An Introduction to Multi Threaded Programming with POSIX Threads and Linux

Liam Widdowson [lbw@telstra.com]
Hewlett-Packard Consulting

January 12, 2001

Contents

1 Preface

1.1 General Portability

New open source applications are being released each day. Many of the programmers who develop open source software may do so on a single platform such as Linux. Far too much open source software is not portable out of the box. Such software typically requires non-trivial changes to compile or work reliably on other operating systems.

Portability is a key component to producing good quality, reliable software. Developing and testing software on multiple platforms helps iron out obscure bugs that many seldom be seen on a single hardware architecture or operating system.

It is a disservice to the developer's efforts if no attempt is made to make software portable. Wherever possible, a developer should use what is available in the POSIX APIs and not unnecessarily use operating system specific system calls or third party libraries. If system specific functions must be used, they should be placed in pre-processor condition statements. When designing software, thought should be given to how the software may behave on other software and hardware platforms (e.g different byte order).

Our great and fearless leader Linus Torvalds one said:

> *Porting this new operating systems to other platforms was really not on my mind at the beginning. At first I just wanted something that would run on my 386* [1]

It is important that developers do not fall into the same trap as Linus did. This oversight meant that a great deal of internal re-development was required to get Linux to a state where it could be ported in a straight forward manner. Most of the popular open source software has been extensively ported to other platforms (e.g Sendmail, BIND, NetBSD). It could indeed be argued that portability is a key component to the success of a particular piece of software.

1.2 POSIX Threads Portability

Portability is imperative when developing with POSIX threads. The POSIX thread API abstracts its data types from the underlying data storage structures. Developers should never make any assumptions in regards to how POSIX thread data types, the scheduler or the threading model are implemented by the vendor. All POSIX thread data types should be considered *opaque* [2].

The importance of portability will become more apparent as we delve further into POSIX threads.

2 Introduction

2.1 Definition

In a general sense a thread can be considered as a scheduling entity. More specifically consider a thread as an independent flow of control within a scheduling entity. In a typical Unix implementation, a thread will share the same address space, file descriptors, text and data segments of its parent process. Each thread has its own private stack, register context and program counter.

2.2 Brief History

The concept of a thread (as an independent flow of control) dates back to 1965 with the Berkeley Timesharing System. However, at that time they were known as processes, not threads [3]. Processes interacted with what would now be considered traditional Inter-Process Communication (IPC) means such as semaphores and message passing. One of the first thread implementations was developed on Multics at Bell Laboratories during early 1970. It used multiple stacks in a single process to support multiple background compilations [4]. In the early 1980s, micro-kernel based operating systems such as Amoeba and Chorus provided *lightweight processes* that shared the address space of a single standard process [4].

Threads as we have come to know them today were created as a part of the Open Software Foundation's Distributed Computing Environment. OSF's DCE threads is a complete user space implementation based on an early POSIX threads draft. It has been ported extensively to many operating systems thanks to its modular design. DCE could typically be ported to a new operating system in a matter of days with *some assembly required* for context switch routines [2]. Today, threads are defined by the POSIX 1003.1c-1995 standard. Extensions to the standard have been defined by The Open Group as part of the Single UNIX Specification V2. Additionally, many vendors have added non-portable system specific extensions.

2.3 Motivation

Most non-trivial applications must perform several tasks at once. Of those applications some or all of those tasks may be independent of each other and may lend themselves to parallel execution. A mail transport agent provides an excellent example - a traditional implementation uses a listener process that waits for requests and *forks* a new process to service each new client. Such an architecture has some disadvantages - forking each process adds significant overhead as *fork(2)* is an expensive system call (even if copy-on-write functionality is provided within the kernel) [5]. As each process has it's own address space it must use standard IPC facilities such as semaphores or shared memory to communicate. Some of these calls are relatively expensive and retard the application's performance.

Threads address many of the aforementioned issues. Threads have a common address space which provides simplified data sharing and lightweight process creation. Additionally, threading environments provide advanced data types which assist in providing a more natural style of parallel programming.

3 Thread Models

There are three generally accepted models used in the implementation of threads on Unix variant operating systems. Each model has its own set of advantages and disadvantages. The following section examines the three models.

3.1 User Space Threads

3.1.1 Overview

A multi-threading sub system may be implemented entirely in user space. This model is typically referred to as M x 1 threads (i.e M user space threads are contained within one kernel space scheduling entity). Typically, implementations of this model are based on the POSIX threads draft 4. OSF's DCE is one such implementation. A user space library creates, terminates, schedules and synchronises threads. These threads are not directly visible to the kernel, in fact, the kernel is unaware that the single process which holds the threads is any different from other processes.

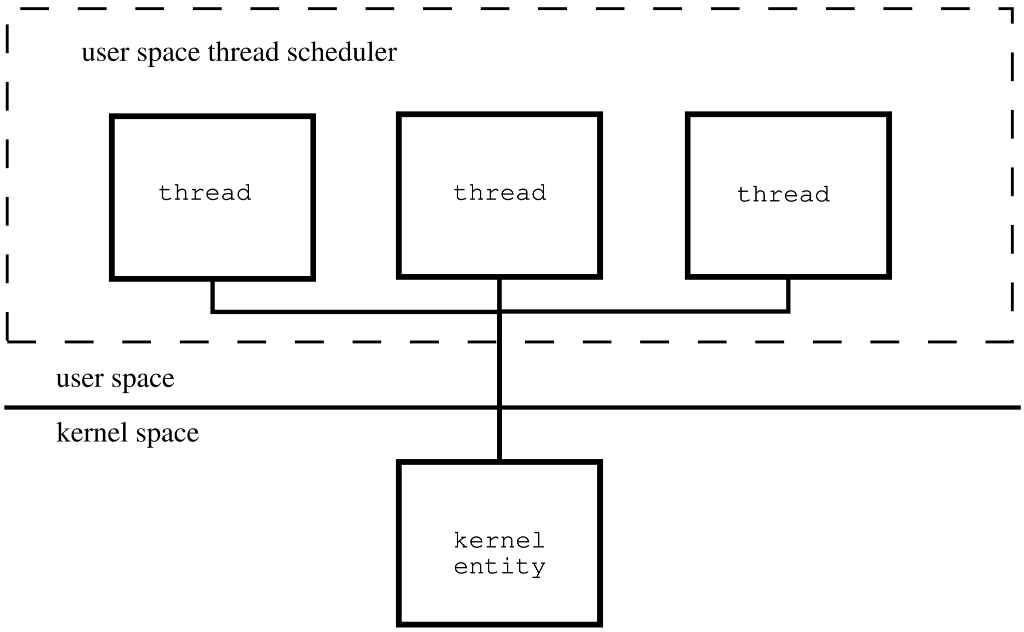
This model provides a performance benefit as each thread context switch does not traverse the user/kernel space boundary. However, there are also disadvantages associated with this model - All threads reside within a single kernel space scheduling entity (typically a Unix process). This means that a only single processor may be utilised to schedule all threads. Within uniprocessor environments this may not represent a significant issue, however in multi-processor environments this model may be unacceptable.

