Concurrent Programming with TreadMarks™

This documentation pertains to versions 0.9.8 and 0.10.1 of the TreadMarks™software package.

TreadMarks™is a software *distributed shared memory* (DSM) system that enables shared-memory concurrent programs to execute on a network of ordinary workstations. It runs over Ethernet, FDDI, and ATM networks of

- DEC Alpha-based workstations under DEC Unix version 3.2C,
- HP PA RISC-based workstations under HP-UX 9.X,
- IBM RS/6000-based workstations, including the SP2, under AIX version 3.2.5 and 4.1.4,
- Intel x86-based PCs under FreeBSD 2.1 and Linux 1.2.13,
- MIPS R3000-based DECstation-5000s under Ultrix version 4.3,
- MIPS R8000-based SGI servers under IRIX 6.1, and
- Sun SPARC-based workstations under SunOS 4.1.3_U1 and Solaris 2.5.

This manual explains first how to write TreadMarks™programs, and then how to compile, link, run and debug them.

Contents

1 Concurrent Programming **3**
1.1 von Neumann's Curse . . . 3
1.2 Practical concurrent programming . . . 3
1.3 TreadMarks™ . . . 4

2 Programming with TreadMarks™ **4**
2.1 Program startup and termination . . . 6
2.2 Memory allocation and data distribution . . . 7
2.3 Synchronization . . . 9

3 Understanding TreadMarks™performance **11**
3.1 Consistency models . . . 11
3.2 Granularity . . . 12

4 Compiling, Linking, and Running a TreadMarks™Program **13**
4.1 Initialization Files . . . 13
4.2 Compiling and Linking . . . 14
4.3 Running a Program . . . 15

5 Debugging a TreadMarks™Program **15**
5.1 Using a Debugger . . . 15
5.2 Using `printf` . . . 17

6 Troubleshooting Guide **17**

1 Concurrent Programming

1.1 von Neumann's Curse

In the traditional, von Neumann computer model, events happen sequentially, one after the other. A computer programmer describes the events that will achieve the goal of the program, and provides an ordering for those events. Sometimes, a pair of events can happen in either order, or even concurrently, but the von Neumann model requires an ordered sequence, so the programmer must provide one.

For decades, the von Neumann model accurately described computers. They executed one instruction at a time. More recently, though, the fastest processors have provided only the appearance of a von Neumann machine. Technological limits have made it progressively more difficult to get one computer processor to execute instructions, in sequence, at significantly faster rates. Therefore, modern superscalar processors execute several instructions concurrently, but only when it is obvious that doing so produces the same results as would strictly sequential execution of the instructions. These superscalar processors are ultimately limited by the fact that automatic detection of fine-grained concurrency, where a handful of simple tasks can happen at once, is difficult.

The most powerful computers of today are composed of many processing units, working as one. Now that fast microprocessors are reasonably cheap and abundant, it makes sense to build powerful, multiprocessor computers by assembling many such microprocessors. To effectively use such multiple processor computers, programs have to offer many opportunities for concurrency. Automatic detection of coarse-grained concurrency, in which complex, multistep procedures can be executed at once, is practically impossible. The programmer must help, by avoiding sequential constraints where they are not necessary, and by pointing out, explicitly, opportunities for concurrency where they arise.

1.2 Practical concurrent programming

Although there are many theoretical models of concurrent programming, most make unrealistic assumptions about hardware that render them essentially unimplementable. Two practical methods for concurrent programming have arisen, based on two different multiple processor computer designs. In shared-memory multiprocessors, each processor can read, or write, any element in a common memory. Each processor has a large cache to reduce the amount of traffic between processors and common memory. In distributed-memory multiprocessors, each processor has its own memory, and data in the memory of one processor is inaccessible to other processors. Where processes must share data, they explicitly send and receive messages that contain data.

Two aspects, not present in sequential programming, complicate distributed-memory concurrent programming. One is data movement. In the von Neumann model, every bit of data is always accessible. In distributed-memory computing, data sought by one processor, but available only on another, must be explicitly sent by the one that has it to the one that needs it, and explicitly received by that one. In scientific applications, it is common for the data that one processor requires from another to be scattered in nonconsecutive memory locations, requiring that it be gathered into a contiguous region of memory before it is sent, and scattered back out into nonconsecutive memory locations when it is received. Though some distributed memory programming systems offer tools to simplify the packing and unpacking of data, the work of data management stil occupies a considerable part of the effort involved in making a sequential program run concurrently.

