wait. The defect, which could cause a thread to block indefinitely, did not show up in over a year of use; eventually, however, something would have failed. Simple code inspection and knowledge of the condition variable model revealed the error.

Debugging is also problematic because debuggers change timing behavior, masking the very race conditions that you wish to expose. For example, debugging is unlikely to find a problem with an incorrect choice of event type (auto-reset or manual-reset) and `SetEvent`/`PulseEvent`. You have to think carefully about what you wish to achieve.

Having said all that, testing on a wide variety of platforms, which must include multiprocessor systems, is an essential part of any multithreaded software development project.

Avoiding Incorrect Code

Every bug you don't put in your code in the first place is one more bug you won't find in testing or production. Here are some hints, most of which are taken, although rephrased, from Butenhof's *Programming with POSIX Threads (PWPT)*.

- **Avoid relying on thread inertia.** Threads are asynchronous, but we often assume, for example, that a parent thread will continue running after creating one or more child threads. The assumption is that the parent's "inertia" will keep it running before the children run. This assumption is especially dangerous on a multiprocessor system, but it can also lead to problems on single-processor systems.
- **Never bet on a thread race.** Nearly anything can happen in terms of thread scheduling. Your program has to assume that any ready thread can start running at any time and that any running thread can be preempted at any time. "No ordering exists between threads unless you cause ordering" (*PWPT*, p. 294).
- **Scheduling is not the same as synchronization.** Scheduling policy and priorities cannot ensure proper synchronization. Use synchronization objects instead.
- **Sequence races can occur even when you use locks to protect shared data.** Just because data is protected, there is no assurance as to the order in which different threads will access the shared data. For example, if one thread adds money to a bank account and another makes a withdrawal, there is no assurance, using a lock alone, that the deposit will be made before the withdrawal. Exercise 10–14 shows how to control thread execution order.
- **Cooperate to avoid deadlocks.** You need a well-understood lock hierarchy, used by all threads, to ensure that deadlocks will not occur.

- **Never share events between predicates.** Each event used in a condition variable implementation should be associated with a distinct predicate. Furthermore, an event should always be used with the same mutex.
- **Beware of sharing stacks and related memory corrupters.** Always remember that when you return from a function or when a thread terminates, memory that is local to the function or thread is no longer valid. Memory on a thread's stack can be used by other threads, but you have to be sure that the first thread continues to exist. This behavior is not unique to thread functions, of course.
- **Be sure to use the `volatile` storage modifier.** Whenever a shared variable can be changed in one thread and accessed in another, the variable should be `volatile` to ensure that each thread stores and fetches the variable to and from memory, rather than assuming that the variable is held in a register that is specific to the thread. However, do not overuse `volatile`; any function call or return will assure that registers are stored; furthermore, every synchronization call will erect a memory barrier.
- **Use memory barriers so that processors have coherent memory views** (see Chapter 8 and Figure 8–2). `volatile` is not sufficient. Memory barriers assure that memory accesses issued by the processors are visible in a particular order.

Here are some additional guidelines and rules of thumb that can be helpful.

- **Use the condition variable model properly,** being certain not to use two distinct locks with the same event. Understand the condition variable model on which you depend. Be certain that the invariant holds before waiting on a condition variable.
- **Understand your invariants and condition variable predicates,** even if they are stated only informally. Be certain that the invariant always holds outside the critical code section.
- **Keep it simple.** Multithreaded programming is complex enough without the burden of additional complex, poorly understood thread models and logic. If a program becomes overly complex, assess whether the complexity is really necessary or is the result of poor design. Careful use of standard threading models can simplify your program and make it easier to understand, and lack of a good model may be a symptom of a poorly designed program.
- **Test on both single-processor and multiprocessor systems and on systems with different clock rates, cache architectures, and other characteristics.** Some defects will never, or rarely, show up on a single-

processor system but will occur immediately on a multiprocessor system, and conversely. Likewise, a variety of system characteristics helps ensure that a defective program has more opportunity to fail.

