System Performance and Highly Concurrent Systems

Sam Kumar
CS 162: Operating Systems and System Programming
Lecture 14
https://inst.eecs.berkeley.edu/~cs162/su20

Read: A&D 7.5,
SEDA Section 2
Recall: Deadlock

• Starvation vs. Deadlock
  • Starvation: Thread indefinitely unable to make progress
  • Deadlock: Thread(s) unable to make progress due to circular wait

• Four conditions for deadlock:
  • Mutual exclusion
  • Hold and wait
  • No preemption
  • Circular Wait

• Three different approaches to address deadlock:
  1. **Deadlock avoidance**: dynamically delay resource requests so deadlock doesn’t happen
  2. **Deadlock prevention**: write your code in a way that it isn’t prone to deadlock
  3. **Deadlock recovery**: let deadlock happen, and then figure out how to recover from it
  4. Or **deadlock denial**: ignore the possibility of deadlock in applications
System Performance

• “Back of the Envelope” calculation and modeling
• Get the rough picture first... and don’t lose sight of it
Times (s) and Rates (op/s)

- **Latency** – time to complete a task
  - Measured in units of time (s, ms, us, ..., hours, years)

- **Response Time** - time to initiate and operation and get its response
  - Able to issue one that *depends* on the result
  - Know that it is done (anti-dependence, resource usage)

- **Throughput** or **Bandwidth** – rate at which tasks are performed
  - Measured in units of things per unit time (ops/s, GLOP/s)

- **Performance???
  - Operation time (4 mins to run a mile...)
  - Rate (mph, mpg, ...)

7/15/2020
Kumar CS 162 at UC Berkeley, Summer 2020
Sequential Server Performance

- Single sequential “server” that can deliver a task in time $L$ operates at rate $\leq \frac{1}{L}$ (on average, in steady state, ...)
  - $L = 10 \text{ ms} \rightarrow B = 100 \text{ op/s}$
  - $L = 2 \text{ yr} \rightarrow B = 0.5 \text{ op/yr}$

- Applies to a processor, a disk drive, a person, a TA, ...
Single Pipelined Server

- Single pipelined server of $k$ stages for tasks of length $L$ (i.e., time $T/L$ per stage) delivers at rate $\leq k/L$.
  - $L = 10$ ms, $k = 4 \rightarrow B = 400 \text{ op/s}$
  - $L = 2$ yr, $k = 2 \rightarrow B = 1 \text{ op/yr}$
Example Systems “Pipelines”

- Anything with queues between operational process behaves roughly “pipeline like”
- Important difference is that “initiations” are decoupled from processing
  - May have to queue up a burst of operations
  - Not synchronous and deterministic like in 61C
Multiple Servers

- $k$ servers handling tasks of length $L$ delivers at rate $\leq \frac{k}{L}$.
  - $L = 10 \text{ ms}, \; k = 4 \rightarrow B = 400 \text{ op/s}$
  - $L = 2 \text{ yr}, \; k = 2 \rightarrow B = 1 \text{ op/hr}$

- In 61C you saw multiple processors (cores)
  - Systems present lots of multiple parallel servers
  - Often with lots of queues
Example Systems “Parallelism”

I/O Processing

User Process → syscall → File System → Upper Driver → Lower Driver

Communication

Parallel Computation, Databases, ...
A Simple Systems Performance Model

Bandwidth ($B$): Rate, Op/s
- e.g., flow: gal per min

Latency ($L$): time per op
- How long does it take to flow through the system

“Service Time”

If $B = 2 \text{ gal/s}$ and $L = 3 \text{ s}$
How much water is “in the system?”
A Simple Systems Performance Model

Latency ($L$): time per op
- How long does it take to flow through the system

“Service Time”

Bandwidth ($B$): Rate, Op/s
- e.g., flow: gal per min

If $B = 2 \text{ op/s}$ and $L = 3 \text{ s}$
How many ops are “in the system?”

