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

Professor Adam Bates
Spring 2017
Goals for Today

• **Learning Objective:**
  • Conclude discussion of scheduling by exploring real-time scheduling in embedded systems.

• Announcements, etc:
  • 4 Credit Project Description was released.
  • MP1 Due 2/20 23:59 UTC-11

**Reminder**: Please put away devices at the start of class
What happens when the system can’t create new log entries?
A: Among other things, authentication fails.

Remedy: Symlink /var/log to a larger partition

```
mkdir /srv/log
rsync -auv /var/log/ /srv/log/
rm -rf /var/log
ln -s /srv/log /var/log
reboot

Monitor size of /srv with df -h
```
Consider a control system in a drive-by-wire 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 2 ms 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 (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)

45% + 50% + 03% = 98%
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)
How do we assign task priorities? (SCHED_FIFO)

- Rate Monotonic (large rate = higher priority)

Intuition: Urgent tasks should be higher in priority
How to assign task priorities?
- Rate Monotonic (large rate = higher priority)

Intuition: Urgent tasks should be higher in priority

Is there a problem here??
- 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)
- 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
- 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)
Semaphores in POSIX

- int sem_init(sem_t *sem, int pshared, unsigned value);
- int sem_destroy(sem_t *sem);
- int sem_trywait(sem_t *sem);
- int sem_wait(sem_t *sem);
- int sem_post(sem_t *sem);
Mutex (Lock)

- Simplest and most efficient thread synchronization mechanism
- A special variable that can be either in
  - **locked state**: some thread holds or owns the mutex; or
  - **unlocked state**: no thread holds the mutex
- When several threads compete for a mutex, the losers block at that call
  - The mutex also has a queue of threads that are waiting to hold the mutex.
POSIX Mutex Functions

- int `pthread_mutex_init`(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);
- int `pthread_mutex_destroy`(pthread_mutex_t *mutex);
- int `pthread_mutex_lock`(pthread_mutex_t *mutex);
- int `pthread_mutex_trylock`(pthread_mutex_t *mutex);
- int `pthread_mutex_unlock`(pthread_mutex_t *mutex);
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.

![Diagram showing priority inversion]
Locks and priorities may be at odds. Locking results in priority inversion.

Diagram:
- High-priority task
  - Lock S
  - Attempt to lock S results in blocking
- Low-priority task

Preempt.

Priority Inversion
How should we account for priority inversion?

Priority Inversion

High-priority task

Low-priority task

Attempt to lock S results in blocking

Preempt.

Lock S

Unlock S

Unlock S

Priority Inversion

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

**Unbounded Priority Inversion**

Attempt to lock S results in blocking

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

Priority Inheritance Protocol:
Consider the instant when a high-priority task that arrives.
- What is the most it can wait for lower priority ones?

If I am a task, priority inversion occurs when
(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)
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
Least Slack Time (LST)

- Idea: There’s no point in completing a task earlier than its deadline. Other tasks many be executed first
- LST: Orders queue with nondecreasing slack times
- Pros: Can run online, might improve response times
- Cons: Needs computing times, only works for preemptive tasks, difficult implementation

Diagram:

```
  A
  |   B
```
Goals for Today

• **Learning Objective:**
  • Conclude discussion of scheduling by exploring real-time scheduling in embedded systems.

• Announcements, etc:
  • 4 Credit Project Description was released.
  • MP1 Due 2/20 23:59 UTC-11