

TOTALVIEW

**UNDERSTANDING
THREADS,
PROCESSES, AND
GROUPS**



VERSION 8.6



Copyright © 2007–2008 by TotalView Technologies. All rights reserved

Copyright © 1998–2007 by Etnus LLC. All rights reserved.

Copyright © 1996–1998 by Dolphin Interconnect Solutions, Inc.

Copyright © 1993–1996 by BBN Systems and Technologies, a division of BBN Corporation.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise without the prior written permission of TotalView Technologies.

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

TotalView Technologies has prepared this manual for the exclusive use of its customers, personnel, and licensees. The information in this manual is subject to change without notice, and should not be construed as a commitment by TotalView Technologies. TotalView Technologies assumes no responsibility for any errors that appear in this document.

TotalView and TotalView Technologies are registered trademarks of TotalView Technologies.

TotalView uses a modified version of the Microline widget library. Under the terms of its license, you are entitled to use these modifications. The source code is available at <http://www.totalviewtech.com/Products/TotalView/developers>.

All other brand names are the trademarks of their respective holders.

Contents



Using Threads, Processes, and Groups

A Couple of Processes	1
Threads	3
Complicated Programming Models	4
Types of Threads	6
Organizing Chaos	8
Creating Groups	12
Simplifying What You're Debugging	16
INDEX	19

Using Threads, Processes, and Groups



While the specifics of how multi-process, multi-threaded programs execute differ greatly from one hardware platform to another, from one operating system to another, and from one compiler to another, all share some general characteristics. This chapter defines a general model for conceptualizing the way processes and threads execute.

This chapter contains the following sections:

- “A Couple of Processes” on page 1
- “Threads” on page 3
- “Complicated Programming Models” on page 4
- “Types of Threads” on page 6
- “Organizing Chaos” on page 8
- “Creating Groups” on page 12
- “Simplifying What You’re Debugging” on page 16

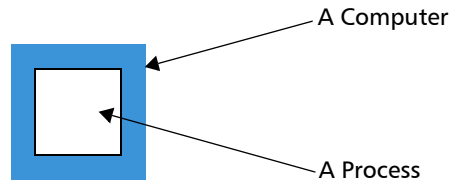
A Couple of Processes

breakpoint When programmers write single-threaded, single-process programs, they can almost always answer the question “Do you know where your program is?” These types of programs are rather simple, looking something like what’s shown in the [following](#) figure on the next page.

If you use any debugger on these types of programs, you can almost always figure out what’s going on. Before the program begins executing, you set a breakpoint, let the program run until it hits the breakpoint, and then inspect variables to see their values. If you suspect that there’s a logic problem, you can step the program through its statements, seeing what happens and where things are going wrong.

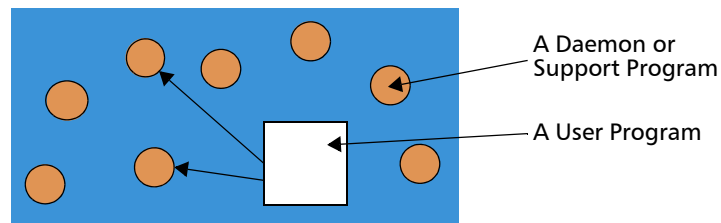
A Couple of Processes

Figure 1: A Uniprocessor



What is actually occurring, however, is a lot more complicated, since a number of programs are always executing on your computer. For example, your computing environment could have daemons and other support programs executing, and your program can interact with them.

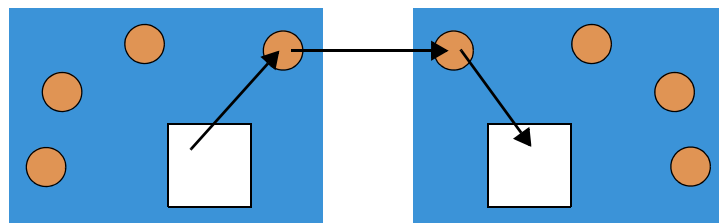
Figure 2: A Program and Daemons



These additional processes can simplify your program because it no longer has to do everything itself. It can hand off some tasks and not have to focus on how that work gets done.

The preceding figure shows an architecture where the application program just sends requests to a daemon. This architecture is very simple. The type of architecture shown in the next figure is more typical. In this example, an email program communicates with a daemon on one computer. After receiving a request, this daemon sends data to an email daemon on another computer, which then delivers the data to another mail program.

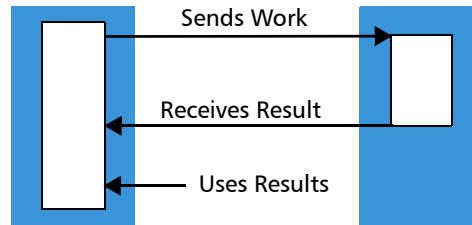
Figure 3: Mail Using Daemons to Communicate



This architecture has one program handing off work to another. After the handoff, the programs do not interact. The program handing off the work just assumes that the work gets done. Some programs can work well like this. Most don't. Most computational jobs do better with a model that allows a program to divide its work into smaller jobs, and parcel this work to other computers. Said in a different way, this model has other machines do some of the first program's work. To gain any advantage, however, the work a program parcels out must be work that it doesn't need right away. In

this model, the two computers act more or less independently. And, because the first computer doesn't have to do all the work, the program can complete its work faster.

Figure 4: Two Computers Working on One Problem



Using more than one computer doesn't mean that less computer time is being used. Overhead due to sending data across the network and overhead for coordinating multi-processing always means more work is being done. It does mean, however, that your program finishes sooner than if only one computer were working on the problem.

