CS1632: PERFORMANCE TESTING

Wonsun Ahn

What do we mean by Performance?

- If you look it up in a dictionary …
  - *Merriam-Webster*: the ability to perform
    - Dictionaries can be self-referential like this ☹
  - *Cambridge*: how **well** a person or machine does a piece of work
  - *Macmillan*: the **speed** and **effectiveness** of a machine or vehicle

- In software QA: it is a **non-functional** requirement (**quality attribute**)
  - Narrow sense: **speed** of a program
  - Broad sense: **effectiveness** of a program
  - In this chapter, we will refer to performance in the broad sense

Performance Indicators

- Quantitative measures of the performance of a system under test
- Examples (in the narrow sense, speed):
  - How long does it take to respond to a button press? (*response time*)
  - How many users can the system handle at one time? (*throughput*)
- Examples (in the broad sense, effectiveness)
  - How long can the system go without a failure? (*availability*)
  - How much memory is used in megabytes? (*memory efficiency*)
  - How much energy is used per second in watts? (*energy efficiency*)

Key Performance Indicators (KPIs)

- **KPI**: a performance indicator important for the performance goal
- Select only a few KPIs that can measure success
  - Being indiscriminate means important performance goals will suffer
  - e.g. miles-per-gallon (mpg) *should* be a KPI for a hybrid-electric car
  - e.g. maybe mpg *should not* be a KPI for a formula-1 race car

- **Performance target**: a number that KPI should reach ideally
- **Performance threshold**: bare minimum a KPI should reach
  - Bare minimum to be considered production-ready
  - Typically more lax compared to performance target

But Performance is Hard to Quantify

- Even performance in the narrow sense (speed) is hard to quantify
- Speed for a *web browser*
  - How quickly a web page responds to user interactions (clicking, typing, …)?
  - Speed is measured in terms of **response time.**
- Speed for a *web server*
  - How many web pages can the server process per second?
  - Speed is measured in terms of **throughput**.
  - Note: Throughput != 1 / response time, due to parallel processing.
    (Page load time may be 1 second, but throughput may be 1000 pages / sec)
- We need more than one metric to quantify performance

Example: KPIs for a Web Application

- KPI: Response time
  - Performance target: 100 milliseconds
  - Performance threshold: 500 milliseconds
- KPI: Availability
  - Performance target: More than 99.999% of HTTP requests serviced
  - Performance threshold: More than 99.9% of HTTP requests serviced
- KPI: Throughput
  - Performance target: 1000 requests / second
  - Performance threshold: 500 requests / second
- KPI: Memory Utilization
  - Performance target: 10 MBs of memory per request
  - Performance threshold: 100 MBs of memory per request

Performance Indicators: Categories

- There are largely two categories of performance indicators

- Service-Oriented

- Efficiency-Oriented

Service-Oriented Performance Indicators

- Measures how well a system is providing a service to the users
  - Measures how users experience your system, the *QoS (Quality of Service)*
  - Often codified in *SLA (Service Level Agreement)* between user and provider

- Two subcategories:
  - **Response Time**
    - How quickly system responds to a user request.
    - E.g. How long does a web page take to load? 10 ms? 100 ms?
  - **Availability** (a.k.a. uptime)
    - Percentage of time users can access the services of the system.
    - E.g. How many days in a year is the website up and running? 99%? 99.9%? 99.99%?

Efficiency-Oriented Performance Indicators

- Measures how efficiently a system makes use of system resources:
  - CPU time, memory space, battery life, network bandwidth, …
  - Is not directly observed by user but impacts QoS if resource is limited

- Two subcategories:
  - **Utilization**
    - Given a *workload*, amount of *resources* system uses.
    - E.g. How many CPU clock ticks are needed to service a web page?
  - **Throughput**
    - Given certain *resources*, amount of *workload* system can handle per time unit.
    - E.g. How many web pages can a web server service per second?

Efficiency-Oriented Indicators Impact QoS

- CPU *utilization*
  - Number of CPU clock ticks needed to handle request
    (e.g. 1 second of CPU time translates to 1 billion clock ticks on a 1 GHz CPU)

  → Translates to *response time*, given a certain number of CPUs with a certain clock rate

- Memory *utilization*
  - Bytes of memory needed to handle request

  → Translates to *availability*, given a certain amount of memory and a flood of requests

- Server *throughput*
  - Maximum number of requests handled per second

  → Translates to both *response time* and *availability*, given a certain number of servers

- Efficiency-oriented indicators are crucial in analyzing QoS problems!