- **Testing is necessary but not sufficient to ensure correct behavior.** There have been a number of examples of programs, known to be defective, that seldom fail in routine or even extensive tests.
- **Be humble.** After all these precautions, bugs will still occur. This is true even with single-threaded programs; threads simply give us more, different, and very interesting ways to cause problems. This adage has been proved many times in preparing this book, where several reviewers and I found bugs (not always subtle bugs, either) in the example programs.

Beyond the Windows API

We have intentionally limited coverage to the Windows API. Microsoft does, however, provide additional access to kernel objects, such as threads. For example, the .NET `ThreadPool` class, accessible through C++, C#, and other languages, allows you to create a pool of threads and to queue work items to the threads (the `ThreadPool` method is `QueueUserWorkItem`).

Microsoft also implements the Microsoft Message Queuing (MSMQ) service, which provides messaging services between networked systems. The examples in this chapter should help show the value of a general-purpose message queuing system. MSMQ is documented in MSDN.

Summary

Multithreaded program development is much simpler if you use well-understood and familiar programming models and techniques. This chapter has shown the utility of the condition variable model and has solved several relatively complex but important programming problems. APCs allow one thread to signal and cause actions in another thread, which allows thread cancellation so that all threads in a system can shut down properly.

Synchronization and thread management are complex because there are multiple ways to solve a given problem, and the different techniques involve complexity and performance trade-offs. The three-stage pipeline example was implemented several different ways in order to illustrate the options.

Use of careful program design and implementation is the best way to improve program quality. Overdependence on testing and debugging, without attention to detail, can lead to serious problems that may be very difficult to detect and fix.

Looking Ahead

Chapter 11 introduces interprocess communication using Windows proprietary anonymous and named pipes. The named pipe example programs show a multithreaded server that can process requests from multiple networked clients. Chapter 12 then converts the example to sockets, which are an industry standard and allow interoperability with Linux, UNIX, and other systems.

Additional Reading

David Butenhof's *Programming with POSIX Threads* was the source of much of the information and programming guidelines at the end of the chapter. The threshold barrier solution, Programs 10–1 and 10–2, was adapted from Butenhof as well.

"Strategies for Implementing POSIX Condition Variables in Win32," by Douglas Schmidt and Irfan Pyarali (posted at http://www.cs.wustl.edu/~schmidt/win32-cv-1.html), discusses Windows event limitations along with condition variables emulation, thoroughly analyzing and evaluating several approaches. Reading this paper will increase your appreciation of the new functions. Another paper by the same authors (http://www.cs.wustl.edu/~schmidt/win32-cv-2.html) builds object-oriented wrappers around Windows synchronization objects to achieve a platform-independent synchronization interface. The open source Pthreads implementation, which is based on the Schmidt and Pyarali work, is available at http://sources.redhat.com/pthreads-win32/.

Exercises

10–1. Revise Program 10–1 so that it does not use the `SignalObjectAndWait` function.

10–2. Modify `eventPC` (Program 8–2) so that there can be multiple consumers and so that it uses the condition variable model. Which event type is appropriate?

10–3. Change the logic in Program 10–2 so that the event is signaled only once.

10–4. Replace the mutex in the queue object used in Program 10–2 with a CS. What are the effects on performance and throughput? The solution is in the *Examples* file, and Appendix C contains experimental data.

10–5. Program 10–4 uses the broadcast CV model to indicate when the queue is either not empty or not full. Would the signal CV model work? Would the signal model even be preferable in any way? Appendix C contains experimental data.

10–6. Experiment with the queue lengths and the transmitter-receiver blocking factor in Program 10–5 to determine the effects on performance, throughput, and CPU load.

10–7. *For C++ programmers*: The code in Programs 10–3 and 10–4 could be used to create a synchronized queue class in C++; create this class and modify Program 10–5 to test it. Which of the functions should be public and which should be private?

10–8. Study the performance behavior of Program 10–5 if `CRITICAL_SECTION`s are used instead of mutexes.

10–9. Improve Program 10–5 so that it is not necessary to terminate the transmitter and receiver threads. The threads should shut themselves down.

10–10. The *Examples* file contains `MultiSem.c`, which implements a multiple-wait semaphore modeled after the Windows objects (they can be named, secured, and process shared, and there are two wait models), and `TestMultiSem.c` is a test program. Build and test this program. How does it use the CV model? Is performance improved by using a `CRITICAL_SECTION` or Windows condition variable? What are the invariants and condition variable predicates?

