Writing multithreaded programs under RiscOS

Neil A Carson Simtec Electronics

Sun 17th November 1996

Contents

Abstract

Acorn/ART RISC computers have a fast and efficient 32–bit operating system called RiscOS. It does, however, fail to include the ability to write multithreading and multiprocessing applications, meaning that hardware such as SIMTEC's *HYDRA* is unable to endow most applications with possible performance gains. Also, existing applications suffer from sluggish user response when trying to do any work in the background, and RiscOS makes it *very* difficult to do any form of smooth concurrent processing.

The RiscOS thread manager module (simply called `Thread`) fills these holes, is available now, by providing simple to use support for both application and module level multithreading on all Acorn machines from the A310 to the Hydra–equipped multiprocessing StrongARM RiscPC. It is part of a family of programs that will, eventually, provide RiscOS with a full multitasking, multithreading, and multiprocessing user interface.

1 Introduction

This document, together with the `Thread` module, forms the first release of the SIMTEC RISCOS multithreading development toolkit. This is by no means the last release of `Thread`: The planned evolution is as follows (depending on interest in this initial release):

Date	Stage
October 1996	Initial release of `Thread` toolkit (single CPU)
March 1997	New version of `Thread` module supporting HYDRA
Q3 1997	Process manager for multiprocessing task and thread management
Q1 1998	New window manager using the process manager for desktop apps

Our eventual goal is to provide a multiprocessing, multithreading, and preemptively multitasking environment that is faster, smoother, and generally more pleasant to the user. **The `Thread` API has how been *frozen*, so any applications that work well with the initial release will *not* need changing at a later date.**

Thread is now the preferred way of writing applications to take advantage of the HYDRA; for high speed, low-level multiprocessing code, the **Hydra API** exists, and is available from SIMTEC. Although the current version of the `Thread` module only works on a single ARM processor, a new version will be shipped with HYDRA boards towards the beginning of next year (and will be available freely to existing HYDRA users, subject to adoption of the API) that will be able to take advantage of the speed increase gained by having up to five ARM processors fitted to a RiscPC.

2 Multithreading

2.1 Concurrent programming

Concurrent (multithreaded) programming has evolved into quite an art over the years, and several good textbooks are available on the subject. For more information, see appendix B on page 27. A particularly good book is "Modern Operating Systems" by Tannenbaum. Readers not familiar with the concept should read the appendix before carrying on with the rest of this guide.

2.2 Terminology

A task is either a WIMP application, or a single tasking full screen RISCOS program. A thread is a child process of a task.

2.3 What is multithreading?

A multithreading program is able to create child processes (called "threads") that run at the same time as itself, possibly at an altered scheduling priority. On a multiprocessor computer system, such as a high end RiscPC, these threads actually run concurrently, and can be distributed across any number of ARM processors; on uniprocessor systems, they are preempted on just one CPU in order to give the illusion of running at the same time.

2.4 Technicalities of threads

As currently no interpreted language, including BASIC, exists for the Acorn platform that supports threads, all code that runs as a thread must be written in either a compiled language (such as C, C++, Pascal, Fortran, etc.) or ARM assembler. A C library has been provided to simplify the writing of multithreaded programs in both C and C++.

All threads run in 26-bit USR mode. They may call SWIs, make use of undefined instructions (however, only the floating point ones) and even use virtual memory[1] to allow the threads to have a larger workspace. This will even be the case on a system running with HYDRA. Each thread runs sequentially, and has its own program counter, stack (which is APCS-compliant) etc. to keep track of where it is. Threads can create child threads, and can 'block' waiting for messages or semaphores to enforce mutual exclusion. While one thread is blocked, another may continue running. All threads share the same address space as the rest of the task, and because of this may read, write or even wipe out another thread's stack. There is no protection between threads, because:

- It is impossible
- It should not be necessary, anyway

Different threads may run the same code, but should be careful to make sure that new instances of any variables are created, or that access to them is managed properly with semaphores.

