GDB Debugging in both User and Kernel Space

Mike Anderson

mailto://mike@theptrgroup.com

http://www.ThePTRGroup.com

Lab 1. Running gdb

Objective

Verify that gdb is installed and working on your platform.

Procedure

In this lab we will learn how to debug a program using gdb.

On your development host, make sure that you have the following packages installed. These will be the Debian names of the packages, but Fedora and other Linux distros should have a similar package:

build-essential
gdb
ddd

In a Debian-based distro this would be accomplished using:

```
$ sudo apt-get install build-essential gdb ddd
```

Fedora and others may use the “yum” package manager instead. You will have to install these as the superuser.

For your Pocketbeagle, you will need the same packages and their dependencies from the instructor.

You will install them on the Pocketbeagle using the dpkg command:

```
$ sudo dpkg –i *.deb
```

Once these are installed, cd into debug-labs/Lab01 on your development host.

There is a small test program in there that we will use for the first debug session.

Compile the code with debugging enabled:

```
$ gcc –g3 –o bubblesort bubblesort.c
```

Start gdb and pass to it the name of the executable:

```
$ gdb bubblesort
```

This will run gdb in commandline mode. Once the program is loaded, run the program passing the parameter value 5:

```
(gdb) run
```

This will run the application and print a sorted list.

Run the list command to see the code:

```
(gdb) list
```

Running in the normal command mode is painful. So, we'll switch to the TUI mode of gdb using the **<CTRL> X A** keyboard sequence. Run the **list** command again.

Next, we'll insert a break point and run it again. Set a breakpoint on line 37 of the code:

```
(gdb) b 37
```

A breakpoint indicator should become visible in the gutter on the left-hand side. Next, run the program again to encounter the breakpoint. Let's set another breakpoint at line 20. We will also set a display up for the array arr

```
(gdb) b 20
```

```
(gdb) display *arr@n
```

This display will the entire array. Next, use the `cont` command to continue execution. Each time the program hits the breakpoint, the array will be displayed. You can see the array contents sorting.

List the current breakpoints, delete the breakpoint at line 20 and continue execution.

```
(gdb) info breakpoints
```

```
(gdb) d 2
```

```
(gdb) cont
```

Next delete the breakpoint at line 37 and continue execution. This should exit the program and return to gdb. Type quit to exit.

Next, we'll do the same sequence on the Pocketbeagle. Cd to the base of the debug-labs directory and tar the labs.

```
$ cd ~
```

```
$ tar cvf debug-labs.tar ./debug-labs
```

Copy them to the Pocketbeagle via sftp and untar them into the Debian user home directory.

```
debian@beaglebone:~$ sftp <user>@192.168.7.1
debian@beaglebone:~$ get debug-labs.tar
debian@beaglebone:~$ tar xvf debug-labs.tar
debian@beaglebone:~$ cd ./debug-labs/Lab01
debian@beaglebone:~$ make clean
debian@beaglebone:~$ make
```

Go through the same steps from above for building and debugging using gdb on the pocketbeagle.

Lab Completed.

Lab 2. Debugging a Core File

Objective

Perform post mortem debugging of a program using a core file.

Procedure

In this lab we will learn how to debug a program using a core file that was generated in a previous run of the program.

Change to your Lab02 directory on the Pocketbeagle. Build the program "debugcore" by using the **make** command.

```
debian@beaglebone:~$ cd
debian@beaglebone:~$ cd debug-labs/Lab02
debian@beaglebone:~$ make
```

As in the previous labs, our Makefile copies the "debugcore" program to the compiled directory.

Do not examine the source code in the file debugcore.c yet. Now that the code is built and copied to the target, we will run it from the command line. First, we will enable the ability for the target system to create a core dump. Go to ssh session to the Pocketbeagle. On the target, execute:

```
debian@beaglebone:~$ ulimit -c unlimited
debian@beaglebone:~$ ./compiled/debugcore
```

The ulimit command allows us to specify that a core file can be generated in the event of an exception and that this core file is not limited in size. By default, the kernel will not generate a core file, making this command necessary for the exercise. The ./debugcore command runs the program.

At the moment, only the debian user has permissions to access the core file. In order to use the core file on the host to determine the error, we give permission to read the file to normal users. On the target, issue this command:

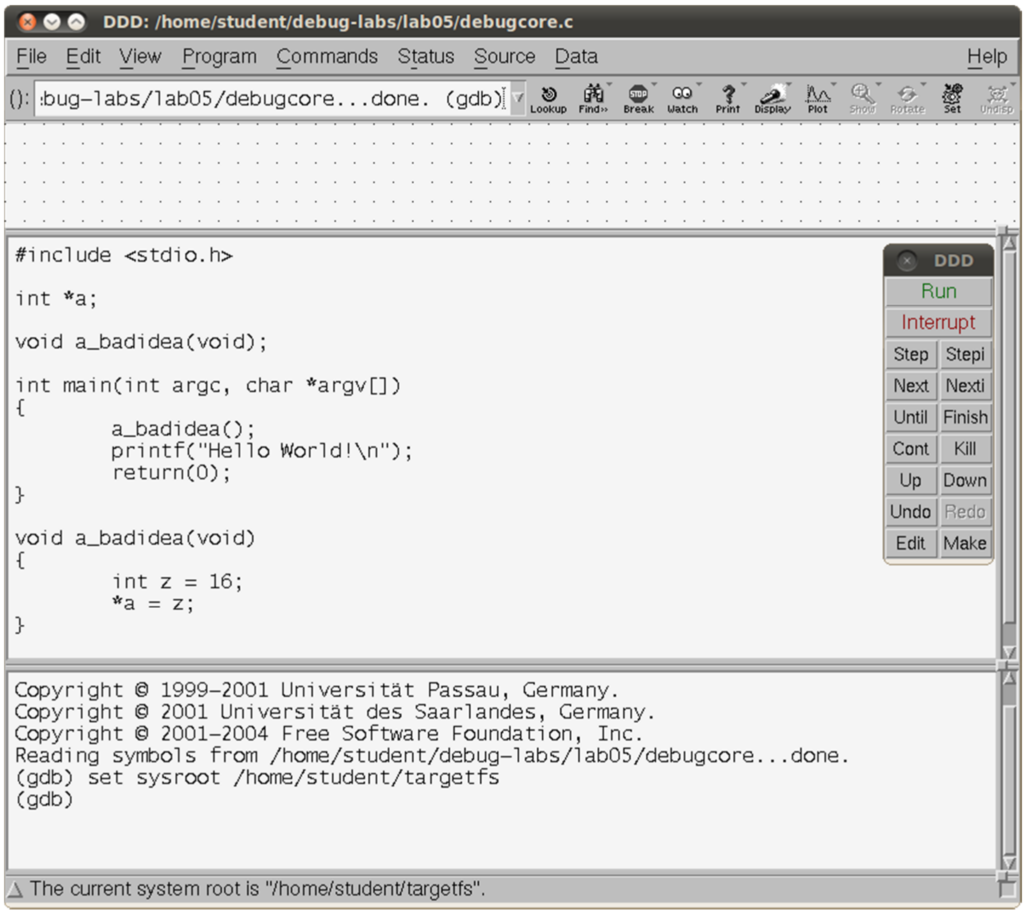
```
debian@beaglebone:~$ chmod a+rw ./compiled/core
```

Now, sftp the core file into the Lab02 directory on the host.   We will be using ddd to load the core file into memory. Core files are a snapshot of the process's register set and memory at the time of the failure.

