15. Length-Aware Job Scheduling

In this chapter we introduce single-resource scheduling techniques that take into account the length of scheduled jobs. We also briefly discuss a few techniques to (1) estimate and (2) predict the length of a job, to enact the considered techniques.

15.1 Considering Additional Parameters

So far we have considered tasks and corresponding job as fundamentally "unknown" entities in the system: a job arrives at the system, but: (i) we assume nothing about when the job can arrive; (ii) we assume nothing about the length of the job; and (iii) we assume nothing about when the job would like to finish being serviced (deadline). In reality, it is often the case that some of these parameters are know, or at least derivable.

In this lecture we will explore few scheduling strategies that rely on some knowledge about the length of a job, rather than just its arrival time. The first algorithm that we explored in the previous lectures was FCFS. FCFS is by nature non-preemptive. Thus, we wonder how we could "improve" scheduling decisions for inherently non-preemptive processors *if* we know the length of jobs in addition to their order of arrival.

From now on, let us call C_i the length of a generic job j_i. Let us use the same task parameters used in the previous lecture for our examples, reported in Table 15.1.

15.1.1 Shortest Job Next (SJN)

- **Invocation:** the scheduler is invoked at the completion of a job (non-preemptive).
- **Policy:** the ready job with the shortest length is picked next to execute on the processor.

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

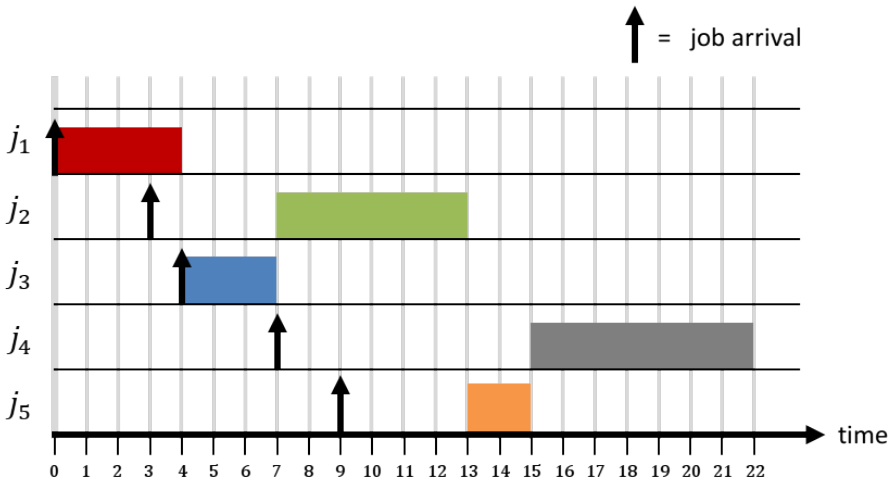

Figure 15.1: Example of SJN schedule.

One of the problems that we discussed with FCFS is that short jobs arrived after long jobs will have to wait a long time before acquiring the processor, despite needing little processor time to complete. SJN tries to patch this problem by sorting in ascending order the jobs in the ready queue. Whenever the processor becomes available to start a new job, the job at the top of the queue is picked and non-preemptively executed.

Figure 15.1 depicts the resulting schedule when SJN is used. Note that if FCFS was used (see previous lecture), jobs $j_1, \ldots, j_5$ would have executed in the order: j_1, j_2, j_3, j_4, j_5 (i.e. in their order of arrival). Under SJN, the order is j_1, j_3, j_2, j_5, j_4. The order clearly depends on the job length **and** the order of arrival. In fact, j_5, which is the shortest job in the system, executes after j_2 because it was not arrived by the time j_2 starts execution. However, it is scheduled before j_4 because it arrives "early enough" and because it is shorter than j_4.

The whole point of preferring short jobs is to try to reduce their response time, and to overall lower the average response time. Let us now take a look at the resulting response times for our jobs. We have $R_i = 4-0 = 4; R_2 = 13-3 = 10; R_3 = 7-4 = 3; R_4 = 22-7 = 15$; and $R_5 = 15-9 = 6$. The average response time, thereby, is $\bar{R} = 7.6$. With the same job-set, the average response time of FCFS was 9.2. As it turns out, there is no non-preemptive algorithm that can perform better than SJN with respect to the average response time metric. In other words, SJN is the optimal non-preemptive strategy to minimize average response time (see proof of this statement in the appendix).