Further, if a single thread blocks on a system call such as *read(2)*, the entire process and all associated threads block. Many user space libraries help alleviate this problem by using non-blocking I/O. However, there are still performance and compatibility issues associated with this work-around (e.g not all interfaces support non-blocking I/O).

Essentially, a user-space thread library can provide *concurrency* (i.e the appearance of multiple tasks being performed at once) but can never provide *parallelism* (i.e multiple tasks performed at the same instance in time on separate processors).

Operating systems which implement this threading model include:
FreeBSD and OpenBSD

3.1.2 Architecture



Advantages	*Disadvantages*
Context-switches performed entirely in user mode	Poor performance due to blocking system calls and/or associated code complexity to avoid them
Straight forward implementation	Unable to use multiple processors

3.2 Kernel Space Threads

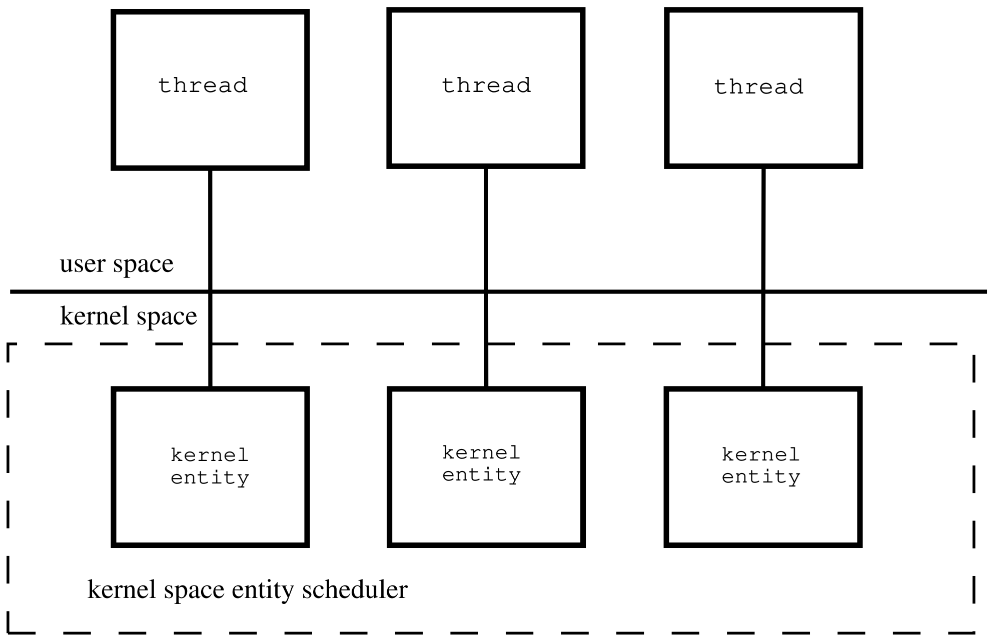
3.2.1 Overview

Unlike user space threads which do not involve any thread specific kernel interaction, kernel space threads rely on it. The kernel space thread model is often referred to as the 1 x 1 thread model (i.e one thread maps to one kernel space entity). Typically, implementations may map each single thread to a single kernel entity such as a standard Unix process or a more lightweight *kernel thread*.

Each thread is scheduled independently by the kernel scheduler so if a thread blocks it will not effect any other thread. However, creation, synchronisation and termination of kernel threads may suffer some performance problems when compared to user space threads. The kernel must be involved in operations such as requesting a memory lock or thread scheduling. Further, each kernel thread or process takes up valuable kernel resources which inhibits the creation of a large number of threads (most operating systems will only allow an arbitrary number of threads to be created system wide). Whilst the overhead may be greater, kernel threads do offer true *parallelism* as each thread may run independently on separate physical processors.

Operating systems which implement this threading model include:
HP-UX 11 and Linux

3.2.2 Architecture



Advantages	*Disadvantages*
Able to use multiple processors	Context switches performed entirely in kernel mode
Improved performance when coping with blocked system calls, etc	Decreased performance with large number of threads

3.3 Kernel and User Space Threads

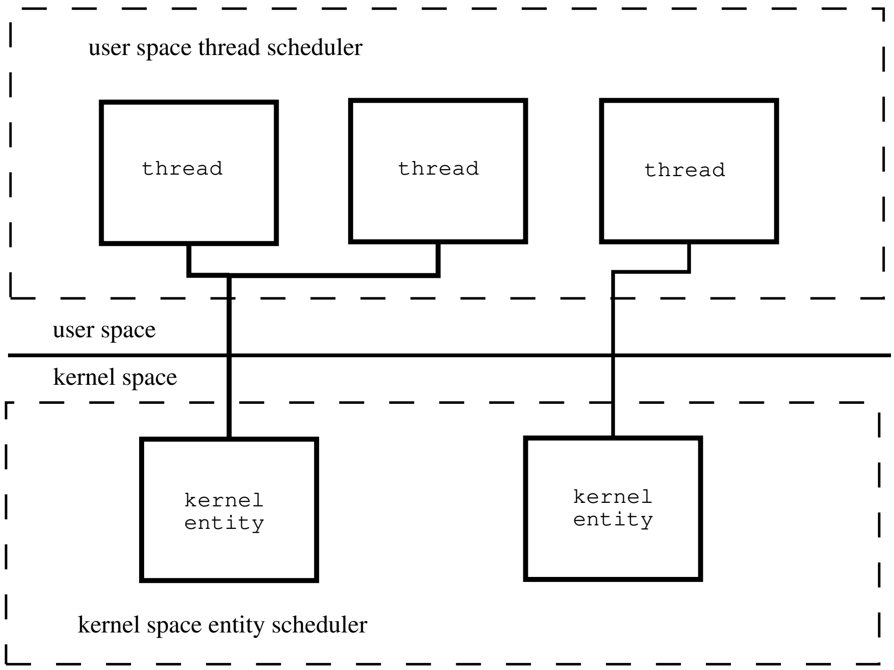
3.3.1 Overview

Kernel and user space threads are often referred to as M x N threads (i.e M user space threads within N kernel entities). The kernel and user space threading model provides a "best of both worlds" approach.