One problem with this model is how a programmer debugs what's happening on the second computer. One solution is to have a debugger running on each computer. The TotalView solution to this debugging problem places a server on each remote processor as it is launched. These servers then communicate with the main TotalView process. This debugging architecture gives you one central location from which you can manage and examine all aspects of your program.



You can also have TotalView attach to programs that are already running on other computers. In other words, programs don't have to be started from within TotalView to be debugged by TotalView.

In all cases, it is far easier to write your program so that it only uses one computer at first. After you have it working, you can split up its work so that it uses other computers. It is likely that any problems you find will occur in the code that splits up the program or in the way the programs manipulate shared data, or in some other area related to the use of more than one thread or process. This assumes, of course, that it is practical to write your program as a single-process program. For some algorithms, executing a program on one computer means that it will take weeks to execute.

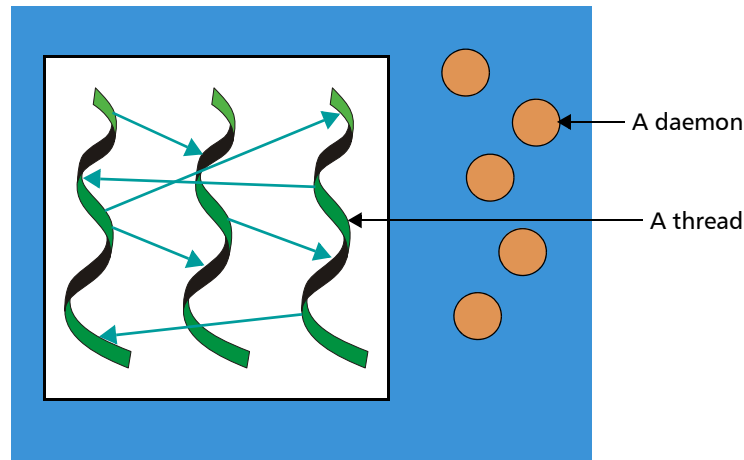
Threads

breakpoint The operating system owns the daemon programs discussed in the previous section. These daemons perform a variety of activities, from

managing computer resources to providing standard services such as printing.

If operating systems can have many independently executing components, why can't a program? Obviously, a program can and there are various ways to do this. One programming model splits the work off into somewhat independent tasks within the same process. This is the *threads* model.

Figure 5: Threads



This figure also shows the daemon processes that are executing. (The figures in the rest of this chapter won't show these daemons.)

In this computing model, a program (the main thread) creates threads. If they need to, these newly created threads can also create threads. Each thread executes relatively independently from other threads. You can, of course, program them to share data and to synchronize how they execute.

The debugging issue here is similar to the problem of processes running on different machines. In both, a debugger must intervene with more than one executing entity. It has to understand multiple address spaces and multiple contexts.



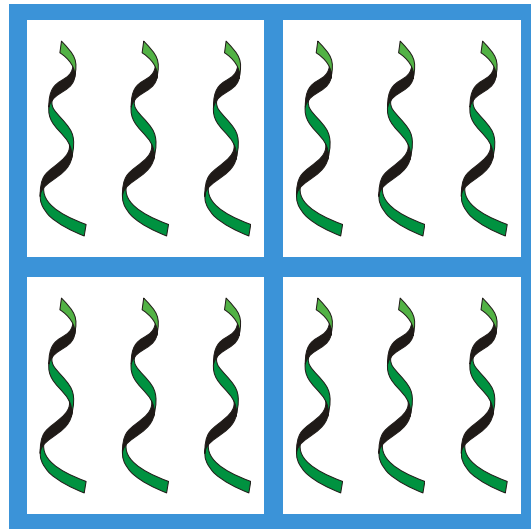
There's not a lot of difference between a multi-threaded or a multi-process program when you are using TotalView. The way in which TotalView displays process information is very similar to how it displays thread information.

Complicated Programming Models

breakpoint While most computers have one or two processors, high-performance computing often uses computers with many more. And as hardware prices decrease, this model is starting to become more widespread. Having more than one processor means that the threads model shown in the figure

in the previous section changes to look something like what is shown in Figure 6. (Only four cores are shown even though many more could on a chip.)like this:

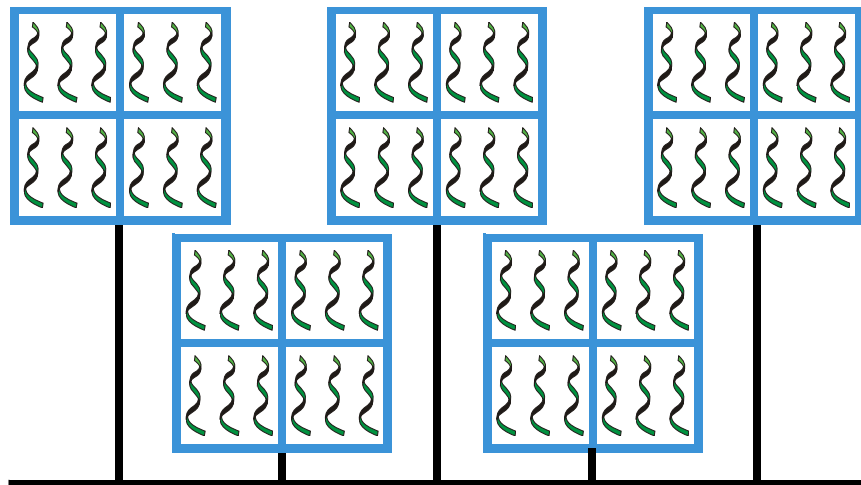
Figure 6: Four-Processor Computer



This figure shows four cores in one computer, each of which has three threads. This architecture is an extension to the model that links more than one computer together. Its advantage is that the processor doesn't need to communicate with other processors over a network as it is completely self-contained.

