CS 423
Operating System Design: Scheduling Periodic Tasks In Embedded Systems

Professor Adam Bates
Spring 2017
Goals for Today

- **Learning Objective:**
  - Conclude discussion inner workings of modern OS schedulers
  - Explore real-time scheduling for embedded systems

- **Announcements, etc:**
  - MP1 is out! Due Feb 20
  - Midterm Exam — Wednesday March 6th (in-class)
  - Updates to C4 reading lists; should be locked-in for the rest of the semester now.

**Reminder**: Please put away devices at the start of class
Example

- Three tasks A, B, C accumulate virtual time at a rate of 1, 2, and 3, respectively.
- What is the expected share of the CPU that each gets?

Strategy: **How many quantums required for all clocks to be equal?**

- Least common multiple is 6
- To reach VT=6...
  - A is scheduled 6 times
  - B is scheduled 3 times
  - C is scheduled 2 times.
- $6 + 3 + 2 = 11$
- A => 6/11 of CPU time
- B => 3/11 of CPU time
- C => 2/11 of CPU time

Q01: A => {A:1, B:0, C:0}
Q02: B => {A:1, B:2, C:0}
Q03: C => {A:1, B:2, C:3}
Q04: A => {A:2, B:2, C:3}
Q05: B => {A:2, B:4, C:3}
Q06: A => {A:3, B:4, C:3}
Q07: A => {A:4, B:4, C:3}
Q08: C => {A:4, B:4, C:6}
Q09: A => {A:5, B:4, C:6}
Q10: B => {A:5, B:6, C:6}
Q11: A => {A:6, B:6, C:6}
We’ve had lots of great (abstraction-violating) questions about how multiprocessor scheduling works in practice…

• To answer, consider **CPU Affinity** — scheduling a process to stay on the same CPU as long as possible
  
  • Benefits?

• Soft Affinity — Natural occurs through efficient scheduling
  
  • Present in $O(1)$ onward, absent in $O(N)$

• Hard Affinity — Explicit request to scheduler made through system calls (Linux 2.5+)
Multi-Processor Scheduling

• CPU affinity would seem to necessitate a multi-queue approach to scheduling… but how?

• **Asymmetric Multiprocessing (AMP):** One processor (e.g., CPU 0) handles all scheduling decisions and I/O processing, other processes execute only user code.

• **Symmetric Multiprocessing (SMP):** Each processor is self-scheduling. Could work with a single queue, but also works with private queues.
  
  • Potential problems?
SMP systems require load balancing to keep the workload evenly distributed across all processors.

Two general approaches:

- **Push Migration**: Task routinely checks the load on each processor and redistributes tasks between processors if imbalance is detected.

- **Pull Migration**: Idle processor can actively pull waiting tasks from a busy processor.
Other scheduling policies

- What if you want to maximize throughput?
Other scheduling policies

- What if you want to maximize throughput?
  - Shortest job first!
Other scheduling policies

- What if you want to maximize throughput?
  - Shortest job first!
- What if you want to meet all deadlines?
Other scheduling policies

- What if you want to maximize throughput?
  - Shortest job first!

- What if you want to meet all deadlines?
  - Earliest deadline first!
  - Problem?
Other scheduling policies

- What if you want to maximize throughput?
  - Shortest job first!

- What if you want to meet all deadlines?
  - Earliest deadline first!
  - Problem?
  - Works only if you are not “overloaded”. If the total amount of work is more than capacity, a domino effect occurs as you always choose the task with the nearest deadline (that you have the least chance of finishing by the deadline), so you may miss a lot of deadlines!
Problem:
- It is Monday. You have a homework due tomorrow (Tuesday), a homework due Wednesday, and a homework due Thursday
- It takes on average 1.5 days to finish a homework.

Question: What is your best (scheduling) policy?
Problem:
- It is Monday. You have a homework due tomorrow (Tuesday), a homework due Wednesday, and a homework due Thursday
- It takes on average 1.5 days to finish a homework.

Question: What is your best (scheduling) policy?
- You could instead skip tomorrow’s homework and work on the next two, finishing them by their deadlines
- Note that EDF is bad: It always forces you to work on the next deadline, but you have only one day between deadlines which is not enough to finish a 1.5 day homework – you might not complete any of the three homeworks!
• Consider a control system in a autonomous vehicle
  • Steering wheel sampled every 10 ms – wheel positions adjusted accordingly (computing the adjustment takes 4.5 ms of CPU time)
  • Breaks sampled every 4 ms – break pads adjusted accordingly (computing the adjustment takes 2ms of CPU time)
  • Velocity is sampled every 15 ms – acceleration is adjusted accordingly (computing the adjustment takes 0.45 ms)
  • For safe operation, adjustments must always be computed before the next sample is taken

• How to assign priorities?
Find a schedule that makes sure all task invocations meet their deadlines

- Steering wheel task (4.5 ms every 10 ms)
- Breaks task (2 ms every 4 ms)
- Velocity control task (0.45 ms every 15 ms)
Sanity check: Is the processor over-utilized?

- E.G.: If you have 5 homeworks due this time tomorrow, each takes 6 hours, then you are over utilized (5x6 = 30 > 24).

**Steering wheel task (4.5 ms every 10 ms)**

**Breaks task (2 ms every 4 ms)**

**Velocity control task (0.45 ms every 15 ms)**
Sanity check: Is the processor over-utilized?

- E.G.: If you have 5 homeworks due this time tomorrow, each takes 6 hours, then you are over utilized ($5 \times 6 = 30 > 24$).

