change information, as a CPU does, but just moves information from one place to another.

Now that we have covered the basic types of storage devices and ways to connect them to the CPU, we are ready to look at ways to evaluate the performance of storage systems.

6.4 I/O Performance Measures

I/O performance has measures that have no counterparts in CPU design. One of these is diversity: Which I/O devices can connect to the computer system? Another is capacity: How many I/O devices can connect to a computer system?

In addition to these unique measures, the traditional measures of performance, response time and throughput, also apply to I/O. (I/O throughput is sometimes called *I/O bandwidth*, and response time is sometimes called *latency*.) The next two figures offer insight into how response time and throughput trade off against each other. Figure 6.16 shows the simple producer-server model. The producer creates tasks to be performed and places them in a buffer; the server takes tasks from the first-in-first-out buffer and performs them.

Response time is defined as the time a task takes from the moment it is placed in the buffer until the server finishes the task. Throughput is simply the average number of tasks completed by the server over a time period. To get the highest possible throughput, the server should never be idle, and thus the buffer should never be empty. Response time, on the other hand, counts time spent in the buffer and is therefore minimized by the buffer being empty.

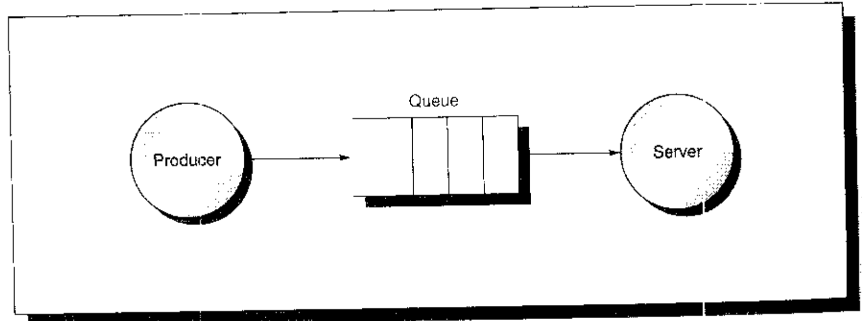

FIGURE 6.16 The traditional producer-server model of response time and throughput. Response time begins when a task is placed in the buffer and ends when it is completed by the server. Throughput is the number of tasks completed by the server in unit time.

Another measure of I/O performance is the interference of I/O with CPU execution. Transferring data may interfere with the execution of another process. There is also overhead due to handling I/O interrupts. Our concern here is how many more clock cycles a process will take because of I/O for another process.

Throughput versus Response Time

Figure 6.17 shows throughput versus response time (or latency) for a typical I/O system. The knee of the curve is the area where a little more throughput results in much longer response time or, conversely, a little shorter response time results in much lower throughput.

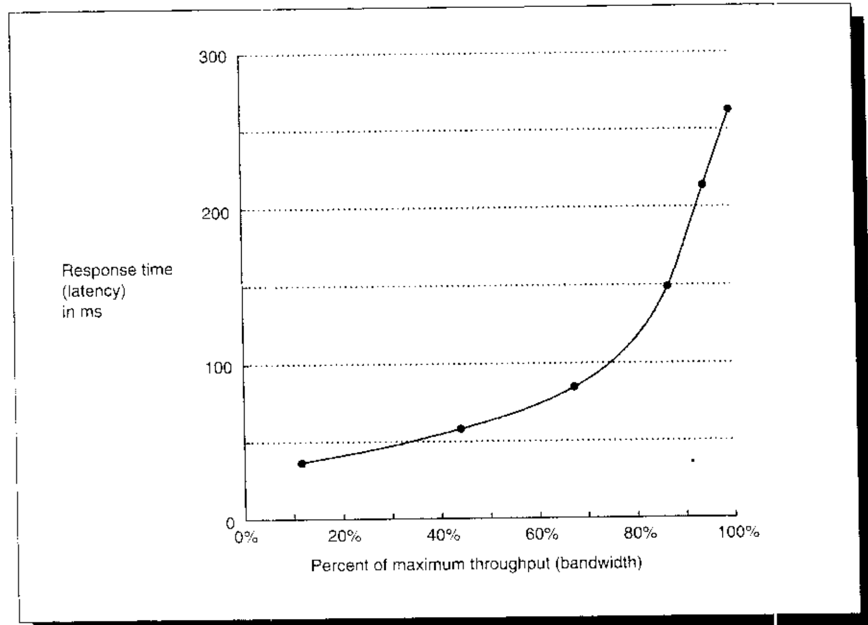

FIGURE 6.17 Throughput versus response time. Latency is normally reported as response time. Note that the minimum response time achieves only 11% of the throughput, while the response time for 100% throughput takes seven times the minimum response time. Note that the independent variable in this curve is implicit: To trace the curve, you typically vary load (concurrency). Chen et al. [1990] collected these data for an array of magnetic disks.

Life would be simpler if improving performance always meant improvements in both response time and throughput. Adding more servers, as in Figure 6.18, increases throughput: By spreading data across two disks instead of one, tasks may be serviced in parallel. Alas, this does not help response time, unless the workload is held constant and the time in the buffers is reduced because of more resources.

