TIME-SHARING SYSTEMS

A time-sharing system is a computer operating system that permits multiple programs to appear to run simultaneously. Time-sharing systems are also called multiprogrammed or multitasking systems. The programs are often called tasks or processes; a process is more formally defined as the active execution state of a program. In what follows, we first step through the fundamental requirements and workings of a time-sharing system. These requirements reveal time-sharing systems to be primarily resource-sharing systems, where resources are the physical components of the machine: disk, central processing unit (CPU), memory, and so on. We then show how a time-sharing system manages, or schedules, the resources to improve system throughput and efficiency. Several state-of-the-art scheduling algorithms are described, and

we conclude with a review of a recently proposed scheduling algorithm still in the research stage.

The first computer systems did not have time-sharing capabilities; in fact, they did not even have operating systems. In these early systems, each program was loaded from punch cards or disk and would run to completion. The next program was then loaded and run, and so on. This one-at-a-time type of scheduling is called a batch system. Early programs were also difficult to code because each program was responsible for doing its own input and output (I/O). I/O device programming is tedious and error-prone, and the basic functionality is similar across many programs. The first operating system was born when programmers decided to write a single set of I/O handlers and place them in memory where all programs could access them.

Early batch systems were soon found to be inefficient because of the significant time wasted between when one program finished and the next one could be started. The next step in operating system development was to make it autonomous, so that the system could automatically load the next program as the current one finished. This helped, and system developers soon turned their attention to reducing response time, the time each user has to wait from when a job is submitted until it completes. The first systems used simple first come first served policies, which are straight forward and fair, but it was seen that such policies as shortest job first could reduce the average response time, albeit at the expense of higher variance in completion time (especially for long running jobs).

The breakthrough that formed the basis for today's time-sharing systems was a technique called preemptive scheduling. The problem with nonpreemptive scheduling is that once control is given to a program, it monopolizes the system until it is finished or voluntarily relinquishes control. A preemptive scheduler divides time into equal sized timeslices, or quantums. The scheduler gives control to a program (i.e., dispatches the process) and sets a timer to interrupt the system after one quantum. When the quantum expires, the operating system regains control as a result of the timer interrupt. The system can now save the state of the interrupted program and then reload and restart a different program. We say the currently executing process is preempted by the quantum expiry, and the system switches context to the next ready process. It is important to pick a good timeslice value: too short a quantum will lead to many context switches, which involve the overhead of saving and restoring process state; too long a quantum will reduce the inefficiency due to context switching but will make the system appear sluggish and slow to respond.

Figure 1 illustrates how time-sharing can both reduce response time and increase resource utilization. Figure 1(a) shows the execution timeline of three tasks: process 1 is CPU bound, which means it spends all its time computing; process 2 does some computing and some disk I/O; and process 3 is I/O bound, since it spends most of its time waiting for network I/O to complete. Clearly, using batch or nonpreemptive scheduling will not yield maximum CPU utilization for this workload mix, since processes 2 and 3 spend significant amounts of time waiting for other resources to service their requests. Figure 1(b) shows the execution timeline when the three jobs are run on a time-sharing system. The timeline is structured so that it shows the utilization of the three different resources (the CPU, disk, and network) over time. When process 3 is started first, it computes for only a short time before it initiates a network request. At this point, the operating system dispatches the operation to the network device, and then context switches to process 2. Now, both the CPU and network are busy, and work is accomplished on behalf of both processes. When process 2 initiates the disk request, the operating system starts the operation and context switches to process 1. Figure 1(b) shows that at this time, all three resources are busy, each on behalf of a different process. Since process 1 is compute bound, it expires its quantum and returns control to the operating system. By this time, both the network and disk operations have completed, so both process 2 and process 3 are ready for execution.

The key point of the figure is that the CPU is always kept busy, and that overall utilization is increased by keeping multiple resources busy simultaneously. As a result, average response time is reduced because the time to complete all three jobs is less than if nonpreemptive scheduling were used.

