Real-Time Operating Systems M

4. Process Scheduling • Files • Shell
Notice

The course material includes slides downloaded from:

http://codex.cs.yale.edu/avi/os-book/


and

http://retis.sssup.it/~giorgio/rts-MECS.html


which has been edited to suit the needs of this course.

The slides are authorized for personal use only.

Any other use, redistribution, and any for profit sale of the slides (in any form) requires the consent of the copyright owners.
Chapter 5: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Operating Systems Examples
- Algorithm Evaluation
Quizzes

- SJF is a kind of priority scheduling algorithm
- Multilevel feedback queues and FCFS are incompatible
- FCFS is a kind of priority scheduling algorithm
- RR and SJF have no relation with one another
- Multilevel feedback queues implement aging
- SJF may cause a process’ starvation
- Multilevel queue scheduling may cause starvation
Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- M:1 and M:M → thread library schedules user-level threads to run on LWP
  - scheduling competition within the process (process-contention scope, PCS)
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is system-contention scope (SCS) – competition among all threads in system

LWP
- Appears to be a virtual processor on which process can schedule user thread to run
- Each LWP attached to kernel thread

Lightweight process

Kernel thread

User thread

LWP

Process-contention scope (PCS)

System-contention scope (SCS)
Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - `PTHREAD_SCOPE_PROCESS` schedules user-level threads using PCS scheduling
    - Onto available LWPs
  - `PTHREAD_SCOPE_SYSTEM` schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow `PTHREAD_SCOPE_SYSTEM`
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 (use cache efficiently)
  - soft affinity vs hard affinity
  - Variations including processor sets
  - Memory architecture can affect processor affinity issues
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** – 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

- Both techniques can co-exist
  - load balancing algorithm run periodically or if empty run queue
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

- **C**: compute cycle
- **M**: memory stall cycle

**Thread**
- Thread 1: C M C M C M C
- Thread 0: C M C M C M C

**Time**
Multithreaded Multicore System

- Coarse-grained multithreading: execute thread until memory stall
  - When high cost of switching (need to flush instruction pipeline)
- Fine-grained multithreading: interleaved
  - When dedicated logic for thread switching

- Two scheduling levels
  - Which software thread to run on each hardware thread (logical processor)
  - Which hardware thread to run inside each core
Operating System Examples

- Solaris scheduling
- Windows scheduling
- Linux scheduling
Solaris

- Priority-based preemptive scheduling
- Six classes available
  - Time sharing (default) (TS) / dynamic priorities
  - Interactive (IA) / dynamic priorities
  - Real time (RT) / highest priority
  - System (SYS) / kernel threads (scheduler & paging daemons)
  - Fair Share (FSS) / CPU shares allocated to projects
  - Fixed priority (FP) / same priority as in time-sharing
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
  - 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

Scheduling order:
- First
- Last
Scheduler converts class-specific priorities into a per-thread global priority

- Thread with highest priority runs next
- Runs until it either
  1. blocks, or
  2. uses time slice, or
  3. is preempted by higher-priority thread
- Multiple threads at same priority selected via RR
Windows Scheduling

- Priority-based preemptive scheduling
- **Dispatcher** is scheduler
  - Highest-priority thread runs next
  - Thread runs until it either
    1. blocks, or
    2. uses time slice, or
    3. is 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
- If wait occurs, priority boosted depending on what was waited for
- Foreground window given 3x quantum priority boost
  - delays time-sharing preemption
### Windows Priorities

<table>
<thead>
<tr>
<th>Priority Level</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>
Linux Scheduling Through Version 2.5

- Priority-based preemptive 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
  - Two priority ranges: time-sharing and real-time
  - **Real-time** range from 0 to 99 and **nice** value from 100 to 139
  - Map into global priority (lower values is higher priority)
  - Higher priority gets larger $q$
  - Task run-able as long as time left in time slice (**active**)
  - If no time left (**expired**), not run-able until all other tasks use their slices
  - All run-able 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
## List of Tasks Indexed According to Priorities

**active array**

<table>
<thead>
<tr>
<th>priority</th>
<th>task lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td></td>
</tr>
<tr>
<td>[1]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>[140]</td>
<td></td>
</tr>
</tbody>
</table>

**expired array**

<table>
<thead>
<tr>
<th>priority</th>
<th>task lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td></td>
</tr>
<tr>
<td>[1]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>[140]</td>
<td></td>
</tr>
</tbody>
</table>
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
  - Quantum based on proportion of CPU time (rather than on fixed time allotments)
  - 2 scheduling classes included, others can be added
    1. default
    2. real-time

- Quantum calculated based on *nice value* from -20 to +19
  - 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 \( v\text{runtime} \). This tree is shown below:

![CFS Tree Diagram]

