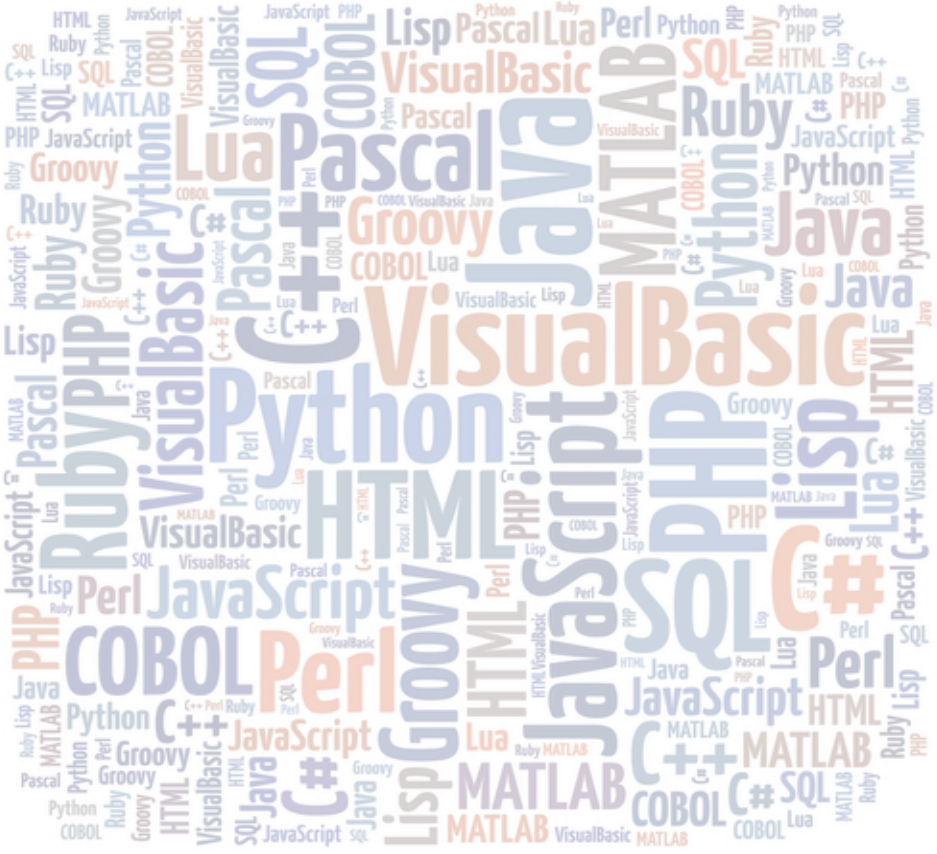
COMP 348: Principles of Programming Languages

Section 21: Concurrency and Erlang



Introduction

- Concurrency versus parallelism
- Race conditions and critical regions
- Locks
- Atomic instructions
- Deadlock and Livelock
- Message passing
- Asynchronous communication

Concurrency

- Concurrency refers to the concept of separate threads of control/execution within a program.
- Often, people use the terms *parallelism* and *concurrency* as synonyms.
- Strictly speaking, this is incorrect.
- Parallel programs are those that support simultaneous execution of distinct instruction streams.
  - By definition, they are run on multiple CPUs or cores.
  - Typically they are used to increase the performance of a specific algorithm
    - For example, we might implement a parallel sort, in which each thread computes part of the sorted result simultaneously.

Concurrency…cont'd

- Concurrent programs are those that provide distinct instruction streams that may be scheduled and run independently.
  - They do not require distinct CPUs and can be run on single core one at a time, giving the appearance that each is running.
  - Typically, they are used to divide the logic of a complex programs into distinct units
    - For example, one thread for GUI interaction and one thread for interfacing with a database and preparing results.
    - Or an application with a large number of tasks that require minimal communication with one another
- To summarize, all parallel programs provide concurrency, but concurrent programs are not necessarily parallel.

Concurrency issues

- It is not difficult to create a concurrent (or parallel) program.
- Languages provide functions to create and run new "threads".
- Unfortunately, it can be VERY difficult to write concurrent programs correctly.
- The primary problem is controlling access to shared data
  - If no write-able data is shared between the threads, then there are no problems.
    - Sharing read-only data is never a problem.
  - Of course, it can be difficult to do anything useful if none of the threads communicate or share data in any way.