RESOURCE MANAGEMENT

We have seen that one of the primary goals of a time-sharing system is to achieve low response times and high throughput

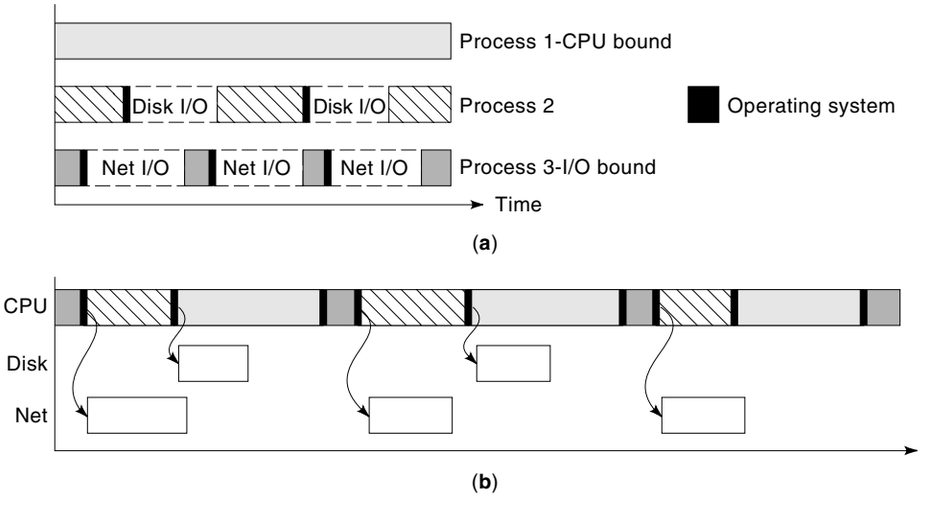

Figure 1. (a) The execution timeline of three processes: process 1 spends all its time computing; process 2 computes for a while, then calls the operating system to do some disk I/O; process 3 spends most of its time doing network I/O. (b) Resource utilization of the CPU (top), disk (center), and network (bottom) over time. The timeline assumes a time-sharing system schedules the three processes of part (a) in round-robin order.

when multiple programs are simultaneously active. This goal is accomplished primarily through efficient management of the physical resources of the computer. When multiple programs are active concurrently, they may all have simultaneous demands on the physical resource. This means that the resources must be shared between the processes, which in turn means that accesses to the resources must be managed or scheduled in some way. In Fig. 1, we saw that the CPU resource was shared between the processes using what appeared to be a round-robin protocol. Of course, real schedulers are much more complex, and there are other resources like memory and disk that must be taken into account as well.

A scheduling algorithm has several objectives: it must be fair; it should minimize response times; it should maximize utilization; and it should have low overhead. Meeting all these objectives simultaneously is a challenging task indeed. In this section, we show typical resource management techniques for three classes of resources: CPU, memory, and disk.

Few operating systems schedule resources at the physical level directly. Instead, they abstract the properties of the resource into a data structure which is used for management purposes. In particular, processes are the logical abstraction of a CPU; virtual memory is used to abstract physical memory, and files are the abstract representation of a disk. These abstractions are important because they give the operating system independence from the details of the physical devices, which makes the system more portable to different architectures and more flexible in the face of differing device parameters. For our purposes, the job of the scheduler is to map abstractions back to their physical counterpart in a way that meets the goals of the system. Thus, processes are mapped to execution timeslices, virtual memory is mapped to real memory pages, and files are mapped to disk blocks. In what follows, we shall see how this mapping is achieved.

Processor Scheduling

Every runnable thread of execution is represented by a process in the operating system. Each process has a process descriptor that holds the state of the process, including (among other things) its priority, whether it is waiting for I/O or not, and its saved state (e.g., its register set) if it is inactive for some reason. Those processes that are ready to run are placed on a ready queue, which is a list of process descriptors usually kept sorted in priority order. The currently active process, the one actually using the CPU, is simply the process at the head of the ready queue. The CPU is never running without some process attached to it—even an idle system is actively running the idle process, which is typically the lowest priority task in the system.

The scheduler comes into play whenever a decision must be made about which process is to be run next. Scheduling decisions must be made in several situations: the quantum of the currently active process expires, a process which was not runnable because it was waiting for an external event becomes runnable, or a new process is created.