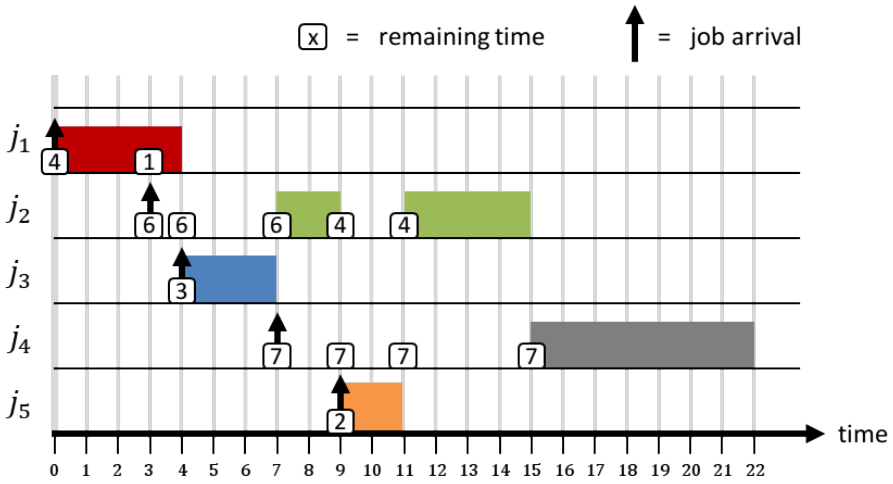

Figure 15.2: Example of SRT schedule.

Taking a look a the slowdown next, we notice that the slowdown for the shortest job in the system j_5 is $\frac{R_5}{C_5} = 3$, which is lower than what we calculated under FCFS (6.5) as well as RR (3.5, with $q = 1$). In this case SJN performed even better (at lest in terms of slowdown on short jobs) than RR, which is preemptive.

Despite displaying some ideal properties, however, SJN suffers from a major flaw. In fact, consider what happens to a long job j_l if the system receives a steady arrival of short jobs $j_{s,i}$. Well, if $j_{s,0}$ arrived before j_l, it will definitely execute before j_l. Now if before the completion of $j_{s,0}$, a second job $j_{s,1}$ arrives, such that $C_{s,1} < C_l$, then $j_{s,1}$ will also execute before j_l. And this sequence could repeat over and over again. In this case, we say that j_l can suffer from **starvation**, because under certain conditions, it many never get a hold of the processor and exhibit an infinite slowdown.

Clearly, SJN tries to favor short jobs (it is in its name after all). But there a limit to what this algorithm can do. In fact, consider the case in which we had kept all the other task parameters, but made j_2 very long. In this case, j_5. despite being the shortest job in the system, would still need to wait a long time before executing on the processor. This problem is inherently embedded in the fact that SJN is non-preemptive. In these cases, we say that j_2 is **blocking** j_5 from executing. We will see that the concept of blocking is central when studying non-preemptive scheduling policies.

15.1.2 Shortest Remaining Time (SRT)

- **Invocation:** the scheduler is invoked at the completion of a job, or at the arrival of a new job (preemptive).
- **Policy:** the ready job with the shortest remaining execution time is picked next to execute on the processor.

