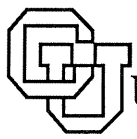
Awesime/Cthreads Library

David B. Wagner

CU-CS-567-91 December 1991



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Awesime/Cthreads Library
User's Manual

David B. Wagner
Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

CU-CS-567-91 December 1991

University of Colorado at Boulder

Technical Report CU-CS-567-91
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Awesime/Cthreads Library
User's Manual

David B. Wagner
Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

December 1991

Abstract

The Awesime/Cthreads library is a software package that provides a Mach Cthreads [1] interface on top of the Awesime thread library. The library allows Cthreads programs to be written and debugged on machines that do not support the Mach operating system. Such programs typically run in the Mach environment with no changes required. The underlying runtime system, Awesime [2], is very portable and currently runs on the DECstation, Sparc (including multiprocessors such as the Solbourne), Sequent, and Encore architectures.

The Cthreads interface is simple and is ideal for educational use. It has been used as the implementation environment for programming assignments in CSCI 5573, a graduate-level operating systems course at the University of Colorado. It should also be suitable for an undergraduate-level operating systems course, or a course in parallel programming.

1 Introduction

The Cthreads lightweight process interface, first developed for the Mach operating system [1], is a set of extremely simple, easy-to-use parallel programming primitives. A thread is an independent stream of execution (a "lightweight process"). The interface allows threads to be created and waited for, and allows threads to return values to other threads. For additional flexibility the interface also provides monitor and condition variable primitives. It is well known that such primitives provide a complete foundation on which other synchronization mechanisms (such as semaphores) can be constructed.

The Awesime/Cthreads library is a software package that provides a Cthreads interface on top of the Awesime [2] thread library. The library allows Cthreads programs to be written and debugged on machines that do not support the Mach operating system. Such programs typically run in the Mach environment with no changes required. The underlying runtime system, Awesime, is very portable and currently runs on the DECstation, Sparc (including multiprocessors such as the Solbourne), Sequent, and Encore architectures.

Although the Awesime library is written in C++, the Cthreads interface works in both C and C++ programs.

2 The Interface

2.1 Primitive Types

The types defined in `cthreads.h` are:

`any_t` — anything that will fit in four bytes, e.g., an integer or a pointer (but *not* a double).

`cthread_any_t` — a pointer to a function that takes an `any_t` as a parameter and returns an `any_t`.

`cthread_t` — a pointer to a cthread.

`mutex_t` — a pointer to a mutex, i.e., a lock or monitor.

`condition_t` — a pointer to a condition variable.

`string_t` — a pointer to a character string.

2.2 Initialization

`void cthread_init();` According to the standard Cthread interface, a Cthread program must call `cthread_init()` before using any other Cthreads routine. In this implementation, this routine does nothing. However, you should call it in order to make your code portable.

In this implementation, the actual initialization is done before the users's code runs. To accomplish this, it is required that you declare your main routine as `_CTHREAD__MAIN_` rather than `main`. (Note that there are *two* underscores between the D and the M.) `_CTHREAD__MAIN_` will be called by the runtime system with the usual parameters (`argc` and `argv`).

2.3 Thread Management

`cthread_t cthread_fork( cthread_fn_t func, any_t parm );` This routine is used to create a new thread. Note that there are only two parameters: the name of the function to be executed by the new thread and *one* parameter to be passed to the function. If the function requires more than one parameter, or if the parameter won't fit into four bytes, then the function should be rewritten to take a pointer to a struct containing the multiple parameters, and `parm` should be that pointer.

It is usually necessary to explicitly coerce the parameters to be of the appropriate types.

`any_t cthread_join( cthread_t child );` This routine is called by a thread to receive a value returned by some other thread. The parameter is the value returned by a previous call to `cthread_fork`. The return value should fit in four bytes; if it won't, it should be pointer to the value rather than the value itself.

Note that a call to `cthread_join` does not return until the thread begin joined terminates.

`void cthread_exit( any_t result );` This call returns a value that can be retrieved using `cthread_join`. The same effect can be achieved simply by executing the statement "`return result`" in the function executed by the thread.

`void cthread_detach( cthread_t child );` This tells the runtime system that `child` will never be the subject of a `cthread_join`; it is necessary only for efficiency, not correctness.

It is an error to attempt to join a thread that has been detached. (The probable result is a core dump.)

