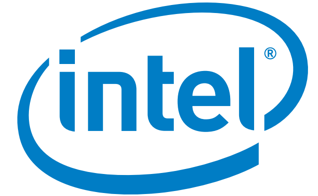


White Paper
Intel® Threading Analysis Tools
Multi-Core Simulation

by Orion Granatir

OMG, Multi-Threading is Easier Than Networking

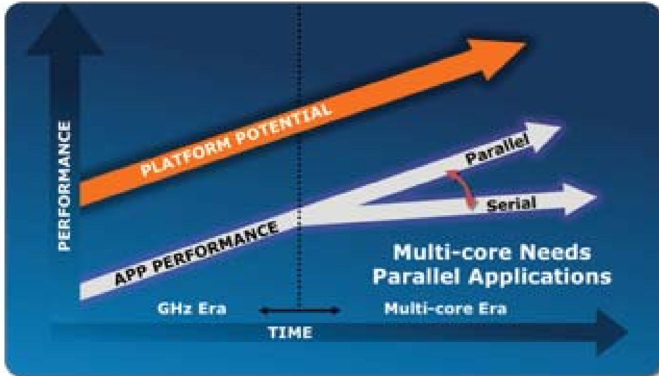
How threading is easy and similar to network code

Prior to working at Intel, I worked on PlayStation* 3 games at Insomniac Games. Most of the work I did at Insomniac dealt with multiplayer network code. When I came to Intel, I was able to focus on threading, eventually realizing that threading and network programming are similar in several ways.

It can be a challenge to understand threading. Using network programming as a comparison, this article intends to give you an introduction to threading so that by the end you will understand the basics of threading in a Microsoft Windows* environment.

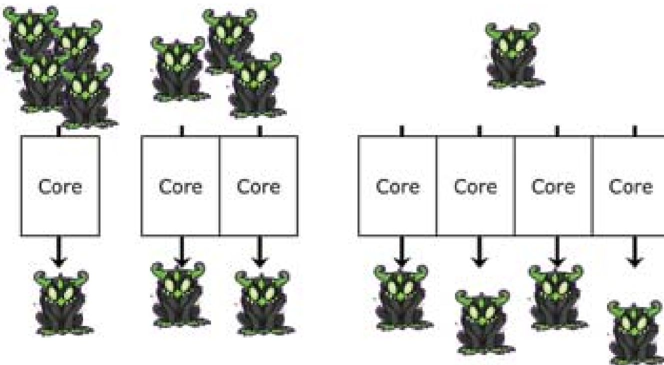
Why Thread?

Why thread? That's a good question. Why put up with all the challenges that come with threading an application?



In the past, CPUs saw consistent performance gains because of the increases in frequency. In modern microprocessors, however, this frequency gain is marginal, and most performance benefits come from an increased number of cores. Having more cores means the CPU can do more things at the same time. To maximize performance and features, an application needs to fully utilize the CPU—and that means threading!

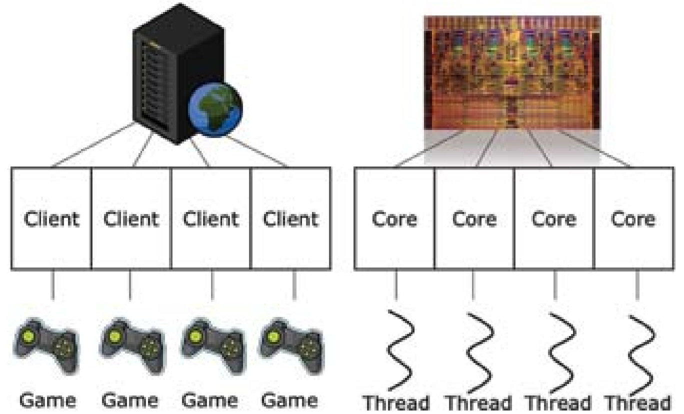
Imagine you want to update the artificial intelligence (AI) for a group of monsters in your game. If you have only one core, all of those monsters will need to be processed in order. However, if you have multiple cores, you can process several monsters at the same time.



More cores, and therefore more threads, means you can have more monsters in your game. Huzzah!

