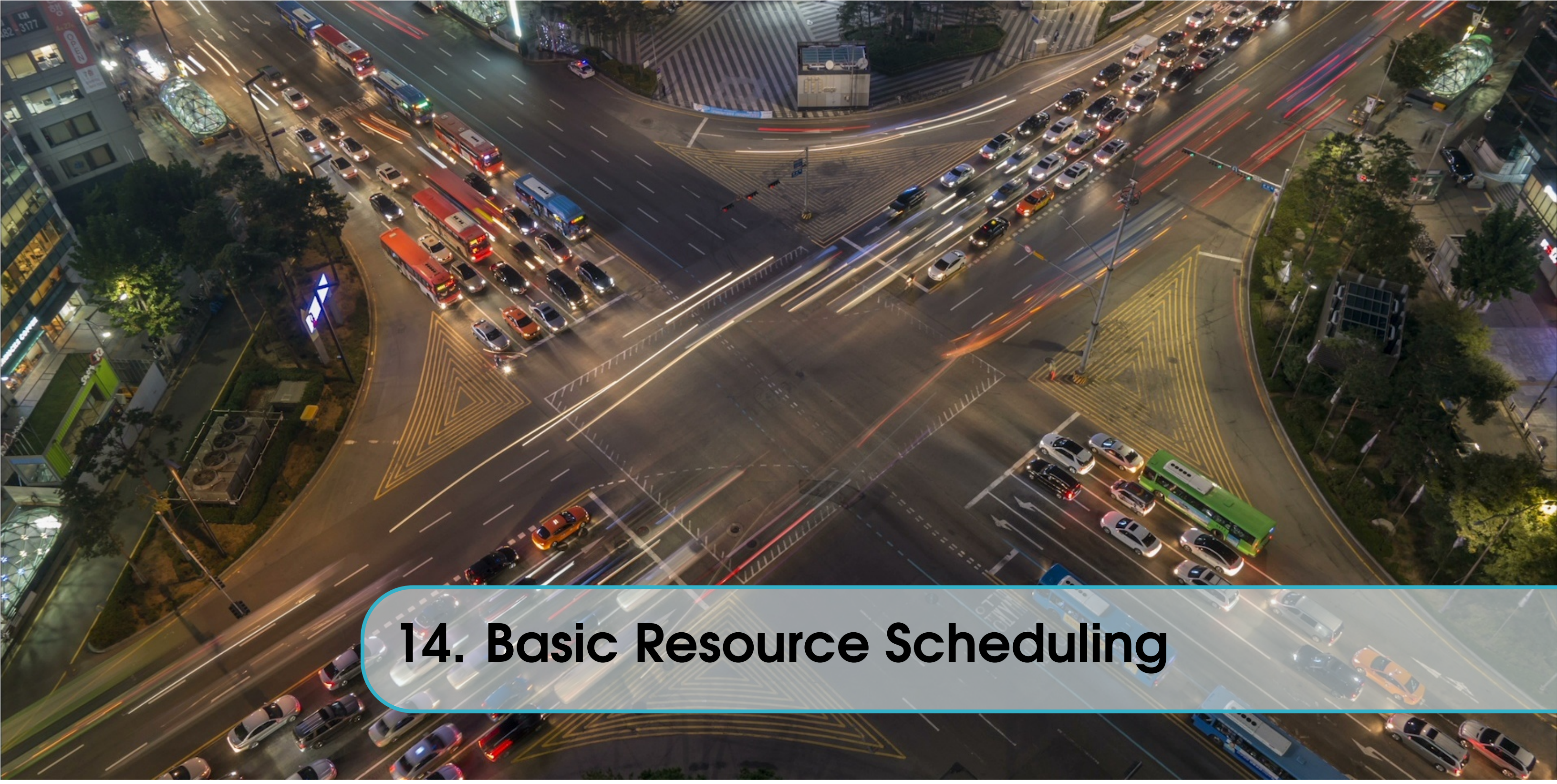

14. Basic Resource Scheduling

In this chapter we will be exploring the common denominator among strategies to perform scheduling for a single resource. We will also cover the basic metrics to evaluate the performance of a scheduler.

14.1 Average vs. Actual Case

In the previous lectures, we have analyzed the properties of a system from a statistical point of view. One of the metrics on which we extensively focused was "average behavior". Reasoning in terms of average behavior is often useful to have a rough idea on a system's performance. Average measurements, however, do not always tell the full story about how well a system is behaving.

