A picture containing text

Description automatically generated

World Of Warcraft

addon Development Using Asynchronous, Non-preemptive Multithreading

Other books by M. Terry Peterson

Peterson, Michael T., DCE: A Guide To Developing Portable Applications, McGraw-Hill, 1995

## World Of Warcraft

## addon Development Using Asynchronous, Non-preemptive Multithreading

## M. Terry Peterson, PhD

Rock Creek Publishing, LLC

Missoula, MT

Copyright © 2022 M. T. Peterson

All rights reserved. No part of this book may be reproduced in any form or by any electronic or mechanical means, including information storage and retrieval systems, without permission in writing from the publisher, except by reviewers, who may quote brief passages in a review.

Library of Congress Control Number: 987865431  
ISBN 1234567891011  
Printed in the United States of America

©2004 Blizzard Entertainment, Inc. All rights reserved. World of Warcraft, Warcraft, and Blizzard Entertainment are trademarks or registered trademarks of Blizzard Entertainment, Inc. in the U.S. and/or other countries

©1996 - 2014 Blizzard Entertainment, Inc. All rights reserved. Battle.net and Blizzard Entertainment are trademarks or registered trademarks of Blizzard Entertainment, Inc. in the U.S. and/or other countries.

.

# Preface

If your only tool is a hammer, all problems look like nails

-- Abraham Maslow

I’ve never met a young engineer who didn’t think threads were great. I’ve never met an older engineer who didn’t shun them like the plague.

-- Anonymous

I am a developer of the second kind. I’m old, I’m a [former] software engineer, and I have a love-hate relationship with threading. I’ve written thousands of lines of multithreaded user-mode applications and protected-mode[[1]](#footnote-1) operating system internals for Linux and Microsoft NT. Over the years, I’ve learned that while experience with multithreading can be impressive on a resume, using multiple threads seldom live up to their hype. And so it is with threads when applied to WoW addon programming. They are quite useful but lead to madness if not applied appropriately.

This book was motivated by some WoW addon work I did for personal use. Over the course of playing World of Warcraft, I noted that many of the WoW addons I used were quite impressive. The code was clean and functional. None seemed overly complex, especially for complexity’s sake, as is so often the case with code written by less experienced engineers. Writing addons looked to be fun (and useful), so I jumped in. Initially, I wrote a couple of simple addons, one to display information when my character experienced a skillup and [yet another] bank management addon[[2]](#footnote-2). But, as I gained more familiarity with Blizzard’s addon [Application Programming Interface](https://wowpedia.fandom.com/wiki/World_of_Warcraft_API) (API) and especially its event-driven execution model, I couldn’t help but wonder if threading, rightly applied, might be useful. So, I wrote a simple combat logger and immediately ran into difficulty.

I needed a more flexible way to control the spacing between lines of text as they scrolled across the screen. Since most WoW addons are required to handle asynchronous events to a greater or lesser extent, I thought I’d write a small prototype threads facility that would enable me to dedicate threads to receiving, processing, and displaying individual combat events and their associated tasks. As it happened, the prototype thread package worked. My application didn’t run faster, but that wasn’t the goal. Instead, I needed to slow down the rate at which text frames scrolled up the screen. ***Talon***, my code name for [what became] the WoWThreads facility described in this book, offered a set of primitives that allowed me to easily assign tasks to specific threads and coordinate their execution using a signals-based communication scheme.

So, is multithreading the way to go when developing WoW addons? Probably not, and here’s why. First, most of my professional software career required designing multithread libraries and multithreaded applications. I thought, “Well, it was straightforward getting a minimal thread facility working using Blizzard’s addon API,” how much more difficult would it be to write a more functional one – especially with robust error checking and some simple debugging tools? I sank a couple of months into the project, found that it wasn’t particularly difficult, and am reasonably satisfied with its functionality.

But, and I can’t say this strongly enough, I’m not convinced threading brings much to a WoW addon developer’s toolkit. Threading is sexy, and it’s way cool to see one’s threaded code in action. But does threading make for better addons? In my opinion, most improvements will be marginal, but in some cases, they may be significant to be sure. But for ***general*** purposes, no.

Second, and maybe most important, there seems to be some confusion about threads – both what they are and how best to use them. I hope this document will help clear up some of that confusion and maybe help developers understand more clearly whether their addon might benefit from threading.

Table of Contents

[World Of Warcraft 4](#_Toc137479595)

[addon Development Using Asynchronous, Non-preemptive Multithreading 4](#_Toc137479596)

[M. Terry Peterson, PhD 4](#_Toc137479597)

[Preface 6](#_Toc137479598)

[Introduction 11](#_Toc137479599)

[Concepts 11](#_Toc137479600)

[Execution Models 14](#_Toc137479601)

[Multithreading – Preemptive and Non-preemptive threads 17](#_Toc137479602)

[Preemptive Threads 18](#_Toc137479603)

[Non-preemptive threads 19](#_Toc137479604)

[Concurrency 19](#_Toc137479605)

[Thread Safety In WoWThreads 20](#_Toc137479606)

[The WoW Multithread Library API 21](#_Toc137479607)

[Design Considerations 21](#_Toc137479608)

[Patterns 21](#_Toc137479609)

[Code Structure 23](#_Toc137479610)

[Getting Started 25](#_Toc137479611)

[Installation 25](#_Toc137479612)

[Threads 25](#_Toc137479613)

[The Thread Handle 26](#_Toc137479614)

[Thread Creation 27](#_Toc137479615)

[The Thread Control Block 27](#_Toc137479616)

[The Thread Scheduler 27](#_Toc137479617)

[Signaling – Thread-to-Thread Communication 28](#_Toc137479618)

[API Entry Points 31](#_Toc137479619)

[thread:create() 31](#_Toc137479620)

[thread:yield() 32](#_Toc137479621)

[thread:wait() 32](#_Toc137479622)

[thread:join() 32](#_Toc137479623)

[thread:exit() 33](#_Toc137479624)

[thread:self() 33](#_Toc137479625)

[thread:getId() 34](#_Toc137479626)

[thread:areEqual() 34](#_Toc137479627)