Figure 15.2 depicts the schedule that results by applying SRT to the considered job-set. In the figure, the remaining time for ready tasks is annotated in a box. The annotation is reported in

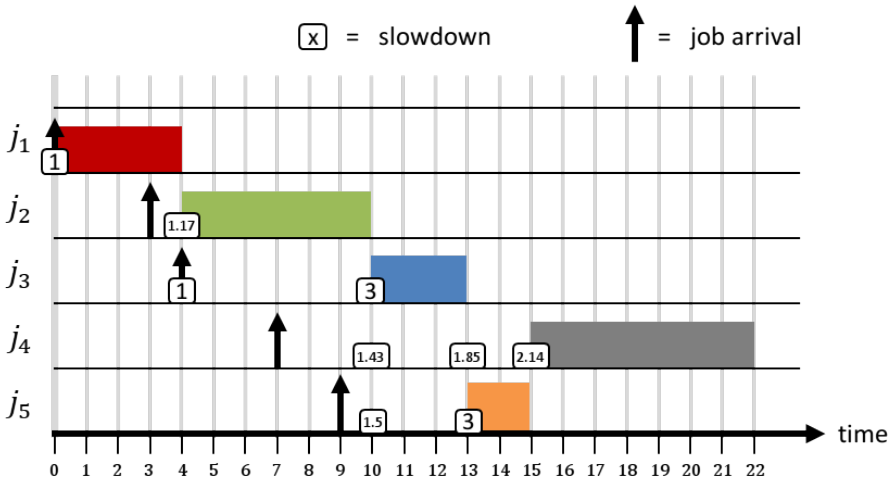

Figure 15.3: Example of HSN schedule.

correspondence of scheduling events (i.e. whenever a job arrives or completes). Let us look at the main improvement over SJN: the long job j_2 is not able to arbitrarily delay the execution of j_5. If we look at the slowdown for the two shortest jobs in the system (j_3 and j_5) we find that it is 1 (minimum) in both cases.

How about starvation for long tasks ? Unfortunately, SRT does even worse than SJN on starvation. Under SRT, not only long tasks may not even begin to execute on the processor in case of a steady arrival of shorter jobs. But it could also be the case that an already running long job may never complete due to the arrival of jobs shorter than its remaining time. To visualize the latter case, consider what would happen to j_2 if a job j_6 with length $C_6 = 3$ was released at time 10; then a job j_7 with $C_7 = 3$ was released at time 13 and so on. Once again, in case of starvation, the slowdown of starved tasks can be infinite.

15.1.3 Highest Slowdown Next (HSN)

- **Invocation:** the scheduler is invoked at the completion of a job (non-preemptive).
- **Policy:** the ready job with the largest slowdown is picked next to execute on the processor. At a scheduling event, the slowdown for ready tasks is calculated as the slowdown that the task would have if it was executed next.

In order to fix the starvation problem of SRT/SJN, s good way could be to order tasks by their slowdown. Clearly, short tasks accumulate slowdown faster and are hence favored by the scheduler. However, the more long tasks are penalized by the execution of shorter tasks, the higher they will be ranked at the next scheduling event.

Figure 15.3 depicts the schedule created by HSN with the job parameters considered so far. The number in the boxes report the calculated slowdown for ready tasks in correspondence of scheduling events. Suppose that we have a scheduling event at time t, then the slowdown for any ready job j_i

arrived at a_i is calculated in the following way:

$$\text{slowdown} = \frac{t + C_i - a_i}{C_i} \tag{15.1}$$

Note that in the equation, the quantity $(t + C_i - a_i)$ captures the response time of job j_i is the job was picked to execute on the processor at t. If we compare what happens in this case compared to Figure 15.1, we can see that j_3 is executed before j_5, despite j_5 being shorter than j_3. This case represents well how a job can increase its chances of acquiring the processor as it waits for execution. In fact, at time 10, j_3 has been waiting for 6 time units, hence having a slowdown of $\frac{13-4}{3} = 3$. Conversely, j_5 which has been waiting for only one time unit, has a slowdown equal to $\frac{12-9}{2} = 1.5$.

15.2 Estimating Job Lengths

So far we have seen that we improve the performance of a scheduler if we can exploit one piece of additional information about the job at hand: its length. In general however, this parameters is not known until the job has terminated. In practice, this parameter can be derived in few ways.