Go back to your host terminal, and make sure that you are in the lab02 directory. Start ddd as follows:

```
$ ddd --debugger arm-none-linux-gnueabi-gdb ./debugcore
```

The ddd session will look something like this:



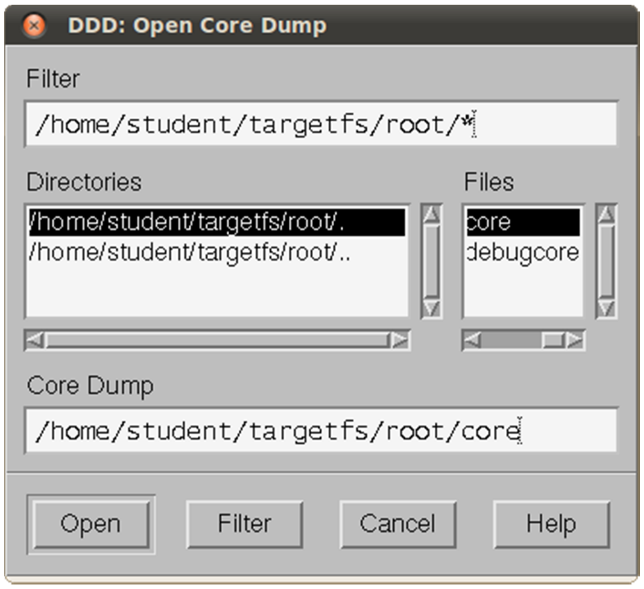
```
():  :bug-labs/lab05/debugcore...done. (gdb)
```

```
#include <stdio.h>

int *a;

void a_badidea(void);

int main(int argc, char *argv[])
{
        a_badidea();
        printf("Hello World!\n");
        return(0);
}

void a_badidea(void)
{
        int z = 16;
        *a = z;
}
```

```
Copyright © 1999-2001 Universität Passau, Germany.
Copyright © 2001 Universität des Saarlandes, Germany.
Copyright © 2001-2004 Free Software Foundation, Inc.
Reading symbols from /home/student/debug-labs/lab05/debugcore...done.
(gdb) set sysroot /home/student/targetfs
(gdb)
```

```
△ The current system root is "/home/student/targetfs".
```

Next we will load the core file into ddd using File -> Open Core Dump… Enter the directory "/home/student/debug-labs/Lab02" in the top filter line as shown below, then hit <Enter> in that filter line to populate the subwindows below it.



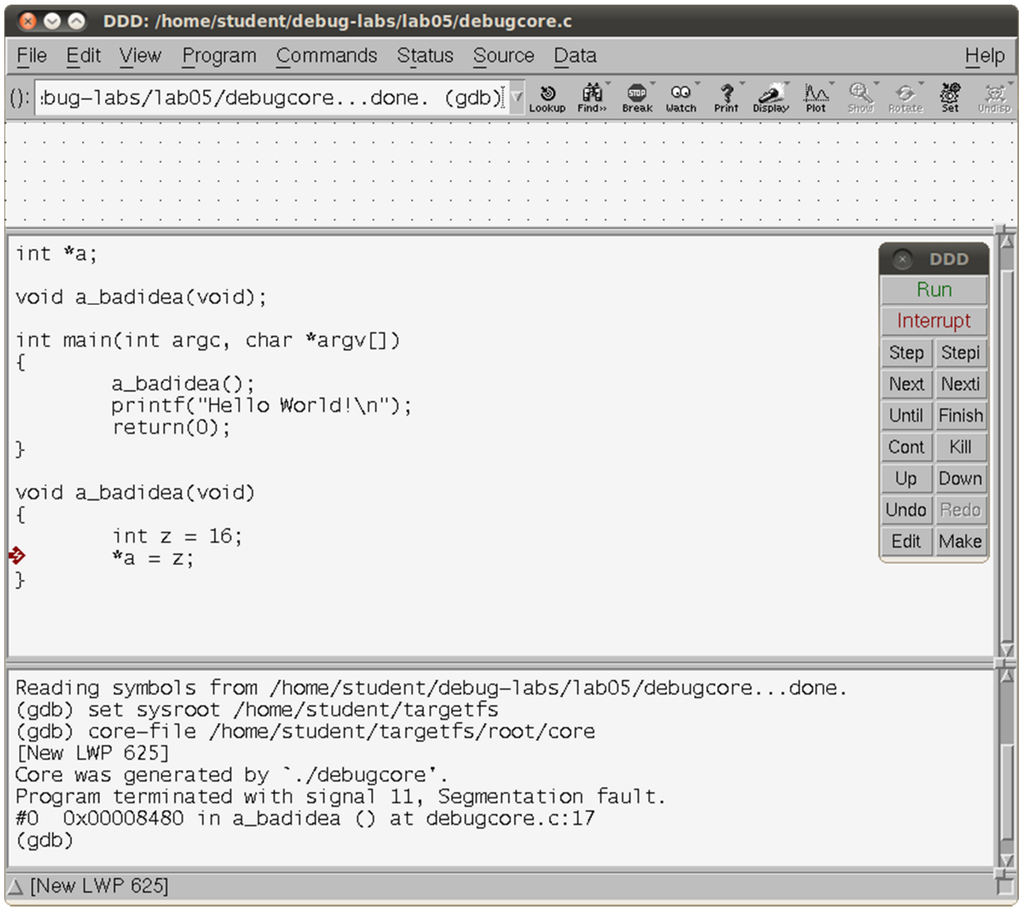
Filter
/home/student/targetfs/root/*

Directories
/home/student/targetfs/root/.
/home/student/targetfs/root/..

Files
core
debugcore

Core Dump
/home/student/targetfs/root/core

Make sure that the file **/home/student/Lab02/core** is selected. Then click **Open**. The output in the lower window indicates that there is a segmentation fault at line 17 of debugcore.c, and the main window shows the offending line of code.