[thread:getParent() 35](#_Toc137479628)

[thread:getChildren() 36](#_Toc137479629)

[thread:getState() 36](#_Toc137479630)

[thread:getSignalName() 37](#_Toc137479631)

[thread:sendSignal() 37](#_Toc137479632)

[thread:getSignal() 38](#_Toc137479633)

[Error Handling 40](#_Toc137479634)

[threadErrors:setResult() 40](#_Toc137479635)

[thread:postResult() 41](#_Toc137479636)

[Debugging Support 42](#_Toc137479637)

[threadErrors:dbgPrint() 42](#_Toc137479638)

[threadErrors:prefix() 42](#_Toc137479639)

[Display Service 44](#_Toc137479640)

[thread:postMsg() 44](#_Toc137479641)

[Management Services 45](#_Toc137479642)

[thread:getCongestion() 45](#_Toc137479643)

[Thread Congestion – discussion 46](#_Toc137479644)

[Appendix 49](#_Toc137479645)

[Installation Examples 49](#_Toc137479646)

[Threads In Perspective 49](#_Toc137479647)

[Example Application(s) 49](#_Toc137479648)

[About the Author 50](#_Toc137479649)

# Introduction

Every so often, a post will appear on one of the WoW Forums suggesting that Blizzard would be better off if their program development philosophy would just get with the times and redesign the WoW client to support multithreading. This would not be wise for all kinds of reasons[[3]](#footnote-3), but what strikes me as fascinating is the misconceptions many thread proponents have about threads and how best they can/should be used. Perhaps the most profound misconception is the belief that the value of multithreading an application resides in the performance improvement gained by parallel execution of the application’s threads. Nothing could be further from the truth (see the links to the two articles at the end of this document).

Actually, multithreading for parallel execution is not that common except on very high-end systems, not infrequently built with special hardware. As it happens, the most common use of multithreading is probably for event-driven programs such as gaming, database management, and other I/O-intensive applications.

## Concepts

In this section, I provide some basic definitions for some of the more important concepts surrounding multithreaded programming.

**Asynchronous** - refers to tasks (threads or processes) that are *scheduled independently* of each other. In other words, the execution of task A is *not* dependent on when another task, B, starts or completes.

**Attribute object** - refers to an object that specifies the attributes of a thread. In WoWThreads, the attributes object is the *thread handle*. In POSIX threads, the attributes object is a separate object, *\*attr*, as shown in their respective declarations below:

|  |  |
| --- | --- |
| POSIX | local thread\_h, result = thread:create( yieldInterval, threadFunction, … ) |
| Talon | int pthread\_create( pthread\_t \*thread, pthread\_attr\_t \*attr, … ) |

Note that POSIX requires the attribute object to be separate from the executable thread. In WoWThreads, a thread’s attributes and its executable thread are both referenced by the thread handle. *Attribute objects* like the attr\* in POSIX and the thread handle in WoWThreads, are opaque to client applications and cannot be directly modified by assignments. In WoWThreads, like POSIX, a set of functions is provided through which some of the attributes contained in the thread handle may be accessed. For WoWThreads, these functions are:

thread:getId()

thread:self()

thread:areEqual()

thread:getParent()

thread:getChildren()

thread:getParent()

thread:getState()

thread:getSignal()

thread:setSignal()

thread:delay()

**Concurrency** - occurs when two or more computational tasks overlap in time. **Multitasking** is a form of concurrency in which two or more tasks share the use of a single processor by alternating execution streams. Parallel execution is another form of concurrency in which execution streams execute simultaneously on separate processors.

**Coroutine**s - (see *thread*) are a specific kind of thread, sometimes referred to as a lightweight thread. Coroutines may be scheduled independently of other coroutines/threads through the use of a scheduler. Coroutines, like POSIX and other threads, only run in user-mode[[4]](#footnote-4). Like all executable objects, coroutines have their own stack, instruction pointer, executable code, data, and symbol table.

**Critical section -** refers to a code path that accesses shared data. To prevent the shared data from being corrupted, critical sections must be executed serially (see *serialization*).

**Mutual exclusion** - refers to a type of serialization. *Mutual exclusion* is commonly implemented by an object called a *mutex* (**Mut**ex-**Ex**clusion). Mutexes are objects that must be acquired prior to accessing a critical section and must be released upon completion of the critical section.

**Reentrant -** refers to a code path that can be executed concurrently by a thread or process and then, before its completion, be reentered by a second thread or process. Reentry can occur by multiple threads reentering the code path or by a single thread reentering the code path via a recursive operation. *Reentrant* code must be made *thread-safe* if operating in a concurrent execution model.

**Serialization** - a programming technique that ensures that shared data are accessed by concurrent tasks sequentially in a one-at-a-time manner (see *critical section*).

**Synchronous** - refers to the sequential execution of tasks in which task B, for example, cannot begin execution until task A completes.

**Thread** - a shorthand reference to a *thread of execution*. Threads are a sequence of instructions that can be managed independently by a scheduler. Under this definition, for example, a Unix process (created by fork(2)) is also a thread of execution, as is the WoW client. POSIX, Windows, and WoW Threads are unlike processes in that these threads do not have their own address space.

**Thread handle** - refers to the attributes object of a WoW thread.

**Thread Safe** - Concurrent tasks communicate by using [shared memory](https://en.wikipedia.org/wiki/Shared_memory_(interprocess_communication)) locations. This style of concurrent programming requires the use of some form of locking (e.g., [mutexes](https://en.wikipedia.org/wiki/Mutual_exclusion), [semaphores](https://en.wikipedia.org/wiki/Semaphore_(programming)), or [monitors](https://en.wikipedia.org/wiki/Monitor_(synchronization))) to **serialize** access to the shared memory. A program that properly implements any of these is said to be [**thread-safe**](https://en.wikipedia.org/wiki/Thread_safety).

## Execution Models

There are essentially two kinds of thread execution models that differ by whether their threads execute synchronously or asynchronously. The asynchronous model can be further subdivided into execution models in which the threads execute serially and those that execute simultaneously (a.k.a parallel execution). In either case, the defining characteristic of asynchronous execution models is that the *threads execute independently* of each other. For this reason, asynchronous models are the *sine qua non* of asynchronous execution models.

