COMP90015 Distributed Systems

Socket Model and Threading Paradigms

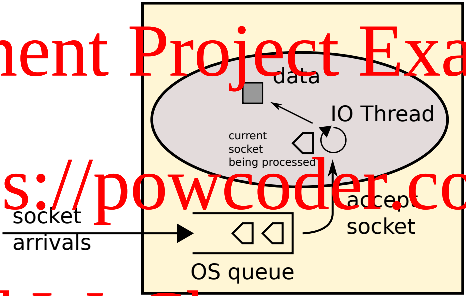
Aaron Harwood

School of Computing and Information Systems

2022 Semester II

1. Socket Model
2. Queueing Theory
3. Threading Paradigms

Essential aspects I



The essential elements of modeling a socket include:

- *incoming socket connections* – communication packets such as used in the TCP protocol, that are requesting that a socket connection be established.
    - The client initiating the request will time out if the request is lost, e.g. by network errors or if the OS of the server drops the request due to lack of resources, and possibly retry several times. This can be a significant source of delay when using connection oriented protocols like TCP over an unreliable network.

Essential aspects II

- Loss of the request in the network is more of a problem than the OS dropping the request since the OS can at least signal the sender that the request was dropped, whereas in the former case the sender will wait for the entire timeout period before it deems the request to be lost.

- *OS queue* – If the socket connection is valid, i.e. to a port that a process is bound to, then the socket connection is put into an OS queue.
  - The OS queue is generally a finite capacity queue, to avoid consuming an unbounded amount of memory, where the OS will offer the request to the queue so that the OS thread does not block, to avoid blocking the OS and/or allocating an unbounded number of OS threads.
  - If the queue is full, or the port is invalid (no process is bound to it), then the socket connection is dropped (lost). The OS may signal the receiver that the connection is refused or dropped in this case.
  - Some OSes have a process like the UNIX `xinetd` process that accepts connections on many of the well known port numbers and starts the relevant server process that is usually listening on that port. This is preferable to having many different server processes started in advance, in the case when connections are infrequent, since even idle processes represent OS overheads.
  - Some OSes let multiple processes accept connections on the same port. All such processes simply make use of the same OS queue.

- *IO thread* – If a process binds to a port for socket based communication then it needs to have at least one thread that accepts socket connections and processes them, e.g. accessing local data, reading/writing to the socket.

Essential aspects III

- At any one time the IO thread is either processing a socket connection, or is blocked (idle) while waiting for a new connection on the queue. The fraction of time that the thread spends processing a socket connection is called the thread's *utilization*.
- The IO thread will usually take from the OS queue, blocking if there are no sockets waiting on the queue. Of course, the IO thread can set a timeout so that it can do other work at regular intervals or can poll the queue rather than block and wait, which will return even if a socket was not waiting on the queue.
- If the thread does not process incoming socket requests fast enough, then the socket requests will start dropping at the OS, i.e. they will be lost, and the client may retry the request later.

Queueing theory

The Socket Model can be described using queueing theory:

- The socket connection requests arrive at a mean rate of λ requets per second, and we can use an Exponential distribution with parameter λ, similar to failure rate models, to describe the probability of a request arriving within the next t seconds, which we recall is the cumulative distribution function of the Exponential distribution.
- The time taken by the IO thread to process a socket connection request, which includes the time from when the IO thread takes the socket off the queue to the time that the socket is closed (no other threads are considered at this point), can be modeled as well using an Exponential distribution with parameter μ, called the service rate, i.e. the mean number of socket connection requests that the IO thread can process per second.
- If the queue has a finite capacity of $k-1$ socket connection requests (and a further 1 socket connection possibly being currently processed by the IO thread), making a maximum of k socket connections in the system at any one time, then the essential aspects of the Socket Model are described using a $M/M/1/k$ queue.

Queueing theory results I

https://sites.pitt.edu/~dtipper/2130/2130_Slides4.pdf

Without looking at deriving queueing theory results, some of the most relevant results are:

- Since the queue has finite capacity $k-1$ it can sometimes become full and new socket requests received by the OS will have to be dropped. The drop rate is equivalently the blocking probability of the queue:

$$P_{drop} = \begin{cases} \frac{\rho^k(1-\rho)}{1-\rho^{k+1}} & \rho \neq 1 \\ \frac{1}{1+k} & \rho = 1 \end{cases}$$

  - For constant $\rho < 1$, if $k \to \infty$ (unbounded queue) then the drop rate approaches 0.

- The portion of socket requests dropped is λP_{drop} and so we consider the effective socket request rate that the IO thread sees as $\lambda_{eff} = \lambda(1 - P_{drop})$.
- The effective thread utilization is then $\frac{\lambda_{eff}}{\mu}$, which is the fraction of time that the thread will be busy, rather than idle (waiting for a request to arrive), and the system is said to be *stable*.
  - When $\lambda_{eff} \geq \mu$ then the thread will eventually be busy 100% of the time, meaning it will eventually fall behind, and the system is said to be *unstable*.

