TreadMarks^{TM} Documentation

This documentation pertains to version 0.7 of the TreadMarks^{TM} software package.

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

- SUN SPARCstations under SunOS version 4.1.x,
- DECstation-5000s based on the MIPS R3000 processor under Ultrix version 4.3,
- DEC workstations based on the Alpha processor under OSF/1 version 1.3, and
- IBM RS/6000 workstations under AIX version 3.2.

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

Contents

1 Introduction

1.1 Shared-Memory Parallel Programming

TreadMarks^{TM} is a software package that supports parallel programming on a network of ordinary workstations. It enables a programmer to create processes on different machines that work together to solve a single problem. There are two fundamental programming models used for parallel computing: shared memory and message passing. In the shared-memory model, every process has direct access to all shared data. In contrast, data sharing in the message-passing model depends on each process sending messages to the other processes; the primitives used to send messages are called explicitly by the program. TreadMarks^{TM} allows the programmer to write programs using the shared-memory model and then run those programs on hardware that supports message passing, such as a network of workstations. The interface for TreadMarks^{TM} is modeled after PARMACS, a widely-used parallel programming interface from Argonne National Labs. The motivation underlying TreadMarks^{TM} is twofold.

- Shared-memory programs are easier to write and debug than message-passing programs.
- Networks of workstations are more widely available than shared-memory multiprocessors.

1.2 The Need for Synchronization

Shared memory means that it is possible to use variables that are visible to all the processes, as if the processes had access to the same memory. Shared memory can give rise to *data races.* Data races happen when accesses to shared variables by different processes are interleaved in a way that the programmer did not intend. For instance, suppose one process writes several fields of a shared record, while another one tries to read those same fields. It is quite possible that the second process reads some of the newly written fields but also reads some of the old values before they are overwritten by the first process. This situation is usually undesirable. To avoid it, shared-memory parallel programs contain *synchronization.* In this particular case, a *lock* could be used to guarantee each of the processes gets exclusive access to the record. TreadMarks^{TM} provides locks as well as other synchronization primitives.

1.3 Implementation

TreadMarks^{TM} uses several techniques to implement shared memory efficiently on a network of workstations. To avoid communication at every shared memory access, TreadMarks^{TM} maintains copies of shared memory pages at the machines currently accessing them. The challenge then becomes how to keep these copies consistent without excessive overhead. Among the more important techniques used by TreadMarks^{TM} for doing so are lazy release consistency and multiple-writer protocols. Roughly, these techniques make sure that shared memory is consistent after synchronization operations, such as acquiring a lock. They do not guarantee that shared memory is consistent all of the time. This is what lets them implement shared memory much more efficiently than other methods.

The programmer need not be too concerned with this. If the program does not have data races, it will behave as if it were executing on an ordinary shared-memory system, **on one condition:** All synchronization must be done using the TreadMarks^{TM} supplied primitives, otherwise TreadMarks^{TM} cannot tell when to make shared memory consistent.

2 Programming with TreadMarks^{TM}

This section discusses in more detail how to write a C or C++ program that uses TreadMarks^{TM}. The interface to TreadMarks^{TM} is defined in the `Tmk.h` file which you will find in the `include` directory in the distribution. This file should be included in any TreadMarks^{TM} program. An annotated copy of this file is included in the next subsection. The important interface functions (startup, memory allocation, synchronization, and termination) are explained next. This section includes a simple but complete example of a TreadMarks^{TM} program. In addition, the distribution directory `apps` contains sample programs for SOR (successive over-relaxation) and TSP (traveling salesman problem).

