED_RR**: Round-robin real-time threads
- **SCHED_OTHER**: Other, non-real-time threads
Linux Real-Time Scheduling

(a) Relative thread priorities

(b) Flow with FIFO scheduling

(c) Flow with RR scheduling

Figure 10.11 Example of Linux Real-Time Scheduling
Non-Real-Time Scheduling

- Linux 2.6 uses a new scheduler the O(1) scheduler
- Time to select the appropriate process and assign it to a processor is constant
  - Regardless of the load on the system or number of processors
struct prio_array {
    int nr_active;  /* number of tasks in this array */
    unsigned long bitmap[BITMAP_SIZE]; /* priority bitmap */
    struct list_head queue[MAX_PRIOR]; /* priority queues */
}

Active Queues:
140 queues by priority; each queue contains ready tasks for that priority

Expired Queues:
140 queues by priority; each queue contains ready tasks with expired time slices for that priority

Figure 10.12 Linux Scheduling Data Structures for Each Processor
Roadmap

- Multiprocessor Scheduling
- Real-Time Scheduling
- Linux Scheduling
- Unix SVR4 Scheduling
- Windows Scheduling
SVR4 Scheduling

• A complete overhaul of the scheduling algorithm used in earlier UNIX systems.

• The new algorithm is designed to give:
  – highest preference to real-time processes,
  – next-highest preference to kernel-mode processes,
  – and lowest preference to other user-mode processes, referred to as time-shared processes.
UNIX SVR4 Scheduling

- New features include:
  - Preemptable static priority scheduler
  - Introduction of a set of 160 priority levels divided into three priority classes
  - Insertion of preemption points
### SVR Priority Classes

<table>
<thead>
<tr>
<th>Priority Class</th>
<th>Global Value</th>
<th>Scheduling Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real-time</td>
<td>159</td>
<td>first</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Kernel</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td></td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>Time-shared</td>
<td>59</td>
<td>last</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 10.13  SVR4 Priority Classes*
SVR Priority Classes

- **Real time (159 – 100)**
  - Guaranteed to be selected to run before any kernel or time-sharing process
  - Can preempt kernel and user processes

- **Kernel (99 – 60)**
  - Guaranteed to be selected to run before any time-sharing process

- **Time-shared (59-0)**
  - Lowest-priority
SVR4 Dispatch Queues

Figure 10.14  SVR4 Dispatch Queues
Roadmap

• Multiprocessor Scheduling
• Real-Time Scheduling
• Linux Scheduling
• Unix SVR4 Scheduling
• Windows Scheduling
Windows Scheduling

• Priorities organized into two bands or classes
  – Real time
  – Variable

• Priority-driven preemptive scheduler
Windows Thread Dispatching Priorities

Figure 10.15 Windows Thread Dispatching Priorities
Figure 10.16 Example of Windows Priority Relationship
Multiprocessor Scheduling

- With multiprocessors, multiple threads with the same highest priority share the processor in a round robin fashion
  - Lower-priority, threads must wait until the other threads block or have their priority decay.
- Lower-priority threads may also have their priority boosted briefly to 15 if they are being starved, to prevent priority inversion.