We will describe a process scheduling algorithm called multilevel feedback queuing, which is by far the most common algorithm in use today (1). The algorithm employs multiple run-queues, where each queue holds processes that fall in a particular priority range. The process priorities are split into two fields: a fixed base component (P_{base}) and a dynamically changing component (P_{cpu}). Thus, at any time the effective priority of a process is $P_{eff} = P_{base} + P_{cpu}$. Processes can never have a higher priority than their base component, but their dynamic component is updated every timeslice to reflect their CPU usage pattern.

Processes in a particular run queue are scheduled in round-robin, first in, first out (FIFO) order, and processes in a lower priority run queue are only dispatched for execution if no higher priority processes are ready. Every time a process expires its quantum, it moves to the next lowest run queue until it reaches the last, or lowest priority, run queue. Processes in this run queue are typically long running and heavily CPU bound because of the nature of the multiple levels of queues.

The feedback part of the algorithm allows processes to move back up the queue levels as the process dynamics change. To see how the feedback works, we must look closer at how the dynamic priority is set. In a multilevel feedback queuing system, quantums are actually multiple timeslices. Each time the clock ticks (usually about once every 10 ms), all the dynamic priorities are adjusted. The dynamic priority of the actively running process is degraded to reflect that it has consumed more CPU time, and the dynamic priority of all the waiting processes is increased to reflect that they are now more deserving of a chance to run. The dynamic priority of these other processes is usually increased logarithmically as $P_{cpu} = P_{cpu}/2$, so that the longer a process waits to run, the closer it gets to its base priority. This property allows processes to move back up the levels of run queues.

Multilevel feedback queues have the desirable tendency of favoring I/O bound or interactive processes, while still preventing CPU bound jobs from starving. Consider again the three processes of Fig. 1(a). Process 1 is CPU bound and will quickly move to the lower queue levels as it continually expires its quantum. Process 3 is I/O bound, and as a result, its dynamic priority is degraded only slightly before it yields the CPU to do I/O. As the process waits for the I/O to complete, its dynamic priority is increased so that it will likely be scheduled immediately once it is ready. The rationale here is that it makes sense to favor I/O bound jobs because there is a high probability that they will soon yield the processor to do more I/O. Process 2 is intermediate in its CPU usage relative to the other two jobs, and so will likely reside in the middle queue levels. Process 2 will be favored when Process 3 yields the CPU unless the CPU bound process has waited long enough for its dynamic priority to decay to a higher effective priority.

Multilevel feedback queues do require some tuning in the way that time-slices are chosen and in the way that dynamic priorities are decayed. If the timeslice or decay factor is too slow, the system will not be responsive to changes in workload; if the timeslice is too short, then context switch overhead increases; and if the decay is too fast, then the differentiation between CPU utilizations is lost. Still, the strengths of this scheduling policy make it one of the most popular in modern time-sharing systems.

Memory Management

To run, a program must have the code it is actively executing and the data it is actively accessing resident in memory. When multiple programs are running simultaneously, memory quickly becomes a scarce resource that must be carefully managed. Early time-sharing systems used a technique called swapping, which loads and unloads the entire memory image

of a program from disk on each context switch. Although swapping may be appropriate for nonpreemptive batch systems, the relatively high latency of disk accesses has the effect of making a context switch very expensive. Devoting all of the physical memory to a single program is simple to implement but has the disadvantage of being slow and of limiting the size of the program to the size of real memory.

In the 1980s, Denning (1) developed the philosophy of working sets, which recognize that a running program only actively accesses a subset of its total code and data space at any given time. For example, a program loop that inverts a matrix will spend a significant period of time accessing the instructions that form the loop and the elements of the array. The loop and the matrix form the working set of the program while the algorithm is executed. We say the fragment exhibits a high degree of both spatial and temporal locality. Spatial locality expresses the probability that memory near a particular datum will be accessed again in the near future. In this example, the next instruction in the loop or the next element of the array are likely to be accessed. Temporal locality expresses the probability that the same datum will be accessed again in the near future. In the example, the instructions of the loop are accessed repeatedly over time. Denning realized that only the working set of a program needs to be resident in memory for the program to run efficiently. Of course, a program's working set will change over time, but at any particular instance, the size of the working set is likely to be much smaller than the entire program.

