KODEWERK

Java™ Performance Diagnostic Methdology Workshop

Workshop Format

- Goal is to learn how to effectively identify root causes of poor performance
- Mix of lectures and hands on labs
  - ask questions and experiment
- Breaks as needed
- Don’t lets others steal your attention
  - turn off email
  - turn off messengers
  - turn off cell phone

Performance Tuning

↑Work = Fn(↓resources, ↓time)

- **Resources:** inputs and machinery needed to produce the desired outputs
- **Time:** two measures, latency and throughput
  - Latency: time needed to convert inputs into desired outputs
  - Throughput: rate at which desired outputs are produced
- Performance Tuning is making adjustments to the system to reduce the time and resources needed to convert inputs into outputs

Performance Tuning Skills

- Creativity
  - Devise new algorithms or techniques that
    - get the job done
    - improve efficiency
      - minimize resource consumption
      - reduce latency

- Diagnostic
  - Systematic approach to troubleshooting
    - reliably and repeatedly characterize underlying pathology
  - determine the source of performance bottlenecks and regressions

Performance Diagnostics

- With all performance talks we need to cover
  - **technology**
    - dive into or technology stack to better understand how things work
  - **tooling**
    - gives us a visualization of how the technology stack is functioning
  - **methodology**
    - provides us with guidance on what to look for, where to start, and how to proceed
- Make use of Specifications and Models

Specifications and Models

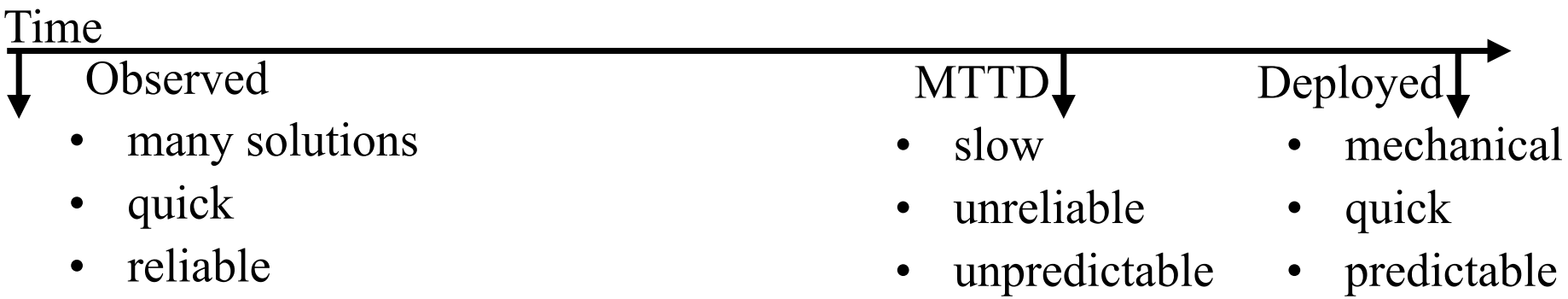
- Specifications define how a things are suppose to work
  - reading specifications helps us understand
    - what the component can do
    - how to best use the component
    - what is normal and is a bug
  - different implementations may vary in behavior in subtile ways
  - our duty as professionals to read specifications

- Models are an abstraction of a real system
  - simplification that can help us understand how the real system works
  - simplification that ignores the complexities in real systems in a useful way
  - building models is a useful way of improving understanding
    - helps us deal with the complexities by focusing on fundamental truths

  - Tool to help us combat Mean Time to Resolution

Mean Time To Resolution

- MTTR is the time from first observation to time when the regression is resolved
  - broken down into 3 steps
    - observed
    - diagnosed
    - repaired and deployed



- many solutions
- quick
- reliable

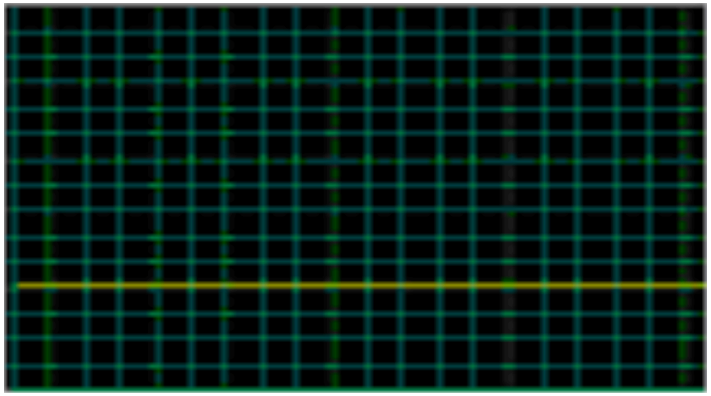
- slow
- unreliable
- unpredictable

- mechanical
- quick
- predictable

Exercise 1

CPU Usage History

Page File Usage History



- Web application running on Windows server
  - Users are complaining about slow response times for simple tasks
- HTTP/Servlet
  - supports up to 20 users
  - makes use of an Oracle DB running on a separate server
    - Oracle JDBC driver/Hibernate
    - dedicated network
    - DBA reports sum millisecond response times
    - Network admin report minimal network saturation and latencies
  - no disk activity
  - CPU and page file usage charts were captured
- Task: Reason through the data to determine the most likely source of the regression

Exercise 1 Redux

- Exercise left us with many choices
  - but no clear solution
- Each choice adds uncertainty
  - impacts schedule as each choice must be explored
  - dilates MTTR
- Even if a fix yields better performance, did it address the bottleneck?
  - more on this when we look at Theory of constraints

Measure Don’t Guess

- Hypothesis free investigation
  - progress through a series of steps to arrive at a conclusion
    - all decisions supported by measurements
  - focus on tuning what matters

- Beware of tools
  - all tools are biassed
    - is that bias hiding the bottleneck
  - they always will point to something
    - does that something matter?

- Beware of your own biases
  - biases clearly plays a role in the diagnosis of Exercise 1
  - GC is rarely the source of performance bottlenecks

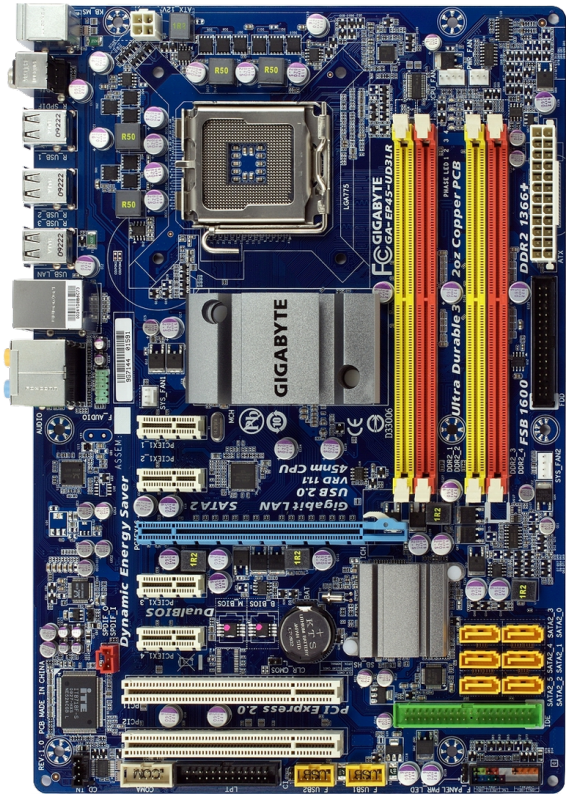
Diagnostic Model

- Build a model
  - Java Performance Diagnostic Model
- Model becomes the basis for Java Performance Diagnostic Methodology (jPDM)
  - a methodical process used to study the performance of a system
    - framework for driving decisions (hypothesis free)
  - basis for a classifier that can differentiate between distinct categories of performance bugs
  - identify key measurements
    - defines requirements for tooling
- Model is based on
  - the components that make up a system
  - how the components behave (specifications)
  - how the components interact (specifications)

The Components of a System

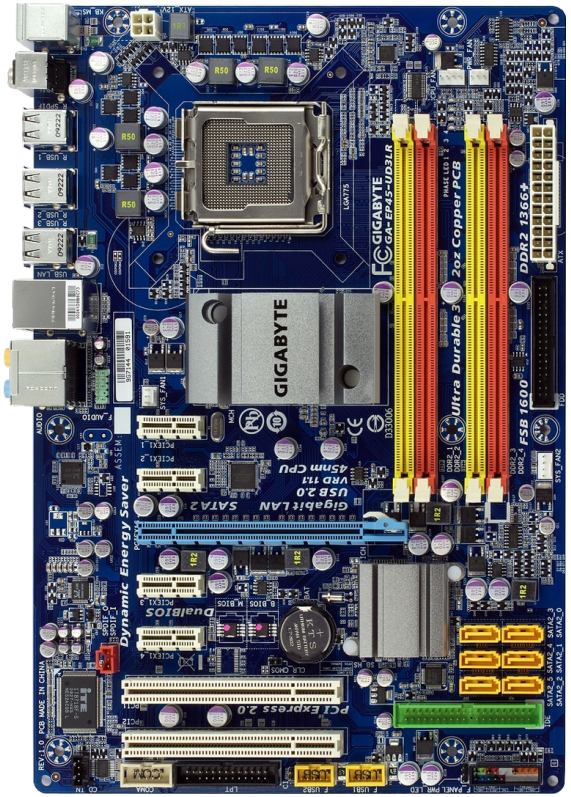
- Hardware is real
  - creates a real physical constraint
  - Max throughput requires perfect conditions
  - every individual component is not shareable

- Properties
  - cardinality
    - how many/much is available?
  - latency
    - what is time to first response?
  - throughput
    - what is the rate of output?
  - granularity
    - What is the data chunk size?



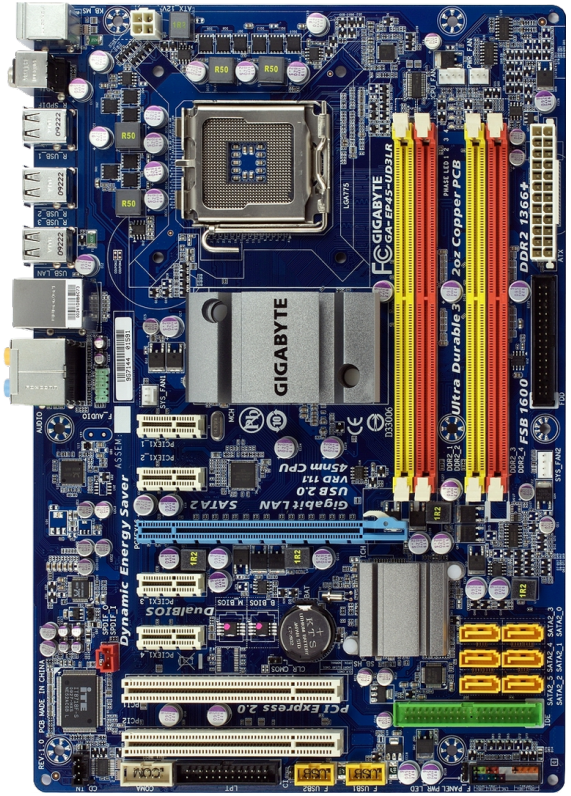
CPU

- Cardinality
  - number of cores
  - number of units in the ALU
- Latency
  - clock speed
  - time to read/write data (latency of QPI and caches)
- Throughput
  - CPI/IPC (typically 4)
- Granularity
  - cache line size (typically 64 bytes)



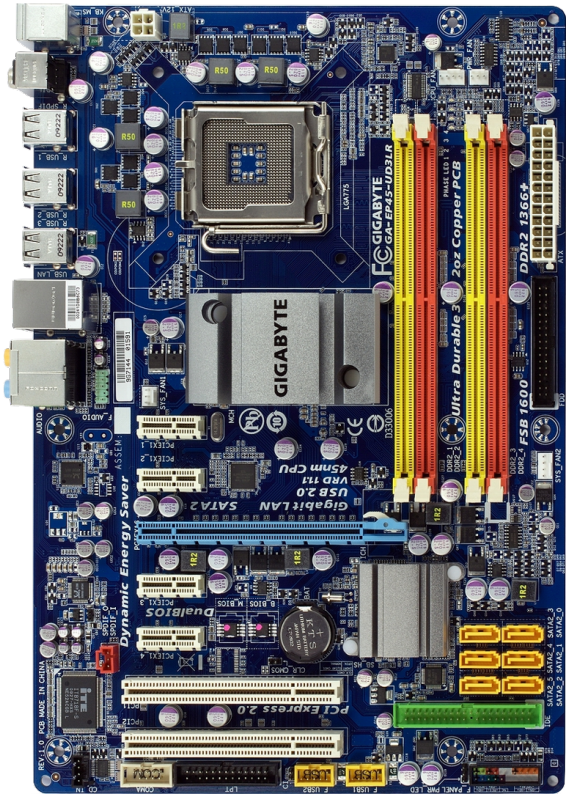
Memory

- Cardinality
  - number of bytes of RAM
  - number of buses on mother board
- Latency
  - clock speed
  - CAS
- Throughput
  - bus speed
  - bus availability
- Granularity
  - Column size
  - bus width



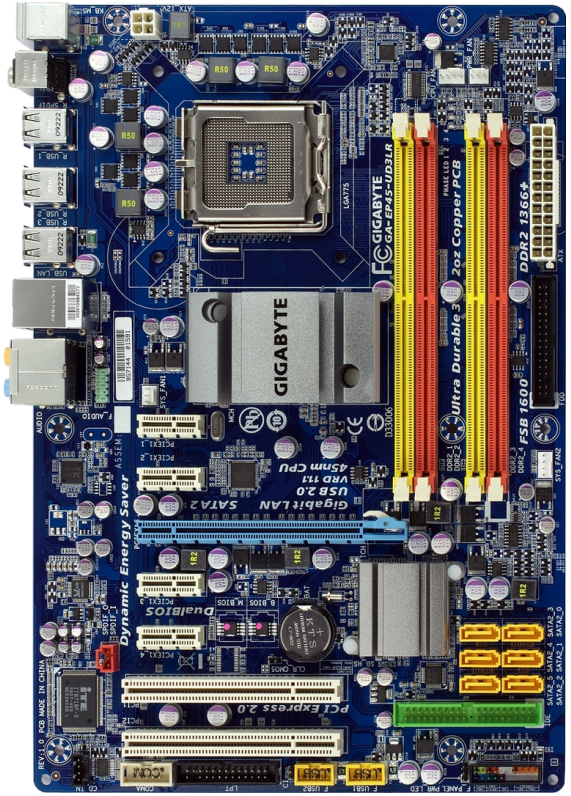
Disk

- Cardinality
  - number of bytes of storage
  - number of I/O channels
- Latency
  - controller clock speed
  - sweep arm speed (and position)
- Throughput
  - IOPS * block size
  - SATA ~2400Gbits/sec
- Granularity
  - disk sector size (block size)
  - bus width



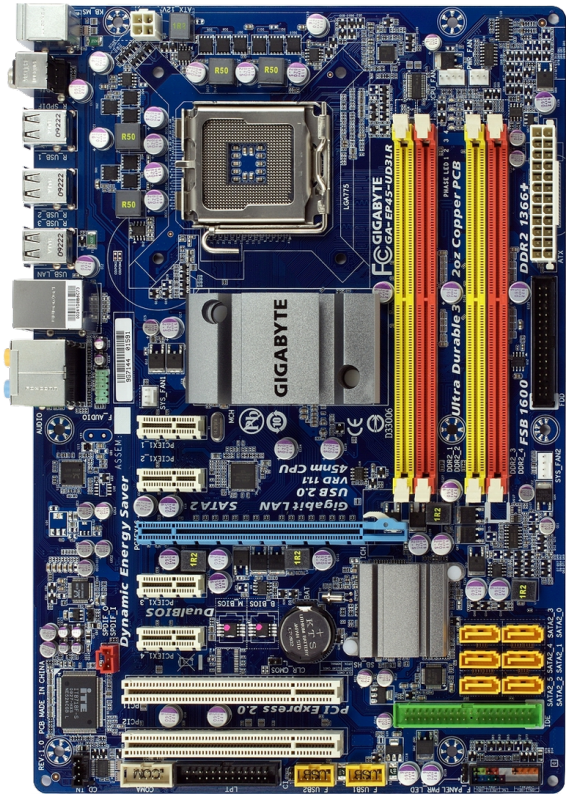
Network

- Cardinality
  - number of network cards
  - number of frame buffers
- Latency
  - time to first payload
- Throughput
  - bits transferred per second
  - clocks
  - many other things
- Granularity
  - packet payload size
  - TCP is typically 1500 bytes



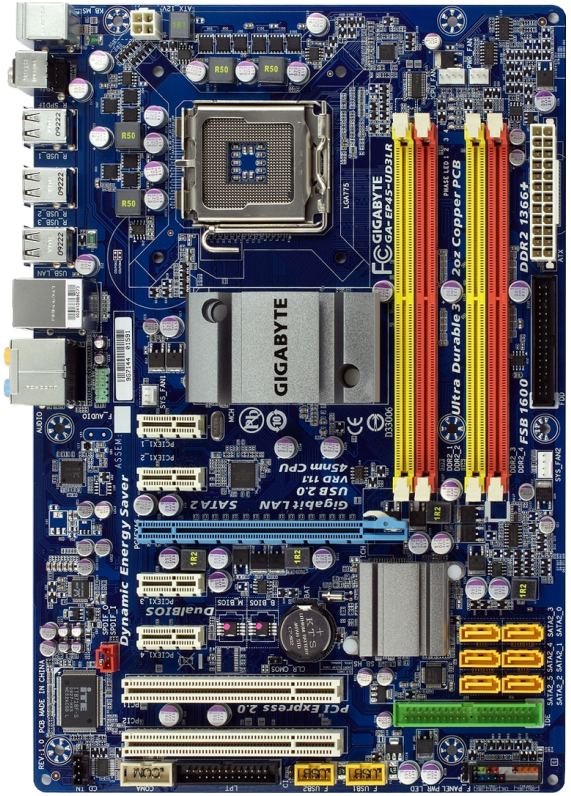
Other Factors

- Other hardware devices
  - video, sound
  - GPU
- Heat
  - limits clock speeds
- Battery
  - electron budgets remain important for portable devices



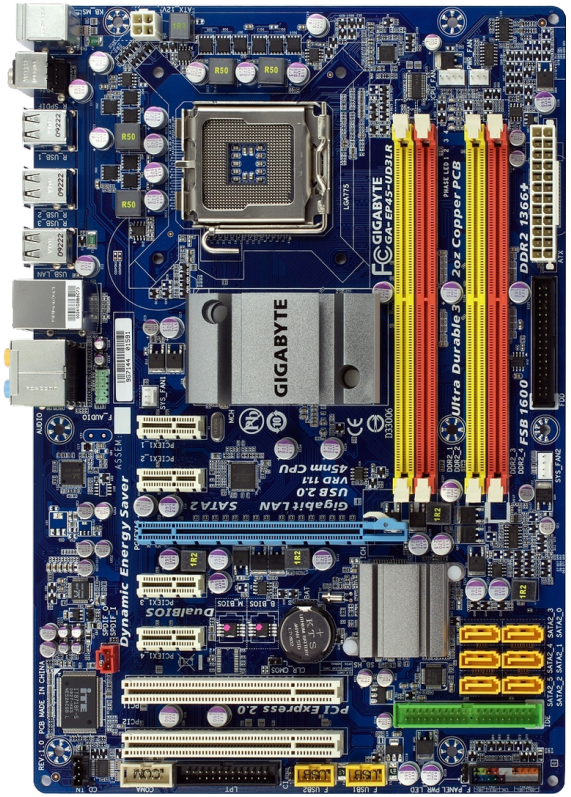
Abstraction Meets Reality

```
public class Software {
    public static void main( String[] args) {
        System.out.println("Software is abstract");
    }
}
```



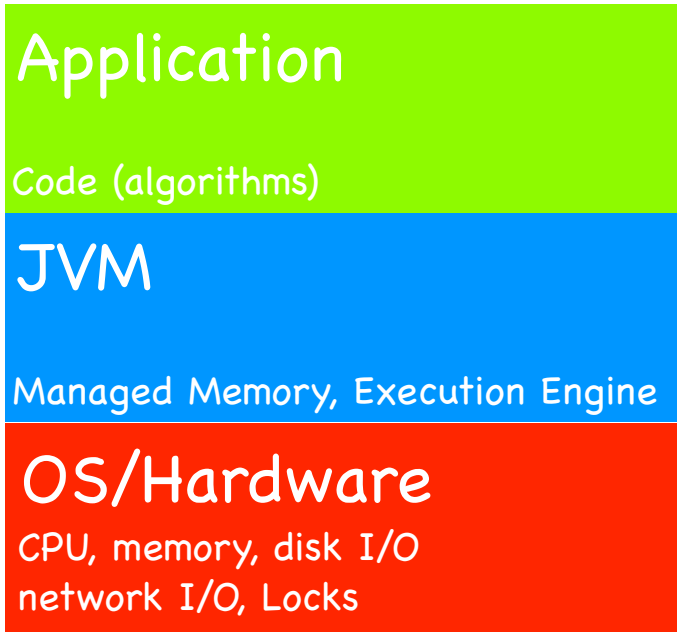
Software Layers

- Divide software into 3 layers
  - **Application**
    - convert inputs into outputs
    - orchestrate flows of data through the hardware
  - **Managed Runtime** (Java Virtual Machine)
    - services
    - runtime with optimizing compilers
    - managed memory
      - garbage collection
  - **Operating System**
    - services
    - metered access to hardware
      - creates the ***illusion*** of sharing
      - queue and schedulers
      - thread scheduler for the CPU



Model

- Create a model with 3 layers
  - Application
    - convert inputs into outputs
    - orchestrate flows of data through the hardware
  - Managed Runtime (Java Virtual Machine)
    - services
    - runtime with optimizing compilers
    - managed memory
      - garbage collection
  - Operating System
    - services
    - metered access to hardware
      - queue and schedulers
      - thread scheduler for the CPU



Question

Which is faster?

a) Bubble sort
b) Quick sort

Hint: In Big O notation...
- Bubble sort is N^2
- Quick sort of Nlog(N)

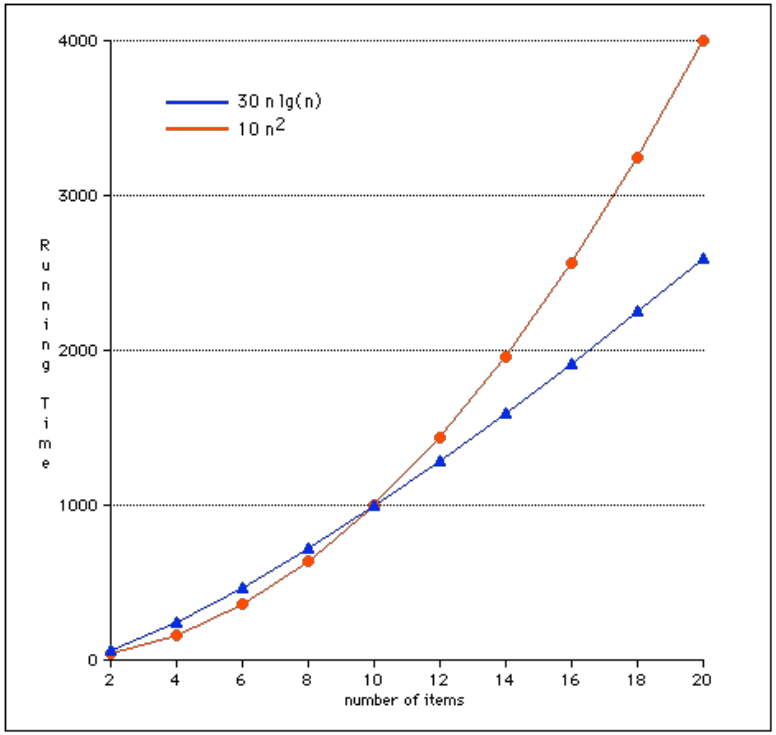
Question

Which is faster?

~~a) Bubble sort~~

~~b) Quick sort~~

c) It depends on the amount of data



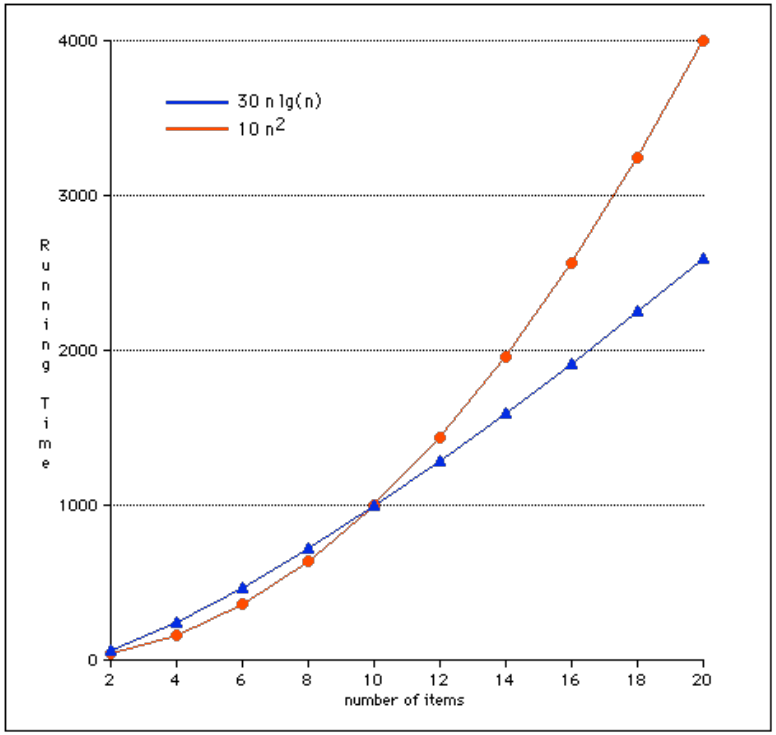
Question

Which is faster?