Threads Are like Clients

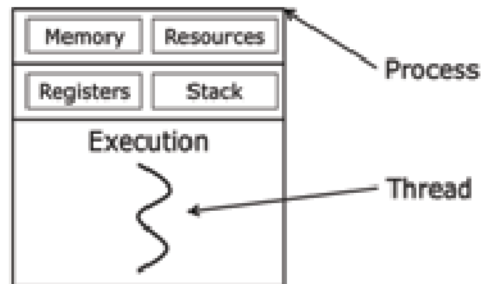
Threads are surprisingly similar to clients, but better. Clients are separate machines that all work and process independently. Like clients, cores all work separately and can process independent work. However, cores don't have to communicate over the slow and scary interwebs. Also, cores can quickly share data and will never lag out.



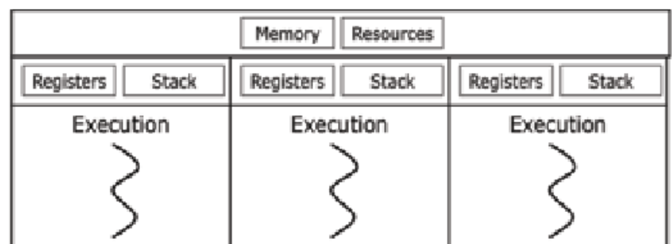
So what exactly is a thread?

- A thread is a series of instructions executed to complete a task.
- Each process can have multiple threads. Threads share the memory and resources within a process.
- Just like clients, threads can work independently.

Here is a process with one thread:



A process can have multiple threads:



Okay, let's see some code. This example creates four threads: one main thread and three threads that run the PrintLetter function.

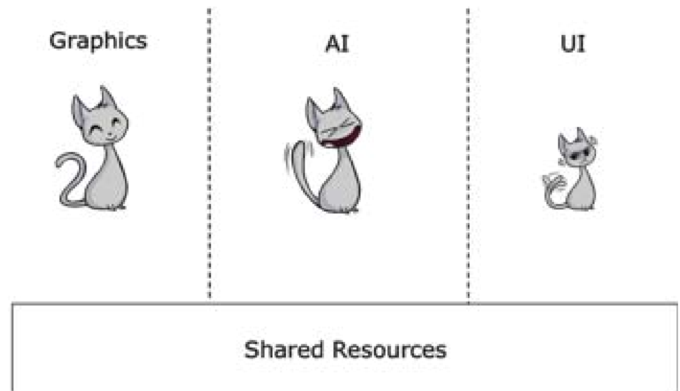
```
DWORD WINAPI PrintLetter( LPVOID data )

#define NUM_THREADS 3

void main( void )
{
    char data[ NUM_THREADS ] = { 'C', 'A', 'T' };

    for( int index = 0; index < NUM_THREADS; index++ )
    {
        CreateThread( NULL,           // Default security
                      0,              // Default stack size
                      PrintLetter,    // Function pointer
                      &data[ index ], // Parameter data
                      0,              // Start thread
                      NULL );         // Default id
    }

    // Do something
    // ...
}

DWORD WINAPI PrintLetter( LPVOID data )
{
    char* letter = (char*)data;

    for( int index = 0; index < 10; index++ )
    {
        printf( "%c", *letter );
    }

    return 0;
}
```

CreateThread spawns a new thread. You tell it which function to run and pass it any associated data. In this example the three threads run in a loop. The first thread prints 'C', the second prints 'A', and the third prints 'T'. The main thread will continue to run and do its own work. Since all threads are running at the same time your output will be a seemingly random combination of 'C', 'A', and 'T'.

Distributing Work

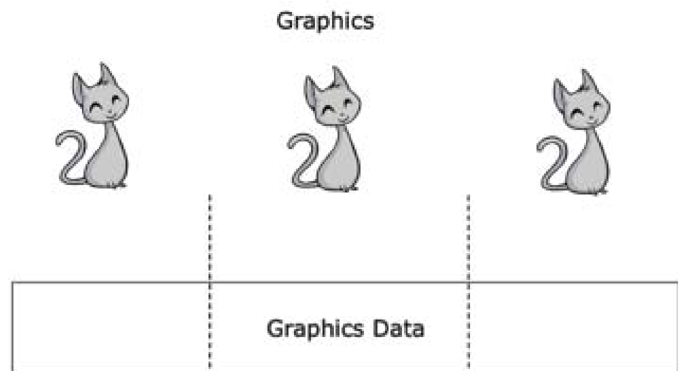
There are two ways you can think about threading code: functional decomposition and data decomposition.

