8

Scheduling: The Multi-Level Feedback Queue

In this chapter, we'll tackle the problem of developing one of the most well-known approaches to scheduling, known as the **Multi-level Feedback Queue (MLFQ)**. The Multi-level Feedback Queue (MLFQ) scheduler was first described by Corbato et al. in 1962 [C+62] in a system known as the Compatible Time-Sharing System (CTSS), and this work, along with later work on Multics, led the ACM to award Corbato its highest honor, the **Turing Award**. The scheduler has subsequently been refined throughout the years to the implementations you will encounter in some modern systems.

兼容的

The fundamental problem MLFQ tries to address is two-fold. First, it would like to optimize *turnaround time*, which, as we saw in the previous note, is done by running shorter jobs first; unfortunately, the OS doesn't generally know how long a job will run for, exactly the knowledge that algorithms like SJF (or STCF) require. Second, MLFQ would like to make a system feel responsive to interactive users (i.e., users sitting and staring at the screen, waiting for a process to finish), and thus minimize *response time*; unfortunately, algorithms like Round Robin reduce response time but are terrible for turnaround time. Thus, our problem: given that we in general do not know anything about a process, how can we build a scheduler to achieve these goals? How can the scheduler learn, as the system runs, the characteristics of the jobs it is running, and thus make better scheduling decisions?

双重

① performance

② fairness

THE CRUX:
HOW TO SCHEDULE WITHOUT PERFECT KNOWLEDGE?

How can we design a scheduler that both minimizes response time for interactive jobs while also minimizing turnaround time without *a priori* knowledge of job length?

TIP: LEARN FROM HISTORY

The multi-level feedback queue is an excellent example of a system that learns from the past to predict the future. Such approaches are common in operating systems (and many other places in Computer Science, including hardware branch predictors and caching algorithms). Such approaches work when jobs have phases of behavior and are thus predictable; of course, one must be careful with such techniques, as they can easily be wrong and drive a system to make worse decisions than they would have with no knowledge at all.

8.1 MLFQ: Basic Rules

To build such a scheduler, in this chapter we will describe the basic algorithms behind a multi-level feedback queue; although the specifics of many implemented MLFQs differ [E95], most approaches are similar.

Fig 8.1

In our treatment, the MLFQ has a number of distinct **queues**, each assigned a different **priority level**. At any given time, a job that is ready to run is on a single queue. MLFQ uses priorities to decide which job should run at a given time: a job with higher priority (i.e., a job on a higher queue) is chosen to run.

Of course, more than one job may be on a given queue, and thus have the *same* priority. In this case, we will just use round-robin scheduling among those jobs.

Thus, we arrive at the first two basic rules for MLFQ:

初始版本

- **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't).
- **Rule 2:** If Priority(A) = Priority(B), A & B run in RR.

放弃

The key to MLFQ scheduling therefore lies in how the scheduler sets priorities. Rather than giving a fixed priority to each job, MLFQ *varies* the priority of a job based on its *observed behavior*. If, for example, a job repeatedly relinquishes the CPU while waiting for input from the keyboard, MLFQ will keep its priority high, as this is how an interactive process might behave. If, instead, a job uses the CPU intensively for long periods of time, MLFQ will reduce its priority. In this way, MLFQ will try to *learn* about processes as they run, and thus use the *history* of the job to predict its *future* behavior.

举例

If we were to put forth a picture of what the queues might look like at a given instant, we might see something like the following (Figure 8.1, page 3). In the figure, two jobs (A and B) are at the highest priority level, while job C is in the middle and Job D is at the lowest priority. Given our current knowledge of how MLFQ works, the scheduler would just alternate time slices between A and B because they are the highest priority jobs in the system; poor jobs C and D would never even get to run — an outrage!

Of course, just showing a static snapshot of some queues does not really give you an idea of how MLFQ works. What we need is to under-

八种级别

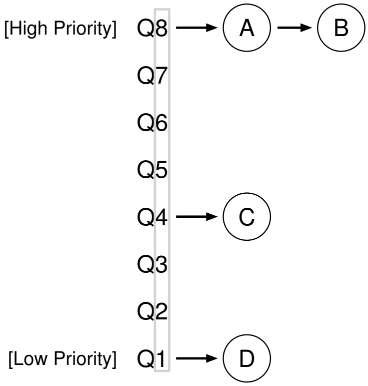


Figure 8.1: **MLFQ Example**

初始版本并未说明 jobs 的 priority 是如何动态变化的

stand how job priority *changes* over time. And that, in a surprise only to those who are reading a chapter from this book for the first time, is exactly what we will do next.

