CPU Scheduling

Thomas Plagemann

(with slides from Otto J. Anshus, Kai Li, Pål Halvorsen and Andrew S. Tanenbaum)

Outline

• Goals of scheduling

• Scheduling algorithms:
  – FCFS/FIFO, RR, STCF/SRTCF
  – Priority (CTSS, UNIX, WINDOWS, LINUX)
  – Lottery
  – Fair share
  – Real-time: EDF and RM
Why Spend Time on Scheduling?

- Optimize the system to the given goals
- Example: CPU-Bound vs. I/O-Bound Processes:
  - Bursts of CPU usage alternate with periods of I/O wait
    - a CPU-bound process
    - an I/O bound process

Scheduling Performance Criteria

- CPU (resource) utilization
  - 100%, but 40-90% normal
- Throughput
  - Number of "jobs" per time unit
  - Minimize overhead of context switches
  - Efficient utilization (CPU, memory, disk etc)
- Turnaround time
  - \( = \text{time}_{\text{process arrives}} - \text{time}_{\text{process exits}} \)
  - \( = \text{sum of all waiting times (memory, R.Q, execution, I/O, etc)} \)
  - How fast a single job got through
- Response time
  - \( = \text{time}_{\text{request starts}} - \text{time}_{\text{response starts}} \)
  - Having a small variance in Response Time is good (predictability)
  - Short response time: type on a keyboard
- Waiting time
  - in the Ready_Queue, for memory, for I/O, etc.
- Fairness
  - no starvation
Scheduling Algorithm Goals

Are we sure about this?

All systems
- Fairness: giving each process a fair share of the CPU
- Policy enforcement: seeing that stated policy is carried out
- Balance: keeping all parts of the system busy

Batch systems

Interactive systems

Real-time systems

Non-Preemptive: FIFO (FCFS) Policy

- Run to
  - to completion (old days)
  - until blocked, yield, or exit

- Advantages

- Disadvantage

### Average Turnaround Time for CPU bursts:

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

**Arrival order: 1 - 2 - 3**

\[
\text{TT}_{\text{average}} = \frac{(24+27+30)}{3} = 27
\]

**Arrival order: 2 - 3 - 1**

\[
\text{TT}_{\text{average}} = \frac{(3+6+30)}{3} = 13
\]
Discussion topic FCFS

How well will FCFS handle:
- Many processes doing I/O arrives
- One CPU-bound process arrives

All the I/O-bound processes execute their I/O instructions

CPU-bound process starts executing.

All I/O-bound processes enter the back of the Ready_Queue

All I/O devices are now IDLE

CPU is IDLE

I/O interrupt

“CONVOY” effect
- Low CPU utilization
- Low device utilization

Round Robin

- FIFO queue
- n processes, each runs a time slice or quantum, q
  - each process gets 1/n of the CPU in max q time units at a time

- Max waiting time in Ready_Queue per process: (n-1) * q

- How do you choose the time slice?
  - Overhead vs. throughputs
  - Overhead is typically about 1% or less
    - interrupt handler + scheduler + dispatch
    - 2 context switches: going down, and up into new process
  - CPU vs. I/O bound processes
FIFO vs. Round Robin

- 10 jobs and each takes 100 seconds

- FIFO

- Round Robin
  - time slice 1s and no overhead

- Comparisons

Case: Time Slice Size

- Resource utilization example
  - A and B each uses 100% CPU
  - C loops forever (1ms CPU and 10ms disk)

- Large or small time slices?
  - nearly 100% of CPU utilization regardless of size
  - Time slice 100ms: nearly 5% of disk utilization with Round Robin
  - Time slice 1ms: nearly 85% of disk utilization with Round Robin

- What do we learn from this example?
  - The right (shorter) time slice can improve overall utilization
  - CPU bound: benefits from having longer time slices (>100 ms)
  - I/O bound: benefits from having shorter time slices (≤10ms)
Shortest Time to Completion First (STCF)  
(a.k.a. Shortest Job First)

- Non-preemptive  
- Run the process having smallest service time  
- Random, FCFS, ... for “equal” processes

- Problem to establish what the running time of a job is  
- Suggestions on how to do this?
  - Length of next CPU-burst
    - Assuming next burst = previous burst  
    - Can integrate over time using a formula taking into account old and new history of CPU burst lengths  
  - But mix of CPU and I/O, so be careful

Shortest Remaining Time to Completion First (SRTCF)  
(a.k.a. Shortest Remaining Time First)

- Preemptive, dynamic version of STCF  
- If a shorter job arrives, PREEMPT current, and do STCF again

- Advantage: high throughput, average turnaround is low  
  (Running a short job before a long decreases the waiting time MORE for the short than it increases for the long!)
- Disadvantage: starvation possible, must know execution time
Priority Scheduling