1. **Static analysis:** in case of a traditional computer application, the corresponding executable is analyzed at compile-time, or after compilation (i.e., analysis is performed on the binary). Static analysis then relies on a model of the hardware on which the application will run, to derive an approximation on the execution time (length) of each job generated by the application. Static analysis is a very complex procedure because it requires specialized knowledge of the hardware and often of the application to yield accurate results.
2. **Offline measurements:** if the set of applications that will request execution on a given system/processor is known, it is possible to measure the individual execution time of each of their jobs. The information, stored in the system, is then used online for scheduling purposes when live jobs arrive at the system. The main drawback of this approach is that it is not always possible to know the entire set of applications/jobs that will request service on a given system.
3. **Online estimation:** usually, applications produce jobs that exhibit regular patterns in their parameters, including their execution time. Suppose we do not have full knowledge of the applications that will execute at runtime. We can observe the length of their jobs as they complete execution and use that information to "predict" the length of subsequent jobs.

Depending on the system at hand, the three approaches mentioned above are more or less desirable. Let us first consider general purpose systems. In these systems, the application workload is not known offline, hence online estimation is the most practical way to derive jobs parameters.

Let us first discuss the notation that we will use in this section. Focus on a given task i, and denote with $C_{i,k}$ the k-th job released by task i. If n ($n \in \{1,2,\ldots\}$) jobs have been released and have completed, we would like to keep an average updated until the n-th sample, and use that to predict the length $C_{i,n+1}$ of job j_{n+1}. Because the quantity is an estimation of $C_{i,n+1}$ computed using n observations, we will use the symbol $\bar{C}(n)$ to indicate this quantity. So $\bar{C}(n)$ is what the scheduler will use to make decisions about job j_{n+1} when it is released. Once j_{n+1} completes, the value of $\bar{C}n+1$ will be calculated and used to make decisions about j_{n+2}, and so on.

15.2.1 Simple Averages

If the parameters of the task under observation change slowly and job length variations are small in magnitude around a central value, then averaging is an effective way to predict the length of the next job that will be released by the task. In this case, the quantity $\bar{C}(n)$ can be calculated with the following equation:

$$\bar{C}(n) = \frac{1}{n}\sum_{k=1}^{n} C_{i,k}. \tag{15.2}$$

Note that if we have a large number of samples, the equation above would require that we keep in memory the observed value of all the completed n jobs. A more efficient way of computing the quantity above is to only keep track of: (i) the last average value $\bar{C}(n-1)$; and n. When a new job, say the n-th job, completes, we can generate the updated value of $\bar{C}(n)$ as follows:

$$\bar{C}(n) = \frac{\bar{C}(n-1)\cdot(n-1) + C_{i,n}}{n} \tag{15.3}$$

The main problem with simple averaging is that observations made "long ago" have the same weight of recent observations. Suppose that the observed application has exhibited a very short job length for, say, the first year it has been running. Now, we have noticed that in the last month or so, it has been consistently releasing very long jobs. With simple averaging it will take a while before the recent story of the application affects the result of the prediction.

15.2.2 Averages over a Sliding Window

One way of reflecting the higher weight of recent observations is to effectively discard older observations. In order to do this, we define a "window" of length w. The last w observations contribute to the current estimation, while samples older than w are discarded. One can modify Equation 15.2 and 15.3 to consider the average in the sliding window w as follows:

$$\bar{C}(n) = \frac{1}{w}\sum_{k=n-w+1}^{n} C_{i,k}. \tag{15.4}$$

Note that when a new observation is available, the current value of moving average can be updated by dropping the observation that falls outside the window, i.e. the $(n-w)$-th observation.

$$\bar{C}(n) = \bar{C}(n-1) - \frac{C_{i,n-w}}{w} + \frac{C_{i,n}}{w} \tag{15.5}$$

Note that because at some point we will need to know the value of $C_{i,n-w}$, it is required to keep track and remember the value of all the observations inside the sliding window w.

