EDA040: Concurrent Programming

Michael Noukhovitch

Fall 2015, Lund University

Notes written from Klas Nilsson's lectures.

Contents

1 Introduction

1.1 Concurrency

activity entity performing actions

process entity performing instructions with own resources

job sequential instructions to be performed by an activity

task a set of jobs being performed by some process

thread sequential activity performing instructions

execution thread the thread itself accessed via the `Thread` interface

1.2 Mutual Exclusion

requirements:

- mutual exclusion
- no deadlock
- no starvation
- efficiency

2 Semaphores

semaphore simple counting interface for concurrency

2.1 Mutex

used to lock and unlock critical sections

```
MutexSem mutex;
mutex.take()
// critical section
mutex.give()
```

2.2 Signaling

calls used to block or unblock a thread

```
CountingSem mutex = new CountingSem();
// thread A
*
mutex.take() // block this thread
*
// thread B
*
mutex.give() // unblock thread A
*
```

2.3 Other Types

- blocked-set: arbitrary thread `take`s
- blocked-queue: `take` in FIFO order
- blocked-priority: highest priority `take`
- binary semaphore: efficient mutex implementation in RTOS
- multi-step semaphore: reserve several resources at once

3 Monitors

3.1 Introduction

As opposed to using `take/give` throughout the program, we instead can limit our mutual exlcusions to specific function.
Monitor: interface for mutually exclusive access to a function, in java using `synchronized`

- `wait` stateless wait for signal
- `notify` notify first (or highest prio) waiting task
- `notifyAll` notify all waiting task

3.2 Rules

- don't mix a thread and monitor
- all public methods should be synchronized
- wrap thread-unsafe classes by monitor
- don't use (spread-out) synchronized blocks

4 Deadlock

4.1 Introduction

deadlock a circular chain of tasks trying to allocate resources

starvation when a task is never prioritized to execute

livelock a running circular chain that is unable to allocate resources

Deadlock **detection** is not feasible as a resolution for real-time applications so we will looks at **prevention** which deals with eliminating one of the conditions for deadlock:

- mutual exclusion
- hold and wait
- no preemption
- circular wait

4.2 Dining Philosophers

Five dining philosophers with five forks between them but each needs two to eat, proves to be a deadlock-able situation if they circularly pick up one fork. We can solve it with:

- One left-handed philosopher that picks up left fork first (not circular)
- Only allowing four philosophers into the room at a time (monitor)
- Philosophers picking up both forks or neither (using a `Multistep Sem`, starvation possible)

5 Message-based Synchronization

5.1 Mailboxes

Message-based communication is useful for:

- producer-consumer relations
- signaling (one thread never waits)
- information transfer (in data of message)
- buffering
- distributed concurrency
- encapsulation (concurrent object properties)

5.2 Unbounded mailbox

Use copy-on-send to create limitless mailboxes

+ flexible code
+ no need to assume shared memory
+ thread safety, message not accessible by sender
- can run out of memory, increased memory use
- unpractical when immediate response is required
- recycling via message pools is difficult

5.3 Implementation

We will use `java.util.EventObject` as our message and `RTEvent` for async communication because it includes a timestamp. As such we will use `RTEventBuffer` as our circular mailbox

6 Scheduling

6.1 Timing

We want to perform something periodically:

CyclicThread no specific period

PeriodicThread specific periodic

SporadicThread minimum period

We are also interested in maintaining strict deadlines, we define some terms:

latency time between release time and start time

execution time time from start until finish

response time latency + execution time

6.2 Metrics

Use **WCET** (worst case execution time) or **R** (worst case response time)

- high priority
  - max latency: time to context switch
  - max execution: maximum blocking time + execution time
- low priority
  - max latency: WCET for all higher and equal priority threads
  - max execution: WCET for all higher priorities (preemption) and max blocking time

6.3 Static Scheduling

Time is divided into short slots and all activities are scheduled in advance, with cyclical execution

+ guaranteed scheduling

+ easy to calculate WCRT

- fragmentation and lost CPU time

- complex schedule must be redone when program changes

6.4 Dynamic Scheduling

6.4.1 Rate Monotonic Scheduling

Priority-based scheduling with priority according to period. We assume an interrupt-driven fixed-priority scheduler:

- periodic threads
- no blocking
- deadline (D) = period (T)
- instantaneous context switch

Generally possible to guarantee schedulability if:

$$\sum \frac{C_i}{T_i} < n(2^{1/n} - 1)$$

$$\begin{aligned} \text{where } n =&\ \text{number of threads} \\ C_i =&\ \text{execution time} \\ T_i =&\ \text{period} \end{aligned}$$

But it is sometimes possible to have schedulability above the limit, to check this look at what happens during the **critical instant**, when all threads are released at the same time, if they meet their first deadline, they will meet all subsequent ones.
We can calculate the worst case response time, R, of each thread:

$$R_i^{k+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_j$$

$$\begin{aligned} \text{where } hp(i) =&\ \text{set of threads with higher priority than i} \\ \left\lceil \frac{R_i^k}{T_j} \right\rceil =&\ \text{number of times i is preempted by j} \\ R_i^0 =&\ 0 \end{aligned}$$