10–11. Illustrate the various guidelines at the end of this chapter in terms of bugs you have encountered or in the defective versions of the programs provided in the *Examples* file.

10–12. Read "Strategies for Implementing POSIX Condition Variables in Win32" by Schmidt and Pyarali (see the Additional Reading section). Apply their fairness, correctness, serialization, and other analyses to the CV models (called "idioms" in their paper) in this chapter. Notice that this chapter does not directly emulate condition variables; rather, it tackles the easier problem of emulating normal condition variable usage, whereas Schmidt and Pyarali emulate condition variables used in an arbitrary context.

10–13. Convert one of Chapter 9's `statsXX` programs to create a thread pool using APCs.

10–14. Two projects in the *Examples* file, `batons` and `batonsMultipleEvents`, show alternative solutions to the problem of serializing thread execution. The code comments give background and acknowledgments. The second solution associates a unique event with each thread so that specific threads can be signaled. The implementation uses C++ in order to take advantage of the C++ Standard Template Library (STL). Compare and contrast these two solutions, and use the second as a means to become familiar with the STL.

10–15. Perform tests to compare NT6 condition variable performance with the other `ThreeStage` implementations.

10–16. Modify `QueueObjectCV.c` (which implements the message queue management functions with condition variables) so that it uses SRW (slim reader/writer) locks. Test with `ThreeStage.c` and compare performance with the original implementation. Further modify the implementation to use thread pooling.

CHAPTER

11 Interprocess Communication

Chapter 6 showed how to create and manage processes, and Chapters 7 to 10 showed how to manage and synchronize threads within processes. So far, however, we have not been able to perform direct process-to-process communication other than through shared memory (Chapter 5).

The next step is to provide sequential interprocess communication (IPC) between processes[1] using filelike objects. Two primary Windows mechanisms for IPC are the anonymous pipe and the named pipe, both of which are accessed with the familiar `ReadFile` and `WriteFile` functions. Simple anonymous pipes are character-based and half-duplex. As such, they are well suited for redirecting the output of one program to the input of another, as is common with communicating Linux and UNIX programs. The first example shows how to do this with Windows anonymous pipes.

Named pipes are much more powerful than anonymous pipes. They are full-duplex and message-oriented, and they allow networked communication. Furthermore, there can be multiple open handles on the same pipe. These capabilities, coupled with convenient transaction-oriented named pipe functions, make named pipes appropriate for creating client/server systems. This capability is shown in this chapter's second example, a multithreaded client/server command processor, modeled after Figure 7–1, which was used to introduce threads. Each server thread manages communication with a different client, and each thread/client pair uses a distinct handle, or named pipe instance. Mailslots, which allow for one-to-many message broadcasting and are also filelike, are used to help clients locate servers.

[1] The Windows system services also allow processes to communicate through mapped files, as demonstrated in the semaphore exercise in Chapter 10 (Exercise 10–10). Additional mechanisms for IPC include files, sockets, remote procedure calls, COM, and message posting. Chapter 12 describes sockets.

Anonymous Pipes

Windows anonymous pipes allow one-way (half-duplex), byte-based IPC. Each pipe has two handles: a read handle and a write handle. The `CreatePipe` function is:

```
BOOL CreatePipe (
   PHANDLE phRead,
   PHANDLE phWrite,
   LPSECURITY_ATTRIBUTES lpsa,
   DWORD cbPipe)
```

The pipe handles are often inheritable; the next example shows the reasons. `cbPipe`, the pipe byte size, is only a suggestion, and `0` specifies the default value.

In order to use the pipe for IPC, there must be another process, and that process requires one of the pipe handles. Assume that the parent process, which calls `CreatePipe`, wishes to write data for a child to use. The problem, then, is to communicate the read handle (`phRead`) to the child. The parent achieves this by setting the child procedure's input handle in the start-up structure to `*phRead` (see Chapter 6 for process management and the start-up structure).

