Lab Manual

TotalView and MemoryScape Training

Table of Contents

Lab 1: Debugger Basics: Startup, Basic Process Control, and Navigation

This lab covers basic process control, including stepping, breakpoint basics and source code navigation.

There are some basic commands you need to know to drive the debugger. You can step through your source code one line at a time (single-stepping) and examine the program state. This state includes global and local variables, the stack frame, and the stack trace. Or, you can tell TotalView to run your program and stop at a particular line in the source code (setting a breakpoint). You also need to know how to halt a running program as well as resume it later. These commands are probably all you need to debug simple programs.

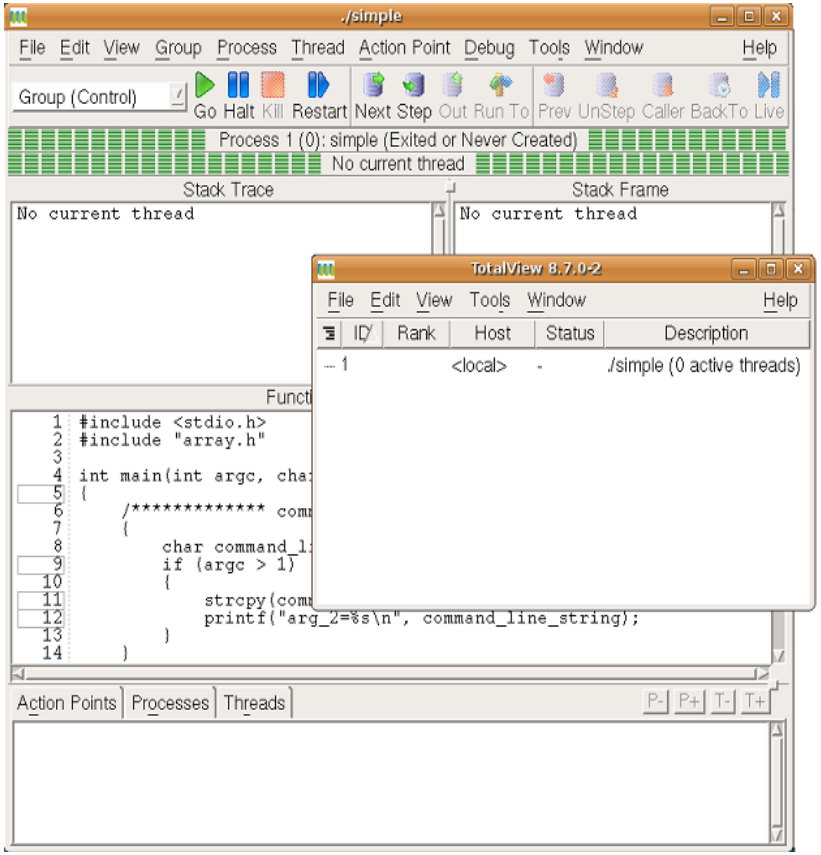
Expected Time: 30 minutes

Step 1: First Steps

- Open a Terminal Window
- Change directory to `$LABS`
  `cd $LABS`
- Compile your program
  `gcc -g $SRC/array.c $SRC/simple.c -lm -o simple`
- Start TotalView by typing
  `totalview ./simple -a hello`

A Startup Parameters Window may open at this point. You can press OK to clear it. It is not used for this lab.

You are now seeing two windows. Depending on your screen size you may need to rearrange them to see both windows. The larger is the Process Window. The smaller is the Root Window.



Note: If you are seeing assembler, you probably forgot to use the –g option when you compiled your program.

Notes:

- The `-a` argument indicates that all arguments after it will be arguments to the target program. In this case, “hello” is sent to `simple`.
- The Stack Trace Pane (the top left area of the window) shows if there are any active threads. When you first start the debugger session, it should show `No current thread`.
- The Stack Frame Pane on the right shows the same message.
- The middle area contains your source code for the `main()` function. We call this area the Source Pane. Note how the line above the Source Pane indicates the function and filename the Source Pane is currently focused on.

Step 2: Navigation

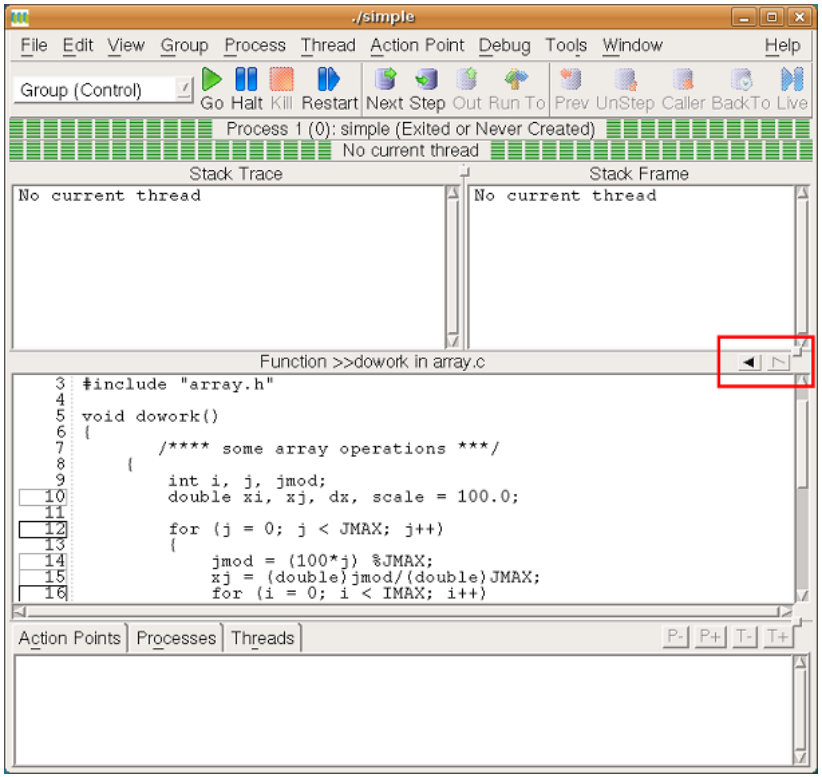
Dive on the `array()` function in the Source Pane:

Place your cursor on the word **`array`** in line 16 or 18 and double-click.

This focuses you on the `array` function in the `array.c` file.

• Select the **View > Lookup Function** command • Type **dowork** • Press **OK**

The Source Pane should now be focused on the `dowork` function in `array.c`.



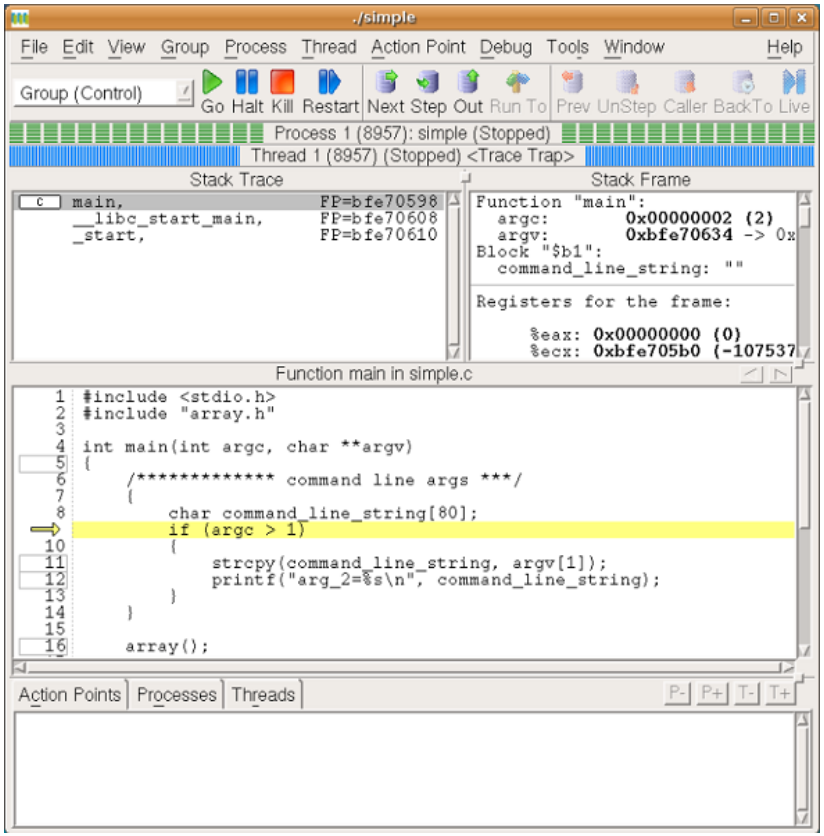
Notes

Note the left and right arrows in the top right hand corner of the Source Pane. We refer to this as the dive stack. Click on the left arrow. It will focus you back to the source you were looking at prior to your last dive. Note that now the right arrow will be enabled, allowing you to 're-dive.'

Step 3: Stepping

Press the **Step** icon in the toolbar

TotalView stops right before the first executable statement.



Notes

- You could also choose the Process > Step command. Also, notice that the command on the Process menu shows the 's' keyboard shortcut. As you gain experience, you'll find the keyboard shortcuts are convenient and speedup debugging.
- Notice the yellow arrow on the left. This is the PC or the Program Counter. It shows you where you are in the program. The Stack Trace Pane (top left) now shows that the program `main` is active and that it is C language code. The Stack Frame is now loaded by the C runtime library and the function `main` is on the stack frame with its command line arguments. It also shows local variables in scope. (Uninitialized variables contain random information.)

Question

1. Where can you find the state of your currently focused process/thread?

_____

Step 4: More Stepping

Select the **Process > Step Instruction** command

Questions

2. What did choosing the Step Instruction command do, if anything?

3. Why doesn't it look like anything changed?

4. What can you do to see the effect of stepping an instruction?

_____

Step 5: Run To and Step

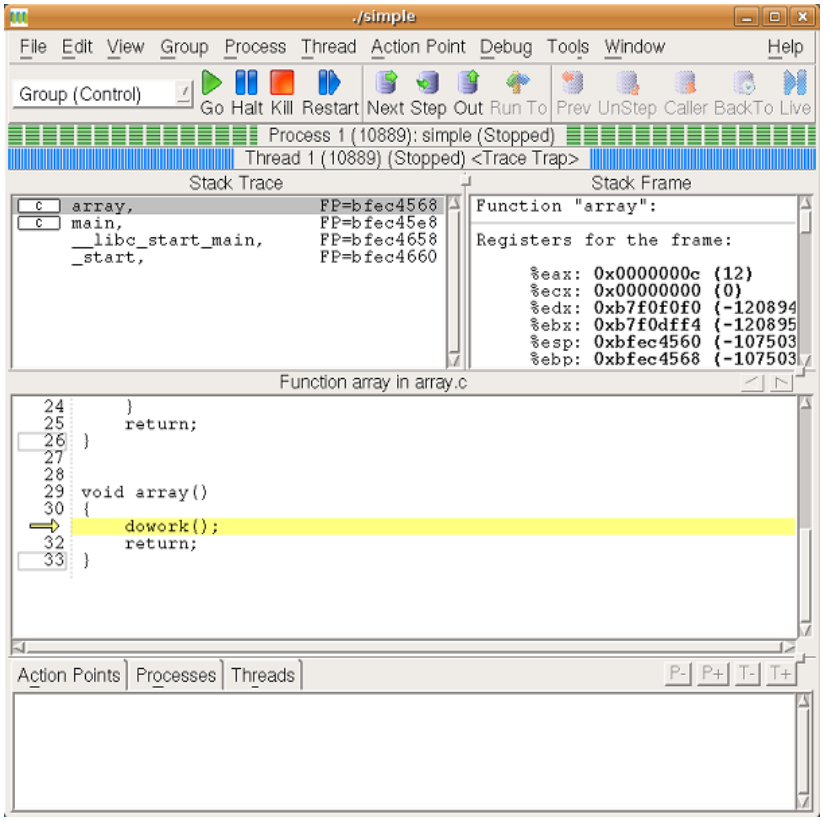
Select line 16 (**not the line number**), the first call to `array()`

Selecting the line will highlight it with a gray bar.

Press the **Run To** icon

This runs the process up to that line of code.

Do a **Step**



Note how the Stack Trace, Stack Frame, and Source Panes all change.

- Run To line 22 – the call to `printf( )`
- Do a **Step**

Questions

5. Why didn't you step into `printf()`?

_____

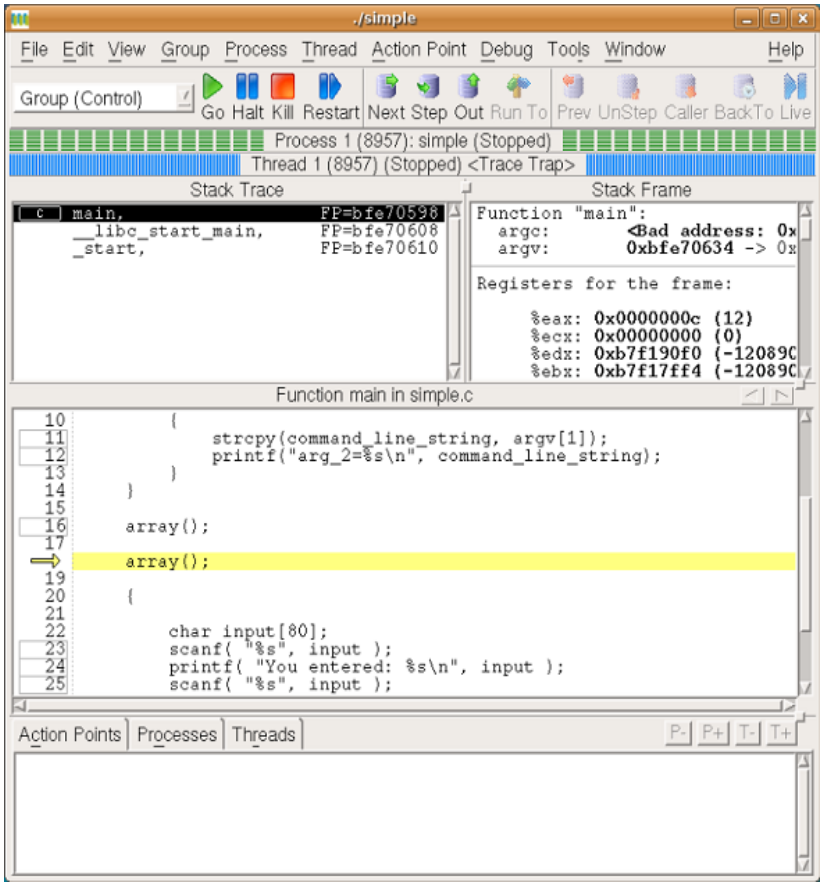
6. Where did the output from the `printf()` call go?

_____

Step 6: Moving Out

- Select `main` in the Stack Trace Pane
- Press the **Out** button

Your Process Window should look as follows:



Question

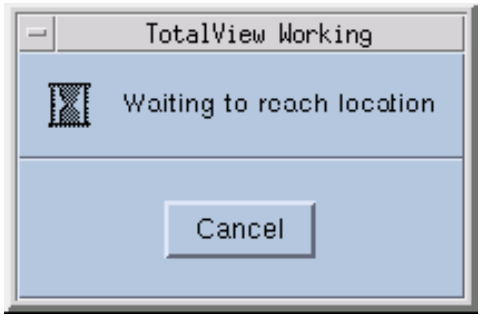
7. What did this do?

_____

Step 7: Waiting

- Run to line 23
- Select the **Next** icon

After a couple of seconds, TotalView displays the following dialog box:



Questions

8. Why did TotalView display this message?

_____

9. What will happen if you enter input on `stdin`?

_____

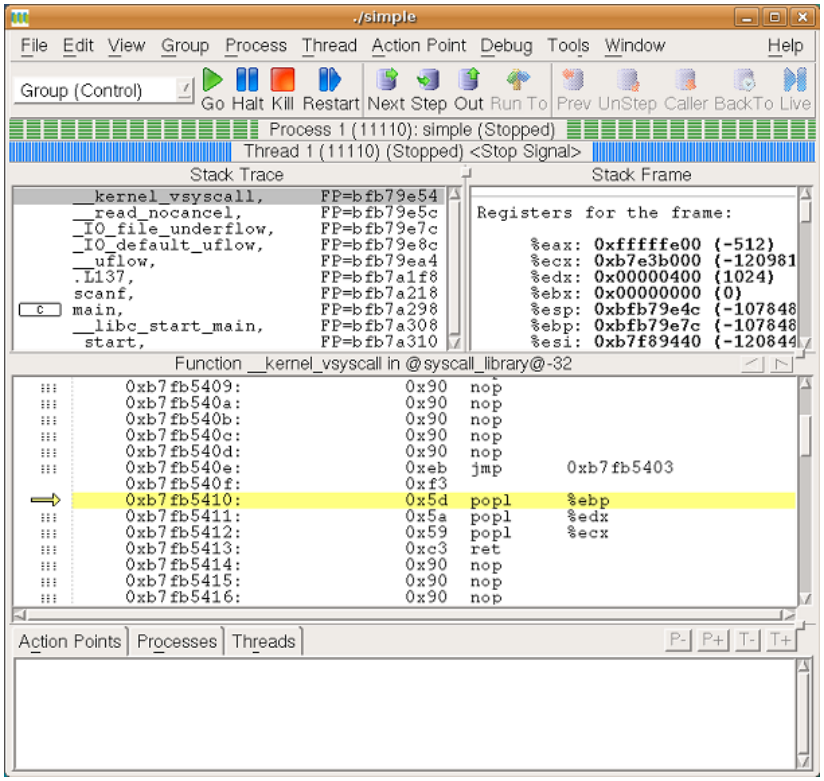
10. What will happen if you press Cancel?

_____

Step 8: Canceling

Press **Cancel**

Your Process Window will look something like this.



The Stack Trace Pane shows you that the process is currently within a system call. The Source Pane shows you assembler code, and the line immediately above the Source Pane tells you the library you're in

rather than the source file. This is because this module was not built with debug information and TotalView always focuses you on the stack frame where your PC currently is, regardless of whether there is debug information or not.

Within the Stack Trace Pane, `main` is preceded by `C`. This means that TotalView has debug information for that frame and the language is C.

Click on `main` within the Stack Trace Pane

You should now see the source code and the PC arrow should be pointing at the `scanf()` call.

Press the **Out** button

After a few seconds you should again see the Waiting to reach location dialog box. Do not click the Cancel button.

- Go back to your Terminal Window
- Type **hello**

TotalView removes the dialog box and the thread's state should be halted (status T in the Threads Pane and <Trace Trap> in the Process Window header).

Step 9: Breakpoints

Click on **`#26` (the line number)** on the left of the Source Pane

You've now set a breakpoint. A bright red stop icon appears over the line number indicating that TotalView has set a breakpoint.

Press the **Go** icon  in the toolbar

You can tell that the process is running by examining the top title bar. Note that TotalView does not alter the PC arrow until the process has stopped. That is, until it stops, the PC arrow indicates the last stopped location.

Type **hello** within the Terminal Window

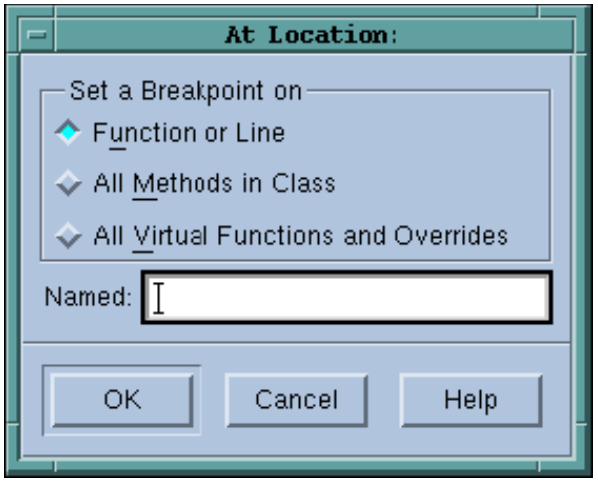
TotalView now halts your program at the breakpoint. The process status is At Breakpoint.

Note
When debugging a program, you'll often want the program to execute until it reaches a particular line. The way you tell TotalView to stop the program's execution is to set a breakpoint, which is a stopping point. The easiest way to set a breakpoint is to click on the line number.

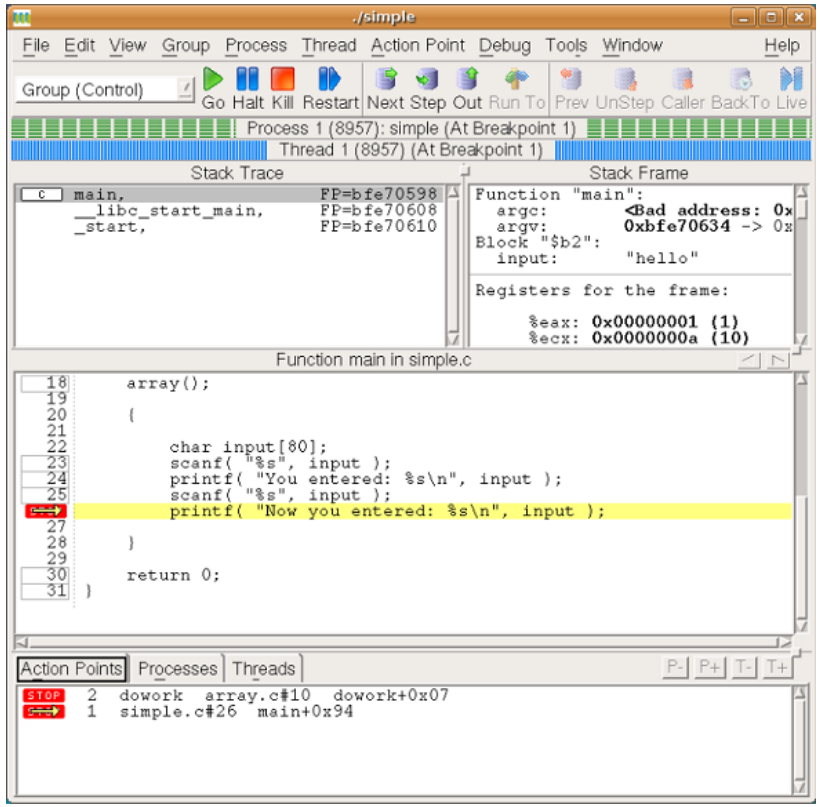
Step 10: Breakpoints: At Location

Create a breakpoint using the At Location dialog box.

- Select the **Action Point > At Location** command
- Type **dowork** in the displayed dialog box
- Press **OK**



You should now see two breakpoint icons in the Action Point Tab. The PC arrow in this pane indicates the breakpoint at which the thread is stopped.



Dive on the newly created breakpoint—this is the one not having the PC arrow

This focuses your Source Pane at that breakpoint location.

Diving on breakpoints is another way that you can navigate to different locations in your program. You could use this as a way to bookmark places you refer to often in your source.

- Select **File > Exit** to exit TotalView
- Select **Yes**

Notes

- The At Location dialog box lets you set breakpoints on all methods of a class or all virtual functions. This is handy for C++ applications but that is beyond the scope of this lab.
- If you left-click on an icon in the Action Point Pane, the icon will dim because you will have disabled the breakpoint. You can re-enable it by left clicking a second time.
- If you right-click on an icon, a context menu appears. Among other choices, you can now delete or enable/disable the breakpoint.
- The square boxes around line numbers also provide information. If they are in bold, there is more than one code address associated with the breakpoint.
- The Action Points > Save All command saves your breakpoints in a file. This file can be loaded in a later TotalView session. By default, TotalView saves the breakpoints in a file in the directory containing the executable. The next time you debug that executable, TotalView automatically loads these breakpoints.
- Use preferences to control preference behavior. Select File > Preferences, and click on the Action Points Tab. From here you can tell TotalView to automatically load action points when it starts and automatically save them when it exits.

END OF LAB 1

Lab 2: Viewing, Examining, Watching, and Editing Data

This lab shows many of the ways in which TotalView displays data values.

Expected time: 45 minutes

Step 1: Preliminary Steps

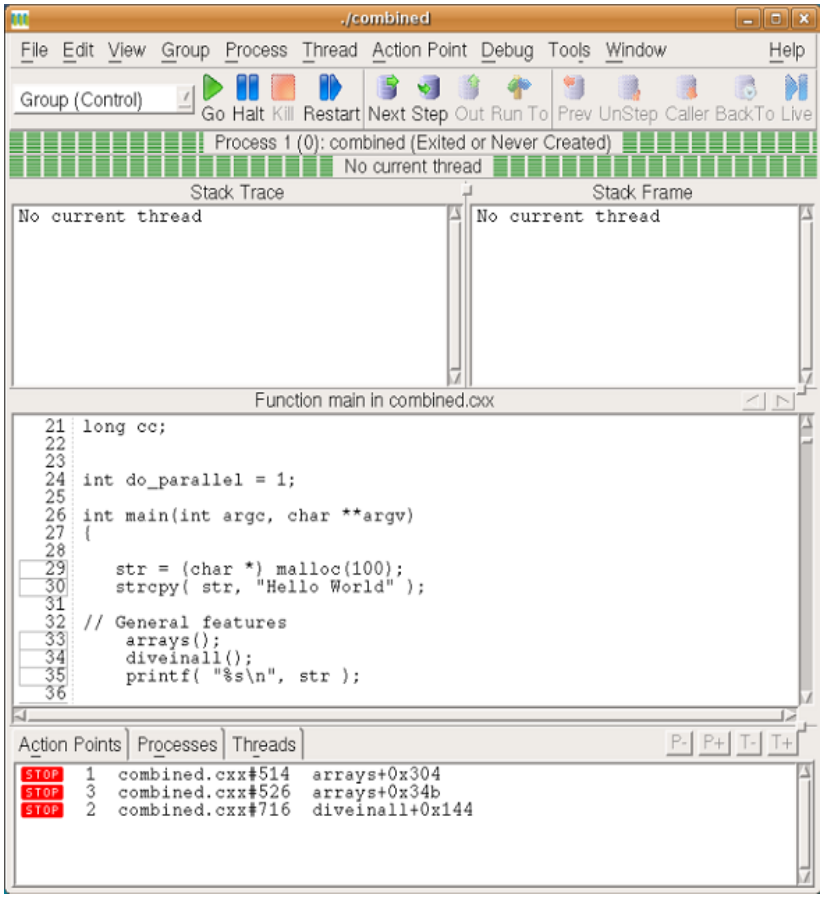
In a Terminal Window:

- Change directory

```
cd $LABS
```

- Start TotalView

```
totalview ./combined -a Thanks for attending
```



Step 2: Looking at Data

There are four ways to look at data. This step looks at three of them. The next will look at the fourth.

Stack Frame Pane (Method 1)

Press **Go** 

TotalView should halt the program at a saved breakpoint inside a function named `arrays`.

Observe the Stack Frame Pane in the top right corner of the Process Window.

Questions

1. What kind of information is displayed in this pane?

2. Some of the values displayed in the Stack Frame are bold while others aren't. What does the bold text mean?

3. What do the `Block` designations mean?

_____

Tool Tips (Method 2)

Place your cursor on `i` in the Source Frame and hold it there for a couple seconds

TotalView displays the value of `i` in a Tool Tips popup.

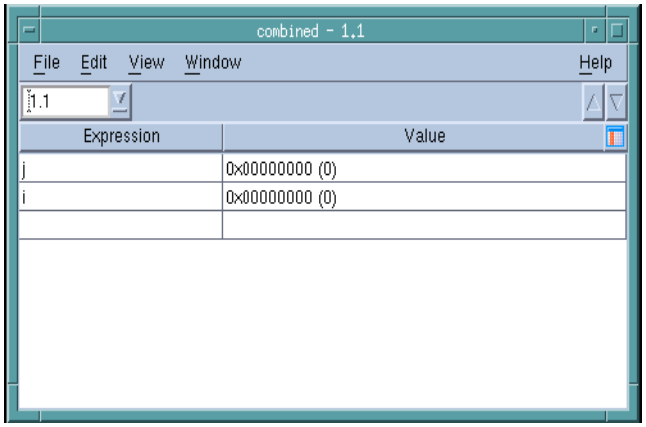
- On line 512, select (i.e., highlight)
  `start + j*step`
- Place your cursor over the selected text for a couple seconds

Note that Tool Tips work with simple expressions as well.

Expression List Window (Method 3)

You can think of an expression list as a kind of watch list that shows value that you can easily keep an eye on.

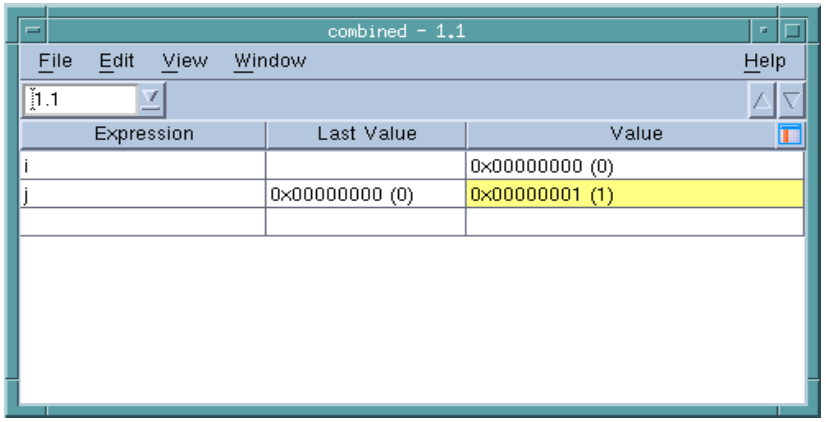
- Right-click on the variable `i`
- Select **Add to Expression List**
- Right-click on the variable `j`
- Select **Add to Expression List**



Press **Go** 

TotalView executes one iteration of the loop and again stops at the breakpoint.

TotalView updates the Expression List Window with the current value. It also highlights the value for `j`, indicating that the value has changed.



- Right-click on the column header
- Select **Last Value**
- Expand the column by dragging to see the whole value
- Click on the third row in the Expression field and enter `i + j + 5`

You can also enter expressions directly into the window.

- Select the `cylinder.volume()/cylinder.area()` expression in the Source Pane on line 513
- Right-click
- Select **Add to Expression List**

The Expression List Window can contain functions calls.

Note
Don't do this for functions that cause side-effects as this window is updated each time the program gets updated. If you don't want to re-evaluate an expression all the time, use Tools > Evaluate instead.