More specifically, when we are interested in studying how well as system is behaving from a more process-centric perspective, it may be important to understand what is the service "perceived" by individual requests. In order to zoom-in onto what happens to each request in our system, we need to consider more details about how the traffic is actually routed and serviced throughout the system. Since we consider a system as a network of interconnected resources, it makes sense to talk about "resource management".

Take a web-server as an example. The moment you bring online your page its popularity is probably low, so the web-server sits in a room servicing a request every once in a while. Because the server is underutilized, it does not matter how you decide to use the various hardware resources to satisfy the few incoming requests. As the popularity of your page grows, you start receiving a large incoming traffic of requests. Now, if all the requests are valuable customer inquiries, then there is not much you can do, again. But then you notice that a lot of these requests are network ping requests. Then perhaps it makes sense to prioritize HTTP traffic to your pages rather than responding with pongs. You are now *managing* your network interface. Moreover, you notice that some requests are for free content on your webpage(s), while others are originated by paying premium customers. In this case, it makes sense to *manage* how you assign CPU/memory/network resources differently to make sure that the paying customers are satisfied with the service.

To summarize, resource management can make the difference for systems in which:

1. Resources are fairly utilized;
2. and requests can be classified according to some metric of importance.

Resource management entails a number of aspects, but often the way a resource is managed can be summarized by discussing about how workload is dispatched/allocated to use a given resource (or resources) under analysis. This goes under the name of "scheduling".

14.2 Resources and scheduling

Unfortunately, it is not possible to devise a scheduling strategy that always achieves the best performance on any type of resource. This is because both the metric on which performance is measured, as well as the specific characteristics of the resource being managed can vary. Often, even a scheduling discipline that works optimally on a given resource of type A can yield sub-optimal performance in a system composed by two or more resources of type A. In general, devising optimal or near-optimal scheduling strategies for any non-trivial system represents an open research field.

Consider now a single resource. Scheduling policies yield different results according to specific properties of the resource. An important distinction can be made between: (i) stateful and stateless resources; and (ii) preemptive and non-preemptive resources.

14.2.1 Stateful & stateless resources

A given resource is said to be **stateless** if the amount of time to service a given request is independent from the history of the requests that have been served before.

Conversely, a resource is said to be **stateful** is the time it takes to satisfy a request does depend on previously serviced workload.

One note here: because all the real implementations of a system are subject to the limitation of the physical world, no resource is *truly* stateless. Rather, we classify a resource as stateless if the impact of previously executed requests on new workload is negligible.

The CPU is traditionally considered an example of a stateless resource if we ignore or abstract away the cost that it takes to switch between two tasks. On the other hand, any resource that involves a cache is typically a stateful resources. Then how about CPU caches ? As we will see, it depends on the time-scale at which scheduling is considered.

14.2.2 Preemptive & non-preemptive resources

Another important distinction needs to be made between resources that cannot be interrupted while servicing a request, and resources that can be temporarily interrupted to serve a different request, and eventually switched back to resume the interrupted work.

The former type of resources are called **non-preemptive**, as opposed to the latter category of resources that are **preemptive**.

Whether a resource is preemptive or not, once again, depends on the time-scale at which we schedule the resource. A network interface can transmit different packets belonging to different flows in an interleaved fashion, but once a packet is being transmitted, the interface is non-preemptive.

Another example of a non-preemptive resource is a printer. Until the printing job at hand has been completed, the job will not be interrupted to start processing a different printing job. A CPU on the other hand is preemptive at the timescale considered for a typical OS scheduler. Conversely, many modern GPUs are non-preemptive: when a kernel is launched for execution, it needs to be necessarily finished (or aborted) before launching a new kernel.

As we will see over the next lectures, a resource that is typically considered preemptive can become temporarily non-preemptive under certain circumstances. In these cases, we say that the resource is entering/exiting a **critical section**.

14.2.3 Scheduling overhead, time-scale, events, and policy

As can be seen, the way a certain resource is considered depends on the time-scale of scheduling decisions. Let us look at the CPU, for instance. Suppose that the CPU is offered multiple streams of instructions and that we want to schedule which instruction stream gets a hold of a given CPU component (e.g. the ALU). This level of scheduling is the finest, and typically performed in hardware. It is the case of Intel hyper-threading technology, as well as the thread block scheduler in modern GPUs. In this case, we are considering a scheduling time-scale of nanoseconds. In this case, there is not much room for complex scheduling strategies.

