OLATINWO MUBARAK

16/SCI01/034

1. **Concept of Operational Laws in Computer Systems-** Operational laws are equations which may be used as an abstract representation of a model of the average behavior of any system. They are general and make no assumptions about the behavior of the random variable that characterize the system. Using these abstractions, these laws can be applied to any device and gradually build a more complex system.
2. **Operational Laws**
   1. Little’s Law- Little’s Law is a theorem that determines the average number of items in a stationary queuing system based on the average waiting time of an item within a system and the average number of items arriving at the system per unit of time.

The law provides a simple and intuitive approach for the assessment of the efficiency of queuing systems. The concept is hugely significant for business operations because it states that the number of items in the queuing systems primarily depends on two key variables, and it is not affected by other factors such as the distribution of the service or service order.

Little’s Law can only be used in queuing systems.  In addition, the theorem can be applied in different fields, from running a small coffee shop to the maintenance of the operations of a military airbase.

Formula for Little’s Law

L = λ x W

L – the average number of items in a queuing system

λ – the average number of items arriving at the system per unit of time

W – the average waiting time an item spends in a queuing system

* 1. Space-Time Product Laws- it states that the throughput is equal to average amount of memory in use divided by average space-time product. Space-time products are often used to evaluate program performance and assign accounting charges to programs in virtual memory systems. Essentially, a program’s space-time product is equal to its execution time multiplied by the average amount of money allocated to it during its execution. Since space-time products, response time, and throughput are all used as indicators of system performance, it is interesting to examine the manner in which these quantities are related.
  2. Forced-Flow Law- The Forced-Flow Law (FFL) relates throughputs at individual resources within a system to the overall system throughput. It is the average no of visits that a system level job makes to that resource. The general residence time law is the sum of the product of its average residence time at each resource and the number of visits it makes to that resource.

The Forced-Flow Law:  
 *λk* = *V kλ.*The average arrival rate to resource *k* is the total system arrival rate times the expected number of visits made to resource *k*. Because of the Conservation Law, we could also state the FFL in terms of output rates, Λ and Λ*k*.

* 1. Conservation Law- What goes in must (normally) come out.

Consider a system with arrival rate of λ and an output rate of Λ. If the system is not overloaded and no customers are created or destroyed inside the system, then λ = Λ.

Creating customers in a system is called forking. Destroying customers is called joining. It’s possible to model systems with these behaviors, but usually difficult, so we won’t see any examples until the end of

class.

* 1. Utilisation Law- It states that, the utilization of a resource is equal to the product of the throughput of that resource and the average service requirement at that resource. If we know the amount of processing that each job requires at a resource then we can calculate the utilisation of the resource. The total amount of service that a system job generates at the ith resource is called the service demand,

Di : Di = SiVi

The utilisation of a resource, the percentage of time that the ith resource is in use processing to a job, is denoted Ui

Ui = XiSi = XD

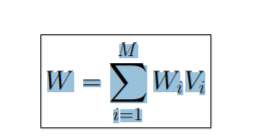
* 1. Interactive Response Time Law- The name of this law dates back to the time when most of the systems which were being modelled were mainframes processing both interactive jobs and batch jobs. The *think time*, *Z*, was quite literally the length of time that a programmer spent thinking at his terminal before submitting another job. More generally interactive systems are those in which jobs spend time in the system not engaged in processing, or waiting for processing: this may be because of interaction with a human user, or may be for some other reason.

The think time represents the time between processing being completed and the job becoming available as a request again. Thus the residence time of the job, as calculated by Little’s law as the time from arrival to completion, is greater than the system’s response time. The *interactive response time law* reflects this: it calculates the *response time*, *R* as follows:  
 *R* = *N=X - Z  
The response time in an interactive system is the residence time minus the think time.*Note that if the think time is zero, *Z* = 0 and *R* = *W*, then the interactive response  
time law simply becomes Little’s law.

* 1. General Residence Time Law- One method of computing the mean residence or response time per job in a system is to apply Little’s law to the system as a whole. However, if the mean number of jobs in the system, N, or the system level throughput, X, are not known an alternative method can be used. Applying Little’s law to the ith resource we see that Ni = XiWi , where Ni is the mean number of jobs at the resource and Wi is the average response time of the resource. From the forced flow law we know that Xi = XVi . Thus we can deduce that

Ni/X = ViWi .

