PIX, the latest NeWS

Wm Leler

Cogent Research, Inc.
wm@cse.ogc.edu

1. Introduction

One of the more innovative features of the NeWS™ window server from Sun Microsystems is the addition of quasi-parallelism to PostScript®. NeWS uses this parallelism to deal with the inherent concurrency of human/computer interaction, as manifested in the following ways:

- Windows on the screen serving separate, simultaneously running applications.
- Asynchronous user input.
- Advanced graphics hardware incorporating multiple processors executing in parallel.

NeWS provides multiple, light weight processes to deal with this parallelism; as a result, user interfaces are demonstrably easier to implement using NeWS than using other window servers.

The pseudo-concurrency of NeWS naturally suggests that it might be an excellent candidate for execution on a true parallel processor. The goal of the PIX project was to produce a version of NeWS that runs on parallel processors, and to fix some problems with NeWS. In addition, executing NeWS on a parallel processor does more than just provide execution speed advantages, it begins to redefine the role of server-based user interfaces in modern computer systems.

2. PIX

PIX stands for Parallel Interactive eXecutive, although PIX runs well and provides worthwhile advantages even when run on a single processor. This document describes the differences between PIX and the NeWS window server.

The PIX extensions to NeWS are:

1. Processes are scheduled preemptively.
2. Linda operators are used for safe interprocess communication.
3. The event mechanism has been decentralized.
4. Magic.

All of these extensions are upwardly compatible with NeWS, with the exception of preemptive processes, but we are assuming that the following quote from page 13 of the NeWS manual is prophetic.

> *The current scheduling policy might change to a pre-emptive scheduling policy in the future. Thus, it is unwise to write PostScript code that relies on the behavior of a non-pre-emptive scheduling policy.*

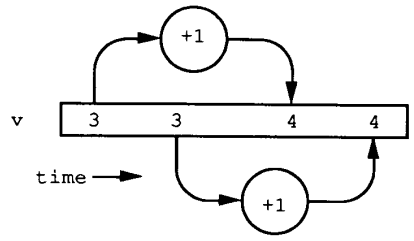
PIX is a complete rewrite of NeWS, written in C++. We found the storage management and strong typing of C++ to be of great benefit. Garbage collection was *the* major implementation headache for NeWS, but C++ abstract data types allow storage management to be encapsulated into the type definitions. In addition, our experience indicates that C++ helps produce code that is much more portable, especially to non-UNIX™ systems.

2.1. Preemptive processes

Concurrently running processes can potentially access shared data simultaneously, often with disastrous consequences. For example, consider the following PostScript code to increment the value of v in dictionary d.

```
d /v d /v get 1 add put
```

If two processes execute this code simultaneously, errors can result. In the following diagram, v is incremented by two different processes, but the value of v only increases by one.



Although NeWS adds multiple processes to PostScript, these processes are scheduled nonpreemptively. A NeWS process is guaranteed to have exclusive control of the window server until it explicitly gives up control by executing an I/O operation, or by calling the NeWS *pause* operator. Unfortunately, this means that PostScript code not written explicitly for NeWS, or code containing time-consuming primitives (such as *imagecanvas*) can tie up the NeWS system, even to the exclusion of mouse tracking. Even when processes are well behaved, things don't actually happen in parallel, which is disconcerting in a windowing interface. Even on a single processor, preemptive behavior is desirable to make multiple processes at least appear to run concurrently.

Nonpreemptive scheduling allows a process to modify shared data without worrying about whether another process might modify the same data concurrently. Consequently, constructs such as monitors or critical

regions are not strictly required to protect shared data, although their use is advised. Unfortunately, Sun's PostScript code often ignores this advice, and depends on nonpreemptive behavior.

Nonpreemptive behavior is only possible if one process can have exclusive control of the window server at a time. On a parallel processor, where multiple processes are executing concurrently, nonpreemptive behavior is meaningless. Some other convenient and safe way to access shared data is required.

2.2. Linda

The Linda communication primitives, developed by David Gelernter at Yale, are a perfect answer to the communication needs of NeWS. With the Linda primitives, processes communicate via shared data spaces called tuple spaces. In PIX, we use PostScript dictionaries as tuple spaces. There are three Linda primitives:

```
dict key value  out  –
dict key  rd  value
dict key  in  value
```

The Linda `out` operator is similar to the PostScript `put` operator; it places a key/value pair into `dict`. The difference is that if multiple `out`s are performed to the same key in a dictionary, the values are all queued (instead of overwriting each other).

