Chapter 9

Memory

9.1. Introduction

Memory and its management affect the performance of computer systems in two major ways. First, almost every system has a *memory constraint*: a limit on the number of "threads of control" that can be active simultaneously, imposed by the availability of memory. A memory constraint places an upper bound on the extent to which processing resources (CPUs, disks, etc.) can be utilized concurrently, and thus on the throughput of the system. Second, there is *overhead* associated with memory management. As an example, swapping a user between primary memory and secondary storage places service demands on the I/O subsystem (and the CPU, as well). To the extent that the operating system devotes processing resources to the management of memory, the progress of "useful" work is impeded.

Although memory seldom was mentioned explicitly in Parts I and II, specific implicit assumptions were made in each example:

- When we described the intensity of a workload by its population N (a *closed* model with a *batch* workload), we were assuming that the system had a memory constraint, that this constraint could be expressed in terms of a specific number of jobs (i.e., that all jobs required the same amount of memory), and that there was a sufficient backlog of work that the system was continuously operating at its maximum multiprogramming level.
- When we described the intensity of a workload by its population N and average think time Z (a *closed* model with a *terminal* workload), we were assuming that the system had a fixed number of interactive users, and that enough memory existed to accommodate as many of these users as might concurrently require it (i.e., that there was no memory constraint).

- When we described the intensity of a workload by its arrival rate λ (an *open* model with a *transaction* workload), we were again assuming that there was no memory constraint. The assumption in this case is in fact somewhat more extreme than in the case of a terminal workload, because there is no bound on the central subsystem population of a transaction workload.

In each case we either ignored overhead due to memory management or included an average value in the service demands of every customer.

These simple assumptions about system behavior are encountered frequently in modelling studies because they satisfy the conditions required for queueing network models to be *separable*, i.e., directly amenable to the efficient evaluation techniques described in Part II. The fact that these studies are successful indicates that the assumptions, if not strictly correct, are at least robust:

- In an actual computer system, the multiprogramming level of a batch workload may vary over time for many reasons: the amount of memory available to the batch workload may vary, or the memory requirements of individual batch jobs may differ, or the backlog of work may drop below the memory constraint. However, usually it is possible to validate a model using a single multiprogramming level that represents the time-weighted average of the observed multiprogramming levels. Projecting performance for a modified workload or configuration requires that the analyst estimate the effect of the modification on this average multiprogramming level.
- Although there are times in almost every interactive system when a user must wait for access to memory, these times may be so infrequent that the existence of the memory constraint can be ignored in constructing a model. A modification to the workload or configuration may affect the distribution of the number of users desiring memory, so the validity of the assumption must be checked in modelling such a modification. Doing so usually is not difficult.
- Although detailed paging behavior is difficult to model, many operating systems succeed in maintaining an average page transfer rate that is relatively insensitive to variations in configuration and workload. In such cases it is not difficult to characterize a customer's service demand at the paging device.

Of course, these simple assumptions are not always adequate. In this chapter we will extend the flexibility with which we represent memory and its management in queueing network models. The organization of the chapter reflects our belief that the throughput-limiting effect of a memory constraint is the *primary effect* of memory on performance, while the overhead associated with memory management is a significant *secondary effect*. The chapter has five principal sections. First, we explore

some of the subtleties that can arise in the simple case of a system with a known average multiprogramming level. Next, we show how to represent the effect on system throughput of a memory constraint that is sometimes, but not continuously, reached. Then, we describe how to represent overhead due to swapping (Section 9.4) and paging (Section 9.5). Finally, we use case studies to relate these techniques to one another, supplementing the examples presented in each section.

9.2. Systems with Known Average Multiprogramming Level

This section serves to illustrate that subtleties can arise even in modelling the apparently straightforward case of a batch workload with a known average multiprogramming level.

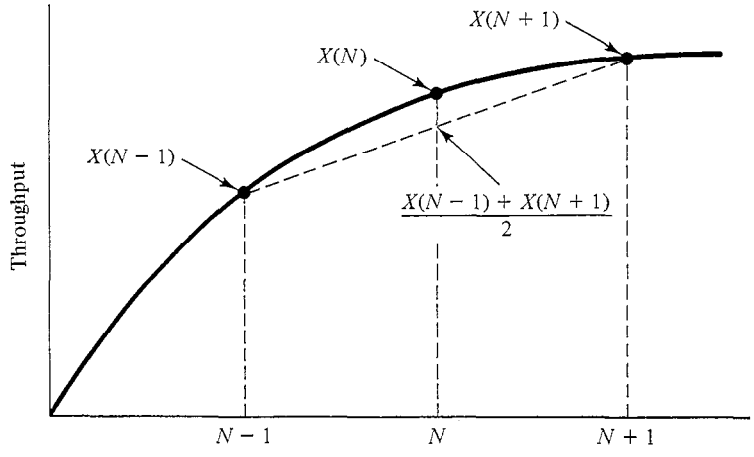

Figure 9.1 — Throughput Versus Multiprogramming Level

In all but the simplest of systems, the multiprogramming level of a workload (the number of active threads of control) is not constant, but varies over time due to factors such as competition for memory from other workloads, differences in the memory requirements of jobs, and the availability of jobs. As the multiprogramming level of a workload varies, so does its throughput. The relationship of throughput to multiprogramming level is illustrated qualitatively by the curve in Figure 9.1. At low multiprogramming levels, the marginal increase in throughput due to an additional active job is relatively large, since this job causes a relatively large increase in the concurrent activity of various processing resources.

As the multiprogramming level increases, the marginal increase in throughput becomes relatively small, because little additional concurrency is realized. (Figure 9.1 assumes that the overhead due to a job can be included as a component of its service demands, and is insensitive to multiprogramming level.)

Imagine that we observe such a workload for a period of time and measure its average multiprogramming level, N. For the sake of argument, let N be an integer. Now, consider two cases:

- If the system had operated at a constant multiprogramming level of N during the entire observation interval, then its throughput would have been $X(N)$, as indicated in the figure.
- If the system had operated at a constant multiprogramming level of $N-1$ during the first half of the interval and at a constant multiprogramming level of $N+1$ during the second half, then its throughput would have been $X(N-1)$ during the first half of the interval, $X(N+1)$ during the second half, and $\frac{X(N-1) + X(N+1)}{2}$ over all, which, as shown in the figure, is less than $X(N)$.

Clearly, if the system actually had operated as in the latter case but a queueing network model of the system is evaluated at the average multiprogramming level N, a discrepancy will result. This discrepancy often is small; systems almost inevitably are modelled successfully using an average multiprogramming level, which almost inevitably represents a time-weighted average of several different multiprogramming levels encountered during an observation interval. However, if greater accuracy is required, the model can be analyzed at each of the observed multiprogramming levels and a weighted average of the results taken. This approach can be applied to multiple class models as well as single class models. Naturally, though, the incentive to be satisfied with the results of an analysis at average workload intensities increases with the number of combinations that would have to be considered to do otherwise.

Here is an example based on actual data collected during a benchmark test of a system with three distinct workloads, each of batch type. As shown in Table 9.1, the multiprogramming levels of these workloads varied in a way that partitions the benchmark into three time periods. These periods are described by the first three lines of the table, which show the elapsed time (in seconds) at which the transitions between periods occurred, the duration of each period (again in seconds), and the proportion of the total observation interval due to each period.

In order to parameterize a queueing network model, we need not only the workload intensities, as shown in Table 9.1, but also the service demands. These service demands, calculated from measurements taken during the benchmark, are shown in Table 9.2.

quantity		period 1	period 2	period 3	average
time interval		0 - 1268	1268 - 1734	1734 - 2108	
duration		1268	466	374	
proportion of total		.602	.221	.177	
MPL	workload 1	2	2	3	2.18
	workload 2	1	0	0	0.60
	workload 3	2	3	0	1.87

Table 9.1 – Variation in Multiprogramming Level (MPL)

device	service demand, seconds/job		
	workload 1	workload 2	workload 3
CPU	12.906	1.315	0.632
disk 1	4.133	0.325	0.004
disk 2	8.580	0	0
disk 3	7.549	0.081	0.305
disk 4	0.424	0.001	0.181
disk 5	4.896	0.053	0.198
disk 6	6.437	0	0
disk 7	3.651	0	0
disk 8	0	0.082	0.888
disk 9	3.057	0.087	0.049
disk 10	4.980	0.141	0.080

Table 9.2 – Service Demands

First we consider a three class model of this system which we evaluate three times, using the three sets of multiprogramming levels corresponding to the three time periods of the benchmark. The results are shown in Table 9.3.

quantity		period 1	period 2	period 3	average
CPU utilization		.925	.782	.557	.825
throughput, jobs/minute	wkld. 1	1.343	1.475	2.498	1.58
	wkld. 2	14.71	0	0	8.86
	wkld. 3	29.71	44.14	0	27.6

Table 9.3 – Model Outputs for Three Time Periods

The alternative is to evaluate the same three class model once, using the average multiprogramming levels for each workload. Table 9.4 compares measurement data, the model using the average multiprogramming level, and the model representing the three time periods.

quantity		actual value	model results			
			average MPL		variable MPL	
			value	discrep.	value	discrep.
CPU utilization		.820	.819	0	.825	+ 1%
t'put., jobs/min.	wkld. 1	1.59	1.51	− 5%	1.58	− 1%
	wkld. 2	8.77	8.72	− 1%	8.86	+ 1%
	wkld. 3	27.0	28.9	+ 7%	27.6	+ 2%

Table 9.4 − Measurements Versus Two Modelling Approaches

Two summary comments, the first of which is technical, the second philosophical:

- As we have observed in other contexts (e.g., Chapter 4), average response time must be calculated in a different and less obvious way than average throughput, queue length, and utilization. These latter quantities are obtained by weighting the performance measure for each period by the relative length of that period. For example:

$$\bar{U} = \sum_{\substack{all \\ periods\ p}} (U \text{ during period } p) \times \frac{\text{duration of period } p}{\text{total duration of observation interval}}$$

  Average response time, on the other hand, is obtained by weighting the performance measure for each period by the relative number of jobs completed during that period:

$$\bar{R} = \sum_{\substack{all \\ periods\ p}} (R \text{ during } p) \times \frac{(X \text{ during } p) \times (\text{duration of } p)}{\sum_{\substack{all \\ periods\ p}} (X \text{ during } p) \times (\text{duration of } p)}$$

- We observe frequently in queueing network modelling that significant increases in effort (both in data collection and in analysis) yield only small increases in accuracy. This is perhaps the most important point illustrated by this example.