8.2 Attempt #1: How To Change Priority

We now must decide how MLFQ is going to change the priority level of a job (and thus which queue it is on) over the lifetime of a job. To do this, we must keep in mind our workload: a mix of interactive jobs that are short-running (and may frequently relinquish the CPU), and some longer-running “CPU-bound” jobs that need a lot of CPU time but where response time isn’t important.

For this, we need a new concept, which we will call the job’s **allotment**. The allotment is the amount of time a job can spend at a given priority level before the scheduler reduces its priority. For simplicity, at first, we will assume the allotment is equal to a single time slice.

Here is our first attempt at a priority-adjustment algorithm:

修正一

- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
- **Rule 4a:** If a job uses up its allotment while running, its priority is *reduced* (i.e., it moves down one queue).
- **Rule 4b:** If a job gives up the CPU (for example, by performing an I/O operation) before the allotment is up, it stays at the *same* priority level (i.e., its allotment is reset).

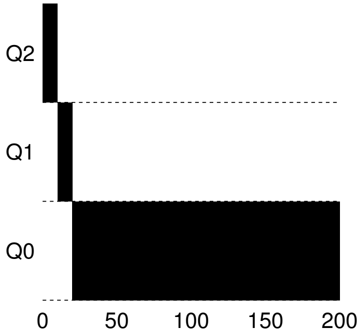


Figure 8.2: **Long-running Job Over Time**

Example 1: A Single Long-Running Job

Let's look at some examples. First, we'll look at what happens when there has been a long running job in the system, with a time slice of 10 ms (and with the allotment set equal to the time slice). Figure 8.2 shows what happens to this job over time in a three-queue scheduler.

As you can see in the example, the job enters at the highest priority (Q2). After a single time slice of 10 ms, the scheduler reduces the job's priority by one, and thus the job is on Q1. After running at Q1 for a time slice, the job is finally lowered to the lowest priority in the system (Q0), where it remains. Pretty simple, no?

Example 2: Along Came A Short Job

Now let's look at a more complicated example, and hopefully see how MLFQ tries to approximate SJF. In this example, there are two jobs: A, which is a long-running CPU-intensive job, and B, which is a short-running interactive job. Assume A has been running for some time, and then B arrives. What will happen? Will MLFQ approximate SJF for B?

Figure 8.3 on page 5 (left) plots the results of this scenario. Job A (shown in black) is running along in the lowest-priority queue (as would any long-running CPU-intensive jobs); B (shown in gray) arrives at time $T = 100$, and thus is inserted into the highest queue; as its run-time is short (only 20 ms), B completes before reaching the bottom queue, in two time slices; then A resumes running (at low priority).

From this example, you can hopefully understand one of the major goals of the algorithm: because it doesn't *know* whether a job will be a short job or a long-running job, it first *assumes* it might be a short job, thus giving the job high priority. If it actually is a short job, it will run quickly and complete; if it is not a short job, it will slowly move down the queues, and thus soon prove itself to be a long-running more batch-like process. In this manner, MLFQ approximates SJF.

近似

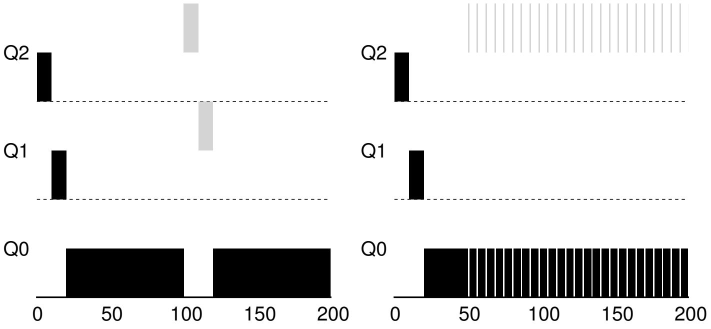


Figure 8.3: **Along Came An Interactive Job: Two Examples**

Example 3: What About I/O?