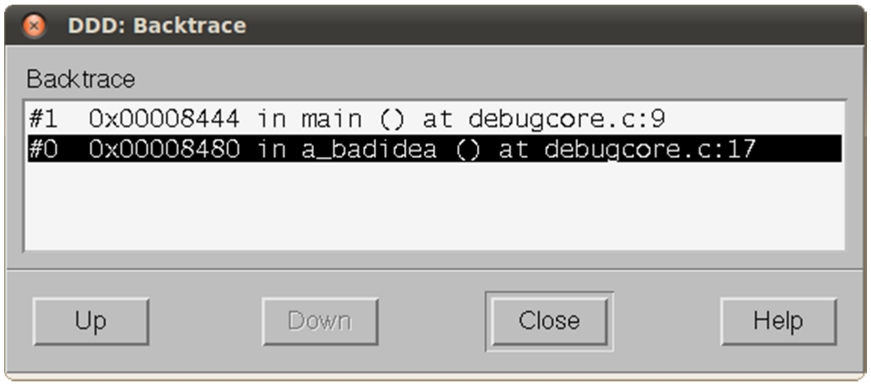
```
(): :bug-labs/lab05/debugcore...done. (gdb)
```

```
int *a;

void a_badidea(void);

int main(int argc, char *argv[])
{
        a_badidea();
        printf("Hello World!\n");
        return(0);
}

void a_badidea(void)
{
        int z = 16;
        *a = z;
}
```

```
Reading symbols from /home/student/debug-labs/lab05/debugcore...done.
(gdb) set sysroot /home/student/targetfs
(gdb) core-file /home/student/targetfs/root/core
[New LWP 625]
Core was generated by `./debugcore'.
Program terminated with signal 11, Segmentation fault.
#0  0x00008480 in a_badidea () at debugcore.c:17
(gdb)
```

```
[New LWP 625]
```

In the ddd main window, select the "Status" button followed by the "Backtrace" button. A new window will open that shows the calling sequence that got us to the failure:



```
#1  0x00008444 in main () at debugcore.c:9
#0  0x00008480 in a_badidea () at debugcore.c:17
```

We see that the ultimate failure was at the line "**`*a = z;`**". "a" is a pointer. But where does it point? Position your mouse cursor over the "*a" expression in line 17 to see the answer.

Because the core dump is post mortem, you can see the values associated with the register set and variables at the time the core dump was generated, but you can't change them nor continue execution of the program. Close ddd.

Lab Completed.

Lab 3. gdbserver and the gdb Debugger

Objective

Debug a program on a remote target using DDD, GDB, and gdbserver.

Procedure

In this lab we will show how to setup and use a cross-debugging environment.

Change to your lab01 directory on the host. Build the program “debug” by using the **make** command.

```
$ cd
$ cd debug-labs/Lab03
$ make
```

As you can see from the Makefile, the “debug” program is copied to the target’s /root directory. Go to the ssh window, and do the following commands on the target.

```
debian@beaglebone:~$ cd /root
debian@beaglebone:~$ gdbserver 192.168.1.2:1929 ./debug 4
Process ./debug created; pid = 623
Listening on port 1929
```

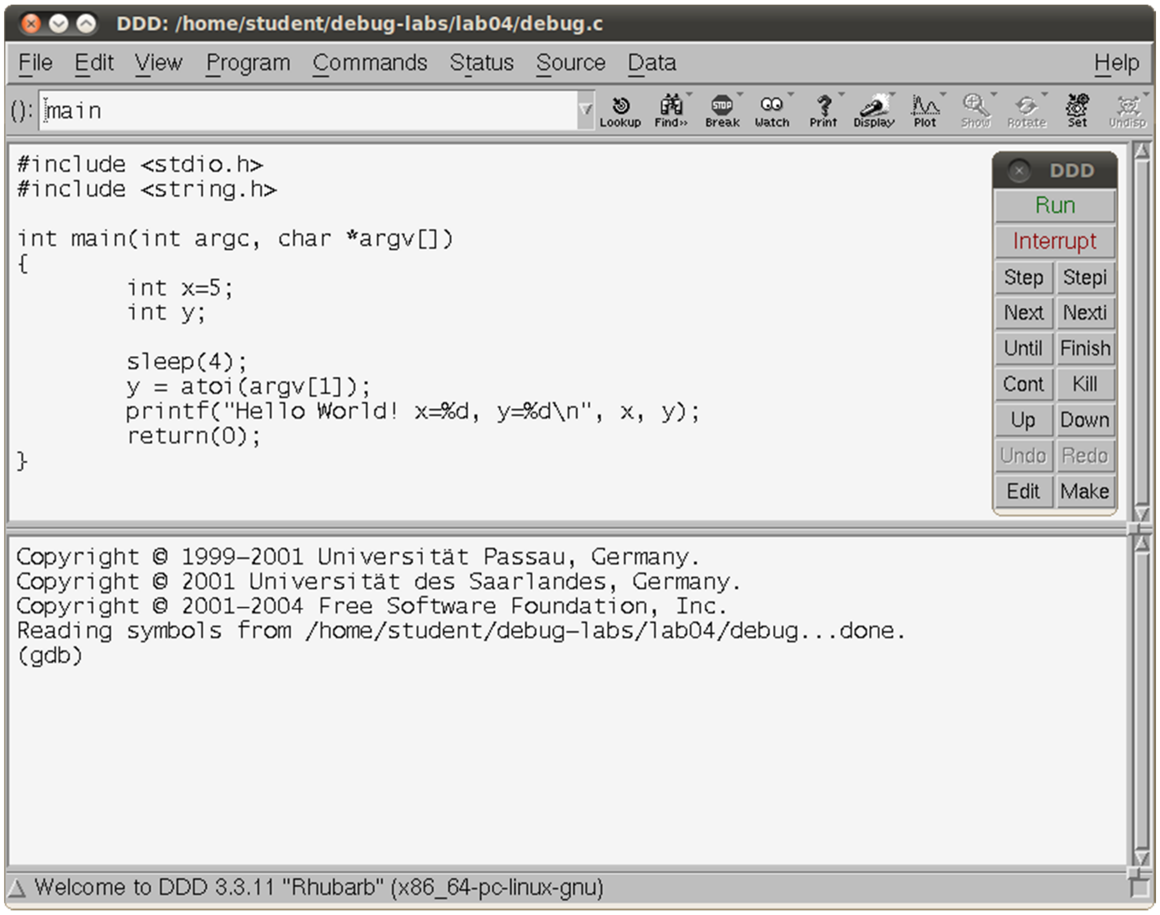
The gdbserver is a "helper" program that runs the application, but waits at "main" for a connection from a remote gdb debugger session. The IP address is the address of the host that will run gdb. The ":1929" is a port number above 1024 that will be used for the connection. Any unused port number will do, but we need to make sure that we remember the port number and use it in the next step. In other debugging situations (e.g. if you were debugging two programs at the same time), it may be necessary to start two different gdbservers. Each must be started with a unique port number. In addition, each must be started from its own shell instance on the target.