The other aspect that distinguishes distributed-memory concurrent programming from sequential computing is synchronization. In the von Neumann model, where instructions are executed one after another, synchronization is abundant. After an instruction is complete, the next instruction

can rely upon the work of the previous instruction having been done. In concurrent programming, each processor in a multiple processor computer executes its instructions sequentially, but instructions on different processors do not generally occur in a fixed order. With several instructions in progress at once on different processors, and when the result of one computation is explicitly required before another computation on another processor can begin, synchronization is required to control the order of the computations. Where one processor is to use a value before a second processor modifies it, synchronization is required again. Without synchronization, the value read by the first processor might be the correct one, or it might be the one written by the second processor. The possibility of computing two different results, depending on relative speeds of two different processors, is called a data race. A program with a data race is not robust, and synchronization is required to eliminate the data race.

Producing a correct program for a shared-memory multiprocessor requires the same concern about synchronization as does distributed-memory multiprocessor programming, but no concern about data movement. Since there is only one, common memory, there is no need to move data to make it accessible to another processor. Because shared-memory programming requires only that the programmer consider synchronization, shared-memory programming is considered easier than distributed memory programming, and the changes required to transform a sequential program into a concurrent one are less dramatic.

1.3 TreadMarks™

For all their advantages, shared-memory multiprocessor computers have the disadvantage that they must be carefully designed and assembled, and are therefore expensive and uncommon. Distributed-memory multiprocessors can be assembled from ordinary computers on a high-speed network. Software interfaces like PVM and MPI make it possible to send and receive messages along the network, from one processor to another, and thus to consider a set of engineering workstations to be a distributed-memory multiprocessor computer. For this reason, distributed-memory multiprocessing has become popular, despite its complexity.

TreadMarks™is software that allows shared-memory concurrent programming on a distributed-memory multiprocessor computer, or on a network of workstations. When "shared" memory is accessed on one processor, TreadMarks™determines whether the data is present at that processor, and if necessary transmits the data to that processor without programmer intervention. When shared memory is modified on one processor, TreadMarks™ensures that other processors will be notified of the change, so that they will not use obsolete data values. This notification is neither immediate nor global; immediate notification would be required too often and global notification would usually provide processors with information that was not relevant to the majority of them. Instead, this notification takes place between two processors when they are synchronized. Programs free of data races produce the same results whether notification is immediate, or delayed until synchronization, and the accumulation of many such notifications into a single message at synchronization time greatly reduces the overhead of interprocessor communications.

2 Programming with TreadMarks™

This section introduces TreadMarks™with an example of a simple C program, "app", that stores elements in a global array, prints it, and computes and prints the sum of its elements. The program exhibits all of the synchronization primitives available in TreadMarks™, and illustrates the code that typically starts up a TreadMarks™program. It can serve as a template for general TreadMarks™programs. Explanations of the TreadMarks™function calls in the program follow.

```
/* program app */
#include <stdio.h>
#include "Tmk.h"

struct shared {
  int sum;
  int turn;
  int* array;
} *shared;

main(int argc, char **argv)
{
    int start, end, i, p;
    int arrayDim = 100;

    /* Read array size from command line */ {
        int c;
        extern char* optarg;
        while ((c = getopt(argc, argv, "d:")) != -1)
            switch (c) {
            case 'd':
                arrayDim = atoi(optarg);
                break;
            }
    }

    Tmk_startup(argc, argv);

    if (Tmk_proc_id == 0) {
        shared = (struct shared *) Tmk_malloc(sizeof(shared));
        if (shared == NULL)
            Tmk_exit(-1);
        /* share common pointer with all procs */
        Tmk_distribute(&shared, sizeof(shared));

        shared->array = (int *) Tmk_malloc(arrayDim * sizeof(int));
        if (shared->array == NULL)
            Tmk_exit(-1);
        shared->turn = 0;
        shared->sum = 0;
    }

    Tmk_barrier(0);

    /* Determine array range for each processor */ {
        int id0 = Tmk_proc_id, id1 = Tmk_proc_id+1;
        int perProc = arrayDim / Tmk_nprocs;
        int leftOver = arrayDim % Tmk_nprocs;
        start = id0 * perProc + id0 * leftOver / Tmk_nprocs;
        end   = id1 * perProc + id1 * leftOver / Tmk_nprocs;
    }

    for (i = start; i < end; i++)
        shared->array[i] = i;
```

```
    Tmk_barrier(0);

    /* Print array elements, in the natural output order */
    for (p = 0; p < Tmk_nprocs; p++) {
        if (shared->turn == Tmk_proc_id) {
            for (i = start; i < end; i++)
                printf("%d: %d\n", i, shared->array[i]);
            shared->turn++;
        }
        Tmk_barrier(0);
    }

    /* Compute local sum, then add to global sum */ {
        int mySum = 0;
        for (i = start; i < end; i++)
            mySum += shared->array[i];

        Tmk_lock_acquire(0);
        shared->sum += mySum;
        Tmk_lock_release(0);
    }

    if (Tmk_proc_id == 0) {
        Tmk_free(shared->array);
        Tmk_free(shared);
        printf("Sum is %d\n", shared->sum);
    }

    Tmk_exit(0);
}
```