This model requires co-operation between the kernel scheduler and a user space thread library/scheduler. Essentially, sets of threads are *bound* to individual kernel entities. This allows the majority of context-switches to occur in user mode whilst still using multiple kernel entities to provide *parallelism*. For example, an application may have nine threads which are spread across four kernel entities. This would allow the application to take advantage of multi-processor hardware whilst still traversing the user-kernel boundary as little as possible. In situations where a thread blocks in a system call, another thread is immediately scheduled thus not halting the execution of threads that are bound to the same kernel entity.

Operating systems which implement this threading model include:
IRIX, Solaris and Tru64

3.3.2 Architecture



Advantages	*Disadvantages*
Able to use multiple processors	Complex system implementation
Improved performance with blocked system calls, etc	
Most context switches in user mode	

4 LinuxThreads

4.1 Overview

Linux implements the 1 x 1 kernel space threading model. Thread creation at a kernel level is performed with *clone(2)* system call. Under Linux, the *clone(2)* system call is a generalisation of *fork(2)*. Thus a thread under Linux is not significantly different from a standard heavy-weight Unix process. When a process creates its first thread, a total of three threads will be active. The initial or *main()* thread (process), the newly created thread and a manager thread which helps performs scheduling activities. This is significantly different from implementations such as Solaris and Tru64 which have separate kernel and user space entities to represent threads.

Linux's kernel developers have made a specific decision to support this model as they believe it removes the complexity associated with implementing another entity and scheduler within the kernel. Threads are created, terminated and scheduled by the same code that does so for processes. Apart from providing simplicity, the rationale behind this architecture choice is that Linux has exceptionally fast context-switch code and this negates what is one of the biggest problems associated with 1 x 1 implementations - context switch performance. However, this simplicity comes with an inherent problem - Linux will never truly be POSIX 1003.1c-1995 compliant without specific support within the kernel for threads.

It is for this reason that threads under Linux do not provide a significant performance benefit as they do on other platforms. Nonetheless, threads provide a more natural parallel style suitable for parallel programming. However, keep in mind portability - a threaded application will perform significantly better than a forking application on most other operating systems.

4.2 Simple Kernel Threads Example

The following source code was posted by Linus Torvalds to the Linux kernel mailing list in 1996. It provides a basic example of how the *clone(2)* system call may be used to implement kernel threads.

linux-clone.c - Linus Torvalds

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <linux/unistd.h>

#define STACKSIZE 16384

/* signal mask to be sent at exit */
#define CSIGNAL         0x000000ff
/* set if VM shared between processes */
#define CLONE_VM        0x00000100
/* set if fs info shared between processes */
```

```
#define CLONE_FS        0x00000200
/* set if open files shared between processes */
#define CLONE_FILES     0x00000400
/* set if signal handlers shared */
#define CLONE_SIGHAND   0x00000800

int start_thread(void (*fn)(void *), void *data) {
        long retval;
        void **newstack;

        /*
         * allocate new stack for subthread
         */
        newstack = (void **) malloc(STACKSIZE);
        if (!newstack)
                return -1;
        /*
         * Set up the stack for child function, put the (void *)
         * argument on the stack.
         */
        newstack = (void **) (STACKSIZE + (char *) newstack);
        *--newstack = data;

        /*
         * Do clone() system call. We need to do the low-level stuff
         * entirely in assembly as we're returning with a different
         * stack in the child process and we couldn't otherwise guarantee
         * that the program doesn't use the old stack incorrectly.
```

```
         *
         * Parameters to clone() system call:
         *       %eax - __NR_clone, clone system call number
         *       %ebx - clone_flags, bitmap of cloned data
         *       %ecx - new stack pointer for cloned child
         *
         * In this example %ebx is CLONE_VM | CLONE_FS | CLONE_FILES |
         * CLONE_SIGHAND which shares as much as possible between parent
         * and child. (We or in the signal to be sent on child
         * termination into clone_flags: SIGCHLD makes the cloned
         * process work like a "normal" unix child process)
         *
         * The clone() system call returns (in %eax) the pid of the newly
         * cloned process to the parent, and 0 to the cloned process. If
         * an error occurs, the return value will be the negative errno.
         *
         * In the child process, we will do a "jsr" to the requested
         * function and then do a "exit()" system call which will
         * terminate the child.
         */
        __asm__ __volatile__(
                "int $0x80\n\t"        // Linux/i386 system call
                "testl %0,%0\n\t"      // check return value
                "jne 1f\n\t"           // jump if parent
                "call *%3\n\t"         // start subthread function
                "movl %2,%0\n\t"
                "int $0x80\n"          // exit syscall: ex subthread
                "1:\t"
```

```
                :"=a" (retval)
                :"0" (__NR_clone),"i" (__NR_exit),
                 "r" (fn),
                 "b" (CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND |
                      SIGCHLD),
                 "c" (newstack));

        if (retval < 0) {
                errno = -retval;
                retval = -1;
        }
        return retval;
}

int show_same_vm;

void cloned_process_starts_here(void * data) {
        printf("child:\t got argument %d as fd\n", (int) data);
        show_same_vm = 5;
        printf("child:\t vm = %d\n", show_same_vm);
        close((int) data);
}

int main() {
        int fd, pid;

        fd = open("/dev/null", O_RDWR);
        if (fd < 0) {
```

```
                perror("/dev/null");
                exit(1);
        }
        printf("mother:\t fd = %d\n", fd);

        show_same_vm = 10;
        printf("mother:\t vm = %d\n", show_same_vm);
        pid = start_thread(cloned_process_starts_here, (void *) fd);

        if (pid < 0) {
                perror("start_thread");
                exit(1);
        }

        sleep(1);
        printf("mother:\t vm = %d\n", show_same_vm);
        if (write(fd, "c", 1) < 0)
                printf("mother:\t child closed our file descriptor\n");
}
```

linux-clone.c

The output from Linus' threading example is as follows:

