Chapter 6

Debugging In A Multiple-Activities Context

Debugging is difficult to begin with, and it becomes even more challenging when the misbehaving application is trying to coordinate multiple, simultaneous activities: client/server network programming, programming with threads, and parallel processing are examples of this paradigm. This chapter presents an overview of the most commonly used multiprogramming techniques and offers some tips on how to deal with bugs in these kinds of programs, focusing on the use of GDB/DDD in the debugging process.

6.1 Debugging Client/Server Network Programs

Computer networks are extremely complex systems, and rigorous debugging of networked software applications can sometimes require the use of hardware monitors to collect detailed information about the network traffic. An entire book could be written on this debugging topic alone. Our goal here is simply to introduce the subject.

Our example consists of the following *client/server pair*. The client application allows a user to check the load on the machine on which the server application runs, even if the user does not have an account on the latter machine. The client sends a request for information to the server—here, a query about the load on the server's system, via the Unix **w** command—over a network connection. The server then processes the request and returns the results, capturing the output of **w** and

sending it back over the connection. In general, a server can accept requests from multiple remote clients; to keep things simple in our example, we assume there is only instance of the client.

The code for the server is shown below:

```
1  // srvr.c
2
3  //  a server to remotely run the w command
4  //  user can check load on machine without login privileges
5  //  usage:  svr
6
7  #include <stdio.h>
8  #include <sys/types.h>
9  #include <sys/socket.h>
10 #include <netinet/in.h>
11 #include <netdb.h>
12 #include <fcntl.h>
13 #include <string.h>
14 #include <unistd.h>
15 #include <stdlib.h>
16
17 #define WPORT 2000
18 #define BUFSIZE 1000  // assumed sufficient here
19
20 int clntdesc,  // socket descriptor for individual client
21     svrdesc;  // general socket descriptor for server
22
23 char outbuf[BUFSIZE];  // messages to client
24
25 void respond()
26 {  int fd,nb;
27
28    memset(outbuf,0,sizeof(outbuf));  // clear buffer
29    system("w > tmp.client");  // run 'w' and save results
30    fd = open("tmp.client",O_RDONLY);
31    nb = read(fd,outbuf,BUFSIZE);  // read the entire file
32    write(clntdesc,outbuf,nb);  // write it to the client
33    unlink("tmp.client");  // remove the file
34    close(clntdesc);
35 }
36
37 int main()
38 {  struct sockaddr_in bindinfo;
39
40    // create socket to be used to accept connections
41    svrdesc = socket(AF_INET,SOCK_STREAM,0);
42    bindinfo.sin_family = AF_INET;
43    bindinfo.sin_port = WPORT;
44    bindinfo.sin_addr.s_addr = INADDR_ANY;
45    bind(svrdesc,(struct sockaddr *) &bindinfo,sizeof(bindinfo));
46
47    // OK, listen in loop for client calls
48    listen(svrdesc,5);
49
```

```
50      while (1)  {
51         // wait for a call
52         clntdesc = accept(svrdesc,0,0);
53         // process the command
54         respond();
55      }
56   }
```

Here is the code for the client:

```
1    // clnt.c
2
3    // usage:  clnt server_machine
4
5    #include <stdio.h>
6    #include <sys/types.h>
7    #include <sys/socket.h>
8    #include <netinet/in.h>
9    #include <netdb.h>
10   #include <string.h>
11   #include <unistd.h>
12
13   #define WPORT 2000  // server port number
14   #define BUFSIZE 1000
15
16   int main(int argc, char **argv)
17   {  int sd,msgsize;
18
19      struct sockaddr_in addr;
20      struct hostent *hostptr;
21      char buf[BUFSIZE];
22
23      // create socket
24      sd = socket(AF_INET,SOCK_STREAM,0);
25      addr.sin_family = AF_INET;
26      addr.sin_port = WPORT;
27      hostptr = gethostbyname(argv[1]);
28      memcpy(&addr.sin_addr.s_addr,hostptr->h_addr_list[0],hostptr->h_length);
29
30      // OK, now connect
31      connect(sd,(struct sockaddr *) &addr,sizeof(addr));
32
33      // read and display response
34      msgsize = read(sd,buf,BUFSIZE);
35      if (msgsize > 0)
36         write(1,buf,msgsize);
37      printf("\n");
38      return 0;
39   }
```

For those unfamiliar with client/server programming, here is an overview of how the programs work:

On line 41 of the server code, we create a *socket*, which is an abstraction similar to a file descriptor; just as one uses a file descriptor to perform I/O operations on a filesystem object, one reads from and writes to a network connection via a socket. On line 45, the socket is bound to a specific *port number*, arbitrarily chosen to be 2000. (User-level applications such as this one are restricted to port numbers of 1024 and higher.) This number identifies a "mailbox" on the server's system to which clients send requests to be processed for this particular application.

The server "opens for business" by calling `listen()` on line 48. It then waits for a client request to come in, by calling `accept()` on line 52. That call blocks until a request arrives. It then returns a new socket for communicating with the client. (When there are multiple clients, the original socket continues to accept new requests even while an existing request is being serviced, hence the need for separate sockets. This would require the server to be implemented in a threaded fashion.) The server processes the client request with the `respond()` function and sends the machine load information to the client by locally invoking the **w** command and writing the results to the socket in line 32.

The client creates a socket on line 24, and then uses it on line 31 to connect to the server's port 2000. On line 34, it reads the load information sent by the server, and then prints it out.

Here is what the output of the client should look like:

```
$ clnt laura.cs.ucdavis.edu
 13:00:15 up 13 days, 39 min,  7 users,  load average: 0.25, 0.13, 0.09
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU WHAT
matloff  :0       -                14Jun07 ?xdm?  25:38   0.15s -/bin/tcsh -c /
matloff  pts/1    :0.0             14Jun07 17:34   0.46s  0.46s -csh
matloff  pts/2    :0.0             14Jun07 18:12   0.39s  0.39s -csh
matloff  pts/3    :0.0             14Jun07 58.00s  2.18s  2.01s /usr/bin/mutt
matloff  pts/4    :0.0             14Jun07  0.00s  1.85s  0.00s clnt laura.cs.u
matloff  pts/5    :0.0             14Jun07 20.00s  1.88s  0.02s script
matloff  pts/7    :0.0             19Jun07  4days 22:17   0.16s -csh
```

Now suppose the programmer had forgotten line 26 in the client code, which specifies the port on the server's system to connect to:

```
   addr.sin_port = WPORT;
```

Let's pretend that we don't know what the bug is, and see how we might track it down.

The client's output would now be

```
$ clnt laura.cs.ucdavis.edu

$
```

It appears that the client received nothing at all back from the server. This of course could be due to a variety of causes, in either the server or the client, or both.

Let's take a look around, using GDB. First, let's check to see that the client actually did succeed in connecting to the server. We'll set a breakpoint at the call to `connect()`, and run the program:

```
(gdb) b 31
Breakpoint 1 at 0x8048502: file clnt.c, line 31.
(gdb) r laura.cs.ucdavis.edu
Starting program: /fandrhome/matloff/public_html/matloff/public_html/Debug/Book/DDD/clnt laura.cs.ucda

Breakpoint 1, main (argc=2, argv=0xbf81a344) at clnt.c:31
31          connect(sd,(struct sockaddr *) &addr,sizeof(addr));
```

We use GDB to execute the `connect()` and check the return value for an error condition:

```
(gdb) p connect(sd,&addr,sizeof(addr))
$1 = -1
```

It is indeed -1, the code for failure. That is a big hint. (Of course, as a matter of defensive programming, when we wrote the client code, we would have checked the return value of `connect()` and handled the case of failure to connect.)

By the way, note that in manually executing the call to `connect()`, we had to remove the cast. With the cast retained, we get an error:

```
(gdb) p connect(sd,(struct sockaddr *) &addr,sizeof(addr))
No struct type named sockaddr.
```

This is due to a quirk in GDB, and arises because we haven't used the struct elsewhere in our program.

Also note that if the `connect()` attempt had succeeded in our GDB session, we could NOT have then gone ahead and executed line 31. Attempting to open an already-open socket is an error.

We would have had to skip over line 31, and go directly to line 34. We could do this using GDB's `jump` command, issuing `jump 34`, but in general one should use this

command with caution, as it might result in skipping some machine instructions which are needed further down in the code. So, if the connection attempt had succeeded, we probably would want to rerun the program.

Let's try to track down the cause of the failure, by checking the argument `addr` in the call to `connect()`:

```
(gdb) p addr
...
connect(3, {sa_family=AF_INET, sin_port=htons(1032), sin_addr=inet_addr("127.0.0.1")}, 16) = -1 ECONNR
...
```

Aha! Port 2052, indeed! Not the 2000 we expect. This suggests that we either misspecified the port, or forgot to specify it altogether. We check, and quickly discover that the latter was the case.

Again, it would have been prudent to include a bit of machinery in our source code to help the debugging process, such as checking the return values of system calls. Another helpful step is inclusion of the line

```
#include <errno.h>
```

which, on our system, creates a global variable `errno`, whose value can be printed out from within our code, or from within GDB:

```
(gdb) p errno
$1 = 111
```

From the file `/usr/include/linux/errno.h`, we find that this error number codes a *connection refused* error.

However, the implementation of the `errno` library may differ from platform to platform. For example, the header file may have a different name, or `errno` may be implemented as a macro call instead of a variable.

Another approach would be to use `strace`, which traces all system calls made by a program:

```
$ strace clnt laura.cs
...
connect(3, {sa_family=AF_INET, sin_port=htons(1032), sin_addr=inet_addr("127.0.0.1")}, 16) = -1 ECONNR
...
```

This gives us two important pieces of information for the price of one. First, we see right away that we had a `ECONNREFUSED` error. Second, we also see that the port was `htons(1032)`, which has the value 2052. We can check this latter value by issuing a command like

```
(gdb) p htons(1032)
```

from within GDB, which shows the value to be 2052, which obviously is not 2000, as expected.