Let's now look at an example with some I/O. As Rule 4b states above, if a process gives up the processor before using up its allotment, we keep it at the same priority level. The intent of this rule is simple: if an interactive job, for example, is doing a lot of I/O (say by waiting for user input from the keyboard or mouse), it will relinquish the CPU before its allotment is complete; in such case, we don't wish to penalize the job and thus simply keep it at the same level.

Figure 8.3 (right) shows an example of how this works, with an interactive job B (shown in gray) that needs the CPU only for 1 ms before performing an I/O competing for the CPU with a long-running batch job A (shown in black). The MLFQ approach keeps B at the highest priority because B keeps releasing the CPU; if B is an interactive job, MLFQ further achieves its goal of running interactive jobs quickly.

新问题
（三个）

Problems With Our Current MLFQ

We thus have a basic MLFQ. It seems to do a fairly good job, sharing the CPU fairly between long-running jobs, and letting short or I/O-intensive interactive jobs run quickly. Unfortunately, the approach we have developed thus far contains serious flaws. Can you think of any?

(This is where you pause and think as deviously as you can)

迂回地

First, there is the problem of **starvation**: if there are "too many" interactive jobs in the system, they will combine to consume *all* CPU time, and thus long-running jobs will *never* receive any CPU time (they **starve**). We'd like to make some progress on these jobs even in this scenario.

Second, a smart user could rewrite their program to **game the scheduler**. Gaming the scheduler generally refers to the idea of doing something sneaky to trick the scheduler into giving you more than your fair share of the resource. The algorithm we have described is susceptible to

偷偷摸摸的

易受影响的

TIP: SCHEDULING MUST BE SECURE FROM ATTACK

You might think that a scheduling policy, whether inside the OS itself (as discussed herein), or in a broader context (e.g., in a distributed storage system's I/O request handling [Y+18]), is not a **security** concern, but in increasingly many cases, it is exactly that. Consider the modern datacenter, in which users from around the world share CPUs, memories, networks, and storage systems; without care in policy design and enforcement, a single user may be able to adversely harm others and gain advantage for itself. Thus, scheduling policy forms an important part of the security of a system, and should be carefully constructed.

the following attack: before the allotment is used, issue an I/O operation (e.g., to a file) and thus relinquish the CPU; doing so allows you to remain in the same queue, and thus gain a higher percentage of CPU time. When done right (e.g., by running for 99% of the allotment before relinquishing the CPU), a job could nearly monopolize the CPU. 独占，垄断

Finally, a program may *change its behavior* over time; what was CPU-bound may transition to a phase of interactivity. With our current approach, such a job would be out of luck and not be treated like the other interactive jobs in the system.

8.3 Attempt #2: The Priority Boost

Let's try to change the rules and see if we can avoid the problem of starvation. What could we do in order to guarantee that CPU-bound jobs will make some progress (even if it is not much?).

The simple idea here is to periodically **boost** the priority of all the jobs in the system. There are many ways to achieve this, but let's just do something simple: throw them all in the topmost queue; hence, a new rule:

修正二

- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

Our new rule solves two problems at once. First, processes are guaranteed not to starve: by sitting in the top queue, a job will share the CPU with other high-priority jobs in a round-robin fashion, and thus eventually receive service. Second, if a CPU-bound job has become interactive, the scheduler treats it properly once it has received the priority boost.

Let's see an example. In this scenario, we just show the behavior of a long-running job when competing for the CPU with two short-running interactive jobs. Two graphs are shown in Figure 8.4 (page 7). On the left, there is no priority boost, and thus the long-running job gets starved once the two short jobs arrive; on the right, there is a priority boost every 100 ms (which is likely too small of a value, but used here for the example),

