1980

Operating Systems Articles

Peter J. Denning

Dorothy E. Denning

Report Number:
80-336

OPERATING SYSTEMS ARTICLES

Peter J. Denning
Dorothy E. Denning

Computer Sciences Department
Purdue University
W. Lafayette, IN 47907

CSD-TR-336

April 12, 1980

PREFACE

This report contains a series of revised articles on aspects of operating systems:

Kernel
Thrashing
Working set
Swapping
Distributed Systems
Virtual Memory
Queueing Network Models

These articles will appear in the 1981 revised edition of the Ralston and Meek Encyclopedia of Computer Science, published by Van Nostrand Reinhold.

KERNEL

Peter J. Denning
Dorothy E. Denning

The term kernel (and sometimes nucleus) is applied to the set of programs in an operating system which implement the most primitive of that system's functions. The precise interpretation of kernel programs, of course, depends on the system; however, typical kernels contain programs for four types of functions:

1. Process Management: routines for switching processors among processes; for scheduling; for sending messages or timing signals among processes; for creating and removing processes.

2. Memory Management: routines for placing, fetching, and removing pages or segments in or from main memory.

3. Basic I/O Control: routines for allocating and releasing buffers; for starting I/O requests on particular channels or devices; for checking the integrity of individual data transmissions.

4. Security: routines for enforcing the access and information-flow control policies of the system; for changing protection domains; and for encapsulating programs.

In some systems the kernel is larger and provides for more than these classes of functions; in others, it is smaller. Each of the classes of kernel programs contains routines for handling interrupts pertaining to the class function; for example, clock interrupts are handled in Class 1, page faults in Class 2, channel completion interrupts in Class 3, and protection violations in Class 4. Some systems order the classes hierarchically (e.g., in order 1, 2, 3, 4) so that programs in the given class can invoke services of programs of lower classes. For example, memory management (Class 2) can be implemented by a collection of processes, the coordination of which is managed by process management routines (Class 1).

The reader should not confuse the system kernel with the portion of the operating system which is continuously resident in main memory. Two criteria determine whether a particular system module (either routine or table) should be resident: its frequency of use, and whether the system can operate at all without it. For example, file directories can be maintained in address spaces, so that they can be swapped out of main memory when not in use. Status information for inactive processes can similarly be swapped out. The resident part of the operating system is a subset of the kernel.

THRASHING

Peter J. Denning
Dorothy E. Denning

Thrashing is a collapse of processing efficiency caused by attempted overcommitment of multiprogrammed main memory. If memory is overcommitted, at least one process will not have its working set fully present, and so will operate inefficiently. (A working set is the smallest set of instructions and data needed in main memory for efficient processing of a program. It changes as the program executes.) If the memory management policy attempts to satisfy the inefficient process by preempting space from other working sets, they, too, will join in the mode of inefficient operation. When the system is thrashing, most of the processes will be waiting in the swapping queue rather than the CPU ready list.

To avoid thrashing, it is necessary both to have a good estimate of a process's working set, and to control the level of multiprogramming so that the totality of active working sets does not exceed the main store. A working set can be measured accurately by sampling page (or segment) usage bits of a process regularly (Rodriguez, 1973) or by measuring the processing time since last use of a page (Morris, 1972). Global memory policies (such as the "clock" policy used in VM/370 or Multics) do not distinguish pages by process when measuring usage; these policies tend toward inaccurate working set estimates because the measured

utility of pages of one process is confounded by the behavior of all the active processes.

One method of controlling the level of multiprogramming is to defer activating the highest priority waiting process until the pool of unused space is sufficient to contain its working set. This method was used successfully in a CP-67 system (Rodriguez, 1973). Another method, which avoids a pool of unused space, assigns fixed priorities to active processes; if the total of active working sets exceeds memory, space is preempted from the lowest priority active working set. In this way the k highest priority processes at a given time have their whole working sets in memory (for some k); the remaining process has at most a portion of its working set present. With a sufficiently large main memory, the mean value of k will be high enough to maintain high processing efficiency. This control mechanism prevents thrashing by preventing feedback: No process can preempt space held by higher-priority working sets. (See Wilkes, 1975.)

Thrashing can also be mitigated by reducing the access-time ratio between the main store and backing store.