In the interests of clarity, consider the following analogy (not mine, I heard this many years ago). Suppose a chef needs to fulfill an order for eggs and toast. Here is how these execution models apply to the chef’s need to prepare meals for the restaurant’s customers.

1. Synchronous Execution:

Tasks are not independent. Thus, the chef makes the eggs. When the eggs are done, the chef makes the toast. When the toast is made, the chef serves the customer cold eggs along with hot toast. This model is commonly used by soon-to-be unemployed chefs but also reflects traditional programming in which procedure calls proceed synchronously, one after the other.

1. Asynchronous Execution I – Serial, non-preemptive execution

In this execution model, the chef starts the eggs cooking and then turns to other kitchen tasks like washing dishes, cutting onions, and so forth. When the eggs are almost done, s/he starts the toast so that it completes when the eggs are done. The waiters are then able to serve the customer hot toast and hot eggs.

Most of us work this way naturally. It just makes sense to turn to some other task while waiting for another task to complete.

1. Asynchronous Execution II – Parallel, preemptive execution

In this model, the chef uses two cooks. Accordingly, s/he assigns a cook to the eggs and a cook to the toast. The chef then coordinates the two cooks such that the eggs and toast finish at the same time. As in the Type I model, the customer gets hot eggs and toast.

However, the chef has to synchronize the two cooks (for some reason, they don’t communicate with each other and often block each other when, for example, they both need to use the sink or divvy up the last few onions). In general, the chef is now required to ensure the two cooks will be able to share resources without unduly interfering with each other. In addition, since the employer now has to provide a salary for the two additional cooks, the cost of the meal is correspondingly higher, and the customer is charged accordingly. This model makes sense only when favored by economies of scale.

In this analogy, the asynchronous parallel model might be thought of as *employee-oriented*, while the non-preemptive model is better thought of as *task-oriented*. The key here is that both models are asynchronous, both are useful, and both have their places. In terms of software applications, the parallel model is unbeatable for large-scale, high-end, compute-intensive applications that, and this is key, can be structured to run multiple threads simultaneously and independently on separate processors – or, per the analogy, giving each of the cooks their own kitchen – a very expensive but sometimes necessary proposition.

In Blizzard's World of Warcraft (WoW) programming environment, the event handling system is synchronous (like the unemployed chef above). When an event occurs for which an addon has registered (say a combat event), the WoW client delivers the event and its payload to the addon. Because the OnEvent service is synchronous, the game must wait for the addon to complete its handling of the event before it continues with other tasks. This synchronous behavior means that the addon's event handling code blocks the WoW client’s game loop until it completes the OnEvent() function.

Talon is a facility that enables developers to structure the design of their addon such that the addon’s execution is decoupled from the WoW client. Thus the WoW client and the addon execute asynchronously from each other. As it happens, gaming is event-driven and largely task-oriented. You ought not to be surprised, therefore, to learn that almost all Android games and many (most?) modern game engines[[5]](#footnote-5) employ Talon-like facilities to enable asynchronous behavior.

## Multithreading – Preemptive and Non-preemptive threads

The concept of threads as an executable object can be confusing, so I’d like first to clarify not only what threads are but for what they are best used. A thread is nothing more than a data structure (a table in Lua) containing as one of its elements a single procedure call. Nominally, a thread, like a process, is a set of data (a stack, registers, the instruction pointer, a symbol table, etc.) and a sequence of executable code able to be *independently* scheduled for execution.

But there is an important distinction between a process (like the WoW client) and a thread. A process encompasses a single address space. Traditional threads, by contrast, contain no address space but instead, share the address space of a process (like the WoW Client). The practical consequences of this distinction are that creating a new thread is far less expensive than creating a new process. Moreover, since threads within the same process share the process’s address space, data produced by one thread is immediately available to all other threads in the process. This makes communicating between threads simple and fast.

At the same time, they are similar to processes in that, just as processes can execute independently of each other, threads are similarly able to *execute independently* of other threads in that process. But, there are some complicating factors, and they arise from the nature of how the scheduler switches between threads. These switches can be preemptive or non-preemptive (also called cooperatively self-scheduling).

### Preemptive Threads

Thread preemption requires operating system support[[6]](#footnote-6) and is very resource intensive. So unless the cost of a thread context switch[[7]](#footnote-7) can be offset by an increase in parallelism, programmers should avoid preemption as they would a plague. Why? Because the difficulties involved in managing preemptive threads are legion and arise from the observation that a thread can be preempted at any point in its execution stream. This means that if the operating system decides to switch to another thread while the current thread is updating your bank account, you could be left broke. But there are well-understood, albeit risky, techniques for dealing with this problem. In my experience, the most difficult problem facing programmers are those that arise from the indeterminacy of preemptive thread execution. For example, fixing a timing bug that reveals itself only intermittently can lead a programmer to madness.

A second difficulty was hinted at a few sentences previously. Programmers must serialize access to critical sections (e.g., shared data such as your bank account), and this means locking. The requirement for locking not only substantially compromises performance but is rife with problems such as race conditions, deadlocks, and other horrors. On August 14, 2003, for example, a race condition arose in General Electric Energy’s Unix-based energy management system that caused a power outage affecting 55 million people. Once triggered, the bug stalled FirstEnergy’s control room alarm system for over an hour. The failure deprived the system operators of both audio and visual alerts. Thus, when a series of faults caused by tree branches touching power lines in Ohio went unnoticed, a cascade of failures was the result. This cascade led to the most widespread blackout in North American history.

Nevertheless, and this is really important, threads of this class can be made to run simultaneously. Performance is paramount in high-end, compute-intensive application software, and preemptively scheduled threads are one of the best ways to satisfy high-performance requirements. Ironically, this class of threads is what many of us think of when discussions about multithreading arise, yet it is the model least applicable to general software.

### Non-preemptive threads

In the non-preemptive model, a thread executes until it [programmatically] yields the processor to another thread or terminates. In either case, the scheduler then submits the next thread eligible for execution to the run queue.

