An Introduction to Programming with Threads

Andrew D. Birrell
January 6 1989

Presenter: Sara Dadizadeh
Andrew Birrell

• Got his PhD from Trinity College, Cambridge in 1978
• In 2001 he got a job with Microsoft Research
Original From 1989

An Introduction to Programming with Threads

Andrew D. Birrell

This paper provides an introduction to writing concurrent programs with “threads”. A threads facility allows you to write programs with multiple simultaneous points of execution, synchronizing through shared memory. The paper describes the basic thread and synchronization primitives, then for each primitive provides a tutorial on how to use it. The tutorial sections provide advice on the best ways to use the primitives, give warnings about what can go wrong and offer hints about how to avoid these pitfalls. The paper is aimed at experienced programmers who want to acquire practical expertise in writing concurrent programs.

CR categories and Subject Descriptors: D.1.4 [Programming Techniques]: Concurrent Programming; D.3.3 [Programming Languages]: Language Constructs—Concurrent programming structures; D.4.1 [Operating Systems]: Process Management

General Terms: Design, Languages, Performance

Additional Key Words and Phrases: Threads, Concurrency, Multi-processing, Synchronization
An Introduction to Programming with C# Threads

Andrew D. Birrell

[Revised May, 2005]

This paper provides an introduction to writing concurrent programs with “threads”. A threads facility allows you to write programs with multiple simultaneous points of execution, synchronizing through shared memory. The paper describes the basic thread and synchronization primitives, then for each primitive provides a tutorial on how to use it. The tutorial sections provide advice on the best ways to use the primitives, give warnings about what can go wrong and offer hints about how to avoid these pitfalls. The paper is aimed at experienced programmers who want to acquire practical expertise in writing concurrent programs. The programming language used is C#, but most of the tutorial applies equally well to other languages with thread support, such as Java.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming; D.3.3 [Programming Languages]: Language Constructs and Features—Concurrent programming structures; D.4.1 [Operating Systems]: Process Management

General Terms: Design, Languages, Performance

Additional Key Words and Phrases: Threads, Concurrency, Multi-processing, Synchronization
What is a thread?

• A single sequential flow of control
• Multiple threads execute simultaneously
Properties of threads

• Processes are expensive
• Threads are cheap
  – They execute within a “single address space”
• Threads have shared memory
  – Except for local variables, which are kept on each threads private stack
Why use concurrency?

• Utilize multi-processors
• Do useful work while waiting for slow devices like disks, networks, terminals
• Create responsive window systems
• Build distributed systems so servers can service many clients at a time
  – Instead of creating a process per client, which is expensive
Basic operations

- Thread creation
- Mutual exclusion
- Waiting for events
- Some arrangement for getting a thread out of an unwanted long-term wait
Alerts

• Mechanism to interrupt a thread
• Used to terminate a long running computation or long term wait
Alerts

• Mechanism to interrupt a thread
• Used to terminate a long running computation or long term wait

• Q: What function do alerts map to in a POSIX-thread library?
• Q: Can other primitives replace the effect of alert?
Cheating

• Maybe we can skip locking if shared variables are accessed atomically, but only if:
  – Your variable is word aligned
  – If you are running on a uni processor
  – If your compiler generates obvious instruction sequences and doesn’t slave variables into registers
Question about cheating

IF NOT initDone THEN
  LOCK m DO
    IF NOT initDone THEN
      Initialize();
      initDone := TRUE;
    END;
  END;
END;
Using conditions

- Use a while to check for conditions that wait for a thread

```plaintext
WHILE NOT expression DO
  Thread.Wait(m,c) END;
```
Using broadcast

• Signal wakes (at least) one thread
• Broadcast awakens all threads that are waiting
  – Sometimes, only one woken thread will get the mutex and continue
    • Performance loss vs. simplicity, don’t have to separate different wait reasons into different condition variables
Q: Broadcast’s efficiency

• What happens if 100 blocked threads are awakened (with 100 times context switch?) and only 1 thread grabs processor while other 99 have to wait again. That must be a heavy loss in performance. So is there any solution like ‘partial broadcast’?
Q: Broadcast’s efficiency

• What happens if 100 blocked threads are awakened (with 100 times context switch?) and only 1 thread grabs processor while other 99 have to wait again. That must be a heavy loss in performance. So is there any solution like ‘partial broadcast’?
  – If only one thread is going to execute, then signal instead of broadcast.
  – If only a subset need to be awoken, you can signal to the next thread from the previous thread, in a ring
Problems

- Deadlock
- Livelock
- Starvation
- Poor performance through lock conflicts
**Simple deadlock**

- Two threads and two mutexes

![Diagram of a simple deadlock]