- Assign each process a priority
- Run the process with highest priority in the ready queue first

- Multiple queues

  - **Advantage**
    - (Fairness)
    - Different priorities according to importance

  - **Disadvantage**
    - Users can hit keyboard frequently
    - Starvation: so should use dynamic priorities

- Special cases (RR in each queue)
  - FCFS (all equal priorities, non-preemptive)
  - STCF/SRTCF (the shortest jobs are assigned the highest priority)

Multiple Queue

- Good for classes of jobs
  - real-time vs. system jobs vs. user jobs vs. batch jobs

- Multi level feedback queues
  - Adjust priority dynamically
    - Aging
    - I/O wait raises the priority
    - Memory demands, #open files, CPU:I/O bursts
  - Scheduling **between** the queues
    - Time slice (and cycle through the queues)
    - Priority typical:
      - Jobs start at highest priority queue
      - If timeout expires (used current time slices), drop one level
      - If timeout doesn’t expires, stay or push up one level
  - Can use different scheduling per queue
  - A job using doing much I/O is moved to an “I/O bound queue”
Compatible Time-Sharing System (CTSS)

- One of the first (1962) priority schedulers using multiple feedback queues (moving processes between queues)
- One process in memory at a time (high switch costs)
- Large slices vs. response time \( \rightarrow \) priority classes
- Each time the quantum was used, the process dropped one priority class (larger slice, less frequent)
- Interaction \( \rightarrow \) back to highest priority class
- Short, interactive should run more often

<table>
<thead>
<tr>
<th>”Priority“</th>
<th>Time slices</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

Scheduling in UNIX

- Many versions
- User processes have positive priorities, kernel negative
- Schedule lowest priority first
- If a process uses the whole time slice, it is put back at the end of the queue (RR)
- Each second the priorities are recalculated:
  
  priority =
  
  CPU_usage (average #ticks)
  + nice (+- 20)
  + base (priority of last corresponding kernel process)
Scheduling in UNIX (cont.)

- SVR3 and 4.3 BSD UNIX: designed primarily for time-sharing interactive environment
- Scheduling algorithm objectives
  - Provide good response time for interactive users
  - Ensure that low-priority background jobs do not starve
- Scheduling algorithm implementation
  - Multilevel feedback using round robin within each of the priority queues
  - 1 second preemption
  - Priority based on process type and execution history
  - Priorities are recomputed once per second
  - Base priority divides all processes into fixed bands of priority levels
  - Bands used to optimize access to block devices (e.g., disk) and to allow the OS to respond quickly to system calls

Priority calculation

\[ P_j(i) = Base_j + \frac{CPU_j(i-1)}{2} \]

Where

- \( CPU_j(i) \) = Measure of processor utilization by process \( j \) through interval \( i \)
- \( P_j(i) \) = Priority of process \( j \) at beginning of interval \( i \); lower values equal higher priorities
- \( Base_j \) = Base priority of process \( j \)
Scheduling in UNIX (cont.)

- Bands in decreasing order of priority
  - Swapper
  - Block I/O device control
  - File manipulation
  - Character I/O device control
  - User processes

- Goals
  - Provide the most efficient use of I/O devices
  - Within the user process band, use the execution history to penalize processor-bound processes at the expense of I/O bound processes

- Example of process scheduling
  - Processes A, B, and C are created at the same time with base priorities of 60
  - Clock interrupts the system 60 times a second and increments counter for running process

Scheduling in Windows 2000

- Preemptive kernel
- 32 priority levels - Round Robin (RR) in each
- Schedules threads individually
- Processor affinity

- Default time slices (3 quantums = 10 ms) of
  - 120 ms – Win2000 server
  - 20 ms – Win2000 professional/workstation
  - may vary between threads

- Interactive and throughput-oriented:
  - "Real time" – 16 system levels
    - fixed priority
    - may run forever
  - Variable – 15 user levels
    - priority may change – thread priority = process priority ± 2
    - uses much CPU cycles → drops
    - user interactions, I/O completions → increase
  - Idle/zero-page thread – 1 system level
    - runs whenever there are no other processes to run
    - clears memory pages for memory manager
Windows 2000 Scheduling

- **Goal:** optimize response for
  - Single user in highly interactive environment, or
  - Server
- **Implementation**
  - Priority-driven preemptive scheduler with round-robin within a priority level
    - When a thread becomes ready and has higher priority than the currently running thread, the lower priority thread is preempted
  - Dynamic priority variation as function of current thread activity for some levels
- Process and thread priorities organized into two bands (classes), each with 16 levels
  - Real-time priorities:
    - Real-time tasks or time-dependent functions (e.g., communications)
    - Have precedence over other threads
  - Variable priorities
    - Non-real-time tasks and functions

CS-550 (M.Soneru): Scheduling in Representative Operating Systems: [Sta’01]
Windows 2000 Scheduling (cont.)

