they did get more accurate information than did Gray, who relied on a form that the operator filled out and then sent up the management chain. The hardware/operating system went from causing 70% of the failures in VAX systems in 1985 to 28% in 1993, and failures due to operators rose from 15% to 52% in that same period. Murphy and Gent expected managing systems to be the primary dependability challenge in the future.

The final set of data comes from the government. The Federal Communications Commission (FCC) requires that all telephone companies submit explanations when they experience an outage that affects at least 30,000 people or lasts 30 minutes. These detailed disruption reports do not suffer from the self-reporting problem of earlier figures, as investigators determine the cause of the outage rather than operators of the equipment. Kuhn [1997] studied the causes of outages between 1992 and 1994, and Enriquez [2001] did a follow-up study for the first half of 2001. Although there was a significant improvement in failures due to overloading of the network over the years, failures due to humans increased, from about one-third to two-thirds of the customer-outage minutes.

These four examples and others suggest that the primary cause of failures in large systems today is faults by human operators. Hardware faults have declined due to a decreasing number of chips in systems and fewer connectors. Hardware dependability has improved through fault tolerance techniques such as memory ECC and RAID. At least some operating systems are considering reliability implications before adding new features, so in 2011 the failures largely occurred elsewhere.

Although failures may be initiated due to faults by operators, it is a poor reflection on the state of the art of systems that the processes of maintenance and upgrading are so error prone. Most storage vendors claim today that customers spend much more on managing storage over its lifetime than they do on purchasing the storage. Thus, the challenge for dependable storage systems of the future is either to tolerate faults by operators or to avoid faults by simplifying the tasks of system administration. Note that RAID 6 allows the storage system to survive even if the operator mistakenly replaces a good disk.

We have now covered the bedrock issue of dependability, giving definitions, case studies, and techniques to improve it. The next step in the storage tour is performance.

D.4 I/O Performance, Reliability Measures, and Benchmarks

I/O performance has measures that have no counterparts in 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 (namely, 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

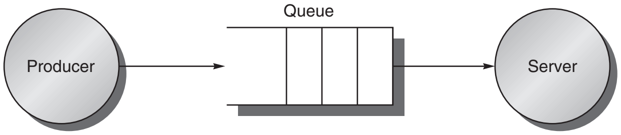

Figure D.8 The traditional producer-server model of response time and throughput. Response time begins when a task is placed in the buffer and ends when it is completed by the server. Throughput is the number of tasks completed by the server in unit time.

against each other. Figure D.8 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, thus the buffer should never be empty. Response time, on the other hand, counts time spent in the buffer, so an empty buffer shrinks it.

Another measure of I/O performance is the interference of I/O with processor 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 much longer a process will take because of I/O for another process.

Throughput versus Response Time

Figure D.9 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.

How does the architect balance these conflicting demands? If the computer is interacting with human beings, Figure D.10 suggests an answer. An interaction, or *transaction*, with a computer is divided into three parts:

1. *Entry time*—The time for the user to enter the command.
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. The results in Figure D.10 show that cutting system response time by 0.7 seconds saves 4.9 seconds (34%) from the conventional transaction and 2.0 seconds (70%) from

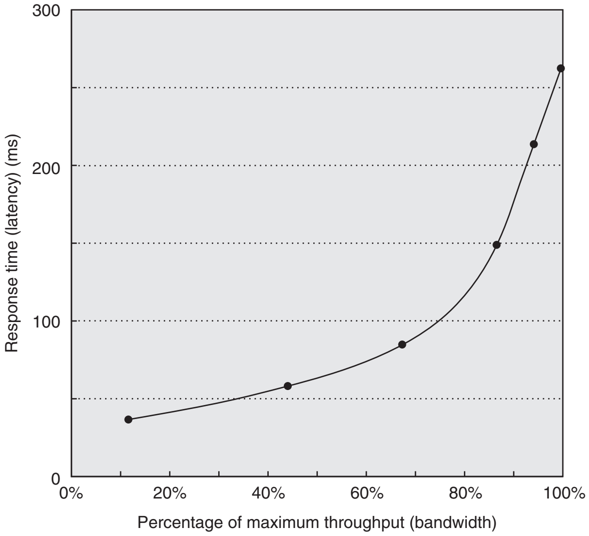

Figure D.9 Throughput versus response time. Latency is normally reported as response time. Note that the minimum response time achieves only 11% of the throughput, while the response time for 100% throughput takes seven times the minimum response time. Note also that the independent variable in this 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.

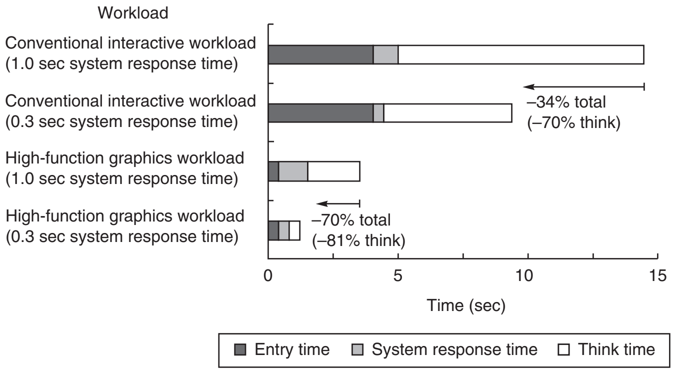

Figure D.10 A user transaction with an interactive computer divided into entry time, system response time, and user think time for a conventional system and graphics system. The entry times are the same, independent of system response time. The entry time was 4 seconds for the conventional system and 0.25 seconds for the graphics system. Reduction in response time actually decreases transaction time by more than just the response time reduction. (From Brady [1986].)

