RunTime : Pipes in Linux, Windows 2000, and Windows XP

High-performance programming techniques on Linux and Windows

Dr. Edward G. Bradford (egb@us.ibm.com)
Senior programmer, IBM
October 2001

This month Ed begins a series of investigations into operating system programming interfaces, starting with pipes. He has also added a new operating system to the lineup; Windows XP has recently released. In this installment, Ed will be working with pipes on Windows 2000 Advanced Server (with Service Pack 2 installed), Linux (based on Red Hat 7.1), and the newly released Windows XP professional. Share your thoughts on this article with the author and other readers in the discussion forum by clicking **Discuss** at the top or bottom of the article.

Before we get started, please note that our nomenclature will be slightly different now that there are two versions of Windows on the market. I will refer to "Windows" when making no distinction between Windows 2000 and Windows XP. When there is a distinction, I will use "Windows 2000" or "Windows XP".

Pipes

A pipe is an interprocess communication mechanism available on both Windows and Linux (and UNIX). Pipes originally appeared in the Bell Laboratories version of UNIX and have remained in all UNIXes and Linux since their inception. A pipe is a stream of bytes accessed through normal IO interfaces. It is created, and then written to or read from using whatever read or write IO system calls are available on the operating system. In the UNIX and Linux case, the IO calls are `read()` and `write()`. In the Windows world, the APIs are `ReadFile()` and `WriteFile()`. Windows pipes differ from Linux pipes in that a single handle (analogous to a Linux file descriptor) supports bi-directional IO. Linux pipes return two file descriptors to effect bi-directional IO.

Windows pipes

Windows pipes are more complicated than Linux pipes. Windows supports named and unnamed pipes. The unnamed variety are simply named pipes where the interface doesn't divulge the name. Windows supports asynchronous IO on pipes whereby a single thread will not block on an IO call to a pipe. Different IO interfaces are required to use the asynchronous IO feature. Windows pipes are of two varieties: byte type and message type. The byte-type pipes are similar to UNIX pipes and support byte streams. I won't be investigating the message-type pipes in this article, although a complete comparison would have to include them. Windows pipes are created with the `CreateNamedPipe()` API. Once created, the `OpenFile()` API is used to access the other end of the newly created named pipe. Pipe names live in a flat name space. For example, `\\.\pipe\anyname` would be a legitimate name for a Windows named pipe. In C or C++ the name is expressed as:

```
char *pipeAdult = "\\\\.\\pipe\\anyname";
```

Only the *anyname* part can be specified. To use a Windows pipe, it must be created by one API, and opened with another. The example code snippet shows how this is done.

Creating a Windows named pipe

```
        //
        // Create named pipe in Windows
        // nbytes -- block size from command line arguments.
        //
        int mult = 1;
        int x;
        x = mult*nbytes + 24;
        handleA = CreateNamedPipe(pipeAdult,
                                  PIPE_ACCESS_DUPLEX,
                                  PIPE_TYPE_BYTE,
                                  2,              // two connections
                                  x,              // input buffer size
                                  x,              // output buffer size
                                  INFINITE,       // timeout
                                  NULL);          // security
        if(handleA == INVALID_HANDLE_VALUE) {
                printf("CreateNamedPipe() FAILED: err=%d
", GetLastError());
                return 1;
        }
        handleB = CreateFile(pipeAdult,
                                  GENERIC_READ|GENERIC_WRITE,
                                  FILE_SHARE_READ|FILE_SHARE_WRITE,
                                  NULL,
                                  OPEN_EXISTING,
                                  FILE_ATTRIBUTE_NORMAL,
                                  NULL);
        if(handleB == INVALID_HANDLE_VALUE) {
                printf("CreateFile() FAILED: err=%d
", GetLastError());
                return 1;
        }
```

The number 24 in the first executable line of code above was determined experimentally. I found no mention of it anywhere in the Platform SDK. If it is not present, the program doesn't work. Apparently, the pipe facility requires a 24-byte header on each write to the pipe.