~~a) Bubble sort~~

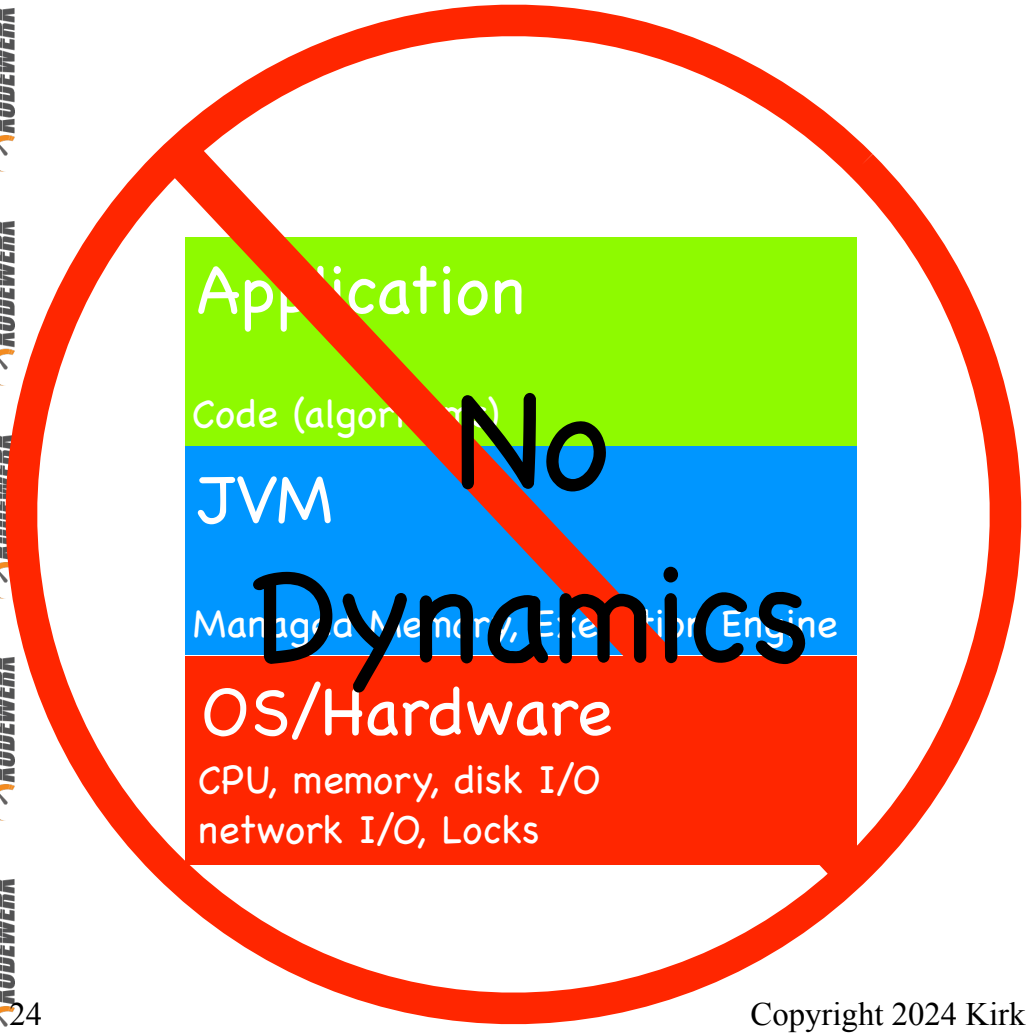
~~b) Quick sort~~

c) It depends on the amount of data

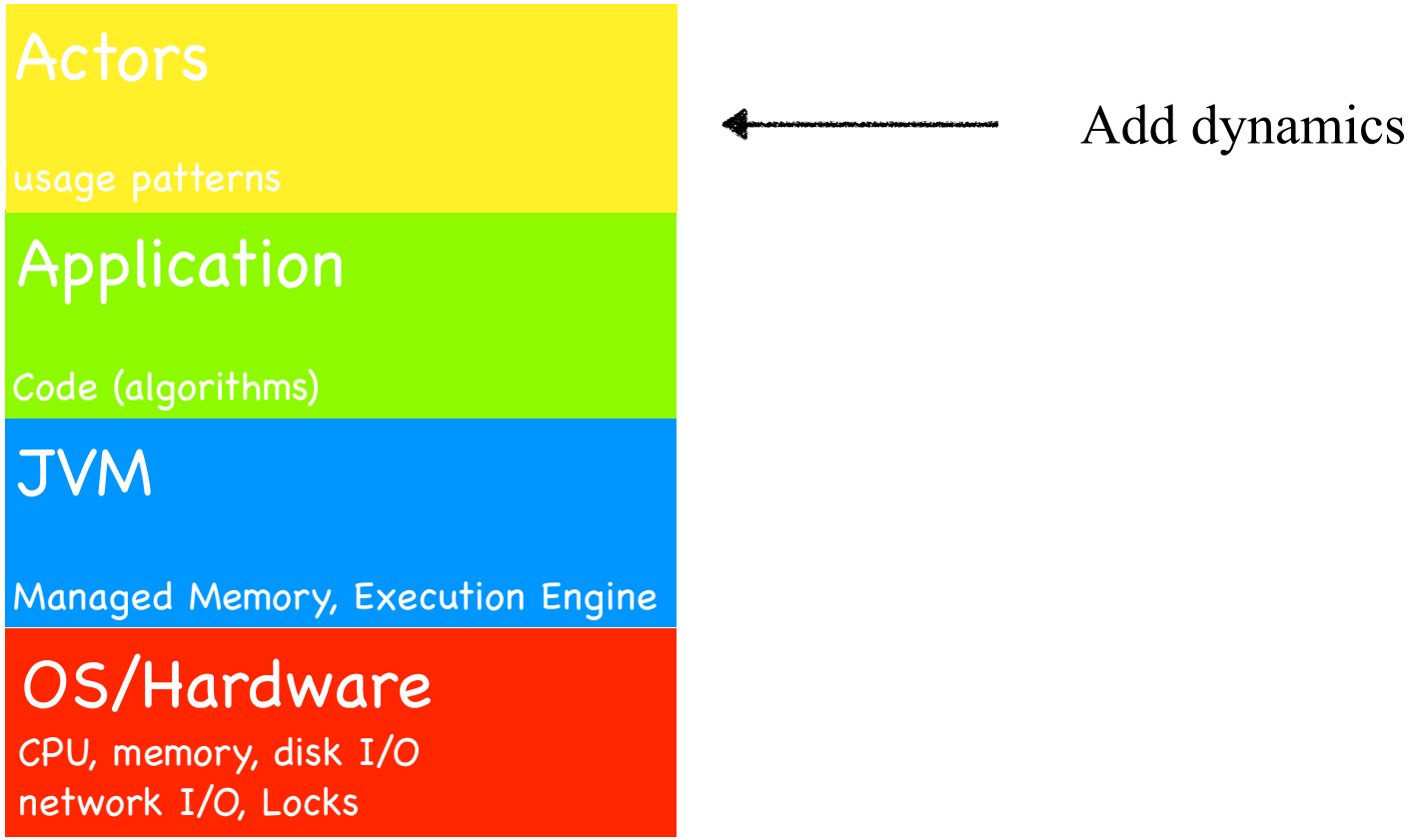


Where in the model is this information?

Model

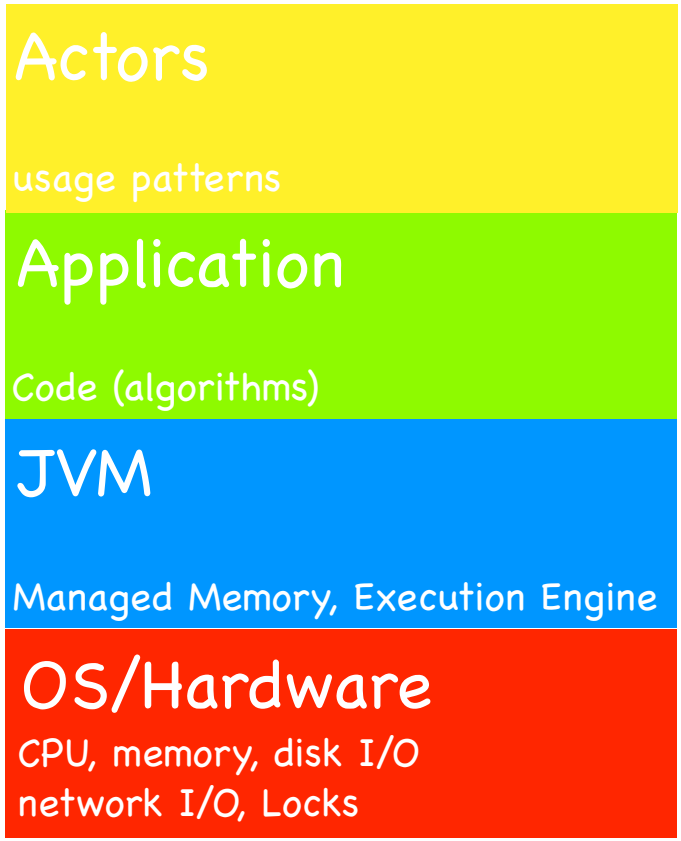


Model



Add dynamics

Actors



- Add dynamics by engaging with the system
  - other applications or nodes
  - end users
- A Usage Pattern is an aggregation of all activity over a window of time for a given machine
  - types of business transactions in the mix
  - the frequency of each business transactions
- Systems tend to enter a steady state driven by the Usage Pattern
  - we'll refer to this steady state as a **mode**
  - the mode will have a unique bottleneck

Theory of Constraints

From Wikipedia

The **theory of constraints** (**TOC**) is a management paradigm that views any manageable system as being limited in achieving more of its goals by a very small number of constraints. There is always at least one constraint, and TOC uses a focusing process to identify the constraint and restructure the rest of the organization around it. TOC adopts the common idiom "a chain is no stronger than its weakest link". That means that organizations and processes are vulnerable because the weakest person or part can always damage or break them, or at least adversely affect the outcome.

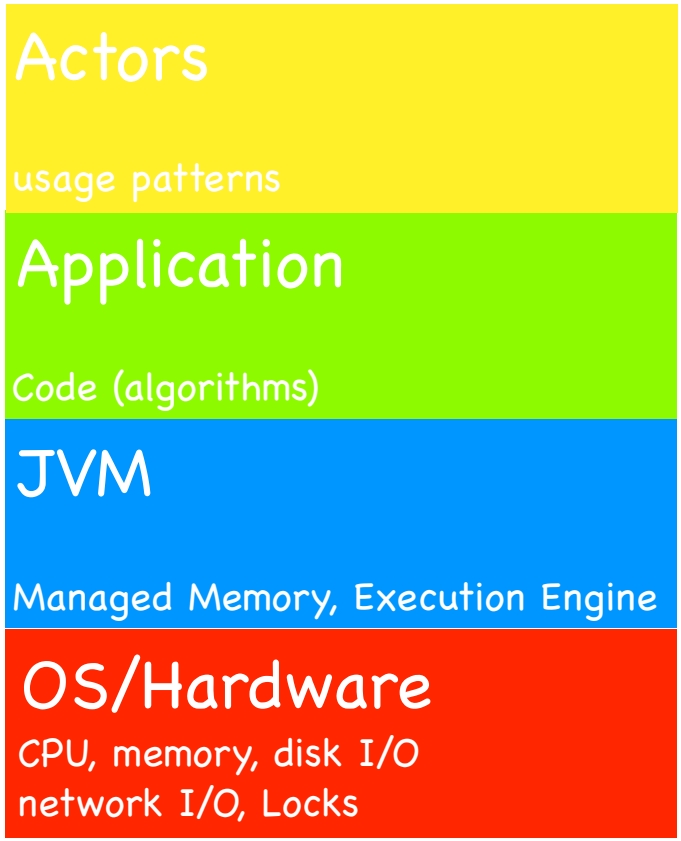
Eliyahu M. Goldratt, 1984.

Theory of Constraints

- Any system is limited by a very small number of constraints
  - Normally 1 resource
  - **Resource:** an asset that is drawn on in order for a system to function
  - **Scarce resource:** an asset whose available quantity is less than desired or cannot be replenished quickly enough

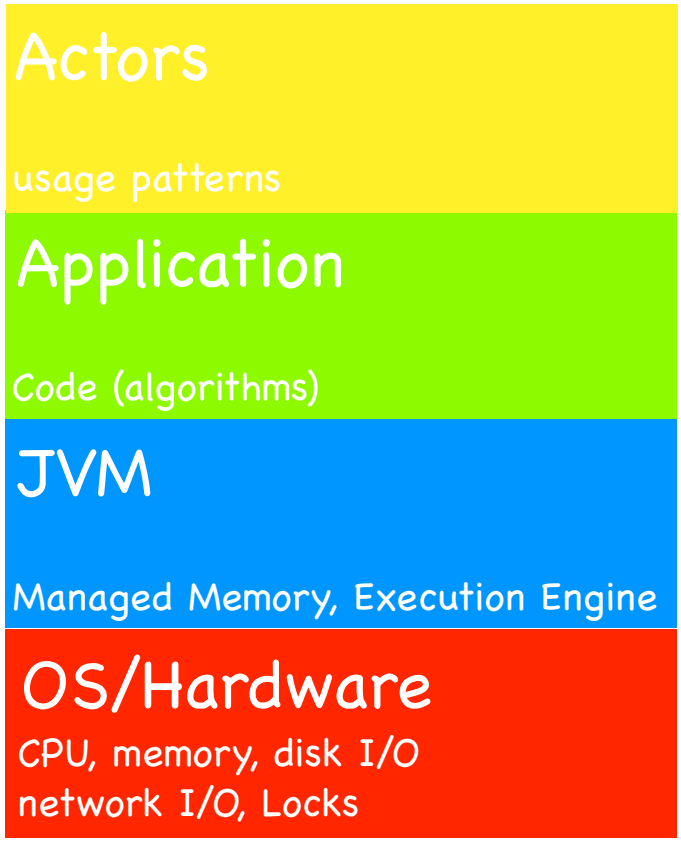
- All systems contain a critical path that is dependent on a **scarce resource**
  - Limits throughput
  - Call this the bottleneck

System Modality



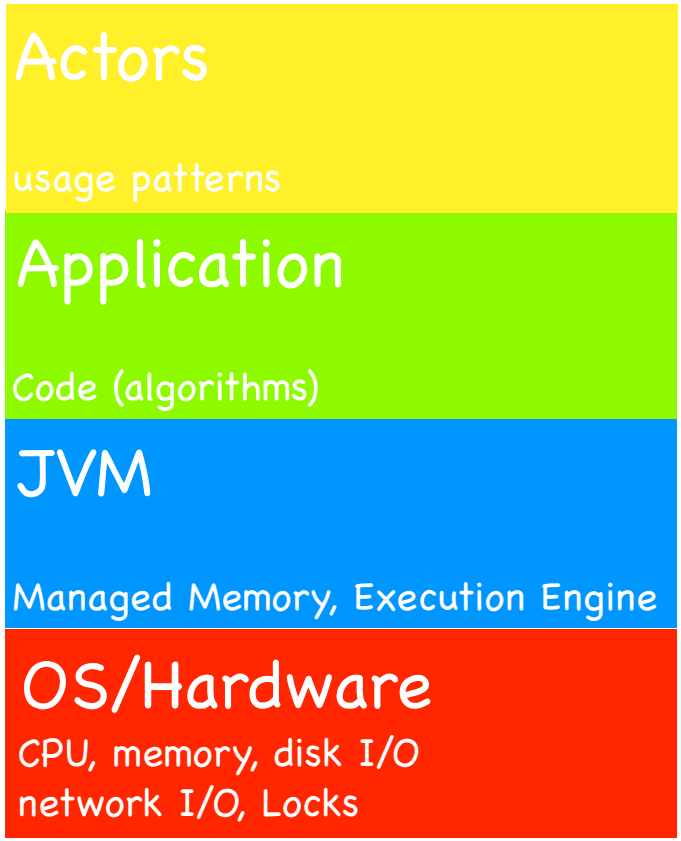
- Most system are multimodal
  - will transition between modes as Usage Patterns change
    - system performance will be dominated by a shift to the new bottleneck
- This effect is at the hardware level
  - in environments with many processes running, you will need to tease out contributions to the overall problem to focus on the main contributor
- Theory of constraints tells us that
  - within a chain of processes, one process will limit throughput
  - within a process, one activity will limit throughput
  - within the hardware, a single component will limit throughput
  - suggests a methodology mix of bottom-up and top-down is needed
    - jPDM makes use of both
      - latency investigation to identify the rate limiting component
      - bottom up to characterize the bottleneck

System Model in Action



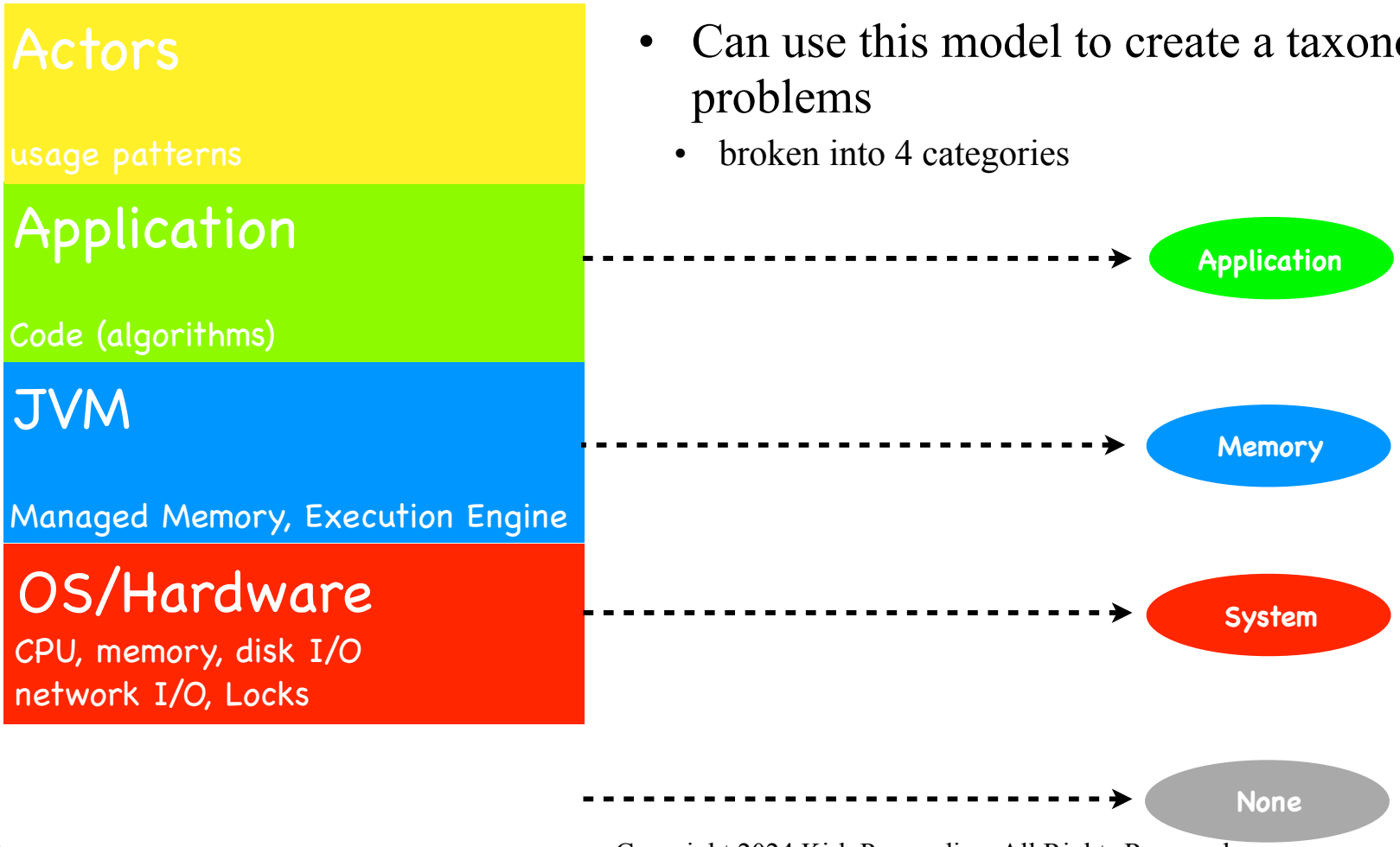
- Actors drive the application
- Application code orchestrates flows of data through JVM services
- JVM services orchestrate the flow of instructions to the OS/ Hardware
- OS works to manage hardware consumption
  - consumption is limited by capacity
  - need effective orchestration of data flows through the hardware
    - stalls in resourcing prevent capacity from being utilized

Classification of Performance



- Can use this model to create a taxonomy of performance problems
  - broken into 4 categories

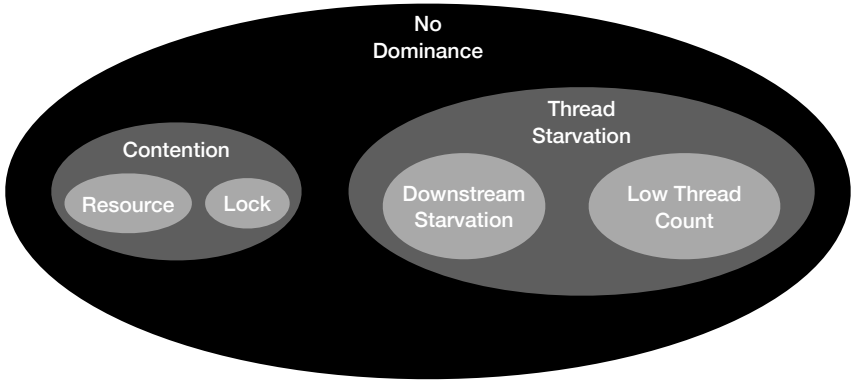
Classification of Performance



- Can use this model to create a taxonomy of performance problems
  - broken into 4 categories

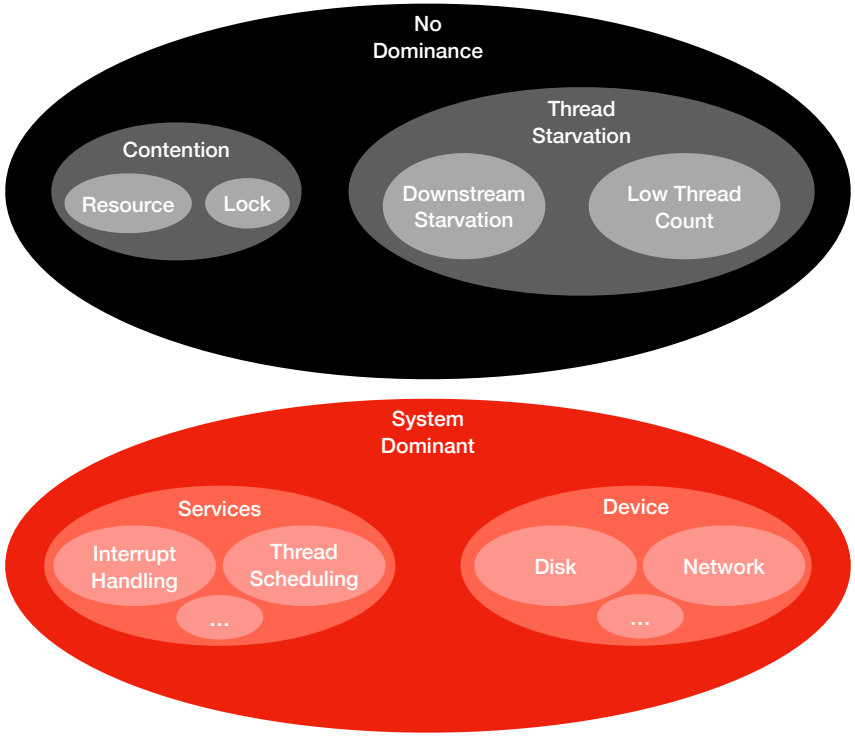
Nothing Dominant

- Characterized by low CPU utilization
  - thread counts maybe low for the available CPU
  - threads maybe blocked waiting for downstream services to complete
  - threads maybe blocked on a synchronized block of code
  - upstream may not be feeding this node as expected



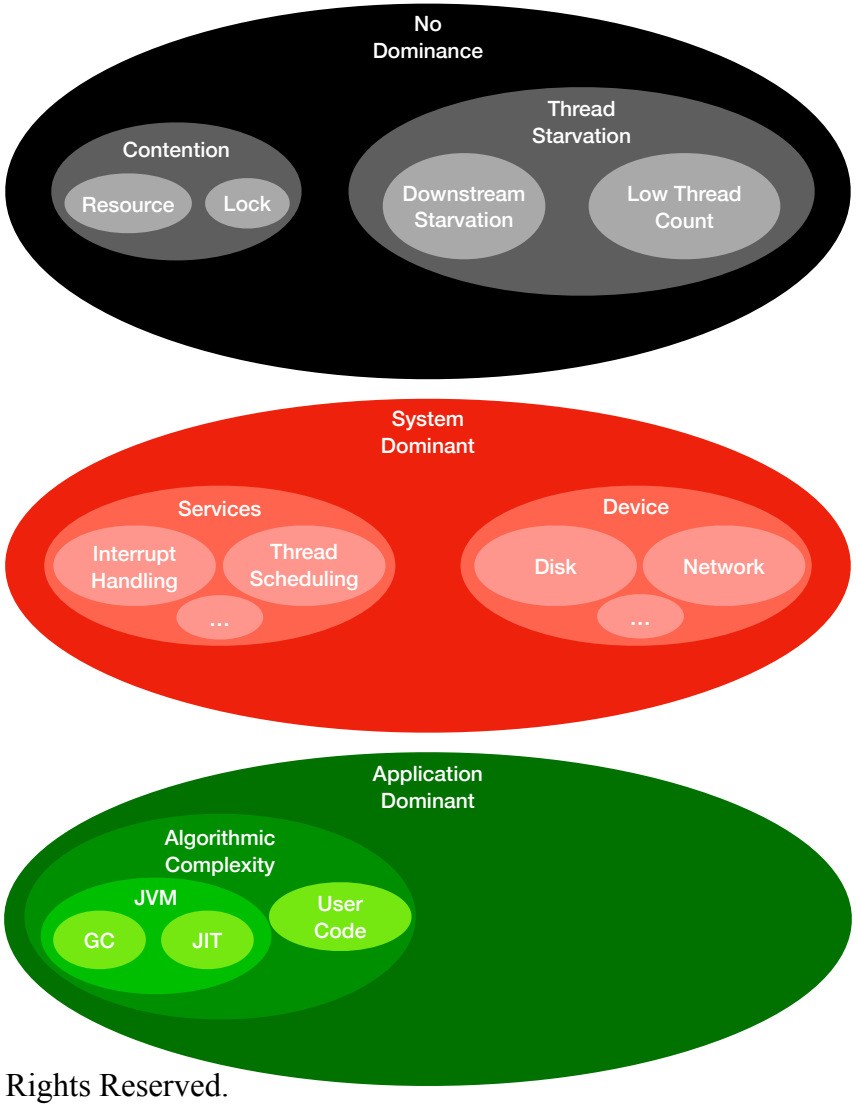
System Dominant

- Characterized by low CPU utilization
  - can be quite high in some cases
  - high % of kernel to user time
    - very little user activity drives a lot of pressure on the OS
  - Amdahl's law applies
    - all hardware is single threaded



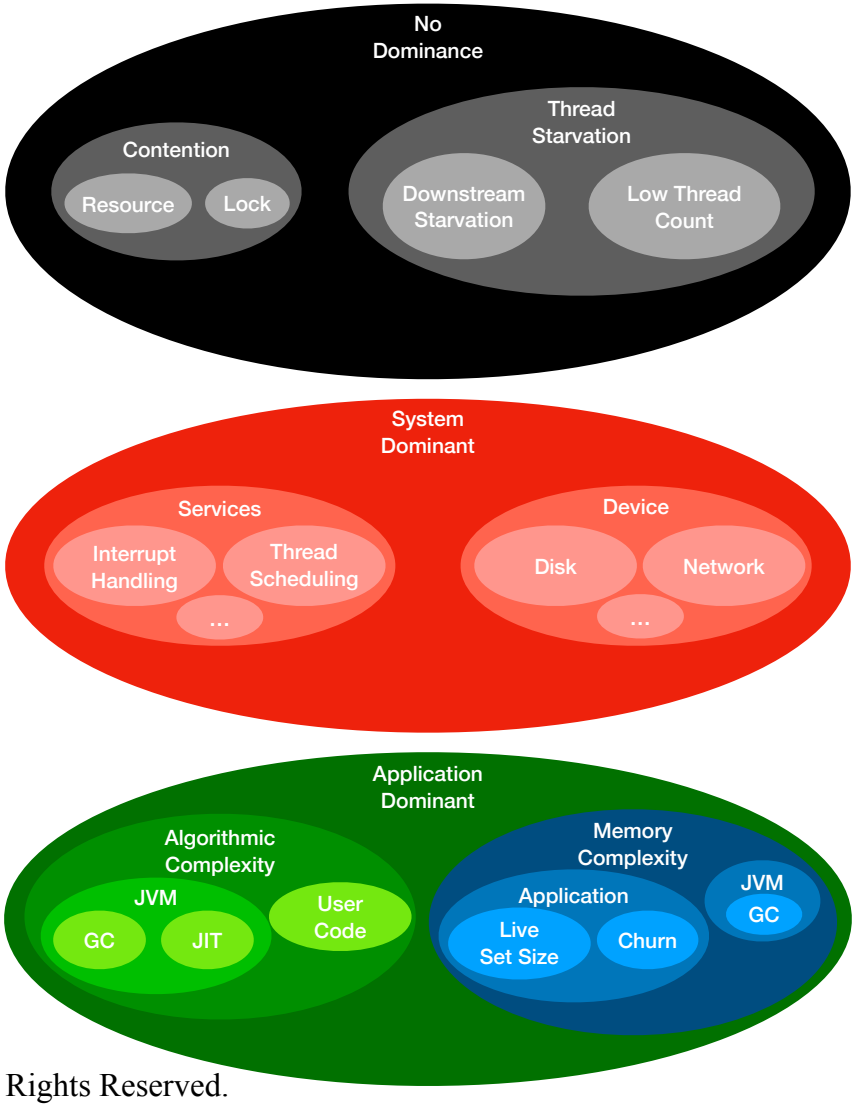
Application Dominant

- Characterized by high CPU utilization
  - rooted in algorithmic complexity
  - user code strength
  - JVM code strength