I/O benchmark	Response time restriction	Throughput metric
TPC-C: Complex Query OLTP	≥90% of transaction must meet response time limit; 5 seconds for most types of transactions	New order transactions per minute
TPC-W: Transactional Web benchmark	≥90% of Web interactions must meet response time limit; 3 seconds for most types of Web interactions	Web interactions per second
SPECsfs97	Average response time ≤40 ms	NFS operations per second

Figure D.11 Response time restrictions for three I/O benchmarks.

the graphics transaction. This implausible result is explained by human nature: People need less time to think when given a faster response. Although this study is 20 years old, response times are often still much slower than 1 second, even if processors are 1000 times faster. Examples of long delays include starting an application on a desktop PC due to many disk I/Os, or network delays when clicking on Web links.

To reflect the importance of response time to user productivity, I/O benchmarks also address the response time versus throughput trade-off. Figure D.11 shows the response time bounds for three I/O benchmarks. They report maximum throughput given either that 90% of response times must be less than a limit or that the average response time must be less than a limit.

Let's next look at these benchmarks in more detail.

Transaction-Processing Benchmarks

Transaction processing (TP, or OLTP for online 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. Suppose, for example, that a bank's computer fails when a customer tries to withdraw money from an ATM. 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.

As mentioned in Chapter 1, 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 [Anon. et al. 1985]. This report led to the *Transaction Processing Council*, which in turn has led to eight benchmarks since its founding. Figure D.12 summarizes these benchmarks.

Let's describe TPC-C to give a flavor of these benchmarks. TPC-C uses a database to simulate an order-entry environment of a wholesale supplier, including

Benchmark	Data size (GB)	Performance metric	Date of first results
A: debit credit (retired)	0.1–10	Transactions per second	July 1990
B: batch debit credit (retired)	0.1–10	Transactions per second	July 1991
C: complex query OLTP	100–3000 (minimum 0.07 * TPM)	New order transactions per minute (TPM)	September 1992
D: decision support (retired)	100, 300, 1000	Queries per hour	December 1995
H: ad hoc decision support	100, 300, 1000	Queries per hour	October 1999
R: business reporting decision support (retired)	1000	Queries per hour	August 1999
W: transactional Web benchmark	≈50, 500	Web interactions per second	July 2000
App: application server and Web services benchmark	≈2500	Web service interactions per second (SIPS)	June 2005

Figure D.12 Transaction Processing Council benchmarks. The summary results include both the performance metric and the price-performance of that metric. TPC-A, TPC-B, TPC-D, and TPC-R were retired.

entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. It runs five concurrent transactions of varying complexity, and the database includes nine tables with a scalable range of records and customers. TPC-C is measured in transactions per minute (tpmC) and in price of system, including hardware, software, and three years of maintenance support. Figure 1.17 on page 42 in Chapter 1 describes the top systems in performance and cost-performance for TPC-C.

These TPC benchmarks were the first—and in some cases still the only ones—that have these unusual characteristics:

- *Price is included with the benchmark results.* The cost of hardware, software, and maintenance agreements is included in a submission, which enables evaluations based on price-performance as well as high performance.
- *The dataset generally must scale in size as the throughput increases*. The benchmarks are trying to model real systems, in which the demand on the system and the size of the data stored in it increase together. It makes no sense, for example, to have thousands of people per minute access hundreds of bank accounts.
- *The benchmark results are audited*. Before results can be submitted, they must be approved by a certified TPC auditor, who enforces the TPC rules that try to make sure that only fair results are submitted. Results can be challenged and disputes resolved by going before the TPC.
- *Throughput is the performance metric, but response times are limited*. For example, with TPC-C, 90% of the new order transaction response times must be less than 5 seconds.

- *An independent organization maintains the benchmarks*. Dues collected by TPC pay for an administrative structure including a chief operating office. This organization settles disputes, conducts mail ballots on approval of changes to benchmarks, holds board meetings, and so on.

SPEC System-Level File Server, Mail, and Web Benchmarks

The SPEC benchmarking effort is best known for its characterization of processor performance, but it has created benchmarks for file servers, mail servers, and Web servers.

Seven companies agreed on a synthetic benchmark, called SFS, to evaluate systems running the Sun Microsystems network file service (NFS). This benchmark was upgraded to SFS 3.0 (also called SPEC SFS97_R1) to include support for NFS version 3, using TCP in addition to UDP as the transport protocol, and making the mix of operations more realistic. Measurements on NFS systems led to a synthetic mix of reads, writes, and file operations. 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-C, SFS scales the amount of data stored 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 40 ms. Figure D.13