`strace` is a handy tool in many contexts (networked and otherwise) for checking the results of system calls.

As another example, suppose that we accidentally omit the write to the client in the server code (line 32):

```
write(clntdesc,outbuf,nb);  // write it to the client
```

In this case, the client program would hang, waiting for a reply that is not forthcoming. Of course, in this simpleminded example we'd immediately suspect a problem with the call to `write()` in the server, and quickly find that we had forgotten it. But in more complex programs the cause may not be so obvious. In such cases, we would probably set up *two* simultaneous GDB sessions, one for the client and one for the server, stepping through *both* of the programs in tandem. We would find that at some point in their joint operation the client hangs, waiting to hear from the server, and thus obtain a clue to the likely location of the bug within the server. We'd then focus our attention on the server GDB session, trying to figure out why it did not send to the client at that point.

In really complex network debugging cases, the open source `ethereal` program can be used to track individual TCP/IP packets.

6.2 Debugging Threaded Code

Threaded programming has become quite popular. For Unix, the most widespread threads package is the POSIX standard, Pthreads, so we will use it for our example in this section. The principles are similar for other thread packages.

6.2.1 Review of Processes and Threads

Modern operating systems use *timesharing* to manage multiple running programs in such a way that they appear to the user to execute simultaneously. Of course, if the machine has more than one CPU, more than one program actually can run simultaneously, but for simplicity we will assume just one processor, in which case the simultaneity is only apparent.

Each instance of a running program is represented by the OS as a *process* (in Unix terminology) or a *task* (in Windows). Thus, multiple invocations of a single program that execute at the same time (e.g., simultaneous sessions of the `vi` text editor) are distinct processes. Processes have to "take turns" on a machine with one CPU. For concreteness, let's assume that the "turns," called *timeslices*, are of length 30 milliseconds.

After a process has run for 30 milliseconds, a hardware timer emits an interrupt, which causes the OS to run. We say that the process has been *pre-empted*. The OS saves the current state of the interrupted process so it can be resumed later, then selects the next process to give a timeslice to. This is known as a *context switch*, because the CPU's execution environment has switched from one process to another. This cycle repeats indefinitely.

A turn may end early. For example, when a process needs to perform input/output, it ultimately calls a function in the OS that carries out low-level hardware operations; for instance, a call to the C library function `scanf()` results in a call to the Unix OS `read()` system call, which interfaces with the keyboard driver. In this manner the process relinquishes its turn to the OS, and the turn ends early.

One implication of this is that scheduling of timeslices for a given process is rather random. The time it takes for the user to think and then hit a key is random, so the time its next timeslice starts is unpredictable. Moreover, if we are debugging a threaded program, we do not know the order in which the threads will be scheduled; this may make our debugging more difficult.

Here is a bit more detail: The OS maintains a *process table* that lists information about all current processes. Roughly speaking, each process is marked in the table as being in either the Run state or the Sleep state. Let's consider an example in which a running program reaches a point at which it needs to read input from the keyboard. As just noted, this will end the process's turn. Because the process is now waiting for the I/O to complete, the OS marks it as being in the Sleep state, making it ineligible for timeslices. Thus, being in Sleep state means that the process is blocked, waiting for some event to occur. When this event finally occurs

later on, the OS will then change its state in the process table back to Run.

Non-I/O events can trigger a transition to Sleep state as well. For instance, if a parent process creates a child process and calls `wait()`, the parent will block until the child finishes its work and terminates. Again, exactly when this happens is usually unpredictable.

Furthermore, being in the Run state does not mean that the process is actually executing on the CPU; rather, it merely means that it is *ready* to run—that is, eligible for a processor timeslice. Upon a context switch, the OS chooses the process that is next given a turn on the CPU from among those currently in the Run state, according to the process table. The scheduling procedure used by the OS to select the new context guarantees that any given process will keep getting timeslices, and so eventually finish, but there is no promise of *which* timeslices it will receive. Thus, exactly when a sleeping process actually "awakens" once the event that it awaits has occurred is random, as is the exact rate of the process's progress toward completion.

A *thread* is much like a process, except that they are designed to occupy less memory and to take less time to create and switch between than processes do. Indeed, threads are sometimes called "lightweight" processes and, depending on the thread system and run-time environment, may even be implemented as operating system processes. Like programs that spawn processes to get work done, a multithreaded application will generally execute a `main()` procedure that creates one or more child threads. The parent, `main()`, is also a thread.

A major difference between processes and threads is that although each thread has its own local variables, just as is the case for a process, the global variables of the parent program in a threaded environment are shared by all threads, and serve as the main method of communication between the threads. (It is possible to share globals among Unix processes, but inconvenient to do so.)

On a Linux system, you can view all the processes and threads currently on the system by running the command `ps axH`.

There are nonpreemptive thread systems, but Pthreads uses a *preemptive* thread management policy, and a thread in a program can be interrupted at any time by another thread. Thus, the element of randomness described above for processes in a timeshared system also arises in the behavior of a threaded program. As a result, some bugs in applications developed using Pthreads are not readily reproducible.

6.2.2 Basic Example

We'll keep things simple and use the following code for finding prime numbers as our example. The program uses the classic Sieve of Eratosthenes. To find all the primes from 2 through n, we first list all the numbers, then cross out all the multiples of 2, then all the multiples of 3, and so on. Whatever numbers remain at the end are prime numbers.

```
1  // finds the primes between 2 and n; uses the Sieve of Eratosthenes,
2  // deleting all multiples of 2, all multiples of 3, all multiples of 5,
3  // etc.; not efficient, e.g. each thread should do deleting for a whole
4  // block of values of base before going to nextbase for more
5
6  // usage:  sieve nthreads n
7  // where nthreads is the number of worker threads
8
9  #include <stdio.h>
10 #include <math.h>
11 #include <pthread.h>
12
13 #define MAX_N 100000000
14 #define MAX_THREADS 100
15
16 // shared variables
17 int nthreads,  // number of threads (not counting main())
18     n,  // upper bound of range in which to find primes
19     prime[MAX_N+1],  // in the end, prime[i] = 1 if i prime, else 0
20     nextbase;  // next sieve multiplier to be used
21
22 int work[MAX_THREADS];  // to measure how much work each thread does,
23                         // in terms of number of sieve multipliers checked
24
25 // lock index for the shared variable nextbase
26 pthread_mutex_t nextbaselock = PTHREAD_MUTEX_INITIALIZER;
27
28 // ID structs for the threads
29 pthread_t id[MAX_THREADS];
30
31 // "crosses out" all multiples of k, from k*k on
32 void crossout(int k)
33 {  int i;
34
35    for (i = k; i*k <= n; i++)  {
36       prime[i*k] = 0;
37    }
38 }
39
40 // worker thread routine
41 void *worker(int tn)  // tn is the thread number (0,1,...)
42 {  int lim,base;
43
44    // no need to check multipliers bigger than sqrt(n)
45    lim = sqrt(n);
```

```
46
47      do  {
48         // get next sieve multiplier, avoiding duplication across threads
49         pthread_mutex_lock(&nextbaselock);
50         base = nextbase += 2;
51         pthread_mutex_unlock(&nextbaselock);
52         if (base <= lim)  {
53            work[tn]++;  // log work done by this thread
54            // don't bother with crossing out if base is known to be
55            // composite
56            if (prime[base])
57               crossout(base);
58         }
59         else return;
60      } while (1);
61   }
62
63   main(int argc, char **argv)
64   {  int nprimes,  // number of primes found
65          totwork,  // number of base values checked
66          i;
67      void *p;
68
69      n = atoi(argv[1]);
70      nthreads = atoi(argv[2]);
71      for (i = 2; i <= n; i++)
72         prime[i] = 1;
73      crossout(2);
74      nextbase = 1;
75      // get threads started
76      for (i = 0; i < nthreads; i++)  {
77         pthread_create(&id[i],NULL,(void *) worker,(void *) i);
78      }
79
80      // wait for all done
81      totwork = 0;
82      for (i = 0; i < nthreads; i++)  {
83         pthread_join(id[i],&p);
84         printf("%d values of base done\n",work[i]);
85         totwork += work[i];
86      }
87      printf("%d total values of base done\n",totwork);
88
89      // report results
90      nprimes = 0;
91      for (i = 2; i <= n; i++)
92         if (prime[i]) nprimes++;
93      printf("the number of primes found was %d\n",nprimes);
94
95   }
```

There are two command-line arguments in this program, the upper bound `n` of the range to be checked for primes, and `nthreads`, the number of worker threads we wish to create.

`main()` creates the worker threads, each of which is an invocation of the function `worker()`. The workers share three data items: the upper bound variable, `n`; the variable specifying the next number whose multiples are to be eliminated from the range $2..n$, `nextbase`; and the array `prime[]` that records, for each number in the range $2..n$, whether or not it has been eliminated. Each invocation repeatedly fetches a yet-to-be-processed elimination multiplicand, `base`, and then eliminates all multiples of `base` from the range $2..n$. After spawning the workers, `main()` uses `pthread_join()` to wait for all these threads to finish their work before resuming itself, at which point it counts the primes that are left and issues its report. The report includes not only the prime count, but also information on how much work each worker thread did. This assessment is useful for *load balancing* and performance optimization purposes on a multiprocessor system.

Each instance of `worker()` fetches the next value of `base` by executing the following code (lines 49–51):

```
pthread_mutex_lock(&nextbaselock);
base = nextbase += 2;
pthread_mutex_unlock(&nextbaselock);
```

Here, the global variable `nextbase` is updated and used to initialize the value of the `worker()` instance's local variable `base`; the worker then crosses out multiples of `base` in the array `prime[]`. (Note that we started by eliminating all multiples of 2 at the beginning of `main()`, and thereafter only need to consider odd values for `base`.)