The next step is to join many multi-processor computers together. (See Figure 7 on page 5.)The following figure shows five computers, each with four processors, with each processor running three threads. If this figure shows the execution of one program, then the program is using 60 threads.

Figure 7: Four Processors on a Network About Threads, Processes, and Groups



This figure depicts only processors and threads. It doesn't have any information about the nature of the programs and threads or even whether the programs are copies of one another or represent different executables.

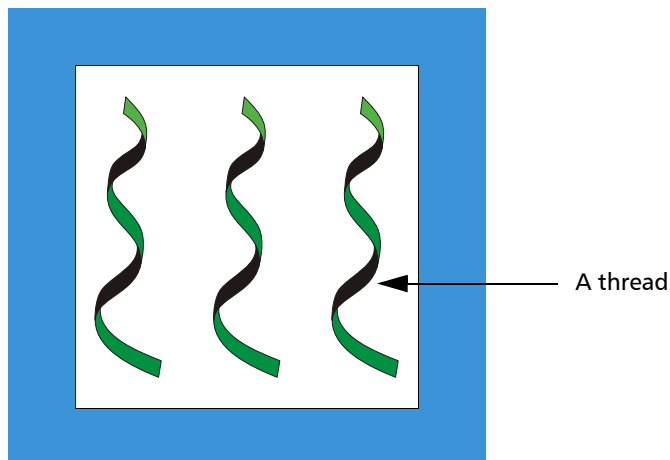
At any time, it is next to impossible to guess which threads are executing and what a thread is actually doing. To make matters worse, many multi-processor programs begin by invoking a process such as **mpirun** or IBM **poe**, whose function is to distribute and control the work being performed. In this kind of environment, a program is using another program to control the workflow across processors.

When there are problems working this way, traditional debuggers and solutions don't work. TotalView, on the other hand, organizes this mass of executing procedures for you and lets you distinguish between threads and processes that the operating system uses from those that your program uses.

Types of Threads

breakpoint All threads aren't the same. The following figure shows a program with three threads. (See Figure 8.)

Figure 8: Threads (again)



Assume that all of these threads are *user threads*; that is, they are threads that perform some activity that you've programmed.

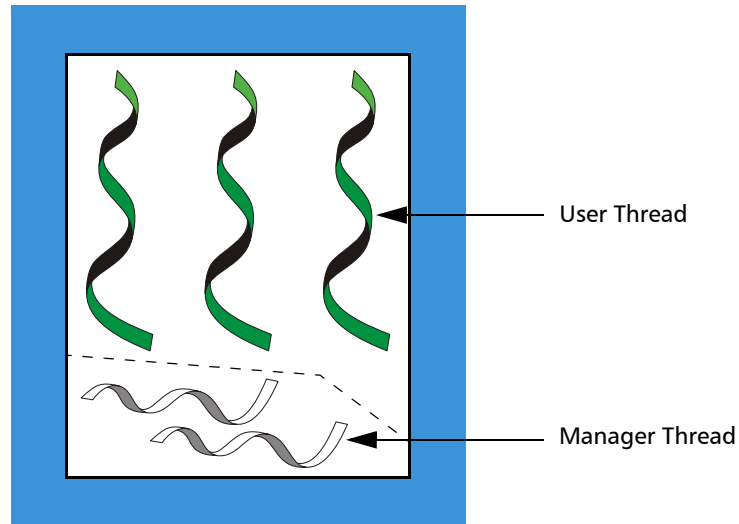


Many computer architectures have something called *user mode*, *user space*, or something similar. *User threads* means something else. The TotalView definition of a *user thread* is simply a unit of execution created by a program.

Because the program creates user threads to do its work, they are also called *worker threads*.

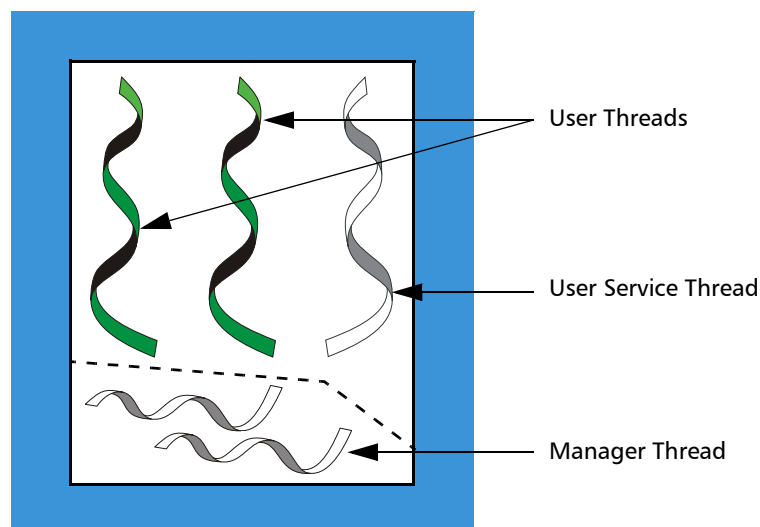
Other threads can also be executing. For example, there are always threads that are part of the operating environment. These threads are called *manager threads*. Manager threads exist to help your program get its work done. In the following figure, the horizontal threads at the bottom are user-created manager threads.

Figure 9: User and Service Threads



All threads are not created equal and all threads do not execute equally. Many programs also create manager-like threads. Since these user-created manager threads perform services for other threads, they are called *service threads*. (See Figure 10 on page 7.)

Figure 10: User, Service, and Manager Threads



These service threads are also worker threads. For example, the sole function of a user service thread might be to send data to a printer in response to a request from the other two threads.