The priorities of threads are decided by shortest period (shortest period = highest priority)
If we factor in blocking, we add B_i to our response time, which is the sum of:

normal blocking waiting for another thread to release a resource

push-through blocking blocking resolved by priority inheritance

ceiling blocking blocking because of priority ceiling

6.4.2 Generalized RMS

We relax some of the assumptions from RMS to generalize it:

- non-periodic threads (more optimistic)
- blocking (on shared resources with priority inheritance)
- shorter deadlines (D < T)
- non-instantaneous context switch (clock interrupts, etc)

6.4.3 Deadline Monotonic Scheduling

Usually $D < T$, so we change the priorities of threads to be in order of earliest deadlines (earlies deadline = highest priority)

6.4.4 Earliest Deadline First

Assign CPU to thread with the earliest deadline

- + 100% CPU usage possible
- - expensive to implement
- - misses all deadlines at overload

6.4.5 Run-Time Overhead

Real systems include:

release jitter variations in time to release a thread

context switch time to switch to another thread

clock interrupts periodic interrupts driving preemption and context switches

We can get realistic measurements either through actual measurement, taking the worst case runtimes, or through formal analysis, but new ideas like pipelining and cacheing make this difficult

6.5 Blocking

If a lower priority thread holds a resource needed for a higher priority thread it can end up blocking it (against priority protocol), this is called **priority inversion**. We can solve this using a **priotiy inheritance protocol**:

6.5.1 Basic Priority Inheritance

Temporarily raise the priority of low blocking thread to release the resource, locally for each resource
Blocking time for thread i is at most the minimum of the number of lower priority tasks that can block i and the number of semaphores that can be used to block i

6.5.2 Priority Ceiling

Allow only one low priority thread to access resources of high priority thread at any time, this allows us to avoid mulitple blockings for highest proirity thread at the expense of average blocking times and runtime overhead
restrictions on `lock` and `unlock`:

- a task must release all resources between invocation
- computation time for task i while holding semaphore s is bounded to the length of the critical section for s

- a task may only lock semaphores from a fixed set known beforehand

If `PC(s)` is the highest priority of a task that may lock semaphore s, and `pri(i)` is the priority of task i then at runtime:

- if task i wants to lock s, it can only do so if `pri(i) > PC(s)`
- else, i is blocked and the task currently holding s inherits the priority of i

This means that our code is deadlock free and blocking time for a task is at most the critical section of *one* lower priority task

6.5.3 Immediate Inheritance

Always raise the priority of a thread to the ceiling of the lock during its critical region

- same worst case time as priority ceiling
- easy to implement
- no need for block queues on single-processor

6.6 Sporadic Threads

A thread triggered at unpredictable points in time
We can pessimisticall model as a periodic thread with period equal to minimum time between events or take a more realistic approach with a **sporadic server**, a periodic thread that handles all sporadic jobs

- apply standard schedulability test
- does not assume a minimum time between events
- cannot guarantee that deadlines are met for sporadic events

7 Real-Time Java

7.1 Language Safety

With real-time development we want to manage:

- system complexity
- system development

For complexity, we use a multi-stage deploy process but unsafe languages can still hinder development process with hard to find errors:

- manual memory management
- type casting
- pointer arithmetic

- arrays without bound checking

Safe languages help make code correctness easier to guarantee:

- compiler checks
- automatic memory management
- errors leading to error messages

7.2 Garbage Collection

7.2.1 Background

Manual memory management can lead to problems with

- dangling pointers
- memory leaks
- fragmentation

7.3 Reference Counting

Count the number of references to each object and deallocate when counter reaches 0

+ easy to implement, short pause

- expensive (overhead)

- no compaction

- doesn't detect circular structures

7.4 Traversing Algorithms

Periodically traverse pointer graph from root pointers outside heap, and delete objects not encountered in traversal
Compacting can also be done simultaneously:

Mark-compact determine where objects will be after compaction, update pointers, and slide objects into new position

Copying alternate between using two subheaps, when current subheap is full, copy over all live objects into other empty subheap, compacting in the process

7.4.1 Generation-based GC

Partition into several "generations" that are garbage collected separately

- young generations need to be checked most often, more efficient
- complex to keep track of inter-generation pointers

7.4.2 Conservative GC

When we don't know runtime type information, we have to regard everything as a pointer, but we can't modify pointers which means no compaction

7.4.3 Incremental GC

"Stop-the-world" pauses to do garbage collection, needs to be an incremental variant of a GC algorithm (can be easy as reference counting is incremental by nature)

7.4.4 Real-Time GC

Real-time adds another layer of complexity because we have soft and hard deadlines to meet and the GC can't interfere with them

- small number of periodic threads with high priority (hard requirements)
- large number of low priority threads (soft requirements)

Requirements:

- minimal response time and latency for high priority
- predictable (and low) worst case response time
- guaranteed schedulability

Idea:

- perform GC work in pauses between high priority executions
- low priority threads should do their own incremental GC
- interruptible GC, minimum locking