Click on the **X button** in the Expression List Window to close it

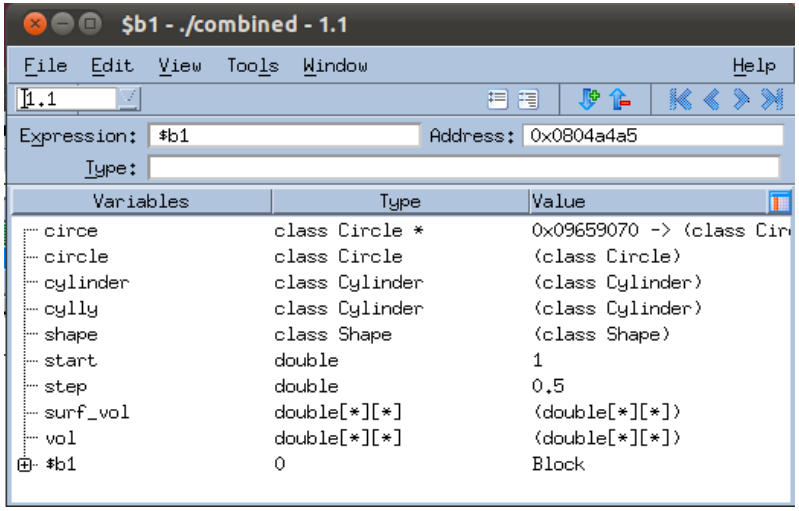
Step 3: Looking at Data (Part 4)—Variable Window

The Stack Frame Pane, Tool Tips, and Expression list are excellent for viewing and watching variables that have built-in types and that you don't need to examine in different ways, such as with a memory dump or as a different data type. These methods do not work with structures, classes, arrays, common blocks, or data types. To view this data, use the Variable Window.

You can open a Variable Window by diving any place you see a variable (sometimes an expression) or by using the View > Lookup Variable command, which is particularly good for global variables. You can dive by a double left-click, a middle-click, or by right-clicking and selecting Dive on the context menu.

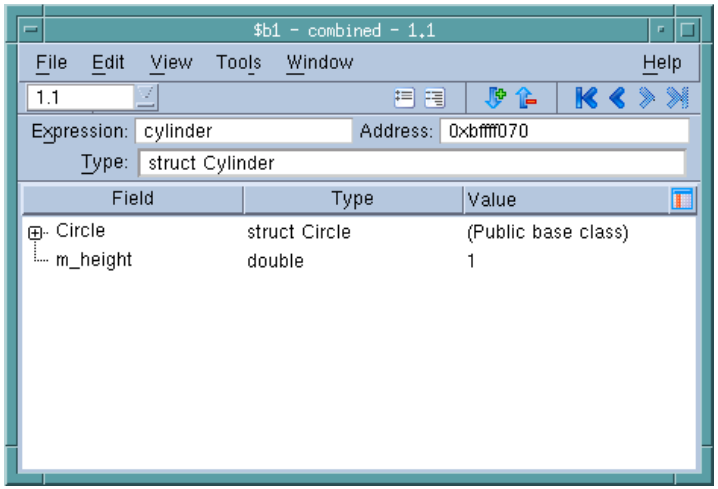
Dive on Block `$b1` – this is displayed in the Stack Frame Pane

TotalView opens a Variable Window which contains all the variables and blocks within `$b1`.



Dive on the `cylinder` variable within the Variable Window

You can dive on any field if the Variable Window is displaying something that does not have `scalar` type. Note that this shows you the contents of the `cylinder`, which has a type of `Cylinder`.



Q

4. When would you want to use the Expression List as opposed to a Variable Window or a Tool Tip?

Step 4: Examining the Variable Window

Part 1: Features

- Press the **X** button in the Variable Window to close the window
- Dive on the `cylinder` variable in the Source Pane

You can also dive on variables in the Source Pane, expressions in the expression list or any place else you see a variable.

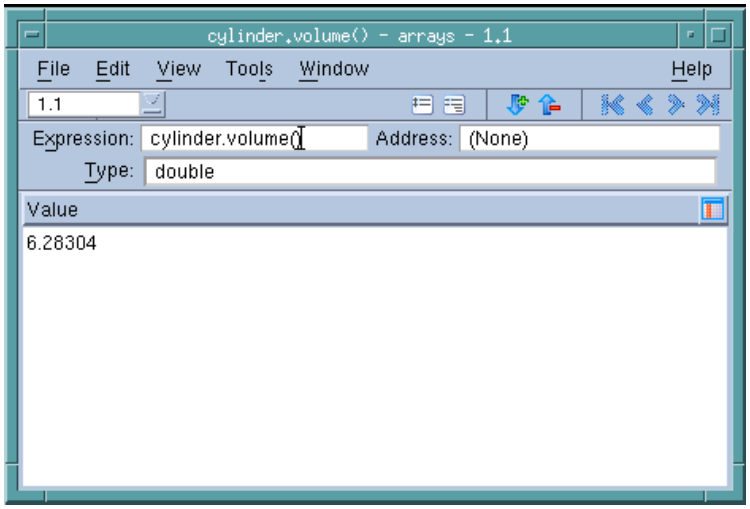
Let's examine the Variable Window a bit more.

- The  icon on the menu bar lets you expand and collapse the contents of a compound type.
- The  up/down icon on the menu bar lets you see more or less meta information about the data you are viewing.
- The dive/redive arrow icons let you undive and redive within the variable.
- The Expression field indicates the variable `cylinder`. You can edit this field and it can contain general expressions in the language your program is written in. For example,

In the Variable Window Expression field, type: `cylinder.m_height`

  TotalView now shows you the value of the `m_height` field in the object.
- You can even call functions here. For example,

Type: `cylinder.volume()`



Note
As with the Expression List Window, do not enter expressions that cause side-effects in this field as it will be evaluated each time the program is updated.

In the Variable Window, select the **Edit > Reset Defaults** command

This resets the window to the contents you originally dived on.

Select the **View > Expand All** command

This shows all of the base class.

Select the **View > Freeze** command

This command tells TotalView that it should not update the Variable Window. At a later time, you can compare these frozen contents against an updated Variable Window.

Part 2: A Second Variable Window

Dive on the variable `j` in the Process Window

TotalView displays a Variable Window containing the `j` variable.

- In the Action Point Pane at the bottom of the Process Window, click on the **Stop** icon for line 514, disabling the breakpoint and graying out the icon
- Select **line 510 (the line, not the number)** in the Source Pane
- Press the **Run To** button

TotalView executes the program to line 510.

The Variable Window containing `j` now shows a highlighted value, indicating that its value has changed. Note that the Variable Window is reporting a status of `stale`.

Right click on the column header and select **Last Value**

Questions

5. Why does the Last Value field report the value of `j` as 1 instead of 19?

   _____

6. What does `Stale` mean and why are the Expression List and Variable Windows reporting this?

   _____

Part 3: Evaluations

Click on the dimmed **Breakpoint** icon in the Action Points Pane

This re-enables the breakpoint.

Press **Go** 

The program executes until it reaches line 514.

The Variable and Expression List Windows no longer report that they are `Stale`.

- Go to the Variable Window you had frozen for the `cylinder` variable
- Select the **Window > Duplicate** command

The Freeze and Duplicate commands allow you to compare values at a later time. In contrast, the Highlight/Last Value features show you what last changed.

You can delete the breakpoint at line 514 in two ways:

- Click on the icon in the Source Pane, or
- Right-click on its number in the Action Points Tab, then select Delete from the context menu.

Try deleting the breakpoint

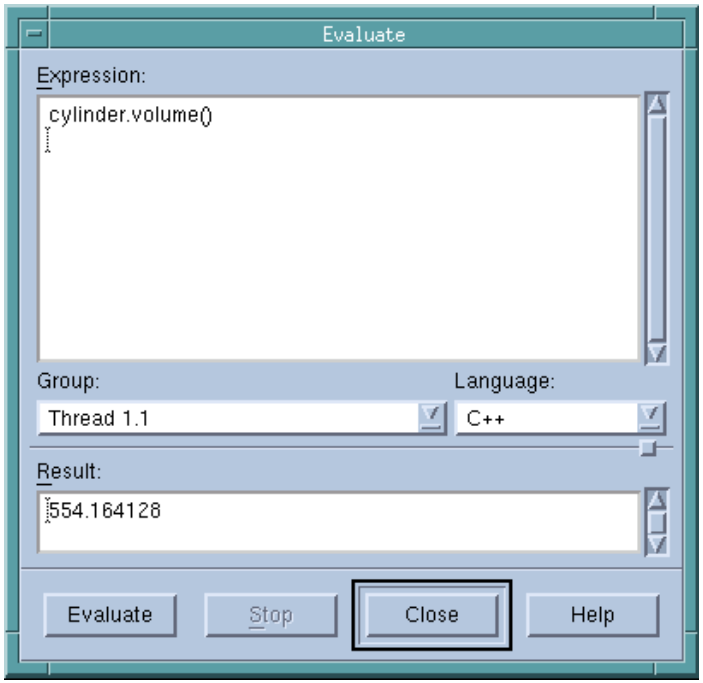
You can type more than just expressions in the expression field. You can type entire program fragments.

- Press **Go** to run the program up to line 526
- Open the **Tools > Evaluate** dialog box

Most language constructs are supported, including declaring variables of `non-object` type, `for`, `while`, and `do` loops.

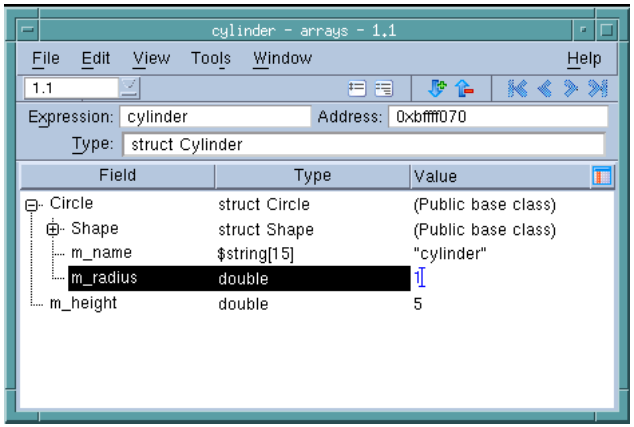
- Type `cylinder.volume()`
- Press the **Evaluate** button

Observe the result in the Result field.



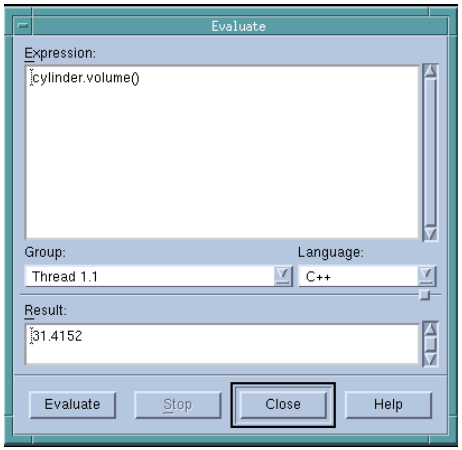
- Go to the unfrozen Variable Window containing the `cylinder` variable
- Click on the value field for `m_radius` – this is in the base class `Circle`
- Change the value to `1` and press Return

You could also press F2 to edit the field.



This edit changes the value in the target program. Check that this has occurred:

- Go back to your Evaluate Window
- Re-evaluate the expression you typed earlier



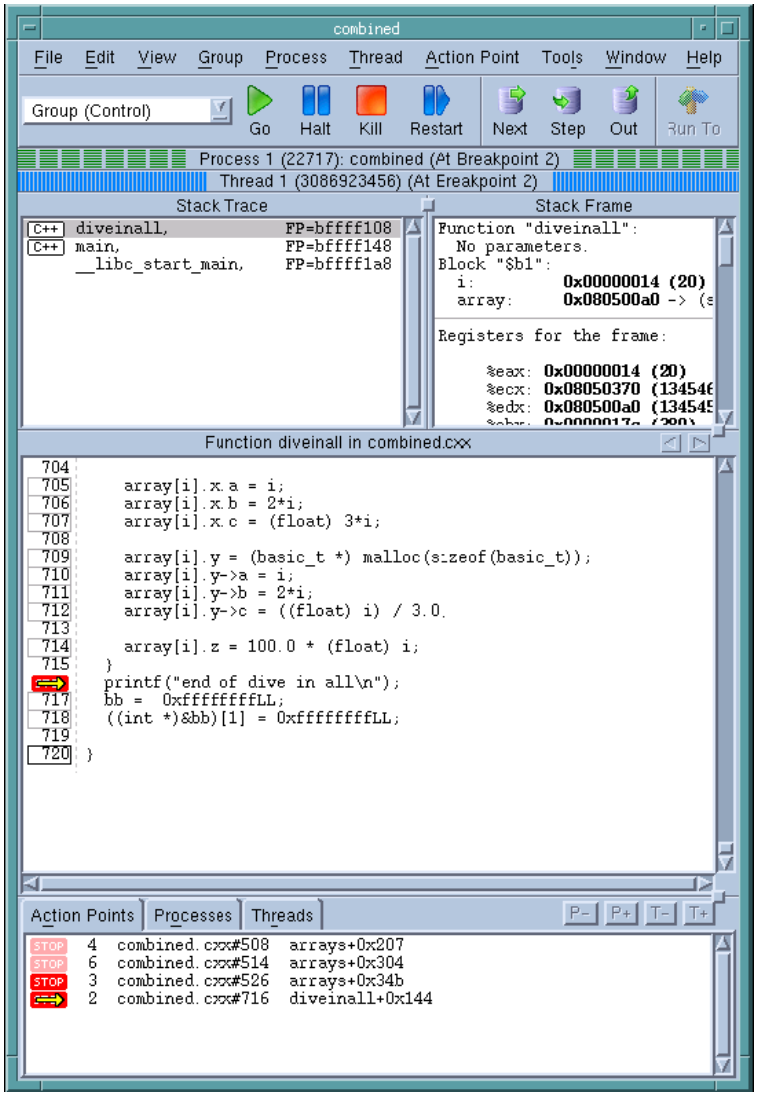
- Close the Expression List Window by pressing the **X** button
- Go to a Variable Window
- Select the **File > Close Similar** command – this closes all your Variable Windows

Step 5: Arrays

This step explores some of TotalView's array features and its typecasting ability.

Press **Go** 

TotalView runs up to the breakpoint in the `diveinall()` function.



Dive on the variable `array`

This opens up a Variable Window displaying this variable. Note that it has a type of `struct compound_t*`, which is a pointer to a `compound_t` structure. However, we know the variable to be an array of `compound_t` and not just a pointer to a single `compound_t`.

Question

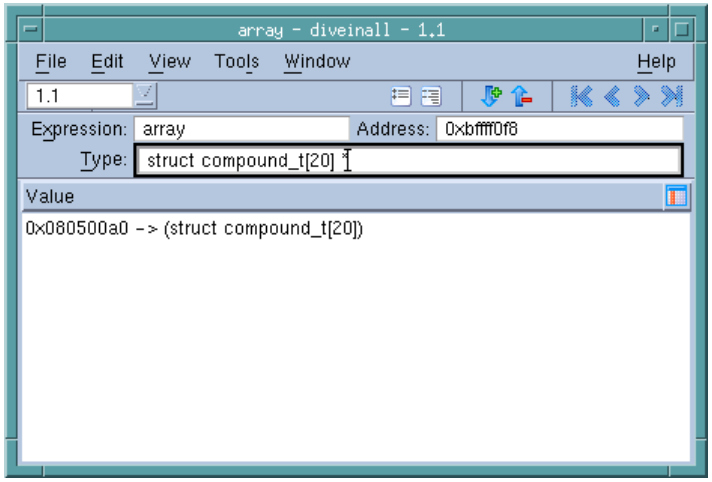
7. Why doesn't TotalView display this as an array?

A debugger cannot tell the difference between a pointer and an array as the semantics of the pointer are defined at runtime. This means that the compiler can't tell the debugger the size at compile time. However, you, the programmer, know when you are looking at an array, and TotalView has the ability to display your data the way you want. We call this typecasting.

TotalView types are read a little differently than they are in C and C++. TotalView reads types from right to left. To view `array` as an array:

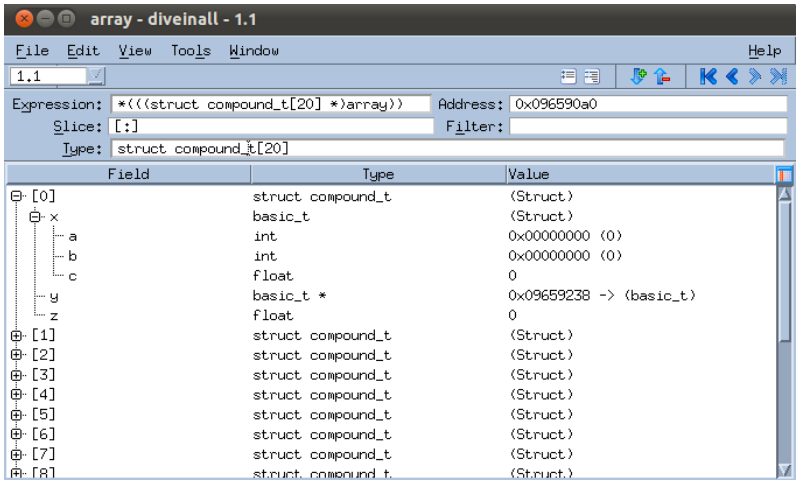
Change the type from
`struct compound_t*`
to
`struct compound_t[20]*`

TotalView now interprets the data at this location as an array of 20 elements of type `struct compound_t`.



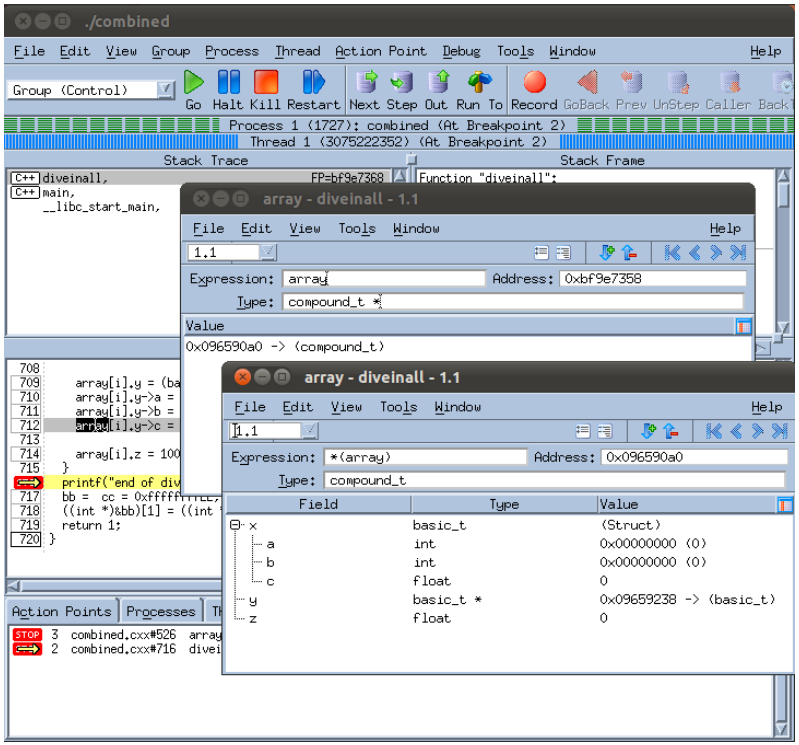
To dereference the pointer:

Dive on the **Value** field



- Close the Variable Window
- Reopen it by diving again

Another way to view a dynamic array, which might be easier to remember, is to first dive on the pointer to display a structure, which happens to be the first element in the array.

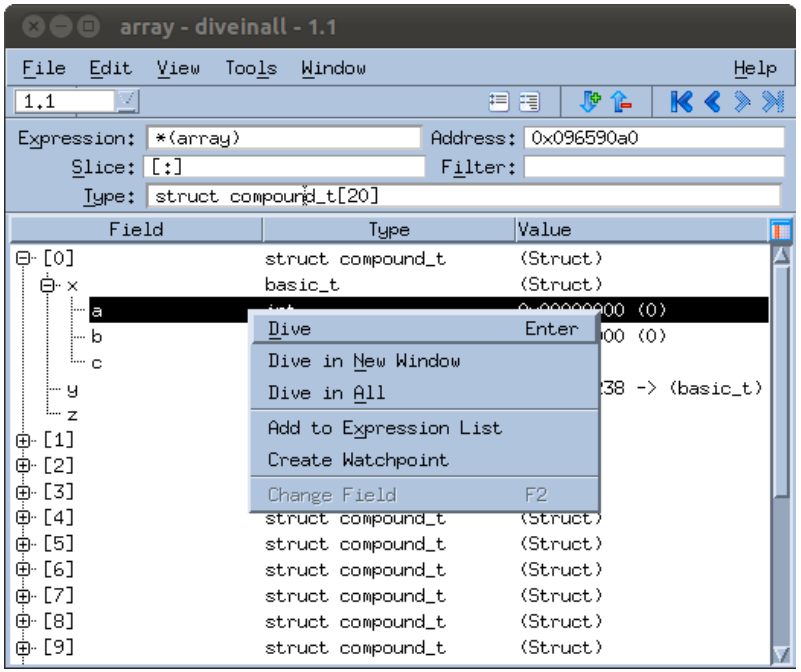


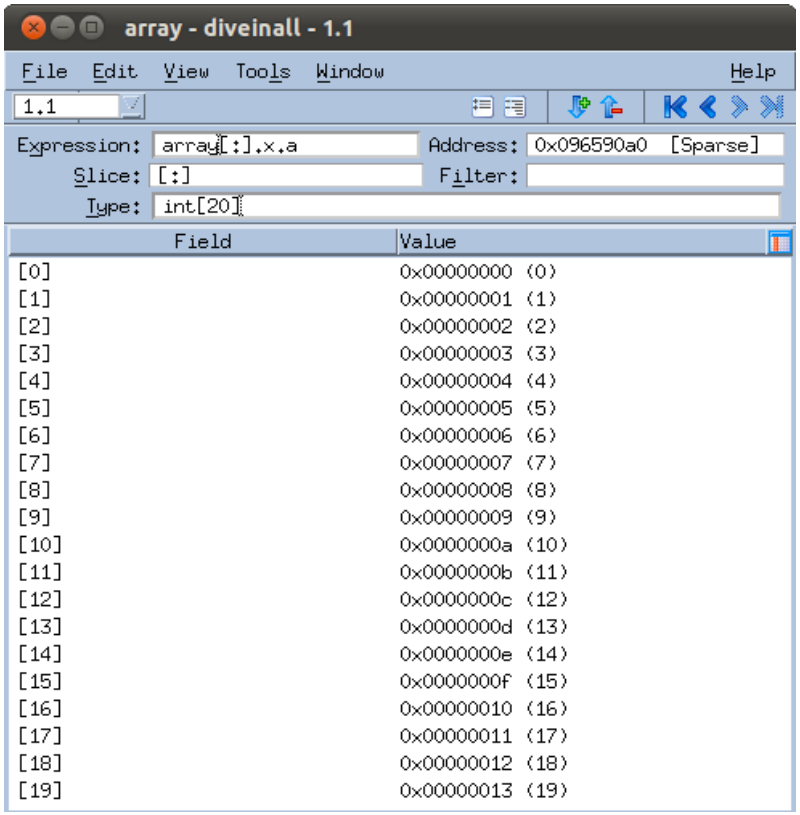
- Dive on the Value field
- Change the type from
  `struct compound_t`
  to
  `struct compound_t[20]`

One great thing you can do for arrays of structures, classes, and data types is to focus on one field in the object in every element in the array. This is called Dive in All.

- Expand out the first element `a`
  This is within the `basic_t` structure, which is within the `compound_t` structure
- Right click and select **Dive in All**

TotalView displays the field `a` within the structure as if it were an array.





Question

8. What does `Sparse` mean in the Address field?

   _____

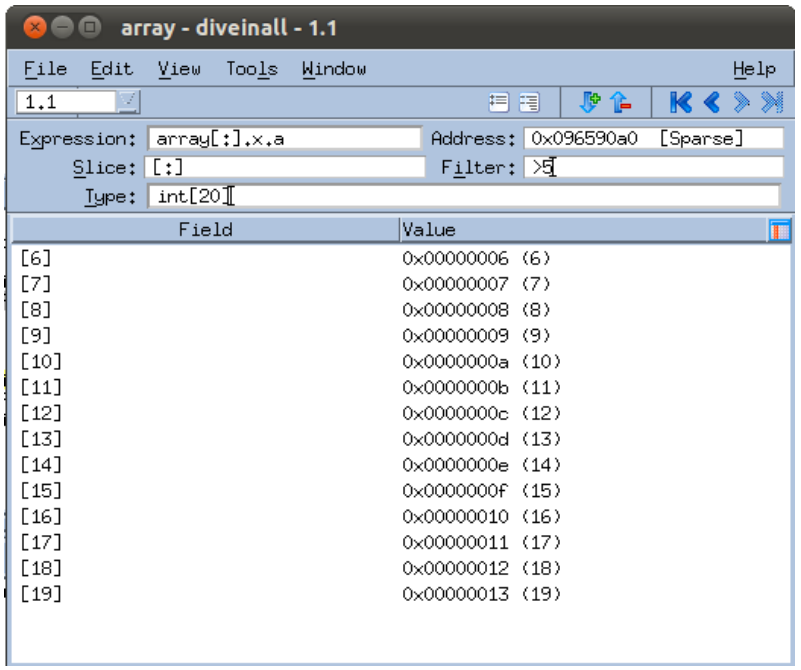
Click on the **Value** column header

This sorts the array in descending order. Clicking a second time sorts the array in ascending order.

Type the following in the Filter field:

```
>5
```

This is short for `$value > 5`. TotalView now shows you all elements greater than 5.



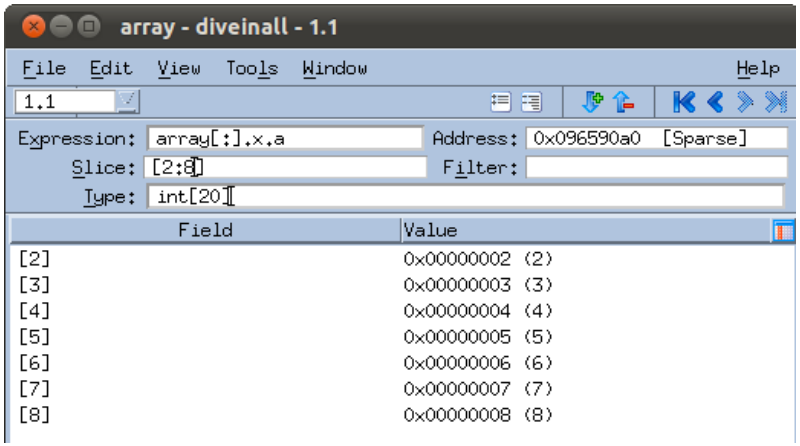
- Type **F1** to open Help
- Click on the **Filter** field

Observe other things you can filter on, such as nans and infinities.

- Delete what you typed in the Filter field
- Type the following in the Slice field

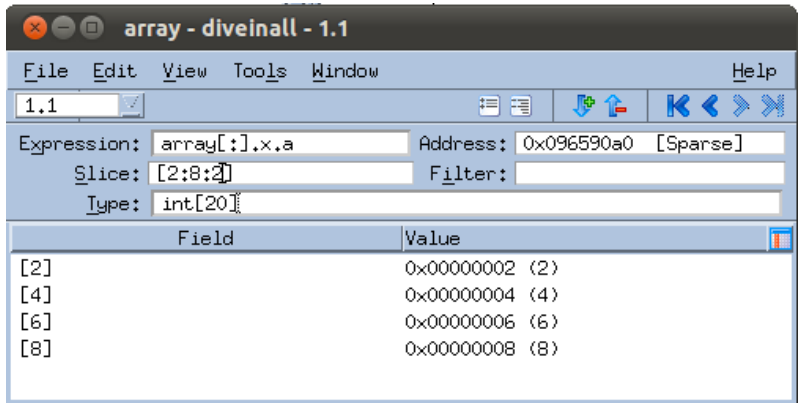
```
[2:8]
```

This slices your array and shows you array elements 2 through 8.



Edit the Slice field to `[2:8:2]`

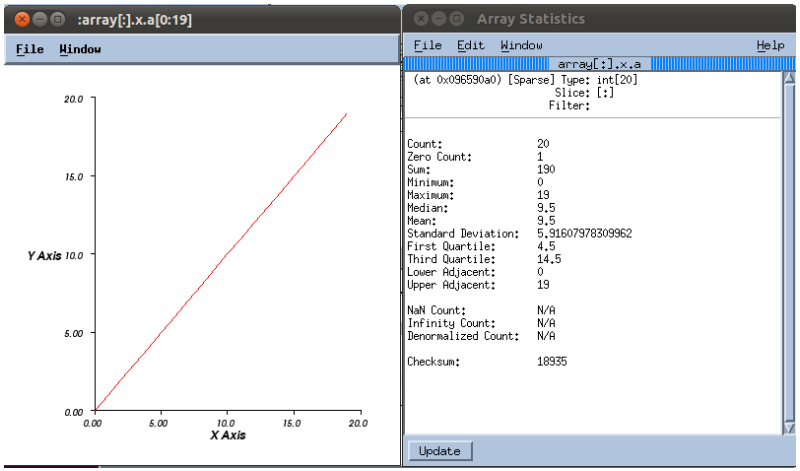
The integer after the second colon indicates the stride of the array. For example, “2” tells TotalView to display every second element in the range [2,8].



Delete what you typed in the Slice field

If you do not delete the slice (or filter), TotalView applies future actions to what is being displayed. So if you add a filter, a slice only displays the elements that meet both the filter and slice criteria, and visualizing or generating statistics on an array will only generate statistics or visualize based on the portion of the array displayed.

