4.2. SYSTEM MODELS

Processes run on processors. In a traditional system, there is only one processor, so the question of how the processor should be used does not come up. In a distributed system, with multiple processors, it is a major design issue. The processors in a distributed system can be organized in several ways. In this section we will look at two of the principal ones, the workstation model and the processor pool model, and a hybrid form encompassing features of each one. These models are rooted in fundamentally different philosophies of what a distributed system is all about.

4.2.1. The Workstation Model

The workstation model is straightforward: the system consists of workstations (high-end personal computers) scattered throughout a building or campus and connected by a high-speed LAN, as shown in Fig. 4-10. Some of the workstations may be in offices, and thus implicitly dedicated to a single user, whereas others may be in public areas and have several different users during the course of a day. In both cases, at any instant of time, a workstation either has a single user logged into it, and thus has an "owner" (however temporary), or it is idle.

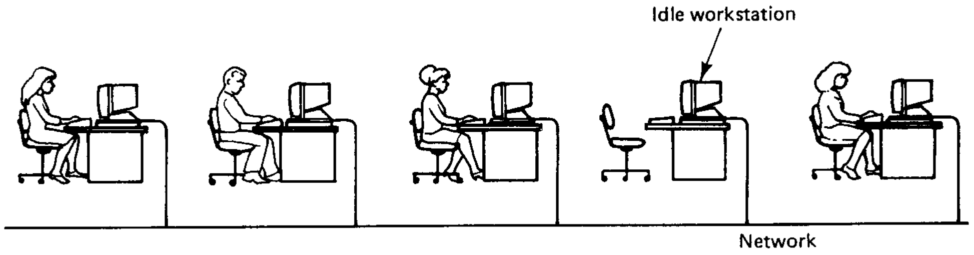

Fig. 4-10. A network of personal workstations, each with a local file system.

In some systems the workstations have local disks and in others they do not. The latter are universally called **diskless workstations**, but the former are variously known as **diskful workstations**, or **disky workstations**, or even stranger names. If the workstations are diskless, the file system must be implemented by one or more remote file servers. Requests to read and write files are sent to a file server, which performs the work and sends back the replies.

Diskless workstations are popular at universities and companies for several reasons, not the least of which is price. Having a large number of workstations equipped with small, slow disks is typically much more expensive than having one or two file servers equipped with huge, fast disks and accessed over the LAN.

A second reason that diskless workstations are popular is their ease of maintenance. When a new release of some program, say a compiler, comes out, the system administrators can easily install it on a small number of file servers in the machine room. Installing it on dozens or hundreds of machines all over a building or campus is another matter entirely. Backup and hardware maintenance is also simpler with one centrally located 5-gigabyte disk than with fifty 100-megabyte disks scattered over the building.

Another point against disks is that they have fans and make noise. Many people find this noise objectionable and do not want it in their office.

Finally, diskless workstations provide symmetry and flexibility. A user can walk up to any workstation in the system and log in. Since all his files are on the file server, one diskless workstation is as good as another. In contrast, when all the files are stored on local disks, using someone else's workstation means that you have easy access to *his* files, but getting to your own requires extra effort, and is certainly different from using your own workstation.

When the workstations have private disks, these disks can be used in one of at least four ways:

1. Paging and temporary files.
2. Paging, temporary files, and system binaries.
3. Paging, temporary files, system binaries, and file caching.
4. Complete local file system.

The first design is based on the observation that while it may be convenient to keep all the user files on the central file servers (to simplify backup and maintenance, etc.) disks are also needed for paging (or swapping) and for temporary files. In this model, the local disks are used only for paging and files that are temporary, unshared, and can be discarded at the end of the login session. For example, most compilers consist of multiple passes, each of which creates a temporary file read by the next pass. When the file has been read once, it is discarded. Local disks are ideal for storing such files.