### Concurrency

Programmers can achieve concurrency in two ways: multitasking and simultaneity (a.k.a., in parallel processing). Given two tasks, concurrency occurs when a cook works on both tasks by switching from one to the other (*single-threaded*, *asynchronous execution*). The second way to achieve concurrency is to assign a cook to each task, but in the absence of multiple kitchens, they must alternate. So, when the first cook reaches a point where s/he has to wait for the oven temperature to get to 400 degrees, s/he turns over the kitchen to the second cook (*multithreaded asynchronous execution).* This model is common in event-driven applications because the tasks in these environments are often *wait-blocked*. For example, suppose a thread’s task is to retrieve a record from a table of records, and the table is empty. In this example, the thread’s task cannot be completed until at least one record appears. The task is, therefore, wait-blocked.

This second model has become the most common strategy for achieving asynchronous, concurrent execution for programs whose tasks are subject to indeterminate waiting.

## Thread Safety In WoWThreads

The requirement for thread safety in Blizzard’s addon execution environment can arise in two situations. The first situation occurs when a programmer implements code that yields the processor to the next scheduled thread prior to exiting a critical section. The second can arise when two or more independent addons share data. The former is easily avoided, and the second is very rare[[8]](#footnote-8).

# The WoW Multithread Library API

The WoW Multithread Library (code name, Talon) is a collection of services that enable WoW addon developers to incorporate multiple, asynchronous, non-preemptive threads into their addons. Talon threads can increase an addon’s concurrency and reduce coding complexity[[9]](#footnote-9) , making debugging and support easier.

## Design Considerations

### Patterns

In keeping with the analogy of how eggs and toast are made (see above), Talon programmers assign a thread to make the eggs and a thread to make the toast. In so doing, they arrange for each thread to pause its execution at appropriate times so the other threads can do their work. In other words, Talon employs multiple threads, each of which has a specific task to perform. If that task has to wait (e.g., the eggs to cook), it can yield to let other threads execute their tasks (clean the kitchen, sharpen knives, and … cook the toast).

For example, consider how Talon might handle WoW events: one thread may do nothing more than wait for events to be received (e.g., from the *OnEvent()* service) and then notify an event-handler thread to process the event. As it happens, this design pattern is one of the most common patterns in gaming applications and is pretty straightforward. When tasks arise, say in the form of events, the receiver signals a handler thread that an event is ready for processing. The handler thread then “wakes up,” fetches the event and its payload, and then processes the event accordingly.

Here’s a typical *OnEvent()* snippet in which Talon’s *thread:sendSignal()* service is used to notify a thread that a combat event has just arrived and is available for processing (i.e., inserted into a table).

-- this is executed by the WoW client

local function OnEvent( self, event, ... )

local eventPayload = {CombatLogGetCurrentEventInfo()}

local subEvent = eventPayload[2]

if subEvent == “SPELL\_DAMAGE” then

table.insert( spellDamageTable, eventPayload )

local result = thread:sendSignal( spellDmg\_h, SIG\_ALERT )

if not result[1] then thread:postResult( result ) return end

end

end --- OnEvent() returns

In this pattern, when the event fires, the event payload is delivered to the addon through the OnEvent() service, which then signals a handler thread (spellDmg\_h) to process the event. This pattern is illustrated in the snippet above. Once the signal is sent, the addon returns from the client’s OnEvent() service leaving the spellDmg\_h thread to process and log the event.

This is probably the most common design pattern for using multiple threads to handle asynchronous events. More generally, this pattern is one of a small collection of standard problems in multithreaded programming called the producer-consumer pattern. In this snippet, the WoW client ‘produces’ an event that is ‘consumed’ by an addon. Asynchrony happens because the producer of the event doesn’t have to process the event. It just hands the event to another thread and goes back to retrieving more events.

### Code Structure

The art and practice of multithreading a WoW addon involve, in part, understanding the distinction between thread context and WoW client context. Code executed by a Talon thread is said to be executed in a thread context. Code executed by the WoW client is said to be running in the client context. Why is this important?

As it happens, while all Talon services can be executed by a Talon thread, only some can also be executed in both WoW client and thread contexts. For example, in the snippet above, the *thread:sendSignal()* can be executed both by non-threaded code (the WoW client) and threaded code. On the other hand, the corresponding *thread:getSignal()* service, cannot be executed by the WoW client because the WoW client is not programmed to receive Talon signals. In other words, the WoW client can signal one of its threads, but a thread cannot signal the WoW client. Programmers must be aware of which services can be executed by client code, Talon code, or both.

Because ALL Talon services can run in a thread context, my practice is to create a main[[10]](#footnote-10) thread that creates the addons threads. like so:

local function main(…)

local signal = SIG\_NONE\_PENDING

<… initialize addon state …>

<… create all the threads necessary to process addon tasks …>

while signal ~= SIG\_TERMINATE do

thread:yield()

local signal, sender\_h = thread:getSignal()

<process event/task>

end

end

This initialization code is usually executed when the addon is loaded.

local yieldInterval = 3.0 -- seconds

local main\_h, result = thread:create( yieldInterval, main )

if not result[1] then thread:postResult( result ) return end

This is by no means required. A couple of thoughts on the snippets: The while-loop contains two thread services, *thread:yield()* and *thread:getSignal()*. Both of these only execute in a thread context – in this case, within the main thread, *main\_h*. The purpose of the loop is to prevent the main thread from exiting[[11]](#footnote-11). In fact, it will not exit until another thread sends it a SIG\_TERMINATE signal.

Here is an example of how to shut down the addon.

local function main()

local signal = SIG\_NONE\_PENDING

<… create one or more threads to process addon tasks …>

while signal ~= SIG\_TERMINATE do

thread:yield()

signal, sender\_h = thread:getSignal()

< do while loop specific stuff>

end

signal, sender\_h = thread:getSignal()

if signal == SIG\_TERMINATE then

<terminate each thread using SIG\_TERMINATE>

end

end

## Getting Started

### Installation

<Not Ready for Prime Time>

## Threads

A WoW thread is a coroutine initialized to execute a function call along with additional state information that enables the library’s dispatcher to schedule the thread’s execution. The additional state information is contained within a table called the thread handle (see below). My personal naming convention is to use *thread\_h* for a thread handle. More specifically, I append an \_h to any thread handle name. For example,

local self\_h, selfId = thread:self()

In common threading parlance, the thread’s function is usually referred to as the *thread function, thread procedure,* and the *start* or *action routine.* Also note that in this guide, a thread is a composite object consisting of its handle, *thread\_h,* and a coroutine initialized to execute the thread function, *threadFunc()*.