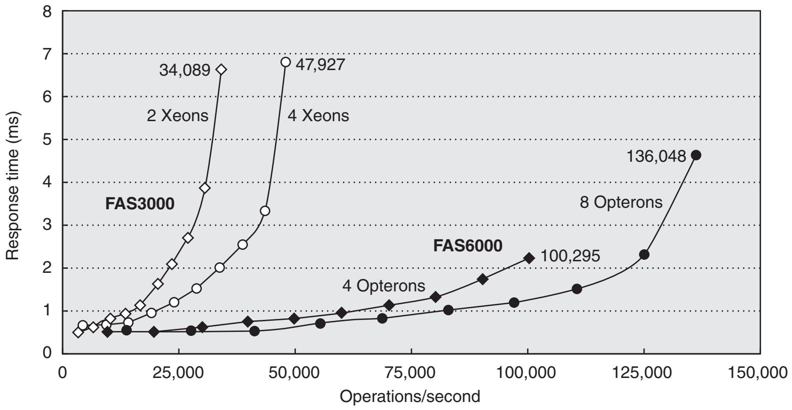

Figure D.13 SPEC SFS97_R1 performance for the NetApp FAS3050c NFS servers in two configurations. Two processors reached 34,089 operations per second and four processors did 47,927. Reported in May 2005, these systems used the Data ONTAP 7.0.1R1 operating system, 2.8 GHz Pentium Xeon microprocessors, 2 GB of DRAM per processor, 1 GB of nonvolatile memory per system, and 168 15 K RPM, 72 GB, Fibre Channel disks. These disks were connected using two or four QLogic ISP-2322 FC disk controllers.

shows average response time versus throughput for two NetApp systems. Unfortunately, unlike the TPC benchmarks, SFS does not normalize for different price configurations.

SPECMail is a benchmark to help evaluate performance of mail servers at an Internet service provider. SPECMail2001 is based on the standard Internet protocols SMTP and POP3, and it measures throughput and user response time while scaling the number of users from 10,000 to 1,000,000.

SPECWeb is a benchmark for evaluating the performance of World Wide Web servers, measuring number of simultaneous user sessions. The SPECWeb2005 workload simulates accesses to a Web service provider, where the server supports home pages for several organizations. It has three workloads: Banking (HTTPS), E-commerce (HTTP and HTTPS), and Support (HTTP).

Examples of Benchmarks of Dependability

The TPC-C benchmark does in fact have a dependability requirement. The benchmarked system must be able to handle a single disk failure, which means in practice that all submitters are running some RAID organization in their storage system.

Efforts that are more recent have focused on the effectiveness of fault tolerance in systems. Brown and Patterson [2000] proposed that availability be measured by examining the variations in system quality-of-service metrics over time as faults are injected into the system. For a Web server, the obvious metrics are performance (measured as requests satisfied per second) and degree of fault tolerance (measured as the number of faults that can be tolerated by the storage subsystem, network connection topology, and so forth).

The initial experiment injected a single fault—such as a write error in disk sector—and recorded the system's behavior as reflected in the quality-of-service metrics. The example compared software RAID implementations provided by Linux, Solaris, and Windows 2000 Server. SPECWeb99 was used to provide a workload and to measure performance. To inject faults, one of the SCSI disks in the software RAID volume was replaced with an emulated disk. It was a PC running software using a SCSI controller that appears to other devices on the SCSI bus as a disk. The disk emulator allowed the injection of faults. The faults injected included a variety of transient disk faults, such as correctable read errors, and permanent faults, such as disk media failures on writes.

Figure D.14 shows the behavior of each system under different faults. The two top graphs show Linux (on the left) and Solaris (on the right). As RAID systems can lose data if a second disk fails before reconstruction completes, the longer the reconstruction (MTTR), the lower the availability. Faster reconstruction implies decreased application performance, however, as reconstruction steals I/O resources from running applications. Thus, there is a policy choice between taking a performance hit during reconstruction or lengthening the window of vulnerability and thus lowering the predicted MTTF.

