1 Short Answer

(a) Typically, warmups are used to control whether programs performing extensive disk I/O work are tested on cold or warm cache. For our purposes, we discard the first N runs since we want to test a program at consistent performance levels, we discard the runs completed during the "cold cache" phase and only use warm cache temperatures cycles (we are interested in the performance overtime after all and lengthy programs naturally lead to warm cache environments).

(b) From the lambda given in the question, we know it takes 150 seconds to serve each customer (i.e. 1/lambda where lambda = 150s). We also know that s = 90 seconds, which allows us to calculate our utilization, rho. Given all these variables, we can calculate the average completion time, Tq. Note, Tq takes longer than the length to serve a customer.

Rho = lamda * s = 1/150s * 90s = 3/5
Tq = s/(1-rho) = 90s/(1-3/5) = 225s

To improve on Tq, we can increase the number of cashiers taking orders (assuming we can handle more than one toast order at a time). Say we double the number of cashiers, then we have to calculate two new values K and C to find our new Tq.

$$K = \frac{\frac{lambda^0}{0!} + \frac{lambda^1}{1!}}{\frac{lambda^0}{0!} + \frac{lambda^1}{1!} + \frac{lambda^2}{2!}} = 0.8989$$

$$C = \frac{1 - K}{1 - Lambda * S * K/N} = 0.1384$$

Note, N = 2

$$T_q = \frac{Cs}{K(1-\rho)} + s = \frac{0.1384 * 90}{0.8989(1 - 3/5)} + 90 = 124.64\ seconds$$

(c) An example of setting a maximum time limit on a decision can be explored in a garbage-collecting service. For example, Gmail saves deleted information for 30 days before defaulting to permanently removing all deleted material. Such a practice decreases the amount of unnecessary memory managed on Google's servers while taking the reasonable action of deleting unwanted emails. This system provides a buffer for the user to reclaim lost emails while reducing resource computation.

(d) A Poisson probability distribution makes sense in this scenario because each user request sent to Amazon.com servers is discrete and independent of others. Although the demands are will

most likely rise on Black Friday, the Poisson distribution still appropriately models the requests; the distribution will simply be skewed in favour of more events happening on Black Friday than typically expected.

(e) Dead store elimination reduces code size and improves performance as a result of the unnecessary code pruning, which is indeed beneficial to the code. Similarly, dead store elimination saves time in areas of repeatedly overwriting variables with constant values (e.g. a while loop constantly executing x = 1 + 2 would be converted to a single assignment command. Despite these benefits, there are possible cases where the compiler scrubs data that the programmer intended to clear upon its last use; this leads to rising security concerns such as information leaks.

(f) The timing information may be inaccurate for 2 reasons: (1) the printing of megabytes to the terminal causes significant decrease in performance relative to the performance without printing to terminal and (2) measuring the performance of a program (e.g. time, memory, function frequency calls) typically causes program overhead (e.g. helgrind, valgrind, flame graph).

Despite /dev/null's incredible speed relative to printing output to the terminal, it still provides some overhead to build strings, send the data to the /dev/null directory, and have /dev/null drop all the imported data. Naturally, removing any code output would still perform faster after redirecting to /dev/null.

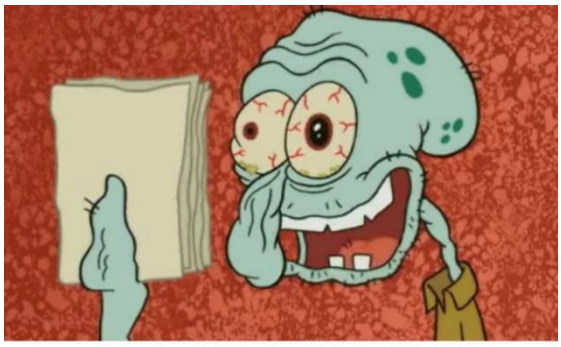
(g) An example for when shared memory is better to implement than message passing in inter-thread communication is when multiple writers are writing to a large buffer. In such an application, each thread can perform its own computation and write to a buffer in shared memory regulated by a mutex for all threads. Implementing this with message passing would be difficult as less processing is required for inter-thread communication and more of a collaborative framework is used to write to the buffer.

(h) A general-purpose memory allocator may only support one thread allocating or deallocating at a time, producing a bottleneck in a multithreaded context (i.e. scales poorly). One way to avoid this bottle neck would be to use a thread pool, which generates a pool of workers created once and only once.

(i) I would consider three aspects of the service: scalability, cost, and ease of migration. If the scalability of AWS' new service implementation is greater and I am indeed in search of a more-scalable system, then the service is more appealing. Cost is also a large factor because I might not want to add significant expenses to my operation, and if I can avoid these costs by simply improving my implementation, I will choose to not use the service. Finally, how easily I can migrate my data or code onto the service would affect my decision as well. Time is money and spending additional time moving all the "set up" involved in my implementation to AWS may not be worth it.

(j)

BONUS MEME:



Me submitting this final on 4 hours of sleep, 6 coffees, and a prayer.

2 Self-Optimizing Payment Processing

Code Overview for Junior

We want our system to handle the current sequential invoice processing much quicker. One way to do this is to parallelize the code, and here's how you're going to do it. We want a multi-threaded system using a queue in shared memory to process our invoices in order. This system overall isn't so difficult to implement, but what we have to consider is our third-party's rate-limiting service. It only allows us to process so many invoices within a given time interval before we register it as a denial-of-service attack. With increasing denial-of-service registrations, however, we have to reduce the level of parallelism.

Details

In detail, generate at most 10 threads to get and process each batch of invoices. Divide the number of threads evenly to get and process invoices. The get threads will call get_next_invoice_batch() and add it to the programs shared memory queue in order. Note, you will have to parallelize the get_next_invoice_batch() function by passing in a fixed set of invoices (e.g. 5). Note, you will have to use a mutex to control how threads enqueue invoices once they are obtained. Now that we have our threads building the queue, we must begin the processing threads. These threads will dequeue the latest invoice, process its content, and mark the payment as paid or failed. Using a queue is helpful, as we pop each element from shared memory, other threads cannot access the already popped element; this saves us from needing to regulate the data structure in the event of multiple threads accessing the same element. Once this implementation is done, we can focus on the rate-limiting aspect of the problem. Since there are at most 10 threads for the program's read and write functionality, we scale the number of threads down based on the number of Err returns we get from our third party software, try_to_pay(). If the third-party software starts to return errors, scale the number of working threads down by 2 (one reader and one writer) for every 10 consecutive errors (obviously from below by 0 which is just the sequential implementation). The probability of failing 10 requests is moderately low for our systems, so we limit the number of false positive rate-limiting errors while maximizing performance. We increase the number of threads once again by 2 when we receive 10 consecutive successful invoice processing. I realize waiting 10 consecutive successions causes considerable overhead, but it would be worse to swiftly return to our rate-limiting state and repeatedly process invoices initially written off due to rate-limiting. For invoices that were discarded (10 consecutive fails) due to rate-limiting, add them to our bucket as backlog, which will be processed well below our proprietary rate limit overnight.