2.1 Interface

Applications running on TreadMarks^{TM} should include the file `Tmk.h`. It defines several useful constants as well as providing external definitions for the routines and variables below.

```
/*
 * the maximum number of parallel processes supported by TreadMarks
 */
#define TMK_NPROCS

/*
 * the actual number of parallel processes after Tmk_startup
 * (Initialized by Tmk_startup.)
 */
extern  unsigned        Tmk_nprocs;

/*
 * the process id, an integer in the range 0 ... Tmk_nprocs - 1
 * (Initialized by Tmk_startup.)
 */
extern  unsigned        Tmk_proc_id;

/*
 * the maximum number of shared memory pages
 */
#define TMK_NPAGES

/*
 * the shared memory page size
 * (Initialized by Tmk_startup.)
 */
extern  unsigned        Tmk_page_size;

/*
 * the number of lock synchronization objects provided by TreadMarks
 */
#define TMK_NLOCKS

/*
 * the number of barrier synchronization objects provided by TreadMarks
 */
#define TMK_NBARRIERS
```

```
/*
 * Initialize TreadMarks and start the remote processes.
 */
void    Tmk_startup(argc, argv)
        int     argc;
        char  **argv;

/*
 * Terminate the calling process.  Other processes are unaffected.
 */
void    Tmk_exit(status)
        int     status;

/*
 * Block the calling process until every other process arrives
 * at the specified barrier.
 */
void    Tmk_barrier(id)
        unsigned id;

/*
 * Block the calling process until it acquires ownership
 * of the specified lock.
 */
void    Tmk_lock_acquire(id)
        unsigned id;

/*
 * Release the specified lock.
 */
void    Tmk_lock_release(id)
        unsigned id;

/*
 * Allocate the specified number of bytes of shared memory, aligning
 * the block on a page boundary if the block's size is at least
 * Tmk_page_size.  Return NULL if the shared memory heap is exhausted.
 */
char   *Tmk_malloc(size)
        unsigned size;

/*
 * Free shared memory allocated by Tmk_malloc.
 */
void    Tmk_free(ptr)
        char   *ptr;

/*
 * Allocate the specified number of bytes of shared memory.  Return
 * NULL if the shared memory heap is exhausted.  N.B.  It is
 * impossible to free a block of memory allocated by Tmk_sbrk.
 */
char   *Tmk_sbrk(incr)
```

```
        int     incr;

/*
 * Distribute (copy) the contents of a block of PRIVATE memory on the
 * calling process to every other process.  After the call completes
 * every process has identical values in the specified memory
 * locations.  Ordinarily used by a program's initialization code to
 * distribute ``root'' pointers to shared data structures.
 */
void    Tmk_distribute(ptr, size)
        char    *ptr;
        unsigned size;
```

2.2 Startup and Termination

The routine `Tmk_startup` initializes the TreadMarks^{TM} library and creates remote processes on other machines. Section 3 describes a variety of ways to specify which machines are used, including command-line arguments. Ordinarily, a program interprets the application specific command-line arguments before passing the remainder of the command-line arguments to `Tmk_startup`. For example, in the sample programs, the application specific command-line arguments are handled using `getopt`[1], and then `argc` and `argv` are passed to `Tmk_startup`. The following code fragment is typical of the sample programs.

```
extern char *optarg;
int c;

while ((c = getopt(argc, argv, "...")) != -1)
  switch (c) {
  ...
  }
Tmk_startup(argc, argv);
```

A program should **never** call any other TreadMarks^{TM} routine before `Tmk_startup`. After `Tmk_startup` both private and shared memory are identical for all processes, with the exception of the global variable `Tmk_proc_id`, which is set to the id of the process.

After each process completes its part of the computation, it calls `Tmk_exit` to exit the program. In order to obtain correct run-time statistics for a program, it must terminate via a call to `Tmk_exit`.

2.3 Getting Processes to Do Different Things

All TreadMarks^{TM} processes execute the same program. It is, however, easy to get different processes to do different things using the process id. The variable `Tmk_proc_id`, supplied by TreadMarks^{TM}, has value i for process i and can be used to identify a process. The execution of a program always starts with process 0.

For example, as we will see shortly in Section 2.4, memory allocation is typically done by process 0. Using `Tmk_proc_id` we can accomplish this as follows.

```
if (Tmk_proc_id == 0) {
  /* memory allocation code */
}
```

[1]See Section 3 of the Unix manual.

In addition to `Tmk_proc_id`, TreadMarks^{TM} supplies the total number of processes in a variable called `Tmk_nprocs`.

