

Learning to Program with Threads

Andrew D. Birrell

DEC Systems Research Center (1989)

presented by Kevin Loken

kloken@cs.ubc.ca

Concurrent Programming

- It's here to stay
 - Nearly all new computers are dual-core
- It's hard
 - Some algorithms are difficult to parallelize
 - Concurrency brings it's own set of problems
 - We generally don't have experience with it

Threading paradigms

- Boss-worker model
 - one master thread, sends work to subservient threads
 - work-crew model (limited number of threads that pick work off of an unbounded queue)

Threading paradigms

- Peer model

- a lot like boss-worker, but boss becomes 'worker' after assigning jobs

- Pipeline model

- work is done piece-meal and passed from thread to thread

Basic Thread Operations

- SRC (Modula-2+)

- Fork
 - Join

- Mutex

- Condition Variable

- Alert

- pthreads

- pthread_create(),
pthread_join()

- pthread_mutex_t

- pthread_cond_t

- pthread_cancel()

Using a mutex

- Shared data access must be protected (synchronized) through mutual exclusion
- Unprotected access will lead to errors

```
void* table[1000];  
int i = 0;  
  
void insert(void* p)  
{  
    table[i] = p;  
    i = i + 1;  
}
```

Using a mutex

- With unprotected access, results are unpredictable (and often wrong)

```
int table[1000];
int i = 0;

void insert(int p)
{
    table[i] = p;
    i = i + 1;
}
```

```
int x = 100;
int y = 200;

void* thread_a()
{
    insert(x);
    return 0;
}

void* thread_b()
{
    insert(y);
    return 0;
}
```

Using a mutex

- thread_a calls insert

```
table[0] = x;
```

```
int table[1000];  
int i = 0;
```

- thread_b preempts thread_a and calls insert

```
table[0] = y;  
i = 0 + 1;
```

```
void insert(int p)  
{  
    table[i] = p;  
    i = i + 1;  
}
```

- thread_b exits, thread_a continues

```
i = 1 + 1;
```

Using a mutex

- We were expecting x and y to be in the table
- We're left with y at `table[0]`, and an undefined value at `table[1]`, with $i = 2$
- That's bad news!

Using a mutex

- The proper approach is to synchronize access to table and i with a mutex

```
int table[1000];
int i = 0;
pthread_mutex_t mtx;

void insert(int p)
{
    pthread_mutex_lock(&mtx);
    table[i] = p;
    i = i + 1;
    pthread_mutex_unlock(&mtx);
}
```

Mutex usage is not without perils

- Deadlock
 - Two threads waiting on each other forever
- Poor performance
 - Mutexes introduce synchronization, poor granularity of locking can block threads resulting in worse performance than single threaded program

Deadlock

```
void* thread_a()  
{  
    pthread_mutex_lock(&mtx1);  
    i = i + 1;  
    pthread_mutex_lock(&mtx2);  
    j = j + 1;  
    pthread_mutex_unlock(&mtx2);  
    pthread_mutex_unlock(&mtx1);  
    return 0;  
}
```

```
void* thread_b()  
{  
    pthread_mutex_lock(&mtx2);  
    j = j + 1;  
    pthread_mutex_lock(&mtx1);  
    i = i + 1;  
    pthread_mutex_unlock(&mtx1);  
    pthread_mutex_unlock(&mtx2);  
    return 0;  
}
```

- thread_a locks mtx1
- thread_b preempts thread_a and locks mtx2
- thread_b now tries to lock mtx1 which is held by thread_a
- thread_a now tries to lock mtx2 which is held by thread_b

Deadlock

- Make sure you unlock your mutex
- Don't interleave mutex locking
- Span of a lock should be minimal amount required to ensure correctness
- Deadlock is relatively easy to debug, since your program hangs and you can examine the stack in a symbolic debugger

Poor performance

- Let's go back to the table example
- We want to add the values for all integers stored in the table