Race conditions

- The central problem is associated with what we call “race conditions”
- Specifically, when two or more threads must access and potentially update a data structure, it is possible that the final state of the data will depend on a “race” to the data.
  - The thread winning the race may update the data first, though the thread losing the race is likely to make the last update.
  - In such cases, one of the threads will have an incorrect view of the data.

Critical regions

- The block of code in which an update to shared data is made is often referred to as the “critical region”.
- Clearly, me must ensure that only one thread at a time enters and exits the critical region.
- How do we ensure this?
- Traditionally, we do this with *locks*.
  - In its simplest form, a lock is a variable that can be set to open/closed (0/1).
  - So a lock is set at the start of the critical region and then released when this thread leaves the critical region.
  - When a second thread encounters a closed lock, it must wait or “block” until the lock has been released.

Locks…cont'd

- There is one small complication with this simple model.
- While setting a variable can be done with a single high level language instruction, this instruction is in fact compiled into multiple machine level instructions.
- Why is this a problem?
- When applications/threads are run, they are swapped on and off the CPU by the operating system.
  - This is how we give the appearance of many applications running simultaneously.
  - Typically, this happens when a hardware timer expires

Locks…cont'd

- The OS has no control over the exact machine level instruction that is currently executing (i.e., when the timer fires)
- So it is possible to swap a thread just after it reaches an open lock but just before it finalizes the action of updating the lock.
- When this happens, it is possible for multiple threads to think they have the lock.
  - Very BAD things happen at this point.

Locks…cont'd

- For this reason, all modern CPU instruction sets provide a special "test and swap" instruction.
  - This is an atomic assignment instruction that always completes.
- The operating systems provides a "wrapper" function for this instruction so that language libraries can provide reliable application locks.
- So when you use locks in your application code, this is really what you are getting.

So what's the problem?

- All of this works properly and has been the basis of multi-threaded code for many years.
- C (pthreads) and Java both provide support for this model
  - In Java, for example, a lock is built into the Class object.
  - When you use a "synchronized" method, it is this lock that is being tested to ensure that only one synchronized method can execute at a time.
- If used properly, this will function as expected.

But…

- Trivial use of locks is, well, trivial.
- But real applications may consist of many threads and many shared data structures and/or critical regions.
- This creates a couple of problems
  1. It can be very easy to **deadlock**.
     - In short, this occurs when thread A is waiting for a lock held by Thread B, while the thread B is waiting n a lock held by Thread A.
     - The application will hang forever in this state.
  2. Livelock is another possibility.
     - Threads contending for locks spend a great deal of time repeatedly releasing locks in order to make them available to one another, but not actually getting much actual work done.
- Even when things work properly, a lot of time can be spent setting, releasing, and most importantly waiting for locks
  - This can seriously impact performance so that much of the benefit of concurrency/parallelism is wasted.

Alternatives

- The bottom line is that “low level” thread management doesn’t scale very well
  - This is of great concern in a future world of high core counts.
- So a number of alternatives are provided by various languages.
- One of the most flexible is message passing.
  - This is the option provided by Erlang

Message Passing

- Message passing consists of a set of message queues associated with individual processes or tasks.
- Data is not shared directly. Instead, information is sent to and received from each process.
  - These messages are placed into a queue of messages that the process can examine.
  - Typically, the messages have a sender ID and perhaps a “tag” to identify the message type.

Asynchronous

- Messages can be viewed as *asynchronous*.
  - In other words, sending and receiving processes do not block during transfer, so sending and receiving are "de-coupled".
- Instead, the message passing system simply appends the message to the queue, which can be checked at any time.
- Note that it is still possible for processes to "hang".
  - You can, for example, explicitly wait for a message from a task that has crashed.
  - However, this is not a true deadlock as the programmer has the ability to address this (as we will see later).
  - It can't happen "by accident" based upon the arbitrary order in which locks were acquired.

Graphically speaking

- The illustration below depicts an application with three threads, each with its own asynchronous message queue
  - Each node consist of a sender ID and a data element.
- Again, note that no data structure is directly shared by any of the tasks