2.1 Program startup and termination

Removing most of the details of what the program actually does leaves this shell:

```
#include "Tmk.h"
/* ... */
main(int argc, char **argv)
{
    /* ... */
    /* Read from command line */ {
        int c;
        extern char* optarg;
        while ((c = getopt(argc, argv, /* ... */)) != -1)
            switch (c) {
            /* ... */
            }
    }

    Tmk_startup(argc, argv);
    /* ... */
    Tmk_exit(0);
}
```

The header file "Tmk.h" must be included in any file that calls the TreadMarks™software library. It provides function prototypes for the various TreadMarks™functions, and defines important constants and global variables as well.

The invocation of the function `Tmk_startup(argc, argv)` begins the concurrent phase of the computation. Before that statement, there is one process running the program. After that statement, there are several. They begin with identical values in all local and global variables except one, `Tmk_proc_id`, which holds a distinct nonnegative value less than the number of processes and serves to identify each process.

The arguments to `Tmk_startup` are the arguments to `main` that represent the command line arguments. The call to `Tmk_startup` should follow the decoding of other command line options, since evaluating the arguments concurrently would serve little purpose. `Tmk_startup` looks for command line arguments that follow a double-dash "`--`", checking for several arguments that affect how the concurrent program is executed.

After the program invocation

```
<application> [<app arguments>]... -- [<TreadMarks arguments>]...
```

TreadMarks™looks for the following command line arguments:

-f *file name*. Identifies the machine list file, which contains a list of machines one per line. Defaults to `.Tmkrc`.

-h *machine name*. One or more instances of this argument define the machine list. If this argument is not used, the machine list is defined by the contents of the machine list file. The machine list identifies the machines that comprise the multiprocessor computer. The ith machine listed runs with `Tmk_proc_id == i-1`.

-n *number*. Number of processors from the machine list that comprise the concurrent machine. By default, it is the number of machines in the machine list. In the program, this quantity is available in the global variable `Tmk_nprocs`. In no case may it exceed the value of the global constant `TMK_NPROCS`.

-r Use `rexec` rather than `rsh` to start remote processes. Default: off.

-s Display run-time statistics before exiting. Default: off.

-x Open a separate X window attached to each of the remote processes. Anything a process writes to `stdout` or `stderr` or reads from `stdin` is displayed to or read through its window. Default off.

The invocation of the function `Tmk_exit(int)` terminates the execution of a single concurrent execution. Program text following the call to `Tmk_exit` executes only on one processor. The argument to `Tmk_exit` is zero to indicate successful termination and nonzero to indicate an error. There is no particular assignment of nonzero values to kinds of errors; the programmer is responsible for deciding how to associate errors and error codes.

2.2 Memory allocation and data distribution

This segment of the program, run as soon as the concurrent phase of the program begins, allocates shared memory, and ensures that all processes can access it. Later, that shared memory is released.

```
/* ... */
struct shared {
  int sum;
  int turn;
  int* array;
} *shared;

main(int argc, char **argv)
{
    /* ... */

    if (Tmk_proc_id == 0) {
        shared = (struct shared *) Tmk_malloc(sizeof(shared));
        if (shared == NULL)
            Tmk_exit(-1);
        /* share common pointer with all procs */
        Tmk_distribute(&shared, sizeof(shared));

        shared->array = (int *) Tmk_malloc(arrayDim * sizeof(int));
        if (shared->array == NULL)
            Tmk_exit(-1);
        shared->turn = 0;
        shared->sum = 0;
    }

    /* ... */

    if (Tmk_proc_id == 0) {
        Tmk_free(shared->array);
        Tmk_free(shared);
        /* ... */
    }
    /* ... */
}
```

The function `Tmk_malloc(unsigned)`, like the function `malloc` from the standard C library, allocates memory dynamically. However, memory allocated with `Tmk_malloc` is automatically shared, so that changes made to shared memory by one process are eventually visible to other processes. The function `Tmk_free(char*)`, like the function `free`, releases dynamically allocated memory. A function not shown here, `Tmk_sbrk(unsigned)`, can be used for lower level memory management, and is similar to the standard C library function `sbrk`. The function `Tmk_sbrk` is provided so that carefully tailored shared memory managers can be implemented; it is rarely appropriate to use it in conjunction with `Tmk_malloc`.

The function `Tmk_distribute(char* ptr, unsigned size)` is used to explicitly share private memory between processes. When invoked by a single process, it causes values in private memory on that processor to be replicated in corresponding memory locations in all other processes. Usually, the private memory is a global variable common to all the processes; automatic variables could appear in different places on different machines, so the result of a distribution could be surprising. `Tmk_distribute` is generally used to solve the bootstrapping problem; when one process allocates shared memory, how do the other processes know where to find it? As in this example, a call to `Tmk_distribute` is the most common way to share that information.

