Frysk: Debugger/Monitoring Tool

Overview

The primary goal of the Frysk project is to provide an intelligent, state-of-the-art tool for developers and systems administrators that enables them to monitor the inner-workings of processes within a system or multiple systems and alert them if/when problems occur. Frysk is designed to "observe" processes as unobtrusively as possible and detect problems at a point in time when the user can interrogate the errant process to get useful information before it exits the cpu queue. This information will help either resolve the problem or will result in data that will help narrow the problem down to where the next debug run can be more intelligently instrumented so it will harvest even more and better data or resolve the problem altogether.

The key difference between Frysk and most of the available open source Linux debuggers is that once the errant process has been identified, the user can then utilize Frysk to attach "observers" to it that will stop the process at any sign of trouble or a user-defined behavior is detected. In the case of an errant process, it is stopped while useful information can be retrieved via backtraces, retrieving variable values, setting breakpoints, etc. When a Frysk-observed process misbehaves, the user can have Frysk perform several actions including having e-mails sent, having warning windows appear, having a "debug" window appear with the source code in it that the user can set breakpoints, look at variable values, etc. and/or having other processes/scripts activated and a whole list of other options.

Frysk is basically divided into two major pieces of functionality: a monitoring function where the user can watch the actions of various processes through a feature called "observers"; and a debugger that the user can activate either via an observer from the monitoring side of Frysk or as an independent task that has no connection to the monitoring side.

Observer Concept

The "observer" model Frysk uses is based on the Java "Observable" class. Basically the theory of operation here is that when an "observer" is attached to a process and/or a thread within a process(from now on when process is mentioned, the entire process or a thread within a process is implied), it reports back to the process that initiated the observer(Frysk in this case) that the behavior has occurred.

Now the question is, what kinds of observers may be attached? The following observers may currently be attached to a process:

Exec – Monitor a process for a call to the "exec" function

Fork – Monitor a process for a call to the “fork” function

Task Clone – Monitor a process for a clone operation

Task Terminating – Monitor a process for any activity for it to exit the cpu queue

Syscall – Monitor a process for any system call

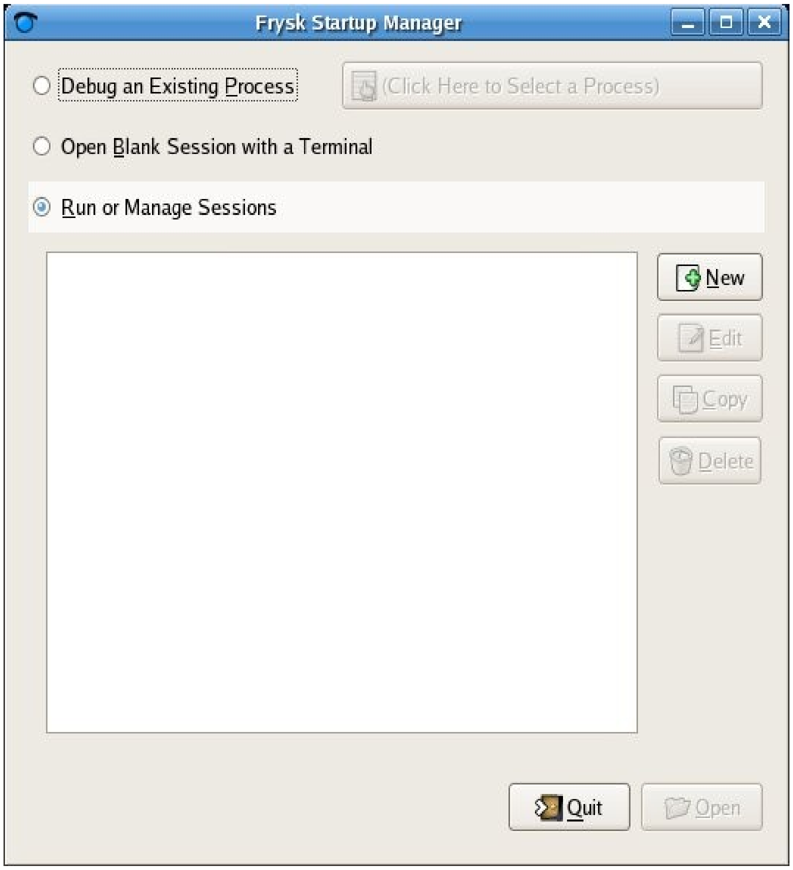
Custom – Apply one of the above filters with filtering

When these observers are attached to a process, the user can define how they want Frysk to respond to the observer being triggered. These user-defined responses are enumerated in the previous section and described later in this document.

The **custom observer** is basically any of the other five observers with filtering applied. For example, if the user wants to monitor a task with an **exec observer**, but only wants an action to occur only when a certain task or certain tasks are exec'ed, then the **custom observer** would be used. If the user wants an action to occur when any exec call is made, the **exec observer** would be used.

Defining a Debug Session

On a system that Frysk has been installed on via an rpm, start Frysk from the “System Tools” menu by clicking on the Frysk icon. Here is the first window that appears for Frysk.

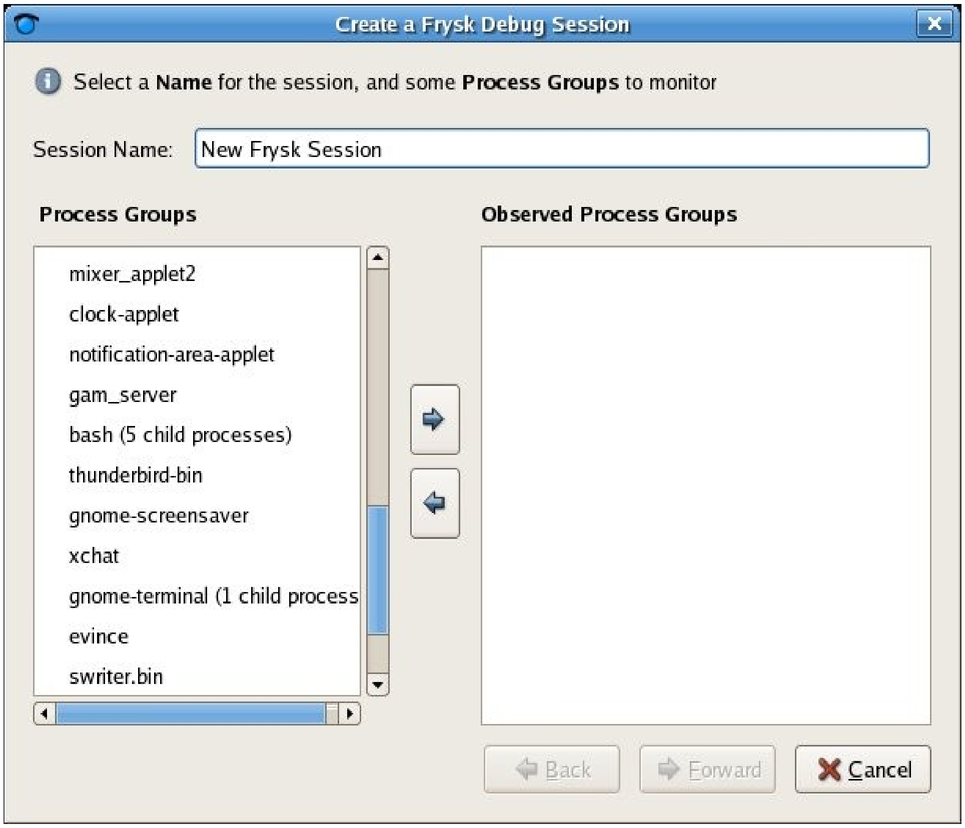

Frysk Startup Manager
○ Debug an Existing Process
(Click Here to Select a Process)
○ Open Blank Session with a Terminal
● Run or Manage Sessions
New
Edit
Copy
Delete
Quit
Open

Frysk Initial Start-up Screen

This is the initial Frysk start-up screen. From this screen the user may launch into either the monitoring part of Frysk or the debug part of Frysk. This document will begin with the monitoring side.