Select the **Tools > Visualize** and **Tools > Statistics** commands and observe what happens

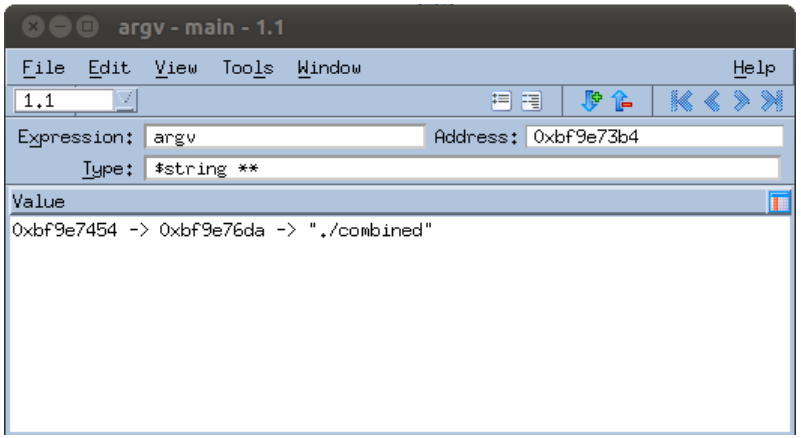


Select the (stepping command) **Out** button

TotalView runs the program out one stack frame.

Dive on `argv`

Here, TotalView shows `argv` to be a pointer to a pointer to a string.

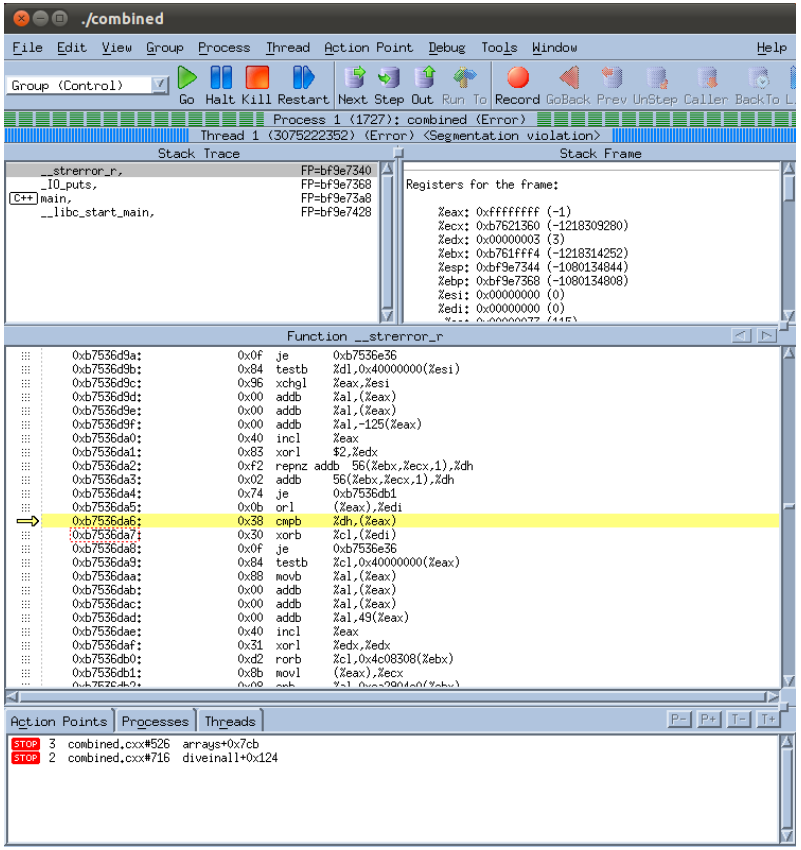


Challenge
Do what is needed to change the variable to display the arguments to your program.

Step 6: A Crash Problem

Press **Go** 

Your program continues executing, and then it should crash with a Segmentation Violation. Note the status in the status bar.



Let's try and figure out what happened.

Click on `main()` in the Stack Trace Pane to focus on the program's source

The program crashed after calling `printf()`. Let's investigate the argument passed to it.

Dive on the `str` variable

The variable is a pointer to a string, but the pointer's value is `0xffffffff`, which is not a valid address. TotalView reports `Bad Address` whenever the program tries to access a memory region in which the operating system won't allow access. When your program tries to read this address, it crashes with a segmentation violation just as this program did.

It is often helpful to look at the raw memory surrounding a corrupted memory region.

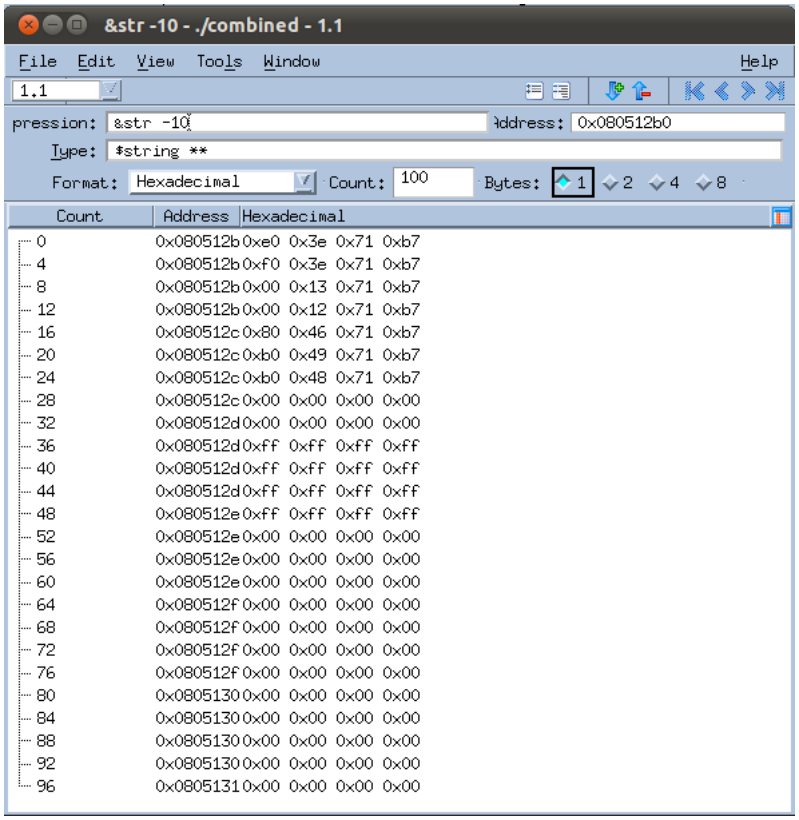
Move the pointer's address:

Change the Variable Window's expression from `str` to `&str -10`

This backs the base address in the Variable Window up by 10 words.

- Select the **View > Examine Format > Raw** command
- Change the Columns field to `4` and the Count to `100`

The original `str` pointer is at index `40` in the display. Observe that you have the same value at index `36`. This is a clue. It appears that what was writing into the address at index `36` is also writing into index `40`.

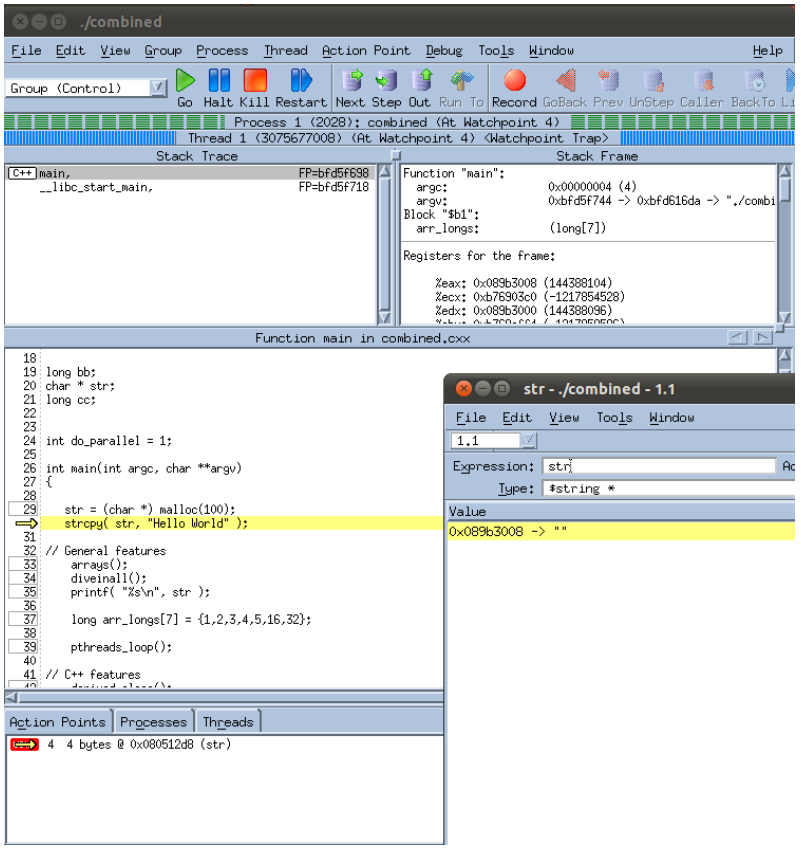


Here's how you can find out what's happening.

- Select the **Action Point > Delete All** command to delete all of your breakpoints
- Select the **Edit > Reset Defaults** command to reset your Variable Window
- Select the **Tools > Create Watchpoint** to plant a watchpoint
- Click **OK** after the dialog box appears

- Press the **Restart** button to restart your program

TotalView halts the program when it first stores a value into the pointer. At this time, the program is behaving correctly.



If you continue the program by pressing the Go button it will now halt when that memory location is overwritten and you have found your bug.

Questions

9. When will a watchpoint trigger?

   _____

10. What precautions do you need to take when planting a watchpoint on a local variable?

    _____

END OF LAB 2

Lab 3: Examining and Controlling a Parallel Application

TotalView 8.3 introduced a new way of launching MPI applications. This new way of launching is designed to be easy to use and to be able to work with any MPI implementation. The old way of starting, known as 'classic launch' is still accessible and needed in a few circumstances. The purpose of this laboratory is to familiarize you with both methods, and with TotalView features that will assist you in debugging your MPI applications.

Expected Time: 45 minutes

Step 1: Start-up (new launch)

- Change directory to `$LABS` by typing
  `cd $LABS`
- Type: `export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH`
- Start TotalView without any command line options
  `totalview`

This starts TotalView and it displays the New Program dialog box.

- Type the following in the Program field
  `./demoMpi`
- Click on the **Parallel** Tab 

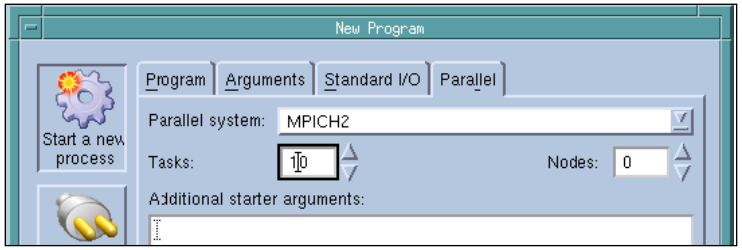
This is where you tell TotalView what MPI you are using, and what arguments you want to pass to the starter program or script (for example, `mpirun`, `mpiexec`, etc.).

Click on the **Parallel System** list control

You will see several options.

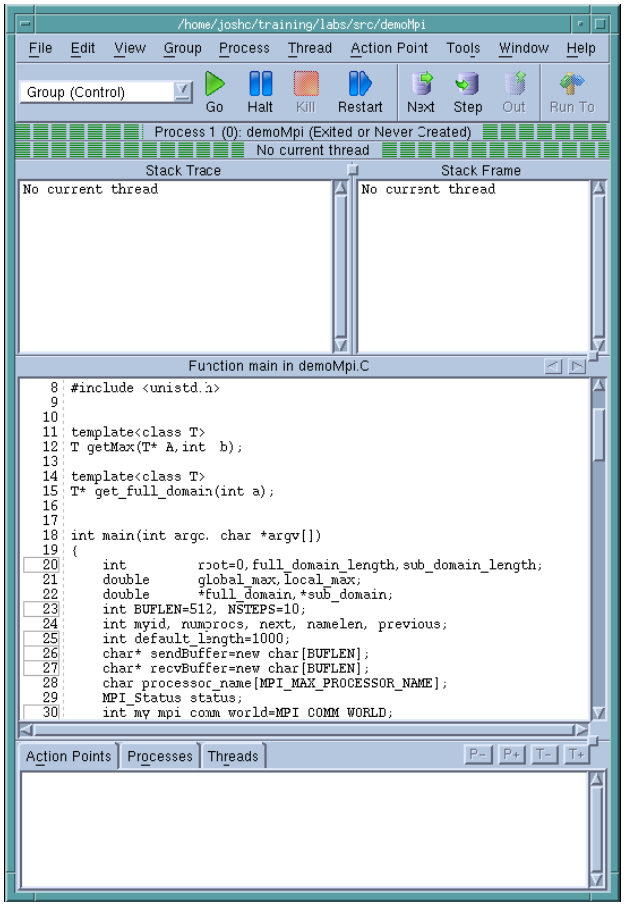
- Select `MPICH2` from the list
- Edit the Tasks field to `10` – this indicates 10 processes

Do not change other fields.



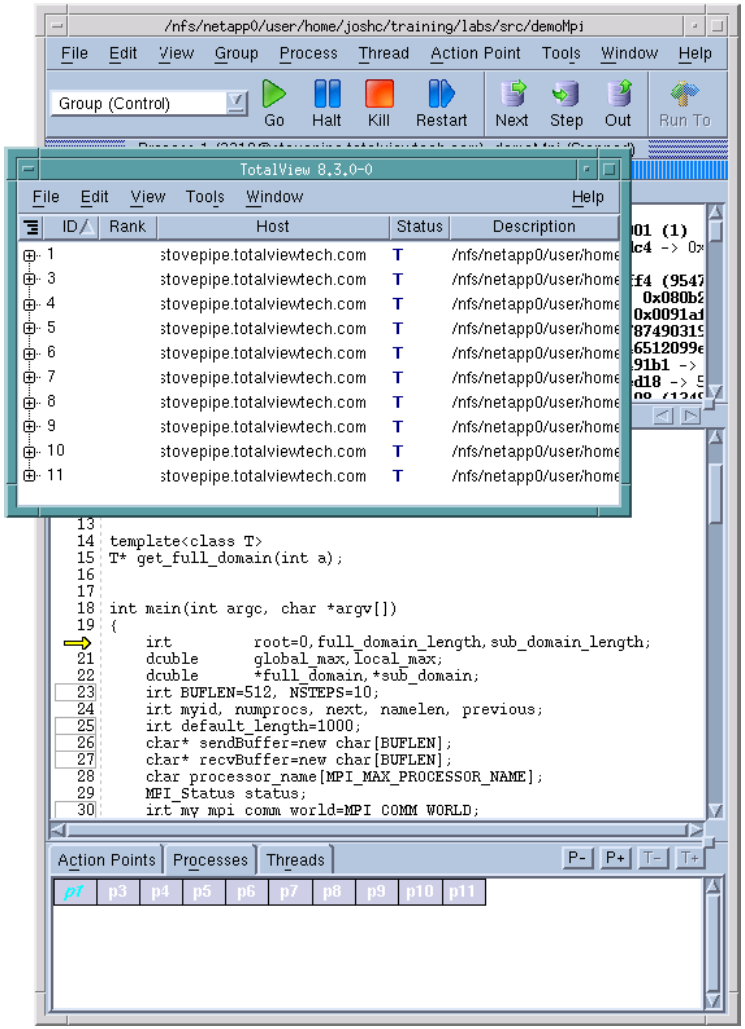
Press **OK**

TotalView opens a Process Window focused on the `main()` function in your MPI application.



Press the **Next** button

TotalView steps into the first line of `main()`, and your Root Window as well as your Processes Tab in the Process Window fill up with processes. This launch method lets you debug your job prior to the processes calling `MPI_init` — this is not possible using classic launch.



Notes

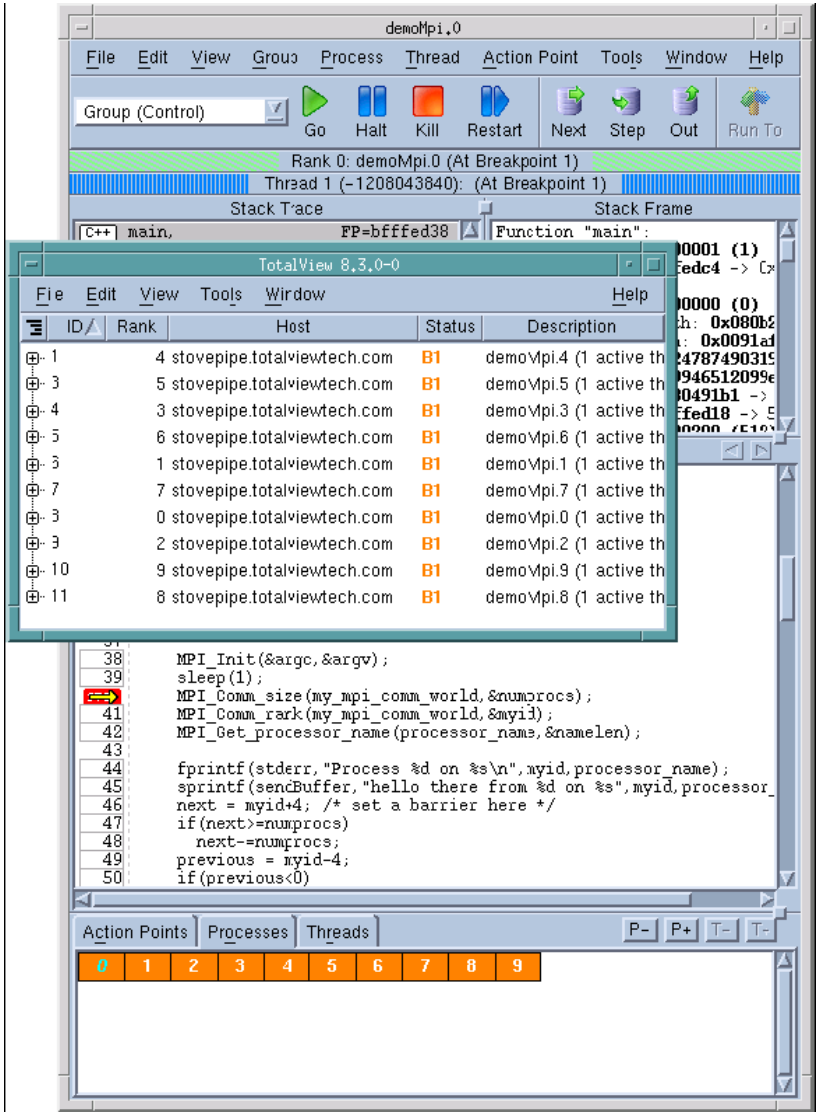
Look at the Root Window.

- The window has one line for each process. This line contains the host name, the status for the process, the name of the process, as well as how many threads are within the process. The ID column is the debugger ID for the process.
- You can sort the Rank, Host, and Status columns. This can be particularly helpful when debugging a job at scale and you want to find a process located on a specific host or a process (or set of processes) that is in a particular state.

- Set a breakpoint on line 40
- Press **Go** 

Observe that:

- All the processes get halted at the breakpoint.
- The Root Window now shows the ranks of the processes as do the nodes in the Process Tab in the Process Window.
- The status `B1` in the Root Window indicates that all the processes are halted at Breakpoint 1.



Click on the **Rank** column header

This sorts the processes in the Root Window.

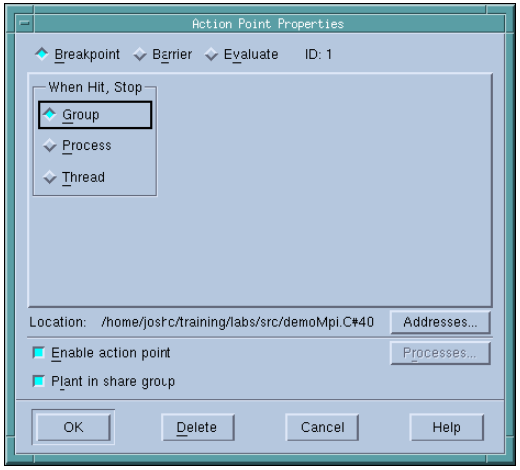
Tip: Sorting in the Root Window can be helpful when debugging at scale for locating processes in a particular state or for finding processes on a particular node.

Question

Why are all ranks stopped at exactly the same point in the program? Is that a coincidence?

_____

- Right click on the **Breakpoint** icon
- Select **Properties**
- Change the "When Hit, Stop" property from **Process** to **Group**



Press the **OK** button

This closes the Action Point Properties dialog box.

- Press the **Restart** button in the Process Window
- (If a dialog box opens, press **Yes**)

This restarts the job.

Question

Note that only a subset of the processes is halted at Breakpoint 1. Why is this?

_____

Notes

- You can change the **When Hit, Stop** property of a breakpoint—which defaults to **Process**—by selecting the **File > Preferences > Action Points** command.
- You may have noticed that your Process Window did not focus you on a process that was halted at a breakpoint. This is because TotalView tries not to steal keyboard focus; it will not focus to a different process when another process hits a breakpoint. You can change this preference by going to the **File > Preferences > Action Points** and checking the **Open Process Window at breakpoint** entry.

Step 2: Process Navigation

The Process Window focuses on only one thread within one process. However, many commands can act on a set of processes. This section gives you three ways to focus on different processes (there are other ways as well):

- Click the **P+/P-** buttons on the lower side to cycle through processes in a job
- Dive on a node in the Process Tab to focus on a particular process
- Dive on a row in the Root Window to focus on that process

- Go to the Root Window
- Select a process that is not halted at a breakpoint
- Dive on it to focus on it

- Go to the Root Window
- Select a process that is halted at a breakpoint
- Right click on it
- Choose the **Dive in New Window** command

The Dive in New Window command allows you to have more than one Process Window open, which means that you can see more than one thread/process.

Close the Process Window that is not focused on a process that is stopped at the breakpoint

Step 3: Multi-Process Control

TotalView allows you to execute process control commands (**Go, Halt, Next, Step, Out, Run To**) on a set of processes, on a single process, or even for a single thread. Here are a few ways to do this:

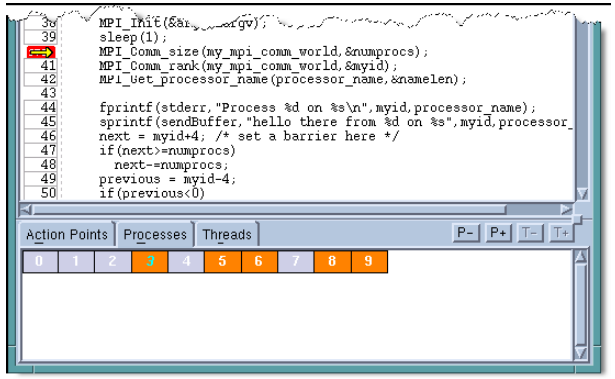
- Changing the focus control of the process control buttons.
- Using the Group, Process, and Thread menus.
- Keyboard accelerators (Group and Process menu options are annotated with the applicable accelerator).

The process control buttons above the Stack Trace and Stack Frame have a focus that is controlled by the combo box to their left. The default setting is **Group** (Control), which essentially means all the processes in the MPI job.

- Set the focus control to its default "**Group** (Control)" setting
- Press **Go** 

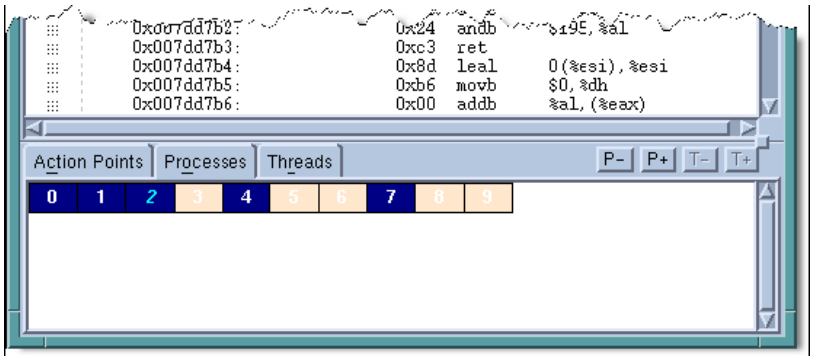
- Expand the list control on the left to see **Group (LockStep)**

The Processes Tab shows group membership by highlighting the processes which belong to the group.



- Go to the **Processes Tab**
- Dive on one of the processes not halted at Breakpoint 1

You can see TotalView selects a different set of processes.



Press **Go** 

You can see that a subset of the processes is still halted at Breakpoint 1. The important thing to note here is that the set of processes that were halted at the breakpoint before you issued the **Go** command are still at the same point because they were not continued.

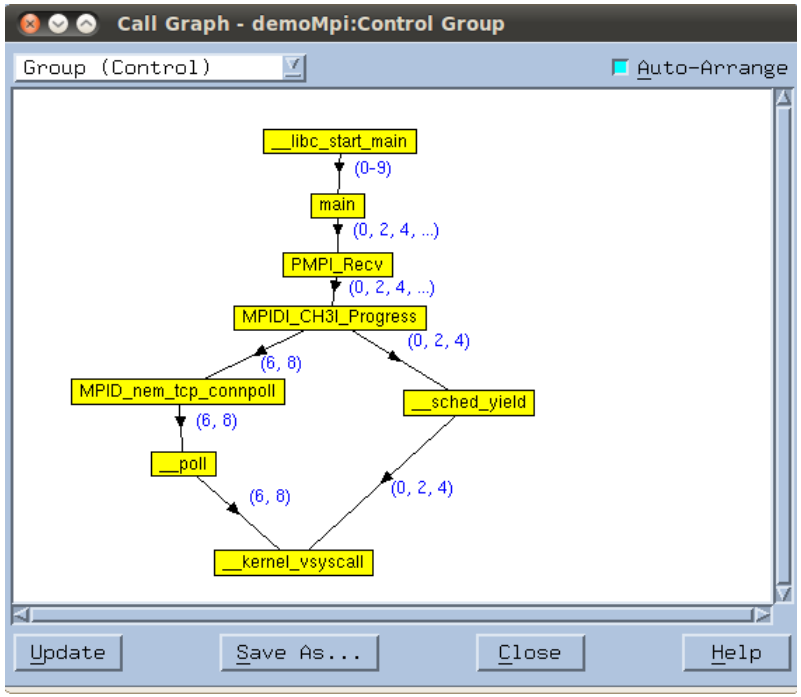
- Press the **Restart** button
- (If a dialog box opens, press **Yes**)

This restarts your application.

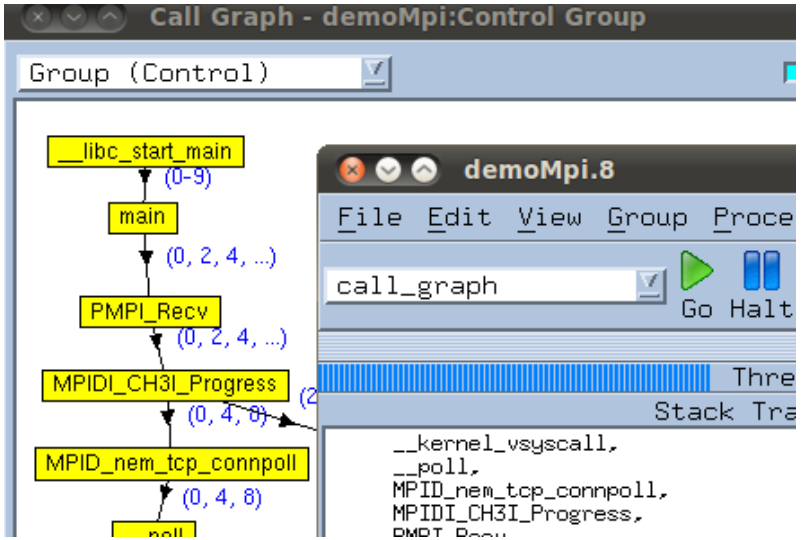
Note that the Kill and Restart buttons are not affected by the focus control; they always apply to the entire control group.

Select the **Tools > Call Graph** command

The Call Graph shows you a graph of all the stack traces in all the threads in all the processes in the control group, which can be filtered by the combo box on the top of the window. This window is often helpful for debugging at scale because it shows you all processes graphically. In this simple example, all processes run through to *main.*
To make the Call Graph more interesting, disable the breakpoint at line number 40, and set a breakpoint at at line 60. Then select the GO button and halt the program execution and examine the Call Graph. The following Call Graph will likely look a little different but there should be similarities in the nodes and edges of the graph.



TotalView creates a process group containing the processes which are within that function. Note that after diving on a node, "`call_graph`" is added to the focus control menu.



Delete the breakpoint at line number 60, and enable the breakpoint at line number 40. Select the Restart button.

Questions

What do the edges in the graph represent?

What do the nodes in the graph represent?

_____

Dive (double click) on a node in the graph

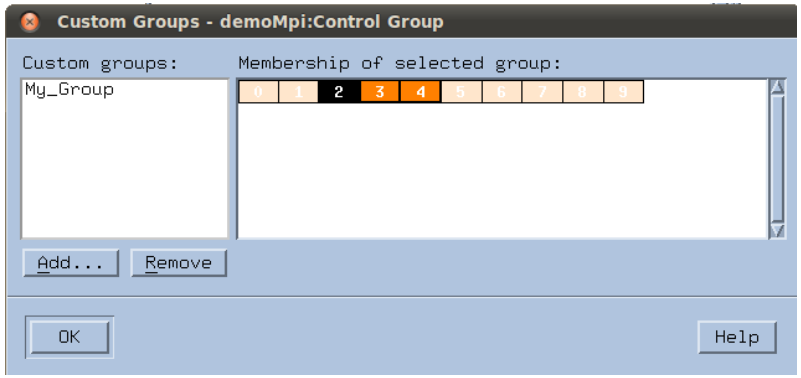
Delete the **Breakpoint** at line number **60**, and enable the **Breakpoint** at line number **40**.
Select the **Restart** button.
The program should stop at the first **Breakpoint**
Select the **Group > Custom Group** command

This dialog box lets you create and edit process groups.