The second model is a variant of the first one in which the local disks also hold the binary (executable) programs, such as the compilers, text editors, and electronic mail handlers. When one of these programs is invoked, it is fetched from the local disk instead of from a file server, further reducing the network load. Since these programs rarely change, they can be installed on all the local disks and kept there for long periods of time. When a new release of some system program is available, it is essentially broadcast to all machines. However, if that machine happens to be down when the program is sent, it will miss the program and continue to run the old version. Thus some administration is needed to keep track of who has which version of which program.

A third approach to using local disks is to use them as explicit caches (in addition to using them for paging, temporaries, and binaries). In this mode of operation, users can download files from the file servers to their own disks, read and write them locally, and then upload the modified ones at the end of the login session. The goal of this architecture is to keep long-term storage centralized, but reduce network load by keeping files local while they are being used. A disadvantage is keeping the caches consistent. What happens if two users download the same file and then each modifies it in different ways? This problem is not easy to solve, and we will discuss it in detail later in the book.

Fourth, each machine can have its own self-contained file system, with the possibility of mounting or otherwise accessing other machines' file systems. The idea here is that each machine is basically self-contained and that contact with the outside world is limited. This organization provides a uniform and guaranteed response time for the user and puts little load on the network. The disadvantage is that sharing is more difficult, and the resulting system is much closer to a network operating system than to a true transparent distributed operating system.

The one diskless and four diskful models we have discussed are summarized in Fig. 4-11. The progression from top to bottom in the figure is from complete dependence on the file servers to complete independence from them.

The advantages of the workstation model are manifold and clear. The model is certainly easy to understand. Users have a fixed amount of dedicated computing power, and thus guaranteed response time. Sophisticated graphics programs can be very fast, since they can have direct access to the screen. Each user has a large degree of autonomy and can allocate his workstation's resources as he sees fit. Local disks add to this independence, and make it possible to continue working to a lesser or greater degree even in the face of file server crashes.

However, the model also has two problems. First, as processor chips continue to get cheaper, it will soon become economically feasible to give each user first 10 and later 100 CPUs. Having 100 workstations in your office makes it hard to see out the window. Second, much of the time users are not using their workstations, which are idle, while other users may need extra computing capacity and cannot get it. From a system-wide perspective, allocating resources in such a way that some users have resources they do not need while other users need these resources badly is inefficient.

The first problem can be addressed by making each workstation a personal multiprocessor. For example, each window on the screen can have a dedicated CPU to run its programs. Preliminary evidence from some early personal multiprocessors such as the DEC Firefly, suggest, however, that the mean number of CPUs utilized is rarely more than one, since users rarely have more than one

	Disk usage	Advantages	Disadvantages
Dependence on file servers ↑	(Diskless)	Low cost, easy hardware and software maintenance, symmetry and flexibility	Heavy network usage; file servers may become bottlenecks
	Paging, scratch files	Reduces network load over diskless case	Higher cost due to large number of disks needed
	Paging, scratch files, binaries	Reduces network load even more	Higher cost; additional complexity of updating the binaries
	Paging, scratch files, binaries, file caching	Still lower network load; reduces load on file servers as well	Higher cost; cache consistency problems
	Full local file system	Hardly any network load; eliminates need for file servers	Loss of transparency

Fig. 4-11. Disk usage on workstations.

active process at once. Again, this is an inefficient use of resources, but as CPUs get cheaper nd cheaper as the technology improves, wasting them will become less of a sin.

4.2.2. Using Idle Workstations

The second problem, idle workstations, has been the subject of considerable research, primarily because many universities have a substantial number of personal workstations, some of which are idle (an idle workstation is the devil's playground?). Measurements show that even at peak periods in the middle of the day, often as many as 30 percent of the workstations are idle at any given moment. In the evening, even more are idle. A variety of schemes have been proposed for using idle or otherwise underutilized workstations (Litzkow et al., 1988; Nichols, 1987; and Theimer et al., 1985). We will describe the general principles behind this work in this section.