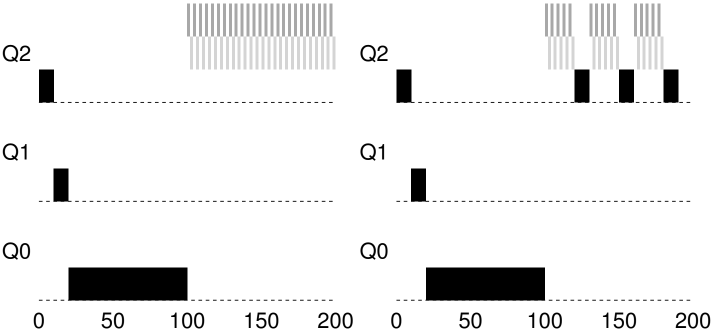


Figure 8.4: **Without (Left) and With (Right) Priority Boost**

and thus we at least guarantee that the long-running job will make some progress, getting boosted to the highest priority every 100 ms and thus getting to run periodically.

Of course, the addition of the time period S leads to the obvious question: what should S be set to? John Ousterhout, a well-regarded systems researcher [O11], used to call such values in systems **voo-doo constants**, because they seemed to require some form of black magic to set them correctly. Unfortunately, S has that flavor. If it is set too high, long-running jobs could starve; too low, and interactive jobs may not get a proper share of the CPU. As such, it is often left to the system administrator to find the right value – or in the modern world, increasingly, to automatic methods based on machine learning [A+17].

巫毒教

需专业知识去设置的参数（变量）

8.4 Attempt #3: Better Accounting

罪魁祸首

We now have one ~~more~~ problem to solve: how to prevent gaming of our scheduler? The real culprit here, as you might have guessed, are Rules 4a and 4b, which let a job retain its priority by relinquishing the CPU before its allotment expires. So what should we do?

> TIP: AVOID VOO-DOO CONSTANTS (OUSTERHOUT'S LAW)
>
> Avoiding voo-doo constants is a good idea whenever possible. Unfortunately, as in the example above, it is often difficult. One could try to make the system learn a good value, but that too is not straightforward. The frequent result: a configuration file filled with default parameter values that a seasoned administrator can tweak when something isn't quite working correctly. As you can imagine, these are often left unmodified, and thus we are left to hope that the defaults work well in the field. This tip brought to you by our old OS professor, John Ousterhout, and hence we call it **Ousterhout's Law**.

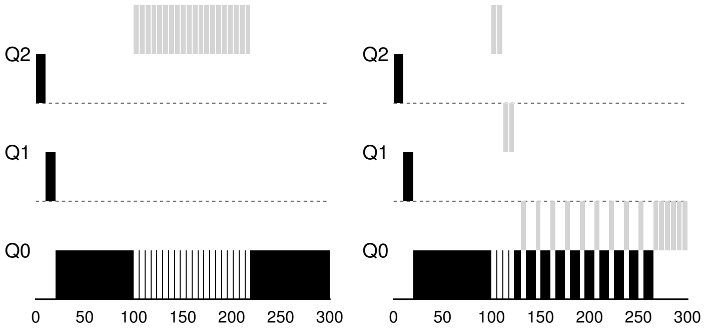


Figure 8.5: **Without (Left) and With (Right) Gaming Tolerance**

The solution here is to perform better **accounting** of CPU time at each level of the MLFQ. Instead of forgetting how much of its allotment a process used at a given level when it performs I/O, the scheduler should keep track; once a process has used its allotment, it is demoted to the next priority queue. Whether it uses its allotment in one long burst or many small ones should not matter. We thus rewrite Rules 4a and 4b to the following single rule:

降级

修正三

- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).

Let's look at an example. Figure 8.5 shows what happens when a workload tries to game the scheduler with the old Rules 4a and 4b (on the left) as well the new anti-gaming Rule 4. Without any protection from gaming, a process can issue an I/O before its allotment ends, thus staying at the same priority level, and dominating CPU time. With better accounting in place (right), regardless of the I/O behavior of the process, it slowly moves down the queues, and thus cannot gain an unfair share of the CPU.

8.5 Tuning MLFQ And Other Issues

参数化

voo-doo constants

A few other issues arise with MLFQ scheduling. One big question is how to **parameterize** such a scheduler. For example, how many queues should there be? How big should the time slice be per queue? The allotment? How often should priority be boosted in order to avoid starvation and account for changes in behavior? There are no easy answers to these questions, and thus only some experience with workloads and subsequent tuning of the scheduler will lead to a satisfactory balance.