Thrashing has been studied as a natural phenomenon of queueing networks by Courtois (1977). A detailed survey of its causes and prevention has been published by Denning (1980).

REFERENCES

1977. Courtois, P. J. Decomposability, Academic Press.

1980. Denning, P. J., "Working sets past and present," IEEE Trans. Software Engrg. SE-6, 1 (January), 64-84.

1973. Rodriguez-Rosell, J., and Dupuy, J. P., "The design, implementation, and evaluation of a working set dispatcher," Comm. ACM 16, 4 (April), 247-253.

1972. Morris, J. B., "Demand paging through the use of working sets on the MANIAC II," Comm. ACM 15, 10 (October), 867-872.

1975. Wilkes, M. V., Time Sharing Computer Systems, 3rd Ed., Elsevier/MacDonald.

WORKING SET

Peter J. Denning
Dorothy E. Denning

Working set, short for "set of working information," is a concept as old as electronic computing. It denotes the smallest subset of a program's address space which can be loaded into the main memory without generating excessive overhead from overlaying, the process of replacing old program segments by new ones fetched from the secondary memory. The working set varies in size and content as the program executes.

The working set is a measure of the program's dynamic memory demand -- the information that ought to be in the main memory. A separate concept is the program's resident set -- the information actually in main memory. A memory policy is a "working set policy" only if its resident set closely approximates (but seldom underestimates) the working set at any given time.

A memory policy that assigns a fixed size resident set to a program is not a working set policy. In some systems, resident sets are called "working sets", while in others (such as IBM's MVS) mean resident set sizes are called "working sets" -- even though there is no attempt to determine a program's true dynamic memory demand. These false "working set policies" do not perform as well as true working set policies.

Working sets can be determined from program structure or from measurement. A program's working set on the Burroughs B6700 series is determined by the semantics of Algol; it comprises the current procedure segment, the stack, and all arrays accessible from activated procedures. A program's working set on most paging machines is determined by measurement; it is typically the set of pages or segments whose usage bits are found ON when sampled at regular intervals in the program's virtual time. (Virtual time is program execution time with all interruptions removed; it is usually measured by counting memory references.) It is possible to specify precisely the relationships among working set size, processing efficiency, and mean time between references to objects not in the working set (Denning 1968, 1978).

It is important that working sets be measured separately in each program's virtual time. Many systems lump all active programs together while sampling usage bits; in this case, the working set estimate of one program is confounded by the behavior of all programs. This leads to suboptimal resident sets and can significantly degrade throughput and response time.

The working-set principle of dynamic multiprogramming asserts that a task may be active only if its working set is in main memory. A memory management policy that implements this principle will guarantee each active task a minimal level of processing efficiency, and will usually protect a virtual memory system from thrashing (q.v.). When installed on an experimental CP-67, the working set policy significantly improved the system's

throughput and response time (Rodriguez 1973). The available theory and experimental data strongly support the hypothesis that the working set policy is near optimal among nonlookahead policies (Denning 1980).

Working set policies exploit the property called locality (of reference) in program behavior. This is the experimentally-observed property that a program's virtual time can be divided into a succession of phases, intervals during which the program restricts all its references to a subset of its information. The information referenced during a given phase is called the locality set of that phase. The working set estimates the current locality set. When most virtual time is covered by phases long compared to the secondary memory access time (the usual case), the working set is an excellent predictor of memory demand in the immediate future. In paging systems it is sometimes advantageous to "restructure" a program by assigning small, logical segments to large pages so as to preserve in the page references the locality originally present in the segment references. This technique can reduce a program's space-time product by factors of 3 to 10 as compared to assigning segments to pages in the order of appearance in the program's text. (The space-time product is the integral, over virtual time, of the size of the resident set.)

Because the performance of a virtual memory system depends on program locality, and because locality is a direct result of the way a programmer designs an algorithm and its associated data

structure, it is not true (as is often claimed) that virtual memory systems insulate the programmer completely from memory management. The performance payoff of virtual memory accrues to those who invest the small effort required to achieve good locality in their programs.

REFERENCES

1968. Denning, P. J., "The working set model for program behavior," Comm. ACM 11, 5 (May), 323-333.

1973. Rodriguez-Rosell, J., and Dupuy, J. P., "The design implementation, and evaluation of a working set dispatcher," Comm. ACM 16, 4 (April).