2.4 Memory Allocation

In TreadMarks^{TM}, shared memory must be allocated dynamically using `Tmk_malloc` or `Tmk_sbrk`. The deallocation routine is called `Tmk_free`. There is no statically allocated shared memory. Although shared memory can be allocated and freed at any time and by any process, typically process 0 allocates all of the shared memory required by the program immediately after `Tmk_startup`.

`Tmk_malloc` returns a pointer to the newly allocated region of shared memory. The value of this pointer can be stored in either a private or a shared variable by the allocating process (On the first shared memory allocation, it obviously has to be stored in a private variable, since there is no shared memory yet). If the pointer is stored in private memory, the TreadMarks^{TM} routine `Tmk_distribute` is often used to inform the other processes of the pointer's value. `Tmk_distribute` is called with two arguments, the address and the size of the pointer.

The following example shows how to allocate a shared array of `int`'s of size `arrayDim`. Process 0 does the actual allocation, and then informs all of the other processes of the shared array's address via the `Tmk_distribute` call. Note the use of a `Tmk_barrier` call by all of the processes after the memory allocation. The purpose of this barrier is to prevent any process from using `array` until its value has been distributed.

```
if (Tmk_proc_id == 0) {

    array = (int *) Tmk_malloc(arrayDim * sizeof(int));

    Tmk_distribute(&array, sizeof(array)); /* Send 4-byte ptr value */
}
Tmk_barrier(0);
```

2.5 Accessing Shared Data

The next step is to split up the computation among the different processes. TreadMarks^{TM} does not automatically parallelize a program, it is the programmer's responsibility to do so. How this is done is of course very much dependent on the application. Many considerations go into this decision, such as balancing the load among processes and minimizing interactions between processes. These considerations are not discussed any further here. Instead, we show a simple example. Suppose we want to initialize the array `array` of `int`'s of size `arrayDim` and we want to do so in parallel using all of the available processes. The code for allocating the array has been shown before. Here we show some sample code to execute the initialization in parallel.

```
start = Tmk_proc_id * (arrayDim / Tmk_nprocs);

end = (Tmk_proc_id + 1) * (arrayDim / Tmk_nprocs);

if (end > arrayDim) end = arrayDim;

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

The important aspect to note about this piece of code is what is *not* there: there is no code to move data between processes, the shared-memory system takes care of any data movement that might be required. Not having to worry about data movement is *the* advantage of shared-memory over message-passing.

As an aside, the distinction between shared data and private data is different from and orthogonal to the distinction between global data and local data. Shared vs. private distinguishes how many (really many vs. one) processes can access the data from somewhere in the code. Global vs. local distinguishes from which syntactic points in the program a process can access a data structure.

2.6 Synchronization

TreadMarks™ currently provides two synchronization primitives: locks and barriers. This section explains for each primitive the circumstances under which its usage is appropriate and the syntax for the code.

2.6.1 Locks

A *lock* is the simplest form of synchronization object. It supports two operations: acquire and release. The *acquire* operation obtains control of a lock for a single process. The *release* operation relinquishes control of a lock. Once a process controls a lock, subsequent acquires by different processes cannot complete until the process with control releases the lock.[2] Only the process that acquired the lock can release it. At an abstract level, a programmer associates a set of privileges with a lock. If a process wants to use one of those privileges, it must acquire the lock. Then it can use the privilege. After the process is done using the privilege, it should release the lock. A typical privilege for which we may use a lock is "Read the value in shared-memory location X" or "Write an updated value to shared-memory location X". The association of the lock with this privilege must be enforced *by the programmer*. The lock is not associated with the shared-memory location X in any formal way whatsoever.

There are two typical uses for locks:

1. one process at a time needs to read an updated value written by other processes or
2. multiple processes may be writing to the same shared memory location and we do not want the writes to conflict.

Regarding the second reason it is important to understand that simultaneous, unsynchronized writes to a shared memory location can cause the program to behave incorrectly. Even worse, this type of bug can be very difficult to find.

TreadMarks™ provides `TMK_NLOCKS` lock objects. Each lock is identified by an integer in the range from 0 to `TMK_NLOCKS` − 1. All of the locks are created and initialized during `Tmk_startup`. Thereafter, the program independently controls the allocation of locks. In other words, the program may allocate a lock without calling the TreadMarks™ library. Lock allocation is simply the *de facto* result of reserving and using a particular lock for a given purpose.

To acquire or release a lock, a process passes the lock's identifier to the appropriate routine. The syntax for acquiring a lock is:

```
Tmk_lock_acquire(lock_identifier);
```

[2] Currently, waiting processes obtain control of a lock in a FIFO order.

The syntax for releasing a lock is:

```
Tmk_lock_release(lock_identifier);
```

Typically an acquire-release pair will have a small amount of code, known as a "critical section" between them.

A lock identifier is ordinarily represented by an `unsigned` integer. It may be stored in private or shared memory, but if it is stored in private memory, all of the processes should be careful to initialize the identifier to the same value.

2.6.2 Barriers

A *barrier* impedes the progress of a process until all of the other processes reach the designated barrier. Arrival is the only operation provided by the barrier abstraction. TreadMarks^{TM} provides `TMK_NBARRIERS` barrier objects. Each barrier is identified by an integer in the range from 0 to `TMK_NBARRIERS` $- 1$. The details of barrier initialization, allocation and use are analogous to locks.

We return to our running example of the array `array` for which we have shown the memory allocation and the parallel initialization procedures. First, consider the memory allocation procedure. Process 0 allocates the array using `Tmk_malloc` and distributes the address using `Tmk_distribute`[3]. The other processes skip the body of the `if` statement. However, they must not proceed to the rest of the code, until they have the correct value in their local `array` variable. Therefore, a barrier is inserted after the allocation code, causing all processes to wait until process 0 has completed the distribution of `array`. In this example, we use barrier object #0. Any other mutually agreed upon barrier object would work as well. The resulting piece of code is typical of TreadMarks^{TM} memory allocation.

```
if (Tmk_proc_id == 0) {

    array = (int *) Tmk_malloc(arrayDim * sizeof(int));
    if (array == NULL)
        Tmk_errexit(``MALLOC ERROR - array\n'');

    Tmk_distribute(&array, sizeof(array)); /* Send 4-byte ptr value */
}
Tmk_barrier(0);
```

Furthermore, suppose we want process 0 to print out the values in `array` after initialization. Before we let process 0 do so, all other processes must have completed their portion of their work. Again, a barrier is inserted between the initialization and the print statement to get the desired behavior.

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

Tmk_barrier(0);

if (Tmk_proc_id == 0)
    for (i = 0; i < arrayDim; i++)
        printf("%d: %d\n", i, array[i]);
```

[3]`Tmk_distribute` is used because the variable holding the address is located in private memory

It is important to understand that the different processes can be at different syntactic locations in the program when all reach the same barrier at run-time. A typical situation where this arises is master/slave parallelism where the master would typically come to a barrier in a "master" routine and the slaves would come to the same barrier in a "slave" routine.

2.7 A Simple Example

The following is a complete, runnable program that contains all of the sample pieces of code given so far. The program takes a single application-dependent option specifying the size of the array.

```
/*
 * File app.c
 */
#include <stdio.h>

#include "Tmk.h"

extern  char   *optarg;

int     arrayDim = 100;
int    *array;

void main(int argc, char **argv)
{
    int     c, start, end, i;

    while ((c = getopt(argc, argv, "d:")) != -1)
        switch (c) {
        case 'd':
            arrayDim = atoi(optarg);
            break;
        }
    Tmk_startup(argc, argv);

    if (Tmk_proc_id == 0) {

        array = (int *) Tmk_malloc(arrayDim * sizeof(int));
        if (array == NULL)
            Tmk_errexit(``MALLOC ERROR - array\n'');

        Tmk_distribute(&array, sizeof(array)); /* Send 4-byte ptr value */
    }
    Tmk_barrier(0);

    start = Tmk_proc_id * (arrayDim / Tmk_nprocs);

    end = (Tmk_proc_id + 1) * (arrayDim / Tmk_nprocs);

    if (end > arrayDim) end = arrayDim;

    for (i = start; i < end; i++)
        array[i] = i;

    Tmk_barrier(0);
```

```
    if (Tmk_proc_id == 0 )
        for (i = 0; i < arrayDim; i++)
            printf("%d: %d\n", i, array[i]);

    Tmk_exit(0);
}
```

3 Compiling, Linking, and Running a TreadMarks™ Program

This section assumes that you have successfully installed the TreadMarks library following the procedure given in the installation guide.

