```
    printf("%s %ld bytes in %.1f seconds",
           direction, amount, delta);
    printf(" [%.0f bits/sec]", (amount*8.)/delta);
}
```

On the implementation front, when you consider alternatives, you should always use measurement data to evaluate them; then, when you write the corresponding code, measure its performance-critical components early and often, to avoid nasty surprises later on.

You will read more about measurement techniques in the next section. In many cases, a major source of performance improvements is the use of a more efficient algorithm; this topic is covered in Section 4.2. Having ruled out important algorithmic inefficiencies, you can begin to look for expensive operations that impact your program's performance. Such operations can range from expensive CPU instructions to operating system and peripheral interactions; all are covered in Sections 4.3–4.6. Finally, in Section 4.7, we examine how caching is often used to trade memory space for execution time.

Exercise 4.1 Choose five personal productivity applications and five infrastructure applications your organization relies on. For each application, list its most important time-related attribute.

Exercise 4.2 Your code provides a horrendously complicated yet remarkably efficient implementation of a simple algorithm. Although it works correctly, measurements have demonstrated that a simple call to the language's runtime library implementation would work just as well for all possible input cases. Provide five arguments for scrapping the existing code.

4.1 Measurement Techniques

Humans are notoriously bad at guessing why a system is exhibiting a particular time-related behavior. Starting your search by plunging into the system's source code, looking for the time-wasting culprit, will most likely be a waste of your time. The only reliable and objective way to diagnose and fix time inefficiencies and problems is to use appropriate measurement tools. Even tools, however, can lie when applied to the wrong problem. The best approach, therefore, is to first evaluate and understand the type of workload a program imposes on your system and then use the appropriate tools for each workload type to analyze the problem in detail.

4.1.1 Workload Characterization

A loaded program on an otherwise idle system can at any time instance be in one of the three different states:

1. Directly executing code. The time spent directly executing code is termed *user time* (u), denoting that the process operates in the context of its user.
2. Having the kernel execute code on its behalf. Correspondingly, the time the kernel devotes to executing code in response to a process's requests is termed *system time* (s), or *kernel time*.
3. Waiting for an external operation to complete. Operations that cause a program to wait are typically read or write requests to slow peripherals, such as disks and printers, input from human users, and communication with other processes, often over a network link. This time is referred to as *idle time*.

The total time a program spends in all three states is termed the *real time* (r) the program takes to operate, often also referred to as the program's wall clock time: the time we can measure using a clock on the wall or a stopwatch. The sum of the program's user and system time is also referred to as CPU time.

The relationship among the real, kernel, and user time in a program's (or complete system's) execution is an important indicator of its workload type, the relevant diagnostic analysis tools, and the applicable problem-resolution options. You can see these elements summarized in Table 4.1; we analyze each workload type in a separate section.

On Unix-type systems, you can specify a process as an argument of the *time*[11] command to obtain the user, system, and real time the process took to its completion. On Windows systems, the *taskmgr* command can list a process's CPU time and show a chart indicating the time the system spends executing kernel code. For nonterminating processes, you will have to obtain similar figures on Unix systems through the commonly available *top* command. On an otherwise unloaded system, you can easily determine a process's user and system time from the corresponding times of the whole system. When analyzing a process's behavior, carefully choose its execution environment: Execute the process either in a realistic setting that reflects the actual intended use or on an unloaded system that will not introduce spurious noise in your measurements.

[11] netbsdsrc/usr.bin/time

Table 4.1 Timing Profile Characterization, Diagnostic Tools, and Resolution Options

Timing Profile	$r \gg u + s$	$s > u$	$u \simeq r$
Characterization	I/O-bound	Kernel-bound	CPU-bound
Diagnostic tools	Disk, network, and virtual memory statistics; network packet dumps; system call tracing	System call tracing	Function profiling; basic block counting
Resolution options	Caching; efficient network protocols and disk data structures; faster I/O interfaces or peripherals	Caching; a faster CPU	Efficient algorithms and data structures; other code improvements; a faster CPU or memory system

4.1.2 I/O-Bound Tasks

Programs and workloads whose real time r is a lot larger than their CPU time $u + s$ are characterized as I/O-bound. Such programs spend most of their time idle, waiting for slower peripherals or processes to respond. Consider as an example the task of creating a copy of the word dictionary on a diskless system with an NFS-mounted disk and a 10Mb/s network interface:

```
$ /usr/bin/time cp /usr/share/dict/words wordcopy
        5.68 real         0.00 user         0.32 sys
```

It would be futile to try to analyze the *cp*[12] command, looking for optimization opportunities that would make it execute faster than the 5.68 seconds it took. The results of the *time* command indicate that *cp* spent negligible CPU time; for 94% of its clock time, it was waiting for a response from the NFS-mounted disk.