### The Thread Handle

The thread handle is a table of attributes and state variables that are used to uniquely identify a thread and manage its semantics. The thread handle is opaque to the client code. Below is the handle table:

|  |  |
| --- | --- |
| **Element** | **Description** |
| TH\_EXECUTABLE\_IMAGE | The initialized coroutine |
| TH\_SEQUENCE\_ID | The thread’s unique [integer] Id |
| TH\_SIGNAL\_QUEUE | A table of signals queued for this thread. |
| TH\_TICKS\_PER\_YIELD | The number of ticks specified in thread:create() |
| TH\_REMAINING\_TICKS | The number of ticks remaining before the thread can be dispatched. |
| TH\_YIELD\_COUNT | The number of times the thread has returned from thread:yield() |
| TH\_LIFETIME | The number of ticks between thread creation and thread completion |
| TH\_ACCUM\_YIELD\_TIME | The total time (milliseconds) the thread has been suspended |
| TH\_JOIN\_DATA | Data is to be delivered to threads that have joined with this thread. |
| TH\_JOIN\_QUEUE | A queue of threads joined with this thread. |
| TH\_CHILD\_THREADS | A table of all threads created by this thread |
| TH\_PARENT\_THREAD | The thread that created this thread. |
| TH\_EXECUTION\_STATE | The execution states of the thread are either “running,” “suspended,” “completed,” or “queued.” |

### Thread Creation

When a thread is created, its state variables (used by the dispatcher to control its execution) are stored in the thread handle. Upon successful completion, the handle is returned to the caller. The code snippet below is the service that creates and initializes a thread and returns its handle, ***thread\_h*** (bolded), the caller.

Local **thread\_h**, result = thread:create( yieldInterval, threadFunc, … )

### The Thread Control Block

The *Thread Control Block* (TCB) is a table of thread handles. When a thread is created, its handle is inserted into the TCB. The thread ceases execution when its function runs to completion or until it suspends itself via a call to *thread:yield(), thread:join(),* or *thread:delay().* Upon entering any of these three calls, the thread enters the *suspended* state.

## The Thread Scheduler

The scheduler runs during every clock interval (the default interval is the reciprocal of your computer’s frame rate[[12]](#footnote-12)). During this interval, the dispatcher examines the state of each thread in its TCB and does the following:

1. For each thread in the “suspended” state, the dispatcher decrements its remaining tick count (TH\_REMAINING\_TICKS). When the tick count reaches zero, the original tick count is replenished, and the thread is queued for execution. The start routine of the newly resumed thread will continue execution at the point where it yielded.
2. For each thread in the “running” state, no action is taken.
3. For each thread in the “completed” state, the dispatcher moves the expired thread to the *graveyard* (a table of thread handles whose coroutines are in the “dead” state).

# Signaling – Thread-to-Thread Communication

In Talon, threads communicate via signals. When a thread sends a signal to a target thread, that signal is inserted into the target thread’s signal queue. Signal send and receive semantics are as follows:

1. No semantics are attached to a signal’s name. Programmers are responsible for how their addon responds to different signals. For example, when a thread receives a SIG\_TERMINATE signal, the addon is NOT required to terminate the receiving thread.
2. The signal queue can accommodate multiple signals, and programmers should structure signal retrieval in a while loop to ensure all signals are retrieved. When iterating over the signal queue, the last signal is always the SIG\_NONE\_PENDING signal.
3. When a signal is delivered to a target thread, its yield interval is canceled and the target thread is immediately resumed.
4. Signals sent to threads whose thread function has been completed result in an illegal operation. These constants are found in the file, WowThreads.lua

In Talon, threads communicate via signals. Signal handling is implemented by the following three services:

|  |  |
| --- | --- |
| Service | Function |
| thread:getSignalName() | Get a string representation of a signal’s name. |
| thread:sentSignal() | Send a signal to another thread |
| thread:getSignal() | Get a pending signal |

The following signals are currently supported in WoWThreads:

|  |  |
| --- | --- |
| SIG\_ALERT | Upon receipt, the signaled thread is immediately scheduled for execution. Upon regaining the processor, the thread should continue processing within its while loop. This signal is intended for use in time-critical applications. |
| SIG\_TERMINATE | Upon receipt, the signaled thread is immediately scheduled for execution. Upon regaining the processor, the thread should exit its while loop and return from its function via normal return semantics (i.e., calling *return* or *thread:exit().* |
| SIG\_METRICS | Not yet been implemented. |
| SIG\_JOIN\_DATA\_READY | Upon receipt, the signaled thread is immediately scheduled for execution. This signal is automatically sent to all threads that have previously joined a “producer” thread. The signal is sent when a producer thread completes its function and calls thread:exit(). |
| SIG\_NONE\_PENDING | No signals pending |

# API Entry Points

These functions are defined in the WoWThreads.lua, Manager.lua, MsgFrames.lua, and Errors.lua files

## thread:create()

Creates a new thread of execution.

|  |  |
| --- | --- |
| SIGNATURE: | local thread\_h, result = thread:create( ticks, func [, parameter list] ) |
| PARAMETERS: | **ticks**: the time, in clock intervals (a.k.a. ticks), a thread will wait after calling thread:yield(). A tick is equal to the reciprocal of the computer’s framerate. At 60 FPS, this equals about 16.7 milliseconds. Sixty ticks are about a second.  **func**: the function the thread is to execute.  **parameter list**: an optional variable argument list for the thread’s function, func |
| RETURNS: | **thread\_h**: a handle to the newly created thread  **result**: a result table containing error information, if any. |
| EXAMPLE 1: | yieldInterval = 60 -- about 1 second  local thread\_h, result = thread:create( yieldInterval, myFunc ) |
| EXAMPLE 2: | local thread\_h, result = thread:create( 60, printString, "Hello world!") |