How does the architect balance these conflicting demands? If the computer is interacting with human beings, Figure 6.19 suggests an answer. This figure presents the results of two studies of interactive environments: one keyboard oriented and one graphical. An interaction, or *transaction,* with a computer is divided into three parts:

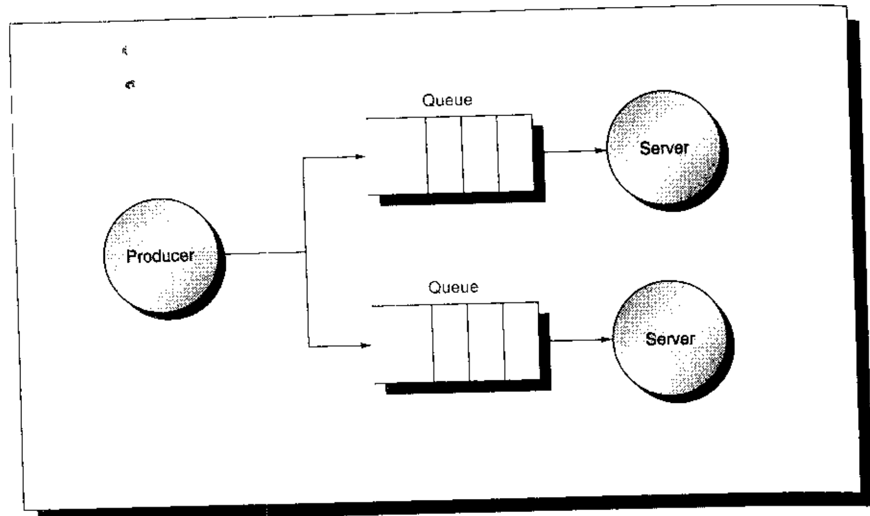

FIGURE 6.18 The single-producer, single-server model of Figure 6.16 is extended with another server and buffer. This increases I/O system throughput and takes less time to service producer tasks. Increasing the number of servers is a common technique in I/O systems. There is a potential imbalance problem with two buffers: Unless data is placed perfectly in the buffers, sometimes one server will be idle with an empty buffer while the other server is busy with many tasks in its buffer.

1. *Entry time*—The time for the user to enter the command. The graphics system in Figure 6.19 took 0.25 seconds on average to enter a command versus 4.0 seconds for the keyboard system.
2. *System response time*—The time between when the user enters the command and the complete response is displayed.
3. *Think time*—The time from the reception of the response until the user begins to enter the next command.

The sum of these three parts is called the *transaction time*. Several studies report that user productivity is inversely proportional to transaction time; *transactions per hour* is a measure of the work completed per hour by the user.

The results in Figure 6.19 show that reduction in response time actually decreases transaction time by more than just the response time reduction: Cutting system response time by 0.7 seconds saves 4.9 seconds (34%) from the conventional transaction and 2.0 seconds (70%) from the graphics transaction. This implausible result is explained by human nature: People need less time to think when given a faster response.

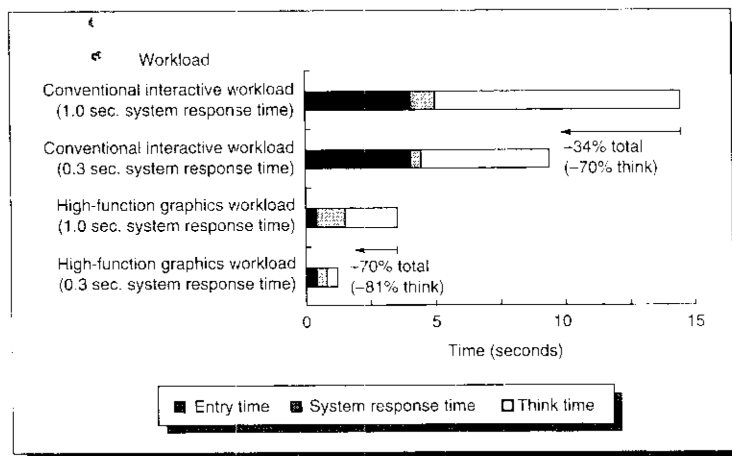

FIGURE 6.19 A user transaction with an interactive computer divided into entry time, system response time, and user think time for a conventional system and graphics system. The entry times are the same, independent of system response time. The entry time was 4 seconds for the conventional system and 0.25 seconds for the graphics system. (From Brady [1986].)

Whether these results are explained as a better match to the human attention span or getting people "on a roll," several studies report this behavior. In fact, as computer responses drop below one second, productivity seems to make a more than linear jump. Figure 6.20 compares transactions per hour (the inverse of transaction time) of a novice, an average engineer, and an expert performing physical design tasks on graphics displays. System response time magnified talent: a novice with subsecond response time was as productive as an experienced professional with slower response, and the experienced engineer in turn could outperform the expert with a similar advantage in response time. In all cases the number of transactions per hour jumps more than linearly with subsecond response time.

Since humans may be able to get much more work done per day with better response time, it is possible to attach an economic benefit to the customer of lowering response time into the subsecond range [IBM 1982], thereby helping the architect decide how to tip the balance between response time and throughput.

A Little Queuing Theory

With an appreciation of the importance of response time, we can give a set of simple theorems that will help calculate response time and throughput of an entire I/O system. Let's start with a black box approach to I/O systems, as in Figure 6.21. In our example the CPU is making I/O requests that arrive at the I/O device, and the requests "depart" when the I/O device fulfills them.