The default selection is for the Frysk monitoring side as indicated by the **Run or Manage Sessions** radio button being selected. Since there are currently no sessions defined as indicated by the blank white area of the screen, debug sessions must be defined before monitoring can occur. Basically, a "debug session" consists of a process or group of processes that a user has attached observers to. Several debug sessions can be defined.

So, to start the monitoring part of Frysk, at least one debug session must be defined. The user can define a debug session by clicking the **New** button to cause the following window to appear.



Select a **Name** for the session, and some **Process Groups** to monitor

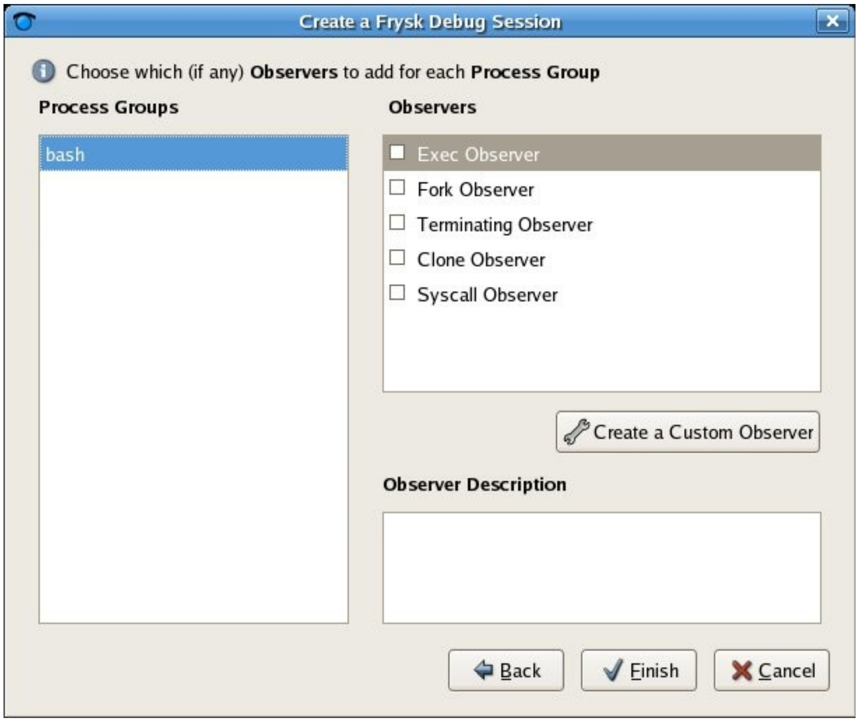
Session Name: New Frysk Session

Frysk Create Debug Session

From this window a debug session can be created by selecting the process or group(s) of processes. The list of processes is derived from the cpu queue list of processes that the current user owns. (Frysk does not allow a user to monitor/debug processes that they do not own.) For example, if **bash** was chosen from the list on the left, all of the child processes that bash has activated would also be selected.

So, select the process(es) of interest from the **Process Groups** list and place the in the **Observed Process Groups** list by either double-clicking them or single-clicking and then clicking on the right-arrow button. Processes mistakenly selected can be removed by selecting them and clicking the left-arrow button.

So, select a process and place it into the **Observer Process Groups**. The **Forward** button now becomes active enabling the user to progress to the next step in defining a debug session. Clicking on that button brings up the next window enabling the user to attach whatever observers they desire.



Choose which (if any) **Observers** to add for each **Process Group**

bash

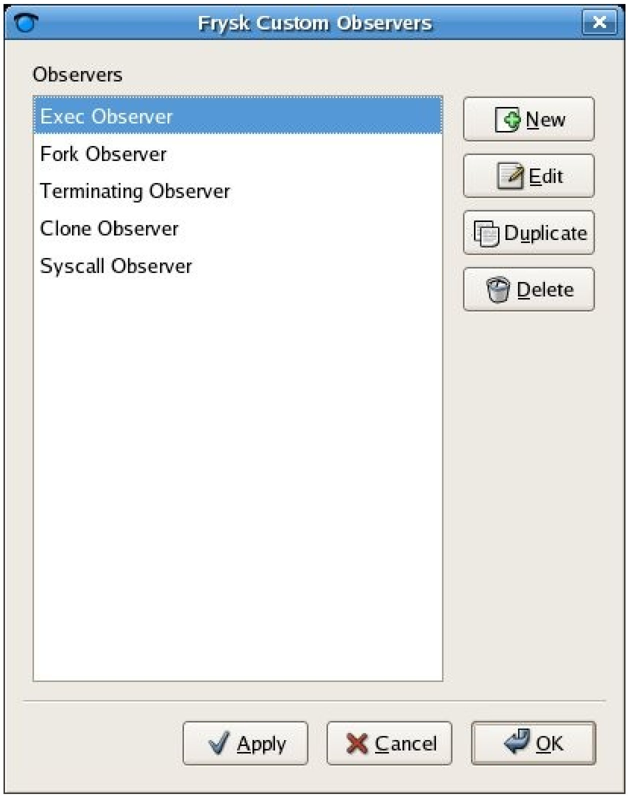
☐ Exec Observer
☐ Fork Observer
☐ Terminating Observer
☐ Clone Observer
☐ Syscall Observer

Create a Frysk Debug Session (Page 2)

This window is where the **Observers** get attached to the Process Groups. If there are multiple **Process Groups**, each one can have its own unique set of observers. If all a user wants to do is monitor a process group with one of the five observers and have a log or graph of each time a selected observer “fires”, then simply select the desired observer(s) and click **Finish**.

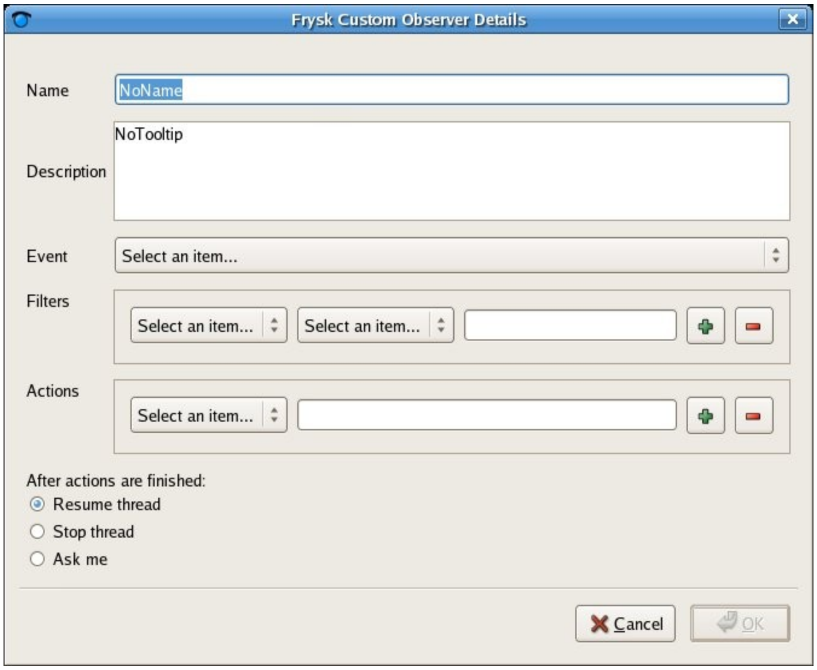
Custom Observer Creation

If a user wants to do more than log/graph observer firings, then a **custom observer** must be created. This allows more control over an observer by allowing filters to be applied that more narrowly focuses one of the five other observer and gives much more precise control over when the observer fires. Click on the **Create a Custom Observer** button to begin the custom observer creation process.