To complete setting up the debug environment, go to the terminal window on the host (make sure you are in the lab03 directory) and start the ddd program:

```
$ ddd --debugger arm-none-linux-gnueabi-gdb ./debug
```

The ddd program is a graphical front end to gdb. The above command says to use the cross-debugger arm-none-linux-gnueabi-gdb with the symbols contained in the “./debug” file.

Click through the “Tip of the Day” window and the “DDD Help” window that appear by default, to get to the following display.


DDD: /home/student/debug-labs/lab04/debug.c
File Edit View Program Commands Status Source Data Help
(): main
Lookup Find» Break Watch Print Display Plot Show Rotate Set Undisp
DDD
Run
Interrupt
Step Stepi
Next Nexti
Until Finish
Cont Kill
Up Down
Undo Redo
Edit Make

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
        int x=5;
        int y;

        sleep(4);
        y = atoi(argv[1]);
        printf("Hello World! x=%d, y=%d\n", x, y);
        return(0);
}
```

```
Copyright © 1999-2001 Universität Passau, Germany.
Copyright © 2001 Universität des Saarlandes, Germany.
Copyright © 2001-2004 Free Software Foundation, Inc.
Reading symbols from /home/student/debug-labs/lab04/debug...done.
(gdb)
```

Welcome to DDD 3.3.11 "Rhubarb" (x86_64-pc-linux-gnu)

For the debugger to find the program's shared library information, we next set the sysroot using the following command in the command line frame in ddd:

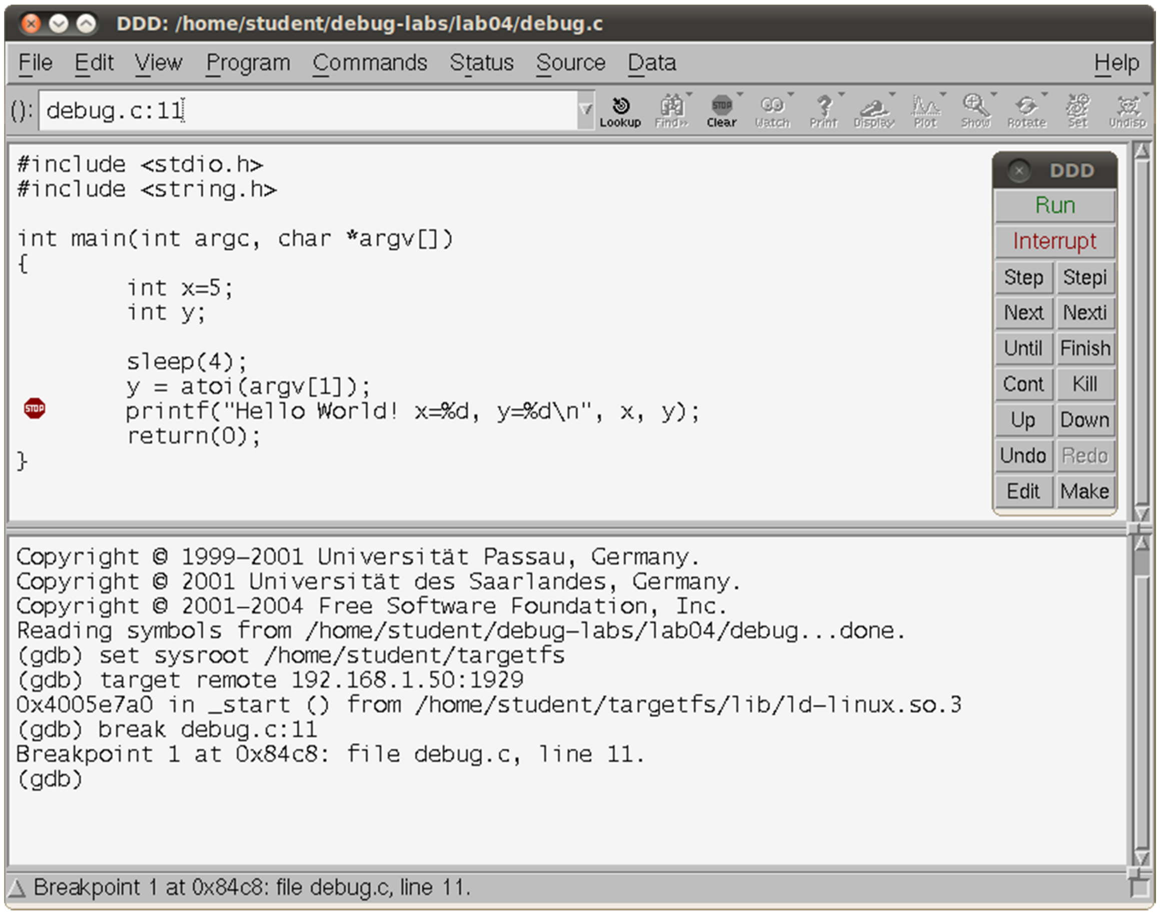
```
(gdb) set sysroot /home/student
```

To complete the connection with the target (gdbserver) perform the following from the command line frame in ddd:

```
(gdb) target remote 192.168.1.50:1929
```

The host ddd program is connected with the gdbserver program on the target.

In the source display frame in the ddd window, right-click on the leftmost part of line 11 (the one with the printf) and select "Set Breakpoint". A small "Stop" sign icon should appear on the left-hand side of the screen.

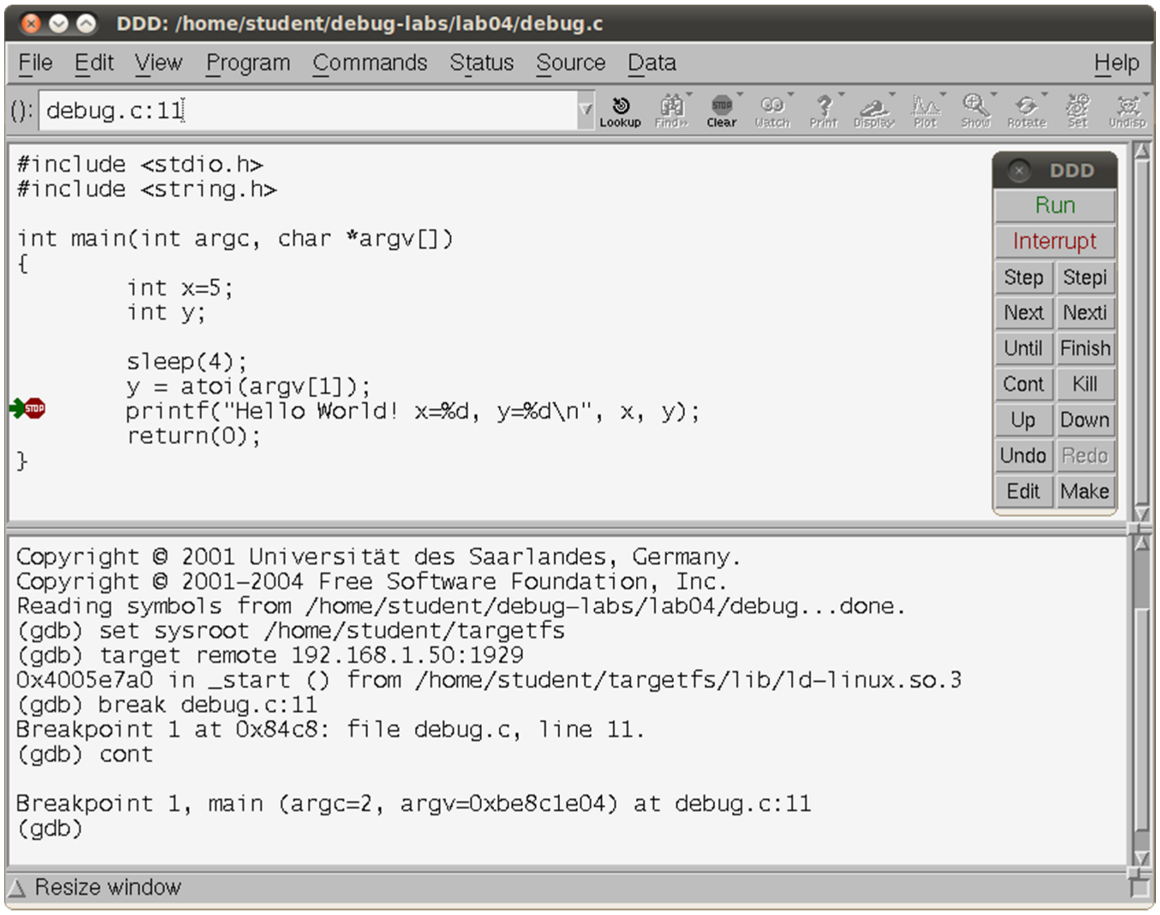

DDD: /home/student/debug-labs/lab04/debug.c
File Edit View Program Commands Status Source Data Help
(): debug.c:11
Lookup Find» Clear Watch Print Display Plot Show Rotate Set Undisp
DDD
Run
Interrupt
Step Stepi
Next Nexti
Until Finish
Cont Kill
Up Down
Undo Redo
Edit Make

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
        int x=5;
        int y;

        sleep(4);
        y = atoi(argv[1]);
        printf("Hello World! x=%d, y=%d\n", x, y);
        return(0);
}
```

```
Copyright © 1999-2001 Universität Passau, Germany.
Copyright © 2001 Universität des Saarlandes, Germany.
Copyright © 2001-2004 Free Software Foundation, Inc.
Reading symbols from /home/student/debug-labs/lab04/debug...done.
(gdb) set sysroot /home/student/targetfs
(gdb) target remote 192.168.1.50:1929
0x4005e7a0 in _start () from /home/student/targetfs/lib/ld-linux.so.3
(gdb) break debug.c:11
Breakpoint 1 at 0x84c8: file debug.c, line 11.
(gdb)
```

Breakpoint 1 at 0x84c8: file debug.c, line 11.

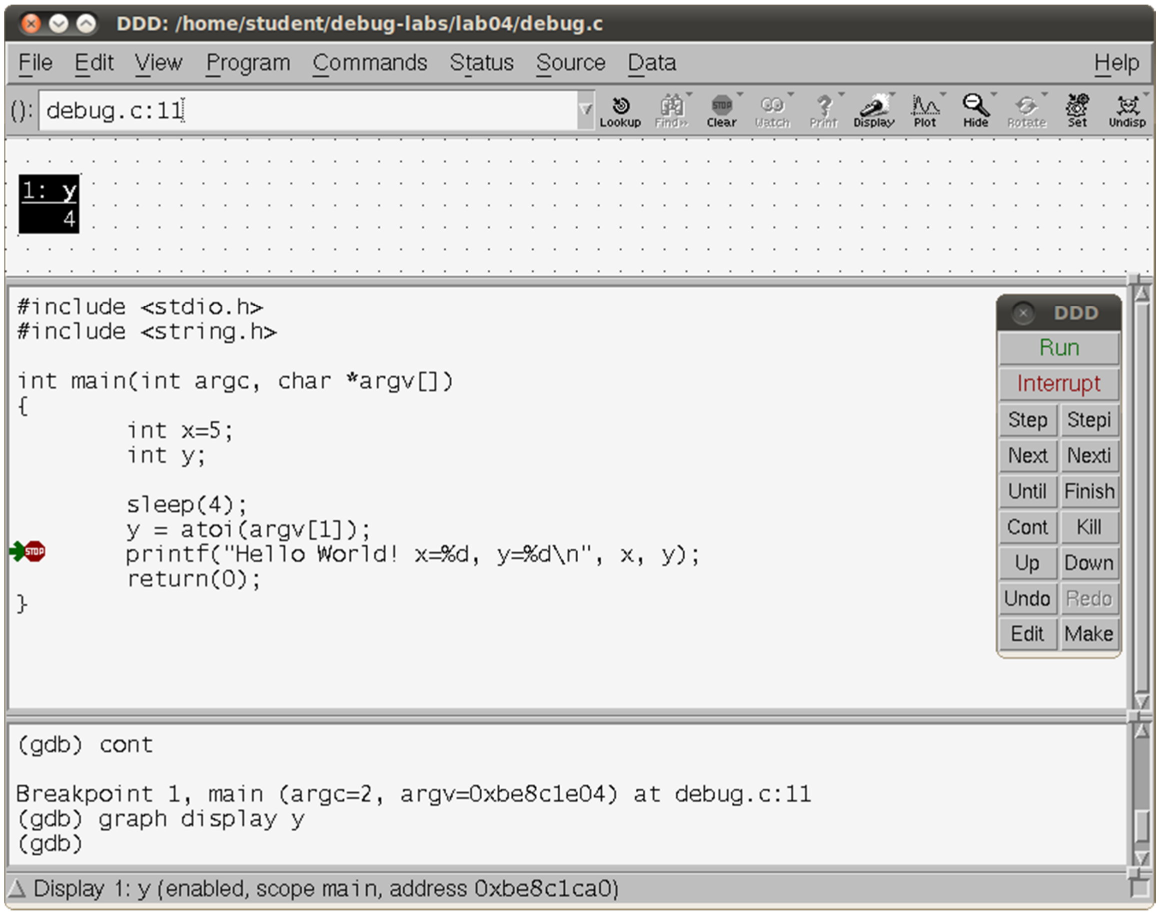
Next, either type “cont” at the gdb prompt or select the“Cont” button from the ddd sub-window and allow the program to continue. Note that you’re not going to use the “run” command, because the program is already running under the control of the gdbserver. Once the program stops at the breakpoint, your ddd display should have a green arrow pointing to the current execution line.