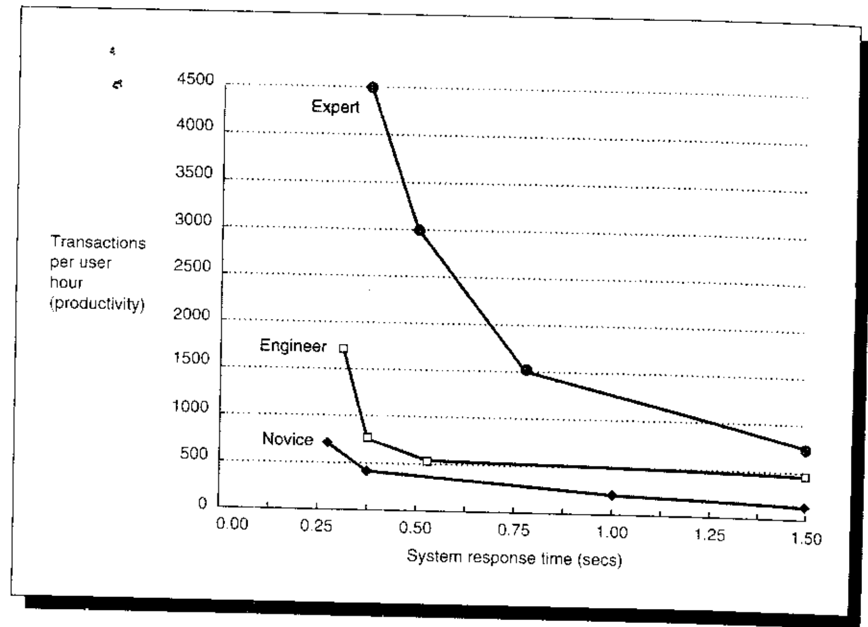

FIGURE 6.20 Transactions per hour versus computer response time for a novice, experienced engineer, and expert doing physical design on a graphics system. *Transactions per hour* is a measure of productivity. (From IBM [1982].)

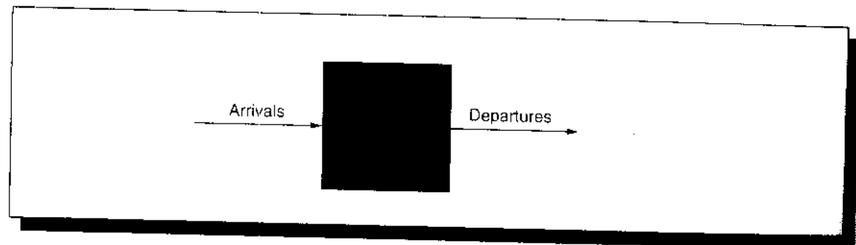

FIGURE 6.21 Treating the I/O system as a black box. This leads to a simple but important observation: If the system is in steady state, then the number of tasks entering the systems must equal the number of tasks leaving the system.

We are usually interested in the long term, or steady state, of a system rather than in the initial start-up conditions. Hence we make the simplifying assumption that we are evaluating systems in equilibrium: the input rate must be equal to the output rate. This leads us to *Little's Law*, which relates the average number of tasks in the system, the average arrival rate of new tasks, and the average time to perform a task:

$$\text{Mean number of tasks in system} = \text{Arrival rate} \times \text{Mean response time}$$

Little's Law applies to any system in equilibrium, as long as nothing inside the black box is creating new tasks or destroying them. This simple equation is surprisingly powerful, as we shall see.

If we open the black box, we see Figure 6.22. The areas where the tasks accumulate, waiting to be serviced, is called the *queue,* or *waiting line*, and the device performing the requested service is called the *server.*

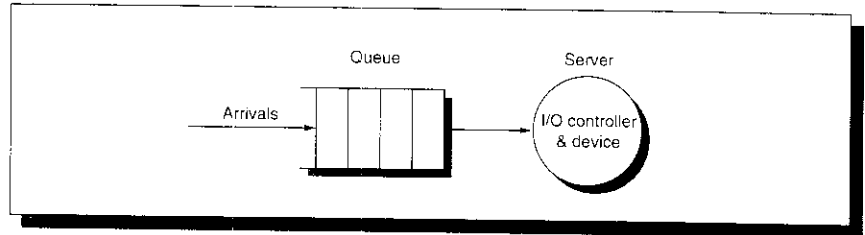

FIGURE 6.22 The single server model for this section. In this situation an I/O request "departs" by being completed by the server.

Little's Law and a series of definitions lead to several useful equations:

$Time_{server}$—Average time to service a task; service rate is $1/Time_{server}$, traditionally represented by the symbol μ in many texts.

$Time_{queue}$—Average time per task in the queue.

$Time_{system}$—Average time/task in the system, or the response time, the sum of $Time_{queue}$ and $Time_{server}$.

Arrival rate—Average number of arriving tasks/second, traditionally represented by the symbol λ in many texts.

$Length_{server}$—Average number of tasks in service.

$Length_{queue}$—Average length of queue.

$Length_{system}$—Average number of tasks in system, the sum of $Length_{queue}$ and $Length_{server}$.

One common misunderstanding can be made clearer by these definitions: whether the question is how long a task must wait in the queue before service starts ($Time_{queue}$) or how long a task takes until it is completed ($Time_{system}$). The latter term is what we mean by response time, and the relationship between the terms is $Time_{system} = Time_{queue} + Time_{server}$.

Using the terms above we can restate Little's Law as

$$Length_{system} = \text{Arrival rate} \times Time_{system}$$

We can also talk about how busy a system is from these definitions. Server utilization is simply

$$\text{Server utilization} = \frac{\text{Arrival rate}}{\text{Service rate}}$$

The value must be between 0 and 1, for otherwise there would be more tasks arriving than could be serviced, violating our assumption that the system is in equilibrium. Utilization is also called *traffic intensity* and is represented by the symbol ρ in many texts.

