Round robin scheduling is a preemptive version of first-come, first-serve scheduling. In round robin scheduling processes are queued in a first-in-first-out order but each process is allowed to run for only a limited amount of time. This time interval is known as a time-slice or quantum. If a process does not complete (or get blocked because of an I/O operation), the quantum expires and the process is interrupted. It is then placed at the back of the run queue where it must wait for all the processes that were already in the queue to cycle through the CPU.

Advantage:

• Every process gets an equal share of the CPU.

• It dramatically improves average response times over non-preemptive schedulers.

• By limiting each task to a certain amount of time, the operating system can ensure that it can cycle through all ready tasks.

Disadvantage:

• Giving every process an equal share of the CPU is not always a good idea. For instance, highly interactive processes will get scheduled no more frequently than CPU-bound processes.

Most systems have a large number of processes with short CPU bursts interspersed between I/O requests, e.g., a text editor, and a small number of processes with long CPU bursts, e.g., video editor. To provide good time-sharing performance, we may preempt (interrupt) a running process to let another one run. The ready list, also known as a run queue, in the operating system keeps a list of all processes that are ready to run and not blocked on input/output or another blocking system request, such as a semaphore.

The process scheduler is the component of the operating system that is responsible for deciding whether the currently running process should continue running and, if not, which process should run next. The decisions that the scheduler makes concerning the sequence and length of time that processes may run is called the scheduling algorithm (or scheduling policy). These decisions are not easy ones, as the scheduler has only a limited amount of information about the processes that are ready to run. A good scheduling algorithm should:

* Be fair – give each process a fair share of the CPU, allow each process to run in a reasonable amount of time.
* Be efficient – keep the CPU busy all the time.
* Maximize throughput – service the largest possible number of jobs in a given amount of time; minimize the amount of time users must wait for their results.
* Minimize response time – interactive users should see good performance.
* Be predictable – a given job should take about the same amount of time to run when run multiple times. This keeps users sane.
* Minimize overhead – don’t waste too many resources. Keep scheduling time and context switch time at a minimum.
* Maximize resource use – favor processes that will use underutilized resources. There are two motives for this. Most devices are slow compared to CPU operations. We’ll achieve better system throughput by keeping devices busy as often as possible. The second reason is that a process may be holding a key resource and other, possibly more important, processes cannot use it until it is released. Giving the process more CPU time may free up the resource quicker.
* Avoid indefinite postponement – every process should get a chance to run eventually.
* Enforce priorities – if the scheduler allows a process to be assigned a priority, it should be meaningful and enforced.
* Degrade gracefully – as the system becomes more heavily loaded, performance should deteriorate gradually, not abruptly.

It is clear that some of these goals are contradictory. For example, minimizing overhead means that jobs should run longer, thus hurting interactive performance. Enforcing priorities means that high-priority processes will always be favored over low-priority ones, causing indefinite postponement. These factors make scheduling a task for which there can be no perfect algorithm.

Preemptive scheduling allows the scheduler to control response times by taking the CPU away from a process that it decided has been running too long in order to let another process run. It incurs more overhead than nonpreemptive scheduling since it has to deal with the overhead of context switching processes instead of allowing a process to run to completion or run until the next I/O operation or other system call. However, it allows for higher degrees of concurrency and better interactive performance.

The scheduling algorithm has the task of figuring out whether a process should be switched out for another process and which process should get to run next. The dispatcher is the component of the scheduler that handles the mechanism of actually getting that process to run on the processor. This requires loading the saved context of the selected process, which is stored in the process control block and comprises the set of registers, stack pointer, flags (status word), and a pointer to the memory mapping (typically a pointer to the page table). Once this context is loaded, the dispatcher switches to user mode via a *return from interrupt* operation that causes the process to execute from the location that was saved on the stack at the time that the program stopped running — either via an interrupt or a system call.