Linux pipes

Linux pipes are simpler to create and use if, for no other reason, fewer parameters are needed. To accomplish the same pipe creation task as Windows, Linux and UNIX use the following snippet:

Creating a Linux named pipe

```
        int fd1[2];
        if(pipe(fd1)) {
                printf("pipe() FAILED: errno=%d
",errno);
                return 1;
        }
```

Linux pipes have a limitation on the size of a write before it blocks. The kernel level buffer dedicated to each pipe is exactly 4096 bytes big. A write of more than 4K will block unless a reader empties the pipe. In practice this is not much of a limitation because the read and write actions are performed in different threads.

Pipe speeds for a single threaded process

I wrote a program (pipespeed2.cpp) to test how fast the operating system pipe code executes. It creates a pipe and writes and reads data, all in a single thread. Because Linux can support only a 4K write before the writer blocks, our test stopped at the 4K block size. The bash shell script that generated our numbers was simple:

Bash shell file for generating test results

```
list="1 2 3 4 6 8 10 12 14 16 20 24 28 32 36 40 44 48 52 56 60 64 72 80 88"
list="$list 96 104 112 120 128 144 160 176 192 208 224 240 256 288 320 352"
list="$list 384 416 448 480 512 576 640 704 768 832 896 960 1024 1280 1536"
list="$list 1792 2048 2560 3k 3584 4k"
uname -s -r
for bytes in $list
do
    case $bytes in
    ????|?k ) count=100k;;
    ?????|??k) count=100k;;
    ?|??|??? ) count=500k;;
    esac
    pipespeed2 $count $bytes
done
```

The output of each run was saved to a file and edited with a text editor to produce a text file easily imported into Microsoft Excel.

Compiling the programs
Programs in this article are compiled with:

- Linux: gcc -O2 pipespeed2.cpp -o pipespeed2
- Windows: cl -O2 pipespeed2.cpp

and

- Linux: gcc -O2 pipespeed2t.cpp -lpthread -o pipespeed2t
- Windows: cl -O2 pipespeed2t.cpp

Figure 1 shows the results for the Linux 2.4.2 kernel delivered with Red Hat 7.1.

