Atomicity

Concurrent Data Access Without Blowing Up

Written by Jonathan ‘Wolf’ Rentzsch (jon at redshed dot net)

This paper explains the dangers of concurrent access of data (in particular, linked lists). I provide an overview of interrupts, multitasking and callbacks. Lulling you into a false sense of security, I show you how simply incrementing an integer can go horribly wrong. After scaring you, this paper details a method to sidestep the dangers: atomicity. Then this paper goes through the trouble of offering (1) a pronunciation of atomicity (atom-ih-sit-ee), (2) a new method to create an atomic queue and (3) sneaky references to lyrics of cheesy 1980s songs.

Introduction

Life is so strange when you don’t know
How can you tell where you’re going to?
You can’t be sure of any situation
Something could change
And then you won’t know

Modern operating systems like to do many things at once. Multitasking, and its finer-grained cousin, multithreading, allow many programs to progress seemingly in parallel.

Of course, on single processor machines, only one program can actually proceed at a time. A modern operating system gives individual tasks a small slice of the processor’s time. These “time slices” are so small that programs appear to be running in parallel from the user’s standpoint. On multiprocessor machines, two or more programs can actually progress at the same time.

While the ability to do more than one thing at a time is crucial for modern software, it has a dark side. Programs can interfere with each other in subtle ways, leading to corruption of data and to crashes.

Interrupts *or I’m about to lose control, and I think I like it*

It helps to understand interrupts since it is how operating systems accomplish multitasking.

Inside your computer, your microprocessor (be it x86, 68K or PowerPC) sits on a **bus** along with a bunch of other black boxes. In reality, a bus is just a series of wires used for communication between the black boxes.

Along the bus sits various interesting devices. The memory controller, responsible for reading and writing to RAM. A time chip, watching the world grow old. A SCSI controller here, a USB controller there.

These black boxes send signals to the microprocessor when events occur. An event may be that a microsecond has passed or the user moved the mouse or the hard drive is finished reading data. More often then not, when the processor receives a signal, it stops what it was doing to handle the signal. The processor is **interrupted**.

The interrupt procedure is well-defined among microprocessors (see Figure 1):
1) The processor pushes the context onto the stack. A context is the address of the next instruction (I call this address the **resume address**) and the contents of the processor’s registers.
2) The processor will load the address of an **Interrupt Handler** corresponding to the interrupt’s type.
3) The processor will then jump to the beginning of the Interrupt Handler.
4) The Interrupt Handler is responsible for handling the interrupt as it sees fit. When the Interrupt Handler is finished, it executes a special **Return From Interrupt** instruction.
5) When the processor encounters the Return

From Interrupt instruction, it restores the processor’s state by popping the saved context off the stack and jumping to the resume address.
6) With any luck, the interrupted program is blissfully unaware it was ever interrupted.

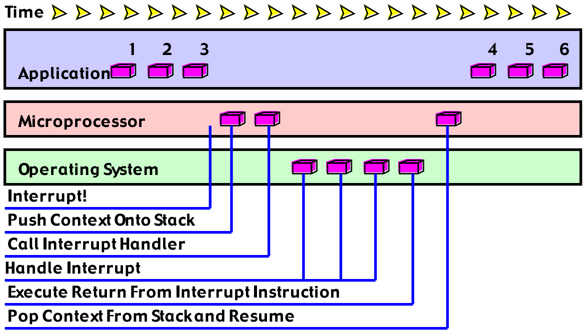


Figure 1.

Interesting Factoid: The processor needs to know where to find these Interrupt Handlers. On the original 68000 processor, the machine would load these addresses from the block of memory beginning at address 0. This is why writing to a nil pointer or handle is such a bad thing: you can easily fill the Interrupt Handler Table with data. During an interrupt, the processor attempts to interpret your data as addresses of an Interrupt Handler, leading to destinations unknown.

Multitasking
or Wake me up, before you go go

Modern operating systems use interrupts to multitask. Before the Interrupt Handler executes the Return From Interrupt instruction, it swaps out the old context. In its place, the operating system swaps in a context of a suspended program. When the processor executes the Return From Interrupt instruction, it unknowingly suspends one program and resumes another.

This is known as **preemptive multitasking** and stands in stark contrast to the Mac OS’ **cooperative multitasking**. The Mac OS will not preemptively switch from one task to another. Instead, it requires each program to call `WaitNextEvent()` or its cronies. Only then will one program be suspended and another resumed.

On modern operating systems, the operating system has the final say about which task gets to proceed and which get suspended until later. This leads to unpredictable behavior from the programmer’s viewpoint. How your program proceeds depends on what other programs are running. You cannot be sure that your program will proceed to completion without another program mucking with your data.

While the Mac OS lacks preemptive multitasking, you get the same effect from **callbacks**.

Callbacks
or Who can it be now?

Callbacks are application-supplied pointers to application-defined routines (or nowadays, routine descriptors pointing to application-defined routines). Typically, the application passes an address to a routine to the Mac OS, and later the Mac OS will “call back” into the application via the supplied routine.

Some callbacks are nice. Examples include the Dialog Manager and Apple Event Manager’s callbacks. Your application is called when its context is current and valid. You can allocate memory, dispose handles, open files, almost anything you would want.

Other callbacks, like the File Manager’s or Open Transport’s, are not so nice. You cannot depend on your application’s context being current. You cannot allocate memory using the standard Memory Manager, or even depend on unlocked handles. You have serious restrictions. Apple did not lay all these restrictions at your feet just to be mean — they did it because your callback is being called at **interrupt time**. Interrupt time is the time when the processor is handling an interrupt.

