The Feel of Pi

T. A. Cargill

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Pi is an interactive debugger for C and C++ on Eighth Edition UNIX® systems. Its user interface uses multiple windows on a DMD 5620 terminal. *Pi* does not feel like a debugger with a sequential command language, nor does it feel like a debugger where commands from a bitmap display are translated into a sequential command language. In contrast, *pi*'s multiple windows display multiple active views of its multiple subject processes, allowing the programmer to browse through a network of information. The programmer interactively explores a set of executing processes, probing for insight with a tool that really helps.

Each window displays a specific view of a subject process in parallel with the other windows. The contents of pop-up menus are determined by context; the current window and the line of text selected within it.

Pi is written in C++ and uses Eighth Edition's `/proc` to access arbitrary live subject processes.

Introduction

Pi (Process Inspector) is an experiment in debugging with an interactive, bitmap graphics user interface. The debugging technology is conventional: breakpoints are planted in the subject process so that the states the process moves through may be examined. But the user interface is unconventional.

In a conventional debugger, the programmer inputs a sequence of commands that are interpreted by the debugger. The debugger responds with information about the subject process. Several problems arise. First, the debugger can usually accept only the subset of its commands applicable to the debugger's current state. For example, breakpoints can only be set in the current source file, or expressions can only be evaluated in the current activation record. Second, the debugger's output is passive and cannot be used to obtain further information about, or other views of, the process. For example, if a value is displayed by some command in an inappropriate format, the programmer must re-issue the command, specifying another format, or take the value and manipulate it elsewhere. The effect is that any non-trivial debugging is accomplished by combining the debugger with some of our oldest tools pencil and paper. Third, a debugging command language must necessarily be very large, if it is to be useful. Generally, keyboard languages are complicated, and often cryptic.

The goal in writing *pi* was to create a full-function interactive debugger with a good user interface: menu-driven, reactive, usable without a scratch pad or reference manual.

Interface Model

Pi's user interface assigns each view of a subject process to a separate window. Each window has its own menu of operations, appropriate to the view presented. Within each window are lines of text providing details of the window's view. Each line has its own menu of operations, appropriate to the information presented. Interaction is driven by the programmer selecting operations from these menus. In response to each operation, the debugger adds or removes windows, or lines, or their menus. A window or line may also choose to accept a line of input from the keyboard.

On the DMD 5620, a layer is subdivided into a set of scrolling, overlapping windows. The mechanics of the user interface are derived from *jim*, a text editor by Pike [3], [1]. There is a current window (with a heavy border), and within it a current line (video-inverted). Each button on the three-button mouse serves a specific role. Button number 1 is for pointing. If the cursor is outside the current window, button 1 selects a new current window. If the

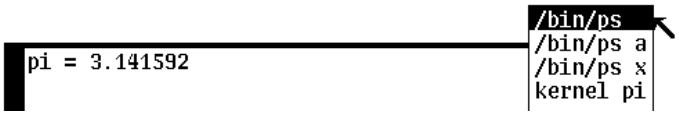
cursor is inside and over a line of text, that line becomes current. If inside and in the scroll zone, the window scrolls to center the proportional scroll bar over the cursor. Buttons 2 and 3 raise the pop-up menus for the current line and window, respectively. Menus also scroll and may have pop-up sub-menus, making large menus relatively easy to use.

An Example

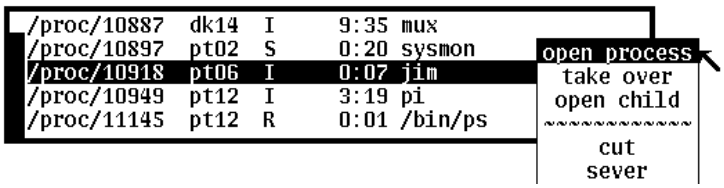
I will demonstrate *pi* by examining the copy of *jim* that I am using to write this paper. *Jim* is two processes, one in the host computer and one in the terminal. I will work with its host process. I create a new layer on my 5620's screen and simply invoke *pi*:

```
pi
```