Figure 15.4 depicts the trend of an average performed over a sliding window. Specifically, the plot shows the trend of the calculated average compared to the real observations, for different window

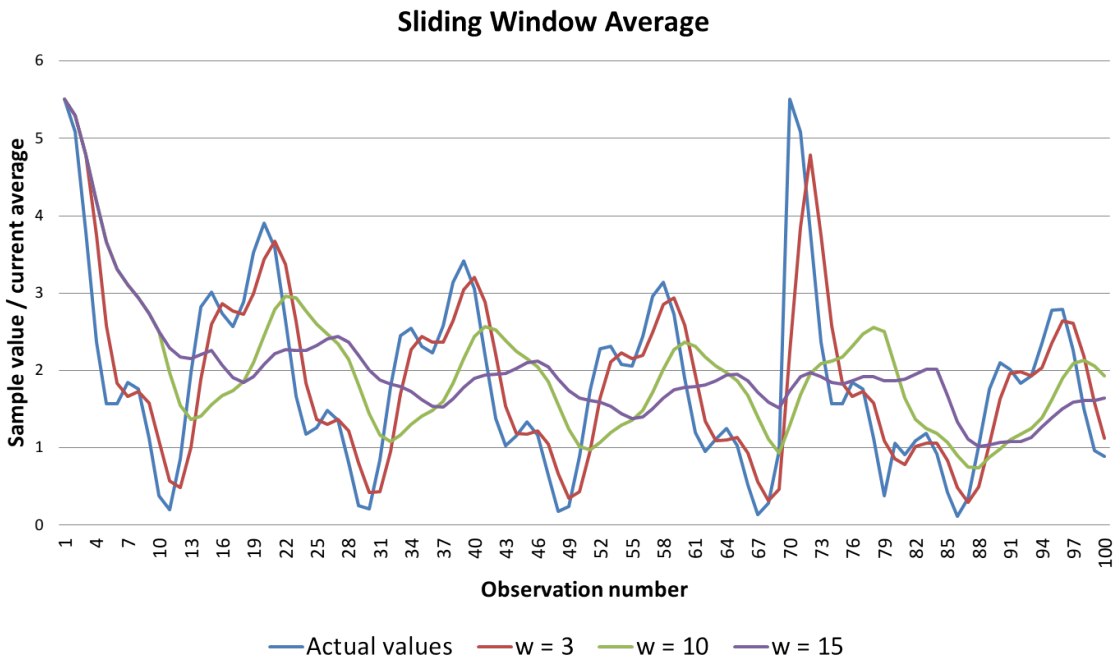

Figure 15.4: Example of Sliding Window Average for $w \in \{3, 10, 15\}$.

sizes $w \in \{3, 10, 15\}$. As can be seen, smaller window sizes follow the actual values more closely, but are easily "fooled" by a temporary perturbation of the system (see what happens around sample number 71). On the other hand, large window sizes can lag significantly behind the actual values (see line for $w = 15$).

Averaging over a sliding window better follows the recent history of the system, and hence produces better estimates about the next future compared to simple averages. Nonetheless, for fast-changing systems, the prediction may still lag behind the actual behavior of the system because equal weights have been assigned to all the observations inside the window w.

15.2.3 Exponentially Weighted Moving Averages (EWMA)

To keep track of the system's past, but to better consider recent observations, it is useful to assign a weight to our measurements. So the idea is to multiply the value of each observation by some weight, and then compute the weighted average. The magnitude of weights from the most recent observation (n) to the oldest follows a decreasing exponential trend. This method can be applied considering observations from the beginning of the sampling, or only within a window w. In practice, since weights decrease exponentially, there is always a threshold after which old observations become irrelevant and can be discarded. Hence the name *exponentially weighted* moving average.

In order to do this, we pick a coefficient α such that $\alpha \in (0, 1]$. Then, we calculate the average by constructing the current average as follows:

$$\bar{C}(n) = \alpha C_{i,n} + (1 - \alpha)\bar{C}(n-1). \tag{15.6}$$

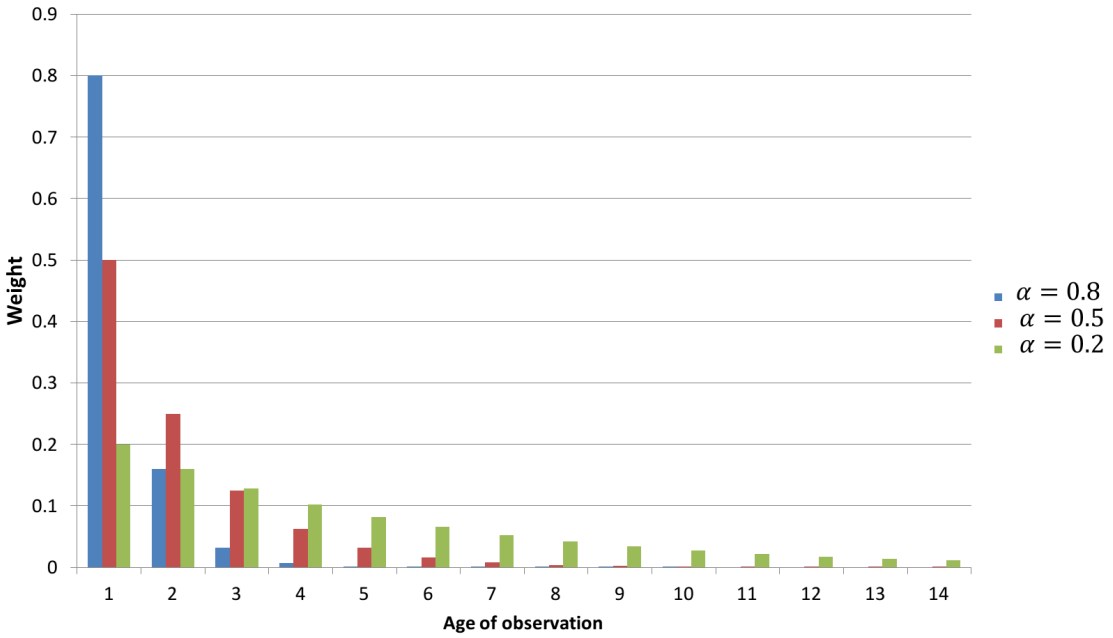

Figure 15.5: Weight for progressively old observations with $\alpha \in \{0.8, 0.5, 0.2\}$.

To properly initialize the first step in the recursive definition above, which corresponsds to the estimate $\bar{C}(1)$ for the second job right after having observed the length of the first job of τ_i, we simply set $\bar{C}(1) = C_{i,1}$.

But why is this type of estimator called an *exponentially* weighted moving average? Where does the fact that the importance of past samples is given an exponentially decaying weight comes from? To answer that, let us try to expand the recursive relation by one step. We have:

$$\bar{C}(n) = \alpha C_{i,n} + (1-\alpha)[\alpha C_{i,n-1} + (1-\alpha)\bar{C}(n-2)] \tag{15.7}$$
$$= \alpha[C_{i,n} + (1-\alpha)C_{i,n-1}] + (1-\alpha)^2\bar{C}(n-2) \tag{15.8}$$

After k steps, the formula above becomes:

$$\bar{C}(n) = \alpha[C_{i,n} + (1-\alpha)C_{i,n-1} + (1-\alpha)^2 C_{i,n-2} + \ldots + (1-\alpha)^k C_{i,n-k}] + \tag{15.9}$$
$$+ (1-\alpha)^{k+1}\bar{C}(n-(k+1)) \tag{15.10}$$

As can be seen, values obtained k observations ago, are weighted exponentially less by a factor $\alpha(1-\alpha)^k$. How this trend changes according to the value of α is depicted in Figure 15.5.

The resulting shape of the exponentially weighted average for values of $\alpha \in 0.8, 0.5, 0.2$ is depicted in Figure 15.6. Once again, it can be noted that higher values of α follow better the observations, but are more influenced by temporary fluctuations, while smaller values of α yield to predictions that lag behind the real values.