- Real-time priority class
  - All threads have a fixed priority that never changes
  - All active threads at a given priority level are in a round-robin queue

- Variable priority class
  - A thread’s priority begins at some initial assigned value and then may change, up or down, during the thread’s lifetime
  - There is a FIFO queue at each priority level, but a thread may migrate to other queues within the variable priority class
  - The initial priority of a thread is determined by
    - Process base priority: attribute of the process object, from 0 to 15
    - Thread base priority: equal to that of its process or within two levels above or below that of the process
  - Dynamic priority
    - Thread starts with base priority and then fluctuates within given boundaries, never falling under base priority or exceeding 15

Windows 2000 Scheduling (cont.)

- Variable priority class: dynamic priorities
  - If thread is interrupted because it has used its current time quantum, executive lowers its priority: processor-bound threads move toward lower priorities
  - If thread is interrupted to wait on an I/O event, executive raises its priority: I/O-bound threads move toward higher priorities
    - Priority raised more for interactive waits (e.g., wait on keyboard or display)
    - Priority raised less for other I/O (e.g., disk I/O)
  - Example
    - A process object has a base priority of 4
    - Each thread associated with this process can have an initial priority between 2 and 6 and dynamic priorities between 2 and 15
    - The thread’s priority will change based on its execution characteristics
Windows 2000 Scheduling (cont.)

- Single processor vs. multiprocessor scheduling
  - Single processor
    - Highest-priority thread is always active unless it is waiting on an event
    - If there is more than one thread at the highest priority, the processor is shared, round-robin
  - Multiprocessor system with N processors
    - The \((N - 1)\) highest-priority threads are always active, running on the \((N - 1)\) processors
    - All other lower-priority threads share the remaining processor
    - Example: three processors, two highest-priority threads run on two processors, all other threads run on the remaining processor
    - Exception for threads with processor affinity attribute
Scheduling in Linux

- Preemptive kernel
- Threads and processes used to be equal, but Linux uses (in 2.6) thread scheduling

- **SHED_FIFO**
  - may run forever, no timeslices
  - may use its own scheduling algorithm

- **SHED_RR**
  - each priority in RR
  - timeslices of 10 ms (quantums)

- **SHED_OTHER**
  - ordinary user processes
  - uses "nice"-values: 1 ≤ priority ≤ 40
  - timeslices of 10 ms (quantums)

Threads with highest goodness are selected first:
- realtime (FIFO and RR):
  - goodness = 1000 + priority
- timesharing (OTHER):
  - goodness = (quantum > 0 ? quantum + priority : 0)

- Quantums are reset when no ready process has quantums left:
  - quantum = (quantum/2) + priority

<table>
<thead>
<tr>
<th></th>
<th>nice</th>
</tr>
</thead>
<tbody>
<tr>
<td>-20</td>
<td>-19</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
</tr>
</tbody>
</table>

Linux Scheduling

- Enhances the traditional UNIX scheduling by adding two new scheduling classes for real-time processing

- 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

- Multiple priorities can be used within each class; priorities for real-time threads higher than for non-real-time threads
Linux Scheduling (cont.)

- Scheduling for the FIFO threads is done according with the following rules
  - An executing FIFO thread can be interrupted only when
    - Another FIFO thread of higher priority becomes ready
    - The executing FIFO thread becomes blocked, waiting for an event
    - The executing FIFO thread voluntarily gives up the processor (sched_yield)
  - When an executing FIFO thread is interrupted, it is placed in a queue associated with its priority
  - When a FIFO thread becomes ready and it has a higher priority than the currently running thread, the running thread is pre-empted and the thread with the higher priority is executed. If several threads have the same, higher priority, the one that has been waiting the longest is assigned the processor
- Scheduling for the Round-Robin threads is similar, except for a time quota associated with each thread
  - At the end of the time quota, the thread is suspended and a thread of equal or higher priority is assigned the processor

Example: The distinction between FIFO and RR scheduling

- Assumptions (a)
  - A program has four threads with three relative priorities
  - All waiting threads are ready to execute when the current thread waits or terminates
  - No higher-priority thread is awakened while a thread is executing
- Scheduling of FIFO threads (b)
  - Thread D executes until it waits or terminates
  - Thread B executes (it has been waiting longer than C) until waits or terminates
  - Thread C executes
  - Thread A executes
- Scheduling of Round-Robin threads (c)
  - Thread D executes until it waits or terminates
  - Threads B and C execute time-sliced
  - Thread A executes
- Scheduling of non-real-time threads
  - Execute only if no real-time threads are ready using the traditional UNIX scheduling algorithm
Linux Scheduling: FIFO vs. RR

(a) Relative thread priorities

(b) Flow with FIFO scheduling

(c) Flow with RR scheduling

Lottery Scheduling

- Motivations
  - SRTCF does well with average response time, but unfair
  - Guaranteed scheduling may be hard to implement
  - Adjust priority is a bit ad hoc. For example, at what rate?