```
mother:  fd = 3
mother:  vm = 10
child:   got argument 3 as fd
child:   vm = 5
mother:  vm = 5
mother:  child closed our file descriptor
```

4.3 Developing with LinuxThreads

A variety of threading libraries (including user space implementations) have been developed for Linux. However, the defacto standard library is now LinuxThreads. LinuxThreads 0.8 provides a mostly compliant POSIX 1003.1c implementation and forms part of glibc 2.x. Linux initially diverges from the POSIX standard with one fundamental difference - each thread has an independent PID. This occurs because each thread is essentially a standard Unix process pretending to be a thread. POSIX requires that all threads in a process share a single PID. In turn, this violation effects the behavior of *fork(2)* and signals, none of which are entirely POSIX compliant.

The following are minimum requirements for developing threaded software with Linux: gcc 2.8, gdb 4.18, LinuxThreads 0.8, glibc 2.0.1 and kernel 2.x

The following table provides examples of common linker and compiler flags required when compiling POSIX threaded software:

Utility	*Parameter*	*Platform*
ld	-lpthread	All platforms
gcc/cc	-D_REENTRANT	All platforms
	-pthread	Free/OpenBSD
	-D_POSIX_C_SOURCE=199506L	HP-UX
	-mt (SunSoft CC only)	Solaris

5 Programming with POSIX threads

This Section outlines how POSIX threads functions and data types may be used to create multi threaded applications. Within Section 5.2 performance comparisons will be made between a forking and threaded example application on Linux, Solaris and FreeBSD. This comparison is by no means a definitive benchmark, however, it does provide some insight into the performance benefits of threaded applications. The following systems were used to test each piece of code.

Hostname	*Operating System*	*Processor(s)*	*Memory*
athena	Linux 2.2.17	AMD K6-2/300MHz	64MB
fatso	FreeBSD 4.2	Intel 100MHz Pentium	32MB
helios	Solaris 8	160MHz TuroboSPARC	96MB
prometheus	Solaris 8	2 x 150MHz HyperSPARC	256MB

5.1 POSIX thread facilities

In addition to providing facilities to create, terminate and synchronise threads and shared data, POSIX threads implementations should provide features such as:

- A per-thread *errno* global variable;
- Thread safe versions of functions such as *malloc(3)* and *free(3)*.

5.2 POSIX thread creation

A POSIX thread is created using the *pthread_create()* function. This function is defined as:

```
int pthread_create(pthread_t *thread_id, pthread_attr_t *attr,
                void *(*startroutine)(void *), void *arg)
```

The *pthread_create()* function returns 0 upon success, ENOMEM if there are insufficient resources to create the thread, EINVAL if the attributes are invalid or EPERM if there are insufficient priviladges to create the thread.

The *thread_id* pointer stores the thread's unique identifier (analogous to a process PID), *attr* stores the attributes that the new thread should possess upon creation and *startroutine* points to the function that should be called upon thread creation. Finally, *arg* is a pointer to data which should be supplied as an argument to *startroutine*. Note, only one argument is allowed so multiple data objects may be placed within a struct.

Functions which are called by *pthread_create()* should be of the form:

```
void *function(void *argument);
```

The argument may then be re-cast at a later date to the desired type. When a thread wishes to exit, it may do so in a variety of ways - either via a *return* statement or by calling the following function:

```
void pthread_exit(void *value_ptr);
```

The value of the exiting thread is lost unless the caller waits for it with the following function:

```
void pthread_join(pthread_t thread_id, void *value_ptr);
```

The *pthread_join()* function is analogous to *waitpid(2)* for processes.

Thread attributes are set by the *pthread_attr_*()* routines and define details such as what state the thread should begin execution in, scheduling information, etc. If NULL is specified as an argument to *pthread_create()*, the thread will be created with a default set of attributes. An examination of all pthread attributes is beyond the scope of this paper, but one noteworthy attribute must be mentioned. When a thread exits, its resources are not automatically *recycled*. The thread will continue to exist in a zombie state until it is joined with *pthread_join()*.

However, in some situations it may not be convenient to use *pthread_join()*. The thread's id may be lost, the caller may not wish to collect the thread's return value or suspend its execution while it waits for the thread to finish. In such a case, the thread may be created with the PTHREAD_CREATE_DETACHED attribute set. This will create the thread in a detached state. When the thread exits all resources will be automatically *recycled*.

The attribute may be set with code such as the following or by calling the *pthread_detach(pthread_t *thread_id)* function at any time:

```
pthread_attr_t  attr;
pthread_t       t;

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(&t, &attr, some_function(), (void *) some_arg);
..
or
..
pthread_detach(&t1)
```

A thread that is detached can never be joined again with *pthread_join()*. It is also worth mentioning again, that all pthread data types are *opaque*. One should never make any assumptions about the underlying data types. Further, the result of copying pthread data variables is undefined as is performing arithmetic (e.g equality) operations upon them. In order to check if two *pthread_t* thread IDs are the same, the following function should be used:

```
int pthread_equal(pthread_t thread0, pthread_t thread1);
```

The *pthread_equal()* function returns a non-zero value if the thread IDs are equal, otherwise zero is returned.

A thread's thread identifier may be retrieved at any time by using the following function:

```
pthread_t pthread_self(void);
```

The following code demonstrates thread creation by creating a pre-defined number of threads and prints a message from each. To contrast, a forking version is provided.

5.2.1 Threaded version

test2-thread.c

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#define MAX_THREADS 512

void *say_hello(void) {
        printf("[thread %d] new thread created\n", (int) pthread_self());
        sleep(1);
        return NULL;
}

int main(void) {
        pthread_t       t1;
```

```
	int		x, status;

	#ifdef __sun__
	pthread_setconcurrency( sysconf(_SC_NPROCESSORS_ONLN)+1 );
	#endif

	for (x = 0; x < MAX_THREADS; x++) {
		status = pthread_create(&t1, NULL, say_hello, NULL);
		if (status != 0) {
		fprintf(stderr, "[thread %d] error creating thread %s\n",
			(int) pthread_self(), (char *) strerror(status));
		exit(0);
		}
	}
	printf("[thread %d] this is the end\n", (int) pthread_self());
	exit(0);
}
```