3.1 Initialization Files

Optionally, two initialization files can be defined to improve the convenience and performance of using TreadMarks™. The use of these files, however, entails tradeoffs with flexibility and security.

If you will 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 must contain the network name of exactly one host. If you specify on the command line that N hosts 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 instead.

The alternative to using a `.Tmkrc` file is to specify the names of the host using the `-h` option on the command line.

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` will be 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™[4]. 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 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.

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

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
```

3.2 Compiling and Linking

TreadMarks™ is a run-time library. Program are compiled as usual, but in the linking phase you include the TreadMarks™ library. TreadMarks™ is currently available for SPARCstations running SunOS 4.x, R3000-based DECstations running Ultrix 4.3, Alpha-based DECs running OSF/1 V1.3, and IBM RS/6000-based machines running AIX version 3.2. To compile and link for a particular platform, an `ARCH` flag should be set 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
```

3.3 Running a Program

Both TreadMarks™ and the application program can accept command-line arguments. The command-line arguments for TreadMarks™ are normally separated from the application's arguments by a double-dash argument. The syntax is the following:

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

TreadMarks™ accepts the following flags, all of which are optional.

- -h *machine name*. One or more instances of this flag can be used to specify the machine(s) on which to run. The first machine will be designated process 0, and so on. When this flag is used, the "-n" flag described below is optional. Defaults to the machines listed in `.Tmkrc`.
- -f *file name*. File containing a list of machines on which to run, one per line. Defaults to `.Tmkrc`.
- -n *number*. Run on the specified number of machines. Defaults to the number of machines in your `.Tmkrc` file. This flag is ordinarily used to say that you only want to run on the first n machines in the `.Tmkrc` file.
- -r Use `rexec` rather than `rsh` to start remote processes. Default: off.
- -s Display the run-time statistics before exiting. Default: off.
- -x Open a separate X window attached to each of the remote processes. Anything these processes write to `stdout` or `stderr` or read from `stdin` is displayed to or read through their window. Default off.
- -D *size*. Change the size of the diff heap. The default is 2 Mbytes. (See the trouble-shooting section for more details.)
- -I *size*. Change the size of the interval heap. The default is 0.5 Mbytes. (See the trouble-shooting section for more details.)

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.

3.4 Other Sample Applications

At this point, it might be a good idea to try running the sample application we have used throughout this manual. In addition, the distribution provides a few sample applications in the `apps` directory.

4 Debugging a TreadMarks^{TM} Program

4.1 Using a Debugger

The following describes one method for debugging TreadMarks^{TM} programs. We will use `gdb` in our examples, but any other debugger with the capability to attach to a running process could be utilized.[5] Conceptually there can be a debugger process running on each processor that controls the behavior of the TreadMarks^{TM} 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`.[6] That file contains commands that will be executed whenever you start up the `gdb`. The one command required by TreadMarks^{TM} to be in this file is

```
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^{TM} 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^{TM} 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^{TM} 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.

[5] 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.

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

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.

On your processor B status window, you'll obtain the Unix process id for the TreadMarks™ 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.ether
```

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.

4.2 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.

5 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.
- **Internal TreadMarks™ data structures and the -I and -D flags -** in programs that have an extraordinary amount of synchronization, TreadMarks™ may run out of internal storage space. If this happens, an error message will be printed to `stderr` on the processor(s) that ran out of space. For example, this message might appear as "Tmk_diff_create: diff space exhausted (diff_SIZE == NNN)". If so, you can increase the amount of *diff* space with the command-line argument "-D MMM" where MMM > NNN. Analogous messages are generated, specifying "interval space", if that storage pool runs low. You can increase the size of the interval pool with the command-line argument "-I MMM".
- **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.

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