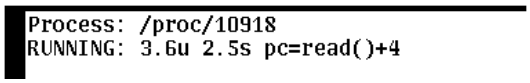
After about 20K bytes of user interface code has downloaded into the 5620, *pi*'s cursor icon requests me to sweep a rectangle for a new window the ''Pi'' window, the master window through which *pi* may be bound dynamically to processes and core dumps. I now have one (almost empty) window in *pi*'s layer:



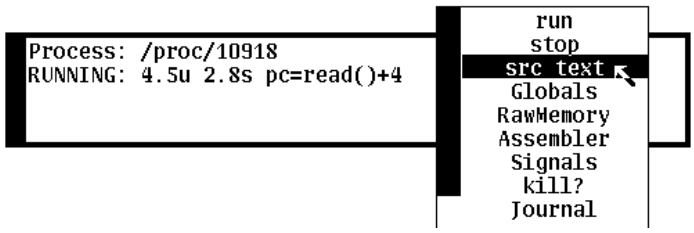
Selecting '`/bin/ps`' from this window's menu runs the *ps* command and lists the output in the window, one process per line:



It shows me with a light load I am only editing. To examine *jim*, I point to process 10918 in this list and select '`open process`' from its menu. I am now requested to sweep a ''Process'' window. The Process window has overall control of the process and can create windows with more detailed views. The process window shows the state of the process, and a callstack if the process is stopped. The state of process 10918 is:



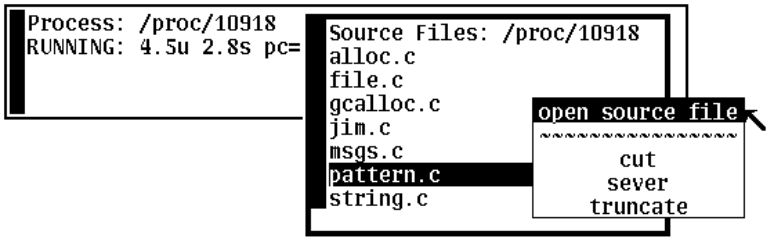
This is the usual state for *jim*'s host process it is blocked reading from the terminal. *Pi* polls the state of the process every second and updates the Process window asynchronously with respect to the user and the subject process. After some more editing the consumed processor time has increased.. I raise the Process window's menu:



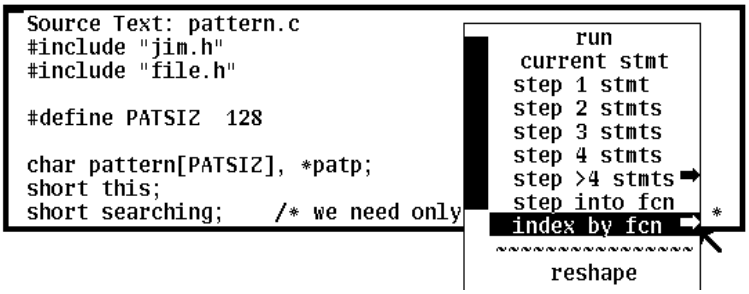
'`stop`' stops the process asynchronously. '`run`' restarts it. '`src text`' creates windows for viewing source text. '`Globals`' creates a window for evaluating expressions in global scope. '`RawMemory`' creates a ''memory editor,'' in which uninterpreted memory cells may be viewed and modified. '`Assembler`' creates a window that dis-

assembles memory and provides instruction level operations. '`Signals`' creates a window that monitors signals to the process. '`kill?`' kills the process; the question mark calls for a confirming button hit. '`Journal`' creates a window that records significant events in the process a trace.