One reason you need to know which of your threads are service threads is that a service thread performs different types of activities than your other threads. Because their activities are different, they are usually developed separately and, in many cases, are not involved with the fundamental problems being solved by the program. Here are two examples:

- The code that sends messages between processes is far different than the code that performs fast Fourier transforms. Its bugs are quite different than the bugs that create the data that is being transformed.
- A service thread that queues and dispatches messages sent from other threads might have bugs, but the bugs are different than the rest of your code and you can handle them separately from the bugs that occur in nonservice user threads.

Being able to distinguish between the two kinds of threads means that you can focus on the threads and processes that actively participate in an activity, rather than on threads performing subordinate tasks.

Although this last figure shows five threads, most of your debugging effort will focus on just two threads.

Organizing Chaos

breakpoint It is possible to debug programs that are running thousands of processes and threads across hundreds of computers by individually looking at each. However, this is almost always impractical. The only workable approach is to organize your processes and threads into groups and then debug your program by using these groups. In other words, in a multi-process, multi-threaded program, you are most often not programming each process or thread individually. Instead, most high-performance computing programs perform the same or similar activities on different sets of data.

TotalView cannot know your program's architecture; however, it can make some intelligent guesses based on what your program is executing and where the program counter is. Using this information, TotalView automatically organizes your processes and threads into the following predefined groups:

- **Control Group:** All the processes that a program creates. These processes can be local or remote. If your program uses processes that it did not create, TotalView places them in separate control groups. For example, a client/server program that has two distinct executables that run independently of one another has each executable in a separate control group. In contrast, processes created by `fork()/exec()` are in the same control group.
- **Share Group:** All the processes within a control group that share the same code. *Same code* means that the processes have the same execut-

able file name and path. In most cases, your program has more than one share group. Share groups, like control groups, can be local or remote.

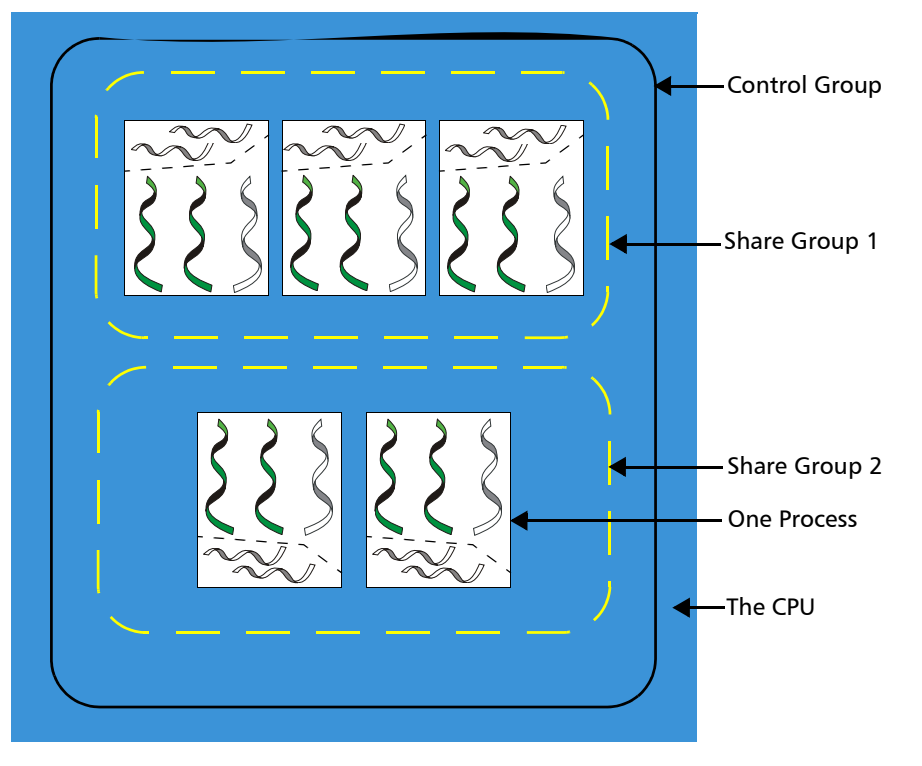
- **Workers Group:** All the worker threads within a control group. These threads can reside in more than one share group.
- **Lockstep Group:** All threads that are at the same PC (program counter). This group is a subset of a workers group. A lockstep group only exists for stopped threads. By definition, all members of a lockstep group are within the same workers group. That is, a lockstep group cannot have members in more than one workers group or more than one control group. A lockstep group only means anything when the threads are stopped.

The control and share groups only contain processes; the workers and lockstep groups only contain threads.

TotalView lets you manipulate processes and threads individually and by groups. In addition, you can create your own groups and manipulate a group's contents (to some extent).

The following figure shows a processor running five processes (ignoring daemons and other programs not related to your program) and the threads within the processes. This figure shows a control groups and two share groups within the control group.

