Debugging Applications

- At some point in the development process the use of debugging tools is inevitable. Students learning programming rely heavily on simple techniques such as “printf”, “cout”, statements and such. There are many reasons why such simplistic techniques must not be used in professional environments:

    - Significant loss of valuable time
    - Simplistic techniques are not very informative
    - Results in a very incoherent and mind-numbing debugging process
    - Inserting and removing these print statements results in large amounts wasted time

- At a professional level (where applications are much more complex than simple assignments) sophisticated debugging tools need to be brought to bear on the problem.

- Professional software developers make heavy usage of debugging tools in order to save time, costs, and minimize frustration.

- The first thing to keep in mind is that a debugging tool will not tell you what the bug is in the software, but it will direct you to the location where the problem is.

- The main purpose of a debugger is to allow you to see what is going on inside a program as it executes its code, or what the applications was doing at the moment it crashed.

- Any good debugger will provide the following (minimal) functionality:

    - Run a program and supply any initial arguments that can be used to trace its execution.
    - Ability to insert “break points” that will stop the execution of code at specified points in the application.
    - Ability to stop code execution given a specified set of conditions.
    - Examine variables to determine what caused the program to stop or crash.
    - Modify program variables so you can experiment with possible solutions for observed bugs.

- Two of the most popular debugging tools are “**gdb**” and “**gprof**”, released under the GNU General Public License (GPL).

Debugging Programs on Linux

- The debugging tool of choice on *nix systems is the **GNU debugger** or "**gdb**". Using gdb we can examine the values of variables while the program is running, set breakpoints, backtrace and so on.

- The debug compilation option works by storing the names and source code line-numbers of functions and variables in a *symbol table* in the object file or executable.

- The debugging information has to added into the executable and object files at compile time using the "-g" *debug option*.

- This debugging information allows errors to be traced back from a specific machine instruction to a corresponding line in the original source code.

"core" Files

- An important function of the debug option is that it provides a developer with the ability to examine the cause of a program crash from what is called a "core dump".

- When a program exits abnormally (i.e. crashes) the operating system creates a *core file* (usually named 'core') which contains the complete memory image of the program at the time it crashed. This file is often referred to as a ***core dump***.

- Combined with information from the symbol table produced by '-g', the core dump can be used to find the line where the program crashed, and the values of its variables at that point.

- This is useful both during the development of software and after deployment, since it allows problems to be investigated when a program has crashed "in the field".

Example

- Consider the simple code example provided that contains an invalid memory access bug, which we will use to produce a core file.

- The program attempts to dereference a null pointer p, which is an invalid operation. On most systems, this will cause the program to crash.

- The first step is to compile the program with the debug option as follows:

```
$ gcc -Wall –ggdb -o bptr  bptr.c
```

- Note that a null pointer will only cause a problem at run-time, so the **–Wall** option does not produce any warnings.

- Here is an example of the output that will be produced by running the executable on a Linux system:

```
$ ./bptr
Segmentation fault
$
```

- Notice that the system reported a segmentation fault but there was no core dump. This is because some systems are configured not to create core files by default, since the core files can be very large and rapidly fill up the disk space on a system.

- In the *Bash* shell the command ***ulimit -c*** controls the maximum size of core files. If the size limit is zero, no core files will be produced. The current size limit can be shown by typing the following command:

```
$ ulimit -c
0
```

- A zero result as above indicates that the creation of core files is disabled. This can be changed to allow core files of any size as follows (The size limit for core files can also be set to a specific value in kilobytes):

```
$ ulimit -c unlimited
```

- Now if we run the program again, we get the following output:

```
$ ./bptr
Segmentation fault (core dumped)
$
```

- Note that now the error message "**core dumped**" is displayed, and the operating system will generate a file called "**core.xxxx..**" (where "**xxxx..**" is a long name containing the executable filename, pid, etc.) in "**/var/lib/systemd/coredump**".

- This core file contains a complete image of the pages of memory used by the program at the time it was terminated.

- The term ***segmentation fault*** refers to the fact that the program tried to access a restricted memory "segment" outside the area of memory which had been allocated to it.

- Now, in order to read the core files, we have to use the have to use "***coredumpctl***" utility.