Once the worker knows the value of `base` to use, it can safely cross out the multiples of `base` from the shared array `prime[]`, because no other worker will use that value of `base`. However, we have to place *guard statements* around the update operation to the shared variable `nextbase` that `base` depends upon (line 26). Recall that any worker thread can be preempted, at an unpredictable time, by another worker thread, which will be at an unpredictable place in the code for `worker()`. In particular, it might just happen that the current worker is interrupted in the midst of the statement

```
base = nextbase += 2;
```

and the next timeslice is given to another thread that is also executing the same statement. In this case, there are two workers trying to modify the shared variable `nextbase` at once, which can lead to insidious and hard to reproduce bugs.

Bracketing the code that manipulates the shared variable—known as a *critical section*—with the guard statements prevents this from happening. The calls to `pthread_mutex_lock()` and `pthread_mutex_unlock()` ensure that there is at most only ever one thread executing the enclosed program fragment. They tell the OS to allow a thread to enter the critical section only if there is no other thread currently executing it, and to not preempt that thread until it completes the entire section. (The *lock variable* `nextbaselock` is used internally by the thread system to ensure this "mutual exclusion.")

Unfortunately, it's all too easy to fail to recognize and/or properly protect critical sections in threaded code. Let's see how GDB can be used to debug this sort of error in a Pthreads program. Suppose we had forgotten to include the unlock statement,

```
pthread_mutex_unlock(&nextbaselock);
```

This of course causes the program to hang once the critical section is first entered by a worker thread, as the other worker threads will wait forever for the lock to be relinquished. But let's pretend we don't already know this. How do we track down the culprit using GDB?

We compile the progam, making sure to include the flags `-lpthread -lm` in order to link in the Pthreads and math libraries (the latter is needed for our call to `sqrt()`). Then we run the code in GDB, with `n = 100` and `nthreads = 2`:

```
(gdb) r 100 2
Starting program: /debug/primes 100 2
[New Thread 16384 (LWP 28653)]
[New Thread 32769 (LWP 28676)]
[New Thread 16386 (LWP 28677)]
[New Thread 32771 (LWP 28678)]
```

Each time a new thread is created, GDB announces it, as seen here. We'll look into which thread is which in a moment.

The program hangs, and we interrupt it by pressing CTRL-C. Our GDB session now looks like this:

```
(gdb) r 100 2
Starting program: /debug/primes 100 2
[New Thread 16384 (LWP 28653)]
[New Thread 32769 (LWP 28676)]
[New Thread 16386 (LWP 28677)]
```

```
[New Thread 32771 (LWP 28678)]

Program received signal SIGINT, Interrupt.
[Switching to Thread 32771 (LWP 28678)]
0x4005ba35 in __pthread_sigsuspend () from /lib/i686/libpthread.so.0
```

At a point like this it's crucial to know what each threads is doing, which we can determine via GDB's `info threads` command:

```
(gdb) info threads
* 4 Thread 32771 (LWP 28678)  0x4005ba35 in __pthread_sigsuspend ()
   from /lib/i686/libpthread.so.0
  3 Thread 16386 (LWP 28677)  0x4005ba35 in __pthread_sigsuspend ()
   from /lib/i686/libpthread.so.0
  2 Thread 32769 (LWP 28676)  0x420db1a7 in poll () from
/lib/i686/libc.so.6
  1 Thread 16384 (LWP 28653)  0x4005ba35 in __pthread_sigsuspend ()
   from /lib/i686/libpthread.so.0
```

The asterisk means that we are currently in thread 4. Let's see what's going on with that thread:

```
(gdb) bt
#0  0x4005ba35 in __pthread_sigsuspend () from /lib/i686/libpthread.so.0
#1  0x4005adb8 in __pthread_wait_for_restart_signal ()
   from /lib/i686/libpthread.so.0
#2  0x4005d190 in __pthread_alt_lock () from /lib/i686/libpthread.so.0
#3  0x40059d77 in pthread_mutex_lock () from /lib/i686/libpthread.so.0
#4  0x0804855f in worker (tn=1) at Primes.c:49
#5  0x40059881 in pthread_start_thread () from /lib/i686/libpthread.so.0
#6  0x40059985 in pthread_start_thread_event () from
/lib/i686/libpthread.so.0
```

(This works under the LinuxThreads implementation of Pthreads but may not be possible on some other platforms.)

Aha—we see in frames 3 and 4 that this thread is on line 49 of the source code and is trying to acquire the lock and enter the critical section:

```
pthread_mutex_lock(&nextbaselock);
```

Note also from frame 0 above that the thread is apparently suspended pending the lock's being relinquished by another thread. It will not get any timeslices until this happens *and* the thread manager arranges for it to acquire the lock.

What are the other threads doing? We can inspect any thread's stack by switching to that thread and then issuing the `bt` command:

```
(gdb) thread 3
[Switching to thread 3 (Thread 16386 (LWP 28677))]#0  0x4005ba35 in
__pthread_sigsuspend () from /lib/i686/libpthread.so.0
(gdb) bt
#0  0x4005ba35 in __pthread_sigsuspend () from /lib/i686/libpthread.so.0
#1  0x4005adb8 in __pthread_wait_for_restart_signal ()
   from /lib/i686/libpthread.so.0
#2  0x4005d190 in __pthread_alt_lock () from /lib/i686/libpthread.so.0
#3  0x40059d77 in pthread_mutex_lock () from /lib/i686/libpthread.so.0
#4  0x0804855f in worker (tn=0) at Primes.c:49
#5  0x40059881 in pthread_start_thread () from /lib/i686/libpthread.so.0
#6  0x40059985 in pthread_start_thread_event () from
/lib/i686/libpthread.so.0
```

Recall that we created two worker threads. We saw above that thread 4 was one of them (frame 4 from its `bt` output), and now we see from frame 4 of the output here that thread 3 is the other one. We also see that thread 3 is trying to acquire the lock as well (frame 3).

There shouldn't be any other worker threads, but one of the fundamental principles of debugging is that nothing is taken on faith, and everything must be checked. We do this now by inspecting the status of the remaining threads. We find that the other two threads are nonworker threads, as follows:

```
(gdb) thread 2
[Switching to thread 2 (Thread 32769 (LWP 28676))]#0  0x420db1a7 in poll
()
   from /lib/i686/libc.so.6
(gdb) bt
#0  0x420db1a7 in poll () from /lib/i686/libc.so.6
#1  0x400589de in __pthread_manager () from /lib/i686/libpthread.so.0
#2  0x4005962b in __pthread_manager_event () from
/lib/i686/libpthread.so.0
```

So thread 2 is the threads manager. This is internal to the Pthreads package. It is certainly not a worker thread, partially confirming our expectation that there are only two worker threads. Checking thread 1,

```
(gdb) thread 1
[Switching to thread 1 (Thread 16384 (LWP 28653))]#0  0x4005ba35 in
__pthread_sigsuspend () from /lib/i686/libpthread.so.0
(gdb) bt
#0  0x4005ba35 in __pthread_sigsuspend () from /lib/i686/libpthread.so.0
#1  0x4005adb8 in __pthread_wait_for_restart_signal ()
   from /lib/i686/libpthread.so.0
#2  0x40058551 in pthread_join () from /lib/i686/libpthread.so.0
#3  0x080486aa in main (argc=3, argv=0xbfffe7b4) at Primes.c:83
#4  0x420158f7 in __libc_start_main () from /lib/i686/libc.so.6
```

we find it executes `main()`, and thus confirm that there are only two worker threads.

However, both of the workers are stalled, each waiting for the lock to be relinquished. No wonder the program is hanging! This is enough to pinpoint the location and nature of the bug, and we quickly realize that we forgot the call to the unlocking function.

6.2.3 A Variation

What if we hadn't realized the necessity of guarding the update of the shared variable `nextbase` in the first place? What would have happened in our previous example if we'd left out *both* the unlock and the lock operations?

A naive look at this question might lead to the guess that there would have been no harm in terms of correct operation of the program (i.e. getting an accurate count of the number of primes), albeit possibly with a slowdown due to duplicate work (i.e. using the same value of `base` more than once). It would seem that some threads may duplicate the work of others, namely when two workers happen to grab the same value of `nextbase` to initialize their local copies of `base`. Some composite numbers might then end up being crossed out twice, but the results (i.e., the count of the number of primes) would still be correct.

But let's take a closer look. The statement

```
base = nextbase += 2;
```

compiles to at least two machine language instructions. For instance, using the GCC compiler on a Pentium machine running Linux, the C statement above translates to the following assembly language instructions (obtained by running GCC with the `-S` option, and then viewing the resulting **.s** file):

```
addl $2, nextbase
movl nextbase, %eax
movl %eax, -8(%ebp)
```

This code increments `nextbase` by 2, then copies the value of `nextbase` to the register EAX, and finally, copies the value of EAX to the place in the worker's stack where its local variable `base` is stored.

Suppose we have only two worker threads and the value of `nextbase` is, say, 9, and the currently running `worker()` invocation's timeslice ends just after it executes the machine instruction

```
addl $2, nextbase
```

setting the shared global variable `nextbase` to 11. Suppose the next timeslice goes to another invocation of `worker()`, which happens to be executing those same instructions. The second worker now increments `nextbase` to 13, uses this to set its local variable `base`, and starts to eliminate all multiples of 13. Eventually, the first invocation of `worker()` will get another timeslice, and it will then pick up where it left off, executing the machine instructions

```
movl nextbase, %eax
movl %eax, -8(%ebp)
```

Of course, the value of `nextbase` is now 13. The first worker thus sets the value of its local variable `base` to 13 and proceeds to eliminate multiples of this value, not the value 11 that it fetched during its last timeslice. Neither worker does anything with the multiples of 11. We end up not only duplicating work unnecessarily, but also skipping necessary work!

