The Elusive 20% Time

This is the original text of Alexei Zheglov's sidebar on capacity utilization in Chapter 2. We had to edit it for space reasons for the book.

The 20% time has become popular in the software industry in recent years. Even though most software engineers don't work at companies that have 20% time, most have heard or know someone who works at a place like Google, where employees spend 80% of their hours working on what the company requires them to do and 20% on their own projects. Or so we have been told.

A shop across town is doing it and now we want to do it too. Many have tried to introduce 20% time in their workplaces and that proved to be very difficult. So, how can we do it? What are the dos and don'ts? Is there some theory behind this practice?

The main reason for 20% time is to keep the average capacity utilization at 80% rather than at 100%.

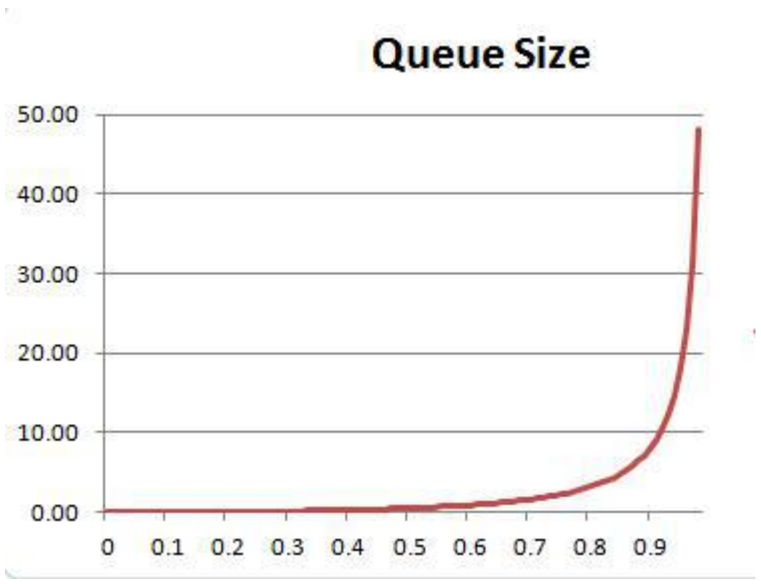
We can think of a software development organization as a system that turns feature requests into developed features. We can then model its behaviour using queuing theory.

Theory

If requests arrive faster than the system can service them, they queue up. When arrivals are slower, the queue size decreases. Because the arrival and service processes are random, the queue size changes randomly with time. The mathematically inclined can ask about this randomness: there must be some probability distribution, so what will the queue size be on average? Math (queuing theory) has an answer to that:

$$N = \frac{\rho^2}{1 - \rho}$$

(Here the Greek letter rho denotes the utilization coefficient equal to the ratio of service and arrival rates. This simple formula owes to the assumption that the arrival and service processes are so-called Markov processes. It's a close approximation of the real-world arrival and service processes. The math is more complicated for non-Markov processes, but leads to the same conclusion.)



By plotting this function we can see that the average queue size remains low while utilization is up to 0.8, then rises sharply and goes to infinity. We can understand this intuitively by thinking about our computer's CPU: when its utilization is low, it responds instantly to our inputs, but when a background task pushes its utilization close to 100%, the computer becomes frustratingly slow to respond to every click.

Practice

The economics of software development is such that software companies incur very high **costs of delay** when their queues are in high-queue states. This includes missed market opportunities, obsolete products, late projects, and waste caused by building features in anticipation of demand. The 20% time is thus the scientific answer to the problem of optimizing economic outcomes: avoid high-queue states by avoiding the high utilization causing them. It is essentially the slack that keeps the system responsive.

Several practical conclusions about **what not to do** follow immediately:

- cost accounting (engineers' time costs X, but/and the company can/cannot afford it). The economic benefit comes from reducing the cost of delay.
- setting up a 20% time project proposal submission-review-approval system
- tracking the 20% time by filling out timesheets
- using innovation as a motivator for the 20% time. While new products have come out of 20% projects, they were not the point. If your company cannot innovate during its core hours, that's a problem!
- relying on the 20% time to encourage creativity. Saying you'll unleash your creativity with 20% time begs the question why you're not creative enough already during your core hours.
- allocating the 20% time to a Friday every week...

Those are All Don'ts, Where Are the Dos?

OK, what about doing it right? Let's answer with the best question we've heard while discussing this subject with practitioners: "If 20% of your capacity is mandated to be filled with non-queue items, then you've just shrunk your capacity to 80%, and 80% is your new 100%. Right?"

Yes, "80 is the new 100" highlights the main problem with the attempts to mandate the 20% time without understanding the theory. You want to escape the utilization trap, not to stay in the trap and allocate time differently.

Remember that the utilization depends on two processes: arrival process (demand) and service process (capability). You can't really choose your utilization. It is what it is because the processes are what they are. You can, however, work on the processes: by improving your company's software delivery capability and shaping the demand. As you make progress, slack will emerge.

Do escape the utilization trap!