

## **A Threads Tutorial<sup>†</sup>**

*Thomas W. Doepner Jr.  
Department of Computer Science  
Brown University  
Providence, RI 02912*

*March 30, 1987  
Revised December 16, 1987 and September 16, 1988*

*Technical Report CS-87-06*

---

<sup>†</sup>This work has been partially supported by grants from the Digital Equipment Corporation and by the Encore Computer Corporation.

## 1. Introduction

Threads is a system for the efficient support of concurrency. It runs on either single-processor or multiprocessor computers and presents to the programmer the same “view of the world” no matter how many processors are available for use. This view is that a number of concurrent *threads of control* are executing in a single shared address space and share a common view of which files are open. Thus threads may communicate very efficiently through this shared memory and all threads may participate in I/O on any file.

In this paper we discuss the most important routines provided by the Threads system and give examples of their use. Where possible, we try to give “realistic” examples, but often realism is sacrificed in favor of toy examples intended to demonstrate a feature simply.

Threads is still an evolving system. We would appreciate any form of feedback on either the contents of this document or the system itself.

## 2. Creating Threads

The Threads systems is initiated by calling *THREADgo*:

```
int
THREADgo(nr_procs, data_size, func, args, argsize, stacksize, priority)
int nr_procs;
int data_size;
void (*func)();
int *args;
int argsize;
int stacksize;
int priority;
```

This allocates *nr\_procs* processors for use by the Threads system. It sets an upper bound of *data\_size* bytes for representing additional threads and the shared heap (in addition to the UNIX data space already owned by the caller; the UNIX stack segment is not available to the user of Threads). (The *nr\_procs* and *data\_size* arguments are ignored on the uniprocessor implementations: there will of course be only one processor; the size of the heap will be limited by whatever constraints there are on the size of the data region of the underlying UNIX process.)

The (single) initial thread starts execution by calling *func*. It is passed a single argument *args* (which might be either a value or a pointer to a value or set of values). If *argsize* is zero, then *args* is passed to the

thread unchanged. However, if *argsize* is nonzero, then *argsize* bytes of data pointed to by *args* are placed on the new thread's stack and the thread is passed a pointer to this location instead of the original value of *args*. (Thus if arguments are copied to the thread's stack, they will be private to that thread, but if they are not copied, they are accessible by other threads.)

The newly created thread is given a stack of maximum size *stacksize*. Unlike stacks for UNIX processes, there is no hardware assistance in monitoring the growth of a thread's stack, and there is no mechanism for growing a stack once it has achieved its maximum size. The Thread runtime does check the extent of a thread's stack "when convenient," but it is unlikely to detect an overextended stack until well after it has overextended. Thus it is a good idea to be liberal in estimating *stacksize* (library routines such as *printf* use a surprising amount of stack space).

Finally, the new thread is given a runtime priority of *priority*. Priorities range from 0 to 31, with 0 being the best priority.

Any thread may create a new thread with the *THREADcreate* call:

```
THREAD
THREADcreate(function, args, argsize, detached, stacksize, priority)
int (*function)();
int *args;
int argsize;
char detached;
int stacksize;
int priority;
```

This call creates a new thread and returns that thread's *handle*, of type *THREAD*. (The handle is used to refer to a thread in many of the routines that follow; a thread's own handle will always be in *THREAD-current*.) The arguments *func*, *args*, *argsize* and *priority* are the same as in *THREADgo*. The additional argument *detached* determines if the new thread's termination is to be synchronized with its creator (i.e. parent). If *detached* is 1, then the child thread bears no relation to its parent: it is totally independent. If *detached* is 0, then the parent-child relationship is maintained. The parent will not be able to terminate until the child terminates, and the parent may execute a call to wait until the child terminates (*THREADwaitforchild*). Furthermore, when the child terminates it may return a value to its parent (which the parent obtains by calling *THREADreturnvalue*). Finally, after the return value has been obtained, the storage occupied by the thread should be freed by a call to *THREADeliminatechild*.

Thus, the proper way for a parent to synchronize with a nondetached terminating child is as follows:

```
THREAD child;
int returnvalue;
.
.
.
child = THREADwaitforchild();
returnvalue = THREADreturnvalue(child);
THREADEliminatechild(child);
```

One should be careful when creating threads on a multiprocessor system; it is likely that a child thread will have begun execution before the parent thread returns from the *THREADcreate* call. If this could result in a race condition, then some explicit form of synchronization should be used in the two threads.

A thread terminates when it returns from its first function. *THREADmurder* may be used to kill off a thread (either the caller or any other thread that is not an ancestor of the caller).