Let us take the File Manager as an example. Your application goes to read a block of data from a file from a SCSI hard disk. Being a Leading Edge Developer, you use the Mac OS’ `PBReadAsync()` routine and pass a File Manager callback (called a **completion routine**). When the SCSI controller is finished reading from the disk, it signals the microprocessor. The microprocessor interrupts the current flow of software and jumps to the Interrupt Handler, supplied by the Mac OS. The Mac OS finds your supplied completion routine and dutifully jumps to it. See Figure 2.

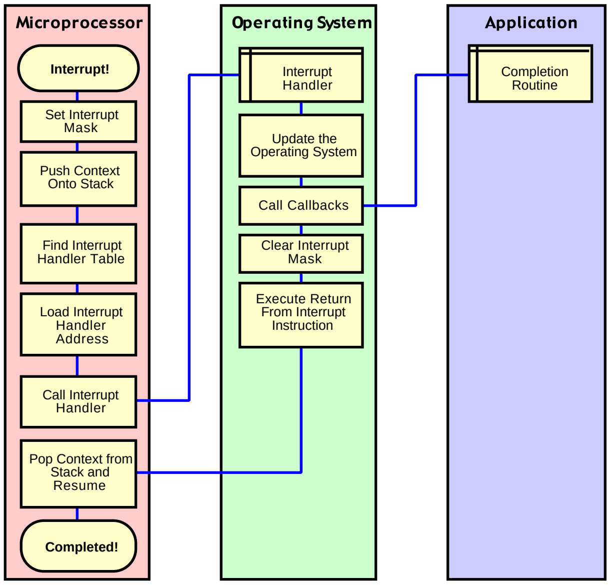


Figure 2.

Now your software is in an interesting state. It is likely your application has been interrupted, and now re-entered via the callback mechanism. It is possible your application just called `NewHandle()` and the Mac OS was in the middle of moving blocks of memory around when it was interrupted.

Now it becomes clearer why certain callbacks are limited in what they can do. Most of the Mac OS is not **reentrant**, meaning Bad Things happen if you interrupt the Mac OS and try to re-enter it during the interrupt (see [Apple1998a]).

Writing reentrant software is not easy, since something as simple as incrementing a shared counter can go horribly wrong.

Reentrancy *or Take on me*

Pretend you give up your life as a corporate lacky and start your own business selling a high-performance Open Transport-based server. Your server allows multiple connections at once, which presents a problem when the user quits your server. All connections must be gracefully closed before the server exits, otherwise your server will crash.

The solution is simple: with each new connection, a global counter is incremented. As each connection closes, the global counter is decremented. Now when the user goes to quit, your server blocks new connections and simply waits to the global counter to decrement to zero before quitting. The code you wrote for incrementing the counter is shown in Listing 1.

Listing 1.

```
unsigned long gConnectCount;

void  IncrementConnectCount()
{
  ++gConnectCount;
}
```

Many months pass and you finally ship your first product. Soon complaints roll in that your server randomly crashes upon quitting. You go over your code with a fine-toothed comb and cannot find the bug. Your server gets a reputation for being buggy and soon purchases cease. Finally you go bankrupt and go to work (shudder) admistrating corporate Wintel systems. It turns out that `IncrementConnectCount()` has a bug in it.

Bug? It is one line of code! What possibly could go wrong with one line of code? Hint: it is not reentrant (okay, that is the full answer, not a hint).

Your precious C compiler translated your one-liner into five separate PowerPC instructions, into something like Listing 2.

Listing 2.

```
//  Load address of gConnectCount into register r3.
lwz   r3, gConnectCount(rtoc)
//  Load integer from RAM into register r4.
lwz   r4, 0(r3)
//  Add 1 to register r4.
addi  r4, r4, 1
//  Store incremented value from register r4 back into RAM.
stw   r4, 0(r3)
//  Return from subroutine.
```

```
blr
```

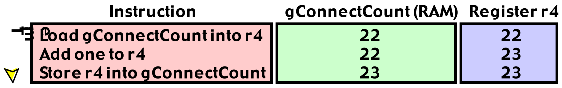
The initial `lwz` and final `blr` can be safely ignored — it is just overhead to access the global variable and return from the subroutine, respectively.

In this function, the integer `gConnectCount` is loaded into register r4 (the second `lwz`), one is added to register r4 (`addi`), and the contents of register r4 are written back to `gConnectCount` (`stw`). Like it or not, the PowerPC is a RISC processor with a load / store architecture. There is not an instruction to simply add 1 to a value in memory.

The Window of Death
or Once in a lifetime

It turns out there is a small possibility that your `IncrementConnectCount()` will be interrupted. When that happens, there is a small possibility that your function will be reentered. When the happens, your program will do the wrong thing.

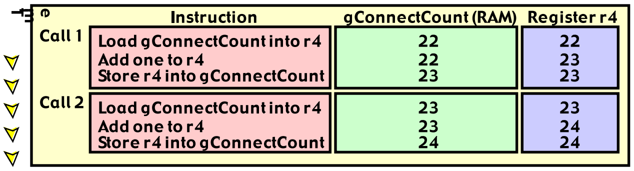
If your function is interrupted after it initially loads the value of `gConnectCount` and reentered before it stores the updated value back into `gConnectCount`, you will lose an increment. It helps if you visualize the flow of your function. Figure 3 depicts the function `IncrementConnectCount()` in action. For visualization purposes, the `gConnectCount` is assumed to hold 22.



Instruction	gConnectCount (RAM)	Register r4
Load gConnectCount into r4	22	22
Add one to r4	22	23
Store r4 into gConnectCount	23	23

Figure 3.

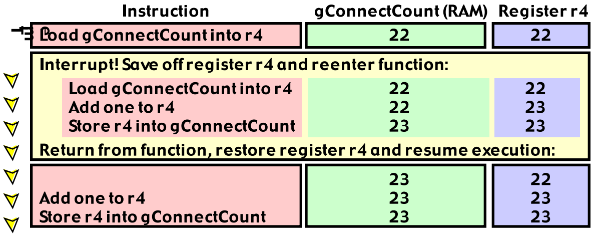
Figure 4 depicts `IncrementConnectCount()` being called twice in a row.