Functional decomposition means separate threads work on different tasks. Here is an example of an application that is running Graphics, AI, and UI *tasks*.



Graphics, AI, and UI can all be running on separate threads. They can all work together through shared resources (for example, main memory).

Data decomposition means separate threads work on different *data*. Here is an example of an application that is updating Graphics on three threads:



The Graphics data can be broken apart and run on separate threads. For example, the first thread updates the first third of the graphics object, the second thread the middle third, and the last thread the remaining third.

Whenever it makes sense, use both functional and data decomposition. In general, data decomposition allows you to break your work into finer pieces and makes load balancing easier.

To see an example that uses both functional and data decomposition with typical game systems, check out the Smoke demo on Whatif.intel.com.

Authority—It's a Lock

Threads need to share resources, just like clients need to share resources in an online game. Clients need authority before they can interact with items shared in a game.

Say everyone needs a shovel to complete a quest, but there is only one shovel.

If a player wants to use the shovel, it must first become the authority of the shovel. It must agree with all the clients or the server that it has the right to use the shovel.

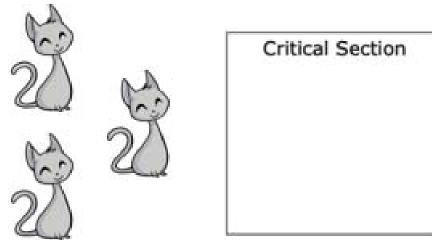
While this client is using the shovel, all other clients must wait for their turn. If multiple clients all want the shovel at the same time, a mechanism needs to ensure that only one client can access the shovel at a time.

This kind of activity is called "locking" for threads. There are a lot of ways to support using shared resources, but we are going to cover only critical sections, mutexes and semaphores, events, and atomic operations.

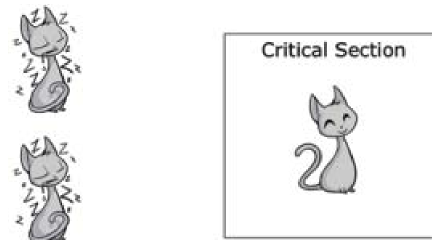
Critical Sections

A critical section is a piece of code that only one thread can execute at a time. If multiple threads try to enter a critical section, only one can run and the others will sleep.

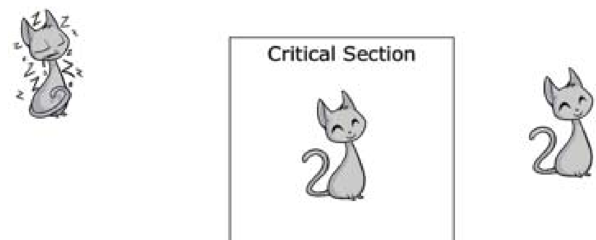
Imagine you have three threads that all want to enter a critical section.



Only one thread can enter the critical section; the other two have to sleep. When a thread sleeps, its execution is paused and the OS will run some other thread.



Once the thread in the critical section exits, another thread is woken up and allowed to enter the critical section.



It's important to keep the code inside a critical section as small as possible. The larger the critical section the longer it takes to execute, making the wait time longer for any additional threads that want access.

Here is some code that demonstrates critical sections.

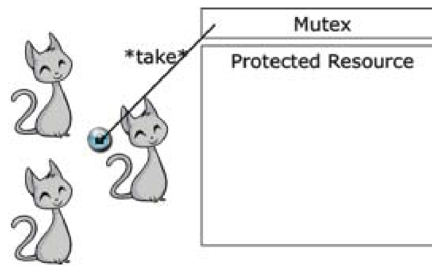
```
static int count = 0;

CRITICAL_SECTION critical_section;

void main( void )
{
    // Initialize the critical section
    InitializeCriticalSection( &critical_section );

    for( int index = 0; index < NUM_THREADS; index++ )
    {
        CreateThread( NULL,         // Default security
                      0,            // Default stack size
                      PrintNumber,  // Function pointer
                      NULL,         // No parameters
                      0,            // Start thread
                      NULL );       // Default id
    }

    // Do something
    // ...
}

DWORD WINAPI PrintNumber( LPVOID data )
{
    for( int index = 0; index < 10; index++ )
    {
        EnterCriticalSection( &critical_section );
        printf( "Count %d\n", count++ );
        LeaveCriticalSection( &critical_section );
    }

    return 0;
}
```

This code creates multiple threads that all print the value of the global variable count. Since multiple threads want to access count, all access to this variable is protected by a global critical section. Without the critical section, multiple threads will try to use count at the same time causing repeated numbers to print out.

