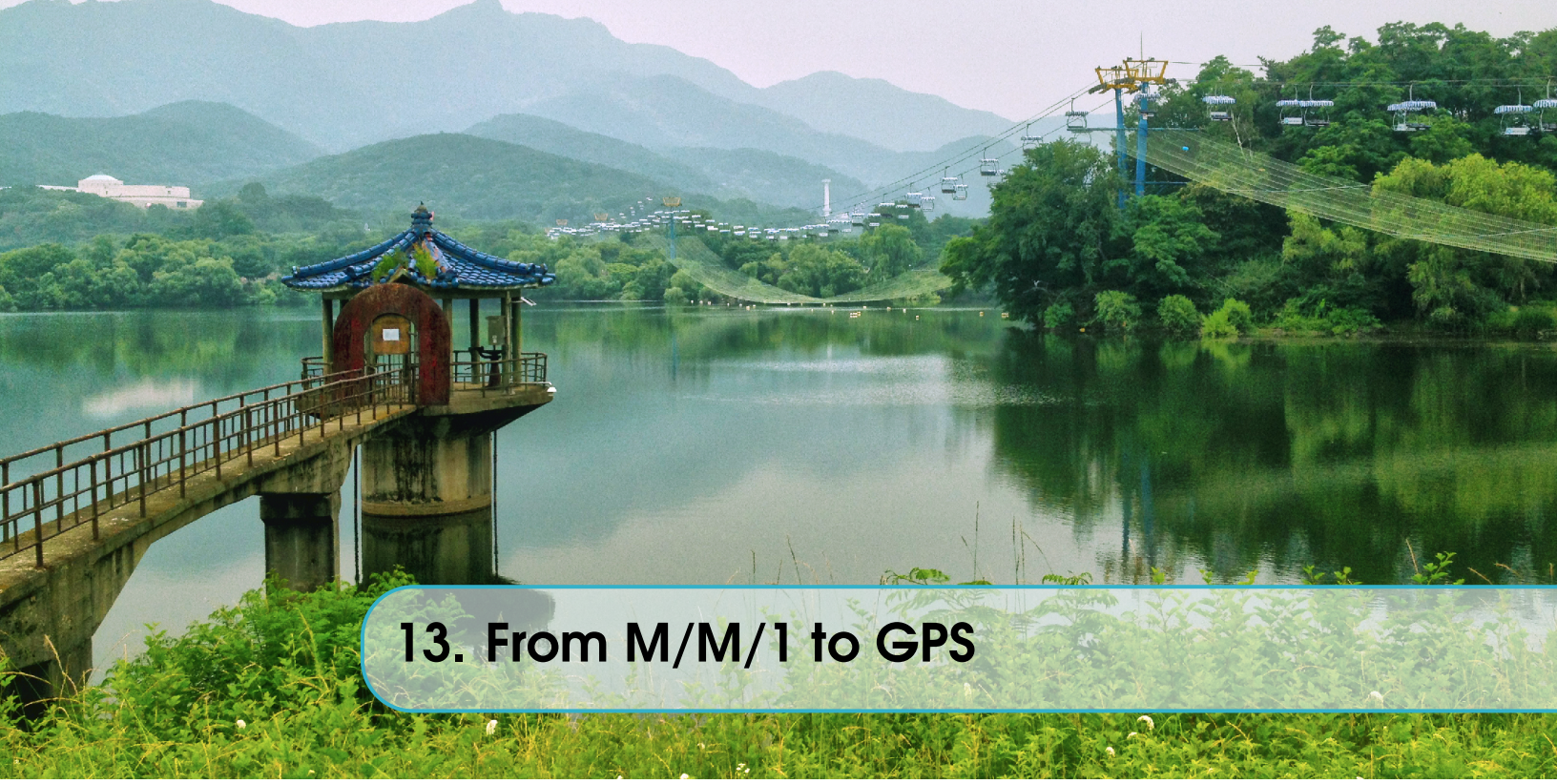

13. From M/M/1 to GPS

13.1 Generalized Processor Scheduling

A key feature of a M/M/1 system is that new workload, i.e. a newly arrived job A, needs to wait in the queue until all the other jobs ahead of A have completed their execution. If you look at the amount of work being done by the server for A – a.k.a. the service being provided, it will be 0 until all the jobs ahead of A have completed and it will be equal to the server capacity C_s as soon as A starts being processed and until the end of A.

