In this chapter:

- *Suitable Tasks for Threading*
- *Models*
- *Buffering Data Between Threads*
- *Some Common Problems*
- *Performance*
- *Example: An ATM Server*
- *Example: A Matrix Multiplication Program*

2

Designing Threaded Programs

So far you've seen only a couple of Pthreads calls, but you know enough about the programming model to start considering real design issues. In this chapter we'll examine a number of broad questions. How much work is worth the overhead of creating a thread? How can I subdivide my program into threads? What relationship do the threads have to the functions in my program?

To give us a sample application worth threading, we'll introduce an application that will take us through most of the book: a server for automatic teller machines (ATMs). We'll try out our design ideas on this server.

Suitable Tasks for Threading

To find the places in your program where you can use threads, you essentially must look for the properties we identified in Chapter 1, *Why Threads?*: potential parallelism, overlapping I/O, asynchronous events, and real-time scheduling. Whenever a task has one or more of these properties, consider running it in a thread. You can identify a task that is suitable for threading by applying to it the following criteria:

- It is independent of other tasks.

  Does the task use separate resources from other tasks? Does its execution depend on the results of other tasks? Do other tasks depend on its results? We want to maximize concurrency and minimize the need for synchronization. The more tasks depend on each other and share resources, the more the threads executing them will end up blocked waiting on each other.

- It can become blocked in potentially long waits.

  Can the task spend a long time in a suspended state? A program can typically perform millions of integer operations in the time it would take to perform a single I/O operation. If you dedicate a thread to the I/O task, the rest of the program could accomplish a lot more work in less time.

- It can use a lot of CPU cycles.

  Does the task perform long computations, such as matrix crunching, hashing, or encryption? Time-consuming calculations that are independent of activities elsewhere in the program are good candidates for threading. In a multiprocessing environment, you might let a thread executing on one CPU process a long computation while other threads on other CPUs handle input.

- It must respond to asynchronous events.

  Must the task handle events that occur at random intervals, such as network communications or interrupts from hardware and the operating system? Use threads to encapsulate and synchronize the servicing of these events, apart from the rest of your application.

- Its work has greater or lesser importance than other work in the application.

  Must the task perform its work in a given amount of time? Must it run at specific times or specific time intervals? Is its work more time critical than that of other tasks? Scheduling considerations are often a good reason for threading a program. For instance, a window manager application would assign a high priority thread to user input and a much lower priority thread to memory garbage collection.

Server programs—such as those written for database managers, file servers, or print servers—are ideal applications for threading. They must be continuously responsive to asynchronous events—requests for services coming over communications channels from a number of client programs. Processing these requests typically requires I/O to secondary storage.

Computational and signal-processing applications that will run on multiprocessing systems are another good candidate for threading. They contain many CPU-intensive tasks that can be spread out over a number of available CPUs.

Finally, real-time developers are attracted to threads as a model for servers and multiprocessing applications. Multithreaded applications are more efficient than multiprocess applications. The threads model also allows the developers to set specific scheduling policies for threads. What's more, threads eliminate some of the complexity that comes with asynchronous programming. Threads wait for events whereas a serial program would be interrupted and would jump from context to context.

Models

There are no set rules for threading a program. Every program is different, and when you first try to thread a program, you'll likely discover that it's a matter of trial and error. You may initially dedicate a thread to a particular task only to find that your assumptions about its activity have changed or weren't true in the first place.

Over time a few common models for threaded programs have emerged. These models define how a threaded application delegates its work to its threads and how the threads intercommunicate. Because the Pthreads standard imposes little structure on how programmers use threads, you would do well to start your multithreaded program design with a close look at each model. Although none has been explicitly designed for a specific type of application, you'll find that each model tends to be better suited than the others for certain types. We discuss:

- The boss/worker model
- The peer model
- The pipeline model

Boss/Worker Model

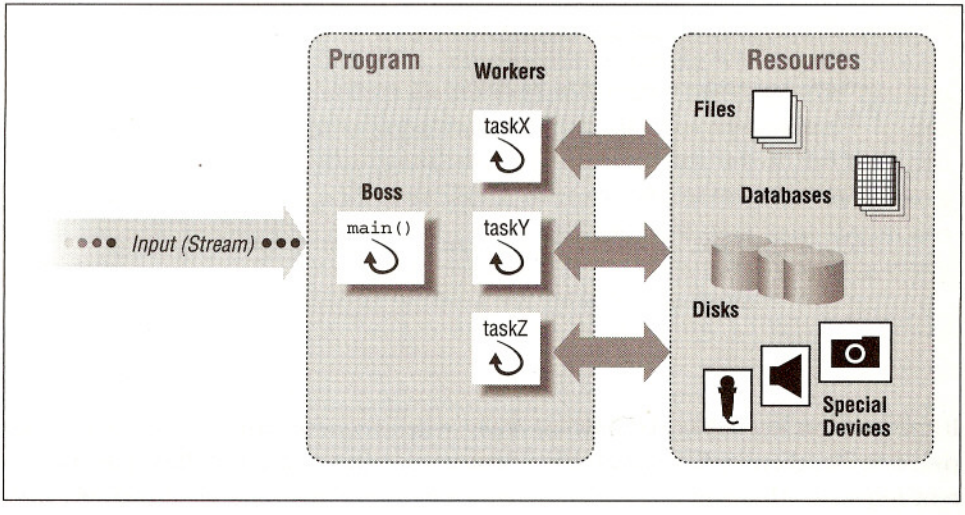

Figure 2–1: The boss/worker model

Figure 2-1 depicts the boss/worker model. A single thread, the *boss*, accepts input for the entire program. Based on that input, the boss passes off specific tasks to one or more *worker* threads.