The earliest attempt to allow idle workstations to be utilized was the *rsh* program that comes with Berkeley UNIX. This program is called by

```
rsh machine command
```

in which the first argument names a machine and the second names a command to run on it. What *rsh* does is run the specified command on the specified machine. Although widely used, this program has several serious flaws. First, the user must tell which machine to use, putting the full burden of keeping track of idle machines on the user. Second, the program executes in the environment of the remote machine, which is usually different from the local environment. Finally, if someone should log into an idle machine on which a remote process is running, the process continues to run and the newly logged-in user either has to accept the lower performance or find another machine.

The research on idle workstations has centered on solving these problems. The key issues are:

1. How is an idle workstation found?
2. How can a remote process be run transparently?
3. What happens if the machine's owner comes back?

Let us consider these three issues, one at a time.

How is an idle workstation found? To start with, what is an idle workstation? At first glance, it might appear that a workstation with no one logged in at the console is an idle workstation, but with modern computer systems things are not always that simple. In many systems, even with no one logged in there may be dozens of processes running, such as clock daemons, mail daemons, news daemons, and all manner of other daemons. On the other hand, a user who logs in when arriving at his desk in the morning, but otherwise does not touch the computer for hours, hardly puts any additional load on it. Different systems make different decisions as to what "idle" means, but typically, if no one has touched the keyboard or mouse for several minutes and no user-initiated processes are running, the workstation can be said to be idle. Consequently, there may be substantial differences in load between one idle workstation and another, due, for example, to the volume of mail coming into the first one but not the second.

The algorithms used to locate idle workstations can be divided into two categories: server driven and client driven. In the former, when a workstation goes idle, and thus becomes a potential compute server, it announces its availability. It can do this by entering its name, network address, and properties in a registry file (or data base), for example. Later, when a user wants to execute a command on an idle workstation, he types something like

```
remote command
```

and the *remote* program looks in the registry to find a suitable idle workstation. For reliability reasons, it is also possible to have multiple copies of the registry.

An alternative way for the newly idle workstation to announce the fact that it has become unemployed is to put a broadcast message onto the network. All other workstations then record this fact. In effect, each machine maintains its own private copy of the registry. The advantage of doing it this way is less overhead in finding an idle workstation and greater redundancy. The disadvantage is requiring all machines to do the work of maintaining the registry.

Whether there is one registry or many, there is a potential danger of race conditions occurring. If two users invoke the *remote* command simultaneously, and both of them discover that the same machine is idle, they may both try to start up processes there at the same time. To detect and avoid this situation, the *remote* program can check with the idle workstation, which, if still free, removes itself from the registry and gives the go-ahead sign. At this point, the caller can send over its environment and start the remote process, as shown in Fig. 4-12.

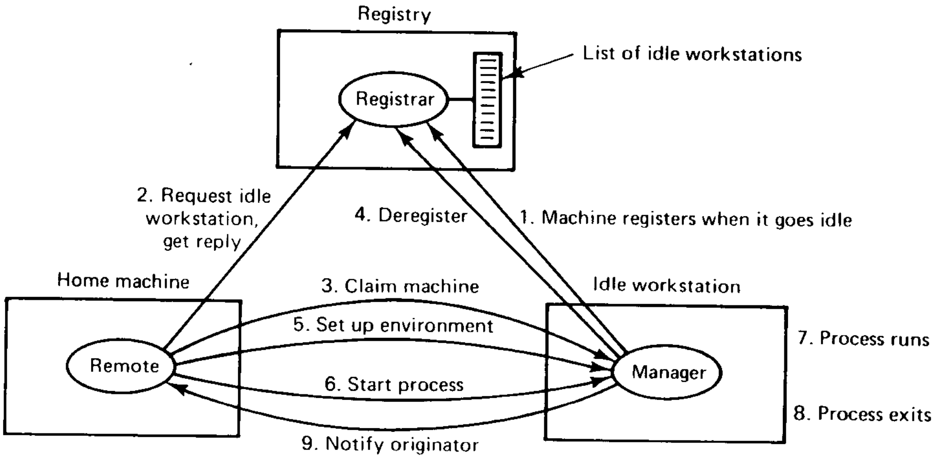

Fig. 4-12. A registry-based algorithm for finding and using idle workstations.