Reading a pipe read handle will block if the pipe is empty. Otherwise, the read will accept as many bytes as are in the pipe, up to the number specified in the `ReadFile` call. A write operation to a full pipe, which is implemented in a memory buffer, will also block.

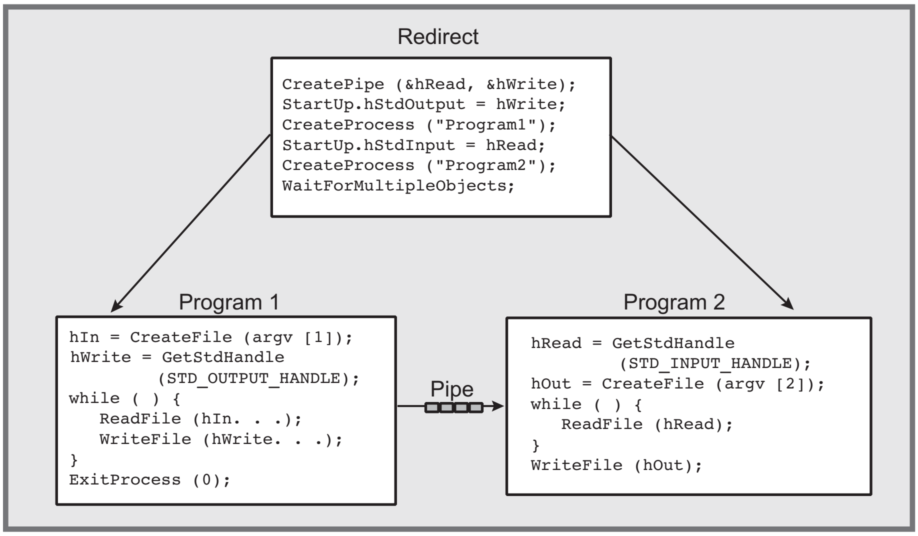
Finally, anonymous pipes are one-way. Two pipes are required for bidirectional communication.

Example: I/O Redirection Using an Anonymous Pipe

Program 11–1 shows a parent process, `Redirect`, that creates two processes from the command line and pipes them together. The parent process sets up the pipe and redirects standard input and output. Notice how the anonymous pipe handles are inheritable and how standard I/O is redirected to the two child processes; these techniques were described in Chapter 6.

The location of `WriteFile` in `Program2` on the right side of Figure 11–1 assumes that the program reads a large amount of data, processes it, and then writes out results. Alternatively, the write could be inside the loop, putting out results after each read.

Close the pipe and thread handles at the earliest possible point. Figure 11–1 does not show the handle closings, but Program 11–1 does. The parent should



```
CreatePipe (&hRead, &hWrite);
StartUp.hStdOutput = hWrite;
CreateProcess ("Program1");
StartUp.hStdInput = hRead;
CreateProcess ("Program2");
WaitForMultipleObjects;
```

```
hIn = CreateFile (argv [1]);
hWrite = GetStdHandle
         (STD_OUTPUT_HANDLE);
while ( ) {
   ReadFile (hIn. . .);
   WriteFile (hWrite. . .);
}
ExitProcess (0);
```

```
hRead = GetStdHandle
         (STD_INPUT_HANDLE);
hOut = CreateFile (argv [2]);
while ( ) {
   ReadFile (hRead);
}
WriteFile (hOut);
```

Figure 11–1 Process-to-Process Communication Using an Anonymous Pipe

close the standard output handle immediately after creating the first child process so that the second process will be able to recognize an end of file when the first process terminates. If there were still an open handle, the second process might not terminate because the application would not indicate an end of file.

