CS162 Discussion #6: Scheduling & More Networking

Announcements

- Project 1 code due Thursday, 10/06 @11:59pm (`submit proj1-code`)
  - Final design doc and group evals due Friday, 10/07 (`submit proj1-final-design`)
  - Project 2 starts once project 1 ends!
- Midterm next Thursday, 10/13 – review session TBA (probably next Monday)
- My office hours have moved from Friday to today, 4-5pm in 651 Soda for this week only

Scheduling

So far, we've talked about how to get threads to cooperate under the assumption that they'll pretty much all get to run at some point. But how does the OS actually decide how to pick which threads get to run at what time? It does this through scheduling. However, as usual, there are tradeoffs between different scheduling policies – depending on what our goals are for the scheduler, we'll want to use different scheduling algorithms. Here are some different criteria we can consider:

- *Response time*. The amount of time it takes to perform an operation (for example, showing a character after a key is pressed, etc.).
- *Throughput*. The number of operations performed per second.
- *Fairness*. A measure of how equally the CPU is shared among users/processes.

We have to make tradeoffs between these criteria; for example, minimizing response time might mean more context switching, but this will increase overhead and consequently lower throughput. Therefore, it's important to decide what scheduling criteria we want to follow and choose a scheduling policy that will achieve them. For each policy, we can compute the following parameters to determine how they differ (these can be computed per process, or as averages across all processes):

- *Waiting time*. The sum of all the chunks of time where the process is waiting on the ready queue, not running.
- *Completion time*. The amount of time it takes from the time the process arrives to the time it finishes, including both waiting and running time.

These are some policies we could use:

- *First Come First Served (FCFS, FIFO)*. Runs each process until completion, in the order they arrive.
- *Round Robin (RR)*. Gives each process a quantum of CPU time. If the process does not finish within that time quantum, it gets preempted and put back onto the end of the queue.
- *Shortest Job First (SJF, STCF)*. Runs the process which requires the least amount of computation first.
- *Shortest Remaining Time First (SRTF, SRTCF)*. Like SJF, but allows for preemption – switches to another process if that process has less time remaining to complete.

Note that SJF and SRTF are more difficult to implement than the others because they require some knowledge about the future to correctly determine which job to run. We can also try to condition our choices based off of previously observed behavior, for example, with an exponential averaging function or multi-level feedback scheduling.

Responding to Requests

We've talked about sending data across networks, but data isn't usually just arbitrarily sent from one point to another for no reason. Whoever's receiving all of this data needs to figure out what to do with it, and how to do this most efficiently. Not only that, one endpoint could be receiving data from more than one source at a time, and needs to be able to handle simultaneous requests. For example, a server (like your Go game server!) needs to deal with incoming requests from all of its clients, but it can choose to handle these requests in multiple ways. Here are two possible options:

- *One accept thread that spawns worker threads.* In this scheme, we have one thread that accepts all requests and then spawns a worker thread for each request. The worker threads then take the request they're assigned and handle it – all the accept thread does is accept and create threads. This option results in high overhead with creating new threads on the fly all the time, and doesn't really scale well (when lots of requests come in, you'll be creating tons and tons of threads). However, the upside is that all requests are handled independently and taken care of relatively quickly.
- *One accept thread with a thread pool.* In this scheme, we still have one thread that accepts all requests, but instead of spawning threads, we have a set of worker threads created beforehand that wait to handle requests. The accept thread pushes requests onto a queue, and threads from the thread pool pull those requests off the queue to handle. This strategy avoids the problems of overhead and scalability – we don't incur any overhead because we don't create any new threads, and the number of threads is constant and bounded. However, this also means that if all the worker threads are busy, any remaining requests on the queue may have to wait a long time to be handled.

Some Networking Principles

- *End-to-End Principle.* The End-to-End Principle states that functionality should be implemented at the hosts (endpoints) if possible, and to avoid putting functionality into the network itself. This makes the underlying network more flexible, and avoids having endpoints that don't need that extra functionality still suffering additional overhead or complexity.
- *Fate-Sharing Principle*. The Fate-Sharing Principle states that in a distributed system, information should be kept with the entities that the information is most important to. That way, if the information fails or is lost somehow, we know the entity that cares about must also be lost…in which case, it doesn't matter quite as much.