8

Testing the results

Finally *make* has run through to the end and has not reported errors. Your source tree now contains all the objects and executables. You're done!

After a brief moment of euphoria, you sit down at the keyboard and start the program:

```
$ xterm
Segmentation fault - core dumped
```

Well, maybe you're not quite done after all. Occasionally the program does not work as advertised. What you do now depends on how much programming experience you have. If you are a complete beginner, you could be in trouble—about the only thing you can do (apart from asking somebody else) is to go back and check that you really did configure the package correctly.

On the other hand, if you have even a slight understanding of programming, you should try to analyze the cause of the error—it's easier than you think. Hold on, and try not to look down.

There are thousands of possible reasons for the problems you encounter when you try to run a buggy executable, and lots of good books explain debugging techniques. In this chapter, we will touch only on aspects of debugging that relate to porting. First we'll attack a typical, if somewhat involved, real-life bug, and solve it, discussing the pros and cons on the way. Then we'll look at alternatives to traditional debuggers: kernel and network tracing.

Before you even start your program, of course, you should check if any test programs are available. Some packages include their own tests, and separate test suites are available for others. For other packages there may be test suites that were not designed for the package, but that can be used with it. If there are any tests, you should obviously run them. You might also consider writing some tests and including them as a target `test` in the *Makefile*.

What makes ported programs fail?

Ported programs don't normally fail for the same reasons as programs under development. A program under development still has bugs that prevent it from running correctly on any platform, while a ported program has already run reasonably well on some other platform. If it doesn't run on your platform, the reasons are usually:

- A latent bug has found more fertile feeding ground. For example, a program may read from a null pointer. This frequently doesn't get noticed if the data at address 0 doesn't cause the program to do anything unusual. On the other hand, if the new platform does not have any memory mapped at address 0, it will cause a segmentation violation or a bus error.
- Differences in the implementation of library functions or kernel functionality cause the program to behave differently in the new environment. For example, the function `setpgrp` has completely different semantics under System V and under BSD. See Chapter 12, *Kernel dependencies*, page 171, for more details.
- The configuration scripts have never been adequately tested for your platform. As a result, the program contains bugs that were not in the original versions.

A strategy for testing

When you write your own program with its own bugs, it helps to understand exactly what the program is trying to do: if you sit back and think about it, you can usually shorten the debugging process. When debugging software that you have just ported, the situation is different: you *don't* understand the package, and learning its internals could take months. You need to find a way to track down the bug without getting bogged down with the specifics of how the package works.

You can overdo this approach, of course. It still helps to know what the program is trying to do. For example, when *xterm* dies, it's nice to know roughly how *xterm* works: it opens a window on an X server and emulates a terminal in this window. If you know something about the internals of X11, this will also be of use to you. But it's not time-effective to try to fight your way through the source code of *xterm*.

In the rest of this chapter, we'll use this bug (yes, it was a real live bug in X11R6) to look at various techniques that you can use to localize and finally pinpoint the problem. The principle we use is the old *GIGO* principle—garbage in, garbage out. We'll subdivide the program into pieces which we can conveniently observe, and check which of them does not produce the expected output. After we find the piece with the error, we subdivide it further and repeat the process until we find the bug. The emphasis in this method is on *convenient*: it doesn't necessarily have to make sense. As long as you can continue to divide your problem area into between two and five parts and localize the problem in one of the parts, it won't take long to find the bug.

So what's a convenient way to look at the problems? That depends on the tools you have at your disposal:

- If you have a symbolic debugger, you can divide your problem into the individual functions and examine what goes in and what goes out.
- If you have a system call trace program, such as *ktrace* or *truss*, you can monitor what the program says to the system and what the system replies.

- If you have a communications line trace program, you can try to divide your program into pieces that communicate across this line, so you can see what they are saying to each other.

Of course, we have all these things. In the following sections we'll look at each of them in more detail.

Symbolic debuggers

If you don't have a symbolic debugger, get one. Now. Many people still claim to be able to get by without a debugger, and it's horrifying how many people don't even know how to use one. Of course you can debug just about anything without a symbolic debugger. Historians tell us that you can build pyramids without wheels—that's a comparable level of technology to testing without a debugger. The GNU debugger, *gdb*, is available on just about every platform you're likely to encounter, and though it's not perfect, it runs rings around techniques like putting *printf* statements in your programs.

In UNIX, a debugger is a process that takes control of the execution of another process. Most versions of UNIX allow only one way for the debugger to take control: it must start the process that it debugs. Some versions, notably SunOS 4, but not Solaris 2, also allow the debugger to *attach* to a running process.

Whichever debugger you use, there are a surprisingly small number of commands that you need. In the following discussion, we'll look at the command set of *gdb*, since it is widely used. The commands for other symbolic debuggers vary considerably, but they normally have similar purposes.

- A *stack trace* command answers the question, "Where am I, and how did I get here?", and is almost the most useful of all commands. It's certainly the first thing you should do when examining a core dump or after getting a signal while debugging the program. *gdb* implements this function with the `backtrace` command.
- *Displaying data* is the most obvious requirement: what is the current value of the variable `bar`? In *gdb*, you do this with the `print` command.
- *Displaying register contents* is really the same thing as displaying program data. In *gdb*, you display individual registers with the `print` command, or all registers with the `info registers` command.
- *Modifying data and register contents* is an obvious way of modifying program execution. In *gdb*, you do this with the `set` command.
- *breakpoints* stop execution of the process when the process attempts to execute an instruction at a certain address. *gdb* sets breakpoints with the `break` command.
- Many modern machines have hardware support for more sophisticated breakpoint mechanisms. For example, the i386 architecture can support four hardware breakpoints on instruction fetch (in other words, traditional breakpoints), memory read or memory write. These features are invaluable in systems that support them; unfortunately, UNIX usually

does not. *gdb* simulates this kind of breakpoint with a so-called *watchpoint*. When watchpoints are set, *gdb* simulates program execution by single-stepping through the program. When the condition (for example, writing to the global variable `foo`) is fulfilled, the debugger stops the program. This slows down the execution speed by several orders of magnitude, whereas a real hardware breakpoint has no impact on the execution speed.*