The global structure called `shared` is not a feature of TreadMarks™, but it is a common idiom

in TreadMarks™programming. To avoid several explicit memory allocation calls, global shared variables are packed into a single structure, and a global instance of that structure is allocated at once. This idiom has the additional benefit that references to shared memory are easy to find in the program, so potential data races are indicated by occurrences of the word "shared" where no synchronization is apparent.

2.3 Synchronization

The part of the program that does the work has three parts, each with different synchronization requirements.

```
/* ... */
main(int argc, char **argv)
{
    /* ... */
    Tmk_barrier(0);

    /* Determine array range for each processor */ {
        int id0 = Tmk_proc_id, id1 = Tmk_proc_id+1;
        int perProc = arrayDim / Tmk_nprocs;
        int leftOver = arrayDim % Tmk_nprocs;
        start = id0 * perProc + id0 * leftOver / Tmk_nprocs;
        end   = id1 * perProc + id1 * leftOver / Tmk_nprocs;
    }

    for (i = start; i < end; i++)
        shared->array[i] = i;
    Tmk_barrier(0);

    /* Print array elements, in the natural output order */
    for (p = 0; p < Tmk_nprocs; p++) {
        if (shared->turn == Tmk_proc_id) {
            for (i = start; i < end; i++)
                printf("%d: %d\n", i, shared->array[i]);
            shared->turn++;
        }
        Tmk_barrier(0);
    }

    /* Compute local sum, then add to global sum */ {
        /* ... */
        Tmk_lock_acquire(0);
        shared->sum += mySum;
        Tmk_lock_release(0);
    }
    /* ... */
}
```

A *barrier* is a synchronization device that requires all processes to wait for the last of them to "catch up". The function `Tmk_barrier(unsigned b)` in TreadMarks™causes all processes to pause until all of them have invoked `Tmk_barrier` with the same barrier b as argument. The argument identifies the barrier; TreadMarks™allows a fixed number of barriers in the program, numbered from 0 to `TMK_NBARRIERS-1`. In a correct program, one barrier is sufficient; multiple barriers exist to make the program clearer. In a program fragment like this,

```
{/* block A */}
Tmk_barrier(0);
{/* block B */}
Tmk_barrier(0);
```

it is not clear whether or not the program could be running correctly with one process executing block A as another concurrently executes block B. In this alternative program fragment,

```
{/* block A */}
Tmk_barrier(0);
{/* block B */}
Tmk_barrier(1);
```

it is clear that all processes execute block A concurrently, and then all execute block B concurrently. If a programmer mistakenly directed control flow into block B in one process while the other processes executed block A, the code that uses distinct barriers would produce deadlock, as no process could pass beyond its next barrier. For one debugging the program, a deadlock that occurs shortly after the program goes wrong provides a valuable clue that can help track the error down. The same control flow error in a program that uses only one barrier produces less predictable results.

After carefully dividing the array into pieces of approximately equal size, and concurrently initializing the array, the program prints the contents of the array in order. To make the pieces come out in order, the processes must coordinate to take turns printing, by using a barrier and a shared variable `turn`. The value of `turn` seen by each processor for each value of variable `p` is the same, because the barrier at the bottom of the loop synchronizes all processors after every modification of `turn`.

The third phase of the code sums the elements of the array. First, each processor computes the sum of elements of the subarray assigned to it. Then the local sums are gathered into the shared variable `sum` with a *lock* used for synchronization. A lock is a synchronization device that enforces one-process-at-a-time access. The function `Tmk_lock_acquire(unsigned L)` in TreadMarks™causes a process to pause while another process owns lock `L`, and to take ownership of the lock and continue when the lock becomes available. The lock is never owned by more than one process at a time, but the lock may be owned by no processes. The process that gets the lock when several contend for it is chosen arbitrarily. TreadMarks™allows a fixed number of locks in the program, numbered from 0 to `TMK_NLOCKS-1`. The function `Tmk_lock_release(unsigned L)`, when invoked in the process that owns lock `L`, releases the lock so that another process may acquire it.

Locks are used to manage access to shared resources. In this case, the lock controls access to the shared variable `sum`. Were two processors to modify `sum` at once, the results would be unpredictable. Perhaps each would read the same value from `sum`, modify the value locally, and then write back the modified values into `sum`. The one that wrote first would have its contribution to the global sum obliterated by the second, producing an erroneous result.

A section of code that cannot be executed by more than process at a time is called a *critical section*. Locks can be used to enforce critical sections, with one lock being used for each critical section. More generally, locks can protect critical resources, like the variable `sum`; were `sum` examined or modified at another point in the program the same lock would be used to protect it from concurrent modification.