The total number jobs in the system is clearly the sum of the number of jobs at each resource, i.e. N = N1 + · · · + NM if there are M resources in the system. We know from Little’s law that W = N/X and from this we arrive at the general residence time, or general response time law:



The average residence time of a job in the system will be the sum of the product of its average residence time at each resource and the number of visits it makes to that resource.

# Residence time, service demand, contention- The residence time of a job within a system will always be at least as large as the total amount of processing that each job requires.

The total amount of processing that a job requires is *D*, the total service demand,

*M*

*D* = \ *Di*

*i* =1

In general, there will be some contention in the system meaning that jobs have to wait for processing so the residence time will be larger than this, i.e. *W ≥ D*

1. **Basic queuing models-**

a. Calling population: the population of potential customers, may be assumed to be finite or infinite.

I. Finite population model: if arrival rate depends on the number of customers being served and waiting, e.g., model of one corporate jet, if it is being repaired, the repair arrival rate becomes zero.

II. Infinite population model: if arrival rate is not affected by the number of customers being served and waiting, e.g., systems with large population of potential customers.

b. System Capacity: a limit on the number of customers that may be in the waiting line or system.

I. Limited capacity, e.g., an automatic car wash only has room for 10 cars to wait in line to enter the mechanism.

II. Unlimited capacity, e.g., concert ticket sales with no limit on the number of people allowed to wait to purchase tickets.

C. Arrival Process: In terms of inter-arrival times of successive customers.

I. Random arrivals: inter-arrival times usually characterized by a probability distribution.

II. Poisson arrival process (with rate λ), where An represents the inter-arrival time between customer n − 1 and customer n, and is exponentially distributed (with mean 1/λ).

III. Scheduled arrivals: inter-arrival times can be constant or constant plus or minus a small random amount to represent early or late arrivals.

D. Arrival Processes - Finite population models: Customer is pending when the customer is outside the queueing system, e.g., machine-repair problem: a machine is “pending” when it is operating, it becomes “not pending” the instant it demands service form the repairman.

**Basic Queuing Disciplines-**

1. First-in-first-out (FIFO)- this means the oldest inventory items are recorded as sold **first** but do not necessarily mean that the exact oldest physical object has been tracked and sold. In other words, the cost associated with the inventory that was purchased **first** is the cost expensed **first**.
2. Last-in-first-out (LIFO)- this describes a method for accounting for inventories. Under this system, the last unit added to an inventory is the first to be recorded as sold.
3. Service in random order (SIRO)- Under this type of queue structure, the customer is chosen for service randomly and hence all the customers are equally likely to be selected. Therefore, the time of arrival of the customer has no consequence on the selection of the customer.
4. Shortest processing time first (SPT)- Its principle is to order jobs according to their duration and schedule them by beginning by the shorters.
5. **How to resolve basic queuing problems**

This could be by evaluating the system’s performance by either the measurement, simulation or analytical technique;

1. **Let Customers Know How Long The Wait Is**: The uncertainty of how long it will take to wait is often the cause of queue anxiety. Because of this the customers are impatient and this is a major cause of queuing problems as people want to jump the queues or altogether leave the queue.
2. **Assess and improve your queue management strategy:** How do you currently handle a long line of customers? Think about what works well and what doesn’t. Assessing the tactics used to manage the queue in the particular organization will really help solve the queuing problems being encountered.
3. **Design Your Environment To Be Able To Accommodate Queues:** Studies have shown that one of the most common issues found in lines is queue anxiety. A well-designed queuing area can help organize waiting lines, remove the possibility of queue jumpers and generally ease customer flow management.
4. I**mplement Digital Queuing Software**: Long queues can inspire customer’s irritation even disgust. But anyone can learn how to reduce queues the use of a nifty technology called a queue management system (QMS).Automating the queuing process creates more labor efficient customer lines, decreases the overall amount of walkaways as well as ultimately reducing queue times. When it’s their turn, a teller calls them to the counter to be served. They can see where they are in line by observing HDTVs hung on the walls of the organization and therefore customers are free to sit or wander and maybe grab a coffee across the street as they wait. They’re not corralled into the line like sheep. By giving customers back their time (their autonomy) one enable customers to wait in leisure. Now that’s effective queue management.
5. **Occupy Customers in The Queue**: Boredom in the queue can often lead to longer perceived waiting times. Queue solutions is to provide a distraction to people in the queue and help them continue shopping while waiting, easing up frustrations etc. Display entertaining programming on HDTVs. Prompt customers to answer surveys to report on their experience. Engaging customers is the best way to reduce the tension inherent in queueing. Because it’s typically the psychology behind queueing rather than the queues themselves that makes queues feel unbearable.
6. **Keep The Rules Of Queuing Fair And Consistent:** One of the most important characteristics of any queue problem solving method is the queuing discipline used. Simply put, the queuing discipline is the rule used to decide who goes next in a queue.