Threads may also send messages between one another, either to communicate data, or to achieve code synchronisation (one thread may not be able to continue until another has reached a given point) in a similar vein to occam's CSP architecture. In order to allow both possibilities as efficiently as possible, `Thread` supports a simple (and efficient) messaging system that does the job, and nothing more.

2.5 Why bother?

All of the above sounds very good, but may leave you wondering what the point of all this is. Multithreading is in fact very useful when writing applications software. For example:

Modelling: Concurrency is natural—in the real world, things do tend to happen at the same time. Right now, ships are sailing round the world, the planet rotates around the sun, weather systems form in the Atlantic and move northwest, all largely independently.

Responsive Programs: For most WIMP programs, user responsiveness is a crucial factor in making the application friendly. When multithreading, user input can be monitored and responded to at *all* times, regardless of what the rest of the program is doing (with the exception of redraw event processing, because the WIMP does not have a per-task redraw queue). A tool, such as a World Wide Web browser, could fetch pages, render them and monitor user input all at the same time, making the application much more responsive for people to use.

Blocking programs: When an application needs to frequently wait for things to happen, such as waiting for some data to arrive, it could quite happily do something else whilst this is happening in a multithreaded environment.

[1] Only when a suitable system, such as Clares Virtualise, is running.

Expensive Programs: Applications that make use of a large amount of CPU power (for instance, JPEG plotters, ray tracers, complex renderers) can simply start off other threads doing another aspect of the same job. Although on a uniprocessor system this will have a small overhead (around 2%) this would allow that program to transparently take advantage of any extra processors, such as those provided by the *HYDRA* if present.

If your software does not easily fit into one of the above categories, then don't bother multithreading it.

3 Writing multithreading tasks

3.1 Root environment

As soon as the original (or "root") task registers itself with the thread manager, it itself will become a thread, running with a priority of 0 (priorities can be summed up as $-20 \leq Pri_{task} \leq 20$) such that it can start other threads with a priority greater than, or less than, itself. Remember that all threads, *including* the root, will execute concurrently. If a thread starts execution of another thread (a "child" thread) then it becomes a "parent" and the threads it started are it's "children" (which are all siblings of each other).

3.2 Thread environment

The entry condition for a thread (the state of its registers as control of the CPU arrives to it) are as described in SWI `Thread_Begin`. The stack pointed to by R13 is around 4Kb in size, and is an APCS-compliant stack chunk. (note that if this is done from the Norcroft C run-time system, the reserved words must be copied from the bottom of the existing stack; this is done automatically by the supplied C library).

3.3 Properties of threads

Threads can be created, at which point they will start executing, and will be destroyed upon their completion. Completion of a thread's execution is done by the standard subroutine return instruction:

```
MOV pc, lr
```

The C compiler's variant that sets the PC flags will also work fine.

The threads may also do many other things:

Sleep: In order to facilitate waiting for a while, a thread can go to sleep (ie. stop) for a given number of centiseconds. Meanwhile, any other threads will continue to run. The time by which the thread goes to sleep, however, is in fact the number of *CPU centiseconds* that it will sleep for. This timing is currently very inaccurate, but will be refined later on.

Yield: If a thread decides it has done enough for now, it can yield some CPU time in order to allow a different thread to have a turn, before it is automatically preempted.

Semaphore: Semaphores may be initialised, signalled, and waited on. This means that a thread that is waiting on an already used semaphore will go to sleep until it is signalled upon elsewhere. Of course, other threads will continue to execute while the waiting one is asleep.

Messages: A thread may opt to receive messages asynchronously or synchronously. Messages are described in more detail below.

4 Interaction between threads

The two ways that threads may interact with each other are by using the messaging and/or semaphore systems provided by `Thread`. This section describes in more detail how they have been implemented.