1978. Denning, P. J., and Slutz, D. R., "Generalized working sets for segment reference strings," Comm. ACM 21, 8 (September).

1980. Denning, P. J., "Working sets past and present," IEEE Trans. Softw. Engrg. SE-6, 1 (January), 64-84.

SWAPPING

Peter J. Denning

Swapping is a name for the information transfer that occurs when a program is temporarily unloaded from main to secondary storage and later reloaded to continue processing. The term originated in the time sharing systems of the early 1960s. Because there was no memory protection hardware to isolate multiple programs, these early systems permitted only one user program at a time to reside and execute in the main memory. When a program reached the end of a time slice or stopped for I/O, the operating system exchanged it for another waiting program.

Most modern operating systems use multiprogrammed virtual memory. In these systems, there are two kinds of information transfer between main and secondary memory:

1. Loading a program at the start of an execution period, and unloading it at the end of that period.

2. Fetching new pages or segments on demand during the execution period.

"Swapping" is often used to name the first type of information transfer, and "demand fetching" (or "demand paging") the second type. The term "roll-in" is sometimes used to name the process of loading a program, and "roll-out" the process of unloading.

Both types of transfer need not be used in the same system. CDC 6000 series computers, for example, load a complete program for execution; these machines employ swapping but no form of demand fetching. On the other hand, paged virtual memory systems are capable of starting a program with no initial loading; in this case demand paging also serves to load the program after the start of execution. This is not an effective use of demand paging. It is much more efficient to load and unload full working sets at the starts and ends of execution periods; demand fetching should be used to add pages to the working set during the execution period.

Early time sharing systems had to control the overhead of swapping. In CTSS, for example, the CPU would be idle during a swap because the user memory was uniprogrammed. The CTSS multilevel scheduler started programs at priority levels whose quanta were at least as long as the swap time; this limited CPU idle time due to swapping to 50%. (See Corbato, 1962.) Schedulers in modern multiprogramming systems do not pay as much attention to this because the swapping of one program occurs in parallel with the execution of another.

REFERENCE

1962. Corbato, F. J., M. Merwin-Daggett, and R. C. Daley, "An experimental time-sharing system," Proc. Spring Joint Computer Conference, Vol. 21, pp. 335-344. In Rosen (ed.), Programming Languages and Systems, McGraw-Hill (1967).

DISTRIBUTED SYSTEMS
(Addition to Operating Systems Article)

Peter J. Denning

Distributed Systems. Mini and micro computer technology now make it feasible to spread operating systems functions over a collection of different computers, each with its own private processor and memory. These computers are usually all at the same site and they communicate by means of a high speed data network. Important functions, such as a file system or the login authenticator, can be permanently housed in one (or several) of the computers. A task requiring text-editing can be routed to a computer specializing in word processing, while another requiring large-scale computation can be routed to a computer specializing in vector operations.

Two kinds of data network ("net" for short) are coming into common use. One is the contention network, a cable up to 1 km long to which the computers are connected. Each computer monitors all transmissions, looking for packets containing its name as destination. A computer may not transmit if the net is busy. If a computer finds its transmission jammed by another computer (because both thought the net was free), it stops, waits a random interval, and retries the transmission. Because this type of network is passive, the failure of any computer will not put the entire net out of commission. (But it may bring the

system down if it is a critical computer, such as the one housing the file system.) This type of network is sometimes called an "ethernet" because the cable is a passive medium, an ether; it was pioneered at Xerox Palo Alto Research Center.

The second type of network is the ring network. Here the stations act as repeaters arranged in a circle; each is constantly receiving packets from one neighbor and retransmitting them to its other neighbor. A computer wishing to transmit awaits an empty packet, which it replaces by a message-containing packet. A computer monitors packets coming by, looking for those with its name as destination; when one such comes by, the computer removes the message and uses the same packet to return an acknowledgement to the sender. Because this type of network is active, a failure of any station will put the net out of commission. This form of network was pioneered at the Computer Laboratory of the University of Cambridge.

Both the contention network and ring network are examples of broadcast networks: all stations can monitor all transmissions. Long distance communications are usually handled with packet-switching networks, which deliver packets only to the stations addressed. (See Networks.)