- For example, to view all the core files we can use:

```
# coredumpctl
TIME                            PID   UID   GID SIG PRESENT EXE
Mon 2016-05-16 10:22:47 PDT   62524     0     0   6   /usr/libexec/drkonqi
Mon 2016-07-18 13:12:28 PDT     774     0     0   6   /usr/lib/systemd/systemd-logind
Wed 2017-01-11 14:52:43 PST   26583     0     0  11 * /home/aman/debugging/bptr
Wed 2017-01-11 14:53:01 PST   26594     0     0  11 * /home/aman/debugging/bptr
Wed 2017-01-11 14:53:45 PST   26644     0     0  11 * /home/aman/debugging/bptr
Wed 2017-01-11 14:58:04 PST   26719     0     0  11 * /home/aman/debugging/bptr
```

- If we want to extract a core file and analyze it in our current directory we can do so as follows:

  coredumpctl -o bptr.core dump /home/aman/debugging/bptr

- This will create a file called "**bptr.core**", which we can load into **gdb**.
- Alternatively we can automatically load the last core dump directly into ***gdb*** as follows:

  coredumpctl gdb

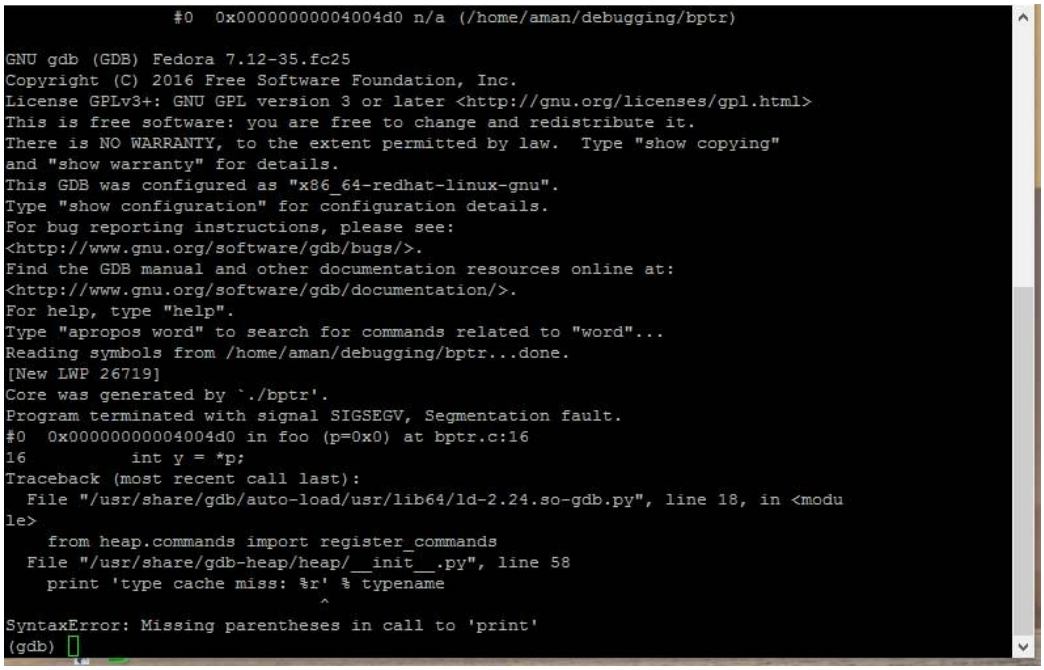
- Or we load a specific core file using its pid as follows:

  coredumpctl gdb [pid]

- Let us load a core file into the GNU Debugger GDB as follows:

  # coredumpctl gdb 26719

- The following screenshot shows the relevant details from gdb:



- Note that both the original executable file and the core file are required for debugging—it is not possible to debug a core file without the corresponding executable.

- The debugger immediately begins printing diagnostic information, and shows a listing of the line where the program crashed (line 16). The displayed information also indicates that the value of p is a null pointer.

- The last line (**gdb**) is the GNU Debugger prompt—it indicates that further commands can be entered at this point.

- As a start we can display the value of the pointer p using the ***print*** command as follows:

```
(gdb) print p
$1 = (int *) 0x0
```

- This shows that p is a null pointer **(0x0) of type 'int *'**, so we know that dereferencing it with the expression ***p** in this line has caused the segmentation fault.