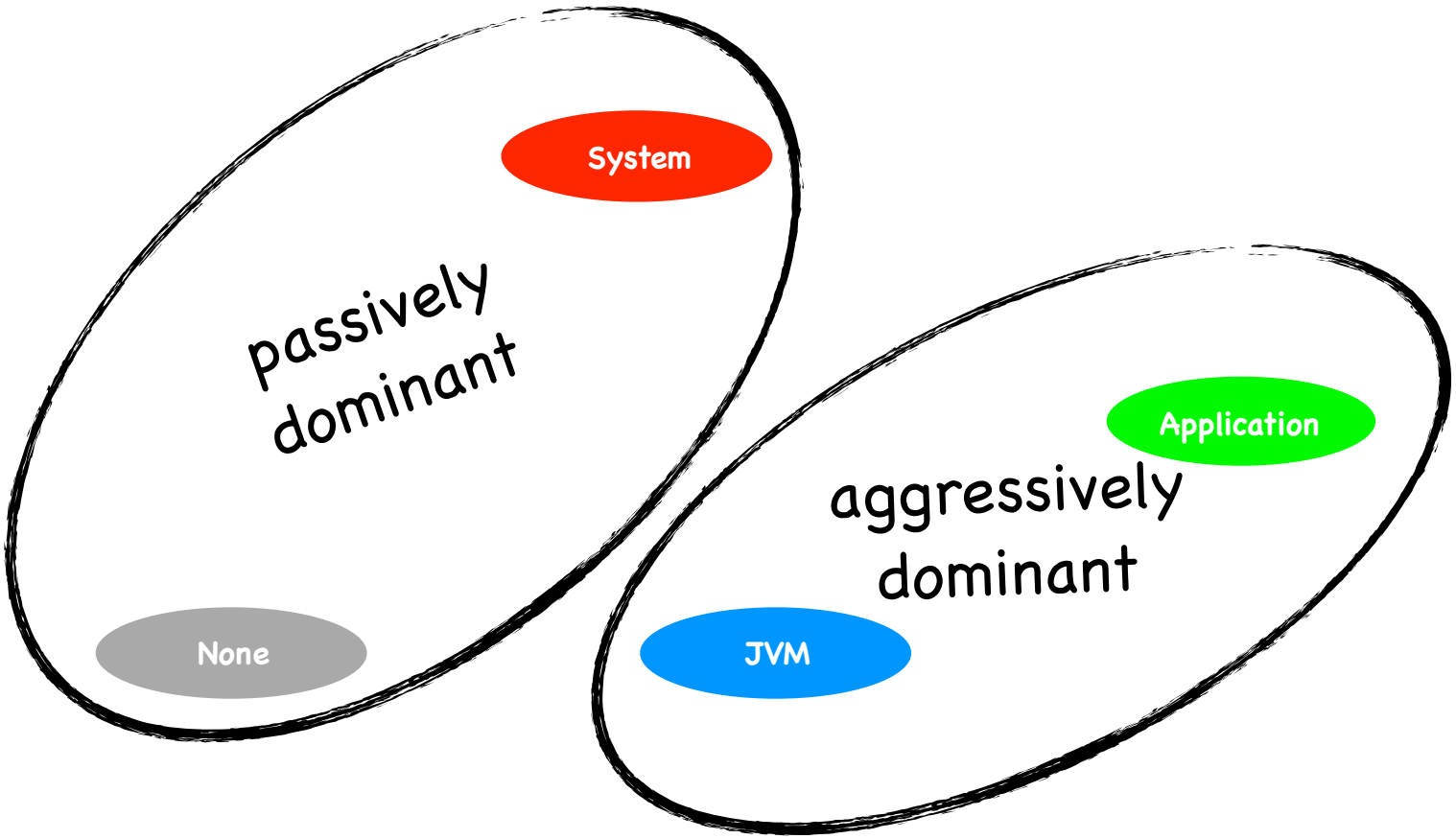


Memory Dominant

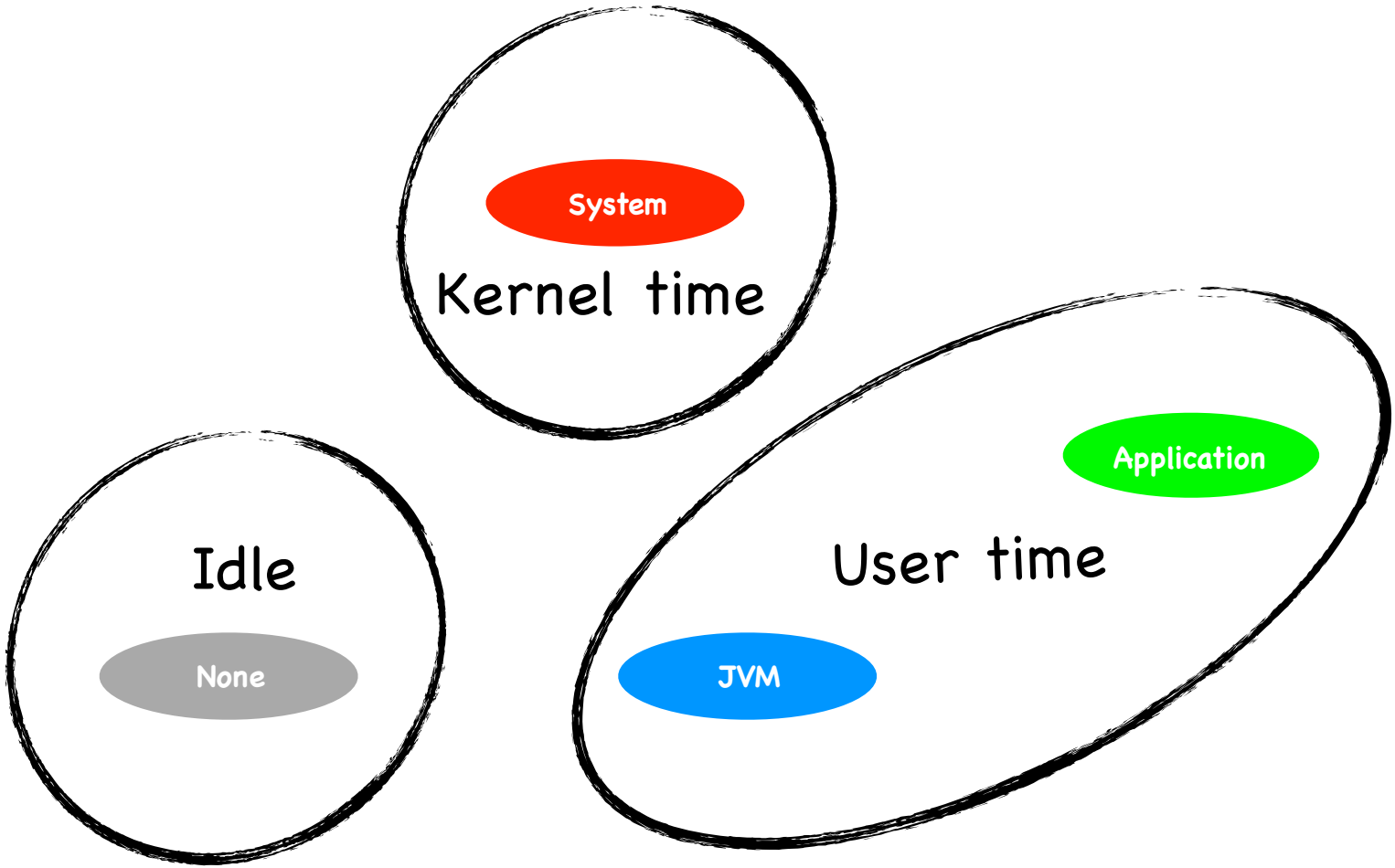
- Characterized by high CPU utilization
- Allocation pressure
  - frequent GC cycles
- Poorly configured Java heap
  - frequent collections of tenured
- High memory complexity
  - data structure/query mismatch
    - LinkedList for data lookup
    - HashMap for queuing



Expressions of CPU Consumption

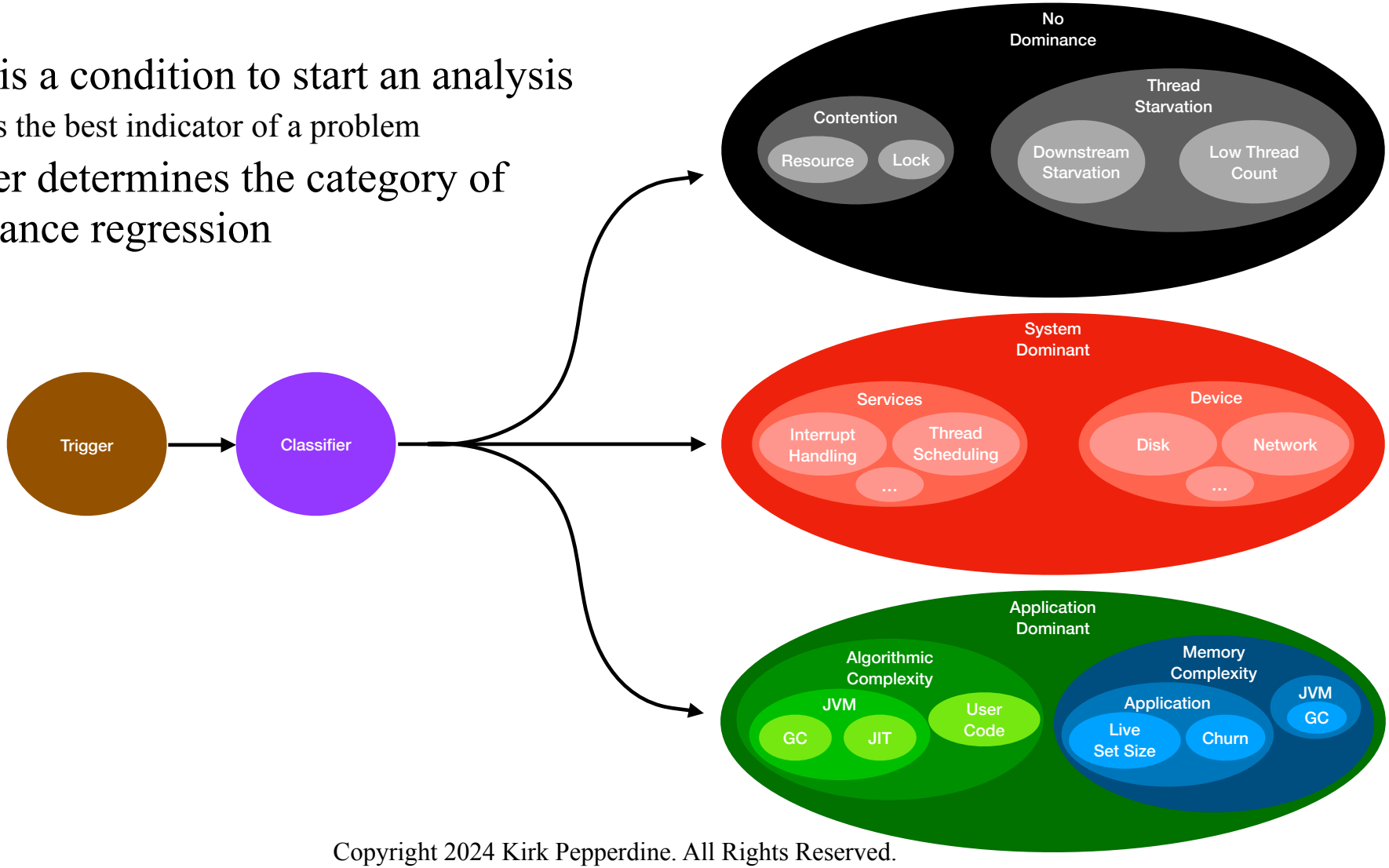


Expressions of CPU Consumption



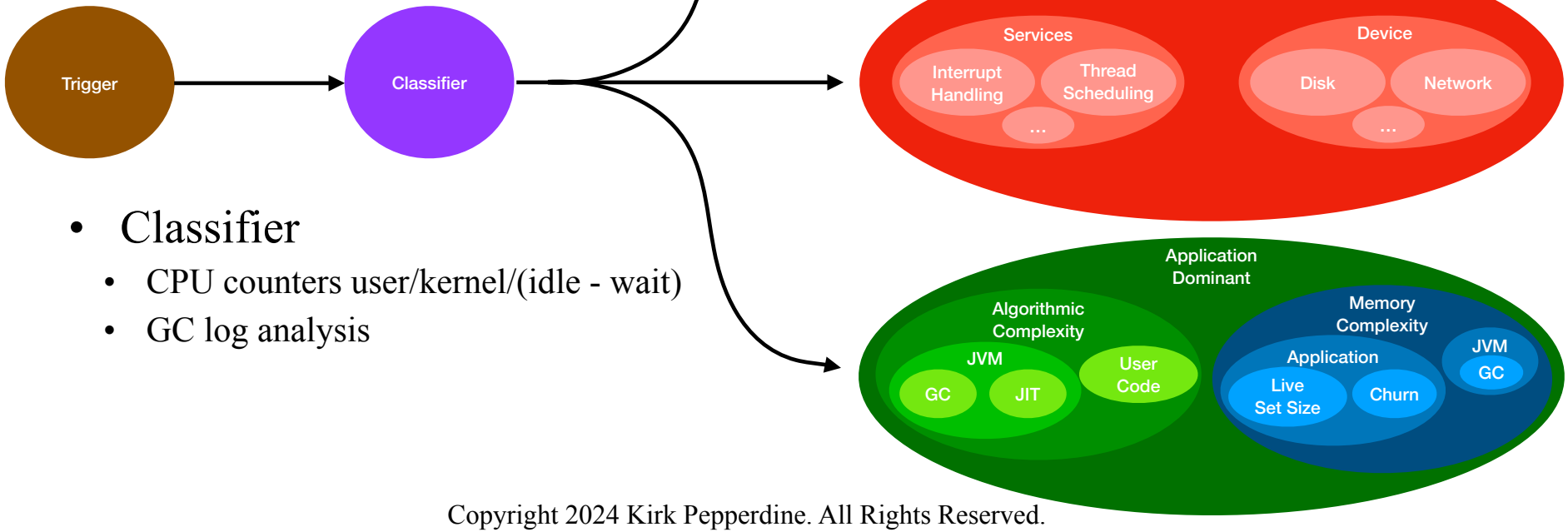
Trigger & Classifier

- Trigger is a condition to start an analysis
  - latency is the best indicator of a problem
- Classifier determines the category of performance regression



Trigger & Classifier

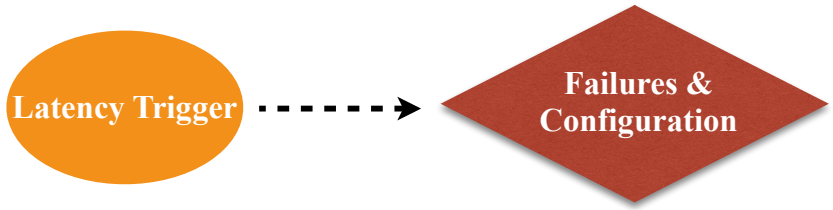
- SLA breach triggers a diagnostic cycle



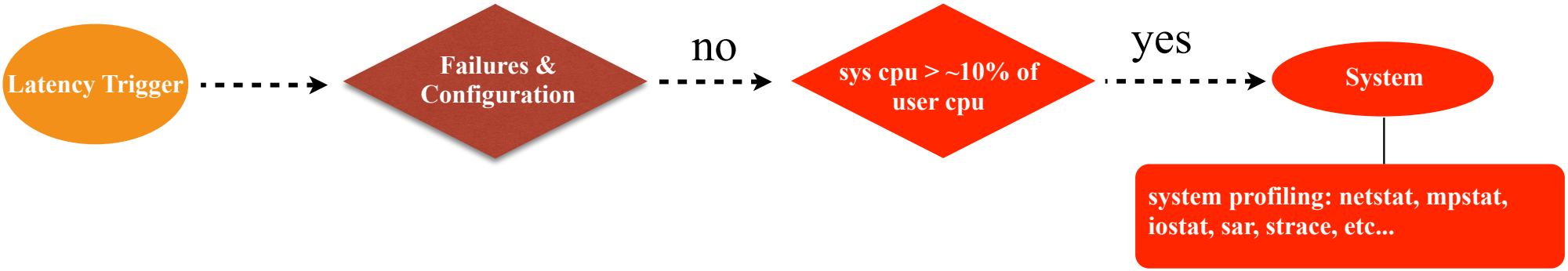
  - Classifier
    - CPU counters user/kernel/(idle - wait)
    - GC log analysis

jPDM Classifier

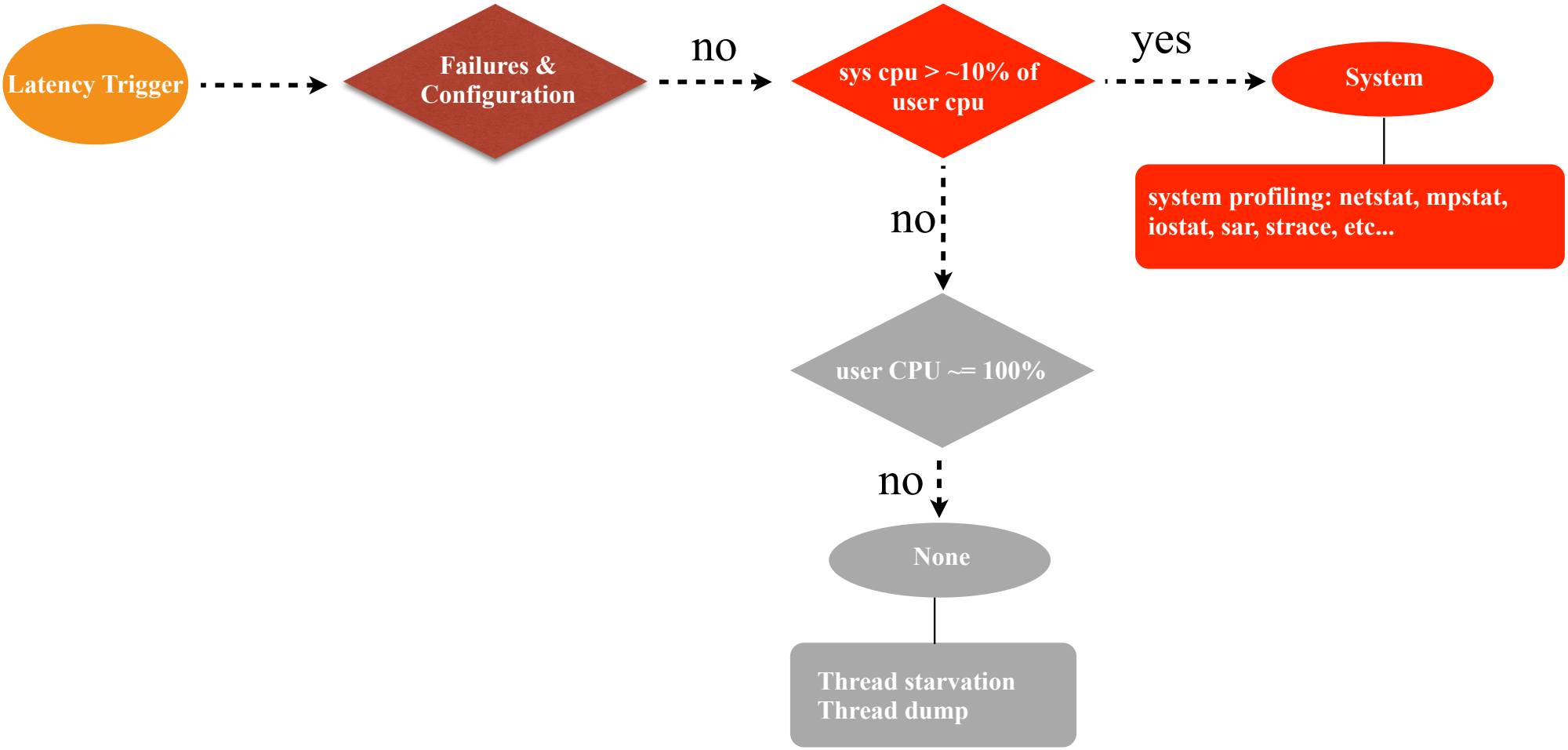
jPDM Classifier



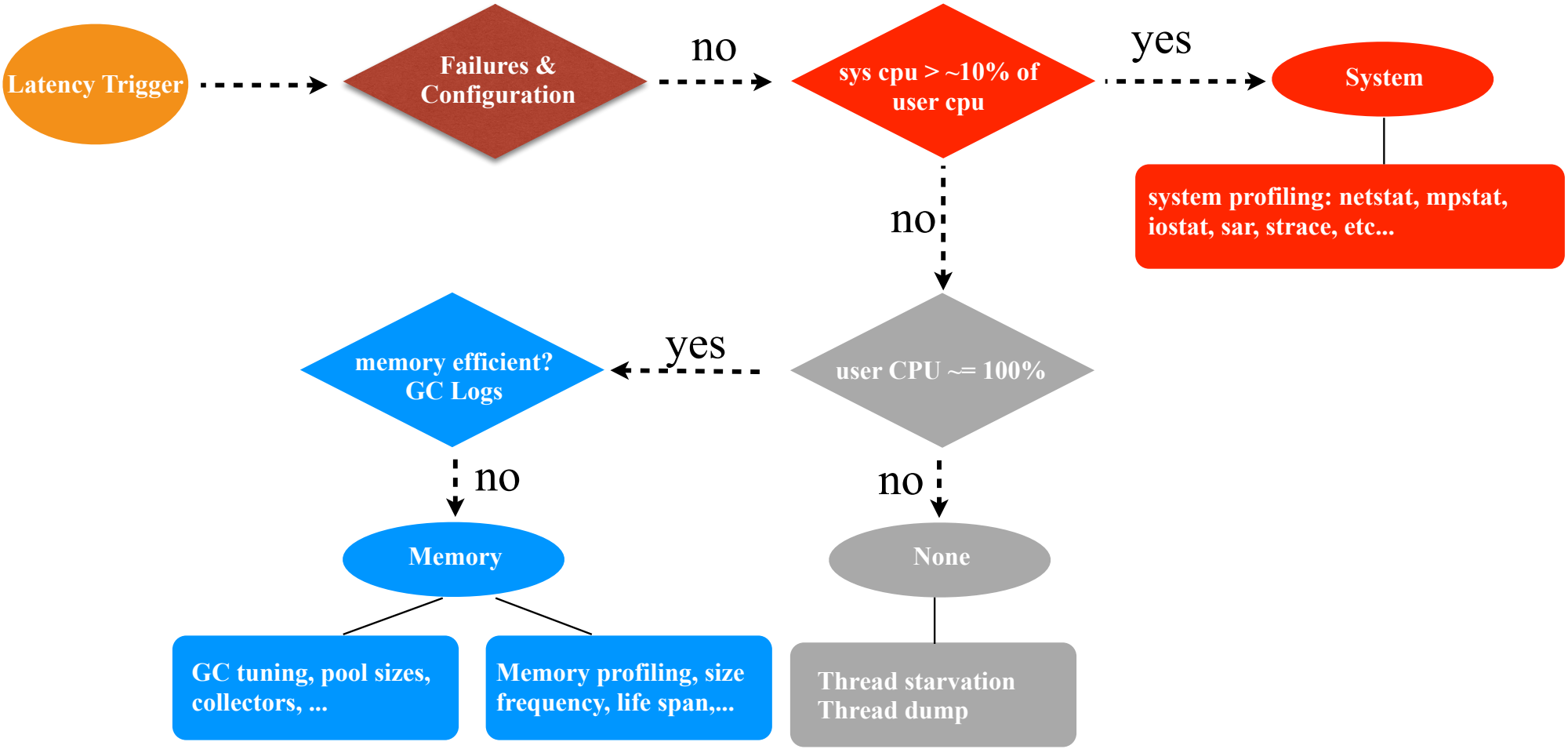
jPDM Classifier



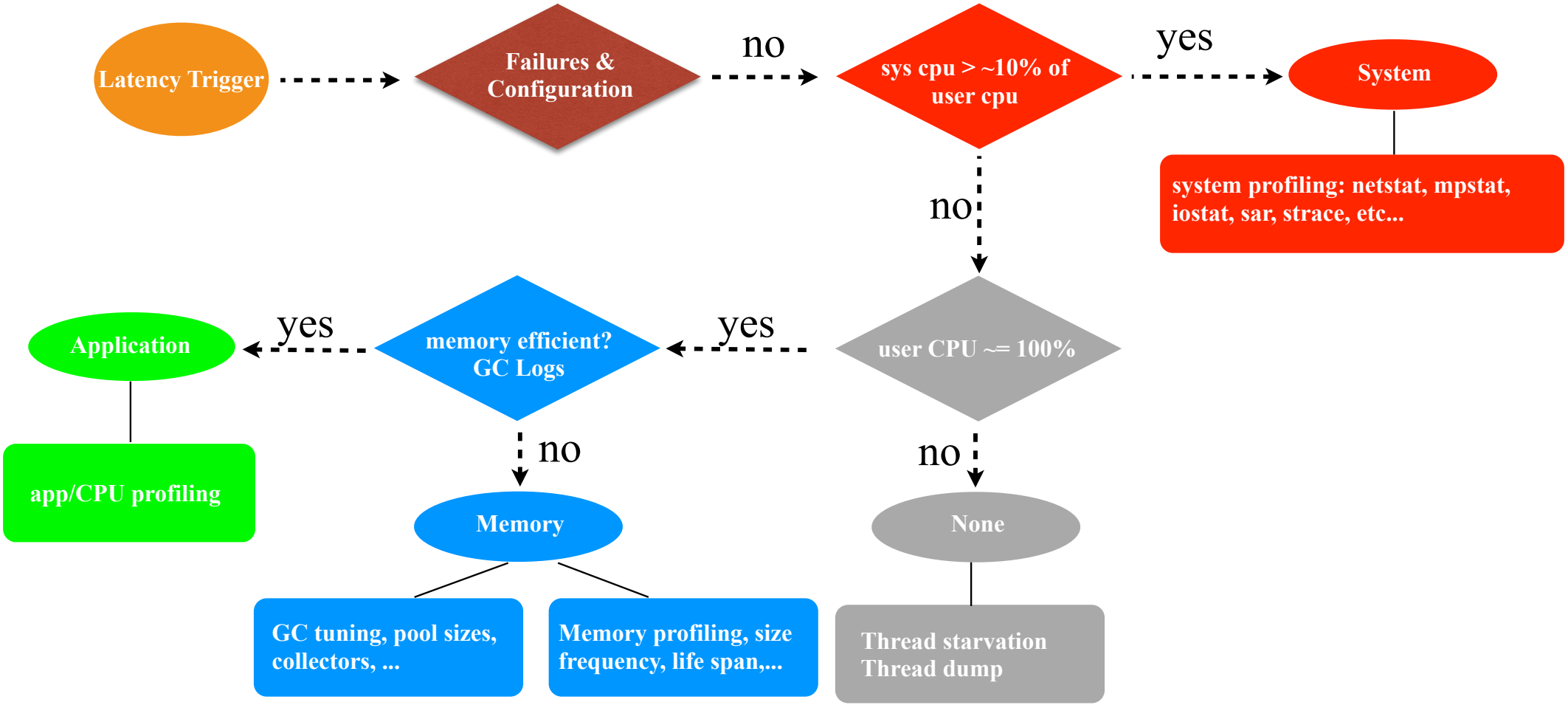
jPDM Classifier



jPDM Classifier



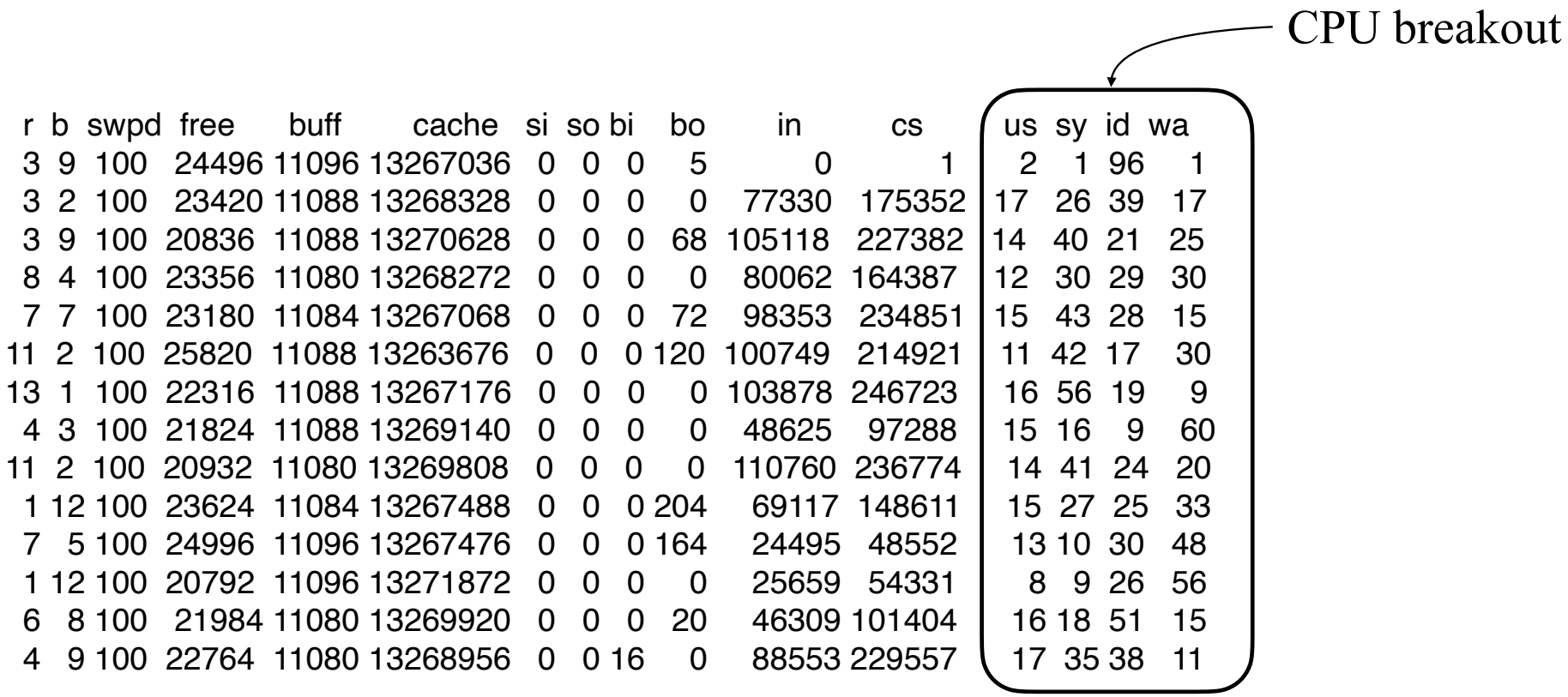
jPDM Classifier



vmstat Analysis

r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa
3	9	100	24496	11096	13267036	0	0	0	5	0	1	2	1	96	1
3	2	100	23420	11088	13268328	0	0	0	0	77330	175352	17	26	39	17
3	9	100	20836	11088	13270628	0	0	0	68	105118	227382	14	40	21	25
8	4	100	23356	11080	13268272	0	0	0	0	80062	164387	12	30	29	30
7	7	100	23180	11084	13267068	0	0	0	72	98353	234851	15	43	28	15
11	2	100	25820	11088	13263676	0	0	0	120	100749	214921	11	42	17	30
13	1	100	22316	11088	13267176	0	0	0	0	103878	246723	16	56	19	9
4	3	100	21824	11088	13269140	0	0	0	0	48625	97288	15	16	9	60
11	2	100	20932	11080	13269808	0	0	0	0	110760	236774	14	41	24	20
1	12	100	23624	11084	13267488	0	0	0	204	69117	148611	15	27	25	33
7	5	100	24996	11096	13267476	0	0	0	164	24495	48552	13	10	30	48
1	12	100	20792	11096	13271872	0	0	0	0	25659	54331	8	9	26	56
6	8	100	21984	11080	13269920	0	0	0	20	46309	101404	16	18	51	15
4	9	100	22764	11080	13268956	0	0	16	0	88553	229557	17	35	38	11