`void cthread_yield();` This causes the calling thread to yield the processor to the Cthreads scheduler. This implementation of Cthreads does not incorporate timer interrupts, so if you want your threads to take turns running on only a single processor, you must insert calls to `cthread_yield` in strategic places.

2.4 Synchronization

`mutex_t mutex_alloc();` Creates a mutex and returns a pointer to it.

`void mutex_free();` Deallocates a mutex.

`void mutex_lock( mutex_t m );` Locks a mutex. If the mutex is unavailable, the thread may be blocked or it may spin indefinitely: the exact behavior is a function of the particular version of the library in use and is not guaranteed to be either one way or the other in the future. A spinning lock primitive can always be achieved by using `mutex_try_lock` (described below) inside a looping construct. A blocking lock primitive can always be achieved by using a mutex and a condition variable to implement a binary semaphore.[1]

`int mutex_try_lock( mutex_t m );` Attempts to lock the mutex, but does not block the thread. The return value indicates whether the mutex was acquired (1) or not (0).

`void mutex_unlock( mutex_t m );` Self-explanatory.

`condition_t condition_alloc();` Creates a condition variable and returns a pointer to it.

`void condition_free();` Deallocates a condition variable.

`void condition_wait( condition_t c, mutex_t m );` Causes the calling thread to block until some other thread signals the condition variable. Assumes that the caller already has locked `m`; `m` will automatically be released before blocking and re-acquired before returning.

`void condition_signal( condition_t c );` Wakes up exactly one thread that is waiting on this condition variable (if any). If several threads are waiting for the same condition variable, there is no way to specify which one will be awakened.

`void condition_broadcast( condition_t c );` Wakes up *all* threads that are waiting on this condition variable (if any).

Note that *condition variables have no memory.* In other words, `condition_signal` or `condition_broadcast` operations that are performed while no threads are waiting do not have any effect on subsequent calls to `condition_wait`.

[1] A blocking lock can also be achieved by combining `mutex_try_lock`, `cthread_yield`, and a looping construct, but this may cause unnecessary context switches.

2.5 Unsupported Routines

The following routines are part of the standard Cthreads interface but are UNSUPPORTED by this implementation:

```
void mutex_init( mutex_t m );

void mutex_clear( mutex_t m );

void condition_init( condition_t c );

void condition_clear( condition_t c );
```

This is a consequence of the fact that the implementation does not allow declaration of a mutex or a condition variable statically; you must dynamically allocate them using `mutex_alloc()` and `condition_alloc()`.

2.6 Extensions

The Awesime/Cthreads library supports the type `barrier_t`, which is not part of the standard Cthreads interface. The following operations are supported on objects of this type:

`barrier_t barrier_alloc( int n );` Creates a barrier of *height* `n` (see below) and returns a pointer to it.

`void barrier_free();` Deallocates a barrier.

`void barrier_rendezvous( barrier_t );` For a barrier of height `n`, this call blocks the first `n`-1 callers. When the `n`-th call is made, all `n` callers are released, and the barrier is reset to its original state.

2.7 Miscellaneous

The following routines are useful for printing debugging or error messages; their use should be self-explanatory:

```
cthread_t cthread_self();

void cthread_set_name( cthread_t, string_t );

string_t cthread_name( cthread_t );

void mutex_set_name( mutex_t, string_t );

string_t mutex_name( mutex_t );

void condition_set_name( condition_t, string_t );
```

```
string_t condition_name( condition_t );

void cthread_set_data( cthread_t, any_t );

any_t cthread_data( cthread_t );
```

3 Compiling, Linking, and Running

The program must include the definition file `cthreads.h`.

Because the Awesime/Cthreads library is written in the GNU C++ programming language, the GNU C++ compiler must be used to link a program to the library. Libaries should be linked into the application in the following order: `-lcthread -lawe2 -lg++`.

A sample GNUmakefile is given in Appendix A. The locations of all relevant files on machines at the University of Colorado are given in Appendix B.

The number of processors to be used by the program is specified by the UNIX environment variable `CTHREAD_CPUS` (default: 1). Unfortunately, it is not possible for the user to specify this number programmatically; it needs to be known before the main thread is created.

4 An Example

The code shown in Figures 1–3 illustrates a solution to the producer-consumer problem with multiple buffers.