The other way to locate idle workstations is to use a client-driven approach. When *remote* is invoked, it broadcasts a request saying what program it wants to run, how much memory it needs, whether or not floating point is needed, and so on. These details are not needed if all the workstations are identical, but if the system is heterogeneous and not every program can run on every workstation, they are essential. When the replies come back, *remote* picks one and sets it up. One nice twist is to have "idle" workstations delay their responses slightly, with the delay being proportional to the current load. In this way, the reply from the least heavily loaded machine will come back first and be selected.

Finding a workstation is only the first step. Now the process has to be run there. Moving the code is easy. The trick is to set up the remote process so that it sees the same environment it would have locally, on the **home workstation**, and thus carries out the same computation it would have locally.

To start with, it needs the same view of the file system, the same working directory, and the same environment variables (shell variables), if any. After these have been set up, the program can begin running. The trouble starts when the first system call, say a READ, is executed. What should the kernel do? The answer depends very much on the system architecture. If the system is diskless, with all the files located on file servers, the kernel can just send the request to the appropriate file server, the same way the home machine would have done had the process been running there. On the other hand, if the system has local disks, each with a complete file system, the request has to be forwarded back to the home machine for execution.

Some system calls must be forwarded back to the home machine no matter what, even if all the machines are diskless. For example, reads from the keyboard and writes to the screen can never be carried out on the remote machine. However, other system calls must be done remotely under all conditions. For example, the UNIX system calls SBRK (adjust the size of the data segment), NICE (set CPU scheduling priority), and PROFIL (enable profiling of the program counter) cannot be executed on the home machine. In addition, all system calls that query the state of the machine have to be done on the machine on which the process is actually running. These include asking for the machine's name and network address, asking how much free memory it has, and so on.

System calls involving time are a problem because the clocks on different machines may not be synchronized. In Chap. 3, we saw how hard it is to achieve synchronization. Using the time on the remote machine may cause programs that depend on time, like *make*, to give incorrect results. Forwarding all time-related calls back to the home machine, however, introduces delay, which also causes problems with time.

To complicate matters further, certain special cases of calls which normally might have to be forwarded back, such as creating and writing to a temporary file, can be done much more efficiently on the remote machine. In addition, mouse tracking and signal propagation have to be thought out carefully as well. Programs that write directly to hardware devices, such as the screen's frame buffer, diskette, or magnetic tape, cannot be run remotely at all. All in all, making programs run on remote machines as though they were running on their home machines is possible, but it is a complex and tricky business.

The final question on our original list is what to do if the machine's owner comes back (i.e., somebody logs in or a previously inactive user touches the keyboard or mouse). The easiest thing is to do nothing, but this tends to defeat the idea of "personal" workstations. If other people can run programs on your workstation at the same time that you are trying to use it, there goes your guaranteed response.

Another possibility is to kill off the intruding process. The simplest way is to do this abruptly and without warning. The disadvantage of this strategy is that all work will be lost and the file system may be left in a chaotic state. A better way is to give the process fair warning, by sending it a signal to allow it to detect impending doom, and shut down gracefully (write edit buffers to the disk, close files, and so on). If it has not exited within a few seconds, it is then terminated. Of course, the program must be written to expect and handle this signal, something most existing programs definitely are not.