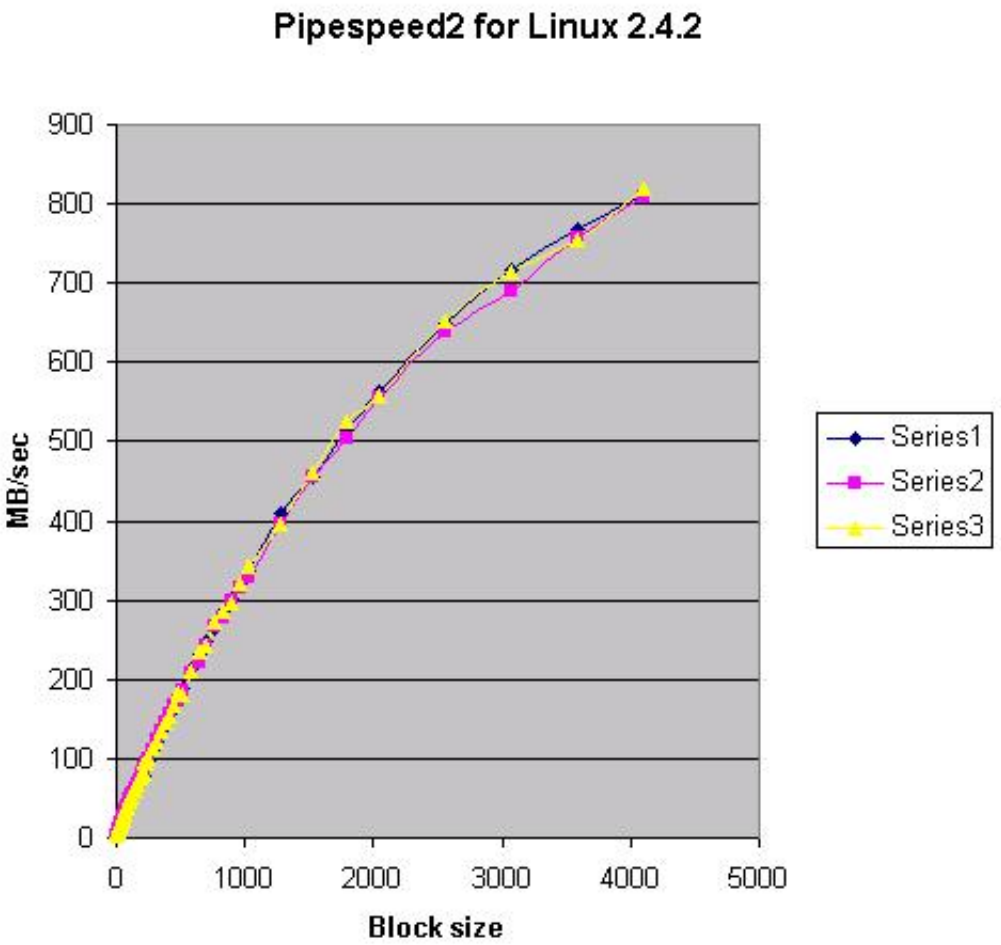

Figure 1

Figure 2 shows the same program compiled to run on Windows for Windows 2000 Advanced Server. Most of the non-essential services were disabled.

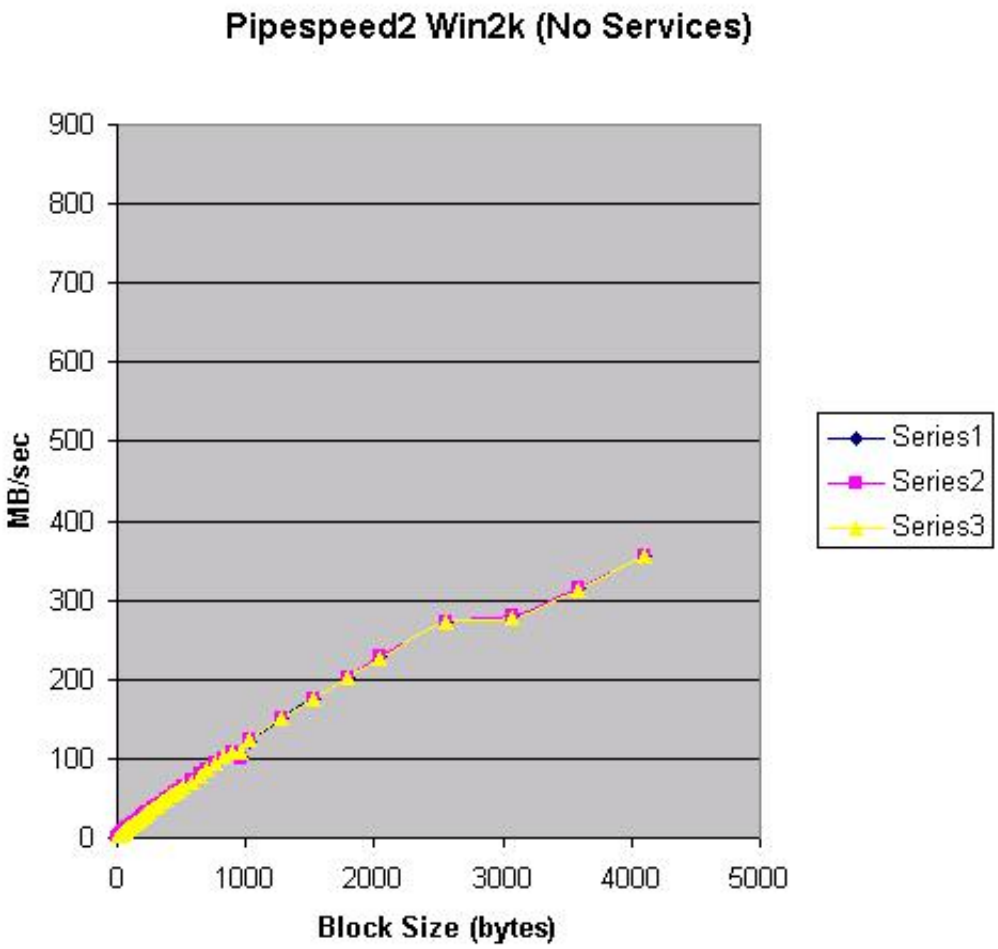

Figure 2

Figure 3 shows the results for the newly released Windows XP.