## thread:yield()

Suspends the calling thread until its yield interval expires.

|  |  |
| --- | --- |
| SIGNATURE: | thread:yield() |
| PARAMETERS: | none |
| RETURNS: | none |
| EXAMPLE 1: | thread:yield() |

## thread:wait()

Instructs the dispatcher to suspend the calling thread for a specified number of clock intervals. This service differs from thread:yield() in that the programmer can delay a thread an arbitrary number of clock intervals.

|  |  |
| --- | --- |
| SIGNATURE: | thread:wait( waitTime ) |
| PARAMETERS: | **waitTime**: the time expressed in clock intervals the calling thread is to suspend itself. |
| RETURNS: | none |
| EXAMPLE 1: | local waitTime = 30 -- about ½ second  thread:wait( 30 ) |

## thread:join()

Suspends the calling thread until the specified thread’s data is ready.

|  |  |
| --- | --- |
| SIGNATURE: | local result = thread:join( thread\_h ) |
| PARAMETERS: | **thread\_h**: Thread handle of thread for which to wait (a.k.a. the producer thread) |
| RETURNS: | **result**: a result table containing error information, if any. |
| EXAMPLE: | local result = thread:join( producer\_h ) |

## thread:exit()

This function is called in lieu of return and is used to pass data to threads that have joined the caller. In the example below, the producer thread passes its data to thread:exit(). Internally, thread:exit() sends a SIG\_JOIN\_DATA\_READY signal to all waiting (joiner) threads.

|  |  |
| --- | --- |
| SIGNATURE: | thread:exit( joinData ) |
| PARAMETERS: | **joinData**: data to be returned to waiting threads |
| RETURNS: | none |
| EXAMPLE: | local function producer()  < do stuff >  local joinData = getData()  thread:exit( joinData )  end  local thread\_h, result = thread:create( yieldInterval, producer() ) |

## thread:self()

Returns the handle and thread Id of the calling thread.

|  |  |
| --- | --- |
| SIGNATURE: | local self\_h, selfId = thread:self() |
| PARAMETERS: | none |
| RETURNS: | **self\_h**: handle of the calling thread.  **selfId**: the numerical (unique) Id of the calling thread. |
| EXAMPLE: | local self\_h, selfId = thread:self() |

## thread:getId()

Returns the numerical Id of the specified thread. If no thread is specified, then the thread Id of the calling thread is returned.

|  |  |
| --- | --- |
| SIGNATURE: | local threadId, result = thread:getId( [thread\_h] ) |
| PARAMETERS: | **thread\_h**: handle of the thread whose Id is to be obtained |
| RETURNS: | **threadId**: the numerical (unique) Id of the calling thread.  **result**: a result table containing error information, if any. |
| EXAMPLE: | local selfId, result = thread:getId()  local threadId, result = thread:getId( thread\_h ) |

## thread:areEqual()

Returns true if the two threads are identical

|  |  |
| --- | --- |
| SIGNATURE: | local areEqual = thread:areEqual( thread1\_h, thread2\_h ) |
| PARAMETERS: | **thread1\_h**: thread handle to be evaluated  **thread2\_h**: thread handle to be evaluated |
| RETURNS: | **areEqual**: (boolean) true if both threads are the same, false otherwise. |
| EXAMPLE: | local areEqual = thread:areEqual( thread1\_h, thread2\_h ) |

## thread:getParent()

Obtains the handle of the thread that created the specified thread (a.k.a. the parent thread).

|  |  |
| --- | --- |
| SIGNATURE: | local parent\_h, result = thread:getParent( [thread\_h] ) |
| PARAMETERS: | **thread\_h**: the thread handle whose parent is to be returned. If not specified, the parent of the calling thread is returned.  NOTE: threads created by Blizzard’s WoW client do not have parent threads. |
| RETURNS: | **parent\_h**: the parent of the specified thread. If no parent exists (i.e., the specified thread was created by the WoW client, nil is returned.  **result**: a result table containing error information, if any |
| EXAMPLE: | local parent\_h, result = thread:getParent()  local parent\_h, result = thread:getParent( thread\_h ) |

## thread:getChildren()

Obtains the handle of the thread that created the specified thread (a.k.a. the parent thread).

|  |  |
| --- | --- |
| SIGNATURE: | local childTable, result = thread:getChildren( [thread\_h] ) |
| PARAMETERS: | **thread\_h**: handle of the thread whose children (if any) are to be obtained. If not specified, the children of the calling thread are returned. |
| RETURNS: | **childTable**: if the specified thread has one or more child threads, a handle for each child thread is returned in a table of thread handles. If no child thread(s) exist, nil is returned.  **result**: a result table containing error information, if any |
| EXAMPLE: | local childTable, result = thread:getChildren()  local childTable, result = thread:getChildren( thread\_h ) |

## thread:getState()

Obtains the execution state of the thread

|  |  |
| --- | --- |
| SIGNATURE: | local state, result = thread:getState( thread\_h ) |
| PARAMETERS: | **thread\_h**: handle of the thread whose state is to be obtained. NOTE: by construction, the calling thread is ALWAYS in the “running” state. |
| RETURNS: | **state:** an enumerated set of 3 values: “suspended," “queued,” or “completed.”  **result**: a result table containing error information, if any |
| EXAMPLE: | local state, result = thread:getState( thread\_h ) |

## thread:getSignalName()

A utility service that returns the name of the input signal

|  |  |
| --- | --- |
| SIGNATURE: | local signalName, result = thread:getSignalName( signal ) |
| PARAMETERS: | **signal:** the signal whose name (string) is to be returned. |
| RETURNS: | **signalName:** a string representation of the signal’s name.  **result**: a result table containing error information, if any |
| EXAMPLE: | local signal, sender\_h = thread:getSignal()  local state, result = thread:getState( sender\_h )  if not result[1] then thread:postResult( result ) return end |