The Linda `in` operator is similar to the PostScript `get` operator; it returns the `value` corresponding to `key` in `dict`. In addition, the `in` operator removes the key/value pair from the dictionary. If the key has multiple values (from repeated `out` operations) then only one of them is removed. If the key is not defined in the dictionary, then this call blocks until it is defined.

The `rd` (pronounced "read") operator is just like the `in` operator, except that the key/value pair is not removed from the dictionary.

The Linda primitives can be used to make access to shared memory safe. Consider the previously discussed PostScript code to increment the value of `v` in the dictionary `d`.

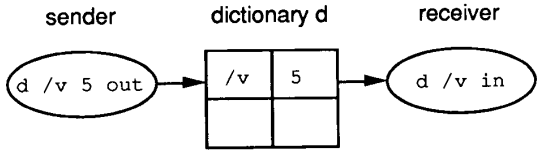
```
d /v d /v get 1 add put
```

Using the Linda primitives, we replace `get` with `in` and `put` with `out`.

```
d /v d /v in 1 add out
```

The `in` operator removes `v` from `d`, so if another process tries to execute this code while `v` is being incremented, that process will block until the current process replaces the value (using `out`, although `put` could be used here).

The Linda primitives can also be used to perform a message send, by having one process send values using `out`, and another process receive values using `in`.



A message broadcast can be performed by having multiple processes each perform a `rd` to receive a message sent by a single process using an `out`.

Even on a sequential processor, the Linda primitives make it easy to write safe code. Programs that depend on nonpreemptive behavior often contain subtle bugs that are difficult to find. Programs that depend on nonpreemptive behavior are also harder to maintain, because a seemingly simple change can easily introduce a race condition or some other bug.

2.3. Events

Many people have noted that the NeWS event mechanism is one of the hardest parts of NeWS to use and understand. In addition, it depends on a highly centralized event matching and distribution mechanism that would be an unacceptable bottleneck on a parallel computer. In PIX, we can use the Linda primitives (instead of NeWS events) to send messages. For example, PIX uses the following code to repaint the framebuffer when part of it becomes damaged

```
{
      newprocessgroup {
      framebuffer /DamageEvent in pop
      damagepath clipcanvas PaintRoot
      newpath clipcanvas
      } loop
} fork pop
```

This code forks a process that loops, waiting on an `in` operation on the name `/DamageEvent` in the framebuffer dictionary. When the framebuffer becomes damaged, the PIX system defines a key called `DamageEvent` in the framebuffer dictionary, as follows

```
framebuffer /DamageEvent 0 put
```

This wakes up the waiting process, which then repaints the damaged parts of the framebuffer, and then loops to wait on another damage event.

This example also shows how events in PIX are associated with a dictionary. In this case, the damage event on a canvas is naturally associated with the appropriate NeWS canvas dictionary, rather than being distributed from a centralized event queue. This localization of interaction not only makes it easier to write programs that use events, it also makes them more efficient to implement on a parallel processor.

The Linda primitives can also be used to wait for processes to finish. When a process finishes, it places the top of its operand stack in its process object (a dictionary) under the name `/Finished`, as follows

```
currentprocess exch /Finished exch put
```

If a process has access to a process object for another process, it can wait for that process to finish by executing the code

```
otherprocess /Finished in
```

which will return the top of stack for the other process. To maintain upward compatibility with NeWS, we define the following operator

```
waitprocess { /Finished in } def
```

Consequently, in PIX, processes can wait for other processes to finish in the same way that they wait for any other event, instead of having two separate mechanisms, as in NeWS.

Using the Linda operators, we were able to implement the entire NeWS event mechanism *entirely in PostScript.* Of

course, it ran horribly slow, so we sped up some parts of it by implementing them in C++ (mainly the code to do event matching). Since our implementation of the NeWS event mechanism still uses the Linda primitives to interface to the rest of the system, it can easily be replaced by some other mechanism, if desired.

Compatibility issues aside, we would really prefer to replace the NeWS event mechanism with a distributed event server, or at least something that is easier to use. We would especially like to speed up the response to user events. Linda-based events can give an order of magnitude improvement in the immediacy of interaction, so more powerful forms of user input, such a gestural input, handwritten input, or continuous input become not only possible, but reasonable.