9.3. Memory Constraints

Since the throughput-limiting effect of a memory constraint is the primary effect of memory on performance, its accurate representation can be important. We have noted that separable queueing network models allow the direct representation of certain extreme cases, such as a memory con-

straint that is continuously reached (batch workloads) and a memory constraint that is never reached (terminal or transaction workloads). Unfortunately, the interesting general case of a memory constraint that is sometimes, but not continuously, reached, is an instance of *simultaneous resource possession*, which violates the conditions required for separability. Fortunately, rather elegant techniques exist for the indirect representation of such a memory constraint in separable models. These techniques are the subject of the present section.

Our approach is based on the concepts of flow equivalence and hierarchical modelling, as described in Chapter 8. As shown in boxes 1 and 2 of Figure 9.2, we initially are confronted with a queueing network model that is non-separable because of the existence of a memory queue. First, we decompose the model into two parts: the *central subsystem* plus the *memory queue* (box 2) and the *external environment* (box 1). Next, we define a load dependent service center (shown in box 3) that is flow equivalent to 2 from the point of view of the external environment. We do this using a separable subsystem model, which can be evaluated efficiently. Finally, we analyze a high-level model consisting of this FESC and the external environment (1 and 3 taken together). The joint analysis of 1 and 3, which again can be carried out efficiently, will yield nearly the same results as the joint analysis of 1 and 2, which cannot.

This hierarchical analysis coincides nicely with the users' view of the system. Referring again to Figure 9.2, each customer can be in one of two principal *states*: *thinking* (i.e., at the terminals; equivalently, within box 1) or *ready* (i.e., desiring to compute; equivalently, within box 2). The primary concern of a user is the average time spent in the ready state (box 2), which corresponds to average response time. It happens that, because of the memory constraint, ready customers can be in one of two sub-states: *waiting* (i.e., in the memory queue; equivalently, above the dashed line in box 2) or *active* (i.e., memory resident and competing for the processing resources of the central subsystem; equivalently, below the dashed line in box 2). This influences the completion rate of customers — the rate at which customers flow from box 2 back to box 1 — and thus average response time. The objective of our analytic approach is to define an FESC that characterizes this completion rate as a function of the customer population within box 2. This characterization will account for competition within the central subsystem (i.e., below the dashed line in box 2), and also for the effect of the memory constraint on the actual population of the central subsystem.

We first discuss single class memory constrained systems, and then extend our discussion to the multiple class case.

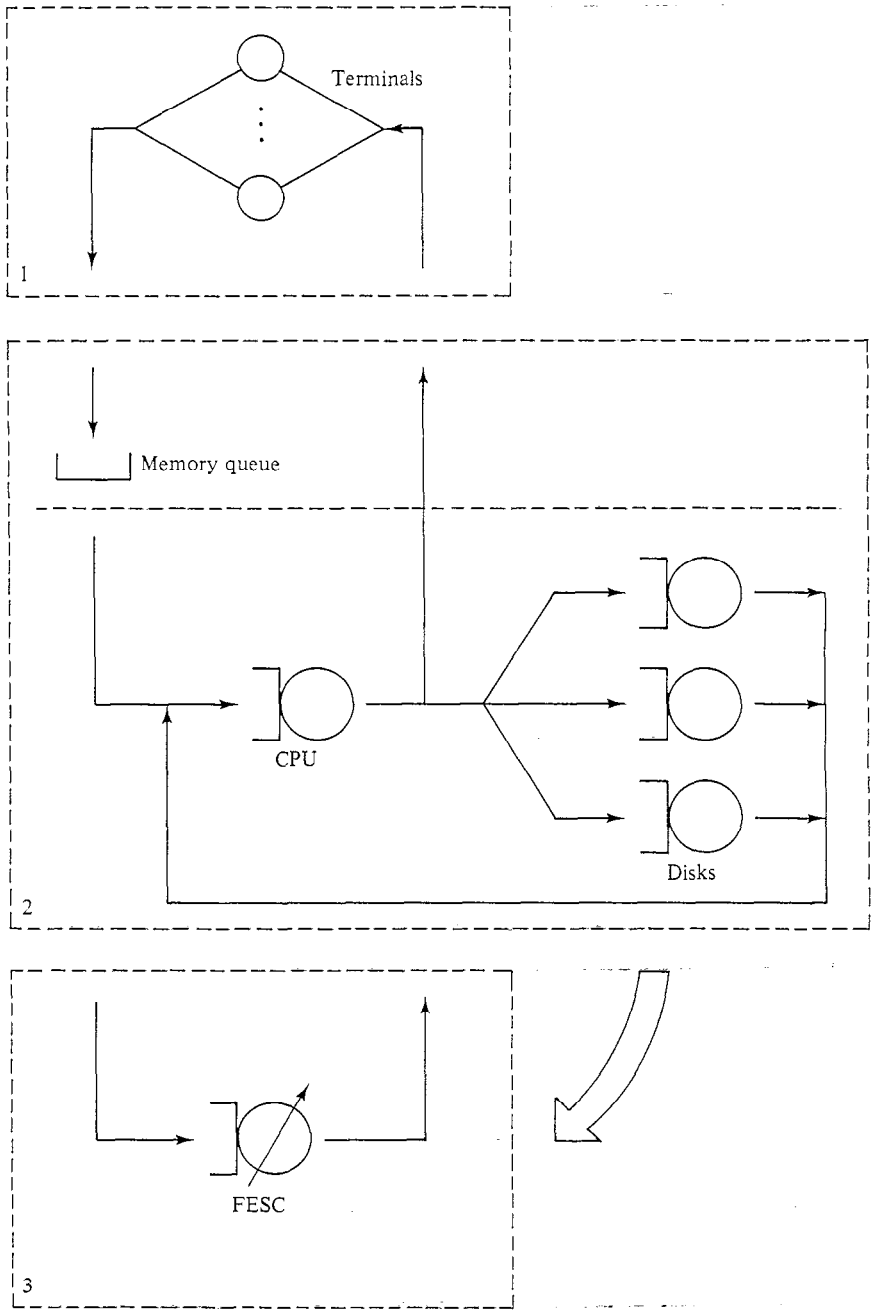

Figure 9.2 – Modelling a Memory Constrained System

9.3.1. The Single Class Case

We assume that customers have indistinguishable memory requirements, as well as service demands. We denote the memory constraint by M. If a customer becomes ready when there are fewer than M other ready customers (i.e., when there are $N-M$ or more thinking customers) then that customer becomes active immediately. If a customer becomes ready when there are M or more other ready customers (and thus M active customers fully occupying memory) then that customer must wait until memory becomes available.

Our task is to define an FESC for the central subsystem plus the memory queue. As noted in Chapter 8, a load dependent service center has a throughput that varies with its queue length. The queue length at the FESC in box 3 corresponds to the number of ready customers — the number of customers anywhere within box 2. In the actual system, how does throughput vary with the number of ready customers? The answer to this question is displayed qualitatively in Figure 9.3 both with the memory constraint (the solid curve) and without (the dashed curve). Once the memory constraint is reached (once there are M ready customers), no further increase in throughput results from an increase in the number of ready customers. Why is this the case? Because these additional ready customers are not active, but rather are waiting (for memory). This is made explicit by Table 9.5, in which $X(n)$ denotes the throughput of the central subsystem with a population of n customers.

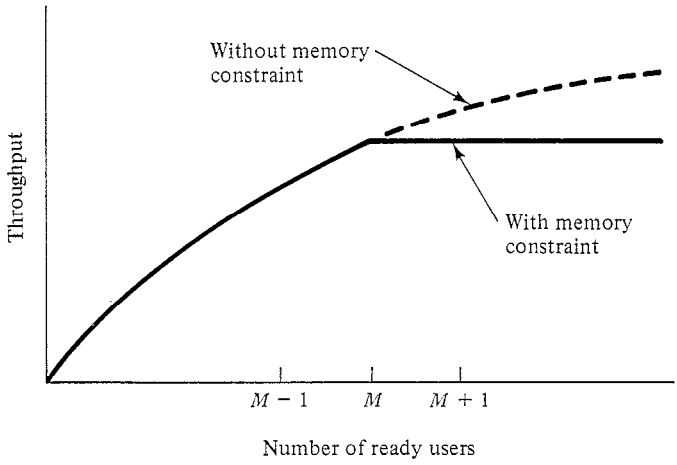

Figure 9.3 — Throughput Versus Number of Ready Customers

FESC queue length	ready customers	active customers	throughput
1	1	1	$X(1)$
2	2	2	$X(2)$
⋮	⋮	⋮	⋮
$M-1$	$M-1$	$M-1$	$X(M-1)$
M	M	M	$X(M)$
$M+1$	$M+1$	M	$X(M)$
⋮	⋮	⋮	⋮
N	N	M	$X(M)$

Table 9.5 – Throughput of a Memory Constrained System

It is a simple matter to determine $X(n)$. We define a *low-level model* consisting of the processing resources comprising the central subsystem. We evaluate this model for each feasible customer population n, i.e., for each number of active customers from 1 to M. For each population, we note the throughput. These throughputs are the $X(n)$ that are used to define the FESC used in the *high-level model.* This is stated more precisely in Algorithm 9.1.

1. Define a low-level model consisting of the service centers representing the processing resources that comprise the central subsystem.
2. Evaluate this model, which is separable, for each feasible population, $n = 1, \ldots, M$. Note the load dependent throughputs, $X(n)$.
3. Create a load dependent service center that is flow equivalent to the central subsystem plus the memory queue, by setting its throughput with queue length n, $\mu(n)$, to:

$$\mu(n) = \begin{cases} X(n) & n = 1, \ldots, M \\ X(M) & n > M \end{cases}$$

4. Define a high-level model consisting of this FESC and the external environment: if a terminal workload, then N customers with think time Z; if a transaction workload, then an external arrival rate λ. Evaluate this model, which is separable.