- Lottery method
  - Give each job a number of tickets
  - Randomly pick a winning tickets
  - To approximate SRTCF, short jobs gets more tickets
  - To avoid starvation, give each job at least one ticket
  - Allows ticket exchange

Fair Share

• Each PROCESS should have an equal share of the CPU
• History of recent CPU usage for each process
• Process with least recently used CPU time := highest priority
  – an editor gets a high priority
  – a compiler gets a low priority

• Each USER should have an equal share of the CPU
• Take into account the owner of a process
• History of recent CPU usage for each user

Real-Time Scheduling

round-robin
request

priority, non-preemptive
request

priority, preemptive
request

NOTE: preemption may also be limited to preemption points (fixed points where the scheduler is allowed to interrupt a running process) giving larger delays
Real-Time Scheduling

- Real-time tasks are often **periodic** (e.g., fixed frame rates and audio sample frequencies)

- Time constraints for a periodic task:
  - $s$ – starting point (first time the task requires processing)
  - $e$ – processing time
  - $d$ – deadline
  - $p$ – period
  - $r$ – rate ($r = 1/p$)
  - $0 \leq e \leq d$ (often $d \leq p$: we’ll use $d = p – \text{end of period}$, but $\sum d \leq \sum p$ is enough)
  - the $k$th processing of the task
    - must start at time $s + (k - 1)p$
    - must be finished at time $s + (k - 1)p + d$
  - the scheduling algorithm must account for these properties

Schedulable Real-Time Systems

- Given
  - $m$ periodic events
  - event $i$ occurs within period $P_i$ and requires $C_i$ seconds

- Then the load can only be handled if
  $$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

- Can we process 3 video streams, 25 fps, each frame require 10 ms CPU time?
  - $3 \times (10\text{ms}/40\text{ms}) = 3 \times 25 \times 0.010 = 0.75 < 1 \rightarrow \text{YES}$
Earliest Deadline First (EDF)

- Preemptive scheduling based on dynamic task priorities

- Task with closest deadline has highest priority
  \( \rightarrow \) priorities vary with time

- Dispatcher selects the highest priority task

- Assumptions:
  - requests for all tasks with deadlines are periodic
  - the deadline of a task is equal to the end on its period (starting of next)
  - independent tasks (no precedence)
  - run-time for each task is known and constant
  - context switches can be ignored

---

Earliest Deadline First (EDF)

- Example:
Rate Monotonic (RM) Scheduling

- Classic algorithm for hard real-time systems with one CPU
- Pre-emptive scheduling based on static task priorities
- Optimal: no other algorithms with static task priorities can schedule tasks that cannot be scheduled by RM

- Assumptions:
  - requests for all tasks with deadlines are periodic
  - the deadline of a task is equal to the end on its period (starting of next)
  - independent tasks (no precedence)
  - run-time for each task is known and constant
  - context switches can be ignored
  - any non-periodic task has no deadline

Rate Monotonic (RM) Scheduling

- Process priority based on task periods
  - task with shortest period gets highest static priority
  - task with longest period gets lowest static priority
  - dispatcher always selects task requests with highest priority

Example:

Task 1

Task 2

Dispatching

\[ P_1 < P_2 \implies P_1 \text{ highest priority} \]
EDF Versus RM – I

- Task A
- Task B
- Rate monotonic
- Earliest deadline first
- Deadline miss

**EDF Versus RM – II**

- **EDF**
  - dynamic priorities changing in time
  - overhead in priority switching
  - QoS calculation – maximal throughput:
    \[ \sum_{\text{all streams } i} R_i \times P_i \leq 1, \quad R \text{ – rate, } P \text{ – processing time} \]

- **RM**
  - static priorities based on periods
  - may map priority onto fixed OS priorities (like Linux)
  - QoS calculation:
    \[ \sum_{\text{all streams } i} R_i \times P_i \leq \ln(2), \quad R \text{ – rate, } P \text{ – processing time} \]
Summary

- Scheduling performance criteria and goals are dependent on environment.

- There exists several different algorithms targeted for various systems.

- Traditional OSes like Windows, UniX, Linux, ... usually use a priority-based algorithm.

- The right time slice can improve overall utilization.