Operating systems designed to exploit these architectures were still highly experimental in 1980. They are best modeled in the traditional way, as a collection of processes

exchanging messages; but each process may have its own computer and each message is sent by the data network. Aside from the high degree of actual concurrency among processes, these experimental systems are functionally not much different from traditional operating systems.

Three major research problems confront the designers of distributed operating systems: 1) Separating functions into enough modules so that the failure of any one (e.g., the file system computer) does not disable the whole system. 2) Learning how to recover from errors which occur while a synchronization operation is in progress. 3) Learning how to distribute operating system functions, perhaps with multiple copies, among the nodes of long-distance networks to facilitate load sharing.

VIRTUAL MEMORY

Peter J. Denning

The term virtual memory (or virtual storage) denotes the memory of a virtual (i.e., simulated) computer. An address (or name) space N of a program is the set of all addresses that can be generated by the processor as it executes the program; if the processor generates a-bit addresses, N contains at most 2^a words. The name space N is frequently referred to as the "virtual address space" or the "virtual memory" within which the processor operates. The memory space M of the machine is the set of all real location addresses recognized by the main memory hardware; if this hardware recognizes b-bit addresses, M contains 2^b words.

As shown in Figure 1, an address translation mechanism is interposed between the processor and memory. This mechanism uses a mapping table, f, that specifies the correspondence between virtual addresses (in N) and real addresses (in M). If the mapping table entry for a given virtual address is marked as "undefined", the mapper will generate an addressing fault in case the processor attempts reference to that address. The fault handler will locate the missing information in the auxiliary memory, move it into main memory (perhaps also moving out other information to make room), and update the table f to show the changes.

Then, when the interrupted program is resumed, it will find the required table entry defined and can proceed.

An example of the scheme of Figure 1 is the core/drum configuration, in which M is a core memory and A is a drum. Another example is the cache/core configuration, in which M is a high-speed semiconductor register memory (called a "cache") and A is a core memory or a slow-speed semiconductor memory.

Virtual memory solves the relocation problem because it permits program pieces to be moved around in memory without altering their virtual addresses. It solves the memory protection problem because each program piece can have its own access mode. In case the size of N exceeds the size of M, the virtual memory system also solves the memory management problem by determining which subset of N will reside in M. The CDC 6000 series illustrates that N can be smaller than M, although in this case N is not called "virtual memory".

Implementation. The map f is usually a directly indexed table. Its entries correspond to program blocks rather than individual virtual addresses. The blocks are normally used both as units of auxiliary memory storage and as units of transfer. Each block of N is a set of contiguous addresses with a base address and a length. A virtual address x must be represented (or translated to) the form (b,w), where b is

a block index and w is an offset (relative address) in block b. The mapping table's entry for block b, denoted f(b), gives the base of the block of M containing b. The translation process consists of three steps:

$$x \longrightarrow (b,w) \longrightarrow (f(b),w) \longrightarrow f(b)+w,$$

as shown in Figure 2. The translation of x to (b,w) takes time T_1; the translation of (b,w) to (f(b),w) takes time T_2 (the time to look up f(b) in the table); and the translation of (f(b),w) to f(b)+w takes time T_3.

The translation function would be impractical unless the total translation time $T_1+T_2+T_3$ can be made small compared to the main memory reference time. Because the time to add two numbers is small, the efficiency of the translation operation actually depends on T_1+T_2.

The two most common methods of making T_1 negligible or zero are segmentation and paging. A segmented name space is partitioned into blocks of various sizes (segments), usually corresponding to logical regions. In the Burroughs B5000 and later series, for example, the Algol compiler creates segments corresponding to the block structure of the language and the organization of the data. (Organick, 1973) The Honeywell 6180 (which implements a segmented name space under Multics) requires the programmer to define the seg-

ments and refer to words by symbolic two-part addresses of the form (segment-name, word-name). (Organick, 1972) A paged name space is partitioned into blocks of the same size (pages). Since the page boundaries bear no prior relation to logical boundaries in the name space, the programmer is not generally apprised of the pagination of his name space; however, the compiler or loader may be designed to reorganize information among pages to improve performance when the cost of such reorganization (which is high) can be justified. Paging's principal attraction has been its simple design.

Under segmentation, the address space is partitioned into regions by the programmmer or compiler; each region becomes a block. In this case all virtual memory references are programmed or compiled in the form of pairs (b,w). Thus, $T_1 = 0$ when segmentation is used.