Little’s Law

• The number of “things” in a system is equal to the bandwidth times the latency (on average)

\[ n = L B \]

• Applies to any stable system (arrival rate = departure rate)

• Can be applied to an entire system:
  • Including the queues, the processing stages, parallelism, whatever

• Or to just one processing stage:
  • i.e., disk I/O subsystem, queue leading into a CPU or I/O stage, ...
A Simple Systems Performance Model

Latency ($L$)

Operation Time: $t$

Service Rate: $\mu$

Request Rate: $\lambda$

Queuing delay: $d$

The maximum service rate $\mu_{max}$ is a property of the system – the “bottleneck”

Utilization: $\rho = \frac{\lambda}{\mu_{max}}$
Ideal System Performance

• How does $\mu$ (service rate) vary with $\lambda$ (request rate)?

Request Rate ($\lambda$) - “offered load”

Service Rate ($\mu$) - “delivered load”

$\mu_{\text{max}}$

asymptotic peak rate

$\mu_{\text{max}}$
Two Related Questions

The maximum service rate $\mu_{max}$ is a property of the system – the “bottleneck”

What determines $\mu_{max}$?

Service Rate: $\mu$

Request Rate: $\lambda$

Latency ($L$)

Operation Time

Queuing delay: $d$

What about “internal” queues?

Utilization: $\rho = \frac{\lambda}{\mu_{max}}$
Bottleneck Analysis

Overall System: Series of Stages

Request Rate: $\lambda$

Service Rate: $\mu$

$L$ $L$ $L$ $L$

...
Bottleneck Analysis

• Each stage has its own queue and maximum service rate
• Suppose the green stage is the bottleneck
Bottleneck Analysis

• Each stage has its own queue and maximum service rate
• Suppose the green stage is the bottleneck
• The bottleneck stage dictates the maximum service rate $\mu_{\text{max}}$

System Model: Bottleneck Stage

Request Rate: $\lambda$

Service Rate: $\mu$

$\mu_{\text{max}} = \mu_{\text{max},3}$
Example: Servicing a Highly Contended Lock

Queue of waiting threads

$\mu_{\text{max}} = \frac{1}{X}$

Critical section guarded by lock

$X \text{ sec in critical section}$

All try to grab lock

Time $= p \cdot X \text{ sec}$

Rate $= \frac{1}{X} \text{ ops/sec}$, regardless of # cores
Two Related Questions

Latency ($L$)

Operation Time

Queuing delay: $d$

Request Rate: $\lambda$

Tank represents queue of bottleneck stage
- Including queues of previous stages, in case of backpressure

The maximum service rate $\mu_{\text{max}}$ is a property of the system – the “bottleneck”

Utilization: $\rho = \frac{\lambda}{\mu_{\text{max}}}$

$\mu_{\text{max}}$ is service rate of bottleneck stage

Service Rate: $\mu$
A Simple Systems Performance Model

Latency ($L$)
Operation Time
Service Rate: $\mu$
Request Rate: $\lambda$

Queuing delay: $d$

The maximum service rate $\mu_{\text{max}}$ is a property of the system – the "bottleneck"

Utilization: $\rho = \frac{\lambda}{\mu_{\text{max}}}$

Useful to apply this model to:
- Bottleneck stage
- Entire system up to and including bottleneck stage
- Entire system
Rest of Today’s Lecture

Using this system model, we will:

• Explore latency in more depth
• Discuss how to build systems that perform well under load
Announcements

• Homework 3 is due on Friday

• Change in policy: if you use a slip day on project, your final report deadline also gets pushed back a day
Rest of Today’s Lecture

Using this system model, we will:

- **Explore latency in more depth**
- Discuss how to build systems that perform well under load
Latency (Response Time)

• Total latency (response time): queuing time + service time
• Service time depends on the underlying operation
  • For CPU stage, how much computation
  • For I/O stage, characteristics of the hardware
• What about the queuing time?
A Simple Systems Performance Model

Request Rate: $\lambda$

Service Rate: $\mu$

Operation Time: $t$

Latency ($L$)

Queuing delay: $d$

The maximum service rate $\mu_{\text{max}}$ is a property of the system – the “bottleneck”

Utilization: $\rho = \frac{\lambda}{\mu_{\text{max}}}$

When will the queue(s) start to fill?
Queuing

• What happens when request rate ($\lambda$) exceeds max service rate ($\mu_{max}$)?
• Short bursts can be absorbed by the queue
  • If on average $\lambda < \mu$, it will drain eventually
• Prolonged $\lambda > \mu \rightarrow$ queue will grow without bound
A Simple, Deterministic World