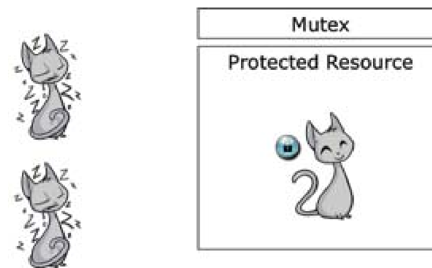
Mutexs and Semaphores

A mutex works like a critical section. You can think of a mutex as a token that must be grabbed before execution can continue. Here is an example of three threads that all want access to a shared resource.

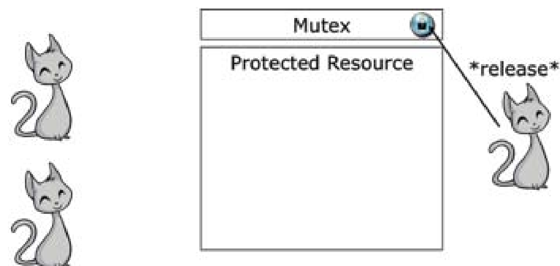
Each of the three threads tries to grab the mutex, but only one thread will be successful.



During the time that a thread holds the mutex, all other threads waiting on the mutex sleep. This behavior is very similar to that of a critical section.



Once a thread has finished using the shared resource, it releases the mutex. Another thread can then wake up and grab the mutex.



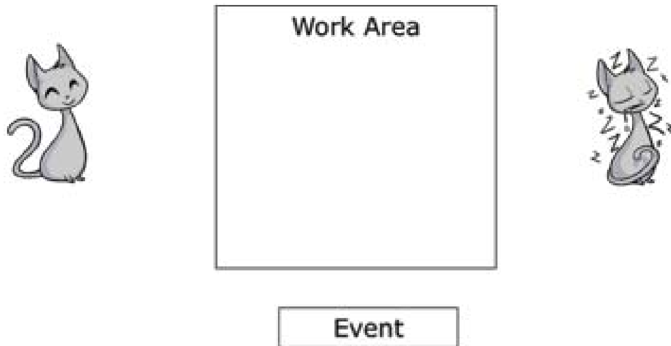
A semaphore is a mutex that multiple threads can access. It's like having multiple tokens.

Should you use a mutex or a critical section? Critical sections work within only a single process; a mutex works across multiple processes. Therefore, a critical section is a lot faster than a mutex, so use a critical section whenever possible.

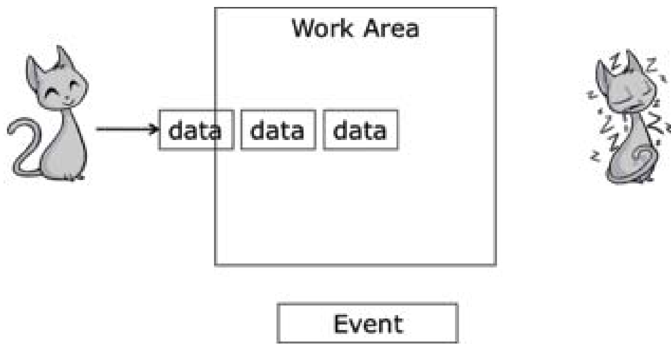
Events

Events are a way of signaling one thread from another, allowing one thread to wait or sleep until it's signaled by another thread.

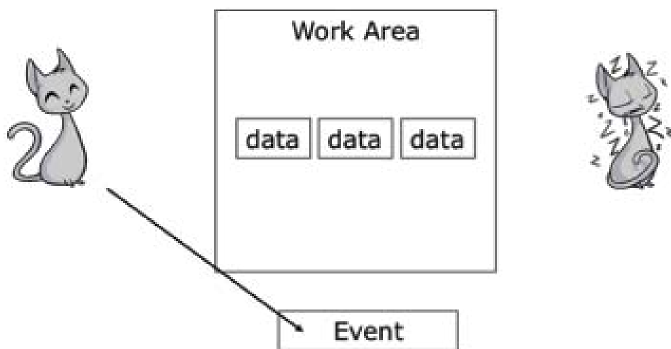
This example shows two threads using an event: the thread on the left is producing data, and the one on the right is consuming data.