As you can see, the server/processor is never really shared among processes/jobs. If all the systems were implemented in this way, you would not be able to take a shower while your neighbor Nick is watering his plants. This is because typically yours and your neighbor's water lines are both connected to the same city water pipeline. If M/M/1 was implemented a the fork between you and Nick's water line, and Nick got up nice and early to water his plants, you would have no choice but to wait for his gardening session to be over before you can cleanse yourself. Thankfully the same city water line is **shared** between you and Nick.

In general, sharing a resource means distributing the capacity of a given resource among all or a portion of the jobs arrived at the system. Generalized Processor Scheduling (GPS) is a generalization of a resource sharing scheme with **ideal** properties. Under GPS, when a single job A needs to be processed, it is allocated the full processing capability of the server, just like in M/M/1 systems. However, as soon as a second job B arrives and requests service, instead of queuing the second job, we *distribute* the capacity of the server in two portions. If we want an equal distribution, you would assign 50% of the server capacity to A and the remaining 50% to B. But that is not mandatory. In general, we would assign some fraction w_A of the server capacity to A and some other fraction w_B to B. The quantities w_A and w_B can be seen as "weights" and are such that $w_A, w_B \in [0,1]$, and that $w_A + w_B \leq 1$, i.e. together they can utilize up to 100% of the server's capacity.

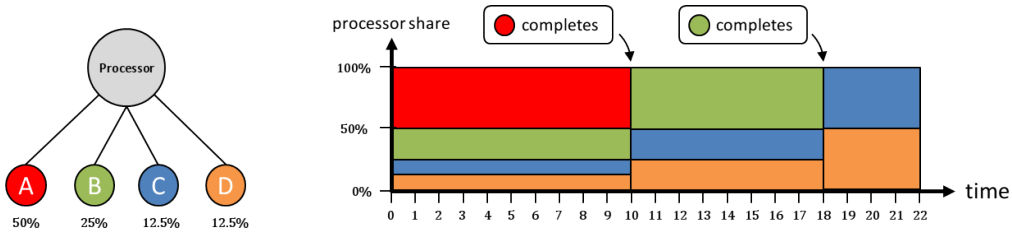

Figure 13.1: Example of GPS schedule.

Just like for a simple water pipeline: when your neighbor is watering his plants and you are still sound asleep, all the water on the pipeline is directed to his garden hose. As soon as you get up and "request" water for your shower, some of the water on the shared pipeline is directed to you, leaving some other portion of the available water to Nick and his plants[1].

The example above only considered two jobs: A and B. In general however there could be N jobs, and each would be assigned its own weight w_i. Clearly in this case it must hold that $i \in \{1,2,\ldots,N\}$, and that $w_1 + w_2 + \ldots + w_N = \sum_{i=1}^{N} w_i \leq 1$. So, if the server under consideration has capacity C_s, then the capacity C_i assigned to each job will be:

$$C_i = \frac{w_i}{\sum_{i=1}^{N} w_i} \cdot C_s \tag{13.1}$$

A representation of how GPS entirely assigns the available processor capacity to requests is provided in Figure 13.1. In the figure, four types of requests A,B,C,D are considered. For this example, the weights are: $w_A = 0.5, w_B = 0.25$, and $w_C = w_D = 0.125$. Moreover, we assume that one request for each of the four types is ready at time 0; that the type-A request will complete at time 10; and that the type-B request will complete at time 18.

Note that if the capacity of the server is entirely assigned to ready jobs (see time interval 0 to 10 in Figure 13.1), Equation 13.1 simplifies to: $C_i = w_i \cdot C_s$. Now we know what we would like to achieve. We also know, however, that a realistic processor (a network switch, a CPU, ...) can only be used by a single process at any instant of time. Then how could we ever implement any processor sharing policy, let alone GPS ? Read the next section to figure that out.

13.2 Round-Robin as Practical GPS

Let us go back to the water pipeline example. Suppose that by specification, 50% of the water should be allowed on the two sides of the fork. Is there any way to achieve a perfect 50-50 split ? In short, NO. If in the observed time window there are an odd number of water molecules passing through the fork, it would be necessary to split a molecule in two to have a perfect 50-50 split. The problem is always *granularity*. You can try to split things across at a finer and finer granularity

[1]None of you two may notice a change in water flow, however. And that is because city water is typically **over-provisioned**, meaning that a typical house plumbing is not able to consume 100% of the available city water at any time. Not even half of it, but only a fraction.

until you hit an unsplittable quantity. In case of water pipes, the unsplittable quantity is the water molecule. In networking it could be packets, in CPUs it could be single instructions or atomic blocks of instructions.