- *Jumping* (changing the address from which the next instruction will be read) is really a special case of modifying register contents, in this case the *program counter* (the register that contains the address of the next instruction). This register is also sometimes called the *instruction pointer*, which makes more sense. In *gdb*, use the `jump` command to do this. Use this instruction with care: if the compiler expects the stack to look different at the source and at the destination, this can easily cause incorrect execution.
- *Single stepping* in its original form is supported in hardware by many architectures: after executing a single instruction, the machine automatically generates a hardware interrupt that ultimately causes a `SIGTRAP` signal to the debugger. *gdb* performs this function with the `stepi` command.
- You won't want to execute individual machine instructions until you are in deep trouble. Instead, you will execute a *single line* instruction, which effectively single steps until you leave the current line of source code. To add to the confusion, this is also frequently called *single stepping*. This command comes in two flavours, depending on how it treats function calls. One form will execute the function and stop the program at the next line after the call. The other, more thorough form will stop execution at the first executable line of the function. It's important to notice the difference between these two functions: both are extremely useful, but for different things. *gdb* performs single line execution omitting calls with the `next` command, and includes calls with the `step` command.

There are two possible approaches when using a debugger. The easier one is to wait until something goes wrong, then find out where it happened. This is appropriate when the process gets a signal and does not overwrite the stack: the `backtrace` command will show you how it got there.

Sometimes this method doesn't work well: the process may end up in no-man's-land, and you see something like:

```
Program received signal SIGSEGV, Segmentation fault.
0x0 in ?? ()
(gdb) bt                        abbreviation for backtrace
#0  0x0 in ?? ()                  nowhere
(gdb)
```

Before dying, the process has mutilated itself beyond recognition. Clearly, the first approach won't work here. In this case, we can start by conceptually dividing the program into a number of parts: initially we take the function `main` and the set of functions which `main` calls. By single stepping over the function calls until something blows up, we can localize the function in which the problem occurs. Then we can restart the program and single step through this

* Some architectures slow the overall execution speed slightly in order to test the hardware registers. This effect is negligible.

function until we find what it calls before dying. This iterative approach sounds slow and tiring, but in fact it works surprisingly well.

Libraries and debugging information

Let's come back to our *xterm* program and use *gdb* to figure out what is going on. We could, of course, look at the core dump, but in this case we can repeat the problem at will, so we're better off looking at the live program. We enter:

```
$ gdb xterm
(political statement for the FSF omitted)
(gdb) r -display allegro:0      run the program
Starting program: /X/X11/X11R6/xc/programs/xterm/xterm -display allegro:0

Program received signal SIGBUS, Bus error.
0x3b0bc in _XtMemmove ()
(gdb) bt                                look back down the stack
#0  0x3b0bc in _XtMemmove ()    all these functions come from the X toolkit
#1  0x34dcd in XtScreenDatabase ()
#2  0x35107 in _XtPreparseCommandLine ()
#3  0x4e2ef in XtOpenDisplay ()
#4  0x4e4a1 in _XtAppInit ()
#5  0x35700 in XtOpenApplication ()
#6  0x357b5 in XtAppInitialize ()
#7  0x535 in main ()
(gdb)
```

The stack trace shows that the main program called `XtAppInitialize`, and the rest of the stack shows the program deep in the X Toolkit, one of the central X11 libraries. If this were a program that you had just written, you could expect it to be a bug in your program. In this case, where we have just built the complete X11 core system, there's also every possibility that it is a library bug. As usual, the library was compiled without debug information, and without that you hardly have a hope of finding it.

Apart from size constraints, there is no reason why you can't include debugging information in a library. The object files in libraries are just the same as any others—we discuss them in detail on page 369. If you want, you can build libraries with debugging information, or you can take individual library routines and compile them separately.

Unfortunately, the size constraints are significant: without debugging information, the file *libXt.a* is about 330 kB long and contains 53 object files. With debugging information, it might easily reach 20 MB, since all the myriad X11 global symbols would be included with each object file in the archive. It's not just a question of disk space: you also need virtual memory during the link phase to accommodate all these symbols. Most of these files don't interest us anyway: the first one that does is the one that contains `_XtMemmove`. So we find where it is and compile it alone with debugging information.

That's not as simple as it sounds: first we need to find the source file, and to do that we need to find the source directory. We could read the documentation, but to do that we need to know that the *Xt* functions are in fact the X toolkit. If we're using GNU *make*, or if our *Makefile*

documents directory changes, an alternative would be to go back to our *make* log and look for the text *Xt*. If we do this, we quickly find

```
make[4]: Leaving directory '/X/X11R6/xc/lib/Xext'
making Makefiles in lib/Xt...
        mv Makefile Makefile.bak
make[4]: Entering directory '/X/X11R6/xc/lib/Xt'
make[4]: Nothing to be done for 'Makefiles'.
make[4]: Leaving directory '/X/X11R6/xc/lib/Xt'
```

So the directory is */X/X11R6/xc/lib/Xt*. The next step is to find the file that contains XtMemmove. There is a possibility that it is called *XtMemmove.c*, but in this case there is no such file. We'll have to grep for it. Some versions of *grep* have an option to descend recursively into subdirectories, which can be very useful if you have one available. Another useful tool is *cscope*, which is supplied with System V.

```
$ grep XtMemmove *.c
Alloc.c:void _XtMemmove(dst, src, length)
Convert.c:      XtMemmove(&p->from.addr, from->addr, from->size);
... many more references to XtMemmove
```

So XtMemmove is in *Alloc.c*. By the same method, we look for the other functions mentioned in the stack trace and discover that we also need to recompile *Initialize.c* and *Display.c*.