A completely different approach is to migrate the process to another machine, either back to the home machine or to yet another idle workstation. Migration is rarely done in practice because the actual mechanism is complicated. The hard part is not moving the user code and data, but finding and gathering up all the kernel data structures relating to the process that is leaving. For example, it may have open files, running timers, queued incoming messages, and other bits and pieces of information scattered around the kernel. These must all be carefully removed from the source machine and successfully reinstalled on the destination machine. There are no theoretical problems here, but the practical engineering difficulties are substantial. For more information, see (Artsy and Finkel, 1989; Douglis and Ousterhout, 1991; and Zayas, 1987).

In both cases, when the process is gone, it should leave the machine in the same state in which it found it, to avoid disturbing the owner. Among other items, this requirement means that not only must the process go, but also all its children and their children. In addition, mailboxes, network connections, and other system-wide data structures must be deleted, and some provision must be made to ignore RPC replies and other messages that arrive for the process after it is gone. If there is a local disk, temporary files must be deleted, and if possible, any files that had to be removed from its cache restored.

4.2.3. The Processor Pool Model

Although using idle workstations adds a little computing power to the system, it does not address a more fundamental issue: What happens when it is feasible to provide 10 or 100 times as many CPUs as there are active users? One solution, as we saw, is to give everyone a personal multiprocessor. However this is a somewhat inefficient design.

An alternative approach is to construct a **processor pool**, a rack full of CPUs in the machine room, which can be dynamically allocated to users on demand. The processor pool approach is illustrated in Fig. 4-13. Instead of giving users personal workstations, in this model they are given high-performance graphics terminals, such as X terminals (although small workstations can also be used as terminals). This idea is based on the observation that what many users really want is a high-quality graphical interface and good performance. Conceptually, it is much closer to traditional timesharing than to the personal computer model, although it is built with modern technology (low-cost microprocessors).

The motivation for the processor pool idea comes from taking the diskless workstation idea a step further. If the file system can be centralized in a small

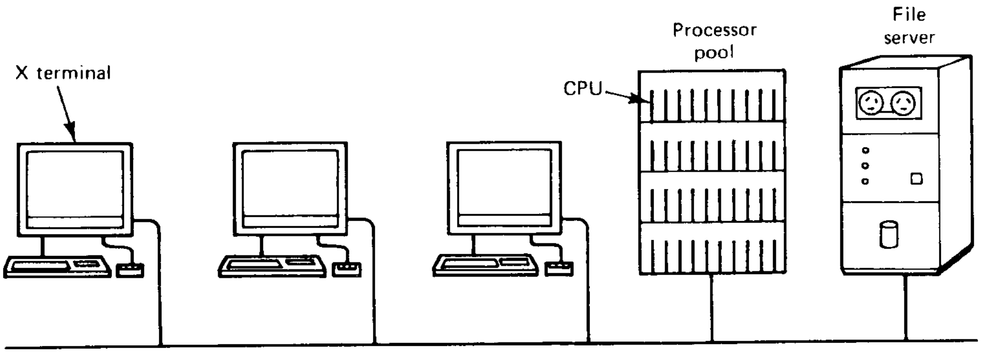

Fig. 4-13. A system based on the processor pool model.

number of file servers to gain economies of scale, it should be possible to do the same thing for compute servers. By putting all the CPUs in a big rack in the machine room, power supply and other packaging costs can be reduced, giving more computing power for a given amount of money. Furthermore, it permits the use of cheaper X terminals (or even ordinary ASCII terminals), and decouples the number of users from the number of workstations. The model also allows for easy incremental growth. If the computing load increases by 10 percent, you can just buy 10 percent more processors and put them in the pool.

In effect, we are converting all the computing power into "idle workstations" that can be accessed dynamically. Users can be assigned as many CPUs as they need for short periods, after which they are returned to the pool so that other users can have them. There is no concept of ownership here: all the processors belong equally to everyone.

The biggest argument for centralizing the computing power in a processor pool comes from queueing theory. A queueing system is a situation in which users generate random requests for work from a server. When the server is busy, the users queue for service and are processed in turn. Common examples of queueing systems are bakeries, airport check-in counters, supermarket check-out counters, and numerous others. The bare basics are depicted in Fig. 4-14.