4.1 Messages

Each thread has an incoming message queue, the size of which is limited only by memory. All messages must have their space allocated and freed by the user's code; all `Thread` does is keep track of where they are. A given message may not exist in more than one thread's incoming message queue; it must also exist only once in any given queue. Failure to observe either of these points will result in a corrupt messaging system for a given task.

Each message consists of several word-sized fields, namely:

1. Two reserved words
2. Source thread's handle
3. Destination thread's handle
4. Message number
5. Data field

Although the data field is only four bytes, it can be used to pass a pointer; a large data field, like in WIMP messages, is not required, because all threads share the same address space. This would only serve to force complex memory allocation, and generally slow things down.

When a thread sends a message to a destination thread, it is simply added onto the destination's list of incoming messages. The destination thread must look at this queue every now and again to see if there are any messages in it, if it expects any. On looking at the queue, the receiving thread can opt to:

- Receive an error if the queue is empty
- Be suspended until there is at least one item in the queue, then receive that item

It is thus obvious how to use threads to send data between each other. An example program later on shows how to pass messages in C, using dynamic memory allocation and deallocation. These messages can also be used to provide code synchronisation. Thread 'A' sends a message to thread 'B', which is waiting. Thread 'A' then in turn starts waiting for a response from 'B.'

4.2 Semaphores

Semaphores are defined in more detail in the appendix. `Thread` provides support for initialising, signalling and waiting upon counting semaphores. When waiting on a semaphore, the thread can specify as to whether or not it wants to be suspended if the semaphore is about to go negative. If not, an error is simply returned.

5 `Thread` SWIs

The SWIs on the following page are currently provided by the `Thread` module. Their functionality will probably not change in the future; if, however, changes are made, they will be ensured to be backwardly compatible. The entries and exits are the same as for normal, standard SWIs. If something goes wrong, the 'V' flag is set on exit, and R0 points to a standard RiscOS error block. In the entry and exit conditions, if a register's value is not documented, it will be preserved. If a register's value is a pointer, it is signified as $\rightarrow$; if it is a numeric value, it is shown as $=$. In reality, both of these are the same, as all pointers are, of course, numeric values.

5.1 SWI `Thread_Initialise` (&4DA40)

On entry:

Register	Contents
R0 →	Task name, NULL terminated (which may be truncated)
R1 →	Context switch code (NULL for none)

On exit:

Register	Contents
R0 =	Task handle
R1 =	Root thread handle

This SWI is called by a task early in its initialisation to register itself with `Thread`. From this stage onwards, the task is under `Thread`'s control, albeit with only one running thread, and should behave accordingly (eg. turning threads off across calls to SWI `Wimp_Poll`).

The context switch function, if supplied in R1, will be called each time `Thread` performs a context switch. On entry to this special function, the processor will be in SVC mode, R1 will be the thread handle of the thread being switched out, and R2 the thread handle of the thread being switched in. Together with SWI `Thread_Context`, this can be used to allow the user to add to a given thread's context (eg. for saving the VDU state). The function *must* preserve all registers and the CPSR flags.

The root thread runs with a priority of 0, and can do anything that a normal thread can.

5.2 SWI `Thread_CloseDown` (&4DA41)

This SWI does not take arguments via registers. All registers are preserved. Whichever task calls this SWI will be deregistered by `Thread`, and will revert a to normal, single threaded execution.

5.3 SWI `Thread_On` (&4DA42)

On entry:

Register Contents
R0 = Task handle of new task, or NULL

If a task handle is supplied in R0, this code will effect a task switch. It is normally called in this manner on return from SWI `Wimp_Poll`. If, however R0 is NULL, then this will re-enable multithreaded operation if it has been turned off by SWI `Thread_Off`.

Typical action for a task whilst doing a WIMP Poll would be:

```
thread_off();
wimp_poll(blah, blah, blah);
if (event_got == Redraw) handle_the_redraw_event();
thread_on(my_task_handle); /* Do a context switch back */
thread_on(NULL); /* Turn threads back on */
```

The action of turning threads back on has been split into two parts because, on a multi-processor system, the overheads in doing a task switch are great.

