This page intentionally left blank.

Chapter 6

Analysis Techniques

But then chiefly do they disdain the unhallowed crowd, as often with their triangles, quadrangles, circles, and the like mathematical devices, more confounded than a labyrinth, and letters disposed one against the other, as it were in battle-array, they cast a mist before the eyes of the ignorant. Nor is there wanting of this kind some that pretend to foretell things by the stars, and make promises of miracles beyond all things of soothsaying, and are so fortunate as to meet people that believe them.

Desiderius Erasmus (1509)
on philosophers, from "The Praise of Folly"

6-0 INTRODUCTION

Throughout this book we have analyzed disks, file organizations, and transaction patterns. So we could proceed rapidly in the prior chapters, details of the methods used for the analyses have been ignored. Some of the techniques will now be covered in this chapter, always in the context of file design.

The analytical techniques applied when designing data-processing systems are derived from applications in areas beyond the subject of this book. Many important tools are available from the field of statistics. Statistics may have been applied first to agriculture, but the techniques can be used in all fields where events are frequent and varied. In order to apply statistical techniques, the distribution patterns of the events which are to be analyzed have to be understood. A typical collection of events are the transactions placed on the file system. The lengths of fields in variable-length records will also show a certain distribution pattern.

Section 6-1 discusses types of distributions which are frequently encountered and applies basic statistical tools to some file problems. Section 6-2 introduces simulation as a means to obtain performance estimates. A high level of demand for service leads to delays and generates waiting lines or queues which are discussed in Sec. 6-3. Some aspects of queuing theory and other disciplines from the area of operations research are touched upon in Sec. 6-4.

The objective of this chapter is to demonstrate the availability of well-developed tools which should be in the satchel of a serious computer system designer. To make the techniques easy to use, many algorithms are presented in programmed form.

The casual reader may skim over the topics in this chapter and proceed to the remainder of this book, which will address general issues of system design and operation.

6-1 STATISTICAL METHODS

Analysis and design often can make use of information collected from the observation of the operation of related systems. Statistics provides a means to summarize and simplify detailed data. The results which are obtained allow transfer of the experience to improved or new systems.

In order to apply statistical techniques to a process, a number of sequential activities must take place:

1 The process being analyzed has to be understood, so that a model can be built.
2 Parameters which affect the operation will be listed.
3 Data about the operation will be
   a Collected
   b Displayed with the relevant parameters to identify patterns
4 The cause of patterns seen in the data values will be explained in terms of the parameters used.
5 Transfer functions will be developed to permit the application of the collected information to new systems.
6 The validity of the transfer will be tested in the new setting.

We will touch upon these points while using examples from file system applications. The actual knowledge of statistics should be obtained from a course or textbooks in this area. The section is kept simple so that a background in statistics is not needed to follow the arguments.

Models and Parameters In the preceding analyses of file systems a number of basic parameters were used to explain the operation of files. The set of functions or logical rules using these parameters form the *model* for the systems being described. If the system being observed can be described with the parameters used throughout this book, then this description provides an adequate model. A model should be complete, that is, it should include all the variables that contribute to the observed results. In practice, completeness is difficult to achieve. We are happy if a model describes the result modestly well; in engineering sciences, obtaining 20% precision is often satisfactory. The remainder is the *unexplained* portion of the result, comprising all effects of variables which were not considered in the model.

An instance of the model is defined by setting parameters for all variables of the model. The basic operational parameters are often difficult to measure, and performance measurements taken often represent the combined effects of many parameters. The power of statistical techniques is that observations of complex and random phenomena can be analyzed if the underlying model is understood. The results obtained can be used for further system development, since arbitrary parameters can be submitted to the model and the expected results computed. Care must be taken that the parameters do not take on values for which the model is not valid.

6-1-1 Typical Demand Distributions

We analyzed many problems using averages. There is an inherent danger in these evaluations, since we must realize that average records and average files are as rare as average people. In most instances where averages were used, the measures we obtained were reasonably robust, that is, not greatly affected by a certain amount of variation from the mean or average value. Unless rigidly constrained, however, the observed values such as record lengths and field lengths, and frequencies of events such as queries will deviate from the mean.

Measurements of varying events are often profitably presented by graphical techniques, and explored in that form. Data presented as a histogram can show the frequencies of occurrence of events, classified by value type. We say that the histogram presents a *distribution* of occurrences. For this short summary all values on the ordinate are considered to be frequencies and values along the abscissa to be categorized values of events. If there is no variation, all events will fall into the same column of the histogram, and we have a *constant* distribution.

A number of possible distributions are shown in Fig. 6-1. We will describe conditions which lead to these distributions and then describe their behavior.

Data from file system operations can be sampled and plotted for data exploration. When frequencies of events are finely categorized, the graph will often have the shape of one of the popular distributions. Rarely will the match be exact, but if events show consistent patterns, the rules associated with the matching distribution type can be used to develop appropriate analysis tactics.

How Distributions Happen When we understand a process and its events we can often expect a characteristic distribution. Then it is possible to validate our expectations for correctness, and use this knowledge to help build a model. Before analyzing the distribution types individually we will consider the type of events occurring in file systems which lead to certain distributions.

Uniform distributions arise when all events being considered are equally likely to occur. This means they are obtained when simple independent events are categorized. Uniformity may be obtained through truncation of high-order effects. For instance, the latency time when accessing data within a cylinder of a disk, given that the time between accesses, c, is many times greater than the revolution time for the disk, $2r$, is apt to be uniformly distributed.

Uniformity is often a desirable feature of file system processes. Uniformity of record distribution into a direct file minimizes interference and collisions. The trun-

cation of high-order effects is the idea behind the key-to-address transformations discussed in Chap. 3-5, typified by the remainder-of-division method.

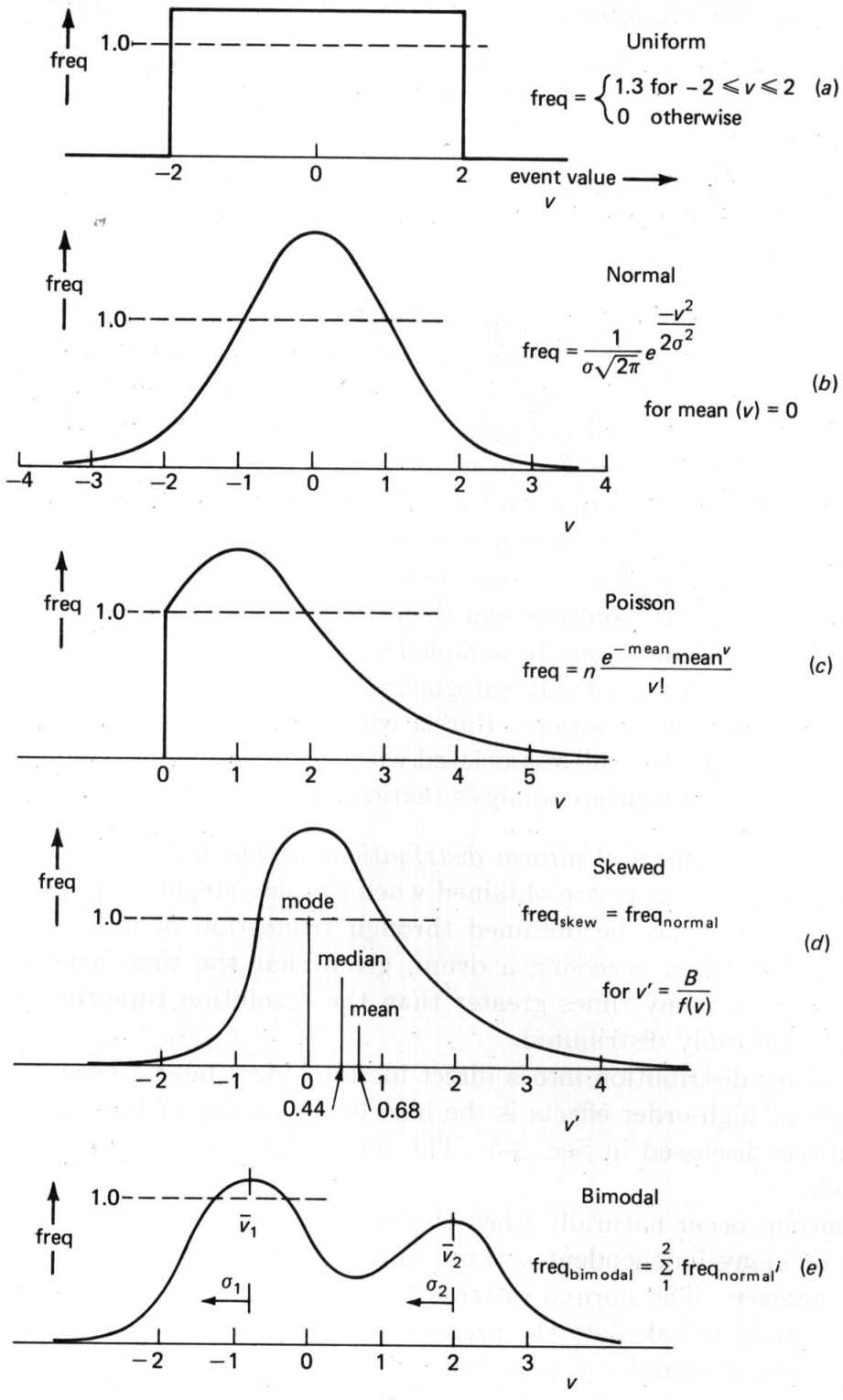


Figure 6-1 Popular distributions.

Normal distributions occur naturally when the events being measured are the sums and products of many independent actions, and the events have a constant probability, p, of occurrence. The normal distribution is, in fact, the limit of the binomial distribution, used to calculate the probabilities of discrete events, which is reached as the number of samples, f, tends to infinity. If the product $fp > 15$, the normal curve becomes an adequate approximation.

Consider placing into buckets the catenation of a fixed number, f, of records of two different sizes which occur with equal probability ($p = 0.5$). If $f > 30$, the expected length of the catenation will be normally distributed. Figure 6-2 shows the distribution developing for three distinct sizes. If the distribution of the record length varies uniformly, the normal distribution is an adequate approximation with even fewer records per block.

To visualize what happens as a distribution becomes normal, we use a record type which can be 30, 40, or 50 bytes long, with equal probability, and then construct sequences composed of three of these records. The sequence can be from 90 to 150 bytes long, each of the 27 possible record arrangements has been drawn on the left. Each sequence has equal probability. Next to them is the distribution of record lengths ($p = {}^1\!/_3, {}^1\!/_3, {}^1\!/_3$) and of the total lengths ($p = {}^1\!/_2 7, {}^3\!/_2 7, {}^6\!/_2 7, {}^7\!/_2 7, {}^6\!/_2 7, {}^3\!/_2 7, {}^1\!/_2 7$). Even with this small aggregation of values from a discrete uniform distribution, the total begins to look like a normal distribution. The final shape, for larger sets of values, was shown as Fig. 6-1b.

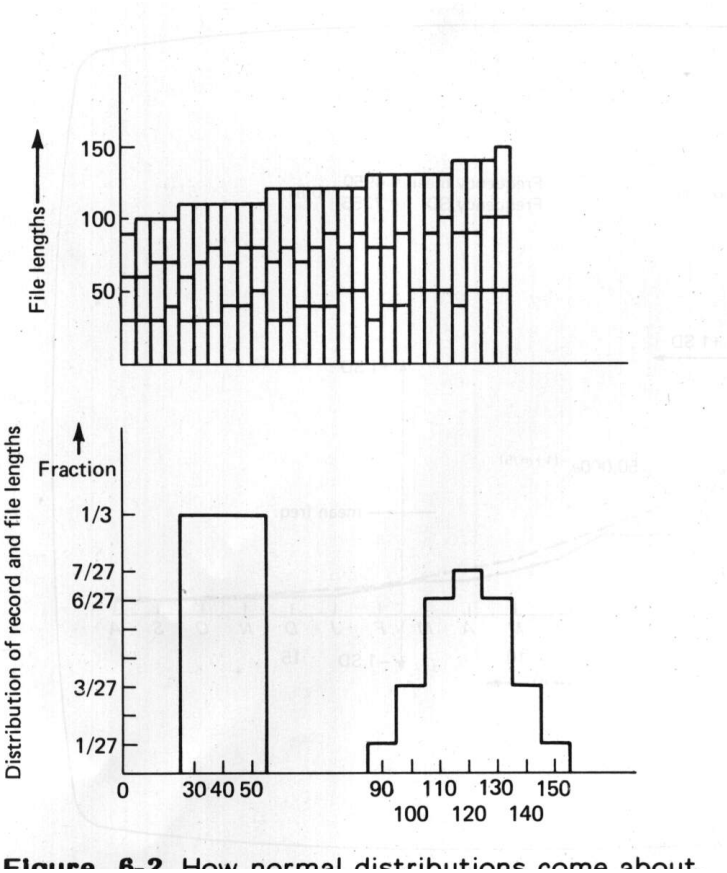


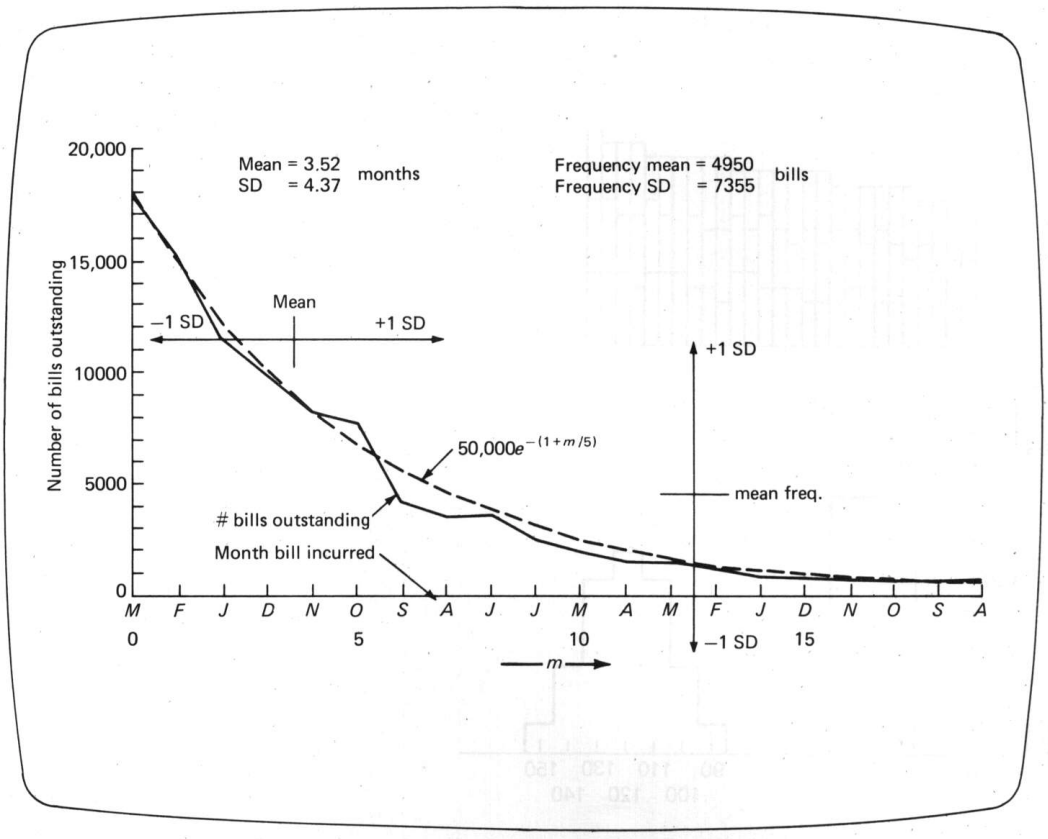
Figure 6-2 How normal distributions come about

Catenations of records whose lengths are symmetrically distributed according to any patter will rapidly become normally distributed. If the record lengths are already normally distributed, their catenations will be always normally distributed.

Exponential distributions occur when the probability of values of a series of event types increases or decreases rapidly as the event values increase. Increasing exponentials are difficult to manage in models and in reality. An example of such disconcerting behavior in a database is the number of interconnection possibilities among its elements, and hence the time required for thorough checking of a complex database.

Negative exponentials decay initially quite rapidly, but then reach their asymptotic value of zero slowly. An example is seen in Fig. 6-3. Distributions of a similar shape arise when measurements of terminal requests and of access frequencies to records of files are made. These are best described by the Erlang and Zipf distributions noted in Sec. 6-1-6.

Figure 6-3 shows the distribution by date of charges for hospital services which have not yet been settled. Bills older than 18 months (August a year past) are either turned over to a collection agency or written off. A total of 94 048 bills were outstanding. The distribution has the form of a negative exponential, as shown by the curve fitted to the data using visual comparison on a display screen. The long tail of this distribution is handled through the imposition of the arbitrary cutoff date.



A negative exponential distribution is typically generated by a succession of mutually dependent events of the same probability. In Figure 6-3 it appears that the probability of a customer paying an outstanding bill is about 20% every month. Negative exponential distributions are also found when the times between random arrivals of service requests are displayed. Exponential and Erlang distributions are important when queues of service requests are analyzed.

Poisson distributions occur when important contributing events have a small probability p. This distribution is the limit of the binomial distribution for the case where the likelihood of a particular event is small. The Poisson distribution becomes a valid approximation when the number of events $f > 25$ while $fp = mean < 5$. When the mean becomes large ($fp > 30$), the peak is sufficiently away from the left boundary that a normal distribution can be used to approximate the Poisson distribution.

For a moderate sample size, the Poisson distribution diminishes more rapidly than the exponential to the asymptotic value of zero. A Poisson distribution is typically caused by independent events, whereas an exponential distribution occurs more often where successive events are related. Poisson distributions which have a mean value of less than one do not show the initial rise shown in Fig. 6-1 but decline immediately and hence look similar to exponential distributions.

Example 6-1 Using a Poisson distribution.

We wish to model overflows due to insertions in an indexed-sequential file organization for a file of 400 blocks. We expected to insert 550 records between reorganizations. The probability of a block being updated is only 1/400 per insertion, and fp is 550/400 = 1.38. With this information we know that a Poisson distribution is likely. This means the distribution curve will be as shown in Fig. 6-1c. The intersection at freq 1.0 gives the relative count fr_0 of 0 insertions in a block. About the same number will receive two insertions (fr_2), but most blocks ($fr_1 = 1.38fr_0$) will receive one insertion. Some blocks will receive up to 6 insertions. The specific details are worked out in Example 6-5.

It can be shown that, for events which have an exponential interarrival-time distribution, the number of expected arrivals in a given time period has a Poisson distribution.

Skewed distributions occur whenever nonlinear transformations affect the outcome of events. A distribution as shown in Fig. 6-1d can represent the number of records, *Bfr*, in a block. Since the operation $Bfr = B/R$ is not linear for R, the normal distribution of record sizes, R, is distorted. Skewed distributions are characterized by the fact that their mean occurs at a different point than their median; the median is the value with an equal number of observations to either side; in Fig. 6-1d the mean is at 0.63, but the median is 0.44. Yet another measure of *central tendency* is the mode, the position of the most frequent value, here at zero. For moderately skewed distributions,

$$(mean - mode) \approx 3(mean - median) \qquad 6\text{-}1$$

This relationship can provide a quick estimate of the median from the mode and the mean, which are easier to compute.

Bimodal distributions occur when the events being measured are due to two separate, underlying phenomena. An example is the response time for queries, where some can be answered by using an index and others require an exhaustive reading of the file or a subset of the file.

Cumulative Distribution Function Often a frequency distribution is used to obtain an estimate of the fraction of cases exceeding a certain limit. For each distribution shown a *cumulative distribution function (c.d.f.)* can be obtained by summing or integrating the frequency distribution from left to right over time. The height of the curve is directly proportional to the number of events occurring with values less than the corresponding value. A few c.d.f.s are shown with their source functions in Fig. 6-4. Table 6-7 and Fig. 6-15 use c.d.f.s to predict the fraction of desirable or undesirable events.

