Concurrency

CS 463

Outline

Levels of concurrency

Locations of levels of concurrency

- machine instruction
- (high-level) language statement
- unit level
- program level

Only statement- and unit-level are interesting for us.

Categories/Terminology

- **physical concurrency:** multiple independent processors
- **logical concurrency:** time-sharing one processor to simulate physical concurrency
- **Coroutines:** (quasi-concurrency) have a single thread of control
  - but method calls/returns aren't strictly nested: also can yield/resume

Tasks

task: some unit of code that can run concurrently with others.

- usually work together (if not: "disjointed" task)
- program can start task without having to pause itself.
- task's completion doesn't always return to caller

Kinds of Tasks:

- heavy-weight: has its own address space (like separate processes)
- light-weight: shared address space.
  - easier to implement
  - easier to share (whether on purpose or not...)

Task Synchronization

Can communicate through:

- shared state
- parameters
- message passing

Cooperation: tasks each help coordinate sharing of resources or timing of execution

- awaiting completed results

Competition: tasks fight for resources.

- mutually exclusive access
- Example: consider two read-modify-write tasks on same memory.

Task States

- new: created, hasn't started.
- ready: able to run but currently not.
- running: currently executing.
- blocked: can't run just now.
- dead: can't run any more (whether it finished its work or not).

Scheduler: handles waiting, notifying, etc. between tasks

Liveness, Deadlock

liveness: task's ability to make progress towards completion

- tasks can lose their liveness.

deadlock: all tasks have lost their liveness.

- example: each waiting on each other to do something first.

Various Approaches

- **semaphores** (signals to other tasks)
- **mutexes** (“locks” to limit access to resources)
- **monitors**
  - hide shared data in monitor instead of direct-access sharing
- **message passing** (“mailboxes” between tasks)

Semaphores

Semaphore (Dijkstra, 1965)

data structure that provides controlled access to a shared resource.

- tasks **wait** for access and **release** access when done.
- can implement with counter and queue

- Uses
  - competition (who manages to stop waiting first?)
  - cooperation (releasing acts as a yield/cooperation)

Producer/Consumer example (cooperation)

- Shared buffer: values can be inserted or removed, as space allows.
  - might implement as array and first/last pointers
- Producer: generate values, put in buffer
  - must wait if there's no room in buffer at the moment
- Consumer: takes values from buffer
  - must wait if there are no values in buffer at the moment

Example Buffer: Producer/Consumer

definition:

```
class Buffer {
   private int[] val; // array of values
   private int head, last; // indexes

   public void insertValue(v) {...}
   public int  takeValue  (v) {...}
```

- semaphore usage:
  - two semapores - `emptySpots`, `fullSpots`
    - `insertValue` and `takeValue` increment/decrement them

Examples

→ see `ProduceConsume.java`

- includes Buffer class
- uses `synchronized` keyword

→ see `ProducerConsumer.hs`

- many versions inside

→ see: pc1.c, pc2.c, pc3.c

- variations on producer/consumer
- uses mutexes and condition variables
- uses int[] as buffer

Semaphore Implementation Basics

Semaphore pseudocode

```
class Semaphore {
    int counter;
    Queue waitingTasks;

    public void wait(Task t){...}
    public void release(Task t) {...}
}
```

How are wait and release implemented?

basics of wait(sem)

```
if sem.counter>0:
    counter -= 1
else:
    sem.enqueue(theCaller)
    wakeup_any_task()  // if we can't: deadlock
```

basics of release(sem)

```
if empty(sem.queue):
    sem.counter += 1
else:
    put caller in ready-queue
    activate (sem.queue.next())
```

Semaphore-based Producer/Consumer

Producer Pseudocode

```
loop:
  << generate value v >>
  wait(emptySpots)
  insertValue(v)         // the guarded action
  release(fullSpots)
```

Consumer Pseudocode

```
loop:
  wait(fullSpots)        // the guarded action
  v <- takeValue()
  release(emptySpots)
  << consume value >>
```

Semaphore issues

- Brittle
  - rely on producer/consumer code to correctly call `wait` / `release` on correct semaphores.
    - missing waits: underflow or overflow occurs
    - missing releases: deadlock occurs (nobody else is woken up)
  - semaphore implementation needs a single-instruction test-and-set to be successfully implemented (took us a long time to realize this!)
- language support
  - usually provided as libraries.
  - very similar: **mutex**. "Mutual Exclusion".
    - can lock/unlock to gain access to resource.
    - sort of like a semaphor where counter can't go above 1.

Monitors

Monitor: abstracts both shared resource (data) and operations that interact with it all into one place.

- mutual exclusion is thus a given.
- programmer still must coordinate between tasks that use the monitor.
- concurrent calls are implicitly blocked.

Java's `synchronized` methods act like monitors.

Monitor notes

Competition:

- straightforward with monitors (mutual exclusion is guaranteed)

Cooperation:

- programmer still does bookkeeping (e.g., # items in buffer now)
- comparison with semaphors:
  - Monitors are ‘better’ for competition
  - both struggle with cooperation
  - equally powerful: semaphors/monitors can implement each other.

Message Passing

- tasks don't interrupt each other; instead, they send messages to each other
    - like mailboxes between tasks
    - tasks can check their mailboxes when they want, and respond to messages or empty mailboxes as appropriate
- may be **synchronous** or **asynchronous**
- non-deterministic which messages arrive first in a mailbox.
    - consider multiple mailboxes to help tame ordering issues.
- `Erlang` is a functional language allowing concurrent 'processes' to send messages to each other.

Java and Synchronization

Java synchronized methods

- Threads and `synchronized` methods
  - Create `Thread` objects (or `Runnable` ones, same effect):
    - the tasks are the `run` methods of such objects.
    - these tasks can run concurrently
  - Java threads are light-weight: share address space; low overhead to create.
- other options
  - `java.util.concurrent.Semaphore` : a counting semaphore (counter, no queue)
  - `java.util.concurrent.atomic` : variable-level synchronization!
  - Explicit Locks: `Lock` interface (with `lock`, `unlock`, `tryLock` methods)

Using Java's synchronized keyword

- add `synchronized` modifier to any method: **bam!**, mutual exclusion is guaranteed here and all other `synchronized` method calls on the same object.
- the object itself acts as the monitor. All uses of the object must obtain the lock to call synchronized methods.
- static and synchronized? Still useful: all class-members are grouped, and the monitor is the object `ClassName.class`.
- also: synchronized block:

```
synchronized (objectExpr){  stmts...}
```

- behaves like sync'd method of result of `objectExpr`

Java: cooperation vs competition using `synchronized`

Finally, we learn about all those "other" methods of `java.lang.Object` !

Cooperation:

`wait(..)` enter ready queue (voluntarily pause)

`notify()` wake up one arbitrary ready thread (maybe no specific enough!)

`notifyAll()` wake up all arbitrary threads (helps maintain liveness vs. `notify()`).

Competition:

`synchronized` achieves mutual exclusion.