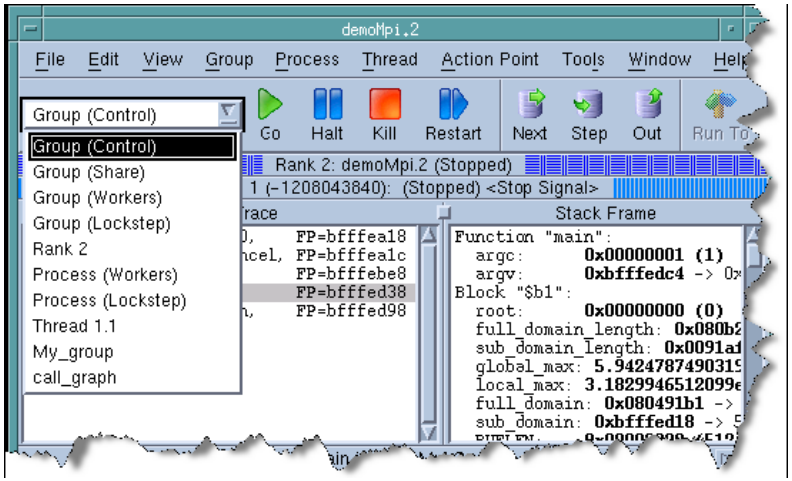
Click the **Add** button to create a new group

This creates a new group.

- Enter `My_Group` as the name
- Select a subset of ranks. (i.e. 2, 3 and 4)



- Press the OK button
- Answer Yes to apply changes to My_Group
- Change the focus control in the Process Window to My_Group
- Press Go 



The processes in `My_Group` progress up to the breakpoint.

Question

What do you expect to happen if you now select the `call_graph` group in the focus control and continue the process?

_____

- Change the Property of the breakpoint you have set back to **When Hit, Stop Process**
- Press the **Restart** button

All your processes should now be halted at the breakpoint.

If your Process Window is not already focused on `Rank 0`, focus there and change the focus control to `Rank 0`.

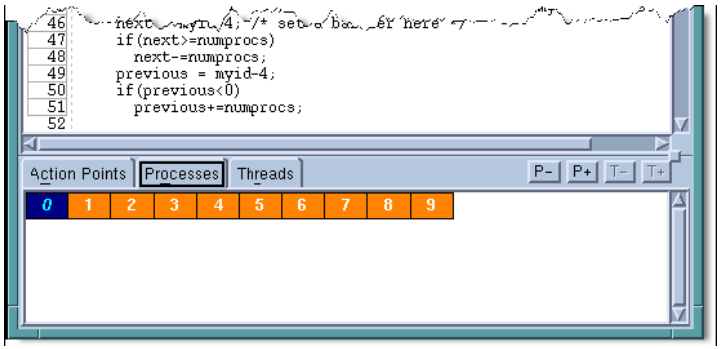
Now when you press the process control buttons, they will only act on the `Rank 0` process. This can be helpful if you only want to control and query one process at a time. It is extremely helpful when you are debugging a race condition.

Tips

- You can force determinism into a race condition by controlling processes and threads independently because this makes debugging easier.
- When debugging jobs at scale, it is recommended that you single step individual or a subset of the processes in your job rather than single stepping the entire job (Control (Group)).

Press the **Next** button twice

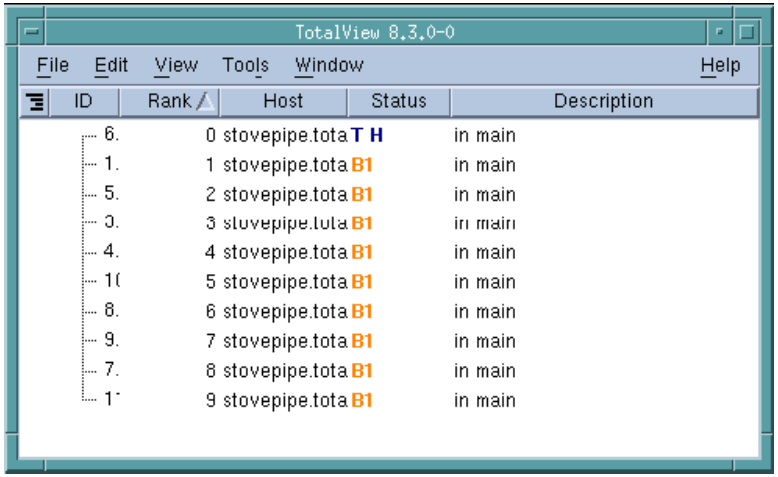
The process steps two line numbers. All other processes still show orange in the Process Tab because they remained at the breakpoint.



There may be times when you want to hold a Process or Thread. Holding a process or thread means that the Process or Thread will remain halted until you release it, regardless of what process control command you issue.

Select the **Process > Hold** command to hold Rank 0

Observe that the process status in the process status bar is `Held`. In the Root Window, this is indicated by an `H`.



With the focus control still set to `Rank 0`, press **Go** 

You should get a warning message that states that this is not possible.

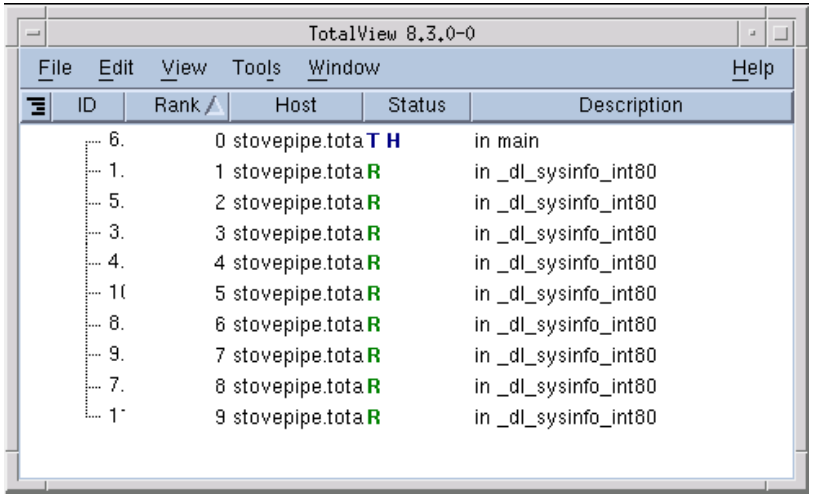
Question

What do you expect to happen if you change the focus to the control group and press **Go**?

_____

- Change the focus control to Control (Group) if you haven't already done so
- Press **Go** 

This continues all the processes except the one that is held.

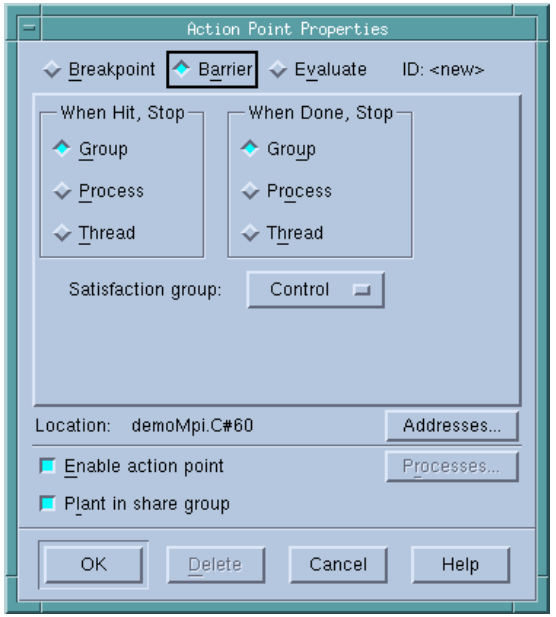


Scroll down to line 60

Observe the call to `MPI_Barrier` on `COMM_WORLD`.

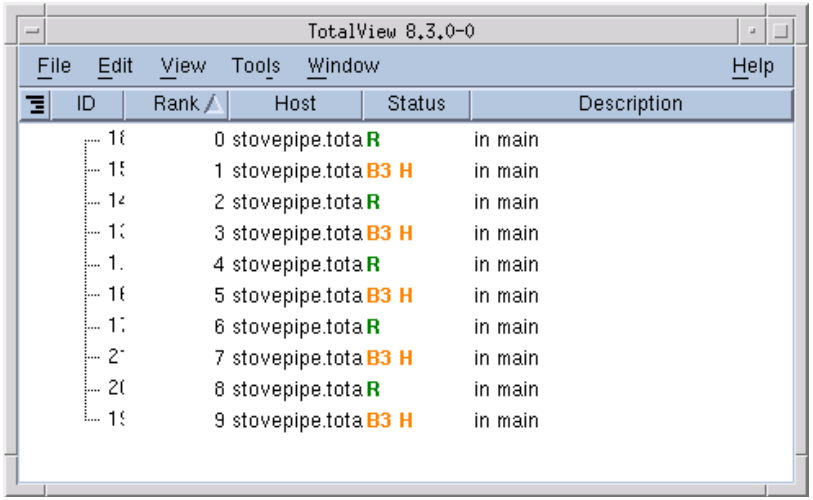
TotalView has the notion of a barrier that is implemented as an action point called a Barrier Point.

- Click on line 60 as if to set a breakpoint
- Right click on the **Breakpoint** icon
- Select **Properties**
- Click the **Barrier** check box at the top



- Set the When Hit, Stop option to **Process**
- Press **OK**
- Delete the breakpoint you previously set on line 40
- Press the **Restart** button

Your Root Window shows that half the ranks are held at the barrier point and the other half aren't.



Dive on one of the held processes

TotalView reminds you that the barrier point was set on a line that calls `MPI_Barrier` on `COMM_WORLD`. The fact that all processes are not reaching the barrier point indicates that the job is deadlocked.

Questions

What is a common cause of a deadlock in an MPI application?

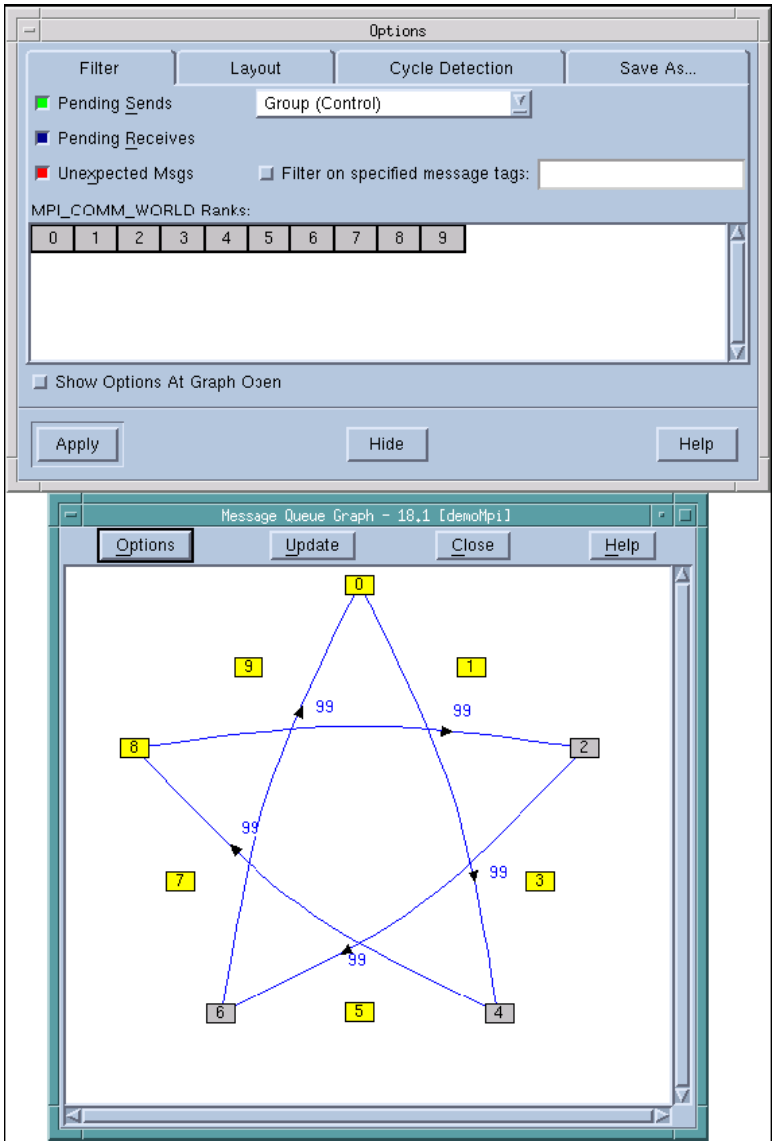
What features does TotalView provide to help you with this type of problem?

_____

Step 4: The Message Queue Graph and Viewing Data across Processes

- Select the **Tools > Message Queue Graph** command
- Press the **Options** button

TotalView displays a key to the messages.



Questions

What does the Message Queue Graph tell you?

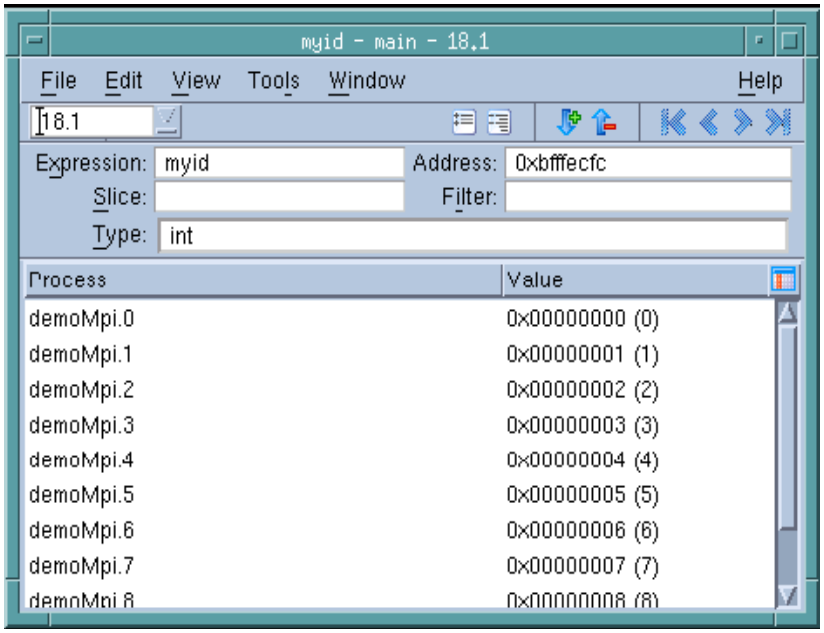
Does the Message Queue Graph show you all messages ever sent or just the pending ones?

_____

Choose different Message Queue options and observe what occurs

These options let you filter the information displayed and the layout of the graph, save the graph, and perform cycle detection, which is helpful if there are lots of processes with lots of messages.

- Press the **Halt** button
- Click in the Stack Trace Pane if you are not already focused on a process showing the program's source
- Right click on the `myid` variable
- Select **Across Processes**



If you already have a Variable Window open on an expression, you can do the same thing by using the View > Show Across > Processes command.

For SIMD applications, TotalView can display the contents of a variable across all processes.

Step 5: Classic Launch

The remainder of this lab presents a second way of launching TotalView, which is called classic launch. There are a few reasons for using classic launch:

- You are using a BlueGene, Cray XT3, or a SiCortex system.
- You want to use the TVD subset_attach feature. This feature allows you to attach to a subset of a job either for scalability reasons or to limit your license usage.
- You want to detach from a running job and later reattach to it.

If you used TotalView before version 8.3, you were using classic launch. This mechanism requires the MPI to collaborate with TotalView by storing information about how to attach to a job in symbols within the MPI program itself. Consequently, many MPIs require special build options to work with TotalView.

For example, with MPICH2, you must use the `-enable-debuginfo` and `-enable-totalview` configuration options.

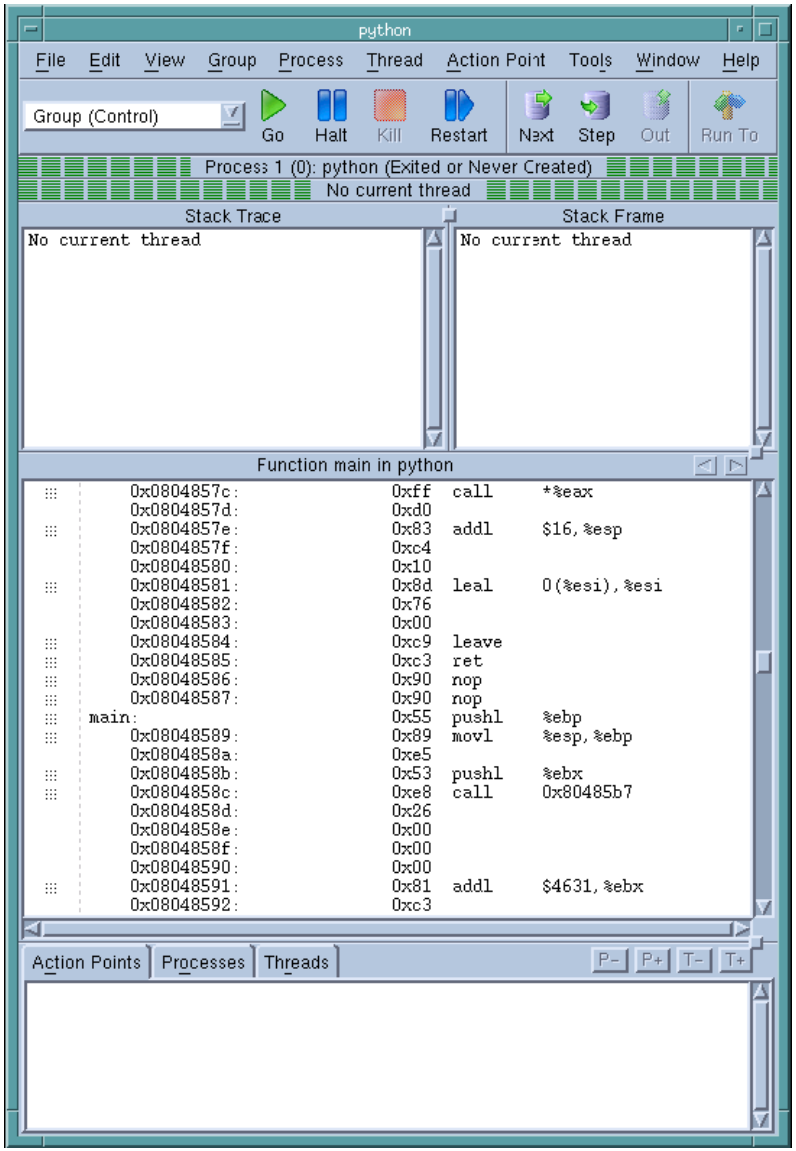
Within classic launch, to enable TotalView on an MPICH2 job:

- Type the following command:
- totalview mpiexec –a –n 10 $LABROOT/demoMpi

The Startup Parameters – mpiexec menu box opens

- Press the **OK Button**
- Press **Go** 

This starts TotalView on an MPICH2 job using classic launch. When TotalView starts up, it will be focusing you on assembler code.



Question

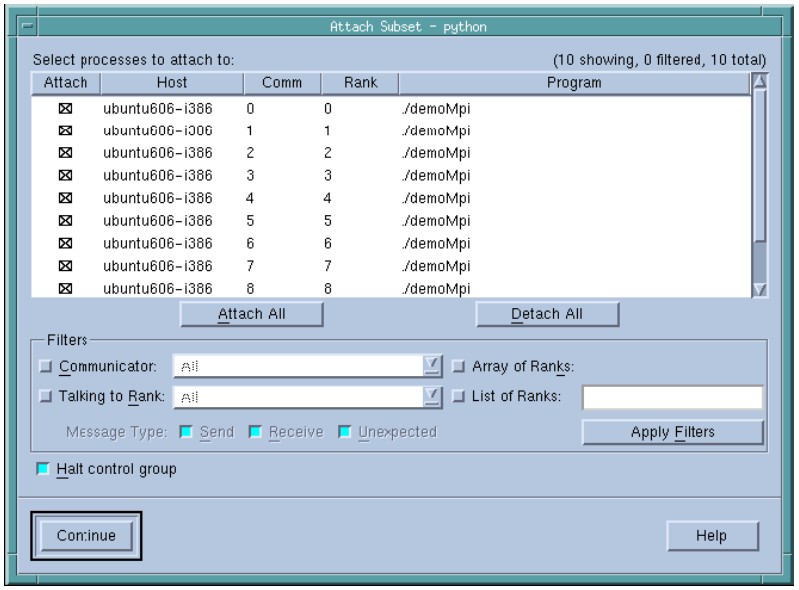
Why does it focus you on assembler code?

_____

You cannot set breakpoints in your code yet because your code has not yet been loaded. If you were to press Go now, python would execute and launch your job and TotalView would ask you if you want to stop at that point, perhaps to set breakpoints. This is TotalView's default behavior. Rather than using the default, we will show you how to use TotalView's Subset Attach feature.

- Select the **File > Preferences > Parallel** command
- Select **Ask What to do** under "**When a job goes parallel**"
- Press **OK**
- Press **Go** 

TotalView now displays its Attach Subset dialog box.

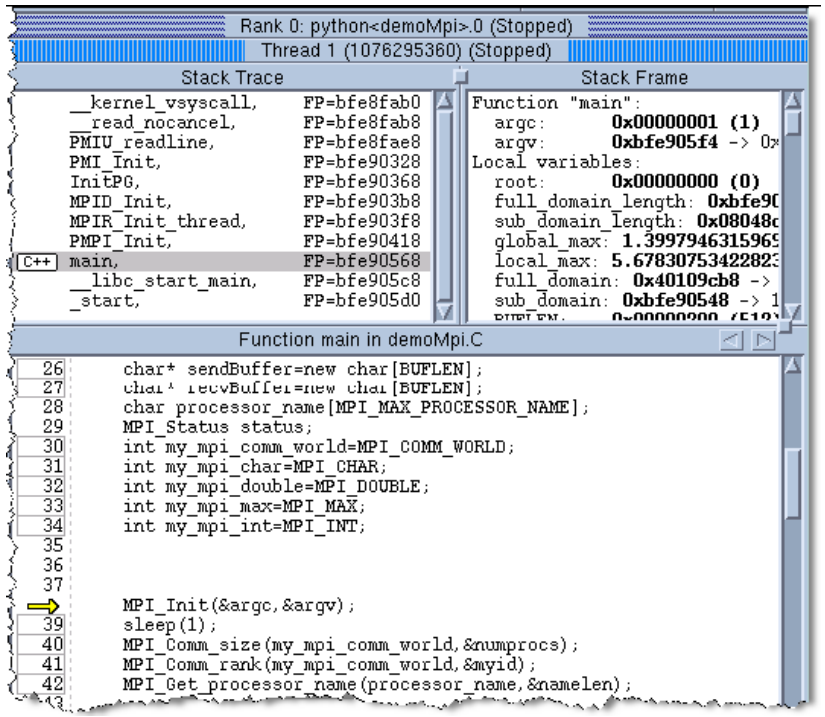


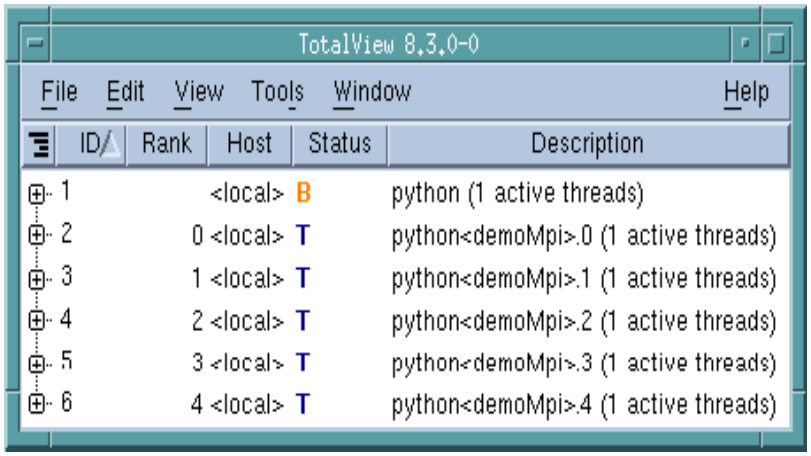
This dialog box allows you to attach to a subset of the processes in your job, listing the host names and ranks of each process. If you have a TotalView Team license, TotalView will count only the processes that you actually attach to for license purposes. You could also use this feature to debug an extremely large job by debugging a subset of it at a time.

- Unselect ranks `5` through `9` in the Attach column
- Press the **Continue** button

TotalView will only attach to the first five ranks and will focus your Process Window on `Rank 0`.

Click on `main()` in the Stack Trace Pane to view the source code





In the Root Window, observe an additional process besides the five processes you attached to. You will see a row for the python process, which is your starter process. In almost all cases, you should ignore this process.

- Click the **Group > Attach Subset** command
- Click **Detach All** to unselect ranks `0` through `4`
- Select ranks `5-9`
- Press the **OK** button

TotalView is now attached to the other half of the job.

- Exit your TotalView session

Step 6: Attaching to a Running Job

If your MPI is built with TotalView support, you can attach to an already-running job across an entire cluster by just a couple mouse clicks. The trick to this is to attach to the starter process.

Tip: To enable this behavior in MPICH2, you must use the `-tvsu` option when you launch the job.

Challenge
Start your job using:
mpiexec –n 4 $LABROOT/demoMpi

Now attach to the entire job.
Hint: Attach to mpiexec.hydraprocess

END OF LAB 3

Lab 4: Exploring Heap Memory in an MPI Application

This lab will explore using MemoryScape, the TotalView memory debugger, within an MPI application.

Expected Time: 30 minutes

Step 1: Start TotalView

- Change directories to `$LABS` by typing:
- `cd $LABS`
- Start TotalView by typing:
  `totalview ./memory-mpi`

If you haven't started TotalView on this program before you will need to tell TotalView that it is a parallel application.

- If the Startup Parameters Window is not automatically displayed, select the **Process > Startup Parameters** command
- Click on the **Parallel** Tab
- Select **MPICH2** as the parallel system
- Set the number of tasks to `4`
- Click on the **Arguments** Tab
- In the Command-line arguments box, enter the letter **R**
- Click **OK**

Step 2: Setting up for Memory Debugging

Under normal circumstances it is not necessary to do anything at compile or link time in order to enable memory debugging.

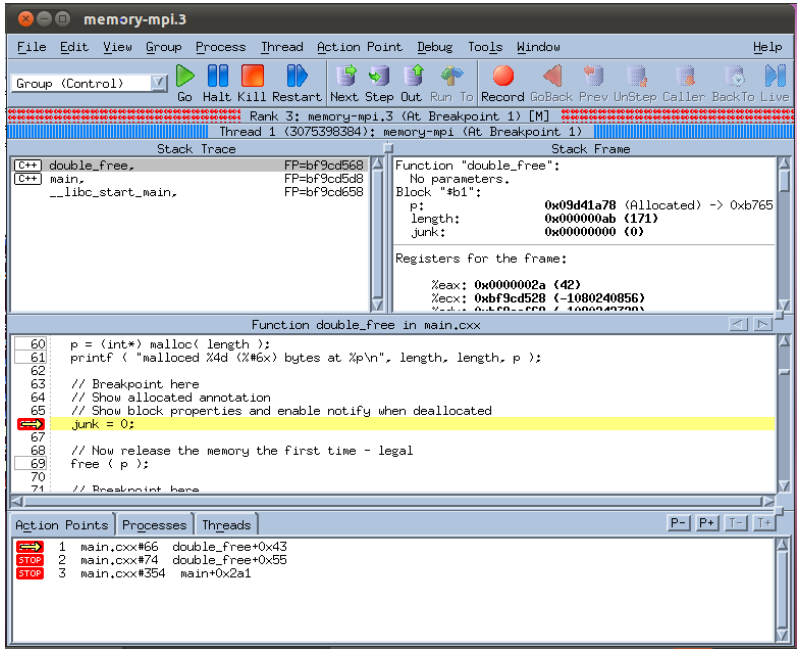
Question

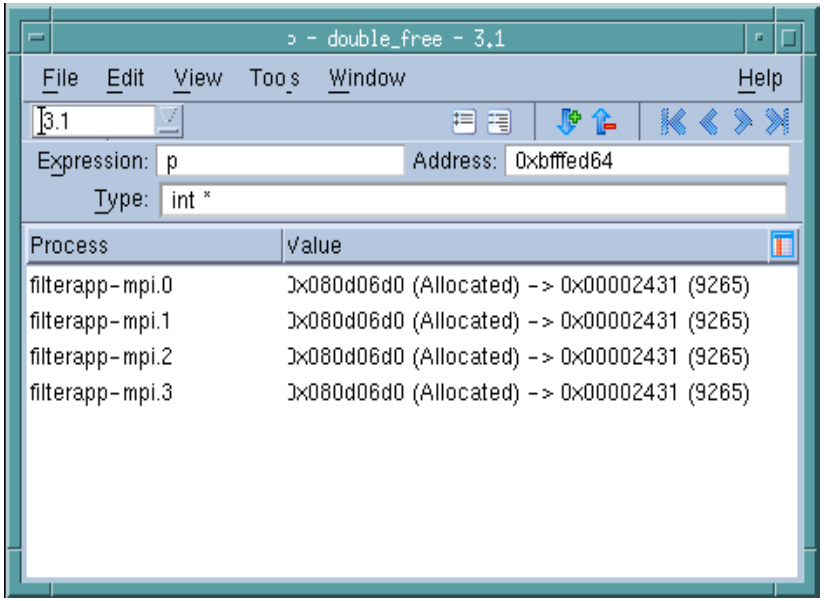
1. Under what circumstances would you have to link with the memory debugging library (`libtvheap.so`)?

   _____

- Select the **Debug > Enable Memory Debugging** command to enable memory debugging
- If not already enabled, enable Stop on Memory Errors: check that the **Debug > Stop on Memory Errors** box is selected.
- Press **Go** 

The application will stop at a breakpoint on line 66. Observe that the Process status bar is annotated with `[M]` to indicate that you are memory debugging the processes.