test2-thread.c

5.2.2 Forking version

test2-process.c

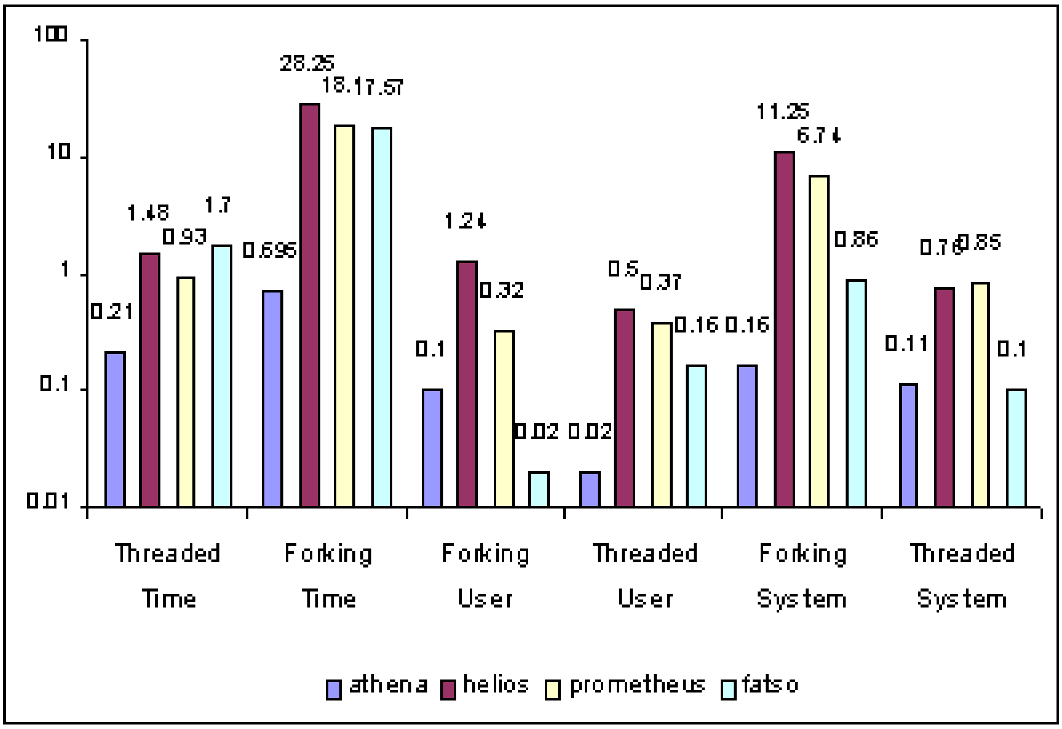
```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>
#define MAX_PROCESSES 512
```

```

void say_hello(void) {
        printf("[pid %d] I am a new child\n", (int) getpid());
        sleep(1);
}

int main(void) {
        int     x;
        pid_t   pid;
        for (x = 0; x < MAX_PROCESSES; x++) {
                pid = fork();
                if (pid == (pid_t) -1) {
                        perror("fork");
                        exit(0);
                }
                if (pid == (pid_t) 0) {
                        // in the child
                        say_hello();
                        exit(0);
                } else {
                        // in the parent
                }
        }
        printf("[pid %d] this is the end\n", (int) getpid());
        return 0;
}
```

test2-process.c

5.2.3 Performance Comparison Graph

The following graph displays the time taken by the threaded and forking test code. Note that the Y axis represents time using a logarithmic scale.



5.2.4 Performance Comparison Analysis

It is evident from the data provided that threaded applications perform significantly better. On Solaris, the threaded application performed its tasks almost twenty times faster than the forking version. It spent almost half as much time in user mode and sixteen times less time in kernel mode. Attention should be paid to the difference in results between the single and multi-processor Solaris machines. The multi-processor machine spent slightly more time in kernel mode and slightly less in user mode. This is because some of the threads could run on different processors thus slightly more kernel intervention was required for context switching and thread creation.

On Linux, the gap was much smaller, but threads still provide a performance advantage finishing their tasks three times faster whilst spending five times less time in user mode and about the same time in kernel mode.

FreeBSD's user space threads implementation also provided excellent results. The threaded application performed almost ten times faster and spent significantly less time in both user and kernel mode.

The code used in this simple experiment does not mimic the behavior of a typical application (which would be both computational and I/O bound). In such a situation a threading model which includes a kernel and user space scheduler (such as Solaris or Tru64) would most likely perform best.

5.3 Mutual Exclusion

As threads share the same address space, they must possess a synchronisation method to ensure that shared resources are not corrupted by concurrent reading and writing. POSIX threads provide a synchronisation type called a mutex. A mutex (short for mutual exclusion) may be used to protect access to any resource. A mutex locked within a single thread may not be unlocked by any other thread. A thread which attempts to acquire a locked mutex will block until it is released.

The mutex type within POSIX threads is *pthread_mutex_t* and may be initialized at deceleration with the PTHREAD_MUTEX_INITIALIZER value or with the following function:

```
int pthread_mutex_init( pthread_mutex_t *mutex,
                        pthread_mutexattr_t *attr);
```

The mutex initialization returns 0 if successful, EBUSY if it has already been initialised or EINVAL if the mutex or attributes are invalid.

A mutex may have a variety of attributes accrued to it at initialisation. Discussion of these attributes is beyond the scope of this paper.

Mutexes should be used to protect multiple threads from reading *or* writing shared data at once. In addition, a mutex should be used to protect a non-thread safe function from multiple invocations across threads.

The following code provides an example of how a mutex is used to protect stdout. When a program is linked with the pthread library, the library will transparently perform internal mutex locking for functions such as *malloc(3)* and buffering/locking of stdout. In the following code explicit mutex locking of stdout is required as characters are being written to stdout individually.

file-mutex.c

```
#include <pthread.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>

pthread_mutex_t stdout_mutex = PTHREAD_MUTEX_INITIALIZER;
const char array[] = "http://www.linux.conf.au/\n";

void *display(void) {
        int x, y;
        struct timespec tv = { 0, 100 };
        for(y=0;y<5;y++) {
                pthread_mutex_lock(&stdout_mutex);
                printf("[thread %d] ", (int) pthread_self());
                for (x=0;x<strlen(array);x++)
                        putchar(array[x]);
                pthread_mutex_unlock(&stdout_mutex);
```

```
		nanosleep(&tv, NULL);
	}
	return NULL;
}

int main(void) {
	pthread_t	t1, t2;
	int		status;
	#ifdef __sun__
	pthread_setconcurrency( sysconf(_SC_NPROCESSORS_ONLN)+1 );
	#endif
	status = pthread_create(&t1, NULL, (void *) display, NULL);
	if (status != 0) {
		fprintf(stderr, "[thread %d] error creating thread: %s\n",
			pthread_self(), (char *) strerror(status));
		exit(0);
	}
	status = pthread_create(&t2, NULL, (void *) display, NULL);
	if (status != 0) {
		fprintf(stderr, "[thread %d] error creating thread: %s\n",
			pthread_self(), (char *) strerror(status));
		exit(0);
	       }
	pthread_join(t2, NULL);
	return 0;
}
```
file-mutex.c