Drive-By-Wire Example

Steering wheel task (4.5 ms every 10 ms)
- Breaks task (2 ms every 4 ms)
- Velocity control task (0.45 ms every 15 ms)

45% + 50% + 03% = 98%
Task Scheduling

How do we assign task priorities? (SCHED_FIFO)

- Steering wheel task (4.5 ms every 10 ms)
- Breaks task (2 ms every 4 ms)
- Velocity control task (0.45 ms every 15 ms)
Task Scheduling

How do we assign task priorities? (SCHED_FIFO)

- **Rate Monotonic** (large rate = higher priority)

  Breaks task (2 ms every 4 ms)

  Steering wheel task (4.5 ms every 10 ms)

  Velocity control task (0.45 ms every 15 ms)

Intuition: Urgent tasks should be higher in priority
How do we assign task priorities? (SCHED_FIFO)

- *Rate Monotonic* (large rate = higher priority)

  Breaks task (2 ms every 4 ms)

  Steering wheel task (4.5 ms every 10 ms)

  Velocity control task (0.45 ms every 15 ms)

Intuition: Urgent tasks should be higher in priority

*Is there a problem here??*
Task Scheduling

- Deadlines are missed!
- Average Utilization < 100%

- Breaks task (2 ms every 4 ms)
- Steering wheel task (4.5 ms every 10 ms)
- Velocity control task (0.45 ms every 15 ms)
Task Scheduling

- Deadlines are missed!
- Average Utilization < 100%

Breaks task (2 ms every 4 ms)
Steering wheel task (4.5 ms every 10 ms)
Velocity control task (0.45 ms every 15 ms)

Fix:
Give this task invocation a lower priority (EDF)
- Deadlines are missed!
- Average Utilization < 100%

Fix: Give this task invocation a lower priority (EDF)
Task Scheduling

- **Static versus Dynamic priorities?**
  - Static: Instances of the same task have the same priority
  - Dynamic: Instances of same task may have different priorities

Intuition: Dynamic priorities offer the designer more flexibility and hence are more capable to meet deadlines.
Re: Real Time Scheduling of Periodic Tasks...

- Result #1: Earliest Deadline First (EDF) is the optimal dynamic priority scheduling policy for independent periodic tasks (meets the most deadlines of all dynamic priority scheduling policies)

- Result #2: Rate Monotonic Scheduling (RM) is the optimal static priority scheduling policy for independent periodic tasks (meets the most deadlines of all static priority scheduling policies)
Re: Real Time Scheduling of Periodic Tasks...

- **Result #1:** Earliest Deadline First (EDF) is the optimal dynamic priority scheduling policy for independent periodic tasks (meets the most deadlines of all dynamic priority scheduling policies)

- **Result #2:** Rate Monotonic Scheduling (RM) is the optimal static priority scheduling policy for independent periodic tasks (meets the most deadlines of all static priority scheduling policies)
Locking vs. Priority

- What if a higher-priority process needs a resource locked by a lower-priority process?
  - How long will the higher priority process have to wait for lower-priority execution?
Locks and priorities may be at odds. Locking results in priority inversion.

- **High-priority task**
- **Low-priority task**

Diagram:
- Preempt.
- Lock S
Locks and priorities may be at odds. Locking results in priority inversion.

Priority Inversion

Attempt to lock S results in blocking

Preempt.
Priority Inversion

How should we account for priority inversion?

High-priority task

Low-priority task

Attempt to lock S results in blocking

Preempt.

Lock S

Priority Inversion

Unlock S

Lock S

Unlock S
Consider the case below: a series of intermediate priority tasks is delaying a higher-priority one.

Unbounded Priority Inversion

High-priority task

Preempt.

Intermediate-priority tasks

Lock S

Low-priority task

Attempt to lock S results in blocking

Preempt.

Unbounded Priority Inversion
Consider the case below: a series of intermediate priority tasks is delaying a higher-priority one.

How can we prevent unbounded priority inversion?
Solution: Let a task inherit the priority of any higher-priority task it is blocking.

Priority Inheritance Protocol

- High-priority task
  - Lock S
  - Attempt to lock S results in blocking
  - Unlock S

- Intermediate-priority tasks
  - Lock S
  - Preempt.

- Low-priority task
  - Lock S
  - Unlock S
Priority Inheritance Protocol:
What is the longest period of time a high priority task will wait on a resource?

Two priority inversion scenarios to consider:
(a) Lower priority task holds a resource I need (direct blocking)
(b) Lower priority task inherits a higher priority than me because it holds a resource the higher-priority task needs (push-through blocking)
Priority Ceiling Protocol

- Definition: The priority ceiling of a semaphore is the highest priority of any task that can lock it.

- A task that requests a lock $R_k$ is denied if its priority is not higher than the highest priority ceiling of all semaphores currently locked by other tasks (say it belongs to semaphore $R_h$).
  - The task is said to be blocked by the task holding lock $R_h$.

- A task inherits the priority of the top higher-priority task it is blocking.
Priority Ceiling Protocol:

Need Blue but priority is lower than Red ceiling

Need Yellow but priority is lower than Red ceiling

Needs Red, waits for 1 critical section to complete.

Done
Compare to the Priority Inheritance Protocol!!

Need Red

Need Blue

Need Yellow