Poor performance

```
void* thread_a()  
{  
    static int result;  
    int j;  
    result = 0;  
    pthread_mutex_lock(&mtx);  
    for ( j = 0; j < i / 2; j++ )  
    {  
        result = result + table[j];  
    }  
    pthread_mutex_unlock(&mtx);  
    return &result;  
}
```

```
void* thread_b()  
{  
    static int result;  
    int j;  
    result = 0;  
    pthread_mutex_lock(&mtx);  
    for ( j = i/2; j < i; j++ )  
    {  
        result = result + table[j];  
    }  
    pthread_mutex_unlock(&mtx);  
    return &result;  
}
```

Poor performance

- Spawn new threads to do work after locking the table (OpenMP has simpler syntax)

```
int sum_table()
{
    int j;
    int result = 0;
    pthread_mutex_lock(&mtx);

#pragma omp parallel for reduction (+:result) private (j)
    for ( j = 0; j < i; j++ )
    {
        result = result + table[j];
    }

    pthread_mutex_unlock(&mtx);
    return result;
}
```

Poor performance

- Priority inversion occurs when threads of two different priorities acquire the same mutex
- It raises the question as to why to different priorities are assigned to the same resource
- Can be avoided through raising priority of holding thread --- would be nice if this happened automatically

Poor performance

- pthread library can set mutex attributes such that the current locking thread inherits the priority of the highest blocking thread

```
pthread_mutexattr_setprotocol(&mtx,  
PTHREAD_PRIO_INHERIT);
```

Scheduling through condition variables

- It is often nice to be able to wait for a certain condition arise (work to be done, queue is empty, etc.)
- Use condition variables for this

Scheduling through condition variables

```
pthread_mutex_lock(&q_mtx);  
while (queue.size() == 0)  
{  
    pthread_cond_wait(&q_notempty, &q_mtx);  
}  
/* pop something off the queue */  
work* w = queue.pop();  
pthread_mutex_unlock(&q_mtx);  
/* work on 'w' */
```

Scheduling through condition variables

```
work* w = (work*)malloc(sizeof(work));  
/* fill in some fields */
```

```
pthread_mutex_lock(&q_mtx);  
queue.push(w);  
pthread_mutex_unlock(&q_mtx);
```

```
pthread_cond_signal(&q_notempty);  
/* pthread_cond_broadcast(&q_notempty);  
*/
```

Scheduling through condition variables

- Signal vs. Broadcast
 - signal wakes a single thread waiting on the condition variable
 - broadcast wakes all threads waiting on the condition variable

Perils of condition variables

- Deadlock (again)
- Spurious wake-ups
 - result from using Broadcast when Signal should be used
- Starvation
- Handling these conditions leads to complex logic and code

Alerts

- Canceling a thread is wrought with hazards
 - fully asynchronous vs. specific cancel points

```
pthread_cancel(),  
pthread_testcancel(),  
PTHREAD_CANCEL_DEFERRED ...
```

- be wary of immediate cancellation while in system calls

Discussion

Questions

- Are Release(m) and Acquire(m) useful enough to justify?
- What might be performance impact of “lazy forking”?
- Birrell did not discuss effects of threading on cache. What principles could be conveyed in this regard?

Questions

- Why is communicating between different address spaces expensive (i.e. IPC vs. shared memory of threads)?
- page 9 -- "in some circumstances you might not get the correct answer" ... give examples
- Could you provide more details on lazy-fork?

Questions

- Please give detailed example of how to use Wait, Signal and Broadcast.
- How does the solution “to arranging to end LOCK clause before calling down” solve the nested monitor problem?

Questions

- What are the equivalents in Java of the Modula-2+ operations shown in the paper?
- The paper emphasizes performance improvement through the use of threads. Are the arguments justified?
- Is it irresponsible to discuss priority inversion without a solution?