Working sets were first exploited in a memory management technique called overlays. This technique divides memory into chunks that can hold either code or data, so that each overlay holds some subset of the program's total storage. At run time, only the overlays that form the working set need to be resident in memory. Overlays allow a time-sharing system to have parts of multiple programs simultaneously resident, which reduces context switch overhead. The overlays can be swapped in and out as the working sets of the currently active programs change. Also, the total space requirement of a program can be larger than the size of physical memory, provided the overlays that form its largest working set can still reside in memory. The difficulty with overlays is that they are typically not transparent to the programmer, who may spend many tedious hours explicitly placing subroutines and data onto different overlays so that the program runs efficiently.

Most modern time-sharing systems use a technique called demand paged virtual memory. Virtual memory allows each application to access memory as if it had its own private address space. The relationship between these virtual memory addresses and their physical counterparts are kept in a set of translation tables maintained by the operating system. To keep the number of translation entries manageable, each entry typically applies to a range of addresses called a page (usually a power of two between 256 bytes and 8 kbytes). Thus, a virtual address space can be viewed as a contiguous array of pages, where each virtual page maps, through the translation entry, to a particular page of physical memory. Note that contiguous virtual pages do not have to map to contiguous physical pages. The translation entries are usually grouped into data structures called page tables, which are, themselves, stored in memory.

In a demand paged system, the page tables are initially empty, which means no virtual to physical mapping exists. As each virtual address is referenced, the page tables are searched to find the mapping; if none exists, a free page of physical memory is allocated to hold the data, and the translation entry is entered into the page tables. This is called a page miss and usually results in the data for the corresponding virtual page being brought in, on demand, from disk. Once the mapping is established, future references to the virtual page can be translated to their physical counterpart. Because each reference to every virtual address must be translated, most systems keep a fast Translation Lookaside Buffer that caches recently accessed translation entries.

With respect to resource management, the physical memory pages are usually managed as a cache of disk blocks. Because of the properties of spatial and temporal locality, a recently referenced page is likely to be referenced again in the near future. Thus, if all the real memory pages are in use and a page miss requires that a new one be allocated, many systems pick the least recently used (LRU) physical page as the victim to be ejected. This is known as an approximate LRU replacement policy (4). As with overlays, each active program only requires the pages that form its working set to be resident in memory. In practice, a typical system can comfortably accommodate the working sets of several programs without excessive paging. Unlike overlays, hardware support has made virtual memory transparent to the programmer and provides access protection, so that programs cannot modify real memory unless the translation entries explicitly permit it. These features can be used to manage memory even more efficiently. For example, shared libraries allow different virtual address spaces to map to common physical pages, thus reducing overall memory requirements.

Disk Scheduling

Because disks (or CDs) have moving parts, the latency to access data can be very high. To access a random block of data on one of the these devices, the arm must first be positioned over the correct track. This is called a seek, and can be several milliseconds. Once the arm is positioned, the disk controller must wait until the block passes under the read/write head. This waiting time is called the rotational latency and is determined by how fast the disk is spinning. Finally, the block is transferred to/from memory, but this transfer rate is also limited by the speed at which the disk spins.

Classically, two orthogonal approaches have been used to reduce latency and improve throughput. The first approach is to place or cluster blocks on the disk so that the transfer rate is maximized. This placement is achieved by taking the rotational latency into account when positioning logically adjacent blocks so that they appear under the read/write heads with little or no delay. For files that span multiple tracks or cylinders, the placement algorithm can choose adjacent tracks to minimize arm movement. Clearly, these placement techniques are applicable for single files and can do little to improve performance when multiple clients simultaneously request data from different files. Still, placement techniques are effective enough that they are now directly incorporated into most disk controllers.