Create Custom Observer First Window

From this window the user is given several options. Usually the five core observers shown here should be left alone and a new observer created. Click the **New** button to create a totally new observer from the following window.



Name: NoName

Description: NoTooltip

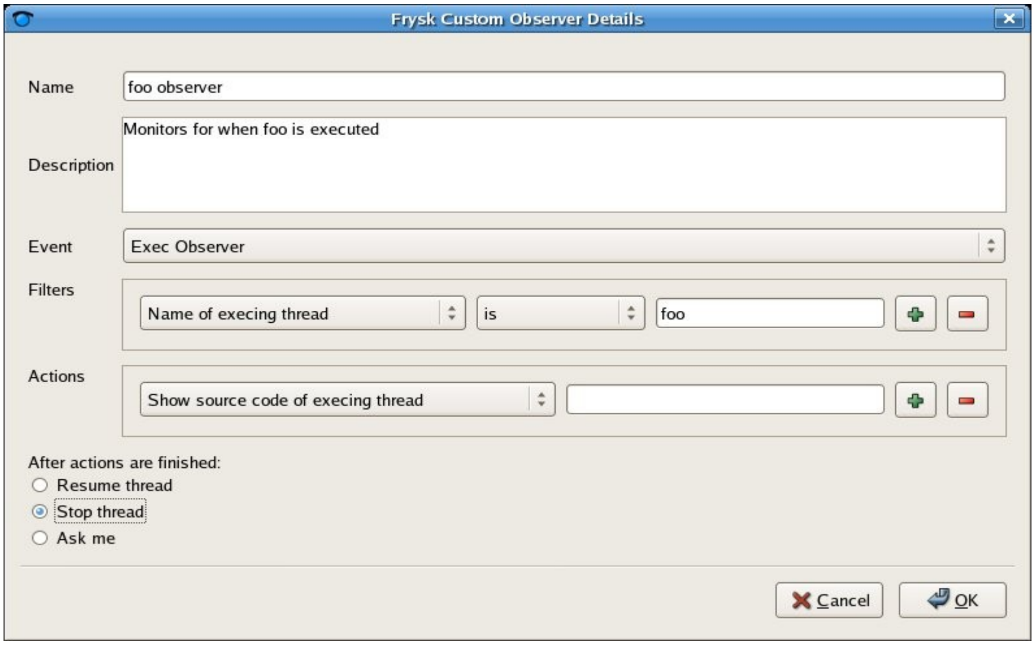
Event: Select an item...

After actions are finished:
● Resume thread
○ Stop thread
○ Ask me

Frysk Custom Observer Details Window

From this window the name of the custom observer is defined and a brief description can be entered in the first two fields. The **Event** field is where one of the five core observers is chosen as a basis for this observer. The **Filters** field is where one or more filters may be applied to control when this observer fires. As many filters as is desired may be applied for a custom observer. Just click the "+" button to add more filters and click the "–" button to delete the currently-selected filter. The **Actions** field allows the definition of an action or a series of actions to be performed when the observer fires.

For example, suppose a user wanted to observe a process and stop it and look at the source code whenever it tried to execute a process named **foo**. The **Frysk Custom Observer Details** window would look something like the following.



Name: foo observer

Description: Monitors for when foo is executed

Event: Exec Observer

Filters: Name of execing thread | is | foo

Actions: Show source code of execing thread

After actions are finished:
○ Resume thread
● Stop thread
○ Ask me

Example of a Defined Custom Observer Based on an Exec Observer Filtered on Foo

The above-defined custom observer, when attached to a process, looks for that process to exec another process named **foo** and when that occurs the observer would perform the selected action. In this case the custom observer would bring up a source window showing the code that is being executed and the process exec'ing **foo** would be stopped per the instructions defined for **After actions are finished**. As mentioned earlier, more filters/action can be applied. For example, if the user wanted to also watch for any occurrences of the process **bar** being executed, another filter could be added the same as for **foo**. So the process would be stopped upon any execution of either **foo** or **bar**.

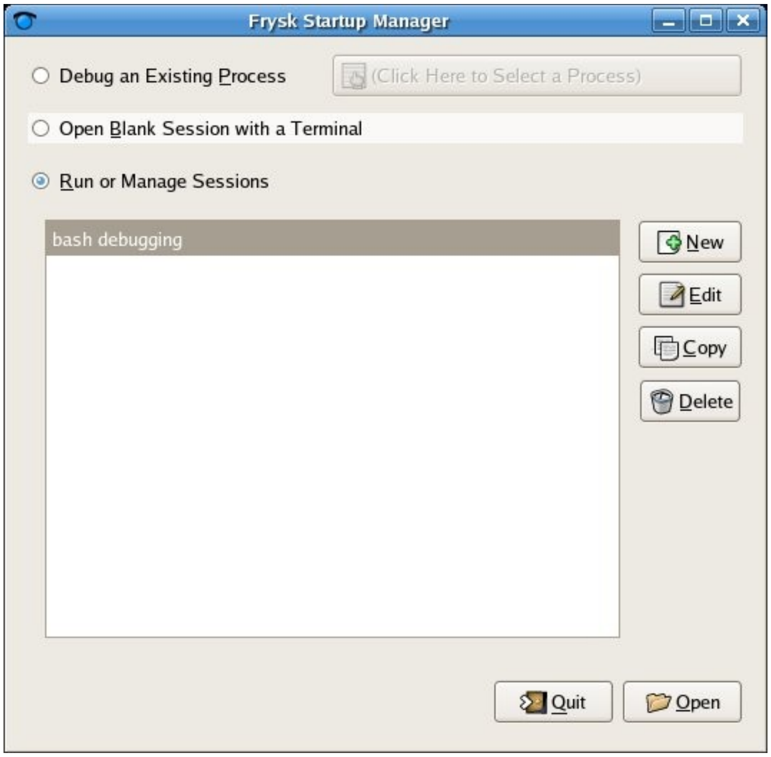
Logic can also be reversed for filters. That is, if you wanted to stop the process upon execution of any process other than bar, the filter could be changed from **is** to **is not**. There are also several other actions that can be performed when the observer fires such as printing the state of the exec'ing thread, showing the memory or register window of the executing thread, and others.

As can be seen from this example, custom observers provide very powerful capabilities to the user. This feature is a great differentiator from the rest of the open source debuggers and adds a great deal of utility for Frysk.

When all definitions are complete, click on **OK** to complete the definition of this custom observer and return to the **Frysk Custom Observers** window where it should be listed in the **Observers** pane. This observer is now available to be attached to any process or group of processes just like any of the core observers.

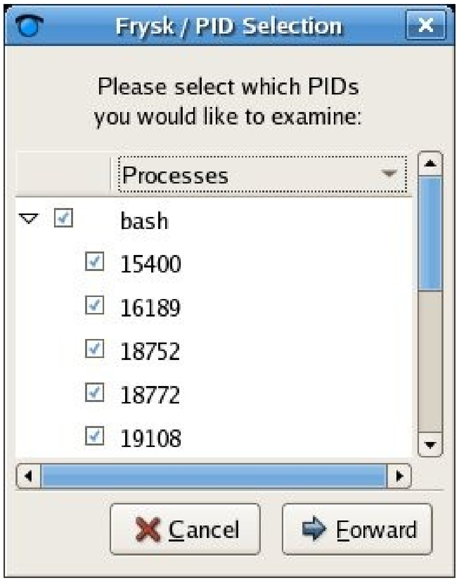
Starting a Debug Session

Once at least one debug session has been defined, the monitoring part of Frysk can be activated. So to start a debug session, make sure the **Run or Manage Sessions** radio button is selected, then select the desired session in the window of the startup screen and either double-click it or single-click the session and then click **Open**.