In order to compile debugging information, we add the compiler option `-g`. At the same time, we remove `-O`. *gcc* doesn't require this, but it's usually easier to debug a non-optimized program. We have three choices of how to set the options:

- We can modify the *Makefile* (*make World*, the main *make* target for X11, rebuilds the *Makefile*s from the corresponding *Imakefile*s, so this is not overly dangerous).
- If we have a working version of *xterm*, we can use its facilities: first we start the compilation with *make*, but we don't need to wait for the compilation to complete: as soon as the compiler invocation appears on the screen, we abort the build with CTRL-C. Using the *xterm* copy function, we copy the compiler invocation to the command line and add the options we want:

  ```
  $ rm Alloc.o Initialize.o Display.o   remove the old objects
  $ make                                and start make normally
  rm -f Alloc.o
  gcc -DNO_ASM -fstrength-reduce -fpcc-struct-return -c     -I../..  \
  -DNO_AF_UNIX -DSYSV -DSYSV386     -DUSE_POLL  Alloc.c
  ^C                                  interrupt make with CTRL-C
  make: *** [Alloc.o] Interrupt
  copy the invocation lines above with the mouse, and paste below, then
  modify as shown in bold print
  $ gcc -DNO_ASM -fstrength-reduce -fpcc-struct-return -c     -I../..  \
  -DNO_AF_UNIX -DSYSV -DSYSV386     -DUSE_POLL  Alloc.c -g
  ```

  You can also use `make -n`, which just shows the commands that make would execute, rather than aborting the make, but you frequently find that `make -n` prints out a whole lot of stuff you don't expect. When you have made *Alloc.o*, you can repeat the process

for the other two object files.

- We could change CFLAGS from the *make* command line. Our first attempt doesn't work too well, though. If you compare the following line with the invocation above, you'll see that a whole lot of options are missing. They were all in CFLAGS; by redefining CFLAGS, we lose them all:

```
$ make CFLAGS=-g
rm -f Alloc.o
gcc -DNO_ASM -fstrength-reduce -fpcc-struct-return -c -g  Alloc.c
```

CFLAGS included all the compiler options starting from `-I/../..`, so we need to write:

```
$ make CFLAGS='-g -c -I../.. -DNO_AF_UNIX -DSYSV -DSYSV386 -DUSE_POLL'
```

When we have created all three new object files, we can let *make* complete the library for us. It will not try to remake these object files, since now they are newer than any of their dependencies:

```
$ make                                  run make to build a new library
rm -f libXt.a
ar clq libXt.a ActionHook.o Alloc.o ArgList.o Callback.o ClickTime.o Composite.o \
Constraint.o Convert.o Converters.o Core.o Create.o Destroy.o Display.o Error.o \
Event.o EventUtil.o Functions.o GCManager.o Geometry.o GetActKey.o GetResList.o \
GetValues.o HookObj.o Hooks.o Initialize.o Intrinsic.o Keyboard.o Manage.o \
NextEvent.o Object.o PassivGrab.o Pointer.o Popup.o PopupCB.o RectObj.o \
Resources.o Selection.o SetSens.o SetValues.o SetWMCW.o Shell.o StringDefs.o \
Threads.o TMaction.o TMgrab.o TMkey.o TMparse.o TMprint.o TMstate.o VarCreate.o \
VarGet.o Varargs.o Vendor.o
ranlib libXt.a
rm -f ../../usrlib/libXt.a
cd ../../usrlib; ln ../lib/Xt/libXt.a .
$
```

Now we have a copy of the X Toolkit in which these three files have been compiled with symbols. Next, we need to rebuild *xterm*. That's straightforward enough:

```
$ cd ../../programs/xterm/
$ pwd
/X/X11R6/xc/programs/xterm
$ make
rm -f xterm
gcc -DNO_ASM -fstrength-reduce -fpcc-struct-return -fwritable-strings -o xterm \
-L../../usrlib main.o input.o charproc.o cursor.o util.o tabs.o screen.o \
scrollbar.o button.o Tekproc.o misc.o VTPrsTbl.o TekPrsTbl.o data.o menu.o -lXaw \
-lXmu -lXt -lSM -lICE -lXext -lX11 -L/usr/X11R6/lib -lpt -ltermlib
```

Finally, we try again. Since the library is not in the current directory, we use the `dir` command to tell *gdb* where to find the sources. Now we get:

```
$ gdb xterm
(gdb) dir ../../lib/X11       set source paths
Source directories searched:
/X/X11/X11R6/xc/programs/xterm/../../lib/X11:$cdir:$cwd
(gdb) dir ../../lib/Xt
```

```
Source directories searched:
/X/X11/X11R6/xc/programs/xterm/../../lib/Xt/X/X11/X11R6/xc/programs/xterm/../..\
/lib/X11:$cdir:$cwd
(gdb) r                         and run the program
Starting program: /X/X11/X11R6/xc/programs/xterm/xterm

Program received signal SIGBUS, Bus error.
0x3ced6 in _XtMemmove (dst=0x342d8 "ÐE 03", src=0x41c800 "", length=383) \
at Alloc.c:101
101                 *dst++ = *src++;
(gdb)
```

This shows a typical byte for byte memory move. About the only thing that could cause a bus error on that statement would be an invalid address, but the parameters show that they appear to be valid.

There are at two possible gotchas here:

- The debugger may be lying. The parameters it shows are the parameters on the stack. If the code has been optimized, there is a very good chance that the source and destination addresses are stored in registers, and thus the value of `dst` on the stack is not up to date.
- The destination address may be in the text segment, in which case an attempt to write to it will cause some kind of error. Depending on the system it could be a segmentation violation or a bus error.

