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

But Even Speed is Hard to Define

- Even performance in the narrow sense (speed) is hard to define
- Speed for a *web browser*
  - How quickly a website responds to user interactions
    (Page loads, button clicks, dragging, typing …)
  - Responsiveness is measured in average **response time**
- Speed for a *web server*
  - How quickly a server responds to a page request is a part of it, yes.
  - More importantly, pages served per second (a.k.a. **throughput**)
  - As long as response time is less than a threshold (say, < 100 ms),
    web server performance is measured by throughput, not response time
- We need more than one metric to quantify performance

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)
  - How long can the system go without a failure? (*availability*)
  - How much memory does the program use in megabytes? (*utilization*)
  - How much energy does a program use per second in watts? (*utilization*)

Key Performance Indicators (KPIs)

- **KPI**: a performance indicator important to the user
- Select only a few KPIs that are really important
  - Those that are indicative of success or failure of your software
  - e.g. miles-per-gallon *should* be a KPI for a hybrid-electric car
  - e.g. miles-per-gallon *should not* be a KPI for a formula-1 race car
  - Being indiscriminate means important performance goals will suffer

- **Performance target**: quantitative measure 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

KPI / Performance Target / Performance Threshold

- Let’s say you are developing requirements for a web application
- Here is an example KPI / Performance Target / Performance Threshold
  - KPI: response time
  - Performance target: 100 milliseconds
  - Performance threshold: 500 milliseconds
- Another example KPI / Performance Target / Performance Threshold
  - KPI: throughput
  - Performance target: 100 user requests / second
  - Performance threshold: 10 user requests / second

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 a typical end-user experiences your system
  - Measures *Quality of Service (QoS)*
- Two subcategories:
  - Response Time
    - How quickly does the system respond to a user request?
    - E.g. How long does a web page takes to load?
  - Availability
    - What percentage of time can a user access the services of the system?
    - E.g. How many days in a year is the website up and running?

Efficiency-Oriented Performance Indicators

- Measures how well a system makes use of computational resources
  - Measures efficiency: computational efficiency, energy-efficiency, …
  - Given finite resources, will impact QoS
- Two subcategories:
  - Utilization
    - How much compute resources does the software use?
    - E.g. How many CPU clock ticks are needed to service a web page?
  - Throughput
    - How many requests can be processed in a given amount of time?
    - E.g. How many web pages can a web server service per second?

Service-Oriented vs. Efficiency-Oriented

- Service-oriented indicators measure QoS of the current system
  - Is the current *response time* satisfactory to the user?
  - Is the current *availability* satisfactory to the user?
- Efficiency-oriented indicators measure resources needed to achieve QoS
  - CPU *utilization* → CPUs needed to achieve target *response time*
  - Memory *utilization* → memory needed to guarantee *availability* without crashing
  - Server *throughput* → servers needed to consistently achieve target *response time* (given a certain request rate, according to queueing theory)
  - Server *throughput* → servers needed to guarantee *availability* during high demand (without having request rate overwhelm processing rate)
- Efficiency-oriented indicators point out problem areas in software
  - Given current set of resources, which aspect of software is bringing down QoS?

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!
  - Submit a request to the system
  - Click “start” on stopwatch
  - Wait for response from the system
  - Click “stop” on stopwatch
  - Write down number on stopwatch!

- Any problems with this approach?