How might we discover such an error using GDB? Presumably the "symptom" that surfaced was that the number of primes reported was too large. Thus we might suspect that values of `base` are somehow sometimes skipped. To check this hypothesis, we could place a breakpoint right after the line

```
base = nextbase += 2;
```

By repeatedly issuing the GDB `continue` ("c") command and displaying the value of `base`,

```
(gdb) disp base
```

we might eventually verify that a value of `base` is indeed skipped.

The key word here is *might*. Recall our earlier discussion that threaded programs run in a somewhat random manner. In our context here, it may be the case that on some runs of the program the bug surfaces, i.e. too many primes are reported, but on other runs we may get correct answers!

There is, unfortunately, no good solution to this problem. Debugging threaded code often requires extra patience and creativity.

6.2.4 GDB Threads Command Summary

Here is a summary of the usage of GDB's thread-related commands:

- `info threads` (gives information on all current threads)
- `thread 3` (change to thread 3)
- `break 88 thread 3` (stop execution when thread 3 reaches source line 88)
- `break 88 thread 3 if x==y` (stop execution when thread 3 reaches source line 88 and the variables x and y are equal)

6.2.5 Threads Commands in DDD

In DDD, select Status | Threads, and a window will pop up, displaying all threads, in the manner of GDB's `info threads`, as seen in Figure 6.1. You can click on a thread to switch the debugger's focus to it.

You will probably want to keep this pop-up window around, rather than using it once and then closing it. This way you don't have to keep reopening it every time you want to see which thread is currently running or want to switch to a different thread.

There appears to be no way to make a breakpoint thread-specific in DDD, e.g. with our GDB command `break 88 thread 3` above. Instead, we issue such a command to GDB via the DDD Console.

6.2.6 Threads Commands in Eclipse

Note first that the default makefile created by Eclipse will not include the `-lpthread` command-line argument for GCC (nor will it include the arguments for any other special libraries you need). You can alter the makefile directly if you wish, but it is easier to tell Eclipse to do it for you. While in the C/C++ perspective, right-click on your project name, and select Properties; point the triangle next to C/C++ Build downward; select Settings | Tool Settings; point the triangle next to GCC C Linker downward and select Libraries | Add (the latter is the green + icon); and fill in your library flags minus the `-l`, e.g. filling in `m` for `-lm`. Then build your project.

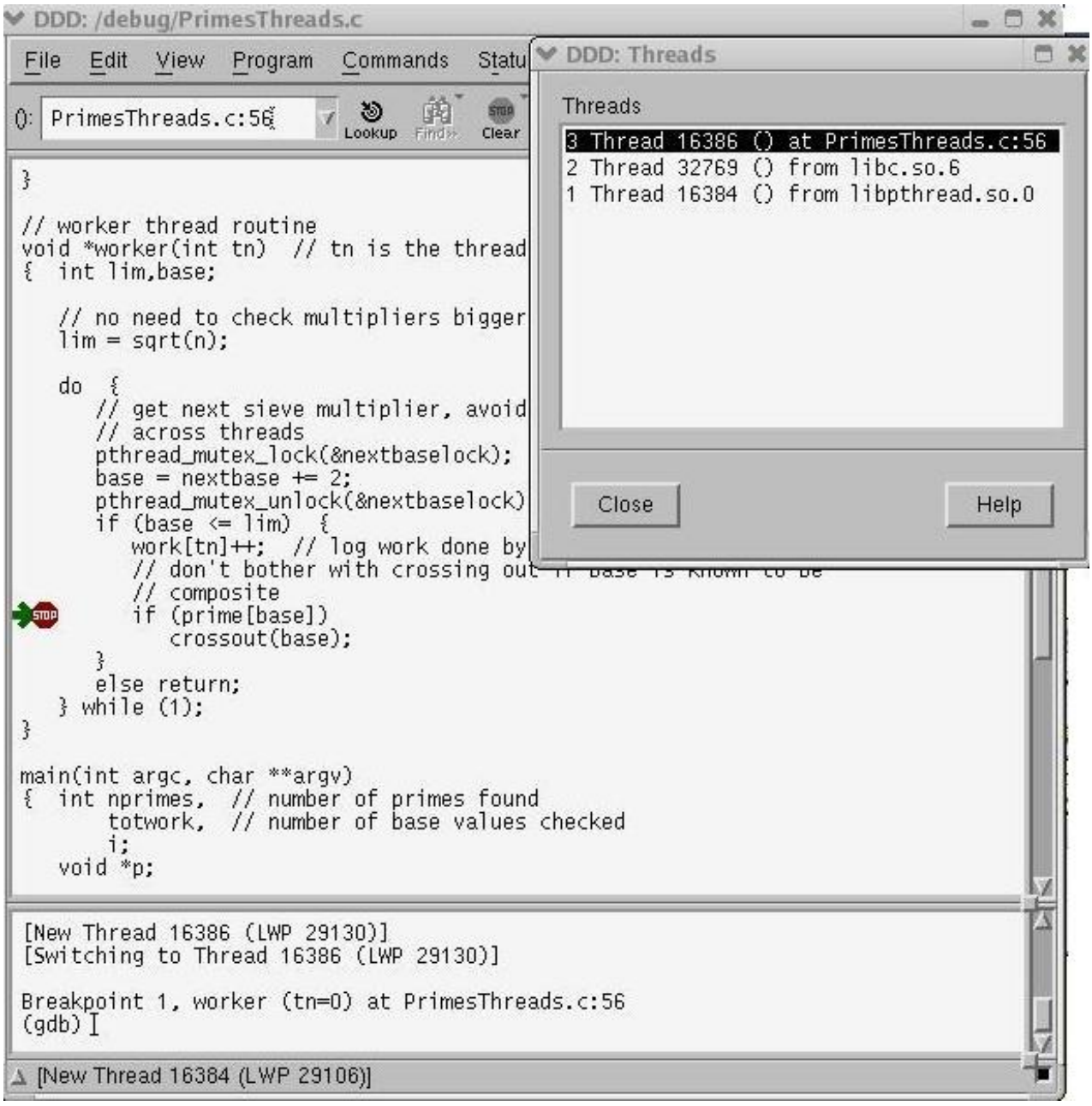

Figure 6.1: Threads window

Recall from Chapter **??** that Eclipse constantly displays your thread list, as opposed to having to request it in the case of DDD. Moreover, we do not need to ask for a backtrace kind of operation as in DDD; the call stack is shown in the thread list. This is depicted in Figure 6.2. As above, we ran the program for a while, then interrupted it by clicking the Suspend icon to the right of Resume. The thread list is in the Debug view, which normally is in the upper-left portion of the screen, but appears here in expanded form due to our having clicked Maximize in the Debug tab. (We can click Restore to return to the standard layout.)

We see that thread 3 had been running at the time of the interruption; it had received a SIGINT signal, which is the interruption ("CTRL-C") signal. We see also that the associated system call had been invoked by `pthread_join()`, which in turn had been called by `main()`. From what we've seen about this program earlier, we see that this indeed is the main thread.

To view the information for another thread, we merely click the triangle next to the thread, to point it downward. To change to another thread, we click on its entry in the list.

We may wish to set a breakpoint that applies only to a specific thread. To do so, we must first wait until the thread is created. Then when execution pauses via a previous setting of a breakpoint, or an interruption as above we right-click on the breakpoint symbol in the same manner as we use to make a breakpoint conditional, but this time select Filtering. A pop-up window like the one in Figure 6.3 will appear. We see that currently this breakpoint applies to all three threads. If we wish it to apply only to thread 2, for instance, we would uncheck the boxes next to the entries for the other two threads.

6.3 Debugging Parallel Applications

There are two main types of parallel programming architectures—*shared memory* and *message passing*.

The term "shared memory" means exactly that: multiple CPUs all have access to some common physical memory. Code running on one CPU communicates with code running on the others by reading from and writing to this shared memory, much as threads in a multithreaded application communicate with one another through a shared address space. (Indeed, threaded programming has become the standard way to write application code for shared memory systems.)