Figure 11: Five-Processes:
Their Control and Share
Groups

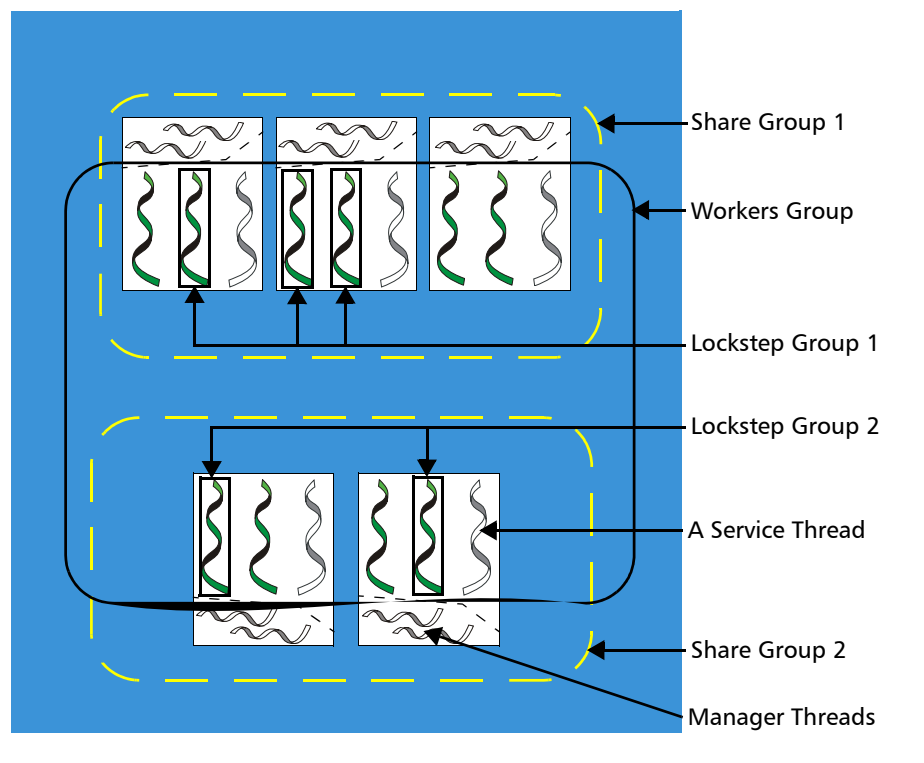


Many of the elements in this figure are used in other figures in this book. These elements are as follows:

- CPU** The one outer square represents the CPU. All elements in the drawing operate within one CPU.
- Processes** The five white inner squares represent processes being executed by the CPU.
- Control Group** The large rounded rectangle that surrounds the five processes shows one control group. This diagram doesn't indicate which process is the main procedure.
- Share Groups** The two smaller rounded rectangles having white dashed lines surround processes in a share group. This drawing shows two share groups within one control group. The three processes in the first share group have the same executable. The two processes in the second share group share a second executable.

The control group and the share group only contain processes. The next figure shows how TotalView organizes the threads in the previous figure. It adds a workers group and two lockstep groups. (See Figure 12 on page 10.)

Figure 12: Five Processes: Adding Workers and Lockstep Groups



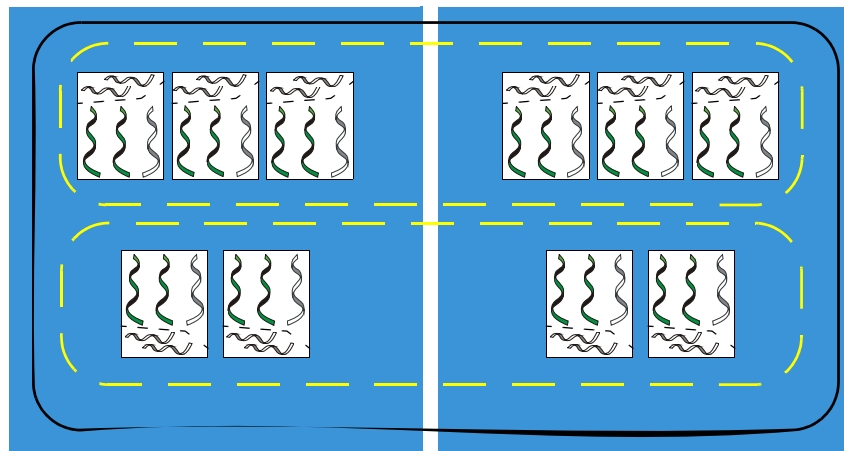
This figure doesn't show the control group since it encompasses everything in this figure. That is, this example's control group contains all of the program's lockstep, share, and worker group's processes and threads.

The additional elements in this figure are as follows:

- Workers Group** All nonmanager threads within the control group make up the workers group. This group includes service threads.
- Lockstep Groups** Each share group has its own lockstep group. The previous figure shows two lockstep groups, one in each share group.
- Service Threads** Each process has one service thread. A process can have any number of service threads, but this figure only shows one.
- Manager Threads** The ten manager threads are the only threads that do not participate in the workers group.

The following figure extends the previous figure to show the same kinds of information executing on two processors.

Figure 13: Five Processes and Their Groups on Two Computers

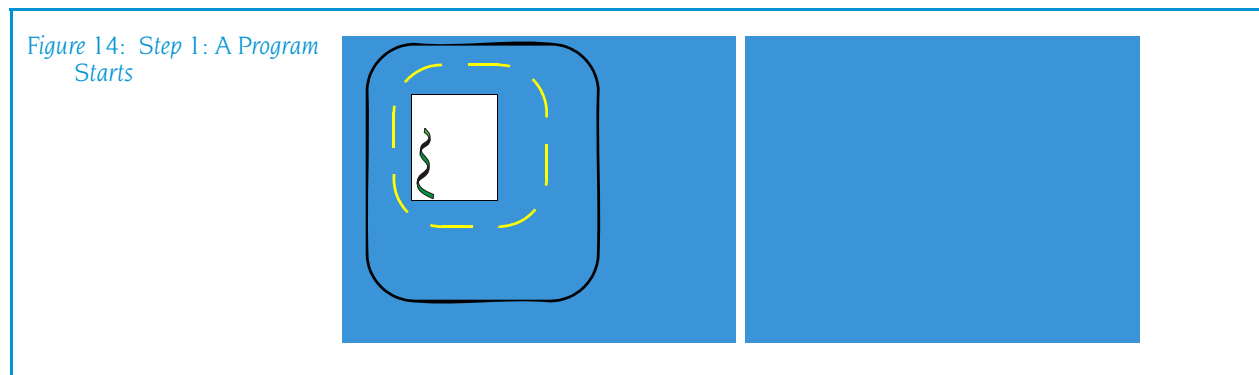


This figure differs from the other ones in this section because it shows ten processes executing within two processors rather than five processes within one processor. Although the number of processors has changed, the number of control and share groups is unchanged. This makes a nice example. However, most programs are not this regular.