Algorithm 9.1 – Single Class Memory Constrained Systems

As an example application of this algorithm, consider a small timesharing system with a CPU, two disks, and 512K bytes of memory. An average interaction requires 3 seconds of CPU service, 4 seconds of service at one of the disks, 2 seconds of service at the other disk, and 100K bytes of memory. The operating system requires 150K bytes of memory, so that at most 3 users can be memory-resident simultaneously. There are 15 users, with average think times of 60 seconds. We wish to know:

- the average response time
- the average number of ready users
- the average number of active users
- the distribution of memory partition occupancy
- the average time spent queued awaiting access to memory
- the utilization of each processing resource
- the improvement in response time that would result if 256K of memory were added

We begin by analyzing the central subsystem for 1, 2, and 3 active users. This low-level model has three centers with service demands of 3, 4, and 2 seconds per interaction respectively. We obtain the load dependent throughputs shown below:

population	throughput, interactions/sec.
1	0.1111
2	0.1636
3	0.1930

Next we define a high-level model with $N = 15$ customers, $Z = 60$ seconds, and a load dependent center that is flow equivalent to the central subsystem plus the memory queue, defined as follows:

queue length	throughput
1	0.1111
2	0.1636
3	0.1930
4	0.1930
⋮	⋮
15	0.1930

We evaluate this model, obtaining the basic outputs shown in Table 9.6. Interactive response time is available directly: 25.7 seconds. So is the average number of ready customers: 4.5. From the queue length distribution at the FESC we see that 3.8% of the time the central subsystem is idle, 8.6% of the time there is a single active customer, 12.2% of the time there are two active customers, and 75.4% of the time there are three

throughput: 0.175 interactions/second
average residence time at the FESC: 25.7 seconds
average queue length at the FESC: 4.5
queue length distribution at the FESC:

queue length	probability
0	.038
1	.086
2	.122
3	.137
4	.142
5	.135
6	.117
>6	.228

Table 9.6 – Basic Outputs

active customers (i.e., 3 or more ready customers). Thus the average number of active customers is 2.6. Substituting this into Little's law, $N = XR$, we find that the average time spent in the central subsystem once a memory partition has been obtained is $2.6/0.175 = 14.9$ seconds. Thus a customer spends $25.7-14.9 = 10.8$ seconds awaiting access to memory. To calculate device utilizations we employ the utilization law, $U_k = XD_k$. At the CPU, utilization must be $0.175 \times 3.0 = 52.5\%$. At the two disks, utilization must be 70% and 35%, respectively.

To assess the impact of additional memory we calculate FESC rates for 4, 5, and 6 customers in the central subsystem. (Three additional users can be accommodated by the new configuration.) The FESC now will have the characteristics shown below:

queue length	throughput
1	0.1111
2	0.1636
3	0.1930
4	0.2110
5	0.2226
6	0.2305
7	0.2305
⋮	⋮
15	0.2305

When we analyze a high-level model consisting of 15 users and this FESC, we obtain a response time of 20.7 seconds, a 20% improvement.

The utility of the technique described in Algorithm 9.1 arises both from its accuracy and from its efficiency. Its accuracy is due to the fact that the terminals and the central subsystem are *decomposable*: the rate at which customers interact in the central subsystem is much greater than the rate at which they flow between the thinking and ready states. Its efficiency is due to two factors:

- The load dependent throughputs used in defining the FESC can be obtained efficiently. In this case, the model of the central subsystem is a single class separable queueing network.
- The resulting high-level model can be analyzed efficiently. In this case, it also is a single class separable queueing network.

This approach to analyzing single class memory constrained systems epitomizes the use of flow equivalence and hierarchical modelling to evaluate non-separable queueing networks efficiently.

9.3.2. Multiple Classes with Independent Memory Constraints

Here we consider a system with C customer classes, $c = 1, \ldots, C$, having independent memory constraints M_c. (The classes may be thought of as differing not only in their workload intensities and service demands, but also possibly in their memory requirements.) There is an obvious generalization of Algorithm 9.1 to this case:

- Define a multiple class low-level model consisting of the service centers representing the processing resources that comprise the central subsystem.
- Evaluate this model for each feasible population vector, $\vec{n} = (n_1, n_2, \ldots, n_C)$, $0 \leqslant n_c \leqslant M_c$. Note the "population vector dependent" throughputs of each class, $X_c(\vec{n})$.
- In a manner analogous to Algorithm 9.1, use these throughputs to define a multiple class FESC.
- Define a multiple class high-level model consisting of this FESC and the external environment of each class. Evaluate this model.

Unfortunately, this generalization possesses neither of the efficiency properties of its single class counterpart:

- Obtaining the throughputs needed to parameterize the FESC requires evaluating the low-level model for every feasible population vector. The cost of this is proportional to:

$$CK \prod_{c=1}^{C} (M_c + 1)$$

- The resulting high-level model is not separable, so can be evaluated only by the global balance technique, which is prohibitively expensive unless there are few classes and the memory constraints are small.

To circumvent these difficulties we introduce two *homogeneity assumptions*:

- We assume that the throughput of class c when its own central subsystem population is n_c depends only on the average central subsystem populations of the other classes.
- We assume that each class sees the other classes as though their central subsystem populations were independent of one another.

The former assumption allows us to determine the load dependent throughputs of any class by analyzing a C class queueing network in which the populations of the other classes are fixed at their *average values.* These average values are determined from the high-level model; the high- and low-level models are solved iteratively, terminating when successive estimates are sufficiently close. The latter assumption allows us to define a separate FESC for each class. In essence, we analyze C separable single class high-level models, rather than a single non-separable C class high-level model.

The result is Algorithm 9.2. This algorithm is applicable to models in which some of the C classes are unconstrained. For ease of expression, we denote the number of constrained classes by $\hat{C} \leqslant C$ and order the classes so that the constrained classes have indices $c = 1, \ldots, \hat{C}$. The algorithm is a good example of the introduction of homogeneity assumptions in order to facilitate evaluation.

9.3.3. Multiple Classes with Shared Memory Constraints

Algorithm 9.2 assumed that each class was subject to a memory constraint that was independent of the behavior of the other classes. Here we generalize that algorithm to shared memory constraints: constraints on the total number of customers in memory (or in a region of memory), rather than on the populations of the individual classes. The only significant change to Algorithm 9.2 will be in the calculation of the $\mu_c(n)$ in Step 3.2.

Let there be F *domains*, or shared regions of memory. Each memory constrained class is assigned to a domain. To simplify the discussion we will assume that all domains are shared; dedicated domains are, of course, a special case of shared domains. Let M_f be the capacity of domain f, i.e., the number of customers that can reside in that domain. (We temporarily assume that the classes assigned to a particular domain have indistinguishable memory requirements.)

1. Obtain initial estimates of the average central subsystem customer population for each memory constrained class, $\bar{n}_c$ for $c = 1, \ldots, \hat{C}$. To do so, ignore all memory constraints in the original C class model, yielding a separable queueing network. Evaluate this network. For each memory constrained class c, set $\bar{n}_c$ to the minimum of M_c and the average class c central subsystem population in the unconstrained model.
2. In preparation for the iteration, modify the original model by changing each of the $\hat{C}$ memory constrained classes into a batch class with population equal to $\bar{n}_c$. Leave the unconstrained classes in their original form. The result is a C class separable queueing network. (The non-integer customer populations of the constrained classes are naturally suited to the MVA-based iterative approximate solution technique.)
3. For each memory constrained class $c = 1, \ldots, \hat{C}$:
   3.1. Replace the $\bar{n}_c$ class c customers with each feasible population of class c, $n_c = 1, \ldots, M_c$. Evaluate the queueing network, obtaining the throughput of class c, $X_c(n_c)$.
   3.2. Create an FESC, a single class load dependent service center whose throughput with queue length n, $\mu_c(n)$, is defined by:

$$\mu_c(n) = \begin{cases} X_c(n) & n = 1, \ldots, M_c \\ X_c(M_c) & n > M_c \end{cases}$$

   3.3. Define and evaluate a single class separable high-level model consisting of this FESC and the external environment of class c (N and Z, or λ). Obtain the queue length distribution at the FESC. (We let $P[Q_{FESC} = i]$ denote the probability that the queue length at the FESC is i.) Use this to calculate a new estimate for the average central subsystem population of class c:

.. continued ..

Algorithm 9.2 – Multiple Classes, Independent Memory Constraints

.. continued ..

$$\bar{n}_c = \sum_{i=1}^{M_c} i\, P[Q_{FESC}=i] + \left[1 - \sum_{i=0}^{M_c} P[Q_{FESC}=i]\right] M_c$$

4. Repeat Step 3 until successive estimates of the $\bar{n}_c$ for each constrained class are sufficiently close.
5. Obtain performance measures for the constrained classes from the $\hat{C}$ high-level models evaluated during the final iteration. Obtain performance measures for the unconstrained classes by solving the queueing network defined in Step 2 using the final estimates of the $\bar{n}_c$ for the constrained classes.

Algorithm 9.2 – Multiple Classes, Independent Memory Constraints

Our approach is to view a domain shared by several classes as several smaller domains, each used by a single class. The memory constraint on a specific class will be determined iteratively, by considering the average central subsystem populations of its *competitor classes*: all other classes sharing the domain, in the case of FCFS domain scheduling; all other classes of greater or equal priority sharing the domain, in the case of priority domain scheduling. This approach is embodied in Algorithm 9.3, parts of which are abbreviated because of their similarity to Algorithm 9.2.

Algorithm 9.3 can be used to evaluate models in which the classes sharing a specific domain have distinct memory requirements. This requires straightforward modifications to the functions M_f and δ_c, defined in the algorithm. Once modified in this way, the algorithm can also be used to evaluate single class memory constrained models in which customers differ in their memory requirements. This is accomplished by defining a single domain shared by several "artificial" classes. Each of these artificial classes corresponds to those customers with a specific memory requirement. Each has service demands identical to those of the "real" class, and a workload intensity adjusted to reflect the proportion of customers having the corresponding memory requirement.

1. Obtain initial estimates of $\bar{n}_c$ for $c = 1, \ldots, \hat{C}$. To do so, ignore all memory constraints in the original C class model. Evaluate the resulting separable network. For each memory constrained class c, set $\bar{n}_c$ to the *minimum* of the average class c central subsystem population in the unconstrained model and a "proportionate share" of its domain, calculated as:

$$M_{F(c)} \times \frac{\alpha_c}{\sum_{i \in \substack{c \text{ plus its com-} \\ \text{petitor classes}}} \alpha_i}$$

where $F(c)$ is a function that gives the domain to which class c is assigned ($M_{F(c)}$ is thus the capacity of the domain to which class c is assigned), and α_i is the average class i central subsystem population in the unconstrained model.