In the following (simplistic) example, we use multiple processors for multiplying matrices.

```
#include <thread.h>
#include <stdio.h>

int A[9] = { 1, 2, 3,
            4, 5, 6,
            7, 8, 9};

int B[9] = { 9, 8, 7,
            6, 5, 4,
            3, 2, 1};

int C[9];

main(argc, argv)
int argc;
char *argv[];
{
    extern void startup();

    if (argc != 2) {
        fprintf(stderr, "usage: tst #processors\n");
        exit(1);
    }

    THREADgo(atol(argv[1]), 2*1024*1024, startup, 0, 0, 20*1024, 2);
}

void
startup()
```

```

{
extern void mult();
struct {
    int i;
    int j;
} ij;

for (ij.i=0; ij.i<3; ij.i++)
    for (ij.j=0; ij.j<3; ij.j++)
        THREADcreate(mult, &ij, sizeof(ij), 0, 20*1024, 2);

while(THREADwaitforchild())
    ;

printf("%3d %3d %3d\n %3d %3d %3d\n %3d %3d %3d\n",
        C[0], C[1], C[2], C[3], C[4], C[5], C[6], C[7], C[8]);
}

void
mult(ij)
struct {
    int i;
    int j;
} *ij;
{
    register int i;
    register int t=0;
    register int col = 3 * ij->i;      /* row i, col 0 */
    register int row = ij->j;        /* row 0, col j */

    for(i=0; i<3; i++) {
        t += A[col] * B[row];
        col++;
        row += 3;
    }

    C[3*ij->i + ij->j] = t;
}

```

### 3. Synchronization

#### 3.1. Monitors

Monitors are the standard synchronization mechanism in this version of Threads. We do not provide any syntactic support for monitors; instead, we provide a set of function calls which form the runtime support for monitors.

A monitor must be created by a thread before any thread can use it. This is done by a call to *THREADmonitorinit*:

```

THREAD_MONITOR
THREADmonitorinit(conditions, resetfunc)
int conditions;
void (*resetfunc)();

```

The *condition* argument gives the number of condition queues (inside the monitor) to be created; *resetfunc* is called if an exception (see below) is raised in a thread that is active in the monitor (it may be set to NULL). *THREADmonitorinit* returns a handle for the newly created monitor, of type *THREAD\_MONITOR*, which is used to identify the monitor.

In its simplest form, a monitor may be used to provide mutually exclusive access to a shared data structure. To achieve this, a thread calls *THREADmonitorentry* before accessing the data structure and *THREADmonitorexit* afterwards:

```

void
THREADmonitorentry(monitor, manager)
THREAD_MONITOR monitor;
THREAD_MANAGER manager;

void
THREADmonitorexit(monitor)
THREAD_MONITOR monitor;

```

Here *monitor* is the handler of the monitor that has been created to protect the data structure. *manager* is the address of a data structure of type *THREAD\_MANAGER* that may be used by the monitor to help it deal with exceptions. (*Managers* are used strictly by the Thread runtime, they are not directly used by users of threads. The only reason they appear to the user at all is that the user can often allocate them more efficiently than can the runtime code.) If the address is NULL, then the Thread runtime allocates this data structure from the shared heap. For fastest performance, the caller should allocate it on its stack, passing the address to *THREADmonitorentry*. *manager* is a pointer to an area of type *THREAD\_MANAGER\_BLOCK*; this storage needs a lifetime long enough that it will still exist when *THREADmonitorexit* is called.

Good programming style dictates that the thread functions supporting monitors should be used as if the language supported the monitor concept – an abstract data type consisting of shared data, initialization code, and a set of access functions. Each of these access functions should start with a call to *THREADmonitorentry* and end with a call to *THREADmonitorexit*. The following is a simple example of the use of

a monitor to prompt for data on a terminal and read it.

```
#include <thread.h>
#include <stdio.h>

extern void promptandread(), output();
THREAD_MONITOR prmonitor;

main(argc, argv)
int argc;
char *argv[];
{
    extern void startup();

    if (argc != 2) {
        fprintf(stderr, "usage: tst #processors\n");
        exit(1);
    }

    THREADgo(atol(argv[1]), 1024*1024, startup, 0, 0, 20*1024, 2);
}

void
startup()
{
    extern void child();

    prmonitor = THREADmonitorinit(0, NULL);

    THREADcreate(child, "prompt1 ", 0, 0, 20*1024, 2);
    THREADcreate(child, "prompt2 ", 0, 0, 20*1024, 2);
}

void
child(prompt)
char *prompt;
{
    char buf[80];

    promptandread(prompt, buf, 80);
    output(buf, strlen(buf));
    promptandread(prompt, buf, 80);
    output(buf, strlen(buf));
}

void
promptandread(prompt, buf, buflen)
char *prompt;
char *buf;
int buflen;
{
    THREAD_MANAGER_BLOCK manager;

    THREADmonitoreentry(prmonitor, &manager);

    write(1, prompt, strlen(prompt));
}
```

```

        read(0, buf, buflen);

    THREADmonitorexit(prmonitor);
}

void
output(buf, len)
char *buf;
int len;
{
    THREAD_MANAGER_BLOCK manager;

    THREADmonitoretry(prmonitor, &manager);

    write(1, buf, len);

    THREADmonitorexit(prmonitor);
}

```

Monitors provide more than just simple mutual exclusion. It is often necessary for a thread to modify the monitor-protected shared data only if certain conditions are true, where the condition itself must be checked inside the monitor. This is accomplished by using a condition queue: a thread enters a monitor and checks a condition (in mutual exclusion). If the condition is true the thread continues, otherwise the thread suspends itself (by calling *THREADmonitorwait*). After some second thread has satisfied the condition that the first thread was waiting for, this second thread may wake up the first thread as it exits the monitor by calling *THREADmonitorsignalandexit* instead of *THREADmonitorexit*. There are often times when it is necessary for a thread which is about to suspend itself in a monitor to signal a condition before suspending itself. This is done via a call to *THREADmonitorsignalandwait*.

```

int
THREADmonitorwait(monitor, condition)
THREAD_MONITOR monitor;
int condition;

int
THREADmonitorsignalandexit(monitor, condition)
THREAD_MONITOR monitor;
int condition;

int
THREADmonitorsignalandwait(monitor, signalcondition, waitcondition)
THREAD_MONITOR monitor;
int signalcondition, waitcondition;