Locks and barriers are the only synchronization primitives provided by TreadMarks™. They are sufficient. Synchronization primitives offered in other concurrent programming systems, like gates, monitors, and condition variables, can be readily imitated with just these two, plus shared variables.

TreadMarks™provides no other functions than the ones mentioned in this section. No others are necessary to produce correct concurrent programs.

3 Understanding TreadMarks™performance

3.1 Consistency models

The TreadMarks™interface is built atop a set of message passing primitives, as are used in distributed-memory concurrent programming and UNIX™ network programming. Although the TreadMarks™programmer need not concern himself with the details of this implementation to produce correct concurrent programs, some understanding of the implementation is useful to the programmer who wants to develop correct programs that run quickly. Since improving program performance is the reason for using concurrent machines and programs in the first place, most programmers should know about these implementation issues.

If a shared-memory concurrent computer were simulated by a single processor dividing time between several processes, memory accesses would be *sequentially consistent*; that is, an access would happen at some time as measured by a common system clock. Consider this code fragment:

```
if (Tmk_proc_id == 0)
    shared->flag = 1;
else
    while (shared->flag != 1)
    { /* do nothing */ }
Tmk_exit(0);
```

Assume that the initial value of the shared variable `flag` is zero. If the concurrent programming model is sequentially consistent, every process will eventually reach the call to `Tmk_exit`. At some time, processor 0 will set `flag` to 1. Other processors check `flag` at monotonically increasing times, and each time is at least one unit greater than the previous one. Consequently, every process will eventually finish.

The code fragment above does not provide for synchronization between accesses and modifications to `flag`, so it has a data race. This data race makes the program behavior nondeterministic, because the number of times each processor but the first tests `flag` cannot be predicted. However, with sequentially consistent concurrency, the program will terminate.

An implementation of TreadMarks™that provided sequential consistency would require that every modification of data in shared memory be reported to all processes with local copies of that data. Consequently, in the program "app" of section 2, every time an array element is initialized, TreadMarks™would have to transmit information to all processes.

The performance of message passing procedures is measured in terms of two quantities: *latency* and *bandwidth*. Latency is the time between the beginning of the transmission of a message and the beginning of its reception. Bandwidth is the rate of data transmission once the data has begun flowing. Typically, latency is large when compared to the time it takes to execute a computer instruction; consequently, the time spent sending many small messages is much larger than the time spent sending the same data in one large message. The implementation of sequential consistency requires the sending of many small messages, so a high-performance implementation of sequentially consistent concurrency is unlikely.

TreadMarks™implements a different kind of consistency, called *release consistency*. With release consistency, there is no common global clock. Temporal relationships between operations in different processes are defined only when they are separated by a synchronization. The looping

example above would not terminate under a release consistency model, because no synchronization follows the modification of `flag` in the first process, nor precedes the examination of `flag` in the other processes. The other processes would never see the new value of `flag`.

Release consistency allows the notification of changes to shared memory to be deferred until the time of synchronization. If a barrier is used to synchronize all processes, then all processes become aware of all recent changes to shared memory that affect their local copies, and these notifications are joined with the messages that must be exchanged to achieve synchronization anyway. Therefore, less time is spent communicating, because fewer messages means less time lost to latency. If a lock is used to synchronize access to shared data, or to a critical code section, the process that releases the lock notifies the process that acquires the lock of all known changes to shared memory. A process knows about the changes that it is responsible for, and about the changes that it learned about in previous synchronizations.

There are two chances to pass information from the lock releaser to the lock acquirer: when the lock is released, and when it is acquired. *Eager* release consistency requires that the releaser of the lock notify all processes of the change to shared memory, since at release time, the next acquirer cannot be identified. The release consistency method used in TreadMarks™is called *lazy* release consistency. With lazy consistency, the acquirer of the lock learns of changes to shared memory only when it receives the lock from the releaser, and no other processes are bothered with the information. There are also two ways to notify a process of changes to shared memory. An *update* strategy sends the modified data with the notification. An *invalidate* strategy, as used in TreadMarks™, sends only news of a change, but not the content of the change. When a process actually attempts to access the modified data, the changes are retrieved; this avoids the retrieval of information that might never be used. Although neither consistency method, and neither update method, is always better than the other, experiments have shown that lazy consistency with an invalidate strategy produces faster running programs than do other choices, in a majority of the cases tested.