Queueing theory results II

https://sites.pitt.edu/~dtipper/2130/2130_Slides4.pdf

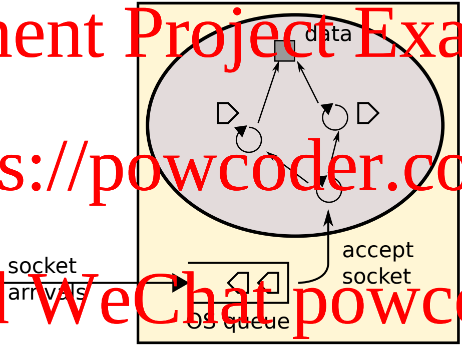
- The average number of socket requests in the system becomes:

$$L = \begin{cases} \frac{\rho}{1-\rho} - \frac{(k+1)\rho^{k+1}}{1-\rho^{k+1}} & \rho \neq 1 \\ \frac{k}{2} & \rho = 1 \end{cases}$$

  - The length of the queue is slightly smaller than the average number of sockets requests in the system because sometimes one of those requests is being processed by the IO thread and so the queue length becomes $L_q = L - \rho_{eff}$.
  - For constant $\rho < 1$, if $k \to \infty$ then this becomes $\frac{\rho}{1-\rho}$.
  - Notice that in either case the mean queue length sharply increases as $\rho \to 1$, as it reaches a vertical asymptote. This is why we prefer using a bounded queue, since memory consumption can sharply increase under high load – *even if the load does not exceed the capacity of the server*.

- The average time that a socket request spends in the system, including the time taken by the IO thread to process the socket request (i.e. to close the socket), is $W = \frac{L}{\lambda_{eff}}$.
  - The average time that a socket request spends waiting in the queue is $W - \frac{1}{\mu}$.

Thread-per-connection I



The thread-per-connection paradigm creates a new thread for every socket connection:

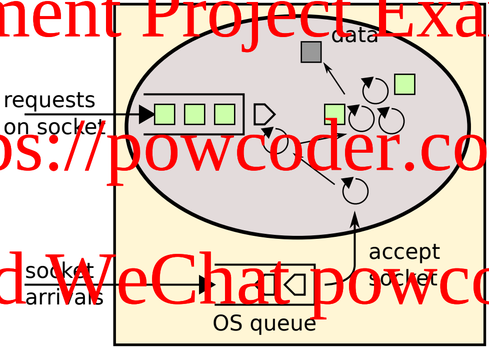
- Creating more threads is one way to utilize more cores of a multi-core server.

Thread-per-connection II

- If the maximum number of threads allowed to be created by the IO thread is c then the system is described by an $M/M/c/k$ queue, which will provide different results to what we have seen so far. In this case the IO thread is not considered to be doing any useful work, i.e. it never actually processes a socket connection itself.
- As the total number of threads on the machine increases, potentially going beyond the total number of cores (or processing unit if hyper-threading is used), then the rate of context switching and context state space increases as well. Context switching happens whenever a thread blocks and another thread is started on the core or processing unit of the blocked thread. Excessive context state needs to be stored in cache and pushes useful data out of the cache.
- Multiple threads leads to concurrency control requirements. Concurrency control leads to further context switching and synchronization requirements between threads.
- Lock-free designs can greatly reduce the amount of context switching required, especially by using special machine instructions, however lock-free designs cannot block and therefore data in a distributed system must be dropped when the distributed systems' resources are exceeded, which is not necessarily desired.

Thread-per-request I

Depicted combined with thread-per-connection



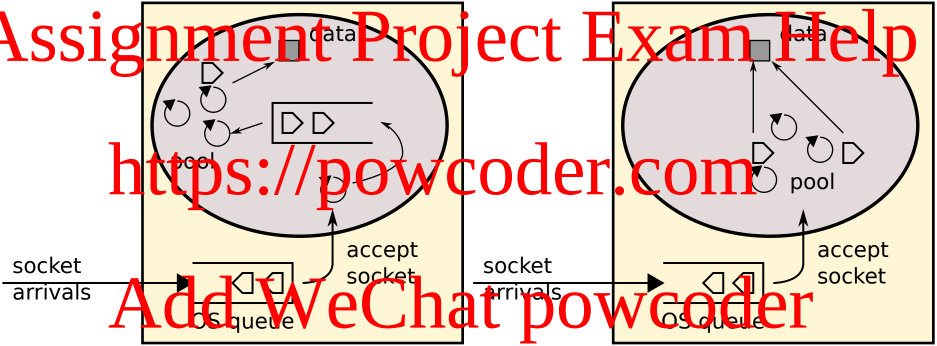
The thread-per-request paradigm goes further and allows the thread that is processing the socket to create threads for each request received on the socket:

Thread-per-request II

Depicted combined with thread-per-connection

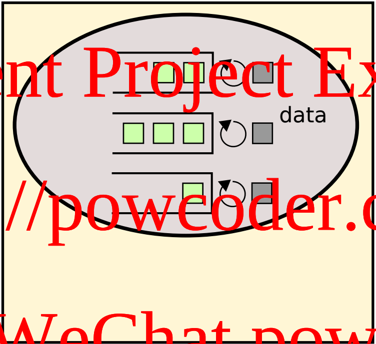
- Even more threads are potentially created than the thread-per-connection paradigm alone.
- Some protocols allow only a single request per socket connection and so thread-per-request is not useful in this case.
- Requests may have dependencies between them, in that the order that the requests are processed may be important.
- If many requests on the socket can be bundled into one larger request as one item on the queue, then this can reduce context switching by an amount proportional to the size of the bundles.

Thread pool



The thread-pool paradigm creates a fixed or dynamically resizable set of threads that either take incoming socket connection requests from a process maintained queue, or, if the accept socket API is thread safe then the pool of threads can take directly from the OS queue.

Thread-per-object paradigm



The thread-per-object paradigm creates a thread for each data object. This can potentially reduce cache miss rates in the machine since for a given data object, only 1 thread ever accesses it and it will reside only in the cache for that thread. This can greatly increase cache efficiency which can significantly improve overall performance of the machine.

Discussion Question

Question (1): Some people argue that using *single-thread-per-process* is better than *multi-thread-per-process*, i.e. using x single threaded processes is better than using 1 multi-threaded process with x threads. Critically compare the two approaches and discuss what are the main aspects of your comparison.