Problem with Manual Testing Response Times

1. Impossible to measure sub-second response times
2. Possibility of human error
3. Time-consuming
4. Impossible to measure response times not visible to end-user
   (e.g. response time of a method call)

☛ 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 }

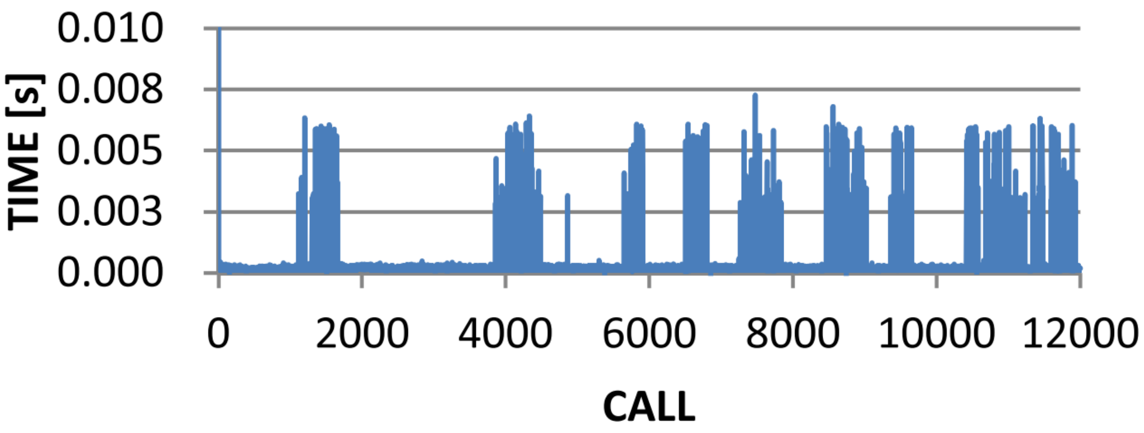
```
-bash$ time curl http://www.example.com
<!doctype html>
<html>
…
</html>

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
  - Data at time of execution can be at different levels of memory hierarchy: cache/memory/disk
  - Network can experience traffic while testing from external sources

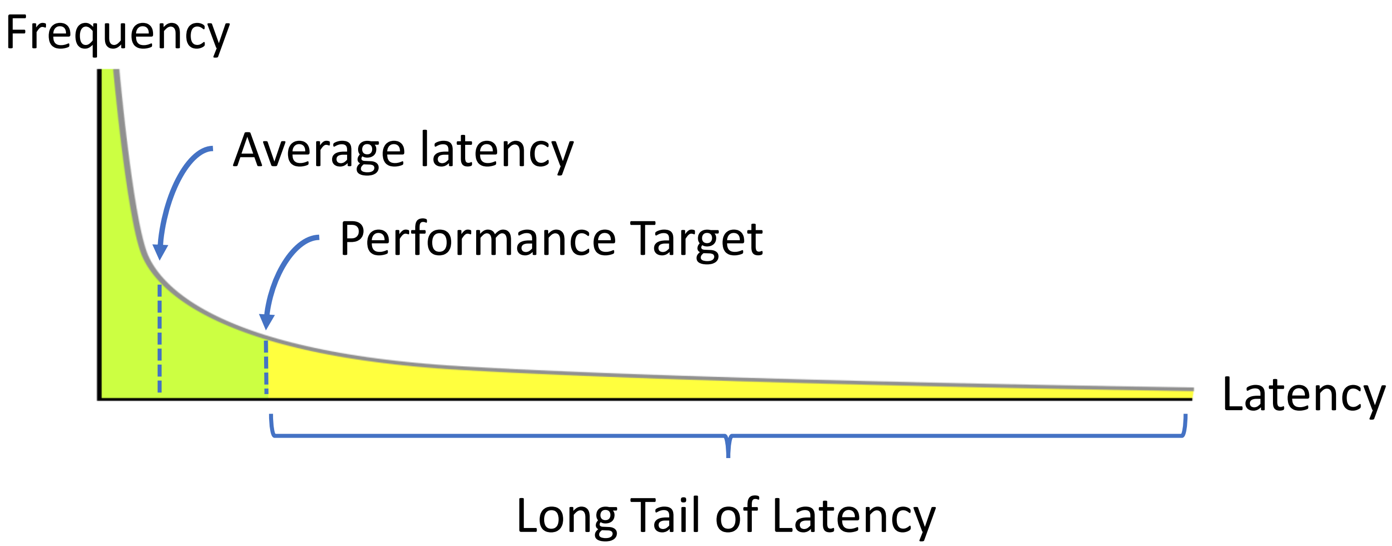
Minimizing and Dealing with Variability

- 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 Component Response Times

- Let's say we want to measure the response time of a website
- It is important to measure each component of the response time
  - Time for web browser to process user input and send request to web server
  - Time for request to travel through the network
  - Time for web server to process user request and send response
  - Time for response to travel back through the network
  - Time for web browser to render response data on to the web page
- Why? They indicate which component has the QoS problem.
  - Is it the web browser?
  - Is it the web server?
  - Is it the routers in the network?

Testing Availability

- Availability - often referred to as uptime
  - What percentage time is the system accessible to the user?

- Often codified in an *SLA (Service Level Agreement)*
  - “I am a web host. I guarantee you that you and your users will be able to access your service 99% of the time in a given month.”

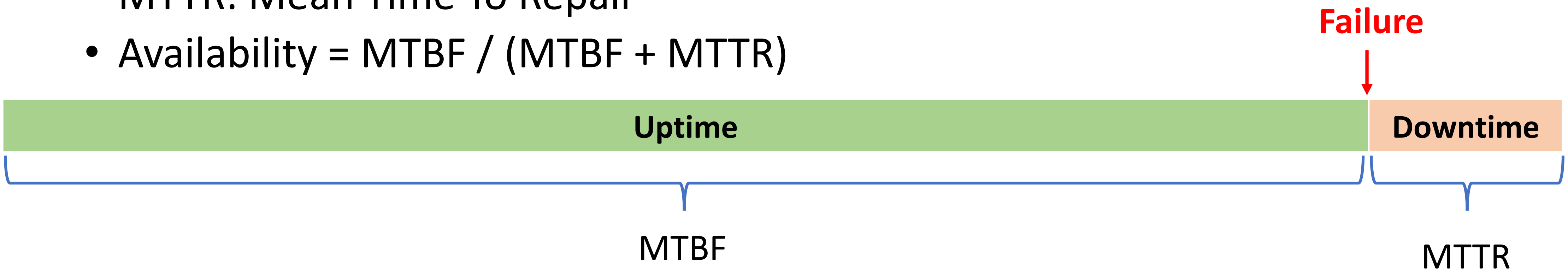