Questions

- Broadcast doesn't guarantee order of thread awakening ... is this a problem? Are there performance issues with Broadcast?
- How do you "wait for multiple conditions" using the primitives in this paper?

Questions

- What is the complexity vs. performance trade-off? For example, partitioning data into smaller chunks each protected by a mutex vs. adding more code to optimize code but increasing complexity of logic?
- What are the problems with pipelining?

Questions

- How do threads differ on multi-processor systems compared to uni-processor systems?
- This is derived from comments that pipelining can achieve linear speed up.

Questions

- When you use threads to defer work, how do you handle consistency of data structures?
- Can hardware support make mutex lock/unlock cheap?

Questions

- What is the difference between thread programming and process programming?
- The paper doesn't discuss semaphores. What are the advantages / disadvantages of semaphores?

Questions

- What is a thread?
- What is difference between signal and broadcast on a condition variable?
- What is alert-pending and what is it used for?
- How can we avoid (guarantee?) deadlocks when only mutexes are used?

Questions

- How does one make sure that processes (threads) with high priority are run when waiting on processes (threads) with low priority?
- When do we use monitors instead of semaphores?
- Can threading cause slower performance?

Questions

- Could you point out some difficulties in programming with threads.
- What do you do if your program produces a deadlock?

Questions

- Birrell talks about “rarely” Signal awakening more than one thread for efficiency ... why?
- Suppose there is a single integer that has multiple readers (and writers), but there is low contention. How can you eliminate the locking overhead of the mutex?

Questions

- Threading seems to rely heavily on programmer to be implemented properly. What other general rules can we follow to take some of the load off the programmer?
- Birrell seems to like the idea of Broadcast to awaken all threads ... is this better or should a programmer use signal threads that need to be signaled?

Questions

- In pipelining, why does optimal efficiency depend on equal sized stages?
- How is synchronization tougher when code has to be made machine independent?

Questions

- The paper focuses on complexity of adding threads to a program. Are there examples where a threaded version is simpler and more intuitive?
- The paper implies that performance tweaking multiple-threaded programs is fragile. For portability how significant is the cost of maintaining the troublesome architecture dependent threaded parts?

Questions

- What sort of impact do you think introduction of coding has caused to the thought process of the programmers?
- Despite the age of the paper, how useful do you think it is to current programmers as an introduction to threads?

Questions

- Can you give some examples where the Broadcast function is better suited than a Signal call repeated a specific number of times?
- Are these synchronization techniques from 17 years ago essentially the same as what is used nowadays, or has technology improved significantly?

Questions

- Why provide the low-level lock/unlock interface when it can be done using two LOCK blocks? It's not obvious that "child" inherits locks from "parent"
- Are the newer implementation of asynchronous calls using threads necessarily better than the old schemes using interrupts? Is there a performance / flexibility tradeoff?

Questions

- In the paper for resolving spurious locks; it's stated that moving Signal(m) outside the protected area would indeed increase performance in multi-processor environments. However, isn't it possible that in transition from protected area to non-protected area another thread might call AcquireShared(m) and change the value of i?

Questions

- In the paper for resolving deadlocks between threads it's suggested that a partial ordering between the mutexes be found by the programmer. However, there may exist situations where a partial ordering is not possible. What other deadlock prevention/recovery techniques can be possible?

Questions

- In my opinion, the timeout option is very useful. The question is why the author didn't include this feature in his implementation of SRC? Does it have a high performance cost?
- Although the author has mentioned various tradeoffs (e.g. mutex granularity), he doesn't provide any quantitative metrics to the reader on which he can decide what to do. How do we ... (pick the right tradeoff)

Questions

- What is the relationship between the overhead of locks, join, fork, etc. and the system implementation?
- There is another way to implementing threads which is contrary to mutexes (threads are not put to sleep), but instead try to acquire the lock. They usually have a quicker response time (as no thread needs to be woken up) ... useful in high performance.