The boss creates each worker thread, assigns it tasks, and, if necessary, waits for it to finish. In the pseudocode in Example 2-1, the boss dynamically creates a new worker thread when it receives a new request. In the *pthread_create* call it uses to create each worker thread, the boss specifies the task-related routine the thread will execute. After creating each worker, the boss returns to the top of its loop to process the next request. If no requests are waiting, the boss loops until one arrives.

Once finished, each worker thread can be made responsible for any output resulting from its task, or it can synchronize with the boss and let it handle its output.

Example 2-1: Boss/Worker Model Program (Pseudocode)

```
main()

/* The boss */
{
   forever {
            get a request
            switch request
            case X : pthread_create( ... taskX)
            case Y : pthread_create( ... taskY)
            .
            .
            .
   }
}
taskX() /* Workers processing requests of type X */
{
   perform the task, synchronize as needed if accessing shared resources
   done
}

taskY() /* Workers processing requests of type Y */
{
   perform the task, synchronize as needed if accessing shared resources
   done
}
.
.
.
```

If the boss creates its workers dynamically when requests arrive, as it does in our pseudocode, there will be as many concurrent worker threads as there are concurrent requests. Alternatively, the boss could save some run-time overhead by creating all worker threads up front. In this variant of the boss/worker model, known as a *thread pool* and shown in Example 2-2, the boss creates all worker threads at program initialization. Each worker immediately suspends itself to wait for a wake-up call from the boss when a request arrives for it to process. The boss advertises work by queuing requests on a list from which workers retrieve them.

Example 2-2: Boss/Worker Model Program with a Thread Pool (Pseudocode)

```
main()

/* The boss */
{
 for the number of workers
        pthread_create( ... pool_base )

 forever {
        get a request
        place request in work queue
        signal sleeping threads that work is available
 }
}
pool_base() /* All workers */
{
 forever {
        sleep until awoken by boss
        dequeue a work request
        switch
          case request X: taskX()
          case request Y: taskY()
                .
                .
                .
 }
}
```

The boss/worker model works well with servers (database servers, file servers, window managers, and the like). The complexities of dealing with asynchronously arriving requests and communications are encapsulated in the boss. The specifics of handling requests and processing data are delegated to the workers. In this model, it is important that you minimize the frequency with which the boss and workers communicate. The boss can't spend its time being blocked by its workers and allow new requests to pile up at the inputs. Likewise, you can't create too many interdependencies among the workers. If every request requires every worker to share the same data, all workers will suffer a slowdown.

Peer Model

Unlike the boss/worker model, in which one thread is in charge of work assignments for the other threads, in the peer model, illustrated in Figure 2-2, all threads work concurrently on their tasks without a specific leader.

In the peer model, also known as the *workcrew* model, one thread must create all the other peer threads when the program starts. However, unlike the boss thread in the boss/worker model, this thread subsequently acts as just another peer thread that processes requests, or suspends itself waiting for the other peers to finish.

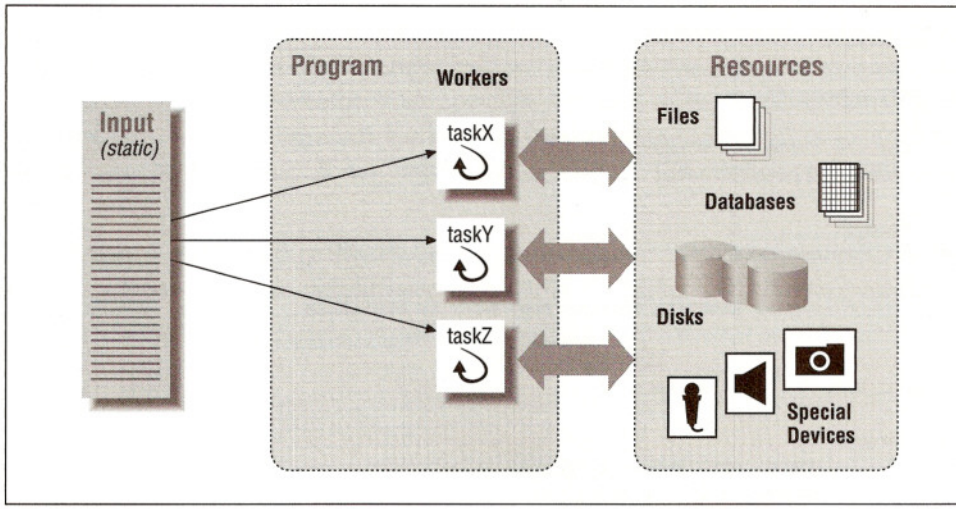

Figure 2–2: The peer model

Whereas the boss/worker model employs a stream of input requests to the boss, the peer model makes each thread responsible for its own input. A peer knows its own input ahead of time, has its own private way of obtaining its input, or shares a single point of input with other peers. The structure of such a program is shown in Example 2-3.

Example 2–3: Peer Model Program (Pseudocode)

```
main()
{
   pthread_create( ... thread1 ... task1 )
   pthread_create( ... thread2 ... task2 )
   .
   .
   .
   signal all workers to start
   wait for all workers to finish
   do any clean up
}

task1()
{
   wait for start
   perform task, synchronize as needed if accessing shared resources
   done
}

task2()
{
   wait for start
```

Example 2-3: Peer Model Program (Pseudocode) (continued)

```
        perform task, synchronize as needed if accessing shared resources
        done
    }
```

The peer model is suitable for applications that have a fixed or well-defined set of inputs, such as matrix multipliers, parallel database search engines, and prime number generators. Well-defined input allows programs to adopt what could be construed as a boss/worker model without the boss. Because there is no boss, peers themselves must synchronize their access to any common sources of input. However, like workers in the boss/worker model, peers can also slow down if they must frequently synchronize to access shared resources.