The following was output from the aforementioned program:

```
[thread 4] http://www.linux.conf.au/
[thread 5] http://www.linux.conf.au/
[thread 4] http://www.linux.conf.au/
[thread 5] http://www.linux.conf.au/
[thread 4] http://www.linux.conf.au/
[thread 4] http://www.linux.conf.au/
[thread 5] http://www.linux.conf.au/
[thread 4] http://www.linux.conf.au/
[thread 5] http://www.linux.conf.au/
[thread 5] http://www.linux.conf.au/
```

The following was output from the aforementioned program with mutex locking removed at lines 16 and 20. Observe that how without correct synchronisation the data is corrupted.

```
[thread 5] [thread 4] http:ht//wwtp://www.w.linux.conflinu.au/x.c
onf.au/
[thread 5] http://www.linux.conf.au/
[thread 4] http://www[thread 5] http://www.linux.conf..linuxau/
.conf.au/
[thread 5] http://w[thread 4] hww.littnux.conf.au/
p://www.linux.conf.au/
[thread 4] http://www.linux.conf.au/
[thread 5] http://www.linux.conf.au/
[thread 4] http://www.linux.conf.au/
```

5.4 Signaling and Synchronisation

Within an application, there may be times where the need arises to signal another process or thread to begin doing some work. There may be a queue of data that needs to be processed or a thread pool waiting to service a newly established connection. POSIX threads offers *condition variables* to perform this function. A thread or many threads may wait on a particular condition and once the condition is met, one or all of the threads will return from their blocked state and begin their work.

Within POSIX threads, a condition variable is always associated with a mutex and indirectly, associated with the data it is waiting upon. A condition variable is defined by the *pthread_cond_t* type. Like a mutex, a *pthread_cond_t* can be initialized either with the constant PTHREAD_COND_INITIALIZER or the following function:

```
pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);
```

The function *pthread_cond_init()* returns 0 if successful, EAGAIN if insufficient resources are available or ENOMEM if insufficient memory is available.

Like most of the POSIX thread functions, condition variables have attributes which may be set upon initialisation. Discussion of these attributes are beyond the scope of this paper.

Once initialised, a condition variable may be waited upon by using either of the following functions:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           struct timespec *wait_time);
```

The above functions will return 0 upon success, EINVAL if either the mutex or condition variable are invalid or ETIMEOUT if *pthread_cond_timedwait()* returns from the timed wait without the condition variable changing state.

Before calling either of the above functions, a thread must have the associated data mutex locked. The function will internally unlock the mutex to allow other operations to occur, but once the condition has been signaled the function will return with the mutex locked.

In order to wake a thread or threads waiting upon a condition variable, one of the following functions may be used:

```
int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);
```

The *pthread_cond_signal()* function will only wake a single thread whilst *pthread_cond_broadcast()* will wake all waiting threads. The above functions return 0 upon success or EINVAL if the condition variable has not been initialised. The following code demonstrates how a condition variable may be used:

cond-wait.c

```
#include <pthread.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>

typedef struct {
        pthread_mutex_t mutex;
        pthread_cond_t  cond;
        char            *data;
} data_t;

data_t data;

pthread_mutex_t        stdout_mutex = PTHREAD_MUTEX_INITIALIZER;

void *wait_thread(void) {
        pthread_mutex_lock(&data.mutex);
        while (1) {

                /* mutex is always returned locked, hence no need to
                  lock or unlock within the loop */
```

```
                pthread_cond_wait(&data.cond, &data.mutex);
                pthread_mutex_lock(&stdout_mutex);

                if (strcmp(data.data, "q\n") == 0) {
                        printf("[thread %d] goodbye!\n",
                               (int) pthread_self());
                } else {
                        printf("[thread %d] user said: %s",
                                (int) pthread_self(), data.data);
                }
                pthread_mutex_unlock(&stdout_mutex);
                free(data.data);
                data.data = NULL;
        }
}

void *input_thread(void) {
        char buf[80];
        struct timespec tv = { 0, 100 };
        while (strcmp(buf, "q\n") != 0) {
                pthread_mutex_lock(&stdout_mutex);
                printf("[thread %d] enter string: ", (int) pthread_self());
                fgets(buf, 79, stdin);
                pthread_mutex_unlock(&stdout_mutex);
                pthread_mutex_lock(&data.mutex);

                if (data.data == NULL)  {
```

```
                        data.data = strdup(buf);
                } else {
                        free(data.data);
                        data.data = strdup(buf);
                }

                pthread_mutex_unlock(&data.mutex);
                pthread_cond_signal(&data.cond);

                nanosleep(&tv, NULL);
        }
        return NULL;
}

int main(void) {

        pthread_t       t1, t2;
        int             status;

        // initialize the data
        pthread_mutex_init(&data.mutex, NULL);
        pthread_cond_init(&data.cond, NULL);
        data.data = NULL;

        #ifdef __sun__
        pthread_setconcurrency( sysconf(_SC_NPROCESSORS_ONLN)+1 );
        #endif

```

```
        status = pthread_create(&t1, NULL, (void *) wait_thread, NULL);
        if (status != 0) {
                fprintf(stderr, "[thread %d] error creating thread: %s\n",
                        pthread_self(), (char *) strerror(status));
                exit(0);
        }
        status = pthread_create(&t2, NULL, (void *) input_thread, NULL);
        if (status != 0) {
                fprintf(stderr, "[thread %d] error creating thread: %s\n",
                        pthread_self(), (char *) strerror(status));
                exit(0);
               }

        pthread_join(t2, NULL);
        return 0;
}
```

cond-wait.c

The following was output from the aforementioned program:

```
[thread 5] enter string: hello
[thread 4] user said: hello
[thread 5] enter string: world
[thread 4] user said: world
[thread 5] enter string: q
[thread 4] goodbye!
```

5.5 UNIX Signal handling

Signal handling can become complex within a multi-threaded application and should, where possible, be avoided. The POSIX thread standard states that application generated signals are sent to any thread within that application. This means that if a SIGCHLD is raised by a child process, it may not be delivered to the thread that originally created that child. However, the signals SIGFPE, SIGSEGV, SIGPIPE and SIGTRAP are always delivered to the thread that caused them [2].

Despite signals being sent to individual threads, they do effect the process as a whole. Unless the relevant signal handler is installed, a signal such as SIGSEGV (whether sent to an individual thread or the process) results in the entire process being killed and a core file generated.

As mentioned previously, LinuxThreads signal handling differs significantly from the POSIX standard. According to the POSIX standard, external signals generated from a command such as *kill* should be sent to the process which in turn delivers it to any thread that does not block the signal. Since a thread within Linux is in fact a process, the external signal will be delivered to that particular process. If another thread is blocked in *sigwait(2)* waiting for that external signal, it will remain blocked. Additionally, SIGUSR1 and SIGUSR2 may not be used within a threaded application under Linux as they are used internally by LinuxThreads.

On POSIX compliant systems the preferred signal handling method is to create a thread specifically for signal handling. The required signals should be masked in all other threads by using the following function:

```
int pthread_sigmask(int how, sigset_t *set, sigset_t *oset);
```

The *pthread_sigmask()* function returns 0 upon success and a non-zero value upon error. A thread inherits the signal masks of its creator. It is therefore possible to set signal masks of all threads by specifying them in *main()* before creating any threads.

Once all threads other than the signal handling thread have the signal masked, the handler thread may wait for the signal with *sigwait()* and perform the appropriate processing when necessary.

However, in order to process external signals in a portable fashion on Linux, a signal handler should be installed process wide with either *signal(2)* or *sigaction(2)*. The code executed within a signal handler is not async-safe so it is generally necessary to wake a thread (using Sys V IPC) to perform the required operations once a signal has been received.

The following code outlines how a portable external signal handler may be implemented:

signals.c

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>
#include <semaphore.h>

sem_t                thread_sem;

void *signal_thread(void) {
        while (1) {
                // wait for the semaphore to be posted
                sem_wait(&thread_sem);
                printf("[thread %d] SIGHUP received\n",
                        (int) pthread_self());
        }
}

void handler(int sig) {
        if (sig == SIGHUP) {
                signal(SIGHUP, handler);
                sem_post(&thread_sem);
        }
}

int main(void) {

```

```
        pthread_t       t;
        int             status;

        #ifdef __sun__
        pthread_setconcurrency( sysconf(_SC_NPROCESSORS_ONLN)+1 );
        #endif

        if ( sem_init(&thread_sem, 0, 0) == -1) {
                printf("[thread %d] error initialising semaphore: ",
                        (int) pthread_self());
                perror("sem_init");
                exit(1);
        }

        signal(SIGHUP, handler);

        status = pthread_create(&t, NULL, (void *) signal_thread, NULL);

        if (status != 0) {
                printf("[thread %d] error creating signal_thread: %s\n",
                        (int) pthread_self(), strerror(status));
                exit(1);
        }

        pthread_join(t, NULL);
        return 0;
}
```

signals.c

5.6 One time operations

Sometimes within a threaded program, it is useful to execute an operation only once within a thread, no matter how many times it is invoked. This may include operations such as initialising a mutex with the *pthread_mutex_init()* function. This may be performed by declaring a variable of type *pthread_once_t* which must be initialized with the value PTHREAD_ONCE_INIT. It is then possible to perform an operation a guaranteed single time by using the following function:

```
int pthread_once(pthread_once_t *once, void (*once_routine,  void));););
```

once_routine is the pointer of the function to be called. Note, no argument can be given to the function unlike with *pthread_create_thread()*. The function *pthread_once()* returns 0 if successful or EINVAL if either the supplied routine or *pthread_once_t* are invalid.

The following code illustrates how *pthread_once()* may be used:

do-once.c

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#define MAX_THREADS 512
pthread_once_t once = { PTHREAD_ONCE_INIT };
```

```c
void say_hello(void) {
        printf("[thread %d] Listen carefully, I will say this only once\n",
                (int) pthread_self());
}

void *thr_ctl(void) {
        pthread_once(&once, say_hello);
        return NULL;
}

int main(void) {
        pthread_t        t1;
        int             x, status;
        #ifdef __sun__
        pthread_setconcurrency( sysconf(_SC_NPROCESSORS_ONLN)+1 );
        #endif
        for (x = 0; x < MAX_THREADS; x++) {
                status = pthread_create(&t1, NULL,  thr_ctl, NULL);
                if (status != 0) {
                fprintf(stderr, "[thread %d] error creating thread %s\n",
                        (int) pthread_self(), (char *) strerror(status));
                exit(0);
                }
        }
        printf("[thread %d] This is the end\n", (int) pthread_self());
        exit(0);
}
```

do-once.c

5.7 Thread Cancellation

In many situations, a programmer may wish to stop a thread's operation and ensure that it shuts down cleanly. An example would be ensuring a dbm database is closed correctly with *dbm_close(3)* before an application is shut down or unlocking shared mutexes when the thread is shutdown or restarted. It would be possible, but messy to implement such a signaling method using UNIX signals or condition variables. Fortunately, POSIX threads provides a set of cancellation functions which many be used to shutdown a thread cleanly.

In order for a thread to be canceled, it must first declare that it is able to be canceled by calling the following functions:

```
int pthread_setcancelstate(int state, int *oldstate);

int pthread_setcanceltype(int type, int *oldtype);
```

POSIX threads provides two types of cancellation - deferred and asynchronous. A thread using deferred cancellation will only allow cancellation at specific cancellation points. Cancellation points include various system and function calls including *read(2), pthread_cond_wait()*, etc. Consult the relevant vendor's operating system documentation for all supported cancellation points. Additionally, POSIX threads defines the following function:

```
void pthread_testcancel();
```

This function may be used to test whether cancellation is required at any point within an application. It is particularly useful for allowing cancellation from a hard loop without any system cancellation points.

Asynchronous cancellation allows the thread to be canceled at any time without encountering a cancellation point. Use of asynchronous cancellation is not encouraged as it may potentially leave resources in an unknown state.

Once cancellation has been enabled within a thread, the following functions may be used to define and place a "clean up function" on the cancellation stack:

```
void pthread_cleanup_push(void (*handler, void *), void *arg);

void pthread_cleanup_pop(int execute);
```

Note, in order for the application to compile, use of *pthread_cleanup_push()* requires that a matching *pthread_cleanup_pop()* is placed within the same caller function.

If multiple cleanup handlers are placed upon the cleanup stack, they will be called in LIFO order once a cancellation request is received. In order to send a cancellation request, the following function is defined:

```
int pthread_cancel(pthread_t thread);
```

The *pthread_cancel()* function returns 0 upon success or ESRCH if no such thread exists. The following code provides an example of deferred cancellation:

cancel.c

```
#include <pthread.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>

double          *current_fn;
pthread_t         t1, t2;

void calc_fn_cleanup(void) {
        printf("[thread %d] cancelled; cleaning up now\n",
                (int) pthread_self());
        if (current_fn != NULL) {
                free(current_fn);
        }
        return;
}

void *calc_fn(void *arg) {
        const double k = 1/sqrt(5.0), p = ((1.0+sqrt(5.0))/2.0);
        double x;
        int i = 1;
```

```
        int *total = arg;

        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
        pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
        pthread_cleanup_push((void *) calc_fn_cleanup, NULL);

        current_fn = (double *) malloc(sizeof(double));

        while (1) {
                x = k * pow(p, ++i);
                *current_fn = x;
                pthread_testcancel();
                *total = *total + 1;
        }

        pthread_cleanup_pop(1);
        return NULL;
}

void *cancel_q(void) {
        char buf[80];

        while(strcasecmp(buf, "y\n") != 0) {
                printf("[thread %d] shall I cancel [y/n]: ",
                        (int) pthread_self());
                fgets(buf, 79, stdin);
        }

```

```
        pthread_cancel(t1);
        return NULL;
}

int main(void) {
        int     status;
        void    *ret;
        int     total = 0;

        #ifdef __sun__
        pthread_setconcurrency( sysconf(_SC_NPROCESSORS_ONLN)+1 );
        #endif

        status = pthread_create(&t1, NULL, (void *) calc_fn, (void *) &total);

        if (status != 0) {
                fprintf(stderr, "[thread %d] error creating thread: %s\n",
                                pthread_self(), (char *) strerror(status));
                exit(0);
        }

        status = pthread_create(&t2, NULL, (void *) cancel_q, NULL);

        if (status != 0) {
                fprintf(stderr, "[thread %d] error creating thread: %s\n",
                                pthread_self(), (char *) strerror(status));
                exit(0);
        }
```

```c

	pthread_join(t1, &ret);

	if (ret ==  PTHREAD_CANCELED) {
		printf("[thread %d] thread %d cancelled; %d ops performed\n",
			pthread_self(), t1, total);
	}

	return 0;
}
```

cancel.c

5.8 Forking and Threads

The use of *fork(2)* within a threaded application should be avoided unless it is used only to invoke another program with *exec(2)*. The POSIX standard states that only the thread which calls *fork(2)* will exist in the child. Additionally, it will own all the mutexes and have the same values for thread specific data as it did before the *fork(2)*. Additionally, to assist in performing cleanup operations before a fork, POSIX threads provide the following function (which is analogous to *atexit(2)*):

```c
int pthread_atfork(void (*prepare)(void),  void  (*parent)(void),
                   void (*child) (void));
```

The function returns 0 if successful or ENOMEM if insufficient memory is available. The *pthread_atfork()* function allows a specified function to be called within a thread prior to *fork(2)* being called. This allows a thread to unlock any required mutexes and protect itself from a deadlock situation in the child. All threaded software which executes non-trivial code within a child process should use *pthread_atfork()*.

Linux diverges from the POSIX standard in regards to the behavior of *fork(2)* within a threaded program. A portable method to use *fork(2)* safely on Linux and other operating systems is to create a "fork thread" which listens for requests and calls *fork(2)* and *wait(2)* on behalf of other threads.

5.9 Using Non-thread Safe Functions

Whilst thread safe versions of many standard POSIX functions (such as *strtok_r()* have been added with the advent of threads, many non-thread safe functions still exist either in third party libraries or within libc. Functions such as *bufsplit()* can still be used within a threaded program but must be protected by a mutex. The following code provides an example of how such a function may be used:

```
...
pthread_mutex_t bufsplit_mutex = PTHREAD_MUTEX_INITIALIZER;
char **st[3];
char *buf;
```

```
buf = strdup("root:x:0:0::/:/sbin/sh");

pthread_mutex_lock(&bufsplit_mutex);
bufsplit(":", 0, NULL);
bufsplit(buf, (size_t) 2, st);
pthread_mutex_unlock(&bufsplit_mutex);
...
```

Third-party libraries should have their function interfaces modified accordingly to follow the standard of functions such as *strtok_r()* where any persistent data is passed as an argument to the function. However, in many cases it may not be possible to alter the interface a function presents to the outside world. In such a situation POSIX thread specific data may be of assistance. Essentially, thread specific data allows the same data structure to be used by multiple threads but each thread has it's own instance of the data and its own associated value. However, due to it's complex nature, detailed discussion of thread specific data is beyond the scope of this paper.

5.10 Advanced Threading Topics Not Covered

Much of the advanced multi-threading functionality provided within the POSIX standard has not been discussed within this paper. Specifically, thread, mutex and condition variable attributes provide the ability to control how such data types behave and what functionality they provide (e.g mutexes shared across multiple processes, etc). The standard also provides real time extensions that provide functionality to alter scheduling characteristics of an application and its threads. Further information can be sought from threading texts, vendor documentation or the standard itself.

Additionally, The X/Open Group have provided extensions to the POSIX thread standard as part of their UNIX98 specification. The most noteworthy extension is the inclusion of read/write locks with the *pthread_rwlock_*()* set of functions. These provide mutex like functionality whilst allowing a distinction between reader and writer threads thus allowing many threads to concurrently access a resource whilst allowing only a single thread to perform updates.

6 Acknowledgments

The author would like to thank the following people for reviewing and providing amendments to this paper:

David Luyer
Pacific Internet, Australia

John Ferlito
Bulletproof Networks, Australia

Nic Grant
Tripfinder Limited, Ireland

Philip Dell
Hewlett-Packard Consulting, Australia

References

[1] Chris DiBona, Sam Ockman and Mark Stone (editors)
Open Sources - Voices from the Open Source Revolution
O'Reilly and Associates, 1999

[2] David R. Butenhof
Programming with POSIX Threads
Addison-Wesley, 1999

[3] E. W. Dijkstra, F. Genuys (editor)
Cooperating sequential processes, Programming languages
Academic Press, 1968

[4] Brian O'Sullivan
comp.os.research FAQ
http://www.serpentine.com/ bos/os-faq/, 1996

[5] Uresh Vahalia
UNIX Internals - The New Frontiers
Prentice Hall, 1996