These cumulative distributions may also appear in the system measurements, since often only the aggregate effect is observed. Differencing or differentiation of these functions can be used to re-create frequency distributions. Skew, multiple modes, etc., are more difficult to recognize when they occur in c.d.f.s instead of in frequency distributions.

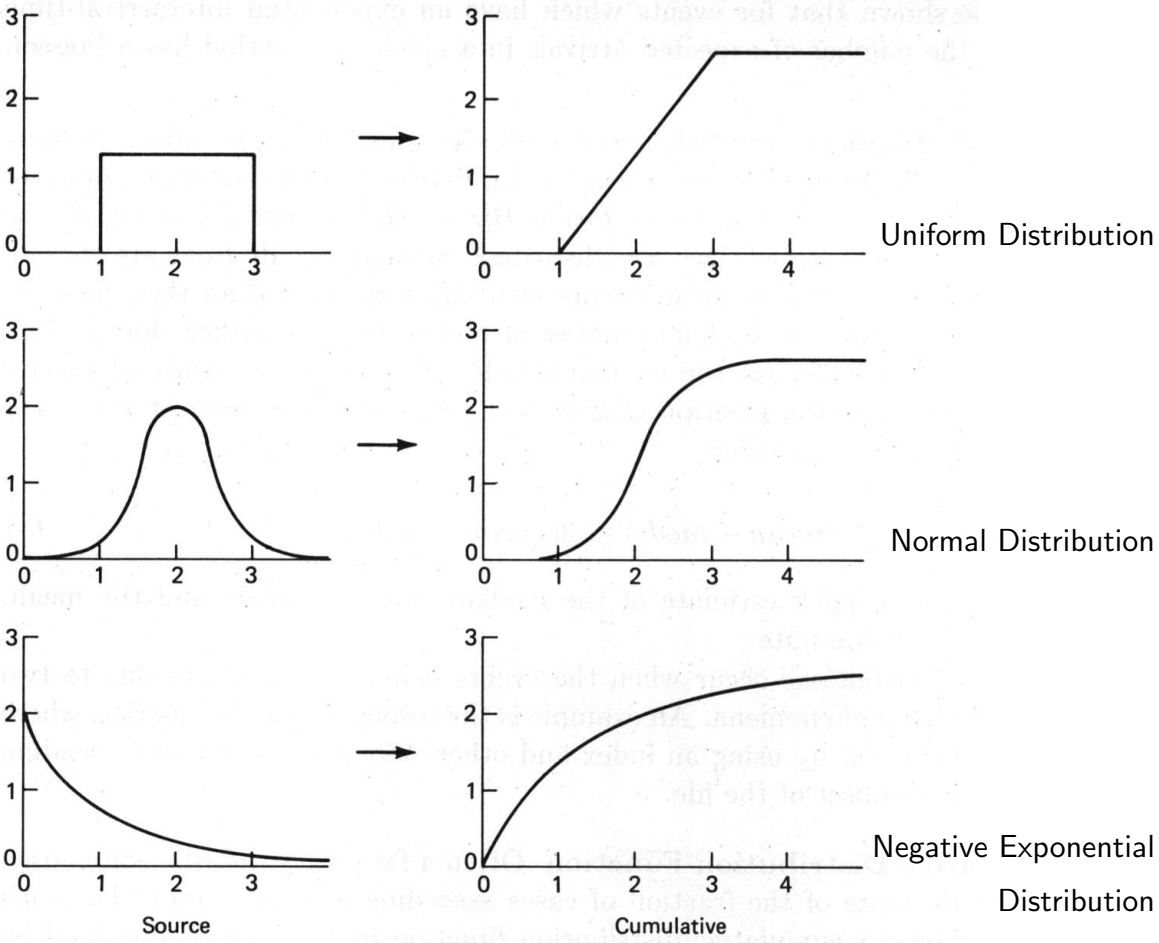


Figure 6-4 Cumulative distributions.

6-1-2 Describing Distributions

When the result of a measurement generates a distribution, a similarity to one of the distribution types shown will provide some clues about the process causing this distribution. As a next step, some quantitative measures can be obtained. Two basic parameters useful with nearly any observed distribution are: the *mean* `xbar` and the standard deviation *sigma* σ of the observations, given `f` samples `x(1),x(2),...,x(f)`. In order to have an unbiased estimate, these sample observations should be a random sample of the events. For the standard deviation we will use the symbol σ in equations, but `sd` in program examples.

For any collection of observed events we can compute the *mean* and σ parameters:

Table 6-1 Mean and Standard Deviation of an array of values.

```
/* Mean (xbar) and Standard Deviation (sd) of array x */   /* 6-1 */
       xsum,xsqsum = 0;
       DO i = 1 TO f;
          xsum = xsum+x(i);
          xsqsum = xsqsum+x(i)**2;
       END;
       xbar = xsum/f;
       sd = SQRT((xsqsum-xsum**2/f)/(f-1));
       ...
```

The number of samples, `f`, should be large enough so that we can have confidence that the sample set used is representative of the events occurring within the system. Any of the statistics texts referenced can be consulted for tests which provide measures of confidence. A *chi-square test* will be used in an example in Sec. 6-1-5.

6-1-3 Uniform Distribution

If a distribution looks flat or uniform over a range of values, many estimation tasks are simplified. A uniform distribution of events has often been assumed in examples in the earlier chapters, since the uniform distribution often gives a relatively poor, and therefore conservative result. A uniform distribution of addresses of seek requests to a disk will cause a higher average seek time than all but a bimodal distribution. The probability that a next record is not available in the current block is greater for a uniform distribution of requests than for any other. The average cost, however, of following overflow chains is lower when update insertions were distributed uniformly and greater for nonuniform update distributions.

Table 6-2 Parameters of the uniform distribution.

```
/* Parameters of the uniform distribution */              /* 6-2 */
       height = f/number_of_categories;
       range = MAX(x) - MIN(x);
```

A uniform distribution is described by its height, that is, the frequency of occurrence of categorized events, and by its range, which has to be finite for a finite number of events as shown in the program segment in Table 6-2.

To visualize the uniformity of the distribution the size and value for each category `icn = 1, ..., number_of_categories` is computed in Table 6-3. `CEIL` and `FLOOR` functions are used to improve the presentation of the information in a histogram.

Table 6-3 Create histogram categories for the frequency distribution.

```
/* Create histogram categories for frequency distribution     6-3 */
      size_of_category = CEIL(range/(number_of_categories-1));
      base = FLOOR(MIN(x));
      category_low_value = base + (icn-1)*size_of_category;
```

If an investigation is made to determine the amount of cylinder overflow for a direct or indexed-sequential file, then f may be the number of records, and the `number_of_categories` may be the number of cylinders allocated to the file. The expected mean and standard deviation of samples which present a perfectly uniform distribution are

Table 6-4 Expected `mean` and `sd` of a uniform distribution.

```
/* mean and sd of a uniform distribution */                 /* 6-4 */
      xbar_uniform = range/2;
      sd_uniform = SQRT(range**2/12);
```

If the observed `xbar` and `sd` from the program segment in Table 6-1 do not match these values well, other distributions should be considered.

Even if the parameters match well, the observed frequencies will vary about the expected `height`, so that it may be desirable to analyze how good the fit is. In the example, variations can cause the capacity of some cylinders to be exceeded. The frequency distribution can be plotted from an array, `fx`, filled from the observed values, `x`, as follows:

Table 6-5 Computation of the frequency histogram of observed values.

```
/* Compute the frequency histogram of observed values */   /* 6-5 */
      DECLARE fx(number_of_categories);
      fx = 0;
      DO i = 1 TO f;
         ifr = CEIL((x(i)-base)/size_of_category);
         fx(ifr) = fx(ifr) + 1;
      END;
```

Computing the standard deviation of the frequencies `sdfreq` and obtaining a value which is small when compared to the height in the graph, the mean frequency, indicate a satisfactory degree of uniformity. We will apply these concepts to a problem posed by the need to combine rapid access with sequential access.

Example 6-2 Creating a uniform distribution for direct access.

In order to use fast direct access to the billing file of Fig. 6-3 for frequent inquiries, while keeping the file in chronological order, a sequentiality-maintaining, exponential key-to-address transformation, based on the inverse of the cumulative distribution function, was investigated. The file was to be stored on 100 cylinders, each with a capacity of 1197 records, so that $m = 119\,700$, for $n = 94\,048$. The key-to-address algorithm computes a cylinder address for a given date, taking into account the weekends, which have lower billing rates. The name of the person billed provides the record address within the area allocated to the date. Figure 6-5 shows the distribution of records over cylinders.

The expected frequency is 940.5 per cylinder. The resulting distribution is quite uniform, with some irregularities due to the fact that the low-order cylinders are affected by the many recent bills generated on single days and also by irregular billings from some holidays. Only one cylinder overflow occurs with this sample. Further analysis is needed to determine the rate of block overflow, given the distribution of names and buckets of 63 records using blocks of $14\,000$ bytes. The results are shown in Fig. 6-5.

Hashing provides the solution for rapid access. We create a specialized key-to-address transformation which is *sequence-maintaining*. For reference the family tree of hash functions, presented in Figure 3-18, may be useful.

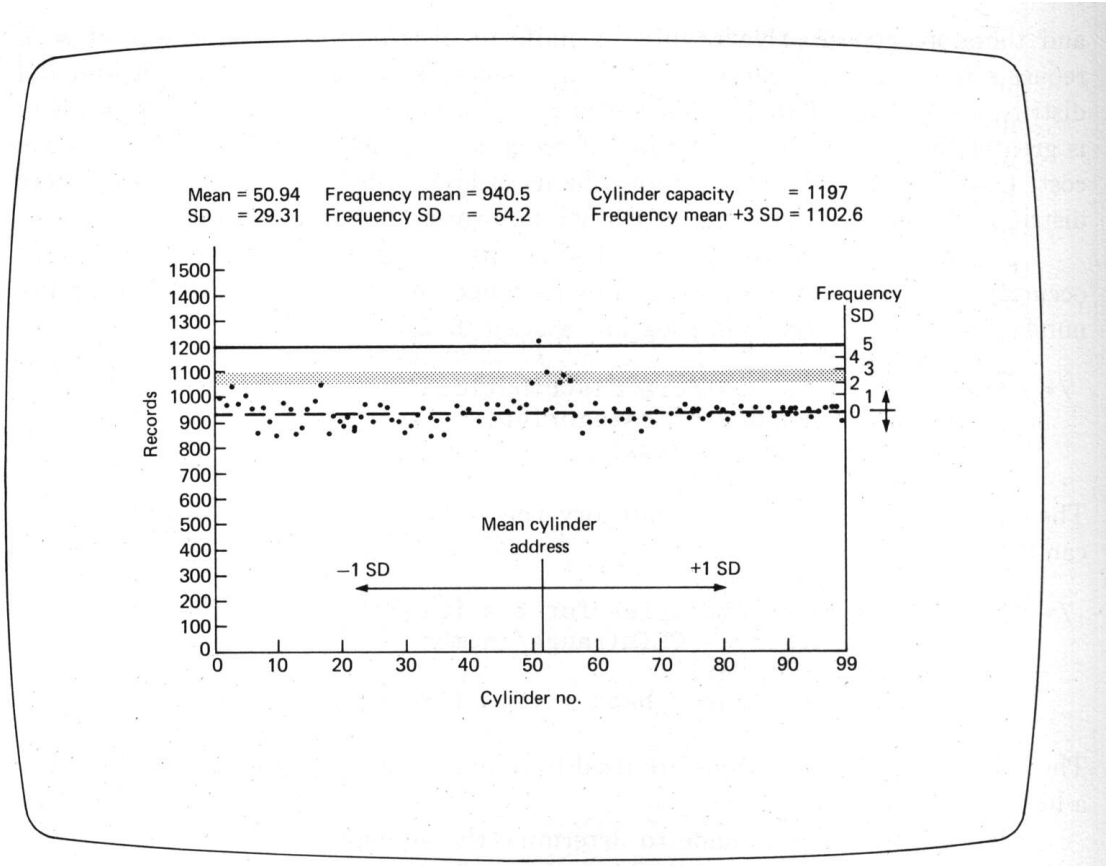


Figure 6-5 Test of uniformity of an algorithmic key-to-address transform.

6-1-4 Normal Distribution

The difference of observed and expected frequencies can often be expected to be a normally distributed variable, since it is based on many independent samples from the observed events.

The parameters which describe a normal-looking distribution are the mean value of events `xbar` and the standard deviation `sd`. They can be computed from an array `x` as shown in Table 6-1. Using these two parameters from the observed values, an expected normal distribution can be generated for comparison purposes. Using the equation for the normal curve, programmed for `f` observations, we obtain values in array `xnf` which should match the histogram values `fx`, computed from `x`, if the distribution was indeed normal.

Table 6-6 Expected normal frequency distribution function `xnf`.

```
/* Expected Normal Frequency Distribution Function xnf */  /* 6-6 */
       DECLARE xnf(number_of_categories);
       scalefactor = f/(sd*2.5066283);  /* 2.5066283 = sqrt(2*pi) */
       shapefactor = -0.5/sd**2;
       DO i = 1 TO number_of_categories;
          category_center_value = base + (icn-0.5)*size_of_category;
          xnf(icn) = scalefactor*EXP(shapefactor*
                 (category_center_value-xbar) **2);
       END;
```

The observed frequencies at `fx(i)` can be compared with the values `xnf(i)` generated by this function.

If a distribution is approximately normal, many useful estimation rules can be applied. In file design, it is often desirable to estimate how often a certain limit will be exceeded; for a distribution which is approximately normal, the cumulative area of the normal curve beyond the limit provides the desired estimate. Table 6-7 lists some of these values in terms of `t`, the difference between the mean and the limit as a ratio of the standard deviation, σ.

Table 6-7 Normal one-tail probabilities.

t	p($value > limit$)
0	0.500
0.25	0.401
0.50	0.309
0.75	0.227
1.00	0.159
1.25	0.106
1.50	0.067
1.75	0.040
2.00	0.023
2.25	0.012
2.50	0.006
2.75	0.003
3.00	0.001 3
4.00	0.000 03
5.00	0.000 000 7

$limit = mean + \mathtt{t}\,\sigma$

or

`t = (limit-mean)/sd`

Infrequent Events Table 6-7 shows that events of value greater than the $mean+3\sigma$ are quite rare.

Applying these values to Example 6-2, we first tabulate the frequencies. We obtain a mean of 940.5 records per cylinder with a σ of 54.2. It takes $t = +4.73\sigma$ to exceed the cylinder capacity of 1197. For a normal distribution this is expected to happen less than once in a million cases (Table 6-7), but the distribution of bills per cylinder was not quite normal, due to seasonal variations. In practice one overflow was found in 100 cylinders, as shown earlier in Example 6-1.

Even when data is truly normally distributed, rare events can, of course, still happen; in fact, the high activity rates in data-processing systems can cause rare events to occur with annoying regularity.

Block Overflow Due to Record Variability When the normal distribution was introduced, the statement was made that distributions, when summed, become more normal. This phenomenon is known as the *the central limit theorem*. In the example which follows that rule will be applied to the problem of packing records into blocks for the tree-structured file evaluated in Chap. 4-4-1, Example 4.8(1b).

In order to pack fixed-length records effectively into one block, the capability of a block in terms of record segments was computed.

$$n_{block} = y - 1 = \left\lfloor \frac{B - P}{R + P} \right\rfloor \qquad 6\text{-}2$$

A certain number of characters per block remained unused. We will name this excess G' and find that

$$G' = B - P - n_{block}(R + P) \qquad 6\text{-}3$$

These files are designed to always place n_{block} entries into a block, so that $G' < R$. The file-space usage for fixed-length records was most efficient if $G' = 0$.

If the records are actually of variable length, the distribution of R has to be described. A normal distribution may be a reasonable assumption if the average length is sufficiently greater than the standard deviation; otherwise, record-length distributions tend to be of the Poisson type. To make a choice, it is best to look at a histogram. When assuming normality, the distribution is determined by the value of the mean record length, *Rbar*, and the standard deviation, σ.

Example 6-3 Fixed space allocation per record.

In Ex. 4-8, case **1b**, we had fixed length records with the following parameters: $B = 1000, R = 180$, and $P = 4$, so that $n_{block} = 5$ and the excess per record was $G' = 1000 - 5 \times 180 - 6 \times 4 = 76 \approx R/2$.

Now we take variable length records that are still on the average 180 characters long, and have a small standard deviation σ of 20 characters. We allocate all the space in the block to the five records ($G = 0$), providing $(B-P)/n_{block} - P = 195$ characters of space for each record. The parameter t of Table 6-7 defines the space between mean and limit in terms of the σ: $t = (195 - 180)/20 = 0.75$ From Table 6-7 we find that 22.7% of the records will not fit into their allotted space. In Example 6-4 we try to do better by using buckets.

Sums of Normally Distributed Values Often, many variable-length records will be packed together in a bucket within a block, so that space not used by one record will be usable by other records. The relative variability of a sum is less than the variability of individual items, so that we can expect that the numbers of overflows will be reduced. For n_{block} records per bucket, the `mean` and `sd` for the sequence of records in a bucket are

$$mean_{block} = n_{block}\, mean_{record} \tag{6-4}$$

$$\sigma_{block} = \sqrt{n_{block}}\, \sigma_{record} \tag{6-5}$$

and, if the ratio of extra space is the same,

$$t_{block} = \sqrt{n_{block}}\, t_{record} \tag{6-6}$$

and the the overflow probability per bucket is less, as evaluated in Example 6-4.

Example 6-4 Allocation of variable-length records to a block.

If five records with the same behavior as those of Example 6-3 share a bucket, then

$$t_{block} = \sqrt{n_{block}}\, t_{record} = \sqrt{5} \times 0.75 = 1.68$$

From Table 6-7 we find that the overflow probability with bucketing is less than 6.7% (actually 4.7%) versus the 22.7% for single record slots, each with their maximum.

To demonstrate the space versus overflow trade-off more vividly we will use the same standard deviation $\sigma = 20$ used in Example 6-3, but applied to smaller records ($Rbar = 66$ vs. 180) so that the effect will be even more pronounced. Given is also $n = 10\,000$ and $B = 762$. Now, with the average $Rbar = 66$, an initial value of $n_{block} = \lfloor 762/66 \rfloor = 11$, and the number of unused characters per block $G' = 762 - 66 \cdot 11 = 36$. The total file will occupy $b = \lceil 10\,000/11 \rceil = 910$ blocks.

With $\sigma_{record} = 20$ the σ_{block} for the sequence of 11 records is $20\sqrt{11} = 66.33$ and the $mean_{block}$ is $11 \cdot 66 = 726$ characters. Table 6-7 indicates for t=36/66.33=0.54 a probability of overflow of about 29.5% per block. If we allocate and link an overflow block to each overflowing primary block $910 \cdot 0.295 = 269$ additional blocks will be required. The total used for this file is now 1179 blocks.

If, on the other hand, only $n_{block} = 10$ entries are placed in every block, then 1000 primary blocks are needed instead of 910, but 102 characters are available to cope with the variability of the length of the entries. This makes $t = 1.54$ and reduces the overflow to 6.2% or 62 blocks. This generates a net saving of $1179 - (1000 + 62) = 117$ blocks. There will also be a proportional savings in expected access time.

A further reduction to nine entries reduces the overflow probability to 0.61% but requires a total of 1118 blocks instead of 1062. The expected access time will still be slightly reduced.

Optimization of Allocation for Variable-Length Records It is sometimes wise to use a value of n_{block} which is less than the value computed based on the average, to reduce the probability of overflows further. In dense files overflows can be costly and using some additional space can provide a good trade-off. Example 6-4 shows this effect also, using small variable-length records.