If we consider the decisions taken by the OS to multiplex different applications on the same CPU, we consider a different time-scale. Specifically, context switching between applications is performed every hundreds of microseconds. At this scale, there is room to perform fairly complex extra computation to decide what is the most desirable way to pick the next application to be executed. Of course the “extra” computation should not jeopardize the benefits of making a more pondered scheduling decision. To put it into more formal terms, we refer to the “extra” computation as **scheduling overhead**; we call the invocation of the scheduler as **scheduling event**; and we refer to the algorithm being implemented by the scheduler to decide what to execute next as **scheduling policy**.

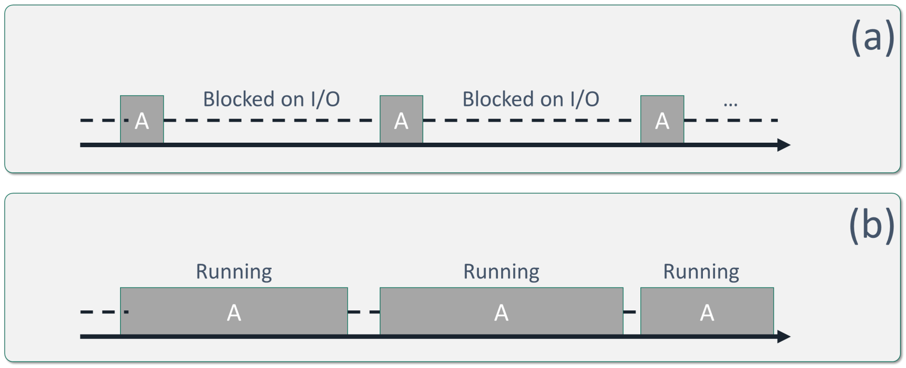

Figure 14.1: Instance of an I/O-bound process (a); and instance of a processor-bound process (b).

14.3 Processor scheduling strategies

Let us now focus on scheduling strategies for a specific type of resource: the CPU. Within these notes, we will refer to the CPU simply as “processor” and talk about “processor scheduling”. This allows us to generalize the discussion, since what will be described over the course of the following sections effectively applies to a number of types of processors, CPUs being only one of the many possible examples.

As you know from the previous lectures, a generic process could be in two main states: (i) ready to be executed or already executing on the processor; or (ii) blocked waiting for one or more external events. In case of a CPU, these external events correspond to the arrival of a certain piece of information in **input** (I) to a device, or the transmission of a piece of information in **output** (O). Hence, they are referred as I/O events.

14.3.1 I/O-bound and processor-bound processes

A generic application will switch mode between intervals that can execute without waiting for external events on the processor, and intervals where the application is blocked on I/O events. Consider the amount of time that an application could spend executing on the processor uninterrupted (assuming that the processor was entirely available for the application). We can consider such an interval a “job”. In this case, an application (a.k.a.) task is nothing but a sequence of jobs, each delimited by the interaction with one or more I/O devices.

This allows us to make a distinction between applications with relatively long jobs and brief interactions with I/O devices. This class of applications is often referred as **processor-bound applications** (see Figure 14.1(b)). Conversely, for applications that are characterized by short jobs and relatively long intervals of I/O waiting time, the term **I/O-bound applications**(see Figure 14.1(a)) is used.

14.3.2 Jobs and tasks

At this point, we can formalize the notation that we will be using from now on:

- **Job**: an interval of execution that can be entirely conducted on the processor. When a job becomes available, we say that a job has been **released**. After release and before completion, a job is considered ready.
- **Task** or **Application**: a sequence of jobs. A task is considered ready if one of its jobs is ready.

The abstraction of job is useful because whenever a job is available, it follows that the corresponding application is ready to execute on the processor. We can then focus on the problem of how the processor is assigned to a group of available jobs. Or in other words, how the processor is *scheduled* to a set of ready applications.

Take as an example a scientific application that requires heavy computational power. Let us assume that this application has been coded in an old-school way: all the heavy calculations will be performed on the CPU. So, our processor now is the CPU, and the scientific application under analysis requires very little inputs (mostly from mass storage) and keep the CPU busy for a while. This application is processor-bound from the point of view of the CPU.