vmstat Analysis



r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa
3	9	100	24496	11096	13267036	0	0	0	5	0	1	2	1	96	1
3	2	100	23420	11088	13268328	0	0	0	0	77330	175352	17	26	39	17
3	9	100	20836	11088	13270628	0	0	0	68	105118	227382	14	40	21	25
8	4	100	23356	11080	13268272	0	0	0	0	80062	164387	12	30	29	30
7	7	100	23180	11084	13267068	0	0	0	72	98353	234851	15	43	28	15
11	2	100	25820	11088	13263676	0	0	0	120	100749	214921	11	42	17	30
13	1	100	22316	11088	13267176	0	0	0	0	103878	246723	16	56	19	9
4	3	100	21824	11088	13269140	0	0	0	0	48625	97288	15	16	9	60
11	2	100	20932	11080	13269808	0	0	0	0	110760	236774	14	41	24	20
1	12	100	23624	11084	13267488	0	0	0	204	69117	148611	15	27	25	33
7	5	100	24996	11096	13267476	0	0	0	164	24495	48552	13	10	30	48
1	12	100	20792	11096	13271872	0	0	0	0	25659	54331	8	9	26	56
6	8	100	21984	11080	13269920	0	0	0	20	46309	101404	16	18	51	15
4	9	100	22764	11080	13268956	0	0	16	0	88553	229557	17	35	38	11

vmstat Analysis

us	sy	
~~2~~	~~1~~	toss
17	26	
14	40	
12	30	
15	43	
11	42	
16	56	
15	16	
14	41	
15	27	
13	10	
8	9	
16	18	
17	35	

vmstat Analysis

us	sy	ratio	> 10%
~~2~~	~~1~~	toss	
17	26	26 > 1.7	T
14	40	40 > 1.4	T
12	30	30 > 1.2	T
15	43	43 > 1.5	T
11	42	42 > 1.1	T
16	56	56 > 1.6	T
15	16	16 > 1.5	T
14	41	41 > 1.4	T
15	27	27 > 1.5	T
13	10	10 > 1.3	T
8	9	9 > 0.8	T
16	18	18 > 1.6	T
17	35	35 > 1.7	T

vmstat Analysis

Conclusion -> System dominant

us	sy	ratio	> 10%
2	1	toss	
17	26	26 > 1.7	T
14	40	40 > 1.4	T
12	30	30 > 1.2	T
15	43	43 > 1.5	T
11	42	42 > 1.1	T
16	56	56 > 1.6	T
15	16	16 > 1.5	T
14	41	41 > 1.4	T
15	27	27 > 1.5	T
13	10	10 > 1.3	T
8	9	9 > 0.8	T
16	18	18 > 1.6	T
17	35	35 > 1.7	T

vmstat Analysis

r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa
~~3~~	~~9~~	~~100~~	~~24496~~	~~11096~~	~~13267036~~	~~0~~	~~0~~	~~0~~	~~5~~	~~0~~	~~1~~	~~2~~	~~1~~	~~96~~	~~1~~
3	2	100	23420	11088	13268328	0	0	0	0	77330	175352	17	26	39	17
3	9	100	20836	11088	13270628	0	0	0	68	105118	227382	14	40	21	25
8	4	100	23356	11080	13268272	0	0	0	0	80062	164387	12	30	29	30
7	7	100	23180	11084	13267068	0	0	0	72	98353	234851	15	43	28	15
11	2	100	25820	11088	13263676	0	0	0	120	100749	214921	11	42	17	30
13	1	100	22316	11088	13267176	0	0	0	0	103878	246723	16	56	19	9
4	3	100	21824	11088	13269140	0	0	0	0	48625	97288	15	16	9	60
11	2	100	20932	11080	13269808	0	0	0	0	110760	236774	14	41	24	20
1	12	100	23624	11084	13267488	0	0	0	204	69117	148611	15	27	25	33
7	5	100	24996	11096	13267476	0	0	0	164	24495	48552	13	10	30	48
1	12	100	20792	11096	13271872	0	0	0	0	25659	54331	8	9	26	56
6	8	100	21984	11080	13269920	0	0	0	20	46309	101404	16	18	51	15
4	9	100	22764	11080	13268956	0	0	16	0	88553	229557	17	35	38	11

vmstat Analysis

r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa
3	9	100	24496	11096	13267036	0	0	0	5	0	1	2	1	96	1
3	2	100	23420	11088	13268328	0	0	0	0	77330	175352	17	26	39	17
3	9	100	20836	11088	13270628	0	0	0	68	105118	227382	14	40	21	25
8	4	100	23356	11080	13268272	0	0	0	0	80062	164387	12	30	29	30
7	7	100	23180	11084	13267068	0	0	0	72	98353	234851	15	43	28	15
11	2	100	25820	11088	13263676	0	0	0	120	100749	214921	11	42	17	30
13	1	100	22316	11088	13267176	0	0	0	0	103878	246723	16	56	19	9
4	3	100	21824	11088	13269140	0	0	0	0	48625	97288	15	16	9	60
11	2	100	20932	11080	13269808	0	0	0	0	110760	236774	14	41	24	20
1	12	100	23624	11084	13267488	0	0	0	204	69117	148611	15	27	25	33
7	5	100	24996	11096	13267476	0	0	0	164	24495	48552	13	10	30	48
1	12	100	20792	11096	13271872	0	0	0	0	25659	54331	8	9	26	56
6	8	100	21984	11080	13269920	0	0	0	20	46309	101404	16	18	51	15
4	9	100	22764	11080	13268956	0	0	16	0	88553	229557	17	35	38	11

Conclusion

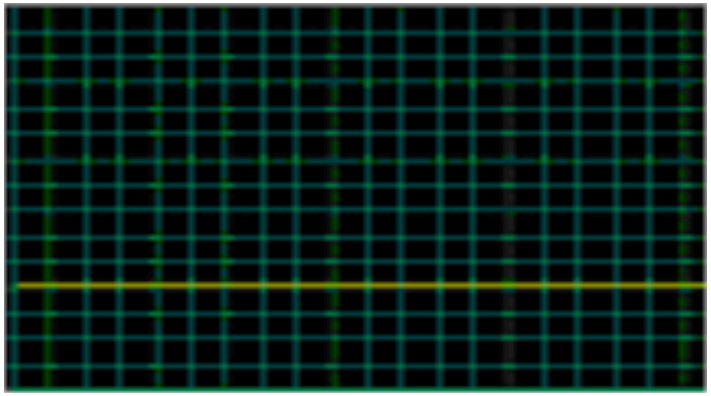
- jPDM makes use of both top-down and bottom-up techniques
  - top-down latency analysis to identify the hotspot
  - bottom-up hardware analysis to classify cause
  - GC logs to differentiate between memory and algorithmic complexity

- Measures need to support
  - course grain latency measurements
  - CPU breakout of user and kernel
    - idle and wait time can be quite helpful
  - GC logs

Exercise 1 Revisited

CPU Usage History

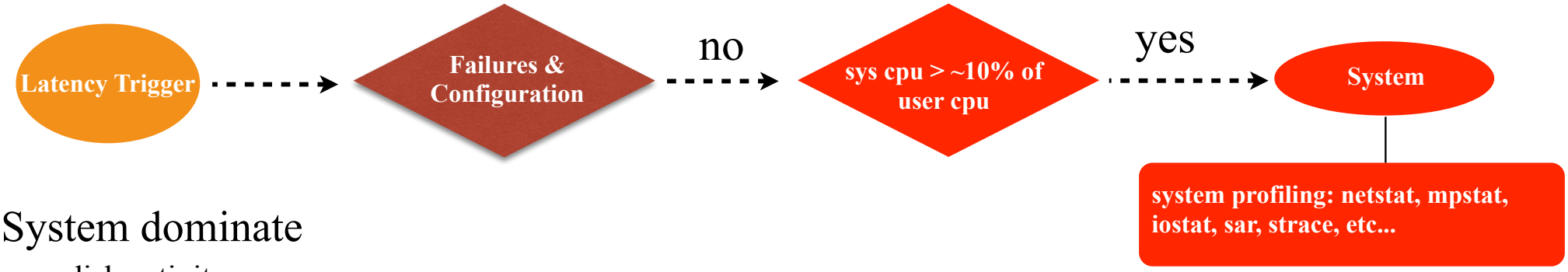
Page File Usage History



- Web application running on Windows server
  - Users are complaining about slow response times for simple tasks
- HTTP/Servlet
  - supports up to 20 users
  - makes use of an Oracle DB running on a separate server
    - Oracle JDBC driver/Hibernate
    - dedicated network
    - DBA reports sum millisecond response times
    - Network admin report minimal network saturation and latencies
  - no disk activity
  - CPU and page file usage charts were captured

- Task: Reason through the data to determine the most likely source of the regression

jPDM Classifier



- System dominate
  - no disk activity
  - network performance is acceptable
    - server view
    - client view is likely an issue
- Diagnosis - many quick calls to the DB driving up context switching rates
  - context switching dominates kernel times
    - combo of excessive calls to the DB with thread scheduling is driving application latency (ripple loading)
- Solution - perform join in the DB

Performance Tuning With Cheap Drink and Poor Tools

Ex 1.1 Identify Dominating Consumer

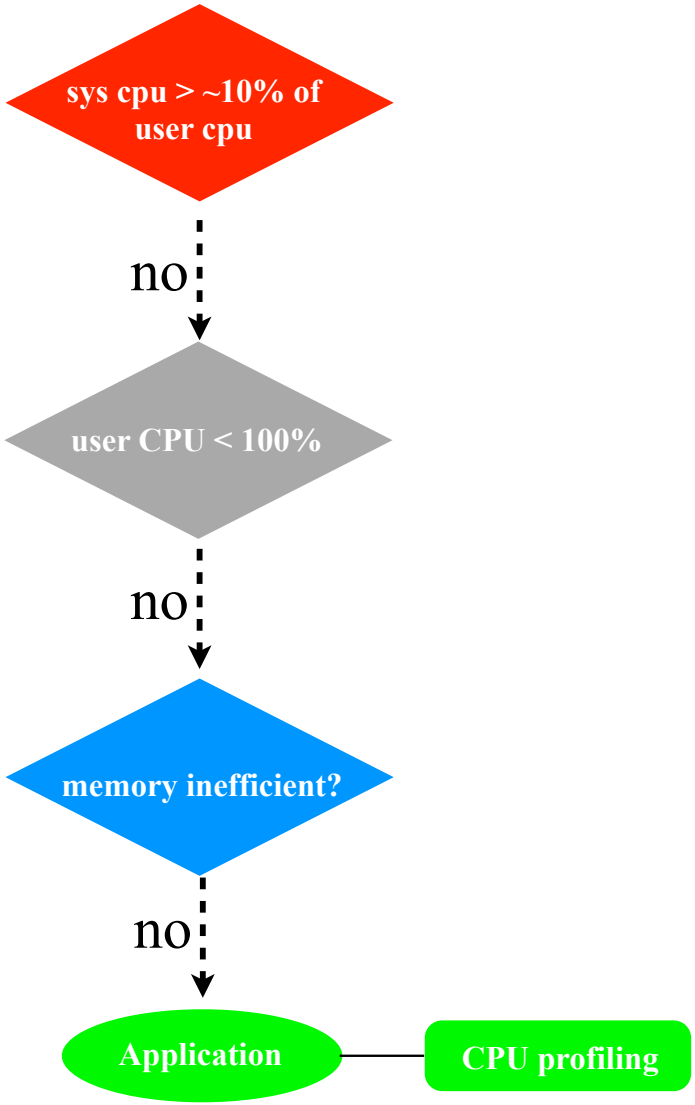
1. Find the setEnv script in the root directory of the exercises
    1. open it and set the JAVA_HOME env variable to your distribution of the JDK
        1. for the windows batch script do remember to escape spaces in the path name

2. cd into the exercise1 directory
    1. run the compile script to compile the application
    2. run the generate script to generate all of the test data
    3. run the run script to run the application

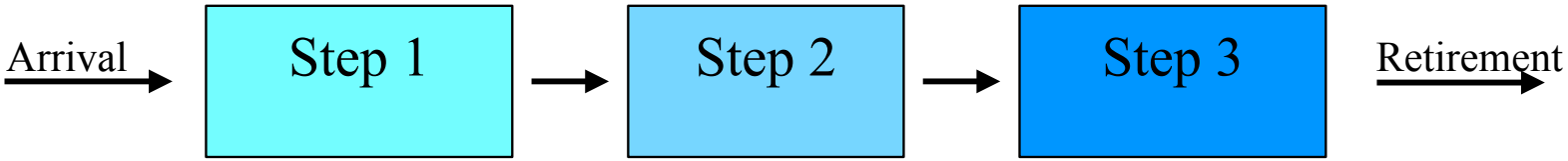
3. Monitor the application to identify the dominating consumer
    1. tooling
        1. Linux: vmstat
        2. Windows: TaskManager
            1. configure to show kernel counters
        3. OSX: Activity Monitor
    2. Note: This is a single threaded benchmark. Perform the analysis on the assumption that only a single core can be used.

Bottleneck #1

- Test is single threaded
  - technically: **Nothing Dominate**
  - adjust the model to account for single threaded
- Kernel time runs >10% of User
  - GC is not an issue
- Conclusion
  - **System Dominate**
- Next Step
  - Thread monitor/OS profiling

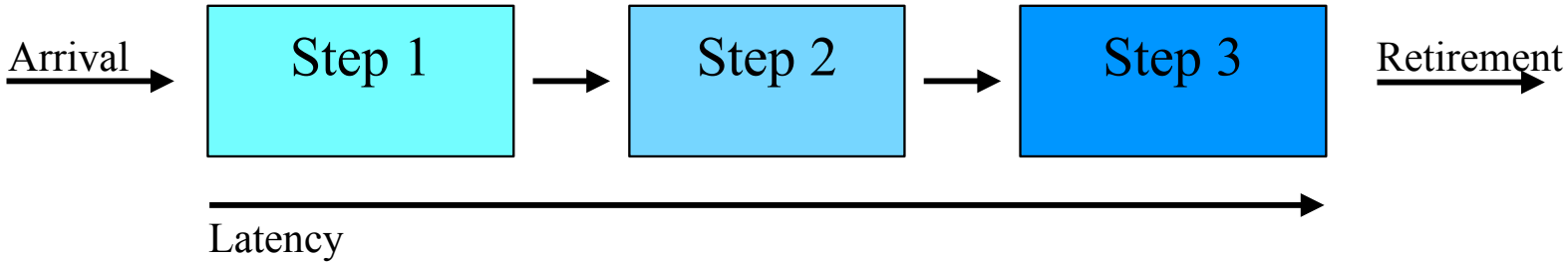


Latency Investigation



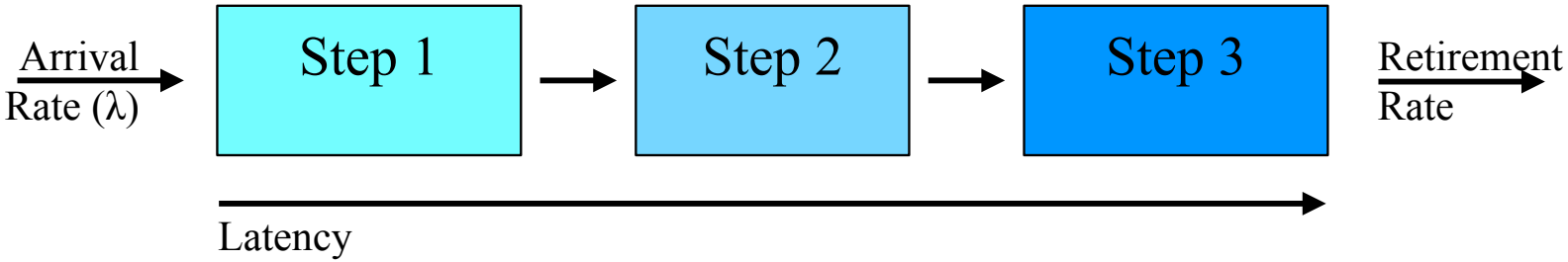
- Use Latency to identify **where** the bottleneck is
- Use of model to work through some concept
  - three step pipeline to convert input to output

Process Model



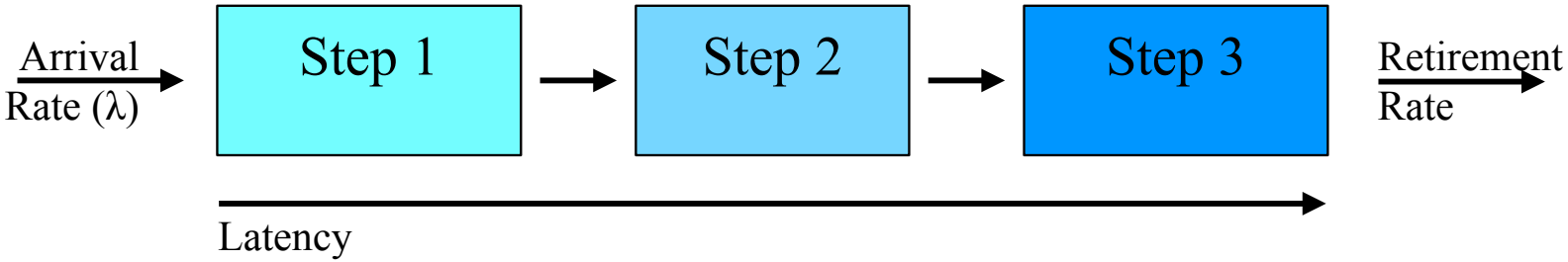
- Three steps to convert input to output
- **Latency** is the time to process a single request
  - time of retirement - time of arrival

Process Model



- Three steps to convert inputs into output
- **Latency** is the time to process a single request
  - time of retirement - time of arrival
- **Throughput** is a measure of how many transactions are retired per unit of time
  - (number of retirements @ t_2 - number of retirements @ t_1) / (t_2 - t_1)
- **Load** is the number of requests in flight
- **Capacity** is the maximum number of requests the system can support
- Little's law can be used to help us understand the performance of this model
  - also tie into ToC

Little's Law



- Little's law can be used to help us understand the performance of this mode

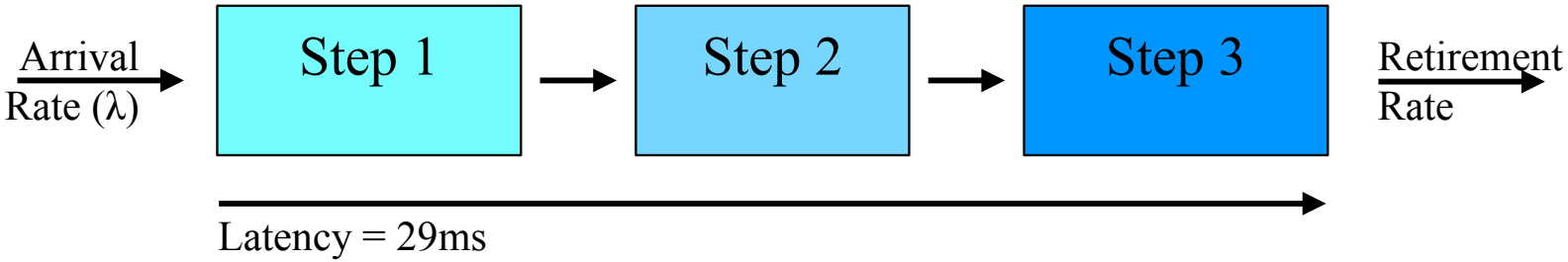
$$L = \lambda W$$

- L - Number of transactions in the system
- λ - Arrival rate
- W - Service time (latency)

$$\lambda = 1 / \mu$$

- λ - Retirement rate
- μ - Service time (latency)

Using Little's Law



$L = \lambda W$

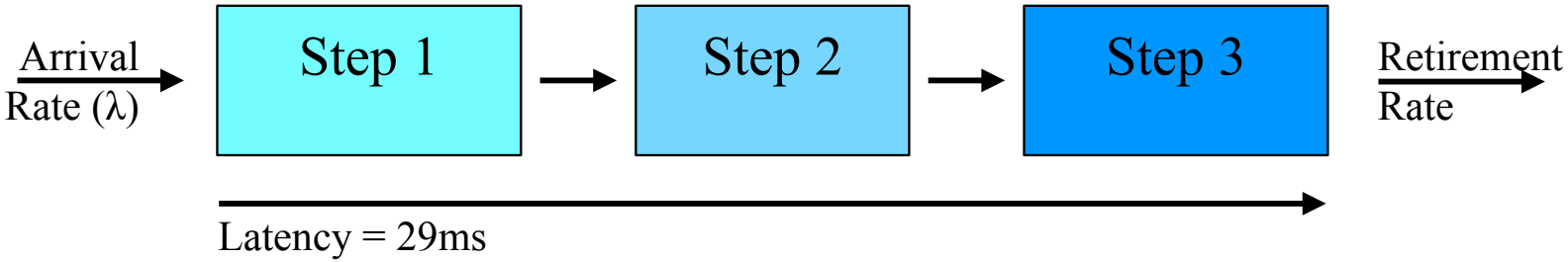
$\lambda = 67$ ← Observed maximal load

$W = 29ms$ ← Measured latency

$L = 67 * 0.029s$

$= 1.94$

Using Little's Law



$L = \lambda W$

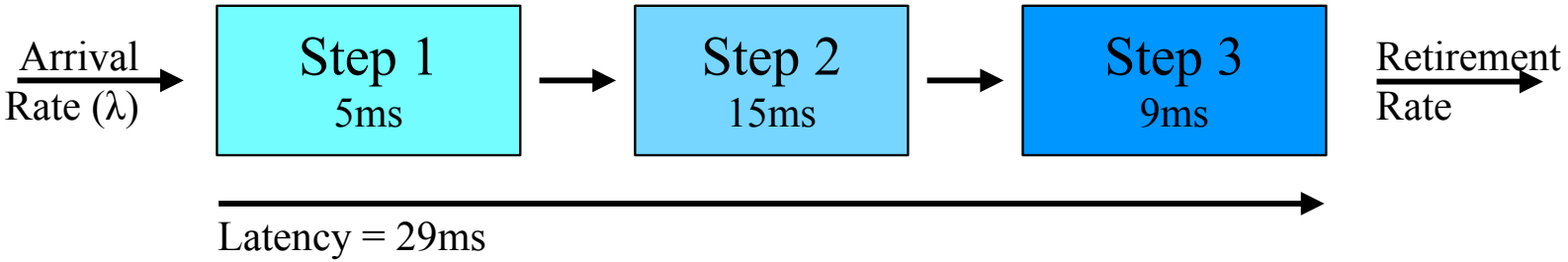
$\lambda = 67$

$W = 29ms$

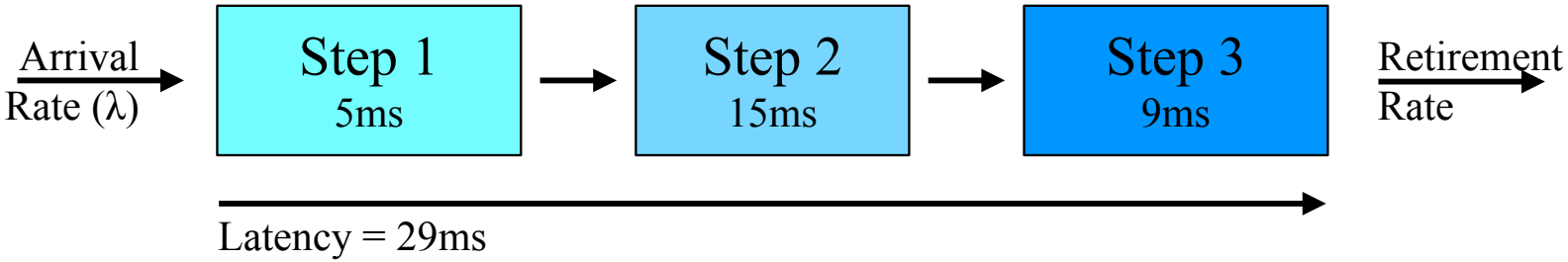
$L = 67 * 0.029s$

$= 1.94$ ← < 3, where 3 is the capacity of this system

Add Time Budgets



Calculate Throughput

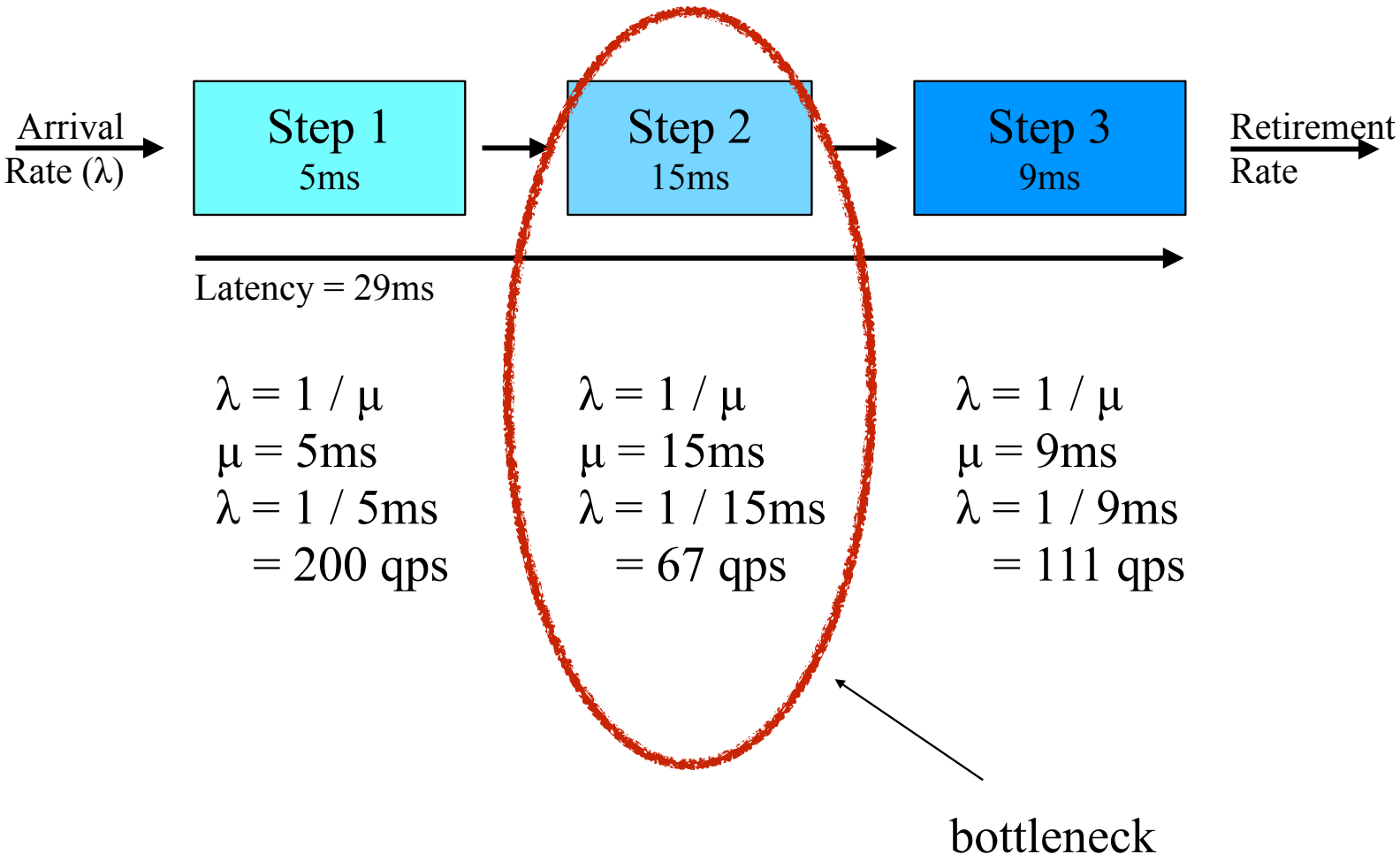


$\lambda = 1 / \mu$
$\mu = 5ms$
$\lambda = 1 / 5ms$
$= 200$ qps

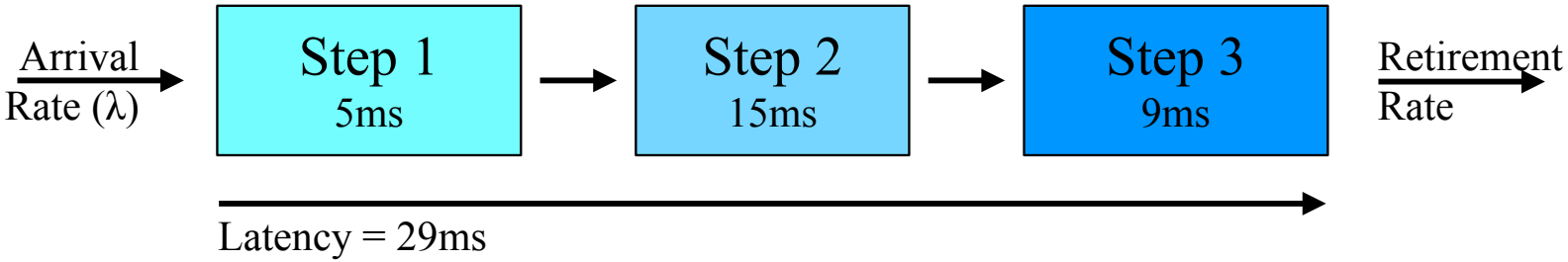
$\lambda = 1 / \mu$
$\mu = 15ms$
$\lambda = 1 / 15ms$
$= 67$ qps