6-1-5 The Poisson Distribution

Poisson distributions occur due to independent events. The shape of the Poisson distribution is fully described by the value of the mean of the observations, since in this distribution the mean and the variance are equal. The variance is the standard deviation squared, σ^2. To avoid introducing another symbol for the variance, we will use σ^2 or `sd**2` as appropriate. To compare an observed distribution with the curve describing a Poisson distribution, the corresponding values can again be computed. The program segment in Table 6-8 computes expected values for event counts from 0 to 20 following a Poisson distribution.

Table 6-8 Poisson distribution for 21 categories.

```
/* Poisson Distribution for 21 Categories  */              /* 6-8 */
       DECLARE expfr(0:20);
       expfr(0) = n*EXP(-mean);
       DO ov = 1 BY 1 TO 20;
          expfr(ov) = expfr(ov-1)*mean/ov;
       END;
```

A sample problem is tabulated in Example 6-5, which shows the observed distribution of insertions and the equivalent Poisson probability computed in Table 6-8 given above. When the two columns are compared, a Poisson distribution appears likely.

The events which lead to this distribution of Table 6-5 are such that a Poisson curve is indeed likely: The distribution is the sum of many events, each with a low probability, and the probability for events in high-numbered categories tends to zero. Table 6-9 modifies the program of Table 6-8 stopping the generation of categories when less than one entry is expected.

Table 6-9 Poisson distribution with grouping of low frequencies.

```
/* Poisson Distribution with Grouping of Low Frequencies      6-9 */
       DECLARE expfr(0:20);
       expfr(0) = n*EXP(-mean);
       left = n-expfr(0);     /* number of blocks for assignment */
       prod = expfr(0)*mean;
       DO ov = 1 BY 1 WHILE(prod<ov);    /* ok if next expfr > 1 */
          expfr(ov) = prod/ov;           /*     as computed now  */
          left = left-expfr(ov);
          prod = expfr(ov)*mean;
       END;
       last_category = ov;     /* used later in Table 6-10  */
       expfr(last_category) = left;
```

Testing for Goodness of Fit A more formal test to determine if a distribution fits the data can be made using the *chi-square* function. The *chi-square*, or χ^2, test compares categories of observations and their expectations, but each category should contain at least one expected sample. To avoid invalid categories, the frequencies of the Poisson distribution that are less than 1 can be grouped with the last good category as shown in the program of Table 6-9 and Example 6-6.

Example 6-5 Poisson distribution of insertions for an indexed-sequential file.

There were 550 insertions into a dense indexed-sequential file of 400 blocks. These insertions caused $0, 1, 2, \ldots$ overflow records to be written for each of the blocks. The mean number of insertions per block is `mean` $= 550/400 = 1.38$. The expected frequency is computed for `n = 400`.

No. of overflows	Observations		Expectations
	Frequency	Records	Frequency
`ov`	`fr(ov)`	`ov·fr(ov)`	`expfr(ov)`
0	101		101.4
1	138	138	139.1
2	98	196	95.6
3	45	135	43.8
4	11	44	15.1
5	5	25	4.1
6	2	12	0.9
7	0	0	0.2
...	0	0	...
	400	550	400

The observed values, listed in Example 6-5 are best inspected again visually. A better presentation for that purpose would be a histogram `fx` as can be generated by the program segment in Table 6-5. The cells of low expected value are combined into a single, namely the last category, by accumulating them in a program loop as shown in Table 6-10.

Then the chi-square values are computed, as shown in Table 6-11, to see how well the actual observations fit the assumed Poisson distribution. The final results are seen together in Example 6-6.

Table 6-10 Combining tail values of an observed frequency histogram.

```
/* Combine tail values of observed frequency histogram */ /* 6-10 */
      DO i = last_category+1 TO number_of_categories;
         fx(last_category) = fx(last_category) + fx(i);
      END;
```

Table 6-11 Computation of Chi-square value for goodness of fit test.

```
/* Computation of Chi-Square Value for Goodness of Fit */ /* 6-11 */
       chisquare = 0;
       DO ov = 0 TO last_category;
          dif = fx(ov)- expfr(ov);
          chisqterm = dif**2 / expfr(ov);
          chisquare = chisquare + chisqterm;
          PUT DATA( ov, fx(ov), expfr(ov), dif, chisqterm);
       END;
       PUT DATA( SUM(fx), SUM(expfr), chisquare);
```

Values obtained for χ^2 can be compared with standard values, which are based on the assumption that the difference of distributions was caused by random events. These standard values for χ^2 can be computed as needed using approximations of binomial distributions or can be found in statistical tables and graphs.

Figure 6-6 presents the standard χ^2 distribution in graphical form. In order to use the χ^2 distribution, the number of degrees of freedom *df* has to be known. When we distribute our samples over a specific number of categories c the value of *df* will be equal to $c - 1$.

Example 6-6 Testing a Poisson distribution.

ov	fx(ov)	expfr	dif	chisqterm
0	101	101.1	0.1	0.001
1	138	139.1	1.1	0.009
2	98	95.6	2.4	0.060
3	45	43.8	1.2	0.033
4	11	15.1	4.1	1.113
last	7	5.3	1.7	0.545
$n =$	400	400		$\chi^2 = 1.761$

Evaluation:

The value for χ^2 is 1.761 at a $df = 5$ for the indexed-sequential file observations shown in Example 6-5. The value for this comparison falls within the area of Example 6-6, which is appropriate for most cases which match an expected distribution. The point is off-center, close to the "good" side, so that it seems likely that the file updates are not quite random, but somewhat uniform. Perhaps many of the insertions are due to some regular customer activity.

A very high value of χ^2 makes it unlikely that the frequencies are related; a very low value could cause suspicion that the data is arranged to show a beautiful fit. The chi-square test is useful when distributions are being compared. Other tests, such as the *t-test* and *F-test*, can be used to compare means and standard deviations obtained from samples with their expected values, if the distribution is known or assumed to be known.

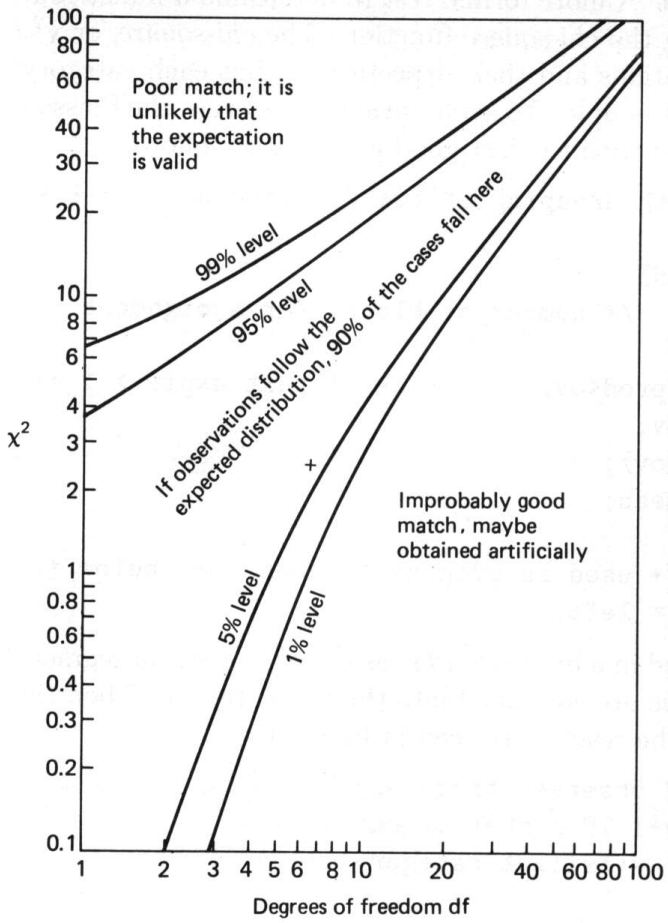


Figure 6-6 Standard chi-square distribution.

Collision Probability To derive Eq. 3-72 in Chap. 3-5-1.4, use was made of a relationship which gave the probability of occurrence of collisions in direct access slots given n entries into m slots. This result will now be derived using the tools developed.

Each record has a probability of $1/m$ of being assigned to any one slot. The probability for one slot to receive q records is given by the binomial distribution

$$P_b(q) = \frac{n!}{q!(n-q)!}\left(\frac{1}{m}\right)^q\left(1-\frac{1}{m}\right)^{n-q} \qquad 6\text{-}7$$

For the case that $n \gg 1, m \gg 1, q \ll n$, we can approximate this expression by the Poisson distribution with the mean density n/m:

$$P_b(q) = e^{-(n/m)}\frac{(n/m)^q}{q!} \qquad 6\text{-}8$$

$P_b(0)$ is the probability of the slot remaining empty, and $P_b(n)$ is the (very small) probability of this slot receiving all n records (see again Fig. 3-19). If q records

appear in one slot, causing $q-1$ overflows, then the number of accesses to fetch all these records, which form one sequential chain, will be

$$0+1+2+\ldots+q-1 = \tfrac{1}{2}q(q-1) \qquad 6\text{-}9$$

Taking the sum of all the $P_b(j), j = 1 \ldots n$, multiplied by their cost in terms of disk accesses, gives for any one slot the expected access load AL

$$AL_{slot} = \sum_{j=0}^{n} \frac{j(j-1)}{2} P_b(j) \qquad 6\text{-}10$$

The successive terms of the Poisson distribution are related to each other by the iteration

$$P(j) = P(j-1)\frac{mean}{j} \qquad 6\text{-}11$$

The mean is here n/m, so that using the Poisson distribution for P_b

$$AL_{slot} = \frac{1}{2}\frac{n}{m}\sum_{j=0}^{n}(j-1)\,P(j-1) \qquad 6\text{-}12$$

Since at $j=0:\quad P(j-1)=0 \qquad$ and at $j=1:\quad j-1=0$

$$\sum_{j=0}^{n}(j-1)P(j-1) = \sum_{j=2}^{n}(j-1)P(j-1) = \sum_{k=1}^{n} k\,P_b(k) \qquad 6\text{-}13$$

where $k = j-1$. The expected record load RL per slot is the mean, which in turn is equal to the mean of the cumulative frequency distribution, specifically the sum of the products of the number of records per slot i and the probability of the occurrence of the event $P(i)$

$$RL_{slot} = \frac{n}{m} = \sum_{i=1}^{m} i\,P(i) \qquad 6\text{-}14$$

and when $m \approx n$ the series terms are equal, so that the expected increase of accesses for overflows due to collisions per record p is

$$p = \frac{AL}{RL} = \frac{1}{2}\frac{n}{m} \qquad 6\text{-}15$$

If multiple records ($Bfr = B/R$) can be kept in a *bucket*, the probability of another record entering a slot can be predicted using $P_b(q)$ as derived above. The number of buckets will be m/Bfr and the number of collisions per bucket will be higher. There will be no overflows, however, until Bfr records have entered the bucket, so that the cost of accessing overflows AL becomes less. For this case Johnson[61] derives similarly,

$$p = \frac{1}{2}\frac{n}{m/Bfr}\sum_{k=Bfr-1}^{n}(k-Bfr)(k-Bfr-1)\,P(k) \qquad 6\text{-}16$$

Of course, the processor's computing effort, cix, to search within one bucket increases now. An evaluation of this effect in indexed files is given in Chap. 4-2-4.

Collision Expectations for Direct Files The results obtained above can be further developed to give the expected overflow space requirement or the average overflow chain length, Lc, due to collisions. These can then be compared with observations from the operational file.

The probability of a slot being empty was approximated by the zero term of the Poisson distribution so that

$$P_b(0) = e^{-n/m} \tag{6-17}$$

The probability of a slot of the primary file being used is then

$$P1u = 1 - e^{-n/m} \tag{6-18}$$

and the number of records in the primary file is then $m\,P1u$, and in the overflow file is

$$o' = n - m\,P1u \tag{6-19}$$

The overflow area contains the tails of all the Poisson-distributed record sequences. Using Eq. 6-5 for the σ of a sequence and the fact that σ^2 of a Poisson distribution is equal to the mean n/m, we expect a standard deviation of o' to be

$$\sigma = \frac{n}{m}\sqrt{m} \tag{6-20}$$

and this σ applies of course to the entire file. If an observed standard deviation is less, we can deduce that the keys are more uniform than a Poisson distribution-based model expects and that a smaller prime or overflow area may suffice. If the observations are worse, the key-to-address transformation is not doing its job.

Example 6-7 Estimating a collision probability.

For a file as used in Example 3-10 (Chap. 3-5-3) $n = 1000, m = 1500$, we find with Eq. 6-18 $P1u = 0.49$, and expect $o' = 265$ with a σ of 25.8. This value can be compared with observed values from operational measurements.

When the buckets contain more than one record ($Bfr > 1$), the above computation can be carried out using the cumulative Poisson probability for the records 1 to Bfr being in the buckets and the records from $Bfr + 1$ to n overflowing. Estimates of the mean total accesses $1 + Lc_I$ are used in Knuth[73S] (page 535) to compute the length for an unsuccessful fetch leading to an insertion to complement Exanple 6-5. The value of

$$P1u = n/m - Lc_I \tag{6-21}$$

so that the case of Example 6-7 corresponds to $Lc_I = 1000/1500 - 0.49 = 0.18$.

6-1-6 Other Distributions

When a distribution does not seem uniform, normal, exponential, or Poisson-like to an acceptable extent, the behavior may follow one of many other functions which are available to describe distributions. An important reciprocal distribution, similar in shape to a negative exponential, is presented by *Zipf's law* (Eq. 13-6). This distribution is used to describe activity ratios of records and type/token ratios of words in text.

Zipf's law can be viewed as a generalization of *Heising's rule*:

> 80 percent of accesses to a file address 20 percent of its records.

An example of its use is given in Chap. 14-3-2.

When requests arrive from users at terminals, queues can form due to conflicting requests of independent users. The requests from each single user are mutually dependent, so that the total set of requests is more complex than a Poisson distribution. This behavior is modeled well by Erlang distributions; they are presented in Sec. 6-4-1.

Some cases where the distributions are not simple can be solved by algebraic transformations of the observations until some fit to a known distribution is obtained. Reciprocal, logarithmic, and hyperbolic functions are used frequently. Observations may also be broken down by differencing or decomposed into sums or products of functions. Differentiation or taking successive differences helps to remove the effect of aggregation if the observations are some form of c.d.f.. A visual display of the transformed data is helpful to rapidly establish the benefit of a transformation.

Any transformations are best based on an understanding of the underlying phenomena, so that we have some model of what is happening. In the examples shown earlier we tried to find a causal mechanism; for instance, utility customers pay in regular cycles and the hospital patients in Fig. 6-3 pay according to individual pressures. If such an empirical model fits the observations well, it can be used with some confidence for extrapolations and the design of databases that work in general and over a long term. The algebraic transformations of the observations based on the simple patient-payment model were adequate to exploit the distribution for effective direct access.

If transformations make observed data tractable, but the underlying model which generates the observations is still not understood, then much care has to be taken in applying the results. The use of many variables or of complex transformations to describe observations is apt to generate very misleading results when the values obtained after the fit for these variables are used to *extrapolate* the behavior of new systems. Empirical functions based on a few reasonable variables can be used for *interpolation*, since this procedure is less risky. Since there is a degree of arbitrariness in assigning a transformation function, subsequent testing for validity should be done with fewer degrees of freedom df.

Skewed Distribution If a distribution appears lopsided or skewed, it is wise to analyze the relationships being considered for nonlinearities. It may be that the parameter being measured is some function which distorts the behavior of parameters of the model at a more basic level.

The skewed distribution in Fig. 6-1 occurred because the blocking factor, Bfr, was described for a fixed number of variable-length records per block. The underlying measure, the distribution of record lengths or the inverse, $1/Bfr$, had a normal distribution.

Bimodal Distribution A bimodal distribution as shown in Fig. 6-1 can best be decomposed by fitting two normal distributions to the data. Four parameters have to be determined: $mean_1$, σ_1, $mean_2$, and σ_2. If the modes are well separated, the means can be estimated at each node and fitting of two normal curves will be simple. It helps if the σ's are equal or have a predetermined relationship. Sometimes the position of the means can be determined by alternative reasoning.

Unrecognizable Distributions Even when the shape of the distribution does not match any known distribution, there is still a conservative statistical tool available, *Chebyshev's inequality theorem*, to describe the behavior of the variables.

Given the mean and the standard deviation, which are always computable for a set of observations, Chebyshev's inequality provides directly the probability of values from the set of observations $x_i, i = 1, \ldots, n$ falling outside a range. The range, rg, is defined in terms of a number, c, of standard deviations, σ, to either side of the mean of x. The mean and standard deviation are computed from the observations without any assumptions regarding the behavior of the distribution. If the sample observations are not biased, then for some k

$$p(x_k \text{ not in } rg) \leq \frac{1}{c^2} \qquad \text{where} \qquad mean - c\sigma < rg < mean + c\sigma \tag{6-22}$$

This relationship can, for instance, be used to state that given any observed distribution, 90% of the values will be within $\sqrt{10}\,\sigma = 3.162\sigma$ to either side of its mean. If the distribution is approximately symmetrical, we can assume that only 5% of the values exceed the $mean + 3.162\sigma$. The worst case is that all 10% of the values exceed the $mean + 3.162\sigma$.

If the distribution has a single mode and is symmetric, then

$$p(x \text{ not in } rg) \leq \frac{2}{3c^2} \tag{6-23}$$

and, correspondingly, the probability that a value, x_k, of the observations exceeds the $mean$ by $c\,\sigma$ is

$$p(x_k > mean + c\sigma) \leq \frac{1}{3c^2} \tag{6-24}$$

Example 6-8 Estmating file overflow with the Chebyshef inequality.

For the direct file used to illustrate Eq. 6-20, the Chebyshev inequality can be used to evaluate the chance of exceeding a primary overflow area of a given size. The distribution of overflows is not symmetric, so that the worst case approximation should be used. Given was $o' = 265$, $\sigma = 25.8$, and we wish to guarantee that in all but 1% of the cases the records fit into the primary overflow area. Here

$$c = \sqrt{100} = 10 \qquad \text{and the required space} \qquad o = o' + 10\sigma = 523$$

This value is based on the sample observations. Multiple samples should be used to verify stability of the result.

6-1-7 Other Statistics

At times, an observed response is a function of multiple variables. With assumption of independence among these variables we can separate the effects given a sufficiently large number of observations. Two types of analysis methods are in use for this work: *multiple regression* and *analysis of variance*. Regression analysis determines a line describing a linear relationship between the individual variables and the resulting dependent variable. Many relationships in computer systems are not linear. Analysis of variance relates causes and effects by discrete groupings and has been used to analyze the performance of paging operating systems based on the variables: memory size, program size, loading sequence, and paging algorithms. Both regression and analysis of variance assume linear relationships of the combination of independent variables on the dependent variable.

6-1-8 Help with Statistics

It is not necessary to be a statistical expert to use the methods and results that statistical research and development have provided. Many statistical packages exist to help perform these calculations. Graphical output can make the models being used accessible to people who are not comfortable with the numerical aspects of statistics.

You cannot expect that statistical experts will solve systems problems. The understanding and modeling of the system being evaluated remains as important as the application of analytical techniques. This also means that expertise about the computer system being evaluated and its applications must be available as well when an analysis is being made. Especially, the actual design process cannot be delegated to specialists in a single narrow area, and neither programmers nor statisticians can be expected to provide acceptable system evaluations by themselves.

6-2 SIMULATION

When no source of data is available which can be used to predict the behavior of a new system, it may be necessary to build a scale model to generate predictive data. Such a simulation model will be based on the process and hardware to be used in reality and will be fed with a sequence of descriptions of the desired computations. The output of a simulation will not be the results of the computation, but rather measurements of parameters collected in the simulation process.