2. In preparation for the iteration, modify the original model by changing each of the $\hat{C}$ memory constrained classes into a batch class with population equal to $\bar{n}_c$.

3. For each memory constrained class $c = 1, \ldots, \hat{C}$:

   3.1. Replace the $\bar{n}_c$ class c customers with each feasible population of class c, n_c. Evaluate the queueing network obtaining the throughput of class c, $X_c(n_c)$. Feasible populations are integers from 1 to $\lfloor M_{F(c)} - \delta_c \rfloor$, where:

$$\delta_c \equiv \sum_{i \in \substack{c\text{'s compe-} \\ \text{titor classes}}} \bar{n}_i$$

   Also evaluate the network at the non-integer population $M_{F(c)} - \delta_c$.

   3.2. Create an FESC, a single class load dependent service center whose throughput with queue length n, $\mu_c(n)$, is defined by:

$$\mu_c(n) = \begin{cases} X_c(n) & n \leqslant M_{F(c)} - \delta_c \\ X_c(M_{F(c)} - \delta_c) & n > M_{F(c)} - \delta_c \end{cases}$$

.. continued ..

Algorithm 9.3 – Multiple Classes, Shared Memory Constraints

.. continued ..

3.3. Define and evaluate a single class separable high-level model consisting of this FESC and the external environment of class c (N and Z, or λ). Obtain the queue length distribution at the FESC. Use this to calculate a new estimate for the average central subsystem population of class c:

$$\bar{n}_c = \sum_{i=1}^{\lfloor M_{F(c)} - \delta_c \rfloor} i\, P[Q_{FESC}=i] + \left[1 - \sum_{i=0}^{\lfloor M_{F(c)} - \delta_c \rfloor} P[Q_{FESC}=i]\right](M_{F(c)} - \delta_c)$$

4. Repeat Step 3 until successive estimates of the $\bar{n}_c$ for each constrained class are sufficiently close.
5. Obtain performance measures as in Algorithm 9.2.

Algorithm 9.3 – Multiple Classes, Shared Memory Constraints

9.4. Swapping

In Section 9.3 we developed techniques for representing the throughput-limiting effect of a memory constraint. While concentrating on this primary effect of memory on performance, we allowed ourselves to ignore the problem of explicitly representing swapping.

On the one hand, swapping devices are no different than other I/O devices: they can be included in a model, and their service demands can be calculated by multiplying device utilization by the length of the measurement interval, then dividing this result by the number of interactions during that interval. In this sense, swapping activity has been included implicitly in all of the models we have constructed. On the other hand, we presently have no way of projecting changes to this service demand that might result from system or workload modifications. Service demand at the swapping device is not an intrinsic property of an interaction, like service demand at the CPU or at a file device. The analyst typically knows how to modify intrinsic parameters to reflect system changes. On the other hand, the influence of system modifications on the level of

swapping activity is something we would like to learn from our model, rather than provide as an input. If the system modifications under consideration can be expected to influence significantly the level of swapping activity, then the modelling approach must include a procedure for estimating swapping device service demand.

The explicit representation of swapping is the subject of the present section. The techniques we develop will use the algorithms of Section 9.3 as a basis, since we wish to represent the effect of the memory constraint in addition to the overhead of memory management. For the sake of simplicity, the algorithms in this section will be expressed for the case of a single workload of terminal type (N customers with think time Z), and a single swapping device. Generalization to multiple workloads and multiple swapping devices is possible.

9.4.1. Swapping to a Dedicated Device

We first consider memory constrained systems with a single workload of terminal type, in which the swapping device is dedicated in the sense that activity there does not affect the throughput of the central subsystem. (The analytic simplicity resulting from this assumption will become apparent.) The basis of our approach is Algorithm 9.1. As shown in Figure 9.4, we modify the high-level model of that algorithm to include a center representing the swapping device, in addition to the FESC representing the central subsystem. The only new issue that we must confront is determining the service demand at the former center.

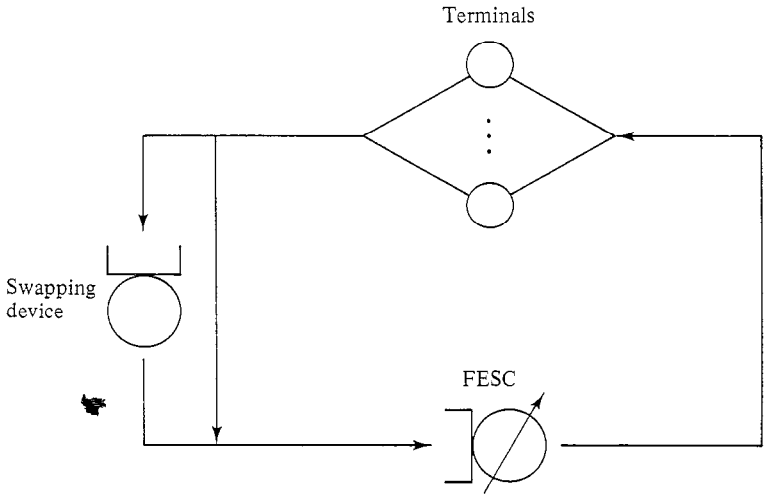

Figure 9.4 – High-Level Model for Swapping to a Dedicated Device

An interaction's service demand at the swapping device, D_{swap}, will be equal to the product of two terms: the probability that a swap precedes an interaction, $P[swap]$, and the service time for a swap in and subsequent swap out (both must occur), S_{swap}. S_{swap} is readily determined, but knowledge of the swapping policy of an operating system is necessary to estimate $P[swap]$. Here is an approach that can serve as a starting point. As in Algorithm 9.1, let there be N customers, M of whom can occupy memory simultaneously. We identify three cases:

- If $N \leqslant M$ then no swapping will occur. Thus $P[swap] = 0$.
- If $N > M$ then there will be some swapping. Let Q_{ready} be the average number of ready customers. If $Q_{ready} \geqslant M$ then a swap will precede every interaction. This is the case because we assume that only ready customers will be occupying memory, so a customer making a transition from the thinking state to the ready state will never be memory resident. Thus $P[swap] = 1$. (This clearly is an approximation, since we consider only the average number of ready customers.)
- If $N > M$ and $Q_{ready} < M$ then a swap will sometimes but not always precede an interaction. On the average there are $N - Q_{ready}$ thinking customers. Of these, $M - Q_{ready}$ are memory resident. So a customer leaving the thinking state requires a swap with probability:

$$P[swap] = 1 - \frac{M - Q_{ready}}{N - Q_{ready}} = \frac{N - M}{N - Q_{ready}}$$

The first of these three cases can be identified easily, since N and M are basic inputs. To distinguish between the second and third cases we need to know Q_{ready}, the average number of ready customers. This is an output of the model, not an input. Iteration is required, as described in Algorithm 9.4. (In the case that $N \leqslant M$, the swapping device can be ignored, and Algorithm 9.1 can be applied directly. For completeness, however, we include this case in Algorithm 9.4.)

From examination of the algorithm, our reliance on the assumption that the swapping device was dedicated should become evident. We constructed a flow equivalent representation of the central subsystem prior to iterating, and did not modify this representation subsequently. This requires that the load dependent throughputs of the central subsystem be independent of the level of swapping activity.

9.4.2. Swapping to a Shared Device

Especially in smaller systems, the swapping device also is apt to be used for other activities. To the extent that swap traffic impedes these activities (and vice versa), the analysis performed in the previous subsection will be invalid. Here, we will represent in our model this contention

1. As in Algorithm 9.1, define a load dependent server that is flow equivalent to the central subsystem.
2. Define a high-level model consisting of the workload (N users with think time Z), the FESC from Step 1, and a center representing the swapping device. Initially, set the service demand at this last center, D_{swap}, to zero.
3. Evaluate this model. Obtain Q_{ready}, the average number of ready customers. This is equivalent to Q_{FESC}, the average queue length at the FESC. Use Q_{ready} to calculate a revised estimate for a customer's service demand at the swapping device, as follows:

$$D_{swap} = S_{swap} \times P[swap]$$

where:

$$P[swap] = \begin{cases} 0 & N \leqslant M \\ 1 & N > M \text{ and } Q_{ready} \geqslant M \\ \dfrac{N-M}{N-Q_{ready}} & N > M \text{ and } Q_{ready} < M \end{cases}$$

4. Based on the discrepancy between the current and previous estimates for D_{swap}, decide whether to repeat Step 3 or to terminate.

Algorithm 9.4 – Swapping to a Dedicated Device

due to swapping. As before, an iterative analysis will be required. We will broaden the scope of the iteration to include the calculation of the load dependent throughputs, which now will vary with our estimate of swapping activity.

In generalizing Algorithm 9.4 a conceptual problem arises: Should the service center representing the swapping device appear in the high-level model (where swapping logically occurs) or in the low-level model (because by assumption this device also is used for file activity, which logically belongs in the low-level model). Fortunately this problem is not of practical concern, because only slight differences in results will occur. We choose to return to the high-level model used in Algorithm 9.1, and to represent all activity at the swapping device, both swapping activity and file activity, in the low-level model.

The low-level model, then, will consist of as many centers as there are processing resources. The service demand at most of these centers will be an intrinsic property of the workload, determined from measurement data. At the center representing the swapping device, however, the service demand will have two components: one due to file activity, determined from measurement data, and one due to swapping activity, determined iteratively as in Algorithm 9.4. The analysis is conducted as stated in Algorithm 9.5.

1. Define a low-level model consisting of the service centers representing the processing resources that comprise the central subsystem. At the center representing the swapping device, the service demand will have two components: one due to file activity, determined from measurement data, and one due to swapping activity, determined iteratively. Initially, assume that this latter component is equal to zero.
2. As in Algorithm 9.1, evaluate this low-level model for each feasible population, create an FESC, and define and evaluate a high-level model.
3. As in Algorithm 9.4, use the value of Q_{ready} obtained from the high-level model to calculate a revised estimate for the swapping activity component of the service demand at the swapping device. Based on the discrepancy between this estimate and the previous one, decide whether to repeat Steps 2 and 3 or to terminate.