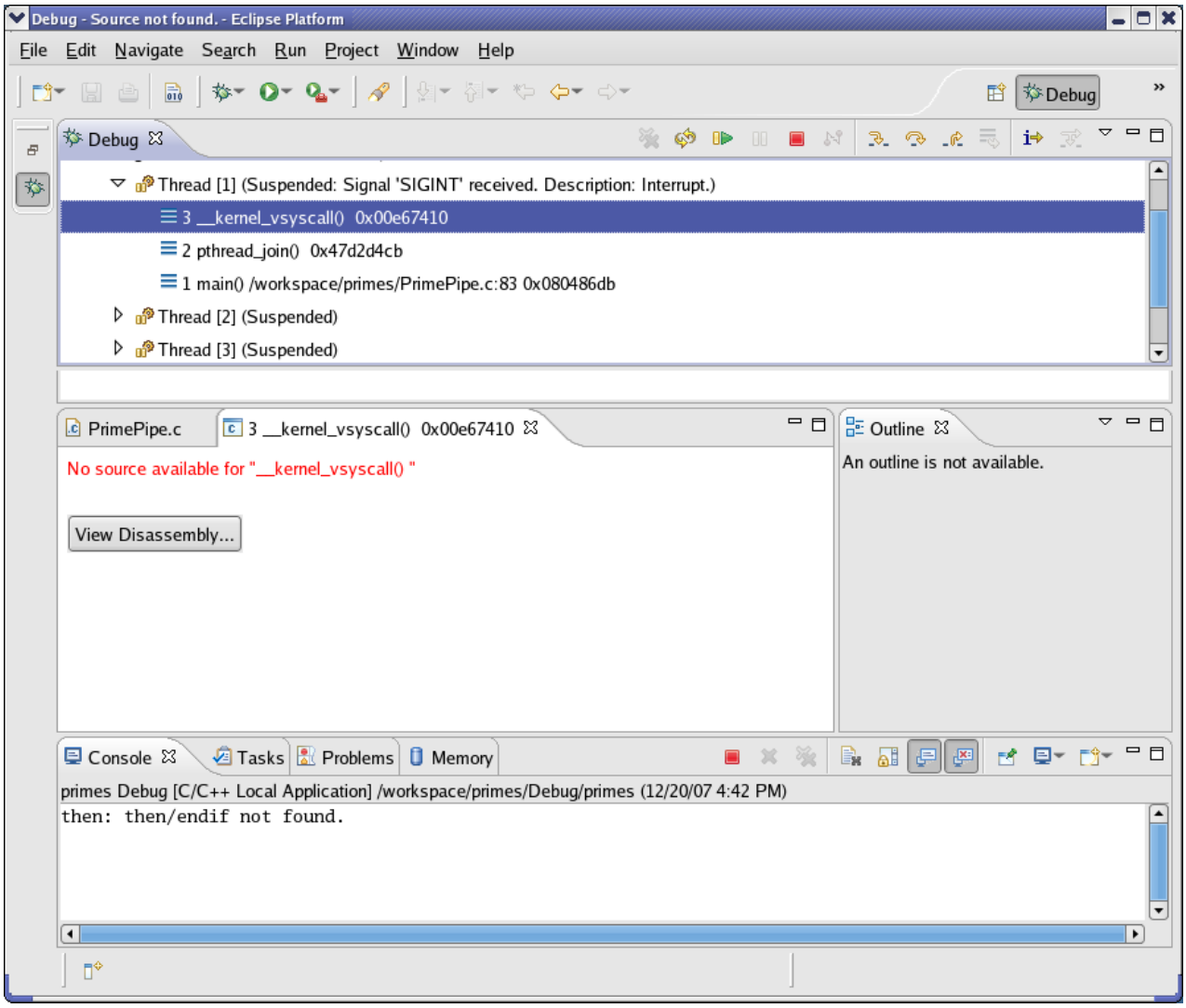

Figure 6.2: Threads Display in Eclipse

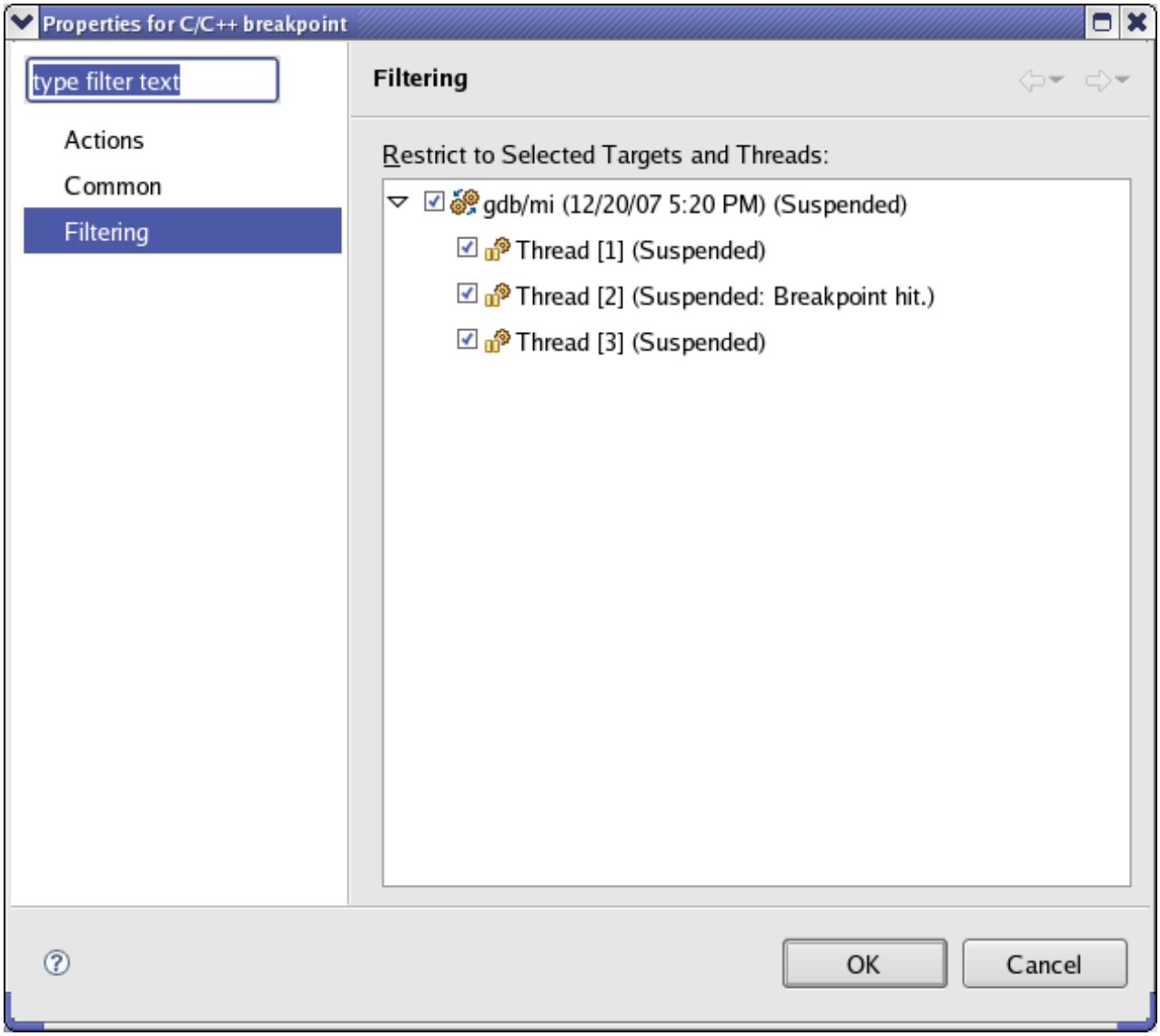

Figure 6.3: Setting a Thread-Specific Breakpoint in Eclipse

By contrast, in a message passing environment, code running on each CPU can only access that CPU's local memory, and communicates with the others by sending strings of bytes called *messages* over a communication medium. Typically this some kind of network, running either a general-purpose protocol like TCP/IP or a specialized software infrastructure that is tailored to message-passing applications.

6.3.1 Message-Passing Systems

We will discuss message passing first, using the popular Message Passing Interface (MPI) package as our example. We use the MPICH implementation here, but the same principles apply to LAM and other MPI implementations.

Let us again consider a prime-number finding program:

```
1  #include <mpi.h>
2
3  // MPI sample program; not intended to be efficient; finds and reports
4  // the number of primes less than or equal to n
5
6  // Uses a pipeline approach:  node 0 looks at all the odd numbers (i.e.,
7  // we assume multiples of 2 are already filtered out) and filters out
8  // those that are multiples of 3, passing the rest to node 1; node 1
9  // filters out the multiples of 5, passing the rest to node 2; node 2
10 // filters out the rest of the composites and then reports the number
11 // of primes
12
13 // the command-line arguments are n and debugwait
14
15 #define PIPE_MSG 0  // type of message containing a number to
16                     // be checked
17 #define END_MSG 1  // type of message indicating no more data will
18                    // be coming
19
20 int nnodes,  // number of nodes in computation
21     n,  // find all primes from 2 to n
22     me;  // my node number
23
24 init(int argc,char **argv)
25 {  int debugwait;  // if 1, then loop around until the
26                    // debugger has been attached
27
28    MPI_Init(&argc,&argv);
29    n = atoi(argv[1]);
30    debugwait = atoi(argv[2]);
31
32    MPI_Comm_size(MPI_COMM_WORLD,&nnodes);
33    MPI_Comm_rank(MPI_COMM_WORLD,&me);
34
35    while (debugwait) ;
36 }
```

```
37
38 void node0()
39 {  int i,dummy,
40        tocheck; // current number to check for passing on to next node
41    for (i = 1; i <= n/2; i++)  {
42       tocheck = 2 * i + 1;
43       if (tocheck > n) break;
44       if (tocheck % 3 > 0)
45          MPI_Send(&tocheck,1,MPI_INT,1,PIPE_MSG,MPI_COMM_WORLD);
46    }
47    MPI_Send(&dummy,1,MPI_INT,1,END_MSG,MPI_COMM_WORLD);
48 }
49
50 void node1()
51 {  int tocheck,  // current number to check from node 0
52        dummy;
53    MPI_Status status;  // see below
54
55    while (1)  {
56       MPI_Recv(&tocheck,1,MPI_INT,0,MPI_ANY_TAG,
57          MPI_COMM_WORLD,&status);
58       if (status.MPI_TAG == END_MSG) break;
59       if (tocheck % 5 > 0)
60          MPI_Send(&tocheck,1,MPI_INT,2,PIPE_MSG,MPI_COMM_WORLD);
61    }
62    // now send our end-of-data signal, which is conveyed in the
63    // message type, not the message itself
64    MPI_Send(&dummy,1,MPI_INT,2,END_MSG,MPI_COMM_WORLD);
65 }
66
67 void node2()
68 {  int tocheck,  // current number to check from node 1
69        primecount,i,iscomposite;
70    MPI_Status status;
71
72    primecount = 3;  // must account for the primes 2, 3 and 5, which
73                     // won't be detected below
74    while (1)  {
75       MPI_Recv(&tocheck,1,MPI_INT,1,MPI_ANY_TAG,
76          MPI_COMM_WORLD,&status);
77       if (status.MPI_TAG == END_MSG) break;
78       iscomposite = 0;
79       for (i = 7; i*i <= tocheck; i += 2)
80          if (tocheck % i == 0)  {
81             iscomposite = 1;
82             break;
83          }
84       if (!iscomposite) primecount++;
85    }
86    printf("number of primes = %d\n",primecount);
87 }
88
89 main(int argc,char **argv)
90 {  init(argc,argv);
91    switch (me)  {
92       case 0:  node0();
```

```
93              break;
94          case 1:  node1();
95              break;
96          case 2:  node2();
97      };
98      MPI_Finalize();
99  }
```

As explained in the comments at the beginning of the program, here our Sieve of Eratosthenes runs on three nodes of a parallel system and works in a pipelined manner. The first node starts with odd numbers and removes all multiples of 3, passing on the remaining values; the second node takes the output of the first and removes all multiples of 5; and the third node takes the output of the second and removes the rest of the nonprimes and reports the number of primes that are left.