Creating Groups

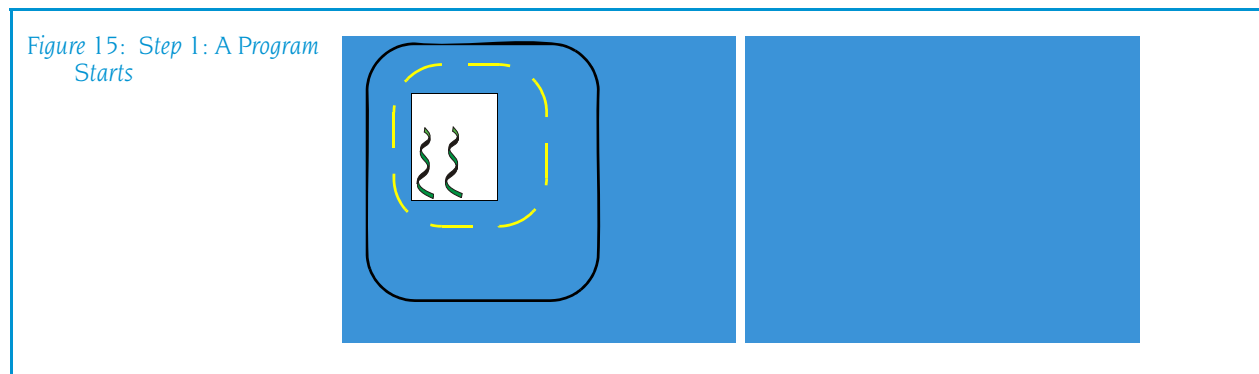
breakpointTotalView places processes and threads in groups as your program creates them. The exception is the lockstep groups that are created or changed whenever a process or thread hits an action point or is stopped for any reason. There are many ways to build this type of organization. The following steps indicate the beginning of how TotalView might do this.

Step 1 TotalView and your program are launched and your program begins executing.



- **Control group:** The program is loaded and creates a group.
- **Share group:** The program begins executing and creates a group.
- **Workers group:** The thread in the `main()` routine is the workers group.
- **Lockstep group:** There is no lockstep group because the thread is running. (Lockstep groups only contain stopped threads.)

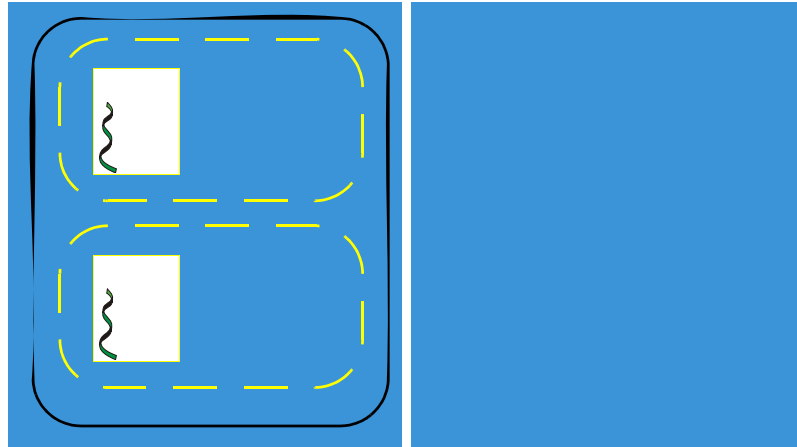
Step 2 The program creates a thread.



- **Control group:** The control group is unchanged.
- **Share group:** The share group is unchanged.
- **Workers group:** TotalView adds the thread to the existing group.
- **Lockstep group:** There are no lockstep groups because the threads are running.

Step 3 The first process uses the `exec()` function to create a second process. (See Figure 16.)

Figure 16: Step 3: Creating a Process using `exec()`



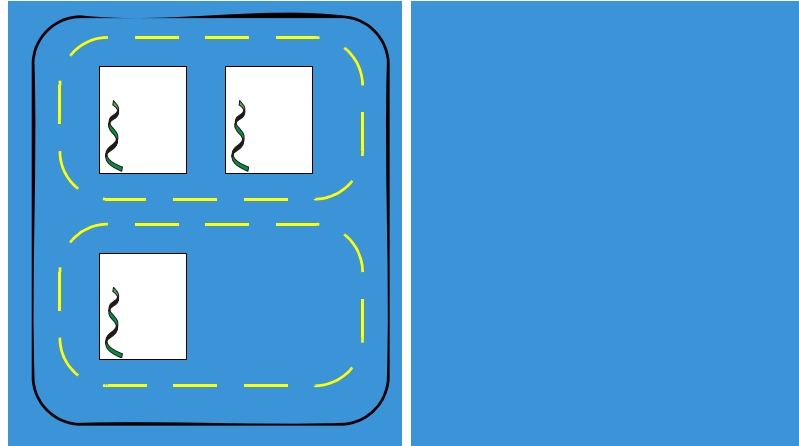
- **Control group:** The group is unchanged.
- **Share group:** TotalView creates a second share group with the process created by the `exec()` function as a member. TotalView removes this process from the first share group.
- **Workers group:** Both threads are in the workers group.
- **Lockstep group:** There are no lockstep groups because the threads are running.