Under paging, the address space is partitioned into blocks all of the same size, say s bytes. All addresses compiled in the program are linear offests relative to the base of the name space. The computation x ⟶ (b,w) is specified by

$$(b,w) = ([x/s], x \bmod s),$$

where the brackets denote "integer part of" and x mod s

denotes the remainder in dividing x by s ($0 \leq x \bmod s < s$). If $s = 2^q$ (for some $q \geq 0$) and binary arithmetic is used, this computation is trivial: w is specified by the q low-order bits of the register containing x and b is specified by the remaining bits. Thus $T_1 = 0$ when paging is used.

This leaves T_2 as the only time of significance in a paging or segmentation scheme. Most systems do not provide special, high-speed memory for storing the entire mapping table. Accordingly, the time T_2 would seem to be comparable with the main memory reference time, and the objective of making T_1+T_2 small would seem to be unrealizable. An ingenious solution has been found. A small associative memory, typically containing at most 16 cells, is included in the mapping mechanism. Its reference time is much faster than the main memory's. Each associative cell contains an entry of the form (x, f(x)). When the block number b of an address is determined, all the cells of the associative memory are searched in parallel using b as a key. If an entry (b, f(b)) is found, the base address f(b) of the block is available immediately without reference to the mapping table in main memory. Otherwise the mapping table is accessed and an entry (b, f(b)) replaces the least recently used entry in the associative memory. Experience has shown that the mean address translation time is typically in the range 1 to 3% of the main memory reference time with the associative memory in operation.

Most commercial virtual memory systems employ paging. This is because the addressing mechanism is especially simple and because managing block allocation and transfer in both the main and auxiliary memories is especially easy if all blocks are of the same size. However, the page size must be chosen carefully. Too small a page size will produce large mapping tables and a greater rate of page transfer between main and auxiliary memory. Too large a page size will produce poor storage utilization, since only a portion of the words on a page are likely to be referenced during its residence in main memory. Paging alone, even if properly designed, does not alter the linearity of the name space, and thus cannot offer the programmer the significant programming advantages possible in a segmented name space. A compromise using both segmentation and paging can be implemented for this purpose (see Denning 1970).

Memory protection is easily implemented within a virtual memory mechanism. This is, in fact, one of the main attractions of virtual memory. No program can access any information other than what is in its address space because each and every reference is translated with respect to the mapping table of the currently running program. Also, each entry of the mapping table is actually of the form

$$b: \quad (d, f(b), pc, L),$$

where d is a single bit set to 1 if and only if f(b) is defined, pc is a protection code indicating which types of

access are permitted to block b (e.g., read or write), and L is the length of block b (omitted in paging systems). If the offset portion w of the (b,w) address does not satisfy 0 $\leq$ w < L, or if the type of access being attempted is not authorized by pc, a protection interrupt is generated by the mapping mechanism. (See Wilkes 1975.)

Performance. The associative memory prevents address translation time from being an important factor in the efficiency of program execution in a virtual memory. The factors that affect performance are, in decreasing order of their significance, the size of the main memory space allocated to the program, locality of reference of the program, and the memory policy used (the paging algorithm in a paging system). The first and third factors are under the system's control, whereas the second is under the programmer's control. (Sometimes the compiler can assist in improving the second factor.) Most memory management policies tend to keep the most recently referenced blocks of the name space N in the main memory space allocated to a program, and to adjust the size of this space to be as small as possible without causing an undue rate of addressing faults.

Although it is true that most virtual memory systems present the programmer with a large linear name space and handle the memory management for him, it is not true that the virtual memory behaves as a random access store. The

nature of the memory management policies causes the access time to an object in the virtual memory to be short when the object or a neighbor has been referenced recently, and long otherwise. The programmer, therefore, can be confident of highly efficient operation of his program in a virtual memory only if he has successfully organized his algorithm and data to maximize "locality of reference." (This means that references are clustered to small groups of objects for extended intervals.)

History and Prospects. Virtual memories have been used to meet one or more of four needs:

1. Solving the overlay problem that arises when a program exceeds the size of the main store available to it. Paging on the Atlas machine at the University of Manchester (1959) was the first example.

2. Storing variable-size program objects off the run time stack. The size of local arrays in Algol, for example, may not be known at compile time; storing them in segments elsewhere, with fixed-size descriptors on the stack, permits the compilation of addresses. Segmentation on the Burroughs B5500 (1963) was the first example.