DDD: /home/student/debug-labs/lab04/debug.c
File Edit View Program Commands Status Source Data Help
(): debug.c:11
Lookup Find Clear Watch Print Display Plot Show Rotate Set Undisp
DDD
Run
Interrupt
Step Stepi
Next Nexti
Until Finish
Cont Kill
Up Down
Undo Redo
Edit Make
Resize window

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
        int x=5;
        int y;

        sleep(4);
        y = atoi(argv[1]);
        printf("Hello World! x=%d, y=%d\n", x, y);
        return(0);
}
```

```
Copyright © 2001 Universität des Saarlandes, Germany.
Copyright © 2001-2004 Free Software Foundation, Inc.
Reading symbols from /home/student/debug-labs/lab04/debug...done.
(gdb) set sysroot /home/student/targetfs
(gdb) target remote 192.168.1.50:1929
0x4005e7a0 in _start () from /home/student/targetfs/lib/ld-linux.so.3
(gdb) break debug.c:11
Breakpoint 1 at 0x84c8: file debug.c, line 11.
(gdb) cont

Breakpoint 1, main (argc=2, argv=0xbe8c1e04) at debug.c:11
(gdb)
```

In the ddd display frame, position your cursor over the variables x and y successively. You should see their values being displayed. Right-click on the variable "y" and select the "Display" command. This will allow you to track the value of "y" as the program runs.


DDD: /home/student/debug-labs/lab04/debug.c
File Edit View Program Commands Status Source Data Help
(): debug.c:11
Lookup Find» Clear Watch Print Display Plot Hide Rotate Set Undisp
1: y
4
DDD
Run
Interrupt
Step Stepi
Next Nexti
Until Finish
Cont Kill
Up Down
Undo Redo
Edit Make

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
        int x=5;
        int y;

        sleep(4);
        y = atoi(argv[1]);
        printf("Hello World! x=%d, y=%d\n", x, y);
        return(0);
}
```

```
(gdb) cont

Breakpoint 1, main (argc=2, argv=0xbe8c1e04) at debug.c:11
(gdb) graph display y
(gdb)
```

Display 1: y (enabled, scope main, address 0xbe8c1ca0)

Next, right-click the variable “y” in the top frame of the ddd window and select "Set Value...". A new dialog window will open that has the current value of "y". Change the value to something new and click OK or press enter. This has changed the value of "y". You should see the new value of "y" in the display frame.

Finally, click the "Cont" button or type "cont" in the gdb command frame to see the program complete and print the last value of "y" that you entered. Note that program output goes to your ssh window. The gdb session indicates that the program exited normally and the gdbserver session should have also exited. Close ddd.

Lab Completed.

Lab 4. Debugging a Multithreaded Program

Objectives

Debug a multithreaded program.
Exercise some of GDB's thread debugging capabilities.

Procedure

In this lab we will learn techniques to use when debugging a multithreaded program.

Change to your lab03 directory on the host. Look at the source code in sched.c. There is a main routine that creates two threads. The threads run, keeping track of scheduler operations by counting thread switches, and calculating the average number of times they loop before a thread switch.

Build the program "sched" by using the **make** command.

```
$ cd
$ cd debug-labs/Lab04
$ make
```

Go to the ssh window, and do the following commands on the target.

```
debian@beaglebone:~$ cd ./Lab04
debian@beaglebone:~$ gdbserver 192.168.7.1:1929 ./sched
Process ./sched created; pid = 663
Listening on port 1929
```

Go to the terminal window on the host (make sure you are in the lab04 directory) and start the ddd program:

```
$ ddd --debugger arm-none-linux-gnueabi-gdb ./sched
```

Set the sysroot as before, using the following command in the command line frame in ddd:

```
(gdb) set sysroot /home/student
```

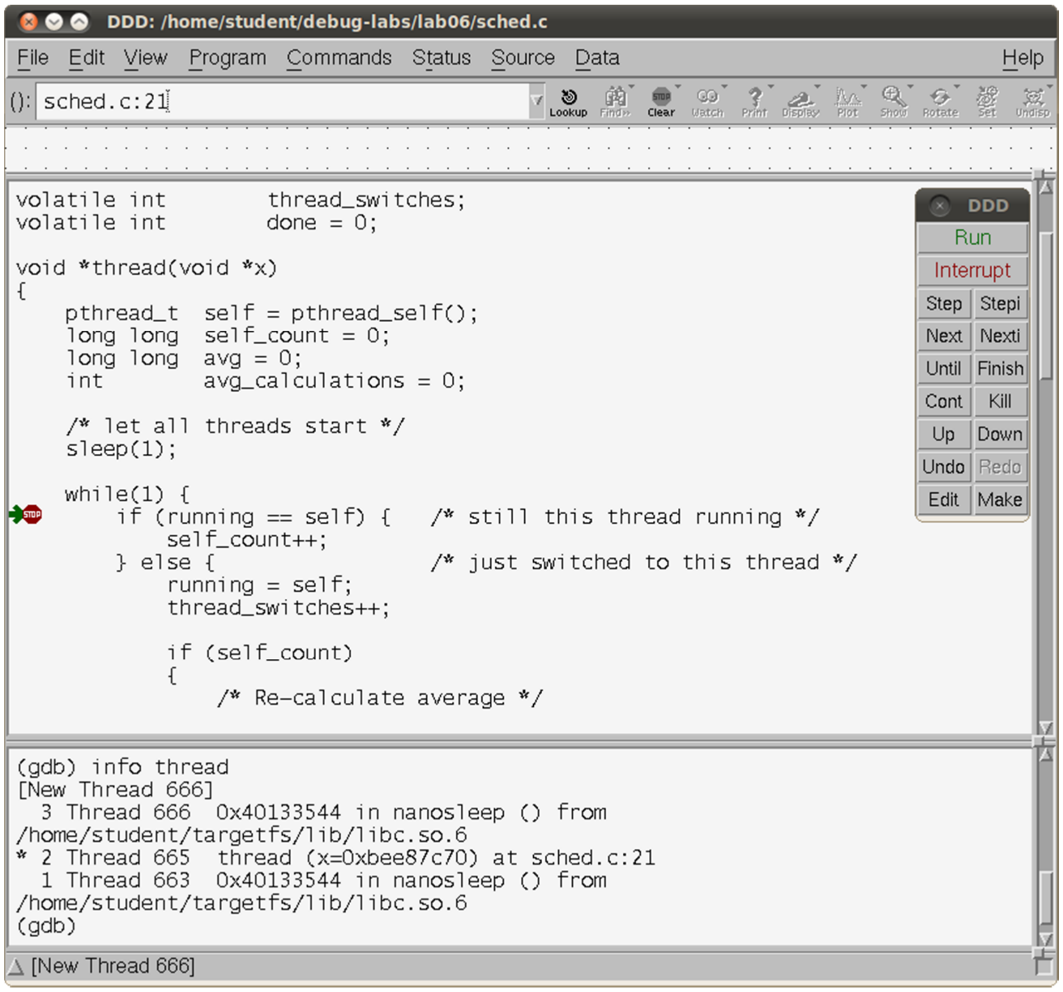
To complete the connection with the target (gdbserver) perform the following from the command line frame in ddd:

```
(gdb) target remote 192.168.7.2:1929
```