$\lambda = 1 / \mu$
$\mu = 9ms$
$\lambda = 1 / 9ms$
$= 111$ qps

Calculate Throughput

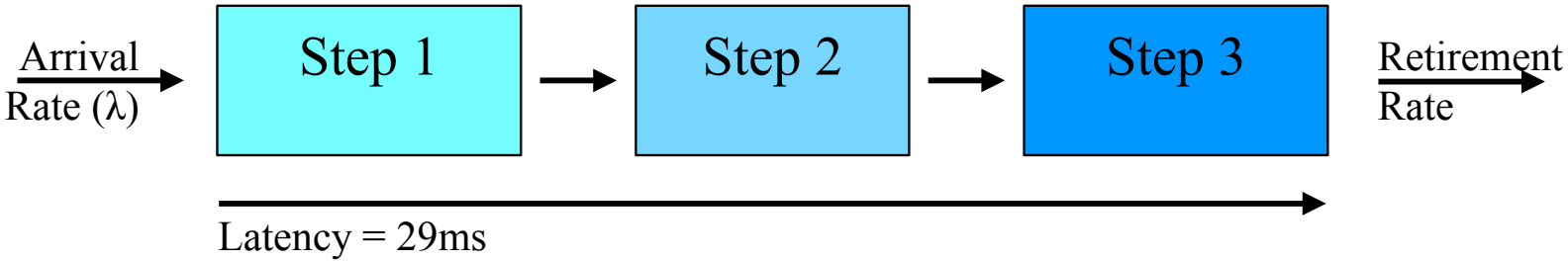


Theory of Constraints



- Tuning before the bottleneck
  - stacks up work in a queue
    - adds to latency
  - increases administration/ coordination costs
    - more work == more latency
  - impact: performance drops

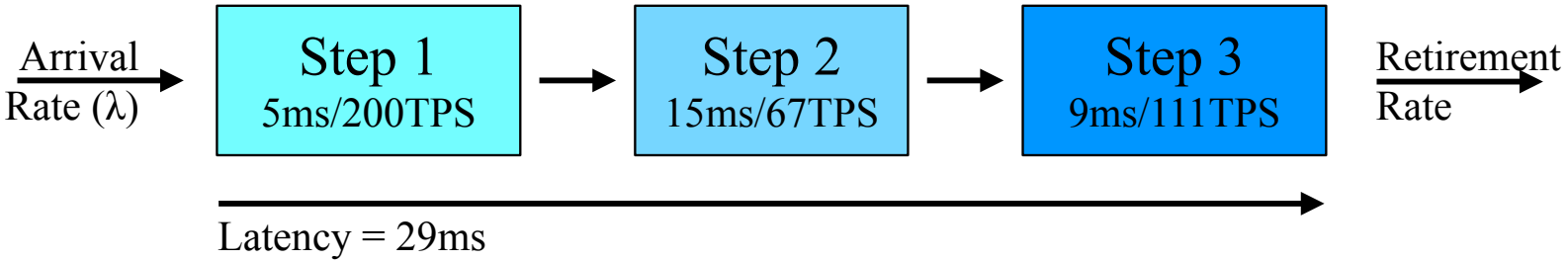
Theory of Constraints



- Tuning before the bottleneck
  - stacks up work in a queue
    - adds to latency
  - increases administration/ coordination costs
    - more work == more latency
  - impact: performance drops

- Tuning after the bottleneck
  - creates an illusion of better performance
  - Throughput will still be limited by Step 2
    - Small fluctuations in load will wipe out gains in Step 3

Throughput Revisited

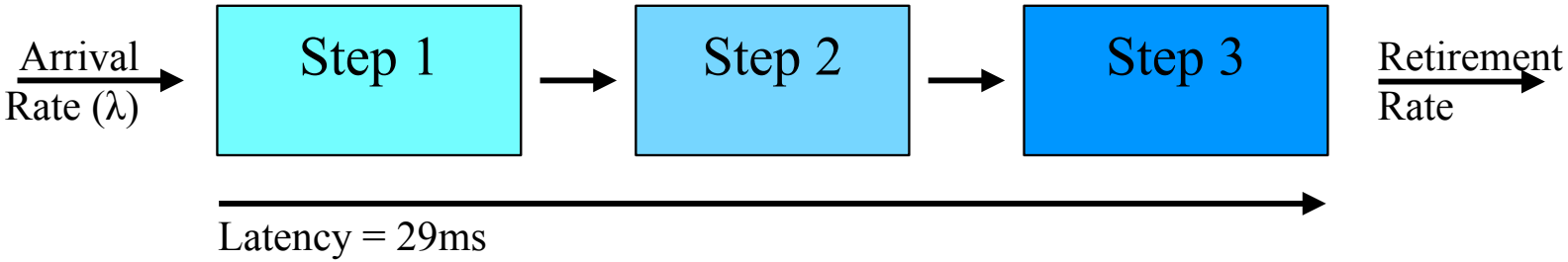


- System can manage a max arrival rate of 67QPS
  - Any more and Step 2 will not be able to catch up

- If arrivals fluctuate above 67 TPS excess arrivals need to be queued or will be lost
  - Random arrivals when service is busy need to be queued
  - Queues add **wait** time to the response time
    - Latency = 29ms + time in queue (wait time)

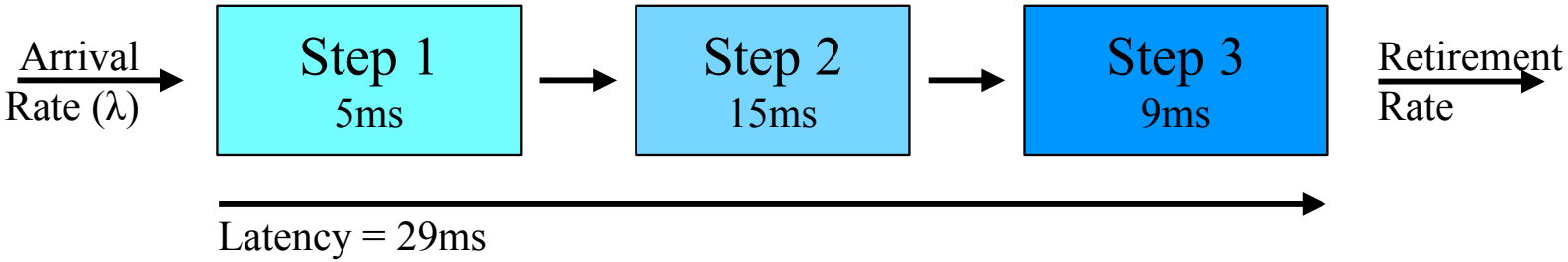
code word for "wait/dead time"

Theory of Constraints



- Tuning before the bottleneck
  - stacks up work in a queue
    - adds to latency
  - increases administration/ coordination costs
    - more work == more latency
  - impact: performance drops
- Tuning at the bottleneck
  - yields the largest performance improvement
    - unless it's masking another significant bottleneck
- Tuning after the bottleneck
  - creates an illusion of better performance
  - Throughput will still be limited by Step 2
    - Small fluctuations in load will wipe out gains in Step 3

Calculate Utilization



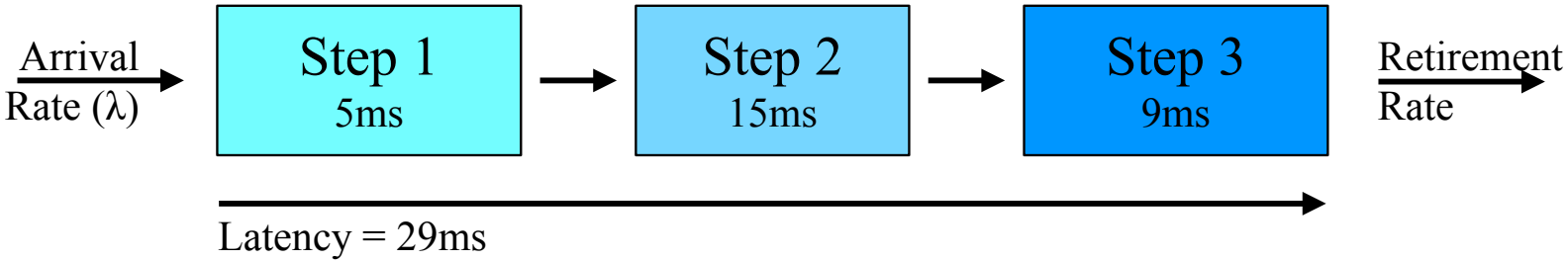
$L = \lambda W$
$\lambda = 67$
$W = 5ms$
$L = 67 * 0.005s$
$= 0.34$

$= 1.94$

$L = \lambda W$
$\lambda = 67$
$W = 15ms$
$L = 67 * 0.015s$
$= 1.0$

$L = \lambda W$
$\lambda = 67$
$W = 9ms$
$L = 67 * 0.009s$
$= 0.6$

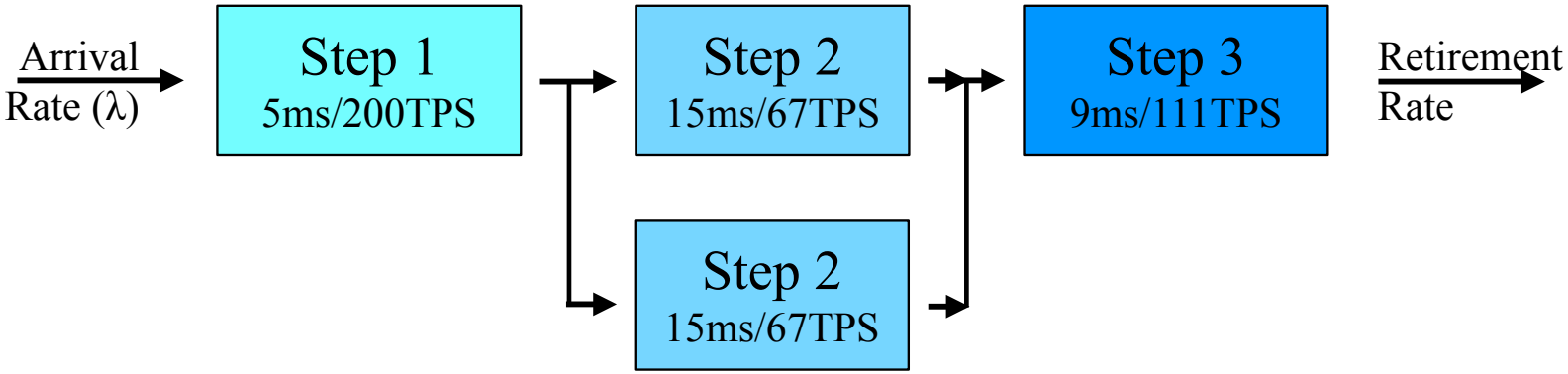
How to Tune?



$$L = \lambda W$$
$$\lambda = 67$$
$$W = 15ms$$
$$L = 67 * 0.015s$$
$$= 1.0$$

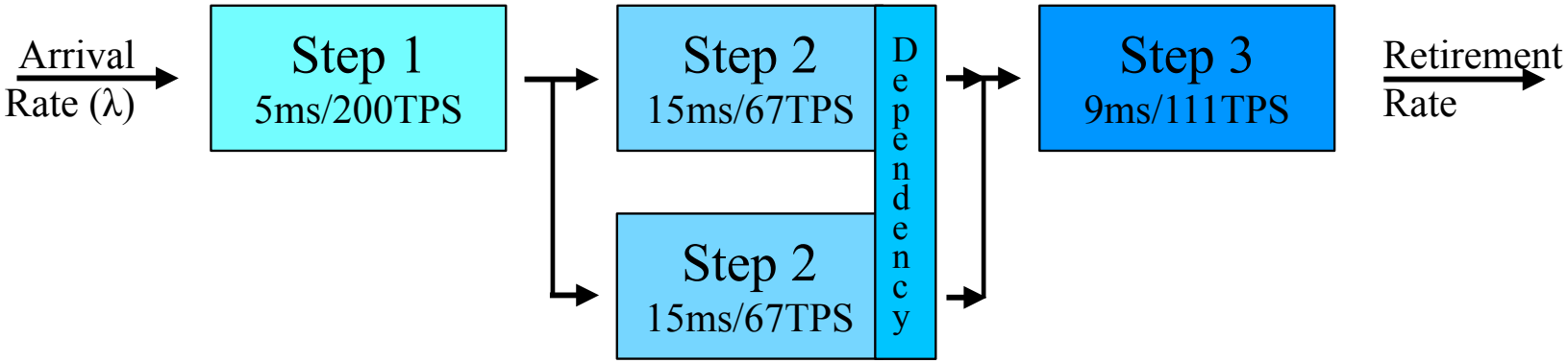
- Should we increase capacity?
  - How much capacity is needed?
- Should we improve the efficiency?
  - will this give us what is needed?

Scale Out



- Assumes node is perfectly scalable
  - no sharing/dependencies
  - if not, Amdahl's law applies
- Scale out to 3 nodes yields 201 TPS capacity
  - 3rd node isn't needed as Step 3 can only manage 111 TPS (<134TPS)
- Uses more compute resources

Scale Out



- Dependency will create a point of serialization
  - The realized throughput of Step 2 will be throttled by the service time of the dependency
  - performance predicted by Amdahl's law
- Overall, latency will remain at 29ms

Amdahl's Law

$$S = \frac{1}{(1 - P) + P / N}$$

- S - predicted speedup
- P - % parallelization, (0,1)
- N - number of cores

Amdahl's Law Example

$$S = \frac{1}{(1 - P) + P / N}$$

- S - predicted speedup
- P - % parallelization, (0,1)
- N - number of cores

P = 100%
N = 24

$$S = \frac{1}{(1 - 1) + 1 / 24}$$

$$= 24$$

Amdahl's Law Example

$$S = \frac{1}{(1 - P) + P / N}$$

- S - predicted speedup
- P - % parallelization, (0,1)
- N - number of cores

P = 100%
N = 24

$$S = \frac{1}{(1 - 1) + 1 / 24}$$

$$= 24$$

P = 0%
N = 24

$$S = \frac{1}{(1 - 0) + 0 / 24}$$

$$= 1$$

Amdahl's Law Example

$$S = \frac{1}{(1 - P) + P / N}$$

- S - predicted speedup
- P - % parallelization, (0,1)
- N - number of cores

P = 100%
N = 24

$$S = \frac{1}{(1 - 1) + 1 / 24}$$

$$= 24$$

P = 50%
N = 24

$$S = \frac{1}{(1 - .5) + .5 / 24}$$

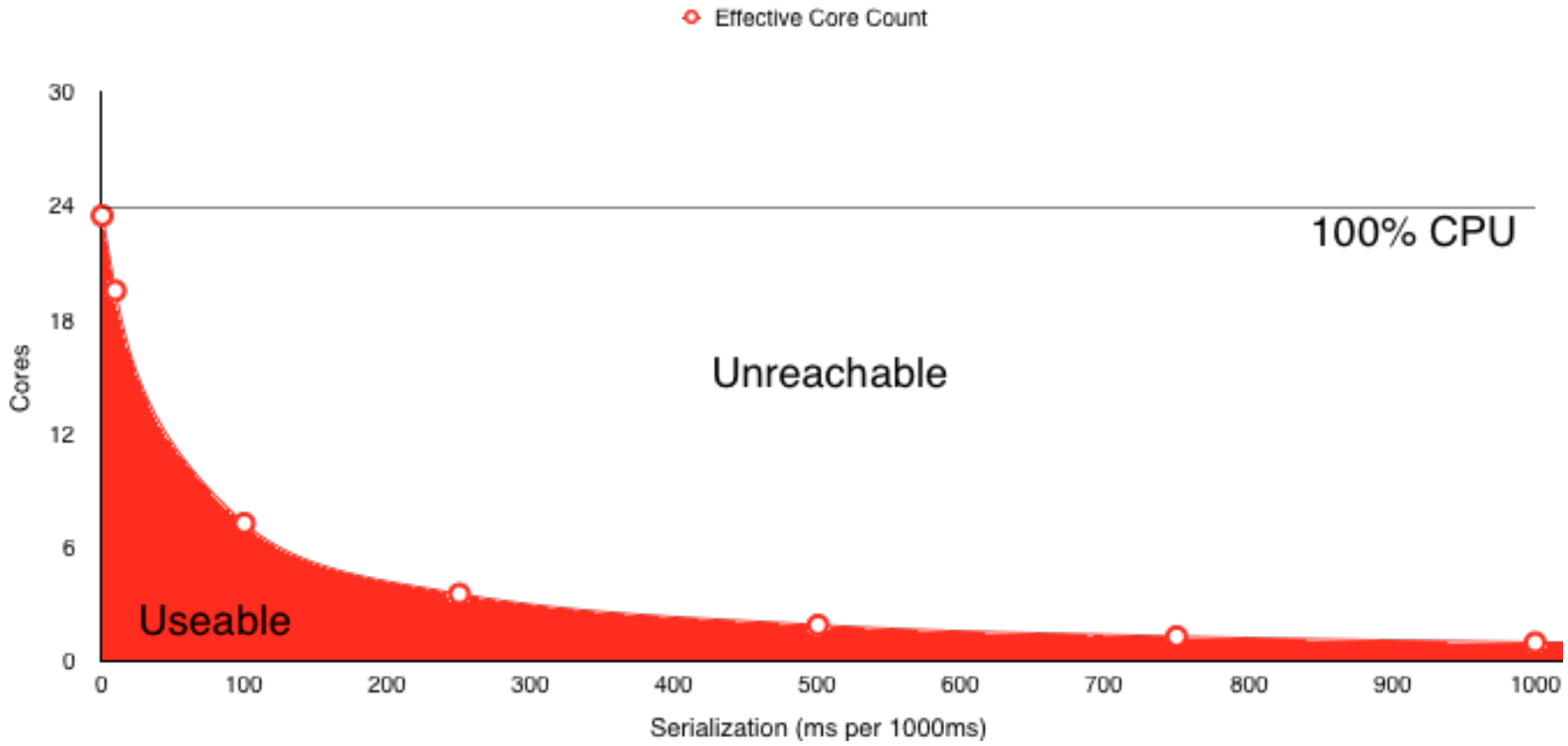
$$= 1.92$$

P = 0%
N = 24

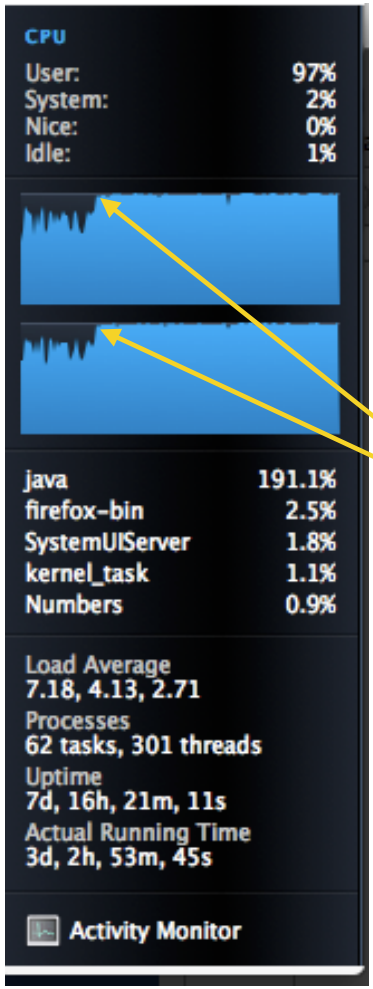
$$S = \frac{1}{(1 - 0) + 0 / 24}$$

$$= 1$$

Amdahl's Law



Effect of Reduced Serialization



Benchmark shifts from a mostly concurrent to a fully concurrent algorithm

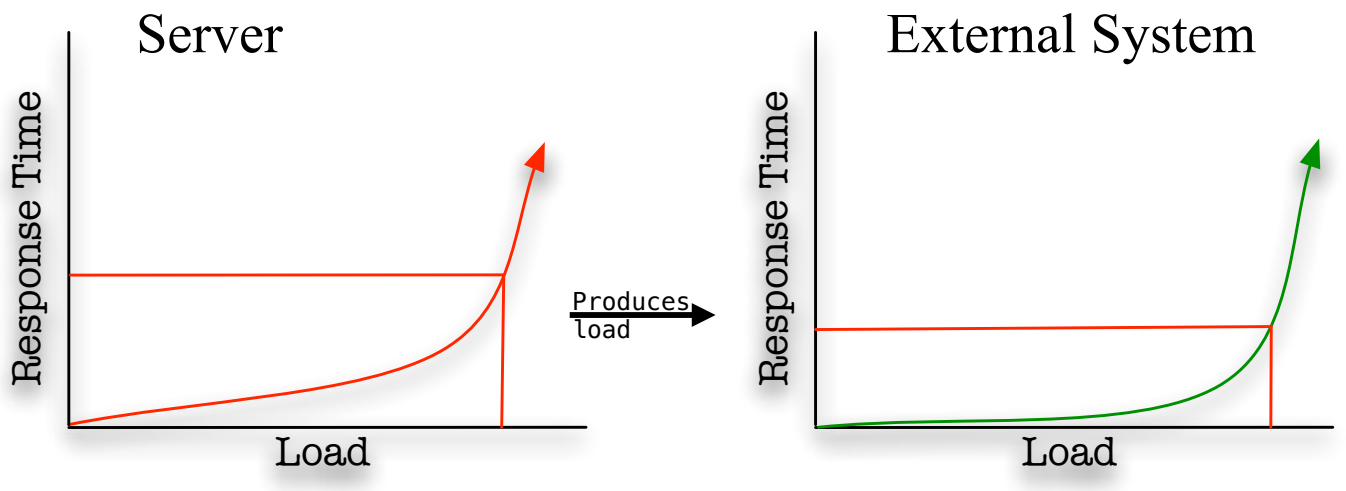
Sources of Serialization

- Software
  - manages soft shareable resources by enforcing single access via locks (mutex)
  - Dependencies between separate threads will limit throughput

- Operating Systems facilitate the competition for access to the hardware
  - scheduling add coherency costs
    - thread scheduling is especially egregious
  - queuing adds to dead time
  - stalls add to dead time

- Hardware/compute platforms are finite
  - execution is necessarily serialized for individual hardware components
  - stalls add to dead time

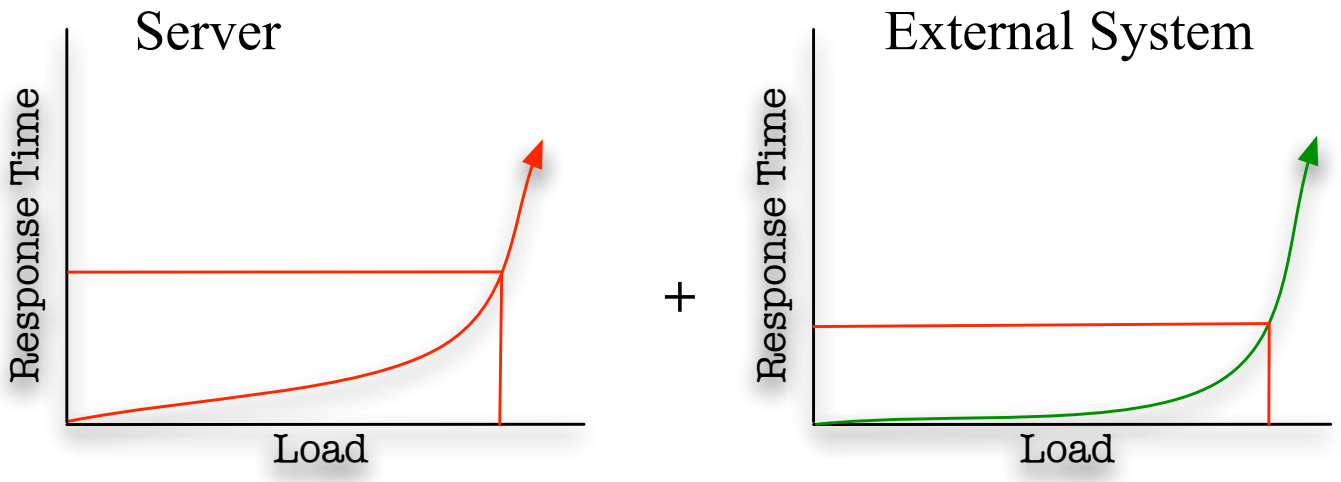
Working With Response Time Curves



- Consider an over simplification of the summation of response time curves.
  - in reality the response time will be the sum of many curves plus other things
- An application consists of a server that communicates with an external service provider
  - database, web service, another server…..
- Behavior is predicted by ToC
  - modeled by Little's Law

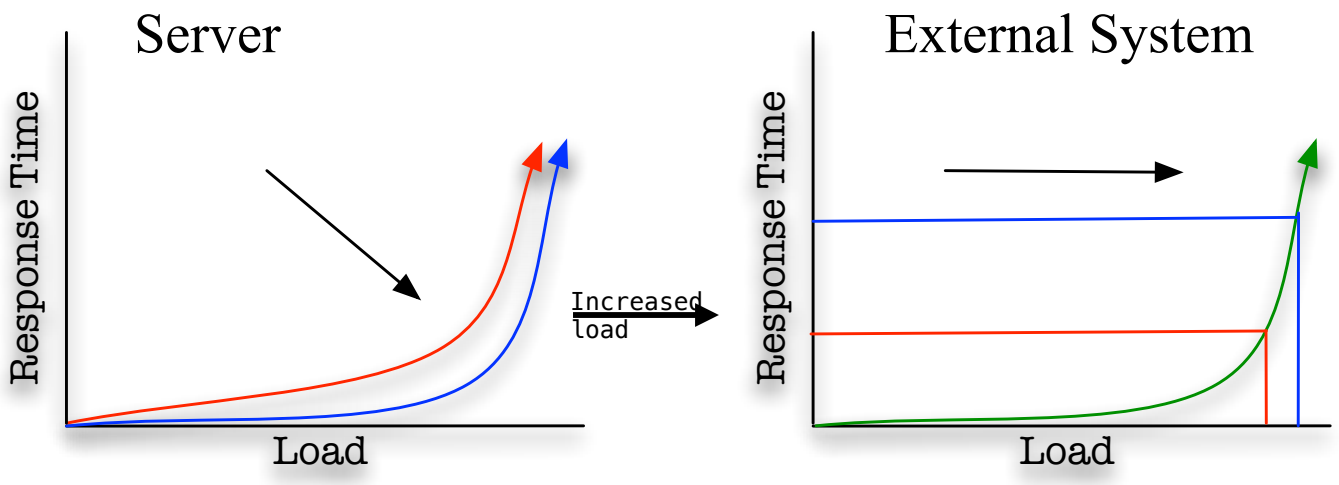
Latency

R_t =



- Response time of the Server is a function of load on the System
- The load on the External System will be a function of the response time of the Server
  - response time of the External System is a function of the load it is experiencing
- End user response time = Server(R_t) + External System(R_t)