EXAMPLE Suppose an I/O system with a single disk gets about 10 I/O requests per second and the average time for a disk to service an I/O request is 50 ms. What is the utilization of the I/O system?

ANSWER The service rate is then

$$\frac{1}{50\text{ ms}} = \frac{1}{0.05\text{ sec}} = 20\text{ I/O per second (IOPS)}$$

Using the equation above,

$$\text{Server utilization} = \frac{\text{Arrival rate}}{\text{Service rate}} = \frac{10\text{ IOPS}}{20\text{ IOPS}} = 0.50$$

So the I/O system utilization is 0.5. ■

Little's Law can be applied to the components of the black box as well, since they must also be in equilibrium:

$$\text{Length}_{\text{queue}} = \text{Arrival rate} \times \text{Time}_{\text{queue}}$$

$$\text{Length}_{\text{server}} = \text{Arrival rate} \times \text{Time}_{\text{server}}$$

EXAMPLE Suppose the average time to satisfy a disk request is 50 ms and the I/O system with many disks gets about 200 I/O requests per second. What is the mean number of I/O requests at the disk server?

ANSWER Using the equation above,

$$\text{Length}_{\text{server}} = \text{Arrival rate} \times \text{Time}_{\text{server}} = \frac{200}{\text{sec}} \times 0.05\text{ sec} = 10$$

So there are 10 requests on average at the disk server. ■

How the queue delivers tasks to the server is called the *queue discipline*. The simplest and most common discipline is *first-in-first-out* (FIFO). If we assume FIFO we can relate time waiting in the queue to the mean number of tasks in the queue:

$$\text{Time}_{\text{system}} = \text{Length}_{\text{queue}} \times \text{Time}_{\text{server}} + \text{Mean time to complete service of tasks when new task arrives}$$

That is, the system response time is the number of tasks in the queue times the mean service time plus the time it takes the server to complete whatever tasks are being serviced when a new task arrives.

The last component of the equation is not as simple as it first appears. A new task can arrive at any instant, so we have no basis to know how long the existing task has been in the server. Although such requests are random events, if we know something about the distribution of events we can predict performance.

To estimate this answer we need to know a little about distributions of *random variables*. A variable is random if it takes one of a specified set of values with a specified probability; that is, you cannot know exactly what its next value will be, but you do know the probability of all possible values.

One way to characterize the distribution of values of a random variable is a *histogram*, which divides the range between the minimum and maximum values into subranges called *buckets*. Histograms then plot the number in each bucket as columns. Histograms work well for distributions that are discrete values—for example, the number of I/O requests. For distributions that are not discrete values, such as time waiting for an I/O request, we need a curve to plot the values over the full range so that we can accurately estimate the value. Stated alternatively, we need a histogram with an infinite number of buckets.

Hence, to be able to solve the last part of the equation above we need to characterize the distribution of this random variable. The mean time and some measure of the variance is sufficient for that characterization. For the first term we use the *weighted arithmetic mean time* (see page 26 in Chapter 1 for a slightly different version of the formula):

$$\text{Weighted mean time} = \frac{f_1 \times T_1 + f_2 \times T_2 + \ldots + f_n \times T_n}{\sum_{i=1}^{n} f_i}$$

where T_i is the time for task i and f_i is the frequency of occurrence of task i.

To characterize variability about the mean, many people use the standard deviation. Let's use the *variance* instead, which is simply the square of the standard deviation. Given the weighted mean, the variance can be calculated as

$$\text{Variance} = \frac{f_1 \times T_1^2 + f_2 \times T_2^2 + \ldots + f_n \times T_n^2}{\sum_{i=1}^{n} f_i} - \text{Weighted mean time}^2$$

The problem with using variance is that it can be large simply because of the units used to measure. Let's assume the distribution is of time. If the times are on the order of 100 milliseconds, then squaring them will lead to large variances on the order of 10,000 milliseconds; if instead the times were reported as 0.1 seconds, then squaring them would lead to small variances on the order of 0.01 seconds (10 milliseconds).

To avoid this unit problem, we use the *squared coefficient of variance*, traditionally called *C*:

$$C = \frac{\text{Variance}}{\text{Weighted mean time}^2}$$

For reasons stated earlier, we are trying to characterize random events, but to be able to predict performance we need random events with certain nice properties. Figure 6.23 gives a few examples. An *exponential distribution*, with most of the times short relative to average but with a few long ones, has a C value of 1. In a *hypoexponential distribution*, most values are close to average and C is less than 1. In a *hyperexponential distribution*, most values are further from the average and C is greater than 1. The disk service is best measured with a C of about 1.5. As we shall see, the value of C affects the simplicity of the queuing formulas.

Distribution type	C	% less than average	90% of distribution is less than
Hypoexponential	0.5	57%	2.0 times average
Exponential	1.0	63%	2.3 times average
Hyperexponential	2.0	69%	2.8 times average

FIGURE 6.23 Examples of value of squared coefficient of variance C and variability of distributions given an unlimited number of tasks (*infinite population*).

Note that we are using a constant to characterize variability about the mean. Since C does not vary over time, the past history of events has no impact on the probability of an event occurring now. This forgetful property is called *memoryless* and is a key assumption used to predict behavior.

Finally, we can answer the question about the length of time a new task must wait for the server to complete a task, called the *average residual service time*:

$$\text{Average residual service time} = 1/2 \times \text{Weighted mean time} \times (1 + C)$$

Although we won't derive this formula, we can appeal to intuition. When the distribution is not random and all possible values are equal to the average, the variance is 0 and so C is 0. The average residual service time is then just half the average service time, as we would expect.