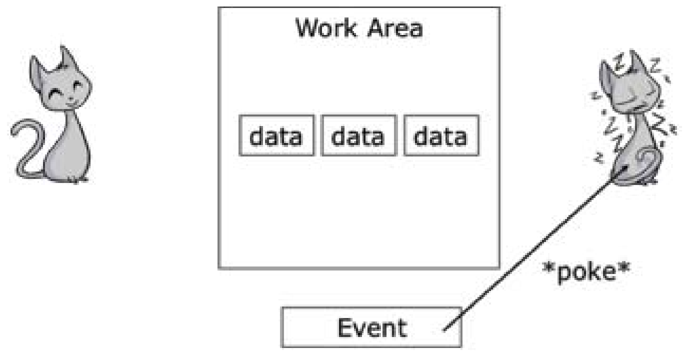
The producing thread generates some data and puts it in a common work area. In this example, the consuming thread is sleeping on the event (waiting for the event to trigger).



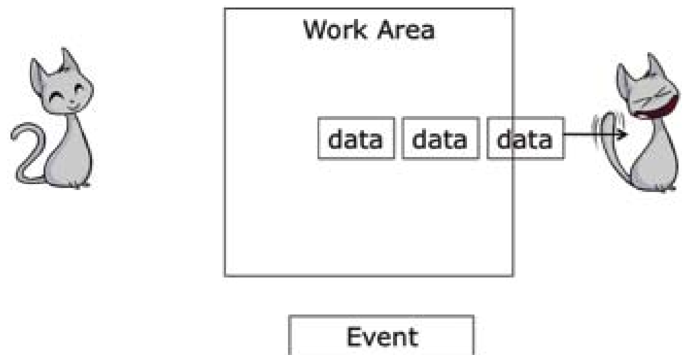
Once the producing thread has finished writing data, it triggers the event.



This signals the consuming thread, thereby waking it up.



Once the consuming thread has woken up, it starts doing work. The *assumption* is that the producing thread will no longer touch the data.



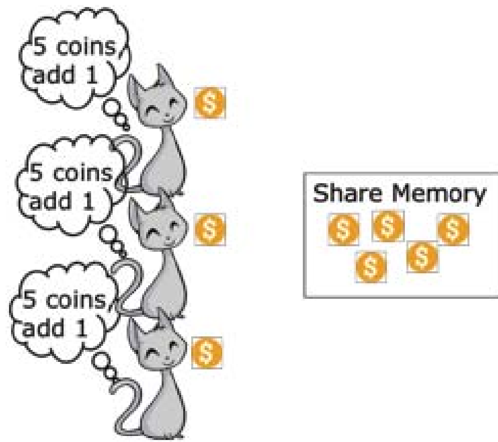
Here's some code that uses events.

```
DWORD WINAPI PrintLog( LPVOID data );

char buffer[ 256 ];
HANDLE event;

void main( void )
{
    // Initialize the event object
    event = CreateEvent( NULL,        // Default security
                         true,        // Allow manual reset
                         false,       // Initial state (not signaled)
                         NULL );      // Default name

    // Create the thread
    CreateThread( NULL, 0, PrintLog, NULL, 0, NULL );

    // Fill the buffer
    strcpy_s( buffer, 256, "Good kitty\n" );

    // Signal event
    SetEvent( event );

    // Wait for "Enter" to exit
    getchar();
}

DWORD WINAPI PrintLog( LPVOID data )
{
    WaitForSingleObject( event, INFINITE );

    printf( "Log: %s\n", buffer );

    return 0;
}
```

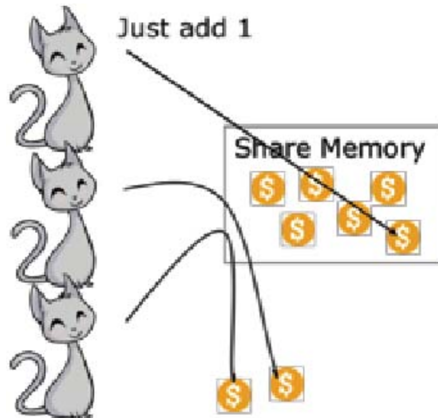
This code creates a thread that just waits for the event to trigger (to be set). The main thread puts some data in the global buffer and sets the event. Once the event is set, the second thread wakes up and prints the contents of the buffer.