Let us now consider that the application is optimized to take advantage of the GPU. The flow is now as follows: some data is read from disk, then the CPU prepares a GPU kernel, and the heavy computation is performed by the GPU. If we take the CPU as our processor, the application is I/O-bound since most of the time the application will appear as blocked waiting for the GPU kernel to complete. Conversely, if we take the GPU as our reference, the application is processor-bound. Because apart from the relatively small amount of time spent fetching data from the disk and preparing the kernel on the CPU, long GPU jobs will be submitted by the application. Interestingly enough, the CPU is just like any other I/O device from the point of view of the GPU. In fact, the CPU provides a kernel and initial data in input; and receives a set of results in output from the GPU.

14.3.3 Anatomy of a scheduler

By now we know that when there is a set of applications that are ready to execute on the processor, we are in the presence of a set of released jobs.

The scheduler is then responsible to decide which job goes into execution on the processor. In order to describe a scheduler, two pieces of information need to be known:

1. the **scheduler invocation**, which describes *when* the scheduler is invoked, or in other words *what* constitutes a scheduling event;
2. and the **scheduling policy**, which describes the function according to which a job in the set of released jobs is selected to run on the processor until the next scheduling event. As we will see, this function effectively sorts the jobs considering their parameters; the job with highest ranking is then picked to execute on the processor.

We are then interested in evaluating the performance of a scheduler. As always, it is important to distinguish between system-centric and process-centric evaluation metrics. The following are some evaluation metrics that are commonly used in evaluating scheduler performances.

Process-centric metrics:

- **Response time:** call t_r the release time of a job, i.e. the first time instant when a job is ready for execution. Call t_f the time at which the considered job has completed its execution on the processor. The response time R is calculated as $R = t_f - t_r$. In other words, the response time is the time that elapses between the job release and the job completion. This is often called *turnaround time*.
- **Fairness:** how different is the quality of service provided to different classes of jobs. Suppose that the scheduler is serving requests from two class of users in the system. A fair scheduler would treat the two classes equally.

System-centric metrics:

- **Utilization:** what is the utilization of the resource achieved with the considered scheduler. This is a very typical way of evaluating a scheduler. Since we are typically not in control of the load, a scheduler that can maximize processor utilization is typically to be preferred.
- **Throughput:** what is the rate at which requests are completed.
- **Complexity:** how difficult is to correctly implement and analyze a scheduling algorithm. This is a very important yet often underestimated metric. Implementation complexity impacts how much memory the scheduler will consume to compute its policy; how much time it will take to run after being invoked (scheduling overhead); how lengthy will be its implementation, and hence how likely it is that the resulting code will be buggy/flawed.

14.3.4 Work-conserving & non-work-conserving

Finally, a scheduler is said to be **work-conserving** if it will always execute something as long as at least one job is ready to execute. Hence, a work conserving scheduler will *only* idle the processor if there is no pending job to be executed.

Conversely, a **non-work-conserving** scheduler may decide to temporarily idle the processor even if there are ready jobs to be executed. Why a scheduler would want to do that may seem counter intuitive at first. Nonetheless, non-work-conserving schedulers are very common in practice. Consider for example any TDMA[1]-based system. If a TDMA slot is not utilized, the corresponding processor remains idle until the beginning of the next utilized time slot. Consider the distributed scheduling algorithms that belong to the class of Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) algorithms. In CSMA/CA the stations transmitting over a shared medium (the medium is the processor in this case!) may decide to wait for a random amount of time before transmitting even if the medium is idle. Clearly this is a non-work-conserving policy.

For the rest of this course, we will mostly focus on work-conserving scheduling algorithms.

[1] Time Division Multiple Access

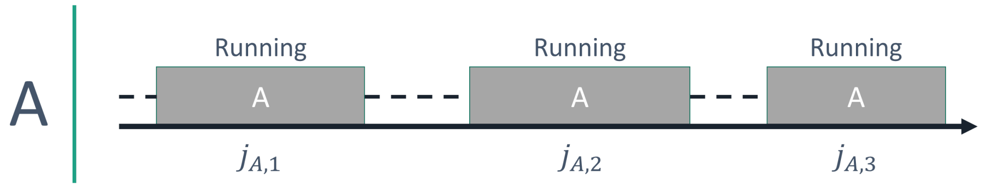

Figure 14.2: Three jobs of task A: $j_{A,1}$, $j_{A,2}$, and $j_{A,3}$.

Job	Arrival time	Job length
j_1	0	4
j_2	3	6
j_3	4	3
j_4	7	7
j_5	9	2

Table 14.1: Summary of job parameters used to illustrate the considered scheduling algorithms.