	Instruction	gConnectCount (RAM)	Register r4
Call 1	Load gConnectCount into r4	22	22
	Add one to r4	22	23
	Store r4 into gConnectCount	23	23
Call 2	Load gConnectCount into r4	23	23
	Add one to r4	23	24
	Store r4 into gConnectCount	24	24

Figure 4.

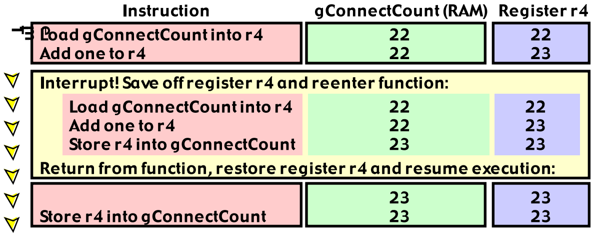
All is peachy so far. However, it is possible for `IncrementConnectCount()` to be interrupted and reentered as illustrated in Figure 5.



Instruction	gConnectCount (RAM)	Register r4
Load gConnectCount into r4	22	22
Interrupt! Save off register r4 and reenter function:		
Load gConnectCount into r4	22	22
Add one to r4	22	23
Store r4 into gConnectCount	23	23
Return from function, restore register r4 and resume execution:		
	23	22
Add one to r4	23	23
Store r4 into gConnectCount	23	23

Figure 5.

Here, `IncrementConnectCount()` is called twice but ends up only incrementing `gConnectCount` once. The problem is that `IncrementConnectCount()` does not finish storing to RAM before it is reentered. If the function is reentered after the load instruction, but before the store instruction, the function will fail. A slight variation, where the function is reentered **after** the increment instruction, will also fail. See Figure 6.



Instruction	gConnectCount (RAM)	Register r4
Load gConnectCount into r4	22	22
Add one to r4	22	23
Interrupt! Save off register r4 and reenter function:		
Load gConnectCount into r4	22	22
Add one to r4	22	23
Store r4 into gConnectCount	23	23
Return from function, restore register r4 and resume execution:		
	23	23
Store r4 into gConnectCount	23	23

Figure 6.

`IncrementConnectCount()`, as it is currently written, has a small (two instruction) **Window of Death**.

Exploring the Window of Death
or She blinded me with science

Most tasks are comprised of a series of instructions. Some tasks require briefly putting the given data into an invalid state before completing. If a task's instruction stream is interrupted before it completes, the data will be in an invalid state. If the invalid data is used during the interrupt, your program may operate incorrectly. This is the fundamental cause of the Window of Death.

Let us look at real-world example: a singly linked list. Every non-trivial program makes use of lists, and singly linked list are flexible and space efficient. The Mac OS uses singly linked lists all over the place. These Mac OS singly linked lists are optimized for adding elements to the end of the list and removing items from the front of the list. These optimized singly linked lists are called **queues** (see [Apple1994]).

Queues are handy method of asynchronous communication. Say two programs want to talk to each other. Program A can place a message in Program B's queue. Eventually Program B will check its queue, take the message and act on it. The programs do not need to be synchronized — Program B can be doing anything while Program A sends the message — and the message still gets there.

Like most operating systems, the Mac OS has device drivers — small programs devoted to control and monitor hardware. Device drivers are low-level and tend to be interrupt-driven. Queues also help out communication here: programs are free to place requests into the device driver's queue at any time. Multiple requests can be outstanding.

Listing 3.

```
typedef  struct  QElem  QElem, *QElemPtr;

struct  QElem  {
  QElemPtr  qLink;
  short     qType;
  short     qData[1];
};

typedef  struct  {
  short     qFlags;
  QElemPtr  qHead;
  QElemPtr  qTail;
} QHdr, *QHdrPtr;
```

Listing 3 details how the Mac OS defines a queue. A queue is made up of a `QHdr`, which has three fields: `qFlags` (ignored here), `qHead` (which always either points to the first element in the queue or is `nil`) and `qTail` (which always either points to the last element in the queue or is `nil`). An empty queue is represented when `qHead` and `qTail` are `nil`. A queue containing one element has `qHead` and `qTail` pointing to the same element.

Note if a queue is not empty, then the last element's `qLink` pointer is always set to `nil`. Otherwise, the queue is invalid (or corrupt). Also note that `qHead` and `qTail` can either both be `nil`, or neither can be `nil`. If one is `nil` and the other is not `nil`, then the `QHdr` is invalid. Figure 7 will make this more clear.

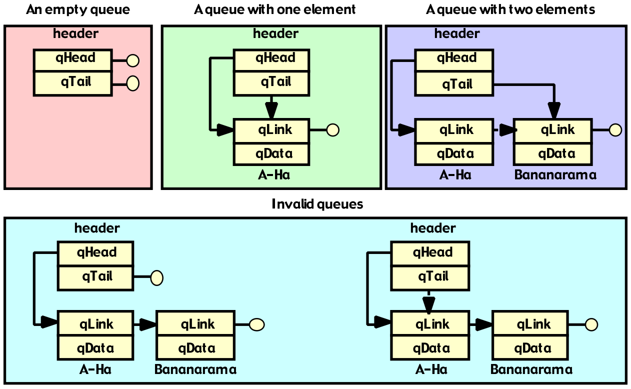


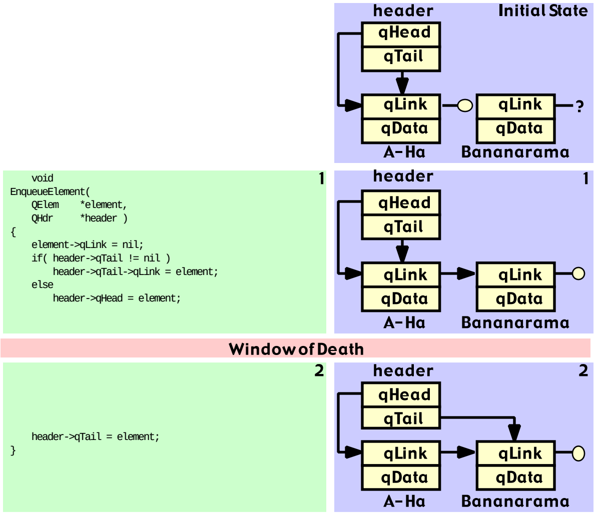
Figure 7.

