Concurrency

CS 463 @ GMU

Outline

- Basic ideas
- Providing Synchronization
- Java and Synchronization

Levels of Concurrency

Locations of levels of concurrency

- Machine instructions
- (high level) language statements
- Unit level
- Program level

Only statement- and unit-level are where we will focus.

Categories/Terminology

- Physical concurrency: multiple independent processors
- Logical concurrency: time-sharing one processor to simulate physical concurrency
- Co-routines (quasi-concurrency): have a single thread of control
  - But method calls/returns aren't strictly nested: you can also yield/resume.

Tasks

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

- usually work together (if not: "disjointed" task)
- a 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: shares address space.
  - Easier to implement (faster)
  - Easier to share memory (whether on purpose or not!)

Task synchronization

Can communicate through:

- Shared state
- Parameters
- Message passing

Cooperation: tasks each help coordinate sharing of resources and timing of execution.

- Awaiting completed results, staying in sync with each other

Competition: tasks fight for resources or get in each others' way

- Mutually exclusive access to resources
- Example: consider two read-modify-write tasks using the same memory.

Task states

- New: created, hasn't started running.
- Ready: able to run but currently not.
- Running: currently executing.
- Blocked: can't run just now.
- Dead: can't run anymore (whether it finished its work or is stuck)

Scheduler: handles waiting, notifying, etc. between tasks.

Liveness, Deadlock

- Liveness: a task's ability to make progress towards completino
  - Tasks can lose their liveness.
- Deadlock: all tasks have lost their liveness.
  - Example: each task is waiting on another to do something first
  - It's like a stalemate.

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)

A data structure that provides controlled access to a shared resource.

- tasks **wait** for access and **release** access when done.

Uses

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

Producer-Consumer example (cooperation)

- Shared "buffer": values can be inserted or removed into a queue.
  - Often implemented as an array with first/last item pointers
- Producer: generates values to put in the buffer.
  - Must wait if there's no space in the buffer at the moment
- Consumer: takes values from the buffer
  - Must wait if there are no values in the buffer at the moment

Example Buffer (producer/consumer)

```
class Buffer {
    private int[] val;
    private in head, last;
    public void insertValue(int v){…}
    public int  take Value ()     {…}
}
```

Semaphore Usage:

- two semaphores: `emptySpots`, `fullSpots`
- `insertValue` and `takeValue` will increment/decrement the counters of how many spots are available
- We can add tasks to a queue of tasks who are waiting their turn.

Code Examples

⟶ see `ProduceConsume.java`

- includes a `Buffer` class
- uses Java's synchronized keyword

⟶ see `pc1.c`, `pc2.c`, `pc3.c`

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

(when we get to "round 2" of learning concurrency:)

⟶ see ProducerConsumer.hs

- *many versions inside*

Semaphore implementation sketch

```
class Semaphore {
    int counter;
    Queue waitingTasks;
    public void wait (Task t) {…}
    public void release (Task t) {…}
}
```

```
public void wait(Task t) {
    if (sem.counter>0) counter-=1;
    else {
        sem.enqueue(t);
        wakeup_any_task();
    }
}
```

```
public void release (Task t) {
    if (sem.queue.empty()) counter+=1;
    else {
        sem.queue.add(t);
        activate(sem.queue.next());
    }
}
```

Producer/Consumer sketch

```
//Producer
loop:
    <<generate value v>>
    wait(emptySpots)
    buffer.insertValue(v)
    release(fullSpots)
```

```
//Consumer
loop:
    wait(fullSpots)
    v = buffer.takeValue()
    release(emptySpots)
    << consume v >>
```

Semaphore Issues

- brittle code
  - relies on producer/consumer code to correctly call wait/release, on the correct semaphores
    - missing waits? dual access is likely
    - missing releases? deadlock quite likely (tasks don't get woken up)
  - semaphore implementation needs a single instruction "test-and-set"
    - took the computing field quite some time to realize this!
  - language support needs:
    - usually provided as libraries
    - very similar: mutex ("mutual exclusion")
      - can lock/unlock to gain access to a resource (task is blocked when resource is in use)
      - sort of like a semaphore where counter can't go above 1.

Monitors

- Monitor: abstracts away both the shared resource (data) and the operations that interact with it, all into one place.
    - very much the OO mentality
    - mutual exclusion is pretty much a given
    - programmer still must coordinate between tasks that use the monitor.
    - concurrent calls are implicitly blocked.

Java's synchronized methods act as monitors.

Monitor notes

- Competition
  - straightforward with monitors (mutual exclusion is guaranteed*)
- Cooperation
  - programmer still does bookkeeping, e.g. tracking #items in the buffer

- comparison with semaphores:
  - monitors are 'better' for competition
  - both struggle with cooperation
  - equally powerful: each can implement the 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 by synchronous or asynchronous
- non-deterministic which messages arrive first in a mailbox
  - consider multiple mailboxes to help tame ordering issues
- Erlang is a programming language that allows concurrent 'processes' to send messages to each other.
  - ("Ericsson Language" - like the telecomm company)

Java and synchronization

- Threads and synchronized methods
  - create `Thread` objects (or `Runnable` ones, same effect)
    - the tasks are the run methods of those objects.
    - We explicitly `start()` them , and once the method is done/returns, the task is complete.
    - these tasks can run concurrently
  - some other options
    - `java.util.concurrent.Semaphore`: a counting semaphore (counter, no queue)
    - `java.util.concurrent.atomic`: variable-level synchronization (protects assignments)
    - `java.util.concurrent.locks`: explicit locks. Lock interface has lock, unlock, tryLock methods.

Using Java's synchronized keyword

- add `synchronized` modifier to any method:
  - mutual exclusion is guaranteed here and all other synchronized method calls on the same instance (object)
  - the object itself acts as the monitor. All uses of the object must be given permission (the lock) to call synchronized methods.
  - static and synchronized? Still useful: all class-members are grouped, and the monitor is the object `ClassName.class`.
- synchronized block:
  - you can synchronize an arbitrary block of code. Give the monitor object and the code:

    ```
    synchronized (objectExpr) { stmts…}
    ```
  - behaves like a synchronized method of `objectExpr`'s class.

Java: cooperation vs competition using **`synchronized`**

- finally you get to see the rest of the `Object` class's methods!
- `wait(..)` enter a ready queue (voluntary pause)
- `notify(..)` wake up any ready thread (maybe not specific enough)
- `notifyAll(..)` wake up all arbitrary threads (helps maintain liveness)

- Competition:
  - `synchronized` achieves mutual exclusion between the synchronized blocks.