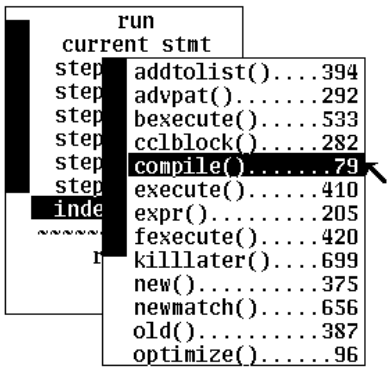
First, I choose to look at some source text. If there were a single source file, '`src text`' would create a ''Source Text'' window for it. *Jim* has several source files; so *pi* asks me to sweep a ''Source Files'' window that lists them:



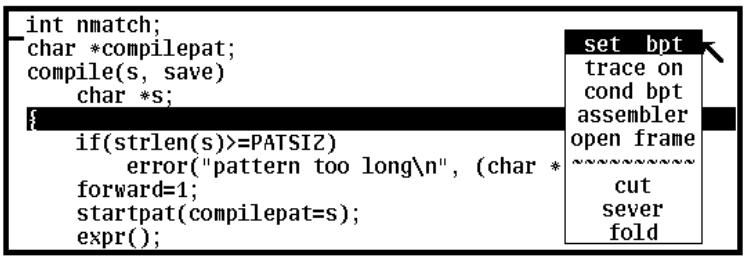
I point to `pattern.c` and choose '`open source file`' from its menu. I sweep a Source Text window. It fills with the first few lines of `pattern.c`. I raise its menu:



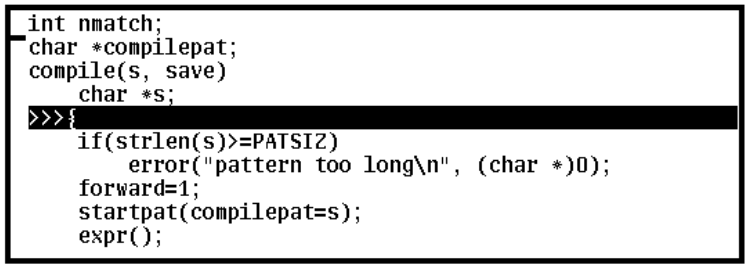
(I have not looked at this code before starting to write this example. I believe I will find *jim*'s regular expression pattern matcher here. I know no details of its implementation. It is as if I were starting from scratch to find a bug in Pike's code.) Moving the cursor over the arrow at the right of '`index by fcn`' pops up a sub-menu that is a table of contents by function (with line number) of `pattern.c`:



It suggests, as I expected, that *jim* compiles regular expressions into a representation from which they can be interpreted efficiently. To see some of this code, I select '`compile().....79`'. This scrolls the window so that the line with the opening brace of `compile()` is in the center:

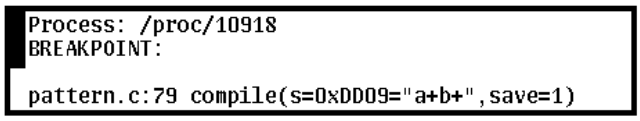


To set a breakpoint, I point to a line of source text, say the opening brace, and select '`set bpt`' from its menu. To indicate the breakpoint, '`>>>`' appears at the beginning of the source line:

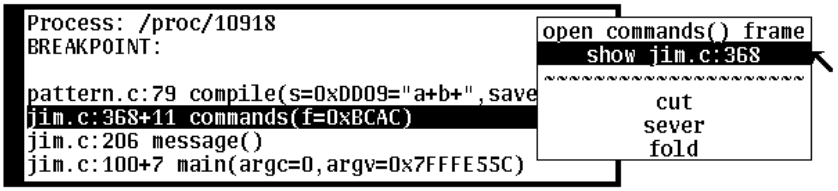


Note that the breakpoint was set while *jim* executed asynchronously.