Atomic Operations

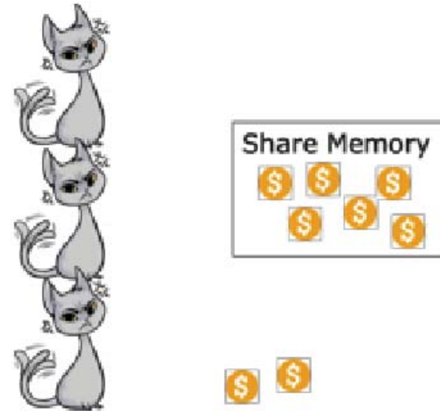
Atomic operations are guaranteed to be "thread safe" because they are atomic (that is, they happen without interrupts). Here is an example with three threads that all want to add 1 to a shared variable.



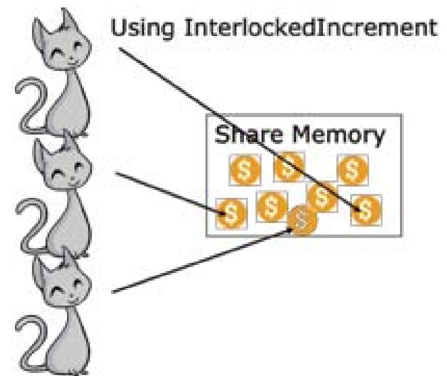
Let's say all threads run at the same time and see that the shared value is 5. Now each thread adds 1 and stores the result.



The correct answer is 5 + 1 + 1 + 1 = 8. However, each thread is going to read 5, add 1, and store the result, which gives an incorrect answer of 6.



Using an atomic operation to increment the value solves this problem. Each use of the atomic operation is guaranteed to finish before another thread can change the value.

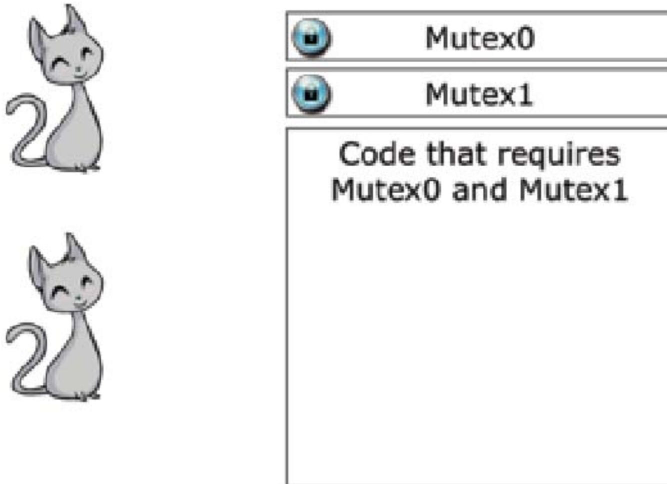


I'm in Your Code, Pwning Your Performance

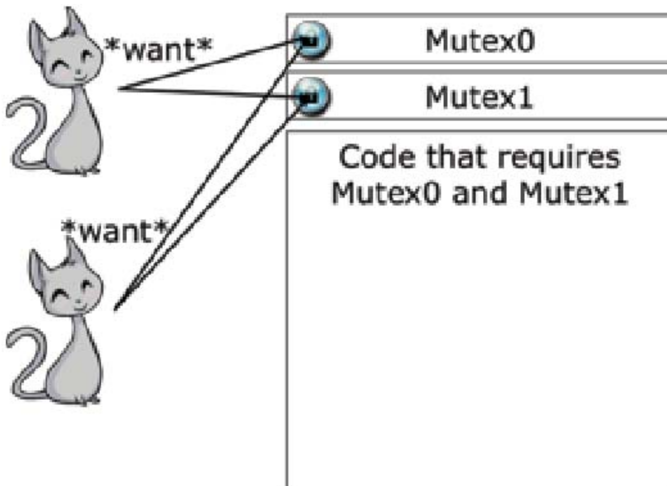
There are some pitfalls you'll want to avoid when threading. The two most common ones are deadlocks and race conditions.