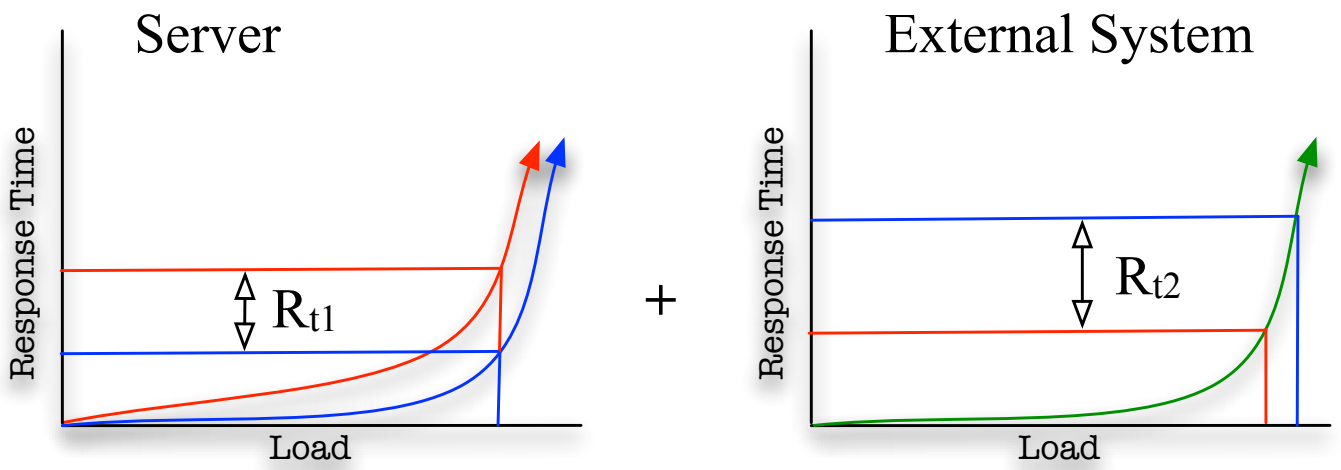
Tune Server



- Tuning the server pushes the response time curve down and out yielding less latency
  - less latency leads to more pressure on downstream systems

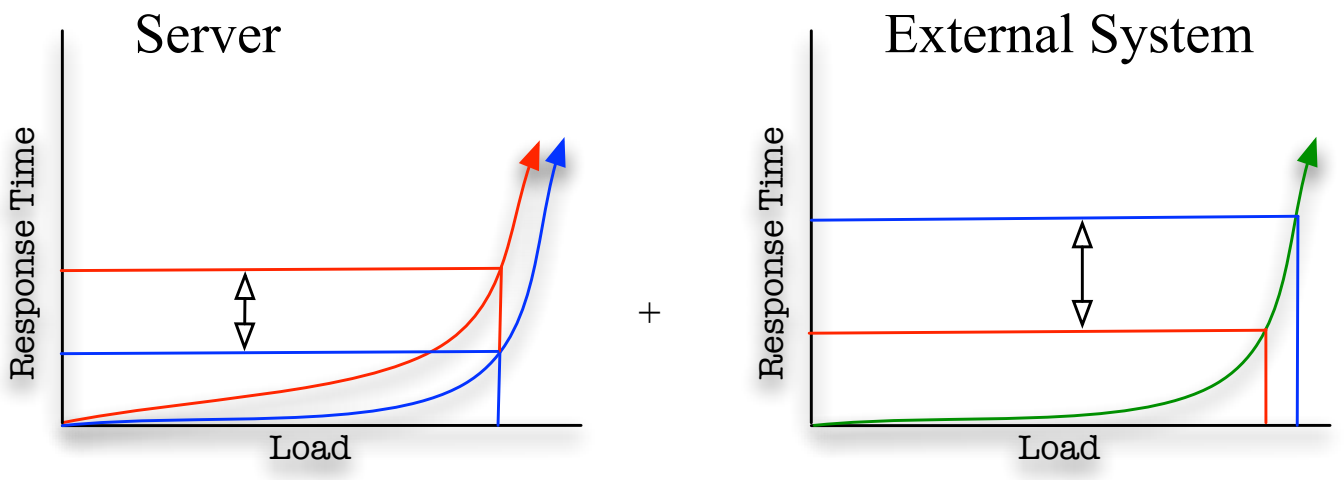
Effect on Response Time

R_t =



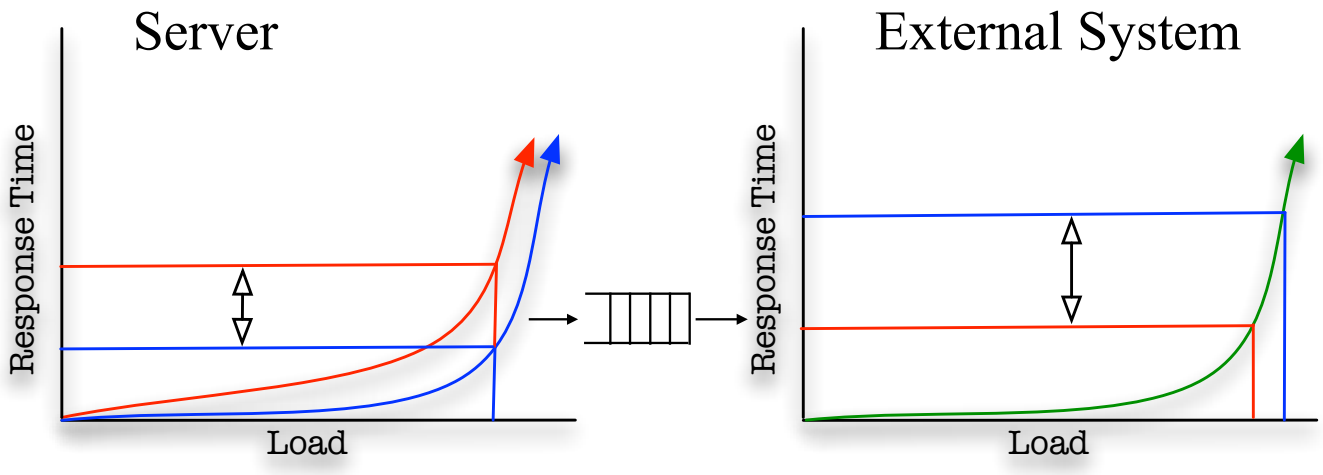
- R_{t1} is the reduction in latency
- R_{t2} is an increase in latency
- Did we tune the system?

Our system is slower, now what?



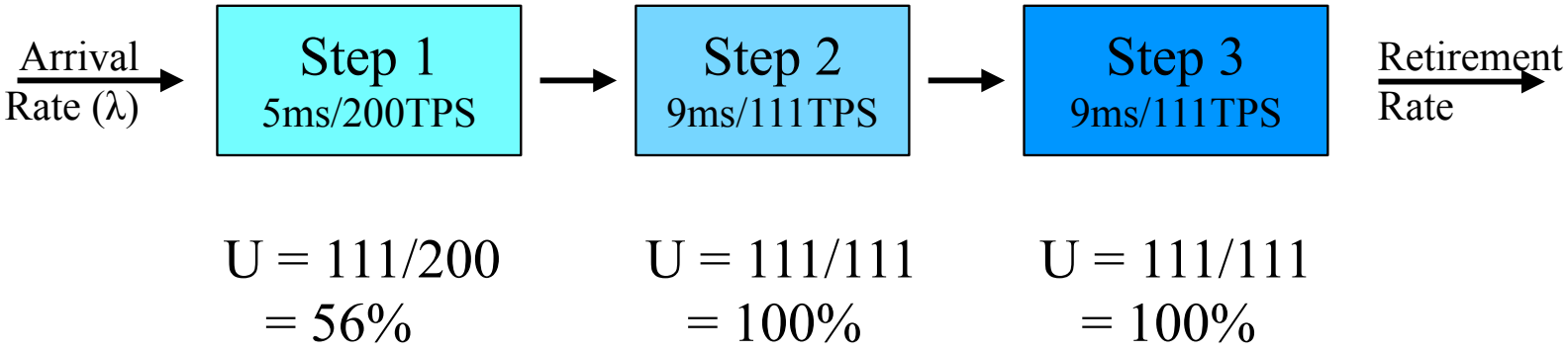
- Tune external system
- Roll back on changes
- Throttle system by introducing some form of a (work) queue

Dead Time



- Queuing introduces dead time into the response time
  - latency = Server(R_t) + External System(R_t) + dead time (in queue)
  - dead time quite often dominates latency
  - generally need to balance the dead time component

Option# 2



- Improve performance to reduce latency
  - manage higher loads without extra hardware
  - assumes step 3 can manage higher loads
    - there is always going to be a bottleneck
- **Stop condition**: latency meets expectations/requirements

Conclusion

- Tuning can have very unpredictable affects on a system
- ToC tells us that tuning before the bottleneck will decrease performance
  - A component that performs well in test maybe the cause of performance regression in production

Note on Measuring Latency

- Measuring latency is fraught with difficulties
  - measuring in the application can be a performance bottleneck
    - depends on how measurement is performed and reported on
  - measurements can be completely wrong
    - does the measurement measure more or less than it should
      - the effects of the pause will be missing from the measurement
      - timer may include other unwanted activities

- Current frameworks
  - come with large overheads
  - may allow for sampling of data that has already been collected
  - track single requests though system
    - perpetuate the assumption that systems are fair

- Recent study showed that 800:1 random sampling of a latency data set yield a statistically similar data set.

Treatment of Measures

- Latency
  - Min, max
  - tail percentiles, 95th, 99th, 99.9, 99.99, …
  - medium and mean are some what meaningless
  - tail latencies are seen more often than you'd think

- Throughput
  - min, max, mean
  - varying time windows, 10s, 20s, 30s, 1m, 5m, 10m
    - trending information

Seeing the 99th %

- What are the chances that your users will experience long latencies?
  - function of the number of requests per transaction
- Calculate likelyhood of p99 for 2 requests in a transaction

$$1 - 0.99^2 \sim= 2\%$$

- Real world example
  - multi-second response time at ~ every 1,000,000th request
  - average of 20,000 requests per transaction (p999999)

$$1 - (1 - 1/1000000)^{20000}$$
$$= 1 - 0.999999^{20000}$$
$$= 2\%$$

Seeing the 99th %

- What are the chances that your users will experience long latencies?
  - function of the number of requests per transaction
- Calculate likelyhood of p99 for 2 requests in a transaction

$$1 - 0.99^2 \sim= 2\%$$

- Real world example
  - multi-second response time at ~ every 1,000,000th request
  - average of 20,000 requests per transaction (p999999)

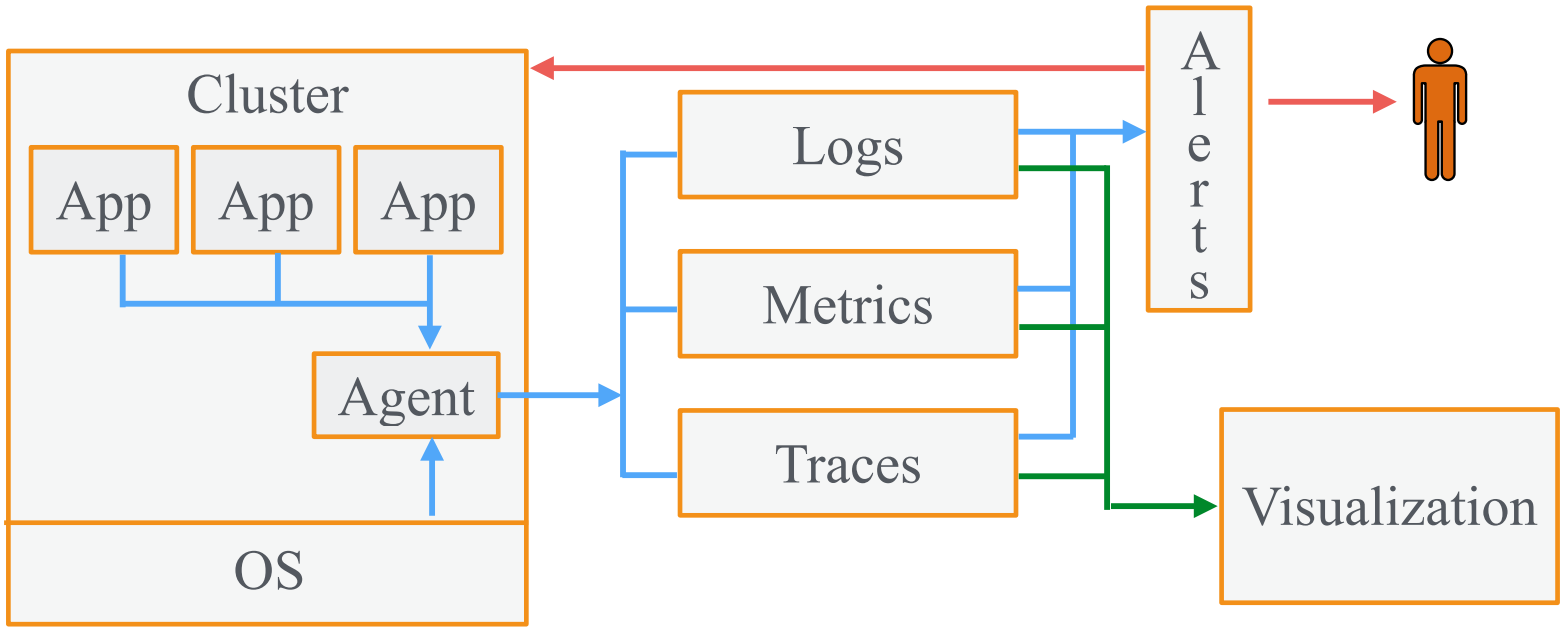
$$1 - (1 - 1/1000000)^{20000}$$
$$= 1 - 0.999999^{20000}$$
$$= 2\%$$

OpenTelemetry

- Open source collection of tools, APIs and SDKs for producing, collecting and exporting telemetry data
  - traces, metrics, and logs
- Cloud Native Computing Foundation project
  - multi-language, multi-platform

OpenTelemetry

- Instrument your code
  - inject happens automatically and out of your control
- Pick off the shelf components to fill in for different functions



Notes on Benchmarking

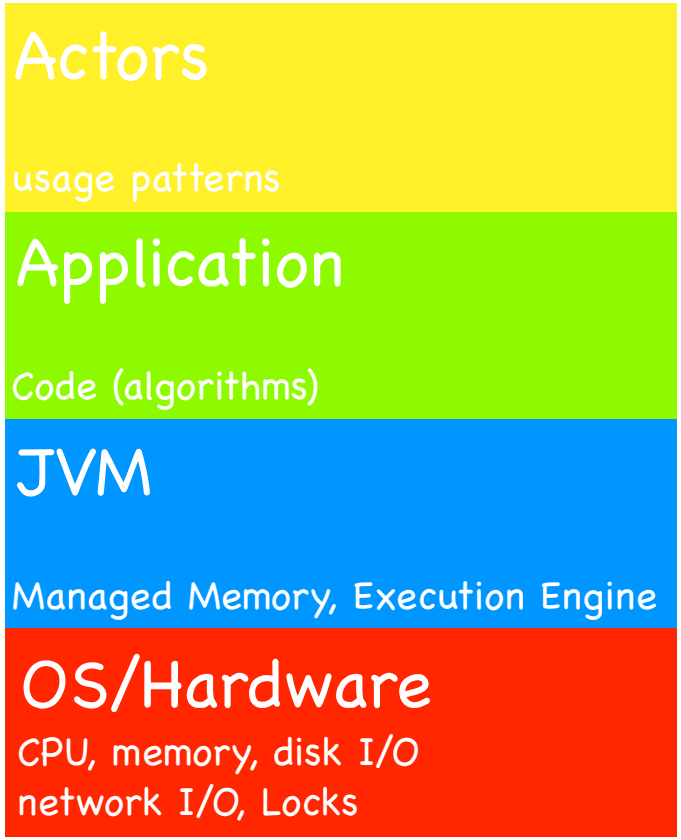
- Benchmark is a standard point of reference that comparisons can be made against
  - run a set of operations in order to compare the relative performance of a target
  - experimental process
    - need to control identify and control variables
- Performance benchmarking
  - Test designed to assess the responsiveness, throughput, and scalability of a UoT under a controlled workload
- Benchmark is broken into 2 parts
  - Unit under Test
  - Test Harness
    - everything else

Quote from a Customer

“the problem was not spotting the problem itself, but that our load generator did not trigger these problem. They were simply not visible in the system under load tests. Just after going productive, the real users behaved differently to the load generators”

What jPDM Tells Us

- If any detail in any layer changes
  - the model has changed
- Implies - benchmarking environment must be identical to the production environment
  - any differences will shift or hide bottleneck
- Generally, emulating the actor layer is the most difficult task
  - many projects fork production load onto test servers
  - good news is that sometimes close is good enough



Workload Defined

- Need a constant so that you can compare results across runs
  - measure how it takes to clear the fixed volume of work
  - measure how many transactions are retired for a fixed time window
- Focus on maintaining the required rate of arrival
  - not being able to maintain a schedule will skew results
- Ensure the bottleneck is in the Unit Under Test
  - a bottleneck in the test harness will interfere with it's ability to apply load at the prescribed schedule

Coordinated Omission

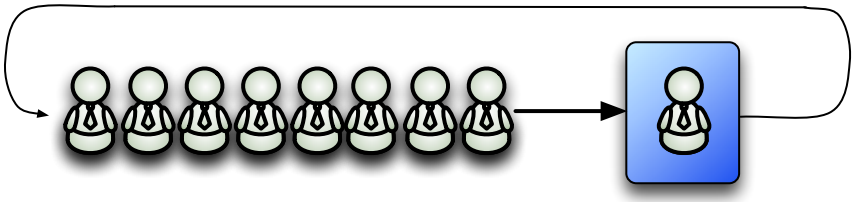
- Several different forms
  - common form occurs when the load injector is unable to send a request due to back pressure from the server
    - reduced load on the server allows for better than realized latency

- Most load test harnesses encourages CO
  - threads are to loop to repeat the script
  - individual steps within a script are delayed because the server wasn't able to respond to the previous request
  - both situations make it difficult to regulate the arrival rate of requests on the server

Closed Systems

- Characterized by
  - limited number of users
  - user re-joins the line as soon as it is finished

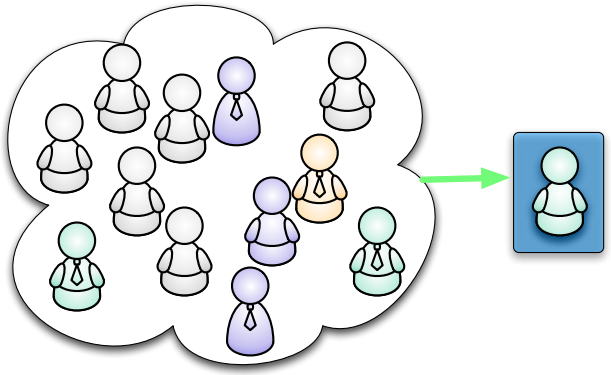
- Difficult to regulate request arrival rates
  - number of thread is fixed (to number of users)
  - system is self throttled
  - can’t start a new request while old ones are still being processed



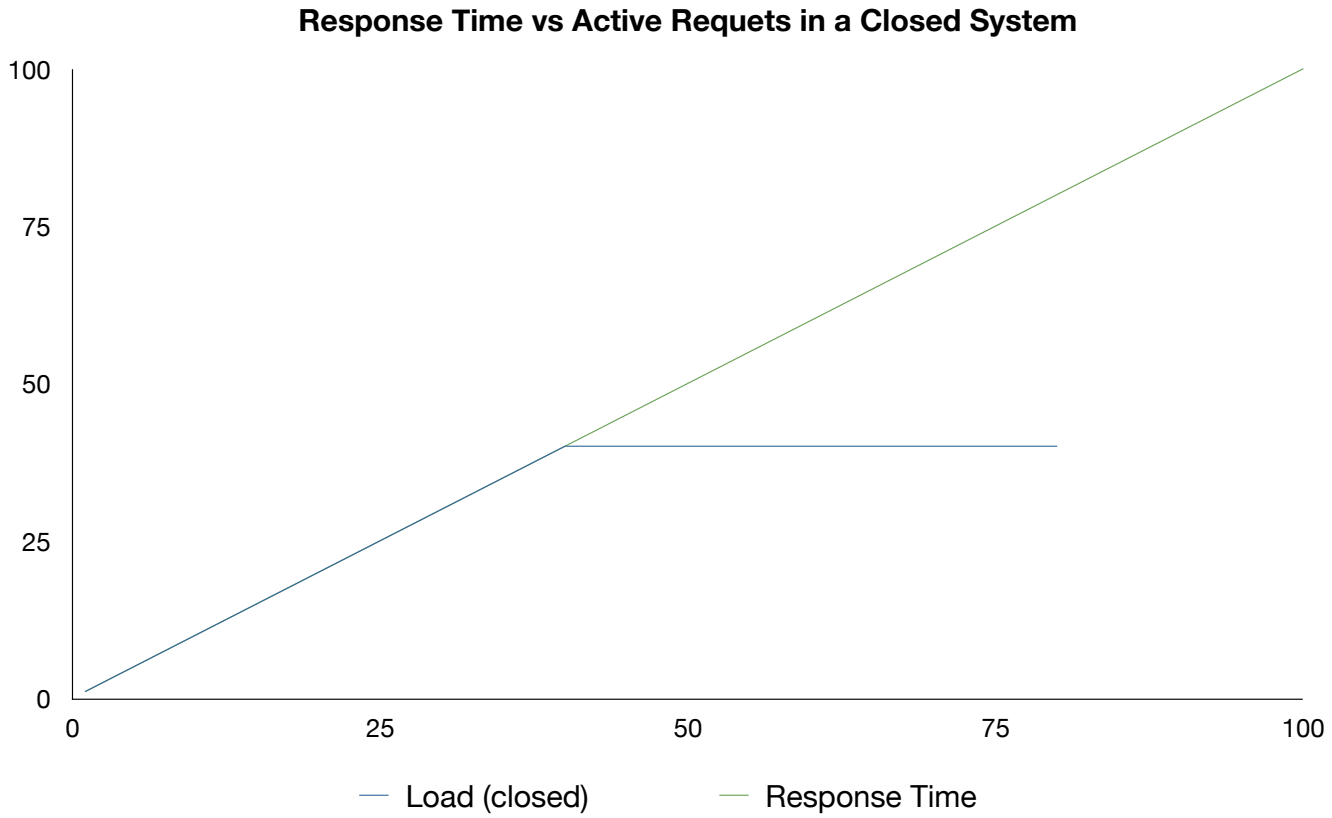
Open Systems

- Characterized by
  - unlimited number of users
  - users arrive independently according to a schedule

- System entry is independent of exit
  - new thread for every entry
  - can completely flood a system

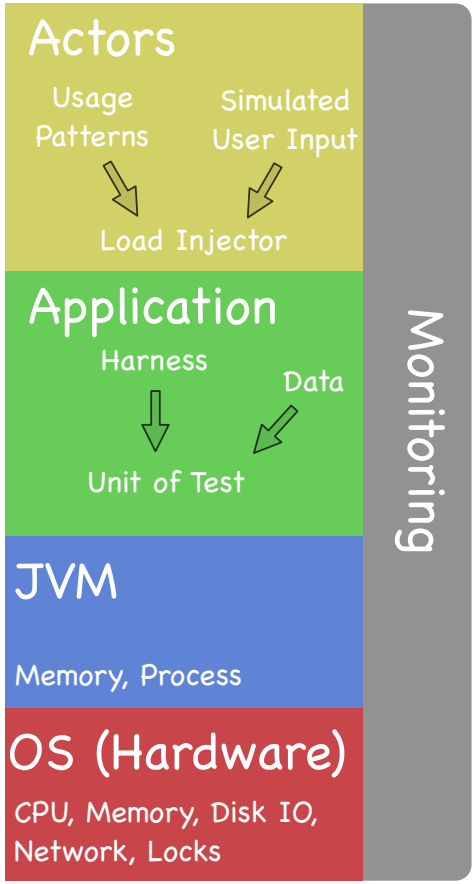


Impact on Response Times



jPDM applied to Benchmarking

- Any difference in any layer will likely produce different results
  - Usage pattern mix
  - Simulated user input
  - Load injection rates
  - How the harness accepts and responds to load
  - Application
  - JVM
  - OS/Hardware



Incorrect Concluions

- Benchmark errors can lead to incorrect interpretations
  - lead to misguided decisions
- Design errors
- Benchmark selection errors
- Execution errors
- Measurement errors
- Interpretation errors

Exercise 2

- The purpose of this exercise is to compare the differences in performance between a closed and open load test script. The scripts have been designed to produce the same load

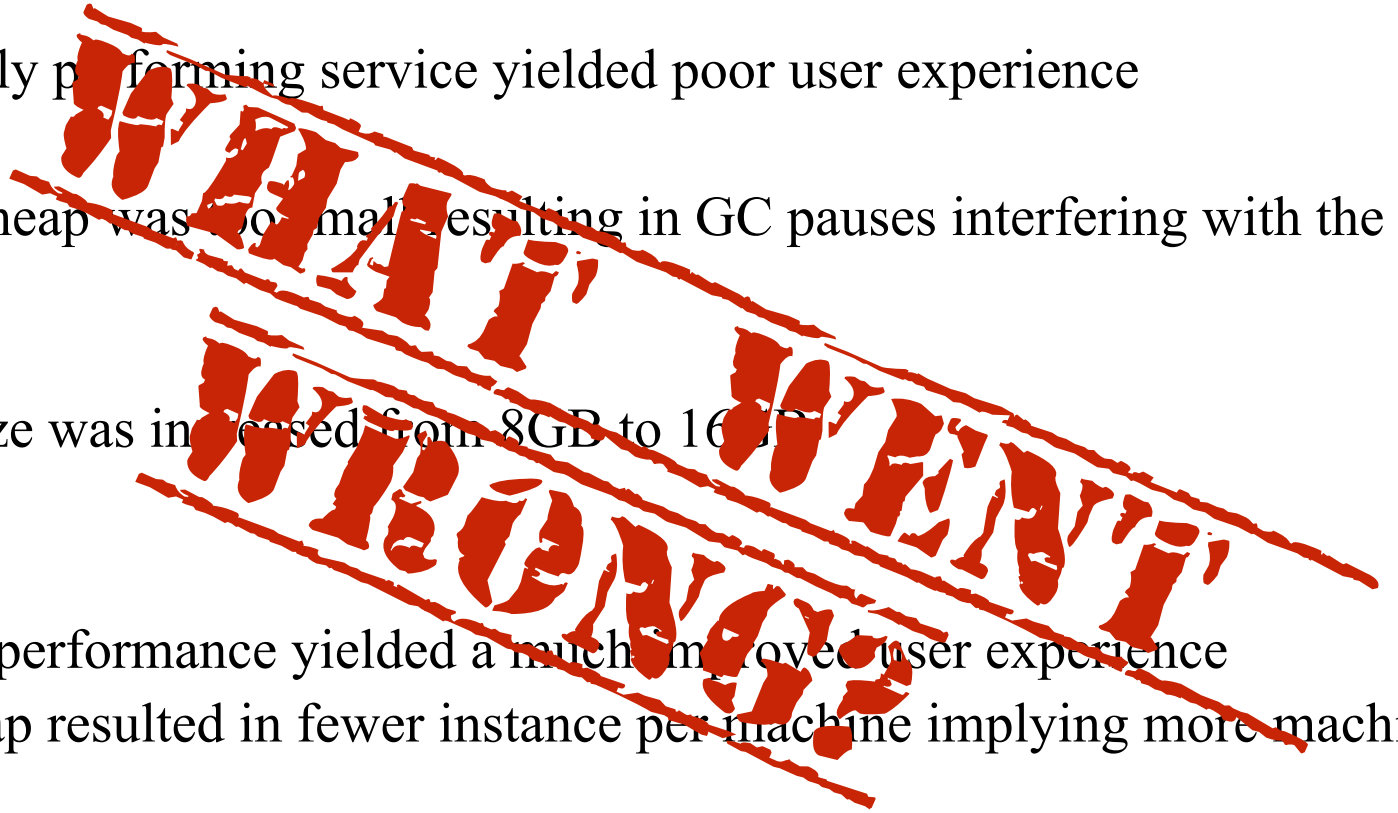
- Steps
  - cd into the exercise2 directory
  - run the compile script
  - run the appServerStart script
  - run the jmeter-closed script
    - press the green go arrow in jmeter to start the test
    - select aggregate view to monitor progress and response times
    - record the average response times for each of the queries
  - restart the application server
  - restart jmeter with the jmeter-open script
    - press the green go arrow in jmeter to start the test
    - select aggregate view to monitor progress and response times
    - record the average response times for each of the queries
  - compare the results from each of the two tests