Queueing systems are useful because it is possible to model them analytically. Let us call the total input rate λ requests per second, from all the users combined. Let us call the rate at which the server can process requests μ. For stable operation, we must have $\mu > \lambda$. If the server can handle 100 requests/sec, but the users continuously generate 110 requests/sec, the queue will grow without bound. (Small intervals in which the input rate exceeds the service rate are acceptable, provided that the mean input rate is lower than the service rate and there is enough buffer space.)

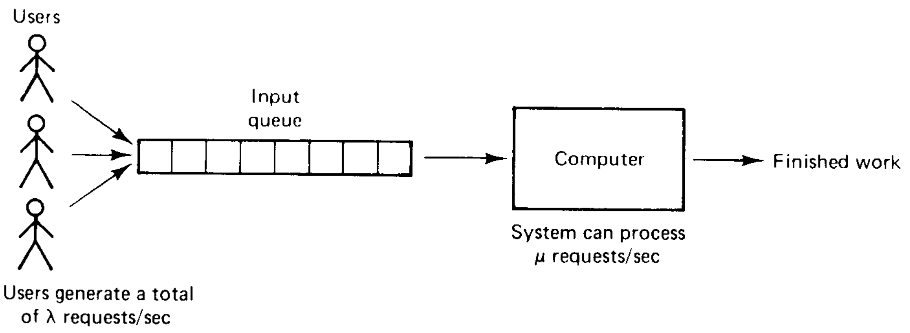
Users
Input queue
Computer
Finished work
System can process μ requests/sec
Users generate a total of λ requests/sec

Fig. 4-14. A basic queueing system.

It can be proven (Kleinrock, 1974) that the mean time between issuing a request and getting a complete response, T, is related to λ and μ by the formula

$$T = \frac{1}{\mu - \lambda}$$

As an example, consider a file server that is capable of handling as many as 50 requests/sec but which only gets 40 requests/sec. The mean response time will be 1/10 sec or 100 msec. Note that when λ goes to 0 (no load), the response time of the file server does not go to 0, but to 1/50 sec or 20 msec. The reason is obvious once it is pointed out. If the file server can process only 50 requests/sec, it must take 20 msec to process a single request, even in the absence of any competition, so the response time, which includes the processing time, can never go below 20 msec.

Suppose that we have n personal multiprocessors, each with some number of CPUs, and each one forms a separate queueing system with request arrival rate λ and CPU processing rate μ. The mean response time, T, will be as given above. Now consider what happens if we scoop up all the CPUs and place them in a single processor pool. Instead of having n small queueing systems running in parallel, we now have one large one, with an input rate $n\lambda$ and a service rate $n\mu$. Let us call the mean response time of this combined system T_1. From the formula above we find

$$T_1 = \frac{1}{n\mu - n\lambda} = T/n$$

This surprising result says that by replacing n small resources by one big one that is n times more powerful, we can reduce the average response time n-fold.

This result is extremely general and applies to a large variety of systems. It is one of the main reasons that airlines prefer to fly a 300-seat 747 once every 5 hours to a 10-seat business jet every 10 minutes. The effect arises because dividing the processing power into small servers (e.g., personal workstations), each with one user, is a poor match to a workload of randomly arriving requests. Much of the time, a few servers are busy, even overloaded, but most are idle. It is this wasted time that is eliminated in the processor pool model, and the reason why it gives better overall performance. The concept of using idle workstations is a weak attempt at recapturing the wasted cycles, but it is complicated and has many problems, as we have seen.

In fact, this queueing theory result is one of the main arguments against having distributed systems at all. Given a choice between one centralized 1000-MIPS CPU and 100 private, dedicated, 10-MIPS CPUs, the mean response time of the former will be 100 times better, because no cycles are ever wasted. The machine goes idle only when no user has any work to do. This fact argues in favor of concentrating the computing power as much as possible.