3.2 Granularity

At synchronization time, TreadMarks™does not transmit messages specifying which shared bits or bytes have changed; TreadMarks™sends information about changes at the *page* level, where a page is the least amount of data that a virtual memory system can store and retrieve in an operation. TreadMarks™works with the virtual memory system of each computer to invalidate modified copies of shared memory pages. When a shared memory access attempts to examine an invalidated page, the consequence is a page fault, just as if the access were to a remote part of private memory that had not been accessed and "paged in" for a long time. When the virtual memory system encounters a page fault, it attempts to retrieve the missing page from swap space, usually a fast disk storage device. In TreadMarks™, when the page fault is the result of an access to invalidated shared memory, the virtual memory system cannot resolve the reference, so it signals that a segmentation fault has occurred. Only then is TreadMarks™software invoked, to produce an up-to-date copy of the shared page. By using the virtual memory system in this way, TreadMarks™does not impose overhead on ordinary private memory accesses, or on ordinary shared memory accesses. TreadMarks™takes over only when the virtual memory system asks it to.

One consequence of this implementation decision is that memory accesses cost less when they involve few pages. This is not a surprise; the same considerations arise in sequential programming in a virtual memory system. Where a program is designed with memory hierarchy in mind, the global variable `Tmk_page_size` gives the size of one page, in bytes.

It would be most inconvenient if two processes could not concurrently modify data on the same page. The program "app" of section 2, for example, would not be correct unless the size of the shared array were a multiple of the product of the page size and the number of processes. Fortunately, TreadMarks™allows concurrent access of distinct computer *words*, whether they are in the same page or not. A word is generally the unit of storage necessary to hold one integer or floating point number, so the program of section 2 is correct. When TreadMarks™is called upon to update a shared memory page, it does so not by retrieving a copy of the page, but by retrieving a list of changes that must be made to the page to bring it up to date. Often, the list of changes is much smaller than would be a copy, and so message traffic is reduced. The changes are represented by a run-length encoding, in which each range of consecutive modified words is sent, preceded by its address and length.

A programmer, armed with this knowledge, can reduce memory traffic by making sure to modify a few, large contiguous blocks, as opposed to many small blocks. If every other word in a page were to be modified, the size of the change list would be larger than the size of a page. Since such a pattern of data modification is rare in practice, the run-length encoding of differences reduces communication in practice.

4 Compiling, Linking, and Running a TreadMarks™Program

4.1 Initialization Files

Two initialization files may be defined to improve the convenience and performance of using TreadMarks™. If you repeatedly run your TreadMarks™programs on the same set of hosts, you can create a `.Tmkrc` file in your home directory. Each line of this file contains the network name of exactly one computer. If you specify on the command line that N computers should be used for a particular run, TreadMarks™will use the first N hosts named. The first line of the `.Tmkrc` file must be the name of the machine from which you will start TreadMarks™. The `-f` command line option allows you to name another file to be used in place of `.Tmkrc`.

TreadMarks™can use one of two mechanisms to distribute work to remote hosts: `rsh` and `rexec`. The main difference between them is in the mechanism used for authentication; `rsh` can be used among "equivalent", i.e. mutually trusting, machines, while `rexec` requires that you provide an explicit username and password for each of the remote hosts. For more detailed discussion, see the relevant Unix `man` pages and system documentation.

Normally, TreadMarks™uses `rsh`. `Rexec` is used if you use the `-r` command line option or if there is a file named `.netrc` in your home directory. There are two ways that you can provide the required username/password pairs. If there is a `.netrc` file and it is readable only by you, then the system will try to use it to look up the host, username, and password. If there is no `.netrc` file, or if the protections are wrong, or if the host is not listed, the system will prompt you for a username and password for that host.

For more information about the alternatives you can consult the Unix `man` pages. You may also want to discuss the issues with your local system/network administrator.

If you create a `.netrc` file, it should contain an entry for each machine that you intend to use with TreadMarks™[1]. Entries should use fully qualified machine names and be of the form:

```
machine helma.cs.rice.edu login myuserid password mypassword
```

Note that `helma` by itself will not work; the machine name has to be fully qualified. Keep in mind that if you change your password on one or more machines listed in the `.netrc` file, you will have

[1]For further information read "man 5 netrc".

to update those entries in the file. Remember to set the file protection bits so that you are the only one with permission to read. The `.Tmkrc` file normally names a subset of the machines that you included in your `.netrc` file.

Here is what a `.netrc` looks like for user `joe` with password `joes-password`, for access to the machines `aurora.cs.rice.edu`, `medea.cs.rice.edu`, and `helma.cs.rice.edu`.

```
machine aurora.cs.rice.edu login joe password joes-password
machine medea.cs.rice.edu login joe password joes-password
machine helma.cs.rice.edu login joe password joes-password
```

The corresponding `.Tmkrc` looks as follows

```
aurora.cs.rice.edu
medea.cs.rice.edu
helma.cs.rice.edu
```

4.2 Compiling and Linking

TreadMarks™is a run-time library. A program is compiled as usual, but in the linking phase you include the TreadMarks™library. To link with the correct library for your platform, set the `ARCH` flag in the Makefile.