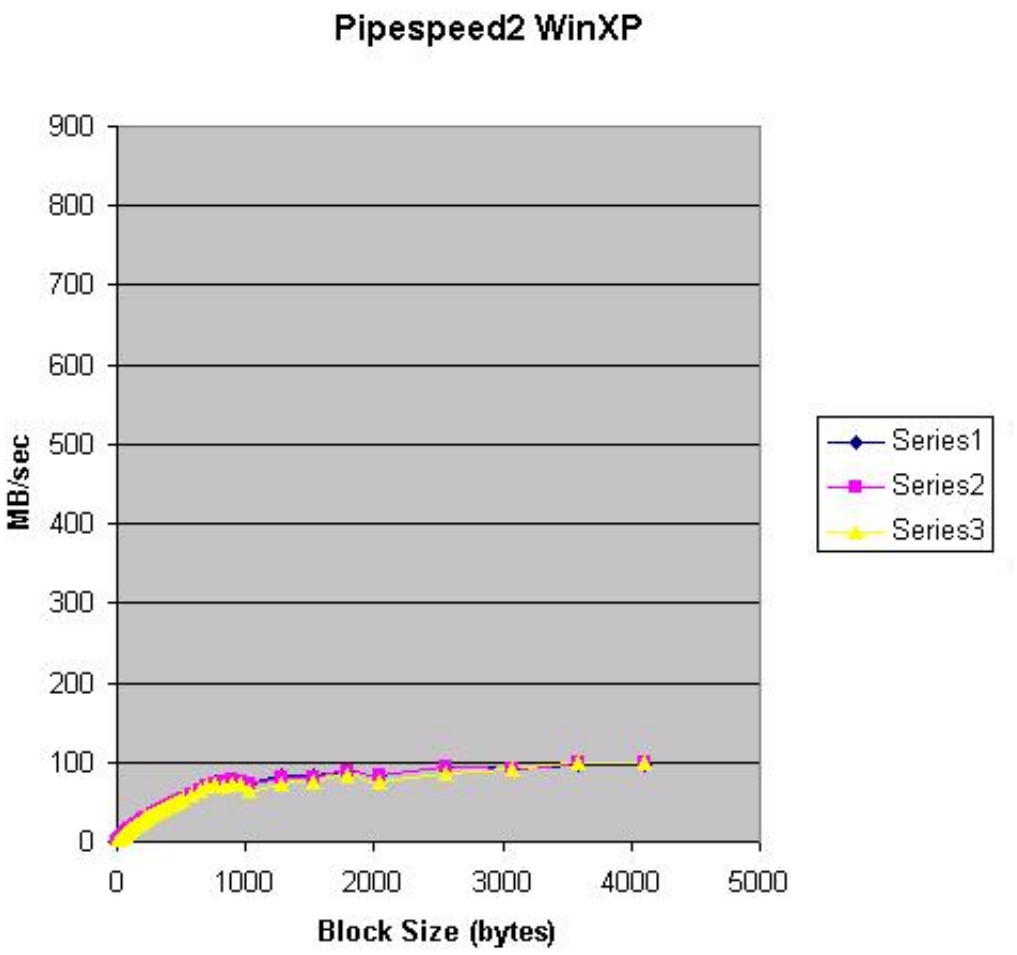

Figure 3

Each test run consisted of three runs (the three series). As you can see in the graphs, Linux pipes are considerably faster than the Windows 2000 namedpipes. The Windows XP named pipes are much slower than Windows 2000 named pipes. The version of Windows XP Professional under test is the released "evaluation" version.

