### The Server

The server will listen to port 6052 waiting for client connections. When a client connection is made, the server will service the connection in a separate thread and will resume listening for additional client connections. Once a client makes a connection to the server, the client will write the IP name it wishes the server to resolve—such as `www.westminstercollege.edu`—to the socket. The server thread will read this IP name from the socket and either resolve its IP address or, if it cannot locate the host address, catch an `UnknownHostException`. The server will write the IP address back to the client or, in the case of an `UnknownHostException`, will write the message "`Unable to resolve host <host name>`." Once the server has written to the client, it will close its socket connection.

### The Client

Initially, write just the server application and connect to it via telnet. For example, assuming the server is running on the localhost, a telnet session would appear as follows. (Client responses appear in blue.)

```
telnet localhost 6052
Connected to localhost.
Escape character is '^]'.
www.westminstercollege.edu
146.86.1.17
Connection closed by foreign host.
```

By initially having telnet act as a client, you can more easily debug any problems you may have with your server. Once you are convinced your server is working properly, you can write a client application. The client will be passed the IP name that is to be resolved as a parameter. The client will open a socket connection to the server and then write the IP name that is to be resolved. It will then read the response sent back by the server. As an example, if the client is named `NSClient`, it is invoked as follows:

```
java NSClient www.westminstercollege.edu
```

and the server will respond with the corresponding IP address or "unknown host" message. Once the client has output the IP address, it will close its socket connection.

## Project 2: Matrix Multiplication Project

Given two matrices, $A$ and $B$, where matrix $A$ contains $M$ rows and $K$ columns and matrix $B$ contains $K$ rows and $N$ columns, the matrix product of $A$ and $B$ is matrix $C$, where $C$ contains $M$ rows and $N$ columns. The entry in matrix $C$ for row $i$, column $j$ ($C_{i,j}$) is the sum of the products of the elements for row $i$ in matrix $A$ and column $j$ in matrix $B$. That is,

$$C_{i,j} = \sum_{n=1}^{K} A_{i,n} \times B_{n,j}$$

For example, if $A$ is a 3-by-2 matrix and $B$ is a 2-by-3 matrix, element $C_{3,1}$ is the sum of $A_{3,1} \times B_{1,1}$ and $A_{3,2} \times B_{2,1}$.

For this project, calculate each element $C_{i,j}$ in a separate *worker* thread. This will involve creating $M \times N$ worker threads. The main—or parent—thread will initialize the matrices $A$ and $B$ and allocate sufficient memory for matrix $C$, which will hold the product of matrices $A$ and $B$. These matrices will be declared as global data so that each worker thread has access to $A$, $B$, and $C$.

Matrices $A$ and $B$ can be initialized statically, as shown below:

```
#define M 3
#define K 2
#define N 3

int A [M][K] = { {1,4}, {2,5}, {3,6} };
int B [K][N] = { {8,7,6}, {5,4,3} };
int C [M] [N];
```

Alternatively, they can be populated by reading in values from a file.

### Passing Parameters to Each Thread

The parent thread will create $M \times N$ worker threads, passing each worker the values of row $i$ and column $j$ that it is to use in calculating the matrix product. This requires passing two parameters to each thread. The easiest approach with Pthreads and Win32 is to create a data structure using a `struct`. The members of this structure are $i$ and $j$, and the structure appears as follows:

```
/* structure for passing data to threads */
struct v
{
   int i; /* row */
   int j; /* column */
};
```

Both the Pthreads and Win32 programs will create the worker threads using a strategy similar to that shown below:

```
/* We have to create M * N worker threads */
for (i = 0; i < M, i++)
   for (j = 0; j < N; j++ ) {
      struct v *data = (struct v *) malloc(sizeof(struct v));
      data->i = i;
      data->j = j;
      /* Now create the thread passing it data as a parameter */
   }
}
```

```
public class WorkerThread implements Runnable
{
    private int row;
    private int col;
    private int[][] A;
    private int[][] B;
    private int[][] C;

   public WorkerThread(int row, int col, int[][] A,
    int[][] B, int[][] C) {
      this.row = row;
      this.col = col;
      this.A = A;
      this.B = B;
      this.C = C;
   }
   public void run() {
      /* calculate the matrix product in C[row] [col] */
   }
}
```

**Figure 4.15** Worker thread in Java.

The data pointer will be passed to either the `pthread_create()` (Pthreads) function or the `CreateThread()` (Win32) function, which in turn will pass it as a parameter to the function that is to run as a separate thread.