14.4 General purpose scheduling policies

Let us now recap a number of commonly used scheduling policies for general purpose processors.

For the purpose of these examples, let us define a notation for jobs. We will identify a job with j_i, where i refers to the task to which the job belongs. It can also be seen as the task ID (or index). When more than one job is considered for the same task, we will use the notation $j_{i,k}$. In this case, $j_{i,k}$ refers to the k-th job of task i. Figure 14.2 depicts this notation. The task ID i can be a number or a letter. In the next subsections, we will consider the schedule that results under different scheduling policies with the jobs considered in Table 14.1. Since only one job per class/task is considered, we use the notation j_i.

14.4.1 First Come First Served (FCFS) – a.k.a First In First Out (FIFO)

- **Invocation:** if there are no jobs in the system, the scheduler is invoked as soon as a new job has been released. Otherwise, the scheduler is invoked only once the currently running job has been completed. The scheduler is hence non-preemptive.
- **Policy:** jobs are sorted in ascending order by arrival time. Hence, the job that has been waiting for longer in the queue is selected for execution.

Let us consider the schedule produced by FCFS with the job parameters in Table 14.1. The schedule is reported in Figure 14.3.

As can be seen from the figure, the jobs are executed and complete in the order of their arrival. The completion times for j_1 to j_5 are respectively: 4, 10, 13, 20, and 22. We can now compute the response times of these jobs as follows: $R_1 = 4 - 0 = 4$; $R_2 = 10 - 3 = 7$; $R_3 = 13 - 4 = 9$; $R_4 = 20 - 7 = 13$; and $R_5 = 22 - 9 = 13$.

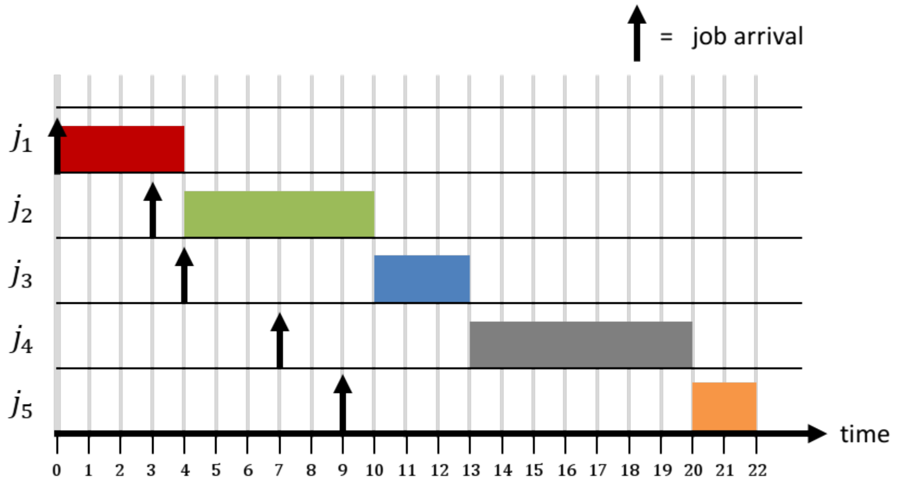

Figure 14.3: Example of FCFS schedule.

We can now calculate the average response time for these jobs, as:

$$\bar{R} = \frac{1}{5} \cdot \sum_{i=1}^{5} R_i = 9.2 \tag{14.1}$$

Could we have achieved a better average response time for our jobs ? For instance, if we swap the order in which j_4 and j_5 are executed, we have the following new values of $R'_4 = 22 - 7 = 15$ and $R'_5 = 15 - 9 = 6$. The new average is $\bar{R}' = 8.2$. Clearly the considered swap improved the average response time.

This example highlights an important limitation of FCFS (and in general of any non-preemptive scheduler): once a long job is given control of the processor, all the other jobs need to wait a long time before having the chance to execute, even if their demand is small. Swapping j_4 with j_5 improved the average response time because a short job was preferred over a long one.

Recall that short jobs represent processor bursts of I/O-bound processes. By making such processes wait for the processor, we may be hurting the system in a different way. In fact, we may be reducing the utilization of I/O devices (and consequently the overall throughput of the system), which may remain idle because I/O-bound processes are made to wait for the processor.

Given the unfairness in dealing with long vs. short jobs, let us try to discuss about a scheduling algorithm that attempts to evenly distribute processor time to jobs, i.e. round-robin.