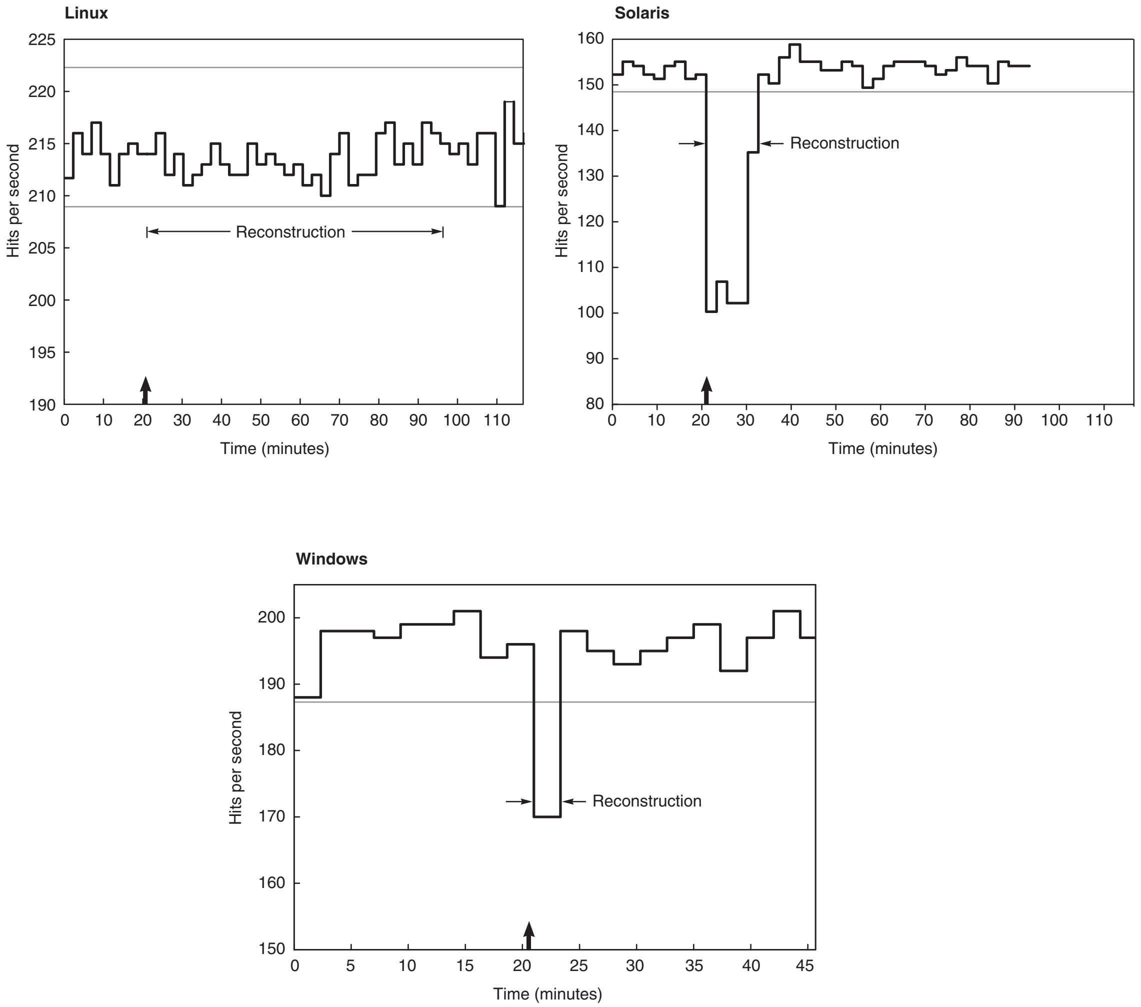

Figure D.14 Availability benchmark for software RAID systems on the same computer running Red Hat 6.0 Linux, Solaris 7, and Windows 2000 operating systems. Note the difference in philosophy on speed of reconstruction of Linux versus Windows and Solaris. The *y*-axis is behavior in hits per second running SPECWeb99. The arrow indicates time of fault insertion. The lines at the top give the 99% confidence interval of performance before the fault is inserted. A 99% confidence interval means that if the variable is outside of this range, the probability is only 1% that this value would appear.

Although none of the tested systems documented their reconstruction policies outside of the source code, even a single fault injection was able to give insight into those policies. The experiments revealed that both Linux and Solaris initiate automatic reconstruction of the RAID volume onto a hot spare when an active disk is taken out of service due to a failure. Although Windows supports RAID reconstruction, the reconstruction must be initiated manually. Thus, without human intervention, a Windows system that did not rebuild after a first failure remains susceptible to a second failure, which increases the window of vulnerability. It does repair quickly once told to do so.

The fault injection experiments also provided insight into other availability policies of Linux, Solaris, and Windows 2000 concerning automatic spare utilization, reconstruction rates, transient errors, and so on. Again, no system documented their policies.

In terms of managing transient faults, the fault injection experiments revealed that Linux's software RAID implementation takes an opposite approach than do the RAID implementations in Solaris and Windows. The Linux implementation is paranoid—it would rather shut down a disk in a controlled manner at the first error, rather than wait to see if the error is transient. In contrast, Solaris and Windows are more forgiving—they ignore most transient faults with the expectation that they will not recur. Thus, these systems are substantially more robust to transients than the Linux system. Note that both Windows and Solaris do log the transient faults, ensuring that the errors are reported even if not acted upon. When faults were permanent, the systems behaved similarly.

D.5 A Little Queuing Theory

In processor design, we have simple back-of-the-envelope calculations of performance associated with the CPI formula in Chapter 1, or we can use full-scale simulation for greater accuracy at greater cost. In I/O systems, we also have a bestcase analysis as a back-of-the-envelope calculation. Full-scale simulation is also much more accurate and much more work to calculate expected performance.

With I/O systems, however, we also have a mathematical tool to guide I/O design that is a little more work and much more accurate than best-case analysis, but much less work than full-scale simulation. Because of the probabilistic nature of I/O events and because of sharing of I/O resources, we can give a set of simple theorems that will help calculate response time and throughput of an entire I/O system. This helpful field is called *queuing theory*. Since there are many books and courses on the subject, this section serves only as a first introduction to the topic. However, even this small amount can lead to better design of I/O systems.

Let's start with a black-box approach to I/O systems, as shown in Figure D.15. In our example, the processor is making I/O requests that arrive at the I/O device, and the requests "depart" when the I/O device fulfills them.

We are usually interested in the long term, or steady state, of a system rather than in the initial start-up conditions. Suppose we weren't. Although there is a mathematics that helps (Markov chains), except for a few cases, the only way to solve the resulting equations is simulation. Since the purpose of this section is to show something a little harder than back-of-the-envelope calculations but less than simulation, we won't cover such analyses here. (See the references in Appendix M for more details.)

Arrivals
Departures

Figure D.15 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 system must equal the number of tasks leaving the system. This *flow-balanced* state is necessary but not sufficient for steady state. If the system has been observed or measured for a sufficiently long time and mean waiting times stabilize, then we say that the system has reached steady state.

Hence, in this section we make the simplifying assumption that we are evaluating systems with multiple independent requests for I/O service that are in equilibrium: The input rate must be equal to the output rate. We also assume there is a steady supply of tasks independent for how long they wait for service. In many real systems, such as TPC-C, the task consumption rate is determined by other system characteristics, such as memory capacity.

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. Note that the arrival rate and the response time must use the same time unit; inconsistency in time units is a common cause of errors.

