CISC 7310X

C05c: Thread, Multiprocessor, and Real-time Scheduling

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College
Acknowledgement

• This slides are a revision of the slides by the authors of the textbook
Outline

• Thread Scheduling
• Multi-Processor Scheduling
• Real-Time CPU Scheduling

• Operating Systems Examples
• Algorithm Evaluation
Thread Scheduling

• Distinction between user-level and kernel-level threads

• When threads supported, threads scheduled, not processes

• 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
  
  • Typically done via priority set by programmer

• Kernel thread scheduled onto available CPU is system-contention scope (SCS) – competition among all threads in system
Example: Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS - Linux and macOS only allow PTHREAD_SCOPE_SYSTEM
Pthread Scheduling API

• pthread_attr_getscope
• pthread_attr_setscope
Questions?

• Thread scheduling
• SCS
• Pthread example
Multiple-Processor Scheduling

• CPU scheduling more complex when multiple CPUs are available

• Multiprocess may be any one of the following architectures:
  • Multicore CPUs
  • Multithreaded cores
  • NUMA systems
  • Heterogeneous multiprocessing
Multiple-Processor Scheduling: SMP

- Symmetric multiprocessing (SMP) is where each processor is self scheduling.
- All threads may be in a common ready queue (a)
- Each processor may have its own private queue of threads (b)
common ready queue

(a)

core_0 \rightarrow T_0 \rightarrow \text{core}_1

core_1 \rightarrow T_1 \rightarrow \text{core}_2

... \rightarrow T_n \rightarrow \text{core}_n

per-core run queues

(b)

core_0 \rightarrow T_0

core_1 \rightarrow T_1 \rightarrow \text{core}_2

... \rightarrow T_n \rightarrow \text{core}_n
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
A timeline showing a sequence of compute cycles (C) and memory stall cycles (M). The diagram illustrates how a thread progresses through a series of cycles, with compute cycles represented by C and memory stall cycles by M.
Each core has > 1 hardware threads.

If one thread has a memory stall, switch to another thread!
Multithreaded Multicore System

- Chip-multithreading (CMT) assigns each core multiple hardware threads. (Intel refers to this as hyperthreading.)
Multithreaded Multicore System: Example

- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.
Multithreaded Multicore System: Scheduling

• Two levels of scheduling:

1. The operating system deciding which software thread to run on a logical CPU

2. How each core decides which hardware thread to run on the physical core.
software threads

hardware threads (logical processors)

level 1

processing core

level 2
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** - periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs

• **Pull migration** - idle processors pulls waiting task from busy processor
Multiple-Processor Scheduling - Processor Affinity

• When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.

• We refer to this as a thread having affinity for a processor (i.e. “processor affinity”)

• 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 of.

• **Soft affinity** - the operating system attempts to keep a thread running on the same processor, but no guarantees.

• **Hard affinity** - allows a process to specify a set of processors it may run on.
NUMA and CPU Scheduling

• Non-uniform memory access

• If the operating system is NUMA-aware, it will assign memory closes to the CPU the thread is running on.
Questions?

• Multiprocessor scheduling
• Design considerations
  • Multicore vs multprocessor
  • SMP
  • NUMA
• Load balancing
• CPU Affinity
Real-Time CPU Scheduling

• Can present obvious challenges

• **Soft real-time systems** - Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled

• **Hard real-time systems** - task must be serviced by its deadline
Real-Time CPU Scheduling

• Event latency – the amount of time that elapses from when an event occurs to when it is serviced.