14.4.2 Round-Robin

- **Invocation:** the scheduler is invoked at the expiration of a fixed time inteval, called *quantum*, denoted with q. The scheduler is also invoked if the currently running job completes.
- **Policy:** the job that has been waiting the longest since the last time a procesor quantum was assigned to it is picked next for execution.

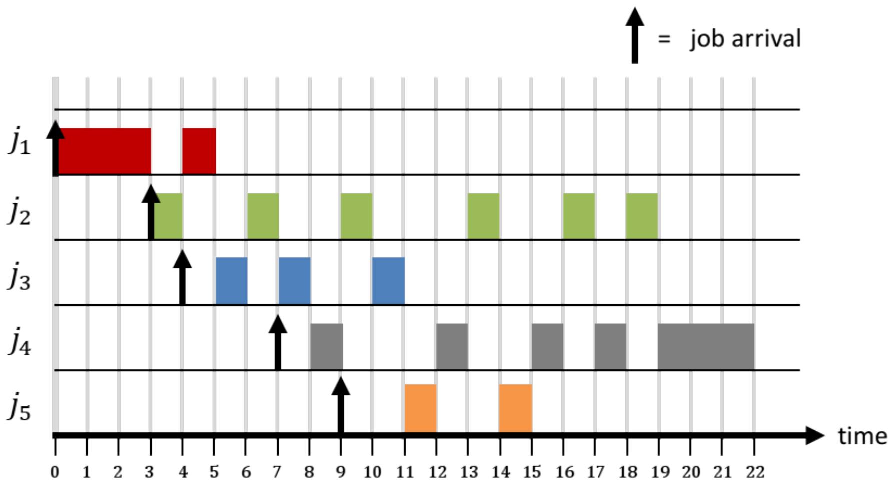

Figure 14.4: Example of Round-Robin schedule when ties are broken based on job index: **higher index** is ranked higher in case of a tie.

Let us consider the schedule produced by RR with the job parameters in Table 14.1 and quantum length $q = 1$. The schedule is reported in Figure 14.4. Often when a scheduling algorithm ranks jobs according to its policy, two or more jobs can be in a tie. Breaking a tie can be done arbitrarily (randomly pick one of the jobs with equal highest ranking), or deterministically. Since there is no advantage in principle to perform a tie-break arbitrarily, many practical implementations use the index/ID of the job/task to break ties. For instance, in Figure 14.4 ties are broken by picking the job with higher index if two or more jobs have been waiting in the queue for the same amount of time. For instance, this situation in Figure 14.4 occurs at time 5. In fact, j_2 has been waiting for 1 time unit, just like j_3. Since $3 > 2$, j_3 is picked for execution.

As a reference, consider Figure 14.5, where ties are broken with the opposite criteria: lower job index results in higher ranking in case of a tie.

How does the performance of RR compare to FCFS ? With the particular job parameters, and using the tie-breaking criteria used in Figure 14.5, we can compute the response time for our jobs. Specifically, $R_1 = 5 - 0 = 5$; $R_2 = 19 - 3 = 16$; $R_3 = 11 - 4 = 7$; $R_4 = 22 - 7 = 15$; and $R_5 = 15 - 9 = 6$. It follows that the average response time is $\bar{R} = 9.8$. As can be seen, in this particular case, RR has an average response time that is worse than FCFS. What is the benefit of RR compared to FCFS ? The benefit lies in the way jobs that require less processor time are treated. Specifically, the slowdown ($\frac{\text{response time}}{\text{requested time}}$) for the shortest job ($j_5$) is now 3 while it was 6.5 under FCFS.

This illustrate the main property of RR: it uses preemption to attempt a fair assign of processor time to ready tasks. In other words, it prevents processor-hungry jobs from taking over the processor for long periods of time. The fair distribution of processor time, however, is performed at a fixed granularity, which corresponds to the size of the quantum q. In order to obtain perfect fairness, one would like to have $q \to 0$. Conversely, it is easy to see that as we enlarge the quantum q, the produced schedule gradually degenerates toward FCFS. Specifically, the schedule produced by RR