While it is surely possible to do otherwise, it is convenient to do all compilation and linking through the use of Makefiles. We usually keep applications in an `apps` directory, with directories underneath for each application, and underneath those directories a directory for the source and a directory for the compiled code for each architecture. Given that directory structure, the following Makefiles can be used for the example program that we have used throughout this manual. The first Makefile, called `.common`, would reside in the directory for the application; the second, called `Makefile`, would reside in the directory underneath that one for the SPARC architecture. The `.common` Makefile is shared by all of the architectures. The other architecture flags are `alpha`, `mips`, and `rs6k`. We strongly suggest that the Makefiles in the directories for the sample applications be used as a template. The contents of `.common` are:

```
TmkDIR = ../../..
TmkLIB = $(TmkDIR)/$(ARCH)

CPPFLAGS= -I$(TmkDIR)/include

OBJ = app.o

app: $(OBJ) $(TmkLIB)/libTmk.a
$(CC) $(CFLAGS) -o $@ $(OBJ) -L$(TmkLIB) -lTmk $(LDFLAGS)

app.o: ../src/app.c
$(CC) $(CFLAGS) $(CPPFLAGS) -c ../src/app.c

clean:
rm -f *.o app
```

The contents of `sparc/Makefile` are:

```
ARCH = sparc

CC = gcc
CFLAGS = -g -O
LDFLAGS =

all: app

include ../.common
```

4.3 Running a Program

Both TreadMarks™and the application program accept command-line arguments. The command-line arguments for TreadMarks™are separated from the application's arguments by a double-dash argument, where the possible arguments are listed in section 2. For instance, to run `app` with a problem size of 500 on the four machines named `machine1`, `machine2`, `machine3`, and `machine4`, use

```
app -d 500 -- -h machine1 -h machine2 -h machine3 -h machine4
```

To run the same program on 4 processors, without specifying their names,

```
app -d 500 -- -n 4
```

This command will run `app` on the first four machines specified in `.Tmkrc`. In either case, the machine you're typing the command on must be first in the command-line list or the `.Tmkrc` file.

5 Debugging a TreadMarks™Program

5.1 Using a Debugger

The following describes one method for debugging TreadMarks™programs. We will use `gdb` in our examples, but any other debugger with the capability to attach to a running process could be utilized.[2] Conceptually there can be a debugger process running on each processor that controls the behavior of the TreadMarks™process on that processor only. Debugger commands such as continue, break, list, etc. apply only to the debugger process within which they are issued. For example, you can require processor 1 to stop every time it reaches line 718, while processor 0 flies right through. The only interactions between the debugger processes occur at startup and at your synchronization points (startup is really just a special kind of automatic synchronization). When a debugged process is stuck at a synchronization point, you cannot force it to advance. The other process(es) that it is waiting for must advance first.

To debug, make sure to compile and link all your code with the -g flag, just as for sequential C code.

In the directory where you will be working, you should create a file called `.gdbinit`; note the initial dot, just as in `.netrc` and `.Tmkrc`.[3] That file contains commands that will be executed whenever you start up the `gdb`. The one command required by TreadMarks™to be in this file is

[2]Unfortunately, none of the debuggers available on Ultrix, including `gdb`, have this capability. This is a limitation of Ultrix, and not `gdb`. For instance, `gdb` running on SunOS can attach to a running process.

[3]See the `gdb` documentation for more details on the `.gdbinit` file.

```
handle 11 nostop noprint
```

This means that `gdb` should not stop at *segmentation faults* (the UNIX abbreviation for a segmentation fault is SIGSEGV, which is the name of the signal that gets sent to your process when a segmentation fault occurs; it has number 11). The reason for this is that unlike regular sequential programs, a segmentation fault in a TreadMarks™execution is not necessarily catastrophic. When you debug a sequential program, the debugger will stop when a segmentation fault occurs to allow you to find the error; the above handle command tells the debugger not to stop. The reason you shouldn't stop at a segmentation fault is that TreadMarks™uses the SIGSEGV signal to retrieve shared memory which is not resident on the processor. Such retrievals are fairly frequent, and you do not want to stop for every one. One difficulty is that if you get a bona fide segmentation fault your execution will crash and you will not get a chance to catch it. Note that outside `gdb`, a *bona fide* segmentation fault causes TreadMarks™to crash and dump a core file (just like when a sequential program has a segmentation fault); however, inside `gdb`, a bona fide segmentation fault will not dump a usable core file.

Suppose you are running on the Ethernet and your program is called program.ether. Suppose you are running on 2 processors (this procedure can be easily generalized to more processors). Suppose the processors are called A and B, with A having the role as processor 0 and B as processor 1. Set up your screen so you have 3 windows, on 1 you should rlogin to processor A (if not already there) and on 2 you should rlogin to processor B (if not already there). You will need one window on each processor to debug and the second processor B window to get status information.