5.4 SWI `Thread_Off` (&4DA43)

Turns multithreading off, leaving the thread of execution with whichever thread called this SWI. For example usage, see SWI `Thread_On`.

5.5 SWI `Thread_Begin` (&4DA44)

On entry:

Register	Contents
R0 →	Thread name, NULL terminated (may be truncated)
R1 =	Execution priority of new thread
R2 =	Argument for thread
R3 →	Address of code for thread
R4 →	Current APCS stack chunk (or NULL for none)

On exit:

Register	Contents
R0 =	Handle of newly created thread

This SWI starts a new thread of execution. The execution priority is in the range −20 to 20, the lower value being of the highest priority. The thread is entered as follows: On entry:

Register	Contents
R0 =	My thread handle
R1 =	Parent's thread handle
R2 =	Argument passed by parent
R10 →	Stack limit
R11 →	Frame pointer (NULL)
R14 =	Link value
R15 =	User mode with interrupts enabled

The values of all other registers, including the floating point ones, are undefined. The link value in fact holds the address of a function that calls SWI `Thread_Zap`, so the thread can terminate by issuing the normal `MOVS pc, lr` instruction.

If a pointer to the current APCS stack chunk is supplied, as is the case with the ThreadLib C library, then `Thread` will copy over the SharedCLibrary's relocation reserved words in order to enable the SharedCLibrary to continue to function properly. See the note on using the SharedCLibrary later on.

5.6 SWI `Thread_Semaphore` (&4DA45)

On entry:

Register	**Contents**
R0 →	Semaphore
R1 =	Initial value

This call initialises a counting semaphore (which must be one word in size, properly aligned etc.) with the given value. The semaphore is held as a signed 32-bit integer. Waiting decrements it, and Signalling increments it. If a user wishes to avoid the overheads of these SWI calls, he or she may choose to examine the contents of the semaphore directly themselves first, to work out whether or not a call will block.

5.7 SWI `Thread_Signal` (&4DA46)

On entry:

Register Contents
R0 → Semaphore

Performs a Signal (increment) operation on the given semaphore. If there is a thread waiting on this semaphore, then it will start executing again.

5.8 SWI `Thread_Wait` (&4DA47)

On entry:

Register Contents
R0 → Semaphore
R1 = Pointer to flag (or NULL)

Performs a Wait (decrement) operation on the given semaphore, if possible. If the operation causes the semaphore to go negative, the action can be twofold:

1. If the pointer in R1 is NULL, then the thread will be suspended until the semaphore is signalled upon

2. If the pointer in R1 is not NULL, the word at the address will be filled with a boolean value indicating whether or not the Wait failed

5.9 SWI `Thread_Send` (&4DA48)

On entry:

Register Contents
R0 → Message

Sends the message, pointed two by R0, to the appropriate destination thread. The message block pointed to by R0 contains the following words:

1. RESERVED
2. RESERVED
3. Source thread handle
4. Destination thread handle
5. Message number
6. Data field

The RESERVED fields should be empty. The only field that *must* be filled in is the destination thread handle; the source thread handle will be useful, though, if the destination thread expects to be able to reply.

The message number and data fields of the message are both totally arbitrary values; there are no standards, it is up to the software author what to use. Several points must be observed:

1. A message block can only exist in one thread's message queue
2. A message block can only exist once in a given queue
3. Messages can only be sent between threads in the same task. If inter-task communication is required, use the Window Manager's own form of messaging. This could easily be extended to allow the encapsulation of a thread message within a WIMP message, should it be desired.