Consider an application in which a single plane or space is divided among multiple threads, perhaps so they can calculate the spread of a life form (such as in the SimLife computer game) or changes in temperature as heat radiates across geographies from a source. Each thread can calculate one delta of change. However, because the results of each thread's calculations require the adjustment of the bounds of the next thread's calculations, all threads must synchronize afterward to exchange and compare each other's results. This is a classic example of a peer model application.

Pipeline Model

The pipeline model assumes:

- A long stream of input
- A series of suboperations (known as stages or filters) through which every unit of input must be processed
- Each processing stage can handle a different unit of input at a time

An automotive assembly line is a classic example of a pipeline. Each car goes through a series of stages on its way to the exit gates. At any given time many cars are in some stage of completion. A RISC (reduced instruction set computing) processor also fits the pipeline model. The input to this pipeline is a stream of instructions. Each instruction must pass through the stages of decoding, fetching operands, computation, and storing results. That many instructions may be at various stages of processing at the same time contributes to the exceptionally high performance of RISC processors.

In each of these examples, a pipeline improves throughput because it can accomplish the many different stages of a process on different input units (be they cars

or instructions) concurrently. Instead of taking each car or instruction from start to finish before starting the next, a pipeline allows as many cars or instructions to be worked on at the same time as there are stages to process them. It still takes the same amount of time from start to finish for a specific car (that red one, for instance) or instruction to be processed, but the overall throughput of the assembly line or computer chip is greatly increased.

Figure 2-3 shows a thread pipeline.

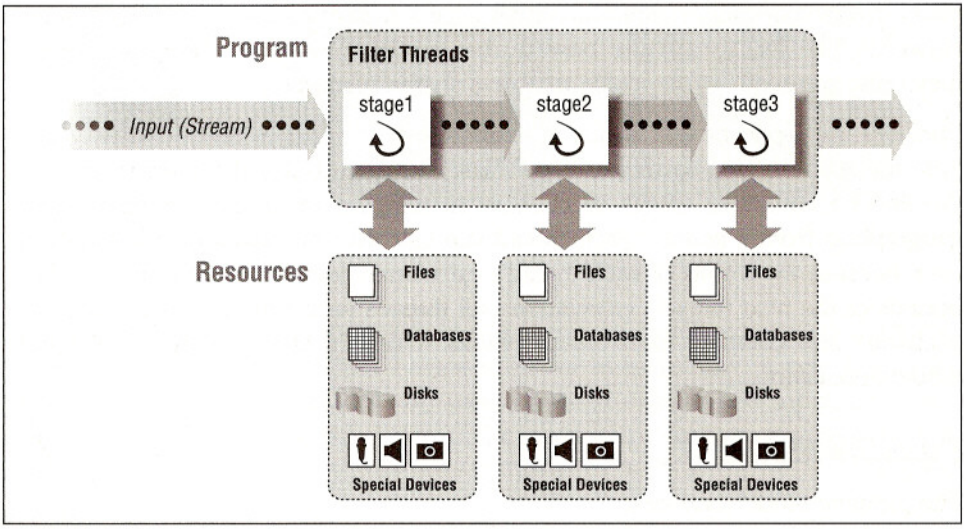

Figure 2-3: A thread pipeline

As the pseudocode in Example 2-4 illustrates, a single thread receives input for the entire program, always passing it to the thread that handles the first stage of processing. Similarly a single thread at the end of the pipeline produces all final output for the program. Each thread in between performs its own stage of processing on the input it received from the thread that performed the previous stage, and passes its output to the thread performing the next. Applications in which the pipeline might be useful are image processing and text processing or any application that can be broken down into a series of filter steps on a stream of input.

Example 2-4: Pipeline Model Program (pseudocode)

```
main()
{
    pthread_create( ... stage1 )
    pthread_create( ... stage2 )
```

Example 2–4: Pipeline Model Program (pseudocode) (continued)

```
        wait for all pipeline threads to finish
        do any clean up
}

stage1()
{
    forever {
            get next input for the program
            do stage 1 processing of the input
            pass result to next thread in pipeline
            }
}

stage2()
{
    forever {
            get input from previous thread in pipeline
            do stage 2 processing of the input
            pass result to next thread in pipeline
            }
}

stageN()
{
    forever {
            get input from previous thread in pipeline
            do stage N processing to the input
            pass result to program output
            }
}
```

We could add multiplexing or demultiplexing to this pipeline, allowing multiple threads to work in parallel on a particular stage. We could also dynamically configure the pipeline at run time, having it create and terminate stages (and the threads to service them) as needed.

Note that the overall throughput of a pipeline is limited by the thread that processes its slowest stage. Threads that follow it in the pipeline cannot perform their stages until it has completed. When designing a multithreaded program according to the pipeline model, you should aim at balancing the work to be performed across all stages; that is, all stages should take about the same amount of time to complete.

Buffering Data Between Threads

The boss/worker, peer, and pipeline are models for complete multithreaded programs. Within any of these models threads transfer data to each other using buffers. In the boss/worker model, the boss must transfer requests to the workers. In the pipeline model, each thread must pass input to the thread that performs the next stage of processing. Even in the peer model, peers may often exchange data.

A thread assumes either of two roles as it exchanges data in a buffer with another thread. The thread that passes the data to another is known as the *producer*; the one that receives that data is known as the *consumer*. Figure 2-4 depicts this relationship.