Frysk Startup Manager
○ Debug an Existing Process
(Click Here to Select a Process)
○ Open Blank Session with a Terminal
● Run or Manage Sessions
bash debugging
New
Edit
Copy
Delete
Quit
Open

Frysk Startup Manager with a Debug Session Defined

Once a debug session has been selected and opened, the monitoring part of Frysk begins. Sometimes an interim window called a **Process Picker** will appear if there are multiple PID's in the cpu queue with the same name. In the above example a **bash** observer has been selected and when the **Open** button is clicked the following screen appears.

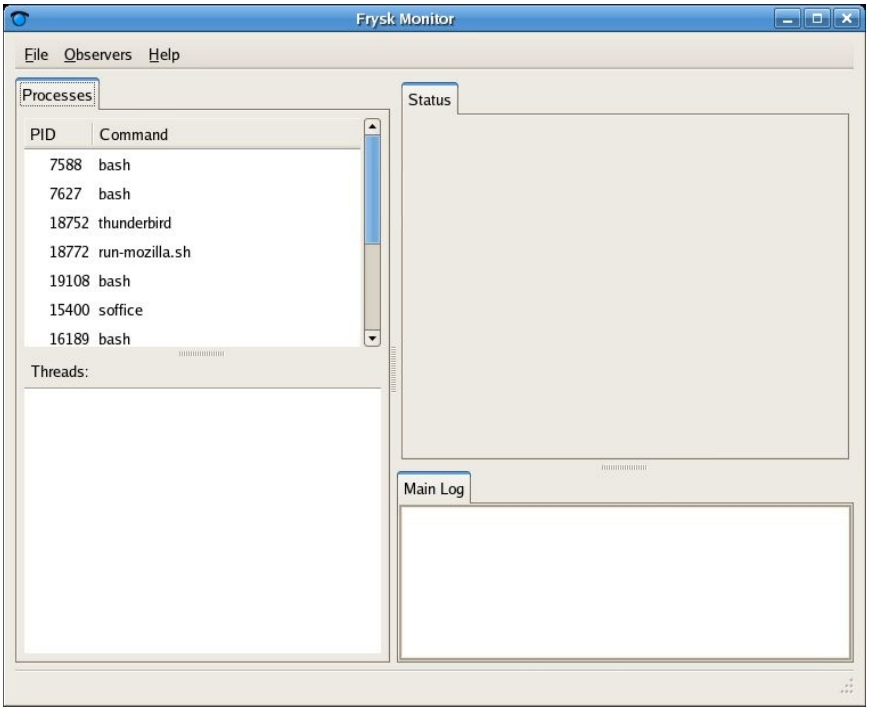


Please select which PIDs
you would like to examine:

- ☑ bash
  - ☑ 15400
  - ☑ 16189
  - ☑ 18752
  - ☑ 18772
  - ☑ 19108

Debug Session Process Picker Window

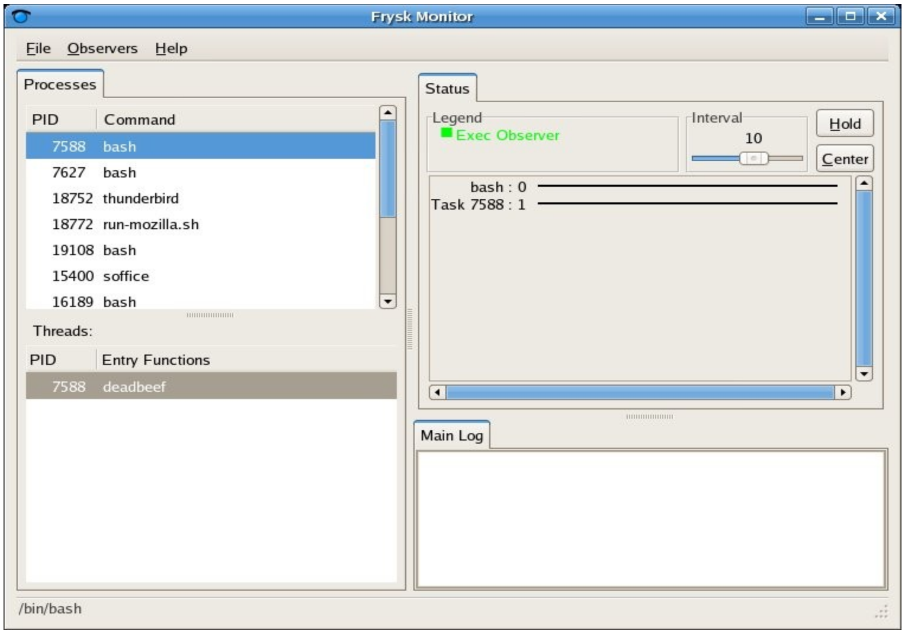
As can be seen above there are multiple bash processes currently in the cpu queue. The user can either select a single PID or all or any number in between to monitor. Once the desired PID(s) have been selected, click the **Forward** button to continue. The main Frysk monitoring window should appear and look something like this.



PID	Command
7588	bash
7627	bash
18752	thunderbird
18772	run-mozilla.sh
19108	bash
15400	soffice
16189	bash

Initial Frysk Monitoring Window

As you can see, the initial Frysk screen is divided into 4 areas or “panes”: **Processes**, **Status**, **Threads** and **Main Log**. The upper left view currently has one tab for **Processes**. In the **Process** pane is a list of all of the processes defined(and possibly selected in the **Process Picker** window) for this debug session. To activate the **Status** and **Threads** panes, simply click on one of the processes in the **Processes** pane.



PID	Command
7588	bash
7627	bash
18752	thunderbird
18772	run-mozilla.sh
19108	bash
15400	soffice
16189	bash

Threads:

PID	Entry Functions
7588	deadbeef

Interval: 10

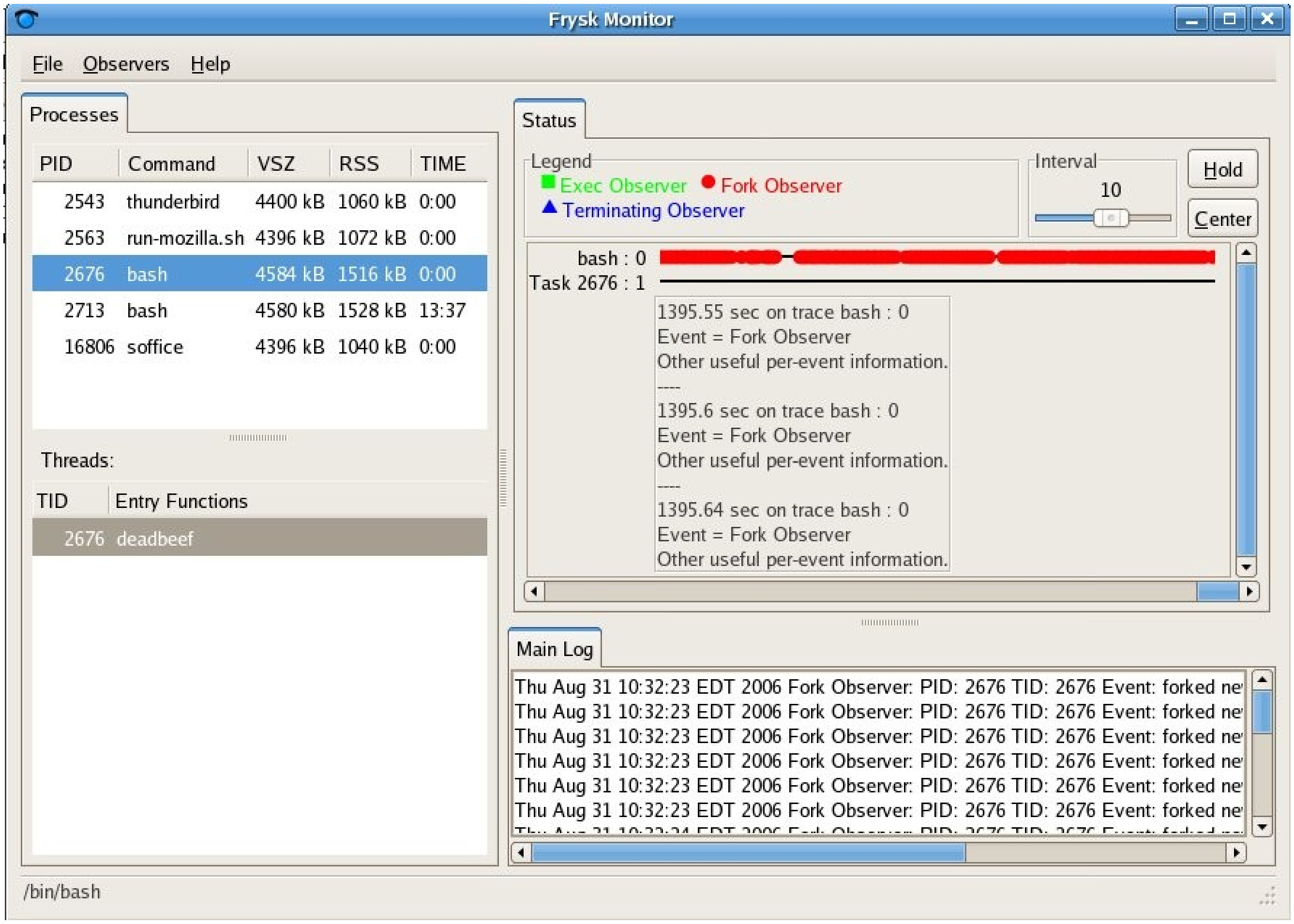
bash : 0
Task 7588 : 1

/bin/bash

Screenshot of a Selected Process

With the selection of a process, the **Status** and **Threads** panes become active. The **Threads** pane is self-explanatory as it will contain a list of the threads of the selected process. The **Status** pane is basically a timeline of the observer firings for the selected process/thread. A blip would appear on the line anytime one of the observers attached to that process fires. There are controls at the top of the timeline to change the length of time viewed in the pane, allows the user to start/stop the timeline and to center the data in the pane.

The following screenshot shows a monitor window where a bash process is has three observers attached, fork, exec and terminating. In this example, many forks have been executed and each one has been recorded in the **Main Log** pane and plotted in the **Status** pane.



PID	Command	VSZ	RSS	TIME
2543	thunderbird	4400 kB	1060 kB	0:00
2563	run-mozilla.sh	4396 kB	1072 kB	0:00
2676	bash	4584 kB	1516 kB	0:00
2713	bash	4580 kB	1528 kB	13:37
16806	soffice	4396 kB	1040 kB	0:00

Threads:

TID	Entry Functions
2676	deadbeef

```
1395.55 sec on trace bash : 0
Event = Fork Observer
Other useful per-event information.
----
1395.6 sec on trace bash : 0
Event = Fork Observer
Other useful per-event information.
----
1395.64 sec on trace bash : 0
Event = Fork Observer
Other useful per-event information.
```

```
Thu Aug 31 10:32:23 EDT 2006 Fork Observer: PID: 2676 TID: 2676 Event: forked ne
Thu Aug 31 10:32:23 EDT 2006 Fork Observer: PID: 2676 TID: 2676 Event: forked ne
Thu Aug 31 10:32:23 EDT 2006 Fork Observer: PID: 2676 TID: 2676 Event: forked ne
Thu Aug 31 10:32:23 EDT 2006 Fork Observer: PID: 2676 TID: 2676 Event: forked ne
Thu Aug 31 10:32:23 EDT 2006 Fork Observer: PID: 2676 TID: 2676 Event: forked ne
Thu Aug 31 10:32:23 EDT 2006 Fork Observer: PID: 2676 TID: 2676 Event: forked ne
Thu Aug 31 10:32:24 EDT 2006 Fork Observer: PID: 2676 TID: 2676 Event: forked ne
```

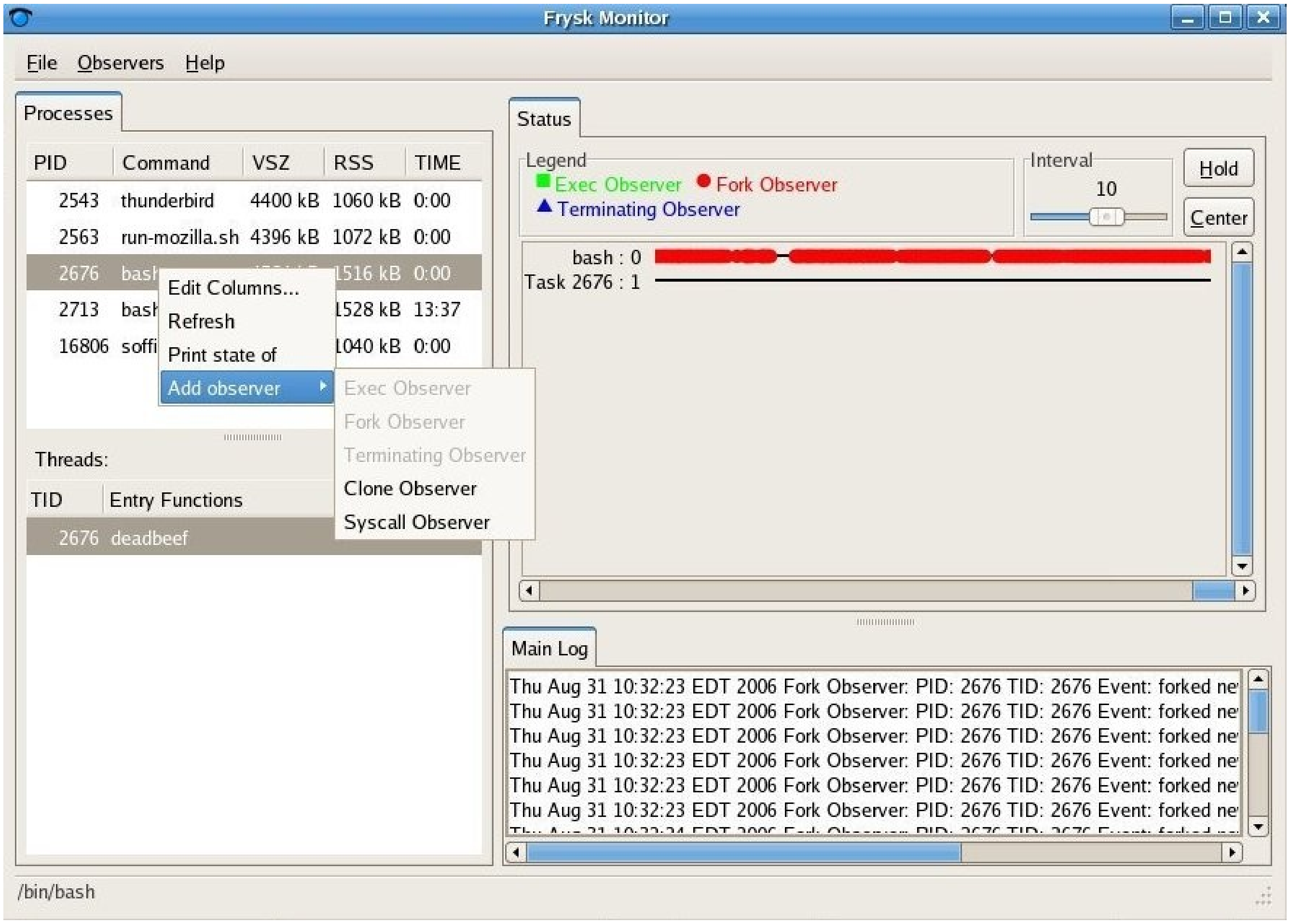
Frysk Monitor Window Showing Numerous Fork Events

The **Main Log** pane will show a line for every observer that fires with a time-stamp. The line will contain information on which observer fired(since multiple observers can be attached to a process/thread) and the host name where the event took place.