Listing 4 shows a reasonable implementation of a function, `EnqueueElement()`, that places an element at the end of a queue.

Listing 4.

```
    void
EnqueueElement(
    QElem    *element,
    QHdr     *header )
{
    element->qLink = nil;
    if( header->qTail != nil )
        header->qTail->qLink = element;
    else
        header->qHead = element;
    header->qTail = element;
}
```

For visualization purposes, let us walk through `EnqueueElement()` (see Figure 8). To keep things simple, we will assume that the queue already has one element it named "A-Ha". The element we are adding is named "Bananarama".



```
    void
EnqueueElement(
    QElem    *element,
    QHdr     *header )
{
    element->qLink = nil;
    if( header->qTail != nil )
        header->qTail->qLink = element;
    else
        header->qHead = element;
```

```
    header->qTail = element;
}
```

Figure 8.

You will notice that `EnqueueElement()` does two things when confronted with non-empty queue: (1) set the last element's `qLink` pointer to the new element and (2) sets the `header`'s `qTail` to point to the new element.

You will also notice that between point 1 and point 2, the queue is in an invalid state. This is the Window of Death we have all come to know and love. If something interrupts `EnqueueElement()` between these two points and accesses the queue, they will find the queue in an invalid state. Let us move on to the other function. Listing 5 is a reasonable implementation of a function that removes the first element from the queue.

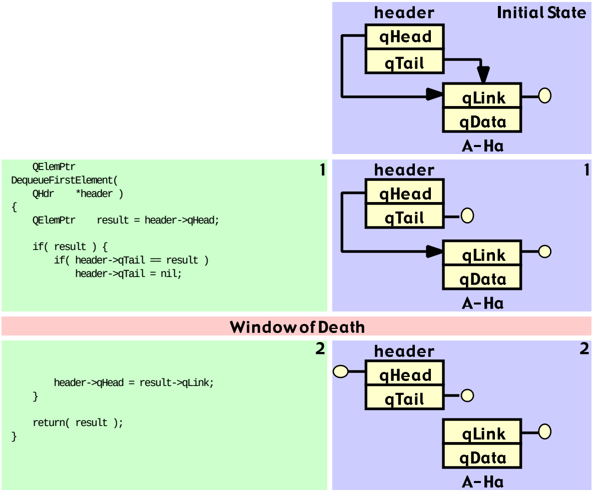
Listing 5.

```
    QElemPtr
DequeueFirstElement(
    QHdr    *header )
{
    QElemPtr    result = header->qHead;

    if( result ) {
        if( header->qTail == result )
            header->qTail = nil;
        header->qHead = result->qLink;
    }

    return( result );
}
```

Let us walk through this `DequeueFirstElement()` assuming a queue containing one element. See Figure 9.



```
    QElemPtr
DequeueFirstElement(
    QHdr    *header )
{
    QElemPtr    result = header->qHead;

    if( result ) {
        if( header->qTail == result )
            header->qTail = nil;
```

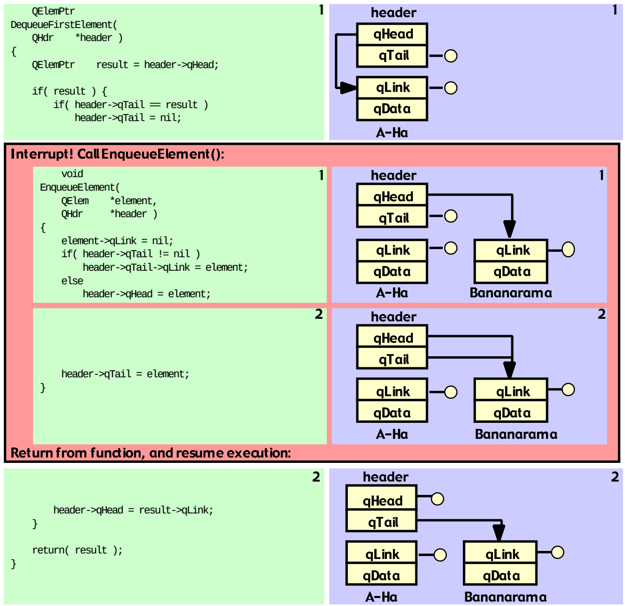
```
        header->qHead = result->qLink;
    }

    return( result );
}
```

Figure 9.

Like `EnqueueElement()`, `DequeueFirstElement()` performs two modifications on an one-element queue: (1) sets the `header`'s `qTail` pointer to `nil` and (2) sets the `header`'s `qHead` to point to the second element. Of course, there is no second element (A-Ha's `qLink` pointer is `nil`), so `qHead` is set to `nil`. Also like `EnqueueElement()`, `DequeueFirstElement()` has a Window of Death.