Testing Service-Oriented Performance Indicators

Response Time / Availability

Rough Response Time Performance Targets

< 0.1 S : Response time required to feel that system is instantaneous

< 1 S : Response time required for flow of thought not to be interrupted

< 10 S : Response time required for user to stay focused on the application

- Taken from "Usability Engineering" by Jakob Nielsen, 1993

Things haven't changed much since then!

Testing Response Time

- Easy to do!
    1. Submit a request to the system, and click “start” on stopwatch
    2. When response comes back, click “stop” on stopwatch

- Any problems with this approach?

Problem with Manual Testing Response Times

1. Limited accuracy: cannot reliably tell sub-second differences
2. Labor intensive: time-consuming to measure all usage scenarios
3. Black box: can only measure end-to-end response times
   - Cannot measure component response times such as response times of:
     - Database queries
     - Calls to microservice endpoints
     - File read/write requests

☛ Performance testing relies heavily on automation

Response Time Testing Relies on Automated Tools

- time command in Unix
  - time java Foo
  - time curl http://www.example.com
  - time ls
- Windows PowerShell has:
  - Measure-Command { ls }

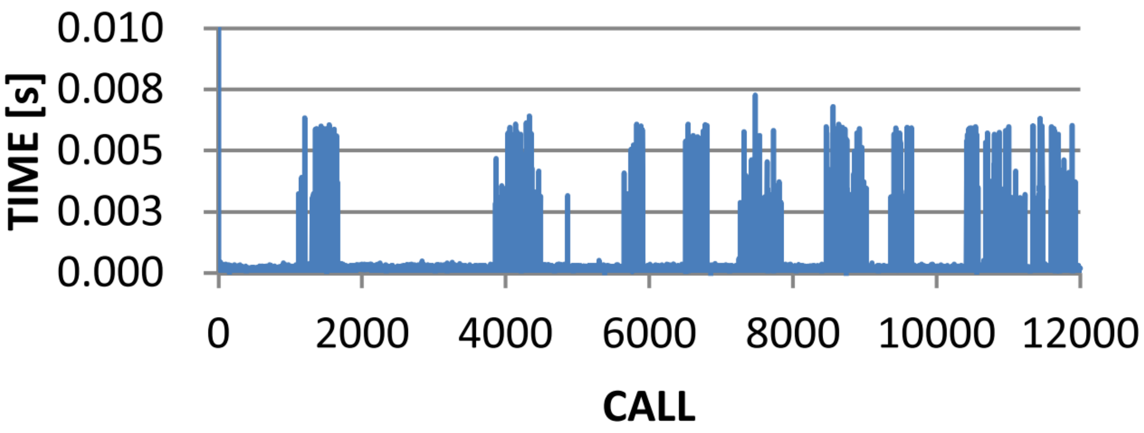
```
-bash$ time curl http://www.example.com
<!doctype html>
<html>
…
</html>

real    0m0.021s
user    0m0.002s
sys     0m0.004s
```

This is the response time

We will discuss these later

Response Time Testing Needs Statistical Reasoning

- Time taken by the same method call when measured 12000 times:



K. Kumahata et al. “A Case Study of the Running Time Fluctuation of Application”, *International Symposium on Computing and Networking*, 2016
See: resources/running_time_case_study.pdf in course repository to read entire paper

- System response times always form a distribution:
  - Other processes can run while testing, taking up CPU time
  - Other processes can take up memory while testing
  - Network can experience traffic while testing from unrelated sources