Here the pipelining is achieved by having each node pass one number at a time to the next. (Much greater efficiency could be attained by passing groups of numbers in each MPI message, thus reducing communications overhead.) When sending a number on to the next node, a node sends a message of type `PIPE_MSG`. When a node has no more numbers to send, it indicates this by sending a message of type `END_MSG`.

As our debugging example here, suppose we forget to include the latter notification at the first node, that is, we forget line 46 in the code for `node0()`:

```
MPI_Send(&dummy,1,MPI_INT,1,END_MSG,MPI_COMM_WORLD);
```

The program will hang at the "downstream" nodes. Let's see how we can track down this bug. (Keep in mind that some line numbers in our GDB session below will differ by 1 from those in the above listing.)

We run an MPICH application program by invoking a script named `mpirun` on one node of the system. The script then starts the application program at each node, via SSH. Here we did this on a network of three machines, which we'll call Node 0, Node 1, and Node 2, with `n` equal to 100. Our bug causes the program to hang at the latter two nodes. The program also hangs at the first node, because no instance of an MPI program will exit until all have executed the `MPI_FINALIZE()` function.

We would like to use GDB, but because we used `mpirun` to invoke the application at each of the three nodes, rather than running them directly on the nodes, we cannot run GDB directly. However, GDB allows one to dynamically *attach* the debugger to an already-running process, using the process number. So let's run `ps` on Node 1, to determine the number of the process that is executing our application there:

```
$ ps ax
...
 2755 ?        S      0:00 tcsh -c /home/matloff/primepipe node 1 3
 2776 ?        S      0:00 /home/matloff/primepipe node1 32812   4
 2777 ?        S      0:00 /home/matloff/primepipe node1 32812   4
```

Our MPI program is running as process 2776, so we attach GDB to the program at Node 1:

```
$ gdb primepipe 2776
...
0xffffe002 in ?? ()
```

This is not very informative! So, let's check to see where we are:

```
(gdb) bt
#0  0xffffe002 in ?? ()
#1  0x08074a76 in recv_message ()
#2  0x080748ad in p4_recv ()
#3  0x0807ab46 in MPID_CH_Check_incoming ()
#4  0x08076ae9 in MPID_RecvComplete ()
#5  0x0806765f in MPID_RecvDatatype ()
#6  0x0804a29f in PMPI_Recv ()
#7  0x08049ce8 in node1 () at PrimePipe.c:56
#8  0x08049e19 in main (argc=8, argv=0xbffffb24) at PrimePipe.c:94
#9  0x420156a4 in __libc_start_main () from /lib/tls/libc.so.6
```

We see from frame 7 that the program is hanging at line 56, waiting to receive from Node 0.

Next, it would be useful to know how much work has been done by the function running at Node 1, `node1()`. Has it just started, or is it almost done? We can gauge the progress by determining the last value processed for the variable **tocheck**:

```
(gdb) frame 7
#7  0x08049ce8 in node1 () at PrimePipe.c:56
56          MPI_Recv(&tocheck,1,MPI_INT,0,MPI_ANY_TAG,
(gdb) p tocheck
$1 = 97
```

(Note that we needed to move to the stack frame for `node1()` first, using GDB's `frame` command.)

This indicates that Node 1 is at the end of execution, as 97 should be the last number that Node 0 passes to it for prime checking. So, currently we would be expecting

a message from node 0 of type END_MSG. The fact that the program is hanging would suggest to us that Node 0 might not have sent such a message, which would in turn lead us to check whether it had. In this manner, we hopefully would zero in quickly on the bug, which was the accidental omission of line 46.

By the way, keep in mind that when GDB is invoked with the command

```
$ gdb primepipe 2776
```

as we did above, GDB's command-line processing first checks for a core file named `2776`. In the unlikely event that such a file exists, GDB will load it instead of attaching to the intended process. Alternatively, GDB also has an `attach` command.

In this example, the bug caused our program to hang. The approach to debugging a parallel program like this one is somewhat different when the symptom is incorrect output. Suppose, for example, that in line 71 we incorrectly initialized `primecount` to 2 instead of 3. If we try to follow the same debugging procedure, the programs running on each node would finish execution and exit too quickly for us to attach GDB. (True, we could use a very large value of `n`, but it is usually better to debug with simple cases at first.) We need some device that can be used to make the programs wait and give us a chance to attach GDB. This is the purpose of line 34 in the `init()` function.

As can be seen in the source code, the value of `debugwait` is taken from the command line supplied by the user, with 1 meaning wait and 0 meaning no wait. If we specify 1 for the value of `debugwait`, then when each invocation of the program reaches line 34, it remains there. This gives us time to attach GDB. We can then break out of the infinite loop and proceed to debug. Here is what we do at Node 0:

```
node1:~$ gdb primepipe 3124
...
0x08049c53 in init (argc=3, argv=0xbfffe2f4) at PrimePipe.c:34
34          while (debugwait) ;
(gdb) set debugwait = 0
(gdb) c
Continuing.
```

Ordinarily we dread infinite loops, but here we deliberately set one up in order to facilitate debugging. We do the same thing at Node 1 and Node 2, and at the latter we also take the opportunity to set a breakpoint at line 77 before continuing:

```
[matloff@node3 ~]$ gdb primepipe 2944
34          while (debugwait) ;
(gdb) b 77
Breakpoint 1 at 0x8049d7d: file PrimePipe.c, line 77.
(gdb) set debugwait = 0
(gdb) c
Continuing.

Breakpoint 1, node2 () at PrimePipe.c:77
77              if (status.MPI_TAG == END_MSG) break;
(gdb) p tocheck
$1 = 7
(gdb) n
78              iscomposite = 0;
(gdb) n
79              for (i = 7; i*i <= tocheck; i += 2)
(gdb) n
84              if (!iscomposite) primecount++;
(gdb) n
75              MPI_Recv(&tocheck,1,MPI_INT,1,MPI_ANY_TAG,
(gdb) p primecount
$2 = 3
```

At this point, we notice that `primecount` should be 4, not 3—the primes through 7 are 2, 3, 5 and 7—and thus have found the location of the bug.

6.3.2 Shared-Memory Systems

Now, what about the shared-memory type of parallel programming? Here we have separate cases for true shared-memory machines and software distributed shared-memory settings.

True Shared Memory

As mentioned earlier, in a true shared-memory environment, application programs are often developed using threads. Our material in Section 6.2 on debugging with GDB/DDD then applies.

OpenMP has become a popular programming environment on such machines. OpenMP supplies the programmer with high-level parallel programming constructs, which in turn make use of threads. The programmer still has thread-level access if needed, but for the most part the threaded implementation of the OpenMP directives are largely transparent to the programmer.

We present an extended example in Section 6.4 of debugging an OpenMP application.

Software Distributed Shared-Memory Systems

Prices of machines with dual-core CPUs are now within reach of ordinary consumers, but large-scale shared-memory systems with many processors still cost hundreds of thousands of dollars. A popular, inexpensive alternative is a *network of workstations* (NOW). NOW architectures use an underlying library that gives the illusion of shared memory. The library, which is largely transparent to the application programmer, engages in network transactions that maintain consistency of copies of shared variables across the different nodes.

This approach is called *software distributed shared memory* (SDSM). The most widely used SDSM library is Treadmarks, developed and maintained by Rice University. Another excellent package is JIAJIA, available from the Chinese Academy of Sciences (`http://www-users.cs.umn.edu/~tianhe/paper/dist.htm`).

SDSM applications exhibit certain kinds of behavior that may baffle the unwary programmer. These are highly dependent on the particular system, so a general treatment cannot be given here, but we will briefly discuss a couple of issues common to many of them.

Many SDSMs are *page-based*, meaning that they rely on the underlying virtual memory hardware at the nodes. The actions are complex, but we can give a quick overview. Consider a variable `X` that is to be shared among the NOW nodes. The programmer indicates this intention by making a certain call to the SDSM library, which in turn makes a certain Unix system call requesting the OS to replace its own seg fault handler with a function in the SDSM library, for page faults involving the page containing `X`. The SDSM sets things up in such a way that only NOW nodes with valid copies of `X` have the corresponding memory pages marked as resident. When `X` is accessed at some other node, a page fault results, and the underlying SDSM software fetches the correct value from a node that has it.

Again, it's not essential to know the precise workings of the SDSM system; rather, the important thing is simply to understand that there *is* an underlying VM-based mechanism that's being used to maintain consistency of local copies of shared data across the NOW nodes. If you don't, you will be mystified when you try to debug SDSM application code. The debugger will *seem* to mysteriously stop for nonexistent seg faults, because the SDSM infrastructure deliberately generates seg

faults, and when an SDSM application program is run under a debugging tool, the tool senses them. Once you realize this, there is no problem at all—in GDB, you'd merely issue a `continue` command to resume execution when one of these odd pauses occurs.