The most reliable way to find out what is really going on is to look at the machine instructions being executed. First we tell the debugger to look at current instruction and the following five instructions:

```
(gdb) x/6i $eip               list the next 6 instructions
0x3ced6 <_XtMemmove+74>:        movb   %al,(%edx)
0x3ced8 <_XtMemmove+76>:        incl   0xc(%ebp)
0x3cedb <_XtMemmove+79>:        incl   0x8(%ebp)
0x3cede <_XtMemmove+82>:        jmp    0x3cec2 <_XtMemmove+54>
0x3cee0 <_XtMemmove+84>:        leave
0x3cee1 <_XtMemmove+85>:        ret
```

The first instruction is a byte move, from register `al` to the address stored in register `edx`. Let's look at the address in `edx`:

```
(gdb) p/x $edx
$9 = 0x342d8
```

Well, this is our `dst` address alright—why can't it store there? It would be nice to be able to try to set values in memory and see if the debugger can do it:

```
(gdb) set *dst = 'X'b
(gdb) p *dst
$13 = 88 'X'
```

That looks writable enough. Unfortunately, you can't rely on the debugger to tell the truth. Debuggers must be able to write to the text segment. If the write had failed, you could have been sure that the address was not writable, but if the write succeeds, you can't be sure. What

we need to know are the exact segment limits. Some debuggers show you the segment limits, but current versions of *gdb* do not. An alternative is the *size* command:

```
$ size xterm
text    data    bss     dec     hex     filename
846204  56680   23844   926728  e2408   xterm
```

The text segment is 846204 decimal bytes long (0xce97c), and on this system (SCO UNIX) it starts at address 0, so the address is, indeed, in the text segment. But where did it come from? To find an answer to that question, we need to look at the calling function. In *gdb*, we do this with the `frame` command:

```
(gdb) f 1                look at the calling function (frame 1)
#1  0x35129 in _MergeOptionTables (src1=0x342d8, num_src1=24,
    src2=0x400ffe, num_src2=64, dst=0x7ffff9c0, num_dst=0x7ffff9bc)
    at Initialize.c:602
602         (void) memmove(table, src1, sizeof(XrmOptionDescRec) * num_src1 );
```

That's funny—last time it died, the function was called from `XtScreenDatabase`,* not from `_MergeOptionTables`. Why? At the moment it's difficult to say for sure, but it's possible that this difference happened because we removed optimization. In any case, we still have a problem, so we should fix this one first and then go back and look for the other one if solving this problem isn't enough.

In this case, the *frame* command doesn't help much, but it does tell us that the destination variable is called `table`, and implicitly that `memmove` has been defined as `_XtMemmove` in this source file. We could now look at the source file in an editor in a different X window, but it's easier to list the instructions around the current line with the `list` command:

```
(gdb) l
597         enum {Check, NotSorted, IsSorted} sort_order = Check;
598
599         *dst = table = (XrmOptionDescRec*)
600             XtMalloc( sizeof(XrmOptionDescRec) * (num_src1 + num_src2) );
601
602         (void) memmove(table, src1, sizeof(XrmOptionDescRec) * num_src1 );
603         if (num_src2 == 0) {
604             *num_dst = num_src1;
605             return;
606         }
```

So, the address is returned by the function `XtMalloc`—it seems to be allocating storage in the text segment. At this point, we could examine it more carefully, but let's first be sure that we're looking at the right problem. The address in `table` should be the same as the address in the parameter `dst` of `XtMemmove`. We're currently examining the environment of `_MergeOptionTables`, so we can look at it directly:

```
(gdb) p table
$29 = (XrmOptionDescRec *) 0x41c800
```

That looks just fine. Where did this strange `dst` address come from? Let's set a breakpoint

* See frame 1 in the stack trace on page 109.

on the call to memmove on line 602, and then restart the program:

Example 8−1:

```
(gdb) b 602
Breakpoint 8 at 0x35111: file Initialize.c, line 602.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /X/X11/X11R6/xc/programs/xterm/xterm

Breakpoint 8, _MergeOptionTables (src1=0x342d8, num_src1=24,
    src2=0x400ffe, num_src2=64, dst=0x7ffff9c0, num_dst=0x7ffff9bc)
    at Initialize.c:602
602         (void) memmove(table, src1, sizeof(XrmOptionDe
(gdb) p table                 look again, to be sure
$31 = (XrmOptionDescRec *) 0x41c800
(gdb) s                       single step into memmove
_XtMemmove (dst=0x342d8 "ÐE 03", src=0x41c800 "", length=384)
    at Alloc.c:94
94          if (src < dst) {
```

This is really strange! table has a valid address in the data segment, but the address we pass to _XtMemmove is in the text segment and seems unrelated. It's not clear what we should look at next:

- The source of the function calls memmove, but after preprocessing it ends up calling _XtMemmove. memmove might simply be defined as _XtMemmove, but it might also be defined with parameters, in which case some subtle type conversions might result in our problem.
- If you understand the assembler of the system, it might be instructive to look at the actual instructions that the compiler produces.

It's definitely quicker to look at the assembler instructions than to fight your way through the thick undergrowth in the X11 source tree:

```
(gdb) x/8i $eip              look at the next 8 instructions
0x35111 <_MergeOptionTables+63>:        movl   0xc(%ebp),%edx
0x35114 <_MergeOptionTables+66>:        movl   %edx,0xffffffd8(%ebp)
0x35117 <_MergeOptionTables+69>:        movl   0xffffffd8(%ebp),%edx
0x3511a <_MergeOptionTables+72>:        shll   $0x4,%edx
0x3511d <_MergeOptionTables+75>:        pushl  %edx
0x3511e <_MergeOptionTables+76>:        pushl  0xfffffffc(%ebp)
0x35121 <_MergeOptionTables+79>:        pushl  0x8(%ebp)
0x35124 <_MergeOptionTables+82>:        call   0x3ce8c <_XtMemmove>
```

This isn't easy stuff to handle, but it's worth understanding, so we'll pull it apart, instruction for instruction. It's easier to understand this discussion if you refer to the diagrams of stack structure in Chapter 21, *Object files and friends*, page 377.