```

*condition* identifies the condition queue to which the thread is referring (numbering starts at 0). A thread calling *THREADmonitorwait* is suspended and placed at the end of the queue of the specified condition. It

is resumed when it is first in that queue and a signal is sent for that condition (by some thread executing *THREADsignalandexit* or *THREADsignalandwait*). A thread calling *THREADsignalandwait* signals *signalcondition* and then is suspended and placed on the condition queue specified by *waitcondition*.

For an example, we give a solution to the standard producer-consumer problem. In our solution, a producer may supply more than the buffer can hold and the consumer may request more than the buffer can hold, so transfer might be incremental.

```
#include <thread.h>
#include <stdio.h>

#define BUFSIZE 80

struct buffer {
    THREAD_MONITOR mon;
    char xbuf[BUFSIZE];
    char *in, *out;
    int nempty, nfull;
    int alldone;
};

extern void produce(), flush();
extern int consume();
extern struct buffer *prodconinit();

main(argc, argv)
int argc;
char *argv[];
{
    extern void startup();

    if (argc != 2) {
        fprintf(stderr, "usage: tst #processors\n");
        exit(1);
    }

    THREADgo(atol(argv[1]), 1024*1024, startup, 0, 0, 20*1024, 2);
}

void
startup() {
    extern void pro(), con();
    THREAD producer, consumer;
    struct buffer *buffer;

    buffer = prodconinit();

    producer = THREADcreate(pro, buffer, 0, 0, 12*1024, 2);
    consumer = THREADcreate(con, buffer, 0, 0, 12*1024, 2);
}
```

```

void
pro(buffer)
struct buffer *buffer;
{
    int count;
    char buf[80];

    for (;;) {
        if ((count = read(0, buf, 80)) <= 0)
            break;
        produce(buffer, buf, count);
    }
    flush(buffer);
}

void
con(buffer)
struct buffer *buffer;
{
    int count;
    char buf[4];

    for (;;) {
        if ((count = consume(buffer, buf, 4)) == 0)
            return;
        write(1, buf, count);
    }
}

/* The "monitor" starts here */

/* this is the initialization and data structure part */

struct buffer *
prodconinit()
{
    struct buffer *b;

    b = (struct buffer *)malloc(sizeof(struct buffer));
    b->mon = THREADmonitorinit(2, NULL);
    b->nfull = b->alldone = 0;
    b->in = b->out = b->xbuf;
    b->nempty = BUFSIZE;
    return(b);
}

int
consume(buffer, buf, n)
struct buffer *buffer;
char *buf;
int n;
{
    register int count, count2;
    register int totcount = 0;
    register char *lin = buf;
    THREAD_MANAGER_BLOCK manager;
    register int bytesleft;

```

```

THREADmonitoreentry(buffer->mon, &manager);

bytesleft = (int)(BUFSIZE - (buffer->out - buffer->xbuf));
/* bytesleft is the size of that portion of the buffer beyond the
   pointer "out" */

while (n > 0) { /* does the caller want more? */
    while (buffer->nfull <= 0) { /* are there more bytes in the buffer? */
        if (buffer->alldone) { /* will there ever be more bytes? */
            THREADmonitorexit(buffer->mon);
            return(totcount);
        }
        THREADmonitorsignalandwait(buffer->mon, 1, 0);
        /* wait for bytes to appear */
    }
    /* how many bytes can we take now? */
    count = (buffer->nfull <= n) ? buffer->nfull : n;
    if (count > bytesleft) { /* does the buffer "wrap around"? */
        bcopy(buffer->out, lin, bytesleft);
        count2 = count - bytesleft;
        bcopy(buffer->xbuf, lin+bytesleft, count2);
        buffer->out = buffer->xbuf + count2;
        bytesleft = BUFSIZE - count2;
    } else {
        bcopy(buffer->out, lin, count);
        buffer->out += count;
        bytesleft -= count;
    }
    buffer->nfull -= count;
    buffer->nempty += count;
    lin += count;
    totcount += count;
    n -= count;
}
THREADmonitorsignalandexit(buffer->mon, 1);
return(totcount);
}

void
produce(buffer, buf, n)
struct buffer *buffer;
char *buf;
int n;
{
    register int count, count2;
    register char *lout = buf;
    THREAD_MANAGER_BLOCK manager;
    register int spaceleft;

    THREADmonitoreentry(buffer->mon, &manager);

    spaceleft = (int)(BUFSIZE - (buffer->in - buffer->xbuf));
    /* spaceleft is the size of that portion of the buffer beyond the
       pointer "in" */

    while (n > 0) { /* is there more to transfer? */
        if (buffer->nempty <= 0) /* is there room for it? */