Pipe speeds for a threaded process
The Windows 2000 graph shows signs of further improvement if the block size is increased beyond 4K To test block sizes greater than 4K an enhanced version of the pipespeed2.cpp program was written. The new program creates a second thread. The first thread writes the data, and the second thread reads the data. The purpose of the first program was to understand the overhead associated with the code paths of pipe support without the context switch overhead. It is an artificial environment probably never used.

The enhanced version of pipespeed2.cpp is called pipespeed2t.cpp. Two threads will necessarily context switch back and forth to transmit all the data. Thus, the second program has an additional overhead of context switching missing in the first. Timing is stopped only after all the data has been sent and the second thread has terminated properly.

Figures 4, 5, and 6 show Linux, Windows 2000, and Windows XP respectively. It is apparent that Windows XP has suffered a serious and visible performance degradation in the named pipe facility. While Linux easily outperforms Windows 2000, Windows 2000 easily outperforms Windows XP.

Figure 4 shows the results for the threaded version of pipespeed2t.cpp running on a Linux 2.4.2 kernel delivered with Red Hat 7.1. Its peak IO rate is around 700 MB/sec.

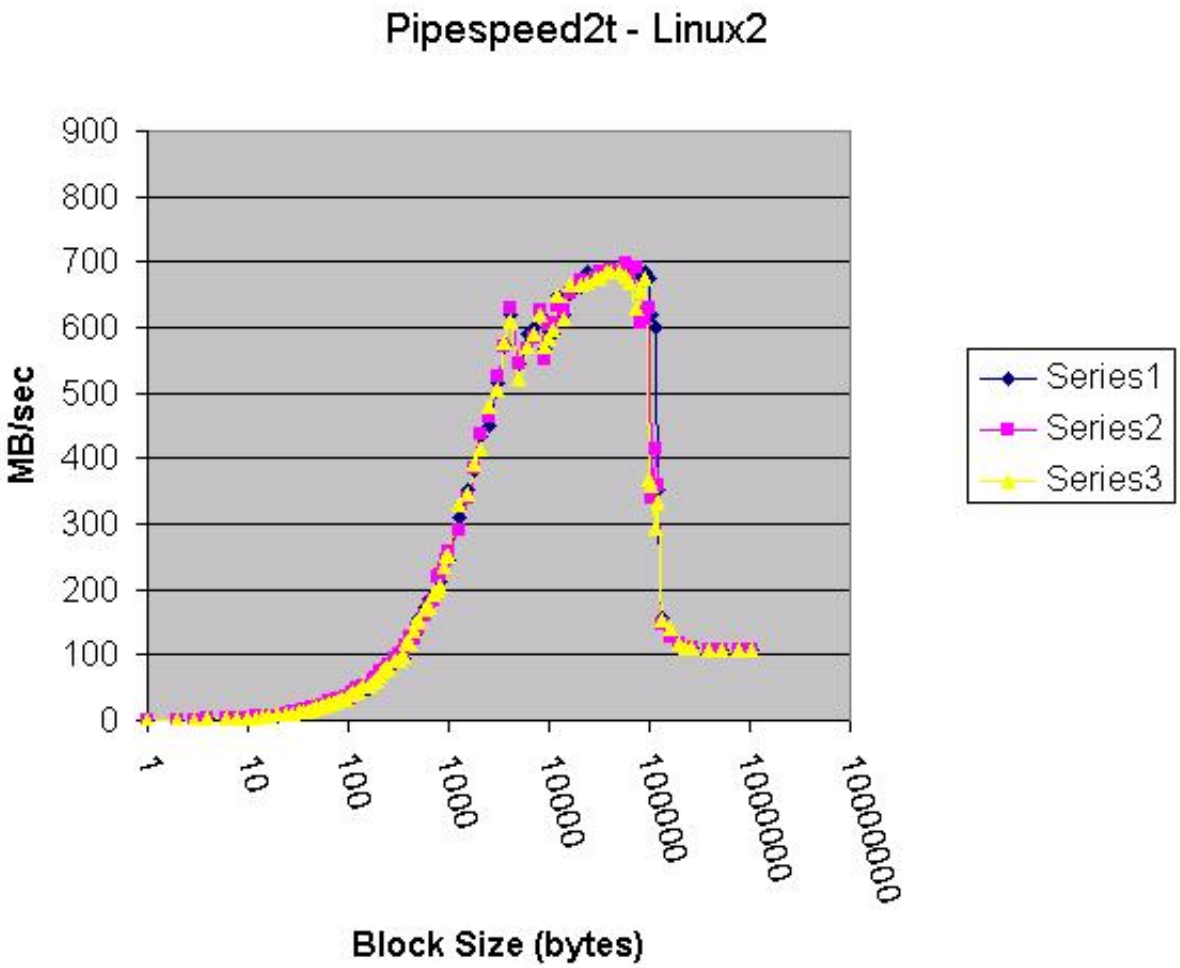

Figure 4

Figure 5 shows the threaded results for Windows 2000. Its peak IO rate is near 500 MB/sec.