The principal components of a discrete, event-driven simulation are shown in Fig. 6-7. The input to the simulation itself is an event queue, which contains entries of events that are created external to the system being simulated, for example, a request to retrieve a record; and internal events, for example, a completion of a seek. The output is a log of events with their times, which can be further analyzed.

External event entries may be generated by a program. It will use the expected distribution of external requests to make up request types and their times.

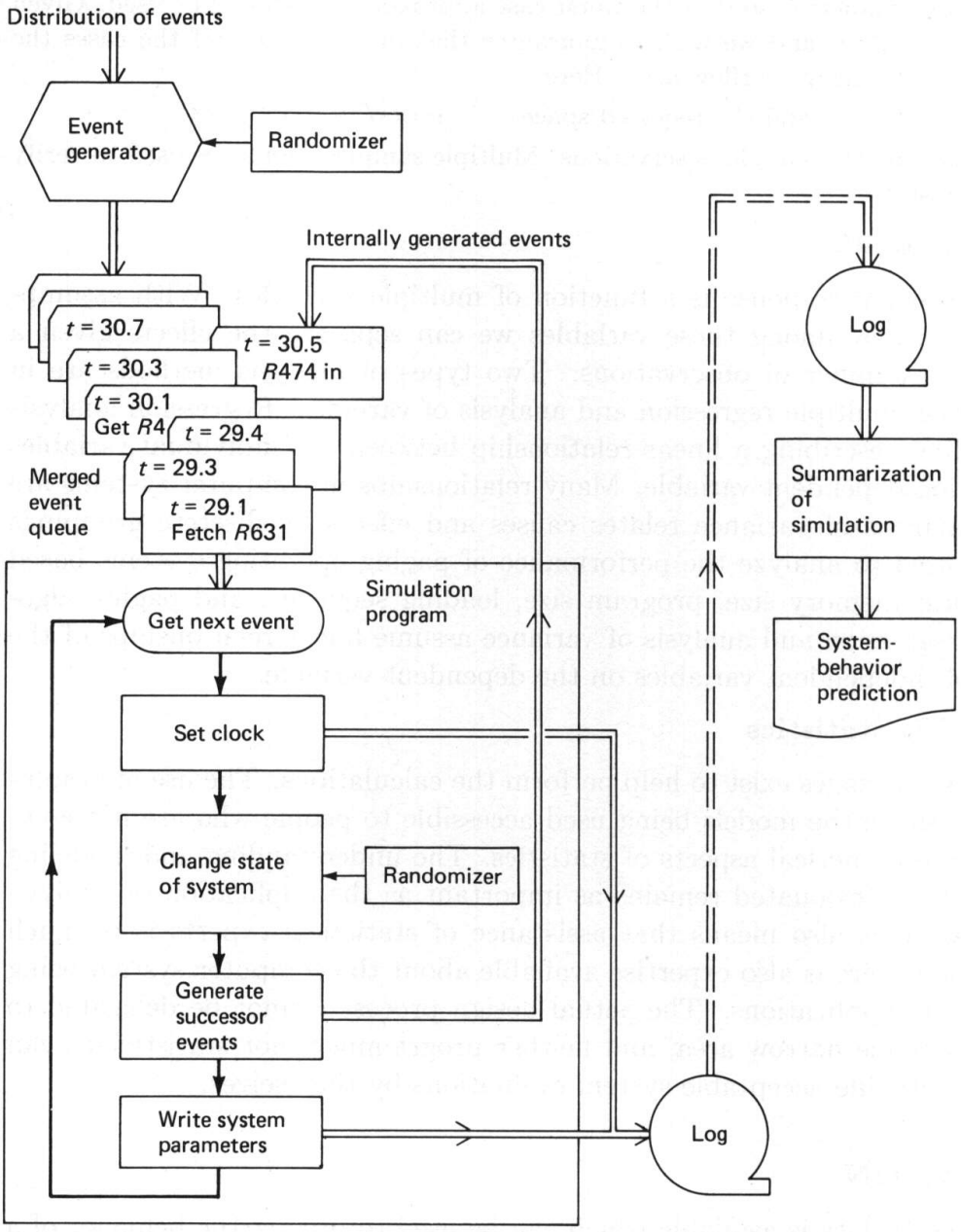
Another source for external events may be a log file from an existing system or a synthetic program which generates event requests during its execution. Logs or programs have the advantage that they provide a standard for system testing.

The Process of Simulation A simulation has its own timing mechanism. To determine how long the simulated computation will take in real life, the simulation uses a variable `clock` to keep the simulated time. The `clock` is incremented to the next event to be simulated. An external event may specify:

"At time = 29.1: `Fetch Record for employee 631 (' Jones' )`"

After setting the clock to 29.1 and logging the receipt of this event, the steps of the process are simulated to the extent required.

If a hashed access to a disk or similar unit is being simulated, the steps shown in Example 6-10 are appropriate. We see that, because of the existence of queues, requests may not be processed when they arrive. Queuing is the subject of Sec. 6-3.



A problem with simulations driven by lists of external events is that there is no feedback from the system to the generator. This is fine if the process requesting services is indeed independent of the system.

Example 6-9 Independent and non-independent request sources.

1 An independent request function would be the distribution of flight inquiries by phone from potential passengers.
2 If the requests are issued by airline reservation agents, then a poor response rate by the system response will reduce their actual rate of entering inquiries.

Dependent requests are best modeled internally within the simulation. A response function based on data as shown in Fig. 5-3 can be used to describe the effects of the user-machine interaction. When randomly generated events are used, multiple tests or partitioned long tests are needed to verify the stability of the simulation results.

The parameters of interest are either processed incrementally or written on a log file for later summarization. Typical parameters of interest are the total time to process all the events, the time required to process individual events, and the activity ratios of the devices used. The data obtained is again conveniently presented as histograms or described as distributions. Then predictions can be made, such as, "The response time to an inquiry is less than 10 s 90% of the time."

Example 6-10 Simulation steps.

1 A status table for the simulated devices, as shown in Fig. 6-8, is checked. If the device being requested is busy skip to step 13.
2 A block address is computed for record with key 631 using the key-to-address transform being considered. A constant value, c, for the KAT is obtained separately.
3 The desired cylinder position is determined.
4 The amount of seek, cyl, from the current position is calculated.
5 The required seek time, s_{cyl}, is found.
6 A random latency, r, is generated.
7 The block transfer time, btt, is calculated from the transfer rate, r, and the block size, B. It will be constant for a given system.
8 A random function based on n/m is used to determine if there is a collision,
if yes, go to step 3 to compute the incremental times, else
9 The completion time is computed as $29.1 + c + s + r + btt$, say, 29.4.
10 The status of the disk unit used is set to "busy".
11 Now an internal event is generated which states that

> "At time 29.4: `Record 631 ('Jones') is in core, and disk D is free`"

12 The internal events are merged with the external events, and the simulation continues until the event list is exhausted.
13 When a device is found busy, the requests are put into a device queue.
14 Eventually A "`device is free`" signal from the internal queue resets the status, and-directs the simulation to look at the device queue, select a request to be processed, and simulate it, continuing with step 1.

Writing a Simulation The use of simulation techniques to investigate file system behavior is simplified by the large gap between processor and file speeds. Adequate results can often be obtained from a simulation which concentrates on file delays and models the computation only in a gross fashion.

In one area, computational delays, other than those affecting bulk transfer, have to be considered. Much of the computational overhead in file operations is due to complexity in operating systems. Frequently, several milliseconds can elapse between the execution of a `CALL` for service and the emitting of a command to actually move the disk arm. Mean overhead times for these operations can sometimes be obtained from manuals, or are otherwise obtained by executing a large number (n) the disk operation in question. The known disk access times can be subtracted from the elapsed time, or the CPU time measurement may be available from the system. Dividing by n gives the mean CPU time for servicing the operation. If no paging of code occurs, this value should have a very low standard deviation.

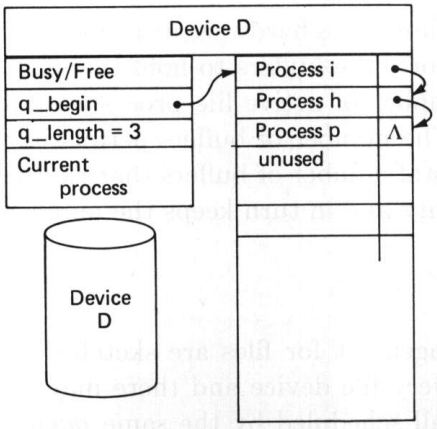
A number of simulation languages are available and have been used extensively for system simulation. The author has found that simple simulations can be written rapidly on an interactive system in any higher-level language. The simulation languages now commonly available use batch processing techniques. When a simulation becomes large, the computing time required can become a significant item in the design budget, and the simulations have to be planned carefully so that they will return an adequate benefit for the expended effort.

A simulation model which includes many variables will first be tested by changing the variables one by one to select those which have a major effect on the outcomes. Then these important variables can be evaluated by having the randomizer generate test values throughout their range, while keeping the variables which had less importance constant. When a desirable model has been found, the minor variables will be tested again to verify that the perceived lack of effect is also true at the final setting of the model variables.

Another approach to handle simulations of large systems is to attack the simulation hierarchically. First, the cost of primitive operations is simulated, and the results are fitted to a simple mathematical formulation. These formulations are then used by simulations of more complex tasks. Several simulation levels may be used until the needed results are obtained. At each level a verification is done so that the parameters used do not exceed the range simulated at the lower level, since extrapolations based on fitted functions are risky. This scheme requires some sophistication so that the mathematical abstractions from each level are viable.

6-3 QUEUES AND SCHEDULING TECHNIQUES

When a computation requests service from a device which is busy, the computation will be delayed. Since the computation should not be abandoned, the request will be noted as an entry in a *queue* for the device. A single process of a transaction is delayed when it requests a file operation until it gets permission to use the file. The user's file requests may be issued directly by the user's programs, or indirectly by systems programs. The elements that make up a queue for a device are shown in Fig. 6-8.



Summary parameters collected during system operation:

Number of requests for the device

Number of requests which found the queue empty

Maximum and average queue size when a request was made

Fraction of time that the device was busy

Figure 6-8 Elements of a queue.

The parameters listed are vital when systems are monitored for planned augmentation and improvement. If the queue is being simulated, then not only these variables will be collected, so that the entire usage distribution can be reconstructed.

Queues and Processes One computation may initiate multiple processes which will independently go to the file system to request services. The entire computation cannot proceed beyond certain joining (P) points until outstanding processes have been completed. Figure 6-9 shows the progress of a small computation, which spawns processes for each file operation. Processes which have to read data frequently require delay of computations. Processes which write data may only prevent the final termination of a computation.

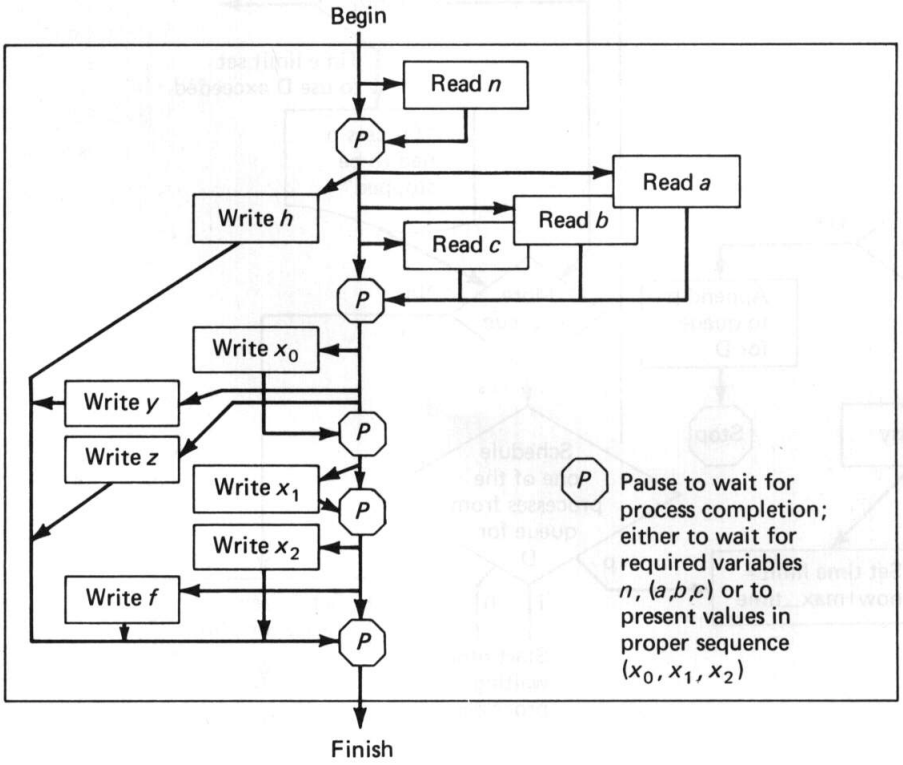


Figure 6-9 A Computation with file processes.

Each service request in a queue is due to a request by one individual process. The sum of all the queue lengths in the system is limited by the number of possible processes in the system. This number may be quite large.

Processes, unless specifically constrained, operate asynchronously. This means it is not wise to generate a bunch of individual processes to write a sequence of blocks into a file: The blocks may not be written in the desired order. If, however, the blocks have individual addresses, a smart queue-management system will be able to write the blocks onto the file in an order which minimizes hardware delays. Every write or read process will be associated with one or more buffers to hold the data. The number of buffer lists possible limits the number of active file processes and hence the degree of parallel operation possible. The number of buffers permitted per file limits the queue lengths. The total number of buffers is limited by memory. This limit in turn keeps the number and sizes of queues manageable.

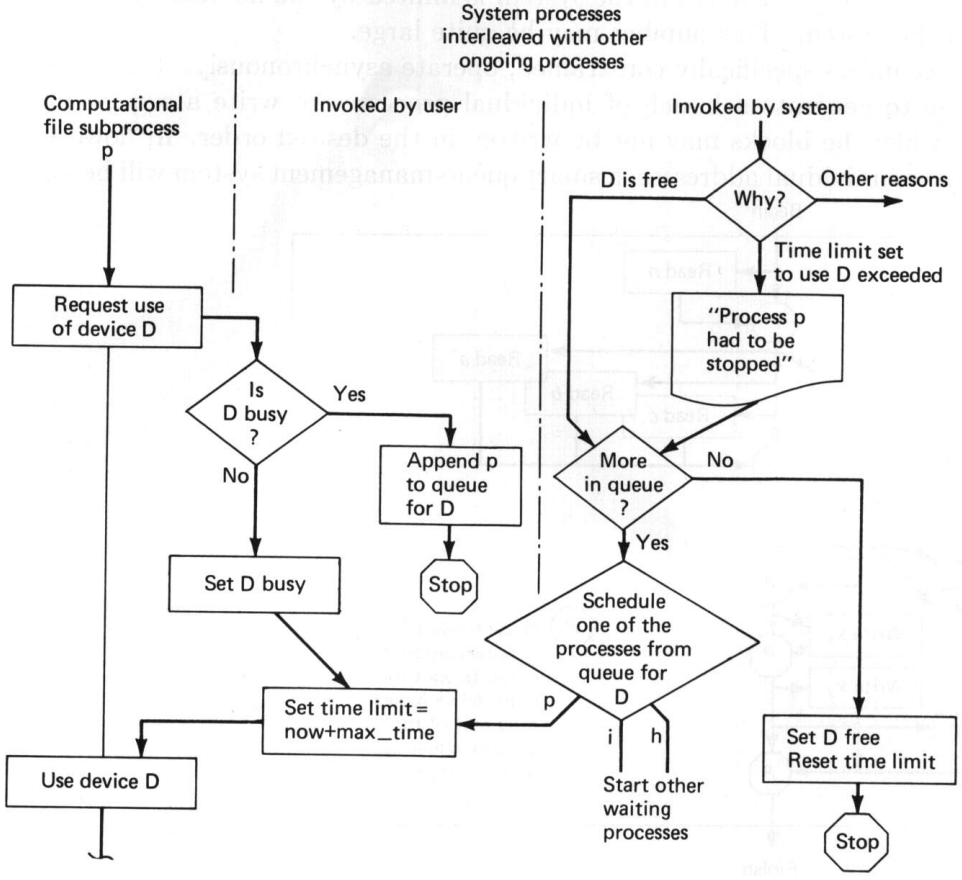


Figure 6-10 Queue management.

6-3-1 Queue Management

The activities which are part of the queue management for files are shown in Fig. 6-10. There will be at least one queue for every file device and there may be also a number of queues for other device types, all scheduled by the same *queue-management program* (QMP). The QMP is invoked either by a user, who wishes access to a device being managed by the QMP, or by the operating system, when it has analyzed that a signal for service involves the QMP.

The typical reason for the operating system to invoke the QMP is that a device has been released by the process that was holding it. Such a release is often initiated by a hardware signal which indicates the completion of the actual data transfer for device D. This will be followed by a verification procedure, after which device D is no longer needed by the process. It can now be assigned to another process.

Occasionally, because of a failure of the hardware or of a program, a process does not release, or maybe does not even begin to use device D. If a time limit or *time-out* is set, the faulty process can be canceled when the allowed time is exceeded, and the device given to another process. In systems where no time limits are set, all computations requesting D will come to a halt, and eventually all activity of a system will cease.

Some simple transaction systems attempt to prevent time-out errors by having all file access processes use code sections which are provided as part of the transaction operating system. This, of course, increases the complexity of the operating system itself. Excessive restrictions imposed by a system on the user for the sake of protection indicate an excessively weak operating system.

In order to provide space for infrequent large queues, a QMP may share the space for all queues active in the system, using linked-list structures. In interacting systems the queues for a few devices are often long, while other queues are empty. The monitoring of queues is important when systems are to be tuned for best performance. Queues may be created dynamically when a new resource is added to the system.

When more than one process is waiting in the queue for the device, one of these processes has to be selected. The selection or *scheduling* of transaction service requests from a queue is the topic of the remainder of this section.

6-3-2 Scheduling from a Queue

The scheduler selects one of multiple candidate processes, which are waiting for service, for execution. The scheduler is the big diamond-shaped box in Fig. 6-10. It is invoked by the queue-management programs but should be a distinct module. Queue management concerns itself with the correct operation of the devices. The scheduler implements management policy, and is frequently subject to change. Many operating systems are built so that correct operation may be compromised when a scheduling policy is changed.

The choice of policy to select request elements from a queue is wide. The intuitively fairest policy is to select the process which has been waiting the longest, that is, use *first in first out* (FIFO) as the scheduling policy. Table 6-12 lists scheduling policies that have been employed, using the abbreviation which is most common in the queuing literature. Most of these policies have also been given computer-oriented names; several of these abbreviations are provided in Appendix A. The reason for alternate scheduling policies is the diversity of objectives to be achieved. Objectives in file and database scheduling are: minimal user delay, minimal variability of user delay, maximum utilization of the processor, maximum utilization of the transfer capability from the device, or maximization of locality for a currently active process so that a single process at a time is flushed through the

system as fast as possible. Objectives may also be combined; a typical combination is the maximal utilization of the device, constrained by a requirement to provide an adequate response time for any user.

The scheduling choice is important to system performance only when the queues are filled; when there is no, or only one, process to be served, all algorithms behave equally well.

Scheduling Policies for a Cylinder We first consider only one cylinder of a disk. When using a FIFO policy, an average of one rotation will be lost between successive block accesses to a cylinder. Since other blocks may pass under the reading heads during this time, it is desirable to arrange that any requested blocks are read when they appear, rather than read strictly in the order in which they were requested. Such an approach is called *shortest service time first*, or SSTF. Control units for disks are available which allow the maintenance of multiple requests, one per sector, and these requests will be processed in sector sequence. Examples are the UNISYS B6375 and the IBM 2835.