You may be tempted to order GDB not to stop or issue warning messages whenever any seg faults occur, using the GDB command

```
handle SIGSEGV nostop noprint
```

You should use this approach with caution, as it may result in your missing any genuine seg faults caused by bugs in the application program.

Yet another, related difficulty with debugging applications that run on page-based SDSMs arises as follows. If a node on the network changes the value of a shared variable, then any other node that needs the value of that variable must obtain the updated value through a network transaction. Once again, the details of how this happens depends on the SDSM system, but this means that if we are single-stepping through the code executing on one node, we may find that GDB mysteriously hangs, because the node is now waiting for an update to its local copy of a variable that was recently modified by another node. If we also happen to be running a separate GDB session to step through the code on that other node as well, the update will not occur on the first node until the debugging session on the second node progresses far enough. In other words, if the programmer is not alert and careful during the debugging of an SDSM application, he can cause his own deadlock situation through the debugging process itself.

The SDSM situation is similar to that of the message-passing case in one sense—the need to have a variable like `debugwait` in our MPI example above, which allowed us to have the program pause at all nodes, giving us a chance to attach GDB at each node and step through the program from the beginning.

6.4 Extended Example

This section presents an example of debugging a shared-memory application developed using OpenMP. The necessary knowledge of OpenMP will be explained below. All that is needed is a basic understanding of threads.

6.4.1 OpenMP Overview

OpenMP is essentially a higher-level parallel programming interface to thread-management operations. The number of threads is set via the environment variable `OMP_NUM_THREADS`. In the C shell, for instance, we type

```
% setenv OMP_NUM_THREADS 4
```

at the shell prompt to arrange to have four threads.

Application code consists of C interspersed with OpenMP directives. Each directive applies to the block that follows it, delimited by left and right braces. The most basic directive is

```
#pragma omp parallel
```

This sets up `OMP_NUM_THREADS` threads, each of which concurrently executes the block of code following the pragma. There will typically be other directives embedded within this block.

Another very common OpenMP directive is

```
#pragma omp barrier
```

This specifies a "meeting point" for all the threads. When any thread reaches this point, it will block until all the other threads have arrived there.

Often we wish to have just one thread execute a certain block, with the other threads skipping it. This is accomplished by writing

```
#pragma omp single
```

There is an implied barrier immediately following such a block.

There are many other OpenMP directives, but the only other one we use in our example here is

```
#pragma omp critical
```

As the name implies, this creates a critical section, in which only one thread is allowed at any given time.

6.4.2 OpenMP Example Program

We implement the famous Dijkstra algorithm for determining minimum distances between pairs of vertices in a weighted graph. We are given distances between adjacent vertices (if two vertices are not adjacent, the distance between them is set to infinity). The goal is to find the minimum distances between vertex 0 and all other vertices.

Following is our source file, **dijkstra.c**. It generates random edge lengths among a specified number of vertices and then finds the minimum distances from vertex 0 to each of the other vertices.

```
1  // dijkstra.c
2
3  // OpenMP example program:  Dijkstra shortest-path finder in a
4  // bidirectional graph; finds the shortest path from vertex 0 to all
5  // others
6
7  // usage:  dijkstra nv print
8
9  // where nv is the size of the graph, and print is 1 if graph and min
10 // distances are to be printed out, 0 otherwise
11
12 #include <omp.h>  // required
13 #include <values.h>
14
15 // including stdlib.h and stdio.h seems to cause a conflict with the
16 // Omni compiler, so declare directly
17 extern void *malloc();
18 extern int printf(char *,...);
19
20 // global variables, shared by all threads
21 int nv,  // number of vertices
22     *notdone, // vertices not checked yet
23     nth,  // number of threads
24     chunk,  // number of vertices handled by each thread
25     md,  // current min over all threads
26     mv;  // vertex which achieves that min
27
28 int *ohd,  // 1-hop distances between vertices; "ohd[i][j]" is
29            // ohd[i*nv+j]
30     *mind;  // min distances found so far
31
32 void init(int ac, char **av)
33 {  int i,j,tmp;
34    nv = atoi(av[1]);
35    ohd = malloc(nv*nv*sizeof(int));
36    mind = malloc(nv*sizeof(int));
37    notdone = malloc(nv*sizeof(int));
38    // random graph
39    for (i = 0; i < nv; i++)
40       for (j = i; j < nv; j++)  {
```

```
41          if (j == i) ohd[i*nv+i] = 0;
42          else  {
43             ohd[nv*i+j] = rand() % 20;
44             ohd[nv*j+i] = ohd[nv*i+j];
45          }
46       }
47    for (i = 1; i < nv; i++)  {
48       notdone[i] = 1;
49       mind[i] = ohd[i];
50    }
51 }
52
53 // finds closest to 0 among notdone, among s through e; returns min
54 // distance in *d, closest vertex in *v
55 void findmymin(int s, int e, int *d, int *v)
56 {  int i;
57    *d = MAXINT;
58    for (i = s; i <= e; i++)
59       if (notdone[i] && mind[i] < *d)  {
60          *d = mind[i];
61          *v = i;
62       }
63 }
64
65 // for each i in {s,...,e}, ask whether a shorter path to i exists, through
66 // mv
67 void updatemind(int s, int e)
68 {  int i;
69    for (i = s; i <= e; i++)
70       if (notdone[i])
71          if (mind[mv] + ohd[mv*nv+i] < mind[i])
72             mind[i] = mind[mv] + ohd[mv*nv+i];
73 }
74
75 void dowork()
76 {
77    #pragma omp parallel
78    {  int startv,endv,  // start, end vertices for this thread
79           step,  // whole procedure goes nv steps
80           mymv,  // vertex which attains that value
81           me = omp_get_thread_num(),
82           mymd;  // min value found by this thread
83       #pragma omp single
84       {  nth = omp_get_num_threads();  chunk = nv/nth;
85          printf("there are %d threads\n",nth);  }
86       startv = me * chunk;
87       endv = startv + chunk - 1;
88       // the algorithm goes through nv iterations
89       for (step = 0; step < nv; step++)  {
90          // find closest vertex to 0 among notdone; each thread finds
91          // closest in its group, then we find overall closest
92          #pragma omp single
93          {  md = MAXINT;
94             mv = 0;
95          }
96          findmymin(startv,endv,&mymd,&mymv);
```

```
97            // update overall min if mine is smaller
98            #pragma omp critical
99            {  if (mymd < md)
100                {  md = mymd;  }
101           }
102           #pragma omp barrier
103           // mark new vertex as done
104           #pragma omp single
105           {  notdone[mv] = 0;  }
106           // now update my section of ohd
107           updatemind(startv,endv);
108       }
109    }
110 }
111
112 int main(int argc, char **argv)
113 {  int i,j,print;
114    init(argc,argv);
115    // start parallel
116    dowork();
117    // back to single thread
118    print = atoi(argv[2]);
119    if (print)  {
120       printf("graph weights:\n");
121       for (i = 0; i < nv; i++)  {
122          for (j = 0; j < nv; j++)
123             printf("%u  ",ohd[nv*i+j]);
124          printf("\n");
125       }
126       printf("minimum distances:\n");
127       for (i = 1; i < nv; i++)
128          printf("%u\n",mind[i]);
129    }
130 }
```

Let's review how the algorithm works. We start with all vertices except vertex 0, which in this case are vertices 1–5, in a "not done" set. In each iteration of the algorithm, we do the following:

- We find the "not done" vertex `v` that is closest to vertex 0, along paths known so far. This checking is shared by all the threads, with each thread checking an equal number of vertices. The function that does this work is `findmymin()`.
- We move `v` to the "done" set.
- Then for all remaining vertices `i` in the "not done" set, we check whether going first from 0 to `v` along the best known path so far, and then from `v` to `i` in one hop, is shorter than our current shortest distance from 0 to `i`. If so, we update that distance accordingly. This function that performs these actions is `updatemind()`.

The iteration continues until the "not done" set is empty.

Since OpenMP directives require preprocessing, there is always the potential problem that we will lose our original line numbers and variable and function names. To see how to address this, we will discuss two different compilers. First we'll look at the Omni compiler (`http://www.hpcc.jp/Omni/`), and then at GCC (version 4.2 or later is required).

We compile our code under Omni as follows:

```
$ omcc -g -o dij dijkstra.c
```

After compiling the program and running it with four threads, we find that it fails to work properly:

```
$ dij 6 1
there are 4 threads
graph weights:
0  3  6  17  15  13
3  0  15  6  12  9
6  15  0  1  2  7
17  6  1  0  10  19
15  12  2  10  0  3
13  9  7  19  3  0
minimum distances:
3
6
17
15
13
```

Analyzing the graph by hand shows that the correct minimum distances should be 3, 6, 7, 8 and 11.

Next, we run the program in GDB. Here it is very important to understand the consequences of the fact that OpenMP works via directives. Although line numbers, function names, and so on are mostly retained by the two compilers we discuss here, there are some discrepancies between them. Look what happens when we try to set a breakpoint in our executable, `dij`, at the outset of our GDB session:

```
(gdb) tb main
Breakpoint 1 at 0x80492af
(gdb) r 6 1
Starting program: /debug/dij 6 1
[Thread debugging using libthread_db enabled]
[New Thread -1208490304 (LWP 11580)]
```

```
[Switching to Thread -1208490304 (LWP 11580)]
0x080492af in main ()
(gdb) l
1       /tmp/omni_C_11486.c: No such file or directory.
        in /tmp/omni_C_11486.c
```

We discover that the breakpoint is not in our source file. Instead, it is in Omni's OpenMP infrastructure code. In other words, `main()` here is Omni's `main()`, not our own. The Omni compiler mangled the name of our `main()` to `_ompc_main()`.