- movl 0xc(%ebp),%edx takes the content of the stack word offset 12 in the current stack frame and places it in register edx. As we have seen, this is num_src1, the second

parameter passed to `_MergeOptionTables`.

- `movl %edx,0xfffffd8(%ebp)` stores the value of `edx` at offset -40 in the current stack frame. This is for temporary storage.
- `movl 0xfffffd8(%ebp),%edx` does *exactly* the opposite: it loads register `edx` from the location where it just stored it. These two instructions are completely redundant. They are also a sure sign that the function was compiled without optimization.
- `shll $0x4,%edx` shifts the contents of register `edx` left by 4 bits, multiplying it by 16. If we compare this to the source, it's evident that the value of `XrmOptionDescRec` is 16, and that the compiler has taken a short cut to evaluate the third parameter of the call.
- `pushl %edx` pushes the contents of `edx` onto the stack.
- `pushl 0xfffffffc(%ebp)` pushes the value of the word at offset -4 in the current stack frame onto the stack. This is the value of `table`, as we can confirm by looking at the instructions generated for the previous line.
- `pushl 0x8(%ebp)` pushes the value of the first parameter, `src1`, onto the stack.
- Finally, `call _XtMemmove` calls the function. Expressed in C, we now know that it calls

```
memmove (src1, table,  num_src1 << 4);
```

This is, of course, wrong: the parameter sequence of source and destination has been reversed. Let's look at `_XtMemmove` more carefully:

```
(gdb) l _XtMemmove
89      #ifdef _XNEEDBCOPYFUNC
90      void _XtMemmove(dst, src, length)
91          char *dst, *src;
92          int length;
93      {
94          if (src < dst) {
95              dst += length;
96              src += length;
97              while (length--)
98                  *--dst = *--src;
99          } else {
100             while (length--)
101                 *dst++ = *src++;
102         }
103     }
104     #endif
```

Clearly the function parameters are the same as those of `memmove`, but the calling sequence has reversed them. We've found the problem, but we haven't found what's causing it.

Aside: Debugging is not an exact science. We've found our problem, though we still don't know what's causing it. But looking back at Example 8-1, we see that the address for `src` on entering `_XtMemmove` was the same as the address of `table`. That tells us as much as analyzing the machine code did. This will happen again and again: after you find a problem, you

discover you did it the hard way.

The next thing we need to figure out is why the compiler reversed the sequence of the parameters. Can this be a compiler bug? Theoretically, yes, but it's very unlikely that such a primitive bug should go undiscovered up to now.

Remember that the compiler does not compile the sources you see: it compiles whatever the preprocessor hands to it. It makes a lot of sense to look at the preprocessor output. To do this, we go back to the library directory. Since we used pushd, this is easy—just enter pushd. In the library, we use the same trick as before in order to run the compiler with different options, only this time we use the options -E (stop after running the preprocessor), -dD (retain the text of the definitions in the preprocessor output), and -C (retain comments in the preprocessor output). In addition, we output to a file *junk.c*:

```
$ pushd
$ rm Initialize.o
$ make Initialize.o
rm -f Initialize.o
gcc -DNO_ASM -fstrength-reduce -fpcc-struct-return -c -g     -I../..  \
   -D_SVID -DNO_AF_UNIX -DSYSV -DSYSV386     -DUSE_POLL  Initialize.c
make: *** [Initialize.o] Interrupt              hit CTRL-C
...  copy the command into the command line, and extend:
$ gcc -DNO_ASM -fstrength-reduce -fpcc-struct-return -c -g -I../..  \
 -D_SVID -DNO_AF_UNIX -DSYSV -DSYSV386     -DUSE_POLL  Initialize.c \
  -E -dD -C >junk.c
$
```

As you might have guessed, we now look at the file *junk.c* with an editor. We're looking for memmove, of course. We find a definition in */usr/include/string.h*, then later on we find, in */X/X11/X11R6/xc/X11/Xfuncs.h*,

```
#define memmove(dst,src,len) bcopy((char *)(src),(char *)(dst),(int)(len))


#define memmove(dst,src,len) _XBCOPYFUNC((char *)(src),(char *)(dst),(int)(len))
#define _XNEEDBCOPYFUNC
```

For some reason, the configuration files have decided that memmove is not defined on this system, and have replaced it with bcopy (which is really not defined on this system). Then they replace it with the substitute function _XBCOPYFUNC, almost certainly a preprocessor definition. It also defines the preprocessor variable _XNEEDBCOPYFUNC to indicate that _XtMemmove should be compiled.

Unfortunately, we don't see what happens with _XNEEDBCOPYFUNC. The preprocessor discards all #ifdef lines. It does include #defines, however, so we can look for where _XBCOPYFUNC is defined—it's in *IntrinsicI.h*, as the last #line directive before the definition indicates.

```
#define _XBCOPYFUNC _XtMemmove
```

IntrinsicI.h also contains a number of definitions for XtMemmove, none of which are used in the current environment, but all of which have the parameter sequence (dst, src, count). bcopy has the parameter sequence (src, dst, count). Clearly, somebody has confused

something in this header file, and under certain rare circumstances the call is defined with the incorrect parameter sequence.

Somewhere in here is a lesson to be learnt: this is a real bug that occurred in X11R6, patch level 3, one of the most reliable and most portable software packages available, yet here we have a really primitive bug. The real problem lies in the configuration mechanism: automated configuration can save a lot of time in normal circumstances, but it can also cause lots of pain if it makes incorrect assumptions. In this case, the environment was unusual: the kernel platform was SCO UNIX, which has an old-fashioned library, but the library was GNU *libc*. This caused the assumptions of the configuration mechanism to break down.