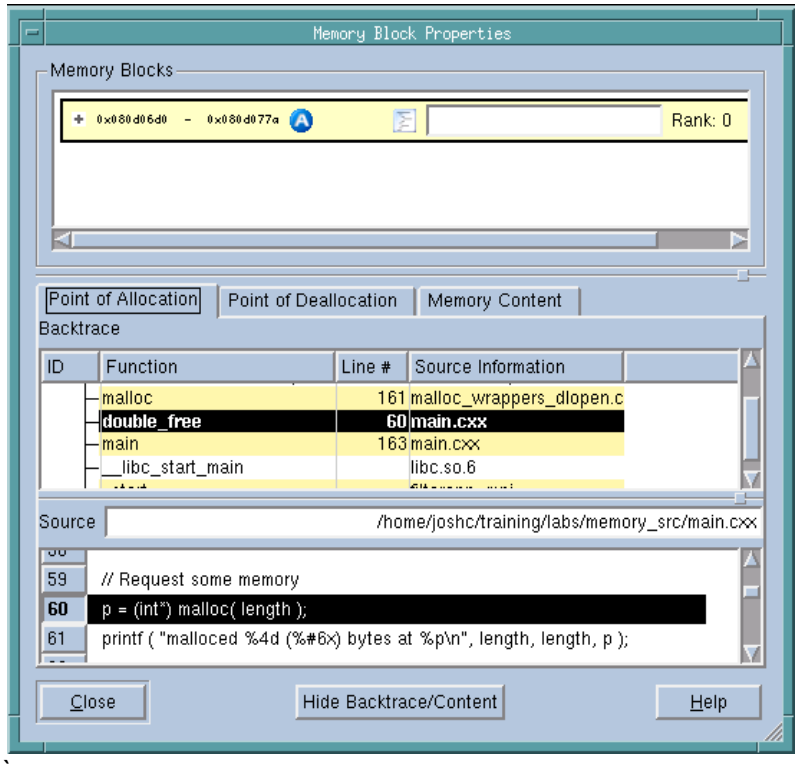
Step 3: Pointers

- Dive on the variable `p`

Note that the debugger has annotated the pointer to tell you that the pointer is allocated. This works when displaying the variable across all processes as well.

- Select the **View > Show Across > Processes** command

- Select the **View > Show Across > None** command
- Dive on the pointer in the Variable Window
- Select the **Tools > Add to Block Properties** command

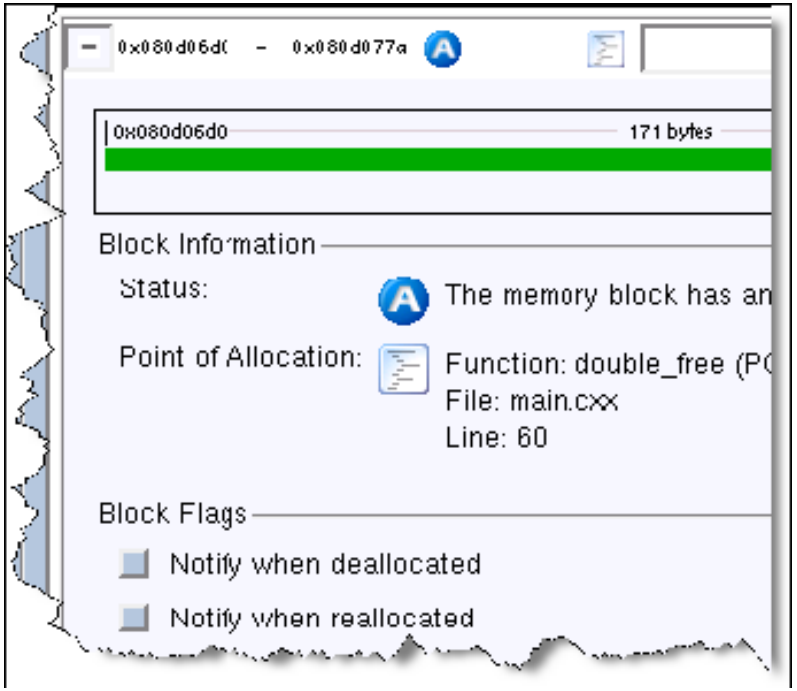


Challenge

Change the Variable Window for `p` to display the same information as the Memory Content Tab in the Memory Block Properties Window.

- Expand out the memory block in the Block Properties Window: Select the (+) icon
- Scroll down to see the Block Flags

If you want to be notified when a particular block is freed or reallocated, set these flags.



Close the Block Properties Window – do not set any flags

Press **Go**  in the Process Window

Your program hits the breakpoint on line 74.

Select the Variable Window containing the `p` variable

Click the **left arrow** (at the top right) to undive

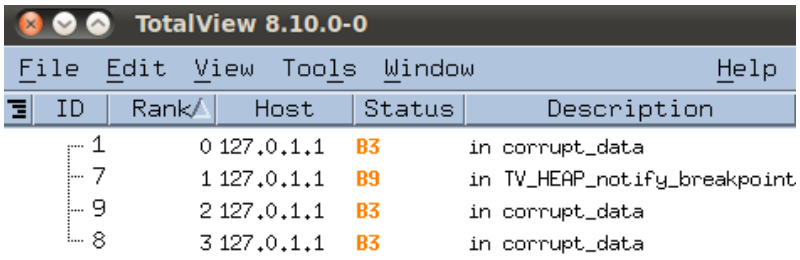
TotalView tells you that the pointer is dangling. If you were to go to the Block Properties dialog, you would notice that the stack trace at the time of deallocation is provided. This tells you where the program freed the memory.

Close this Variable Window

Step 4: Memory Events and Errors

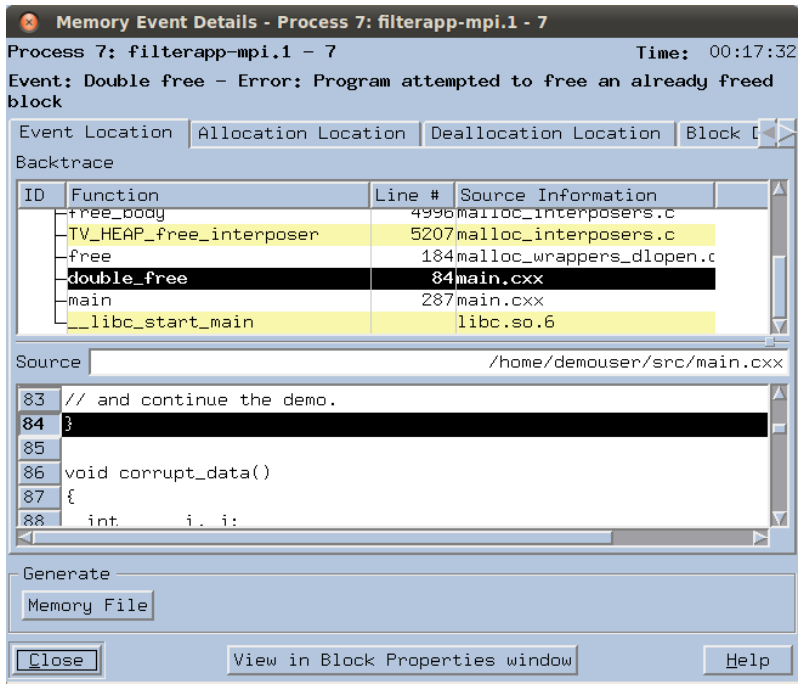
- `Press **Go**

TotalView halts the job. But look closely at the Root Window. Observe that three processes are halted at one breakpoint while one process (`rank 1`) is at another breakpoint.



- Dive on the rank that is not at the same break point as the others.

TotalView opens a Memory Event Details Window, showing you that the “Program attempted to free an already freed block.” TotalView will open the Memory Event dialog when the Process Window is focused on the process which received the event.



Observe that the Event Location Tab in Memory Event Details Window displays the location where the process is currently halted.

Questions

2. Where can you find the same information as is displayed in the Memory Event Details Window's Allocation, Deallocation, and Block Details Tabs?

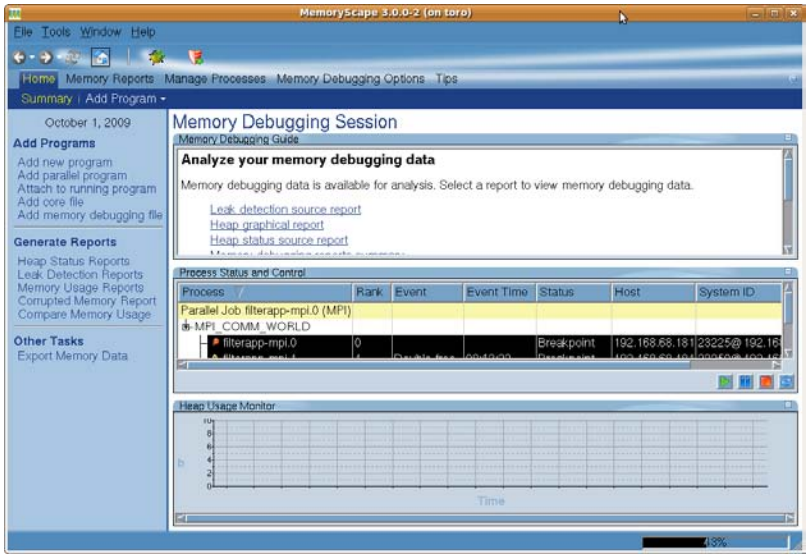
3. What are the stack frames above the frame for free() and why does TotalView report that the process is at a breakpoint? Is this a breakpoint you can disable or delete?

4. Since TotalView does not automatically focus you to a process when it receives a memory event what can you do to make sure you do not miss an event? How can you recover the event information should you miss it?

5. Why does the process list in the bottom left corner only have one process?

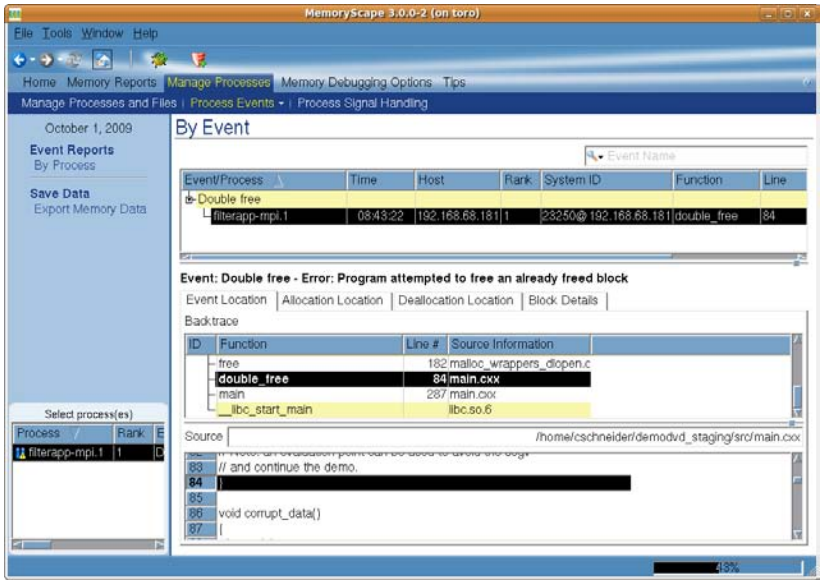
Close the Memory Event Details Window
Select the Debug > Open MemoryScape command

The MemoryScape Window will open. This is where you can run the Heap Memory Reports and enable or disable various memory debugging options.



Click on the **Manage Processes** Tab and on the **Process Event** Subtab
Select **By Event** Report

This report allows you to view an aggregation of all events that have occurred across all processes.



Question

6. What kinds of memory events and errors can MemoryScape provide events for?

_____

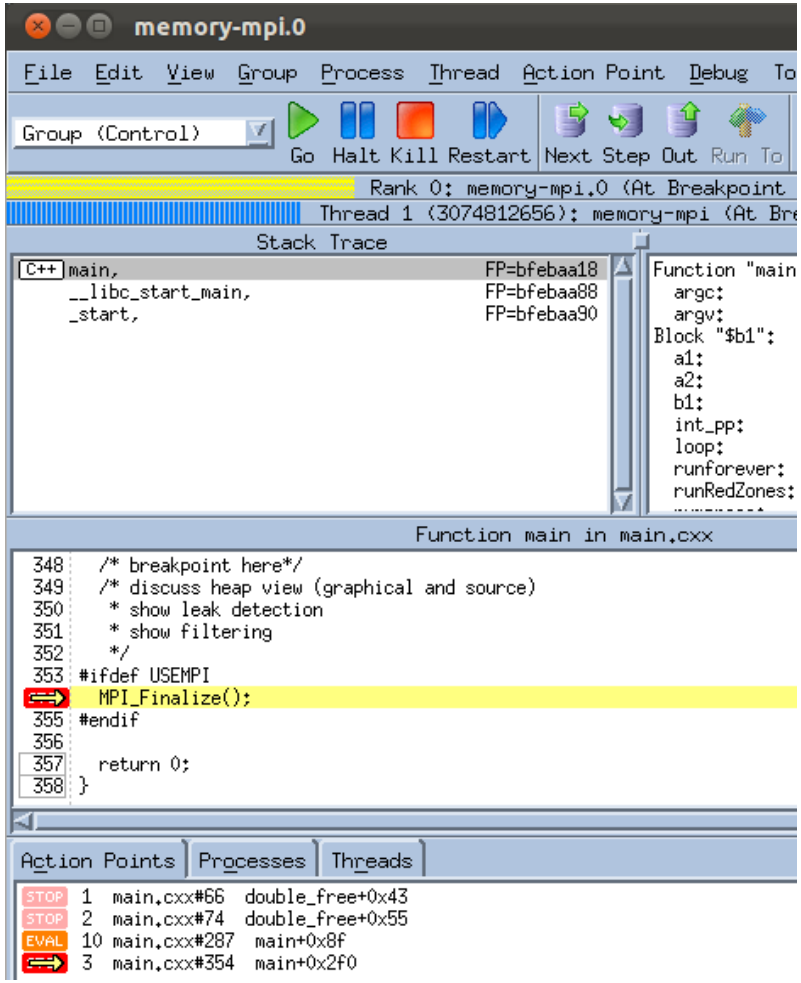
Examine the kinds of memory events and errors MemoryScape provides.

Click the Memory Debugging Options Tab
Press the **Advanced Options** button
Click the **Advanced** button – this is within the Halt execution on memory event or error area

Step 5: Heap Reports and Leak Reports

Dismiss the Memory Event Details Window
In the MemoryScape Window, disable **Use Red Zones for `MPI_COMM_WORLD`**
In the **Halt Execution** > **Advanced Options** disable **Double_free** event
In the Action Points Tab of the **TotalView** Window, disable everything except the evaluation point **line 287** and the last breakpoint at **line 354**
Click **Process/Startup Parameters** and select the Arguments Tab
Delete the entry in Command-line arguments
Click **OK**
Press **Restart** (if a confirmation box appears, click **Yes**)

This should result in all processes being stopped at the remaining active breakpoint. It will take a minute for Rank 1 to reach that breakpoint. The Evaluation action point at line 287 was set to skip over code this lab is not interested at this time.

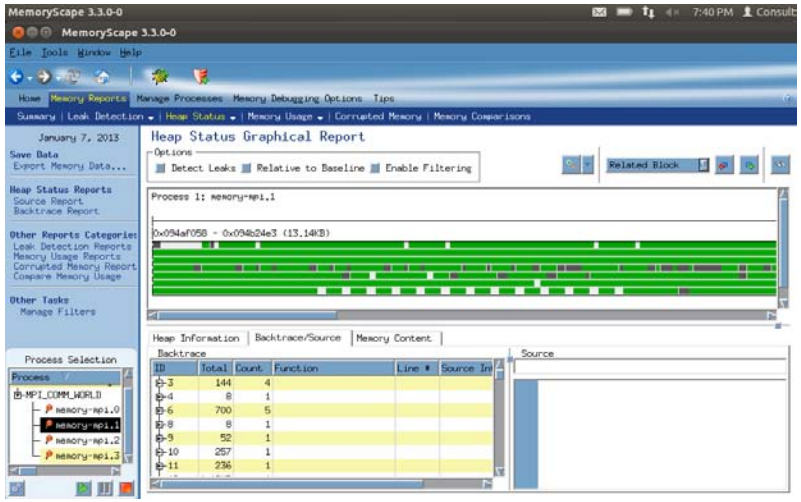


Heap Graphical Report

- Select the **MemoryScape** Window
- Click on the **Memory Reports** Tab, then **Heap Status Reports**
- Select `memory-mpi.1` in the **Process Set** control

Generating memory reports is an expensive operation, so we recommend generating reports one process at a time. That being said, generating memory reports across several processes can be helpful, for example if you want to compare the differences across these processes. However, the Heap Graphical Report only analyzes one process at a time. If you want to view your heap status across several processes we recommend using the Heap Source Report.

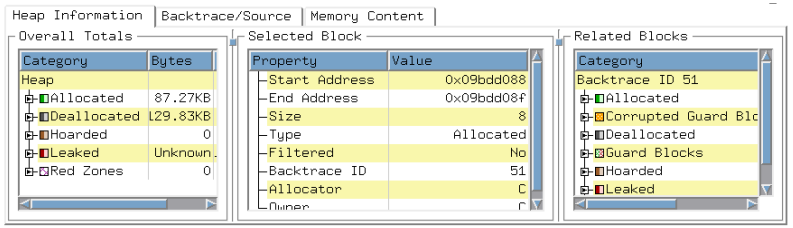
Select **Graphical Report**



The Zoom-In and Zoom-Out controls at the top right of the window will help to zoom out to visualize how your heap memory is fragmented, or to zoom in to particular memory blocks.
Observe the Key and Overall totals in terms of counts and bytes in the Heap Information Tab.

- Scroll to the top of the Graphical Report
- Zoom in so that you can select the very first allocated block displayed.

MemoryScape fills in the Selected Block and Related Blocks Panes.



Questions:

7. How does MemoryScape define a related heap block?

8. Why is it useful to know about the related heap blocks?

_____

At the top right of the Heap Graphical Report you should see a list control that says “Leaked Block.”

Select **Related Block** from this list

This control is provided to navigate between blocks in the heap graphical view.

Click the **Find next** arrow (the right arrow next to the list control)

This will select the next related memory block and focus the graph on it. Note that the Selected Block Pane has been modified to indicate information for the current selection.

Right-click on the block and select **Properties**

Any time you select a memory block in a memory report you can right-click on it to get at the block's properties.

- Close the **Block Properties** Window
- Select the **Backtrace/Source** Pane

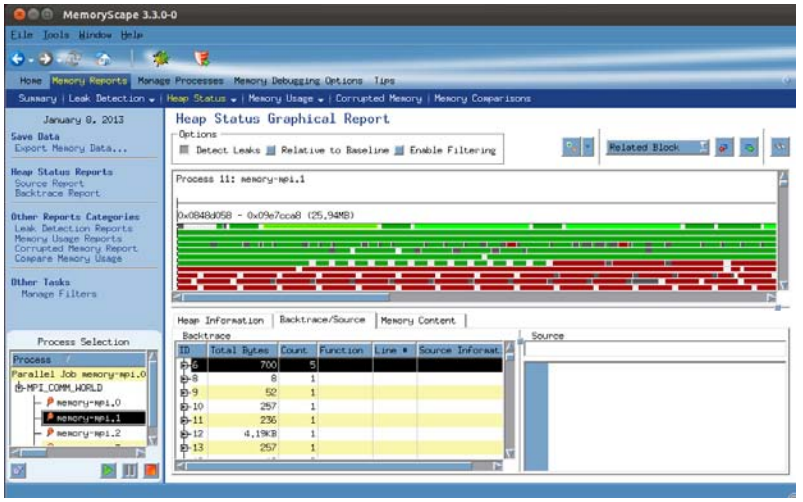
The Backtrace Tab will show information pertaining to all allocations organized by backtrace. The selected backtrace is the backtrace for the current selected block and consequently all related blocks.

Expand the selected backtrace

You can now view the backtrace and source for the allocation point.

Select from the MemoryScape process window a Process other than memory-mpi.1
Check the leak detection option above the graph

Observe that the graph now displays the leaked blocks in red.



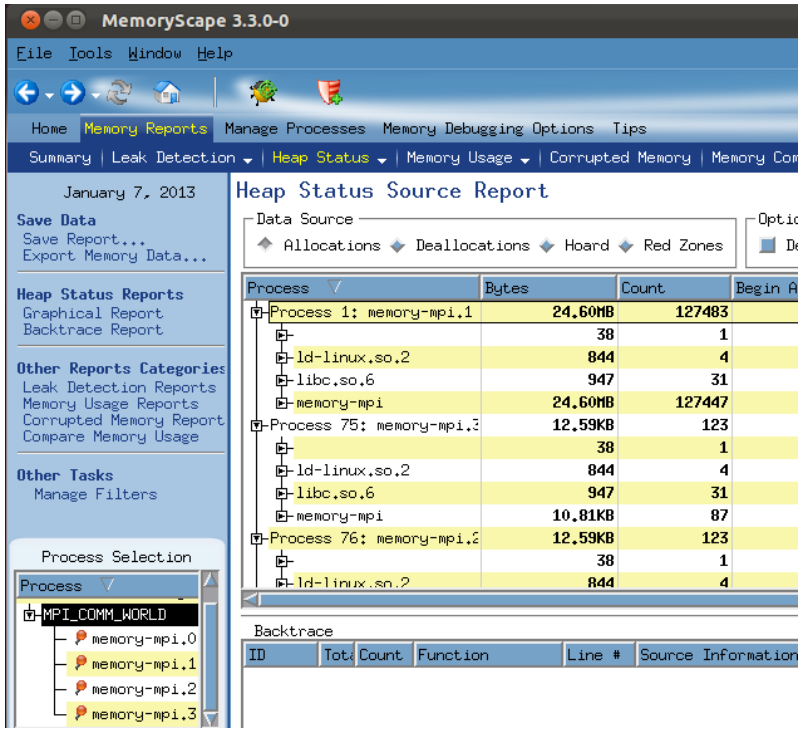
Question

9. Why doesn't the graph display memory leaks by default?

_____

Heap Source View

- Select `MPI_COMM_WORLD` in the Process Set control in the MemoryScape Window
- Select **Source Report** from the list on the left, under Heap Status Reports

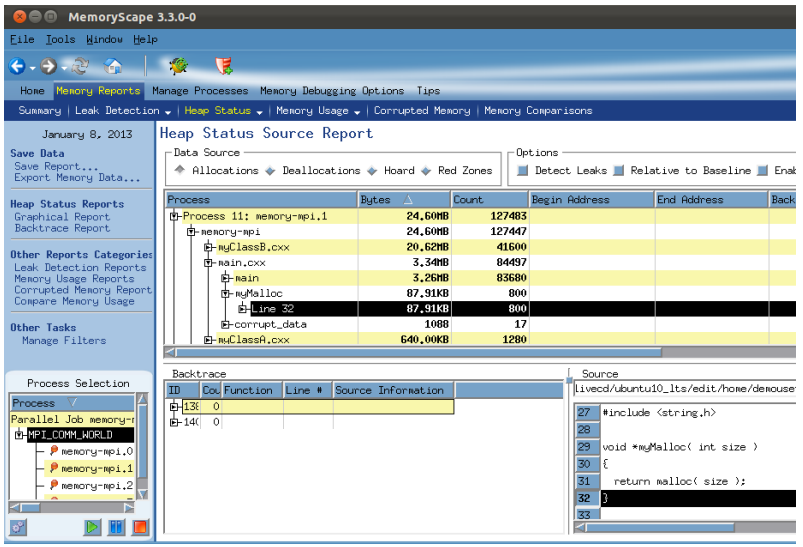


Questions

10. How does the Source View organize information?
11. Do you see anything peculiar with the number of bytes and allocations in the processes?

_____

Expand out the process that is running `memory-mpi.1`, `memory-mpi, main.cxx,` and `myMalloc` to show Line 32 using the tree controls



Select **Line 32** in the Backtrace Pane

This displays several of the backtraces in the Backtrace Pane.

Expand the **first** backtrace

Questions

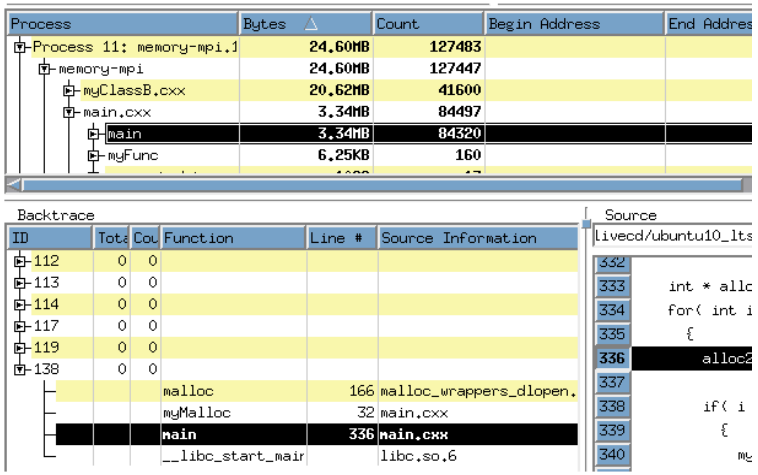
12. Do the memory blocks allocated in `myMalloc()` all have the same backtrace? Why or why not?
13. How does MemoryScape define the allocation focus point for a memory block?
14. Under what circumstances is MemoryScape's choice of the allocation focus point not optimal?

- Right click on **`main()`** in backtrace 138 (or the first backtrace in the list)
- Select **Set Allocation Focus Level**

This tells MemoryScape to focus on the frame directly above `myMalloc()`, since you are probably not interested in the malloc wrapper, `myMalloc()`, but rather the function that called it.

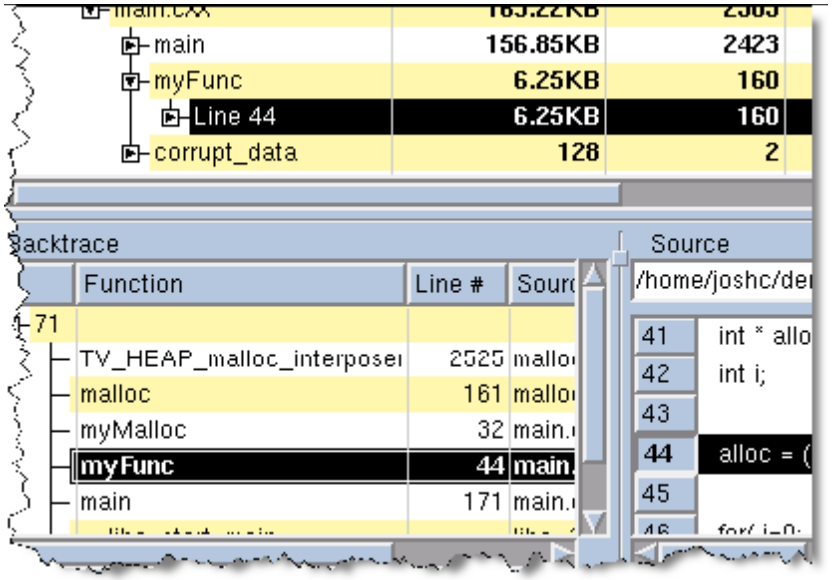
Click on **`main()`**

Observe that the backtrace now appears there.



Click on **`myFunc`**

Observe that the allocations associated with this backtrace now appear under `myFunc` instead of `myMalloc`.

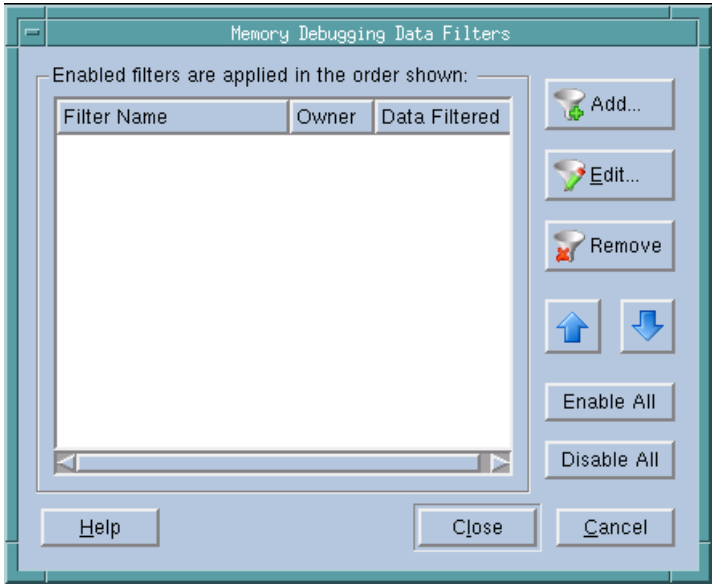


Filters

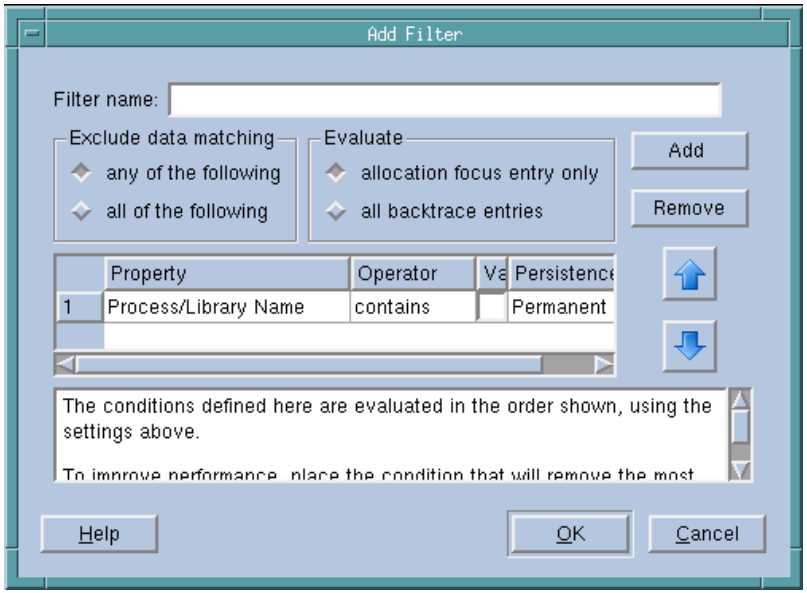
A large application that allocates a lot of memory can generate very large reports. Under these circumstances it may be helpful to focus on the information you are interested in by filtering the memory blocks that are displayed. Filtering memory blocks results in the blocks not being displayed in the Source Report, while filtering in the graphical view results in the blocks being dimmed.

Select the **Tools > Filters** command

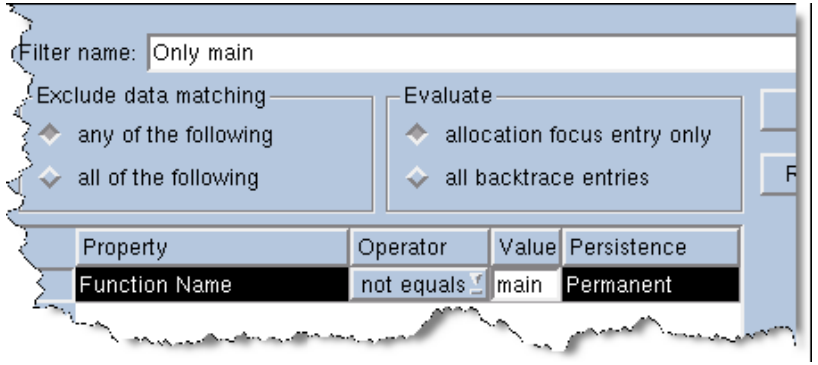
This window allows you to Add, Edit, and Remove various filters. The filters are applied starting from the top and there are controls to move the filters up and down.