So, to set a breakpoint at our `main()`, we type

```
(gdb) tb _ompc_main
Breakpoint 2 at 0x80491b3: file dijkstra.c, line 114.
```

and check it by continuing:

```
(gdb) c
Continuing.
[New Thread -1208493152 (LWP 11614)]
[New Thread -1218983008 (LWP 11615)]
[New Thread -1229472864 (LWP 11616)]
_ompc_main (argc=3, argv=0xbfab6314) at dijkstra.c:114
114         init(argc,argv);
```

OK, there's our familiar `init()` line. Of course, we also could have issued the command

```
(gdb) b dijkstra.c:114
```

Note the creation of the three new threads, making four in all.

However we choose to set our breakpoints, we must go to a bit more work here than normal, so it's extra important to stay within a single GDB session between runs of our program, even when we change our source code and recompile, so that we retain the breakpoints, conditions, and so on. That way we only have to go to the trouble of setting these things up once.

Now, how do we track down the bug(s)? It is natural to approach the debugging of this program by checking the results at the end of each iteration. The main results are in the "not done" set, i.e. in the array `notdone[]`, and in the current list of best known distances from 0 to the other vertices, that is, the array `mind[]`. For

example, after the first iteration, the "not done" set should consist of vertices 2, 3, 4, and 5, vertex 1 having been selected in that iteration.

Armed with this information, let's apply our Principle of Confirmation and check `notdone[]` and `mind[]` after each iteration of the `for` loop in `dowork()`.

We have to be careful as to exactly where we set our breakpoints. Although a natural spot for this seems to be line 108, at the very end of the algorithm's main loop, this may not be so good, as GDB will stop there for *each* thread. Instead, we opt for placing a breakpoint inside an OpenMP `single` block, so that we'll stop for only one thread.

So, instead we'll check the results after each iteration by stopping at the *beginning* of the loop, starting with the second iteration:

```
(gdb) b 92 if step >= 1
Breakpoint 3 at 0x80490e3: file dijkstra.c, line 92.
(gdb) c
Continuing.
there are 4 threads

Breakpoint 3, __ompc_func_0 () at dijkstra.c:93
93              {  md = MAXINT;
```

Let's confirm that the first iteration did choose the correct vertex, vertex 1, to be moved out of the "not done" set:

```
(gdb) p mv
$1 = 0
```

Our hypothesis does not confirm after all. Inspection of our code shows that on line 99 we forgot to set `mv`. We fix it to read

```
              {  md = mymd; mv = mymv;  }
```

So, we recompile and run the program again. As noted earlier in this section (and elsewhere in this book), it is very helpful to not exit GDB when we rerun the program. We could run the program in another terminal window, but just for variety let's take a different approach here. We'll temporarily disable our breakpoints, by issuing the `dis` command, then run the recompiled program from within GDB, and then re-enable the breakpoints using `ena`:

```
(gdb) dis
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
'/debug/dij' has changed; re-reading symbols.
Starting program: /debug/dij 6 1
[Thread debugging using libthread_db enabled]
[New Thread -1209026880 (LWP 11712)]
[New Thread -1209029728 (LWP 11740)]
[New Thread -1219519584 (LWP 11741)]
[New Thread -1230009440 (LWP 11742)]
there are 4 threads
graph weights:
0  3  6  17  15  13
3  0  15  6  12  9
6  15  0  1  2  7
17  6  1  0  10  19
15  12  2  10  0  3
13  9  7  19  3  0
minimum distances:
3
6
17
15
13

Program exited with code 06.
(gdb) ena
```

We're still getting wrong answers. Let's check things at that breakpoint again:

```
(gdb) r
Starting program: /debug/dij 6 1
[Thread debugging using libthread_db enabled]
[New Thread -1209014592 (LWP 11744)]
[New Thread -1209017440 (LWP 11772)]
[New Thread -1219507296 (LWP 11773)]
[New Thread -1229997152 (LWP 11774)]
there are 4 threads
[Switching to Thread -1209014592 (LWP 11744)]

Breakpoint 3, __ompc_func_0 () at dijkstra.c:93
93              {  md = MAXINT;
(gdb) p mv
$2 = 1
```

At least `mv` now has the right value. Let's check `mind[]`:

```
(gdb) p *mind@6
$3 = {0, 3, 6, 17, 15, 13}
```

Note that because we constructed the `mind[]` array dynamically via `malloc()`, we could not use GDB's `print` command in its usual form. Instead, we used GDB's artificial array feature.

At any rate, `mind[]` is still incorrect. For instance, `mind[3]` should be 3+6 = 9, yet it is 17. Let's check the code that updates `mind[]`:

```
(gdb) b 107 if me == 1
Breakpoint 4 at 0x8049176: file dijkstra.c, line 107.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /debug/dij 6 1
[Thread debugging using libthread_db enabled]
[New Thread -1209039168 (LWP 11779)]
[New Thread -1209042016 (LWP 11807)]
[New Thread -1219531872 (LWP 11808)]
[New Thread -1230021728 (LWP 11809)]
there are 4 threads
[Switching to Thread -1230021728 (LWP 11809)]

Breakpoint 4, __ompc_func_0 () at dijkstra.c:107
107             updatemind(startv,endv);
```

First we'll confirm that `startv` and `endv` have sensible values:

```
(gdb) p startv
$4 = 1
(gdb) p endv
$5 = 1
```

Our chunk size is only 1? Let's see:

```
(gdb) p chunk
$6 = 1
```

After checking the computation for `chunk`, we realize that we need the number of threads to evenly divide `nv`. The latter has the value 6, which is not divisible by our thread count, 4. We make a note to ourselves to insert some error-catching code later, and we decide to reduce our thread count to 3 for now.

Once again, we do not want to exit GDB to do this. GDB inherits the environment variables when it is first invoked, but the values of those variables can also be changed or set within GDB, and that is what we do here:

```
(gdb) set environment OMP_NUM_THREADS = 3
```

Now let's run again:

```
(gdb) dis
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /debug/dij 6 1
[Thread debugging using libthread_db enabled]
[New Thread -1208707392 (LWP 11819)]
[New Thread -1208710240 (LWP 11847)]
[New Thread -1219200096 (LWP 11848)]
there are 3 threads
graph weights:
0  3  6  17  15  13
3  0  15  6  12  9
6  15  0  1  2  7
17  6  1  0  10  19
15  12  2  10  0  3
13  9  7  19  3  0
minimum distances:
3
6
7
15
12

Program exited with code 06.
(gdb) ena
```

Aiyah, still the same wrong answers! Let's continue to check the updating process for `mind[]`:

```
(gdb) r
Starting program: /debug/dij 6 1
[Thread debugging using libthread_db enabled]
[New Thread -1208113472 (LWP 11851)]
[New Thread -1208116320 (LWP 11879)]
[New Thread -1218606176 (LWP 11880)]
there are 3 threads
[Switching to Thread -1218606176 (LWP 11880)]

Breakpoint 4, __ompc_func_0 () at dijkstra.c:107
107             updatemind(startv,endv);
(gdb) p startv
$7 = 2
(gdb) p endv
$8 = 3
```

All right, those are the correct values for `startv` and `endv` in the case of `me = 1`. So, let's enter the function:

```
(gdb) s
[Switching to Thread -1208113472 (LWP 11851)]

Breakpoint 3, __ompc_func_0 () at dijkstra.c:93
93              {  md = MAXINT;
(gdb) c
Continuing.
[Switching to Thread -1218606176 (LWP 11880)]
updatemind (s=2, e=3) at dijkstra.c:69
69         for (i = s; i <= e; i++)
```

Note that due to context switches among the threads, we did not enter `updatemind()` immediately. Now let's check the case `i = 3`:

```
(gdb) tb 71 if i == 3
Breakpoint 5 at 0x8048fb2: file dijkstra.c, line 71.
(gdb) c
Continuing.
updatemind (s=2, e=3) at dijkstra.c:71
71              if (mind[mv] + ohd[mv*nv+i] < mind[i])
```

As usual, we apply the Principle of Confirmation:

```
(gdb) p mv
$9 = 0
```

Well, that's a big problem. Recall that in the first iteration `mv` turns out to be 1; why is it 0 here?

After a while we realize that those context switches should have been a big hint to us. Take a look at the GDB output above again. The thread whose system ID is 11851 was already on line 93—in other words, it was already in the next iteration of our algorithm's main loop. In fact, when we hit `c` to continue, it even executed line 94, which is

```
            mv = 0;
```

This thread overwrote `mv`'s previous value of 1, so that our thread that updates `mind[3]` is now relying on the wrong value of `mv`. The solution is to add another barrier:

```
        updatemind(startv,endv);
        #pragma omp barrier
```

After this fix, the program runs correctly.

The foregoing was based on the Omni compiler. As mentioned, beginning with version 4.2, GCC handles OpenMP code as well. All you have to do is add the `-fopenmp` flag to the GCC command line.

Unlike Omni, GCC generates code in such a way that GDB's focus is in your own source file from the beginning. Thus, issuing a command

```
(gdb) b main
```

at the very outset of a GDB session really will cause a breakpoint to be set in one's own `main()`, unlike what we saw for the Omni compiler.

However, at this writing, a major shortcoming of GCC is that the symbols for local variables that are inside an OpenMP `parallel` block (called *private* variables in OpenMP terminology) will not be visible within GDB. For example, the command

```
(gdb) p mv
```

that we issued for the Omni-generated code above will work for GCC-generated code, but our command above

```
(gdb) p startv
```

will fail on GCC-generated code.

There are ways to work around this, of course. For instance, if we wish to know the value of `startv`, we can query the value of `s` within `updatemind()`. Hopefully this issue will be resolved in next version of GCC.