We have been building to it, so let us cut the tension by walking through an innocent application's call to `DequeueFirstElement()`. Unluckily for the application, `DequeueFirstElement()` is interrupted and `EnqueueElement()` is called, specifying the same queue. Witness the carnage in Figure 10.



```
    QElemPtr
DequeueFirstElement(
    QHdr    *header )
{
    QElemPtr    result = header->qHead;

    if( result ) {
        if( header->qTail == result )
            header->qTail = nil;
```

```
    void
EnqueueElement(
    QElem    *element,
    QHdr     *header )
{
    element->qLink = nil;
    if( header->qTail != nil )
        header->qTail->qLink = element;
    else
        header->qHead = element;
```

```
    header->qTail = element;
}
```

```
        header->qHead = result->qLink;
    }

    return( result );
}
```

1
header
qHead
qTail
qLink
qData
A-Ha
Interrupt! Call EnqueueElement():
Bananarama
2
Return from function, and resume execution:

Figure 10.

Interesting tidbit number 1: After `EnqueueElement()` completed task 2, the queue is in a valid state. `EnqueueElement()` "fixes" the invalid queue. However, `EnqueueElement()` returns and `DequeueFirstElement()` is resumed, only to deal the death blow to the queue.

Interesting tidbit number 2: If we reverse this scenario (that is, `DequeueFirstElement()` interrupts `EnqueueElement()` during its Window of Death), the queue is corrupted. However, if the queue holds exactly one item, the functions interact in such a way that both functions succeed without corrupting the queue!

Interesting tidbit number 3: Even though we have seen how `EnqueueElement()` and `DequeueFirstElement()` will corrupt the queue, in fact there is nothing wrong with either of them. Both will function admirably so long as you do not interrupt them during their Window of Death. Indeed, there is no way to write this code in C (or C++) to make it "right". You have to go lower-level than what C offers, which is low indeed.

There is only a small chance of encountering the Window of Death. However, the risk is accumulative. Each time you call a subroutine with a Window of Death, the possibilty increases towards 100%.

Closing the Window of Death
or The safety dance

The cause of the Window of Death is a series of instructions that should not be interrupted. So it stands to reason that by minimizing the number of intructions, the Window of Death would shrink. Indeed, simply turning on your compiler's optimizer will shrink the Windows. I compiled `EnqueueElement()` using Metrowerks's PowerPC compiler without any optimization. It produced 11 instructions, with a 5 instruction-long Window of Death (the Window made up 45% of the function). After turning on the optimizers, the function shrunk to 10 instructions, with a 4 instruction-long Window of Death (reduced to 40% of the function).

Remember our `IncrementConnectCount()` function from before? The PowerPC is a load / store architecture. In order to increment a value in memory, it first needs to load the value into a register, then increment it and finally store it back into memory. That is three instructions, with a two instruction Window of Death.

The 68K is not a load / store architecture: the add instruction can directly modify memory. If you recompile `IncrementConnectCount()` for the 68K, the Window of Death disappears! That is because the 68K `add` instruction reads the value in memory, adds to it and writes it back out in **one indivisible operation**, otherwise known as **an atomic instruction**. The word atomic comes from the word atom, which means "indivisible".

The Mac OS Queue Utilities
or I want to be your sledgehammer

Way back in the early 1980s, the Mac OS system software engineers had a problem. They were using queues for communications between normal event-driven applications and interrupt-driven device drivers.

However, there was not a way to write an atomic linked list. The needed instructions, cas and cas2, were not available until the Mac II, with its 68020, rode into town. Their solution was hard and heavy, but it worked: they disabled interrupts.

The 68K defines a register called the status register. Three bits of this register are set aside as the **interrupt mask**. Interrupts on the 68K come in 7 levels, one through seven. It is possible to set this interrupt mask so that any interrupts at or below the mask are ignored. For example, if you want to disable Time Manager tasks (which execute at interrupt level 1), set the interrupt mask to 1. Any interrupts at level 2 or above will be handled, any interrupts at level 1 are disabled. Setting the interrupt mask to zero enables all interrupts. Application software spends most of its time at with the interrupt mask set to zero.

Disclaimer: It is undocumented that Time Manager tasks execute at interrupt level 1, and is subject to change. This example is for illustrative purposes only (see [Apple1998a]).

Disabling interrupts works, but it is not a very nice thing to do. It hurts performance and can lead to data loss and data corruption. Remember, interrupts are how all those black boxes communicate with the microprocessor. One of those devices might be a serial chip whose puny buffers are filling fast. From the black boxes' point of view, the processor becomes non-responsive. Thus this technique is sometimes (though rarely) known as "when we pretend that we're dead".

The Mac OS system software engineers wrote two functions: Enqueue() and Dequeue(). Both set the interrupt mask to 7, its highest level. By disabling interrupts, the functions know they will complete without tragedy.

It was the right thing to do, however the jump to PowerPC made this operation expensive. In order to provide complete compatibility, the PowerPC version of the Mac OS runs a nanokernal. This nanokernal is responsible for catching PowerPC interrupts and revectoring them through the 68K emulator. As a result, the 68K emulator is tightly wound to the PowerPC. Indeed, the only way to disable interrupts on the Power Macintosh is to switch into 68K mode, set the interrupt mask to 7, and switch back to PowerPC.

As we will see later, it is impossible to write an atomic version of Enqueue() and Dequeue() on the PowerPC. So Enqueue() and Dequeue() are stuck being 68K code. When a PowerPC native application calls these functions, you will get two mixed-mode hits.

To top it all off, Dequeue() works in linear time — meaning the longer the queue, the longer it takes to remove an arbitrary element. If your PowerPC native application attempts to remove the last item of a thousand element queue, you suffer (1) a mixed mode switch entering Dequeue(), (2) interrupts are disabled, (3) wait while Dequeue() iterates over 999 elements to find the element, (4) re-enable interrupts and (5) another mixed mode switch to reenter your application. Whew!