Deadlock

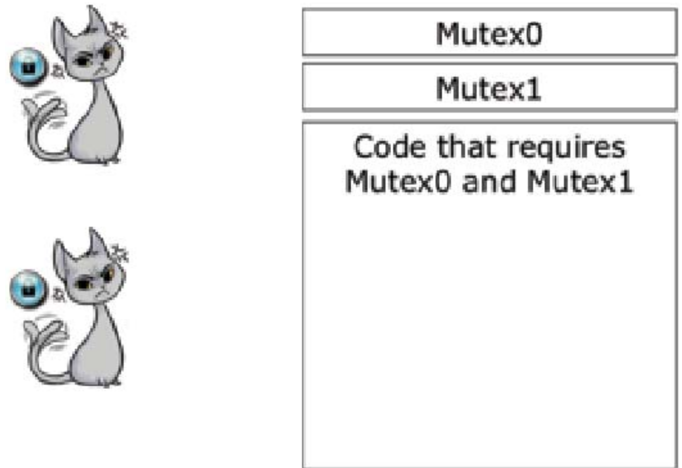
A deadlock happens when two or more threads are waiting on resources that are dependent on each other. Here is an example in which two threads want to do something that requires two mutexes.



In this example, one mutex might be protecting network access and the other might be protecting a log file. If both threads want to read data from the network and log it to a protected file, they need to grab both mutexes.



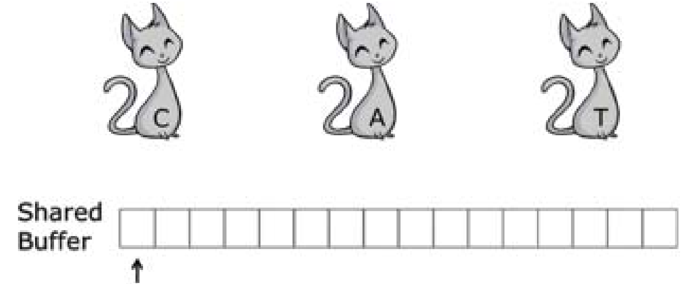
If one thread gets Mutex0 and the other one gets Mutex1, both don't have the required mutex to continue.



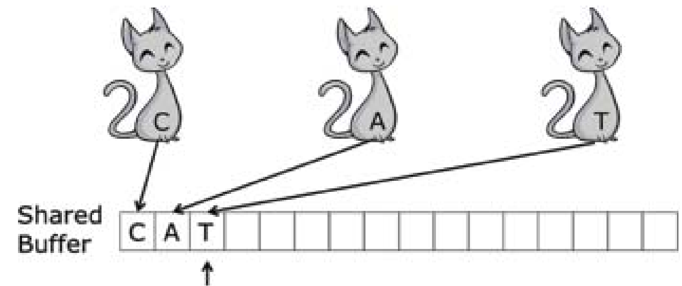
This situation is bad; nothing can happen because neither thread has both mutexes. Worse yet, they will sleep waiting for each other and never wake up.

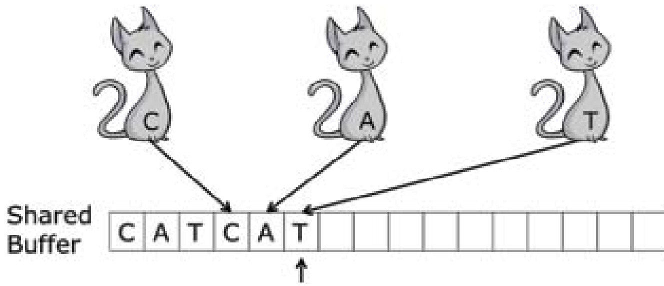
Race Conditions

A race condition can happen if the result of the threads' work is incorrectly dependent on the sequence of execution. In this example three threads are each writing a unique character to a shared buffer.

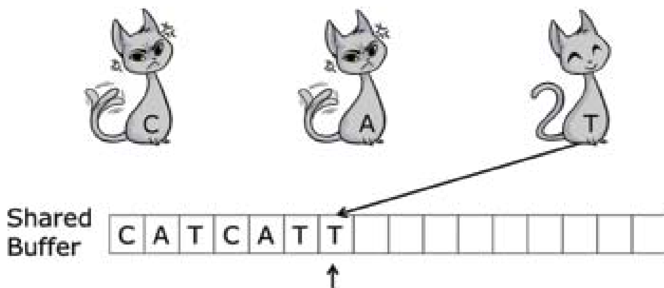


If the threads happen to run in the correct order, they will write 'CAT' over and over again.