5.10 SWI `Thread_Receive` (&4DA49)

On entry:

Register Contents
R0 = Boolean flag

On exit:

Register Contents
R0 → Message (or NULL if flag set, and no messages waiting)

The thread that called the SWI receives a message from its incoming message list, if there is one present. If not, the action can be twofold:

1. If the flag in R0 is NULL, then the thread will be suspended until a message arrives
2. If the flag in R0 is TRUE, then a NULL pointer will be returned in R0

5.11 SWI `Thread_Purge` (&4DA4A)

On entry:

Register Contents
R0 = Thread handle
R1 = Termination flag

On exit:

Register Contents
R1 → Next message (if Termination flag is NULL)

This SWI should be used with care, either when a thread is in serious trouble, or during debugging. The action of this call is twofold:

1. If the Termination flag in R1 is NULL, the call will return a pointer to the next message in a given thread's incoming queue, after removing it, and re-starting the thread if it was waiting for a message from the queue. If the queue is empty, it will return NULL.

2. If the termination flag is TRUE, then the incoming message count in the given thread's queue will be set to zero. The call should only be used in this way after the any messages have been sucked out as above.

5.12 SWI `Thread_Sleep` (&4DA4B)

On entry:

Register Contents
R0 = Time to sleep in centiseconds

The thread that makes this call will go to sleep for the number of centiseconds specified in R0.

5.13 SWI `Thread_Context` (&4DA4C)

On entry:

Register Contents
R0 = Thread handle (or NULL)

On exit:

Register Contents
R0 → Context (or boolean)

Gets a pointer to a thread's context, given a thread handle. The form of the context is documented, in C-style syntax, in `ThreadLib.h.Thread`. The main use of this call is to examine a thread's register set, and allow reading and writing of it's private word (a user-supplied value unique to each thread).

If, however, R0 is NULL on entry, R0 will contain, on exit, a boolean dictating whether or not the task is currently single-threaded (note, in this case, single threaded means *all threads are disabled* not that there is only one thread running).

5.14 SWI `Thread_Zap` (&4DA4D)

The thread that calls this will be terminated.

5.15 SWI `Thread_ZapAnother` (&4DA4E)

This SWI has not been tested yet, so we do not recommend its use.

5.16 SWI `Thread_Self` (&4DA4F)

On exit:

Register Contents
R0 = Handle of calling thread

If a thread calls this SWI, its handle will be returned.

5.17 SWI `Thread_Yield` (&4DA50)

A thread can call this SWI at will, in order to yield CPU time to the rest of the system, if it feels it isn't doing anything particularly important or time-critical.

5.18 *ThreadLicence

Displays the licence for the `Thread` module.

5.19 *ThreadStatus

Displays the status of the `Thread` module. It lists the tasks currently registered with `Thread`, which one is active, together with the total number of tasks. Each task also has listed along with it the handle of its currently active thread, and the status of all the other threads in the queue. The meanings of the thread statuses are as follows:

- 0: Virgin (ready to run).
- 1: Running (being executed, or eligible to be).
- 2: Semaphore wait.
- 3: Time wait (sleeping).

A Bugs, crashes, problems and contacts

A.1 Bugs

This implementation of `Thread` is no doubt not perfect, and contains bugs. The most noticeable problem is that timings, as displayed by `*ThreadStatus`, are highly inaccurate. Also, timings when calling SWI `Thread_Sleep` aren't too great, either. This will be fixed soon. Also, the code's error checking, to make sure that thread handles, task handles, semaphore pointers , messages etc. are valid is minimal to say the most. This will be improved.

A.2 The Toolbox

There appear to be some problems whilst using `Thread` with the Toolbox, that cause applications on the desktop to occasionally crash whilst receiving events. This does not seem to affect (to the extent of our testing, anyway) other WIMP programs, though.

A.3 Crashes

Programs running under `Thread`, and hopefully not very often `Thread` itself, may well crash. If this is the case, then they will probably go out in a big bang. By typing `*ThreadStatus`, you may notice that the task is still registered with `Thread`. This is not a problem; it merely increases RMA usage. In order to rectify the situation, the current solution is to reload `Thread`. `Thread` will be extended to detect crashes shortly.