Let's look more carefully at the part of *Xfuncs.h* where we found the definitions:

```
/* the new Xfuncs.h */

#if !defined(X_NOT_STDC_ENV) && (!defined(sun) || defined(SVR4))
/* the ANSI C way */
#ifndef _XFUNCS_H_INCLUDED_STRING_H
#include <string.h>
#endif
#undef bzero
#define bzero(b,len) memset(b,0,len)
#else /* else X_NOT_STDC_ENV or SunOS 4 */
#if defined(SYSV) || defined(luna) || defined(sun) || defined(__sxg__)
#include <memory.h>
#define memmove(dst,src,len) bcopy((char *)(src),(char *)(dst),(int)(len))
#if defined(SYSV) && defined(_XBCOPYFUNC)
#undef memmove
#define memmove(dst,src,len) _XBCOPYFUNC((char *)(src),(char *)(dst),(int)(len))
#define _XNEEDBCOPYFUNC
#endif
#else /* else vanilla BSD */
#define memmove(dst,src,len) bcopy((char *)(src),(char *)(dst),(int)(len))
#define memcpy(dst,src,len) bcopy((char *)(src),(char *)(dst),(int)(len))
#define memcmp(b1,b2,len) bcmp((char *)(b1),(char *)(b2),(int)(len))
#endif /* SYSV else */
#endif /* ! X_NOT_STDC_ENV else */
```

This is hairy (and incorrect) stuff. It makes its decisions based on the variables X_NOT_STDC_ENV, sun, SVR4, SYSV, luna, __sxg__ and _XBCOPYFUNC. These are the decisions:

- If the variable is *not* defined, it assumes ANSI C, unless this is a pre-SVR4 Sun machine.
- Otherwise it checks the variables SYSV (for System V.3), luna, sun or __sxg__. If any of these are set, it includes the file *memory.h* and defines *memmove* in terms of *bcopy*. If _XBCOPYFUNC is defined, it redefines memmove as _XBCOPYFUNC, reversing the parameters as it goes.
- If none of these conditions apply, it assumes a vanilla BSD machine and defines the functions memmove, memcpy and memcmp in terms of the BSD functions bcopy and bcmp.

There are two errors here:

- The only way that _XBCOPYFUNC is ever defined is as _XtMemmove, which does *not* have the same parameter sequence as bcopy—instead, it has the same parameter sequence as memmove. We can fix this part of the header by changing the definition line to

  ```
  #define memmove(dst,src,len) _XBCOPYFUNC((char *)(dst),(char *)(src),(int)(len))
  ```

  or even to

  ```
  #define memmove _XBCOPYFUNC
  ```

- There is no reason to assume that this system does not use ANSI C: it's using *gcc* and GNU *libc.a*, both of them very much standard compliant. We need to examine this point in more detail:

Going back to our *junk.c*, we search for X_NOT_STDC_ENV and find it defined at line 85 of */X/X11/X11R6/xc/X11/Xosdefs.h*:

```
#ifdef SYSV386
#ifdef SYSV
#define X_NOT_POSIX
#define X_NOT_STDC_ENV
#endif
#endif
```

In other words, this bug is likely to occur only with System V.3 implementations on Intel architecture. This is a fairly typical way to make decisions about the system, but it is wrong: X_NOT_STDC_ENV relates to a compiler, not an operating system, but both SYSV386 and SYSV define operating system characteristics. At first sight it would seem logical to modify the definitions like this:

```
#ifdef SYSV386
#ifdef SYSV
#ifndef __GNU_LIBRARY__
#define X_NOT_POSIX
#endif
#ifndef __GNUC__
#define X_NOT_STDC_ENV
#endif
#endif
#endif
```

This would only define the variables if the library is not GNU *libc* or the compiler is not *gcc*. This is still not correct: the relationship between __GNUC__ and X_NOT_STDC_ENV or __GNU_LIBRARY__ and X_NOT_POSIX is not related to System V or the Intel architecture. Instead, it makes more sense to backtrack at the end of the file:

```
#ifdef __GNU_LIBRARY__
#undef X_NOT_POSIX
#endif
#ifdef __GNUC__
#undef X_NOT_STDC_ENV
#endif
```

Whichever way we look at it, this is a mess. We're applying cosmetic patches to a

configuration mechanism which is based in incorrect assumptions. Until some better configuration mechanism comes along, unfortunately, we're stuck with this situation.

Limitations of debuggers

Debuggers are useful tools, but they have their limitations. Here are a couple which could cause you problems:

Can't breakpoint beyond fork

UNIX packages frequently start multiple processes to do the work on hand. Frequently enough, the program that you start does nothing more than to spawn a number of other processes and wait for them to stop. Unfortunately, the `ptrace` interface which debuggers use requires the process to be started by the debugger. Even in SunOS 4, where you can attach the debugger to a process that is already running, there is no way to monitor it from the start. Other systems don't offer even this facility. In some cases you can determine how the process was started and start it with the debugger in the same manner. This is not always possible—for example, many child processes communicate with their parent.

Unfortunately, SunOS trace doesn't support tracing through `fork`. *truss* does it better than *ktrace*. In extreme cases (like debugging a program of this nature on SunOS 4, where there is no support for trace through `fork`), you might find it an advantage to port to a different machine running an operating system such as Solaris 2 in order to be able to test with *truss*. Of course, Murphy's law says that the bug won't show up under Solaris 2.

Terminal logs out

The debugger usually shares a terminal with the program being tested. If the program changes the driver configuration, the debugger should change it back again when it gains control (for example, on hitting a breakpoint), and set it back to the way the program set it before continuing. In some cases, however, it can't: if the process has taken ownership of the terminal with a system call like *setsid* (see Chapter 12, *Kernel dependencies*, page 171), it will no longer have access to the terminal. Under these circumstances, most debuggers crawl into a corner and die. Then the shell in control of the terminal awakes and dies too. If you're running in an *xterm*, the *xterm* then stops; if you're running on a glass tty, you will be logged out.