```

```

    THREADmonitorsignalandwait(buffer->mon, 0, 1);
    /* how much shall we transfer? */
    count = (buffer->nempty <= n) ? buffer->nempty : n;
    if (count > spaceleft) { /* does the buffer "wrap around"? */
        bcopy(lout, buffer->in, spaceleft);
        count2 = count - spaceleft;
        bcopy(lout+spaceleft, buffer->xbuf, count2);
        buffer->in = buffer->xbuf + count2;
        spaceleft = BUFSIZE - count2;
    } else {
        bcopy(lout, buffer->in, count);
        buffer->in += count;
        spaceleft -= count;
    }
    buffer->nempty -= count;
    buffer->nfull += count;
    lout += count;
    n -= count;
}
THREADmonitorsignalandexit(buffer->mon, 0);
}

void
flush(buffer)
struct buffer *buffer;
{
    THREADmonitoreentry(buffer->mon, NULL);

    buffer->alldone++;

    THREADmonitorsignalandexit(buffer->mon, 0);
}

```

### 3.2. Semaphores

Threads supplies an alternative synchronization mechanism in the form of semaphores. The use of semaphores is not strongly encouraged; being the “go to” of concurrent programming, they are not terribly well-structured, and furthermore (in this implementation) they are not safe for use with exceptions, as monitors are. However, semaphores are implemented very efficiently and perhaps should be used where simple mutual exclusion is required and speed is of the essence.

To create a semaphore, one calls *THREADseminit*:

```

SEMAPHORE
THREADseminit(initialvalue)
int initialvalue;

```

*initialvalue* is the initial value given to the semaphore. The value returned is the handle for referencing the semaphore and is of type *SEMAPHORE*. *P* and *V* operations on semaphores are performed using

*THREADpsem* and *THREADvsem*:

```
void  
THREADpsem(sem)  
SEMAPHORE sem;
```

```
void  
THREADvsem(sem)  
SEMAPHORE sem;
```

*sem* is the handle of the relevant semaphore. *THREADpsem* decrements the value of the semaphore by 1 if the result will be nonnegative; otherwise the caller is suspended and is queued at the end of a queue associated with the semaphore. *THREADvsem* releases the first thread on the semaphore's queue, if there is one; otherwise, it increments the value of the semaphore by 1.

In the following simple example, a number of threads should execute a loop for a total (over all threads) of 1000 times:

```
#include <thread.h>  
#include <stdio.h>  
  
SEMAPHORE sem;  
int count;  
  
main(argc, argv)  
int argc;  
char *argv[];  
{  
    extern void startup();  
  
    if (argc != 2) {  
        fprintf(stderr, "usage: tst #processors\n");  
        exit(1);  
    }  
  
    THREADgo(atol(argv[1]), 1024*1024, startup, 0, 0, 20*1024, 2);  
}  
  
void  
startup()  
{  
    extern int child();  
    THREAD tcb;  
    int i;  
    int total_iterations = 0;  
  
    sem = THREADseminit(1);  
  
    for (i=0; i<10; i++)  
        THREADcreate(child, 0, 0, 0, 20*1024, 2);
```

```

while((tcb = THREADwaitforchild()) != NULL) {
    /* wait for the children to terminate */
    total_iterations += THREADreturnvalue(tcb);
    THREADdelimitatechild(tcb);
}

printf("count = %d, total iterations = %d\n", count, total_iterations);
fflush(stdout);
}

int
child()
{
    int i;

    for(i=0;i++) {
        THREADpsem(sem);
        if (count >= 1000)
            break;
        count++;
        THREADvsem(sem);
    }
    THREADvsem(sem);
    return(i);
}

```

#### 4. Thread Control Blocks

As has been discussed, associated with every thread are a stack and a thread control block. The thread's handle is actually a pointer to the thread control block, in which a fair amount of per-thread information is stored. The application programmer may also store per-thread information. To do this, one must request that a certain amount of additional space be allocated in each thread control block and must provide a set of functions for initializing and cleaning up this space. To accomplish this, the function *THREADtype1\_register* must be called (before *THREADgo* is called). It may be called any number of times.

```

int
THREADtype1_register(size, threadinitfunc, threadterminatefunc,
    systeminitfunc, systemterminatefunc)
int size;
void (*threadinitfunc)();
void (*threadterminatefunc)();
void (*systeminitfunc)();
void (*systemterminatefunc)();

```

*size* is the amount of additional space needed in each thread control block. The first two functional arguments are called as each thread is created and terminated; the second two functional arguments are

called as the system is initialized and shut down. *threadinitfunc* is called in the context of the creating (parent) thread with a single argument – the handle of the newly created thread. *threadterminatefunc* is called in the context of whichever thread is causing this thread to terminate; it is passed the handle of the dying thread.

The value returned by *THREADtype1\_register* is the offset within the thread control block of the location where the application information resides. To convert this offset into a pointer, one calls

*THREADdynamic*:

```
char *  
THREADdynamic(tcb, offset)  
THREAD tcb;  
int offset;
```

Normally this is called as part of a programmer-defined macro, as seen below.