Now let us suppose that N jobs are ready at the processor/server. Let us also suppose that we want to entirely distribute the server capacity among the ready processes/jobs, and that each process will receive the same amount of service, i.e. each job receives C_s/N of the original server capacity C_s. One way to achieve this is to define a **time quantum** during which the server processes one of the ready jobs. As soon as the quantum expires, the server puts back the job in the ready queue (if it has not been completed during the last quantum) and starts processing a different job. No job is picked twice until the "round" is completed. This approach is called **Round-Robin** (RR) scheduling. The time quantum is often referred as round-robin slice.

What happens if we manipulate the RR slice? As we shrink the slice and make it smaller and smaller, jobs being processed by the system alternate at a faster peace. If the length of the slice approaches (but it is not equal to) 0, we achieve perfect GPS. This however is often not possible because at some point we hit the granularity wall.

An important note: round-robin in its "default" formulation assigns equal weights to all the processes. Nonetheless, round-robin can be adapted to approximate the more general formulation of GPS. One of such extensions is called **weighted round-robin**. Hereafter, we will be only focusing on standard round-robin.

13.3 Measuring Performance with Round-Robin

For a M/M/1 queuing system, we have calculated important quantities over the previous lectures. Can we derive the same quantities when a RR server with slice of length s is used instead? This slice of time slice is also called the *quantum* of resource time allocated to each job before they timeout and are sent back to the ready queue.

In order to answer this question, we need to abandon the deterministic world and formulate the problem in terms of probabilities. In a system with Poisson-distributed arrivals, we know that new processes arrive at a rate λ.

However, how can we model the length of a process? Effectively, the length of a process determines how many slices are required to complete the job. From a probabilistic perspective we can imagine that after a first slice, the process may be completed or not according to a probability p. Once a slice is completed, with probability p the job is not finished and re-enters the round. With probability $1-p$, the job is completed and exits the round.

Let us introduce a random variable m that captures how many times the process completes a slice and goes back in the round instead of terminating. Clearly, the job will execute for $m+1$ slices: the first plus any additional slice after re-entering the round. It follows that the total processing time t will be:

$$t = (m+1) \cdot s. \tag{13.2}$$

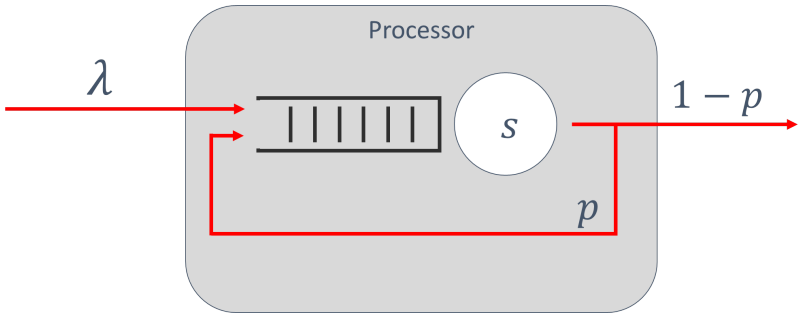

Figure 13.2: A round-robin processor as a network of M/D/1 queues.

But how many times a process re-enters the round ? Clearly, that depends on p. Once a slice is completed, a p-weighted coin is flipped. If the result is head, the process re-enters the round; if the result is tail, the process terminates and exits the round. This way of thinking about a RR processor is depicted in Figure 13.2. As you know, flipping a coin is equivalent to a Bernoulli trial. A sequence of Bernoulli trials, on the other hand, follows a Geometric distribution. Hence, the probability $f(m)$ that a given job re-enters the round m times will be:

$$f(m) = p^m \cdot (1-p). \tag{13.3}$$

Recall that for a Geometric distribution, the expected value is:

$$E[m] = \frac{p}{1-p}. \tag{13.4}$$

With this, we can finally compute the mean for the service time T_s as follows:

$$\begin{aligned} T_s &= E[t] = E[(m+1)\cdot s] \\ &= (E[m]+1)\cdot s = \left(\frac{p}{1-p}+1\right)\cdot s \\ &= \frac{s}{1-p}. \end{aligned} \tag{13.5}$$

13.3.1 System Utilization

Let us put ourselves in a perfect world where there is no limit to the granularity that one can achieve in slicing processes. As previously mentioned, we can approximate even-weighted GPS using round-robin by making the slice $s \to 0$ (read as: s approaches 0). But since the length of the jobs is untouched, as the slice becomes smaller, it takes proportionally more rounds to complete each job. This means that the probability $p \to 1$, and in turns that $(1-p) \to 0$. As mentioned before, the number of rounds m was geometrically distributed. Then, we consider μ such that $1-p = \mu \cdot s$. Recall that when a random variable X follows a Geometric distribution, the scaled variable Xs tends to be exponentially distributed with parameter μ and mean $1/\mu$.