PowerPC Atomicity *or Always something there to remind me*

The PowerPC is a Reduced Instruction Set Computer (RISC) architecture. Only two classes of instructions are allowed to touch memory: load and store. Other instructions are limited only to touching registers. Really, this is all for the best, but it sure makes atomicity difficult. The PowerPC engineers threw us a bone in the form of two special instructions: Load Word and Reserve Index (lwarx) and Store Word Conditional Index (stwcx.).

lwarx works just like the common Load Word and Zero Indexed (lwzx), except it places a **reservation** on the loaded address as well as loading the data. The PowerPC processor can hold only one reservation at a time.

stwcx., is the yin to lwarx's yang. Alone, neither is the life of the party. Together, they make beautiful music. You see, stwcx. works just like any other store instruction, except stwcx. is conditional. It only performs the store if a reservation is present on the given address. If there is a reservation, then it clears the reservation, performs the store and sets the condition register CR0[EQ] to true. Otherwise, the instruction does nothing except set the CR0[EQ] to false.

So what are these “reservations”? The main reference for the software interface to the PowerPC processor, the *PowerPC Microprocessor Family: The Programming Environments*, is vague on the subject. It seems a reservation acts somewhat like a register. During each store instruction, the processor compares the given address to reservation. If they are equal, the reservation is cleared. However, reservations can also work in multiprocessor environment. If processor A places a reservation on address X and processor B stores to address X, processor A’s reservation is cleared. This suggests reservations are more than glorified registers. Chances are the reference manual is vague since reservations are implemented in different ways on different processors.

The Window of Death is caused by a series of instructions that should not be interrupted. The Queue Utilities fight interruptions by disabling interrupts. The 68020 and later fight interruptions by doing more work in each instruction. The PowerPC’s answer is different: go ahead, interrupt all you want — just leave a note that you have mucked with our data.

These two instructions form a foundation for emulating atomicity on the PowerPC. Let us go back to our poor non-atomic `IncrementConnectCount()` function, reviewed in Listing 6.

Listing 6.

```
// Load address of gConnectCount into register r3.
lwz   r3, gConnectCount(rtoc)
// Load integer from RAM into register r4.
lwz   r4, 0(r3)
// Add 1 to register r4.
addi  r4, r4, 1
// Store incremented value from register r4 back into RAM.
stw   r4, 0(r3)
// Return from subroutine.
blr
```

Listing 7 details how `IncrementConnectCount()` can be re-written to be atomic.

Listing 7.

```
    // Load address of gConnectCount into register r3.
    lwz       r3, gConnectCount(rtoc)
again:
    // Load gConnectCount into register r4.
    lwarx     r4, 0, r3
    // Add 1 to register r4.
    addi      r4, r4, 1
    // If we didn’t lose the reservation
    // Then store register r4 into gConnectCount.
    stwcx.    r4, 0, r3
    // Else try again.
    bne-      again
    // Return from subroutine.
    blr
```

The basic idea here is that if the reservation is lost, the function tries again. It is unlikely it will fail the first time around, even more unlikely the second, close to impossible the third, etc. You computer science types may worry that there is no guarantee of forward progress. No one seems to care about this possibility, but you could add a retry counter to make sure the function runs in finite time.

Clever as reservations are, the limit of one per processor limits their use. You only can atomically modify one 32 bit word at a time. This rules out fast doubly linked lists, which require an atomic update of two values in memory. It also rules out all but the most basic singly linked list: a stack. More on this later.

68K Atomicity
or Sweet dreams are made of this

The 68K is fairly awesome when it comes to atomicity. It defines two instructions that really help out: Compare and Swap (`cas`) and Compare and Swap 2 (`cas2`).

The `cas` instruction takes three parameters: a comparison value, an update value and an address. `cas` will atomically compare the value at the given address to the comparison value. If the values are equal, `cas` will atomically update the value in memory with the update value and set the Z flag to true. Otherwise `cas` does nothing except set the Z flag to false.

A program can grab a value from memory, modify it, and then execute the `cas` instruction.

If the Z flag is set after the instruction, the program knows that the value in memory was changed after it was copied, and can retry the operation.

This try-test-retry business is a lot like the PowerPC. In fact, rewriting a 68K version of an atomic PowerPC function is easy since they act similar. But there is nothing like cas2 on the PowerPC.

cas2 works like cas, only with 2 comparison values, 2 update values and 2 addresses. This allows atomic modification of two different values in memory. This allows implementation of atomic doubly linked lists.

Sadly, this instruction is not used much. A compiler would never generate it (indeed, Metrowerks 68K C compiler's inline assembler does not even recognize cas or cas2). Futhermore, there is no equivalent on the PowerPC. It is important to be able to produce 68K and PowerPC binaries from a single source base, so functions need to be implementable on both sides of the fence. Atomic stacks **can** be implemented natively on 68K and PowerPC, so let us concentrate on those.

Atomic Stacks
or Everything counts in large amounts

In the mid 1990s, Apple was working on its new networking architecture, Open Transport. Open Transport would make extensive use of linked lists, and Apple was painfully aware about Enqueue() and Dequeue()'s inefficiencies. Apple needed new list code.

It was a good thing the PowerPC was out, otherwise Apple probably would have chosen cas2 as the basis for the linked list code, resulting in tragedy when the PowerPC came along. It was a good time for new list code: the old 68000 machines were dwindling in numbers, so Apple could take advantage of the 68020's cas and the PowerPC's lwarx/stwcx. deadly duo. The need for atomic stacks is a big reason why Open Transport will not run on a 68000 processor.

Open Transport defines three functions to deal with atomic stacks. By the way, Open Transport does not call them atomic stacks, instead it is the more formal "last in, first out" (LIFO) list. The three functions are (1) OTLIFOEnqueue() which pushes an element onto a stack, (2) OTLIFODequeue() which pops an element off a stack and (3) OTLIFOStealList() which atomically **steals** a list. Stealing an atomic stack means copying the stack's head and setting the original to nil. You now own the stack.

Stacks are not as useful as queues, which Open Transport calls "first in, first out" (FIFO) lists. So Open Transport provides a function called OTReverseList(). This non-atomic function reverses a stack into a more useful queue. The idea is to populate an atomic stack, atomically steal it and non-atomically reverse it into a queue. It is a work-around, but it is better than disabling interrupts.