EXAMPLE Using the definitions and formulas above, derive the average time waiting in the queue ($Time_{queue}$) in terms of the average service time ($Time_{server}$), server utilization, and the squared coefficient of variance (C).

ANSWER All tasks in the queue ($Length_{queue}$) ahead of the new task must be completed before the task can be serviced; each takes on average $Time_{server}$. If a task is at the server, it takes average residual service time to complete. The chance the server is busy is *server utilization*, hence the expected time for service is Server utilization × Average residual service time. This leads to our initial formula:

$$Time_{queue} = Length_{queue} \times Time_{server} + \text{Server utilization} \times \text{Average residual service time}$$

Replacing average residual service time by its definition and $Length_{queue}$ by Arrival rate × $Time_{queue}$ yields

$$Time_{queue} = \text{Server utilization} \times (1/2 \times Time_{server} \times (1 + C)) + (\text{Arrival rate} \times Time_{queue}) \times Time_{server}$$

Rearranging the last term, let us replace Arrival rate × $Time_{server}$ by Server utilization since

$$\text{Server utilization} = \frac{\text{Arrival rate}}{1/Time_{server}} = \text{Arrival rate} \times Time_{server}$$

It works as follows:

$$Time_{queue} = \text{Server utilization} \times (1/2 \times Time_{server} \times (1 + C)) + (\text{Arrival rate} \times Time_{server}) \times Time_{queue}$$
$$= \text{Server utilization} \times (1/2 \times Time_{server} \times (1 + C)) + \text{Server utilization} \times Time_{queue}$$

Rearranging terms and simplifying gives us the desired equation:

$$Time_{queue} = \text{Server utilization} \times (1/2 \times Time_{server} \times (1 + C)) + \text{Server utilization} \times Time_{queue}$$
$$Time_{queue} - \text{Server utilization} \times Time_{queue} = \text{Server utilization} \times (1/2 \times Time_{server} \times (1 + C))$$
$$Time_{queue} \times (1 - \text{Server utilization}) = \frac{\text{Server utilization} \times (Time_{server} \times (1 + C))}{2}$$
$$Time_{queue} = \frac{Time_{server} \times (1 + C) \times \text{Server utilization}}{2 \times (1 - \text{Server utilization})}$$

Note that when we have an exponential distribution, then C = 1.0, so this formula simplifies to

$$\text{Time}_{\text{queue}} = \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})}$$

■

These equations and this subsection are based on an area of applied mathematics called *queuing theory*, which offers equations to predict behavior of such random variables. Real systems are too complex for queuing theory to provide exact analysis, and hence queuing theory works best when only approximate answers are needed. This subsection is a simple introduction, and interested readers can find many books on the topic.

Requests for service from an I/O system can be modeled by a random variable, because the operating system is normally switching between several processes that generate independent I/O requests. We also model I/O service times by a random variable given the probabilistic nature of disks in terms of seek and rotational delays.

Queuing theory makes a sharp distinction between past events, which can be characterized by measurements using simple arithmetic, and future events, which are predictions requiring mathematics. In computer systems we commonly predict the future from the past; one example is least recently used block placement (see Chapter 5). Hence the distinction between measurements and predicted distributions is often blurred here, and we use measurements to verify the type of distribution and then rely on the distribution thereafter.

Let's review the assumptions about the queuing model:

- The system is in equilibrium.
- The times between two successive requests arriving, called the *interarrival times*, are exponentially distributed.
- The number of requests is unlimited (this is called an *infinite population model* in queuing theory).
- The server can start on the next customer immediately after finishing with the prior one.
- There is no limit to the length of the queue, and it follows the first-in-first-out order discipline.
- All tasks in line must be completed.

Such a queue is called *M/G/1*:

M = exponentially random request arrival (C = 1), with M standing for the memoryless property mentioned above

G = general service distribution (i.e., not exponential)

1 = single server

When interarrival times are exponentially distributed, this model becomes an *M/M/1* queue and we can use the simple equation for waiting time at the end of the last example. The M/M/1 model is a simple and widely used model.

The assumption of exponential distribution is commonly used in queuing examples for two reasons, one good and one bad. The good reason is that a collection of many arbitrary distributions acts as an exponential distribution. Many times in computer systems a particular behavior is the result of many components interacting, so an exponential distribution of interarrival times is the right model. The bad reason is that the math is simpler if you assume exponential distributions.

Let's put queuing theory to work in a few examples.

EXAMPLE Suppose a processor sends 10 disk I/Os per second, these requests are exponentially distributed, and the average disk service time is 20 ms. Answer the following questions:

1. On average, how utilized is the disk?
2. What is the average time spent in the queue?
3. What is the 90th percentile of the queuing time?
4. What is the average response time for a disk request, including the queuing time and disk service time?

ANSWER Let's restate these facts:

Average number of arriving tasks/second is 10.

Average disk time to service a task is 20 ms (0.02 sec).

The server utilization is then

$$\text{Server utilization} = \frac{\text{Arrival rate}}{\text{Service rate}} = \frac{10}{1/0.02} = 0.2$$

Since the service distribution is exponential, we can use the simplified formula for the average time spent waiting in line:

$$\text{Time}_{\text{queue}} = \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})} = 20\text{ ms} \times \frac{0.2}{1-0.2} = 20 \times \frac{0.2}{0.8} = 20 \times 0.25 = 5\text{ ms}$$

From Figure 6.23 (page 512), the 90th percentile is 2.3 times the mean waiting time, so it is 11.5 ms. The average response time is

$$\text{Time}_{\text{queue}} + \text{Time}_{\text{server}} = 5 + 20\text{ ms} = 25\text{ ms}$$

■

EXAMPLE Suppose we get a new, faster disk. Recalculate the answers to the questions above, assuming the disk service time is 10 ms.