The **Status** pane shows the fork observer symbol on the timeline each time the observer fires. The user can use the cursor to hover over an item on the line and a description will be displayed as shown above. In the above screenshot there were three fork observers firing about the same time and so the three events are displayed.

Attaching Additional Observers from the Monitor

Additional observers can be attached to the processes from the monitoring screen by right-clicking on a process in the **Processes** pane or a thread in the **Threads** pane as shown below.



PID	Command	VSZ	RSS	TIME
2543	thunderbird	4400 kB	1060 kB	0:00
2563	run-mozilla.sh	4396 kB	1072 kB	0:00
2676	bash		1516 kB	0:00
2713	bash		1528 kB	13:37
16806	soffi		1040 kB	0:00

Threads:

TID	Entry Functions
2676	deadbeef

```
Thu Aug 31 10:32:23 EDT 2006 Fork Observer: PID: 2676 TID: 2676 Event: forked ne
Thu Aug 31 10:32:23 EDT 2006 Fork Observer: PID: 2676 TID: 2676 Event: forked ne
Thu Aug 31 10:32:23 EDT 2006 Fork Observer: PID: 2676 TID: 2676 Event: forked ne
Thu Aug 31 10:32:23 EDT 2006 Fork Observer: PID: 2676 TID: 2676 Event: forked ne
Thu Aug 31 10:32:23 EDT 2006 Fork Observer: PID: 2676 TID: 2676 Event: forked ne
Thu Aug 31 10:32:23 EDT 2006 Fork Observer: PID: 2676 TID: 2676 Event: forked ne
```

Adding Observers Via the Frysk Monitor Screen

Grayed out observers in the list have already been attached to the process. Be aware that any observers added from the monitor window are only temporary for this monitoring session. Once the monitoring window for a session is closed, any observers attached from the monitoring screen will not be attached next time this session is opened again. To permanently attach observers, this must be done from the **Frysk Session Manager**.

An observer can be attached to either the entire process(which basically attaches an observer to all of the threads(tasks) in a process) or it can be attached to a particular thread. To attach an observer to all of the threads in a process, use the process list in the **Processes** pane. To attach to a single thread within a process use the **Threads** pane. Each process can have as many observers attached as the user desires. Please bear in mind that each attached observer will consume a small percentage of system resources.

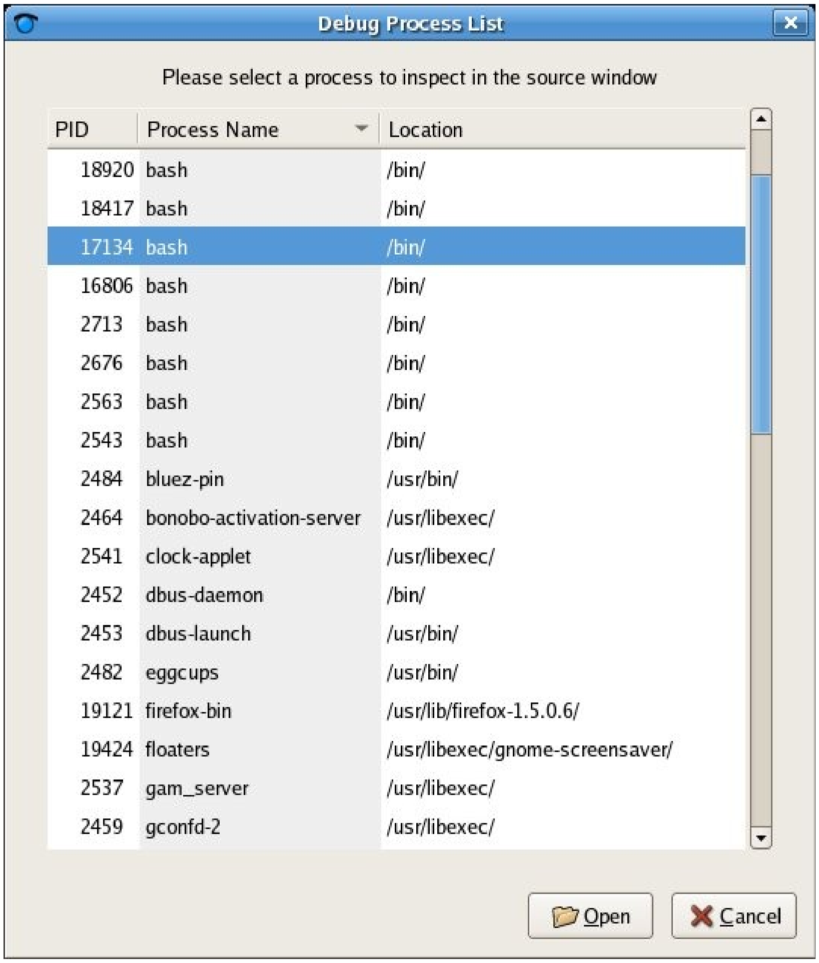
Source Window

The source window feature of Frysk is currently very minimally operational. The only feature that is operational right now is that it can be activated and “marked up” source code can be shown. The term “marked up” means that keywords, variables, comments, etc. are highlighted. In the near future the ability to bring up stacktraces will be added and soon after displaying variable values, stepping, etc.

*****NOTE***** For the source window to work as expected on a process, that process must be compiled with the -g option and the source code must be available somewhere. If it is not, when the source window is activated the message “No debug information for this stack frame” will be displayed where the source code should appear.

Activation

The most common way to activate the source window is from the initial Frysk window. Simply select the Debug an Existing Process radio button and click on the **Click Here to Select an Existing Process** button to bring up a list of processes that the current UID/PID owns.