Because of the preemptive scheduling policy of PIX, we were not able to maintain complete compatibility with the NeWS event mechanism. In particular, NeWS guarantees that all processes interested in a particular event will have a chance to run before the next event is distributed. Even worse, a process receiving an event can execute the `blockinputqueue` operator to suspend event distribution. In a distributed system, such behavior is not only undesirable, it is meaningless. Luckily, we found that such behavior is not necessary; it is always possible to rewrite such code, typically using the Linda operators, so that it behaves correctly without the need to delay or block the input queue.

2.4. Magic

In NeWS, some dictionary entries are magic, and when you access them, mysterious things happen. For example, when you set the value of `/Mapped` in a canvas dictionary to `true`, the canvas appears on the screen. This behavior is entirely alien to the PostScript language, and is possible only because canvasses and processes only behave like dictionaries, and are not actually implemented as them. In PIX, we wanted either to get rid of this magic behavior, or else make it more general. We couldn't get rid of it entirely because that would destroy compatibility, so we made it more general. PIX allows any entry in any dictionary to be made magic. This is done by two new operators: `defmagic` and `undefmagic`.

```
dict key proc defmagic -
dict key undefmagic -
```

The `defmagic` operator defines `key` in `dict` to be magic; when any operator is executed on that key, instead of performing the operation, the procedure is executed instead. This procedure is passed the name of the attempted operation and any other arguments; the dictionary is pushed onto the dictionary stack (and popped off afterwards). For example,

```
d /v { == (executed\n) print } defmagic
d /v 2 put
```

results in the following output:

```
/put
executed
```

and leaves `/v` and `2` on the operand stack.

The dictionary is pushed onto the dictionary stack, instead of on the operand stack, in order to make object-oriented programming easier (at least, the NeWS version of object-oriented programming).

PIX magic is used, not only in the implementation of NeWS magic dictionaries, but also for debugging purposes. It is possible to make any entry in any dictionary be magic, so we can monitor who changes a value, and when. This is similar to active variables in some languages.

2.5. Misc.

We also changed the transmission protocol for client to server connections, but this is transparent to most users and applications. We only allow names to be encoded, as a single byte greater than 127. Other tokens, including numbers and strings, are sent (unencoded) in ASCII format. If a number or string is to be sent repeatedly, it should be assigned to a name. Encoded names are treated like normal PostScript "special" tokens by the scanner (they do not need to be delimited, and they serve as delimiters). Consequently, they can be defined and used just like any other PostScript token, without using any special syntax. Many systems can produce these tokens on their keyboard, so they are easy to define and use. For example, if ∂ and π are two characters greater than 127, then

```
/∂/def load def
/π3.14159∂
```

defines ∂ to be the same as `def`, and then defines π.

Using this encoding scheme, the example from page 147 of the NeWS Manual

```
"10 300 moveto\n(Hello world) show "
```

can be encoded in 21 bytes, using μ for `moveto` and § for `show`:

```
"10 300μ(Hello world)§"
```

This is two characters longer than the (barely human readable) NeWS encoding

```
"\200\012\201\001\054\261\233Hello world\262"
```

but still 12 shorter than the unencoded format. In fact, if `moveto` and `show` were not in the NeWS encoding table of the 32 most common operators, then they would require an extra byte each to encode, so the PIX encoding would be the same length as the NeWS encoding. In addition, the PIX encoding scheme is easier to set up and use, so we expect that its likely greater use will make the actual savings even larger.

3. Separately compiled primitives

A common complaint about NeWS is that in order to extend NeWS a programmer must write programs in PostScript. Although many people would argue that learning PostScript is no harder than learning the various subroutine calls needed by other window servers, there is still the problem that PostScript programs are interpreted, so there is an execution speed penalty. Window systems such as Andrew have attempted to solve this problem by using a compiled language like C for their extension language. Unfortunately, this means that it is easy for an errant user program to crash the entire server, unlike a system that uses an interpreted language, where errors are easily caught by the interpreter.