Algorithm 9.5 – Swapping to a Shared Device

As an example, we return to the simple system considered in Section 9.3.1. Assume that the disk with an intrinsic service demand of 4 seconds also is used for swapping, and that the service time for a one-way swap of a 100K program is 150 msec.

On the first iteration we assume that no swapping occurs, so we evaluate the same low-level model used in Section 9.3.1, obtaining the same load dependent throughputs. We then construct and evaluate the same high-level model used in Section 9.3.1, obtaining the same value for the average number of ready users, 4.5. Now, we iterate. Since $Q_{ready} \geq M$ (the memory capacity was three customers in the example), we assume that a swap precedes each interaction. The service demand at the swapping device is equal to the sum of the intrinsic service demand there (4.0 seconds) and the service demand due to swapping. This latter service demand equals the product of the one-way swap service time (0.15

seconds), the probability that a swap precedes an interaction (1), and 2 (to account for the outswap that also must occur): 0.3 seconds. Total service demand at the swapping device is thus 4.3 seconds. We once again evaluate the low-level model for populations from 1 to 3, obtaining load dependent throughputs of 0.1075, 0.1577, and 0.1851, respectively. Using these rates to define a flow equivalent server, we again evaluate the high-level model, obtaining:

throughput: 0.170 interactions/second
average interactive response time: 28.0 seconds
average number of ready users: 4.8

Since our revised estimate for Q_{ready} still is greater than the capacity of memory, we still estimate that a swap precedes every interaction, and further iteration is unnecessary. As we would expect, throughput and response time are slightly worse than in Section 9.3.1, where swapping activity was ignored.

9.5. Paging

Most computer programs exhibit *locality of reference*: although a program may have a large *address space*, only a small portion of that address space will be referenced during any short time interval. *Virtual memory systems* exploit this property by allocating to each program an amount of (physical) primary memory that is smaller than the program's (virtual) address space, then using a combination of hardware and software to translate virtual addresses into physical addresses and to transfer portions of the virtual address space between primary memory and disk.

There are two principal advantages to virtual memory: the system can accommodate programs whose virtual address spaces are larger than the amount of physical memory that is attached to the CPU, and the number of concurrently active programs can be larger than would otherwise be possible. There is also a disadvantage: CPU and I/O resources must be devoted to the management of the virtual memory.

Virtual memory systems may employ *paging*, or *segmentation*, or both. Our focus in this section will be on paging. We consider the system's physical memory to be divided into some number of fixed-size *page frames*, and the address space of each program to be divided into some number of *pages* of the same fixed size. The operating system must make decisions on both a system level (How many programs should be allowed to compete for memory resources? How many page frames should be allocated to each of these programs?) and on a program level (Which pages should occupy the page frames allocated to a program? Alternatively, which page should be removed from primary memory in order to

accommodate a non-resident page that has just been referenced?) The I/O associated with moving pages between primary memory and disk in response to *page faults* is the aspect of system behavior whose modelling we will study in this section.

Modelling paging has much in common with modelling swapping. The fundamental issue is to determine the contribution of memory management activity to service demands. If it is not anticipated that the system modifications under consideration will have a significant effect on service demands at the paging devices, then these service demands can be taken from measurement data. As with swapping, though, the influence of system modifications on the level of paging activity is something we would like to learn from our model, rather than provide as an input. Paging activity is especially difficult to forecast because it is highly dependent on the characteristics of individual programs and on their interactions with each other through the memory management policies of the operating system.

Consider a simple example: a small multiprogrammed virtual memory system supporting a batch workload. Processing resources include a CPU at which jobs require an average of 3 seconds of service, two file disks at which jobs require an average of 8 and 2 seconds of service, respectively, and a paging disk.

Service demand at the paging disk is determined by considering in more detail the configuration of the system, the policies of the operating system, and the characteristics of the jobs. The system has 512 page frames of physical memory, 300 of which are available to user jobs. The operating system allocates memory on an *equipartition* basis: a multiprogramming level is selected and the available page frames are divided equally among the jobs. The memory reference characteristics of jobs and the page replacement policy of the operating system interact with one another in a manner that is reflected by the *program lifetime function*, shown in Figure 9.5. This function shows, for a single job, the average number of milliseconds of CPU service that elapse between page faults for various numbers of allocated page frames.

Suppose we are asked to model the performance of this system at multiprogramming levels of 2 through 8. A separate analysis must be conducted for each multiprogramming level. Each analysis must begin by determining the service demand at the paging disk. Consider a multiprogramming level of 5. Because 300 page frames are available for users, the equipartition policy will allocate $300/5 = 60$ page frames to each of the 5 jobs. The lifetime function tells us that at this memory allocation a job will experience an average of one page fault every 9 milliseconds of CPU processing. Since the average CPU service requirement of a job is 3 seconds, a job, on the average, will experience $3000/9 = 333$ page faults.

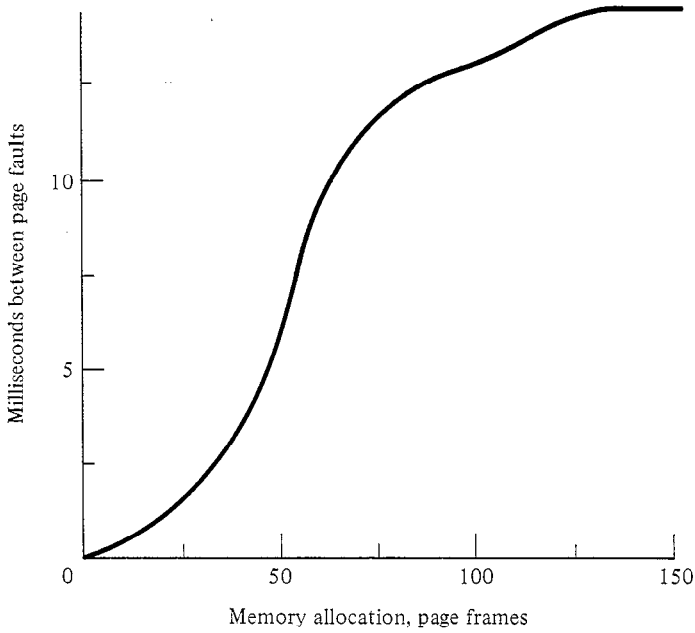

Figure 9.5 – Program Lifetime Function

Suppose we know that an average of 12.5 milliseconds of paging disk service is required to process a single page fault. Then on the average each job will place a service demand of $333 \times .0125 = 4.16$ seconds on the paging disk. The resulting queueing network model will have a population of 5 customers, and four service centers with service demands of 3, 8, 2, and 4.16 seconds.

Figures 9.6, 9.7, and 9.8 show respectively system throughput in jobs/minute, average job response time in seconds, and device utilizations, each as a function of multiprogramming level.

This example illustrates the techniques used to analyze paging systems. The difficulties that arise in such studies are related to the availability of data from which to parameterize the model. The example was very much simplified in this respect. For instance:

- It is extremely difficult to acquire paging lifetime data for a program. Doing so requires detailed tracing of the execution of the program in the context of the page replacement policy used by the operating system.
- The paging characteristics of a program are likely to vary as the program passes through different phases of execution, with each phase requiring a different lifetime function.

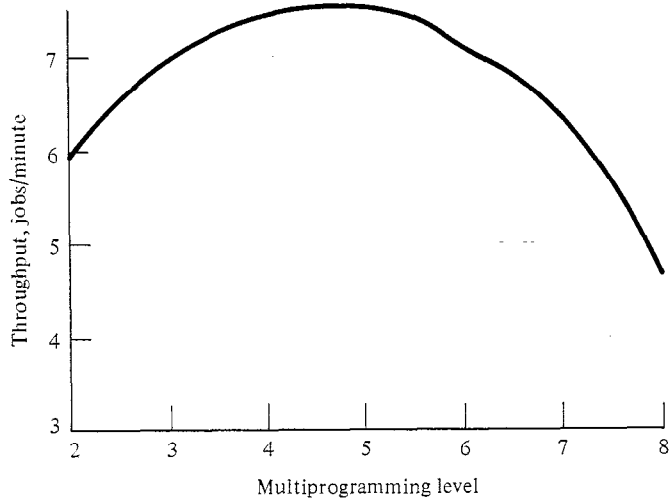

Figure 9.6 – Throughput Versus Multiprogramming Level

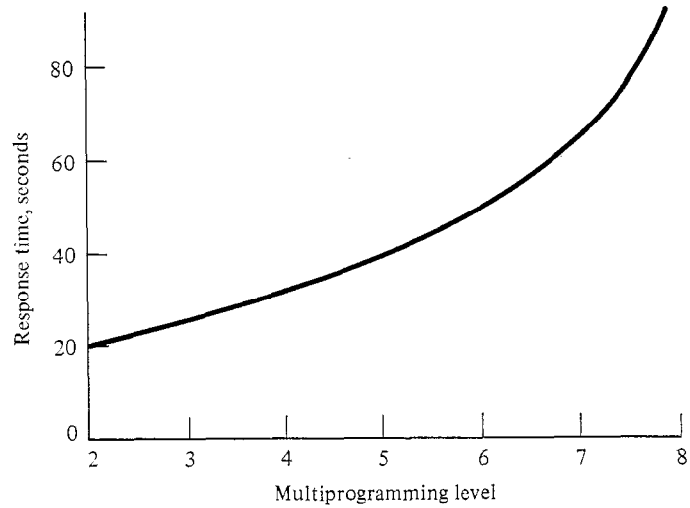

Figure 9.7 – Response Time Versus Multiprogramming Level

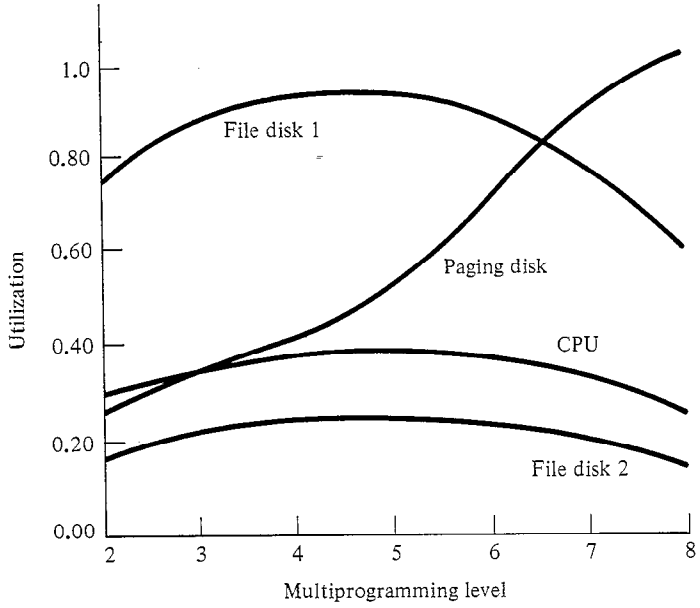

Figure 9.8 – Device Utilizations Versus Multiprogramming Level

- The paging characteristics of different programs will be dissimilar.
- Since different programs exhibit different paging characteristics, operating systems typically do not employ an equipartition strategy. At the very least, a different number of physical page frames will be allocated to each program.
- More likely, the operating system will change the number of physical page frames allocated to a program over the life of that program. Thus the number of programs that can be accommodated in memory simultaneously will vary with time.
- As the number of jobs that can be accommodated in memory varies, preemptive swapping may be employed. The swapping policy in a virtual memory system may be quite complex.