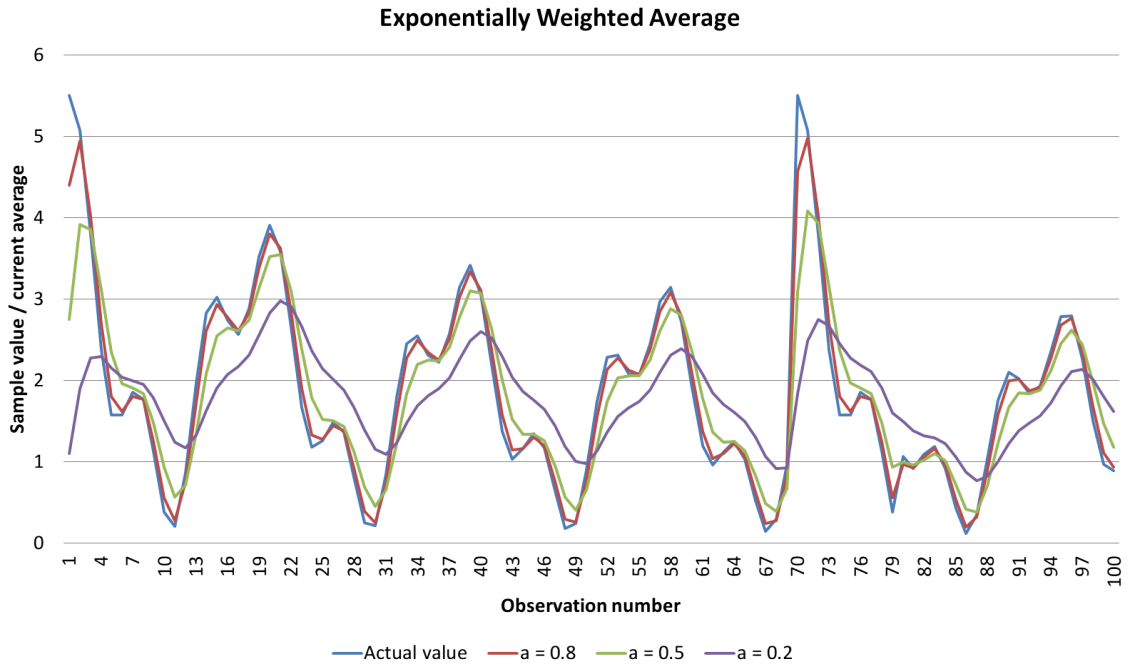

Figure 15.6: Exponentially weighted moving average for $\alpha \in \{0.8, 0.5, 0.2\}$.

15.3 Priority-Based Scheduling

In the previous example of schedulers, we have discussed how a scheduling policy is effectively composed of two parts: (i) job parameters are considered and each job is ranked; then (ii) when a scheduling event occurs, the top-ranked job is picked next for execution on the processor. The *ranking* that is associated to each job if often called the **priority** of the job. In other words, the priority is just a number that expresses an ordering on the ready jobs. The job with highest priority at the scheduling event is executed next.

In all the scheduling algorithms discussed so far, we can re-think how scheduling decisions are taken in terms of job priorities. For instance, in FCFS, the priority of a job is inversely proportional to its arrival time: the higher the *arrival timestamp*, the lower the priority.

Similarly, under SJN, the priority of job j_i is inversely proportional to C_i; in HSN the priority of j_i is directly proportional to the quantity $\frac{t+C_i-a_i}{C_i}$ (see Equation 15.1). And so on.

As we will see in the next lectures, priorities can be assigned by the scheduler, or they can be established offline. We can have three types of priorities:

1. **Static priorities:** the priorities are assigned offline to each task; each job released by a given task inherits the task-level priority.
2. **Dynamic task-level priorities:** the priority is assigned to each job of a task as soon as the job arrives. Once the priority has been assigned, however, it remains the same for the job under analysis. It follows that different jobs of the same task can have different priorities. However,

Algorithm	Priority type
FCFS	Task-level Dynamic
RR / VRR	Job-level Dynamic
SJN	Task-level Dynamic
SRT	Job-level Dynamic
HSN	Job-level Dybamic

Table 15.2: Categorization of general-purpose scheduling algorithms.

if j_i is given a certain priority, it remains the same until j_i completes.

3. **Dynamic job-level priorities:** the priority of a job can change dynamically throughout its execution.

A categorization of the scheduling algorithms encountered so far is reported in Table 15.2. It can be noted that so far we have not discussed about any scheduling algorithm with static priorities.