- $T_A$: time between arrivals
  - $\lambda = \frac{1}{T_A}$
- $T_S$: service time
  - $\mu = \frac{k}{T_S}$
- $T_Q$: queuing time
  - $L = T_Q + T_S$

- Assume requests arrive at regular intervals, take a fixed time to process, with plenty of time between ...
A Simple, Deterministic World

Utilization ($\rho = \frac{\lambda}{\mu} = \frac{T_s}{T_A}$)

Delivered Throughput

Queue delay

time

Queue delay

time

Saturation

Empty Queue

Unbounded
A Bursty World

- $T_A$: time between arrivals
  - Now, a random variable
- $T_S$: service time
  - $\mu = \frac{k}{T_S}$
- $T_Q$: queuing time
  - $L = T_Q + T_S$

- Requests arrive in a burst, must queue up until served
- Same average arrival time, but almost all of the requests experience large queue delays (even though average utilization is low)
How to model burstiness of arrival?

• $T_A$, the time between arrivals, is now a random variable
  • Elegant mathematical framework if we model it as an exponential distribution
  • Probability distribution function of an exponential distribution with parameter $\lambda$ is $f(x) = \lambda e^{-\lambda x}$

“Memoryless”: Likelihood of an event occurring is independent of how long we’ve been waiting

Lots of short arrival intervals (i.e., high instantaneous rate)

Few long gaps (i.e., low instantaneous rate)
A Simple Systems Performance Model

Request Rate: $\lambda$

Service Rate: $\mu$

Operation Time: $t$

Latency ($L$)

Queuing delay: $d$

Queue grows at rate $\mu - \lambda$

After time $t$, it will have grown to length $t(\mu - \lambda)$

Utilization: $\rho = \frac{\lambda}{\mu_{\text{max}}}$
Server spends variable time \( (T) \) with customers

- Mean (Average): \( m = \sum p(T) \cdot T \)
- Variance (stddev\(^2\)): \( \sigma^2 = \sum p(T) \cdot (T - m)^2 = \sum p(T) \cdot T^2 - m^2 \)
- Squared coefficient of variance: \( C = \frac{\sigma^2}{m^2} \)

Important values of \( C \):

- No variance or deterministic \( \Rightarrow C = 0 \)
- “Memoryless” or exponential \( \Rightarrow C = 1 \)
  - Past tells nothing about future
  - Poisson process – purely or completely random process
  - Many complex systems (or aggregates)
    are well described as memoryless
Introduction to Queuing Theory

- Queuing Theory applies to long term, steady state behavior
  - Arrival rate = Departure rate

- Arrivals characterized by some probabilistic distribution

- Departures characterized by some probabilistic distribution
Our Goals with Queuing Theory

• We wish to compute:
  • $T_Q$: Time spent in queue
  • $L_Q$: Length of the queue
Little’s Law Applied to a Queue

• Before, we had \( n = LB \) (for a stable system):
  • \( B \): bandwidth
  • \( L \): latency
  • \( n \): number of operations in the system

• When applied to a queue, we get:

\[
L_Q = \lambda T_Q
\]
Our Goals with Queuing Theory

• We wish to compute:
  • $L_Q$: Length of the queue
  • $T_Q$: Time spent in queue
Some Results from Queuing Theory

• Assumptions: system in equilibrium, no limit to the queue, time between successive arrivals is random and memoryless

![Diagram showing a queue and server with arrival rate $\lambda$, service rate $\mu = 1/T_S$, and utilization $\rho = \lambda/\mu$]

• $\lambda$: arrival rate
• $T_S$: mean time to service a customer
• $C$: squared coefficient of variance ($\sigma^2/T_S^2$)
• $\mu$: service rate ($1/T_S$)
• $\rho$: utilization ($\lambda/\mu$)
Some Results from Queuing Theory

• **Memoryless service distribution** ($C = 1$)—an “M/M/1 queue”:
  • $T_Q = \frac{\rho}{1-\rho} \cdot T_S$

• **General service distribution** (no restrictions)—an “M/G/1 queue”:
  • $T_Q = \frac{1+C}{2} \cdot \frac{\rho}{1-\rho} \cdot T_S$