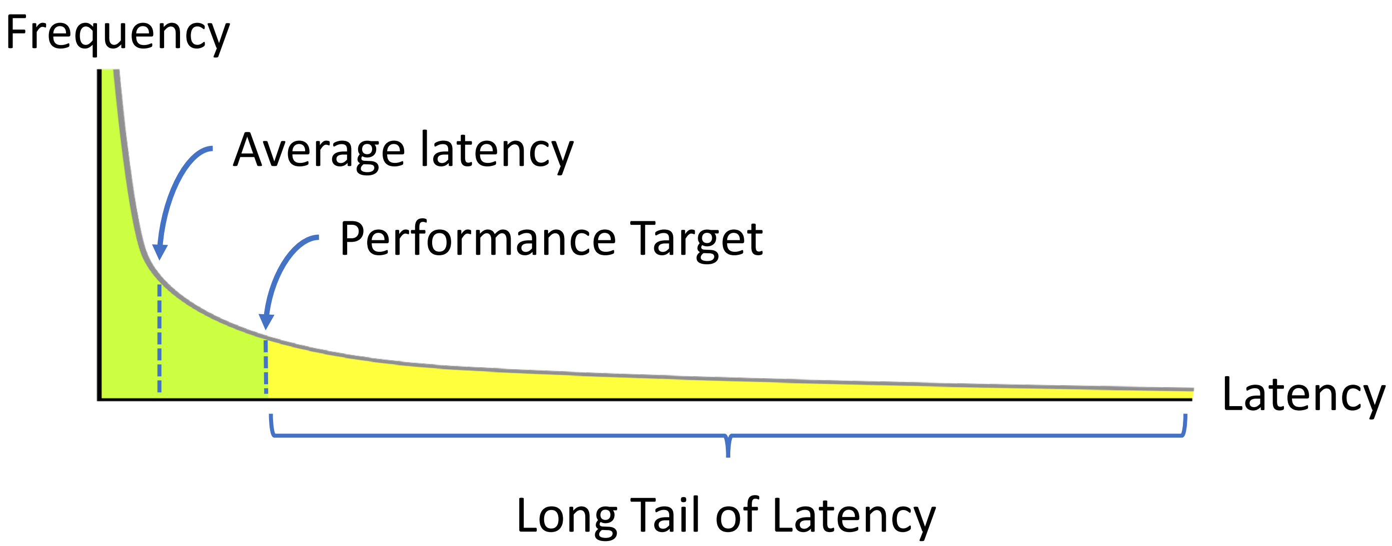
Minimizing and Dealing with Variability

- Eliminate all variables OTHER THAN THE CODE UNDER TEST
  - Make sure you are running with same software/hardware configuration
  - Kill all processes in the machine other than the one you are testing
  - Remove all periodic scheduled jobs (e.g. anti-virus that runs every 2 hours)
  - Fill memory / caches by doing several warm up runs of app before measuring

- Even after doing all of this, there is still going to be variability
  - Try multiple times to get a statistically significant average
  - Also look at min/max values to check for large variances

The Dreaded Long Tail of Latency

- Typically, this is the type of latency distribution you will get

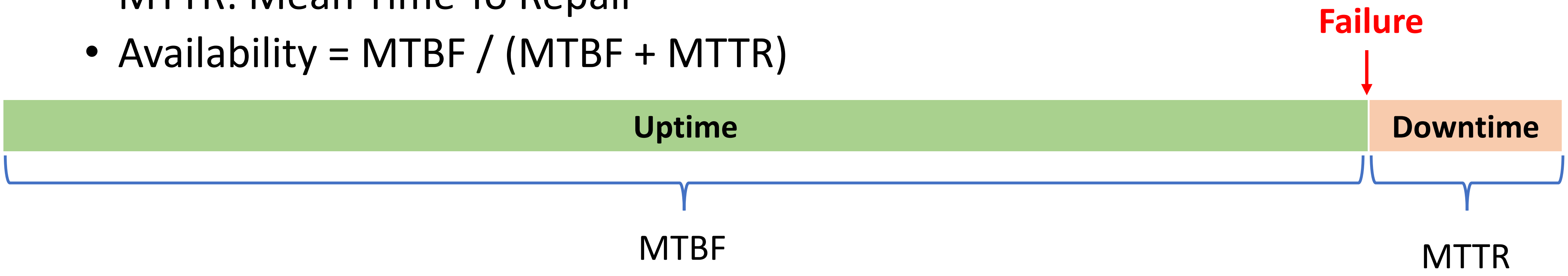


- Often the “long tail” is more important than average latency
  - These are the response times that fail the performance target
- Many runs are required not only to accurately measure the average, but also to detect the length and height of the “long tail”

Testing Availability

- Difficult – not feasible to run a few “test years” before deploying
- Modeling usage and estimating uptime is the only feasible approach