The best way out of this dilemma is to start the child process on a different terminal, if your debugger and your hardware configuration support it. To do this with an *xterm* requires starting a program which just sleeps, so that the window stays open until you can start your test program:

```
$ xterm -e sleep 100000&
[1] 27013
$ ps aux|grep sleep
grog     27025  3.0  0.0   264  132 p6  S+    1:13PM    0:00.03 grep sleep
root     27013  0.0  0.0  1144  740 p6  I     1:12PM    0:00.37 xterm -e sleep 100000
grog     27014  0.0  0.0   100   36 p8  Is+   1:12PM    0:00.06 sleep 100000
$ gdb myprog
(gdb) r < /dev/ttyp8 > /dev/ttyp8
```

This example was done on a BSD machine. On a System V machine you will need to use *ps -ef* instead of *ps aux*. First, you start an *xterm* with *sleep* as controlling shell (so that it will stay there). With *ps* you grep for the controlling terminal of the *sleep* process (the third line in the example), and then you start your program with *stdin* and *stdout* redirected to this terminal.

Can't interrupt process

The *ptrace* interface uses the signal `SIGTRAP` to communicate with the process being debugged. What happens if you block this signal, or ignore it? Nothing—the debugger doesn't work any more. It's bad practice to block `SIGTRAP`, of course, but it can be done. More frequently, though, you'll encounter this problem when a process gets stuck in a signal processing loop and doesn't get round to processing the `SIGTRAP`—precisely one of the times when you would want to interrupt it. My favourite one is the program which had a `SIGSEGV` handler which went and retried the instruction. Unfortunately, the only signal to which a process in this state will still respond is `SIGKILL`, which doesn't help you much in finding out what's going on.

Tracing system calls

An alternative approach is to divide the program between system code and user code. Most systems have the ability to trace the parameters supplied to each system call and the results that they return. This is not nearly as good as using a debugger, but it works with all object files, even if they don't have symbols, and it can be very useful when you're trying to figure out why a program doesn't open a specific file.

Tracing is a very system-dependent function, and there are a number of different programs to perform the trace: *truss* runs on System V.4, *ktrace* runs on BSD NET/2 and 4.4BSD derived systems, and *trace* runs on SunOS 4. They vary significantly in their features. We'll look briefly at each. Other systems supply still other programs—for example, SGI's IRIX operating system supplies the program *par*, which offers similar functionality.

trace

trace is a relatively primitive tool supplied with SunOS 4 systems. It can either start a process or attach to an existing process, and it can print summary information or a detailed trace. In particular, it *cannot* trace the child of a `fork` call, which is a great disadvantage. Here's an example of *trace* output with a possibly recognizable program:

```
$ trace hello
open ("/usr/lib/ld.so", 0, 040250) = 3
read (3, ""..., 32) = 32
mmap (0, 40960, 0x5, 0x80000002, 3, 0) = 0xf77e0000
mmap (0xf77e8000, 8192, 0x7, 0x80000012, 3, 32768) = 0xf77e8000
open ("/dev/zero", 0, 07) = 4
getrlimit (3, 0xf7fff488) = 0
mmap (0xf7800000, 8192, 0x3, 0x80000012, 4, 0) = 0xf7800000
```

```
close (3) = 0
getuid () = 1004
getgid () = 1000
open ("/etc/ld.so.cache", 0, 05000100021) = 3
fstat (3, 0xf7fff328) = 0
mmap (0, 4096, 0x1, 0x80000001, 3, 0) = 0xf77c0000
close (3) = 0
open ("/opt/lib/gcc-lib/sparc-sun-sunos".., 0, 01010525) = 3
fstat (3, 0xf7fff328) = 0
getdents (3, 0xf7800108, 4096) = 212
getdents (3, 0xf7800108, 4096) = 0
close (3) = 0
open ("/opt/lib", 0, 056) = 3
getdents (3, 0xf7800108, 4096) = 264
getdents (3, 0xf7800108, 4096) = 0
close (3) = 0
open ("/usr/lib/libc.so.1.9", 0, 023170) = 3
read (3, "".., 32) = 32
mmap (0, 458764, 0x5, 0x80000002, 3, 0) = 0xf7730000
mmap (0xf779c000, 16384, 0x7, 0x80000012, 3, 442368) = 0xf779c000
close (3) = 0
open ("/usr/lib/libdl.so.1.0", 0, 023210) = 3
read (3, "".., 32) = 32
mmap (0, 16396, 0x5, 0x80000002, 3, 0) = 0xf7710000
mmap (0xf7712000, 8192, 0x7, 0x80000012, 3, 8192) = 0xf7712000
close (3) = 0
close (4) = 0
getpagesize () = 4096
brk (0x60d8) = 0
brk (0x70d8) = 0
ioctl (1, 0x40125401, 0xf7ffea8c) = 0
write (1, "Hello, World!0, 14) = Hello, World!
14
close (0) = 0
close (1) = 0
close (2) = 0
exit (1) = ?
```

What's all this output? All we did was a simple write, but we have performed a total of 43 system calls. This shows in some detail how much the viewpoint of the world differs when you're on the other side of the system library. This program, which was run on a SparcStation 2 with SunOS 4.1.3, first sets up the shared libraries (the sequences of `open`, `read`, `mmap`, and `close`), then initializes the `stdio` library (the calls to `getpagesize`, `brk`, `ioctl`, and `fstat`), and finally writes to *stdout* and exits. It also looks strange that it closed *stdin* before writing the output text: again, this is a matter of perspective. The `stdio` routines buffer the text, and it didn't actually get written until the process exited, just before closing *stdout*.

ktrace

ktrace is supplied with newer BSD systems. Unlike the other trace programs, it writes unformatted data to a log file (by default, *ktrace.out*), and you need to run another program, *kdump*, to display the log file. It has the following options:

- It can trace the descendents of the process it is tracing. This is particularly useful when the bug occurs in large complexes of processes, and you don't even know which process is causing the problem.
- It can attach to processes that are already running. Optionally, it can also attach to existing children of the processes to which it attaches.
- It can specify broad subsets of system calls to trace: system calls, namei translations (translation of file name to inode number), I/O, and signal processing.