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 \( v\text{runtime} \)) 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.
Nodes represent `sched_entity(s)` indexed by their virtual runtime.

```
struct task_struct {
    volatile long state;
    void *stack;
    unsigned int flags;
    int prio, static prio, normal prio;
    const struct sched_class *sched_class;
    struct sched_entity se;
    ...
};
```

```
struct sched_entity {
    struct load_weight
    struct rb_node run;
    struct list_head gr;
    ...
};
```

```
struct ofs_rq {
    struct rb_root tasks_timeline;
    ...
};
```

```
struct rb_node {
    unsigned long rb_pare;
    struct rb_node *rb_r;
    struct rb_node *rb_l;
};
```
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

```
Real-Time          Normal
0                  99 100 139
Higher             Lower
Priority
```
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>$P_1$</td>
<td>10</td>
</tr>
<tr>
<td>$P_2$</td>
<td>29</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3</td>
</tr>
<tr>
<td>$P_4$</td>
<td>7</td>
</tr>
<tr>
<td>$P_5$</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

- \( n = \) average queue length
- \( W = \) average waiting time in queue
- \( \lambda = \) 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
Simulations

- Simulation of actual process execution
- Performance statistics for FCFS
- Performance statistics for SJF
- Performance statistics for RR (q = 14)
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
A Very Short Introduction to Files & Shell

- Using Files
  - File concept
  - File system
  - Common operations on files
  - System calls

- UNIX Shell
  - Useful commands
  - Input/output redirection and command piping
File Concept

- Contiguous logical address space
- Attributes
  - **Name** – only information kept in human-readable form
  - **Type**
  - **Size**
  - **Location** on device
  - **Identifier** within file system
  - **Time, date, and user identification**
  - **Protection**
  - **etc**
- Information kept in the directory structure
File Control Block

- Directory structure organizes the files
- Per-file File Control Block (FCB) contains many details about the file

<table>
<thead>
<tr>
<th>file permissions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>file dates (create, access, write)</td>
<td></td>
</tr>
<tr>
<td>file owner, group, ACL</td>
<td></td>
</tr>
<tr>
<td>file size</td>
<td></td>
</tr>
<tr>
<td>file data blocks or pointers to file data blocks</td>
<td></td>
</tr>
</tbody>
</table>
Common Operations on Files

- **Write** – at *write pointer* location
- **Read** – at *read pointer* location
- **Create**
- **Seek** (reposition pointer)
- **Delete**
- **Truncate**

- Before using a file: need to **open**
Open Files

- Data needed to manage open files:
  - **Open-file table**: tracks open files
  - **File pointer**: pointer to last read/write location, per process that has the file open
  - **File-open count**: counter of number of times a file is open to allow removal of data from open-file table when last process closes it
  - **Access rights**: per-process access mode information
  - etc
In-Memory File System Structures

(a) open (file name)
user space

directory structure
kernel memory

file-control block
secondary storage

(b) read (index)
user space

per-process open-file table
kernel memory

system-wide open-file table

index
data blocks

file-control block
secondary storage
File Descriptor vs File Pointer

- **File descriptor**: a (usually small) integer number
  - Points to entry in process-wide open-file table
  - Value returned by open()
  - Used explicitly when operating on files (read, write, etc)

- Three files open by default when creating new process:
  - 0 – Standard **input** file
  - 1 – Standard **output** file
  - 2 – Standard **error** file

- Process-wide open-file table copied from parent upon fork()
  - Child starts up with same open files as parent
  - File pointers of files opened **before** fork() are **shared**
File Descriptor vs File Pointer

- **File pointer**: a (usually long) integer number
  - Points to byte in open file
  - Created when opening a new file
  - Used implicitly by system calls
  - Updated automatically when reading from/writing to file
    - E.g.: read 5 bytes → file pointers increased by 5
Useful system calls to work with files (unistd.h):

- \texttt{fd = open(pathname, flags, mode)};
- \texttt{read(fd, buffer, count)};
- \texttt{write(fd, buffer, count)};
- \texttt{close(fd)};
UNIX Shell

- Command line interface
  - accepts commands as text input
  - converts commands to appropriate operating system functions
- Each command behaves as a filter
  - Uses input, output, error files
  - Possible to redirect input/output and establish information flows (pipes)
- Some useful shell commands
  - man
  - ls
  - cat
  - less
  - grep
  - ps
  - top
Example of Program Analysis Question

Using the following program, explain what the output will be at LINE A.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <wait.h>

int value = 5;

int main()
{
    pid_t pid;
    pid = fork();

    if (pid == 0) {
        value += 15;
    }
    else if (pid > 0) {
        wait(NULL);
        printf("PARENT: value = %d", value); /* LINE A */
    }
    return 0;
}
```