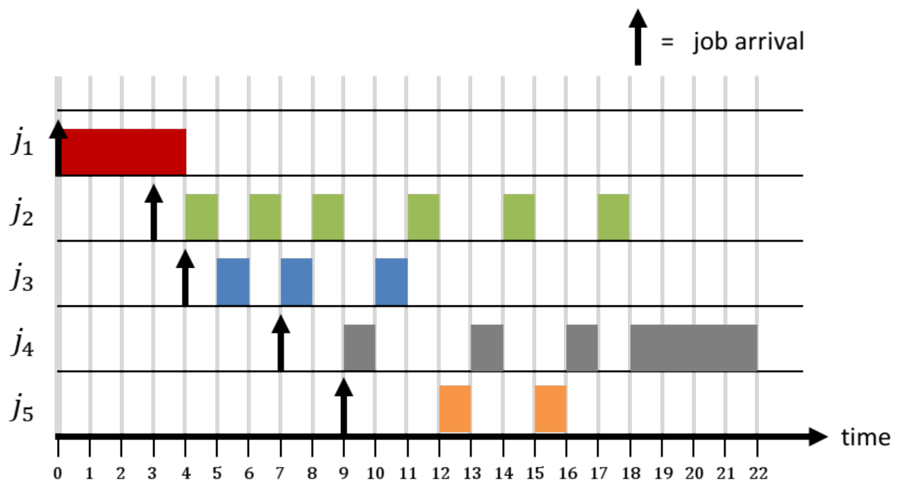

Figure 14.5: Example of Round-Robin schedule when ties are broken based on job index: **lower index** is ranked higher in case of a tie.

is indistinguishable from that of FCFS if q is greater than or equal to the longest job in the system.

So the solution is simple: let us make q as small as possible. Yeah, that only works in a perfect world. In practice, unfortunately, stopping a running job, computing who needs to go next, and context-switching to the next job in line takes time. This time is effectively scheduling overhead. The per-switching overhead is independent from q and hence it represents a constant that depends only on the considered system. Decreasing q results in more context-switches and scheduler invocations, which in turns results in higher total overhead. All is not lost however. A practical way to compute q is to decide how much overhead we are willing to pay for our system. For instance, if we want that under no circumstances the total overhead goes beyond 1%, and we know that the per-invocation overhead is o_{rr}, then we can setup q such that $q = 100 \cdot o_{rr}$.

Note that when jobs can be naturally divided into small pieces of fixed length without incurring in much overhead, round-robin represents a low-complexity, low-overhead, high-fairness scheduling algorithm. For instance, consider as processor the main memory interconnect in a multi-core CPU architecture. Different CPUs submit memory transactions to be sent to main memory on the same interconnect. Since we consider the memory interconnect as our processor, jobs are CPU-originated fetches of relatively large memory blocks. These fetches are naturally carried by the hardware as a sequence of individual main memory transactions. In other words, a single fetch is pre-divided into quantums. These transactions need to be scheduled on the physical bus lines, and an hardware scheduler is used. It is common for the interconnect hardware scheduler to implement RR, where q is taken as the length of a single memory transaction. In this case, the scheduler only needs to keep track of the last time a transaction was scheduled for a given CPU. A transaction for the CPU that has been waiting the longest is performed next.

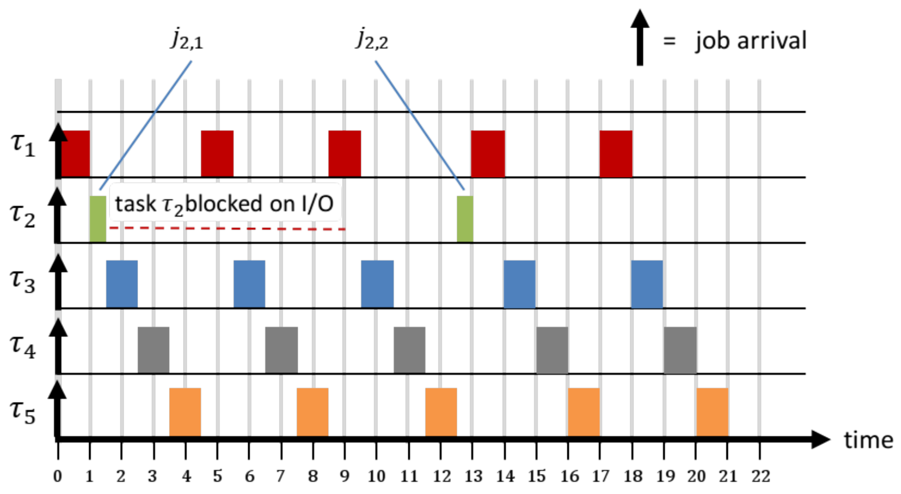

Figure 14.6: Example of Round-Robin schedule when ties are broken based on job index (**lower index** is ranked higher), in presence of an I/O bound task (τ_2); $q = 1$.