Here's an example of *ktrace* running against the same program:

```
$ ktrace hello
Hello, World!
$ kdump
 20748 ktrace   RET   ktrace 0
 20748 ktrace   CALL  getpagesize
 20748 ktrace   RET   getpagesize 4096/0x1000
 20748 ktrace   CALL  break(0xadfc)
 20748 ktrace   RET   break 0
 20748 ktrace   CALL  break(0xaffc)
 20748 ktrace   RET   break 0
 20748 ktrace   CALL  break(0xbffc)
 20748 ktrace   RET   break 0
 20748 ktrace   CALL  execve(0xefbfd148,0xefbfd5a8,0xefbfd5b0)
 20748 ktrace   NAMI  "./hello"
 20748 hello    RET   execve 0
 20748 hello    CALL  fstat(0x1,0xefbfd2a4)
 20748 hello    RET   fstat 0
 20748 hello    CALL  getpagesize
 20748 hello    RET   getpagesize 4096/0x1000
 20748 hello    CALL  break(0x7de4)
 20748 hello    RET   break 0
 20748 hello    CALL  break(0x7ffc)
 20748 hello    RET   break 0
 20748 hello    CALL  break(0xaffc)
 20748 hello    RET   break 0
 20748 hello    CALL  ioctl(0x1,TIOCGETA,0xefbfd2e0)
 20748 hello    RET   ioctl 0
 20748 hello    CALL  write(0x1,0x8000,0xe)
 20748 hello    GIO   fd 1 wrote 14 bytes
       "Hello, World!
       "
 20748 hello    RET   write 14/0xe
 20748 hello    CALL  exit(0xe)
```

This display contains the following information in columnar format:

1. The process ID of the process.
2. The name of the program from which the process was started. We can see that the name changes after the call to `execve`.
3. The kind of event. `CALL` is a system call, `RET` is a return value from a system call, `NAMI` is a system internal call to the function `namei`, which determines the inode number for a pathname, and `GIO` is a system internal I/O call.
4. The parameters to the call.

In this trace, run on an Intel 486 with BSD/OS 1.1, we can see a significant difference from SunOS: there are no shared libraries. Even though each system call produces two lines of output (the call and the return value), the output is much shorter.

truss

truss, the System V.4 trace facility, offers the most features:

- It can print statistical information instead of a trace.
- It can display the argument and environment strings passed to each call to `exec`.
- It can trace the descendents of the process it is tracing.
- Like *ktrace*, it can attach to processes which are already running and optionally attach to existing children of the processes to which it attaches.
- It can trace specific system calls, signals, and interrupts (called *faults* in System V terminology). This is a very useful feature: as we saw in the *ktrace* example above, the C library may issue a surprising number of system calls.

Here's an example of *truss* output:

```
$ truss -f hello
511:    execve("./hello", 0x08047834, 0x0804783C)  argc = 1
511:    getuid()                                        = 1004  [ 1004 ]
511:    getuid()                                        = 1004  [ 1004 ]
511:    getgid()                                        = 1000  [ 1000 ]
511:    getgid()                                        = 1000  [ 1000 ]
511:    sysi86(SI86FPHW, 0x80036058, 0x80035424, 0x8000E255) = 0x00000000
511:    ioctl(1, TCGETA, 0x08046262)                    = 0
Hello, World!
511:    write(1, " H e l l o ,   W o r l d"., 14)      = 14
511:    _exit(14)
```

truss offers a lot of choice in the amount of detail it can display. For example, you can select a verbose parameter display of individual system calls. If we're interested in the parameters to the `ioctl` call, we can enter:

```
$ truss -f -v ioctl hello
...
516:    ioctl(1, TCGETA, 0x08046262)                    = 0
```

```
516:            iflag=0004402 oflag=0000005 cflag=0002675 lflag=0000073 line=0
516:                cc:  177 003 010 030 004 000 000 000
```

In this case, *truss* shows the contents of the *termio* structure associated with the `TCGETA` request—see Chapter 15, *Terminal drivers*, pages 241 and 258, for the interpretation of this information.

Tracing through fork

We've seen that *ktrace* and *truss* can both trace the child of a `fork` system call. This is invaluable: as we saw on page 119, debuggers can't do this.

Unfortunately, SunOS trace doesn't support tracing through `fork`. *truss* does it better than *ktrace*. In extreme cases (like debugging a program of this nature on SunOS 4, where there is no support for trace through `fork`), you might find it an advantage to port to a different machine running an operating system such as Solaris 2 in order to be able to test with *truss*. Of course, Murphy's law says that the bug won't show up under Solaris 2.

Tracing network traffic

Another place where we can trace is at the network interface. Many processes communicate across the network, and if we have tools to look at this communication, they may help us isolate the part of the package that is causing the problem.

Two programs trace message flow across a network:

- On BSD systems, *tcpdump* and the *Berkeley Packet Filter* provide a flexible means of tracing traffic across Internet domain sockets. See Appendix E, *Where to get sources*, for availability.
- *trpt* will print a trace from a socket marked for debugging. This function is available on System V.4 as well, though it is not clear what use it is under these circumstances, since System V.4 emulates sockets in a library module. On BSD systems, it comes in a poor second to *tcpdump*.

Tracing net traffic is an unusual approach, and we won't consider it here, but in certain circumstances it is an invaluable tool. You can find all you need to know about *tcpdump* in *TCP/IP Illustrated, Volume 1*, by Richard Stevens.