- Metrics to model
  - MTBF: Mean Time Between Failures
  - MTTR: Mean Time To Repair
  - Availability = MTBF / (MTBF + MTTR)



Measuring MTTR and MTBF

- Measuring MTTR is easy
  - Average time to reboot a machine
  - Average time to replace a hard disk

- Measuring MTBF is hard
  - Depends on how much the system is stressed
  - Depends on the usage scenario
  - Measure MTBF for different usage scenarios
    → Calculate a (weighted) average of MTBF for those scenarios

Measuring MTBF with Load Testing

- Load testing:
  - Given a load, how long can a system run without failing?
  - Load is expressed in terms of concurrent requests / users

- Kinds of load testing:
  - Soak / Stability Test – Typical usage for extended periods of time
  - Stress Test – High levels of activity typically in short bursts

- Estimate MTBF based on test results and historical load data
  - E.g. if 90% of time is typical usage, 10% of time is peak usage,
    MTBF = Soak Test MTBF * 0.9 + Stress Test MTBF * 0.1

Testing Efficiency-Oriented Performance Indicators

Throughput / Resource Utilization

Testing Throughput

- *Throughput*
  - Number of events a system can handle in a given timeframe

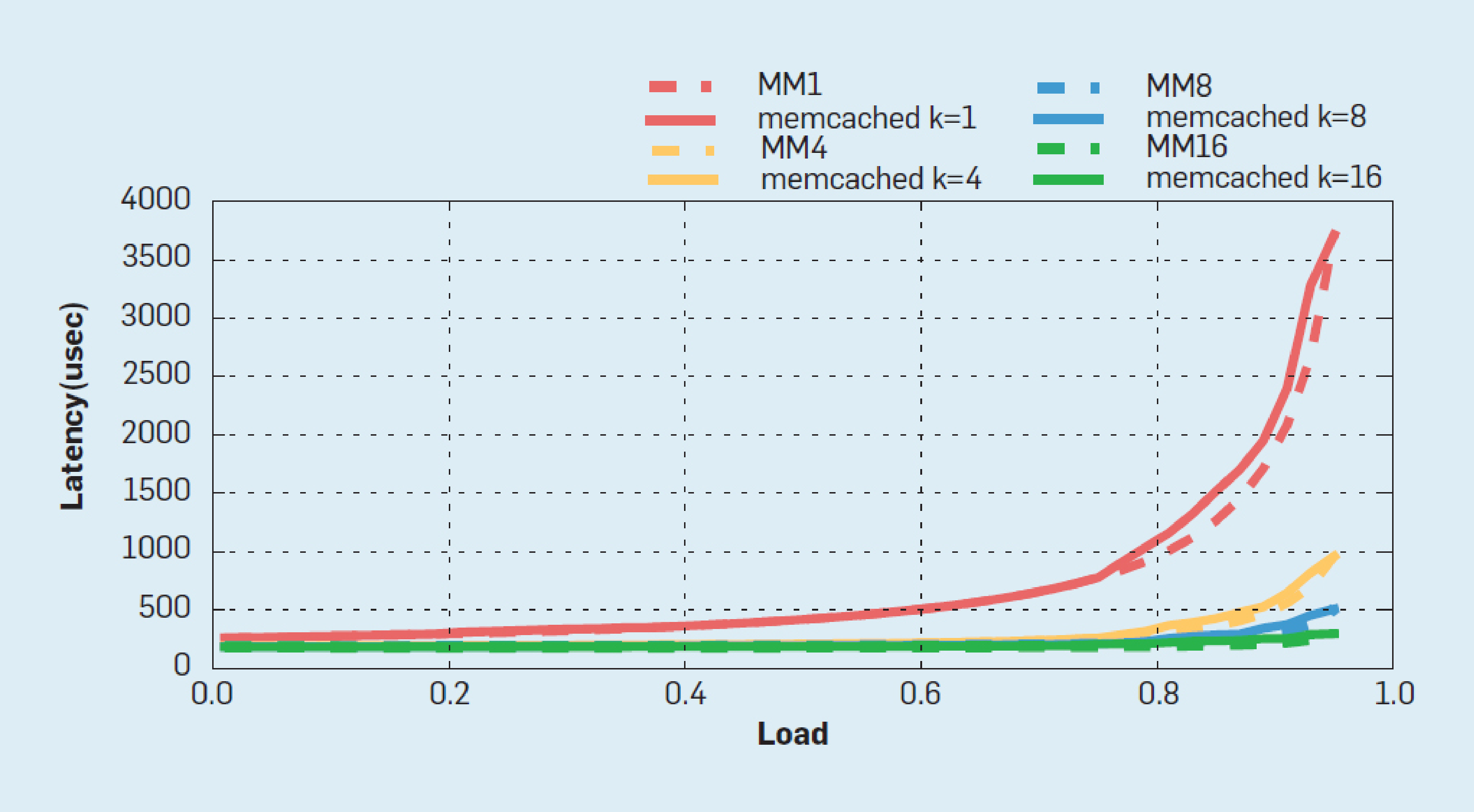
- Examples:
  - Packets per second (that can be handled by a router)
  - Pages per minute (that can be served by a web server)
  - Number of concurrent users (that a game server can handle)