Click the **Add** button



- Enter **Only main** for the Filter name
- In the Property column, click the list control to specify **Function Name**
- Select **not equals** within the Operator column's list control
- Enter **main** for a Value
- Press the **OK** button

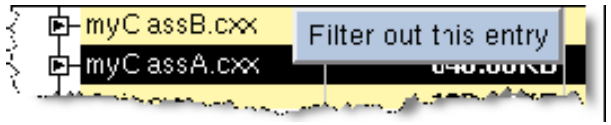


Observe that the Filter gets added to the Filters Window.

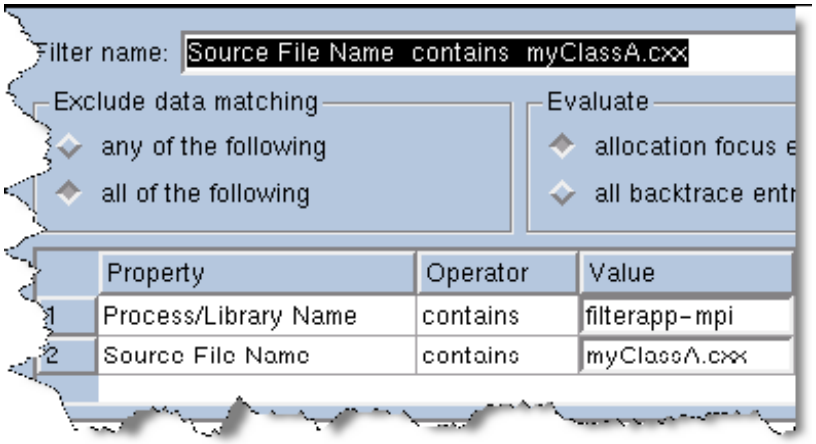
- Press **Ok** in the Filters Window
- Check **Enable Filtering** in the Options box at the top of the report

Explore the resulting Source Report and note that only memory blocks where the allocation focus point is `main()` are displayed.

- Disable filtering
- Right click on **`myClassA.cxx`**
- Select **Filter out this entry**



This initializes the filters dialog with the context from your selection.



Click **OK**

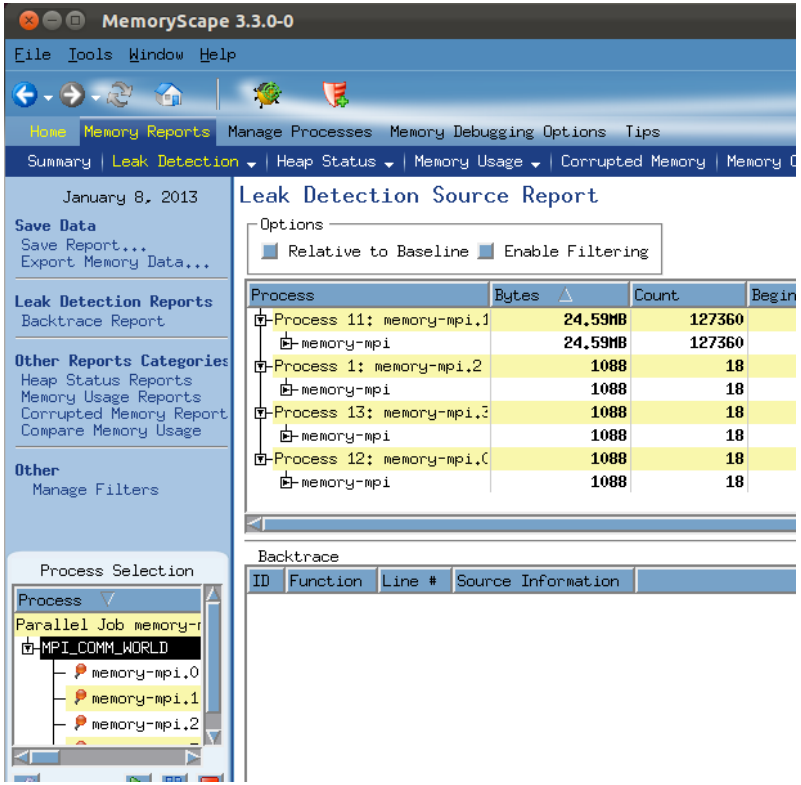
This accepts the filter's pre-filled-in values. It also automatically reenables filtering and regenerates the view for you. However, in this case this is not what we wanted because our only main filter is still active.

- Select the **Tools > Filters** command
- Uncheck the filter with the name `Only main`
- Click **OK**

Observe that the display shows only blocks which do not have allocation focus points in the file `myClassA.cxx`.

Leak Detection

- Disable filtering
- Select **Leak Detection** Reports from the list on the left
- Select **Source Report**



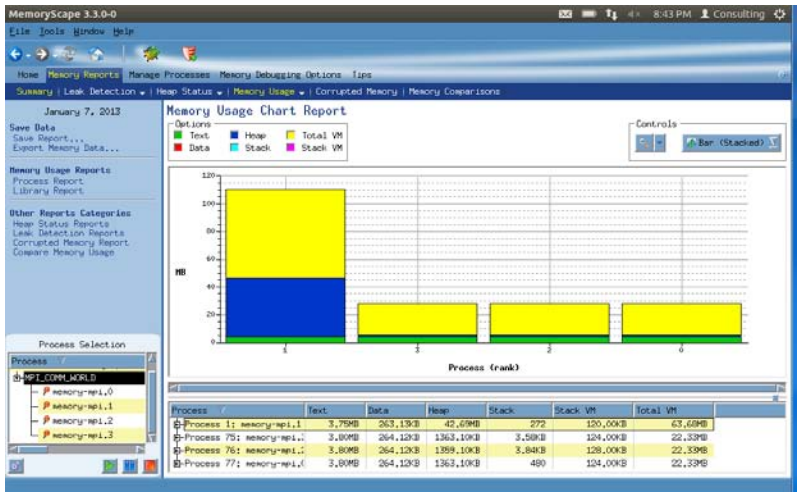
Questions

15. How does MemoryScape define a memory leak?

16. How is generating a Leak Report in the Leak Detection Tab different from detecting leaks in the Heap Graphical or Heap Source Reports?

Memory Usage

- Click on the **Memory Usage** Tab
- Select **Chart Report**
- Unselect Total VM and Stack VM in the Options area
- Select **Bar (Stacked)** in Controls area



Questions

17. What rank is clearly using more heap than the others?
18. Does memory debugging having to be enabled in order to view the Memory Usage Reports?
19. Why do the two reports show different values?
20. What might the Memory Usage Report be useful for during a debug session?

END OF LAB 4

Lab 5 Debugging Memory Comparisons and Heap Baseline

Expected Time: 15 minutes

Step 1: Memory Heap Baseline

Memory heap baseline is used to dynamically observe memory use from one point of a programs execution to another. Once set or reset, the memory Debugger will begin remembering all heap operations that occur within the thread.
The goal of this lab is to demonstrate how to compare memory at different points in the program execution.

- If TotalView is running, close TotalView.
- Change directories to `$LABS` by typing:
  `cd $LABS`
- Start TotalView by typing:
  `totalview ./memorycomp-mpi`
- If the Startup Parameters Window is not automatically displayed, select the **Process > Startup Parameters** command
- Click on the **Parallel** Tab
- Select **MPICH2** as the parallel system
- Set the number of tasks to `4`
- Select the **Debug > Enable Memory Debugging** command to enable memory debugging – hit **OK**
- Press **Go** 

Now at the first breakpoint main.cxx: line 235 (bool loop = false, runforever, runRedZones=false;).

- Select the **Debug > Open MemoryScape** command
- Click on the **Manage Processes** Tab and on the **Process Event** Subtab
- Select **By Event** Report
- Click the **Memory Debugging Options** Tab
- Press the **Advanced Options** button
- In the **Halt Execution > Advanced Options** disable **Double_free** event

Set a Heap Baseline in TotalView and continue the program.

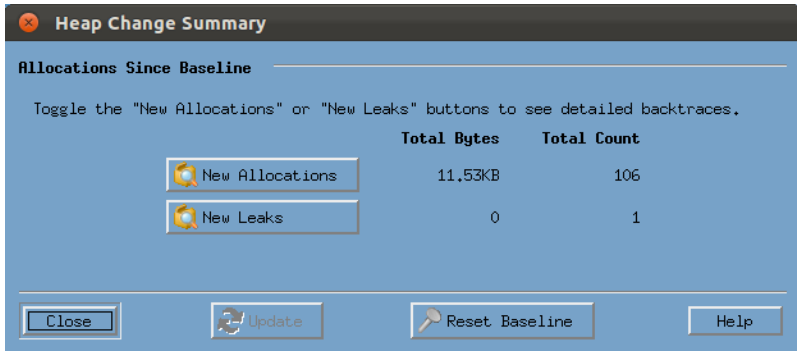
- Select from TotalView gui **Debug > Set Heap Baseline(in Group)**
- Press **Go** 

Now check the memory use from the previous baseline.

You should be at the breakpoint on line 287 (corrupt_data();)
Examine the progress of the Heap Memory versus the baseline:

- Select from TotalView gui
  Debug >Heap Baseline>Heap Change Summary

The report should look similar to the following:



Note that you can select either New Leaks or New Allocation and view the backtrace and the source of the leaks or allocations..

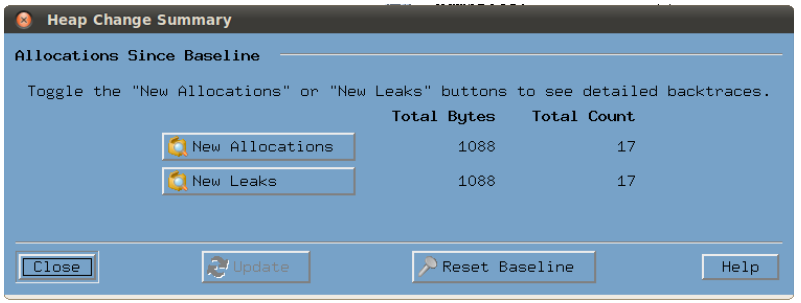
- Select from TotalView gui
  Debug > Set Heap Baseline(in Group)
- Press **Go**
- Select **Debug >Heap Change Summary**

The first Heap Summary should have no new leaks and no new allocation.

Examine the New Allocations and new Leaks

- Press **Go**
- Select
  Debug >Heap Change Summary

The second heap Summary should look like the following:



Questions

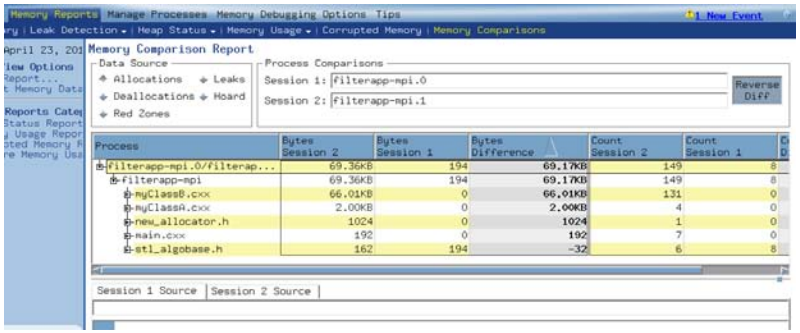
1.) How would you verify the new allocations up to this point?

Step 2: Memory Comparisons

Memory comparisons can be used to compare the allocations, leaks, and deallocations between two processes. They may be two MPI processes, two processes from the same executable operating on different inputs, or a live process and a post-mortem process (whether that is a core file or a memory debug file).

You should be a the breakpoint line 354 (MPI_Finalize();)

- Click on the **Memory Report** -> **Memory Comparisons** report
- Select `memory-comp.0` for Process 1
- Select `memory-comp.1` for Process 2
- Press the **Compare** button
- Expand the process for `memory-comp`



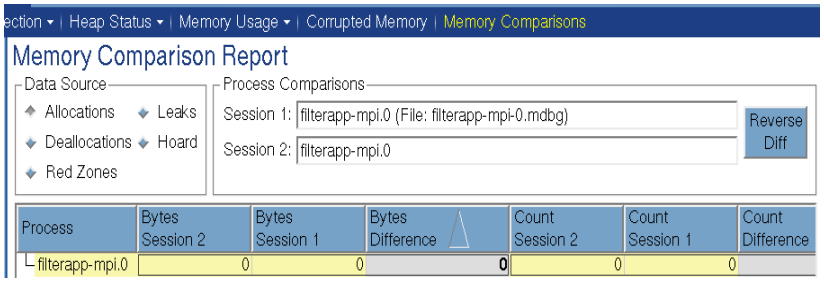
Questions

2.) Observe the byte counts reported for session 1. Why are they zero?

3.) What allocations are not displayed in the Memory Comparison Report?

_____

- Go to the **Home > Summary** Tab
- Select `memory-comp.0` in the Process Status and Control
- Begin creating a memory debugging file by selecting **Export Memory Data** from the task list on the left
- Accept the defaults by pressing the **Export** button
- Select the **Add memory debugging file** under **Add Programs** from the task list on the left
- Select the file you just created, using the Browse button
- Press the **Next** button
- Select **Analyze memory data**
- Select **Memory Comparison Reports**
- Select `memory-comp.0` as Process 1 and the `memory-comp.0` memory debug file (the file you just opened) as Process 2
- Press the **Compare** button



Question

4.) Why is there no difference?

_____

- Select the **Home > Summary** Tab
- Select one of the processes
- Press the **Tools** - > **Debug in TotalView** button on the tool bar to display a Process Window
- Scroll to line **259**
- Right-click on the line number and select **Properties**
- Click on the **Evaluate** check box
- Type the following in the expression field:

```
loop = rank == 0;
goto 354;
```

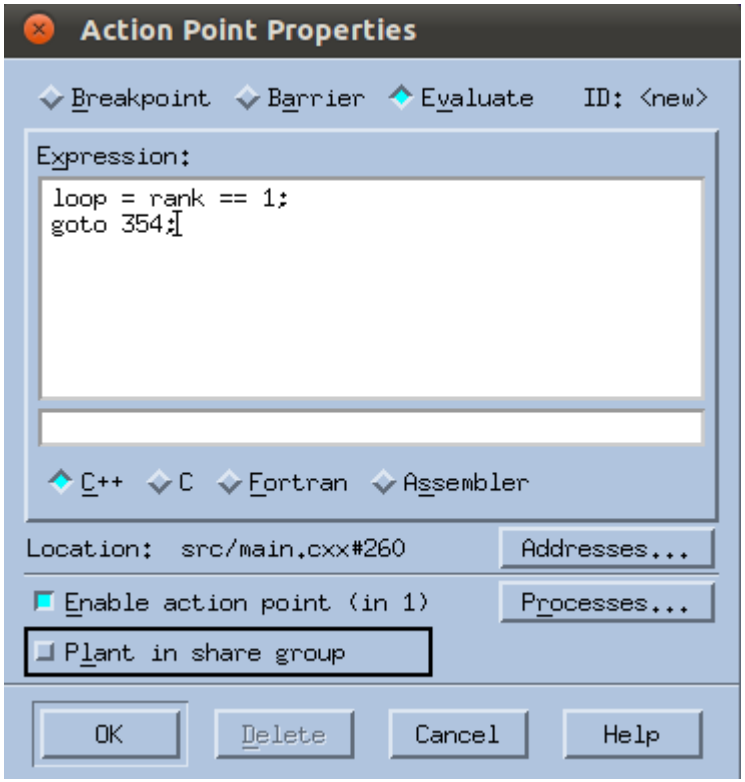
- Press **OK**

You have now set an evaluation point on line 259.

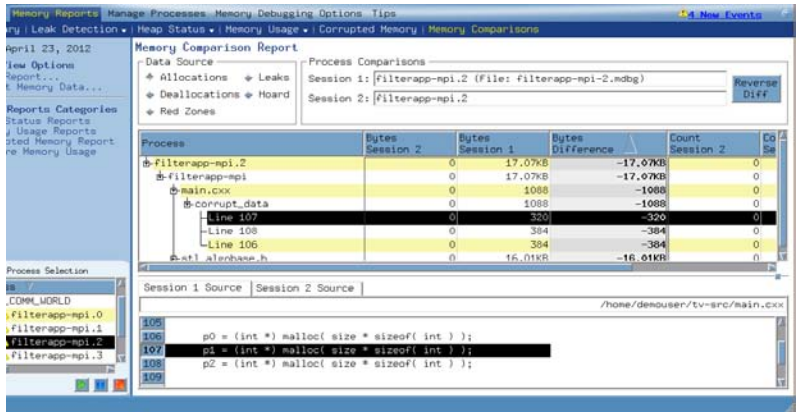
Questions

5.) What does this evaluation point on line 259 do?

a. Hint: Search the **TotalView User Guide** for **Eval Points.**
b. What other commands can be used at an eval point?



- Disable all breakpoints except the last one in the Action Points Pane and the new evaluation point
- Press the **Restart** button (if a confirmation box appears, press **Yes**)
- Wait for all four processes to halt at the breakpoint
- Go to the MemoryScape Window
- Select the **Memory Reports** Tab
- Select the **Memory Compare** Report
- Make sure that both `filterapp-mpi.0` and the `filterapp-mpi.0mdbg` file are selected
- Press the **Compare** button



Questions

6.) Under what circumstances might you want to compare the differences between a live process and a post-mortem process?

7.) What other kinds of memory reports can you generate on a memory debug file?

_____

Most memory debugger reports can be saved for later use. To do so, select **Save Report** from the task list on the left.

END OF LAB 5

Lab 6 Memory Corruption discovery using Red Zones

This lab will explore using Red Zones and Guard Blocks with MemoryScape.

Expected Time: 40 minutes

Step 1: Memory Corruption

Questions

Review: What types of Memory Corruption can Heap Interposition technology help you with?

_____

Review: When can you get notified that your application has corrupted memory?

_____

- Change directories to `$LABS` by typing:
  `cd $LABS`
- Start TotalView by typing:
  `totalview ./memory-redzone`

- If the Startup Parameters Window is not automatically displayed, select the **Process > Startup Parameters** command
- Click on the **Parallel** Tab
- Select **MPICH2** as the parallel system
- Set the number of tasks to `4`
- Click on the **Arguments** Tab
- In the Command-line arguments box, enter the letter **R**
- Click **OK**

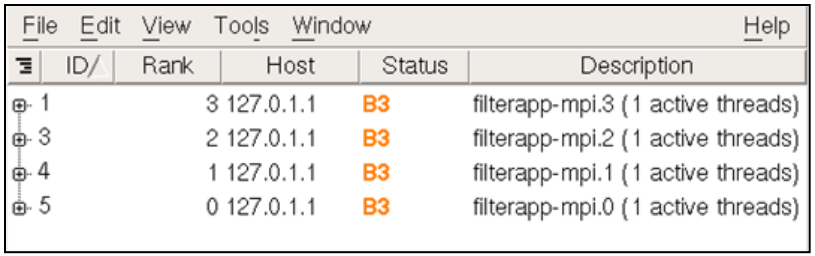
Turn on memory debugging.

- From the TotalView gui: **Debug>Enable Memory Debugging**
- **Debug>Stop on Memory Errors**
- Select the **Process > Go** command

TotalView should stop with a double free event.
Now run Rank 1 alone to synchronize to the other processes.

- Go to the TotalView Window
- Focus on `Rank 1`
- Select the **Process > Go** command

TotalView runs just Rank 1 to the breakpoint at line 109 so that it is synchronized with all other processes.



- If open, close the **Memory Event Details Window**
- Select the **Debug > Open MemoryScape** command

Set up for Guard blocks to be used later in the lab.

From the **Memory Scape** gui:

- Click on the **Manage Processes** Tab and on the **Process Event** Subtab
- Select **By Event** Report

- Click the **Memory Debugging Options** Tab
- Select **`MPI_COMM_WORLD`** in the Process Set control
- Press the **Advanced Options** button
- Check the **Guard allocated memory** option

This will enable the option globally across MPI_COMM_WORLD. If you select an individual process in the Process Set control, any configuration changes will only apply to the process which you select.

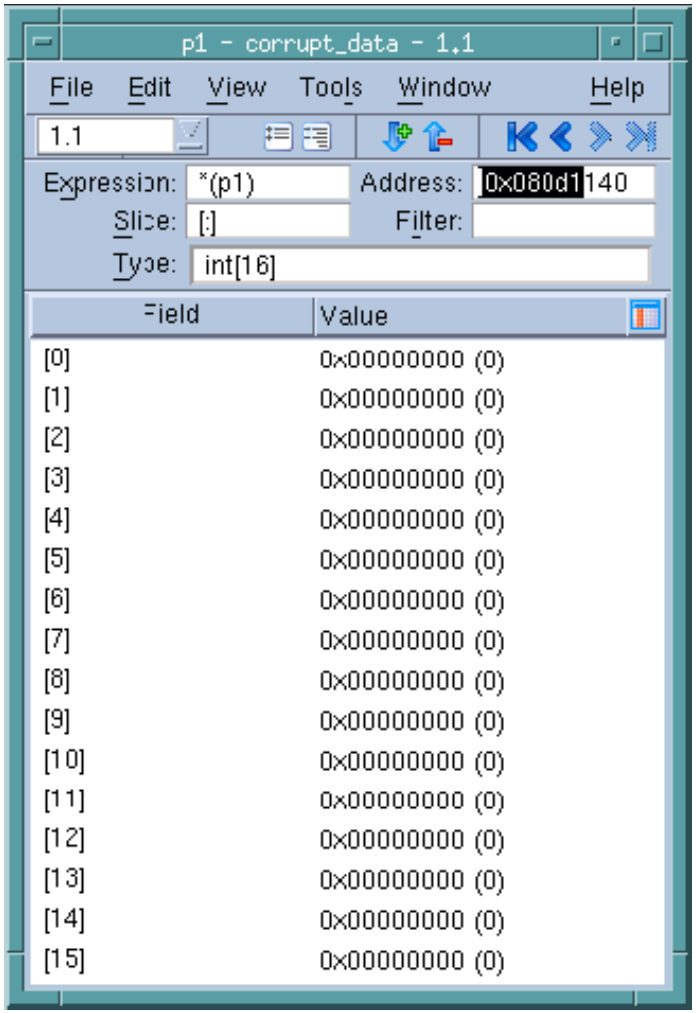
Question

1. What types of memory corruption can the Heap Interposition technology help you with?
2. When can you get notified that your application has corrupted memory?
3. How can you change the bit pattern that TotalView uses to paint the guard regions?

_____

- Dive on the variable `p1`
- Dive through the pointer in the Variable Window
- Cast `p1` to an array of size 16

This shows `p1` as being an array.



Challenge
Modify the Variable Window display to show the contents of the pre- and post guard regions.

- Dismiss the **Variable** Window
- Disable the breakpoint at **line 109**
- Run the whole group using **Go** 

TotalView runs the job to the breakpoint at line 121. Note that the `for` loop overwrites the bounds of the array by 1.

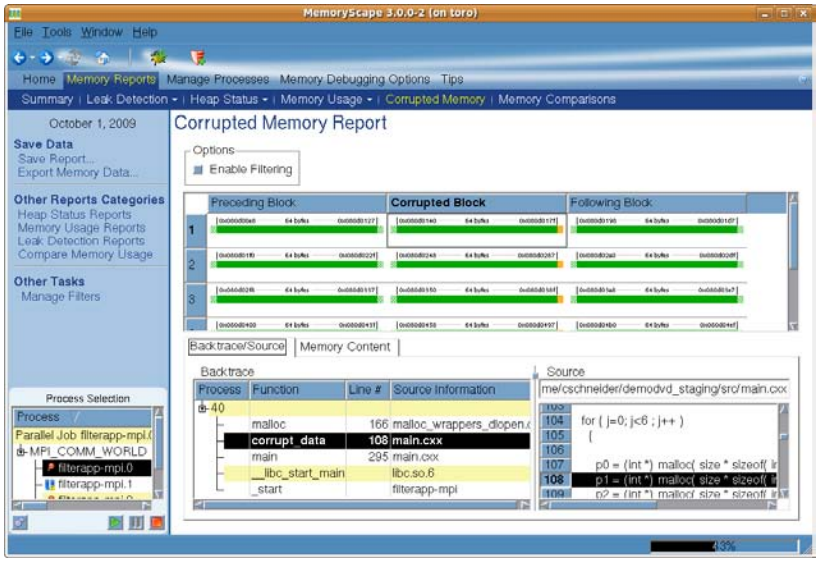
Question

4. Why didn't MemoryScape halt your program at the time it overwrote the bounds of the array?

_____

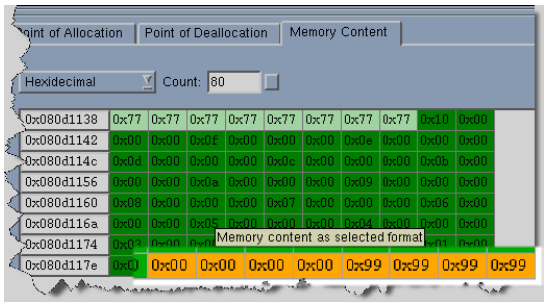
- Go to the MemoryScape Window
- Click on the **Memory Reports** Tab
- Select `memory-redzone.0` in the **Process Set** control
- Select the **Corrupted Memory Report**

The Corrupted Guard Blocks Report will check all heap regions with guards to see if their guard areas have been corrupted. Note that the view shows you the preceding and following memory blocks in order to help locate the problem.

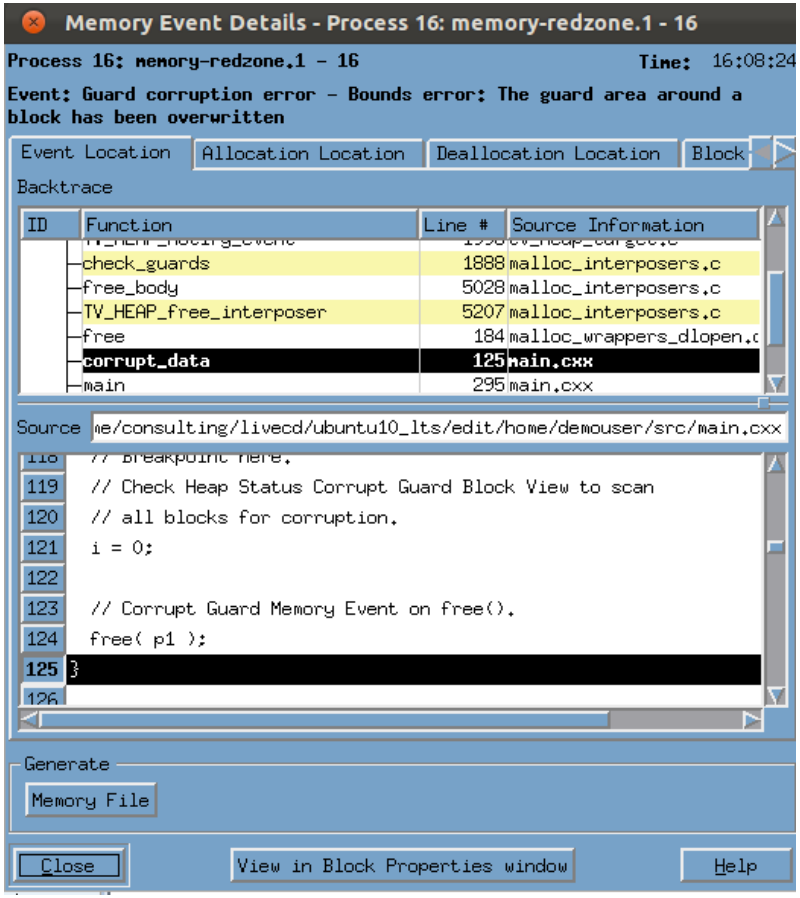


- Right click on a Corrupted Block
- Select **Properties**
- Click on the **Memory Contents** Tab

Observe that memory was overwritten.



- Dismiss the **Memory Block Properties** Window
- Press **Go**  in the TotalView Window



Your job will continue and when the processes actually free the memory block, TotalView will halt your job with a guard corruption event. You can view this information either in the Memory Event Details Window, which opens when you focus on a particular process, or when looking at the Process Events Report.

- Go to the **MemoryScape** Window
- Click on the Memory Debugging Options Tab > **Advanced Options**
- Select **`MPI_COMM_WORLD`** in the Process Set control
- Click twice if necessary on **Guard allocated memory** to disable it

With the exception of enabling memory debugging, memory debugging options can be toggled on and off in the middle of a debug session. The reason you needed to toggle this option twice to turn it off is because you selected `MPI_COMM_WORLD`, which represents a group of processes. MemoryScape does not try to resolve the state of all options for each process in the group.