Program 11–1 uses an unusual syntax; the = sign is the pipe symbol separating the two commands. The vertical bar (|) would conflict with the command processor. Figure 11–1 schematically shows the execution of the command:

```
$ Redirect Program1 arguments = Program2 arguments
```

In UNIX or at the Windows command prompt, the corresponding command would be:

```
$ Program1 arguments | Program2 arguments
```

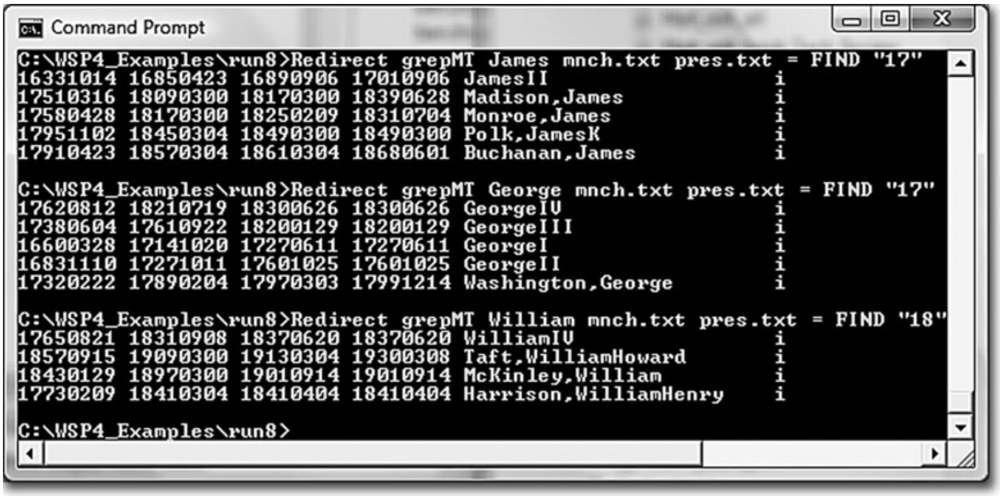
Program 11–1 `Redirect:` Interprocess Communication

```
/* Chapter 11. Redirect: Anonymous Pipe IPC example. */

#include "Everything.h"

int _tmain (int argc, LPTSTR argv[])
```

```
/* Pipe together two programs whose names are on the command line:
      Redirect command1 = command2
   where the two commands are arbitrary strings.
   command1 uses standard input, and command2 uses standard output.
   Use = so as not to conflict with the DOS pipe. */
{
   DWORD i;
   HANDLE hReadPipe, hWritePipe;
   TCHAR command1[MAX_PATH];
   SECURITY_ATTRIBUTES pipeSA =
            {sizeof(SECURITY_ATTRIBUTES), NULL, TRUE};

   /* Initialize for inheritable handles. */
   PROCESS_INFORMATION procInfo1, procInfo2;
   STARTUPINFO startInfoCh1, startInfoCh2;
   LPTSTR targv = GetCommandLine ();

   /* Startup info for the two child processes. */
   GetStartupInfo (&startInfoCh1);
   GetStartupInfo (&startInfoCh2);

   targv = SkipArg (targv);
   i = 0;    /* Get the two commands. */
   while (*targv != _T ('=') && *targv != _T ('\0')) {
      command1[i] = *targv;
      targv++; i++;
   }
   command1[i] = '\0';
   if (*targv == '\0')
      ReportError (_T ("No command separator found."), 2, FALSE);
   targv = SkipArg (targv);

   /* Create anonymous pipe - default size & inheritable handles. */
   CreatePipe (&hReadPipe, &hWritePipe, &pipeSA, 0);

   /* Set the output handle to the inheritable pipe handle,
      and create the first processes. */
   startInfoCh1.hStdInput  = GetStdHandle (STD_INPUT_HANDLE);
   startInfoCh1.hStdError  = GetStdHandle (STD_ERROR_HANDLE);
   startInfoCh1.hStdOutput = hWritePipe;
   startInfoCh1.dwFlags = STARTF_USESTDHANDLES;

   CreateProcess (NULL, command1, NULL, NULL,
         TRUE,     /* Inherit handles. */
         0, NULL, NULL, &startInfoCh1, &procInfo1);

   CloseHandle (procInfo1.hThread); CloseHandle (hWritePipe);

   /* Repeat (symmetrically) for the second process. */
   startInfoCh2.hStdInput  = hReadPipe;
   startInfoCh2.hStdError  = GetStdHandle (STD_ERROR_HANDLE);
   startInfoCh2.hStdOutput = GetStdHandle (STD_OUTPUT_HANDLE);
   startInfoCh2.dwFlags = STARTF_USESTDHANDLES;

   CreateProcess (NULL, targv, NULL, NULL,
         TRUE,     /* Inherit handles. */
         0, NULL, NULL, &startInfoCh2, &procInfo2);
   CloseHandle (procInfo2.hThread);
   CloseHandle (hReadPipe);

   /* Wait for both processes to complete.
      The first one should finish first, but it doesn't matter. */
   WaitForSingleObject (procInfo1.hProcess, INFINITE);
   WaitForSingleObject (procInfo2.hProcess, INFINITE);
   CloseHandle (procInfo1.hProcess);
   CloseHandle (procInfo2.hProcess);
   return 0;
}
```