ANSWER The disk utilization is then

$$\text{Server utilization} = \frac{\text{Arrival rate}}{\text{Service rate}} = \frac{10}{1/0.01} = 0.1$$

Since the service distribution is exponential, we can use the simplified formula for the average time spent waiting in line:

$$\text{Time}_{\text{queue}} = \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})} = 5 \text{ ms} \times \frac{0.1}{1-0.1} = 5 \times \frac{0.1}{0.9} = 5 \times 0.11 = 0.55 \text{ ms}$$

The 90th percentile of the mean waiting time is 1.27 ms.

The average response time is 10 + 0.55 ms or 10.55 ms, 2.4 times faster than the old response time even though the new service time is only 2.0 times faster. ■

Section 6.7 has more examples using queuing theory to predict performance.

Examples of Benchmarks of Disk Performance

The prior subsection tries to predict the performance of storage subsystems. We also need to measure the performance of real systems to collect the values of parameters needed for prediction, to determine if the queuing theory assumptions hold, and to suggest what to do if the assumptions don't hold.

This subsection describes three benchmarks, each illustrating novel concerns regarding storage systems versus processors.

Transaction Processing Benchmarks

Transaction processing (TP, or OLTP for on-line transaction processing) is chiefly concerned with *I/O rate*: the number of disk accesses per second, as opposed to *data rate*, measured as bytes of data per second. TP generally involves changes to a large body of shared information from many terminals, with the TP system guaranteeing proper behavior on a failure. If, for example, a bank's computer fails when a customer withdraws money, the TP system would guarantee that the account is debited if the customer received the money and that the account is unchanged if the money was not received. Airline reservations systems as well as banks are traditional customers for TP.

Two dozen members of the TP community conspired to form a benchmark for the industry and, to avoid the wrath of their legal departments, published the report anonymously [1985]. This benchmark, called *DebitCredit*, simulates bank tellers and has as its bottom line the number of debit/credit transactions per

second (TPS). The DebitCredit performs the operation of a customer depositing or withdrawing money. *TPC-A* and *TPC-B* are more tightly specified versions of this original benchmark. The organization responsible for standardizing TPC-A and TPC-B have also developed benchmarks on complex query processing (*TPC-C*) and decision support (*TPC-D*).

Disk I/O for DebitCredit is random reads and writes of 100-byte records along with occasional sequential writes. Depending on how cleverly the transaction-processing system is designed, each transaction results in between 2 and 10 disk I/Os and takes between 5000 and 20,000 CPU instructions per disk I/O. The variation depends largely on the efficiency of the transaction-processing software, although in part it depends on the extent to which disk accesses can be avoided by keeping information in main memory. Hence, TPC measures the database software as well as the underlying machine.

The main performance measurement is the peak TPS, under the restriction that 90% of the transactions have less than a two-second response time. The benchmark requires that for TPS to increase, the number of tellers and the size of the account file must also increase. Figure 6.24 shows this unusual relationship in which more TPS requires more users. This scaling is to ensure that the benchmark really measures disk I/O; otherwise a large main memory dedicated to a database cache with a small number of accounts would unfairly yield a very high TPS. (Another perspective is that the number of accounts must grow, since a person is not likely to use the bank more frequently just because the bank has a faster computer!)

TPS	Number of ATMs	Account file size
10	1000	0.1 GB
100	10,000	1.0 GB
1000	100,000	10.0 GB
10,000	1,000,000	100.0 GB

FIGURE 6.24 Relationship among TPS, tellers, and account file size. The DebitCredit benchmark requires that the computer system handle more tellers and larger account files before it can claim a higher transaction-per-second milestone. The benchmark is supposed to include "terminal handling" overhead, but this metric is sometimes ignored.

Another novel feature of TPC-A and TPC-B is that they address how to compare the performance of systems with different configurations. In addition to reporting TPS, benchmarkers must also report the cost per TPS, based on the five-year cost of the computer system hardware and software.

SPEC System-Level File Server (SFS) Benchmark

The SPEC benchmarking effort is best known for its characterization of processor performance, but it branches out into other fields as well. In 1990 seven companies agreed on a synthetic benchmark, called SFS, to evaluate systems running the Sun Microsystems network file service NFS. This synthetic mix was based on measurements on NFS systems to propose a reasonable mix of reads, writes, and file operations such as examining a file. SFS supplies default parameters for comparative performance: For example, half of all writes are done in 8-KB blocks and half are done in partial blocks of 1, 2, or 4 KB. For reads the mix is 85% full blocks and 15% partial blocks.

Like TPC-B, SFS scales the size of the file system according to the reported throughput: For every 100 NFS operations per second, the capacity must increase by 1 GB. It also limits the average response time, in this case to 50 ms. Figure 6.25 shows average response time versus throughput for three systems. Unfortunately, unlike TPC-B, SFS does not normalize for different configurations. The fastest system in Figure 6.25 has 12 times the number of CPUs and disks as the slowest system, but SPEC leaves it to you to calculate price versus performance.

Self-Scaling I/O Benchmark

A different approach to I/O performance analysis was proposed by Chen and Patterson [1994b]. The first step is a *self-scaling benchmark,* which automatically and dynamically adjusts *several* aspects of its workload according to the performance characteristics of the system being measured. By doing so, the benchmark automatically scales across current and future systems. This scaling is more general than the scaling found in TPC-B and SFS, for scaling here varies five parameters, according to the characteristics of the system being measured, rather than just one.