Two of the most commonly used rules are:

* First in, First out.
* Last in, First out.

Bottom line, people expect queues to be fair. It’s not like they’re happy to be stuck waiting in line, to begin with. But when everyone abides by the same rules, we can’t help but follow them too.

1. **Reduce Response times:** So, when it comes to providing service, be quick as possible. It's not possible to solve every problem immediately, but customers don’t expect that from you. What they do expect is that you give them some kind of response quickly. Having all information at your fingertips is the next step as these steps will help improve the flow of the queues and have fewer waiting times.

I. Simulation technique

II. I’ll be able to see the how the system works without it being complete. Other techniques need simulation to see if they run properly.

iii. All reports should have it, any report without it is invalid.

iv. Go through the report to check if there are any mistakes

V. The fourth report because measurement technique it shows the level every part of the system can operate to and it is the most accurate; simulation method requires relatively lesser effort, is averagely accurate and it will show the system in action to be able to know where the flaws might be.

OLATINWO MUBARAK

16/SCI01/034

The three main performance evaluation techniques are:

1. Performance measurement

2. Analytic performance modelling

3. Simulation performance modelling

**Using the following performance measurement technique, we can achieve an accurate performance report:**

* •**On-chip Performance Monitoring Counters:** All state-of-the-art high performance microprocessors including Intel's Pentium III and Pentium IV, IBM's POWER 3 and POWER 4 processors, AMD's Athlon, Compaq's Alpha, and Sun's UltraSPARC processors incorporate on-chip performance monitoring counters which can be used to understand performance of these microprocessors while they run complex, real-world workloads. This ability has overcome a serious limitation of simulators, that they often could not execute complex workloads. Now, complex run time systems involving multiple software applications can be evaluated and monitored very closely. All microprocessor vendors nowadays release information on their performance monitoring counters, although they are not part of the architecture. For illustration of on-chip performance monitoring, we use the Intel Pentium processors. The microprocessors in the Intel Pentium contain two performance monitoring counters. These counters can be read with special instructions (eg: RDPMC) on the processor. The counters can be made to measure user and kernel activity in combination or in isolation. A variety of performance events can be measured using the counters
* **Off-chip hardware measurement:** Instrumentation using hardware means can also be done by attaching off-chip hardware, two examples of which are described in this section.
* **Speed-Tracer from AMD:** AMD developed this hardware tracing platform to aid in the design of their x86 microprocessors. When an application is being traced, the tracer interrupts the processor on each instruction boundary. The state of the CPU is captured on each interrupt and then transferred to a separate control machine where the trace is stored. The trace contains virtually all valuable pieces of information for each instruction that executes on the processor. Operating system activity can also be traced. However, tracing in this manner can be invasive, and may slow down the processor. Although the processor is running slower, external events such as disk and memory accesses still happen in real time, thus looking very fast to the slowed-down processor. Usually this issue is addressed by adjusting the timer interrupt frequency. Use of this performance monitoring facility can be seen in Merten and Bhargava.

* **Logic Analyzers:** Poursepanj and Christie use a Tektronix TLA 700 logic analyzer to analyze 3D graphics workloads on AMD-K6-2 based systems. Detailed logic analyzer traces are limited by restrictions on sizes and are typically used for the most important sections of the program under analysis. Preliminary coarse level analysis can be done by performance monitoring counters and software instrumentation. Poursepanj and Christie used logic analyzer traces for a few tens of frames which covered a second or two of smooth motion.
* **Software Monitoring**: Software monitoring is often performed by utilizing architectural features such as a trap instruction or a breakpoint instruction on an actual system, or on a prototype. The VAX processor from Digital (now Compaq) had a T-bit that caused an exception after every instruction. Software monitoring used to be an important mode of performance evaluation before the advent of on-chip performance monitoring counters. The primary advantage of software monitoring is that it is easy to do. However, disadvantages include that the instrumentation can slow down the application. The overhead of servicing the exception, switching to a data collection process, and performing the necessary tracing can slow down a program by more than 1000 times. Another disadvantage is that software monitoring systems typically only handle the user activity.
* **Micro coded Instrumentation:** Digital (now Compaq) used microcoded instrumentation to obtain traces of VAX and Alpha architectures. The ATUM tool [14] used extensively by Digital in the late 1980s and early 1990s uses microcoded instrumentation. This is a technique lying between trapping information on each instruction using hardware interrupts (traps) or software traps. The tracing system essentially modified the VAX microcode to record all instruction and data references in a reserved portion of memory. Unlike software monitoring, ATUM could trace all processes including the operating system. However, this kind of tracing is invasive, and can slow down the system by a factor of 10 without including the time to write the trace to the disk.