When block lengths are fixed and track-switching time is predictable, an SSTF schedule can be implemented through software arrangement of requests. Sector-specific queues are shown in Fig. 6-11, with an example of the selection path for the next sector of a fixed head disk. The gap between sectors provides the time needed for switching tracks. If all sector queues Q1, . . . ,Qb_{track} are filled, then b_{track} blocks can be accessed during one track revolution, giving the maximal *utilization factor* *uf*, defined as #(user blocks accessed)/#(disk blocks passed), of one.

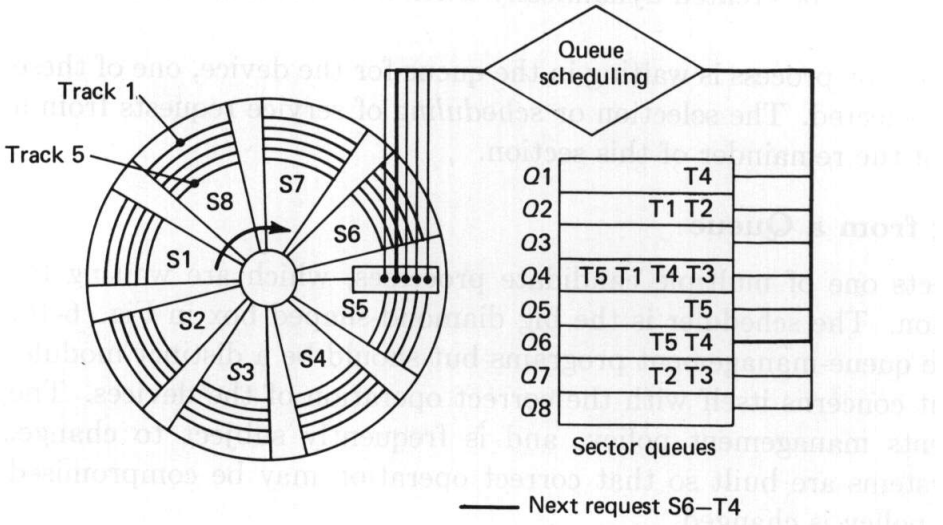


Figure 6-11 Queue optimization for a drumlike device.

Even if b_{track} or more blocks are waiting to be processed, there will usually be empty queues. In Fig. 6-11 there are 12 requests for the five tracks in the 8 sector queues, but Q8 for sector S8 is still empty, and unless a request arrives for this queue during the next two sector times S6,7 the time for passing sector S8 will be unutilized. The probability of having an empty queue will be less for the sectors about to be accessed (S6, 7, . . .) than for the sectors which have just been processed

$(S5, 4, \ldots)$. If the arrival of requests to the cylinder has a certain distribution, the arrival of requests to a sector queue (if they are independent of each other) will have the same distribution.

Long queues increase productivity, so that SSTF scheduling promotes a stable equilibrium as long as

$$uf = \frac{\lambda\, 2r}{b_{track}} < 1 \qquad 6\text{-}25$$

where $\lambda\, 2r$ is the number of arrivals during one revolution.

Example 6-11 Computing the arrival rate.

A $uf = 0.75$ corresponds, for the cylinder of Fig. 6-11 with $b_{track} = 8$ sectors and a rotation time of $2r = 50$ms, to an arrival rate $\lambda = 0.120$ requests per ms, or 120 requests/s.

Unfortunately, a high utilization is associated with long queues, and hence long delays to processing requests. The mean delay for waiting in a sector queue w, to be added to $r + btt$, has been derived for a Poisson-type distribution of arrival times by Coffman[69]. For a moderately large number of sectors ($b_{track} > 3$) the effects of the discrete sector boundaries can be disregarded; then

$$w = \frac{uf}{1 - uf}\, r \qquad \langle \text{Poisson arrivals} \rangle \; 6\text{-}26$$

giving a total average processing time for a request of

$$T_F = w + r + btt \qquad \langle \text{cylinder queue} \rangle \; 6\text{-}27$$

While a high utilization rate does increase the productivity of a cylinder, the waiting times, and hence the queue lengths, increase rapidly with heavy utilization.

Example 6-12 Queue delay due to high utilization

At a utilization $uf = 0.75$ the waiting time, w, becomes $3r$, or the equivalent of one and a half revolutions. Given $r = 25$ms, $w = 75$ms.

At $uf = 0.9$ $w = 9r$, and as $uf \to 1$ the queues get longer and longer, and will never be worked off if the arrivals don't cease.

A result of queuing theory, *Little's formula* (Jewell[67]), states that at equilibrium the total number of requests queued, Rq, depends on their delays, w'

$$Rq = \lambda w' \qquad 6\text{-}28$$

Example 6-13 Computation of the queue length.

For the case of Exanple 6-12 we include in Rq the requests in the queue due w and r. Let $w' = w + r = 100$ms, then $Rq = (0.120 \text{ requests/ms})100 = 12$ requests.

Denning[72] has analyzed the case of uniform distribution of arrivals in the sector queue, taking into account the changing density before and after access. Uniform arrivals are a reasonable assumption if cylinder requests are due to sequential processing or due to direct-access algorithms. Here, the utilization

$$uf = \frac{1}{\frac{1}{2}b_{track}/Rq \ + 1} \qquad \langle \text{uniform arrivals} \rangle \ 6\text{-}29$$

Under extreme conditions ($Rq > 10, \lambda 2r > 10$ or $Rq < 1, \lambda 2r < 1$) the models behave similarly under either arrival distribution.

For cylinders the other scheduling algorithms listed in Table 6-12 are of interest only when variable-length blocks are collected. Since such usage is uncommon, we will proceed to consider disk units.

Scheduling Policies for Disks For disks with many tracks the intensity of use per sector will tend to be small, so that the expected length of the sector queue often becomes less than one and the cost of scheduling by sector does not provide commensurate benefits.

However, an optimization of seek times can improve performance significantly for disklike devices. This requires again the use of scheduling policies other than common queuing (first-in first-out or FIFO). The minimization of seek time is completely independent of the minimization of rotational latency within a cylinder so that two different schedulers may be operative within one file system at the same time. Each of the policies listed in Table 6-12 will now be considered.

LIFO Transaction systems can use LIFO to optimize the use of the disk by minimizing disk distances. The idea here is that, by giving the device to the most recent user, no or little arm movement will be required for sequential reading of the file. LIFO can also minimize congestion and queue lengths, since jobs, as long as they can actively use the file system, are processed as fast as possible.

Table 6-12 Scheduling algorithms.

Name	Description	Remarks
Selection according to requestor:		
FIFO	First in first out	Fairest of them all
PRI	Priority by process	Control outside of QMP
LIFO	Last in first out	Maximize locality and resource utilization
RSS	Random scheduling	For analysis and simulation
Selection according to requested item:		
SSTF	Shortest service time first	High utilization, small queues
SCAN	Back and forth over disk	Better service distribution
CSCAN	One-way with fast return	Lower service variability
N-step-SCAN	SCAN of N records at a time	Service guarantee
FSCAN	N-step SCAN with N = queue size at begin of SCAN cycle	Load-sensitive

Priority The use of *PRIority*, frequently given to small computations, also has the effect of flushing selected computations through the system. If the number of computations having equally high priority becomes large, the disk usage efficiencies diminish. Users with large computations and low priorities often wait exceedingly long times. Priority-driven systems will have a high rate of completed computations and small queues. Statistics from these systems are used by computer center directors to demonstrate the effectiveness of their operations, while the users split their computations into small pieces to share in the benefits. This type of operation tends to be poor for databases.

If the current cylinder position of the disk-access mechanism is known, scheduling can be based on the requested item rather than based on attributes of the queue or the requestor.

Shortest Service Time First The SSTF algorithm for disks is equivalent to highest locality first. All requests on the current cylinder will be processed, and then requests on the nearest cylinder. A random tie-breaking algorithm may be used to resolve the case of equal distances. Since the center tracks are apt to be favored, an alternate tie-breaking rule is to choose the direction of the nearest extremity. This means that if the current position is 115 out of $j = 200$ tracks and requests are waiting at 110 and 120, then 120 will be chosen.

Guaranteeing Service Any of the methods shown above, except FIFO, can leave some request unfulfilled until the entire queue is emptied. It is hence necessary to modify these techniques to reintroduce some minimum service level into the system. One augmentation is the addition of a dynamic priority structure which is based on the length of time that a process has been delayed. The rate of priority increase has to be chosen to satisfy response time constraints while not losing advantage of improved locality.

The SCAN algorithm adds regularity of control to a SSTF method. When all requests on a cylinder have been served, it proceeds to one specific side, and this direction is maintained until the extreme cylinder is reached. The service direction changes whenever the inner or outer cylinder is reached. A modification of SCAN (LOOK) reverses direction as soon as there are no more requests in that direction. A SCAN policy will operate similarly to SSTF unless the request pattern is particularly poor, since the probability of requests in SSTF is biased against the area most recently processed.

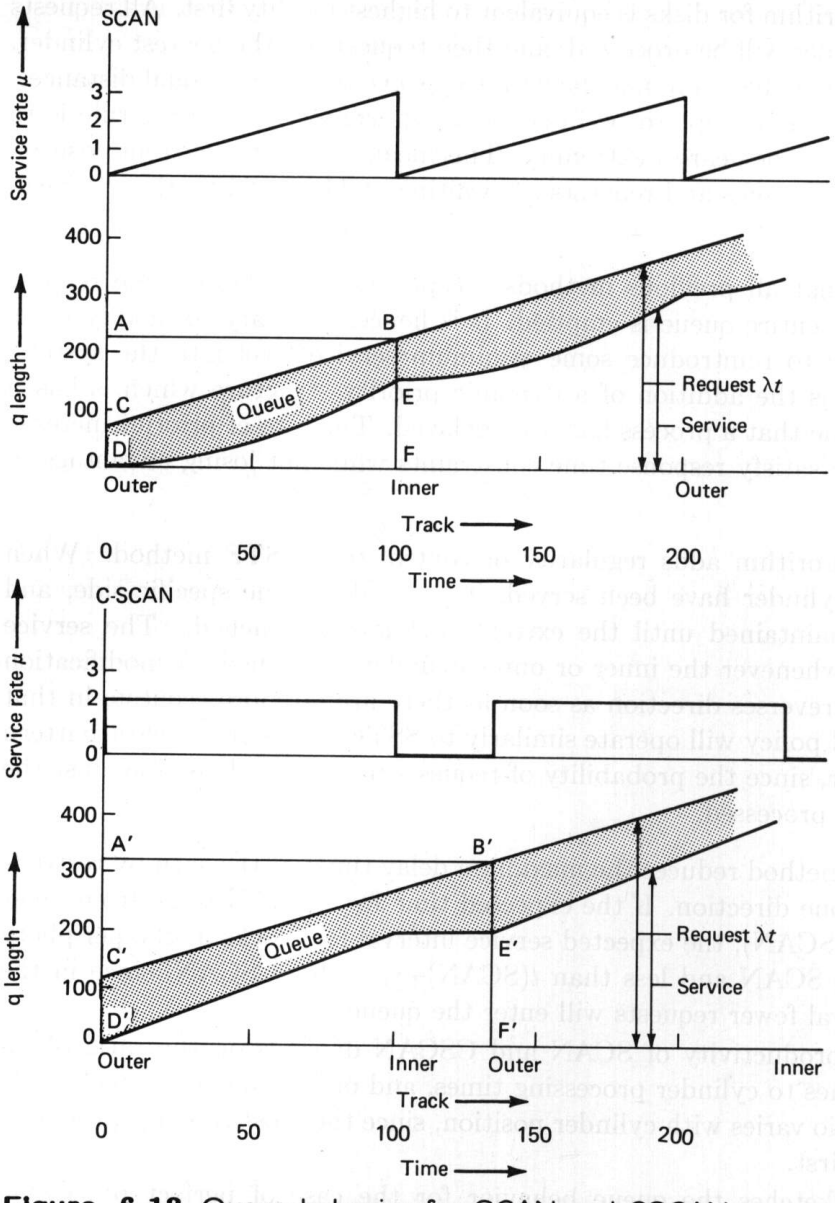
The CSCAN reduces the maximum delay times of the scan by restricting the SCAN to one direction. If the expected time for a SCAN to go from inner to outer track is t(SCAN), the expected service interval for blocks at the periphery is $2t$(SCAN) using SCAN and less than t(SCAN)$+s_{max}$ for CSCAN, since in the shorter time interval fewer requests will enter the queue.

The relative productivity of SCAN and CSCAN depends on the ratio of incremental seek times to cylinder processing times, and on the maximum seek time. For SCAN this ratio varies with cylinder position, since the most recently processed area is processed first.

Figure 6-12 sketches the queue behavior for the case of perfect processing equilibrium. The arrival rate is uniform (1.5 per unit of time). The top boundary of the

queue area B→C→ ... indicates the aggregate arrivals; during the period to point F their volume is proportionate to the area A,B,C. The lower boundary of the queue area D→E→ ... indicates requests that have been served and which have departed from the queue. The areas (A,B,C)=(D,E,F) and the areas (A′,B′,C′)= (D′,E′,F′), since under long-term equilibrium conditions arrivals and departures have to be equal in each cycle. SCAN encounters increasing numbers of requests per track as it goes back and forth. CSCAN encounters a steady load but is idle during the seek to the beginning of the cycle.

The seek time from inner to outer track s_{max} is taken to be one-third of the sum of single-track seek times ($j\, s_1$). It can be observed that the average queue length is slightly greater under CSCAN than under SCAN (about 100 versus 90), since during the time s_{max} no requests are serviced.



The simplifying assumption made here, that processing time is linearly related to track position and does not depend on the number of requests per cylinder, is valid only when the average number of requests per cylinder during a scan is ≤ 1, or at least much less than the number of blocks or sectors on a track, so that no additional rotational latency accrues.

Making Scanning More Fair SSTF, SCAN, and CSCAN may still fail to move the access arm for a long time. If, for instance, two processes have high access rates to one cylinder, they can monopolize the entire device by interleaving. High-density multisurface disks are more likely to be affected by this characteristic than disks with a low capacity per cylinder. To avoid forgetting requests for long periods, the request queues may be segmented, and one segment processed completely before any requests from the next segment are taken. This approach is refined in the N-step-SCAN and the FSCAN policies.

The N-step-SCAN places requests into subqueues of length N. Within a subqueue requests are processed using the SCAN policy. If fewer than N requests are ready at the end of a cycle, all of them are processed as a cycle.

With a large value of N, the performance of the N-step-SCAN approaches the performance of SCAN, while with $N = 1$, the service is equal to FIFO.

The scan has only to cover the range required for the N subqueue requests. User response time within a group of N requests might be best if SSTF were used within a cycle. Policies which leave the arm at the end of a cycle in an extreme position will contribute to the performance of the next cycle.

FSCAN is an implementation of an N-step-SCAN which uses as N the total length of the queue when a cycle commences, and defers all arrivals during the cycle to the next cycle. FSCAN shows improved efficiency as the load increases over CSCAN. The maximum queue length, however, increases to the number of arrivals per cycle. For the values used in Fig. 6-12 the average queue length becomes $1.5j/2 = 1.5 \cdot 200/2 = 150$ entries.

An *Eschenbach scan* extends the optimization used for a cylinder of a disk to the entire disk area. This is done by limiting the time to be spent at each cylinder position. A cylinder with a queue may be allotted between $1 \rightarrow k$ rotations, where k is the number of tracks on the cylinder. A parameter, E, is used to indicate the order of the Eschenbach scan. An order $E = k$ scan will obtain all the blocks from a cylinder position and is hence similar to a CSCAN, except that incremental requests requiring more than k revolutions will not be serviced in the current cycle. A small parameter of E $(1, 2, \ldots)$ will obtain blocks while the queue for the cylinder is still relatively long and will hence deal with higher average sector densities.

Example 6-14 shows the effect of the Eschenbach scheme applied to the single cylinder shown in Fig. 6-11. The results do not account for the small probability of new arrivals for this cylinder during its processing.

It is obvious that the Eschenbach scheme trades longer queues and greater waiting times for the capability to provide a very high utilization rate. It is clear that a high device-utilization rate will provide, at an equal request rate, better aggregate service to the user. For an Eschenbach scheme to be effective, the request rates will be high. This method has been used more in message switching and in

the collection of transactions for subsequent batch processing than in the area of standard file services.

Example 6-14 Utilization factors using Eschenbach scans.

E	Blocks accessed	*uf*
1	6 out of 8	0.75
2	+4 out of the next 8	0.625
3	+1 out of the next 8	0.457
4	+1 out of the next 8	0.375
5	Skip since track is empty	0.375

The Eschenbach scheme does not provide a service guarantee, but since the scanning cycle time, t_{cycle}, is guaranteed

$$t_{cycle} \leq j(s_1 + 2\,E\,r) + s_j \qquad \text{6-36}$$

where j is the number of cylinders allotted to the file, a service guarantee for priority messages can be provided. The required augmentation is the rearrangement of the sector queues to place priority requests at the head of the line. Unless a cylinder has for one sector more than E priority messages, t_{cycle} will be the maximum queue delay.

6-3-3 Use of Scheduling

Other techniques to optimize disk usage include the placement of frequently used blocks in the central cylinders of a disk. A SCAN policy will then access these cylinders most effectively; and even FIFO will show lower average seek times. FIFO and PRIority policies are frequently used because of their simplicity and the fact that the more complex scheduling policies excel at utilization rates not often reached in practice.

Scheduling is needed to overcome for poor performance of systems which collect transactions from many terminals for later batch processing (Winick[69]). If we represent data from a transaction coming from terminal T during the time interval I as (T,I), the file containing data from these transactions will be approximately arranged in the sequence

{ (1,1),(2,1),. . .,(t,1), (1,2),(2,2),. . .,(t,2), . . ., (1,i),(2,i),. . .,(t,i)}.

When this file has to be processed, the data from one source terminal repesents a batch, requiring access in the order { (1,1),(1,2),. . .,(i,1)}, followed by { (2,1),. . . , } etc. This transposition of access places extremely heavy loads on the file services supporting these systems, loads which can be considerably ameliorated by effective queue scheduling policies.

In many operating systems the scheduler is integrated with other functions, so that implemented scheduling policies are difficult to discern and nearly impossible to change. Since the scheduler, after making the scheduling decision, allocates not only resources but also assigns access privileges to users, as described in Chaps. 12 and 13, the interaction can be complex. Some concepts, for instance the notion of a *monitor* to protect critical sections of programs (BrinchHansen[73]), encourage such integration. The lack of modularity which exists in those systems can make it difficult to implement effective scheduling techniques.

6-4 OPERATIONS RESEARCH IN DATABASE DESIGN

Operations Research is an area of applied mathematics which addresses problems faced by the management of enterprises in the scheduling, operations, and development of facilities. In this context, Operations Research (OR) is closely identified with *management science*, although it is obvious that there is more to the science of management than the application of OR techniques.

Tools of OR include:

- Queuing theory
- Inventory theory
- Linear and integer programming
- Decision theory

Each of these fields is now a discipline in its own right.

6-4-1 Queuing Distributions and Their Application

Because of the importance of queues in manufacturing and communications, the subject of queues has been intensively studied. The parameters required for the mathematical treatment of queues include:

Description of the source of requests for service
- r.1 Number of requests
- r.2 Arrival rate distribution

Description of the queue waiting for service
- q.1 Queue capacity
- q.2 Scheduling policy

Description of the service
- s.1 Number of servers
- s.2 Service time distribution

The previous section has described queues as they are managed in file systems. A convention used to classify queuing models for analysis, *Kendall's notation*, specifies

arrival process/service process/number of servers,

using standard symbols for the distributions of arrival or service processes:

M	:	Exponential
D	:	Constant or otherwise determined
$E(m)$	:	Erlang
I	:	General and independent
G	:	General

Of interest to problems in file analysis are primarily queuing models of the type $M/G/1$ and $E/G/1$.