Sharing of data between Java threads is different from sharing between threads in Pthreads or Win32. One approach is for the main thread to create and initialize the matrices *A*, *B*, and *C*. This main thread will then create the worker threads, passing the three matrices—along with row *i* and column *j*—to the constructor for each worker. Thus, the outline of a worker thread appears in Figure 4.15.

### Waiting for Threads to Complete

Once all worker threads have completed, the main thread will output the product contained in matrix *C*. This requires the main thread to wait for all worker threads to finish before it can output the value of the matrix product. Several different strategies can be used to enable a thread to wait for other threads to finish. Section 4.3 describes how to wait for a child thread to complete using the Win32, Pthreads, and Java thread libraries. Win32 provides the `WaitForSingleObject()` function, whereas Pthreads and Java use `pthread_join()` and `join()`, respectively. However, in these programming examples, the parent thread waits for a single child thread to finish; completing this exercise will require waiting for multiple threads.

In Section 4.3.2, we describe the `WaitForSingleObject()` function, which is used to wait for a single thread to finish. However, the Win32 API also provides the `WaitForMultipleObjects()` function, which is used when waiting for multiple threads to complete. `WaitForMultipleObjects()` is passed four parameters:

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

**Figure 4.16** Pthread code for joining ten threads.

1. The number of objects to wait for
2. A pointer to the array of objects
3. A flag indicating if all objects have been signaled
4. A timeout duration (or `INFINITE`)

For example, if `THandles` is an array of thread `HANDLE` objects of size `N`, the parent thread can wait for all its child threads to complete with the statement:

```
WaitForMultipleObjects(N, THandles, TRUE, INFINITE);
```

A simple strategy for waiting on several threads using the Pthreads `pthread_join()` or Java's `join()` is to enclose the join operation within a simple `for` loop. For example, you could join on ten threads using the Pthread code depicted in Figure 4.16. The equivalent code using Java threads is shown in Figure 4.17.

```
final static int NUM_THREADS = 10;

/* an array of threads to be joined upon */
Thread[] workers = new Thread[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++) {
   try {
      workers[i].join();
    } catch (InterruptedException ie) { }
}
```

**Figure 4.17** Java code for joining ten threads.

## Bibliographical Notes

Threads have had a long evolution, starting as "cheap concurrency" in programming languages and moving to "lightweight processes", with early examples that included the Thoth system (Cheriton et al. [1979]) and the Pilot system (Redell et al. [1980]). Binding [1985] described moving threads into the UNIX kernel. Mach (Accetta et al. [1986], Tevanian et al. [1987a]) and V (Cheriton [1988]) made extensive use of threads, and eventually almost all major operating systems implemented them in some form or another.

Thread performance issues were discussed by Anderson et al. [1989], who continued their work in Anderson et al. [1991] by evaluating the performance of user-level threads with kernel support. Bershad et al. [1990] describe combining threads with RPC. Engelschall [2000] discusses a technique for supporting user-level threads. An analysis of an optimal thread-pool size can be found in Ling et al. [2000]. Scheduler activations were first presented in Anderson et al. [1991], and Williams [2002] discusses scheduler activations in the NetBSD system. Other mechanisms by which the user-level thread library and the kernel cooperate with each other are discussed in Marsh et al. [1991], Govindan and Anderson [1991], Draves et al. [1991], and Black [1990]. Zabatta and Young [1998] compare Windows NT and Solaris threads on a symmetric multiprocessor. Pinilla and Gill [2003] compare Java thread performance on Linux, Windows, and Solaris.

Vahalia [1996] covers threading in several versions of UNIX. McDougall and Mauro [2007] describe recent developments in threading the Solaris kernel. Russinovich and Solomon [2005] discuss threading in the Windows operating system family. Bovet and Cesati [2006] and Love [2004] explain how Linux handles threading and Singh [2007] covers threads in Mac OS X.

Information on Pthreads programming is given in Lewis and Berg [1998] and Butenhof [1997]. Oaks and Wong [1999], Lewis and Berg [2000], and Holub [2000] discuss multithreading in Java. Goetz et al. [2006] present a detailed discussion of concurrent programming in Java. Beveridge and Wiener [1997] and Cohen and Woodring [1997] describe multithreading using Win32.