and then waits for a command from the terminal to load another process, overwriting the first one.

4.1.2. Multiprogramming and Memory Usage

Although monoprogramming is sometimes used on small computers, on larger computers with multiple users it is rarely used. In Chap. 2 we already saw one reason for multiprogramming—to make it easier to program an application by splitting it up into two or more processes. Another motivation is that large computers often provide interactive service to several people simultaneously, which requires the ability to have more than one process in memory at once in order to get reasonable performance. Loading a process, running it for 100 msec, and then spending a few hundred milliseconds swapping it to disk is inefficient. But if the quantum is set too much above 100 msec, the response time will be poor.

Another reason for multiprogramming a computer (also applicable to batch systems), is that most processes spend a substantial fraction of their time waiting for disk I/O to complete. It is common for a process to sit in a loop reading data blocks from a disk file and then doing some computation on the contents of the blocks read. If it takes 40 msec to read a block, and the computation takes 10 msec, with monoprogramming the CPU will be idle waiting for the disk 80 percent of the time.

Modeling Multiprogramming

When multiprogramming is used, the CPU utilization can be improved. Crudely put, if the average process computes only 20 percent of the time it is sitting in memory, with five processes in memory at once, the CPU should be busy all the time. This model is unrealistically optimistic, however, since it assumes that all five processes will never be waiting for I/O at the same time.

A better model is to look at CPU usage from a probabilistic viewpoint. Suppose that a process spends a fraction p of its time in I/O wait state. With n processes in memory at once, the probability that all n processes are waiting for I/O (in which case the CPU will be idle) is p^n. The CPU utilization is then $1 - p^n$. Figure 4-2 shows the CPU utilization as a function of n, called the **degree of multiprogramming**.

From the figure it is clear that if processes spend 80 percent of their time waiting for I/O, at least 10 processes must be in memory at once to get the CPU waste below 10 percent. When you realize that an interactive process waiting for a user to type something at a terminal is in I/O wait state, it should be clear that I/O wait times of 80 percent and more are not unusual. But even in batch systems, processes doing a lot of disk or tape I/O will often have this percentage or more.

For the sake of complete accuracy, it should be pointed out that the

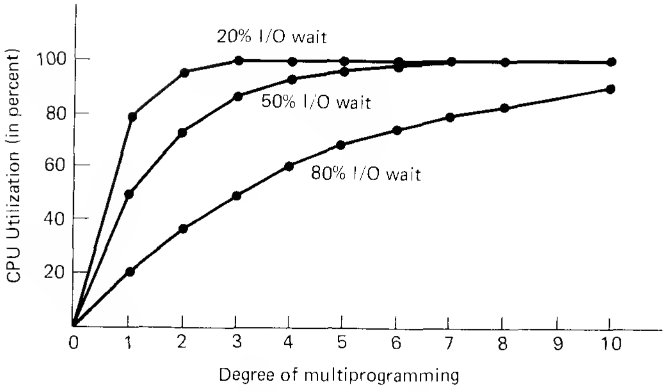


Fig. 4-2. CPU utilization as a function of the number of processes in memory.

probabilistic model just described is only an approximation. It implicitly assumes that all *n* processes are independent, meaning that it is quite acceptable for a system with five processes in memory to have three running and two waiting. But with a single CPU, we cannot have three processes running at once, so a process becoming ready while the CPU is busy will have to wait. Thus the processes are not independent. A more accurate model can be constructed using queueing theory, but the point we are making—multiprogramming lets processes use the CPU when it would be otherwise idle—is, of course, still valid, even if the true curves of Fig. 4-2 are slightly different.

Even though the model of Fig. 4-2 is simple-minded, it can still be used to make specific, although approximate, predictions about CPU performance. Suppose, for example, that a computer has 1M of memory, with the operating system taking up 200K and each user program also taking up 200K. With an 80 percent average I/O wait, we have a CPU utilization (ignoring operating system overhead) of about 60 percent. Adding another megabyte of memory allows the system to go from four-way multiprogramming to nine-way multiprogramming, thus raising the CPU utilization to 87 percent. In other words, the second megabyte will raise the throughput by 45 percent.

Adding a third megabyte would only increase CPU utilization from 87 percent to 96 percent, thus raising the throughput by only another 10 percent. Using this model the computer's owner might decide that a second megabyte was a good investment, but that a third megabyte was not.

Analysis of Multiprogramming System Performance

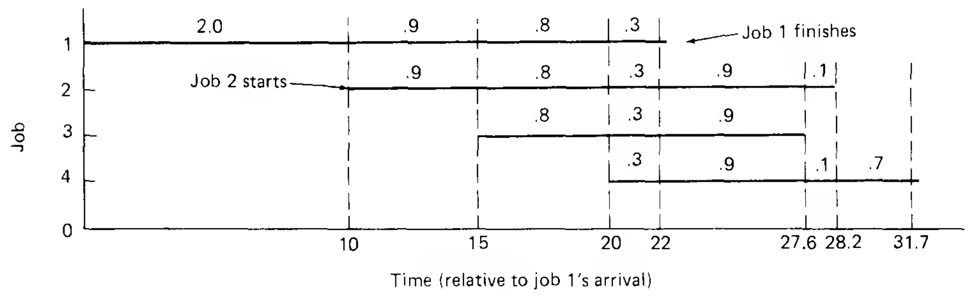
This model can also be used to analyze batch systems. Consider, for example, a computer center whose jobs average 80 percent I/O wait time. On a particular morning, four jobs are submitted as shown in Fig. 4-3(a). The first job, arriving at 10:00 A.M., requires 4 minutes of CPU time. With 80 percent I/O wait, the job uses only 12 seconds of CPU time for each minute it is sitting in memory, even if no other jobs are competing with it for the CPU. The other 48 seconds are spent waiting for I/O to complete. Thus the job will have to sit in memory for at least 20 minutes in order to get 4 minutes of CPU work done, even in the absence of competition for the CPU.

Job	Arrival time	CPU minutes needed
1	10:00	4
2	10:10	3
3	10:15	2
4	10:20	2

(a)

	#Processes			
	1	2	3	4
CPU idle	.80	.64	.51	.41
CPU busy	.20	.36	.49	.59
CPU/process	.20	.18	.16	.15

(b)



(c)

Fig. 4-3. (a) Arrival and work requirements of four jobs. (b) CPU utilization for 1 to 4 jobs with 80 percent I/O wait. (c) Sequence of events as jobs arrive and finish. The numbers above the horizontal lines show how much CPU time, in minutes, each job gets in each interval.

From 10:00 A.M. to 10:10 A.M., job 1 is all by itself in memory and gets 2 minutes of work done. When job 2 arrives at 10:10 A.M., the CPU utilization increases from 0.20 to 0.36, due to the higher degree of multiprogramming (see Fig. 4-2). However, with round robin scheduling, each job gets half of the CPU, so each job gets 0.18 minutes of CPU work done for each minute it is in memory. Notice that the addition of a second job costs the first job only 10 percent of its performance (from 0.20 to 0.18 minutes of CPU per minute of real time).

At 10:15 A.M. the third job arrives. At this point job 1 has received 2.9 minutes of CPU and job 2 has had 0.9 minutes of CPU. With three-way multiprogramming, each job gets 0.16 minutes of CPU time per minute of real time, as shown in Fig. 4-3(b). From 10:15 A.M. to 10:20 A.M. each of the three jobs gets 0.8 minutes of CPU time. At 10:20 A.M. a fourth job arrives. Fig. 4-3(c) shows the complete sequence of events.