In the following example, each thread is given its own private storage (heap), which is found by looking in an area registered in the tcb.

```
#include <thread.h>  
#include <stdio.h>  
  
#define HEAPSIZE      1024  
int heap_offset;  
#define heap(tcb) ((char **)THREADdynamic(tcb, heap_offset))  
  
main(argc, argv)  
int argc;  
char *argv[];  
{  
    extern void startup();  
    extern void allocate_heap(), free_heap();  
  
    if (argc != 2) {  
        fprintf(stderr, "usage: tst #processors\n");  
        exit(1);  
    }  
  
    heap_offset = THREADtype1_register(sizeof(char *),  
        allocate_heap, free_heap, NULL, NULL);  
  
    THREADgo(atol(argv[1]), 1024*1024, startup, 0, 0, 20*1024, 2);  
}  
  
void  
allocate_heap(tcb)  
THREAD tcb;  
{
```

```

char **heapaddr = heap(tcb);

*heapaddr = (char *)malloc(HEAPSIZE);
}

void
free_heap(tcb)
THREAD tcb;
{
    free(*heap(tcb));
}

void
startup()
{
    extern int child();
    THREAD tcb;

    THREADcreate(child, 0, 0, 0, 10*1024, 2);
    THREADcreate(child, 0, 0, 0, 10*1024, 2);
}

int
child()
{
    char *privateheap = heap(THREADcurrent);

    /* Replace this line with code that uses the thread's private storage */
}

```

## 5. Exceptions and Interrupts

Interrupts are causes and exceptions are effects, though interrupts do not always cause exceptions and exceptions are not always the effect of interrupts. In Threads, interrupts are the manifestation of UNIX signals; exceptions are the forced response in a thread to interrupts and certain actions of other threads.

### 5.1. Exceptions

The programmer may establish an exception handler for each “module” through which a thread passes. When an exception is raised in a thread, that thread’s stack is popped back to the point at which the exception handler was established, the handler is called from that point, and, on return from the handler, control resumes (at that point). The idea is that a failure in a module makes control roll back to a “safe point,” the exception handler is called to clean up, and then normal execution resumes.

An exception handler is established by calling *THREADsetexception*:

```
int
THREADsetexception(handler, oldstate)
int (*handler)();
EXCEPTION *oldstate;
```

*handler* is the address of the routine which is to be called in the event of an exception. *THREADsetexception*, much like *setjmp*, returns 0 when it is called and returns 1 when control resumes after the exception handler has been called in response to an exception. The exception handler itself may return a value; this may be obtained after it returns by calling *THREADgetexceptionreturn*. *oldstate* is the address of an area of memory into which will be placed information describing the previous exception handler. This space can be allocated by calling *THREADgetexceptionspace* and freed by calling *THREADfreeexceptionspace*. A saved exception handler may be reinstated by calling *THREADrestoreexception*:

```
void
THREADrestoreexception(oldstate)
EXCEPTION oldstate;
```

This allows one to have a *dynamic chain* of exception handlers: as a thread enters a module, it saves the old exception handler, pushing it onto a stack, and establishes a new handler. When the thread leaves the module, it reestablishes the old handler, popping it off the stack.

When an exception handler is called, it is passed a single argument which represents the *type* of the exception. This type is completely uninterpreted by Threads, other than that it must be greater than zero, and that types 1 through 31 may represent UNIX signals. A thread may explicitly raise an exception in any thread (including itself) by calling *THREADraiseexception*:

```
int
THREADraiseexception(tcb, param)
THREAD tcb;
int param;
```

*tcb* is the handle of the thread in which the exception is to be raised; *param* is the “exception type,” the parameter which is passed to the exception handler.

If a monitor is being executed when an exception occurs, then the system state of the monitor is cleaned up (i.e., the internal queues of threads are brought to a consistent state so that normal use of the monitor may continue) and, if the programmer has defined a reset function in the original call to *THREADmonitorinit*, then this function is called with its argument set to the handle of the monitor. This routine’s

purpose is to clean up the user-defined monitor state (e.g. it might reset the values of monitor-protected data structures).

In the following example, we create a chain of exception handlers as the child thread calls module1 and then module2, which is a monitor.

```
#include <thread.h>
#include <stdio.h>

THREAD_MONITOR tmonitor;

main(argc, argv)
int argc;
char *argv[];
{
    extern void startup();

    if (argc != 2) {
        fprintf(stderr, "usage: tst #processors\n");
        exit(1);
    }

    THREADgo(atol(argv[1]), 1024*1024, startup, 0, 0, 20*1024, 2);
}

void
startup()
{
    extern void child();
    extern void monitorreset();
    THREAD tcb;
    int i;

    tmonitor = THREADmonitorinit(0, monitorreset);

    tcb = THREADcreate(child, 0, 0, 0, 10*1024, 2);

    /* idle for a bit, then raise an exception in the child */

    for (i=0; i<2000; i++)
        ;

    if (THREADraiseexception(tcb, 17) == 0) {
        printf("raiseexception failed\n");
    }
}

void
child()
{
    extern int handler1();
    extern void module1();
    int i;
```

```

    if (THREADsetexception(handler1, NULL)) {
        printf("return from handler1, return value = %d\n",
            THREADgetexceptionreturn());
        return;
    }

    for (i=0; i<200; i++)
        ;

    module1();
}

void
module1()
{
    EXCEPTION oldhandler = THREADgetexceptionspace();
    extern int handler2();
    extern void module2();
    int i;

    if (THREADsetexception(handler2, oldhandler)) {
        printf("return from handler2, return value = %d\n",
            THREADgetexceptionreturn());
        return;
    }

    for (i=0; i<20; i++)
        ;

    module2();

    THREADrestoreexception(oldhandler);
    THREADfreeexceptionspace(oldhandler);
}

void
module2()
{
    EXCEPTION oldhandler = THREADgetexceptionspace();
    extern int handler3();
    THREAD_MANAGER_BLOCK manager;
    int i;

    if (THREADsetexception(handler3, oldhandler)) {
        printf("return from handler3, return value = %d\n",
            THREADgetexceptionreturn());
        return;
    }

    THREADmonitorentry(tmonitor, &manager);

    /* body of monitor */

    THREADmonitorexit(tmonitor);

    THREADrestoreexception(oldhandler);
    THREADfreeexceptionspace(oldhandler);
}