For example, most MLFQ variants allow for varying time-slice length across different queues. The high-priority queues are usually given short time slices; they are comprised of interactive jobs, after all, and thus

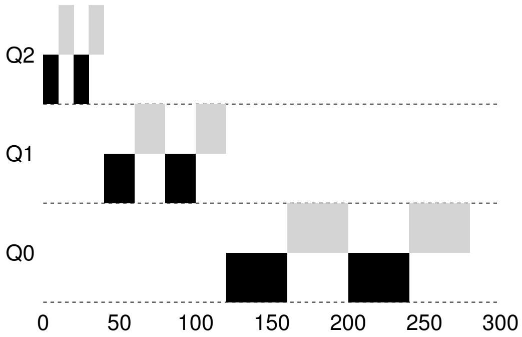


Figure 8.6: **Lower Priority, Longer Quanta**

quickly alternating between them makes sense (e.g., 10 or fewer milliseconds). The low-priority queues, in contrast, contain long-running jobs that are CPU-bound; hence, longer time slices work well (e.g., 100s of ms). Figure 8.6 shows an example in which two jobs run for 20 ms at the highest queue (with a 10-ms time slice), 40 ms in the middle (20-ms time slice), and with a 40-ms time slice at the lowest.

The Solaris MLFQ implementation — the Time-Sharing scheduling class, or TS — is particularly easy to configure; it provides a set of tables that determine exactly how the priority of a process is altered throughout its lifetime, how long each time slice is, and how often to boost the priority of a job [AD00]; an administrator can muck 淤泥 with this table in order to make the scheduler behave in different ways. Default values for the table are 60 queues, with slowly increasing time-slice lengths from 20 milliseconds (highest priority) to a few hundred milliseconds (lowest), and priorities boosted around every 1 second or so.

Other MLFQ schedulers don't use a table or the exact rules described in this chapter; rather they adjust priorities using mathematical formulae. For example, the FreeBSD scheduler (version 4.3) uses a formula to calculate the current priority level of a job, basing it on how much CPU the process has used [LM+89]; in addition, usage is decayed over time, 衰变 providing the desired priority boost in a different manner than described 在此处 herein. See Epema's paper for an excellent overview of such **decay-usage** algorithms and their properties [E95].

Finally, many schedulers have a few other features that you might encounter. For example, some schedulers reserve the highest priority levels for operating system work; thus typical user jobs can never obtain the highest levels of priority in the system. Some systems also allow some user **advice** to help set priorities; for example, by using the command-line utility `nice` you can increase or decrease the priority of a job (somewhat) and thus increase or decrease its chances of running at any given time. See the man page for more.

TIP: USE ADVICE WHERE POSSIBLE

As the operating system rarely knows what is best for each and every process of the system, it is often useful to provide interfaces to allow users or administrators to provide some **hints** to the OS. We often call such hints **advice**, as the OS need not necessarily pay attention to it, but rather might take the advice into account in order to make a better decision. Such hints are useful in many parts of the OS, including the scheduler (e.g., with `nice`), memory manager (e.g., `madvise`), and file system (e.g., informed prefetching and caching [P+95]).

8.6 MLFQ: Summary

We have described a scheduling approach known as the Multi-Level Feedback Queue (MLFQ). Hopefully you can now see why it is called that: it has *multiple levels* of queues, and uses *feedback* to determine the priority of a given job. History is its guide: pay attention to how jobs behave over time and treat them accordingly. 指导思想

The refined set of MLFQ rules, spread throughout the chapter, are reproduced here for your viewing pleasure:

总结

- **Rule 1:** If Priority(A) $>$ Priority(B), A runs (B doesn't).
- **Rule 2:** If Priority(A) $=$ Priority(B), A & B run in round-robin fashion using the time slice (quantum length) of the given queue.
- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

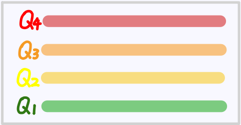
MLFQ is interesting for the following reason: instead of demanding *a priori* knowledge of the nature of a job, it observes the execution of a job and prioritizes it accordingly. In this way, it manages to achieve the best of both worlds: it can deliver excellent overall performance (similar to SJF/STCF) for short-running interactive jobs, and is fair and makes progress for long-running CPU-intensive workloads. For this reason, many systems, including BSD UNIX derivatives [LM+89, B86], Solaris [M06], and Windows NT and subsequent Windows operating systems [CS97] use a form of MLFQ as their base scheduler. 意义

假设：~~①等长~~ ~~②同时~~ ~~③原子~~ ~~④纯算~~ ⑤已知

假设：~~①等长~~ ~~②同时~~ ~~③原子~~ ~~④纯算~~ ~~⑤已知~~

idea：to learn from the past, and to predict the future !

