Infinite work streams are the new reality of many systems. Web servers and application servers serve very large user populations where it is realistic to expect infinite streams of new work. The work never ends. Requests come in 24 hours a day 7 days a week. Work could easily saturate servers at 100% CPU usage.

Traditionally we have considered 100% CPU usage a bad sign. As compensation we create complicated infrastructures to load balance work, replicate state, and cluster machines.

CPUs don't get tired so you might think we would try to use the CPU as much as possible.

In other fields we try to increase productivity by using a resource to the greatest extent possible.

In the server world we try to guarantee a certain level of responsiveness by forcing an artificially low CPU usage. The idea is if we don't have CPU availability then we can't respond to new work with a reasonable latency or complete existing work.

Is there really a problem with the CPU being used 100% of the time? Isn't the real problem that we use CPU availability and task priority as a simple cognitive shorthand for architecting a system rather than having to understand our system's low level work streams and using that information to make specific scheduling decisions?

We simply don't have the tools to do anything other than make clumbsy architecture decisions based on load balancing servers and making guesses at the number of threads to use and the priorities for those threads. We could use 100% of CPU time if we could:

0. Schedule work so that explicit locking is uncessary (though possible). This
   will help prevent dead lock and priority inversion.
1. Control how much of the CPU work items can have.
2. Decide on the relative priority of work and schedule work by
   that priority.
3. Have a fairness algorithm for giving a particular level of service
   to each work priority.
4. Schedule work CPU allowance across tasks.
5. Map work to tasks so as to prevent dead lock, priority inversion, and guarantee
   scheduling latency.
6. Have mechanisms to make work give up the CPU after its CPU budget has been used
   or higher priority works comes in, in such a way to give up locks to prevent
   dead lock and priority inversion.
7. Control new work admission so that back pressure can be put on callers.
8. Assing work to objects and process work in order to guarantee protocol ordering.
9. Ideally, control work characteristics across the OS and all applications.

The problem is we don't have this level of scheduling control. If we did then a CPU can run at 100% because we have complete control of what work runs when and in what order. There's no need not to run the CPU at 100% because we know the things we want to run are running.

It is interesting frameworks like Struts and JSP rarely even talk about threading. I don't think application developers really consider the threading architectures of their applications. Popular thread local transaction management mechanisms, for example, assume a request will be satisfied in a single thread. That's not true of architectures for handling large work loads.

Scaling a system requires careful attention to architecture. In the current frameworks applications have very little say as to how their applications run.

For a discussion of how to handle infinite workloads take a look at Architecture Discussion and specifically App Backplane.

Another way of handling higher work loads is to simply add more machines and have an architecture that supports that type of scaling. While that works, I am addressing the scenarios where you are limitted in your resources, let's say in embedded system or in scenario where you can only afford one or two machine in a colo site.