Displaying a backtrace

- The debugger can also show the function calls and arguments up to the current point of execution—this is called a *stack backtrace* and is displayed with the command backtrace:

```
(gdb) backtrace
#0  0x00000000004004a3 in foo (p=0x0) at bptr.c:16
#1  0x0000000000400495 in main () at bptr.c:11
(gdb)
```

- In this case, the backtrace shows that the crash occurred at **line 16** after the function **foo** was called from main with an argument of **p=0x0** at **line 11** in 'bptr.c'.

- It is also possible to move to different levels in the stack trace, and examine their variables, using the debugger commands up and down:

```
(gdb) up
#1  0x0000000000400495 in main () at bptr.c:11
11        return foo (p);
(gdb) down
#0  0x00000000004004a3 in foo (p=0x0) at bptr.c:16
16        int y = *p;
(gdb)
```

Setting a breakpoint

- A *breakpoint* stops the execution of a program and returns control to the debugger, where its variables and memory can be examined before continuing program execution.

- Breakpoints can be set for specific functions, lines or memory locations with the ***break*** command. To set a breakpoint on a specific function, use the command ***break function-name***.

- For example, the following command sets a breakpoint at the start of the main function in the program above, we first load the executable into gdb:

```
$ gdb ./bptr
GNU gdb Fedora (6.8-32.fc10)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"…
(gdb)
```

- Next we set a breakpoint at the main function:

```
(gdb) break main
Breakpoint 1 at 0x400484: file bptr.c, line 10.
(gdb)
```

- The debugger will now take control of the program when the function main is called. Since the main function is the first function to be executed in a C program the program will stop immediately when it is run:

```
(gdb) run
Starting program: /home/aman/debugging/bptr

Breakpoint 1, main () at bptr.c:10
10        int *p = 0;
(gdb)
```

- The display shows the line that will be executed next (the line number is shown on the left). The breakpoint stops the program *before* the line is executed, so at this stage the pointer **p** is undefined and has not yet been set to zero.

Stepping through the program

- To move forward and execute the line displayed above, we use the ***step*** command:

```
(gdb) step
11      return foo (p);
(gdb)
```

- After executing line 10, the debugger displays the next line to be executed. The pointer **p** will now have been set to zero (null):

```
(gdb) print p
$1 = (int *) 0x0
(gdb)
```

- The command step will follow the execution of the program interactively through any functions that are called in the current line. To move forward without tracing these calls, use the ***next*** command.

Modifying variables

- To temporarily fix the null pointer bug discovered above, we can change the value of p in the running program using the ***set*** variable command.

- Variables can be set to a specific value, or to the result of an expression, which may include function calls. This powerful feature allows functions in a program to be tested interactively through the debugger.

- In this case we will interactively allocate some memory for the pointer **p** using the function ***malloc***, storing the value 255 in the resulting location (note that the value **0x601010** is address assigned to the pointer p by malloc):

```
(gdb) set variable p = malloc(sizeof(int))
(gdb) print p
$2 = (int *) 0x601010
(gdb) set variable *p = 255
(gdb) print *p
$3 = 255
(gdb)
```

- If we now continue stepping through the program with the new value of p the previous segmentation fault will not occur:

```
(gdb) step
foo (p=0x601010) at bptr.c:16
16        int y = *p;
(gdb) step
17        return y;
(gdb)
```

Continuing execution

The finish command continues execution up to the end of the current function, displaying the return value:

```
(gdb) finish
Run till exit from #0  foo (p=0x601010) at bptr.c:17
main () at bptr.c:12
12     }
Value returned is $4 = 255
(gdb)
```

- To continue execution until the program exits (or hits the next breakpoint) use the command continue,

```
(gdb) Quit
(gdb) continue
Continuing.

Program exited with code 0377.
(gdb)
```

- Note that the exit code is shown in octal (**0377 base 8 = 255 in base 10**).

Debugging Programs with Multiple Processes

- When a program forks, GDB will continue to debug the parent process and the child process will run unimpeded. If you have set a breakpoint in any code which the child then executes, the child will get a SIGTRAP signal which (unless it catches the signal) will cause it to terminate.

- However, if you want to debug the child process there is a workaround. Insert a call to the ***sleep*** function in the code which the child process executes after the fork.