In practice, analysts using queueing network models to study virtual memory systems ignore many of these subtleties by making homogeneity assumptions similar to those we have encountered in other contexts. For example, it is common to consider only the average number of page frames allocated to a program, to assume that this average is the same for all programs belonging to the same class, and to assume that this average is largely independent of the load on the system. Studies incorporating such homogeneity assumptions generally are successful even in projecting the effect of modifications to the memory subsystem, e.g., the addition of memory. In the next section we will consider two such studies.

9.6. Case Studies

In this section we consider two successful case studies in which queueing network models were used to explore the effects of modifications to the memory subsystems of virtual memory systems. In the first study, a very simple model was used to evaluate the effects of increased paging device speed and of additional memory on the performance of an early IBM virtual memory system. In the second study, a more sophisticated model was used to evaluate workload and configuration changes to a Digital Equipment Corporation VAX/VMS system.

9.6.1. A Simple Model of an Early IBM Virtual Memory System

This study is from the early days of computer system analysis using queueing network models. At the time it was conducted, techniques for efficiently evaluating separable queueing networks (Chapters 6 and 7) and for representing memory subsystems using flow equivalence and hierarchical modelling (Chapters 8 and 9) were not widely known. This stimulated a number of clever "short cuts". The study serves to illustrate that useful results can be obtained for complex systems even in the presence of rather extreme simplifications. The system under consideration had the following characteristics:

- a small number of interactive users
- a CPU-intensive workload
- a large number of disks
- a low ratio of think time to response time (i.e., slow response)
- a paging virtual memory system
- a multiprogramming level limited to three to avoid thrashing

Figure 9.9 shows the model that was used in the study. It has one customer class. Each customer cycles through periods of thinking, (possibly) queueing for memory, and alternating bursts of CPU and I/O service. Because the multiprogramming level was limited to three and there were many possible paths to the I/O devices, little or no I/O queueing took place. This allowed the model to be simplified by representing the I/O subsystem as a single delay center. (The authors of the study probably evaluated the model by hand. Representing the large number of disks by a single delay center saved much tedious computation. Given a queueing network analysis package, it would be equally easy to represent all disks explicitly. This would be a "safer" procedure, since it would not rely on the assumption that no I/O queueing takes place.)

Because of the memory queue, the model is not separable. Even without the FESC approach described earlier in this chapter, though, it is possible to obtain accurate results in two extreme cases. The first is that

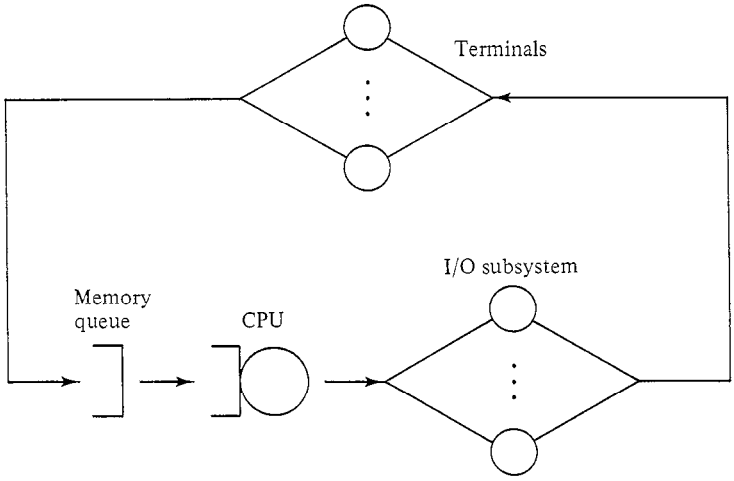

Figure 9.9 – The Model

memory utilization is low, so that little or no memory queueing takes place. This would occur, for instance, if response times were so short that most users spent the majority of their time thinking. Thus, the number of users in memory simultaneously would be small, and the chance that a user ever would need to queue for memory would be negligible. In this extreme case the memory queue could be ignored entirely, yielding a separable model.

The other extreme is that memory is utilized nearly 100%, so that the multiprogramming level of the system remains constant at its maximum. This was in fact the case in the system under consideration. This analytically fortunate situation allowed the model to be evaluated as follows:

- From the full model of Figure 9.9, extract the central subsystem (the queueing center representing the CPU and the delay center representing the I/O subsystem).
- Evaluate this central subsystem model with appropriate service demands and with a fixed population equal to the maximum multiprogramming level (in this case, three). Obtain throughput, X.
- Apply the response time law (N and Z must be provided).

For the system under consideration, evaluation of the central subsystem model gave a throughput of .395 interactions/second. From measurements, the number of interactive users was 10 and their average think time was 4 seconds. Applying the response time law:

$$R = \frac{N}{X} - Z = \frac{10}{.395} - 4 = 21.3 \text{ seconds}$$

The measured response time was 21.0 seconds.

Two changes to the configuration were being considered in an attempt to reduce the effect of the severe memory contention being experienced: upgrading the paging disks to drums, and adding memory. The upgrade to drums can be reflected in the model by adjusting the service demand at the delay center representing the I/O subsystem. The part of this service demand due to paging activity must be reduced to account for the elimination of the seek portion of data access (the drums have fixed heads) and for a decrease in the latency and data transfer portions (the drums have higher rotation speed than the disks). These adjustments can be estimated rather easily. Once a new service demand has been calculated, the evaluation can be carried out as before.

Representing the addition of memory is somewhat more challenging, since this modification affects paging activity (and thus the service demand at the delay center) in a manner that is not easily estimated. The addition of memory was studied for two cases: using the additional memory to increase the maximum multiprogramming level while maintaining the current number of page frames allocated to each active user, and using the additional memory to increase the number of page frames allocated to each active user while maintaining the current maximum multiprogramming level. To model the first case, it was determined that the additional memory would allow two more users to be active while maintaining the current memory allocation per user. Since the memory allocation per user would remain fixed, it was postulated that the page fault count of each user would be unaffected by the increase in the multiprogramming level. The memory addition was therefore modelled by increasing the number of customers in the central subsystem model from three to five and evaluating as before.

The other case, increasing the memory allocation to the three active users, can be expected to reduce the number of page faults per user. The service demand at the delay center in the model must be adjusted to reflect this. To estimate each user's service demand due to paging in the new environment, an experiment was conducted in which the maximum multiprogramming level of the existing system was reduced to two. (It had been determined that the number of page frames available to each of two active users on the existing configuration would be roughly the same as the number of page frames available to each of three active users on the proposed configuration.) I/O subsystem service demand was calculated from measurements during this experiment. The memory addition was modelled by using this value and a customer population of three, evaluating as before.

It is important to note a limitation arising from the fact that the evaluation technique assumes the central subsystem runs continuously at the maximum multiprogramming level. If response times improve significantly, this assumption may no longer be valid. Should this occur, the model may yield optimistic results. For any particular set of parameter values, the validity of the assumption can be checked by computing the average number of customers competing for memory (the average number of ready customers). If there are, on average, at least as many ready customers as can be accommodated in memory, the results of the model can be expected to be accurate. The average number of ready customers can be computed by applying Little's law to the central subsystem plus the memory queue. For the model of the original system:

$$N_{ready} = XR = .395 \times 21.3 = 8.4 \text{ customers}$$

The previous paragraph points out that proposed system modifications can have side effects that invalidate assumptions made by the particular evaluation technique in use. It is also possible for modifications to have side effects that invalidate measurements used to calculate model inputs. In the system described here, the user think time was measured as 4 seconds. This low value probably was due in part to the poor response time of the system: while one request was processing, users had time to prepare their next. If a system modification resulted in significantly improved response times, the think time would likely increase because of a reduction in this overlap.

Much of the success of a modelling study depends on the analyst's ability to anticipate significant side effects.

9.6.2. A Model of VAX/VMS

This section presents a queueing network model of Digital Equipment Corporation's VAX/VMS system. Memory management in VMS includes swapping, paging, and a shared cache of page frames. The questions addressed by this modelling study relate to workload and configuration changes that can be expected to affect paging and swapping behavior. The configuration is a small one, making homogeneity assumptions risky. For these reasons, the example serves to integrate a number of the techniques presented in this chapter, and we will examine it in considerable detail. The model is of an early release of VMS and does not reflect certain major changes in the system that have occurred since that time. The study predates the development of the algorithms for evaluating multiple class memory constrained queueing networks described in Section 9.3, so an alternative technique was employed.

9.6.2.1. Essentials of the System

As noted, memory management in VMS is accomplished through a combination of swapping, paging, and a shared cache of page frames.

A physical memory requirement, the *resident set size*, is associated with each process. An active process is guaranteed a number of page frames equal to its resident set size. Should a page fault occur in a process already using its entire allocation of page frames, a FIFO page replacement policy is used to select a page for removal from the resident set.