A.4 Problems

Thread descriptors, task descriptors and 4Kb stacks are currently allocated from the RMA, occasionally leading to fragmentation. In a not too distant release, these will move into a dynamic area on RiscPC machines.

A.5 The Shared C Library

Just briefly... this shouldn't go here, but the Shared C Library is *not* reentrant, and thus should be treated as a mutually exclusive resource. In particular, do not allocate and free memory at the same time, etc. Liberal use of semaphores is required. For example, the message passing example in the appendix uses a semaphore to make sure memory is not allocated and freed at the same time.

There also exists a function in the `ThreadLib` library called `thread_kludge()`. This modifies the entry vectors for the functions `malloc()`, `calloc()`, `realloc()`, `free()`, and `x$stack_overflow` and places a semaphore round them, to avoid heap modification functions being called at once. These functions may indeed be called from within the shared library itself; unfortunately, until we hear back from ART, there is nothing we can do about it. Many thanks to Miles Sabin for pointing this out, especially the stack extension bit. Note that `thread_kludge` has *not* been extensively tested; the source code is provided to get the general idea, though.

A.6 Contacts

If you wish to use `Thread`, or a future version, in a commercial, profit making application, please contact SIMTEC for details of licencing. Otherwise feel free to use it. If you do plan to use the product, please E-Mail `thread@simtec.demon.co.uk` letting us know, so we can keep you up to date with announcements. Alternatively, just call SIMTEC for more information.

SIMTEC's address is:

SIMTEC ELECTRONICS,
Avondale Drive,
Tarleton,
Preston,
Lancs PR4 6AX (phone: (01772) 812863, e-mail: thread@simtec.demon.co.uk)

B An introduction to concurrent programming

Concurrent programming (writing multithreading programs) and sequential programming (writing *normal* RISCOS programs) are completely different kettles of fish. In a multithreaded execution environment, execution and completion order of threads are both undefined, and each thread must be *mutually exclusive*—in essence, each thread will need to access a resource (memory, disc, etc.). If more than one thread has access to such a resource at one time, then this will cause chaos. Threads may also wish to communicate with one another, either to allow synchronisation, or pass messages. A good book to look at is Tannenbaum's book on Modern Operating Systems.

B.1 Pitfalls

Concurrent and multithreaded programming have evolved into quite an art over the years; several techniques have been developed to aid mutual exclusion, message passing, and illustrate problems that may be encountered. This subsection aims to illustrate these problems, and the next subsection illustrates, in more detail, the methods that `Thread` uses to both enforce mutual exclusion and allow message passing.

B.1.1 The Dining Philosopher's problem

Many problems and their solutions have very fancy names in concurrent programming; this is one such problem, used to illustrate some problems that can arise without mutual exclusion, namely:

- Deadlock
- Infinite overtaking

In ancient times, a wealthy philanthropist endowed a college to accommodate five eminent philosophers. Each philosopher had a room in which he could engage in his professional activity of thinking; there was also a common dining room, furnished with a circular table, surrounded by five chairs, each labelled with the name of the philosopher who was to sit on it. To the left of each philosopher laid a golden fork, and in the centre stood a large bowl of spaghetti which was being constantly replenished.

A philosopher was expected to spend most of his time thinking; but when he felt hungry, he went into the dining room, sat down at his chair, picked up the golden fork on his left, and plunged it into the spaghetti. However, such was the tangled nature of the spaghetti that a second fork was required to lift it to the mouth. The philosopher, therefore, also had to pick up the fork on his right. When he was finished he would put the forks down, get up from his chair, and continue to think. Of course, a fork can only be used by one philosopher at a given time. If the philosopher on his right wants to use it, he has to wait until it is free again.

There are two problems. If all of the philosophers get hungry at the same time, and each picks up his fork and plunges it into the spaghetti at the same time, then each will not be able to get another fork to carry it to his mouth, and the situation is said to be one of *deadlock*. The other danger is that the left neighbour will always beat a philosopher to his fork. This is a case of *infinite overtaking*.