MLFQ



rule 1：先做急事
rule 2：同级平等

但初始版本并未说明jobs的priority是如何变化的。

介绍状况：有两种类型的workload，一种是CPU-intensive，另一种是interactive。

rule 3：新人保护
rule 4："场清黄牌"

三个新问题：饥饿、欺诈、变性。

rule 5：定期重置

rule 4："场清黄牌" ⟶ "累积黄牌"

discussion

References

[A+17] "Automatic Database Management System Tuning Through Large-scale Machine Learning" by Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, Bohan Zhang. SIGMOD '17. *This isn't about the application of machine learning to CPU scheduling in the OS, but rather a cool early example of automatically tuning parameters of a database via ML techniques. Worth a read, if you like ML... which, alas, everyone seems to these days.*

[AD00] "Multilevel Feedback Queue Scheduling in Solaris" by Andrea Arpaci-Dusseau. Available: http://www.ostep.org/Citations/notes-solaris.pdf. *A great short set of notes by one of the authors on the details of the Solaris scheduler. OK, we are probably biased in this description, but the notes are pretty darn good.*

[B86] "The Design of the UNIX Operating System" by M.J. Bach. Prentice-Hall, 1986. *One of the classic old books on how a real UNIX operating system is built; a definite must-read for kernel hackers.*

[C+62] "An Experimental Time-Sharing System" by F. J. Corbato, M. M. Daggett, R. C. Daley. IFIPS 1962. *A bit hard to read, but the source of many of the first ideas in multi-level feedback scheduling. Much of this later went into Multics, which one could argue was the most influential operating system of all time.*

[CS97] "Inside Windows NT" by Helen Custer and David A. Solomon. Microsoft Press, 1997. *The NT book, if you want to learn about something other than* UNIX. *Of course, why would you? OK, we're kidding; you might actually work for Microsoft some day you know.*

[E95] "An Analysis of Decay-Usage Scheduling in Multiprocessors" by D.H.J. Epema. SIGMETRICS '95. *A nice paper on the state of the art of scheduling back in the mid 1990s, including a good overview of the basic approach behind decay-usage schedulers.*

[LM+89] "The Design and Implementation of the 4.3BSD UNIX Operating System" by S.J. Leffler, M.K. McKusick, M.J. Karels, J.S. Quarterman. Addison-Wesley, 1989. *Another OS classic, written by four of the main people behind BSD. The later versions of this book, while more up to date, don't quite match the beauty of this one.*

[M06] "Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture" by Richard McDougall. Prentice-Hall, 2006. *A good book about Solaris and how it works.*

[O11] "John Ousterhout's Home Page" by John Ousterhout. `www.stanford.edu/~ouster/`. *The home page of the famous Professor Ousterhout. The two co-authors of this book had the pleasure of taking graduate operating systems from Ousterhout while in graduate school; indeed, this is where the two co-authors got to know each other, eventually leading to marriage, kids, and even this book. Thus, you really can blame Ousterhout for this entire mess you're in.*

[P+95] "Informed Prefetching and Caching" by R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, J. Zelenka. SOSP '95, Copper Mountain, Colorado, October 1995. *A fun paper about some very cool ideas in file systems, including how applications can give the OS advice about what files it is accessing and how it plans to access them.*

[Y+18] "Principled Schedulability Analysis for Distributed Storage Systems using Thread Architecture Models" by Suli Yang, Jing Liu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. OSDI '18, San Diego, California. *A recent work of our group that demonstrates the difficulty of scheduling I/O requests within modern distributed storage systems such as Hive/HDFS, Cassandra, MongoDB, and Riak. Without care, a single user might be able to monopolize system resources.*