Since VMS makes no attempt to adjust processes' resident set sizes in response to observed behavior, an efficient allocation of page frames among active processes is unlikely. Since FIFO is a notoriously bad page replacement policy, an efficient choice of resident set membership is equally unlikely. To compensate for these shortcomings, VMS maintains a cache of page frames that is shared among the active processes. When a page is removed from a process' resident set it is added to this *shared page cache.* A fault on a page held in the cache can be resolved without disk I/O. Therefore we must distinguish between a page fault, which may not result in I/O, and a *paging transfer*, in which a page is retrieved from disk in response to a page fault. (Actually, pages are *clustered* for efficiency, and several pages are transferred in a single paging transfer.) The maximum and minimum sizes of the shared page cache are regulated by system parameters. If the cache exceeds its maximum size, pages are purged FIFO until the cache reaches its minimum size. Thus, as shown in Figure 9.10, physical memory can be divided logically into four parts: page frames permanently allocated to VMS, page frames containing processes' resident sets, page frames belonging to the shared page cache, and unallocated page frames.

Before a process that is swapped out can become active, it must be allocated sufficient page frames to accommodate its resident set. If enough unallocated page frames are not available, some other process must first be swapped out. Typically this process would correspond to an interactive user in the think state. The swapping rate at saturation is regulated by the *quantum*: a ready process is not eligible to be swapped out until it has acquired one quantum of CPU service.

One final detail. In point of fact, unallocated page frames are added to the shared page cache: the cache is allowed to grow until it reaches a size equal to the larger of its maximum size parameter and the number of page frames left over after VMS and the memory-resident processes have taken their toll. Cache pages that have been modified are written to disk when the maximum size parameter is reached, but the images of these pages are allowed to remain in memory and, if accessed, can be made available without disk I/O. The concept of an unallocated page frame principally is of use in understanding the swapping policy.

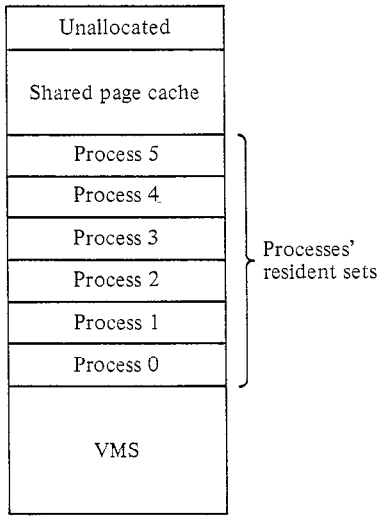

Figure 9.10 – A Logical View of Memory in VAX/VMS

9.6.2.2. The Queueing Network Model

The configuration under study is a small one: 512K bytes of memory and a single disk used for swapping, paging, and file activity. The workload is a benchmark consisting of one batch job (repeated compilation of a 10,000 line program) and 7 simulated interactive users each performing a specific task (compilation, execution, editing, trivial commands). The study involves validating a model of the base system, using this model to project the effect of specific modifications to the workload (eliminating the interactive users and running the batch job in isolation) and to the configuration (doubling the amount of physical memory), and finally making these modifications and comparing the results with the projections of the model. Four aspects of the system are of special interest in the context of the current chapter:

- There is a memory constraint.
- The proposed system modifications can be expected to affect the paging behavior of the system, which therefore must be modelled explicitly.
- The proposed system modifications also can be expected to affect the level of swapping activity, so this also must be modelled explicitly.

- The single disk means that swapping activity can be expected to interfere with the throughput of paging and file I/O.

The basis of the analysis is the familiar two-level hierarchical model: a low-level model that is evaluated at each feasible population in order to define an FESC for use in a high-level model. The low-level and high-level models are described in the following paragraphs.

The Low-Level Model

The low-level model has two service centers, representing the CPU and the disk, and two customer classes, representing the batch job and the interactive users. In the actual system there was a single batch jobstream that was locked into memory to reduce swapping activity, so in the low-level model the batch class has a constant multiprogramming level of one. In the actual system the seven interactive users had various resident set sizes, but the differences were small and on the average six interactive users could be accommodated in addition to the batch job. So in the low-level model there will be from zero to six customers in this class.

For each class, measurement data yields CPU service demand and the file activity component of disk service demand. Since we wish to explore system modifications that will affect paging and swapping behavior, we must develop techniques to estimate for each class the components of disk service demand due to these activities.

First, consider paging activity. Recall that each VMS process has a fixed allocation of page frames when it is memory resident. Because of this, the number of page faults sustained by a process will be insensitive to system load. However, the proportion of those page faults that result in disk I/O will vary with load, since this proportion is related to the number of page images in the shared page cache belonging to the process in question, which in turn depends on the number of processes actively using the cache. Thus the key to representing paging activity is estimating the effectiveness of the shared page cache.

We can measure the average number of page faults per interaction and we can calculate the average disk service time per paging transfer. We expect both of these quantities to be insensitive to the proposed modifications. The effectiveness of the shared page cache is reflected in the ratio of page faults to paging transfers. We can calculate this ratio for the benchmark measurement interval. In order to project performance under system modifications, we make the assumption that this ratio is linearly related to the average number of cache page frames available to each process actively using the cache. As an example, during the benchmark an average interaction caused 158 page faults and the ratio of page faults to paging transfers was 4:1. Thus an average interaction caused

158/4 = 39.5 paging transfers. The average number of active processes was eight: six interactive users, the batch job, and VMS (portions of which are pageable). Our assumption makes it possible to estimate that if the average number of active processes were three, the ratio of page faults to paging transfers would be $4 \times 8/3 = 10.7$, and an interaction would cause $158/10.7 = 14.8$ paging transfers. Our assumption also allows us to estimate that if the size of the shared page cache were doubled by the addition of memory (with three active processes), the ratio of page faults to paging transfers would become $10.7 \times 2 = 21.4$, and an interaction would cause $158/21.4 = 7.4$ paging transfers. Multiplying the average number of paging transfers per interaction by the average disk service time per paging transfer yields the paging activity component of disk service demand.

Next, consider swapping activity. The approach presented in Algorithm 9.5 is suitable except in the case that the average number of ready users exceeds the memory constraint. In this case, VMS will swap once per interaction plus once per quantum. The number of swaps per interaction due to the latter can be approximated by dividing the CPU service requirement per interaction by the quantum length.

The High-Level Model

We begin the analysis of the system by establishing initial values for the average numbers of ready and active interactive customers. These values allow us to estimate disk service demand due to paging (the average number of active customers is used for this) and swapping (the average number of ready customers is used for this). Given disk service demand, we can evaluate the low-level model. We do so for each feasible interactive population (the batch population is always one), obtaining load dependent throughputs which we use to construct an FESC.

The high-level model consists of this FESC and the workload (N customers with think time Z). Evaluation of this model yields revised estimates for the average numbers of ready and active interactive customers. If these revised estimates differ substantially from those used in the previous evaluation of the low-level model, we iterate using the new values.

Interactive response time and throughput, and thus the contribution of interactive users to device utilizations, can be determined directly from the high-level model. Batch throughput is calculated by taking the sum of the batch throughput at each interactive population (obtained from the low-level analysis) weighted by the proportion of time each of those interactive populations is encountered (obtained from the high-level analysis). Average batch response time and the batch contribution to device utilizations then can be determined by application of Little's law.

9.6.2.3. Use of the Model

In this section we illustrate the use of the model in some detail. Table 9.7 displays certain measured characteristics of the benchmark.

average interaction:
0.74 CPU seconds
158 page faults
12.4 file I/O operations
batch job:
330 CPU seconds
101386 page faults
918 file I/O operations

Table 9.7 – Measured Characteristics of the Benchmark Jobstream

Table 9.8 displays certain system parameters relating to paging activity that were measured during the benchmark.

63.4 page faults per second
55.8 pages transferred per second
15.9 paging transfers (physical I/Os) per second

Table 9.8 – Paging Activity Measures

Based upon knowledge of device characteristics, the average number of bytes transferred per swap and per file operation, and the page I/O clustering factors evident from Table 9.8, we calculate the I/O service times shown in Table 9.9.

.150 seconds per two way (in-out) swap
.037 seconds per file I/O
.039 seconds per paging transfer

Table 9.9 – I/O Operation Service Times

First we calculate service demands for interactive users. The CPU service demand is .74 seconds. The disk service demand due to file I/O is $12.4 \times .037 = .46$ seconds. From Table 9.8, the ratio of page faults to paging transfers is $63.4/15.9 = 3.99$. Thus an average interaction will cause $158/3.99 = 39.6$ paging transfers, with a resulting disk service demand of $39.6 \times .039 = 1.54$ seconds. In the benchmark, interactive think times were set to zero. (The system under study had extremely long response times, so users often typed ahead.) Thus there were always 7 ready and 6 active interactive users. We use the third component of the swapping approximation: each interaction requires $1 + .74/1 = 1.74$ swaps (the quantum length was 1 second), so interactive disk service demand due to swapping is $1.74 \times .150 = .26$ seconds. Total disk service demand is therefore $.46 + 1.56 + .26 = 2.26$ seconds.

Next we consider the batch job. CPU service demand is 330 seconds. Disk service demand due to file I/O is $918 \times .037 = 34$ seconds. Each batch job will cause $101386/3.99 = 25410$ paging transfers, with a resulting service demand of $25410 \times .039 = 991$ seconds. Since the batch job is not swapped, its total disk demand is $34 + 991 = 1025$ seconds.