• 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 schedule to take current process off CPU and switch to another
event E first occurs

$t_0$

event latency

$t_1$

real-time system responds to E

Time
Interrupt Latency

- Task T running
- Determine interrupt type
- Context switch
- ISR

Interrupt latency

Time
Dispatch Latency

- 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 $1/p$
Rate Montonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- \( P_1 \) is assigned a higher priority than \( P_2 \).
Missed Deadlines with Rate Monotonic Scheduling

• Process P2 misses finishing its deadline at time 80
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
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 $\frac{N}{T}$ of the total processor time
Example: 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

- pthread_attr_getschedpolicy
- pthread_attr_setschedpolicy
Questions

• Real-time scheduling
• Scheduling algorithms
• Example API
Operating System Examples

• Linux scheduling

• Windows scheduling

• Solaris scheduling
Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm

Version 2.5 moved to constant order $O(1)$ scheduling time

- Preemptive, priority based
- Two priority ranges: time-sharing and real-time
- **Real-time** range from 0 to 99 and **nice** value from 100 to 140
- Map into global priority with numerically lower values indicating higher priority
- Higher priority gets larger $q$
- Task runnable as long as time left in time slice (**active**)
- If no time left (**expired**), not runnable until all other tasks use their slices
- All runnable tasks tracked in per-CPU **runqueue** data structure
  - Two priority arrays (active, expired)
  - Tasks indexed by priority
  - When no more active, arrays are exchanged

**Worked well, but poor response times for interactive processes**
Linux Scheduling in Version 2.6.23 +

• **Completely Fair Scheduler** (CFS)
  
• **Scheduling classes**
  
  • Each has specific priority
  
  • Scheduler picks highest priority task in highest scheduling class
  
  • Rather than quantum based on fixed time allotments, based on proportion of CPU time
  
  • 2 scheduling classes included, others can be added
  
    1. default
    2. real-time
  
• Quantum calculated based on **nice value** from -20 to +19
  
  • Lower value is higher priority
  
  • Calculates target latency – interval of time during which task should run at least once
  
  • Target latency can increase if say number of active tasks increases
  
• CFS scheduler maintains per task **virtual run time** in variable vruntime
  
  • Associated with decay factor based on priority of task - lower priority is higher decay rate
  
  • Normal default priority yields virtual run time = actual run time
  
• To decide next task to run, scheduler picks task with lowest virtual run time
CFS Performance

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of \( vrunt ime \). This tree is shown below:

![Diagram of a red-black tree with tasks T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, with T0 at the root. Arrows indicate the order of tasks based on their \( vrunt ime \) values.]

When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of \( vrunt ime \)) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require \( O(\log N) \) operations (where \( N \) is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable \( rb\_leftmost \), and thus determining which task to run next requires only retrieving the cached value.
Linux Scheduling (Cont.)

- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
  - Nice value of -20 maps to global priority 100
  - Nice value of +19 maps to priority 139
Linux Scheduling (Cont.)

- Linux supports load balancing, but is also NUMA-aware.

- **Scheduling domain** is a set of CPU cores that can be balanced against one another.

- Domains are organized by what they share (i.e. cache memory). Goal is to keep threads from migrating between domains.
Windows Scheduling

• Windows uses priority-based preemptive scheduling
• Highest-priority thread runs next
• **Dispatcher** is scheduler
• Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
• Real-time threads can preempt non-real-time
• 32-level priority scheme
• **Variable class** is 1-15, **real-time class** is 16-31
• Priority 0 is memory-management thread
• Queue for each priority
• If no run-able thread, runs **idle thread**
Windows Priority Classes

• Win32 API identifies several priority classes to which a process can belong
  • REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
  • All are variable except REALTIME
• A thread within a given priority class has a relative priority
  • TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
• Priority class and relative priority combine to give numeric priority
• Base priority is NORMAL within the class
• If quantum expires, priority lowered, but never below base
Windows Priority Classes (Cont.)

• If wait occurs, priority boosted depending on what was waited for
• Foreground window given 3x priority boost
• Windows 7 added user-mode scheduling (UMS)
  • Applications create and manage threads independent of kernel
  • For large number of threads, much more efficient
  • UMS schedulers come from programming language libraries like C++
    Concurrent Runtime (ConcRT) framework
<table>
<thead>
<tr>
<th>Priority</th>
<th>Real-time</th>
<th>High</th>
<th>Above Normal</th>
<th>Normal</th>
<th>Below Normal</th>
<th>Idle Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>time-critical</td>
<td>31</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>highest</td>
<td>26</td>
<td>15</td>
<td>12</td>
<td>10</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>above normal</td>
<td>25</td>
<td>14</td>
<td>11</td>
<td>9</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>normal</td>
<td>24</td>
<td>13</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>below normal</td>
<td>23</td>
<td>12</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>lowest</td>
<td>22</td>
<td>11</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>idle</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Solaris