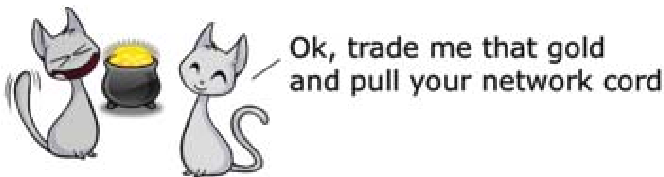
The problem occurs if one of the threads runs in a different order.



If you clearly define all the interactions between threads, it's not too difficult to avoid deadlocks and race conditions. Intel® Thread Checker will help analyze your code and determine locations that might have one of these pitfalls.

Think about thread interactions like multiple clients working on the same data. If you don't completely understand how data is going to be manipulated, you can end up with exploits.

And it's still easier than finding dup bugs or lag issues in games.



Multi-Thread This!

If you have gotten this far, hopefully you now have a grasp of network code and some working knowledge about threading. Now, let's combine the two! By using threads an application can process multiple network packets in parallel and decrease process latency.

Let's start with a simple game loop.

```
void main( void )
{
    // Game loop
    while( true )
    {
        // Process the frame
        UpdateFrame();

        // Process network messages
        UpdateNetwork();
    }
}
```

This game loop does two things: it updates the frame (UpdateFrame) and processes network data (UpdateNetwork). Let's create some threads to process network data.

```
void main( void )
{
    // Create the thread
    for( int index = 0; index < NUM_THREADS; index++ )
    {
        CreateThread( NULL, 0, NetworkThread, NULL, 0, NULL );
    }

    // Game loop
    while( true )
    {
        // Process the frame
        UpdateFrame();

        // Process network messages
        UpdateNetwork();
    }
}
```

We create NUM_THREADS threads before entering the main loop. These threads will run NetworkThread. We still need to signal the threads when it's time to update, so let's add an event.

```
void main( void )
{
    // Initialize the event object
    NetworkUpdateEvent = CreateEvent( NULL, true, false, NULL );

    // Create the thread
    for( int index = 0; index < NUM_THREADS; index++ )
    {
        CreateThread( NULL, 0, NetworkThread, NULL, 0, NULL );
    }

    // Game loop
    while( true )
    {
        // Process the frame
        UpdateFrame();

        // Release all the network threads
        SetEvent( NetworkUpdateEvent );
    }
```

Now let's look at NetworkThread.

```
DWORD WINAPI NetworkThread( LPVOID data )
{
    while( true )
    {
        // Wait for next update
        WaitForSingleObject( NetworkUpdateEvent, INFINITE );

        // Process network messages
        UpdateNetwork();
    }
}

void UpdateNetwork( void )
{
    NetworkMessage* msg;
    GetNetworkMessages( msg );  // Needs to be thread safe

    // Process network messages
    while( msg != NULL )
    {
        ProcessNetworkMessage( msg );  // Needs to be thread safe
        GetNetworkMessages( msg );
    }
}
```

The threads are simple enough: they run in a loop sleeping until the NetworkUpdateEvent event is triggered. Once the event is set, the threads all call UpdateNetwork. This function calls GetNetworkMessages to read network messages and ProcessNetworkMessage to process the packet. These two functions must be thread-safe because multiple threads will call them and access the shared data.

We can improve this example by adding better load balancing. For example, instead of just having the main thread signal NetworkUpdateEvent and wait, it could call UpdateNetwork and help out. Furthermore, the three created threads could help out with UpdateFrame.

For simplicity's sake, one step was omitted from this example: the main thread needs to wait for the helper threads to finish processing the network data before it continues and resets the event.

Conclusion

Threading can be a challenge, but using threads is necessary to unlock the power of the CPU and maximize an application's performance. Current trends indicate that CPU manufacturers will continue to add more and more cores. Hopefully you now feel more comfortable with threading, so give it a try in your own application.

Threading FTW!

About the Author

Orion Granatir is a senior in Intel's Visual Computing Software Division. Prior to joining Intel in 2007, Orion worked on several PlayStation 3 titles as a senior programmer with Insomniac Games. His most recently published titles are Resistance*: *Fall of Man and Ratchet and Clank* Future: Tools of Destruction.*

Copyright © 2009 Intel Corporation. All rights reserved. Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

Printed in USA 0209/WD/CS/PDF

Please Recycle