- While the child is sleeping, use the ***ps*** program to get its process ID. Then tell GDB (a new invocation of GDB if you are also debugging the parent process) to attach to the child process.

- As mentioned above, the default action for GDB is to continue to debug the parent process and the child process will run unimpeded.

- If we want to follow the child process instead of the parent process, the **set follow-fork-mode** command is used:

    - set follow-fork-mode *mode*

- The *mode* can be:

    - parent

      The original process is debugged after a fork. The child process runs unimpeded. This is the default.

    - child

      The new process is debugged after a fork. The parent process runs unimpeded.

    - ask

      The debugger will ask for one of the above choices.

- These commands are best illustrated with examples. Consider the sample code (**pipe3.c**) provided. The program has two processes, the parent process that reads from a pipe and a child process that writes to the pipe.

- We compile with debugging turned on and start GDB:

```
$ gcc -Wall -ggdb -o pipe3 pipe3.c
$ gdb ./pipe3
GNU gdb Fedora (6.8-32.fc10)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
(gdb)
```

- Next, we set some breakpoints (The command **tb** is an abbreviation for **tbreak**, a temporary breakpoint and **c** is an abbreviation for **continue, n** = next, **r** = run, and so on). We set a temporary break at **main** and then a breakpoint in the child process at line 91:

```
(gdb) tb main
Breakpoint 1 at 0x4007a4: file pipe3.c, line 29.
(gdb) b 91
Breakpoint 2 at 0x4008c0: file pipe3.c, line 91.
(gdb) set follow-fork-mode child
```

- Now we set GDB to follow the child process (note that the parent process will continue executing normally) and run the program:

```
(gdb) r
Starting program: /home/aman/debugging/pipe3
main () at pipe3.c:29
29      if (pipe(pfd) < 0)
(gdb)
```

- We observe that execution stops at the temporary breakpoint. We continue execution at the next breakpoint in the child process:

```
(gdb) c
Continuing.
[Switching to process 32567]

Breakpoint 2, child (p=0x7fffffffe550) at pipe3.c:91
91      write (p[1], msg1, MSGSIZE);
(gdb)
```

- Now we observe that execution is halted in the child process at line 91. Say we want to start monitoring a variable in the child process, ***count*** for example. We can set a "**watch**" on that variable as follows:

```
(gdb) watch count
Hardware watchpoint 3: count
(gdb)
```

- Next we step through the code execution one instruction at a time until the child exits. Note how the variable value is continuously updated (and also continue to see the writes to ***stdout*** from the parent process):

```
(gdb) n
89      for (count = 0; count < 3; count ++)
(gdb) MSG = Hello World
n
Hardware watchpoint 3: count

Old value = 0
New value = 1
0x00000000004008df in child (p=0x7fffffffe550) at pipe3.c:89
89      for (count = 0; count < 3; count ++)
(gdb) n

Breakpoint 2, child (p=0x7fffffffe550) at pipe3.c:91
91        write (p[1], msg1, MSGSIZE);
(gdb) n
89      for (count = 0; count < 3; count ++)
(gdb) MSG = Hello World
n
Hardware watchpoint 3: count

Old value = 1
New value = 2
0x00000000004008df in child (p=0x7fffffffe550) at pipe3.c:89
89      for (count = 0; count < 3; count ++)
(gdb) n

Breakpoint 2, child (p=0x7fffffffe550) at pipe3.c:91
91        write (p[1], msg1, MSGSIZE);
(gdb) n
89      for (count = 0; count < 3; count ++)
(gdb) MSG = Hello World
n
Hardware watchpoint 3: count

Old value = 2
New value = 3
0x00000000004008df in child (p=0x7fffffffe550) at pipe3.c:89
89      for (count = 0; count < 3; count ++)
(gdb) n
95      write (p[1], msg2, MSGSIZE);
(gdb) n
96      exit(0);
```

(gdb) End of Conversation

Program exited normally.
(gdb)

Debugging an already-running process

- **attach** ***process-id***

  This command attaches to a running process--one that was started outside GDB. (**info files** shows your active targets.)

- The command takes as argument a process ID. The usual way to find out the process-id of a Unix process is with the ***ps*** utility.

- Note that ***attach*** does not repeat if you press Enter a second time after executing the command.