- Priority-based scheduling
- Six classes available
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
  - Loadable table configurable by sysadmin
## Solaris Dispatch Table

<table>
<thead>
<tr>
<th>priority</th>
<th>time quantum</th>
<th>time quantum expired</th>
<th>return from sleep</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>200</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>160</td>
<td>0</td>
<td>51</td>
</tr>
<tr>
<td>15</td>
<td>160</td>
<td>5</td>
<td>51</td>
</tr>
<tr>
<td>20</td>
<td>120</td>
<td>10</td>
<td>52</td>
</tr>
<tr>
<td>25</td>
<td>120</td>
<td>15</td>
<td>52</td>
</tr>
<tr>
<td>30</td>
<td>80</td>
<td>20</td>
<td>53</td>
</tr>
<tr>
<td>35</td>
<td>80</td>
<td>25</td>
<td>54</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
<td>30</td>
<td>55</td>
</tr>
<tr>
<td>45</td>
<td>40</td>
<td>35</td>
<td>56</td>
</tr>
<tr>
<td>50</td>
<td>40</td>
<td>40</td>
<td>58</td>
</tr>
<tr>
<td>55</td>
<td>40</td>
<td>45</td>
<td>58</td>
</tr>
<tr>
<td>59</td>
<td>20</td>
<td>49</td>
<td>59</td>
</tr>
</tbody>
</table>
Solaris Scheduling

- Interrupt threads
- Realtime (RT) threads
- System (SYS) threads
- Fair share (FSS) threads
- Fixed priority (FX) threads
- Timeshare (TS) threads
- Interactive (IA) threads
Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
- Thread with highest priority runs next
- Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Multiple threads at same priority selected via RR
Algorithm Evaluation

• How to select CPU-scheduling algorithm for an OS?

• Determine criteria, then evaluate algorithms

• Deterministic modeling
  • Type of analytic evaluation
    • Takes a particular predetermined workload and defines the performance of each algorithm for that workload.

• Consider 5 processes arriving at time 0:

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>10</td>
</tr>
<tr>
<td>P₂</td>
<td>29</td>
</tr>
<tr>
<td>P₃</td>
<td>3</td>
</tr>
<tr>
<td>P₄</td>
<td>7</td>
</tr>
<tr>
<td>P₅</td>
<td>12</td>
</tr>
</tbody>
</table>
Deterministic Evaluation

- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:
    - Non-preemptive SFJ is 13ms:
  - RR is 23ms:
Queueing Models

• 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 servers, each with queue of waiting processes
  • Knowing arrival rates and service rates
  • Computes utilization, average queue length, average wait time, etc
Little’s Formula

- $n = \text{average queue length}$
- $W = \text{average waiting time in queue}$
- $\lambda = \text{average arrival rate into queue}$

Little’s law – in steady state, processes leaving queue must equal processes arriving, thus:

$$n = \lambda \times W$$

- Valid for any scheduling algorithm and arrival distribution

- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds
Simulations

• Queueing models limited

• **Simulations** more accurate
  
  • Programmed model of computer system
  
  • Clock is a variable
  
  • Gather statistics indicating algorithm performance
  
  • Data to drive simulation gathered via
    
    • Random number generator according to probabilities
    
    • Distributions defined mathematically or empirically
    
    • Trace tapes record sequences of real events in real systems
Evaluation of CPU Schedulers by Simulation
Implementation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary
End of Chapter 5
Objectives

• Describe various CPU scheduling algorithms

• Assess CPU scheduling algorithms based on scheduling criteria

• Explain the issues related to multiprocessor and multicore scheduling

• Describe various real-time scheduling algorithms

• Describe the scheduling algorithms used in the Windows, Linux, and Solaris operating systems

• Apply modeling and simulations to evaluate CPU scheduling algorithms