In this example, a philosopher can be thought of as a thread, and a fork a mutually exclusive resource.

B.1.2 Livelock

1. *Deadlock* can be summed up as a state in which threads are blocked while waiting for something which will never happen;

2. *Livelock* is a condition in which processes are executing instructions uselessly, in the sense that they will never make constructive progress.

B.1.3 Race Conditions

These are often the hardest of all things to track down. If there is a resource of some sort, occasionally one thread may access it before another, causing a problem. The only way to avoid this happening is to write code very carefully in the first place, and plan it in detail.

B.2 Avoidance of pitfalls

B.2.1 Mutual exclusion using semaphores

Dijkstra introduced the semaphore in 1968. A semaphore is a boolean data type, with values 1 or 0. It has two operations on it, (with two names, the single character being from the Dutch equivalent of the English words):

wait(P): decrement the specified semaphore by 1, providing the result is non-negative, otherwise deschedule the calling thread until an appropriate **signal** is received on that semaphore

signal(V): increment the specified semaphore by 1, providing that there is no process already descheduled on it. Otherwise, reschedule the waiting process.

`Thread` provides a set of SWIs for implementing semaphores in user-level processes.

B.2.2 Counters (or "counting semaphores")

A counting semaphore (or "counter" or "general semaphore") is similar to a binary semaphore (above), however it is allowed to hold values greater than 1.

B.2.3 The Banker's algorithm

Suppose there are 12 instances of a shared resource. Never allocate all of the resource, such that all processes can complete properly ie. impose a process-dependant limit on each resource.

B.2.4 Condition Critical Regions and Monitors

These two approaches to writing complex multi-threaded programs both need to be implemented early in the program design phase.

CCRs involve dividing the program into groups of procedures/functions that can only be executed when given global conditions are True. Compilers are available (though not necessarily for the Acorn platform) which will develop CCR expressions at compile-time.

Monitors rely on the coder dividing resources, and the operations that may need to be performed upon them, into individual objects. That way, each process can keep track, with semaphores or the like, of exactly what is happening.

C Message passing in C

This is a trivial bit of C code that passes a textual message between two threads. Because a message may not be included in a message queue more than once, each message here is dynamically allocated.

```
/* This is the thread that sends messages, saying "fish and chips", at
 * random intervals to the main task, which then prints them out.
 */
void thread_a(thread_handle me, thread_handle dad, int argument)
{
    thread_message *m;

    while (1)
    {
        thread_sleep(rand() & 255); /* Wait for a random amount of time */
        thread_wait(&tweak_mem, 0);
        m = malloc(sizeof(thread_message));
        thread_signal(&tweak_mem);
        if (m == 0)
        {
            printf("Error allocating memory for message!\n");
            exit(0);
        }
        memset(m, 0, sizeof(thread_message));
        m->from = me;
        m->to = dad;
        m->no = 0;
        m->data = (int) "Fish and chips!";
```

```
        thread_send(m);
    }
}

/* This is called at exit time
 */
void at_exit(void)
{
    thread_closedown();
}

int main(void)
{
    thread_handle thread_a_h, my_thread;
    task_handle my_task;
    thread_message *m;

    /* Make sure thread_closedown() is called when we quit
     */
    atexit(at_exit);

    /* Kludge the SharedCLibrary vectors
     */
    thread_kludge();

    /* Start the threads and initialise the semaphore
     */
    thread_initialise("Simtec Demo 1", NULL, &my_task, &my_thread);
    thread_init_semaphore(&tweak_mem, 1); /* Resource is initially free */
    thread_begin("Child", 0, 0, (thread_func) thread_a, &thread_a_h);

    /* Enter into a loop, receiving and freeing messages
     */
    while (1)
    {
        m = thread_receive(0);
        printf("Got message %s\n", (char *) m->data);

        /* Free up the message's memory
         */
        thread_wait(&tweak_mem, 0);
        free(m);
        thread_signal(&tweak_mem);
    }
}
```