There are several noteworthy features of this example:

- The program source must include the file `cthreads.h` before any Cthreads constructs are used.
- The implementation requires that the main routine be declared `_CTHREAD__MAIN_` rather than `main`. (Note that there are *two* underscores between the D and the M.) The Cthreads library will provide the `main` routine expected by the UNIX linker; this will in turn call `_CTHREAD__MAIN_` with the usual parameters (`argc` and `argv`).
- Although the `mutex_t` and `condition_t` types provide the functionality of monitors and conditions variables, there is no syntactic support for monitors. The Cthreads equivalent of the monitor, the `mutex_t`, must be explicitly acquired and released by calls to `mutex_lock` and `mutex_unlock`.
- `wait`ing for a condition variable automatically releases the associated `mutex_t`, and the `mutex_t` is automatically re-acquired after being `signal`ed but before proceeding. Note, however, that the associated `mutex_t` must be supplied explicitly in the call to `condition_wait`; there is no way to permanently bind a particular `mutex_t` to a particular `condition_t`
- The combined semantics of the `mutex_t` and `condition_t` types are those of *Mesa* monitors [4]. In contrast to a Hoare monitor [3], there is no guarantee that a `signal`ed thread will run immediately, or even that it will be the next thread to acquire the `mutex_t`. The ramification of this is that every call to `condition_wait` should be enclosed in a loop that checks for the boolean condition that is implicitly associated with the given `condition_t`.

```
#include "cthreads.h"

mutex_t themonitor;
condition_t full, empty;
#define BUFSZ 1
int buffer[BUFSZ];
int first_empty = 0;
int first_full = 0;

_CTHREAD__MAIN_(argc, argv)
int argc;
char **argv;
{   cthread_t child;

    themonitor = mutex_alloc();
    full = condition_alloc();
    empty = condition_alloc();
    child = cthread_fork(consumer, 0);
    cthread_detach(child);
    child = cthread_fork(producer, 4);
    return 0;
}
```

Figure 1: Global declarations and the main routine.

```
int producer(dummy)
int dummy;   /* not used */
{   int i;
    while (/* not done */)
        mutex_lock(themonitor);
        while ((first_empty+1)%BUFSZ == first_full)
            condition_wait(empty, themonitor);
        buffer[first_empty] = /* data */;
        first_empty = (first_empty+1)%BUFSZ;
        condition_signal(full);
        mutex_unlock(themonitor);
    }
}
```

Figure 2: The producer.

```
int consumer(dummy)
int dummy;   /* not used */
{   int b;
    while(/* not done */) {
        mutex_lock(themonitor);
        while (first_empty == first_full)
            condition_wait(full,themonitor);
        b = buffer[first_full];
        first_full = (first_full+1)%BUFSZ;
        condition_signal(empty);
        mutex_unlock(themonitor);
    }
}
```

Figure 3: The consumer.

A big advantage of Mesa monitor semantics, assuming the programmer is sufficiently disciplined, is that a `condition_signal` or `condition_broadcast` is never an incorrect thing to do.

5 "Unorthodox Behavior"

- The library does not perform preemptive scheduling. Thus, failure to use enough processors can cause a deadlock if busy waiting is used.
- The program will exit silently when all C-threads are dead or blocked. This includes the thread that runs `_CTHREAD__MAIN_`. There are two ramifications of this:
  - If the program deadlocks, it will exit without printing any diagnostic. This can be confusing the first time it happens, since the natural expectation is for a deadlocked program to "hang."

    In the event that this happens, the recommended course of action is to insert print statements (to stderr) immediately before every call to a blocking primitive (`cthread_join`, `mutex_lock`, `condition_wait`) in the program. This can give you the identity of the last thread to block.
  - Since the main thread is a Cthread, not a UNIX thread, the program may not exit after the main thread exits (assuming there are other ready threads). On some other implementations, notably the NeXT machine, the entire program exits if the main thread exits.
- The library creates one UNIX process per processor. In the case of multiple processors, the library also creates a large temporary file (approximately 2 megabytes) into which shared memory is mapped. This file will go away when *all* UNIX processes associated with the program have exited. However, it can occasionally happen that if the program terminates

abnormally, some of the UNIX processes may become orphans. If this happens, the temporary file will not go away until they are hunted down and killed. Failure to notice this situation when it happens can cause your disk space to fill up very quickly.

- A corollary of the previous point is that core files, when dumped, can be very large.