However, mean response time is not everything. There are also arguments in favor of distributed computing, such as cost. If a single 1000-MIPS CPU is much more expensive than 100 10-MIPS CPUs, the price/performance ratio of the latter may be much better. It may not even be possible to build such a large machine at any price. Reliability and fault tolerance are also factors.

Also, personal workstations have a uniform response, independent of what other people are doing (except when the network or file servers are jammed). For some users, a low variance in response time may be perceived as more important than the mean response time itself. Consider, for example, editing on a private workstation on which asking for the next page to be displayed always takes 500 msec. Now consider editing on a large, centralized, shared computer on which asking for the next page takes 5 msec 95 percent of the time and 5 sec one time in 20. Even though the mean here is twice as good as on the workstation, the users may consider the performance intolerable. On the other hand, to the user with a huge simulation to run, the big computer may win hands down.

So far we have tacitly assumed that a pool of *n* processors is effectively the same thing as a single processor that is *n* times as fast as a single processor. In reality, this assumption is justified only if all requests can be split up in such a way as to allow them to run on all the processors in parallel. If a job can be split into, say, only 5 parts, then the processor pool model has an effective service time only 5 times better than that of a single processor, not *n* times better.

Still, the processor pool model is a much cleaner way of getting extra computing power than looking around for idle workstations and sneaking over there while nobody is looking. By starting out with the assumption that no processor belongs to anyone, we get a design based on the concept of requesting machines from the pool, using them, and putting them back when done. There is also no need to forward anything back to a "home" machine because there are none.

There is also no danger of the owner coming back, because there are no owners.

In the end, it all comes down to the nature of the workload. If all people are doing is simple editing and occasionally sending an electronic mail message or two, having a personal workstation is probably enough. If, on the other hand, the users are engaged in a large software development project, frequently running *make* on large directories, or are trying to invert massive sparse matrices, or do major simulations or run big artificial intelligence or VLSI routing programs, constantly hunting for substantial numbers of idle workstations will be no fun at all. In all these situations, the processor pool idea is fundamentally much simpler and more attractive.

4.2.4. A Hybrid Model

A possible compromise is to provide each user with a personal workstation and to have a processor pool in addition. Although this solution is more expensive than either a pure workstation model or a pure processor pool model, it combines the advantages of both of the others.

Interactive work can be done on workstations, giving guaranteed response. Idle workstations, however, are not utilized, making for a simpler system design. They are just left unused. Instead, all noninteractive processes run on the processor pool, as does all heavy computing in general. This model provides fast interactive response, an efficient use of resources, and a simple design.

4.3. PROCESSOR ALLOCATION

By definition, a distributed system consists of multiple processors. These may be organized as a collection of personal workstations, a public processor pool, or some hybrid form. In all cases, some algorithm is needed for deciding which process should be run on which machine. For the workstation model, the question is when to run a process locally and when to look for an idle workstation. For the processor pool model, a decision must be made for every new process. In this section we will study the algorithms used to determine which process is assigned to which processor. We will follow tradition and refer to this subject as "processor allocation" rather than "process allocation," although a good case can be made for the latter.

4.3.1. Allocation Models

Before looking at specific algorithms, or even at design principles, it is worthwhile saying something about the underlying model, assumptions, and goals of the work on processor allocation. Nearly all work in this area assumes that all the machines are identical, or at least code-compatible, differing at most by speed. An occasional paper assumes that the system consists of several disjoint processor pools, each of which is homogeneous. These assumptions are usually valid, and make the problem much simpler, but leave unanswered for the time being such questions as whether a command to start up the text formatter should be started up on a 486, SPARC, or MIPS CPU, assuming that binaries for all of them are available.

Almost all published models assume that the system is fully interconnected, that is, every processor can communicate with every other processor. We will assume this as well. This assumption does not mean that every machine has a wire to every other machine, just that transport connections can be established between every pair. That messages may have to be routed hop by hop over a sequence of machines is of interest only to the lower layers. Some networks support broadcasting or multicasting, and some algorithms use these facilities.