Measuring Throughput: Load testing

- Load testing can also be used to test throughput (as well as availability)

- Measure maximal load system can handle without degrading QoS
  - Increment events / second until response time falls below performance target
  - Resulting events / second is the throughput of the system

Measuring Throughput: Load testing



- From “Amdahl's Law for Tail Latency” C. Delimitrou et al. *CACM*, 2018 https://cacm.acm.org/research/amdahls-law-for-tail-latency/

Testing Utilization

- *Utilization*
  - How much compute resources does the software use?

- Examples:
  - How many *CPU clock ticks* is used to service a request?
  - How much *memory* is used to service a request?
  - How much *network bandwidth* is used to service a request?
  - How much *energy* is used to service a request?

Testing Utilization: Tools

- General purpose
  - Windows Systems – Task Manager, perfmon
  - OS X - Activity Monitor, Instruments, top
  - Unix systems - top, iostat, sar, perf, time

- Program-Specific Tools

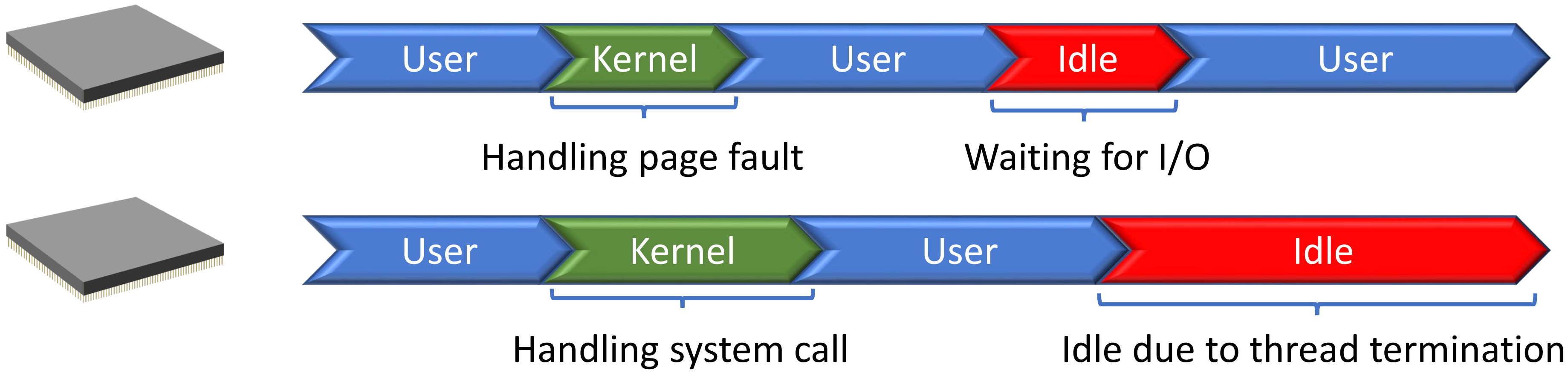
Measuring CPU Utilization

- real time: “Actual” amount of time taken (wall clock time)
- **user time**: Amount of time user code executes on CPU
- **system time**: Amount of time kernel (OS) code executes on CPU
- **total time**: user time + system time = **CPU utilization**

