When a computer is multiprogrammed, it frequently has multiple processes competing for CPU at the same time. This situation occurs whenever two or more processes are simultaneously in the ready state. If only one CPU is available, the choice is to be made which process to run next. The part of the operating system that makes the choice is called the scheduler and the algorithm it uses is called the scheduling algorithm.

**Process behavior.** Nearly all processes alternate bursts of computing with I/O requests. Typically CPU runs for a while without stopping, then a system call is made to read from a file or write to a file. When the system call completes, the CPU computes again until it needs more data or has to write more data, and so on. Some of the processes spend most of their time waiting for I/O, while others spend most of their time computing. The former are called **computer-bound**; the latter are called **I/O-bound**. Compute-bound processes typically have long CPU bursts and thus infrequent I/O waits, whereas I/O-bound processes have short CPU bursts and frequent I/O waits.

**Categories of scheduling algorithms.** The key issue is how to make scheduling decisions. Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts. A **nonpreemptive** scheduling algorithm picks a process to run and then just lets it run until it blocks (either on I/O or waiting for another process) or until it voluntarily releases the CPU. Even if it runs for hours, it will not be forcibly suspended. After clock interrupt processing completed, the process that was running before the interrupt is always resumed.

In contrast, a **preemptive** scheduling algorithm picks a process and lets it run for a maximum of some fixed time. If
it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run (if one is available).

Not surprisingly, in different environments different scheduling algorithms are needed. This situation arises because different application areas (and different types of operating systems) have different goals. Three environments worth distinguishing are

1. Batch.
2. Interactive.
3. Real time.

In batch systems, there are no users impatiently waiting at their terminals for a quick response. Consequently, nonpreemptive algorithms, or preemptive algorithms with long time periods or each process are often acceptable. This approach reduces process switches and thus improves performance.

In an environment with interactive users, preemption is important to keep one process from hogging the CPU and denying service to the others. Even if no process runs forever, due to a program bug, one process might shut out all the others indefinitely. Preemption is needed to prevent this behavior.

In systems with real time constraints, preemption is, oddly enough, sometimes needed because the processes know that they may not run for long periods of time and usually do their work and block quickly. The difference with interactive systems is that real-time systems run only programs that are intended to further application at hand.

**Scheduling in batch systems.** It is time to turn from general scheduling issues to specific scheduling algorithms. The following three parameters are important from point of view of performance and effectiveness: throughput, turnaround time, and CPU utilization. Throughput is the number of jobs per hour that the system completes. We try to increase throughput of the system. Turnaround time is the statistically average time from the moment that a batch job is submitted until the moment it is completed. It measures how long the average user has to wait for the output. Here is the rule: small is beautiful. CPU utilization shows how good CPU has been utilized.

**First-come first-served.** Probably the simplest of all scheduling algorithms is nonpreemptive first-come first-served. With this algorithm, processes are assigned the CPU in the order they request it. Basically, there is a single queue of ready processes. When the first process enters the system from the outside, it is started immediately and allowed to run as long as it wants it. As other jobs come in, they are put onto the end of the queue.
The greater strength of this algorithm is that it is easy to understand and equally easy to program. Unfortunately, first-come first-served also has a powerful disadvantage. If there is one compute-bound process that runs for 1 sec at a time and many I/O-bound processes that use little CPU time but each have to perform 1000 disk reads to complete.

Shortest job first. Let us look at another nonpreemptive batch algorithm that assumes the run times are known in advance. With several equally important jobs are sitting in the input queue waiting to be started, the scheduler picks the shortest job first.

Consider four jobs A, B, C, and D with run times of 8, 4, 4, and 4, respectively. By running them in that order, the turnaround time for A is 8, for B is 12, for C is 16 and for D is 20. The average turnaround time is 14. Now let us consider running these our jobs using shortest job first. The turnaround times are now 4, 8, 12, and 20 for an average of 11.

Shortest remaining time next. A preemptive version of shortest job first is shortest remaining time next. With this algorithm, the scheduler always chooses the process whose remaining run time is the shortest. Again here, the run time has to be known in advance. When a new job arrives, its total time compared to the current process’ remaining time. If the new job needs less time to finish than the current process, the current job is suspended and the new job started. This scheme allows new short jobs to get good service.

Scheduling in interactive systems. We will now look at some algorithms that can be used in interactive systems. All of these can also be used as the CPU scheduler in batch systems.

Round-robin scheduling. In round robin method, each process assigned a time interval, called a quantum, which it is allowed to run. If the process is still running at the end of the quantum, the CPU is preempted and given to another process. If the process has blocked or finished before the quantum has elapsed, the CPU switching is done when the process blocks, of course. Round robin is easy to implement.
All the scheduler needs to do is maintain a list of runnable processes. When the process uses up its quantum, it is put on the end of the list.

In round robin method, setting the quantum too short causes too many process switches and lowers the CPU efficiency, but setting it too long may cause poor response to short interactive requests. A quantum around 20-50 msec is often a reasonable compromise.