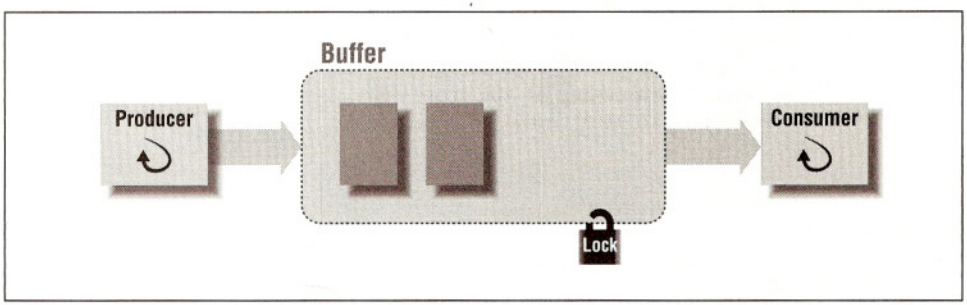

Figure 2-4: Producer-consumer

The ideal producer/consumer relationship requires:

A buffer

The buffer can be any data structure accessible to both the producer and the consumer. This is a simple matter for a multithreaded program, for a such a shared buffer need only be in the process's global data region. The buffer can be just big enough to hold one data item or it can be larger, depending upon the application.

A lock

Because the buffer is shared, the producer and consumer must synchronize their access to it. With Pthreads, you would use a mutex variable as a lock.

A suspend/resume mechanism

The consumer may suspend itself when the buffer contains no data for it to consume. If so, the producer must be able to resume it when it places a new item in the buffer. With Pthreads, you would arrange this mechanism using a condition variable.

State information

Some flag or variable should indicate how much data is in the buffer.

In the pseudocode in Example 2-5, the producer thread takes a lock on the shared buffer, places a work item in it, releases the lock, and resumes the consumer thread. The consumer thread is more complex. It first takes a lock on the shared buffer. If it finds the buffer empty, it releases the lock (thus giving the producer a chance to populate it with work) and hibernates. When the consumer thread awakens, it reacquires the lock, and removes a work item from the buffer.

Example 2-5: Producer/Consumer Threads (Pseudocode)

```
producer()
{
    .
    .
    .
  lock shared buffer
  place results in buffer
  unlock buffer
  wake up any consumer threads
    .
    .
    .
}

consumer()
{
    .
    .
    .
  lock shared buffer
  while state is not full {
        release lock and sleep
        awake and reacquire lock
        }
  remove contents
  unlock buffer
    .
    .
    .
}
```

If the threads share a buffer that can hold more than one data item, the producer can keep producing new items even if the consumer thread has not yet processed the previous one. In this case the producer and consumer must agree upon a mechanism for keeping track of how many items are currently in the buffer.

You can devise other permutations of the producer/consumer relationship based on the number of producer and consumer threads that access the same buffer. For example, an application that adopts the boss/worker model and uses a thread pool must accommodate a single producer (the boss) and many consumers (the workers).

A more specialized producer/consumer relationship, often used in pipelines for signal processing applications, uses a technique known as *double buffering*. Using double buffering, threads act as both producer and consumer to each other. In the example of double buffering shown in Figure 2-5, one set of buffers contains unprocessed data and another set contains processed data. One thread—the I/O thread—obtains unprocessed data from an I/O device and places it in a shared buffer. (In other words, it's the producer of unprocessed data.) The I/O thread also obtains processed data from another shared buffer and writes it to an I/O device. (That is, it's the consumer of processed data.) A second thread—the calculating thread—obtains unprocessed data from the shared buffer filled by the I/O thread, processes it, and places its results in another shared buffer. The calculating thread is thus the consumer of unprocessed data and the producer of processed data.

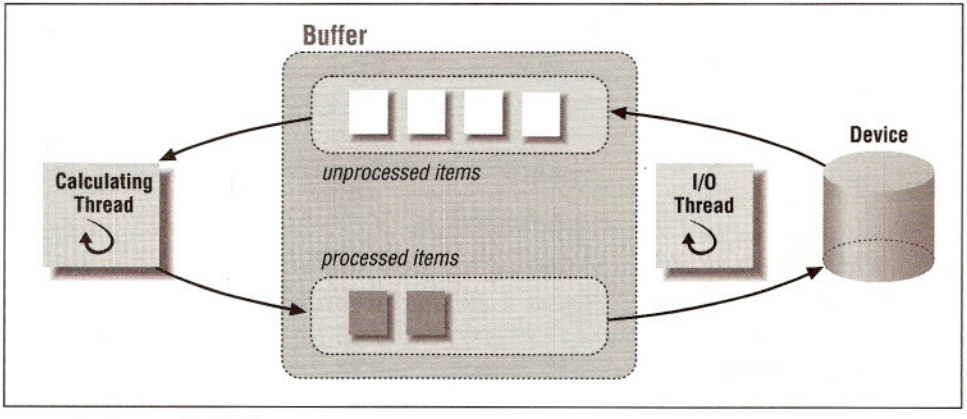

Figure 2–5: Double buffering

Some Common Problems

Regardless of the model you select, a few classes of bugs creep into nearly every threaded application at some point during its development. Avoiding them takes a lot of concentration. Finding them once they've crept in requires patience and persistence. Most bugs result from oversights in the way the application manages its shared resources. Either you forget to keep one thread out of a resource while

another is modifying it, or the way in which you attempt to synchronize access to the resource causes your threads to hang. We'll walk through a debugging session for a multithreaded program in Chapter 6, *Practical Considerations*. For the time being, we'll rest content with pronouncing a few basic rules and noting the most common pitfalls.

The basic rule for managing shared resources is simple and twofold:

- Obtain a lock before accessing the resource.
- Release the lock when you are finished with the resource.

