CPU Scheduling

Knut Omang
IT/Paradial
17 Sep, 2009
(with slides from Thomas Plagemann,
Otto J. Anshus, Kai Li, RII Halvorsen
and Andrew S. Tanenbaum)

Today

- 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:

  (a) Long CPU burst
  - Short CPU burst
  - Waiting for I/O

  (b) Bursts of CPU usage alternate with periods of I/O wait
  - a CPU-bound process
  - an I/O bound process

Scheduling Algorithm Goals

- 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
  - Throughput - maximize jobs per hour
  - Turnaround time - minimize time between submission and termination
  - CPU utilization - keep the CPU busy all the time
- Interactive systems
  - Response time - respond to requests quickly
  - Proportionality - meet users’ expectations
- Real-time systems
  - Meeting deadlines - avoid losing data
  - Predictability - avoid quality degradation in multimedia systems

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
  - = time processing + time waiting
  - = sum of all waiting times (memory, I/O, etc)
  - How fast a single job gets through
- Response time
  - = time waiting + time processing
  - Having low variance in Response Time is good (predictability)
  - Short response time: type on a keyboard, click on GUI
- Waiting time
  - in the Ready Queue, for memory, for I/O, etc.
- Fairness
  - no starvation

Non-Preemptive: FIFO (FCFS) Policy

- Run
  - 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>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TT_{total} = (24+27+30)\times3 = 105

Arrival order: 1 - 2 - 3

<table>
<thead>
<tr>
<th>Arrival order: 2 - 3 - 1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TT_{total} = (3+6+30)\times3 = 123

Arrival order: 2 - 3 - 1
How well will FCFS handle:
* Many processes doing I/O arrives
  * One CPU-bound process arrives

**Discussion topic FCFS**

- All the I/O-bound processes execute their I/O instructions.
- All I/O-bound processes enter the back of the Ready Queue.
- CPU-bound process starts executing.
- CPU-bound does I/O

**CONVOY effect**
- Low CPU utilization
- Low device utilization

---

**FIFO vs. Round Robin**

- 10 jobs and each takes 100 seconds
  - 10 seconds of this is I/O wait

**FIFO**

**Round Robin**
- time slice 1s and no overhead

**Comparisons**

---

**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

**Problems**
- 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

---

**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

---

**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 (>100ms)
  - I/O bound: benefits from having shorter time slices (<10ms)

**[But what about memory bound?]**

---

**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, low average turnaround
  (Running a short job before a long decreases the waiting time MORE for this short than it increases for the long!)
- Memory/cache benefits

**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
  - Different priorities according to importance
- Advantage
  - Users can hit keyboard frequently
  - Starvation (should use dynamic priorities)
- Disadvantage
  - FCFS (all equal priorities, non-preemptive)
  - STCF/SRTCF (the shortest jobs are assigned the highest priority)

**Multiple Queues**

- 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, opened 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 slice), drop one level
      - If timeout doesn’t expire, stay or pushup one level
    - Can use different scheduling per queue
    - A job 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 queues
- One process in memory at a time (high switch costs)
- Large slices vs. response time → priority classes
  - Each time the quantum was used, the process dropped one priority class (larger slice, less frequent)
  - Interaction → back to highest priority class
  - Short, interactive should run more often
  - Proved viability of time sharing

**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, round robin)
  - Each second the priorities are recalculated:
    - Priority = CPU_usage (average #ticks)
    - Nice (+ 20)
    - Base (priority of last corresponding kernel process)

**Scheduling in UNIX (4.4BSD)**

- Similar to last slide
- Time slices of 100 ms
- Priorities is updated every 4th tick (40 ms)
  - `p_usspul = USER + [p_estcpu x ¼] + 2 x p_nice`
    - USER defaults to 50 (min), may be changed but here one uses only values between 50 and 127
    - `p_estcpu =`
      - running process: `[(2 x load)/(2 x load + 1)] x p_estcpu + p_nice`
      - blocked process: `[(2 x load)/(2 x load + 1)] x p_estcpu`
    - `p_nice` defaults to 0

**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 – 10 user levels
    - Priority may change – thread priority = process priority ± 2
    - Uses much CPU cycles → drops
    - User interactions, I/O completion → increase
    - Idle/zero-page thread – 1 system level
    - Runs whenever there are no other processes to run
    - Clear’s memory pages for memory manager
**Scheduling in Linux**

- Linux 2.4.x: Threads = processes
- Linux 2.6.x: thread scheduling

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

**SCHED_RR**
- each process has quantum
- timeslices of 10 ms (quantums)

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

Threads with highest nice are selected first:
- nice (FIFO):
  - good process = nice + priority
  - timeslicing (OTHERS): nice = quantum > 0 ? quantum + priority : 0
  - Quants are reset when no ready
- process has quantum left

---

**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**

- 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 require processing)
  - e = processing time
  - d = deadline
  - p = period
  - r = rate (r = 1/p)
  - 0 <= e <= p
  - (often d = p: we’ll use d = p – end of period, but Xd <= Xp is enough)
  - the kth processing of the task
  - the scheduling algorithm must account for these properties

---

**Real-Time Scheduling**

- Given
  - m periodic events
  - event i occurs within period p, 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 → 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

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

EDF Versus RM – I
- EDF:
  - dynamic priorities changing in time
  - overhead in priority switching
  - QoS calculation – maximal throughput: \( \sum_{p \leq 1} R_p \leq 1 \)
- RM:
  - static priorities based on periods
  - may map priority onto fixed OS priorities (like Linux)
  - QoS calculation:
    - \( \sum_{p \leq \ln(2)} R_p \leq 1 \)

EDF Versus RM – II
- EDF:
  - dynamic priorities changing in time
  - overhead in priority switching
  - QoS calculation – maximal throughput: \( \sum_{p \leq 1} R_p \leq 1 \)
- RM:
  - static priorities based on periods
  - may map priority onto fixed OS priorities (like Linux)
  - QoS calculation:
    - \( \sum_{p \leq \ln(2)} R_p \leq 1 \)
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 uses a priority-based algorithm
- The right time slice can improve overall utilization