To force *jim* to execute the breakpoint, I type (in *jim*'s layer) a search command whose pattern matches a non-empty sequence of '`a`' followed by a non-empty sequence of '`b`': `/a+b+`. When *jim* hits the breakpoint, *pi* asynchronously notices its change of state and reports it in the Process window, along with as much of the callstack as fits (here, only the deepest activation record):



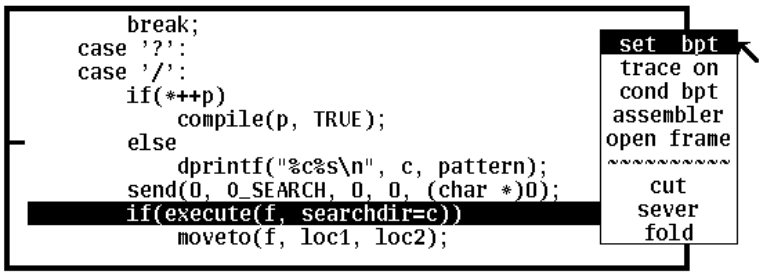
In the Source Text window, the breakpoint source line is selected to show the current context. To see more of the callstack I reshape the Process window, making it larger:



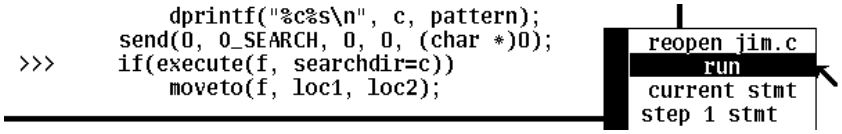
To see the context from which `compile()` was called, I select the `commands(f=0xBCAC)` line from the callstack and choose '`show jim.c:368`' from its menu. I am prompted to sweep another Source Text window, `jim.c`, to see this context. To catch the process before it calls `execute()`, I change the selection from the line

```
compile(p, TRUE);
```

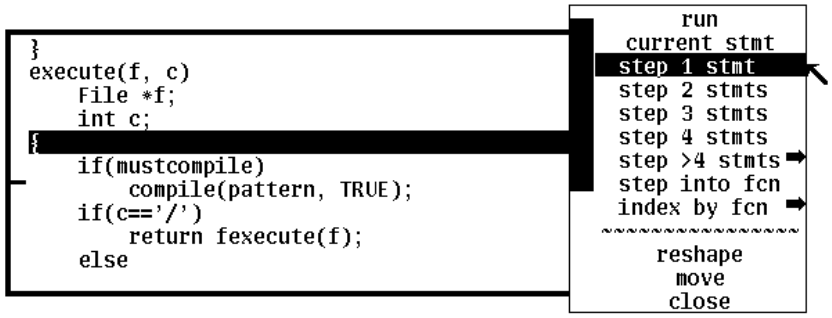
to the `if` statement four lines below and set a breakpoint:



I then '`run`' from the Source Text window's menu:



When *jim* reaches this breakpoint, I choose '`step into fcn`' from the same menu to step the process into `execute()`. (The other source stepping commands step *over* called functions.) The source context for `execute()` is back in the first source file, `pattern.c`. `pattern.c`'s Source Text window moves to the front of the screen and highlights the opening brace of `execute()`:



It appears that the real work will be done by `fexecute()`. I could set a breakpoint there, but I use '`step 1 stmt`' from the source window's menu a few times until I get to:

```
return fexecute(f);
```

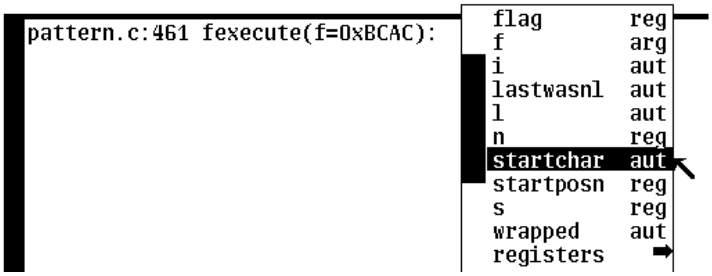
and then use '`step into fcn`' again. The context shown from `pattern.c` changes:

`fexecute()` looks non-trivial. Before going further, I would like to understand the data structure driving it. I do not know what this data structure is. Looking forward through the source text of `fexecute()` I understand very little of the code. But three lines do make sense:

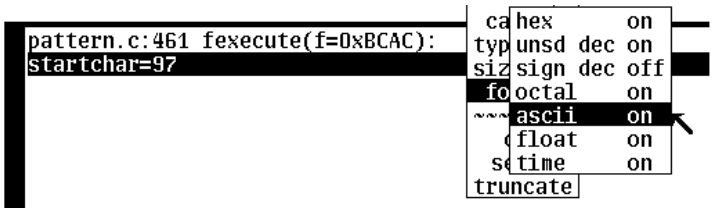
```
/* fast check for first char */
if(startchar && *s!=startchar)
        goto Continue;
```

Surely, `startchar` holds a literal character and `s` is a pointer into a scanned string. To test this I set a breakpoint on the `if` and '`run`'. At the breakpoint I need the value of `startchar`. Choosing '`open frame`' from the source line's menu:

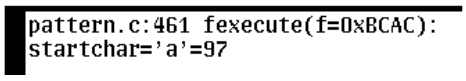
creates a ''Frame'' window for the activation record of the function corresponding to the source line. A Frame window evaluates expressions with respect to its activation record. The menu contains local variables, each flagged as an argument, an automatic or a register:



Choosing `startchar` evaluates that expression:



Is that an '`a`'? The value is in decimal because `startchar` is declared `int`. To override the default format, I select '`format`' from the expression's menu, and '`ascii on`' from the sub-menu. The expression re-displays itself:



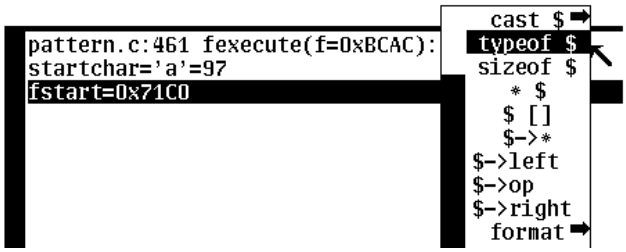
The value of `startchar` looks right and probably came from the data structure I am after. Scrolling back a few lines in `pattern.c` I find an assignment to `startchar`:

```
    nmatch=0;
    match[0].b=0x7FFFFFFF;
    match[0].e=0x7FFFFFFF;
    startchar=0;
    if(fstart->op<0200)
        startchar=fstart->op;
    i=Fseek(f, startp)-1;
    if(i<0 || f->str.s[i]=='\n')
        lastwasnl=TRUE;
Restart:
```

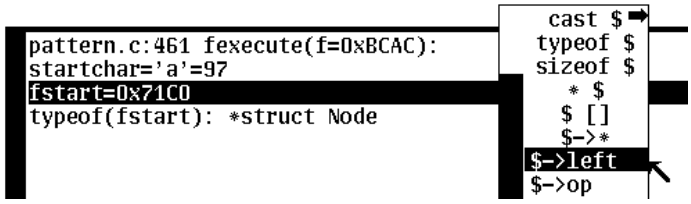
`fstart` may be the pointer I need, but it does not appear in `fexecute()`'s menu. It must be a global. Rather than open the global expression evaluator window and look in its menu, I enter the expression

```
fstart
```

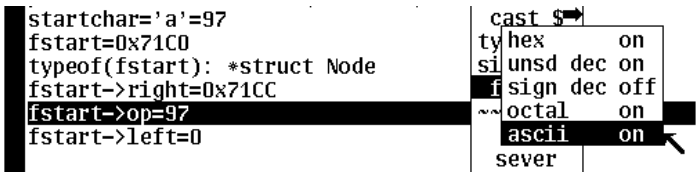
from the keyboard, with `fexecute()`'s Frame window selected as the target. The Frame window now contains two expressions:



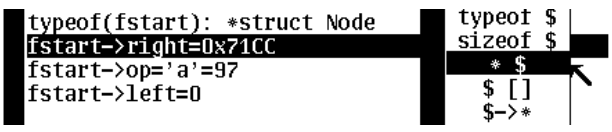
What type is `fstart`? I can almost tell from its menu. Most of the entries in an expression's menu are new expressions that may be derived from it. The `$->`'s tell me that I have a pointer to a structure. (In the menu, and from the keyboard, `$` denotes the current expression.) Choosing '`typeof $`' confirms it:



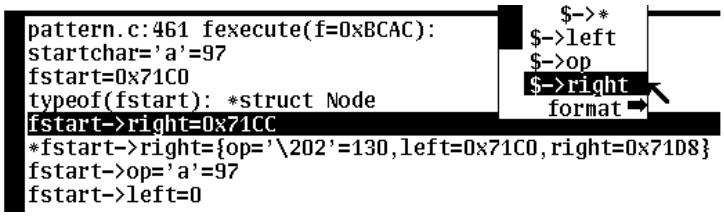
Choosing '`$->left`', followed by '`$->op`', and '`$->right`' yields:



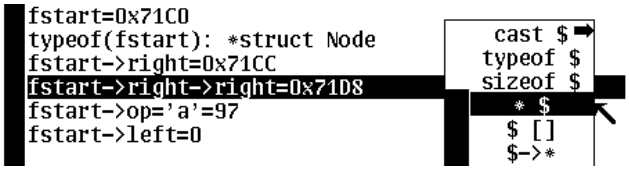
Reformatting `fstart->op` in ASCII leaves:



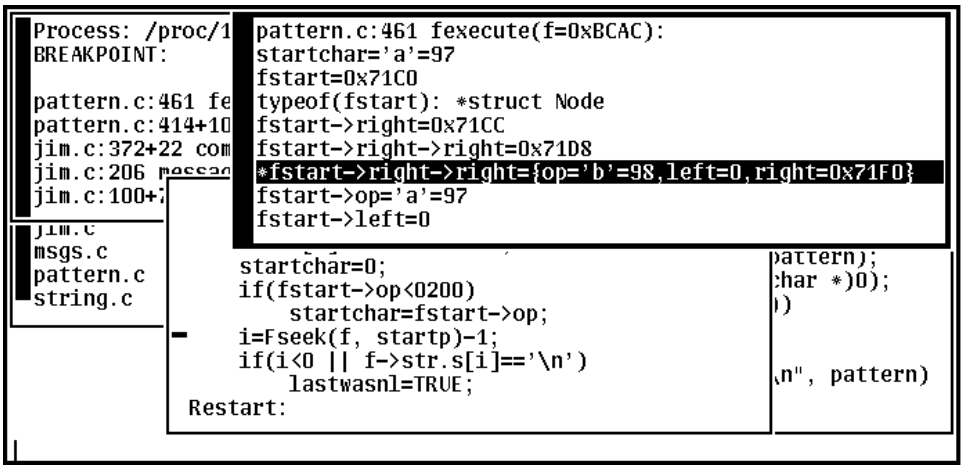
So here is some kind of tree, where an operator code less than octal 200 is to match its own value in the scanned text. The left sub-tree is empty; the right looks promising. Dereferencing with '`* $`' yields:



The `left` field of `fstart->right` is equal to `fstart` itself; maybe this is a doubly-linked list. Applying '`$->right`' to `fstart->right`, I get:



I already know this, but applying '`* $`' produces (showing *pi*'s entire layer for the first time):



Note that the value of the `op` field for the current expression is displayed in ASCII as '`b`'. The ASCII format explicitly requested for that field earlier was saved in the symbol table and is now the default. The `left` pointer is zero here. It now looks as though `left` points back to the beginning of the sub-pattern controlled by the closure operator.

Let me stop here. I have started to unravel the data structure and understand the program. I hope this paper description conveys something of the feel of *pi*.