Case Study

- Symptom: poorly performing service yielded poor user experience
- Analysis: Java heap was too small resulting in GC pauses interfering with the user experience
- Action: heap size was increased from 8GB to 16GB
- Result :
  - improved performance yielded a much improved user experience
  - Larger heap resulted in fewer instance per machine implying more machines were needed
    - increased COGS putting service into the red

Case Study

- Symptom: poorly performing service yielded poor user experience
- Analysis: Java heap was too small resulting in GC pauses interfering with the user experience
- Action: heap size was increased from 8GB to 16GB
- Result :
  - improved performance yielded a much improved user experience
  - Larger heap resulted in fewer instance per machine implying more machines were needed
    - increased COGS putting service into the red



Case Study

- Symptom: poorly performing service yielded poor user experience

- Analysis: ~~Java heap was too small resulting in GC pauses interfering with the user experience~~ jPDM analysis: System dominate

- Action: heap size was increased from 8GB to 16GB

- Result :
  - improved performance yielded a much improved user experience
  - Larger heap resulted in fewer instance per machine implying more machines were needed
    - increased COGS putting service into the red

Case Study

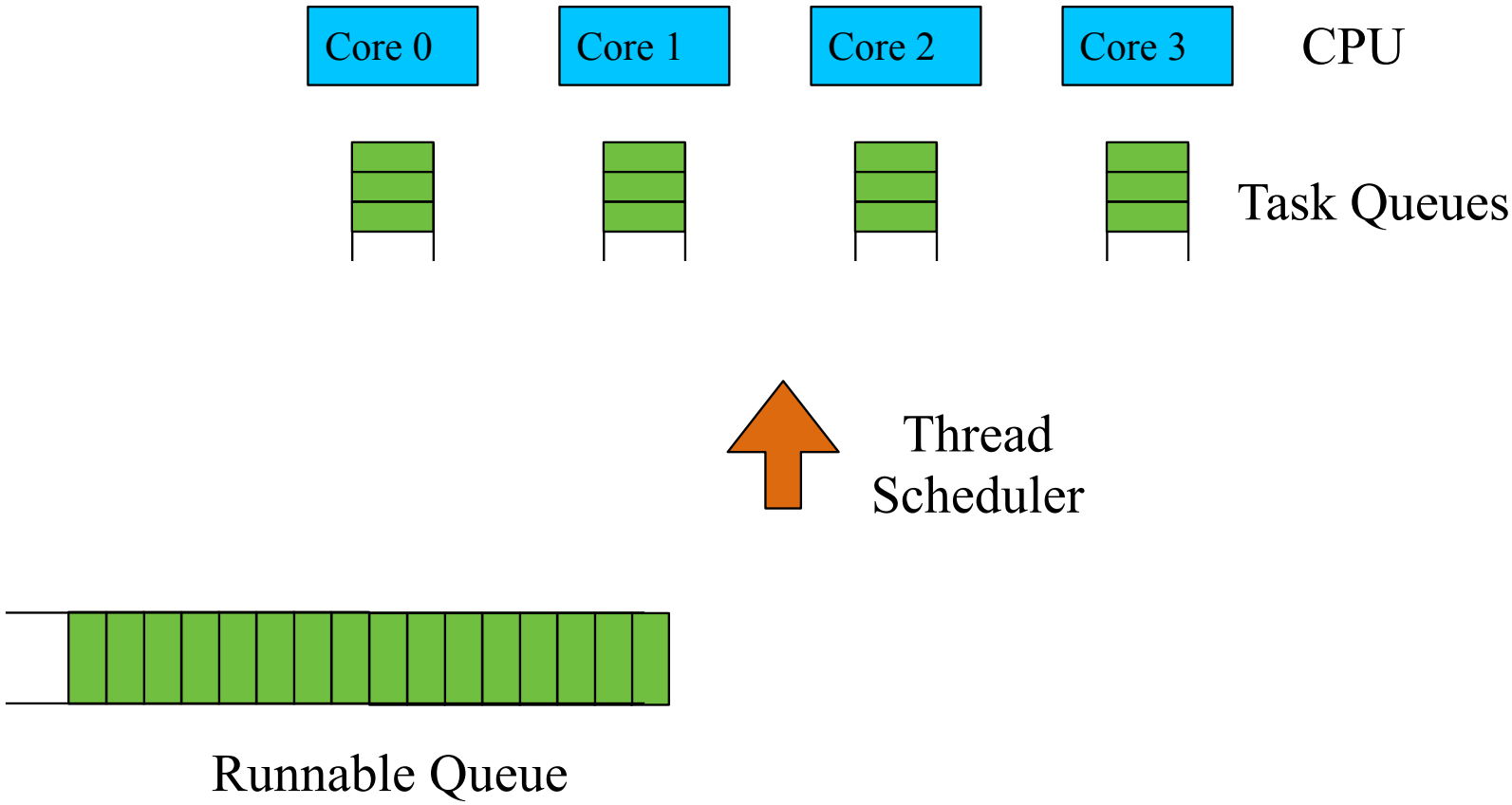
- Symptom: poorly performing service yielded poor user experience

- Analysis: Context switching rates are extremely high, Java heap was 8x larger than needed

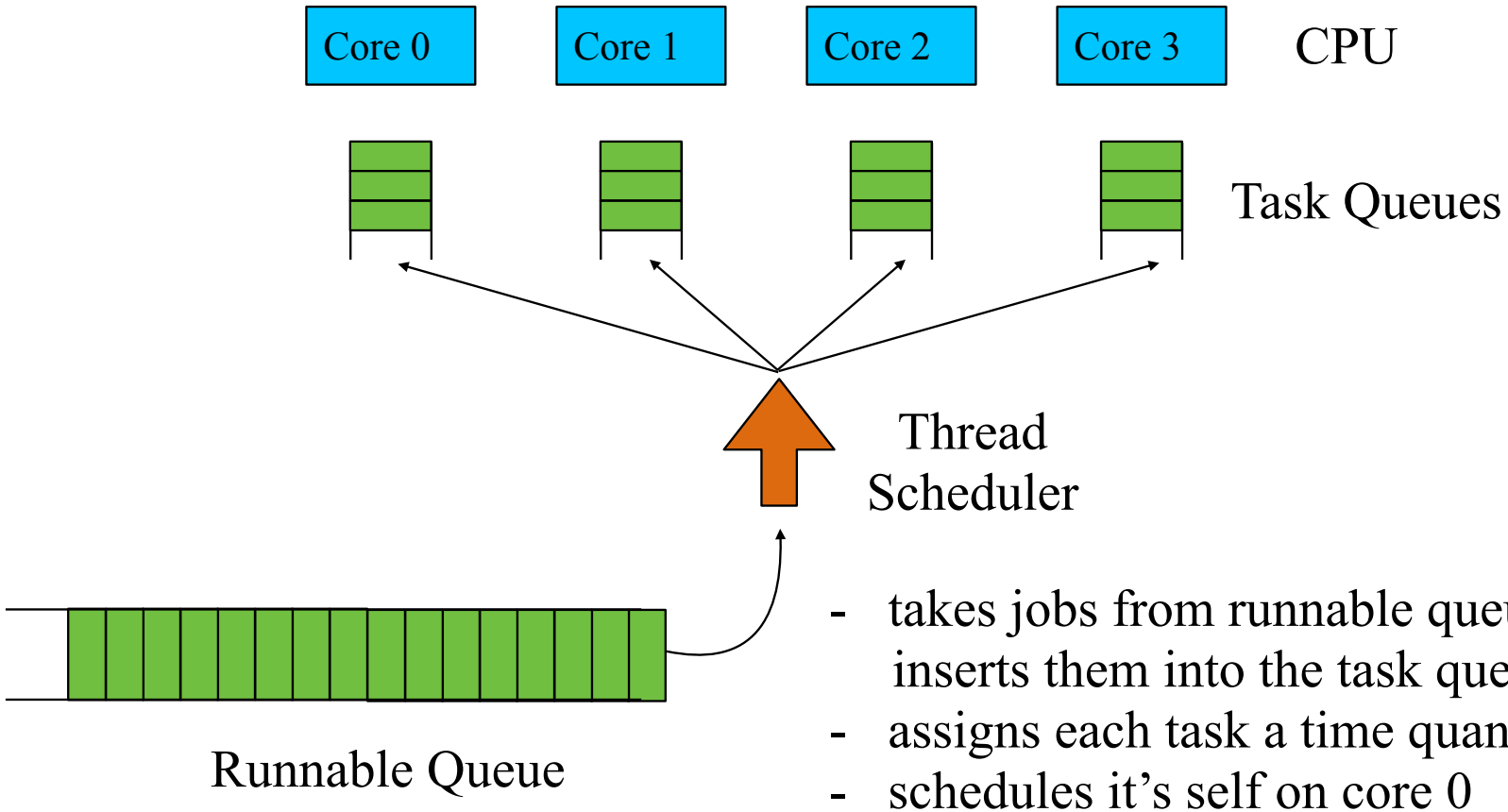
- Action:
  - redeploy onto smaller machines
  - reduce Java heap to 2GB

- Result :
    - improved performance yielded a much improved user experience
    - Using the same number of smaller cheaper machines reduces COGS
      - service now runs in the black

Thread Scheduling



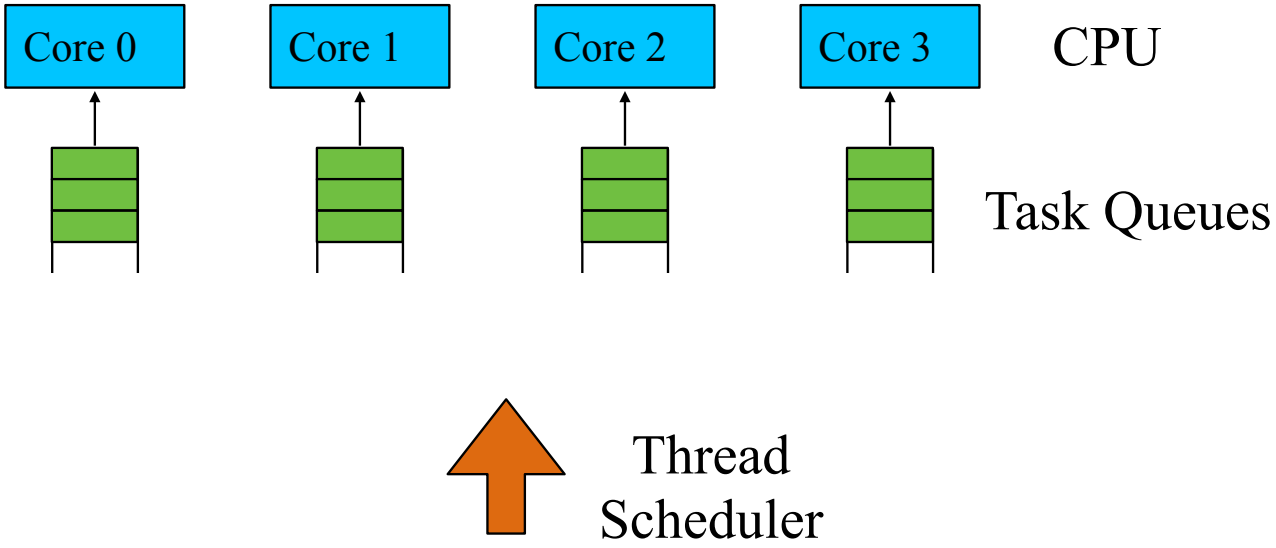
Thread Scheduling



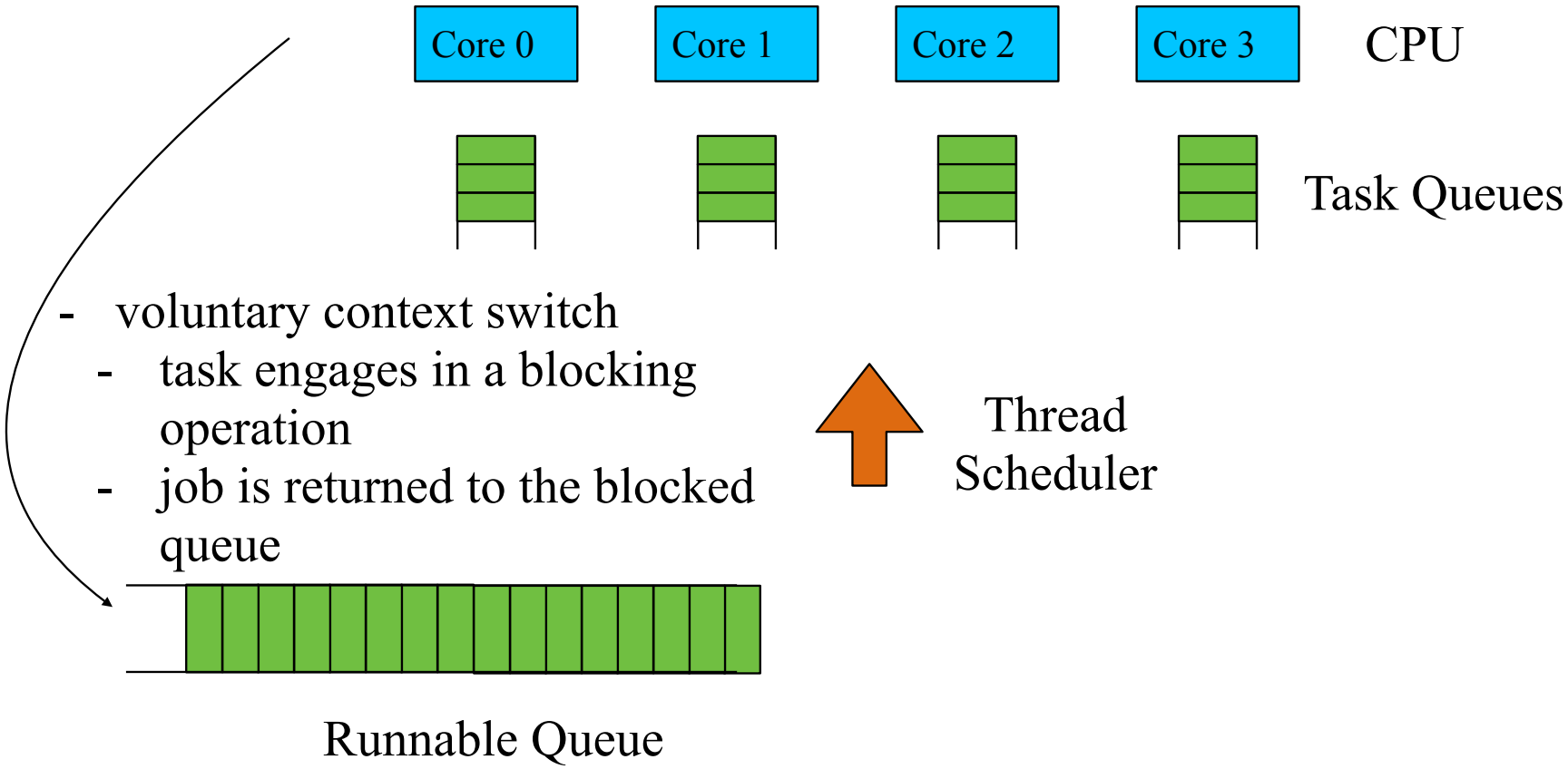
- takes jobs from runnable queue and inserts them into the task queues
- assigns each task a time quantum
- schedules it's self on core 0

Thread Scheduling

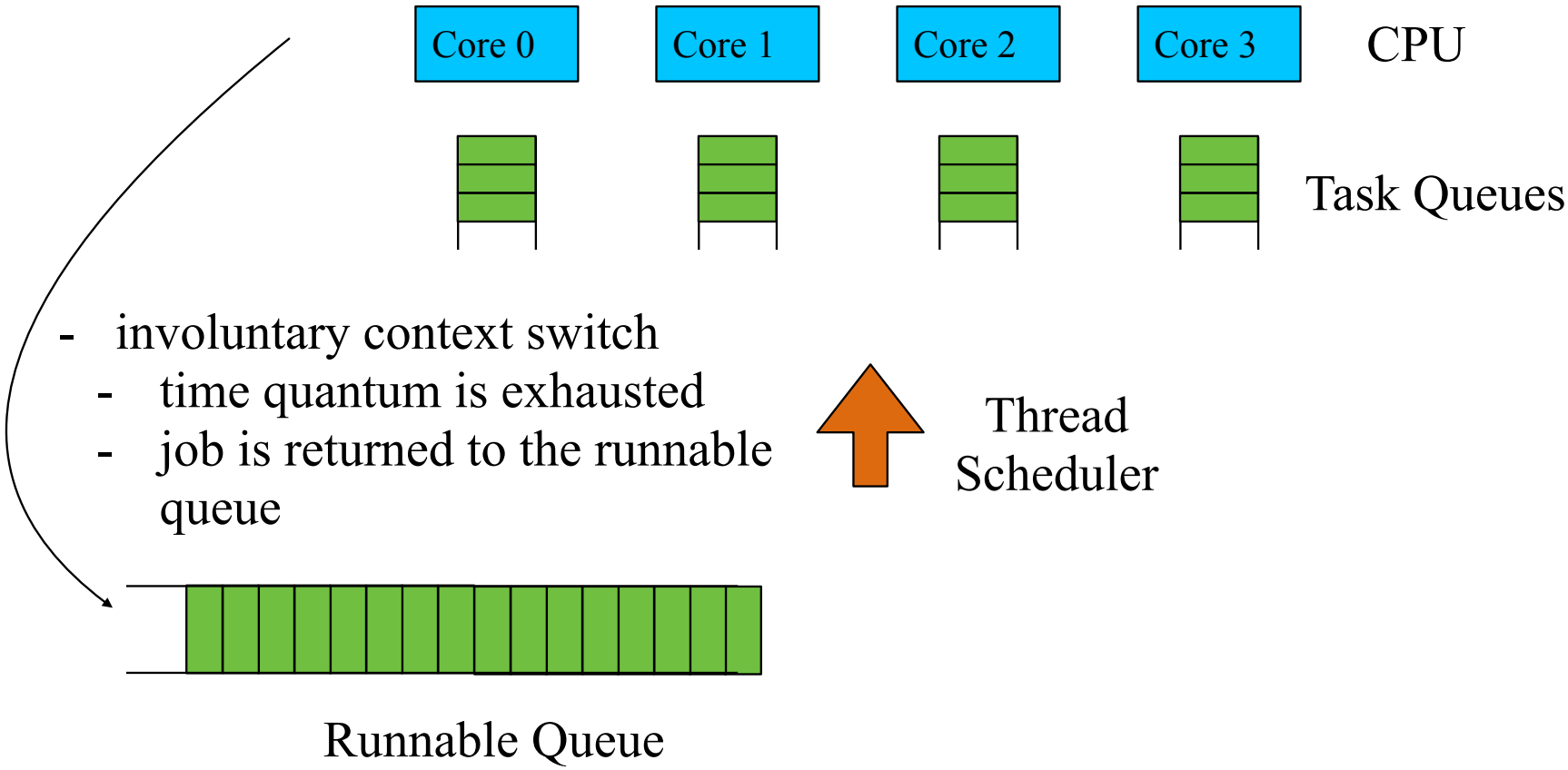
- takes task from task queue and executes it



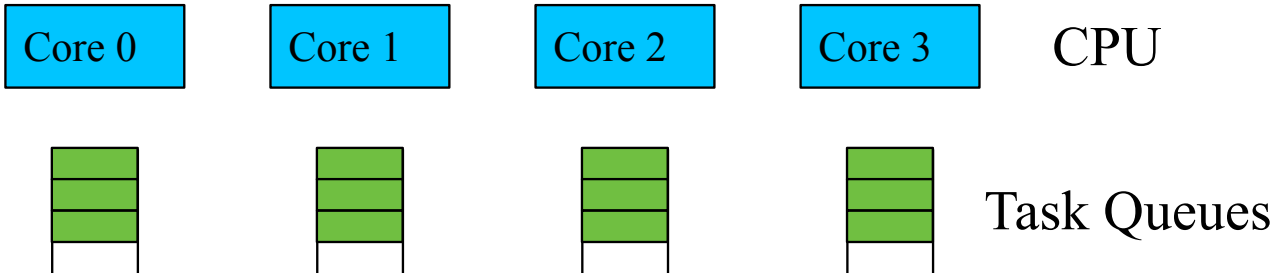
Thread Scheduling



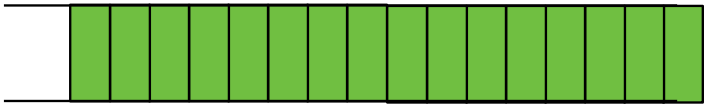
Thread Scheduling



Thread Scheduling



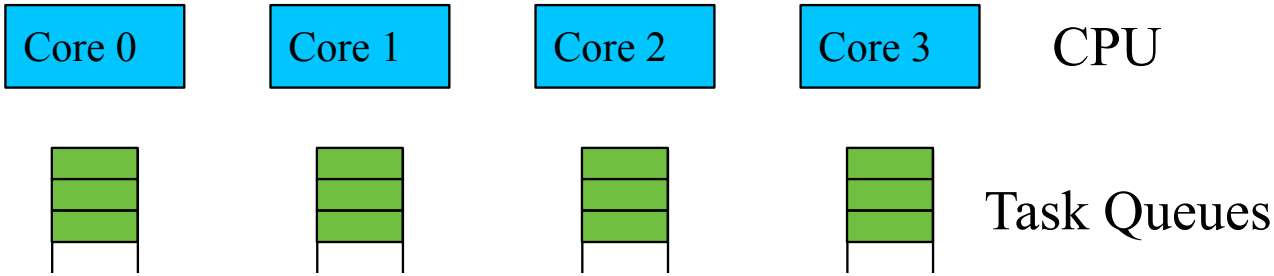
Thread Scheduler



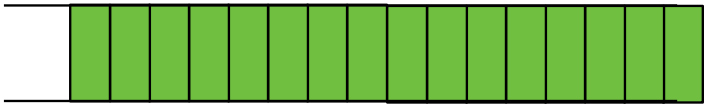
Runnable Queue

- CS rates are commonly seen as a function of the number of threads
- Little's law tells us Context Switching rates are a function of the time quantum

Thread Scheduling



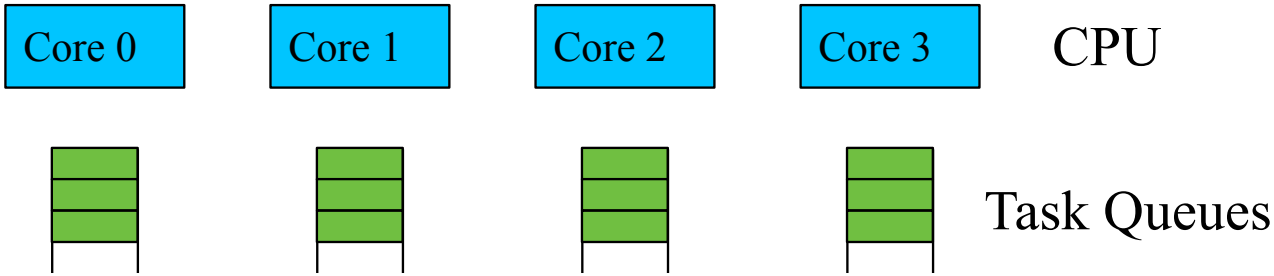
Thread Scheduler



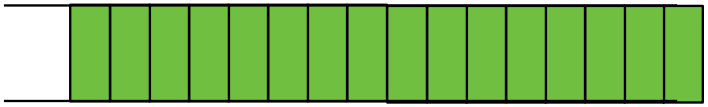
Runnable Queue

- A queued thread
  - accumulates latency
  - does not make forward progress
- more threads == more wait == more latency

Thread Scheduling



Thread
Scheduler



Runnable Queue

Thread scheduler and cores are racing
- core are emptying the task queues
- scheduler is filling task queue
- empty task queue == idle core

Java Heap

- Data structure designed to retain Java objects and arrays
  - reserved as a single contagious block of memory with size -Xmx
    - default size is 1/4 available RAM
  - committed memory size is -Xms
    - default size is 1/16th available RAM
  - ergonomics can resize Java Heap as needed at the end of each GC cycle

- Application thread(s)
  - allocate Java objects in Java heap
  - mutate pointers that connect Java objects to each other
  - mutate data values in arrays or primitive fields
  - access data that a thread can reach

- Garbage Collection thread(s)
  - reclaim memory that application threads have allocated but can no longer reach
  - employ a precise Mark/Sweep algorithm

Java Heap

- A set of shared data structures
  - must be thread safe to ensure no threads see data corruption
- Serial and Parallel collectors require mutually exclusive access
  - application threads are paused while GC is underway
- Concurrent collectors and application threads use GC barrier code to cooperate
  - application threads maybe paused for a very short period of time
- GC overhead effect on application threads is quite different between the two schemes
  - full pause vs constant drag

Safepointing-JVM

- A safepoint is point in the code where execution can be suspended, maintenance can be performed, and the thread can be resumed without being corrupted

- Safepoint “captures” all application threads
  - called for on two conditions
    - immediate need for JVM maintenance
      - GC is an example
    - maintenance that can be delayed to put into a work queue
      - once a second the maintenance work queue is checked
        - if queue has work, call for a safepoint

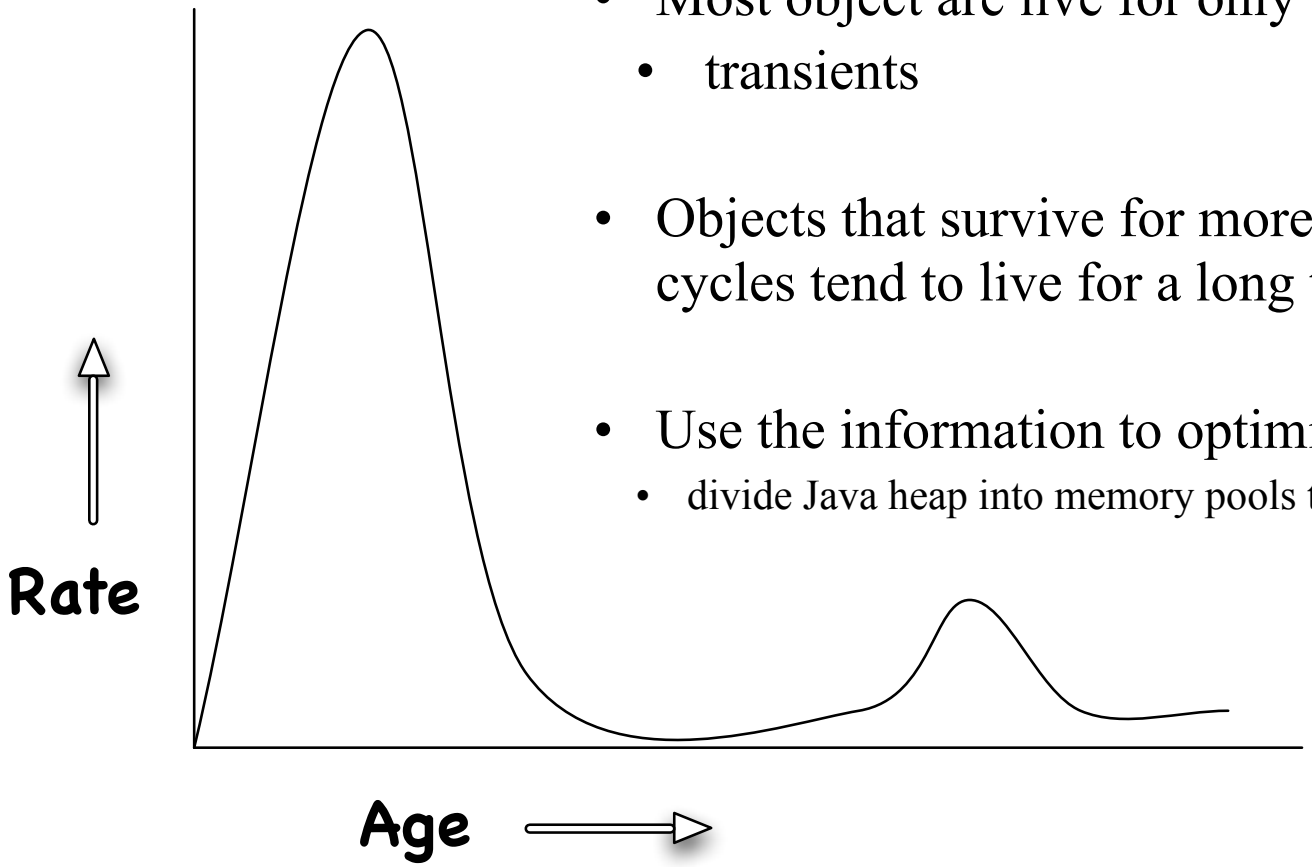
Safepointing-Application

- All application threads must cooperate
  - maintenance doesn't start until all application threads are at the safepoint
  - threads slow to reach a safepoint will delay maintenance and inflate pause times