Disk head scheduling is the second approach used to improve performance and is most effective when there are multiple outstanding read or write requests. Although there are many variants, the general idea is to order the requests by

increasing track/cylinder number, so that the disk arm can move from lower to higher cylinder numbers in a continuous sweep. The arm can then be repositioned back to track 1, or simply reversed in its direction by servicing newly arrived requests in decreasing cylinder order. The former is called the CSCAN algorithm and is very effective in reducing average seek time. Patt (2) has showed that this algorithm is effective for modern SCSI drives that contain track caches and built-in, look-ahead controllers.

LOTTERY SCHEDULING

Lottery scheduling (3) is a novel scheduling approach for time-sharing systems. It was only recently proposed and as such, is an example of current research in operating system design. Lottery scheduling uses randomized resource allocation to apportion resource rights in a way that is probabilistically fair across multiple competing clients.

Conceptually, each resource is abstracted as a pool of lottery tickets, and clients that want to access the resource are granted a set of tickets out of the pool. Each resource allocation is determined by holding a lottery: a random ticket number is selected, and the client holding the ticket is granted the resource. Starvation is prevented because any client that holds tickets has a chance of winning the lottery. As well, the probability of winning the lottery increases with the number of tickets held. This can be used to implement fair share scheduling, where each client pays to get some guaranteed percentage of the resource. For example, if a client has paid to get 25% of the CPU time on a system, that client would receive one quarter of the CPU scheduling tickets and should win one quarter of the lotteries, on average. Fair share scheduling is often used in large time-sharing installations, where corporate clients are charged in accordance with their resource consumption.

The basic ideas in lottery scheduling can be enhanced in several ways. For example, clients requiring service from, say, a database server can transfer their tickets to the server to give it a better chance of running on their behalf. The desirable property of favoring I/O bound processes can be achieved through compensation tickets. If a process has t tickets but uses only a fraction $1/f$ of its CPU quantum, it receives $f \cdot t$ tickets in compensation until its next quantum is awarded. To see how this works, consider two processes: A is compute-bound and has 100 tickets; B is I/O bound and also has 100 tickets. Suppose that B only uses a quarter of its quantum before yielding the processor to do I/O. Without compensation tickets, process B would be awarded the CPU as often as process A but would only get one quarter of the CPU utilization—which is in violation of the 1 : 1 ticket allotment ratio. Using compensation tickets, process B is given 400 tickets to compensate for using one quarter of its quantum. Thus, process B is four times as likely to win the lottery as process A, but since it uses one fourth of its winnings, both processes get one half of the CPU.

Because lottery tickets are abstract representations, they can be used for any resource. For example, network channel bandwidth can be represented by lottery tickets, and accesses to the channel can be granted by holding lotteries. This is useful for multimedia applications that have stringent bandwidth requirements since they can obtain the tickets required to meet their scheduling constraints. Memory management can also be accommodated through lottery scheduling. If some memory has to be cleared to make room for new data, a loser lottery can be held to see whose data is evicted. A loser is selected in inverse proportion to the number of tickets held, so that the more tickets one holds, the less likely it is that one will lose the lottery.

SUMMARY

Multiprogramming is an increasingly important part of today's computing systems. Time-sharing enables interactive and compute intensive programs to progress simultaneously, giving fast response times while still maintaining high throughput overall. Good resource management is at the heart of an effective time-sharing system and must be applied to all shared components to achieve balanced utilization and avoid bottlenecks that could degrade performance.

BIBLIOGRAPHY

1. P. J. Denning, Working Sets Past and Present, *IEEE Trans. Softw. Eng.,* **SE-6**: 64–84, 1980.
2. B. L. Worthington, G. R. Ganger, and Y. N. Patt, Scheduling Algorithms for Modern Disk Drives, *Sigmetrics 94,* 1994, pp. 241–251.
3. C. A. Waldspurger and W. E. Weihl, Lottery Scheduling: Flexible Proportional-Share Resource Management, *1st Symp. Oper. Syst. Des. Implementation,* 1994, pp. 1–12.
4. S. J. Leffler et al., *The Design and Implementation of the 4.3BSD UNIX Operating System,* Reading, MA: Addison-Wesley, 1989.

RONALD C. UNRAU
University of Alberta

TIME SWITCHING. See ISDN.