Programmer Reaction

Most programmers take somewhere from a few hours to a few days to make the transition from drowning in a sea of windows to considering *pi* an indispensable tool. At the outset, they do not expect dynamic binding to subject processes and cannot see why there are so many windows. Invoking a debugger without specifying a dump or program is a foreign notion. Expectations of a debugger are very low: ''I only want the value of *x* when *f()* is called why all the windows?'' With increased confidence and ambition they use *pi* with more sophistication. Styles vary considerably. Each programmer uses idiosyncratic sizes, shapes and placements of windows, especially when debugging multiple processes. Some prefer to enter most of their expressions from the keyboard, others never touch it.

There are two main problems. First, binding *pi* to subject processes is too complicated for novices. Experts demand many special facilities, which have been allowed to complicate what the novice encounters. Second, demand for programmable debugging is growing among the expert users. Programmability was excluded from *pi* in order to concentrate on interactive behavior. *Pi* does have ‘‘spy’’ expressions, which re-display themselves if their values change, and conditional breakpoints, but it is not programmable, say, to step 10 instructions after encountering a breakpoint. It is now time to think about how programmability and interaction can be combined.

Asynchronous Multiple Processes

An arbitrary set of processes may be examined simultaneously. For each subject process there is an independent network of windows. Since all the windows are in a flat space on the screen, each successive action from the programmer may be in an any window, associated with an any process. Events in the set of subject processes are reported as they occur. For example, the programmer might step source statements alternately between a pair of processes while watching the changing values of spy expressions in a third process. This simplifies debugging situations that were difficult or impossible in the past. For example, it becomes straightforward to (i) compare the behavior of two similar programs; (ii) compare the effects of different inputs on a single program; (iii) observe the interaction between related processes, say child and parent.

Implementation

Pi depends on the Research UNIX’s `/proc` [2], [1], and object-oriented programming in C++ [4].

`/proc` permits *pi* to bind itself dynamically to any processes, and execute asynchronously with them. For each process, *pi* can tell the kernel how to handle an *exec()* by the process and signals received from other processes. A breakpoint in code executed by a child of a subject process suspends the child so that it may be opened and examined. Code sharing is managed transparently by `/proc`.

The browsing and asynchrony are driven by object-oriented programming in C++. A large host C++ program communicates with a small 5620 C program. Everything the programmer can identify on the screen is a C++ object, an instance of a class. The host program binds an object identifier (which can be thought of as a host address) and a menu of operations to each window and each line of text as it describes them to the terminal. When the programmer selects an operation from a menu associated with an object’s image, the terminal sends back a remote invocation of one of the object’s member functions. Generally, executing this function creates, changes or removes host objects and their images in the terminal. Host-terminal communication is asynchronous; the programmer need not wait for results to appear on the screen before issuing another operation. There is no ambiguity in this ‘‘mouse-ahead’’; the identity of the object on which a menu operates is frozen when the menu is raised. A crude object registration scheme in the host detects (with high probability) and ignores operations for objects that have been destroyed.

Conclusion

Pi’s easy access to information about arbitrary processes has made programmers more sophisticated in their debugging practices. Programmers working with large programs written by others are happier. Programmers who would not normally read assembly code can sometimes spot code generation bugs in the compiler. Programmers with families of interacting processes have a handle on them. In general, programmers understand their programs better.

References

1. Hume, A.G. and McIlroy, M.D., Eds. *Unix Programmer’s Manual, Tenth Edition*. AT&T Bell Laboratories, Murray Hill, NJ 07974, September 1989.
2. Killian, T.J. Processes as Files. In *USENIX Summer Conference Proceedings*, Usenix, Salt Lake City, June 1984, pp. 203-207.
3. Pike, R. The Blit: A Multiplexed Graphics Terminal. *AT&T Bell Laboratories Technical Journal 63*, 8 (October 1984), 1607-1632.
4. Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, 1986.

photo page

divider with title

Supporting Tools and Languages