We can now compute the service time T_s as:

$$T_s = E[(m+1)\cdot s] = E[m\cdot s] + s \tag{13.6}$$

And when $s \to 0$, we have:

$$T_s = E[m \cdot s] + 0 = \frac{1}{\mu}. \tag{13.7}$$

We also have the following relation:

$$T_s = \frac{1}{\mu} = \frac{s}{1-p}. \tag{13.8}$$

At this point, we can analyze the performance properties of round-robin by reusing some of the results used to analyze networks of M/M/1 systems, as long as we can "convert" that M/D/1 system depicted in Figure 13.2 into a M/M/1 system.

The conversion can be done by saying that instead of each slice having deterministic length s, the length of each slice is exponentially distributed with a mean of s. By playing this trick, we can now reuse Jackson's theorem to analyze our GPS approximation. Because we know that at steady state, what goes into the system has to go out (flow balance), we can compute the aggregated flow λ' as follows:

$$\lambda' \cdot (1-p) = \lambda \tag{13.9}$$

In other words:

$$\lambda' = \frac{\lambda}{(1-p)} \tag{13.10}$$

We know the service time T_s and the flow through the processor λ'. Thus, we can calculate the utilization ρ as $\rho = \lambda' \cdot T_s$. It follows that:

$$\rho = \lambda' \cdot s = \frac{\lambda \cdot s}{1-p}. \tag{13.11}$$

But since $\mu \cdot s = 1 - p$, we have:

$$\rho = \frac{\lambda \cdot s}{\mu \cdot s} = \frac{\lambda}{\mu}. \tag{13.12}$$

13.3.2 Equivalence "on Average"

As can be seen, the average utilization does not change if GPS is implemented instead of a simple FIFO policy. It makes sense because the server will still have to process the same amount of work, only in a different order. GPS and FIFO, however, share two fundamental assumptions: (1) if at least one job is ready to be processed, the server will process *something*. In other words, the server is not allowed to stay idle when there are pending, ready jobs. Scheduling disciplines that adhere to this rule are called "work conserving". Secondly, (2) the next job to be processed is selected from the queue without considering the length of the queued jobs. In other words, the queue dispatch

discipline is length-unaware. FIFO, round-robin, GPS, and random selection are all examples of length-unaware scheduling policies.

Thus, we can also generalize and state that the specific scheduling discipline does not impact the utilization of the server, as long as the discipline is work-conserving and length-unaware. In other words: given a work-conserving length-unaware resource scheduling discipline, the utilization of the resource is only affected by (i) the rate at which requests arrive to the system λ; and (ii) the amount of service that they request $1/\mu$.

Let us take a look at the number of requests in the system q under GPS (and other similar policies).

Recall that in a M/M/1 system, we can exploit flow equation to derive the probability of having $0, 1, \ldots, n$ requests in the system. These probabilities are:

$$p_0 = (1-\rho) \tag{13.13}$$
$$p_1 = p_0 \cdot \rho \tag{13.14}$$
$$p_2 = p_0 \cdot \rho^2 \tag{13.15}$$
$$\ldots \tag{13.16}$$
$$p_n = p_0 \cdot \rho^n \tag{13.17}$$
$$\tag{13.18}$$

In order to find q, we need to calculate the following:

$$q = \sum_{n=0}^{\infty} n \cdot p_n = \sum_{n=0}^{\infty} n \cdot p_0 \cdot \rho^n = \tag{13.19}$$
$$= (1-\rho) \sum_{n=0}^{\infty} n \cdot \rho^n = (1-\rho) \frac{\rho}{(1-\rho)^2} = \tag{13.20}$$
$$= \frac{\rho}{1-\rho}. \tag{13.21}$$

As can be seen, q depends solely on ρ. We have determined at Equation 13.11 that the average utilization is identical between GPS and M/M/1 systems. It follows that q is also the same for GPS and M/M/1 systems.

The same is true for the response time of the system:

$$T_q = \frac{q}{\lambda} = \frac{1}{\mu \cdot (1-\rho)}. \tag{13.22}$$

From all this discussion, it follows that M/M/1 system is a good way to approximate any work-conserving, length-unaware scheduling policy (including GPS) on average. The key thing to remember at this point is that measurements that work *on average* do not always tell the whole story.

In a system, we may have workload that is critical and workload that is non-critical. Critical workload may have timing constraints that need to be deterministically met for the system to work

properly, so that *just* a guarantee "on average" is not enough to formulate the correctness of the system. This will be clear when discussing in detail scheduling policies.