In the source code display frame in the ddd window, right-click on the leftmost part of line 21. This is the "if" statement just inside the "while (1)" loop in the thread() function. Select "Set Breakpoint".

Click "Cont" to continue the program. The breakpoint will be hit.

How can we tell which thread encountered the breakpoint? At the “(gdb)” prompt in the bottom frame, enter the command “info thread”.


DDD: /home/student/debug-labs/lab06/sched.c
File Edit View Program Commands Status Source Data Help
(): sched.c:21
Lookup Find Clear Watch Print Display Plot Show Rotate Set Undisp
DDD
Run
Interrupt
Step Stepi
Next Nexti
Until Finish
Cont Kill
Up Down
Undo Redo
Edit Make

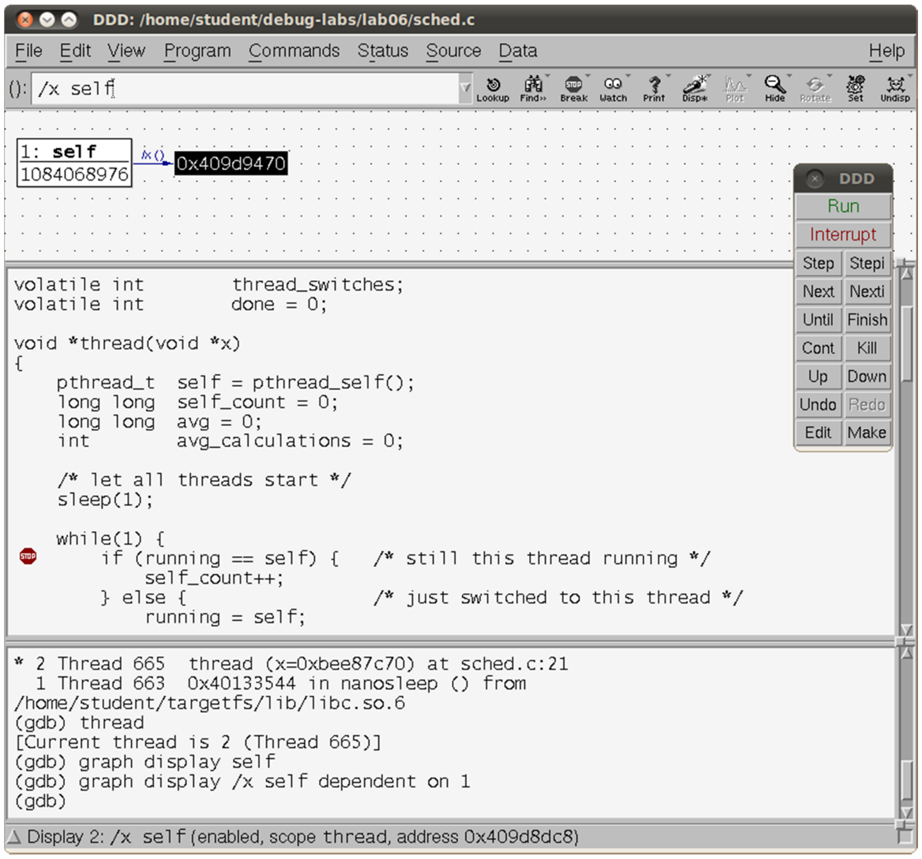
```
volatile int        thread_switches;
volatile int        done = 0;

void *thread(void *x)
{
    pthread_t  self = pthread_self();
    long long  self_count = 0;
    long long  avg = 0;
    int        avg_calculations = 0;

    /* let all threads start */
    sleep(1);

    while(1) {
        if (running == self) {   /* still this thread running */
            self_count++;
        } else {                 /* just switched to this thread */
            running = self;
            thread_switches++;

            if (self_count)
            {
                /* Re-calculate average */
```

```
(gdb) info thread
[New Thread 666]
  3 Thread 666  0x40133544 in nanosleep () from
/home/student/targetfs/lib/libc.so.6
* 2 Thread 665  thread (x=0xbee87c70) at sched.c:21
  1 Thread 663  0x40133544 in nanosleep () from
/home/student/targetfs/lib/libc.so.6
(gdb)
```

[New Thread 666]

There are three threads listed. Notice that the “*” is next to thread 2 in the example above. That means thread 2 is the current thread being debugged – the one that hit the breakpoint. You can also type the command “thread” to list the current thread.

In the source code frame, hover the mouse over the variable “self” and the popup will show the decimal value of the “self” variable. Right-click on “self” and choose “Display self”. In the top frame, the variable self will be displayed. Right-click on that “self” display at the top and choose “New Display -> Convert to Hex”.


DDD: /home/student/debug-labs/lab06/sched.c
File Edit View Program Commands Status Source Data Help
(): /x self
Lookup Find» Break Watch Print Disp* Plot Hide Rotate Set Undisp
1: self
1084068976
0x409d9470
DDD
Run
Interrupt
Step Stepi
Next Nexti
Until Finish
Cont Kill
Up Down
Undo Redo
Edit Make

```
volatile int        thread_switches;
volatile int        done = 0;

void *thread(void *x)
{
    pthread_t  self = pthread_self();
    long long  self_count = 0;
    long long  avg = 0;
    int        avg_calculations = 0;

    /* let all threads start */
    sleep(1);

    while(1) {
        if (running == self) {   /* still this thread running */
            self_count++;
        } else {                 /* just switched to this thread */
            running = self;
```

```
* 2 Thread 665  thread (x=0xbee87c70) at sched.c:21
  1 Thread 663  0x40133544 in nanosleep () from
/home/student/targetfs/lib/libc.so.6
(gdb) thread
[Current thread is 2 (Thread 665)]
(gdb) graph display self
(gdb) graph display /x self dependent on 1
(gdb)
```

Display 2: /x self (enabled, scope thread, address 0x409d8dc8)

Now click "Cont" again to continue. Do this several times, watching the top display of the value of "self". Notice that sometimes the breakpoint is hit by the same thread multiple times, and sometimes there is a thread switch. This breakpoint is not thread-specific. It stops any thread that hits it.