- When ***attach*** is first used, the debugger finds the program running in the process first by looking in the current working directory, then (if the program is not found) by using the source file search path We can also use the file command to load the program.

- The first thing GDB does after arranging to debug the specified process is to stop it. We can then examine and modify an attached process with all the GDB commands that are ordinarily available when we start processes with run.

- We can insert breakpoints, step and continue, modify storage, etc. If you would rather the process continue running, you may use the continue command after attaching GDB to the process.

- **detach**

- When we have finished debugging the attached process, use the **detach** command to release it from GDB control.

- Detaching the process continues its execution. After the detach command, that process and GDB become completely independent once more.

- Note that if you exit GDB or use the **run** command while you have an attached process, you kill that process. By default, GDB asks for confirmation if you try to do either of these things; you can control whether or not you need to confirm by using the **set confirm** command

Multithreaded Application Debugging

- Debugging multithreaded applications is difficult at best. However, GDB does offer some useful capabilities that provide assistance in this area of debugging.

- Just as before, the breakpoint is the most important aspects of debugging, but its behavior is different in multithreaded applications. This is best illustrated with a practical example.

- Consider the simple multithreaded example presented earlier in the course. This example is now converted to a C++ version.

- Each thread just prints out a string associated with its passed argument and its iteration 100 times. After each print to *stdout* it sleeps for a duration of one second.

- As usual, we compile the code with the debug option as follows:

  $ g++ -Wall -ggdb -o threads threads.cpp -lpthread

- So for example, we might see an output as follows:

```
$ ./threads
Hello: 0
World: 0
Hello: 1
World: 1
Hello: 2
World: 2
Hello: 3
World: 3
Hello: 4
World: 4
Hello: 5
World: 5
^C
$
```

- Now in order to trace the program let us examine its behavior using GDB (The command **tb** is an abbreviation for **tbreak**, a temporary breakpoint and **c** is an abbreviation for **continue, n** = next, **r** = run, and so on):

```
$ gdb ./threads
GNU gdb Fedora (6.8-32.fc10)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
(gdb) tb main
Breakpoint 1 at 0x400e02: file threads.cpp, line 16.
(gdb) r
Starting program: /home/aman/debugging/threads
[Thread debugging using libthread_db enabled]
[New Thread 0x7ffff7fd2700 (LWP 29773)]
main (argc=1, argv=0x7fffffffe638) at threads.cpp:16
16      string n1("Hello"), n2("World");
(gdb)
```

- At this point none of the threads have been spawned yet and if we examine the number of threads all we should see is the main thread of execution within the process:

```
(gdb) info threads
* 1 Thread 0x7ffff7fd2700 (LWP 29773)  main (argc=1, argv=0x7fffffffe638) at threads.cpp:16
(gdb)
```

- At this point it would be useful (especially when considering where to insert a breakpoint) to list some of the code associated with the threads:

```
(gdb) list 36,45
36      for (int i = 0; i < 100; i++)
37      {
38        pthread_mutex_lock (&lock);
39        cout << name << ": " << i << endl;
40        pthread_mutex_unlock (&lock);
41
42        sleep.tv_sec = 1;
43        nanosleep (&sleep, NULL);
44      }
45      return NULL;
(gdb)
```

- The following will insert a breakpoint at line 40; and we continue execution up to the breakpoint:

```
(gdb) b 40
Breakpoint 2 at 0x400d78: file threads.cpp, line 40.
(gdb) c
Continuing.
[New Thread 0x7ffff7fd1950 (LWP 29793)]
Hello: 0
[Switching to Thread 0x7ffff7fd1950 (LWP 29793)]

Breakpoint 2, outputMsg (arg=0x7fffffffe530) at threads.cpp:40
40          pthread_mutex_unlock (&lock);
(gdb)
```

- Observe that the first thread gets to write a line to *stdout* before reaching the breakpoint. We can now investigate the number of running threads in our program as follows:

```
(gdb) info threads
* 2 Thread 0x7ffff7fd1950 (LWP 29793)  outputMsg (arg=0x7fffffffe530) at threads.cpp:40
  1 Thread 0x7ffff7fd2700 (LWP 29773)  0x00000035b4ee2bea in mmap64 () from
/lib64/libc.so.6
(gdb)
```

- It appears that so far we have 2 threads, that is only one new thread has been spawned (thread 2) and it is currently the active thread (signified by the asterisk).