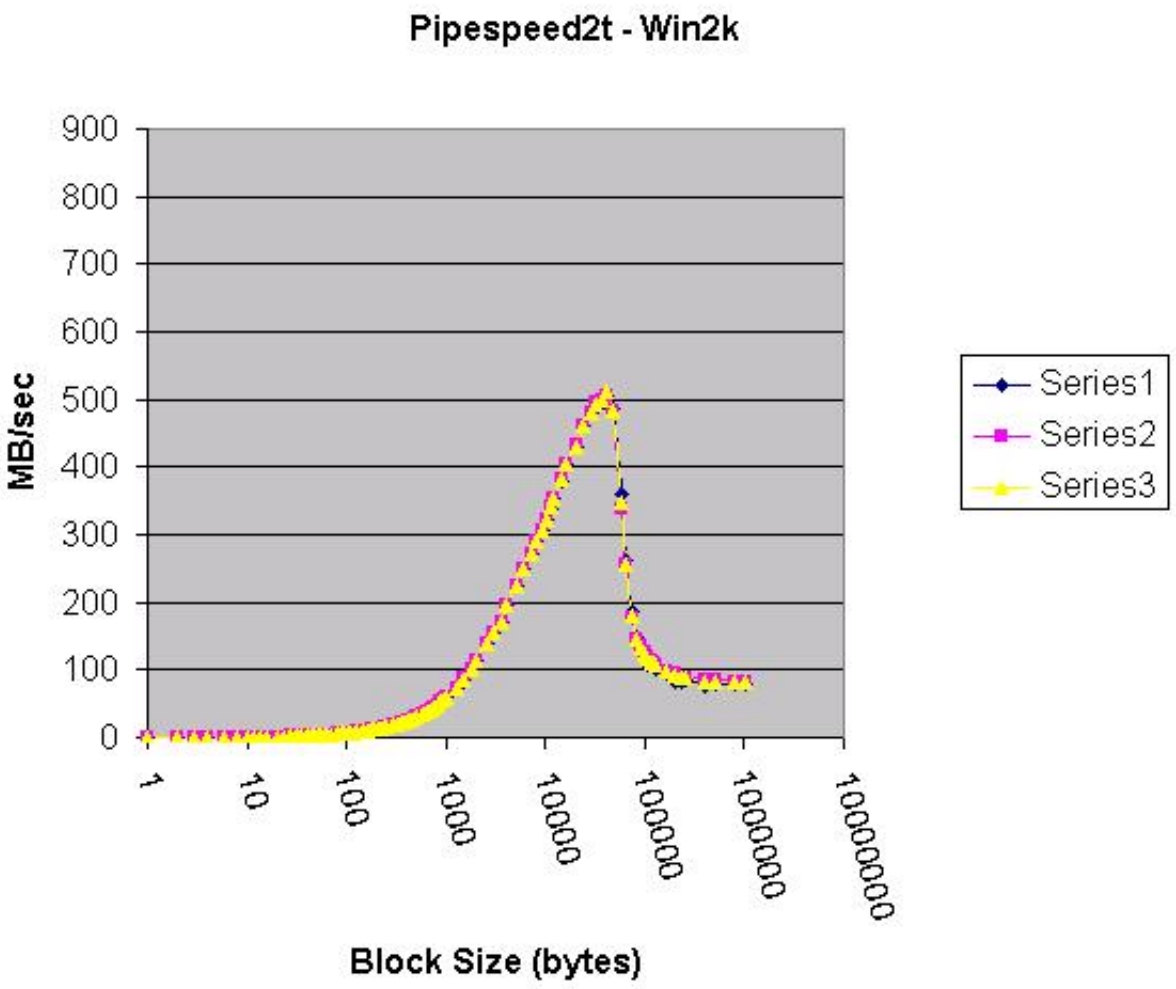

Figure 5

Both graphs show the same shape but Linux reaches a steady state at near 100 MB/sec for very large block sizes. Windows 2000 reaches a steady state for large block sizes also, but it is only at the 80 MB/sec rate.

Figure 6 shows the threaded results for Windows XP Professional (evaluation version). Its peak IO rate is only 120 MB/sec.