Let's try to derive Little's law. Assume we observe a system for $\text{Time}_{\text{observe}}$ minutes. During that observation, we record how long it took each task to be serviced, and then sum those times. The number of tasks completed during $\text{Time}_{\text{observe}}$ is $\text{Number}_{\text{task}}$, and the sum of the times each task spends in the system is $\text{Time}_{\text{accumulated}}$. Note that the tasks can overlap in time, so $\text{Time}_{\text{accumulated}} \geq \text{Time}_{\text{observed}}$. Then,

$$\text{Mean number of tasks in system} = \frac{\text{Time}_{\text{accumulated}}}{\text{Time}_{\text{observe}}}$$

$$\text{Mean response time} = \frac{\text{Time}_{\text{accumulated}}}{\text{Number}_{\text{tasks}}}$$

$$\text{Arrival rate} = \frac{\text{Number}_{\text{tasks}}}{\text{Time}_{\text{observe}}}$$

Algebra lets us split the first formula:

$$\frac{\text{Time}_{\text{accumulated}}}{\text{Time}_{\text{observe}}} = \frac{\text{Time}_{\text{accumulated}}}{\text{Number}_{\text{tasks}}} \infty \frac{\text{Number}_{\text{tasks}}}{\text{Time}_{\text{observe}}}$$

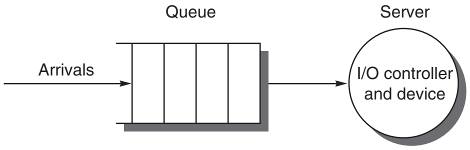

Figure D.16 The single-server model for this section. In this situation, an I/O request "departs" by being completed by the server.

If we substitute the three definitions above into this formula, and swap the resulting two terms on the right-hand side, we get Little's law:

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

This simple equation is surprisingly powerful, as we shall see.

If we open the black box, we see Figure D.16. The area where the tasks accumulate, waiting to be serviced, is called the *queue*, or *waiting line*. The device performing the requested service is called the *server*. Until we get to the last two pages of this section, we assume a single server.

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

- $\text{Time}_{\text{server}}$—Average time to service a task; average service rate is $1/\text{Time}_{\text{server}}$, traditionally represented by the symbol μ in many queuing texts.
- $\text{Time}_{\text{queue}}$—Average time per task in the queue.
- $\text{Time}_{\text{system}}$—Average time/task in the system, or the response time, which is the sum of $\text{Time}_{\text{queue}}$ and $\text{Time}_{\text{server}}$.
- Arrival rate—Average number of arriving tasks/second, traditionally represented by the symbol λ in many queuing texts.
- $\text{Length}_{\text{server}}$—Average number of tasks in service.
- $\text{Length}_{\text{queue}}$—Average length of queue.
- $\text{Length}_{\text{system}}$—Average number of tasks in system, which is the sum of $\text{Length}_{\text{queue}}$ and $\text{Length}_{\text{server}}$.

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 ($\text{Time}_{\text{queue}}$) or how long a task takes until it is completed ($\text{Time}_{\text{system}}$). The latter term is what we mean by response time, and the relationship between the terms is $\text{Time}_{\text{system}} = \text{Time}_{\text{queue}} + \text{Time}_{\text{server}}$.

The mean number of tasks in service ($\text{Length}_{\text{server}}$) is simply Arrival rate × $\text{Time}_{\text{server}}$, which is Little's law. Server utilization is simply the mean number of tasks being serviced divided by the service rate. For a single server, the service rate is $1/\text{Time}_{\text{server}}$. Hence, server utilization (and, in this case, the mean number of tasks per server) is simply:

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

Service utilization must be between 0 and 1; otherwise, there would be more tasks arriving than could be serviced, violating our assumption that the system is in equilibrium. Note that this formula is just a restatement of Little's law. Utilization is also called *traffic intensity* and is represented by the symbol ρ in many queuing theory texts.

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

Answer Using the equation above, with 10 ms represented as 0.01 seconds, we get: 50

$$\text{Server utilization} = \text{Arrival rate} \times \text{Time}_{\text{server}} = \frac{50}{\text{sec}} \times 0.01\,\text{sec} = 0.50$$

Therefore, the I/O system utilization is 0.5.

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:

$$\text{Time}_{\text{queue}} = \text{Length}_{\text{queue}} \times \text{Time}_{\text{server}} + \begin{array}{l}\text{Mean time to complete service of task when}\\ \text{new task arrives if server is busy}\end{array}$$

That is, the time in the queue is the number of tasks in the queue times the mean service time plus the time it takes the server to complete whatever task is being serviced when a new task arrives. (There is one more restriction about the arrival of tasks, which we reveal on page D-28.)

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.

Poisson Distribution of Random Variables

To estimate the last component of the formula 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 may know the probability of all possible values.

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.

One way to characterize the distribution of values of a random variable with discrete values 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 have two choices. Either we need a curve to plot the values over the full range, so that we can estimate accurately the value, or we need a very fine time unit so that we get a very large number of buckets to estimate time accurately. For example, a histogram can be built of disk service times measured in intervals of 10 μs although disk service times are truly continuous.

Hence, to be able to solve the last part of the previous equation we need to characterize the distribution of this random variable. The mean time and some measure of the variance are sufficient for that characterization.