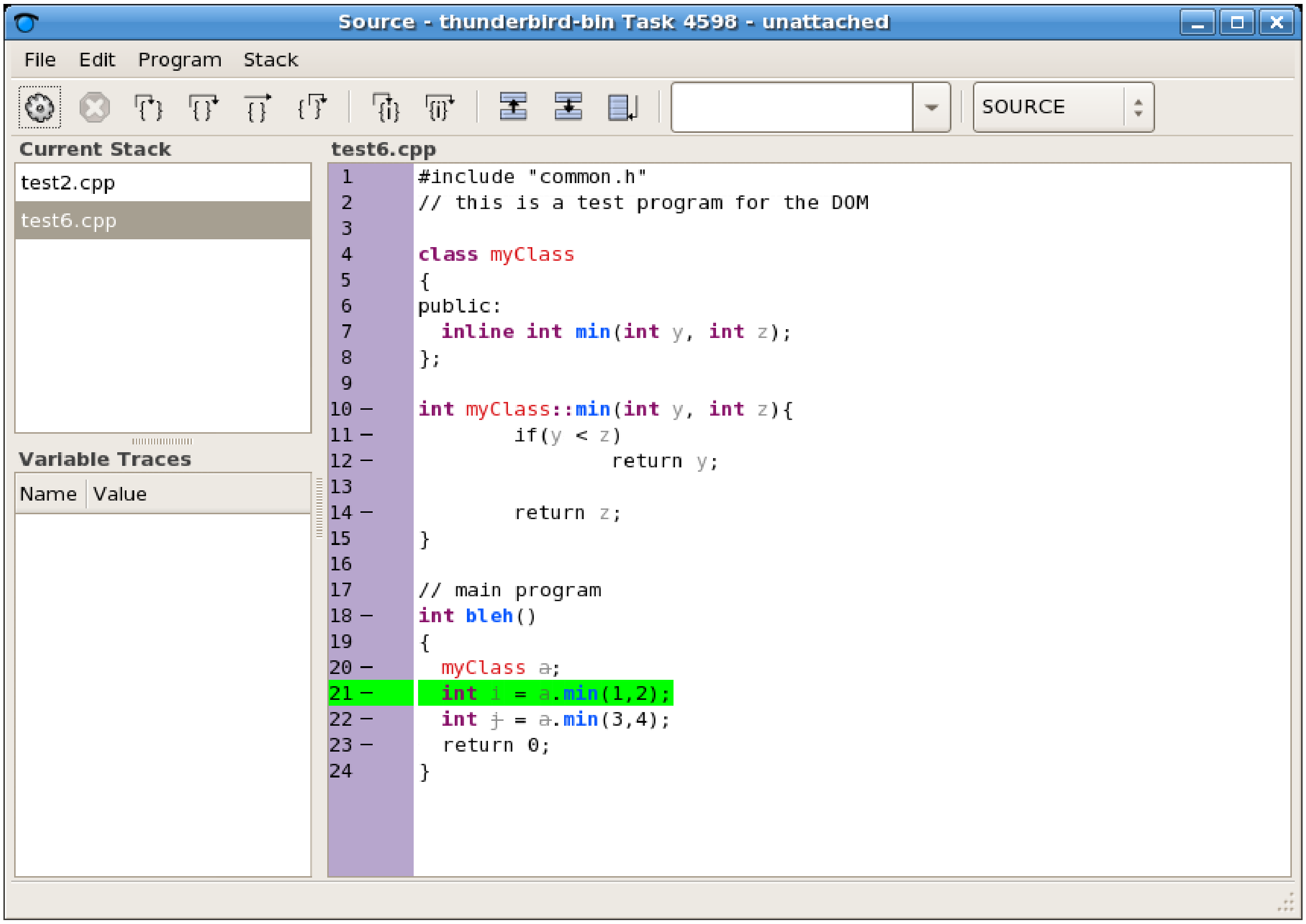


Please select a process to inspect in the source window

PID	Process Name	Location
18920	bash	/bin/
18417	bash	/bin/
17134	bash	/bin/
16806	bash	/bin/
2713	bash	/bin/
2676	bash	/bin/
2563	bash	/bin/
2543	bash	/bin/
2484	bluez-pin	/usr/bin/
2464	bonobo-activation-server	/usr/libexec/
2541	clock-applet	/usr/libexec/
2452	dbus-daemon	/bin/
2453	dbus-launch	/usr/bin/
2482	eggcups	/usr/bin/
19121	firefox-bin	/usr/lib/firefox-1.5.0.6/
19424	floaters	/usr/libexec/gnome-screensaver/
2537	gam_server	/usr/libexec/
2459	gconfd-2	/usr/libexec/

Screenshot of List of Processes to Attach Frysk Debugger To

Another way to activate the source window is to set that as as action to be performed when a custom observer fires. When defining a custom observer the user can define many actions to be performed and activating a source window is one of them. The following screenshot is an example of what a source window is activated.



```
#include "common.h"
// this is a test program for the DOM

class myClass
{
public:
  inline int min(int y, int z);
};

int myClass::min(int y, int z){
        if(y < z)
                return y;

        return z;
}

// main program
int bleh()
{
  myClass a;
  int i = a.min(1,2);
  int j = a.min(3,4);
  return 0;
}
```

Screenshot of Source Window

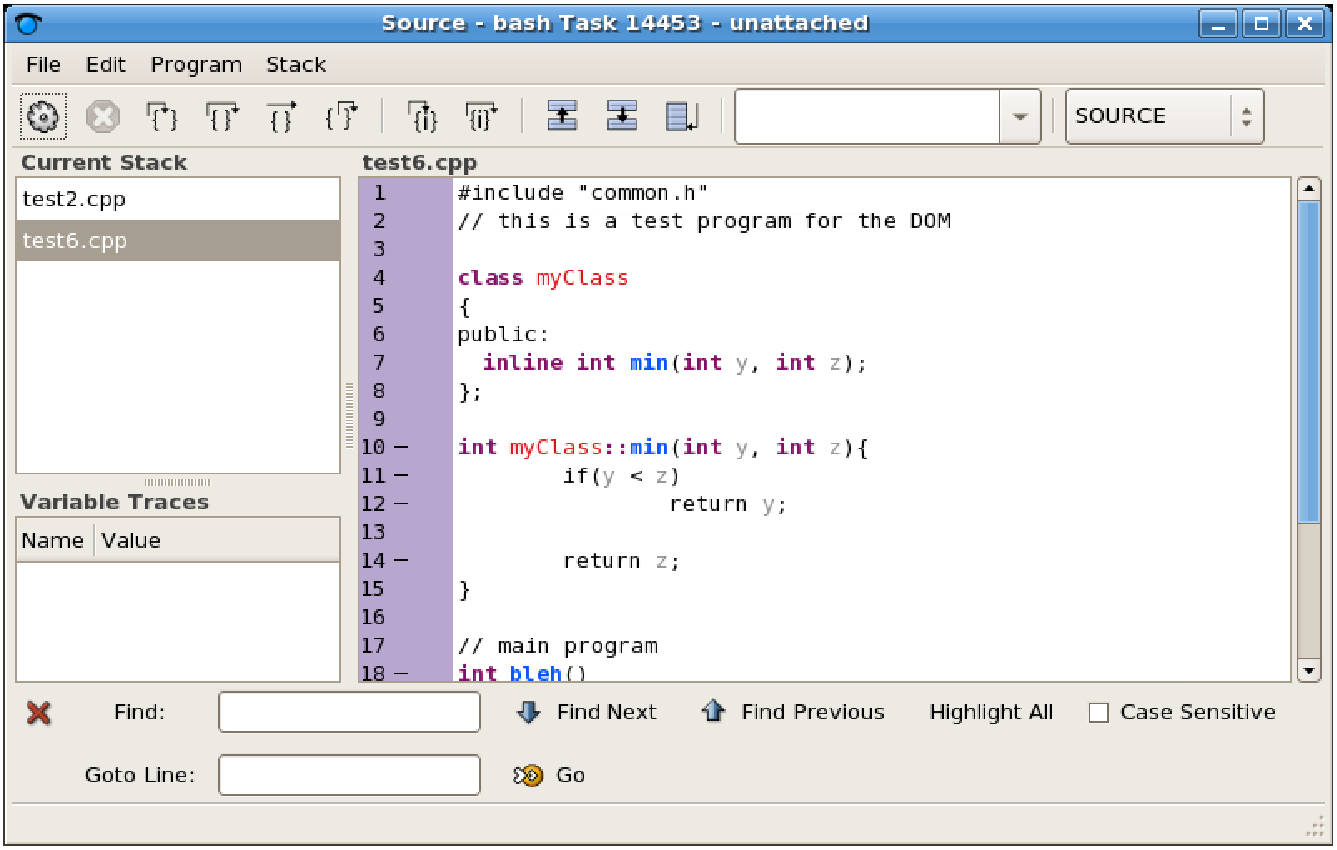
As can be seen above, the source window is divided into 4 sections: the toolbar, **Current Stack**, **Variable Traces** and the view of the source itself. The **Current Stack** view shows the list of functions/tasks that have been defined for this process and the **Variable Traces** view has the variable values of any variables the user has selected to follow. The only windows with meaningful information in them at this point are the **Current Stack** and the window containing the source code.

Toolbar

Hovering the mouse cursor over each icon on the toolbar provides a useful tooltip explaining its function if the icon itself is not explanatory enough. Symbols that are universally-accepted to developers are used. As can be seen in the screenshot, all of the functionality required to easily and efficiently debug applications is available. The grayed out icons are for future functionality.