Events leading to queuing are described by their average arrival rate (λ). An alternative to the arrival rate is the distribution of interarrival times, t. In Fig. 6-12 the requests arrived uniformly distributed, the arrival rate λ was 1.5 per time unit, and the interarrival time t was a constant 0.66.

We have encountered exponential and constant distributions in Sec. 6-1. We associate exponential rates with randomness due to independence of successive events and constant rates with complete dependence of the arrival time on the time of

the preceding event. Erlang distributions describe the important case where the distribution of arrivals is neither completely random nor constant.

The servers are characterized by the potential service (μ) rates they represent. The service types in the systems we describe are dominated by seeks and latencies which can be determined using hardware parameters. Requests will depart from the system at a throughput rate, $\rho = uf\,\mu$, As seen earlier a high utilization, uf creates delays.

During equilibrium arrivals λ and departures ρ are equal. The number of departures is limited by the number of arrivals: at any time t_i after initialization $\lambda\, t_i \geq \rho\, t_i$ and any unprocessed arrivals are held in queues.

When a service request addresses a specific file, the applicable number of servers is one; only in communication networks between processors, access to fully redundant files, or when writing without specifying the disk address can the case of multiple servers occur.

Erlang Distributions When the demands for service to file and communications systems are measured, it is often found that the times between services requests are distributed in a pattern which is due neither to completely random nor to constant behavior. The behavior may be explained by the fact that after a service request has been granted, the time for the service and analysis of the obtained data is not at all random, so that one computation will emit service requests in a cyclical pattern. Similar but randomly interleaving patterns are presented by other, independent computations being processed.

To capture the range between random and constant interarrival times a parameterized distribution, Erlang(m), can be used. Figure 6-13 shows some Erlang distributions for an average arrival rate, $\lambda = 1$. Figure 6-13 and the accompanying Eq. 6-32 show that the Erlang distribution for $m = 1$ is equal to the exponential distribution, while as $m \to \infty$ the distribution becomes constant. The parameters for the Erlang distribution are

$$mean(t) = \frac{1}{\lambda} \qquad \text{(of course) and} \qquad \sigma(t) = \frac{1}{\sqrt{m}\,\lambda} \tag{6-31}$$

Table 6-13 Computation of Erlang parameter, `m`.

```
/* Computation of Erlang Parameter */                    /* 6-13 */
      m_erlang = (mean/sd)**2  ;
```

The appropriate Erlang parameter, m, for an observed Erlang-like distribution can hence be found from the computed mean and standard deviation, as shown in the program statement of Table 6-13. High values of `m_erlang` (> 10) indicate little variability in interarrival times.

The frequency distribution for an Erlang distribution is

$$freq(t) = \frac{(\lambda m)^m t^{m-1}}{(m-1)!} e^{-m\lambda\, t} \quad \text{for } t \geq 0 \tag{6-32}$$

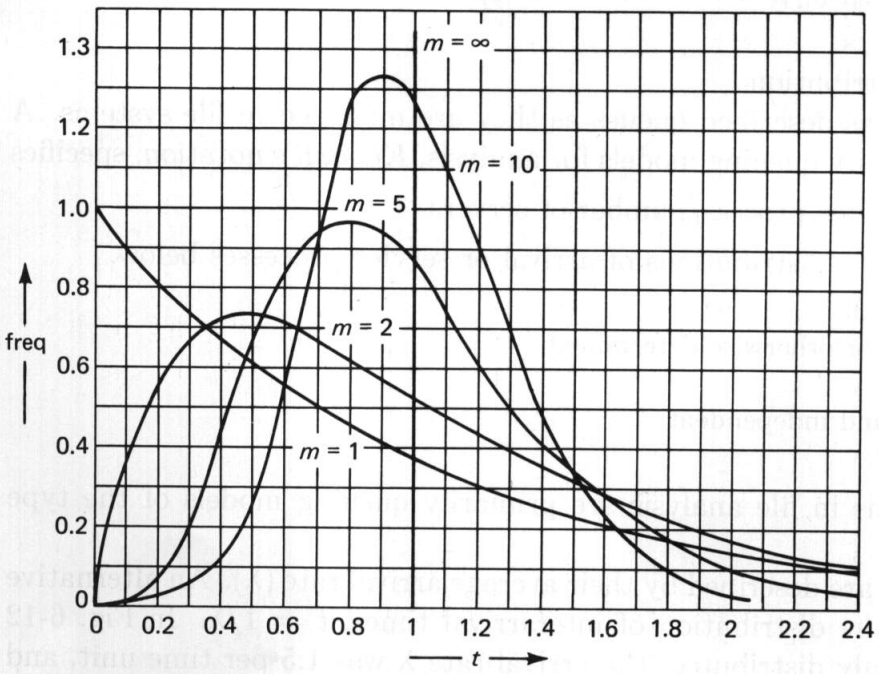


Figure 6-13 Erlang distributions.

The Effect of Variability That random service times or service requests have a detrimental effect on service performance can be demonstrated with the simple example of Fig. 6-14. The probability to arrive at a *bad* time is 50/fully compensated by *good* times.

There are two disks which provide service at mean intervals of 30 ms. The expected delay for one randomly arriving request arriving in the interval $(0, 120)$ to receive service from disk D1 is 15 ms, but the delay for that request for disk service from D2 is

$$\tfrac{15}{120}\tfrac{1}{2}(15) + \tfrac{15}{120}\tfrac{1}{2}(15) + \tfrac{60}{120}\tfrac{1}{2}(60) + \tfrac{30}{120}\tfrac{1}{2}(30) = 20.6\text{ms}$$

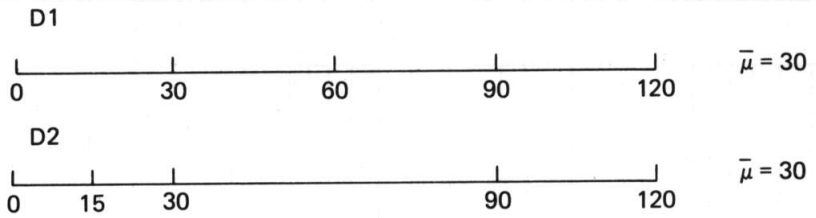


Figure 6-14 Disk service times.

Queue Length and Delays for Erlang Distributions When the order of the Erlang distribution for requests, and the utilization factor of the service facility are known, the queuing behavior is defined. Formulas and graphs (IBM F20-7[71]) can be used to compute queue length and delay times given `m_erlang` and `uf`. The computations required are shown in Table 6-14.

Table 6-14 Computation of expected queue length and waiting time.

```
/* Computation of Expected Queue Length */              / * 6-14a */
      idlef = 1 - uf;   ufsq = uf**2;
      factor1 = 1 - 1/m_erlang;   factor2 = 1 - uf/2*factor1;
 /* Queue Length */
      queue_bar = uf * factor2/idlef;
      queue_sd = SQRT(uf * (1 - uf/2*(3
                  - (10*uf-ufsq)/6 - (3*idlef + ufsq)/m_erlang
                  - (8*uf-5*ufsq)/(6*m_erlang**2)) ) ) / idlef;
/* Waiting Time before Begin of Service */               /* 6-14b */
      w_bar = (s+r+btt) * factor2/idlef;
      w_sd = (s+r+btt) *
          SQRT( (1-(4*uf-ufsq)/6*factor1)*(1+1/m_erlang)
                                              - factor2**2) / idlef;
```

The availability of computers is appreciated when Erlang distributions are used.

The factor `idlef=1-uf` had already been encountered in the analysis of cylinder scheduling. It accounts for the rapid decline of performance when a high utilization is achieved while arrival or service times show random or stochastic variation.

Example 6-15 Calculation of a queue length.

Given that the times needed to fetch a large number of records from a file system have been collected. The mean time was 90 ms, and the standard deviation 40 ms. A histogram of the distribution makes an Erlang distribution likely. The appropriate Erlang parameter, according to the statement in Table 6-13, is

```
m_erlang  = (90/40)**2  ≈ 5.0
```

The disk unit was busy during the test 40% of the time, so that `uf = 0.4`. The queue length expected is computed, using the program in Table 6-14a as

```
queue_bar = 0.56
queue_sd  = 0.84
```

Given that the mean service time `s+r+btt` is 50 ms we find from the program segment in Table 6-14b the mean delay and its standard deviation

```
w_bar = 70.0 ms
w_sd  = 42.8 ms
```

Use of Percentiles to Define Performance The type of distribution with its mean and standard deviation defines the distribution of events in a system precisely. The result has to be translated into terms useful for operational decisions. The use of fractions or percentiles of occurrence of certain events is a well-understood approach. Figure 6-15 provides this information for Erlang and hence exponential distributions.

It provides the values of t where the areas of the Erlang distribution from $0 \rightarrow t$ are 90%, 95%, and 99% of the total area. Events that fall into the tail, from $t \rightarrow \infty$ will have longer delays. The result is that the waiting time is less than $w(p)/\overline{w}$ times the average waiting time $\overline{w}$ in p% of the cases.

Example 6-16 Waiting time limit using Erlang and Chebyshev assumptions.

Let us provide the waiting-time limit for 90% of the record fetches in the example above. For `m_erlang = 5` the 90th percentile in Fig. 6-15 gives a ratio of 1.6, or a delay of $1.6 \cdot 70 = 112$ ms.

A *Chebyshev* inequality can only promise for the 90th percentile a delay of less than $70 + \sqrt{10}\,42.8 = 205$ ms, and this is the value to be used if the distribution does not match the Erlang curve for `m_erlang = 5`.

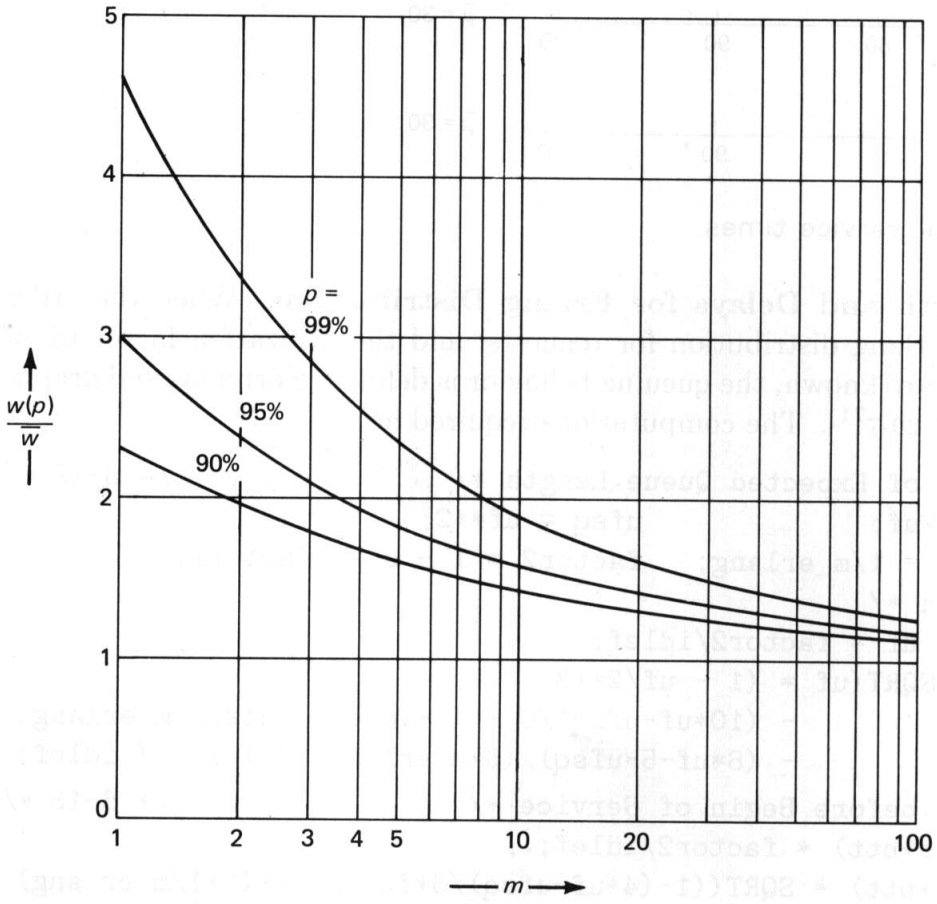


Figure 6-15 Percentiles of Erlang distributions.

Mix of Distributions When the behavior of a mixture of different services is needed, the joint distribution and the mean will be the sum of the scaled distributions. The expected mean and standard deviation in the mix will be

$$mean_{mix} = \sum_h pc_h mean_h \qquad 6\text{-}33$$

$$\sigma_{mix} = \sqrt{\sum_h pc_h^2(\sigma_h^2 + (mean_h - mean_{mix})^2} \qquad 6\text{-}34$$

where pc_h is the proportion of the computation h in the mix (IBM F20-7[71]).

6-4-2 Transients in Service Demands

Up to this point we have discussed queuing models which are in equilibrium, so that within all intervals considered, the service rate was adequate to provide the requested services. In many systems, however, there will be certain periods where the rate of arrival of requests for service, λ, exceeds the service capacity, μ. Such a period is called a *transient*, since it can only be a temporary situation; a situation where the average rate of arrivals, $\overline{\lambda}$, remains greater than μ is of course intolerable.

Examples in Sec. 6-3 have demonstrated how rapidly the expected queue length and the wait time, w, can rise as the utilization factor, $uf = \overline{\lambda}/\mu$, approaches one. If later $\lambda \rightarrow \ll \mu$, the queues will empty and $w \rightarrow 0$. Unused computing capacity is a very perishable commodity; it cannot be put into a bank to provide later benefits.

Transients are fairly difficult to analyze using the probabilistic techniques used earlier, so that graphic constructions based on flow concepts will be used here. We will consider now only the part of a queue due to overload and not the part of the queue which is generated because of the delays caused by seek and rotational latency.

Figure 6-16 shows the cumulative arrivals as line A and the cumulative service requests which are being processed as line D. Processing of requests (D) is limited both by the slope μ giving the service rate capacity and by the arrival of requests, A. The space between A and D depicts the queue which is generated. The queue begins forming at $t = t_0$ where for the first time the instantaneous arrival rate $\lambda(t) > \mu$. At t_2 the arrival rate has subsided and $\lambda(t) = \mu$, but the queue has yet to be worked off. The queue will not be reduced to zero until time t_3, where D, continued at angle μ from t_0, intersects A. The average rate of arrivals λ during the total transient between t_0 and t_3 is equal to μ, but the overall $\overline{\lambda}$ in the cycle of t_c is less than the available service rate μ. The values are given below.

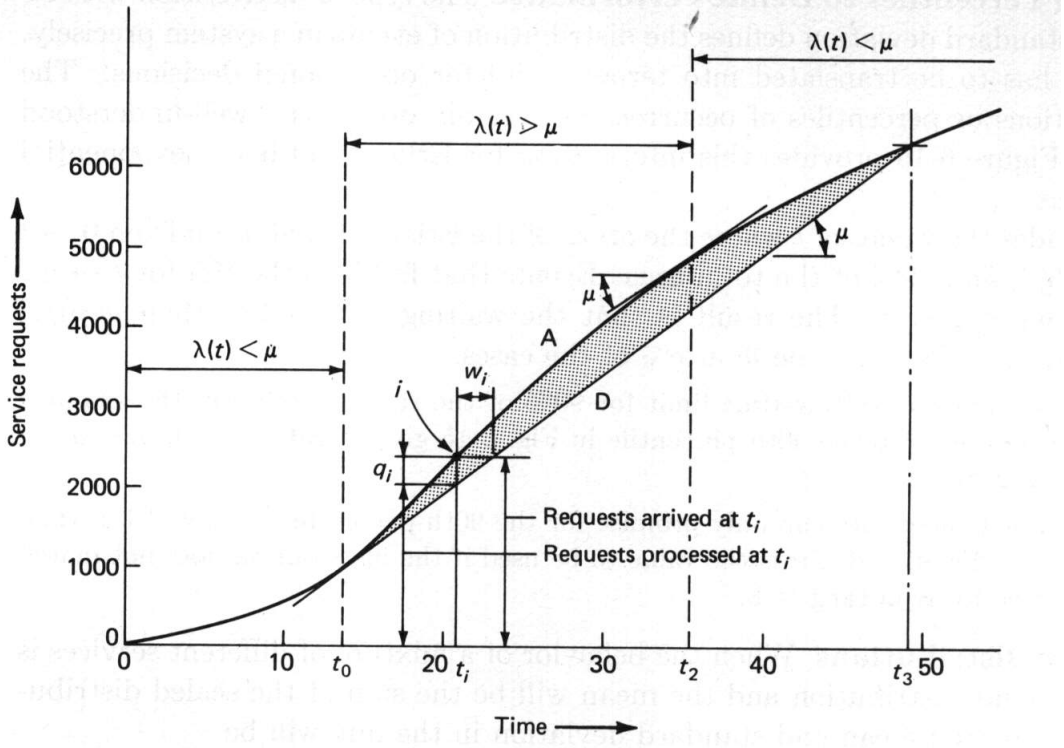
If the shape of the arrival distribution, A, is known, a graphical construction can provide the desired values of queue lengths. This graph can easily be interpreted, for instance, for a FIFO scheduling discipline the waiting time for an arrival at t_i, when the queue length is q_i, is indicated by line w_i.

In order to treat a transient analytically, an assumption regarding its shape has to be made. The simplest relationship which can describe a cyclic transient of $\lambda(t)$ is a sine curve. In Fig. 6-17 we show this curve with a basic cycle of 100 s. To this curve are applied three arrival and service parameters, as listed.

Figure 6-16 shows the queue beginning to form at t_0, where $\lambda(t) = \mu$. Substitution of this value in the sine-wave equation for $\lambda(t)$ allows computation of this point. The arrival rate is maximal at $t_1 = t_c/4$. Symmetry about t_1 determines t_2.

Data for Figures 6-16 and 6-17

Cycle time, t_c	`cycle`	100 s
Average arrival rate, $\overline{\lambda}$	`lambdabar`	100/s
Maximum arrival rate, λ_{max}	`lambdamax`	165/s
Service rate, μ	`mu`	150/s



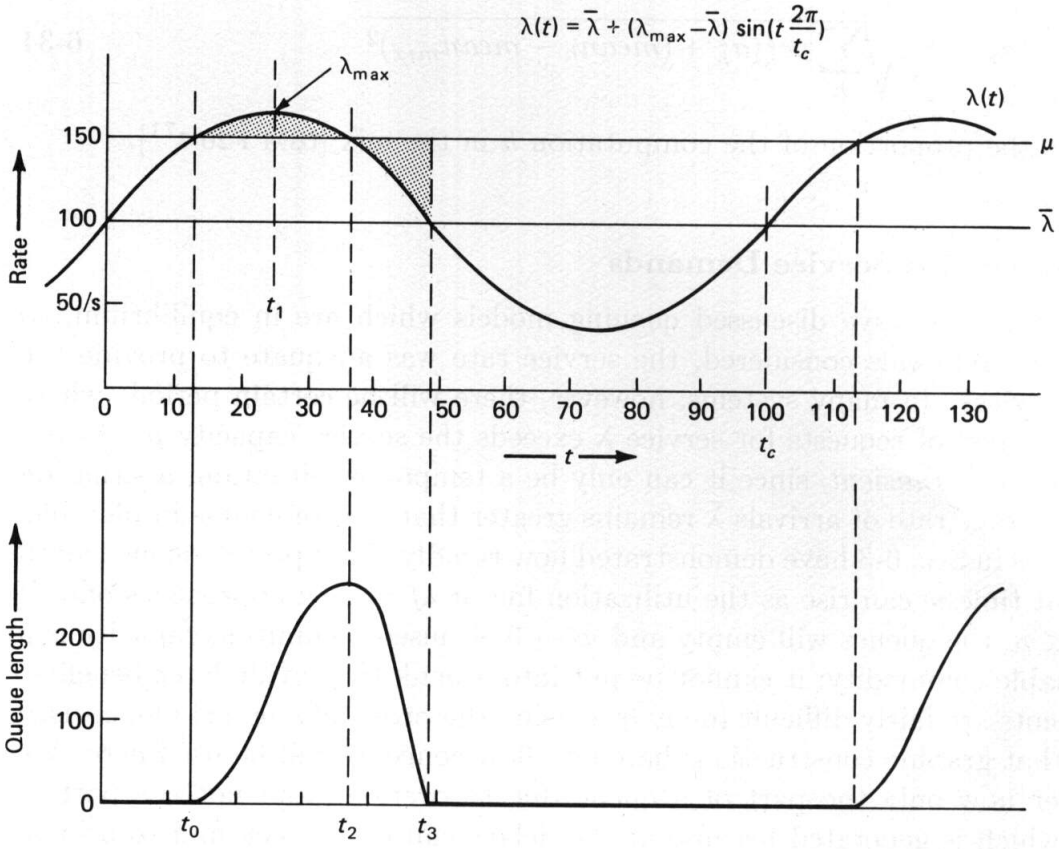


Figure 6-17 Transient queue.