- Thread A tries to lock Mutex 1 but it is already held by Thread B.
- Thread A is blocked.
- Thread B tries to lock Mutex 2 but it is already held by Thread A.
- Thread B is blocked.
- Both threads are blocked waiting for each other to release their respective mutexes.
Spurious wake-ups

• Happen when we awaken threads that cannot make useful progress
  – ex. if you call broadcast when signal is sufficient
Spurious wake-ups

• Happen when we awaken threads that cannot make useful progress
  – ex. if you call broadcast when signal is sufficient

• Solution: use multiple conditional variables
Spurious lock conflicts

LOCK m DO

  // some work
  IF expr THEN Thread.Signal(c) END;

END;
Spurious lock conflicts

LOCK m DO
  // some work
  IF expr THEN Thread.Signal(c) END;
END;

vs.

LOCK m DO
  // some work
  doSignal = expr
END;
  IF doSignal THEN Thread.Signal(c) END;
Starvation

• A kind of livelock where a thread never makes progress
• Can occur in our reader-writer example
Additional Techniques

• Up-calls

• Version stamps
  – Count # of times data is modified and make sure lower level is kept up to date

• Work crews
  – Create pool of threads and re-use them, save overhead of forking and joining
Conclusion

• Birrell concludes that given
  – A system with good primitives and suitable libraries
  – Basic caution and carefulness
  – Armory of useful techniques and
  – Knowledge of common pitfalls

• Writing concurrent programs is neither exotic or difficult
Q: Advantage of pipeline model

• In the pipelining model, what are the advantages? The time consumptions are the same between doing the three steps separately and doing them together, since the three steps are not independent and they need to lock the same mutex.
Q: Advantage of pipeline model

• In the pipelining model, what are the advantages? The time consumptions are the same between doing the three steps separately and doing them together, since the three steps are not independent and they need to lock the same mutex.
  
  – If you are streaming data between steps in real time, then each process can continuously take data from its input queue, process it, then put it on its output queue.
Q: Scalibility

- “By using a lightweight multi-threading facility, the programmer can utilize the processors cheaply. This seems to work well in systems having up to about 10 processors, rather than 1000 processors”
  - Why doesn’t this system scale to 1000 processors?
Q: Language support

• “It would be a good idea for your language system to give you some help [with correctness of locks], for example by syntax that prevents you accessing variables until you have locked the appropriate mutex. But most languages don’t offer this yet”
  – Do they now?
Q: Language support

• “It would be a good idea for your language system to give you some help [with correctness of locks], for example by syntax that prevents you accessing variables until you have locked the appropriate mutex. But most languages don’t offer this yet”
  – Do they now?
  – “…use static or dynamic analysis tools. For example, there are experimental tools that check at runtime which locks are held while accessing each variable, and that warn you if an inconsistent set of locks (or no lock at all) is used”
Q: Explicitly declare memory as shared

• Given that writing correct parallel code (without deadlocks, races, etc) is very difficult, why don’t we put all shared state in one specific well known location, and explicitly declare that memory as being shared using a special construct?
Q: Mapping between vars and mutexes

• The paper says that each condition variable should always be used together with the same mutex. What would happen if one variable is shared by multiple mutexes?
Q: Time slice threads to achieve priority

• To avoid the situation that some high priority thread never makes progress because of lock conflicts, Can we introduce time slice to each threads? When the time up, a certain thread must save its context and release the resources and pass cpu to higher priority threads? How about the time and space tradeoff of doing so?
Q: Releasing a mutex within a LOCK

• What is the rationale behind releasing a mutex inside the LOCK region? What is the advantage of using Acquire and Release over structuring the code so that the mutex is released, the block called and then the mutex re-acquired?

```
LOCK m DO
    First bit of work.....
END

SlowFunctionCall()

LOCK m DO
    Remaining work.....
END
```
• In the paper, the author suggests that the thread scheduler plays major role in avoiding deadlocks and other thread safety conflicts. Thus why can’t we have a one intelligent thread scheduler in the OS level so that each programmer does not have to worry too much. (but still the programmer has to do many things in order to make it work correctly)
• Is there any work in the compiler world on automatically generating signals to condition variables? That is, have a language where you can wait on an expression being true, which will be translated by the compiler into a condition variable. Every place where state is modified such that the waited condition might become true, the compiler automatically generates a call to signal().
• As a more speculative question, most of the points raised about threads and synchronization are still important today. However, recently things like the Grand Central Dispatcher in Snow Leopard are trying to get the developer to use higher level constructs like blocks while leaving the actual thread implementations and synchronization to the OS. With similar technologies, is it likely that thread programming may not be an issue for application and ring 3 developers in the near future?