Using the Analytical performance modelling, we can achieve an accurate performance report: Performance measurement as described in the previous section can be done only if the actual system or a prototype exists. It is expensive to build prototypes for early stage evaluation. Hence one needs to resort to some kind of modeling in order to study systems yet to be built. Performance modeling can be done using simulation models or analytical models.

**Using the following Simulation performance modelling techniques, we can achieve an accurate performance report:**

* **Trace-driven simulation:** consists of a simulator model whose input is modeled as a trace or sequence of information representing the instruction sequence that would have actually executed on the target machine. A simple trace driven cache simulator needs a trace consisting of address values. Depending on whether the simulator is modeling a unified instruction or data cache, the address trace should contain addresses of instruction and data references. Cachesim5 and Dinero IV are examples of cache simulators for memory reference traces. These simulators are not timing simulators. There is no notion of simulated time or cycles, only references. They are not functional simulators. Data and instructions do not move in and out of the caches. The primary result of simulation is hit and miss information. The basic idea is to simulate a memory hierarchy consisting of various caches. The various parameters of each cache can be set separately (architecture, mapping policies, replacement policies, write policy, statistics). During initialization, the configuration to be simulated is built up, one cache at a time, starting with each memory as a special case. After initialization, each reference is fed to the appropriate top-level cache by a single simple function call. Lower levels of the hierarchy are handled automatically. One does not need to store a trace while using cachesim5, because Shade can directly feed the trace into cachesim5
* **Execution Driven Simulation:** There are two meanings in which this term is used by researchers and practitioners. Some refer to simulators that take program executables as input as execution driven simulators. These simulators utilize the actual input executable and not a trace. Hence the size of the input is proportional to the static instruction count and not the dynamic instruction count. Mis-predicted branches can be accurately simulated as well. Thus, these simulators solve the two major problems faced by trace-driven simulators. The widely used Simple-scalar simulator is an example of such an execution driven simulator. With this tool set, the user can simulate real programs on a range of modern processors and systems, using fast execution-driven simulation. There is a fast-functional simulator and a detailed, out-of-order issue processor that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction.
* **Complete system simulation**: Many execution and trace driven simulators only simulate the processor and memory subsystem. Neither I/O activity nor operating system activity is handled in simulators like Simplescalar. But in many 9 workloads, it is extremely important to consider I/O and operating system activity. Complete system simulators are complete simulation environments that model hardware components with enough detail to boot and run a full-blown commercial operating system. The functionality of the processors, memory subsystem, disks, buses, SCSI/IDE/FC controllers, network controllers, graphics controllers, CD-ROM, serial devices, timers, etc are modeled accurately in order to achieve this. While functionality stays the same, different microarchitectures in the processing component can lead to different performance. Most of the complete system simulators use microarchitectural models that can be plugged in and out. For instance, SimOS, a popular complete system simulator provides a simple pipelined processor model and an aggressive superscalar processor model. SimOS and SIMICS can simulate uniprocessor and multiprocessor systems.
* **Stochastic Discrete Event Driven Simulation**: It is possible to simulate systems in such a way that the input is derived stochastically rather than as a trace/executable from an actual execution. For instance, one can construct a memory system simulator in which the inputs are assumed to arrive according to a Gaussian distribution. Such models can be written in general purpose languages such as C, or using special simulation languages such as SIMSCRIPT. Languages such as SIMSCRIPT have several built-in primitives to allow quick simulation of most kinds of common systems. There are built-in input profiles, resource templates, process templates, queue structures, etc. to facilitate easy simulation of common systems. An example of the use of event-driven simulators using SIMSCRIPT may be seen in the performance evaluation of multiple-bus multiprocessor systems in Kurian et. Al
* **Program Profilers**: There are a class of tools called software profiling tools, which are similar to simulators and performance measurement tools. These tools are used to generate traces, to obtain instruction mix, and a variety of instruction statistics. They can be thought of as software monitoring on a simulator. They input an executable and decode and analyze each instruction in the executable. These program profilers can be used as the front end of simulators. A popular program profiling tool is Shade for the UltraSparc.