• $\lambda$: arrival rate
• $T_S$: mean time to service a customer
• $C$: squared coefficient of variance ($\sigma^2/T_S^2$)

• $\mu$: service rate ($1/T_S$)
• $\rho$: utilization ($\lambda/\mu$)

Kumar CS 162 at UC Berkeley, Summer 2020
Key Results from Queuing Theory

\[ T_Q = \frac{\rho}{1-\rho} \cdot T_S \]  (memoryless service distribution)

\[ L_Q = \lambda T_Q \]  (by Little’s Law)

Utilization is \( \rho = \frac{\lambda}{\mu_{max}} = \lambda T_S \), so

\[ L_Q = \lambda T_Q = \frac{\rho}{T_S} \cdot T_Q = \frac{\rho^2}{1-\rho} \]  (for a single server)
Ideal System Performance

Request Rate ($\lambda$) - "offered load"

Service Rate ($\mu$) - "delivered load"

Latency ($\lambda$)

Operation Time

Time

$\mu_{max}$

Service Rate ($\mu$) - "delivered load"

Operation Time

$\mu_{max}$

$T_Q \sim \frac{\rho}{1-\rho}$, $\rho = \frac{\lambda}{\mu_{max}}$

Why does latency blow up as we approach 100% utilization?
- Queue builds up on each burst
- But very rarely (or never) gets a chance to drain

"Half-Power Point": load at which system delivers half of peak performance
- Design and provision systems to operate roughly in this regime
- Latency low and predictable, utilization good: ~50%
Break (If Time)
Rest of Today’s Lecture

Using this system model, we will:
• Explore latency in more depth
• Discuss how to build systems that perform well under load
Ideal System Performance

- A system that behaves this way is **well-conditioned**
  - Delivered load increases with offered load until pipeline saturates
  - As offered load increases further, throughput remains high

- Request Rate ($\lambda$) - “offered load”
- Service Rate ($\mu$) - “delivered load”
- $\mu_{max}$
- Asymptotic peak rate

$$\lambda$$

7/15/2020  Kumar CS 162 at UC Berkeley, Summer 2020
Recall: Sockets with Protection and Concurrency

Client

1. Create Client Socket
2. Connect it to server (host:port)
3. Connection Socket
   - write request
   - read response
4. Close Client Socket

Server

1. Create Server Socket
2. Bind it to an Address (host:port)
3. Listen for Connection
4. Accept syscall()
5. Connection Socket
   - read request
   - write response
6. Child
   - Close Listen Socket
   - Close Connection Socket
7. Parent
   - Close Connection Socket
   - Close Server Socket
Recall: Server Protocol (v3)

// Socket setup code elided...
while (1) {
    // Accept a new client connection, obtaining a new socket
    int conn_socket = accept(server_socket, NULL, NULL);
    pid_t pid = fork();
    if (pid == 0) {
        close(server_socket);
        serve_client(conn_socket);
        close(conn_socket);
        exit(0);
    } else {
        close(conn_socket);
        // wait(NULL);
    }
}
close(server_socket);
Recall: Sockets with Concurrency, without Protection

Client
- Create Client Socket
- Connect it to server (host:port)
  - Connection Socket
  - write request
  - read response
- Close Client Socket

Server
- Create Server Socket
- Bind it to an Address (host:port)
- Listen for Connection
  - Accept syscall()
  - spawn new thread
- Close Server Socket

Connection Socket
- read request
- write response
Non-Well-Conditioned Systems

• A server that spawns a new pthread per request is not well-conditioned!

• Figure from SEDA Section 2 reading (Welsh 2001)

Figure 2: Threaded server throughput degradation: This benchmark measures a simple threaded server which creates a single thread for each task in the
Building Well-Conditioned Systems

• Spawning a new thread or process for each request is not well-conditioned

• Too many threads is bad
  • Scheduling overhead becomes large
  • Context switch overhead becomes large
    • E.g., Poor cache performance
  • Synchronization overhead becomes large
    • E.g., Lock contention

• Was our original (v1) server well-conditioned?
  • The one that handles requests one at a time, with no concurrency?
Building Well-Conditioned Systems

1. Thread Pools
2. User-Mode Threads
3. Event-Driven Execution

We’ll discuss these next time...