- Calls to check for safepoint is embedded in application code
  - method entry
  - end of non-counted loops

```
// counted loop
int total = 1;
int n = Integer.MAX_INTEGER;
for (int i = 0; i < n; i++)
   total *= i;
```

Weak Generational Hypothesis

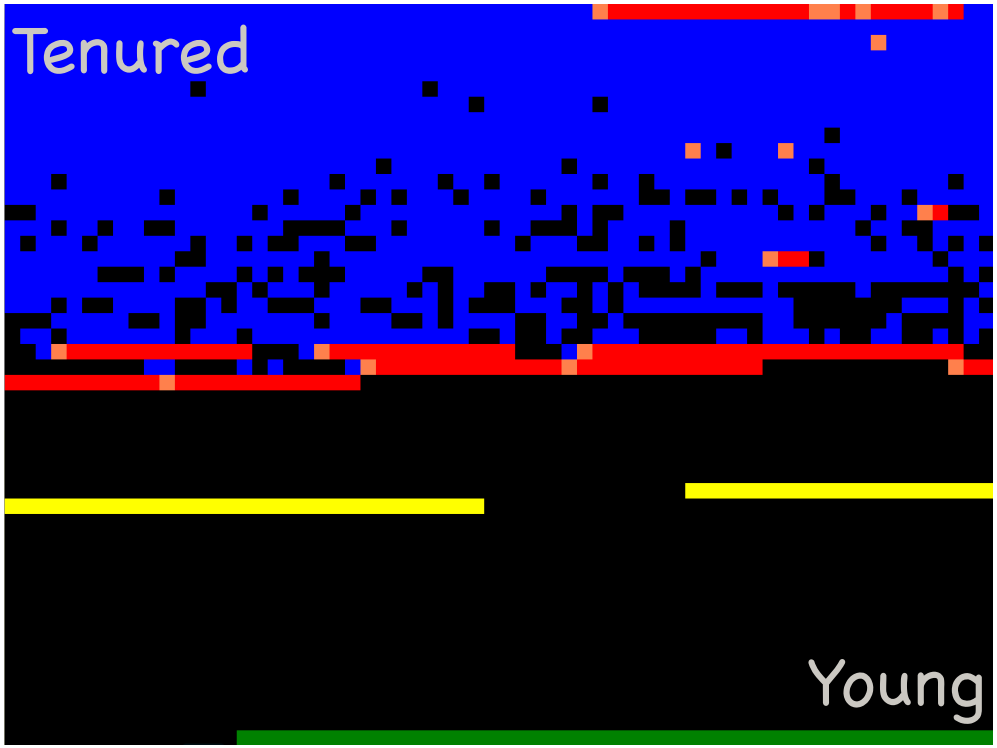


- Most object are live for only a few machine cycles
  - transients
- Objects that survive for more than a few machine cycles tend to live for a long time
- Use the information to optimize Java heap
  - divide Java heap into memory pools that segregate objects by age

GC Memory Pools

- Serial
  - Young
  - Full
- Parallel
  - Young
  - Full
- CMS
  - Tenured only
  - Failure mode: Serial Full

G1 Regions



- Young
  - Young
  - Young-Initial Mark
  - Young-Mixed
- Tenured
  - Mark only
  - Failure mode: Serial Full

Tenured Region

Humungous Region (start)

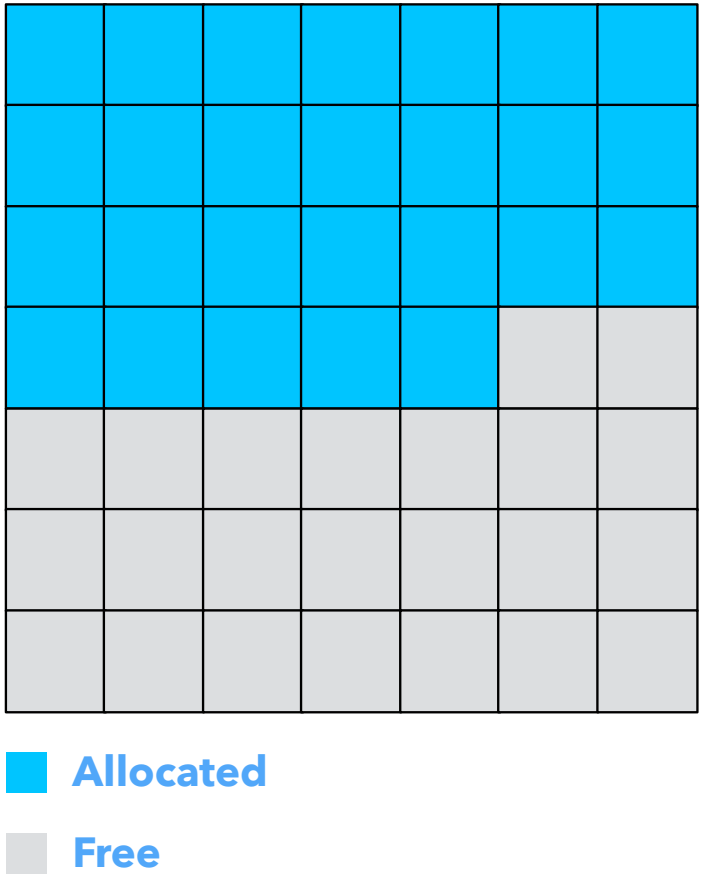
Humungous Region (cont)

Eden Region

Survivor Region
aka: to space

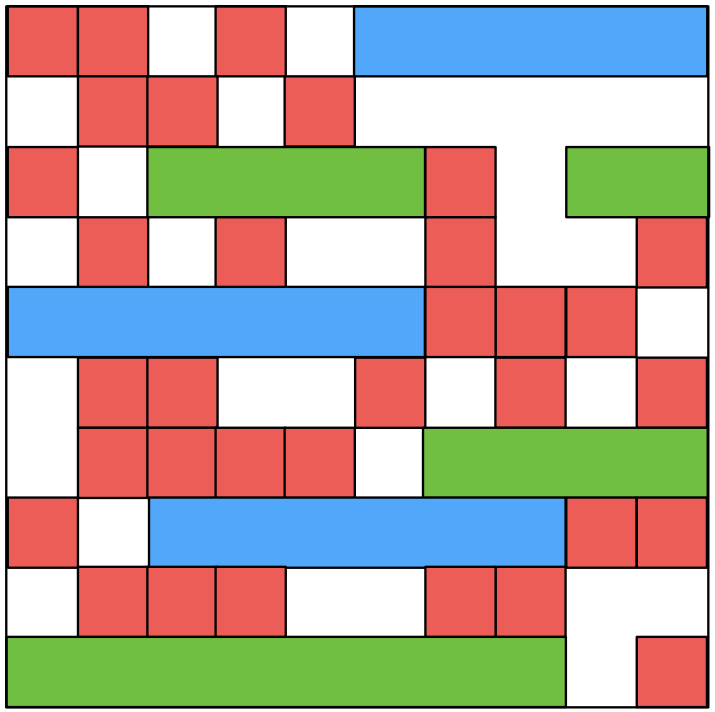
Unallocated Region

Shenandoah Regions



- Concurrent collection
  - 4 very short pauses between phases
  - Failure mode: Parallel Full
- Young
  - recently allocated regions
- Tenured
  - all other regions

ZGC Pages



Small (2mb)

Medium (32mb)

Large (N*2mb)

- Concurrent collection
  - 4 very short pauses between phases
  - Failure mode: page allocation throttling
- Young
  - recently allocated regions
- Tenured
  - all other regions

Reasons to Start a GC Cycle

- Allocation failure
  - heap is full
  - heap is fragmented

- Duty cycle occupancy threshold is reached
  - generally triggers initial mark of tenured space
  - threshold maybe calculated at the end of each GC cycle
  - failure to start soon enough may result in
    - full GC
    - allocation stalls (pacing)

- Speculative or Diagnostic
  - System.gc() or Runtime.gc()

Garbage Collection

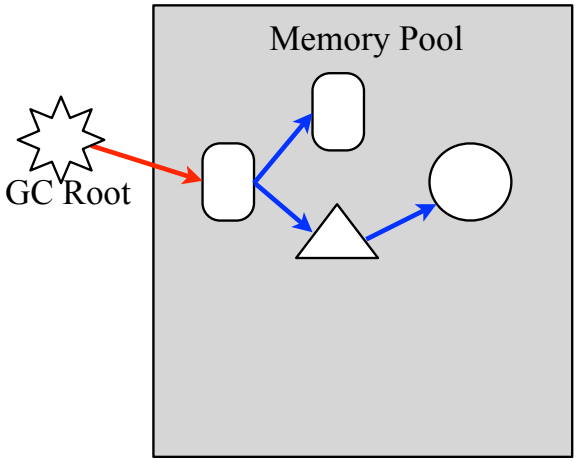
- Multi-step process to recover memory for reuse

- Steps
  - **Scan for roots**
    - find all pointers that point into the memory pool being collected
  - **Mark**
    - trace all references to identify all reachable objects
  - **Sweep**
    - copy live data to another memory pool
      - heal pointers
    - in-place
      - free list management
      - compact to reduce fragmentation
        - generally considered to be a failure mode

Scan for Roots

- Find all pointers that point into the memory pool to be collected

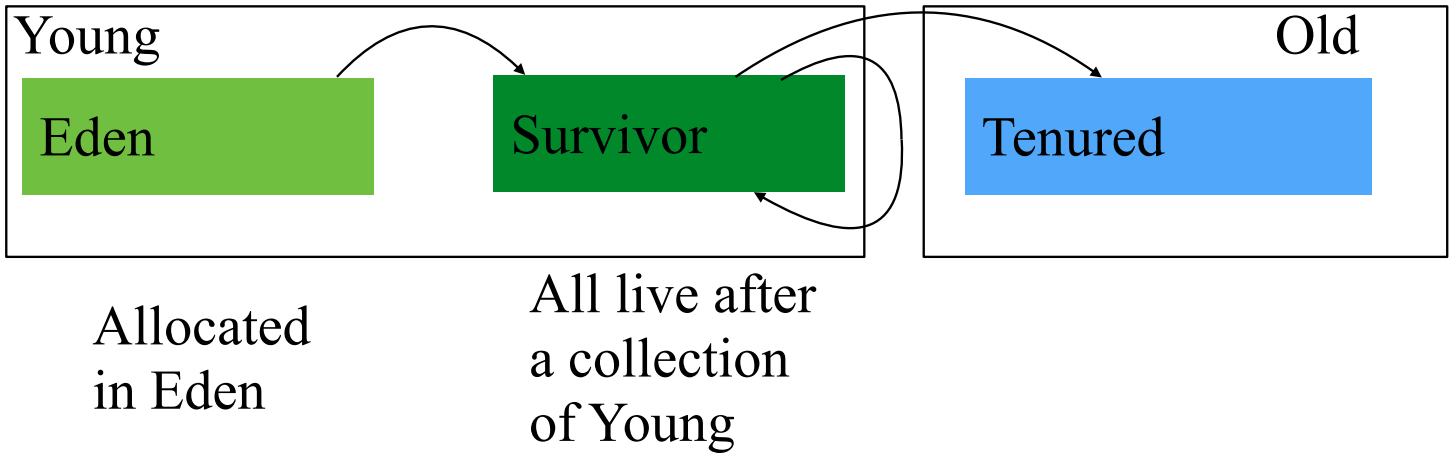
- Sources for GC roots include
  - stack frames
  - CPU registers
  - JNI references
  - class loaders
  - global variables
  - Metaspace
  - Other memory pools



GC Phases

- Mark
  - Serial/Parallel/ParNew/G1 Young
    - mark is run in a full STW pause
  - ZGC/Shenandoah/CMS/G1 Old
    - mark is run concurrently after an Initial Mark
- Sweep
  - Serial/Parallel/ParNew/G1 Young/G1 Mixed
    - sweep us run in a full STW pause
  - ZGC/Shenandoah
    - sweep is run concurrently after an initial STW pause
  - G1 Old does not sweep
    - sweep is deferred to G1 Mixed
  - CMS does not copy
    - compaction is part of a failure mode

Object Lifecycles



- Allocated in Eden
  - heap is fragmented

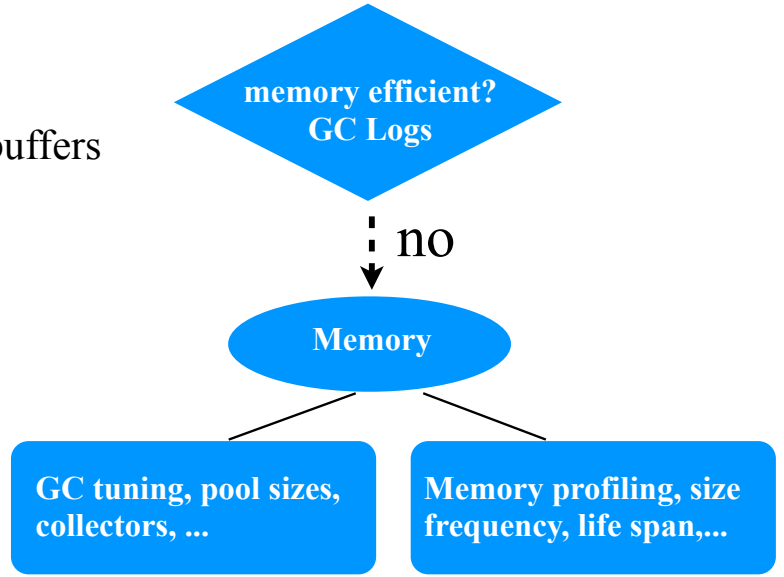
GC Tuning Basics

- The purpose of tenured is to retain long lived data
  - promotion of transients results
    - high rates of fragmentation
    - more frequent collections of this space
  - Recommendation: Old should be 2x the size of the live set

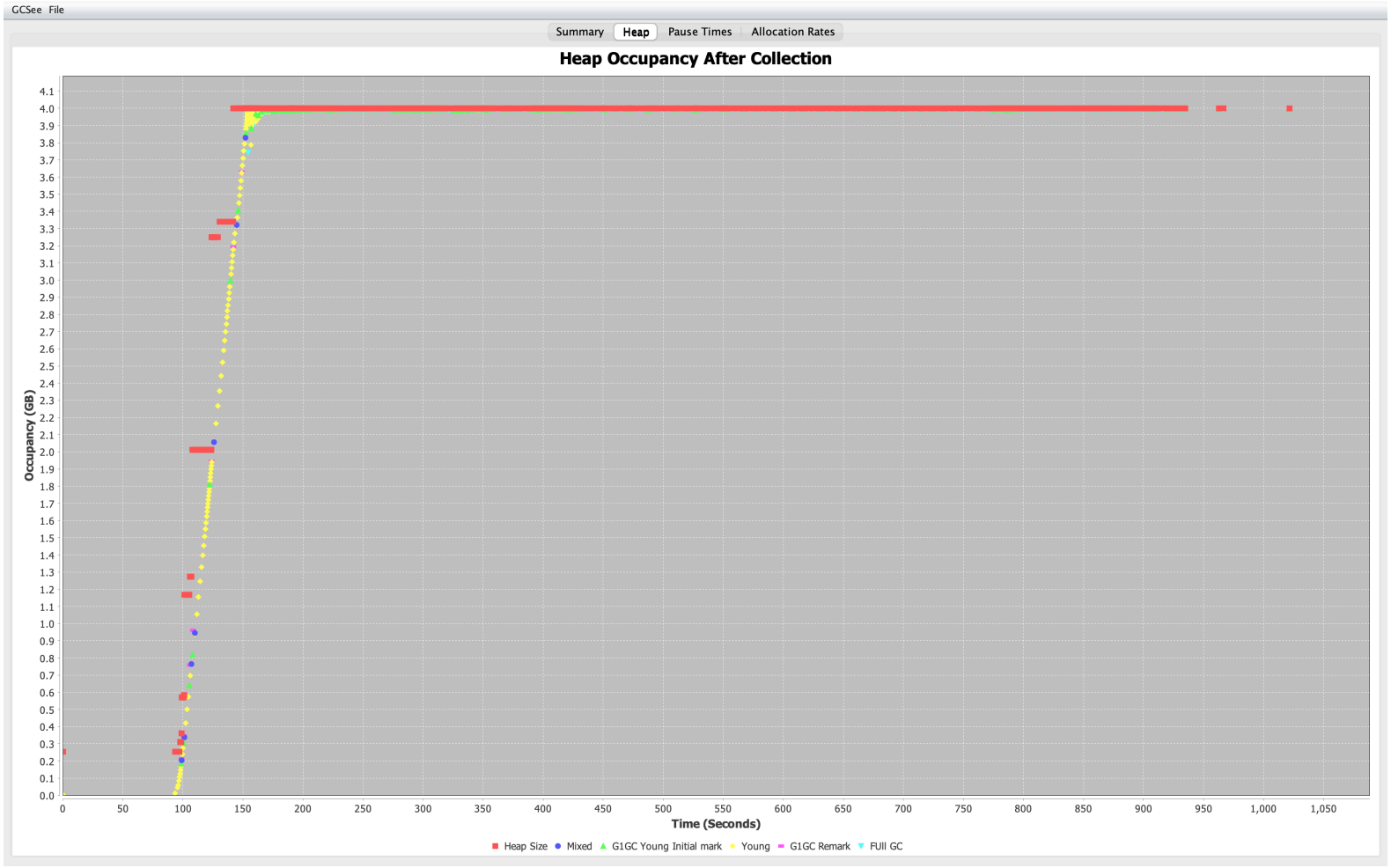
- The purpose of young is to provide working space
  - high allocation rates result in frequent collection of this space
    - may require a larger space to cope
    - may cause promotion of transients to tenured
  - Recommendation: Eden should be large enough to support the allocation rate
    - Survivor should be large enough to prevent promotion of transients

GC Tuning Basics

- GC Overhead: Should be less than 2%
  - Pause time / run time
- Pauses Times: tied to GC Overhead but should be evenly distributed and consistent
- Allocation Rates: Lower is better
  - less than 1GB/sec for normal applications
  - less than 2GB/sec for application that make use of JSON (Rest) or large buffers



Tooling

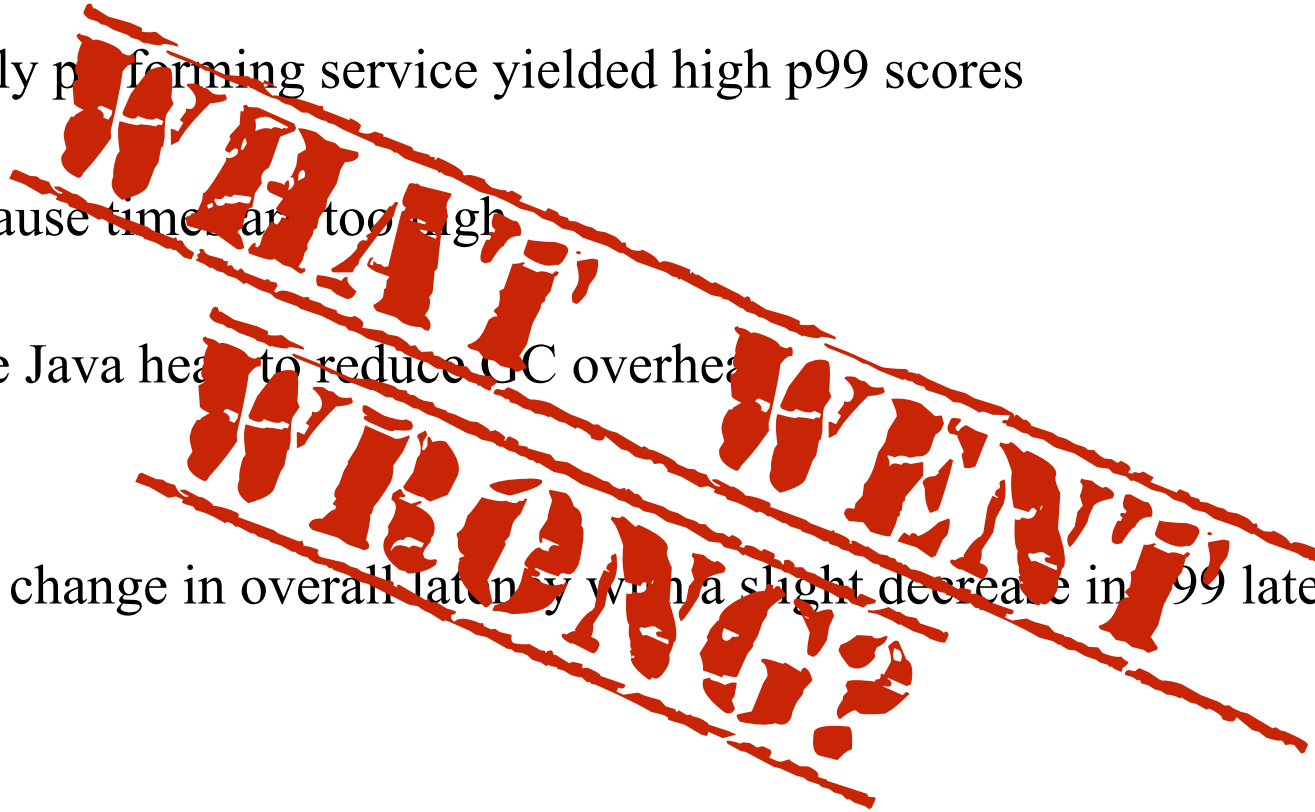


Case Study

- Symptom: poorly performing service yielded high p99 scores
- Analysis: GC pause times are too high
- Action: increase Java heap to reduce GC overhead
- Result :
  - little to no change in overall latency with a slight decrease in p99 latencies

Case Study

- Symptom: poorly performing service yielded high p99 scores
- Analysis: GC pause times are too high
- Action: increase Java heap to reduce GC overhead
- Result :
  - little to no change in overall latency with a slight decrease in p99 latencies



Case Study

- Symptom: poorly performing service yielded high p99 scores

- Analysis: ~~GC pause times are too high~~

  jPDM - Memory dominate, allocations at GB/sec

- Action: increase Java heap to reduce GC overhead

- Result :
  - little to no change in overall latency with a slight decrease in p99 latencies

Case Study

- Symptom: poorly performing service yielded high p99 scores
- Analysis: High allocation rates, memory profile finds Optional to be the source
- Action: Refactor code to eliminate allocation of Optional
- Result :
  - significant drop in p50 and p99 latencies
  - reduce the overall size of heap (less working space needed)
  - delay server upgrades for 1 year (cost savings of $1.7million)

Performance Tuning With Cheap Drink and Poor Tools

Profiling

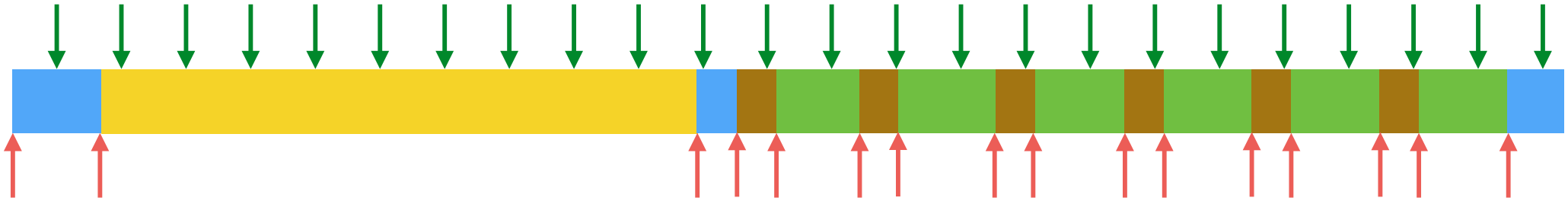
- Would like to develop an overview of which portion of the code base is consuming the time budget
  - use sampling techniques to develop a picture of the application and environment
- Sytem dominant
  - OS/CPU/Hardware counters
- Nothing dominant
  - thread dump
- System dominant
  - CPU profiling
- Memory dominant
  - memory profiling

Profiler Basics

- Profiler should collect
  - a sample from the resource under investigation
  - a stack trace for the code utilizing the resource
- How and when you collect will come with it's own bias
  - create blind spots
  - distorts reality
  - imposes overheads
- All data treatment techniques will have blind spots
  - how data is aggregated may hide effects
- All visualizations will have blind spots
  - how data is drawn in charts will causes visual distortions

Safepoint Bias

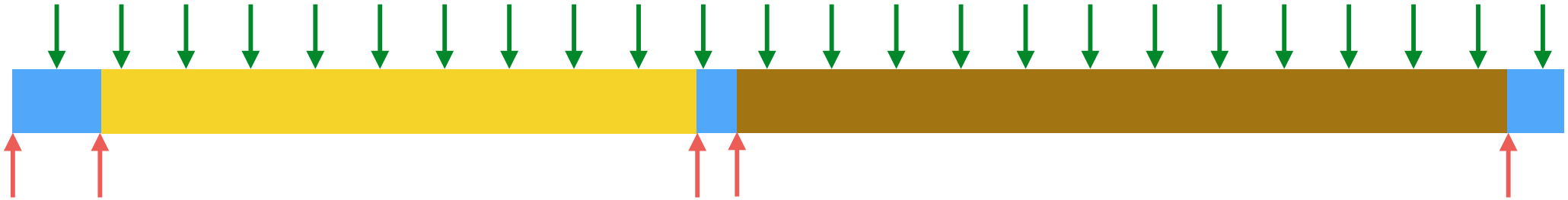
- Some profilers only sample at safepoints
  - creates holes in the profile that maybe important



	main	foo	bar	foe
sample	3	9	6	6
safepoint	3	1	6	6

JIT Inlining

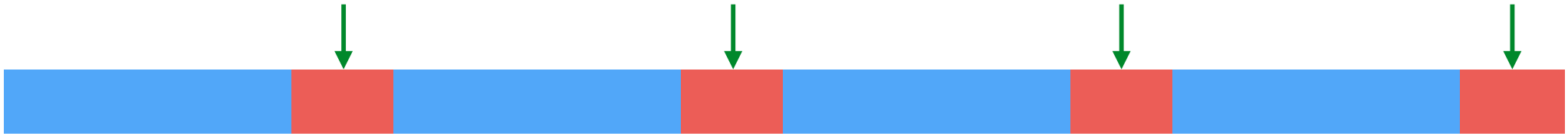
- Just-in-time (JIT) compiler can optimize the code
  - inline frequently called methods to reduce virtual method call overheads
    - pathway to other optimizations
- Can lead to confusing results
  - code being profiled doesn't look like the code that is running



	main	foo	bar	foe
sample	3	9	12	0
safepoint	3	1	1	0

Periodicity bias

- Sampler can become lock-sync'ed with application
  - inline frequently called methods to reduce virtual method call overheads
    - pathway to other optimizations
- Can lead to confusing results
  - code being profiled doesn't look like the code that is running



The world is red

Some Terminology

- Profiling techniques
  - Sampling: collects information about a program by periodically sampling it's state
  - Instrumented: code is instrumented to emit a signal when the the code is executed

- Different types of profiling
  - CPU: Measure of the amount of CPU time spent executing code
  - Wall clock: Measure of the time elapsed to execute code
  - Method duration: focus no the execution time of individual methods
  - Allocation: an analysis of the memory allocations behavior of a program

- Other terms
  - Self time: The amount of time spent executing a method
  - Total time: The amount of time spent executing a method and time in it's callouts

How Much Data is Needed?

- Execution/Allocation profiling
  - millions of data points
- Thread Dump
  - single data point
- Heap Dump
  - single data point
- GC Log
  - audit trail
- A well randomized sampling profiler should be able to detect issues with very few samples
  - timing when to profile is critical

Demo: VisualVM

Demo: JFR

Demo: Async Profiler

Ex 1.2 - Profile to find bottleneck

1. cd into the exercise1 directory
    1. run the compile script
    2. if the data files do not exist, run the generate script
    3. run the appStart script to run the application
    4. attach the profiler of choice to running application
        1. hit enter to run the application once the profiler had attached

2. Use the profile to characterize the bottleneck
    1. use this information to refactor the code

3. Repeat a full tuning cycle on this test app

Java Management eXtensions

- JMX is a standard API for management and monitoring of resources
- MBean consists of
  - attributes
  - operations
  - notifications
- MBeans are registered with a the Platform MBeanServer
  - registered with a unique ObjectName
  - creates a centralized location for clients to discover and interact with MBeans
- Standard set of MBeans defined in java.lang.management
  - wrappers over performance counters in the JVM
- Easily extensible
  - used by many products, frameworks, and components

Demo: JMX Clients

Exercise 2: Tuning Tips

Questions

@kcpeppe
kirk@kodewerk.com