As nice as Open Transport is, it is not available on all machines. I want my software to work on as many machines as possible, so I wrote my own atomic stack functions: PushAtomicStack(), PopAtomicStack() and StealAtomicStack(). Listing 8 and Listing 9 display the implementation of PushAtomicStack() and PopAtomicStack(), respectively. For brevity, I am only listing the Classic 68K implementation. In reality, I use an unholy combination of the C #if/#else/#endif preprocessor commands and Metrowerks C compilers' ability to define assembly functions to automatically generate the correct code for Classic 68K, CFM-68K and PowerPC from the same source file. You can grab the full source for these functions (including the PowerPC implementation) from the MacHack 1999 CD or you can email me.

Listing 8.

```
typedef struct AtomicElement
AtomicElement, AtomicStack;

struct   AtomicElement   {
    AtomicElement  *next;
};

    asm
    pascal
    void
PushAtomicStack(
    AtomicElement    *element,
    AtomicStack      *stack )
{
    // stack.
    movea.l  4(a7), a0
    // element.
    movea.l  8(a7), a1
    // element.
    move.l   a1, d0
again:
    // next = stack->next.
    move.l   (a0), d1
    // element->next = next.
    move.l   d1, (a1)
    // If stack->next didn't change,
    // Then set stack->next to element.
//  cas.l    d1, d0, (a0)
    // CWPro2 doesn't know cas. Here's the raw opcode.
    dc.l     0x0ED00001
    // Else someone else progressed, try again.
    bne.s    again
    // Pop the return address.
    movea.l  (a7)+, a0
    // Pop the parameters.
    addq.l   #8, a7
    // We're outta here.
    jmp      (a0)
}
```

Listing 9.

```
    asm
    pascal
    AtomicElement*
PopAtomicStack(
    AtomicStack    *stack )
{
    // stack.
    movea.l 4(a7), a0
again:
    // element = stack->next.
    movea.l  (a0), a1
    // element = stack->next.
    move.l   a1, d0
    // Is element == nil?
    tst.l    d0
    // If equal, return nil.
    beq.s    done
    // next = element->next.
    move.l   (a1), d1
    // CWPro2 doesn't know cas. Here's the raw opcode.
    dc.l     0xED00040
    // If stack->next wasn't changed, stack->next = next.
//  cas.l    d0, d1, (a0)
    // Else someone else progressed, try again.
    bne.s    again
done:
    // Pop the return address.
    movea.l  (a7)+, a0
    // Pop stack parameter.
    addq.l   #4, a7
    // Return the popped element on the stack.
    move.l   d0, (a7)
    // We're outta here.
    jmp      (a0)
}
```

Atomic Locks
or I'm on the hunt, I'm after you

Atomic stacks are fast and reentrant. However, it is easy to corrupt the stack into a nasty circular list. This occurs if an element is pushed onto the stack more than once. As an example, picture a stack containing two elements: "Bananarama" and "Cheap Trick". You successfully push the new element "A-Ha". Then you push Bananarama again. The list becomes currupted with Bananarama pointing to A-Ha and A-Ha pointing to Bananarama. Cheap Trick is removed from the stack. Witness the entire sorted affair in Figure 11.

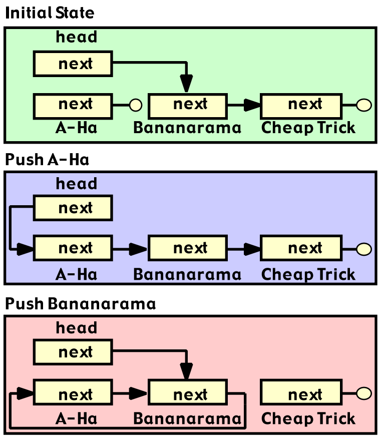


Figure 11.

Often you will be faced with a situation where you should place an element onto a stack if it is not already in the stack. You could walk the stack looking for the given element, however such an operation would be non-atomic and thus could fail if interrupted.

The better approach is to keep a flag in the element. When the element in pushed onto the stack, the flag is set to true. When the element is popped, the flag is cleared. Now it is easy to know whether an element is already in a list.

However, the element's flag is a shared value, which means it is subject to reentrancy issues. The answer is to set the flag atomically. An atomically controlled flag is called an **atomic lock**. We can define two functions that manipulate atomic locks: `GrabAtomicLock()` and `ReleaseAtomicLock()`. Listing 10 shows the prototypes for these functions.

Listing 10.

```
typedef unsigned long   AtomicLock;

    long   // Non-zero if successfully grabbed.
GrabAtomicLock(
    AtomicLock    *lock );

    void
ReleaseAtomicLock(
    AtomicLock    *lock );
```

`GrabAtomicLock()` atomically compares the value in `lock` against zero. If the lock's value is zero, it is atomically set to one and `GrabAtomicLock()` returns true. Otherwise the value is left alone and `GrabAtomicLock()` returns false. `ReleaseAtomicLock()` simply unconditionally sets the `lock`'s value to zero.

Atomic locks are very useful for synchronization, and are found on most operating systems. They can go by different names, like mutex (short for "mutual exclusion") or semaphore. Mutexes and semaphores can also be implemented by disabling interrupts.

Atomic Queues
or You spin me round

As we have seen above, you cannot atomically emulate traditional Mac OS queues on the PowerPC. However, with a slight redefinition of how the Mac OS defines a queue, you can get the same effect as a first in, first out queue.

The Mac OS allows `Dequeue()` to remove an arbitrary element. While this is nice, it is the reason we cannot make it atomic. Instead, I define two functions: `PushAtomicQueue()` and `PopAtomicQueue()`. They both work on a structure called an `AtomicQueue`, which is nothing more than two atomic stacks (see Listing 11).