Run 11–1 shows output from `grepMT`, Chapter 7's multithreaded pattern search program piped to `FIND`, which is a similar Windows command. While this may seem a bit artificial, `cat` is the book's only sample program that accepts standard input, and it also shows that `Redirect` works with third-party programs that accept standard input.

These examples search the presidents and monarchs files, first used in Chapter 6, for individuals named "James" and "George" who lived in any part of the eighteenth century (the search is not entirely accurate) or "William" who lived in any part of the nineteenth century. The file names were shortened to decrease the horizontal space.



```
C:\WSP4_Examples\run8>Redirect grepMT James mnch.txt pres.txt = FIND "17"
16331014 16850423 16890906 17010906 JamesII                   i
17510316 18090300 18170300 18390628 Madison,James             i
17580428 18170300 18250209 18310704 Monroe,James              i
17951102 18450304 18490300 18490300 Polk,JamesK               i
17910423 18570304 18610304 18680601 Buchanan,James            i

C:\WSP4_Examples\run8>Redirect grepMT George mnch.txt pres.txt = FIND "17"
17620812 18210719 18300626 18300626 GeorgeIV                  i
17380604 17610922 18200129 18200129 GeorgeIII                 i
16600328 17141020 17270611 17270611 GeorgeI                   i
16831110 17271011 17601025 17601025 GeorgeII                  i
17320222 17890204 17970303 17991214 Washington,George         i

C:\WSP4_Examples\run8>Redirect grepMT William mnch.txt pres.txt = FIND "18"
17650821 18310908 18370620 18370620 WilliamIV                 i
18570915 19090300 19130304 19300308 Taft,WilliamHoward        i
18430129 18970300 19010914 19010914 McKinley,William          i
17730209 18410304 18410404 18410404 Harrison,WilliamHenry     i

C:\WSP4_Examples\run8>
```

Run 11–1 `Redirect:` Using an Anonymous Pipe

Named Pipes

Named pipes have several features that make them an appropriate general-purpose mechanism for implementing IPC-based applications, including networked file access and client/server systems,[2] although anonymous pipes remain a good choice for simple byte-stream IPC, such as the preceding example, where communication is within a single computer. Named pipe features (some are optional) include the following.

- Named pipes are message-oriented, so the reading process can read varying-length messages precisely as sent by the writing process.
- Named pipes are bidirectional, so two processes can exchange messages over the same pipe.
- There can be multiple, independent instances of pipes with the same name. For example, several clients can communicate concurrently with a single server using distinct instances of a named pipe. Each client can have its own named pipe instance, and the server can respond to a client using the same instance.
- Networked clients can access the pipe by name. Named pipe communication is the same whether the two processes are on the same machine or on different machines.
- Several convenience and connection functions simplify named pipe request/response interaction and client/server connection.

Named pipes are generally preferable to anonymous pipes, although Program 11–1 and Figure 11–1 did illustrate a situation in which anonymous pipes are useful. Use named pipes any time your communication channel needs to be bidirectional, message-oriented, networked, or available to multiple client processes. The upcoming examples could not be implemented using anonymous pipes.

Using Named Pipes

`CreateNamedPipe` creates the first instance of a named pipe and returns a handle. The function also specifies the pipe's maximum number of instances and, hence, the number of clients that can be supported simultaneously.

[2] This statement requires a major qualification. Windows Sockets (Chapter 12) is the preferred API for most networking applications and higher-level protocols (http, ftp, and so on), especially where TCP/IP-based interoperability with non-Windows systems is required. Many developers prefer to limit named pipe usage to IPC within a single computer or to communication within Windows networks.