- real time ≠ total time
  - real time = total time + idle time
  - idle time: time app is idling (not executing on CPU) waiting for some event (where event can be an I/O event, synchronization event, interrupt event, …)

Sometimes, Total Time > Real Time

- Example breakdown of time for an application that runs on 2 CPUs



- Real time:
- User time: Sum of User
- Kernel time: Sum of Kernel
- Idle time: Sum of Idle

- Now we need to revise our previous equation:
  `Real time = Total time + Idle time`
- This works for only one CPU. For multiple CPUs:
  `Real time = (Total time + Idle time) / CPUs`

Time Measurement Using "time"

- time command in Unix
  - time java Foo
  - time curl http://www.example.com
  - time ls
- Windows PowerShell has:
  - Measure-Command { ls }

```
-bash$ time curl http://www.example.com
<!doctype html>
<html>
…
</html>

real    0m0.021s
user    0m0.002s
sys     0m0.004s
```

- `Real time = (User time + Kernel time + Idle time) / CPUs`
- `0.021s = (0.002s + 0.004s + Idle time) / 1 (single-threaded)`
- `Idle time = 0.015s` → Time mostly spent waiting for web server to respond

What does each Indicator Imply?

- Suppose real time does not satisfy response time target

- High proportion of **user time**?
  - Means a lot of time is spent running user (application) code
  - Need to optimize algorithm or use efficient data structure
- High proportion of **kernel time**?
  - Means a lot of time is spent in OS to handle system calls or interrupts
  - Need to reduce frequency of system calls or investigate source of interrupts
- Neither? i.e. High proportion of **idle time**?
  - Means a lot of time is spent waiting for I/O or synchronization
  - CPU utilization is not the problem. Look for efficiency issues somewhere else.
  - Need to reduce I/O bandwidth (by compressing data)?
  - Need to reduce synchronization so that all CPUs can be utilized at the same time?

CPU is not the Only Limited Resource

- CPU Usage
- Physical Memory
- Virtual Memory
- Disk I/O Bandwidth
- Network Bandwidth
- Threads

☛ Excessive utilization of any of these can result in low QoS

Other Utilization Performance Indicators

- Page faults – indicates high physical memory utilization
- Network packets discarded – indicates high network utilization
- Disk cache misses – indicates high hard disk utilization
- CPU cache misses – indicates high memory bandwidth utilization

General purpose tools only give general info

- Lots of CPU being taken up…
  - …but by what methods / functions?
- Lots of memory being taken up…
  - …but by what objects / classes / data?
- Lots of packets sent…
  - …but why? And what's in them?

Testing Utilization: Tools

- General purpose
  - Windows Systems – Task Manager, perfmon
  - OS X - Activity Monitor or Instruments, top
  - Unix systems - top, iostat, sar

- Program-Specific Tools

Program-Specific Tools

- Protocol analyzers
  - e.g., Wireshark or tcpdump
  - See exactly what packets are being sent/received
- Profilers
  - e.g. JProfiler, VisualVM, gprof, and many, many more
  - See exactly what methods are taking up most of the CPU time
  - See exactly what objects are taking up memory

To Wrap it Up …

“Premature optimization is the root of all evil” – Donald Knuth

- Do service-oriented testing first
  - If key performance indicators hit targets, why bother?
  - Only drill down with efficiency-oriented tests if otherwise

From Service-Oriented Test to Solution

- Assume: Rent-A-Cat has list-sorted-cats API listing available cats

1. Service-oriented testing
   - Response time: list-sorted-cats API misses performance target of 100 ms
2. Efficiency-oriented testing – General-purpose testing
   - Utilization testing (per request):
     Network bandwidth usage is 1%
     I/O bandwidth usage is 1%
     Memory usage is 2%
     CPU usage is pegged at 99%
   - Diagnosis: Problem must be inefficient CPU utilization

From Service-Oriented Test to Solution

3. Efficiency-oriented testing – Program-specific testing
   - VisualVM profiling says that the sortCats() method is taking most of the time

4. Solution
   - Cats sorted with insertion sort – Use better sorting algorithm

Track Performance throughout Versions

- Performance testing should be part of your regression test suite
- Just like for functional defects, you should be able to tell exactly when/where a performance defect is introduced
- Allows you to make an informed decision on whether that extra feature or enhancement is worth the performance hit

Now Please Read Textbook Chapter 19