- We step a few more times until we hit the breakpoint again and examine the threads:

```
(gdb) n
42          sleep.tv_sec = 1;
(gdb) n
43          nanosleep (&sleep, NULL);
(gdb) n
[New Thread 0x7ffff75d0950 (LWP 29803)]
World: 0
[Switching to Thread 0x7ffff75d0950 (LWP 29803)]

Breakpoint 2, outputMsg (arg=0x7fffffffe520) at threads.cpp:40
40          pthread_mutex_unlock (&lock);
(gdb) info threads
* 3 Thread 0x7ffff75d0950 (LWP 29803)  outputMsg (arg=0x7fffffffe520) at threads.cpp:40
  2 Thread 0x7ffff7fd1950 (LWP 29793)  0x00000035b5a0e851 in nanosleep () from /lib64/libpthread.so.0
  1 Thread 0x7ffff7fd2700 (LWP 29773)  0x00000035b5a07cb5 in pthread_join
(threadid=140737353947472, thread_return=0x0) at pthread_join.c:89
(gdb)
```

- Now we observe that there are three threads and the active thread is thread 3. This is because the previous thread 2 has already unlocked the mutex.

- We step through a three more times and we notice that control goes back to thread 2:

```
(gdb) n
42        sleep.tv_sec = 1;
(gdb) n
43        nanosleep (&sleep, NULL);
(gdb) info threads
* 3 Thread 0x7ffff75d0950 (LWP 29803)  outputMsg (arg=0x7fffffffe520) at threads.cpp:43
  2 Thread 0x7ffff7fd1950 (LWP 29793)  outputMsg (arg=0x7fffffffe530) at threads.cpp:36
  1 Thread 0x7ffff7fd2700 (LWP 29773)  0x00000035b5a07cb5 in pthread_join
(threadid=140737353947472, thread_return=0x0) at pthread_join.c:89
(gdb) n
[Switching to Thread 0x7ffff7fd1950 (LWP 29793)]
36      for (int i = 0; i < 100; i++)
(gdb) info threads
  3 Thread 0x7ffff75d0950 (LWP 29803)  0x00000035b5a0e851 in nanosleep () from /lib64/libpthread.so.0
* 2 Thread 0x7ffff7fd1950 (LWP 29793)  outputMsg (arg=0x7fffffffe530) at threads.cpp:36
  1 Thread 0x7ffff7fd2700 (LWP 29773)  0x00000035b5a07cb5 in pthread_join
(threadid=140737353947472, thread_return=0x0) at pthread_join.c:89
(gdb)
```

- At this point we can switch to thread 3 if we wish using the thread command as follows:

```
(gdb) info threads
  3 Thread 0x7ffff75d0950 (LWP 29803)  outputMsg (arg=0x7fffffffe520) at threads.cpp:42
* 2 Thread 0x7ffff7fd1950 (LWP 29793)  outputMsg (arg=0x7fffffffe530) at threads.cpp:42
  1 Thread 0x7ffff7fd2700 (LWP 29773)  0x00000035b5a07cb5 in pthread_join
(threadid=140737353947472, thread_return=0x0) at pthread_join.c:89
(gdb) thread 3
[Switching to thread 3 (Thread 0x7ffff75d0950 (LWP 29803))]#0  outputMsg (arg=0x7fffffffe520) at
threads.cpp:42
42        sleep.tv_sec = 1;
(gdb)
```

- We notice that thread 3 is currently in sleep mode. We step a couple more times and observe that the thread 3 gets preempted and thread 2 locks the mutex:

```
(gdb) n
43        nanosleep (&sleep, NULL);
(gdb) n
[Switching to Thread 0x7ffff7fd1950 (LWP 29793)]
38        pthread_mutex_lock (&lock);
(gdb) info threads
  3 Thread 0x7ffff75d0950 (LWP 29803)  0x00000035b5a0e851 in nanosleep () from /lib64/libpthread.so.0
* 2 Thread 0x7ffff7fd1950 (LWP 29793)  outputMsg (arg=0x7fffffffe530) at threads.cpp:38
  1 Thread 0x7ffff7fd2700 (LWP 29773)  0x00000035b5a07cb5 in pthread_join
(threadid=140737353947472, thread_return=0x0) at pthread_join.c:89
(gdb)
```

- As you step through a multithreaded program, you will find that the focus of the debugger can change at any step. This can be annoying, especially when the current thread is what you are interested in debugging.