Listing 11.

```
typedef    struct    {
    AtomicStack    in;
    AtomicStack    out;
}   AtomicQueue;
```

All `PushAtomicQueue()` does is call `PushAtomicStack()` to push the given element onto the `in` stack.

`PopAtomicQueue()`'s job is more complicated. First it tries `PopAtomicStack()` on `out`. If the result is not nil, `PopAtomicQueue()` returns it. Otherwise the stack is empty and needs "refilling".

Refilling is accomplished by using `StealAtomicStack()` on `in`. Now we reverse the stack by popping each element from the stolen stack and pushing it onto `out`. Finally we pop `out` and return the result (which may be nil if the queue is empty). Perhaps Listing 12 will make this more clear.

Listing 12.

```
   AtomicElement*
PopAtomicQueue(
   AtomicQueue  *queue )
{
   AtomicElement  *next, *current;

   current = PopAtomicStack( &queue->out );

   if( current == nil ) {
       // Nothing to pop.
       // Refill the queue from the input stack.
       current = StealAtomicStack(&queue->in);
       while( current ) {
          next = current->next;
          PushAtomicStack(current,&queue->out);
          current = next;
       }
       current = PopAtomicStack( &queue->out );
   }

   return( current );
}
```

The only problem is that if `PopAtomicQueue()` is interrupted while reversing the stack **and** one or more elements are added to the queue using `PushAtomicQueue()` **and then** `PopAtomicQueue()` is called again, the queue will not be first in, first out state — some elements will become out-of-order. No elements are lost, and the queue will not be corrupted, so this is not a big problem.

This is essentially the same thing as Open Transport's atomic-LIFO reversal to normal-FIFO technique, except the mechanics are hidden under two easily understood functions, providing a polished metaphor.

Conclusion *or Shout, shout, let it all out*

Atomicity is important for maintaining correctness in the face of concurrency. Concurrency will become more of an issue as we move towards Mac OS X, so you should become prepared now. This paper helped illustrate the issues involved and offered a new method for implementing atomic queues. If you don't know atomicity by now, you will never ever know it.

Bibliography *or The Policy of Truth*

Dewar, R. B. K. and Smosna, M. , "Microprocessors: A Programmer's View", McGraw-Hill, 1990. A great read comparing the x86, 68K, MIPS, SPARC, i860, RIOS and INMOS Transputer. Dewar and Smosna are particularly moved by the 68K `cas2` instruction, and provide an overview of IBM's RIOS, the code name of the original POWER architecture (which, of course, evolved into the PowerPC).

Motorola, "68000 Family Programmer's Reference Manual", Motorola, 1989. The 680x0 bible. Pages 20 thru 26 include an example of how to use `cas` and `cas2` to implement atomic lists. Chances are my copy is outdated.

IBM, Motorola, "PowerPC Microprocessor Family: The Programming Environments", Motorola, 1994. Often called the Programming Environments Manual (which allows the nice acronym "PEM"), this is the PowerPC programmer's bible. Appendix E details how to use lwarx/stwcx. pair for atomic operations.

[Apple1994] Apple, "Inside Macintosh: Operating System Utilities", Addison-Wesley, 1994. Features a description of the Mac OS' Queue Utilities.

[Apple1998a] Bechtel, Brian and "The Eskimo", Quinn, "Technote 1104: Interrupt-Safe Routines", Apple, 1998. This technote is required reading for all advanced Macintosh programmers. It clears up the "interrupt time" definition mess quite nicely. It even reveals that the beloved `TickCount()` is not interrupt-safe!

[Apple1998b] "The Eskimo", Quinn, "Technote 1137: Disabling Interrupts on the Traditional Mac OS", Apple, 1998. This technote describes how to disable and enable interrupts on the Mac OS. It also cautions against using the PowerPC's lwarx/stwcx. pair since "the behavior of these instructions varies between PowerPC CPU types. Accommodating all these variations is tricky."

[Songs1980s] Listed in order of inclusion.
Format is:
"Quote"
    Artist(s)
    Song Title
    Album
"Life is so strange..." (from the introduction)
    Missing Persons
    Destination Unknown
    Spring Session M
"I'm about to lose control, and I think I like it"
    Pointer Sisters
    I'm So Excited
    Break Out
"Wake me up, before you go go"
    Wham!
    Wake Me Up Before You Go Go
    Make It Big
"Who can it be now?"
    Men At Work
    Who Can It Be Now?
    Business As Usual
"Take on me"
    A-Ha
    Take on Me
    Hunting High And Low
"Once in a lifetime"

Talking Heads
Once in a Lifetime
Remain In Light

“She blinded me with science”
Thomas Dolby
She Blinded Me with Science
Retrospectacle-Best Of

“The safety dance”
Men Without Hats
The Safety Dance
The Safety Dance

“I want to be your sledgehammer”
Peter Gabriel
Sledgehammer
So

“When we pretend that we’re dead”
L7
Pretend We’re Dead
Bricks Are Heavy

“Always something there to remind me”
Naked Eyes
Always Something There to Remind Me
Promises, Promises

“Sweet dreams are made of this”
Eurythmics
Sweet Dreams (Are Made of This)
Sweet Dreams (Are made of this)

“Everything counts in large amounts”
Depeche Mode
Everything Counts
People are People

“I’m on the hunt, I’m after you”
Duran Duran
Hungry like the Wolf
Rio

“You spin me round”
Dead or Alive
You Spin Me Round (Like a Record)
Youthquake

“Shout, shout, let it all out”
Tears For Fears
Shout
Songs From The Big Chair

“If you don’t know atomicity by now, you will never ever know it” (paraphrased)
Simply Red
If You Don’t Know Me by Now
A New Flame

“The Policy of Truth”
Depeche Mode
The Policy of Truth
Violator