Unfortunately, there are many borderline areas of usage where it is difficult to clearly apply this rule. In those applications in which locks and resources must be created dynamically—while multiple threads are already running—you can get into trouble very easily. The symptoms of sharing without proper synchronization are often subtle: incorrect answers and corrupted data. It is often quite hard to track down the point in the program where the error or corruption occurred. Further, the effort of debugging is often exacerbated by difficulties you may have in reproducing the bug. It is much easier to run a single-threaded program with the same inputs and get the same outputs. Programs that take advantage of concurrency aren't like that. A multithreaded program with a synchronization bug may run correctly hundreds of times for every time that it fails.

The other common bug in threaded programs results from assumptions about the "liveliness" of its threads. When a thread attempts to obtain a lock, it assumes that any thread currently holding that lock will eventually let it go. If that thread fails to release the lock for whatever reason—it hangs or it simply forgets—other threads will grind to a halt as they wait for it to release its lock. You can encounter the same problem if you make a thread wait for some variable to reach an unreachable value or wait on a condition variable that is never signaled.

Performance

When considering the performance of a threaded application, note that threads can represent negligible to significant overhead, depending on how they are implemented and how they are used. Before you add threads to a program, be sure that the benefits of threading outweigh the costs. Some of the costs of threading include:

- The memory and CPU cycles required to manage each thread, including the structures the operating system uses to manage them, plus the overhead for the Pthreads library and any special code in the operating system that supports the library.

- The CPU cycles spent for synchronization calls that enforce orderly access to shared data. These calls cost in CPU cycles to execute the calls.
- The time during which the application is inactive while one thread is waiting on another thread. This cost results from too many dependencies among threads and can be allayed by improved program design.

Example: An ATM Server

Example 2-6 is a client/server program that implements an imaginary automated teller machine (ATM) application. This server will give us an opportunity to exercise our thinking about multithreaded program design and explore more realistic—and more complicated—thread handling applications.

As shown in Figure 2-6, the example is made up of a client that provides a user interface* and a server that processes requests from the client. On disk, the server stores a database of bank accounts, each including an account ID, password, and balance.

In a typical ATM operation, a customer chooses a withdrawal from a menu presented by the client and enters the amount to be withdrawn. The client packages this information into a request that it sends to the server. The server spawns a thread that checks the user's password against the one in the database, decrements the amount of money in the user's account, and sends back an indication of whether the operation succeeded. The client and server process communicate using UNIX sockets. The client reports any information returned from the server back to the user. Multiple clients can run simultaneously.

We want the server to be capable of overlapping I/O, because the account data is stored in secondary storage and its access will require a significant amount of time. The environment is asynchronous because multiple clients may exist simultaneously, sending requests of unpredictable type, order, and frequency.

In the following sections of this chapter, we'll discuss two different implementations of this program: a serial version and a multithreaded version that uses Pthreads.† The multithreaded version of the program uses the boss/worker model inside the server. The boss looks at the first field of each request, then spawns a thread or process to handle that request. When the worker completes the request, it communicates the results directly back to the client program.

---

* The client for an ATM application should be an actual machine, but, for the purposes of this book, we'll just make it a command-line program that accepts typed-in requests. (Unfortunately, this type of client isn't realistic enough to spit out ten dollar bills.)

† You can obtain the complete source code for all versions of the ATM example, including that for the multiprocess version used in our performance testing in Chapter 6, from our ftp site. Throughout this chapter, we'll show only those interfaces and routines pertinent to the current discussion.

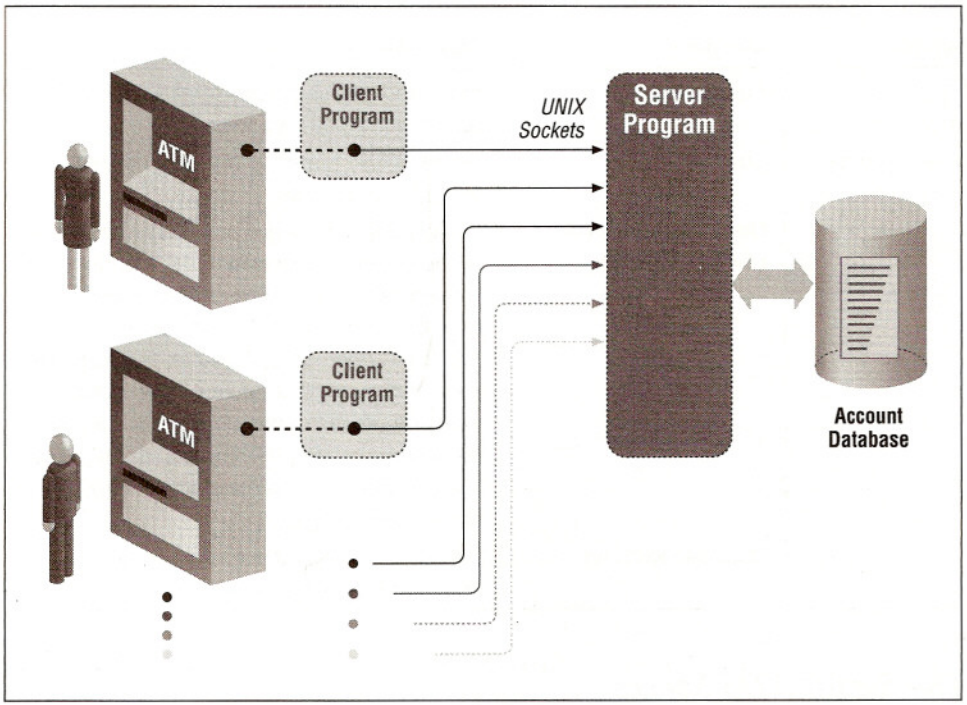

Figure 2–6: The ATM and bank database server example

For simplicity's sake, we've partitioned the client and server into modules. The interfaces between these modules will remain unchanged throughout all versions of our example. We'll change only the dispatch and service routine module from one version to another. Table 2-1 shows the contents of the client and server modules.