## thread:sendSignal()

Sends a signal to the target thread.

|  |  |
| --- | --- |
| SIGNATURE: | local signal, sender\_h = thread:getSignal( thread\_h, signal ) |
| PARAMETERS: | **thread\_h:** the thread to receive the signal (i.e., the “target thread”).  **signal:** the signal to be sent. |
| RETURNS: | **result**: a result table containing error information, if any |
| EXAMPLE: | local result = thread:sendSignal( thread\_h, SIG\_ALERT )  if not result[1] then thread:postResult( result ) return end |

## thread:getSignal()

Checks to see if a signal is pending. If so, the signal and its sending thread are returned.

|  |  |
| --- | --- |
| SIGNATURE: | local signal, sender\_h = thread:getSignal() |
| PARAMETERS: | none |
| RETURNS: | **signal:** the pending signal. If more than one signal is pending, the calling thread must iterate over the signal queue (see the example below). If no signal is pending, then SIG\_NONE\_PENDING is returned.  **sender\_h**: the thread that sends the signal. The semantics are somewhat complex:   1. If sender\_h is nil, then the sending thread was the WoW client, i.e., not a thread. 2. If sender\_h is not nil, then the sender may be in one of two states: “completed” or “suspended.” If “completed,” the thread sent the signal and completed its function before the calling thread retrieved the signal. If “suspended,” the thread is “suspended” and available for all operations. |
| EXAMPLE: | Example 1:  local signal, sender\_h = thread:getSignal()  local state, result = thread:getState( sender\_h )  if not result[1] then thread:postResult( result ) return end  Example 2: thread has multiple threads in its signal queue.  local signal, sender\_h = thread:getSignal()  while signal ~= SIG\_NONE\_PENDING do  < do some signal-specific stuff >  signal, sender\_h, thread:getSignal()  end |
|  |  |

# Error Handling

These two functions are found in threadErrors.lua and MessageFrames.lua, respectively. In my programs, I usually set E equal to threadErrors. Thus, the following services can be coded as follows:

local E = threadErrors

local result = E:setResult( result. debugstack() )

E:dbgPrint()

## threadErrors:setResult()

Initializes and returns an error result table.

|  |  |
| --- | --- |
| SIGNATURE: | local result = threadErrors:setResult( errorMsg, stackTrace ) |
| PARAMETERS: | **errorMsg**: a string error whose message describes the error.  **stacktrace**: a stacktrace generated by debugstack(), usually passed into the function as the second parameter (see below). |
| RETURNS: | **result**: a result table containing error information, if any. The setResult() function ALWAYS sets the status element to FAILURE and ALWAYS generates a stack trace (via debugstack() at the location where setResult() was called.  result = {  status,  errorMsg, -- supplied by the caller  stackTrace -- debugstack()  { |
| EXAMPLE: | local function someFunction()  local result = {SUCCESS, EMPTY\_STR, EMPTY\_STR }  local returnValue = somefunction()  if not returnValue then  result = E:setResult( L[“INCORRECT\_VALUE”], debugstack())  return result  end  end |

## thread:postResult()

Displays the contents of a result table in a scrolling text frame.

|  |  |
| --- | --- |
| SIGNATURE: | thread:postResult( result ) |
| PARAMETERS: | **result:** a result table containing error information. |
| RETURNS: | none |
| EXAMPLE: | local E = threadErrors  local result = {SUCCESS, EMPTY\_STR, EMPTY\_STR }  result = someFunction()  if not result[1] then thread:postResult( result ) return end |

# Debugging Support

These services are found in threadErrors.lua

## threadErrors:dbgPrint()

Displays the location, i.e., filename and line number, from where called

|  |  |
| --- | --- |
| SIGNATURE: | local fileLocation = threadErrors:dbgPrint( [msg] ) |
| PARAMETERS: | **msg:** an optional string. |
| RETURNS: | none |
| EXAMPLE: | If called in MyFile.lua at line number 85, the following will be printed to the DEFAULT\_CHAT\_FRAME.  threadErrors:dbgPrint()  [MyFile.lua:85]  or  threadErrors:dbgPrint( “Hello world! )  [MyFile.lua:85] Hello, world! |

## threadErrors:prefix()

Equivalent to dbgPrint() except that prefix() returns the location string. Its intended use is to embed location information in other strings.

|  |  |
| --- | --- |
| SIGNATURE: | local fileLocation = threadErrors:prefix() |
| PARAMETERS: | none |
| RETURNS: | **fileLocation:** “[Filename:LineNo]” |
| EXAMPLE: | E = threadErrors  local str = sprintf(“%s Hello, world!”, E:prefix() )  print ( str )  [MyFile.lua:85] Hello, world! |

# Display Service

This function is found in MessageFrames.lua

## thread:postMsg()

Display a user/programmer message in a scrolling window frame.

|  |  |
| --- | --- |
| SIGNATURE: | thread:postMsg( msg ) |
| PARAMETERS: | **msg:** a user-defined, usually informative, message. |
| RETURNS: | none |
| EXAMPLE: | thread:postMsg( “Hello world! ) |

# Management Services

These functions are found in the file, Manager.lua

## thread:getCongestion()

Calculates a set of metrics for each completed thread from which thread congestion can be calculated (note: a thread must have completed before its congestion metrics can be obtained.

|  |  |
| --- | --- |
| SIGNATURE: | entry, entriesRemaining, result = thread:getCongestion( thread\_h ) |
| PARAMETERS: | thread\_h: |
| RETURNS: | **entry**: the entry returned for the specified thread.  table = { threadId, -- numerical (unique) Id of the thread.  ticksPerYIeld, -- specified at thread creation  yieldCount, -- number of thread:yield() completions  timeSuspended, -- total time spent in a suspended state  threadLifeTime, -- time from creation to completion  congestion -- the extent to which threads conflict  }  **entriesRemaining**: the number of metric entries remaining in the metrics table.  **result**: a result table containing error information, if any |

### Thread Congestion – discussion

Congestion is defined as the overhead imposed on a specific thread due to the presence of other threads competing for the system processor. In a perfect world, congestion is given by the following formula:

Congestion = [1 – (lifetime with no other threads) / (lifetime with multiple threads)]

The two main variables that influence a thread’s congestion are:

1. The duration of the yield interval. Short yield intervals lead to higher congestion
2. The number of active threads in the addon. The more threads, the higher the congestion.