```

```

}

void
monitorreset(monitor)
THREAD_MONITOR monitor;
{
    /* Reset the internal state of the monitor.
       This is called if there is an exception in the monitor */
}

int
handler1(type)
int type;
{
    printf("exception occurred in handler1; type %d\n", type);

    return (1);
}

int
handler2(type)
int type;
{
    printf("exception occurred in handler2; type %d\n", type);

    return (2);
}

int
handler3(type)
int type;
{
    printf("exception occurred in handler3; type %d\n", type);

    return (3);
}

```

## 5.2. Interrupts

Using the *THREADregistersignal* call, the programmer specifies the response to any of the catchable UNIX signals. This response may be to cause an exception in the current thread, to create a new thread, or to suspend (freeze) the current thread and execute a handler on that thread's stack (an action similar to the way in which signals are handled in UNIX). The type of response is encoded in a structure of type *SIGHANDLER*:

```

typedef struct sighandler {
    int type;
    THREAD (*whichthread)();
    void (*func)();
    int stacksize;
}

```

```

    int priority;
} SIGHANDLER;

/* types: */
#define SIG_EXCEPTION      1
#define SIG_SENDSIG       2
#define SIG_CREATETHREAD  3

```

The *THREADregistersignal* call is as follows:

```

int
THREADregistersignal(signo, newhandler, oldhandler)
int signo;
SIGHANDLER *newhandler, *oldhandler;

```

If a signal occurs which has been registered to be of type SIG\_EXCEPTION, then an exception with parameter equal to the signal number is raised in the current thread.

If a signal occurs which has been registered to be of type SIG\_CREATETHREAD, then a new, detached thread of control is created and is passed one parameter – the signal number. This thread of control will start execution in the routine *func* which was given as part of the *THREADregistersignal* call.

If a signal occurs which has been registered to be of type SIG\_SENDSIG, then the routine *whithread* is called. If this routine returns NULL (or the address of the routine is NULL), then nothing happens. However, if the routine returns a non-NULL value, it is taken to be the handle of a THREAD. This thread is then *frozen* – it will not be allowed to execute until it is unfrozen, though it may be moved among the various queues. A new thread of control is created which will execute *func* and will be passed the signal number. This new thread is considered a child of the frozen thread, and uses the stack of its parent. When this thread of control terminates, the original thread of control will be unfrozen.

This latter behavior was designed to be the analogue of UNIX signal handling. An almost (but not quite!) analogue of the UNIX *kill* system call is *THREADfakesignal*:

```

void
THREADfakesignal(thread, func, priority, sig)
THREAD thread;
void (*func)();
int priority;
int sig;

```

This call causes *thread* to be frozen and a new thread to be created which executes *func*, is passed a single argument *sig* and runs at priority *priority*.

To determine if a particular thread is *frozen*, one may call *THREADfrozen*:

```
int
THREADfrozen(thread)
THREAD thread;
```

It returns *1* if the thread whose handle is *thread* is frozen, *0* otherwise.

If a thread has been created as the result of freezing its parent, its parent can be identified using the call *THREADfrozenparent*:

```
THREAD
THREADfrozenparent(thread)
THREAD thread;
```

The following example illustrates all three types of signal handling. A child thread causes a floating-point overflow, and this is converted into an exception. If the SIGINT signal is sent, it is handled via the creation of a new thread. If a SIGQUIT signal is sent, this is handled by freezing the main thread and then running a new thread on the main thread's stack.

```
#include <thread.h>
#include <stdio.h>
#include <signal.h>

THREAD mainthread;

main(argc, argv)
int argc;
char *argv[];
{
    extern void startup();

    if (argc != 2) {
        fprintf(stderr, "usage: tst #processors\n");
        exit(1);
    }

    THREADgo(atol(argv[1]), 1024*1024, startup, 0, 0, 20*1024, 2);
}

void
startup()
{
    extern void child();
    THREAD tcb;
```

```

SIGHANDLER sigh;
extern void intrhandler();
extern void sigquithandler();
extern THREAD chooser();
int i;

mainthread = THREADcurrent;

sigh.type = SIG_CREATETHREAD;
sigh.whichthread = NULL;
sigh.func = intrhandler;
sigh.stacksize = 10*1024;
sigh.priority = 2;

THREADregistersignal(SIGINT, &sigh, NULL);

sigh.type = SIG_EXCEPTION;

THREADregistersignal(SIGFPE, &sigh, NULL);
sigh.type = SIG_SENDSIG;
sigh.func = sigquithandler;
sigh.whichthread = chooser;

THREADregistersignal(SIGQUIT, &sigh, NULL);

tcb = THREADcreate(child, 0, 0, 0, 10*1024, 2);

/* don't do this if you have only one processor! */
for (i=0; i<1000000; i++)
    ;
}

void
child()
{
    extern void excphandler();

    if (THREADsetexception(excphandler, NULL)) {
        printf("return from excphandler\n");
    } else {
        /* cause a SIGFPE */
        kill(getpid(), SIGFPE);
    }
}

void
intrhandler(sig)
int sig;
{
    printf("signal %d\n", sig);
}

void
excphandler(param)
int param;
{
    printf("exception; param = %d\n", param);
}