Table 6-15 Computation of queue times for transient.

```
/* Compute Queue Times for Transient Arrivals */          /* 6-15 */
         two_pi = 2 * 3.14159;
         sine_t0 = (mu-lambdabar)/(lambdamax-lambdabar);
         t0 = cycle * ARCSIN(sine_t0)/two_pi;
         t1 = cycle/4;
         t2 = t1 + (t1-t0);
```

Executing this program gives for the example values

```
        t0 = 13.6 s
        t1 = 25.0 s
        t2 = 36.4 s
```

The queue length at `t2` can be computed by analytical integration between `t0` and `t2` or by finite, stepwise integration. The time when the queue vanishes, `t3`, is tedious to derive analytically, so that a simple stepwise computation will be used which simulates the queue behavior. This computation will step through time and estimate arrivals λ using the assumed *sine* function superimposed on $\overline{\lambda}$. When λ exceeds the possible service rate, μ, the excess arrivals are added to a queue. If there is a queue, it is decreased by any excess $\mu - \lambda$. When the queue length becomes zero, the time `t3` is computed and printed. At each second of the cycle the queue length is reported.

Table 6-16 Queue length and total time for transient.

```
/* Queue Length and Total Time for Transient */           /* 6-16 */
      radius = lambdamax - lambdabar;
      queue = 0;
      lastlambda = lambdabar;
      DO second = 1 TO cycle;   angle = second/cycle*two_pi;
         lambda = radius*SIN(angle) + lambdabar;
         IF lambda>mu THEN
 /* Approximate Arrivals by Average in Interval */
            queue = queue + (lastlambda+lambda)/2 - mu;
         ELSE DO;
            IF queue > 0 THEN DO;
    /* Work off Queue */
               decrease = mu - (lastlambda+lambda)/2;
               IF decrease < queue THEN queue = queue - decrease;
               ELSE DO;
       /* queue is gone, interpolate between seconds */
                  t3 = second - 1 + queue/decrease;
                  queue = 0;
                  PUT DATA(t3,queue);
               END;
            END;
         END;
         lastlambda=lambda;
         PUT DATA(second, queue);
      END period;
```

Example 6-17 The three phases of a transient.

Applying the program of Table 6-16 to our example computes a completion time of `t3 = 48.2 s`, so that there are here three similar intervals,

Phase 1:	Increasing rate of queue growth: t1-t0 = 11.4 s
Phase 2:	Decreasing rate of queue growth: t2-t1 = 11.4 s
Phase 3:	Queue collapse : t3-t2 = 11.8 s

‡‡‡‡‡ **Behavior of Transients** The pattern of equal thirds observed in Example 6-17 can also be analytically deduced. The time to work off the queue depends on the ratio of $\overline{\lambda}$ to μ. If there is a reasonable margin between $\overline{\lambda}$ and μ, the transient takes less than half of the total cycle, t_c. If now $t_3 \approx \frac{1}{2}t_c$ we find four phases in a cycle as shown in Fig. 6-17: no queue from 0 to t_0 and the three phases of queue growth increase, growth decrease, and collapse to t_3. In the last phase t_2, t_3 the steepness of the arrival rate λ is about twice as great for cyclic or quadratic functions as the steepness of λ during growth in t_0, t_1 and t_1, t_2. Because of this difference the queue will collapse in about half the time it took to grow. The two phases of growth are equal because of the symmetry we assumed for the cycle. The reasoning of double steepness and symmetry continues to hold for transients with $t_3 < \frac{1}{2}t_c$ or transient lengths $t_3 - t_0 < \frac{3}{8}t_c$. This pattern of equal thirds is typical and has also been concluded for other second-order assumptions.

When transients increase in length, the time to work off the queue increases relative to the transient length. At the limit for long-term operation, $\overline{\lambda} = \mu$, $t_2 - t_0 = t_3 - t_2$ and the time for the queue to grow is equal to the time needed for the queue to disappear. This can be easily visualized in Fig. 6-17 by reducing μ to 100; then t_3 occurs at the end of the cycle, namely, at $t_c = 100$ s.

When the transients are not due to cyclic phenomena, the same analysis method can still be used by fitting the sine curve according to observed values of any two of the time points `{t0,t1,t2,t3}` in addition to using the service and arrival rates as before.

The expected queue length, as computed here, is entirely due to the transient, and additional queue capacity is required for random variations of arrivals and service, and for hardware-caused delays. ‡‡‡‡‡

6-4-3 An Application of Inventory Theory

In classical inventory theory, there is a collection of goods which are gradually consumed. Before the inventory is exhausted, an order is to be placed, so that the supply is replenished. A cost is associated with placing an order, with storing the inventory, and with having run out of the goods. There are many models available to solve inventory problems.

A file or database tends to become less efficient over time until an order to reorganize is placed, so that its behavior presents a similar cost function to the one shown in Fig. 6-18. The cost functions depend on the overflow parameter, o, seen throughout Chap. 3. In direct files the cost is related to the degree of clustering and to the file density.

After reorganization at a cost $C_Y = f(T_Y)$, the file will be more efficient. The original efficiency should be regained unless the file has grown. If statistics

about record activity are available, reorganization can be used to improve the file efficiency. While the method and its effect depend on the file organization, usage costs will be lower if records of highest activity are reloaded first. In a direct file, for instance, the frequently accessed records will be found at the initial address.

We will consider only the case of a static file because the principle of the analysis remains the same. The reorganization cost is equivalent to the cost of placing an order, and the time required between the placing of the order and the receipt of the goods can be ignored.

The cost of operating the file system can be split into the component for the static portion $n\,C_n$ and for the overflow portion $o\,C_o$, where $C_{n,o}$ are the costs per record of the file fragments. To simplify matters further, a linear growth of overflow o over time t is assumed, so that $o = c_o t$, $t = 0 \rightarrow t_Y$. The cost of overflows at a given instant can be written as $t\,c_o C_o = t\,C'_o$. Reorganizations are performed after an interval t_Y and the cost of operating the system during such an interval can be estimated from the two areas shown in Fig. 6-17 and the cost of reorganization

$$C_{op}(t_Y) = t_Y n\,C_n + \frac{1}{2} t_Y^2 C'_o + C_Y \qquad 6\text{-}35$$

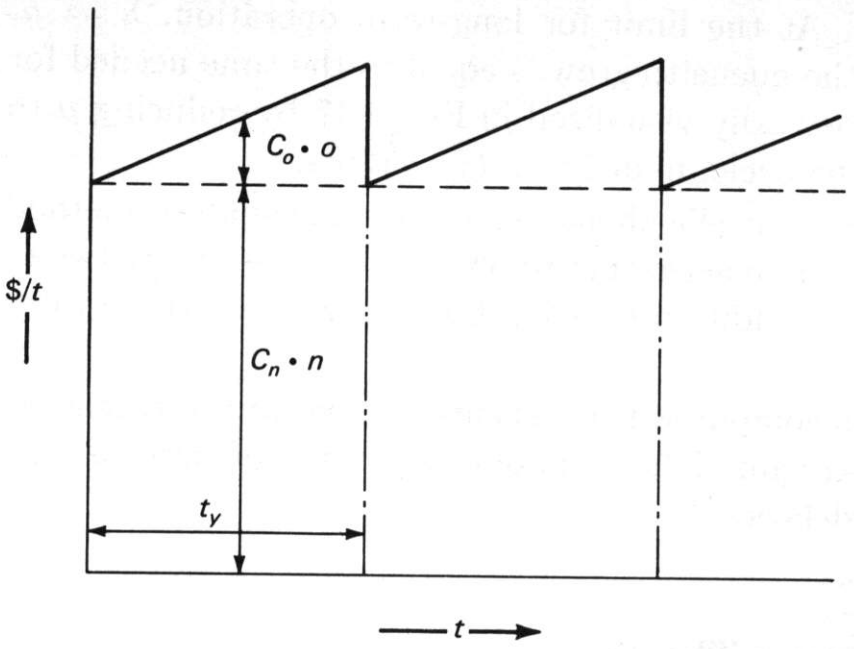


Figure 6-18 Cost function of a file which is being updated.

During a long period, Per, there will be Per/t_Y reorganizations and the cost over Per will be

$$C_{op}(Per) = Per\,n\,C_n + \frac{1}{2} Per\,t_Y C'_o + Per\,\frac{C_Y}{t_Y} \qquad 6\text{-}36$$

Example 6-18 Calculation of reorganization period.

Given are load quantites of

$$L_F = 20\,000/\text{day} \quad \texttt{fetch requests}$$
$$L_I = 300/\text{day} \quad \texttt{new records, causing overflow } o$$

The file has $n = 40000$ records of length $R = 500$ on blocks of size $B = 2000$, the index has a fanout $y = 100$. We derive $Bfr = B/R = 4$, $x = \log_{100}(n/Bfr) = 2$. We assume that the top index level x is in core, also that levels 1 to $(x-1)$ are on the same cylinder as the data and that $n/o \gg Bfr$, so that we can use Eq. 3-34, as derived in Eq. 3-38. We simplify the overflow term $o'/(n+o')$ to o/n, since we use o here to denote the current overflow and $o \ll n$.

$$T_F = c + s + 2(r + btt) + \frac{o}{n}(r + btt)$$
$$T_Y = 2n\frac{R}{t'}\frac{SI}{t}$$

The low ratio of insertions to fetch operations $L_I/L_F = 300/20\,000 = 1.5\%$ permits T_I to be ignored.
Continuing the estimation process, it is assumed that the index SI requires 5% space over the rest of the file and hence a proportionate effect to rewrite. Then

$$C_Y = 2.05\, n\, \frac{R}{t'}\, 0.5\, cost$$

For a 2314-type disk unit the parameters for this file will be $r = 0.025$ s
$t' = 120\,000$ bytes/s
$btt = 0.005$ s and for simplicity, we set the processor cost at

$$cost = \$0.10/\text{s}$$

The actual absolute values of these parameters are not very critical since they tend to cancel each other out. With this $cost$

$$C_Y = \$17.10 \qquad \text{and} \qquad C'_o = \$0.45/day^2$$

so that

$$t_Y(\text{optimum}) = 8.7 \text{ days}$$

The lowest cost as a function of the reorganization interval can be found by taking the derivative with respect to t_Y and setting it to zero.

$$\frac{1}{2}PerC'_o - Per\frac{C_Y}{t_Y^2} = 0 \qquad \text{giving} \qquad t_Y(optimal) = \sqrt{2\frac{C_Y}{C'_o}} \qquad \text{6-37}$$

This rule is applied in Example 6-18 to the indexed-sequential file presented in Chap. 3-3.

For an indexed-sequential file the relationship for $C_o = f(T_F)$ is, given the assumptions in the example, linear with respect to o and T_Y is independent, as required by the assumptions made in the model used to optimize the reorganization time. Then

$$C_o = \frac{r + btt}{n} \qquad 6\text{-}38$$

and with one overflow per insertion L_I

$$C'_o = L_F L_I (r + btt) \frac{cost}{n} \qquad 6\text{-}39$$

where *cost* is the processing cost per second. The reorganization will be scheduled for lower cost time, so that $0.5\,cost$ was used in Fig. 6-36 to compute C_Y.

6-5 FREE STORAGE MANAGEMENT

A function of the operating system is the allocation of storage space to the separate files as indicated in Chap. 1-7. There is an interaction between file structure and allocation policy: file systems that do not use indexes or pointers to locate bocks require contiguous space; other files types can receive space incrementally. Two methods to keep track of the *p portions* allocated to files and of the space which remains free have already been encountered in Chap. 4-3 (VSAM) and 4-5 (MUMPS). In this section we will review this subject more completely.

The assignment problem becomes more complex when a computer system has to support a variety of files of different types. We see at times more than one allocation scheme in use. VSAM, for instance, has its own suballocation scheme within the standard IBM operating system.

Initial State In the beginning the system establishes the file area required for its operation and then defines the areas available for users. The user's area is without form and void. The first task for the system is the creation of a directory to all available storage units. During operation the system has to identify removable diskpacks, tapes, etc., which are currently mounted. The storage-device directory is re-created whenever the system is initialized, and directory entries are updated upon receipt of a "ready" signal from a device on which a diskpack or such has been mounted.

On many machines new diskpacks or floppy disks have to be formatted before use. Formatting defines the blocks that the system will use, giving each block an identifier for future reference. During formatting a verification may take place. Any blocks found that cannot be successfully written and read may be marked "bad" and omitted from the set of blocks available for allocation.

The initialization process has established the *free space* available in the system.

6-5-1 Assigning Portions

When the free space has been defined, users can come forth and demand their share of the resources. We use the term *portion* to designate a unit being assigned to a user's file. The minimum size of a portion will be one block, but if locality is important then portions may consist of many blocks at a time.

We note that the allocation of space is a function carried out at a level which is more basic than the level on which the file systems operate. When multiple file systems share one computer system, the operating system may have to support more than one space-allocation strategy. Good space management is essential to keep the system's ecology in balance.

Portions, Size versus Number Space will be allocated to users in response to their requests. The shape, number, and size of the portions of the system storage space given to users is a compromise between the requirements for efficiency by the individual files and overall system efficiency. The items to be considered in the trade-off are:

1 Contiguity of space increases performance, especially for `get_next` operations, and greatly for transactions running in a transaction-oriented operating system.
2 Having a large number of portions increases the size of tables needed to manage the allocation information.
3 Having fixed-size, simple portions, for instance blocks, simplifies the reallocation of space.
4 Having variable size or small portions minimizes the waste of unused storage space due to overallocation.

These items interact, and there are two major alternatives: variable, large, contiguous portions or blocks. We can summarize the trade-off of the alternatives as follows:

1 Large portions improve performance. They should be variable in size to avoid excessive waste. They can use small allocation tables, but the space is hard to reuse.
2 Small portions improve flexibility. They are typically of fixed size, but may require large tables or complex structures for their allocation. Contiguity has been abandoned, blocks are allocated incrementally as needed.

To achieve high performance it is best to allocate large portions. Some users and some file systems request large amounts of storage prior to need, even if incremental allocation is the provided alternative, and they allocate that space to their files internally. This was the approach used in VSAM, where the user can prespecify eventual file size. With large portions much space may remain unused, but the user will pay for the excess.

If contiguous space is to be allocated, large portions are needed to make the contiguity effective. To prevent waste of storage space for small files, the portions will be of various sizes. Allocations may be restricted to a small number of portions; this number is, for instance, limited to 16 in IBM'S OS. The tables used to keep track of allocated storage in OS are relatively small. Some operating systems are able to utilize the extra space given when the user is done.

In systems with a high degree of multiprogramming it is not very beneficial to provide contiguous space for users, so that allocation can be based on blocks (CDC SIS) or trains of blocks (IBM VSAM). The user's request will be rounded to the next larger unit.

Allocation by variable portion leaves more total free space, but in less usable form, than allocation by block or train. The storage efficiency of block allocation improves with smaller block or train sizes. When allocating blocks or small multiples of blocks, the storage allocation tables will be larger.

Figure 6-19 sketches a storage space layout after an identical space request and release sequence for variable and block allocation. We see that after the release of the variable portions used by process C some space could be used by successive requests of processes D and B, but fragments of those spaces remained unused. It is likely that small fragments will never be used. The blocked allocation assigns more space intially, but uses less space in the end.

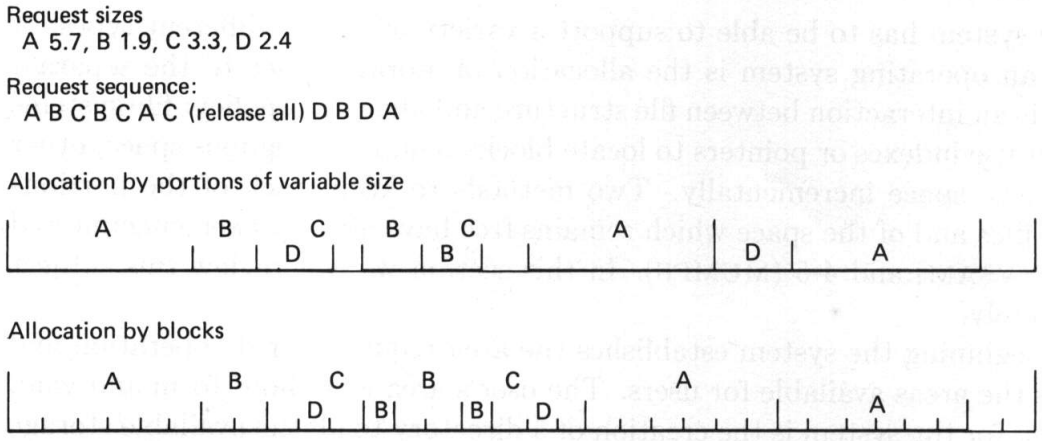


Figure 6-19 Storage-allocation alternatives.

A Dynamic Allocation Policy A scheme to avoid the problem of preallocation versus waste or lack of contiguity is to allocate portions of increasing size as the file grows. If the next portion is always doubled, the size of the portion table for the file is limited to $\log_2(n/Bfr)$ entries. Less than half the allocated file space is unused at any time.

This scheme can be improved by allocating fractional portions, say, four portions of 25% of the previous allocation, before doubling the allocation size. Now the portion table is four times as large, but a file of 10^6 blocks still requires only 80 entries. Less than 25% of the allocated file space is unused at any point.

6-5-2 Management of Allocated Portions

Since external storage is typically allocated over long periods of time, the storage-control tables reside on disk. The record of allocated space has to be updated whenever a portion is allocated to avoid problems when the system fails.

Consider the scenario of Example 6-19.

Example 6-19 Storage allocation disaster.

Step 1 A user, `Y`, stores information on a file in the portion allocated to user `Y`.
Step 2 The system fails before the allocation information is persistently recorded.
Step 3 The system is restarted.
Step 4 Another user, `M`, requests storage and is allocated the same portion, or a part of it.
Step 5 User `Y` accesses the same portion via a reference kept inside the file.

Some schemes can prevent such disasters.

1 One can delay user access to the portion until the update of the allocation table has been completed.
2 One can check on restart after a failure that all blocks used on disk are in the proper allocation table.
3 One can mediate all accesses to disk and verify that any block accessed is allocated to the user or file giving the access command.

In the latter two schemes additional storage is taken from each block to maintain the identification.

The second scheme has a very high cost when a failure occurs, since all blocks have to be read. The cost can be reduced if it is only performed for files that were in use or open when the disaster occurred. In that case, the timestamp for the last `CLOSE` of the file precedes the `OPEN` timestamp. The cost for the second scheme can also be deferred by only checking when a file is `OPEN`ed again for use, but at that time more damage may have been done to that file.