3. Long term storage of files and segments forces the programming of information transfers between the file

system and the virtual memory. The Multics virtual memory (1968) eliminated this by merging the two storage systems. Users can declare their own segments and keep them indefinitely in the address space.

4. Memory protection requires that references to segments be in range and conform to enabled access modes (read, write, or execute). These constraints are easily checked by the hardware in parallel with the main computation. Several experimental machines have been designed explicitly to study descriptor-based addressing as a means of memory protection and improved software reliability (Myers, 1978).

It is sometimes argued that advancing memory technology will soon permit us to have all the real memory we could possibly want and, hence, we will soon be able to dispense with virtual memory. It has long been a favorite assumption of operating systems prognosticators that resources will by next year be plentiful. Users' ambitions for new ways of using resources have, however, continually defied this assumption. It is unlikely that today's predictions of the passing of the overlay problem will prove to be any more reliable than similar predictions made in 1960, 1965, 1970, and 1975.

What is true is that paged virtual stores, which were invented as a simple solution to the overlay problem, will diminish in utility as the last three needs in the list

above become critical parts of programming environments. Virtual memories of the future will rely more on segmentation and tagged memory to achieve greater software error tolerance and to narrow the semantic gap between concepts of programming languages and concepts embodied in hardware.

REFERENCES

1970. Denning, P. J., "Virtual memory," Computing Surveys 2, 3 (September), 153-189.

1972. Organick, E. I., The Multics System: An Examination of Its Structure, MIT Press.

1973. Organick, E. I., Computer System Organization: The B5700/B6700 System, Academic Press.

1975. Wilkes, M. V., Time Sharing Computer Systems (3rd Ed.), Elsevier/North-Holland.

1976. Denning, P. J., "Fault tolerant operating systems," Computing Surveys 8, 3 (December).

1978. Myers, G. J., Advances in Computer Architecture, Wiley.

QUEUEING NETWORK MODELS

Peter J. Denning

A queueing network model embodies a set of devices, jobs, and connections. A device is a service center containing a queue for holding waiting jobs and one or more processors for rendering a given type of service. A job travels through the network, visiting one device at a time; when it completes a task (interval of service) at a device, a job moves to another device along one of the interdevice connections. The time for a job to move between devices is negligible. A device is never idle when jobs are in its queue.

Queueing networks can model a wide variety of important aspects of multiprogrammed computer systems. These include multiple-task jobs, different job classes, queueing for resources, concurrency among devices, bottlenecks, and saturation. Despite their simplicity, these models are remarkably accurate. For typical systems, these models estimate throughput and utilization to within 5% of the true values, and mean queue lengths and mean response times to within 25% of the true values.

The first successful application of a network was Scherr's machine repairman model for MIT's time sharing system, CTSS (1967). In 1971, Buzen introduced the central

server network, a model applicable to many typical configurations; he applied theory dating back to 1957 to develop efficient algorithms for calculating performance quantities in these models. Since that time, numerous validations have shown that these models work well in practice. (See Denning & Buzen, 1978, for a survey.)

Figure 1 illustrates a model of a time sharing system. Device 1 represents all the active terminals on the system and Device 2 represents the central computing facility. A thinking period corresponds to an interval when a user's job is present at Device 1, and a waiting period corresponds to an interval when his job is present at Device 2. This simple network was used by Scherr for CTSS (1967).

Figure 2 depicts the central server network. The boxes represent the devices and their queues. Device 1 is the CPU and Devices 2,...,K are I/O stations. A job enters the system at the IN port and begins with a CPU service interval (burst); it continues with zero or more I/O service intervals which alternate with further CPU bursts; finally, it exits from the OUT port. The time required for a job to travel from the IN port to the OUT port is called the response time.

Each device i has two basic parameters. The visit ratio, V_i, denotes the mean number of times a job requires service at device i. It can be a fraction; for example, V_i

= 0.5 indicates that every other job requires service at device i. The mean service time, S_i, denotes the mean time between completions when device i is busy. If S_i depends on the device's queue length, the device is load-dependent; otherwise, the device is load-independent.