```

```

}

THREAD
chooser(sig)
int sig;
{
    return(mainthread);
}

void
sigquithandler(sig)
int sig;
{
    printf("in sigquithandler\n");
}

```

## 6. Shared I/O

One can safely perform I/O in Threads just by making the appropriate system call or Standard I/O call. (Actually, Threads has “taken over” all I/O system calls and Standard I/O calls. It must make certain that a single thread never inadvertently blocks a processor by performing a blocking system call and it must enforce mutually exclusive access to the Standard I/O buffers.) If a computation needs to access one of several files (e.g. one might want to read from either the keyboard or from a socket, depending upon which one is ready first), then there are two approaches available. The first is to create separate threads to deal with each I/O device of interest. The second makes use of the Thread analogue of the UNIX *select* system call.

An example of the first approach is given below in which each I/O request is handled by a separate thread.

```

#include <thread.h>
#include <stdio.h>

extern THREAD NBread();
extern THREAD NBwrite();

main(argc, argv)
int argc;
char *argv[];
{
    extern void startup();

    if (argc != 2) {
        fprintf(stderr, "usage: tst #processors\n");
        exit(1);
    }
}

```

```

    THREADgo(atol(argv[1]), 1024*1024, startup, 0, 0, 20*1024, 2);
}

void
startup()
{
    char ibuf[100];
    char obuf[200];
    THREAD child;

    /* start a non-blocking read */
    NBread(0, ibuf, 100);

    sprintf(obuf, "output message\n");
    /* start a non-blocking write */
    NBwrite(1, obuf, strlen(obuf));

    while ((child = THREADwaitforchild()) {
        /* wait for the nonblocking I/O to finish */
        THREADdeliminatechild(child);
    }

    sprintf(obuf, "Input was: %s\n", ibuf);
    write(1, obuf, strlen(obuf));
}

struct args {
    int fd;
    char *buf;
    int len;
};

THREAD
NBread(fd, buf, len)
int fd;
char *buf;
int len;
{
    int Tread();
    struct args args;

    args.fd = fd;
    args.buf = buf;
    args.len = len;

    return(THREADcreate(Tread, &args, sizeof(args), 0, 20*1024, 1));
}

int
Tread(args)
struct args *args;
{
    return(read(args->fd, args->buf, args->len));
}

THREAD
NBwrite(fd, buf, len)

```

```

int fd;
char *buf;
int len;
{
    int Twrite();
    struct args args;

    args.fd = fd;
    args.buf = buf;
    args.len = len;

    return(THREADcreate(Twrite, &args, sizeof(args), 0, 20*1024, 1));
}

int
Twrite(args)
struct args *args;
{
    return(write(args->fd, args->buf, args->len));
}

```

In the second approach a single thread may wait for any one of many I/O devices to be ready to transfer and then perform the appropriate transfer request. To avoid any possible race conditions, this is all done within the confines of a monitor. To do this, one uses an additional monitor call, *THREADmonitorwaitevent*:

```

int
THREADmonitorwaitevent(monitor, condition, limit, rmask, wmask, xmask,
timeout)
THREAD_MONITOR monitor;
int condition;
int limit;
fd_set *rmask;
fd_set *wmask;
fd_set *xmask;
struct timeval *timeout;

```

(*fd\_set* is a 4.3-defined *typedef* for a bit vector long enough to represent the files of interest. On 4.3 and Ultrix machines, it is up to 64 bits long (or longer, if the system has been configured for more open files per user); on an Encore Multimax, it is set for a maximum of 31 open files.) Called from within monitor *monitor*, the calling thread is suspended until condition *condition* is signaled and the thread is the first one in line for the signal. The condition will be signaled either if explicitly signaled by some other thread on exiting the monitor, or if any of the events occur which are described by the last five arguments of the call (these have essentially the same meaning as the arguments of the UNIX *select* system call). The call returns -1 if

there was an error. It returns 0 if *timeout* points to a value which has expired and none of the events described by the various masks have occurred. Otherwise it returns a value which is the sum of the number of events that have occurred plus either  $2^{*}30$  if a thread has signaled the condition or 0 if not. (N.B.: as with the other monitor calls, this must be used from within a monitor, i.e., between calls to *THREADmonitorentry* and either *THREADmonitorexit* or *THREADmonitorsignalandexit*.)