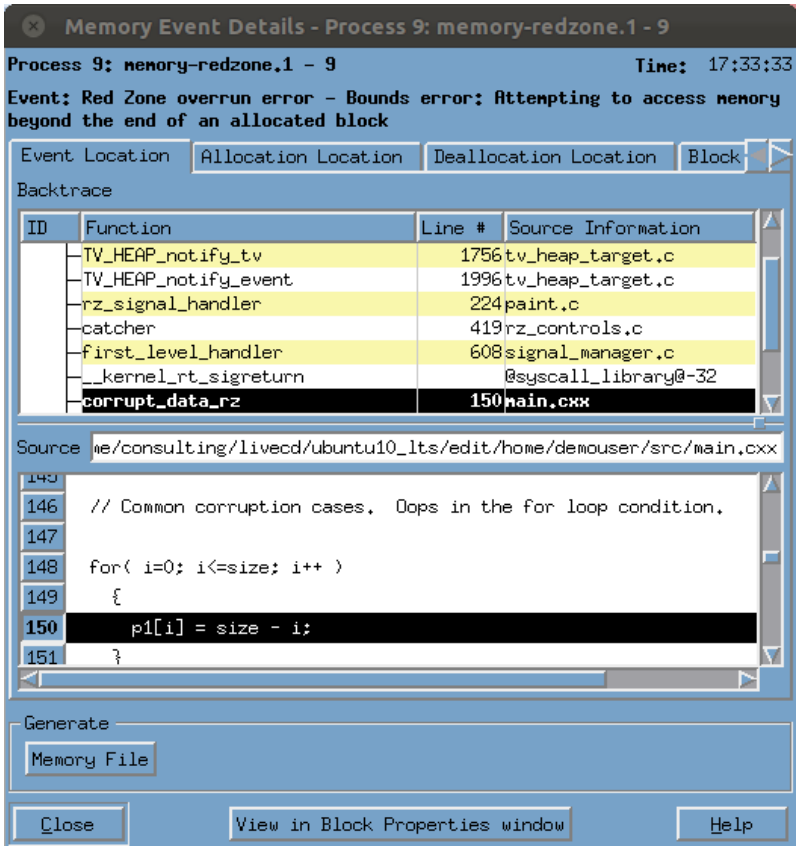
Question

5. Why can't you toggle the Enable memory debugging option while the program is running?

_____

Step 2: Red Zones and Heap Reports

- Under the **Memory Debugging Options** Tab, enable **Use Red Zones** to find memory access violations
- Go to the **TotalView** Window
- Highlight main.cxx line 121
- Click the **Run to** button



A Memory Event occurred in a function called `corrupt_data_rz`, which is similar to the `corrupt_data` function involved in the previous step. With Red Zones, however, the event is triggered immediately at the point when the program tries to write beyond the array bounds. That point is displayed in the Event Window's Source pane.

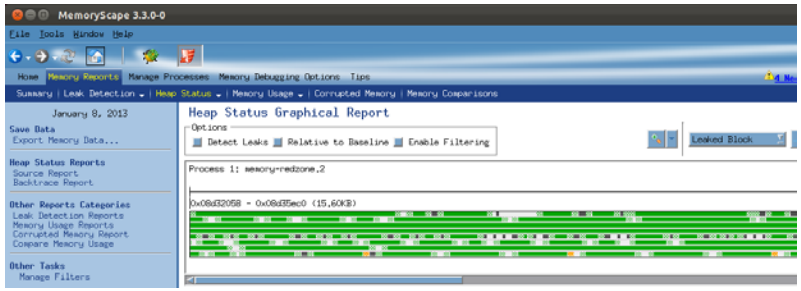
- Click on the **Block Details** Tab in the Memory Event Details Window
- Examine the memory contents.

Question

6. Was the memory beyond the array bounds actually overwritten?

_____

- Dismiss the **Memory Event Details** Window
- In the MemoryScape Window, click on **Memory Reports**, then **Heap Status**
- Select the **Graphical Report**, and scroll down in the graph pane past the end of the mostly-green set of blocks



In the lower set of blocks, the crosshatched areas indicate where Red Zone placements start. The full size of the Red Zone includes some additional memory, which is shown in grey in the graph.

- Go to the Process Window and press **Go** 
- In the window that pops up, press **Let process exit**

Questions

7. Why is there so much empty space with Red Zones?

8. Why is there so much additional overhead for Red Zones?

9. Why can't a program be continued after a Red Zone event?

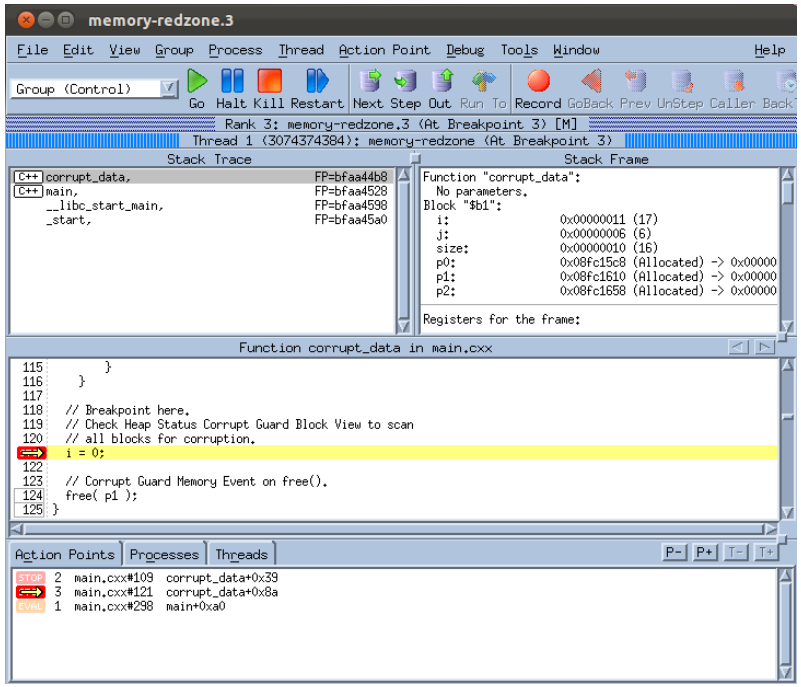
_____

You may need to restart TotalView if TotalView should shut down from the last operation.

Step 3: Restricting Red Zones

Begin the debugging. After starting the debugging up again start the Guard allocated memory as well. This will demonstrate changing the Guard blocks on the fly.

- Disable all of the break points except for line main.cxx: 121
- Press **Go** 



- In the **Memory Debugging Options** turn on the **Guard allocated memory**.
- From the TotalView gui: Press **Go** 

Keep an eye on the Root Window. One of the processes may hit a heap event breakpoint, with the stack trace containing initialization routines.

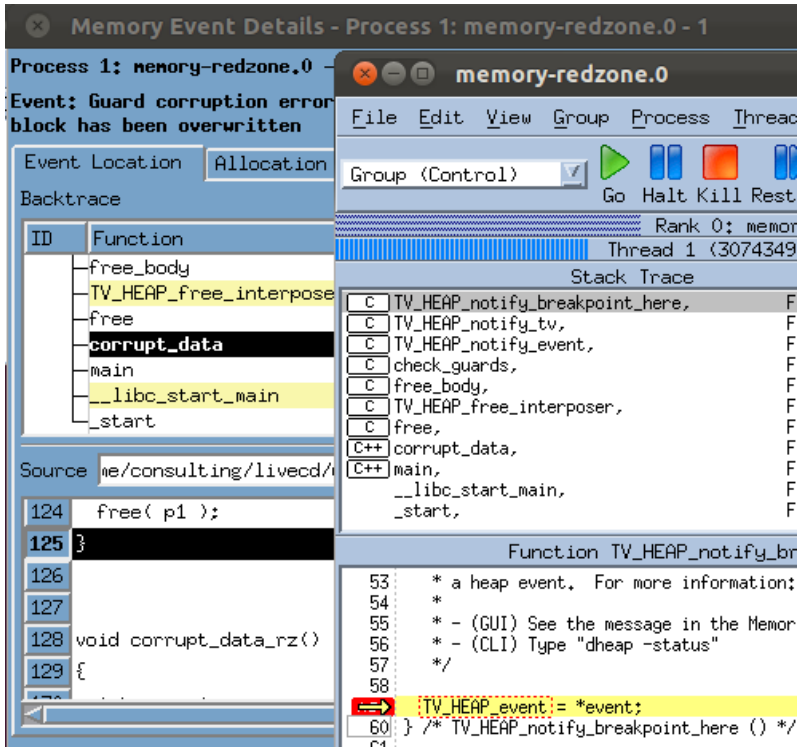
If that happens, select the process that is at the breakpoint, and press from the TotalView menu selection: **Process > Go**

If one of the processes reports a `double_free` event:

- Dismiss the **Event** Window
- Select the process that reported the event
- Press **Process > Go**

This will result in all processes being stopped at the active breakpoint at line 121 in the `corrupt_data` function.

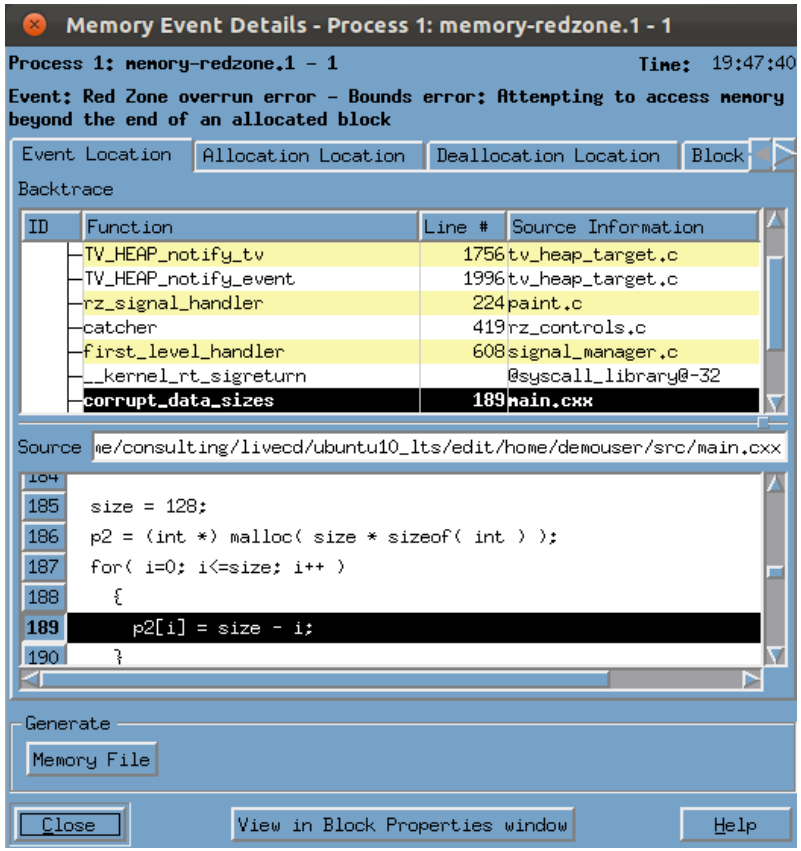
A Memory Event Details “Guard Corruption error” window will appear.



- Dismiss the **Event** Window
- Select the process that reported the event
- Press **Process > Go**
- Dismiss the new Memory Event Window "**Guard corruption error**." – corrupt_data_rz
- In the MemoryScape Window, click the **Memory Debugging Options** Tab
- Disable **Guard allocated memory**.

- In the MemoryScape Window, click the **Memory Debugging Options Tab**
- Select **`MPI_COMM_WORLD`** in the process selector
- Click **Advanced Options**, enable and expand the **Red Zones** option for **`MPI_COMM_WORLD`**
- Enable the **Restrict Red Zones** option
- Click the C**heck Box** of **Restrict red zones to allocation within ranges.**
- Then click on Ranges and in **Red Zone Range** Editor, enter **512** for Lower Limit, and **1020** for Upper Limit and Click **OK**
- In the **TotalView** Window, press **Go** 

A Memory Event Window will appear.



In the **Memory Event Details** Window, scroll up in the Source Pane

The Memory Event occurred in a function called `corrupt_data_sizes`. You can see that a larger and a smaller allocation were also overrun, but the size range restrictions caused those allocations not to be treated with Red Zones.

Question

10. Why wasn't a Red Zone event triggered as before in the `corrupt_data_rz` function?

_____

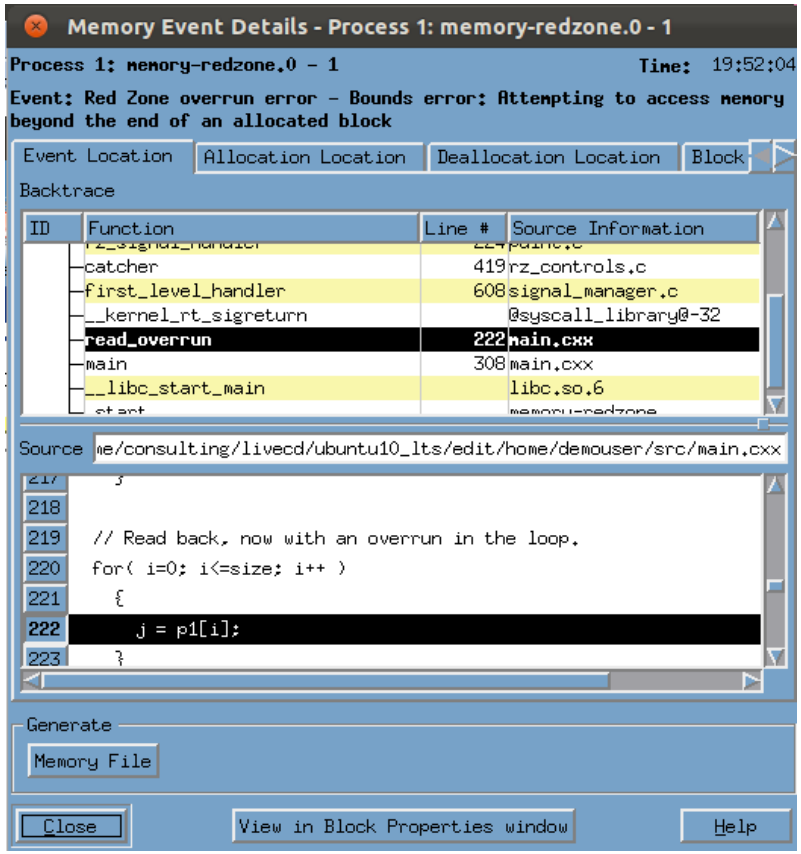
Step 4: Red Zones: Overrun Error

- Dismiss the **Memory Event Details** Window
- In the MemoryScape Window, disable **Use Red Zones for `MPI_COMM_WORLD`**
- In the **Halt Execution > Advanced Options** disable **Double_free** event
- In the Action Points Tab of the **TotalView** Window, enable the evaluation point for **line 298**
- Press **Restart** (if a confirmation box appears, click **Yes**)

This should result in all processes being stopped at the active breakpoint at line 121 in the `corrupt_data` function.

- In the MemoryScape Window, enable **Use Red Zones for `MPI_COMM_WORLD`**
- In the TotalView Window, press **Go** 

A Memory Event Window will appear.



Questions

11. What is different about this latest Red Zones event? Could the memory error in this case have been detected with Guard Blocks?
12. What are some ways of limiting memory space overhead when using Red Zones?

END OF LAB 6

Lab 7: Batch Mode Debugging with TVScript

TVScript provides non-interactive debugging with TotalView. It is especially useful in situations where a program is impractical to debug interactively (for example, due to lengthy run times or system access restrictions), and where debugging needs to be done repetitively (for example, parametric experiments or regression testing). This lab will familiarize you with TVScript operation and features, and will introduce some strategies for batch mode debugging.

Expected Time: 30 minutes

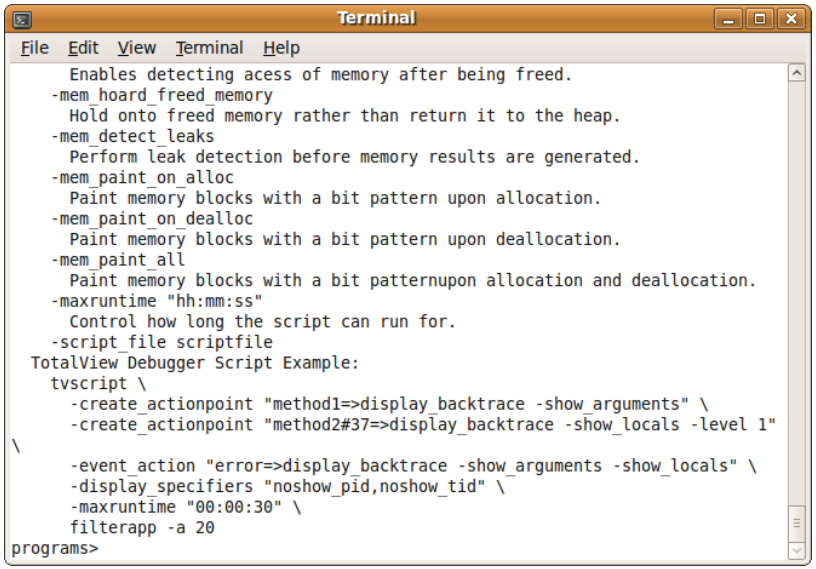
Step 1: Introduction

- Change directories to `$LABS` by typing:

```
cd $LABS
```

- Run TVScript with no arguments by typing:

```
tvscript
```



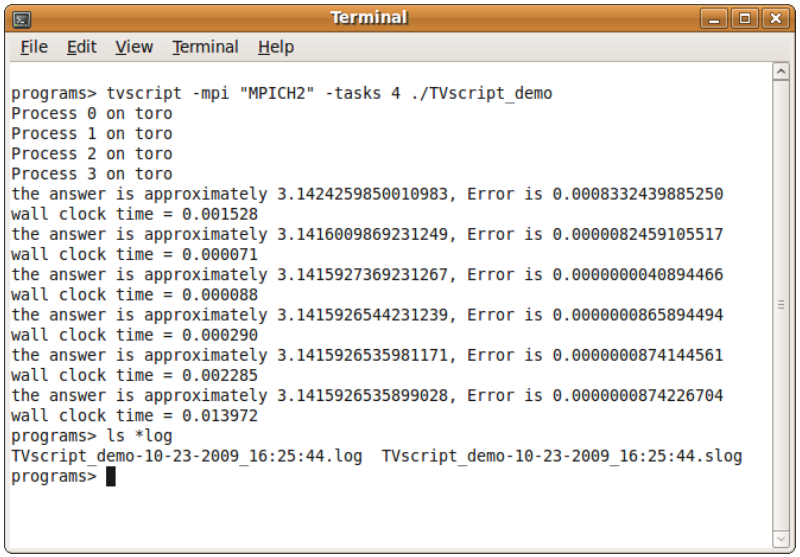
Although it isn't meant to be a substitute for the documentation, TVScript will output a summary of many of its options when invoked without arguments.

- Execute an example MPI program under TVScript, with four MPI ranks and with no debugging actions, by typing:

```
tvscript -mpi "MPICH2" -tasks 4
./TVscript_demo
```

- Examine how TVScript names its log files by typing:

```
ls *log
```



TVScript requires no interactive input once it has started. Unless it needs to report an error condition, it produces no interactive output. Output that the subject program sends to the `stdout` or `stderr` channels will be unchanged. TVScript always produces a summary log file (file extension `.slog`) showing what actions it took, and a detail log file (file extension `.log`) showing any output from those actions. Log file names include the subject program name, date, and time of day, which helps to make the logs from each run uniquely named. All these features make TVScript well-suited to running in batch mode.

Although no actions are recorded in these first log files, feel free to look at their contents with the "`cat`" command or any text editor.

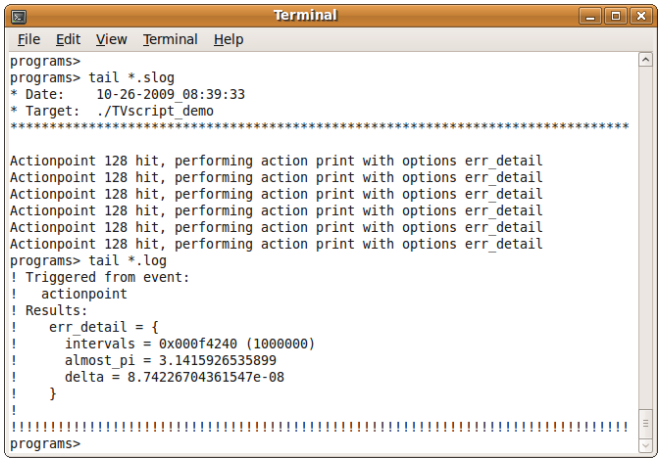
You may also find it helpful to view the source code of the example program with an editor that can display line numbers. The source code is available at this path: `../src/TVscript_demo.c`

The example program uses series expansions to estimate the value of pi. The series lengths are increased in several steps, which would be expected to make the estimates more accurate. That happens for the first few steps, but then the estimation errors stop decreasing, and even increase.

Use your imagination to picture how a problem like this might arise in actual practice. For example, a numerical simulation running on a large batch computing facility might start to show increasing errors only after running for a number of lengthy steps. We will look at a number of techniques that might be used with TVScript to debug in such a situation.

Step 2: Batch Mode Debugging

- Run the program under TVScript with an action by typing:
  `source TVcmd1` or just `./TVcmd1`
- Examine the summary and detail log files



`TVcmd1` is a small file that echoes and then executes a TVScript command. Since TVScript commands can be lengthy, this packaging was done to help the lab go smoothly, but you can try variations on your own. The packaged commands also delete pre-existing log files to make it convenient to find the new ones that are produced by the commands.

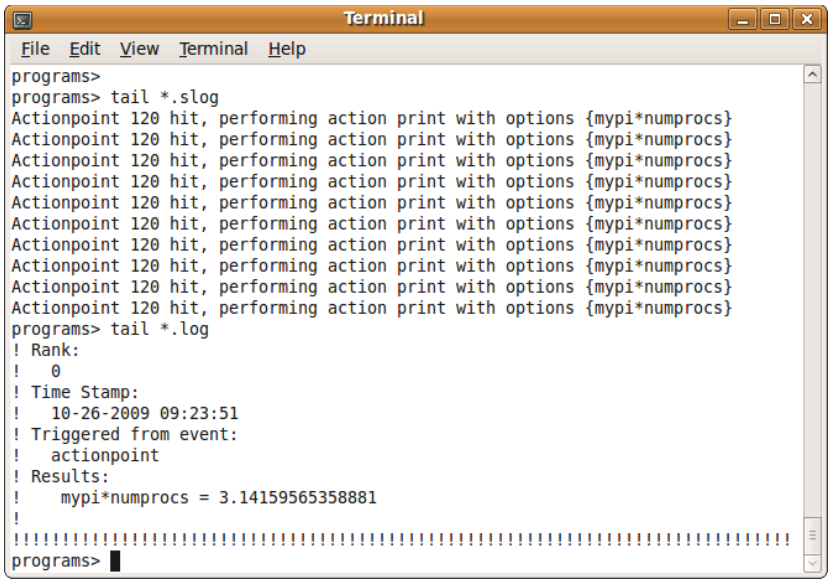
The summary log shows that the action point was hit six times, once for each step of increasing series lengths. The action point was set in code that only the MPI rank 0 process executes, so only that process took the action. The detail log shows output from an action that prints some program variables, one of the steps that would likely be taken in a real debugging situation.

Question

1. What are some ways in which examining these program variables with TVScript is more convenient than the conventional batch debugging practice of inserting `print` statements into the program?

_____

- Run the program under TVScript with an action by typing:
  `source TVcmd2` or just `./TVcmd2`
- Examine the summary and detail log files

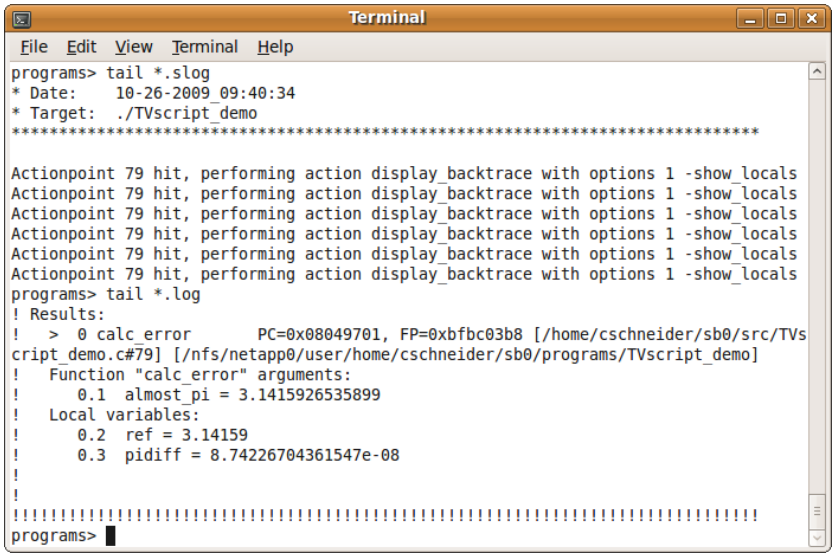


The summary log shows that the action point was hit for each program step, and in every rank. The detail log shows the output from each action, and includes identification of the rank. The output that TVScript was asked to print is the value of an expression in the native language of the program, doing a sanity check on each rank's contribution to the estimate of pi.

- Run the program under TVScript with an action by

typing:
`source TVcmd3` or just `./TVcmd3`

- Examine the summary and detail log files



The summary log shows that the action point was hit once for each program step. The detail log shows selected details of a backtrace of the program's call stack, including the arguments and local variables of the current function.

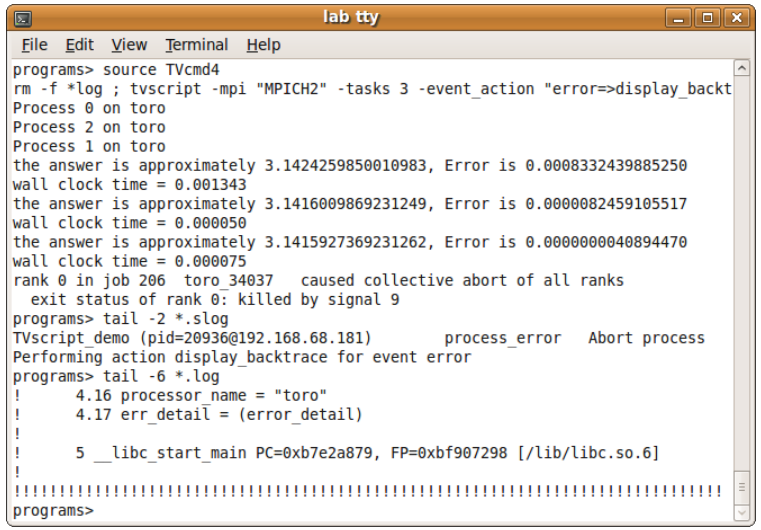
(Note that this is something not easily done with conventional batch debugging via `print` statements.) The function is intended to calculate the error in the estimated value of pi.

Question

2. What stands out in the output of the current estimate of pi (`almost_pi`) compared to the reference value (`ref`)?

Step 3: Batch Mode Debugging with Events

- Run the program under TVScript with an event action by typing:
  `source TVcmd4` or just `./TVcmd4`
- Examine the summary and detail log files



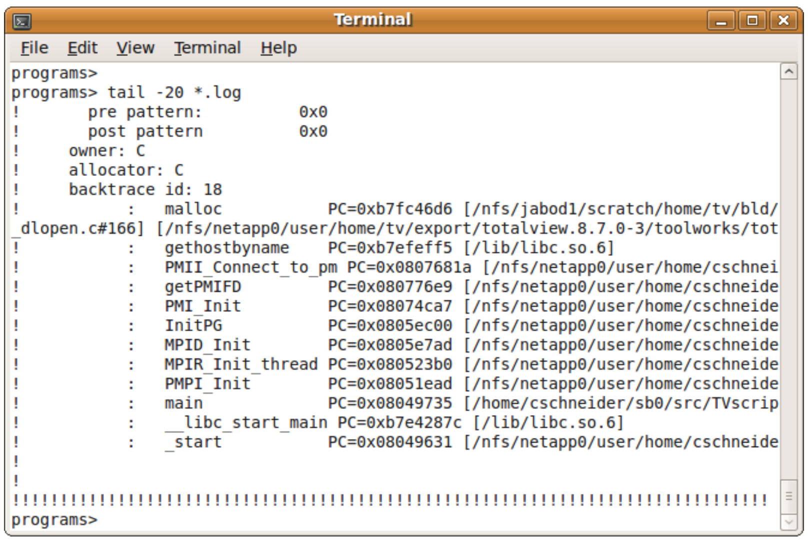
Here, TVScript was set up to detect and respond to an event, specifically an unhandled error. The program is coded to raise an error when run with three MPI ranks, but in general you would use a

similar TVScript setup to catch an asynchronous or unexpected error.

The summary log shows that the action point was hit by the process that encountered the error. The detail log shows a backtrace, with arguments and local variables printed for each level of the backtrace.

Step 4: Introduction to Batch Mode Memory Debugging

- Run the program under TVScript with a memory debugging action by typing:
  `source TVcmd5` or just ./`TVcmd5`
- Examine the summary and detail log files




```
programs>
programs> tail -20 *.log
!       pre pattern:          0x0
!       post pattern          0x0
!     owner: C
!     allocator: C
!     backtrace id: 18
!           :   malloc          PC=0xb7fc46d6 [/nfs/jabod1/scratch/home/tv/bld/
_dlopen.c#166] [/nfs/netapp0/user/home/tv/export/totalview.8.7.0-3/toolworks/tot
!           :   gethostbyname    PC=0xb7efeff5 [/lib/libc.so.6]
!           :   PMII_Connect_to_pm PC=0x0807681a [/nfs/netapp0/user/home/cschnei
!           :   getPMIFD         PC=0x080776e9 [/nfs/netapp0/user/home/cschneide
!           :   PMI_Init         PC=0x08074ca7 [/nfs/netapp0/user/home/cschneide
!           :   InitPG           PC=0x0805ec00 [/nfs/netapp0/user/home/cschneide
!           :   MPID_Init        PC=0x0805e7ad [/nfs/netapp0/user/home/cschneide
!           :   MPIR_Init_thread PC=0x080523b0 [/nfs/netapp0/user/home/cschneide
!           :   PMPI_Init        PC=0x08051ead [/nfs/netapp0/user/home/cschneide
!           :   main             PC=0x08049735 [/home/cschneider/sb0/src/TVscrip
!           :   __libc_start_main PC=0xb7e4287c [/lib/libc.so.6]
!           :   _start           PC=0x08049631 [/nfs/netapp0/user/home/cschneide
!
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
programs>
```

The summary log shows that the action point was hit once for every rank (at program exit). The detail log lists the memory allocations belonging to the program at that time.

While the example program isn't an interesting memory debugging subject, this step highlights that memory debugging is available in batch mode with TVScript (or MemScript, when only memory debugging actions are needed).

END OF LAB 7

Lab 8: Reverse Debugging with ReplayEngine

ReplayEngine provides reverse debugging features to TotalView. By recording program execution history and allowing the user to play it back, reverse debugging can accelerate the solution of many types of code problems. This lab will introduce you to the basics of navigation in reverse debugging, and familiarize you with ReplayEngine features and strategies that help with complex applications.

