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