Table 2–1: The ATM Example Program Modules

Module	Component	Description
Client program	User interface (*main*)	Prompts a customer for a request, parses the response, and makes a remote procedure call (RPC) to access the server.
	RPC	Includes a procedure for each possible type of request. Each procedure copies its arguments into a buffer and passes the buffer to the communication module for transmission to the server. When a response arrives from the server, the procedure checks its return values.

Table 2-1: The ATM Example Program Modules (continued)

Module	Component	Description
	Communication	Finds and passes buffers to and from the server using UNIX sockets.
Server program	Communication	Receives and transmits buffers to clients using UNIX sockets.
	Dispatch (and service) routines (*main*)	Obtains input buffers from clients by means of the communication module, identifies the request type and copies out arguments, and calls the service routine that handles the requested operation. Together, the dispatch and service routines make up the server-side procedures of the client's RPC. When request processing is complete, the dispatch routine prepares and transmits a response buffer to the client.
	Database routines	Reads from and writes to the account database file using standard file I/O.

The Serial ATM Server

If we didn't have threads, what would be the simplest implementation of the ATM server? One that comes to mind is a program that runs in a loop, processing available requests serially. If a request is available, the program processes it in a request-specific service routine and sends a response to the client. The *main* routine for this version of the server is shown in Example 2-6.

Example 2-6: Serial ATM Server: main Routine (atm_svr_serial.c)

```
extern int
main(argc, argv)
int argc;
char **argv;
{
  char req_buf[COMM_BUF_SIZE], resp_buf[COMM_BUF_SIZE];
  int  conn;
  int  trans_id;
  int  done=0;

  atm_server_init(argc, argv);

  /* loop forever */
  for(;;) {

    server_comm_get_request(&conn, req_buf);
    sscanf(req_buf, "%d", &trans_id);
```

Example 2-6: Serial ATM Server: main Routine (atm_svr_serial.c) (continued)

```
            switch(trans_id) {

              case CREATE_ACCT_TRANS:
                   create_account(resp_buf);
                   break;

              case DEPOSIT_TRANS:
                   deposit(req_buf, resp_buf);
                   break;

              case WITHDRAW_TRANS:
                   withdraw(req_buf, resp_buf);
                   break;

              case BALANCE_TRANS:
                   balance(req_buf, resp_buf);
                   break;

              case SHUTDOWN:
                   if (shutdown_req(req_buf, resp_buf)) done = 1;
                   break;

              default:
                   handle_bad_trans_id(req_buf, resp_buf);
                   break;

              }

            server_comm_send_response(conn, resp_buf);

            if(done) break;

          }

          server_comm_shutdown();
        }
    return 0;
```

The serial version of our ATM server can process only a single request at a time, no matter how many clients are requesting service.

Handling asynchronous events: blocking with select

The server handles the asynchronous arrival of requests from clients by waiting. When the server's *main* routine calls *server_comm_get_request*, the server's communication layer uses a UNIX *select* call to determine which channels have data on them waiting to be read. If none do, the *select* call (and consequently the *server_comm_get_request* call) blocks until data arrives.

Handling file I/O: blocking with read/write

The server in Example 2-6 does nothing but block when performing file operations. When it issues a *read* or *write* call to the file, the server waits until the operating system completes the operation and the call returns.

The server could have used UNIX signals to access the file without blocking. If so, it would need to establish a signal handler that processes the results of its I/O requests and to register this handler with the operating system such that it takes control when the completion of an I/O request is signaled. This would allow the server to make asynchronous I/O calls that return immediately. When the request completes at a later time, the server is interrupted and put into its signal handler to process the results.

The big drawback for using asynchronous I/O in a serial server is in the complicated state management and synchronization problems that arise between the server and its signal handler. The program must keep track of the state of all in-progress requests. It must create and maintain locks for various resources (such as account records) so that they are not simultaneously accessed in program and signal contexts. Finally, the clean division of the program into modules breaks down. The communication, server, and database modules all get mixed together.

All in all, the experiment of using asynchronous I/O in a serial ATM server is a good argument for designing such a server to use threads. It's much cleaner to let a single thread wait for I/O to complete than it is to manage the complexity of signals and synchronization.

The serial version of our ATM server works—in fact, it works quite well—when the input stream of requests is light. However, the performance of the serial server degrades rapidly as more and more clients request access to its data. Clients begin to see longer and longer delays in the processing of their requests because all are blocked by server access to the database. In Chapter 6, we'll run some tests on single-threaded and multithreaded versions of the server that show the point at which it becomes inefficient to use the serial version.

What can we do to improve the performance of our server under load? It can help a lot to allow it to move on to another client request while its I/O to the database is proceeding. The next versions of our server will do just that.

The Multithreaded ATM Server

Let's add threads to our example. We'll begin by identifying those tasks we want individual threads to process. Having each request processed by a separate thread may or may not be a good starting point.

Before we pursue our design, let's step back and look again at the general criteria for selecting tasks for threads. In general we'd like to select tasks for our ATM server's threads based on whether:

- They are independent of each other.

  If we assume that simultaneous accesses to the same account are rare, it makes sense to have each request processed by a separate thread. Threads will not compete for account data. No individual thread will rely on the work accomplished by another thread to complete its work.

- They can become blocked in potentially long waits.

  This is true of all requests to our server, because any access to the account database could involve disk access to a file.