Step 4 The first process hits a break point.

- **Control group:** The group is unchanged.
- **Share group:** The groups are unchanged.
- **Workers group:** The group is unchanged.
- **Lockstep group:** TotalView creates a lockstep group whose member is the thread of the current process. (In this example, each thread is its own lockstep group.)

- Step 5** The program is continued and TotalView starts a second version of your program from the shell. You attach to it within TotalView and put it in the same control group as your first process.

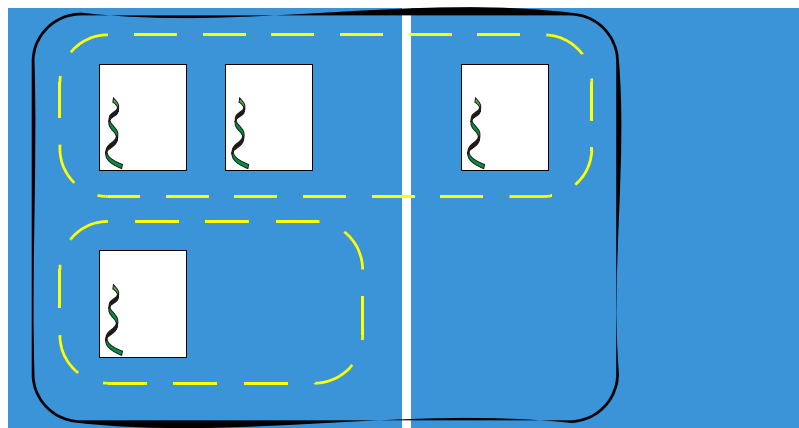
Figure 17: Step 5: Creating a Second Version



- **Control group:** TotalView adds a third process.
- **Share group:** TotalView adds this third process to the first share group.
- **Workers group:** TotalView adds the thread in this third process to the group.
- **Lockstep group:** There are no lockstep groups because the threads are running.

- Step 6** Your program creates a process on another computer.

Figure 18: Step 6: Creating a Remote Process

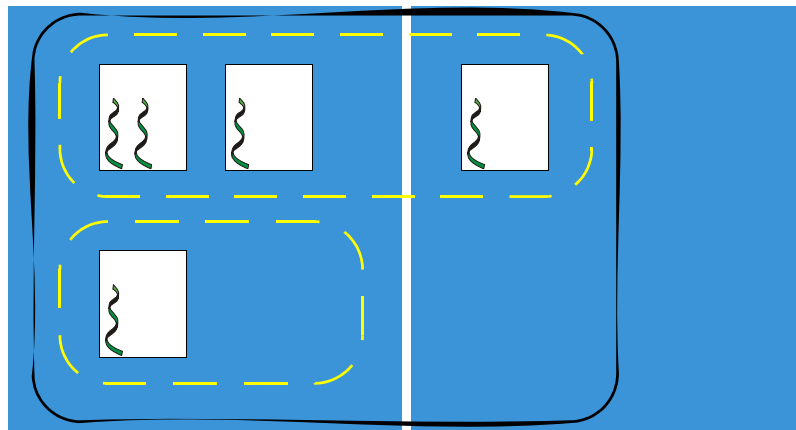


- **Control group:** TotalView extends the control group so that it contains the fourth process, which is running on the second computer.
- **Share group:** The first share group now contains this newly created process, even though it is running on the second computer.
- **Workers group:** TotalView adds the thread within this fourth process to the workers group.

- **Lockstep group:** There are no lockstep groups because the threads are running.

Step 7 A process within the control group creates a thread. This adds a second thread to one of the processes.

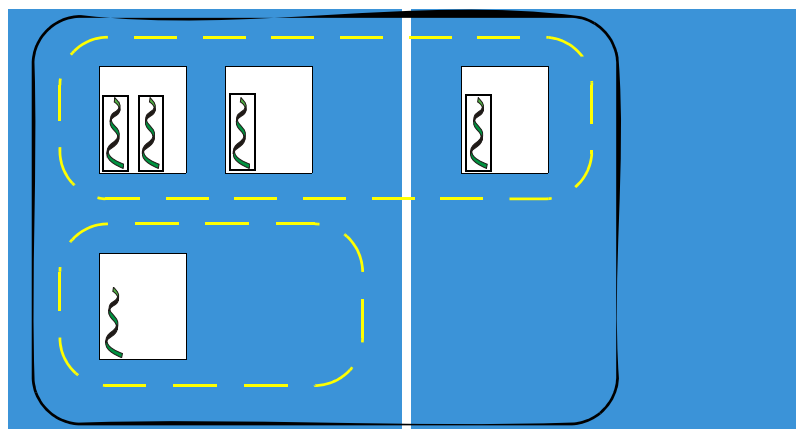
Figure 19: Step 7: Creating a Thread



- **Control group:** The group is unchanged.
- **Share group:** The group is unchanged.
- **Workers group:** TotalView adds a fifth thread to this group.
- **Lockstep group:** There are no lockstep groups because the threads are running.

Step 8 A breakpoint is set on a line in a process executing in the first share group. By default, TotalView shares the breakpoint. The program executes until all three processes are at the breakpoint.