For the first term, we use the *weighted arithmetic mean time*. Let's first assume that after measuring the number of occurrences, say, n_i, of tasks, you could compute frequency of occurrence of task i:

$$f_i = \frac{n_i}{\left(\sum_{i=1}^{n} n_i\right)}$$

Then weighted arithmetic mean is

$$\text{Weighted arithmetic mean time} = f_1 \times T_1 + f_2 \times T_2 + \ldots + f_n \times T_n$$

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, as it will help us with characterizing the probability distribution. Given the weighted arithmetic mean, the variance can be calculated as

$$\text{Variance} = \left(f_1 \times T_1^2 + f_2 \times T_2^2 + \ldots + f_n \times T_n^2\right) - \text{Weighted arithmetic mean time}^2$$

It is important to remember the units when computing variance. Let's assume the distribution is of time. If time is about 100 milliseconds, then squaring it yields 10,000 square milliseconds. This unit is certainly unusual. It would be more convenient if we had a unitless measure.

To avoid this unit problem, we use the *squared coefficient of variance*, traditionally called C^2:

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

We can solve for C, the coefficient of variance, as

$$C = \frac{\sqrt{\text{Variance}}}{\text{Weighted arithmetic mean time}} = \frac{\text{Standard deviation}}{\text{Weighted arithmetic mean time}}$$

We are trying to characterize random events, but to be able to predict performance we need a distribution of random events where the mathematics is tractable. The most popular such distribution is the *exponential distribution*, which has a C value of 1.

Note that we are using a constant to characterize variability about the mean. The invariance of C over time reflects the property that the history of events has no impact on the probability of an event occurring now. This forgetful property is called *memoryless*, and this property is an important assumption used to predict behavior using these models. (Suppose this memoryless property did not exist; then, we would have to worry about the exact arrival times of requests relative to each other, which would make the mathematics considerably less tractable!)

One of the most widely used exponential distributions is called a *Poisson distribution*, named after the mathematician Siméon Poisson. It is used to characterize random events in a given time interval and has several desirable mathematical properties. The Poisson distribution is described by the following equation (called the probability mass function):

$$\text{Probability}(k) = \frac{e^{-a} \times a^k}{k!}$$

where $a =$ Rate of events × Elapsed time. If interarrival times are exponentially distributed and we use the arrival rate from above for rate of events, the number of arrivals in a time interval t is a *Poisson process*, which has the Poisson distribution with $a =$ Arrival rate × t. As mentioned on page D-26, the equation for $\text{Time}_{\text{server}}$ has another restriction on task arrival: It holds only for Poisson processes.

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*, which again assumes Poisson arrivals:

$$\text{Average residual service time} = 1/2 \times \text{Arithemtic mean} \times \left(1 + \text{C}^2\right)$$

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 standard deviation is 0 and so C is 0. The average residual service time is then just half the average service time, as we would expect. If the distribution is random and it is Poisson, then C is 1 and the average residual service time equals the weighted arithmetic mean time.

Example Using the definitions and formulas above, derive the average time waiting in the queue ($\text{Time}_{\text{queue}}$) in terms of the average service time ($\text{Time}_{\text{server}}$) and server utilization.

Answer All tasks in the queue ($\text{Length}_{\text{queue}}$) ahead of the new task must be completed before the task can be serviced; each takes on average $\text{Time}_{\text{server}}$. 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:

$$\begin{aligned}\text{Time}_{\text{queue}} = {} & \text{Length}_{\text{queue}} \times \text{Time}_{\text{server}} \\ & + \text{Server utilization} \times \text{Average residual service time}\end{aligned}$$

Replacing the average residual service time by its definition and $\text{Length}_{\text{queue}}$ by Arrival rate × $\text{Time}_{\text{queue}}$ yields

$$\begin{aligned}\text{Time}_{\text{queue}} &= \text{Server utilization} \times \left[1/2 \times \text{Time}_{\text{server}} \times \left(1 + \text{C}^2\right)\right]\\ &\quad + \left(\text{Arrival rate} \times \text{Time}_{\text{queue}}\right) \times \text{Time}_{\text{server}}\end{aligned}$$

Since this section is concerned with exponential distributions, C^2 is 1. Thus

$$\text{Time}_{\text{queue}} = \text{Server utilization} \times \text{Time}_{\text{server}} + \left(\text{Arrival rate} \times \text{Time}_{\text{queue}}\right) \times \text{Time}_{\text{server}}$$

Rearranging the last term, let us replace $\text{Arrival rate} \times \text{Time}_{\text{server}}$ by Server utilization:

$$\begin{aligned}\text{Time}_{\text{queue}} &= \text{Server utilization} \times \text{Time}_{\text{server}} + (\text{Arrival rate} \times \text{Time}_{\text{server}}) \times \text{Time}_{\text{queue}}\\ &= \text{Server utilization} \times \text{Time}_{\text{server}} + \text{Server utilization} \times \text{Time}_{\text{queue}}\end{aligned}$$

Rearranging terms and simplifying gives us the desired equation:

$$\begin{aligned}\text{Time}_{\text{queue}} &= \text{Server utilization} \times \text{Time}_{\text{server}} + \text{Server utilization} \times \text{Time}_{\text{queue}}\\ \text{Time}_{\text{queue}} - \text{Server utilization} \times \text{Time}_{\text{queue}} &= \text{Server utilization} \times \text{Time}_{\text{server}}\\ \text{Time}_{\text{queue}} \times (1 - \text{Server utilization}) &= \text{Server utilization} \times \text{Time}_{\text{server}}\\ \text{Time}_{\text{queue}} &= \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})}\end{aligned}$$

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

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

If we substitute for $\text{Time}_{\text{queue}}$ from above, we get:

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

Since $\text{Arrival rate} \times \text{Time}_{\text{server}} = \text{Server utilization}$, we can simplify further:

$$\text{Length}_{\text{queue}} = \text{Server utilization} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})} = \frac{\text{Server utilization}^2}{(1 - \text{Server utilization})}$$

This relates number of items in queue to service utilization.