- They can use a lot of CPU cycles.

  Our server contains no tasks that can be defined as compute intensive.

- They must respond to asynchronous events.

  This is true of the manner in which the communication layer of our server accepts client requests.

- They require scheduling.

  In our first pass at a multithreaded version of our ATM server we won't use scheduling. But we can imagine that certain operations could be given higher priority than others. A thread handling a shutdown request might be given priority over other requests. If we used the database to generate monthly account statements, we could give those requests lower priority than direct customer requests to bank accounts.

We must also bear in mind some key constraints our ATM server places on our program design:

- We must maintain correctness.

  For the multithreaded version of our ATM server to produce correct and consistent results, we must ensure that two threads don't corrupt an account by writing information to it simultaneously. Thus, we'll use locks to protect the account data.

- We must maintain the liveliness of the threads.

  We must avoid those types of programming bugs in which a worker thread obtains a lock on account data and then exits without releasing the lock. Other worker threads that subsequently attempt to obtain the lock on the same data will deadlock, waiting forever.

- We must minimize overhead.

  Our threads can't spend all their time synchronizing with each other or they are not worth their overhead. As a simplification, we'll start out in our example by allocating a thread for each request. We can later enhance it by allowing threads to remain active, waiting for new requests (that is, we could use a thread pool).

Model: boss/worker model

Because it's a classic server program, we'll use the boss/worker model for our ATM server. A boss thread accepts input from remote clients through the communication module. Worker threads handle each client account request.

Figure 2-7 shows the structure of our ATM example program using the boss/worker ATM server. The boss thread is neatly encapsulated by the server's *main* routine. Each worker thread runs one service routine: deposit, withdraw, and so on.

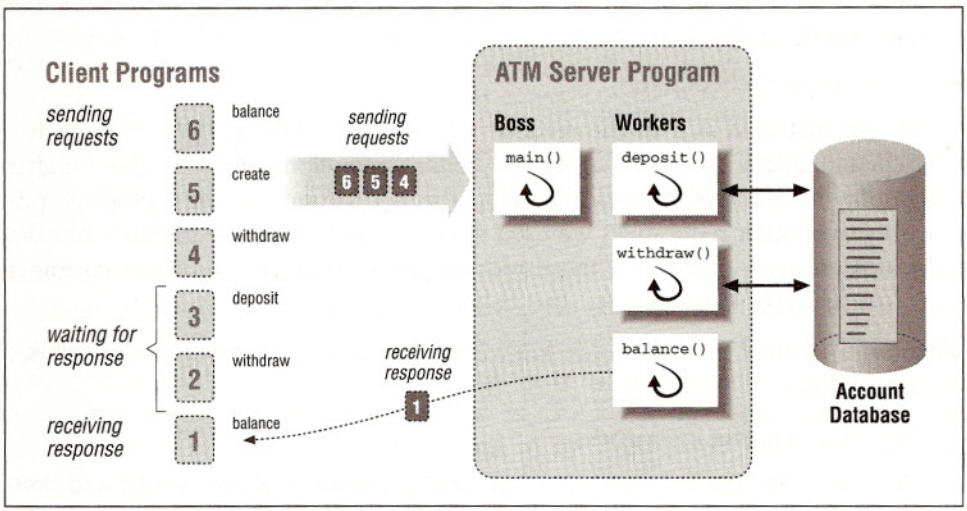

Figure 2-7: The boss/worker Pthreads ATM server

The boss thread

We'll start building our multithreaded ATM server's boss thread from our serial server's *main* routine. The boss thread simply manages the receipt of incoming requests using the *server_comm_get_request* routine. After it obtained each request from the communication module, the serial server's *main* routine unpacked it and called the appropriate service routine. The boss thread's *main* routine will create a

worker thread to which it will pass the request. The worker thread begins by executing a generic request-processing routine called *process_request*, as shown in Example 2-7.

Example 2-7: Multithreaded ATM Server: Boss Thread (atm_svr.c)

```
typedef struct workorder{
        int conn;
        char req_buf[COMM_BUF_SIZE];
        } workorder_t;
extern int
main(argc, argv)
int argc;
char **argv;
{
  workorder_t *workorderp;
  pthread_t   *worker_threadp;
  int  conn;
  int  trans_id;

  atm_server_init(argc, argv);

  for(;;) {

    /*** Wait for a request ***/
    workorderp = (workorder_t *)malloc(sizeof(workorder_t));
    server_comm_get_request(&workorderp->conn, workorderp->req_buf);

    sscanf(workorderp->req_buf, "%d", &trans_id);
    if (trans_id == SHUTDOWN) {
              .
              .
              .
              break;
              }

    /*** Spawn a thread to process this request ***/
    worker_threadp=(pthread_t *)malloc(sizeof(pthread_t));
    pthread_create(worker_threadp, NULL, process_request, (void *)workorderp);

    pthread_detach(*worker_threadp);
    free(worker_threadp);
  }

  server_comm_shutdown();
  return 0;
}
```

Dynamically detaching a thread

In the code for our boss thread's *main* routine, we've introduced a new Pthreads call—*pthread_detach*. The *pthread_detach* function notifies the Pthreads library that we don't want to join our worker threads: that is, we will never request their

exit status. If we don't explicitly tell the Pthreads library that we don't care about a thread's exit status, it'll keep the shadow of the thread alive indefinitely after the thread terminates (in the same way that UNIX keeps the status of zombie processes around). Detaching our worker threads frees the Pthreads library from storing this information, thus saving space and time. We are still responsible for freeing any space we dynamically allocated to hold the *pthread_t* itself.