This first step aids in understanding system performance by reporting how performance varies according to each of five workload parameters. These five parameters determine the first-order performance effects in I/O systems:

1. *Number of unique bytes touched*—This is the number of unique data bytes read or written in a workload; essentially, it is the total size of the data set.
2. *Percentage of reads.*
3. *Average I/O request size*—It chooses sizes from a distribution with a coefficient of variance of (C) of one.
4. *Percentage of sequential requests*—This is the percentage of requests that sequentially follow the prior request. When set at 50%, on average half of the accesses are to the next sequential address.
5. *Number of processes*—This is the concurrency in the workload, that is, the number of processes simultaneously issuing I/O.

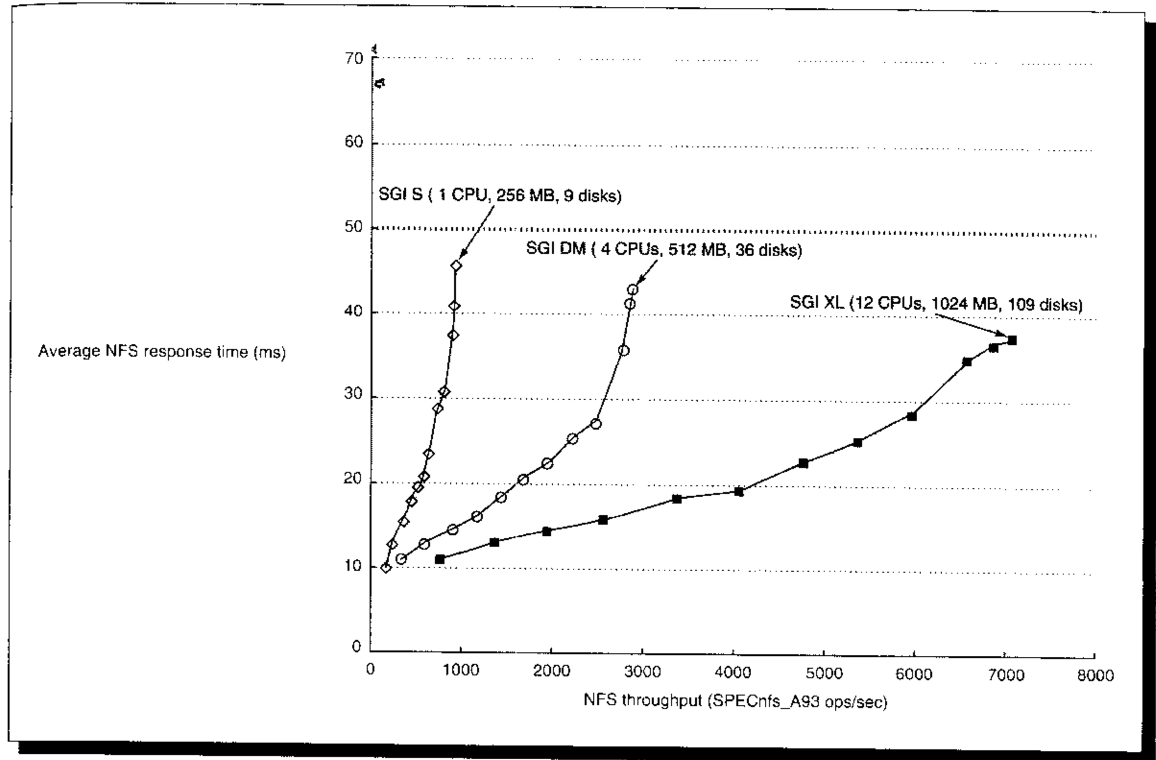

FIGURE 6.25 SPEC SFS performance for three SGI Challenge servers. The dashed line represents the 50-ms average response time limit imposed by SPEC. Reported in March 1995, these systems all ran IRIX version 5.3 with the EFS file system and all used the R4400 microprocessor. The XL model processors ran at 200 MHz and the other two used 150 MHz. Each system had one 1-GB disk with the rest being 2-GB disks, most spinning at 7200 RPM. SPEC SFS also divides the peak rate by 10 and calls this quotient SPECnsf_A93 users/second. The numbers of such users per second for these three machines are 84, 283, and 702, respectively.

The benchmark first chooses a nominal value for each of the five parameters based on the system's performance. It then varies each parameter in turn while the other four parameters remain at their fixed, nominal values. The one exception is the first parameter, since it determines whether all accesses go to the file cache or to disk. Because of the very different performance for file cache and disk accesses, the benchmark automatically picks two values for the number of bytes accessed.

The resulting I/O performance is then plotted for each of the parameters. Figure 6.26 shows the performance for workstations and mainframes, using the nominal parameter values collected by the self-scaling benchmark as a function of unique bytes touched. These plots give insight into appropriate workloads and resulting performance. The width of the high-performance parts of the curves is

determined by the size of the file cache. For example, the HP 730 offers the highest performance, provided the workload fits in its small file cache, and workloads that would need to go to disk on other systems can be satisfied by the very large file cache of the Convex.