A queueing network can be open or closed. If open, the number of jobs in the network, N, varies. If closed, N is fixed. A closed network can model a system that includes the source of jobs -- e.g., the users logged in at their terminals in Figure 1. A closed network can also model a subsystem operating under a backlog -- the moment one job leaves the subsystem, another is immediately injected to replace it.

Network models can be used for a simple bottleneck analysis if the parameters V_i and S_i are independent of the total number of jobs, N, in the system. As N increases, the device b for which the total service demand $V_i S_i$ is largest saturates and limits system throughput to at most $1/V_b S_b$ jobs per second. In this case, the system's mean response time is approximately $NV_b S_b$ seconds. (See Denning & Buzen, 1978.)

Network models are used to calculate the following standard performance metrics:

U_i - Utilization of device i (fraction of time device busy);

X_i - Throughput at device i (jobs per second being

completed);

X_0 - Throughput of the system (jobs per second at the OUT port);

Q_i - Mean queue length at device i;

R_i - Mean response time per visit to device i;

R_0 - Mean response time of the system for one job.

These quantities are easily derived from the basic parameters (V_i and S_i) under two basic assumptions:

Flow Balance: The number of arrivals equals the number of departures at every device of the system.

Homogeneity: The job flow from a source device to a destination device depends at most on the queue length at the source, not on any other queue length.

These assumptions lead to simple calculations of the standard performance metrics from the basic parameters. Because the cost of these calculations is small, and because the results allow useful conclusions to be drawn about real systems, the errors introduced by these assumptions are acceptable.

Solution of a Model. The state of a model is a list $\underline{n} = (n_1,\ldots,n_K)$ specifying the number of jobs present at each of the K devices. In a closed network, $n_1+\ldots+n_K = N$, the fixed load on

the network. The quantity $p(\underline{n})$ denotes the proportion of time state $\underline{n}$ is observed. It can be shown that, under the two assumptions noted above, the quantity $p(\underline{n})$ has the so-called product form,

$$p(\underline{n}) = F_1(n_1)...F_K(n_K) \; / \; G,$$

where $F_i(n_i)$ is a "device factor" that depends only on the queue length n_i and the basic parameters (V_i and S_i) at device i, and G is a normalizing constant chosen so that the sum of $p(\underline{n})$ for all $\underline{n}$ equals 1. Although the number of possible states is large, the standard performance metrics of any device can be calculated from the product form in time proportional to NK. (Bruell & Balbo, 1980).

An example of the computational algorithm derived from the product-form solution is

$$R_i(N) = S_i(1 + Q_i(N-1)) \qquad \text{(all devices i)}$$

$$X_0(N) = N \; / \; (V_1R_1(N)+...+V_KR_K(N))$$

$$Q_i(N) = X_0(N) \; V_i \; R_i(N) \qquad \text{(all devices i)}$$

These equations are evaluated sequentially for each N and iteratively for N = 1,2,... until the desired network load is reached. The initial condition is $Q_i(0) = 0$. For each value of N, the first equation calculates the mean response time per visit

to each device in terms of the mean service time per visit and the mean queue length of the network with one less job in it. The second equation calculates the system throughput as the system load divided by the mean response time. The third equation calculates the queue length from the system throughput, the visit ratio, and the mean response time per visit. (See Buzen & Denning, 1980.) A complete discussion of these algorithms, and their limitations, is given in the book by Bruell and Balbo (1980).

The basic computational algorithms for queueing networks have been extended and refined in many ways. One of the most important is for multiple job classes: different classes of jobs can be identified, and separate values of the basic parameters measured for each. The product form solution tends to be inaccurate when some job classes have priority over others, and when jobs have service time distributions of high variance at devices with first-in-first-out (FIFO) queueing disciplines. Even for these cases, good approximations have been developed to iteratively refine a product form network until the error is minimum.

Queueing network algorithms are available in several commercial, proprietary software packages.

REFERENCES

1967. Scherr, A. L., An Analysis of Time Shared Computer Systems, MIT Press.

1978. Denning, P. J., and Buzen, J. P., "The operational analysis of queueing network models," Computing Surveys 10, 3 (September), 225-261.

1980. Bruell, S. C., and Balbo, G., Computation Algorithms for Closed Queueing Networks, New York, Elsevier/North-Holland.

1980. Buzen, J. P., and Denning, P. J., "Measuring and calculating queue length distributions," IEEE Computer (April) 33-44.