- We can instruct GDB not to preempt the current thread by locking the scheduler as follows. First we set the debugger focus to thread 3:

```
(gdb) info threads
  3 Thread 0x7ffff75d0950 (LWP 29803)  0x00000035b5a0e851 in nanosleep () from /lib64/libpthread.so.0
* 2 Thread 0x7ffff7fd1950 (LWP 29793)  outputMsg (arg=0x7fffffffe530) at threads.cpp:38
  1 Thread 0x7ffff7fd2700 (LWP 29773)  0x00000035b5a07cb5 in pthread_join
(threadid=140737353947472, thread_return=0x0) at pthread_join.c:89
(gdb) Quit
(gdb) thread 3
[Switching to thread 3 (Thread 0x7ffff75d0950 (LWP 29803))]#0  0x00000035b5a0e851 in nanosleep ()
from /lib64/libpthread.so.0
Current language:  auto; currently asm
(gdb) info threads
* 3 Thread 0x7ffff75d0950 (LWP 29803)  0x00000035b5a0e851 in nanosleep () from /lib64/libpthread.so.0
  2 Thread 0x7ffff7fd1950 (LWP 29793)  outputMsg (arg=0x7fffffffe530) at threads.cpp:38
  1 Thread 0x7ffff7fd2700 (LWP 29773)  0x00000035b5a07cb5 in pthread_join
(threadid=140737353947472, thread_return=0x0) at pthread_join.c:89
```

- Next we lock the scheduler and as we step through we notice that the debugger focus stays on thread 3:

```
(gdb) set scheduler-locking on
 (gdb) info threads
* 3 Thread 0x7ffff75d0950 (LWP 29803)  0x00000035b5a0e851 in nanosleep () from /lib64/libpthread.so.0
  2 Thread 0x7ffff7fd1950 (LWP 29793)  outputMsg (arg=0x7fffffffe530) at threads.cpp:38
  1 Thread 0x7ffff7fd2700 (LWP 29773)  0x00000035b5a07cb5 in pthread_join
(threadid=140737353947472, thread_return=0x0) at pthread_join.c:89
(gdb) n
Single stepping until exit from function nanosleep,
which has no line number information.
outputMsg (arg=0x7fffffffe520) at threads.cpp:36
36      for (int i = 0; i < 100; i++)
Current language:  auto; currently c++
(gdb) n
38        pthread_mutex_lock (&lock);
(gdb) n
39        cout << name << ": " << i << endl;
(gdb) n
World: 3

Breakpoint 2, outputMsg (arg=0x7fffffffe520) at threads.cpp:40
40        pthread_mutex_unlock (&lock);
(gdb) n
42        sleep.tv_sec = 1;
(gdb) n
43        nanosleep (&sleep, NULL);
(gdb) n
36      for (int i = 0; i < 100; i++)
(gdb) info threads
```

*** 3 Thread 0x7ffff75d0950 (LWP 29803) outputMsg (arg=0x7fffffffe520) at threads.cpp:36**
** 2 Thread 0x7ffff7fd1950 (LWP 29793) outputMsg (arg=0x7fffffffe530) at threads.cpp:38**
** 1 Thread 0x7ffff7fd2700 (LWP 29773) 0x00000035b5a07cb5 in pthread_join**
(threadid=140737353947472, thread_return=0x0) at pthread_join.c:89
(gdb)

- In order to allow other threads to be active we can turn the locking mode off:

(gdb) show scheduler-locking
Mode for locking scheduler during execution is "on".
(gdb) set scheduler-locking off
(gdb) show scheduler-locking
Mode for locking scheduler during execution is "off".
(gdb)

- Finally we delete the break point and exit:

(gdb) info b
Num Type Disp Enb Address What
2 breakpoint keep y 0x0000000000400d78 in outputMsg(void*) at threads.cpp:40
** breakpoint already hit 7 times**
(gdb) d 2
(gdb) q
The program is running. Exit anyway? (y or n) y
$