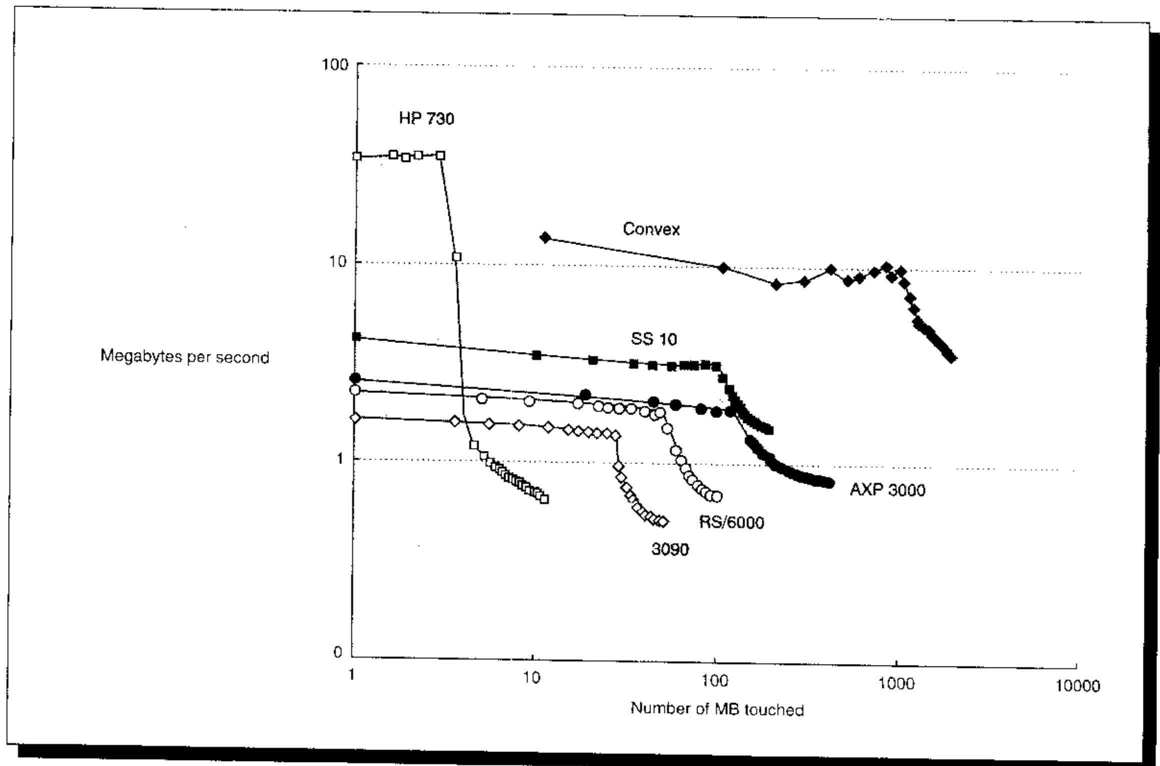

FIGURE 6.26 Performance versus megabytes touched for several workstations and mainframes (see section 6.8). Note the log-log scale. These results use the nominal values selected by the self-scaling benchmark. For example, 50% of accesses are reads and 50% are writes. The primary difference between the systems is the average access size of 120 KB for the Convex; adjusting for a common access size would halve Convex performance but make little change to the other lines in this plot.

The self-scaling benchmark increases our understanding of a system and scales the workload to remain relevant as technology advances. It complicates the task of comparing results from two systems, however. The problem is that the benchmark may choose different workloads on which to measure each system.

Hence, the second part of this new approach is to estimate the performance of other workloads. It estimates performance for unmeasured workloads by assuming

that the *shape* of a performance curve for one parameter is independent of the values of the other parameters. This assumption leads to an overall performance equation of

$$\mathrm{Perf}(X, Y, Z, \ldots) = \mathrm{Perf}(X_{nominal}, Y_{nominal}, Z_{nominal}, \ldots) \times f_X(X) \times f_Y(Y) \times f_Z(Z) \times \ldots$$

where X, Y, Z, ... are the parameters. Suppose the nominal values were 50% reads and 50% of accesses as sequential, but the desired workload had 60% reads and 60% sequential, and all other parameters matched the nominal values. The predicted performance is the nominal performance multiplied by the measured ratio of 60% reads to 50% reads and by the measured ratio at 60% sequential to 50% sequential. Chen and Patterson [1994b] have shown that this technique yields accurate performance estimates, within 10% for most workloads.

We use this benchmark to evaluate systems in section 6.8.

6.5 Reliability, Availability, and RAID

Although throughput and response time have their analogues in processor design, reliability is given considerably more attention in storage than in processors. This brings us to two terms that are often confused—*reliability* and *availability*. The term reliability is commonly used interchangeably with availability: if something breaks, but the user can still use the system, it seems as if the system still works and hence it seems more reliable. Here is a clearer distinction:

Reliability—Is anything broken?

Availability—Is the system still available to the user?

Adding hardware can therefore improve availability (for example, ECC on memory), but it cannot improve reliability (the DRAM is still broken). Reliability can only be improved by bettering environmental conditions, by building from more reliable components, or by building with fewer components. Another term, *data integrity*, refers to consistent reporting when information is lost because of failure; this is very important to some applications.

One innovation that improves both availability and performance of storage systems is *disk arrays*. The argument for arrays is that since price per megabyte is independent of disk size, potential throughput can be increased by having many disk drives and, hence, many disk arms. For example, Figure 6.25 (page 519) shows how NFS throughput increases as the systems expand from 9 disks to 109 disks. Simply spreading data over multiple disks, called *striping*, automatically forces accesses to several disks. (Although arrays improve throughput, latency is not necessarily improved.) The drawback to arrays is that with more devices, reliability drops: N devices generally have $1/N$ the reliability of a single device.