6 Troubleshooting

Table 1 points out some common problems and what to do about them.

A Sample GNUmakefile

The directory pathnames shown in this makefile are valid on *tramp*, a Sequent Symmetry multiprocessor owned by Computing and Network Services of the University of Colorado. This makefile should be used only with the **gnumake** utility.

```
##        TARGET      - program to build
##        OBJS        - object modules ending in .o
##
##        C++ program sources should end in .cc
##        C program sources should end in .c

TARGET      = main
COBJ        = main.o

AWE2        = /users/csci/grunwald/Awe2
AWESIMELIB  = -L$(AWE2)/lib -lawe2-g++

CTHREAD     = /users/csci/wagner/Cthreads
CTHREADINC  = -I$(CTHREAD)
CTHREADLIB  = -L$(CTHREAD) -lcthread

INCL        = -I. $(CTHREADINC)
LIBS        = $(CTHREADLIB) $(AWESIMELIB) -lg++

C           =gcc
CFLAGS      := ${CFLAGS} ${INCL} -g

C++        =g++
C++FLAGS   := ${C++FLAGS} ${INCL} -O -g

$(TARGET): $(COBJ)
        $(C++) $(C++FLAGS) -o $(TARGET) $(COBJ) $(LIBS)
```

Symptom	Possible cause/solution
Program won't link	Are you using Gnu g++ to compile and link?
Program dies as soon as it uses any library primitive	Failure to initialize properly. Is your main routine declared `_CTHREAD__MAIN_`?
Core dump using mutex or condition primitives	Did you remember to call `mutex_alloc()` or `condition_alloc()`, as appropriate?
Core dump using `cthread_join`	Tried to join on a detached thread, or tried to join on the same thread more than once.
Return value from `cthread_join` is nonsense	Trying to return something that doesn't fit in 4 bytes, e.g., a double. Return a pointer instead.
I am returning a pointer, and it's still nonsense	You probably returned a pointer to a value inside the stack of the now-dead thread. Allocate space using `malloc()` for values that are returned by reference.
Program terminates mysteriously without apparent error	Probably a deadlock; see Section 5 for suggestions. Busy waiting with fewer processors than threads will almost surely cause a deadlock, unless a `cthread_yield` is placed inside every busy wait loop.
Program fails immediately with the following error: Assertion bytesWritten == oldData failed:...	Can't write the temporary file. Is the current directory writable? Is there enough disk space? If disk space is tight, try doing a cd to another file system (e.g., /tmp) before running the program.
Program won't execute in parallel on a multiprocessor	Are you setting the `CTHREAD_CPUS` environment variable?
Disk space is steadily leaking away	You have orphaned processes. Hunt them down and kill them.

Table 1: Troubleshooting guide.

B Location of Software at the University of Colorado

Persons associated with the University of Colorado may be able to access the library on the following machines:

tramp: This is a 6-processor Sequent Symmetry owned by Computing and Network Services. The relevant pathnames are:

```
/users/csci/grunwald/Awe2/lib
/users/csci/wagner/Cthreads
```

Tramp is currently running a slightly older version of the library because it has an older version of the Gnu software.

srlnet: This is a group of DECStations owned by the Systems Research Laboratory of the Department of Computer Science; it includes bullwinkle, coco, foobar, goober, mumble, and tile. The relevant pathnames are:

```
/srl/Awe2/lib
/srl/Awe2/Cthreads/include
/srl/Awe2/Cthreads/lib
```

CS sun-4's: This includes anchor as well as eclipse, a 3-processor Solbourne multiprocessor. (Users must sign a non-disclosure agreement in order to get an account on eclipse.) The pathnames are the same as on the srlnet machines.

References

[1] COOPER, E., AND DRAVES, R. C-Threads. Tech. Rep. CMU-CS-88-154, Carnegie-Mellon University, Feb. 1988.

[2] GRUNWALD, D. A user's guide to Awesime, an object oriented parallel programming and simulation system. Tech. Rep. CMU-CS-88-154, Carnegie-Mellon University, Feb. 1988.

[3] HOARE, C. Monitors: An operating system structuring concept. *CACM 21*, 8 (Aug. 1978), 666–677.

[4] LAMPSON, B., AND REDELL, D. Experience with processes and monitors in mesa. *CACM 19*, 5 (Feb. 1980), 105–117.