You can also change the thread being debugged from the "(gdb)" prompt, even without that thread hitting a breakpoint. For example, at the "(gdb)" prompt, enter the following:

```
(gdb) thread 2

(gdb) p/x self
```

Followed by:

```
(gdb) thread 3

(gdb) p/x self
```

This capability lets you view and debug the context of any thread.

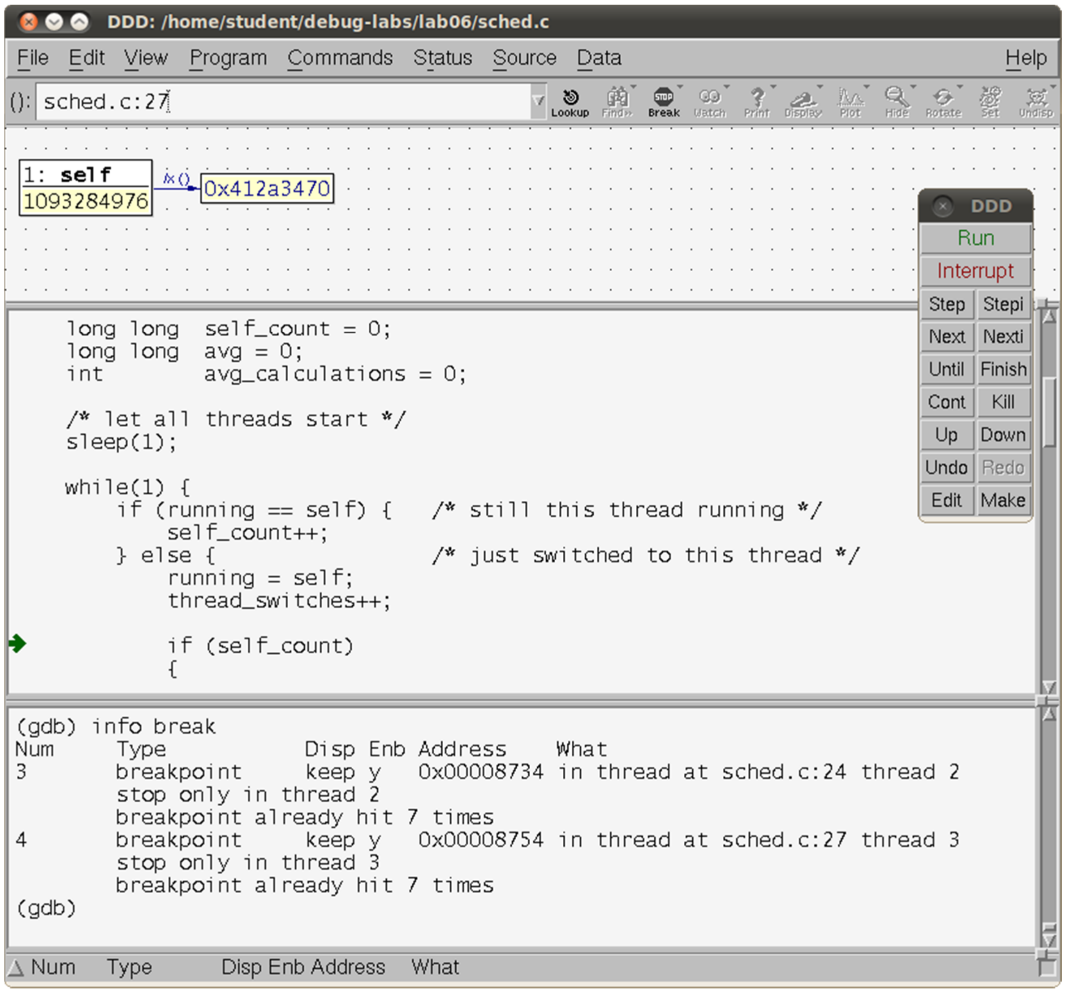
Remove the existing breakpoint by right-clicking the stop sign and choosing "Delete Breakpoint". Now we will set some thread-specific breakpoints. At the "(gdb)" prompt enter the following:

```
(gdb) break sched.c:24 thread 2

(gdb) break sched.c:27 thread 3
```

This sets a thread-specific breakpoint for thread 2 at line 24 of sched.c. Also we set a breakpoint specific to thread 3 at sched.c line 27. We do not see the stop signs for these breakpoints, but they are still effective.

Click "Cont" a few times. You will see that when the program stops at line 24, it is always thread 2. Likewise, when the program stops at line 27, it is always thread 3.

At the "(gdb)" prompt enter the command "info break". This information on our breakpoints is very helpful, including how many times each breakpoint is hit, to which thread the breakpoint applies, etc.


DDD: /home/student/debug-labs/lab06/sched.c
File Edit View Program Commands Status Source Data Help
(): sched.c:27
Lookup Find Break Watch Print Display Plot Hide Rotate Set Undisp
1: self
1093284976
0x412a3470
DDD
Run
Interrupt
Step Stepi
Next Nexti
Until Finish
Cont Kill
Up Down
Undo Redo
Edit Make
Num Type Disp Enb Address What

```
    long long  self_count = 0;
    long long  avg = 0;
    int        avg_calculations = 0;

    /* let all threads start */
    sleep(1);

    while(1) {
        if (running == self) {    /* still this thread running */
            self_count++;
        } else {                  /* just switched to this thread */
            running = self;
            thread_switches++;

            if (self_count)
            {
```

```
(gdb) info break
Num     Type           Disp Enb Address    What
3       breakpoint     keep y   0x00008734 in thread at sched.c:24 thread 2
        stop only in thread 2
        breakpoint already hit 7 times
4       breakpoint     keep y   0x00008754 in thread at sched.c:27 thread 3
        stop only in thread 3
        breakpoint already hit 7 times
(gdb)
```

Our program is written to finish when the global variable "done" is set to nonzero. We can do that at the same command prompt by using the command
"set var done=1". Enter this command.

```
(gdb) set var done=1
```

It is useful during debugging to be able to easily set variable values using this method.

Click “Cont” again, twice, so the threads can get through their breakpoints and finish up. In the ssh window, you will see the output of the program.

Exit from ddd.

Lab Completed.