In addition to the icons on the toolbar, there are two other items. The item on the far right is a pulldown window that provides a selection of how the source window should be displayed. The choices are: SOURCE, SOURCE/ASM, ASM, MIXED. As of right now, only SOURCE is supported. More info will be provided as these features are implemented.

To the left of the pulldown is a "Jump to Function" field. On the right-hand side of the field is a down-arrow indicating there are already some pre-built "jump" entries. Frysk automatically puts the name of each of the functions contained within the current source window there as a help to the developer. If the developer wants to search for other strings within the source window, press CTRL-F and the "Find" toolbar will appear at the bottom of the screen as shown in the following screenshot.



```
#include "common.h"
// this is a test program for the DOM

class myClass
{
public:
  inline int min(int y, int z);
};

int myClass::min(int y, int z){
        if(y < z)
                return y;

        return z;
}

// main program
int bleh()
```

Screenshot of Source Window with "Find" Toolbar Enabled

As can be seen above, the "Find" mechanism within Frysk is based loosely on the Firefox was of searching. Another way of activating the "Find" toolbar is to left-click on the "Edit" pulldown on the toolbar and select "Find" from the items on the pulldown.

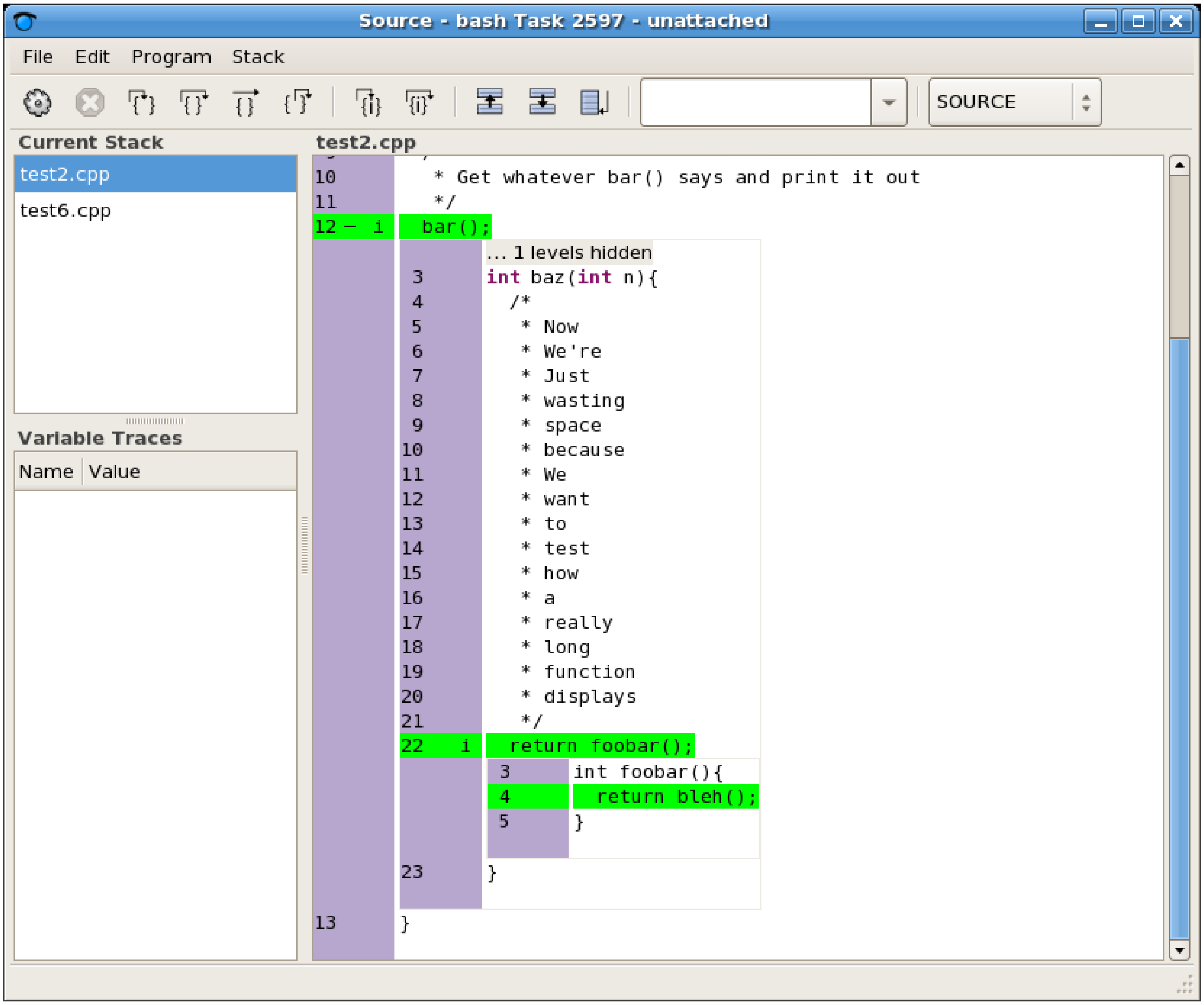
Source Window

Much thought has gone into the design of the source window incorporating many of the nice features of many open source GUI debuggers, plus a few more features have been/are being added. First, in the left-hand column, the line numbers appear and to the right of each line number is a "-"(minus sign) if the line has executable code or is blank if not.

Inside the main source window where the example C++ code resides, notice that there is key-word highlighting. Variable names are in one color, function names in another, C++ keywords in another, etc.

A nice feature that has been implemented in the source window in Frysk is the ability to view expanded inline code on demand. To illustrate this feature, click on the "test2.cpp" in the "Current Stack" window to bring up its source code. Line 12 is outlined in green indicating that line contains an inline function, "bar". Move the cursor over the line number column for line 12 and you will see the cursor convert to a "finger pointer" which means this area can be clicked on. Left-click in this area and notice that the inline function gets expanded to show what the code that makes up the "bar" function. A whole new column with line numbers and all is presented.

Notice that there is another line outlined in green, the line containing "baz", which is also an inline function. Move the cursor over the line number for "baz" and then left-click and the source lines for it will be shown, line numbers and all. The current plan is to enable a developer to be able to put break points on lines within inline functions. Below is a screenshot of this feature.



```
   * Get whatever bar() says and print it out
   */
 bar();
  ... 1 levels hidden
  int baz(int n){
    /*
     * Now
     * We're
     * Just
     * wasting
     * space
     * because
     * We
     * want
     * to
     * test
     * how
     * a
     * really
     * long
     * function
     * displays
     */
    return foobar();
      int foobar(){
        return bleh();
      }
  }
}
```

Screenshot of Source Window Showing Inline Code Expanded

Some Other Planned Features (not a complete list, just the highlights)

Some other planned features are:

- Show variable values when hovering the mouse over a variable
- One-click tracing of variables
- One-click set-up of “printf” statements(aka tagsets)
- Register window view

- Memory window view

Installing/Building Frysk

Please be aware that, as of this writing, Frysk is a very new project, just a little over a year old. As such, it is in a constant state of flux with new features being added almost every day and bugs being fixed constantly. This being the case, it might be best at this stage of the Frysk project to check out the latest source code from the CVS head and build it yourself. This will ensure you have the latest version and all of the latest features and bug fixes.

Beginning with RHEL4U3(Red Hat Enterprise Linux Version 4 Update 3) and FC5(Fedora Core 5), Frysk is available as a "technology preview". Frysk will be added as a full-blown tool in the FC6/RHEL5 releases. Every attempt is made to make sure Frysk will properly build on these platforms as long as the latest compiler and associated libraries are installed.

Please visit the Frysk website at http://sourceware.org/frysk for instructions on building Frysk on various os' and hardware architectures as well as the latest news on Frysk.

Getting Involved with Frysk

Please visit the following link to see how to join the Frysk developers on chat channels and mailing lists: http://sourceware.org/frysk/getinvolved/