New work is generated when a running process decides to fork or otherwise create a subprocess. In some cases the forking process is the command interpreter (shell) that is starting up a new job in response to a command from the user. In others, a user process itself creates one or more children, for example, in order to gain performance by having all the children run in parallel.

Processor allocation strategies can be divided into two broad classes. In the first, which we shall call **nonmigratory**, when a process is created, a decision is made about where to put it. Once placed on a machine, the process stays there until it terminates. It may not move, no matter how badly overloaded its machine becomes and no matter how many other machines are idle. In contrast, with **migratory** allocation algorithms, a process can be moved even if it has already started execution. While migratory strategies allow better load balancing, they are more complex and have a major impact on system design.

Implicit in an algorithm that assigns processes to processors is that we are trying to optimize something. If this were not the case, we could just make the assignments at random or in numerical order. Precisely what it is that is being optimized, however, varies from one system to another. One possible goal is to maximize **CPU utilization**, that is, maximize the number of CPU cycles actually executed on behalf of user jobs per hour of real time. Maximizing CPU utilization is another way of saying that CPU idle time is to be avoided at all costs. When in doubt, make sure that every CPU has something to do.

Another worthy objective is minimizing mean **response time**. Consider, for example, the two processors and two processes of Fig. 4-15. Processor 1 runs at 10 MIPS; processor 2 runs at 100 MIPS, but has a waiting list of backlogged processes that will take 5 sec to finish off. Process *A* has 100 million instructions and process *B* has 300 million. The response times for each process on each processor (including the wait time) are shown in the figure. If we assign *A* to processor 1 and *B* to processor 2, the mean response time will be $(10+8)/2$

or 9 sec. If we assign them the other way around, the mean response time will be (30 + 6)/2 or 18 sec. Clearly, the former is a better assignment in terms of minimizing mean response time.

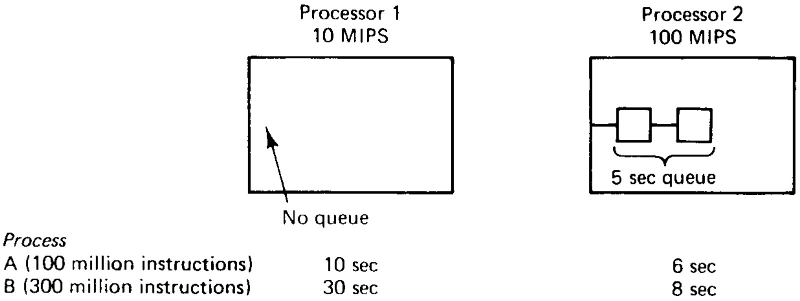

Fig. 4-15. Response times of two processes on two processors.

A variation of minimizing the response time is minimizing the **response ratio**. The response ratio is defined as the amount of time it takes to run a process on some machine, divided by how long it would take on some unloaded benchmark processor. For many users, response ratio is a more useful metric than response time since it takes into account the fact that big jobs are supposed to take longer than small ones. To see this point, which is better, a 1-sec job that takes 5 sec or a 1-min job that takes 70 sec? Using response time, the former is better, but using response ratio, the latter is much better because $5/1 \gg 70/60$.

4.3.2. Design Issues for Processor Allocation Algorithms

A large number of processor allocation algorithms have been proposed over the years. In this section we will look at some of the key choices involved in these algorithms and point out the various trade-offs. The major decisions the designers must make can be summed up in five issues:

1. Deterministic versus heuristic algorithms.
2. Centralized versus distributed algorithms.
3. Optimal versus suboptimal algorithms.
4. Local versus global algorithms.
5. Sender-initiated versus receiver-initiated algorithms.

Other decisions also come up, but these are the main ones that have been studied extensively in the literature. Let us look at each of these in turn.