The diagnostic tools we use to analyze I/O-bound tasks aim to find the source of the bottleneck and any physical or operational constraints affecting it. The physical constraint could be lagging responses from a genuinely slow disk or the network;

[12]netbsdsrc/bin/cp

the corresponding operational constraints could be the overloading of the disk or the network with other requests that are not part of our workload. On Unix systems, the *iostat*,[13] *netstat*,[14] *nfsstat*,[15] and *vmstat*[16] commands provide summaries and continuous textual updates of a system's disk and terminal, network, and virtual memory performance. On Windows systems, the management console performance monitor (invoked as the *perfmon* command) can provide similar figures in the form of detailed charts. After we find the source of the bottleneck, we can either improve the hardware performance of the corresponding peripheral (by deploying a faster one in its place) or reduce the load we impose on it. Strategies for reducing the load include caching (discussed in Section 4.7) and the adoption of more efficient disk data structures or network protocols, which will minimize the expensive transactions. We discuss how these elements relate to specific source code instances in Section 4.5.

Analyzing the disk performance on the NFS server hosting the `words` file in our example using *iostat* shows the load on the disk to be quite low:

```
             ad0
  KB/t tps  MB/s
  0.00   0  0.00
 32.80  15  0.47
 27.12  24  0.63
 37.31  26  0.94
 73.60  10  0.71
 35.24  33  1.14
 25.14  21  0.51
  7.00   4  0.03
  0.00   0  0.00
```

The load never exceeded 1.14MB/s, well below even the lowly 3.3MB/s PIO[17] mode 0 transfer mode limit. Therefore, the problem is unlikely to be related to the actual disk I/O. However, using *netstat* to monitor the network I/O on the diskless machine does provide us with an insight:

[13]netbsdsrc/usr.sbin/iostat
[14]netbsdsrc/usr.bin/netstat
[15]netbsdsrc/usr.bin/nfsstat
[16]netbsdsrc/usr.bin/vmstat
[17]The programmed input/output mode is a legacy ATAPI hard disk data transfer protocol that supports data transfer rates ranging from 3.3MB/s (PIO mode 0) to 16.6MB/s (PIO mode 4). Modern ATAPI drives typically operate using the Ultra-DMA protocol, supporting transfer rates up to 133MB/s.

```
           input        (Total)          output
packets  errs      bytes    packets  errs      bytes colls
      1     0         60          1     0        250     0
    210     0     237744        204     0     230648   113
    417     0     515196        418     0     513722   324
    383     0     467208        402     0     496650   292
    368     0     451418        381     0     470212   259
    425     0     519588        430     0     515714   301
    400     0     488434        400     0     496816   287
      9     0       6106         15     0      11886     7
      1     0         60          1     0        138     0
```

The maximum network throughput attained is 7.9mb/s,[18] which is very near the limit of what the particular machine's half-duplex 10mb/s ethernet interface can deliver in practice. We can therefore safely say that the operation efficiency of the particular command invocation is bound by the capacity of the machine's network interface. Minimizing the network traffic (by adding, for example, a local disk and keeping copies of the data on it) or improving the network interface (for example, to 100mb/s) will most probably correct the particular deficiency.

In some cases, a more detailed examination of the I/O operations may be required to locate the problem. Two tool categories that will provide such details are system call tracers and network packet-monitoring tools. On Unix systems, you will find such commands as *strace*, *dtrace*, *truss*, *ltrace*,[19] *tcpdump*, and *ethereal*;[20] for Windows systems, you will have to download such programs as *apispy* and *windump*. Your objective here is to examine either the sheer volume of the corresponding transactions or the time a single transaction takes to complete. Most tools provide options for time stamping each transaction, thus providing you with an easy way to reason about the program's behavior.