14.4.3 Virtual Round-Robin

So far we have been considering only the cases in which: (i) a task is ready (i.e. a job for that task is ready); or (ii) the corresponding job has been completed. If we are in presence of I/O-bound tasks, however, there could be a third case: (iii) the current job has finished before the end of the quantum, and the next job will not be available until some I/O event has occurred. In this case, we need to treat differently short jobs that complete before the expiration of their quantum because they belong to I/O-bound tasks. Virtual Round-Robin attempts to solve this problem.

- **Invocation:** the scheduler is invoked at the expiration of a fixed time inteval, called *quantum*, denoted with q. The scheduler is also invoked if the currently running job completes.
- **Policy:** the scheduler keeps two queues. A task enters the high-priority queue if a job belonging to that task has completed before the expiration of its assigned quantum. At the expiration of a quantum, if a job for a task in the high-priority queue is ready, it is selected for execution, and the corresponding task is removed from the high priority queue. A low-priority queue is also maintained for newly arrived jobs, or for jobs whose quantum has expired. The job that has been waiting the longest in the high-priority queue is selected for execution. If the high-priority queue is empty, the job that has been waiting the longest in the low-priority queue is selected for execution.

In order to visualize how Virtual RR operates, let us simplify the task parameters. Let us assume that we have 5 tasks $\tau_1, \ldots, \tau_5$. All the tasks are ready at time 0. All the tasks excluding τ_2 generate one job, which we will indicate simply with j_i with $i \in \{1,3,4,5\}$. All of these single jobs require 5 units of processor time to complete. Conversely, τ_2 emits a first job $j_{2,1}$, then blocks over an I/O event for 7.5 time units, and finally emits a second job $j_{2,2}$. Both $j_{2,1}$ and $j_{2,2}$ require 0.5 units of time to complete. Finally, let us consider a setup with $q = 1$. The resulting schedule if we were in presence of simple RR is reported in Figure 14.6.

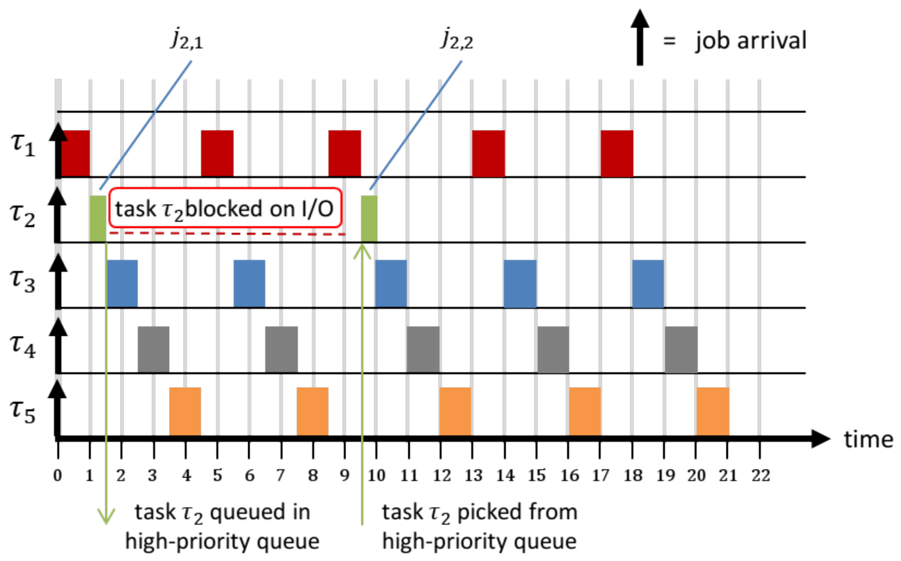

Figure 14.7: Example of Virtual Round-Robin schedule when ties are broken based on job index (**lower index** is ranked higher), in presence of an I/O bound task (τ_2); $q = 1$.

As can be seen, job $j_{2,2}$ is unfairly delayed at time 9. This problem is solved in Virtual RR with the introduction of a special, high-priority queue for tasks whose jobs prematurely release the processor before the expiration of q. Figure 14.7 depicts the schedule that results when Virtual RR is used instead of RR on the same task-set considered for Figure 14.6.