Because there are always 7 ready and 6 active interactive users, we can take a short cut, analyzing the low-level model only one time, with a population of 1 batch job and 6 interactive users. With the exception of interactive response time, all interesting system performance measures can be obtained directly from the results of this analysis. Interactive response time is calculated as in the previous case study, by applying the response time law with $N=7$, $Z=0$, and X equal to the throughput obtained from the evaluation. Table 9.10 displays both observed and projected performance measures.

performance measure	observed	projected
total CPU utilization	.30	.32
swapping rate (swaps/sec.)	.72	.64
interactive		
throughput (int's./min.)	22.2	22.2
response time (secs.)	18.9	19.0
batch		
throughput (jobs/min.)	.0091	.0082

Table 9.10 – Original System

Next, we explore the effect of eliminating the interactive workload, running the batch job in isolation. The swapping rate will be zero. The cache will be shared by VMS and the batch job, rather than among 8 processes. It will expand to occupy the space vacated by the interactive users, increasing in size from 150 to 450 pages, a factor of 3. Our linear

approximation to the effectiveness of the shared page cache estimates that the ratio of page faults to paging transfers will be $3.99 \times 3 \times 8/2 = 47.9$. We therefore calculate that the batch job's disk service demand due to paging will be $101386/47.9 \times .039 = 82.5$ seconds, and that its total disk service demand will be $34+82.5 = 116.5$ seconds. We evaluate the low-level model once, with a single batch job. Table 9.11 displays both observed and projected performance measures.

performance measure	observed	projected
total CPU utilization	.68	.73
batch throughput (jobs/min.)	.124	.133

Table 9.11 – Batch Only

Finally, we explore the effect on the original workload of doubling the size of memory. Once again, the swapping rate will be zero. All seven interactive users will be memory resident, so the page cache will be shared by 9 rather than 8 active processes. The size of the cache will increase from 150 to 1125 pages, a factor of 7.5. The linear approximation to the effectiveness of the shared page cache estimates that the ratio of page faults to paging transfers will be $3.99 \times 7.5 \times 9/8 = 33.7$. Interactive disk service demand due to paging will be $158/33.7 \times .039 = .183$ seconds, and total interactive disk service demand will be $.46+.183 = .643$ seconds. Batch disk service demand due to paging will be $101386/33.7 \times .039 = 117$ seconds, and total batch disk service demand will be $34+117 = 151$ seconds. We simply can evaluate the low-level model with a single batch job and 7 interactive customers. Table 9.12 displays both observed and projected performance.

performance measure	observed	projected
total CPU utilization	.89	.95
interactive throughput (int's./min.) response time (secs.)	55. 7.6	65.7 6.38
batch throughput (jobs/min.)	.040	.026

Table 9.12 – Additional Memory

The projected performance measures shown in Tables 9.10 - 9.12 are sufficiently accurate to be useful. The discrepancies are reasonable when we consider the magnitude of the system modifications, the crudeness of

the linear approximation to shared page cache effectiveness, and the absence of any consideration of the effect of paging and swapping rates on CPU overhead.

9.7. Summary

Memory and its management affect the performance of computer systems in two major ways. The existence of a memory constraint can impose a bound on the multiprogramming level, and thus the throughput, of a system. The overhead associated with memory management can impede the progress of "useful" work. In this chapter we have presented techniques for representing these effects, techniques which extend the flexibility of separable queueing network models.

It never is possible to represent every detail of an operating system's memory subsystem in a queueing network model. However, *nor is it necessary or desirable to do so.* This latter point is a philosophical cornerstone of computer system analysis using queueing network models, and cannot be overemphasized. In each particular modelling study — for each configuration, workload, and set of questions to be investigated — it is imperative to identify the *essential* characteristics of the system — those that can be expected to have primary effects on performance — and to represent these and only these in the model. A large body of case study literature testifies to the success of this approach.

In closing this chapter, we should mention two related points. First, the fact that we have organized Part III on a "subsystem" basis rather than on a "technique" basis means that the broad applicability of certain techniques is not emphasized. As an example, Algorithm 9.1 for evaluating single class memory constrained subsystems is applicable to any subsystem in which there is a population constraint. (See Exercise 2.)

The second related point is a brief mention of cache memory: relatively small, fast memory sometimes interposed between the CPU and primary memory, which is managed by hardware and firmware in a manner not unlike the paging that may occur one level removed in the memory hierarchy. The effect of cache memory is usually included in a queueing network model simply as an adjustment to the service demand at the CPU. This is consistent with the decomposition approach, since memory references occur extremely frequently relative to other events. The analyst must be aware that a statement about the instruction execution rate of a machine with a cache must necessarily rely on some assumption about the cache hit ratio, and that this assumption should be verified, probably by benchmark.

9.8. References

The implications of the fact that throughput is convex with respect to multiprogramming level were noted by Dowdy, Gordon, and Agre [Dowdy et al. 1979].

Brandwajn [1974] first analyzed single class memory constrained systems using a decomposition approach, although he did not couch the analysis in the simple terms of an FESC. Lazowska and Zahorjan [1982] and Brandwajn [1982] independently developed the extension to multiple classes. An interesting alternative for evaluating single class models with non-homogeneous memory requirements was suggested by Brown, Browne, and Chandy [Brown et al. 1977].

The iterative analysis of swapping behavior presented in Section 9.4 is due to Lazowska [1979]. The analysis of a paging system presented in Section 9.5 comes from Graham and Lazowska [1978].

The case study of the early IBM virtual memory system was conducted by Boyce and Warn [1975]. Lazowska [1979] performed the VAX/VMS case study. Hodges and Stewart [1982] use the same techniques to analyze a more recent version of VAX/VMS; this system is described in detail by Levy and Eckhouse [1980]. A good overview of memory management in general, and of paging and segmentation in particular, is provided by Denning and Graham [1975].

[Boyce & Warn 1975]
J.W. Boyce and David R. Warn. A Straightforward Model for Computer Performance Prediction. *Computing Surveys 7*,2 (June 1975), 73-93. Copyright © 1975 by the Association for Computing Machinery.

[Brandwajn 1974]
Alexandre Brandwajn. A Model of a Time-Sharing System Solved Using Equivalence and Decomposition Methods. *Acta Informatica 4*,1 (1974), 11-47.

[Brandwajn 1982]
Alexandre Brandwajn. Fast Approximate Solution of Multiprogramming Models. *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (1982), 141-149.*

[Brown et al. 1977]
R.M. Brown, J.C. Browne, and K.M. Chandy. Memory Management and Response Time. *CACM 20*,3 (March 1977), 153-165.

[Denning & Graham 1975]
Peter J. Denning and G. Scott Graham. Multiprogrammed Memory Management. *Proc. IEEE 63*,6 (June 1975), 924-939.

[Dowdy et al. 1979]
Lawrence W. Dowdy, Karen D. Gordon, and Jonathan R. Agre. On the Multiprogramming Level in Closed Queuing Networks. Technical Report TR-831, Department of Computer Science, University of Maryland, November 1979.

[Graham & Lazowska 1978]
G. Scott Graham and Edward D. Lazowska. Quark: A Performance Evaluation Package for an Operating Systems Course. Technical Report 78-04-01, Department of Computer Science, University of Washington, April 1978.

[Hodges & Stewart 1982]
Larry F. Hodges and William J. Stewart. Workload Characterization and Performance Evaluation in a Research Environment. *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (1982), 39-50.

[Lazowska 1979]
Edward D. Lazowska. The Benchmarking, Tuning and Analytic Modelling of VAX/VMS. *Proc. ACM SIGMETRICS Conference on Simulation, Measurement and Modeling of Computer Systems* (1979), 57-64. Copyright © 1979 by the Association for Computing Machinery.

[Lazowska & Zahorjan 1982]
Edward D. Lazowska and John Zahorjan. Multiple Class Memory Constrained Queueing Networks. *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (1982), 130-140.

[Levy & Eckhouse 1980]
Henry M. Levy and Richard Eckhouse, Jr. *Computer Programming and Architecture: The VAX-11.* Digital Press, 1980.

9.9. Exercises

1. Suppose that in the example of Section 9.2 the observed average multiprogramming levels of the three classes had been 2.60, 0.40, and 1.75, but that no additional information was available (i.e., you did not know the actual distribution of multiprogramming mixes).
   a. How could you analyze this system using approximate MVA?
   b. How could you analyze this system using exact MVA?

2. Consider a Control Data 6000-series batch computer system consisting of a CPU, $K-1$ disks, and P peripheral processors, with a fixed multiprogramming level of N jobs. A job desiring disk service first must contend for access to any one of the PPs. Once allocated, the PP is held while the job contends for and uses the specific disk on which its data resides. At the conclusion of the I/O activity, both the disk and the PP are released, and the job enters the CPU queue. Thus, although there may be N jobs and $K-1$ disks, at most P jobs can be using disks simultaneously. The actual number may be less than P, either because fewer jobs desire disk service, or because several jobs desire access to the same disk.

   a. Draw an analogy between this modelling problem and the single class memory constraint problem discussed in Section 9.3.

   b. Analyze a system in which there are 10 jobs, a CPU at which each job has a service demand of 50 seconds, 3 PPs, and 5 disks at which each job has service demands of 20, 25, 30, 35, and 40 seconds, respectively. Report CPU utilization, disk utilizations, and average job response times. (Use the Fortran program in Chapter 18, extended to accommodate FESCs as described in Chapter 20.)

   c. Analyze the same system ignoring the PP constraint. (That is, represent the system using a separable single class model with 6 centers and 10 jobs.) What error in job response times results from this assumption? How about CPU utilization?

3. Re-work the example of Section 9.3.1 for the following values of think time:

   a. 10 seconds

   b. 180 seconds

   Simpler approaches to modelling memory constraints do not require the use of FESCs. The case study in 9.6.1 presents one such approach. Another approach is simply to ignore the memory constraint, which causes the model to be separable and thus amenable to the standard MVA algorithms.

   c. For think times of 10, 60, and 180 seconds in this example, how well do you think each of the simpler approaches will work?

   d. Test your intuition by applying both approaches in these three cases, and comparing the results to those obtained using the more accurate flow equivalent technique.

4. Some computer systems do not impose a fixed limit on the number of jobs that can be loaded in memory, but instead load jobs in a FCFS manner until either there are no jobs left to be loaded or no memory in which to load them.
   a. In the case where all jobs can be thought of as belonging to a single class, how can Algorithm 9.2 be used to model such systems?
   b. If jobs in the system have widely differing memory requirements (e.g., many small jobs but occasional very large jobs), we may wish to model the system using multiple job classes. In this case, how can Algorithm 9.3 be used?
5. In Section 9.5 a technique was described for modelling the primary effect of the change in page fault rate with system load (or equivalently with main memory allocation per job): the change in the service demand at the paging device. An important secondary effect is a change in CPU overhead per job due to page fault handling.
   a. How would you reflect this secondary effect in the model (i.e., what parameters would you change)?
   b. How would you determine appropriate parameter values for a specific system?
6. Suppose that a system contains a number of disks dedicated to swapping, and a number dedicated to paging.
   a. What modifications to the techniques of this chapter need to be made for such systems?
   b. What additional measurement information would be required to parameterize such models?
   c. In the absence of such measurements, what reasonable guesses could you make to allow you to analyze the model?