If you are using X you get output to appear properly in the debugger as follows. First make sure that your execution path is in your .cshrc file (or whatever file that corresponds to for your shell). Then, make sure that the program `xterm` is included in one these directories. In the windows where you will be running the debugger, make sure that your `DISPLAY` environment variable is correctly set to your display. If not, under `csh` use the command:

```
setenv DISPLAY <workstation name>:0.0
```

where workstation name is the name of the workstation on your desk, not the processor you will be using in that window.

You only need to do this before your first debugging session, per login. On the console window of your X session run the command:

```
xhost +<machine1> +<machine2> ...
```

including each of the machines that you intend to use. This command enables the specified machines to open windows on your display. Again, this needs to be done only once per login session.

In the status window for processor B, you will obtain the Unix process id for the process on that machine. For example, you can use the commands `ps` or `top`. `top` shows the processes that are getting the most CPU time on processor B. When the TreadMarks™process on processor B starts up you will need to watch the top window carefully to get its process number. In the other two windows make sure you are in the directory where program.ether resides. In each of those two windows, execute the command:

```
gdb program.udp
```

This will start up a `gdb` session. In your processor A window you should start up the program by executing in the debugger the command:

```
run -- -n 2 -x
```

Alternatively you can specify the processors to use via "-h". Note the extra x flag at the end. This is used from within `gdb` only if you are debugging from within an X session. The x flag can also be used outside `gdb` when running from an X session. It enables processors other than number 0 to print out values to the screen through their own X windows.

Now watch the window where you are running top carefully as the program.ether process starts up on processor B. It will have a process id (pid) number between 1 and 30000. Suppose that number is 12940. Then in your Processor B `gdb` window, execute the command:

```
attach 12940
```

This will cause that `gdb` process to take on your TreadMarks™process and allow you to debug it. Usually the process on processor B will stop at a TreadMarks™synchronization point just after you attach. You need to tell it to continue execution by giving the command:

```
continue (or just c)
```

in your Processor B `gdb` window.

From here debugging works essentially as for sequential programs. You can set breakpoints, look at values, set values, etc. Sometimes one of the two processors will appear not to be doing anything. This usually means that it is either waiting at a synchronization point or waiting for updated shared memory values.

5.2 Using `printf`

You can also report status and/or debugging information from each process using (f)printf or other stdio functions. Ordinarily, `stdin`, `stdout`, and `stderr` for remote processes are redirected to `/dev/null`. However, there are two ways to direct these streams to useful places: using the "-x" command-line argument displays the stream in an X window and `freopen`'ing `stdout` or `stderr` writes the stream to a log file. When using a log file, be careful that the processes write to physically distinct files. There are two common ways to achieve this. If the files are stored in a shared (NFS) file system, each file is `freopen`'ed with a unique name, e.g., by appending the `Tmk_proc_id` to a file name. Alternatively, the log files are written to a private file system, e.g., the `/tmp` directory. The `freopen` should be performed immediately after `Tmk_startup` returns.

6 Troubleshooting Guide

- **Data granularity -** TreadMarks™detects modifications to shared memory at a four-byte granularity. This means that if you share memory at the character or short level, you may not get correct results. A simple way to avoid problems is to be sure that you don't declare any `char` or `short` variables in shared memory.

- **`read` and `write` calls -** Calling `read` and `write` with shared memory address is *not* guaranteed to work. However, stdio operations such as `fread` and `fwrite` will work. Both of `read` and `write` check the protection level of the specified buffer address and return errors if the required operation cannot be completed. TreadMarks™uses changes in page protection to implement the DSM protocols, and therefore pages in the shared memory address range are often unreadable during the computation.

- **Debugging -** TreadMarks™uses the `SIGSEGV` signal (signal 11) to detect accesses to shared memory. If you use a debugger, therefore, you need to disable the debugger's monitoring

of that signal. In `gdb`, for instance, you should execute "handle 11 nostop noprint" before running your program.

- **Remote processes fail to start -** there are a few common causes. (1) If you're using the "-x" parameter, make sure that your display environment variable is correctly set and that the program `xterm` is on the path in your `.cshrc`. (Your `.login` file won't be executed when a remote process is started, so the path must be in your `.cshrc`.) (2) The remote machine has insufficient swap space to run a TreadMarks™program. Because of preallocation of memory, the minimal TreadMarks™program requires approximately 40 Mbytes to start. The program `pstat` on Ultrix and SunOS reports the amount of available swap space. (3) Is the first machine in your command-line list or `.Tmkrc` file the same as the one you're starting the program on. (n) Most other problems produce a reasonably descriptive error message, e.g., when your `.netrc` file doesn't contain the correct password.

- "<mmap>Tmk_page_initialize: can't allocate the shared memory" **-** You don't have enough available swap space on this machine.