Example For the system in the example on page D-26, which has a server utilization of 0.5, what is the mean number of I/O requests in the queue?

Answer Using the equation above,

$$\text{Length}_{\text{queue}} = \frac{\text{Server uti1ization}^2}{(1 - \text{Server uti1ization})} = \frac{0.5^2}{(1 - 0.5)} = \frac{0.25}{0.50} = 0.5$$

Therefore, there are 0.5 requests on average in the queue.

As mentioned earlier, these equations and this section 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, hence queuing theory works best when only approximate answers are needed.

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 more sophisticated mathematics. In computer systems, we commonly predict the future from the past; one example is least recently used block replacement (see Chapter 2). Hence, the distinction between measurements and predicted distributions is often blurred; 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, which characterizes the arrival rate mentioned earlier.
- The number of sources of requests is unlimited. (This is called an *infinite population model* in queuing theory; finite population models are used when arrival rates vary with the number of jobs already in the system.)
- The server can start on the next job immediately after finishing the prior one.
- There is no limit to the length of the queue, and it follows the first in, first out order discipline, so all tasks in line must be completed.
- There is one server.

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

$M =$ exponentially random request arrival ($C^2 = 1$), with M standing for A. A. Markov, the mathematician who defined and analyzed the memoryless processes mentioned earlier

$M =$ exponentially random service time ($C^2 = 1$), with M again for Markov

$1 =$ single server

The M/M/1 model is a simple and widely used model.

The assumption of exponential distribution is commonly used in queuing examples for three reasons—one good, one fair, and one bad. The good reason is that a superposition 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 fair reason is that when variability is unclear, an exponential distribution with intermediate variability ($C = 1$) is a safer guess than low variability ($C \approx 0$) or high variability (large C). 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 40 disk I/Os per second, these requests are exponentially distributed, and the average service time of an older disk 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 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 40.

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

The server utilization is then

$$\text{Server utilization} = \text{Arrival rate} \times \text{Time}_{\text{server}} = 40 \times 0.02 = 0.8$$

Since the service times are exponentially distributed, we can use the simplified formula for the average time spent waiting in line:

$$\begin{aligned}\text{Time}_{\text{queue}} &= \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})}\\ &= 20 \text{ ms} \times \frac{0.8}{1 - 0.8} = 20 \times \frac{0.8}{0.2} = 20 \times 4 = 80 \text{ ms}\end{aligned}$$

The average response time is

$$\text{Time system} = \text{Time}_{\text{queue}} + \text{Time}_{\text{server}} = 80 + 20 \text{ ms} = 100 \text{ ms}$$

Thus, on average we spend 80% of our time waiting in the queue!

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

$$\text{Server utilization} = \text{Arrival rate} \times \text{Time}_{\text{server}} = 40 \times 0.01 = 0.4$$

The formula for the average time spent waiting in line:

$$\begin{aligned}\text{Time}_{\text{queue}} &= \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})}\\ &= 10 \text{ ms} \times \frac{0.4}{1 - 0.4} = 10 \times \frac{0.4}{0.6} = 10 \times \frac{2}{3} = 6.7 \text{ ms}\end{aligned}$$

The average response time is 10 + 6.7 ms or 16.7 ms, 6.0 times faster than the old response time even though the new service time is only 2.0 times faster.

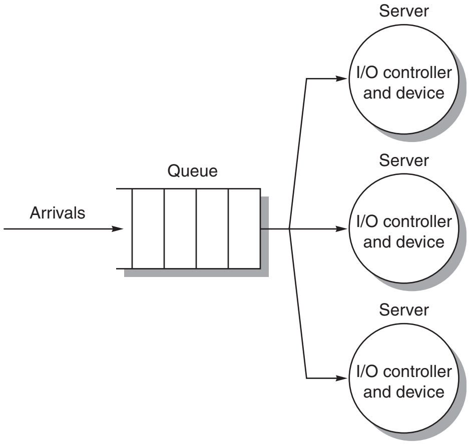
Server
I/O controller and device
Server
I/O controller and device
Server
I/O controller and device
Queue
Arrivals

Figure D.17 The M/M/m multiple-server model.

Thus far, we have been assuming a single server, such as a single disk. Many real systems have multiple disks and hence could use multiple servers, as in Figure D.17. Such a system is called an *M/M/m* model in queuing theory.

Let's give the same formulas for the M/M/m queue, using $\text{N}_{\text{servers}}$ to represent the number of servers. The first two formulas are easy:

$$\begin{aligned}\text{Utilization} &= \frac{\text{Arrival rate} \times \text{Time}_{\text{server}}}{\text{N}_{\text{servers}}}\\ \text{Length}_{\text{queue}} &= \text{Arrival rate} \times \text{Time}_{\text{queue}}\end{aligned}$$

The time waiting in the queue is

$$\text{Time}_{\text{queue}} = \text{Time}_{\text{server}} \times \frac{\text{P}_{\text{tasks} \geq \text{N}_{\text{servers}}}}{\text{N}_{\text{servers}} \times (1 - \text{Utilization})}$$