Priority scheduling. Round robin makes assumption that all processes are equally important. Frequently, the people who own and operate multiuser computers have different ideas on that subject. Or instance, at a university, the pecking order may be deans first, then professors, secretaries, janitors, and initially students. The need to take external factors into account leads to priority scheduling. The basic idea is straightforward: each process is assigned a priority and runnable process with the highest priority is allowed to run first.

The UNIX, system has a command, nice, which allows to a user voluntarily reduce the priority of his/her process, in order to be nice to other users. Nobody ever uses it!

Multiple queues. Giving all processes a large quantum would mean poor response time. The solution is to set up priority classes. Processes in the highest class were run for one quanta. Processes in the next class were run for two quanta. Processes in the next class were run for four quanta, and so on. Whenever a process used up all the quanta allocated to it, it was moved down one class.

As an example, consider a process that needed to compute continuously for 100 quanta. It would initially be given 1 quantum, then swapped out. Next time it would be given 2 quanta before being swapped out. On succeeding runs it would
get 2, 4, 8, 16, 32, and 64 quanta, although it would have used only 37 of the final 64 quanta to complete its work. Only 7 swaps would be needed (including the initial load) instead of 100 with a pure robin-round algorithm.

**Memory management**

The part of operating system that manages memory is called the *memory manager*. It is job to keep track of which parts of memory are not in use, to allocate memory to processes when they need it and deallocate it when they are done, and to manage swapping between main memory and disk when main memory is not big enough to hold all the processes.

**Memory management without swapping or paging.** The memory management systems can be divided into two classes: those that move processes back and forth between main memory and disk (swapping and paging), and those that do not. The second is simpler. So we will study it first.

The simplest possible memory management scheme is to have just one process in memory at a time (monoprogramming), and to allow that process to use all the memory. The user loads the entire memory with a program from disk or tape, and it takes over whole machine.

When multiprogramming is used, the CPU utilization can be improved. If the process computes only 20% 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 the five processes will never wait for I/O at the same time.

![Graph showing CPU utilization with different I/O wait times](image-url)
The better model is to look to CPU usage from probabilistic point of view. 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 processes are waiting for I/O is $p^n$. The CPU utilization then given by the formula

$$\text{CPU utilization} = 1 - p^n.$$  

The CPU utilization as a function of $n$, which is called the degree of multiprogramming, is shown below. The probabilistic model just described is an approximation. It implicitly assumes that all processes are independent, meaning that it is quite acceptable for a system with five processes in memory to have three processes running and two processes 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.

**Exercise.** A computer has enough room to hold four programs in its main memory. These programs are idle waiting for I/O half of the time. What fraction of the CPU time is wasted?

**Solution.** $n = 4, p = \frac{1}{2}, p^n = \left(\frac{1}{2}\right)^4 = 1/16.$

**Multiprogramming with fixed partitions.** It should be clear that it is better to have more than one process in memory at once. The easiest way to do this is to divide memory up into $n$ (possible unequal) partitions. When a job arrives, it can be put into the input queue for the smallest partition large enough to hold it. Since the partitions are fixed in this scheme, any space in a partition not used by a job will be lost.
The disadvantage of sorting the incoming jobs into separate queues becomes apparent when the queue for a large partition is empty but the queue for a small partition is full. An alternate organization is to maintain a single queue. Whenever a partition becomes free, the job closest to the front of the queue that fits in it could be lead into the empty partition and run. Since it is undesirable to waste a large partition on a small job, a different strategy is to search the whole input queue whenever a partition becomes free and pick the largest job that fits. It discriminates small jobs. One way out is to have one small partition around. Such a partition will allow small jobs to run without having to allocate a large partition for them. Another approach is to have a rule stating that a job that is eligible to run not be skipped over more than \( k \) times.

**Exersize.** A memory of a computer is divided into five fixed partitions of size 1200K, 800K, 400K, 200K, and 100K. The jobs of size 50K, 60K, 70K, 120K, 150K, 250K, 280K, 950K, and 1150K have been arrived in order. After all five partitions filled up, operating system sends them into CPU, that is the partitions become empty. Fill the following table to show the allocation of the jobs into partitions according to (a) separate queues model, (b) single queue model with picking up a job closest to the front of queue that fits, and (c) single queue model with picking the largest job up that fits.

**Solution.**

<table>
<thead>
<tr>
<th></th>
<th>1200K</th>
<th>800K</th>
<th>400K</th>
<th>200K</th>
<th>100K</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>950K, 1150K</td>
<td>-</td>
<td>250K, 280K</td>
<td>120K, 150K</td>
<td>50K, 60K, 70K</td>
</tr>
<tr>
<td>(b)</td>
<td>50K, 150K, 950K, 1150K</td>
<td>60K, 250K</td>
<td>70K, 280</td>
<td>120K</td>
<td></td>
</tr>
<tr>
<td>(c)</td>
<td>1150K, 950K, 280K, 120K</td>
<td>250K, 60K</td>
<td>150K, 50K</td>
<td>70K</td>
<td></td>
</tr>
</tbody>
</table>