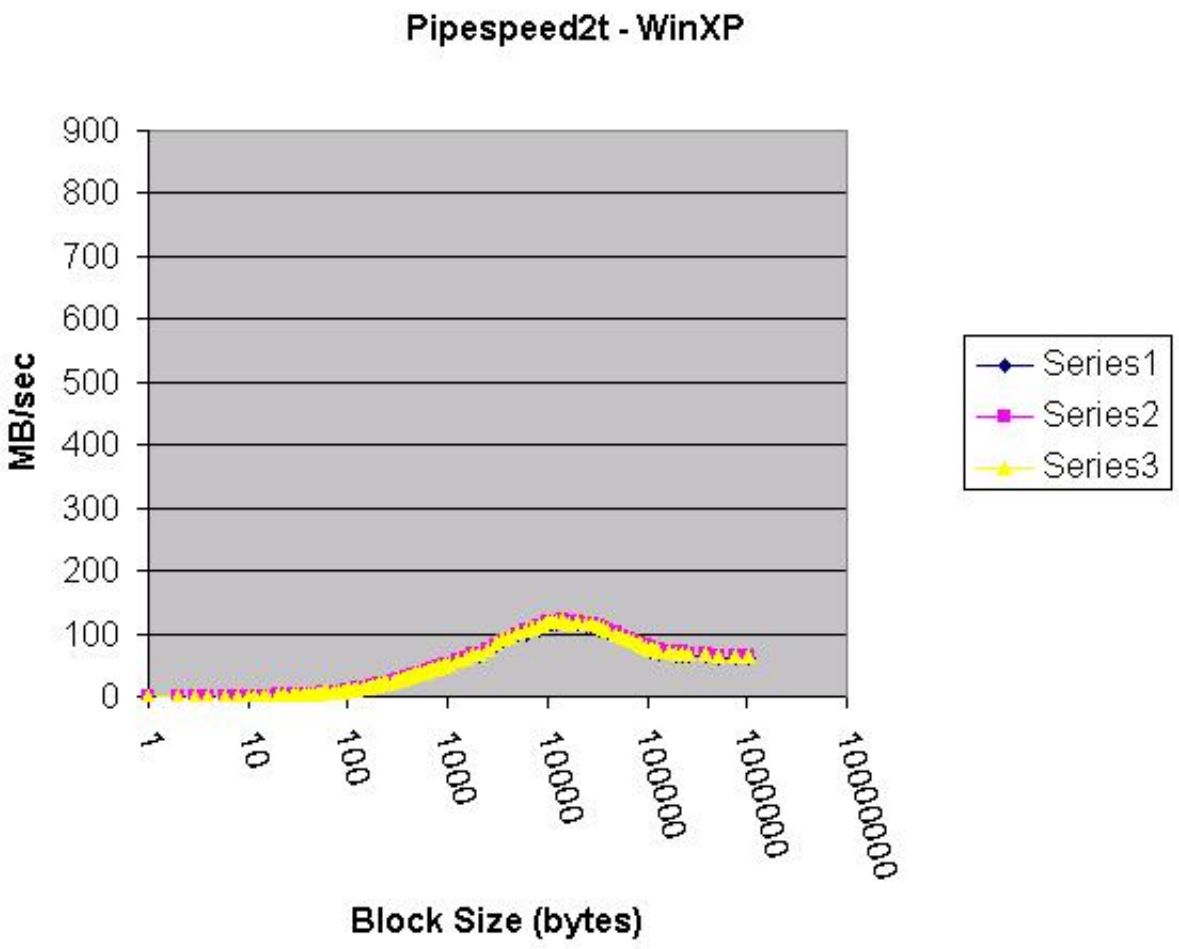

Figure 6

Windows XP shows disappointing results along the full range of block sizes. Perhaps one of the Microsoft experts in our discussion forum will add a useful comment on how to improve our program and the performance of Windows XP named pipes.

Information easily generated by simple programs can inform program designers, software architects, and even system administrators about the features of an operating system. Simplicity is required to avoid endless debate over "realistic" scenarios. "Simplicity" should represent the way people write programs. In this case I have done nothing special to improve performance on any of the platforms.

The poor performance of Windows XP is perplexing. One possible explanation might lie in the existence of a better solution for transferring data -- sockets. I will be looking at sockets in a future column.

Linux also supports named pipes. An early reviewer of these numbers suggested I compare Linux named pipes with Windows named pipes to be fair. I wrote another program that used named pipes on Linux. I found the results for named and unnamed pipes on Linux indistinguishable.

Another distinction might the the "feature" of Windows pipes where there is no fixed buffer size. For the first test we stopped at a 4K buffer size in deference to the Linux buffer. Windows advocates might suggest that the arbitrary buffer sizes associated with Windows named pipes are a benefit. To demonstrate the arbitrary size of the Windows named pipe buffers, we can simply run the single threaded program with arbitrarily large block sizes. I did a run with pipespeed2.cpp on Windows and specified a 256 MB buffer size. Windows obliged by swelling the buffer size to hold 256 MB of data *before* the `ReadFile()` was issued. The system slowed to a crawl and I didn't wait until the operation completed. Whether this "feature" of Windows is useful or not is up