This formula is related to the one for M/M/1, except we replace utilization of a single server with the probability that a task will be queued as opposed to being immediately serviced, and divide the time in queue by the number of servers. Alas, calculating the probability of jobs being in the queue is much more complicated when there are $\text{N}_{\text{servers}}$. First, the probability that there are no tasks in the system is

$$\text{Prob}_{0\ \text{tasks}} = \left[1 + \frac{(\text{N}_{\text{servers}} \times \text{Utilization})^{\text{N}_{\text{servers}}}}{\text{N}_{\text{servers}}! \times (1 - \text{Utilization})} + \sum_{n=1}^{\text{N}_{\text{servers}}-1} \frac{(\text{N}_{\text{servers}} \times \text{Utilization})^{n}}{n!}\right]^{-1}$$

Then the probability there are as many or more tasks than we have servers is

$$\text{Prob}_{\text{tasks} \geq \text{N}_{\text{servers}}} = \frac{\text{N}_{\text{servers}} \times \text{Utilization}^{\text{N}_{\text{servers}}}}{\text{N}_{\text{servers}}! \times (1 - \text{Utilization})} \times \text{Prob}_{0\ \text{tasks}}$$

Note that if $N_{servers}$ is 1, $Prob_{task \geq N_{servers}}$ simplifies back to Utilization, and we get the same formula as for M/M/1. Let's try an example.

Example Suppose instead of a new, faster disk, we add a second slow disk and duplicate the data so that reads can be serviced by either disk. Let's assume that the requests are all reads. Recalculate the answers to the earlier questions, this time using an M/M/m queue.

Answer The average utilization of the two disks is then

$$\text{Server utilization} = \frac{\text{Arrival rate} \times \text{Time}_{\text{server}}}{\text{N}_{\text{servers}}} = \frac{40 \times 0.02}{2} = 0.4$$

We first calculate the probability of no tasks in the queue:

$$\begin{aligned}
\text{Prob}_{0\ \text{tasks}} &= \left[1 + \frac{(2 \times \text{Utilization})^2}{2! \times (1 - \text{Utilization})} + \sum_{n=1}^{1} \frac{(2 \times \text{Utilization})^n}{n!}\right]^{-1} \\
&= \left[1 + \frac{(2 \times 0.4)^2}{2 \times (1 - 0.4)} + (2 \times 0.4)\right]^{-1} = \left[1 + \frac{0.640}{1.2} + 0.800\right]^{-1} \\
&= [1 + 0.533 + 0.800]^{-1} = 2.333^{-1}
\end{aligned}$$

We use this result to calculate the probability of tasks in the queue:

$$\begin{aligned}
\text{Prob}_{\text{tasks} \geq \text{N}_{\text{servers}}} &= \frac{2 \times \text{Utilization}^2}{2! \times (1 - \text{Utilization})} \times \text{Prob}_{0\ \text{tasks}} \\
&= \frac{(2 \times 0.4)^2}{2 \times (1 - 0.4)} \times 2.333^{-1} = \frac{0.640}{1.2} \times 2.333^{-1} \\
&= 0.533/2.333 = 0.229
\end{aligned}$$

Finally, the time waiting in the queue:

$$\begin{aligned}
\text{Time}_{\text{queue}} &= \text{Time}_{\text{server}} \times \frac{\text{Prob}_{\text{tasks} \geq \text{N}_{\text{servers}}}}{\text{N}_{\text{servers}} \times (1 - \text{Utilization})} \\
&= 0.020 \times \frac{0.229}{2 \times (1 - 0.4)} = 0.020 \times \frac{0.229}{1.2} \\
&= 0.020 \times 0.190 = 0.0038
\end{aligned}$$

The average response time is 20 + 3.8 ms or 23.8 ms. For this workload, two disks cut the queue waiting time by a factor of 21 over a single slow disk and a factor of 1.75 versus a single fast disk. The mean service time of a system with a single fast disk, however, is still 1.4 times faster than one with two disks since the disk service time is 2.0 times faster.

It would be wonderful if we could generalize the M/M/m model to multiple queues and multiple servers, as this step is much more realistic. Alas, these models are very hard to solve and to use, and so we won't cover them here.

D.6 Crosscutting Issues

Point-to-Point Links and Switches Replacing Buses

Point-to-point links and switches are increasing in popularity as Moore's law continues to reduce the cost of components. Combined with the higher I/O bandwidth demands from faster processors, faster disks, and faster local area networks, the decreasing cost advantage of buses means the days of buses in desktop and server computers are numbered. This trend started in high-performance computers in the last edition of the book, and by 2011 has spread itself throughout storage. Figure D.18 shows the old bus-based standards and their replacements.

The number of bits and bandwidth for the new generation is per direction, so they double for both directions. Since these new designs use many fewer wires, a common way to increase bandwidth is to offer versions with several times the number of wires and bandwidth.

Block Servers versus Filers

Thus far, we have largely ignored the role of the operating system in storage. In a manner analogous to the way compilers use an instruction set, operating systems determine what I/O techniques implemented by the hardware will actually be used. The operating system typically provides the file abstraction on top of blocks stored on the disk. The terms *logical units*, *logical volumes*, and *physical volumes* are related terms used in Microsoft and UNIX systems to refer to subset collections of disk blocks.

Standard	Width (bits)	Length (meters)	Clock rate	MB/sec	Max I/O devices
(Parallel) ATA	8	0.5	133 MHz	133	2
Serial ATA	2	2	3 GHz	300	?
SCSI	16	12	80 MHz	320	15
Serial Attach SCSI	1	10	(DDR)	375	16,256
PCI	32/64	0.5	33/66 MHz	533	?
PCI Express	2	0.5	3 GHz	250	?

Figure D.18 Parallel I/O buses and their point-to-point replacements. Note the bandwidth and wires are per direction, so bandwidth doubles when sending both directions.