Expected Time: 45 minutes

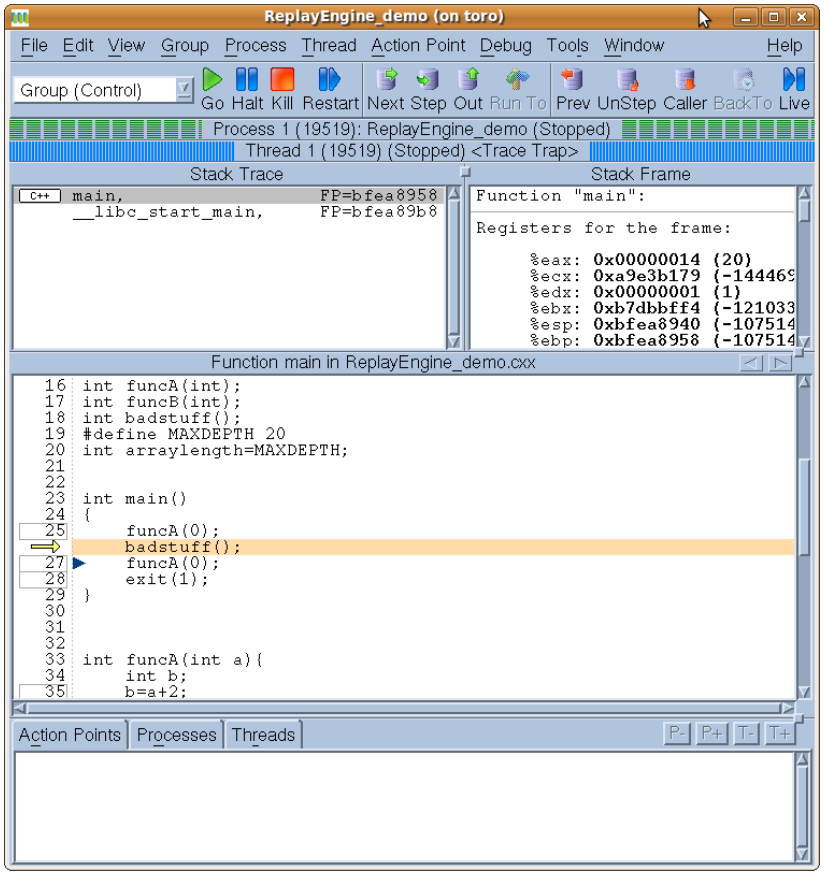
Step 1: Start TotalView

- Change directories to `$LABS` by typing:
  `cd $LABS`
- Start TotalView with ReplayEngine by typing:
  `totalview -replay ReplayEngine_demo`

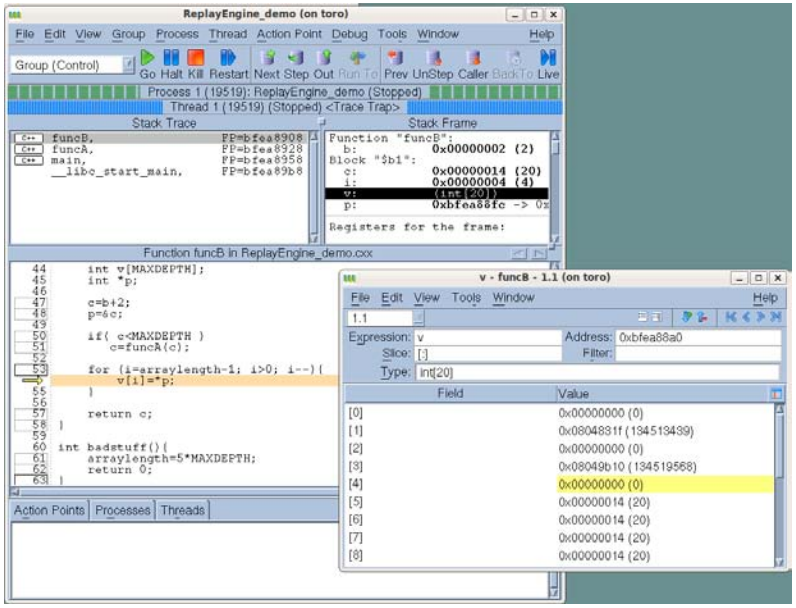
Step 2: Reverse Navigation

- In the Process Window, highlight **line 27** (second call to `funcA`)
- Press **RunTo**
- Press **Prev**

After the application ran forward and stopped at line 27, ReplayEngine replayed execution back to line 26. The black arrow marks where forward execution stopped. The familiar yellow arrow indicates the point to which execution was replayed, and that source line is also highlighted in orange to indicate replay mode.



- Press **UnStep** four times, observing after each time how the replay location changes
- Dive on the `v` array to open a Variable Window
- Press **UnStep** repeatedly until several iterations of the `for` loop have been traversed, noting changes in variables



- Highlight **line 48**
- Press **BackTo**

Note that the stack trace is now deeper. This is because of recursive calling in `funcA` and `funcB`.

- Dismiss the Variable Window with the `v` array
- Set a breakpoint on **line 57**, the return statement of `funcB`
- Press **Go** 

Question

1. Is debugging still in replay mode? How can you tell?

_____

- Delete the breakpoint
- Press **Caller**

Question

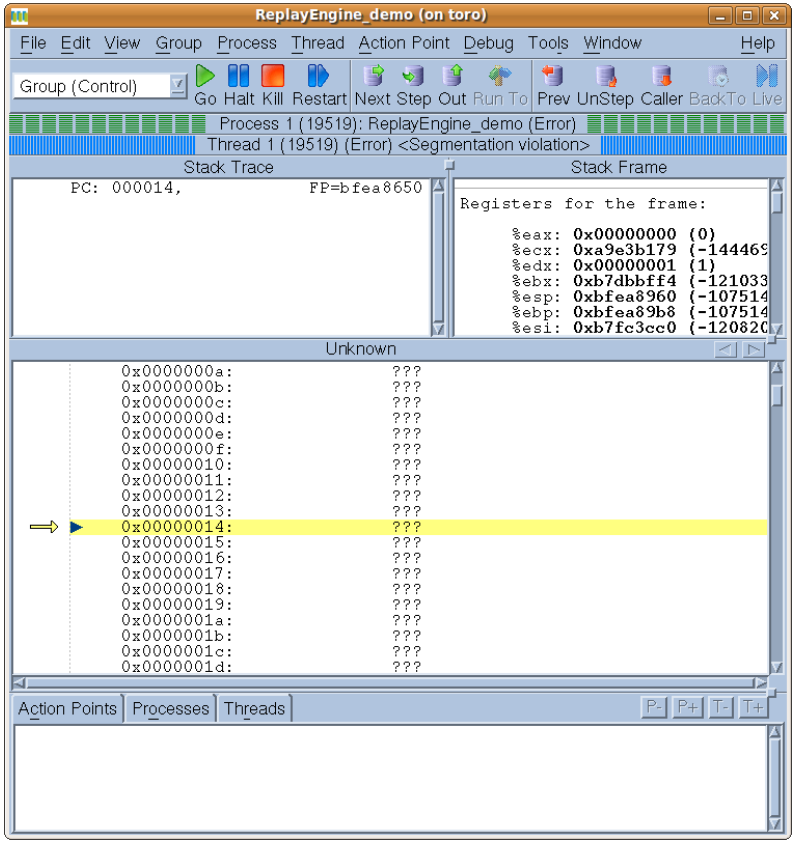
2. What forward debugging actions are analogous to the reverse debugging actions that were used so far?

_____

Step 3: Reverse Debugging a Stack Corruptor

- Press **Live**

- Press **Go** 

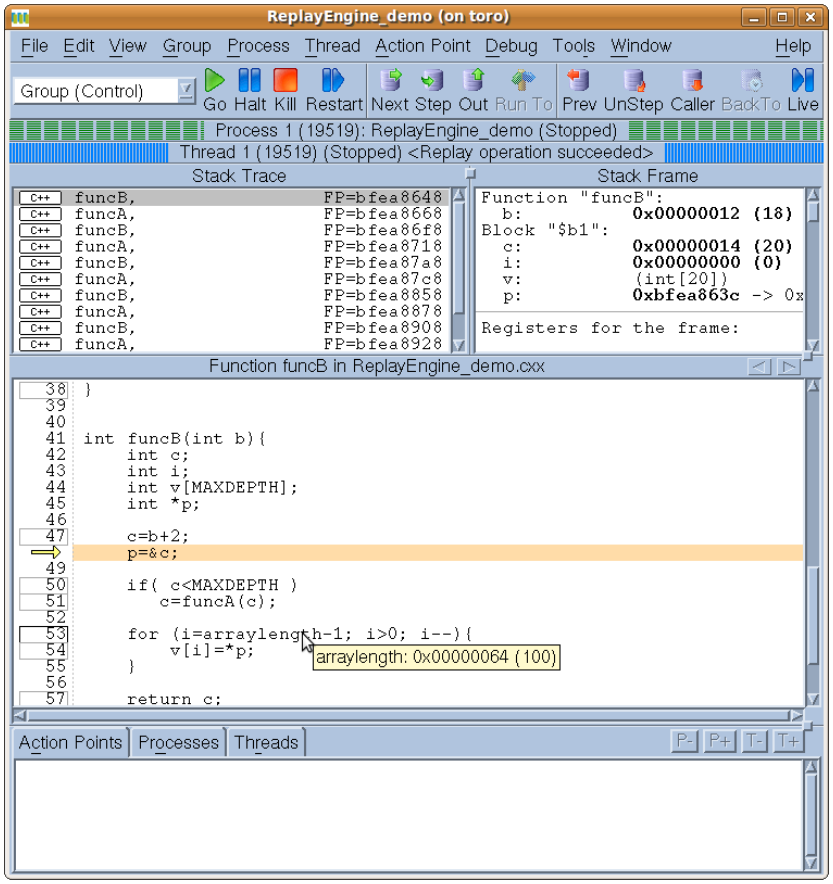


- Press **UnStep**
- Highlight **line 48**
- Press **BackTo**
- Hover over **`arraylength`** to see its value, and compare it with the declared size of the `v` array

Question

3. What is the most likely reason for the loss of program position information?

_____



The proximate cause of the symptom (segmentation violation) is now evident, but the root cause (improper value of `arraylength`) remains obscure.

- Set a watchpoint on arraylength
- In the Stack Trace Pane, click on **`main`**
- Highlight **line 25** (first executable line in `main`)

- Press **BackTo**
- Press **Go**

Question

4. A deterministic bug is possible to find with forward debugging. In this case, once forward debugging had established that the symptom occurs in `funcB`, the rest of the effort would have been similar to using reverse debugging. (That is, most likely a watchpoint would have been set on `arraylength` to lead to the root problem.) What are some advantages of reverse debugging in this case?

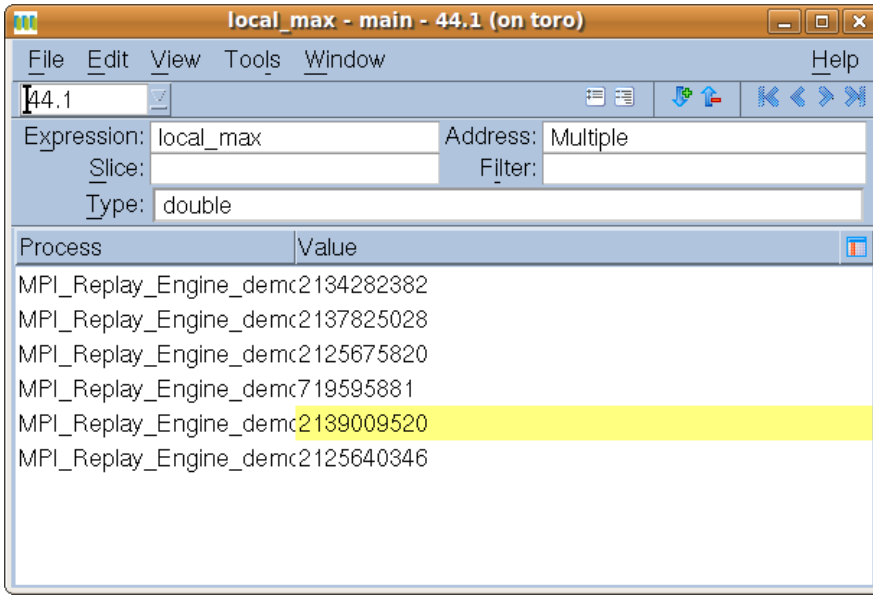
_____

Step 4: Reverse Debugging a Nondeterministic Parallel Program

- Press **Kill**
- Dismiss the **ReplayEngine_demo Process Window**
- Select **File > New Program**

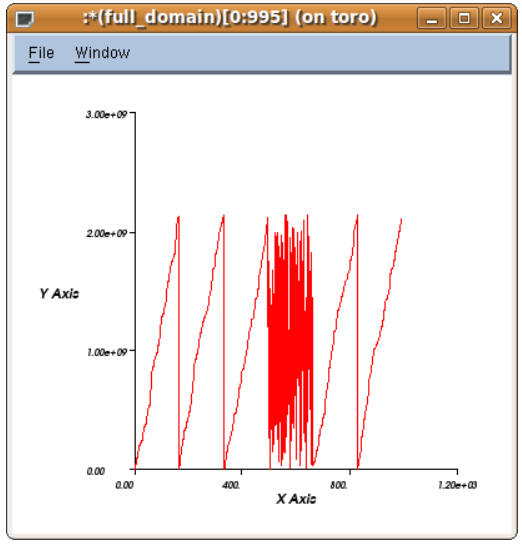
- In the Program entry, browse and select **`MPI_Replay_Engine_demo`**
- Open the **Parallel Tab**
- Select `MPICH2` for the Parallel System
- Set the Tasks entry to **6**
- Press **OK**
- Set a breakpoint at **line 89** (call to `MPI_Reduce`)

- Press **Go** 
- Dive on **`local_max`**
- In the Variable Window, select **View/Show Across/Processes**
- In the Process Window, press **Restart** a few times (if a confirmation box appears, press Yes) and observe the changes in the Variable Window



You should notice that typically most of the processes have values of `local_max` which are around two billion, but there are often one or more processes with a substantially lower value. The distribution of typical or low values across processes is different from run to run.

- Repeat restarts, if necessary, until the Variable Window shows at least one value of `local_max` that is substantially less than two billion
- In the Process Window, highlight **line 100**
- Press **Run To**
- Select the MPI **`rank 0`** process using the P-/P+ buttons
- Dive on **`full_domain`**
- In the `full_domain` Variable Window, dive on the **pointer**
- Cast the type to **`double[996]`**
- Select **Tools/Visualize**



The design of the program is that, at this point, each rank should have completed a sort on its subdomain of data. The `rank 0` process has gathered the subdomains into the `full_domain` array.

Question

5. Can you characterize how the program is failing to operate as designed? Is the misbehavior repeatable from run to run?

_____

- In the toolbar of the Process Window, select **Rank 0** from the pull-down list (to make it the focus for toolbar operations)
- Dive on the **`getMax`** function, which is called at line 88
- Highlight **line 144** (call of the `qsort` function)
- Press **BackTo**

Note that the replay succeeded, as indicated by the orange high-lighting of line 144. Also, if you examine the `local_max` Variable Window, you will see that the Rank 0 value has changed.

- Identify one of the ranks with a substantially lower value of `local_max`, and select that rank in the Process Window, with the P-/P+ buttons; (note that the toolbar focus changes accordingly)
- Dive on the **`getMax`** function, which is called at line 88
- Highlight **line 144** (call of the `qsort` function).
- Press **BackTo**; a warning window will appear

Questions

6. What does the warning mean?

_____

- Dismiss the warning window
- Highlight **line 142**
- Press **BackTo**

7. How does the use of ReplayEngine enhance the debugging of non-repeatable bugs?

_____

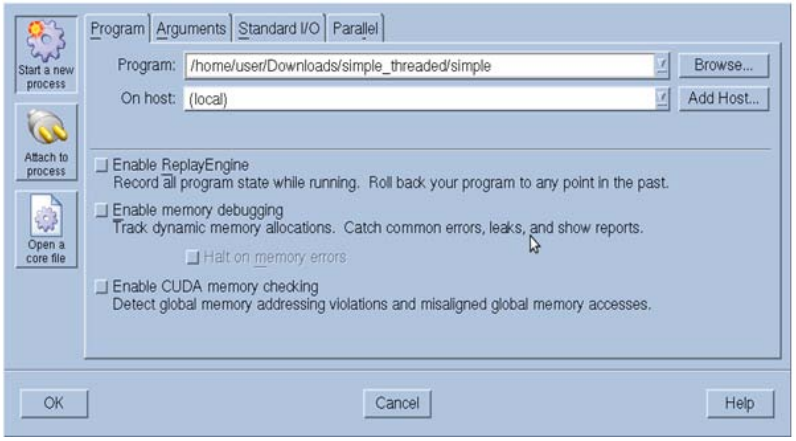
END OF LAB 8

Lab 9: Asynchronous Control Lab

Using the multi-threaded example program, 'simple', we can demonstrate group, process, and thread control contexts.

Step 1: Start TotalView

- Start up the debugger and **Browse** to location of the `**simple**' program to open. Click the **OK** button.

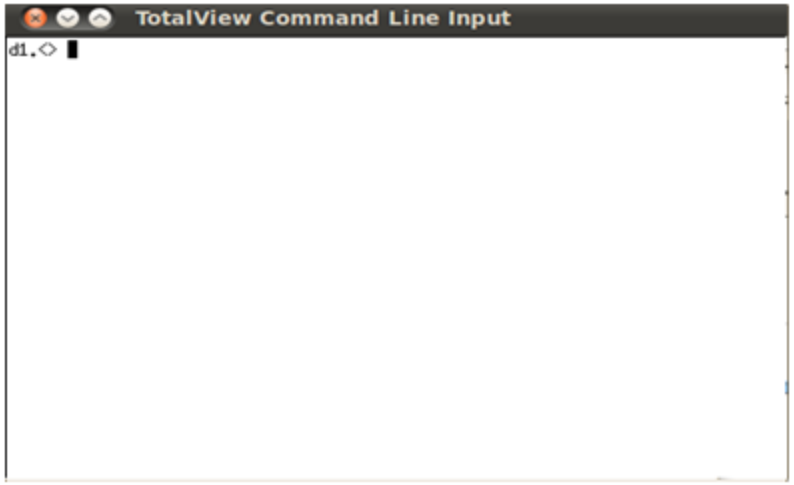


Step 2: Start Command line debugger.

- Once the main window opens go to the Tools menu item and choose **Command Line** at the bottom of the menu box.

This will open an xterm window where debugger commands can be entered.

This will open an xterm window where debugger commands can be entered.



- Start up the debugger and browse to location of the `**simple**' program to open. Click the **OK** button.

Unless noted otherwise all commands will be entered into the xterm command line window.

```
d1.<> dgroups -l *
1: {control 1}
2: {workers}
3: {share 1}
d1.<>
```

- Now put a break at main.

```
d1.<> dbreak main
1
d1.<>
```

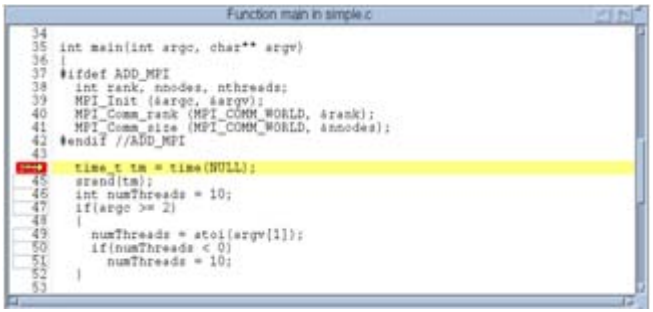
Notice in the main window of the GUI that the breakpoint will appear.

Function main in simple.c

```
34
35 int main(int argc, char** argv)
36 {
37 #ifdef ADD_MPI
38   int rank, nnodes, nthreads;
39   MPI_Init (&argc, &argv);
40   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
41   MPI_Comm_size (MPI_COMM_WORLD, &nnodes);
42 #endif //ADD_MPI
43
STOP  time_t tm = time(NULL);
45   srand(tm);
46   int numThreads = 10;
47   if(argc >= 2)
48   {
49     numThreads = atoi(argv[1]);
50     if(numThreads < 0)
51       numThreads = 10;
52   }
53
```

- Start the program and it will stop at the breakpoint, and there will be a pointer where execution has stopped.

```
d1.<> drun
Thread 1.1 has appeared
Created process 1 (3834), named "simple"
Thread 1.1 has appeared
Thread 1.1 has exited
Thread 1.1 hit breakpoint 2 at line 44 in "main"
d1.<>
```



- Now list the groups once again.

```
d1.<> dgroups -l *
1: {control 1}
2: {workers 1.1}
3: {share 1}
d1.<>
```

The new listing shows {workers 1.1}. 1.1 is the main thread.

- Remove the main thread from workers group.

```
d1.<> dgroups -remove -g 2 1.1
d1.<> dgroups -l *
1: {control 1}
2: {workers}
3: {share 1}
d1.<>
```

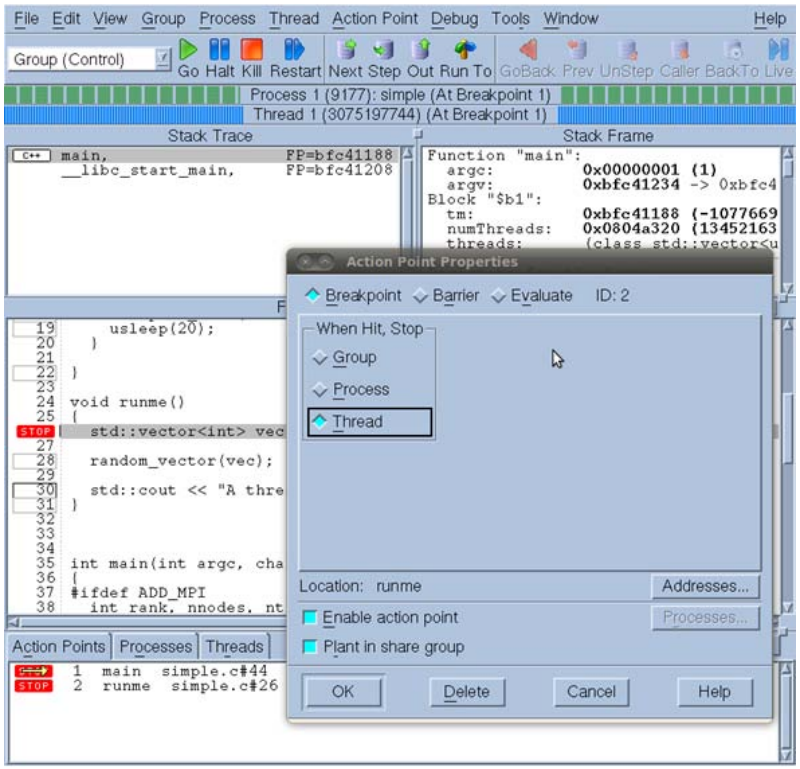
The -g switch is the group ID, in this example: 2: {workers 1.1} . 1.1 is the thread ID which is the main thread.

- Put a breakpoint at the runme function.

This is the start of each thread that will be created. The simple program creates 10 threads by default.

```
d1.<> dbreak runme
2
d1.<>
```

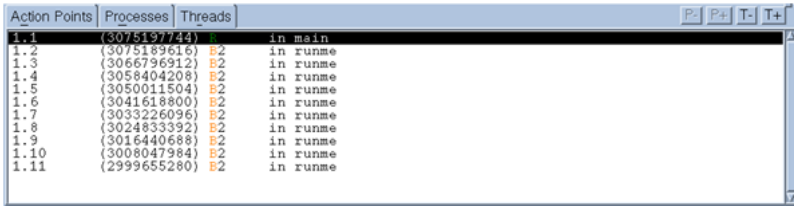
- Now go to the GUI and find the breakpoint at the runme function. Right click on this and choose properties. In the "**When Hit, Stop**" box choose Thread. Click **OK**.



Main is going to create 10 threads.

- In the GUI Press **Go**  .

This will create all the threads and stop at the breakpoint for each thread at the runme function. The threads are listed at the bottom of the GUI. As a reminder, the 1.1 thread is the main process.



.

- And from the command line debugger, list the groups again

All the threads are in the workers group.

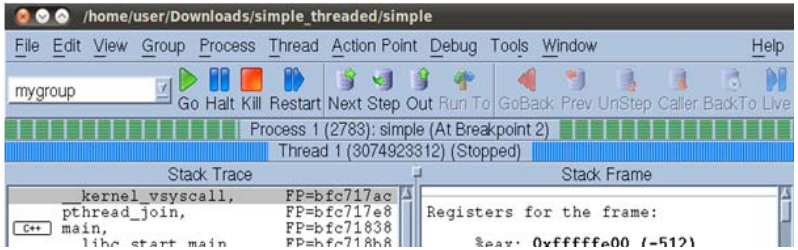
```
d1.<> dgroups -l *
1: {control 1}
2: {workers 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 1.10
1.11}
3: {share 1}
d1.<>
```

- Now a new group can be created and we'll move some threads to this new control group. You'll also have to remove the same threads from the workers group.

```
d1.<> dset -new mythreads {1.2 1.4 1.7 1.11}
1.2 1.4 1.7 1.11
d1.<> dgroups -new t -g mygroup $mythreads
mygroup
d1.<> dgroups -remove -g 2 {1.2 1.4 1.7 1.11}
d1.<> dgroups -l *
1: {control 1}
2: {workers 1.3 1.5 1.6 1.8 1.9 1.10}
3: {share 1}
mygroup: {thread 1.2 1.4 1.7 1.11}
d1.<>
```

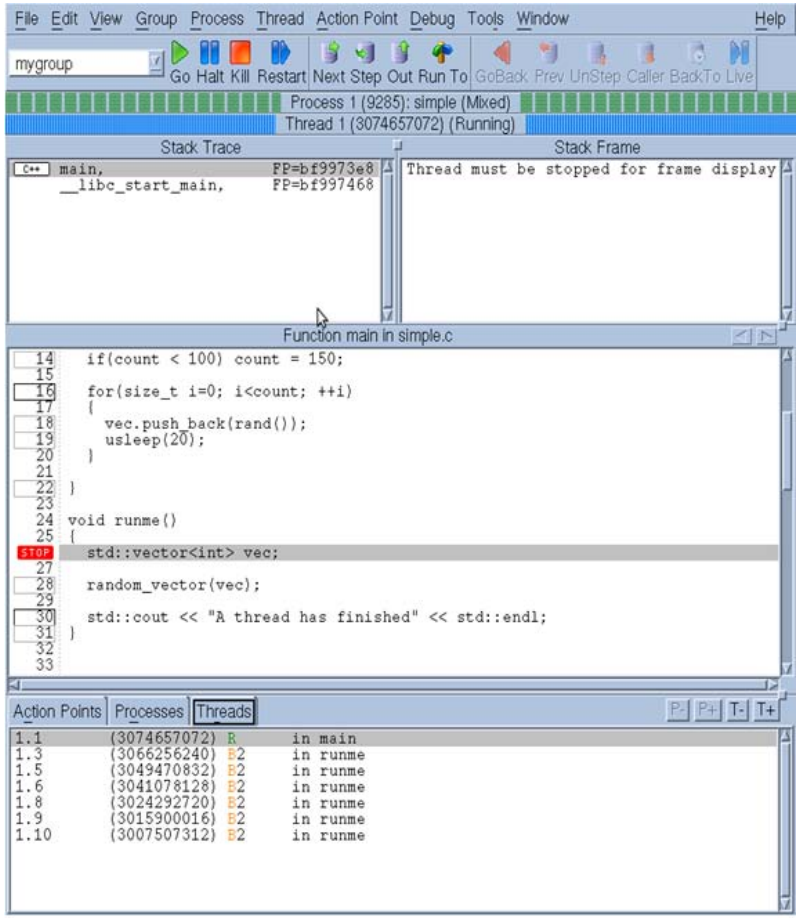
There's a drop down list box in the upper left portion of the GUI that lists all the group types.

- View the list and choose the `mygroup' group that was created.



- Then press the **GO** button  (it has to be pressed twice).

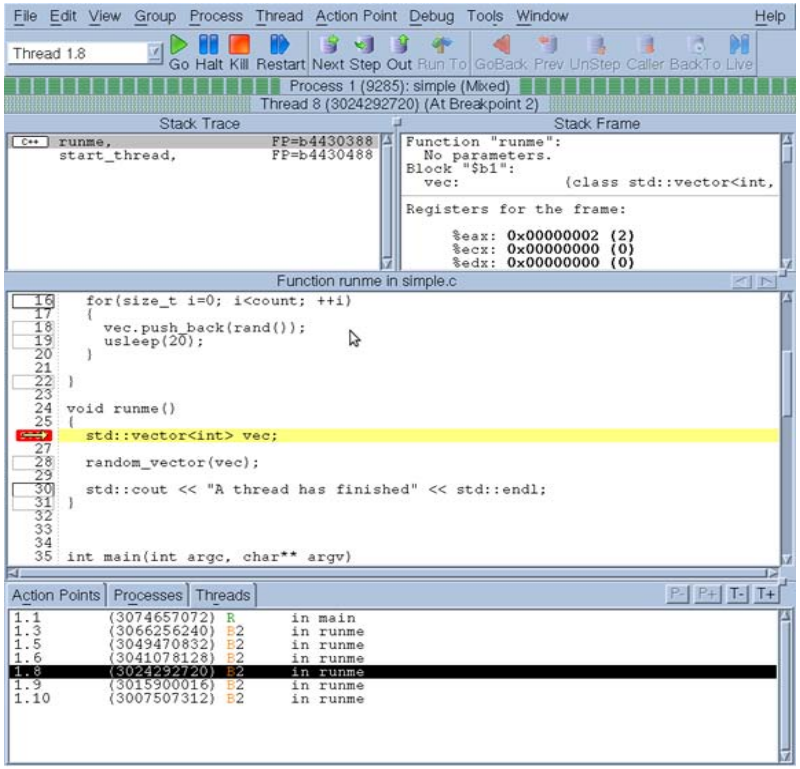
This will run all of the threads in `mygroup'. You can see in the bottom box of the GUI that those threads of this group are gone. They've all completed their execution while all other threads in the `workers' group are held.



Now you can step through an individual thread

- Click on one of the threads to highlight at the bottom of the GUI.
- Then go to the group menu list box at upper left of GUI. This highlighted thread should be near the bottom of the list. Click on it to make it the focus of execution.
- Now you can click on the **next/step** button(s) to execute through this specific thread.

.



- Experiment with switching the focus to each thread and stepping through.

END OF LAB 9