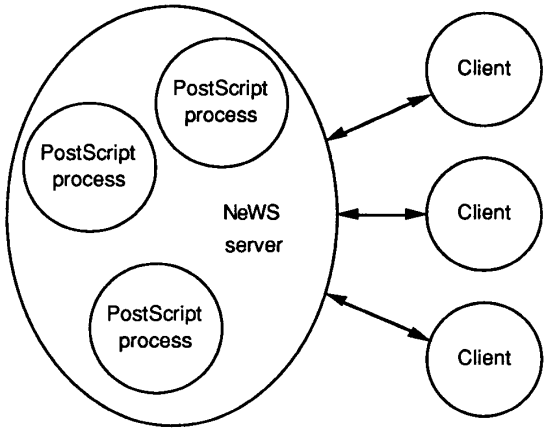
PIX allows separately compiled functions to be executed as PostScript primitives. These functions can be written in any language that has access to Linda, such as C++, C and even FORTRAN, with other languages easily added. In PIX, these separately compiled functions execute as

separate tasks, so they do not share address space with each other (they might not even execute on the same processor) and they communicate with each other using Linda. Consequently, the danger of a separately compiled function crashing the server is greatly reduced. Separately compiled functions communicate with PIX via Linda, so these functions have full access to PostScript dictionaries and other objects.

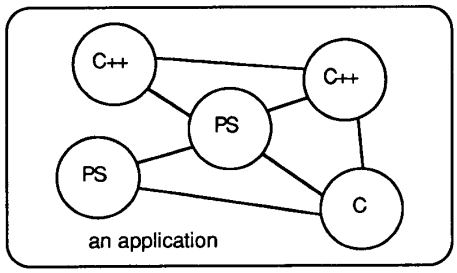
This seemingly simple addition to PIX has other significant implications.

3.1. The NeWS client/server schism

In NeWS, there is a wide conceptual gulf between the clients (the application programs) and the window server (NeWS). Each client communicates with the NeWS server via a single communication channel, and user interface code must be downloaded into the server by sending PostScript programs to be executed.



In PIX, the window server is not a single process, it is a collection of processes communicating with each other via Linda. Likewise, each client is a collection of processes, all communicating with each other (and with PIX processes) via Linda. Because PIX processes are no different from regular client processes, there is no need to treat them any differently.



As mentioned above, PIX processes can be written in languages other than PostScript, but since there is no difference between PIX processes and client processes, this also means that "client" processes can be written in PostScript. Why would anyone want to write part of a application program in an interpreted language like PostScript? For the same reason that part of an application program might be written as an interpreted shell script.

3.2. PIX as a command shell

Until now, graphical user interfaces have not had the ability to define and use shell scripts, such as are commonly provided by command driven interfaces, but in PIX, PostScript programs can be used as shell scripts. Using PostScript for shell scripts has several advantages. PostScript has a better set of data types and control structures than a typical shell language, and it also has a good set of graphical primitives for implementing interactive input and output. In PIX, PostScript also has the ability to create new processes, and to communicate with other processes using Linda, so it is a very powerful shell language. And PostScript, even with its postfix syntax, is hardly any more bizarre than existing shell languages (and its syntax could be changed by a preprocessor, if desired).

Conclusions

PIX is not just NeWS running in parallel, it is a better version of NeWS. Preemptive processes allow user programs to operate truly in parallel, and so free the programmer from scheduling concerns. The Linda communication primitives make it easier for separate processes to communicate safely, and also allow PIX to be portable between different types of parallel processors. Decentralized events allow user interaction to be more easily controlled, and make more advanced forms of user interaction possible. Other PIX features, such as magic, generalize existing constructs in NeWS and make them available to the application programmer. PIX support for separate processes and compiled primitives redefines the role a window server plays in a computer, while retaining conventional features such as shell scripts.

We believe that parallel processing will become much more common in the future, but even single processor machines will run operating systems (such as Mach) that support light weight processes and pseudo-parallelism, and so will be able to support the advanced features of PIX. One of the strengths of NeWS is that it is ready to move into the next generation of computing with only a few minor changes, such as the ones that have been made in PIX.

References

David Gelernter, "Linda and Friends" *IEEE Computer*, 19 : 8 (August 1986).

David Gelernter, "Getting the Job Done," *Byte Magazine* (November 1988).

Philip Goward, *PIX: NeWS for Parallel Computers*, M.Sc. thesis, University of Manchester, England.

Wm Leler, *PIX-Disk*, Cogent Research Tech Report, 1987.

Sun Microsystems, *NeWS Manual*, March 1987.

Sun Microsystems, *NeWS Technical Overview*, January 1988.

Adobe Systems, *PostScript Language Reference Manual*, Addison-Wesley, 1985