to the public. However, the swelling and contracting of internal buffers requires page allocations and deallocations -- operations not generally associated with simple streams of bytes.

Conclusion

We wrote two programs to demonstrate good programming practices when using pipe facilities on Windows and Linux. The first program `pipespeed2.cpp` demonstrated the performance of the pipe facility code path using a single thread to pass data into and out of the operating system. The second program `pipespeed2t.cpp` used two threads to pass arbitrarily large amounts of data. The second program is a more realistic representation of the normal usage of pipe facilities.

Our results showed that Linux pipes are considerably faster than Windows 2000 named pipes, and Windows 2000 named pipes are much faster than Windows XP named pipes.

Resources

- Participate in the discussion forum on this article by clicking **Discuss** at the top or bottom of the article.
- View listings of programs mentioned in this article:
  - pipespeed2.cpp
  - pipespeed2t.cpp
  - pipespeed2-sh.sh
  - pipespeed2t-sh.sh
  - pipespeed2.xls
- Read Ed's previous RunTime columns on *developerWorks*:
  - Introductory column
  - Block memory copy
  - Block memory copy, Part 2
- Read these related articles on *developerWorks*:
  - Operating system flexibility
  - Linux, the server operating system
- Browse more Linux resources on *developerWorks*
- Browse more Open source resources on *developerWorks*.

About the author

Ed currently manages Microsoft Premier Support for IBM Software group and writes a weekly newsletter for Linux and Windows 2000 Software Developers. Ed can be reached at egb@us.ibm.com.