Consider, for example, the performance of the Apache *logresolve*[21] command. The command reads a web server log and replaces numeric IP addresses with the corresponding host name. Examining its operation with *time* reveals that it spends 99.99%[22] of its time sitting idle:

```
$ /usr/bin/time logresolve <httpd-access.log >/dev/null
     1230.55 real         0.04 user         0.03 sys
```

[18]Calculated as $8\,(519,588+515,714)\,/1,024^2$.
[19]freshmeat.net/projects/ltrace
[20]http://www.ethereal.com
[21]apache/src/support/logresolve.c
[22]Calculated as $100\,(1-(0.04+0.03)\,/1,230.55)$.

The output of the *netstat* command also shows an (almost) idle network connection:

```
         input        (Total)          output
packets  errs      bytes    packets  errs      bytes colls
      7     0        486          8     0        108     0
     14     0        229         11     0        383     0
      3     0        336          3     0        324     0
      3     0        216          4     0        301     0
      3     0        667          3     0        216     0
      6     0         98          2     0        301     0
```

However, obtaining a network packet dump with *tcpdump* and examining the timestamps of a single name lookup operation reveal that this may require up to 150 ms:[23]

```
16:15:33.283221 istlab.dmst.aueb.gr.1024 > gns1.nominum.com.domain:
        9529 [1au] PTR? 105.199.133.198.in-addr.arpa. (57)
16:15:33.433305 gns1.nominum.com.domain > istlab.dmst.aueb.gr.1024:
        9529*- 1/2/0 (122) (DF) [tos 0x80]
```

Ignoring the effects of caching (which *logresolve* does perform),[24] we can easily see that processing a 10 million log file may require 17 days.[25] Thus, *logresolve*'s caching is certainly a worthwhile investment.

4.1.3 Kernel-Bound Tasks

Programs and workloads whose system time s is larger than their user time u can be characterized as *kernel-bound*. Strictly speaking, the kernel-bound tasks are also CPU-bound in the sense that improving the processor's speed will often increase their performance. However, we treat such programs as a separate class because they require different diagnostic and resolution techniques. Your objective when dealing with a kernel-bound task is first to determine what kernel system calls the task performs. A system call tracing utility, such as *strace* or *apispy*, will be your tool of choice here.[26] Such a program will provide you with a list of all the system calls a process has performed. As we discuss in Section 4.4, system calls are relatively expensive

[23]Calculated as $(433,305 - 283,221)/1,000$.

[24]apache/src/support/logresolve.c:32–34

[25]Calculated as $0.150 \times 10^7/60/60/24$.

[26]Don't confuse system call tracing with the program comprehension human activity with the same name we discuss in Section 7.2.

operations; therefore, you will then browse the system call list to see whether any calls could be eliminated by the use of appropriate user-level caching strategies.

Consider, as an example, running the directory listing *ls* command to recursively list the contents of a large directory tree:

```
$ /usr/bin/time ls -lR >/dev/null
        4.59 real         1.28 user         2.73 sys
```

We can easily see that *ls* spends twice as much time operating in the context of the kernel than the time it spends executing user code. Examining the output of the *strace* command, we see that for listing 7,263 files, *ls* performs 18,289 system calls. The *strace* command also provides a summary display, which we can use to see the number of different system calls and the average time each one took:

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- -------------
 42.52    5.604588         994      5638           lstat
 13.95    1.838295         842      2183           open
 12.38    1.632320         599      2727           fstat
 12.37    1.630742         747      2182           fchdir
 11.41    1.503261         690      2180           close
  2.94    0.387360         353      1096           getdirentries
[...]
```

Armed with that information, we can reason that *ls* performs an `lstat` or `fstat` system call for every file it visits.

Based on those results, we can look at the source to find the cause of the various `stat` calls:[27]

```
if (!f_inode && !f_longform && !f_size && !f_type &&
    sortkey == BY_NAME)
        fts_options |= FTS_NOSTAT;
```

The preceding snippet tells us that by omitting the “long” option, we will probably eliminate the corresponding `stat` calls. The improvement in the execution performance figures corroborates this insight:

```
$ /usr/bin/time ls -R >/dev/null
        1.08 real         0.28 user         0.77 sys
```

[27]netbsdsrc/bin/ls/ls.c:232–234