Aside from using *pthread_detach* on an existing thread, you can create threads already in the detached state. We'll discuss this method in Chapter 4, *Managing Pthreads.*

A worker thread

In our multithreaded ATM server, each worker thread begins its life in a new request-parsing routine called *process_request.* This is a generic request-parsing routine that all workers use regardless of which requests they actually process. Because different service routines process different requests, the primary job of *process_request* is to select the proper service routine. We accomplish this by means of a simple case statement, shown in Example 2-8.

Example 2-8: Multithreaded ATM Server: Worker Thread process_request

```
void process_request(workorder_t *workorderp)
{
  char resp_buf[COMM_BUF_SIZE];
  int  trans_id;
  sscanf(workorderp->req_buf, "%d", &trans_id);

  switch(trans_id) {

      case CREATE_ACCT_TRANS:
           create_account(resp_buf);
           break;

      case DEPOSIT_TRANS:
           deposit(workorderp->req_buf, resp_buf);
           break;

      case WITHDRAW_TRANS:
           withdraw(workorderp->req_buf, resp_buf);
           break;

      case BALANCE_TRANS:
           balance(workorderp->req_buf, resp_buf);
           break;

      default:
           handle_bad_trans_id(workorderp->req_buf, resp_buf);
           break;
      }
```

exit status. If we don't explicitly tell the Pthreads library that we don't care about a thread's exit status, it'll keep the shadow of the thread alive indefinitely after the thread terminates (in the same way that UNIX keeps the status of zombie processes around). Detaching our worker threads frees the Pthreads library from storing this information, thus saving space and time. We are still responsible for freeing any space we dynamically allocated to hold the *pthread_t* itself.

Aside from using *pthread_detach* on an existing thread, you can create threads already in the detached state. We'll discuss this method in Chapter 4, *Managing Pthreads.*

A worker thread

In our multithreaded ATM server, each worker thread begins its life in a new request-parsing routine called *process_request.* This is a generic request-parsing routine that all workers use regardless of which requests they actually process. Because different service routines process different requests, the primary job of *process_request* is to select the proper service routine. We accomplish this by means of a simple case statement, shown in Example 2-8.

Example 2-8: Multithreaded ATM Server: Worker Thread process_request

```
void process_request(workorder_t *workorderp)
{
  char resp_buf[COMM_BUF_SIZE];
  int  trans_id;
  sscanf(workorderp->req_buf, "%d", &trans_id);

  switch(trans_id) {

      case CREATE_ACCT_TRANS:
           create_account(resp_buf);
           break;

      case DEPOSIT_TRANS:
           deposit(workorderp->req_buf, resp_buf);
           break;

      case WITHDRAW_TRANS:
           withdraw(workorderp->req_buf, resp_buf);
           break;

      case BALANCE_TRANS:
           balance(workorderp->req_buf, resp_buf);
           break;

      default:
           handle_bad_trans_id(workorderp->req_buf, resp_buf);
           break;
      }
```

Example 2-8: Multithreaded ATM Server: Worker Thread process_request (continued)

```
        server_comm_send_response(workorderp->conn,
                                  resp_buf);

        free(workorderp);

    }
```

In our ATM example, the boss thread is always active. It creates worker threads, as needed, to process requests. Each active worker could be processing a request on a different account, or each worker could be performing a separate operation on the same account. It shouldn't matter to our program. The boss thread limits the number of active worker threads in the server.

At any given time in the ATM example, a request could be in one of three places:

- Queued at the server's communication module, waiting to be picked up by the boss thread
- In the boss thread's hands, about to be passed off to a worker thread
- In the hands of a worker thread, being processed

Synchronization: what's needed

So far, we haven't shown any synchronization between the threads in our multithreaded ATM server. We'll go into the details in Chapter 3, *Synchronizing Pthreads*, and Chapter 4.

Right now we'll just list what synchronization we'll need:

- Accounts

  Now that we have multiple workers accessing the database through the service routines (*deposit*, *withdraw*, and *balance*), we'll need to deal with the possibility that two routines may try to manipulate the same account balance at the same time. To prevent simultaneous access, we'll protect database accesses with a mutex variable.

- Limiting the number of workers

  To keep from overloading the CPUs, the boss must limit the number of worker threads that can exist concurrently. It must maintain an ongoing count of worker threads and decrement the count as threads exit. We'll do that and add a check for exiting worker threads.

- Server shutdown

  The ATM client lets privileged users shut down the server. To make our server more robust, we must ensure that the server has completed the requests that are already in progress before it stops accepting new requests and shuts itself down. We'll do this by adding code so that the boss can tell when threads are active.

Future enhancements

We'll add the synchronization we discussed to our multithreaded ATM server in Chapter 3. We'll also enhance our server throughout the remainder of this book. Among the design refinements we'll consider are:

- Thread pools

  Our ATM server creates a worker thread each time it receives a request and pays the cost of thread creation each time. What if we allowed our server to reuse worker threads? When the server starts, it can create a predetermined number of workers in an idle state. Each worker thread could take requests off a queue and return to an idle state (instead of exiting) after completing each request. The reduction in overhead would pay off in performance.

- Cancellation

  In a couple of situations it would be useful if the boss thread could interrupt and terminate a worker thread: to cancel an in-progress request that is no longer wanted or to support a quick shutdown.

- Scheduling

  We could give some threads—possibly shutdown threads and deposit threads—priority over other threads. When a CPU becomes available, we could give these threads first crack at it.

Example: A Matrix Multiplication Program

In this section we'll look at a program very different from the ATM client/server example: one that exemplifies how you can break down a program into tasks. Whereas we used the boss/worker model to design our ATM server, we'll use the peer model for this one.

A large class of programs are computationally intensive and work on large sets of data: image processing, statistical analysis, and finite element modeling, to name a