In the following example, two ‘‘worker’’ threads are created to do I/O. Each calls a monitor which checks to see if reading from *stdin* or writing to *stdout* is possible, and if so, does it. If neither is possible after 10 seconds, it returns with an error indication.

```
#include <thread.h>
#include <stdio.h>
#include <time.h>

main(argc, argv)
int argc;
char *argv[];
{
    extern void startup();

    if (argc != 2) {
        fprintf(stderr, "usage: tst #processors\n");
        exit(1);
    }

    THREADgo(atol(argv[1]), 1024*1024, startup, 0, 0, 20*1024, 2);
}

THREAD_MONITOR iomonitor;

void
startup()
{
    extern void worker();

    iomonitor = THREADmonitorinit(1, NULL);

    THREADcreate(worker, 0, 0, 0, 20*1024, 2);
    THREADcreate(worker, 0, 0, 0, 20*1024, 2);
}

void
worker()
{
    int retcode;
    char ibuf[80];
    int ilen;
    char obuf[80];
    int olen;
```

```

for (;;) {
    sprintf(obuf, "... a message ...\n");

    ilen = 80;
    olen = strlen(obuf);
    retcode = iomon(ibuf, &ilen, obuf, &olen);

    if (retcode <= 0) {
        fprintf(stderr, "timed out\n");
        continue;
    }

    if (ilen > 0) {
        /* do something with the input */
    }

    if (olen > 0) {
        /* do something with the knowledge that a message was outputted */
    }
}

int
iomon(ibuf, ibuflen, obuf, obuflen)
char ibuf[];
int *ibuflen;
char obuf[];
int *obuflen;
{
    THREAD_MANAGER_BLOCK manager;
    fd_set ivec;
    fd_set ovec;
    int code;
    struct timeval t;

    t.tv_sec = 10; /* set a time value of 10 seconds */
    t.tv_usec = 0;

    FD_ZERO(&ivec); /* this macro clears a vector */
    FD_ZERO(&ovec);

    FD_SET(0, &ivec); /* this macro sets the bit corresponding to fd 0 */
    FD_SET(1, &ovec);

    THREADmonitoreentry(iomonitor, &manager);

    /* wait for no more than 10 seconds for I/O to be doable */
    code = THREADmonitorwaitevent(iomonitor, 0, 2, &ivec, &ovec, NULL, &t);

    if (code <= 0) {
        /* an error or timer expiration occurred */
        THREADmonitorexit(iomonitor);
        return(code);
    }

    if (FD_ISSET(0, &ivec)) {
        /* reading from f.d. 0 is possible */

```

```

    *ibufflen = read(0, ibuf, *ibufflen);
} else
    *ibufflen = -2;

if (FD_ISSET(1, &ovec)) {
    /* writing to f.d. 1 is possible */
    *obufen = write(1, obuf, *obufen);
} else
    *obufen = -2;

THREADmonitorexit(iomonitor);
return(code);
}

```

## 7. Bugs, Features, etc.

For the most efficient implementation of I/O, Threads sets all file descriptors to be *nonblocking*. This can cause problems if *stdin* has been set this way and Threads crashes – *stdin* often remains set to non-blocking and the user’s shell will exit after making twenty attempts to read from *stdin*, each one returning the error code *EWOULDBLOCK*. This can be annoying, so the call *THREADsetnonblockinio()* is provided. If this is called before *THREADgo* is called, then Threads won’t use nonblocking I/O, but instead will use a slower, but safer technique.

When a thread performs an I/O operation, the thread blocks, but not the underlying UNIX process. The best implementation of this makes use of the SIGIO signal. This facility did not work in 4.2BSD and, unfortunately, does not work in many systems derived from 4.2 (in particular, it does not work in SunOs, up through at least 3.4 and Ultrix up through at least 2.0; SIGIO does work in 4.3BSD and on UMAX 4.2 on the Encore Multimax). For those systems in which SIGIO does not work, Threads is compiled with the “-DNOSIGIO” option and uses a timer-based polling strategy for I/O, which is slower and less responsive than the SIGIO-based strategy. Even in those systems in which SIGIO does seem to work, it still doesn’t work for output to terminals (due to a “standard” kernel bug), so all terminal-oriented output is handled by timer-based polling. There is still another bug in Berkeley UNIX (related to the problem with nonblocking I/O mentioned above), in which, when one uses SIGIO on terminals, one is forced to set the process group of the terminal to include the process which is to receive the SIGIO signal; this means that SIGIO cannot be used by background processes which use the terminal. Consequently, with the current release of Threads, do not run Threads in the background unless *stdin*, *stdout* and *stderr* are redirected to something other than the terminal (this will be fixed in Threads soon, though the “best” fix would be in the Berkeley

kernel).

Not all of the C library has been modified to be “thread-safe.” The standard I/O library has been made thread-safe, but other routines have not. For example, *gethostbyname* is not thread safe because it returns a pointer to storage that has been statically allocated inside of itself. If two threads attempt to use it at roughly the same time, they will interfere with each other. The routine *sleep* is not safe to be used with Threads because its implementation uses the SIGALRM signal, which is used by the Threads runtime code for different purposes. If it is desired that a thread sleep for a specified interval, use the *select* routine (specifying null vectors for file descriptors). *Select* is a system call, but it is “caught” by the Threads library and made to operate only on the calling thread.

The library and include file for threads is located in the directory \$THREADSDIR/\$MACHINE in the files thread.a and thread.h, where \$THREADSDIR is directory into which the threads distribution has been read (/pro/threads at Brown) and \$MACHINE is the machine type (e.g. multimax, sun3, sun4, vax) Any C program which uses threads should include \$THREADSDIR/thread.h. A typical command line for invoking the C compiler and loader at Brown on a vax would look like:

```
cc -I/pro/threads yourprogram.c /pro/threads/vax/thread.a -o yourprogram
```

(On the Encore Multimax, it is necessary to append “-lpp” to the end of this line).