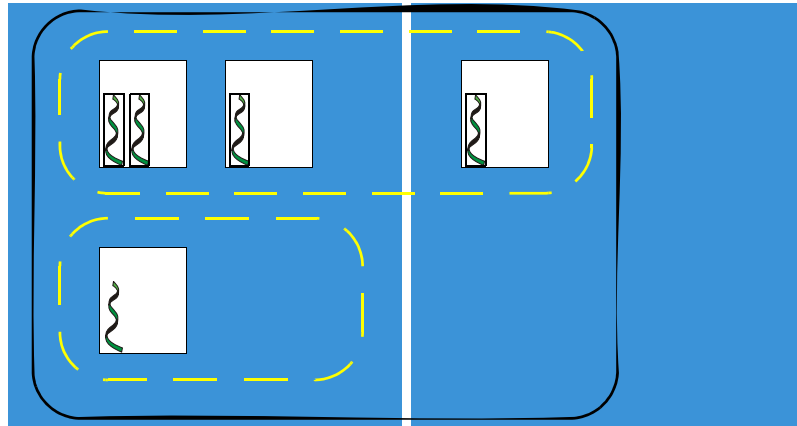
Figure 20: Step 8: Hitting a Breakpoint



- **Control group:** The group is unchanged.
- **Share group:** The groups are unchanged.
- **Workers group:** The group is unchanged.
- **Lockstep groups:** TotalView creates a lockstep group whose members are the four threads in the first share group.

Step 9 You tell TotalView to step the lockstep group.

Figure 21: Step 9: Stepping the Lockstep Group



- **Control group:** The group is unchanged.
- **Share group:** The groups are unchanged.
- **Workers group:** The group is unchanged.
- **Lockstep group:** The lockstep groups are unchanged.

What Comes Next

This example could keep on going to create a more complicated system of processes and threads. However, adding more processes and threads won't change the basics of what has been covered.

Simplifying What You're Debugging

The reason you're using a debugger is because your program isn't operating correctly and the way you think you're going to solve the problem (unless it is a $\&\%\$\#$ operating system problem, which, of course, it usually is) is by stopping your program's threads, examining the values assigned to variables, and stepping your program so you can see what's happening as it executes.

Unfortunately, your multi-process, multi-threaded program and the computers upon which it is executing have lots of things executing that you want TotalView to ignore. For example, you don't want to be examining manager and service threads that the operating system, your programming environment, and your program create.

Also, most of us are incapable of understanding exactly how a program is acting when perhaps thousands of processes are executing asynchronously. Fortunately, there are only a few problems that require full asynchronous behavior at all times.

One of the first simplifications you can make is to change the number of processes. For example, suppose you have a buggy MPI program running

on 128 processors. Your first step might be to have it execute in an 8-processor environment.

After you get the program running under TotalView control, run the process being debugged to an action point so that you can inspect the program's state at that point. In many cases, because your program has places where processes are forced to wait for an interaction with other processes, you can ignore what they are doing.



TotalView lets you control as many groups, processes, and threads as you need to control. Although you can control each one individually, you might have problems remembering what you're doing if you're controlling large numbers of these things independently. TotalView creates and manages groups so that you can focus on portions of your program.

In most cases, you don't need to interact with everything that is executing. Instead, you want to focus on one process and the data that this process manipulates. Things get complicated when the process being investigated is using data created by other processes, and these processes might be dependent on other processes.

The following is a typical way to use TotalView to locate problems:

- 1 At some point, make sure that the groups you are manipulating do not contain service or manager threads. (You can remove processes and threads from a group by using the **Group > Custom Group** command.)

```
CLI: dgroups -remove
```

- 2 Place a breakpoint in a process or thread and begin investigating the problem. In many cases, you are setting a breakpoint at a place where you hope the program is still executing correctly. Because you are debugging a multi-process, multi-threaded program, set a *barrier point* so that all threads and process will stop at the same place.



Do not step your program except where you need to individually look at what occurs in a thread. Using barrier points is much more efficient.

- 3 After execution stops at a barrier point, look at the contents of your variables. Verify that your program state is actually correct.
- 4 Begin stepping your program through its code. In most cases, step your program synchronously or set barriers so that everything isn't running freely.

Things begin to get complicated at this point. You've been focusing on one process or thread. If another process or thread modifies the data and you become convinced that this is the problem, you need to go off to it and see what's going on.

You need to keep your focus narrow so that you're only investigating a limited number of behaviors. This is where debugging becomes an art. A multi-process, multi-threaded program can be doing a great number of things. Understanding where to look when problems occur is the art.

For example, you most often execute commands at the default focus. Only when you think that the problem is occurring in another process do you change to that process. You still execute in the default focus, but this time the default focus changes to another process.

Although it seems like you're often shifting from one focus to another, you probably will do the following:

- Modify the focus so that it affects just the next command. If you are using the GUI, you might select this process and thread from the list displayed in the Root Window. If you are using the CLI, you use the **dfocus** command to limit the scope of a future command. For example, the following is the CLI command that steps thread 7 in process 3:

```
dfocus t3.7 dstep
```

- Use the **dfocus** command to change focus temporarily, execute a few commands, and then return to the original focus.

Index



A

asynchronous processing 2

B

barrier points 17

C

control groups 10
defined 8

creating groups 12
creating threads 4

D

daemons 2, 4
debugging techniques 16
dgroups command
–remove 17
disconnected processing 2
dividing work up 2

F

figures
Five Processes and Their Groups
on Two Computers 11
Two Computers Working on One
Problem 3
Five Processes and Their Groups on
Two Computers figure 11
four linked processors 5

G

Group > Custom Group command 17
groups
creating 12
defined 8, 9
overview 8

L

lockstep group 11
defined 9

M

manager threads 7, 11
multiprocessing 5

P

process groups 9
processors and threads 6

S

server on each processor 3
service threads 7, 11
share groups 10
defined 8
splitting up work 3

T

thread groups 9
threads
creating 4
manager 7
service 7
user 6
workers 6, 8
threads model 4
Two Computers Working on One Prob-
lem figure 3

U

user mode 6
user threads 6

W

worker threads 6
workers group 11
defined 9
working independently 2