A fairly accurate assessment of a thread’s congestion can be obtained from the information in the thread’s entry in the congestion table (see above). Here are two simple tests, the first uses a 10-tick (167 ms) yield interval, and the second uses a 40-tick interval. The congestion seen in the first test is much higher (and is more variable). A congestion example can be found in the file congestion.lua. When executed, the code prints the result. In this test, the experiment was to measure the congestion difference when five threads were run with a yield interval of ten ticks and with forty ticks.

Test 1:

\*\*\* THREAD CONGESTION TEST \*\*\*

Thread 13

ticks per yield: 10

time suspended: 1499.25 ms

Lifetime: 1548 ms.

Congestion: 3.202%

Thread 11

ticks per yield: 10

time suspended: 1499.29 ms

Lifetime: 1532 ms.

Congestion: 2.171%

Thread 10

ticks per yield: 10

time suspended: 1499.25 ms

Lifetime: 1515 ms.

Congestion: 1.101%

Thread 12

ticks per yield: 10

time suspended: 1499.27 ms

Lifetime: 1515 ms.

Congestion: 1.093%

Thread 14

ticks per yield: 10

time suspended: 1499.24 ms

Lifetime: 1515 ms.

Congestion: 1.090%

Test 2:

\*\*\* THREAD CONGESTION TEST \*\*\*

Thread 13

ticks per yield: 40

time suspended: 5997.82 ms

Lifetime: 6047 ms.

Congestion: 0.826%

Thread 11

ticks per yield: 40

time suspended: 5997.85 ms

Lifetime: 6031 ms.

Congestion: 0.551%

Thread 10

ticks per yield: 40

time suspended: 5997.81 ms

Lifetime: 6014 ms.

Congestion: 0.279%

Thread 12

ticks per yield: 40

time suspended: 5997.84 ms

Lifetime: 6014 ms.

Congestion: 0.277%

Thread 14

ticks per yield: 40

time suspended: 5997.76 ms

Lifetime: 6014 ms.

Congestion: 0.276%

# Appendix

The WoWThreads library described in this book will be freely available at

<https://github.com/mtp1032/WoWThreads>

## Installation Examples

<To Be Supplied>

# Threads In Perspective

[Threads are Evil: Parallelism meets programming reality](http://becpp.org/blog/wp-content/uploads/2018/01/Frederik-Vannoote-Threads-are-evil.pdf)

[The Problem with Threads](https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf)

## Example Application(s)

<To be Supplied>

# About the Author

I began playing World of Warcraft shortly after its first release (Vanilla) and have leveled at least one character to max in every expansion since.

I hold a Ph.D. in a field completely unrelated to programming (immunology) that, as such, confers absolutely no credibility insofar as programming is concerned. On the other hand, I am a retired software architect with the following experience:

* Architect and lead developer of Digital Equipment Corporation’s distributed UNIX-based off-line storage facility (Backup/Restore, Disaster Recovery, etc.)
* Lead developer and architect for the disaster recovery facility[[13]](#footnote-13) in the Windows NT operating system kernel.
* Wrote the first POSIX threads implementation for Linux[[14]](#footnote-14).
* Author of DCE: A Guide To Developing Portable Applications[[15]](#footnote-15). This book evolved, in part, from my association with the development of Concert Multithread Architecture (CMA)[[16]](#footnote-16). CMA, and especially its lead developer, David Butenof. David was a formative influence on the development of POSIX threads[[17]](#footnote-17).

1. Code that execute in a kernel’s address space (e.g., drivers and file-systems). [↑](#footnote-ref-1)
2. For what it’s worth, the bag addon was in the service of learning how to adapt to express object-oriented semantics. Again, just for fun. [↑](#footnote-ref-2)
3. One of which is that the WoW client is probably threaded already. [↑](#footnote-ref-3)
4. This can be confusing because some vendors refer to kernel threads as light-weight threads. But these threads are not accessible to the application code. [↑](#footnote-ref-4)
5. Unity uses coroutines and is also a component of approximately ½ of all mobile games in the market today. [↑](#footnote-ref-5)
6. Thread preemption requires explicit hardware support in order to assign threads to specific processors or cores. Most modern operating system have kernel services that are able to trigger the hardware primitives that mediate processor assignment. [↑](#footnote-ref-6)
7. Coroutines, unlike threads, do not involve the operating system and its supporting hardware when switching between coroutines. [↑](#footnote-ref-7)
8. If anyone is aware of independent addons that share data I would be very grateful if someone would let me know. I ache to implement a mutex facility. Not! [↑](#footnote-ref-8)
9. In the formal sense: code complexity is often measured in terms of the McCabe Complexity Metric (also called Cyclomatic Complexity) and is a measure of the linearly independent code paths through a program’s source code. [↑](#footnote-ref-9)
10. I come from a C/C++ background and the use of *main()* seemed appropriate. However, Talon doesn’t care. You can name such functions anything you want. [↑](#footnote-ref-10)
11. However, because the threads are independent of one another, were main\_h to terminate, the rest of the threads would happily continue to execute. [↑](#footnote-ref-11)
12. i.e., 1/*GetFramerate()*. For a framerate of 60 FPS, this gives a tick interval of 0.0167 seconds, or about 17 ms. [↑](#footnote-ref-12)
13. Formally known as ASR (Automated System Recovery) [↑](#footnote-ref-13)
14. These were user-mode, preemptive threads. [↑](#footnote-ref-14)
15. Peterson, Michael T., DCE: A Guide To Developing Portable Applications, McGraw-Hill, 1995 [↑](#footnote-ref-15)
16. While working for Digital Equipment Corporation. [↑](#footnote-ref-16)
17. He was the author of one of the early Bibles of POSIX threads. You can click this link for further information: https://www.amazon.com/Programming-POSIX-Threads-David-Butenhof/dp/0201633922 [↑](#footnote-ref-17)