The third scheme distributes a small loss in performance over all accesses, independent of the failure rate. When the inherent failure rate is low, as we hope, its cost will be relatively greater.

The first scheme is the only one that actually prevents errors. It has, however, a significant effect on performance when small portions are allocated frequently. The cost of keeping storage secure in this manner can triple the users' cost of writing a new block. To avoid this overhead, a batch of entries denoting free portions can be kept in a working area in core storage. The algorithm in Table 6-17 is intended to keep the *batch storage allocation* secure.

Table 6-17 Batch storage allocation.

1 Obtain batch of free portions from the storage-control table on file for allocation into a work area.
2 Mark these portions in the work area as `"in use"`.
3 Rewrite the storage-control information to storage.
4 Allocate portions from the work area to the users as requests are received, and identify the portions allocated.
5 When the batch is used up, rewrite the storage-control table again to incorporate the actual assignment information.
6 Return to step 1.

With a batch storage allocation scheme, there will be entries for portions after a failure marked "`owned by user X`", entries for portions marked "`free`", and entries

for portions marked only "`in use`". The latter portions will not be reallocated. If the allocated portions themselves are identified with their owner, date, and time of most recent `WRITE` operation (as required by the other schemes), it is possible to check the portions whose entries are marked "`in use`" when they are again retrieved by the user and correct the storage-control table eventually. Other techniques for cleanup are possible, depending on the file system's protocol.

Cleanup and other housekeeping tasks may be deferred and combined with file-reorganization tasks, since the storage-control table remains usable. It helps to use a verification such as alternative scheme 2 above routinely when reorganizing files. Chapter 13-3 discusses some long-term aspects of file maintenance.

Systems which ignore the problem altogether lose robustness, and the user may waste much time picking the pieces up after a failure. Even users who were not aware of the failure can be subject to damage.

Allocation of Free Storage The storage-control table also contains, explicitly or implicitly, information about the external storage areas which are not yet allocated and hence free to be given away. To assign storage to a user from the free space, a section of the storage-control table is fetched from the disk and analyzed to determine which space to give to the requesting user. When variable portions are used, decisions have to be made whether to allocate according to *first fit* (done above), *best fit*, or closest to the previous allocation for the file to increase locality. It is not clear which strategy is best.

6-5-3 Storage-Control Tables

The structure of the storage-control table is affected by the allocation procedure it has to support. Three methods are in common use to define the available storage space and its allocation:

1 Table of contents
2 Chaining of portions
3 Bit tables

We will discuss these in turn.

Table of Contents A file maintained by the system on every unit may be used to describe space allocation, using one or several records per file. A file may have an extensive description. Typically included are such items as the owner's identification, the file name, the date of creation, the date of most recent use, the date of most recent updating, and for every portion the position and size. This technique is typically used for systems which allocate large portions. Such a *table of contents* is kept on every disk in the system and is found from a fixed position on every unit.

Free portions could be located by searching the table and finding unallocated space, but this is apt to be costly. To avoid this cost, a dummy file with owner `SYSTEM` and name `FREE_SPACE` is kept in the table of contents. Since the number of portions of free space can be larger than the limit of portions per single file, the free portions may be chained to each other and found from a single or a few entries.

Chaining of Portions The free portions may be chained together through the free-space portions themselves. This method is also applicable when portions are smaller, for instance, in block-based allocation. It has negligible space overhead since it uses free space for storage of the free storage-control data. Only one, or a few, header entries are kept in a pseudofile entry. When space is needed, the headers from the portions are fetched one by one to determine the next suitable free portion in the chain. Then the portion is removed from the chain and the chain is rewritten, taking care again not to leave the free storage control data vulnerable to system failure.

Multiple chains may be kept to group cylinders or sectors in order to create efficient user files. MUMPS uses free chains in that manner (Fig. 4-28). If variable-length portions are being allocated, there may be multiple chains to implement a best-fit algorithm. Files will obtain portions from the chain appropriate for the size of portions required, and large portions are preserved until needed.

Chained free-storage management does not keep track of data allocated to the user's file. It is desirable for the operating system to be cognizant of all allocations made. When there is no table of contents, this can be achieved by an extension of the chaining method. The allocated portions are chained together for each file using a header similar to the header used for free storage management. The relative space overhead to do this depends on the size of the portions allocated but will be bearable unless portions are of a size much less than a thousand bytes.

Bit Tables A third approach to manage free space is to keep a *bit table*. This method uses an array containing one bit per portion in the system. It is used only when all portion sizes are the same, generally equal to one block. Each entry of a "0" indicates a corresponding free portion, and a "1" indicates a portion which is in use. The list of portions allocated to a specific file is kept as part of the file mechanism. A bit table has the advantage that large sections of the table can be kept in core memory and that the allocation and deallocation cost is minimal. This advantage is important in highly dynamic environments. The tables, on the other hand, are quite vulnerable to errors, so that they tend to be practical only where the file system keeps track of its allocation in a manner which can be used by the operating system. It is then possible to reconstruct or verify the bit table by checking through all the individual file tables.

File systems themselves can also use bit tables to locate their member portions. The use of bit tables is especially effective where the portion-allocation data does not have to be kept in serial order. Bit tables for a specific file tend to be very sparse, but their efficiency can be increased by a technique of packing. The packing of a sparse bit table is achieved by indexing groups of the bit array. A group which has only zero entries is indexed by a zero, and not actually represented. The index itself can be recursively grouped and indexed. Figure 6-20 illustrates this technique for File B. Bit arrays have also been used where portions, or records, are being shared by multiple files. Since no space is required within the portion, there is no limit to the number of owners that such a portion can have.

We see that there are a number of possibly conflicting objectives in the design of a free-storage management algorithm. The primary conflict is again speed versus

space. To increase file performance, it is desirable to provide large portions, and also to locate new portions close to the predecessor portions. To decrease the cost of space management, one may also wish to allocate large portions, which reduces the frequency of allocation as well as the size of the required tables and lists. To use storage space well, it is desirable to allocate variable or small portions.

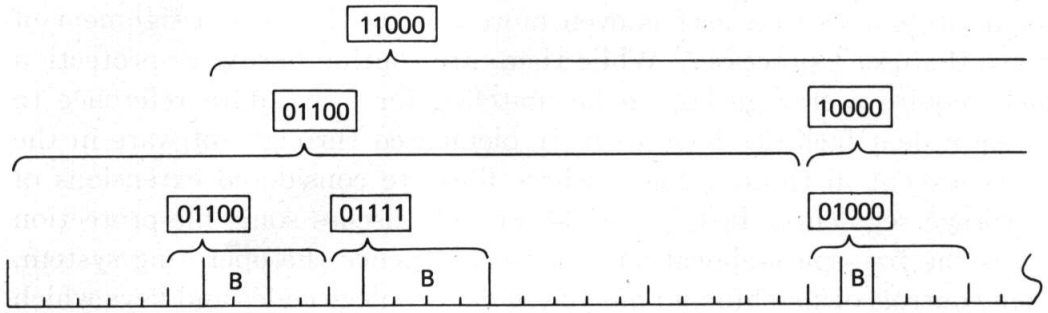


Figure 6-20 A packed bit array.

Summary Table 6-18 summarizes these three methods to manage free space. The basic methods can, of course, be improved at some cost in complexity, time, and core requirements.

Table 6-18 Evaluation of three free-space management techniques.

Criteria \ Methods	1. Contents directory	2. Chained free portions	3. Bit tables
Portion size	Large	Large or small	Small
Portion variability	Variable	Variable or fixed	Fixed
Allocation frequency	Low	Low to high	High
Time to allocate space	Medium	Long	Short
Free space control	Poor	Medium	High
Basic crash security	Medium	Medium	Low
Core space required	Moderate	Low	Moderate

An operating system which does not provide good management of storage space imposes a greater load on the file system. Some file systems allocate separately for files within a larger space assigned to a user (MUMPS), or allocate from a very large space allocated by the operating system for all files in its purview, as seen in VSAM in Chap. 4-3.

6-5-4 Reliability in Allocation

An important aspect of free-space management is file reliability. The assignment of permanent file resources to a user is even more critical than the assignment of core memory to the user's processes. While there are routine hardware-protection

features which require a storage key to be matched for every data reference to core storage, equivalent facilities have to be implemented through software in the file-system area, except in those systems where files are considered extensions of virtual core storage segments. In Chap. 11 we will discuss some file-protection structures. It is the free-space allocation routine and hence the operating system, however, which controls ownership and has to set the privilege codes and keys which enable file protection schemes to perform their work.

Because of the great risk inherent in the reassignment of space, computer systems have been designed which would never have to reuse an address. This is achieved by providing a large virtual address space ($\gg 10^{20}$). The MULTICS system achieves this goal for all data at a given point in time. In more commonly available hardware some of this protection can be achieved by assigning portions for reuse that have been free for the longest period. Then there is increased recovery capability when portions have been released to free space in error, either by the system due to a program error, or by the user due to a mistake. Unfortunately, many systems assign free space on a LIFO basis rather than according to the safer FIFO rule, since the "last in first out" method is much easier to implement when free-space portions are maintained in chains. A ringlike structure could be used to hold free portions of storage for FIFO allocation.

BACKGROUND AND REFERENCES

Basic References Of the many available statistics texts, Dixon[69], Freund[62], and Snedecor[67] provided the required background. Feller[71] presents a comprehensive foundation of probability theory. Moroney[56] is a very readable introduction, and Huff[54] warns of many pitfalls in the use of statistics.

Useful mathematical techniques can be found in Knuth[69,73S] and in Acton[70]. An excellent compendium for statistical and mathematical procedures is Press[86]; programs are available for its procedures.

The application of a variety of techniques to computer-system behavior analysis is demonstrated in Freiberger[72]. An example is given by Hill[87]. There is not much incentive to publish the routine application of statistical techniques to system design, and because of this void, it is difficult to judge how much these techniques are used and how powerful they have shown themselves to be (Ferrari[86]).

Distributions Agarwala[70] addresses the problems of recognizing distributions, and Bhattacharya[67] separates bimodal mixtures of normal distributions. Statistically based analysis of file systems has concentrated on direct access methods: Peterson[57], Johnson[61], and Buchholz[63]. Christodoulakis[84] is justifies optimism about the assumptions of uniformity made commonly.

Simulation Simulation as a tool has been used for planning a large variety of technical and social projects and is often associated with database efforts to provide the required input to the simulation model. A journal, *Simulation*, is devoted to this field. An early textbook in this area is Gordon[69], who is the designer of GPSS, a simulation program often used for computer-system modeling. Kiviat[69] describes SIMSCRIPT and some applications. A careful exposition of simulation is given by MacDougall[82]. Maryanski[83] is a general text which covers both discrete (GPSS) and continuous simulation (CSMP).

Simulation has been used in many performance assessments, other chapters cite results by Effelsberg[84], Salza[83], and Whang[84]. Computer systems have been simulated to evaluate interacting processes (Nielsen[67] and Skinner[69] are examples), but the relative importance of file operations has been low in these studies. Seaman[69] includes input-output processes applicable to files. Glinka[67] discusses a simulation of an information-retrieval design. Reiter in Kerr[75] presents the development of a simulator to handle a variety of files. Fishman[67] analyzes the problem of the validity of simulation results generated from random inputs. Özsu[85] simulates control of distributed files.

Measurements Measurements of access patterns are provided by Lum[71,73], Deutscher[75], and both Ousterhout[85] and Maloney[87] for UNIX. Cheng[69] describes the use of logging files to drive simulations. Buchholz[69] presents a synthetic file-updating program, and Sreenivasan[74] uses this program to derive a workload for testing. Measurement of file operations is described by Krinos[73], Huang[74], Sreenivasan[74], and Heacox in Kerr[75]. Salza[85] describes the workload for testing advanced hardware models.

Queue Scheduling Any computer managing more than one process at a time will need queue management as part of required operating system services. Methods to obtain effective scheduling of queues were initially tried in the teleprocessing environment, where message queuing was a familiar concept. Seaman[66] mentions scanning to improve disk access, and Weingarten[66] presents the case for sector queuing, the Eschenbach scheme, and in Weingarten[68] extends this method to disk. Denning[67] gives a simple analysis of practical queue scheduling policies. These and other early papers take into account track-switching delays, which are no longer important.

Coffman[69] gives results for queue optimization with SSTF. Burge[71] and Denning[72] resolve the problem of drum performance for random arrivals of requests which access sectors uniformly, using Markov processes. Jenq[87] describes experiments.

Frank[69] analyzed SCAN and N-step-SCAN for uniform requests and complex disk devices using discrete probabilities and simulation. Teorey[72] analyzes and simulates a number of scheduling algorithms, and proposes CSCAN as well as switching of scheduling policies. The assumptions made in these analyses have to be considered very carefully before the results can be used. Coffman[72] analyzes FSCAN; it is the technique used in the CII SIRIS operating system which supports the database system SOCRATE. Stone[73] compares SSTF with other scheduling disciplines for drums with variable-length blocks. Wilhelm[76] presents the case where FIFO outperforms SSTF on disk and Goldberg[79] presents a problem in paging. Fuller[75] summarizes queuing policies and their analyses; included are confirmations of results in earlier reports. More recently, Atwood[82] simulated actual hardware with great care and concluded that seek scheduling was ineffective with their assumptions.

Operations Research OR has emerged as an applied science mainly because of the problems faced by the industrial mobilization of World War II. Much of the mathematical background was already available. Queuing theory had been developed and applied to telephone systems by A. K. Erlang in 1909; a recent example of their use is by Kelton[85].

A basic and easily read textbook describing methods used in OR is Wagner[75]; Hillier[67] or Hertz[69] provide fundamentals. Feller[71] includes some examples of queuing in his treat-

ment of probability theory, and Kleinrock[75] covers the theory and practice of queuing systems.

Queuing analysis Hellerman[75] and Stone[75] consider queuing problems in computer systems; Graham[78] and Denning[78] provide surveys. Kobayashi[78] approaches performance issues formally. BrinchHansen[73] analyzes queuing in time-shared processing, but many of the assumptions do not hold for file systems. Allen[75] and IBM F20-7[71] summarize queuing theory for message-processing systems, and Martin[72] includes some approximations to queuing results. Newell[71] applies queuing to rush-hour transients, using a parabolic approximation; the sinewave-based approach presented here was original in Wiederhold[77,83] and appears simpler. Many of the analyses of scheduling disciplines in the previous section use results from queuing theory.

Linear Programming Morgan[74] describes replacement algorithms for files on mountable diskpacks and uses linear programming (LP) for some solutions. Benner[67] uses LP to optimize record allocation in an IMS-like hierarchical structure, and Stanfel[70] uses LP to optimize fetch time in trees. Chu[69] and Severance[72] apply LP to file design. A CODASYL database is optimized by De[78]. Ramamoorthy[79], Chen[80], and Ceri[82] use OR to design distributed databases. Levin[78] uses a branch-and-bound method. Pelegatti in Bernstein[79] and Ceri[82] minimize the communication costs.

Reorganization Analyses The use of techniques developed for inventory control to determine the time interval for file reorganization is due to Shneiderman[73]. Arora[71], Vold[73], Yao[76], Tuel[78], and Batory[82] address the same problem. Sockut[78] models concurrent reorganization and provides a survey (Sockut[79]). Young[74] and Lohman[77] use a similar technique to obtain the required interval for writing file backup copies.

EXERCISES

1 Redo Exercise 4-1000000000000000000000000000000000000000000000000000000000000000000000000000 given that the text lines are of length 70 with a standard deviation of 30 characters.

2 Equation 2-2 is based on uniform track access. Redo this result, given that the request rate for tracks has a Poisson distribution over the track numbers with a mean of 0.5.

3 a Sketch a distribution of the access times between blocks on tape.

b Calculate the standard deviation.

4 Do Exercise 3 for a disk or other direct-access unit.

5 What queue-scheduling policy is used by the file system that you have been using? Do you find the policy reasonable? Which users are being favored and which are being discriminated against? Prepare an argument to management for changing the policy.

6 In Example 6-4, used to demonstrate the normal distribution, space was saved by placing fewer records into a block. When is the technique apt to be effective in an indexed or in a hashed file?

7 Reevaluate the same example for the case that the distribution was not normal, but symmetric and unimodal.

8 Section 6-1-6 stresses understanding of the model when applying transformations. What was the model used for the billing data? Does it make sense? Using this model, what is the expectation of outstanding bills if the hospital doubles size and billings? What can cause failure of the model?

9 Specify the simulation steps for
" 29.4: `Record "Jones" is in core and disk D is free.`"

10 Read a scheduling algorithm other than FIFO used for processor scheduling and apply it to a file system. Compare the effectiveness and the validity of the analysis in the file environment.

11 During lunch hour a large number of orders for stock purchases are placed at a brokerage, creating a queue within the processor which updates the order file. Estimate the earliest time when the entire order file can be tabulated.

12 Write a program to simulate the queue management flow of Fig. 6-10. Include two selection algorithms and compare the processing of identical workloads. Explain the differences in behavior.

13 Describe how a random-number-generating routine generates a uniform distribution. Describe how a normal random-distribution routine generates values which are normally distributed, using a uniform-distribution-generating routine. The listings of the programs used can be found in the documentation of your computer service.

14 Using the Poisson distribution of Example 6-5 as an example, compute the mean and the mode, and use Eq. 6-1 to estimate the median from the mean and the mode. Apply these to some moderately skewed distributions.

15 Recalculate the expected update cost for an indexed file (Chap. 3-4-3), given that one attribute only is updated, and that it has been determined that data changes are distributed normally around the current data value, with a relatively small standard deviation, say 100 for $n = 10\,000$.

16^p We now will apply some distributions to the files of your application and their use. Consider the length of the records. Let the records be of variable length and assume the standard deviation of the length to be of at least 30% of the average record length R. If you use variable-length and unspanned blocking:

- a How many records fit into a block if consider only the average record length?
- b How many records fit into a block if consider only the record length to be normally distributed, and you want to exceed the block less than 1% of the time ($t = 2.32$)?
- c How many records fit into a block, again with less than 1% overflow, if you cannot predict the distribution, so that you have to use the Chebysheff assumption?

17^p Write a simulation where variable length records are placed into blocks, up to the estimated Bfr. Use the same parameters assigned for Exercise 16.

a Assume a uniform distribution with the same length and standard deviation parameters.
b Do the same with a normal distribution. (If you dont have a function to generate normally distributed random numbers take a sum of say, 16 uniform numbers, from a distribution having a quarter of the desired standard deviation, as shown in Eqs. 6-4 and -5.

Count the number of overflows. Compare your results with the prediction. Which distribution is better? Justify the result.

18^p Assume that your record lengths have an Erlang distribution. Compute the Erlang `m` parameter.

19^p Create a scenario where, for a transient period, more requests are coming into your system than it can handle during some period (i.e., $L_{period}T_T > t_{period}$). Assume that the excess load distribution in that period has a sinusoidal shape, as done in Sec. 6-4-2. Establish a start and and end time for the transient.
a Sketch the behavior of the transient.
b Compute the length of the transient.
c Compute the expected waiting time in the queue which will form during he transient. For this problem you may want to compute the standard deviation over the transient, and find a distribution which approximates the excess.
d As an alternative or verification of the result of Exercise 19-c determine the queue length by averaging the queue as computed in steps as shown in Table 6-16.
e Discuss if the distribution you selected to solve Exercise 19-c is an optimistic approximation to the shape of the transient.
f Compute the overall expected waiting time in the queue, considering that that the transient is only in effect some fraction of the day.

20^p Compute the optimal reorganization period for your files.
a Assume that you are using indexed-sequential files.
a Do the same for the file organization you have actually chosen.

State carefully all your assumptions on file usage. Chap. 5-1 gives some examples of usage estimation.