Nines

- Uptime is often expressed in an abbreviated form as 9’s (e.g. 3 nines, 5 nines etc)
- Refers to how many 9’s start out the percentage of time available
  - 1 nine: 90% available (36.5 days of downtime per year)
  - 2 nines: 99% available (3.65 days of downtime per year)
  - 3 nines: 99.9% available (8.76 hours of downtime per year)
  - 4 nines: 99.99% available (52.56 minutes of downtime per year)
  - 5 nines: 99.999% available (5.26 minutes of downtime per year)
  - 6 nines: 99.9999% available (31.5 seconds of downtime per year)
  - 9 nines: 99.9999999% available (31.5 ms of downtime per year)

How to test?

- Difficult – not feasible to run a few “test years” before deploying
- Modeling system 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
    $\rightarrow$ 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:
  - Baseline Test - A bare minimum amount of use, to provide a baseline
  - 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

MTBF is Not Only about Your Software

- For true availability numbers, also need to determine:
  - Likelihood of hardware failure
  - Likelihood of OS crashes
  - Likelihood of data center cooling system failures
  - Planned maintenance
  - etc.

Things can still go wrong

- Even with all this work, things go wrong

- Major service providers “breach” their SLAs once in a while
    - Including Microsoft Azure and Amazon Web Services
    - Usually, money is refunded automatically

Think about Contingency Plans!

- What if performance requirements aren't met?
- What if they can be, but at a high cost in time/resources?
- What if they can't be?
- etc.

Testing Efficiency-Oriented Performance Indicators

Throughput / Resource Utilization

Testing Throughput

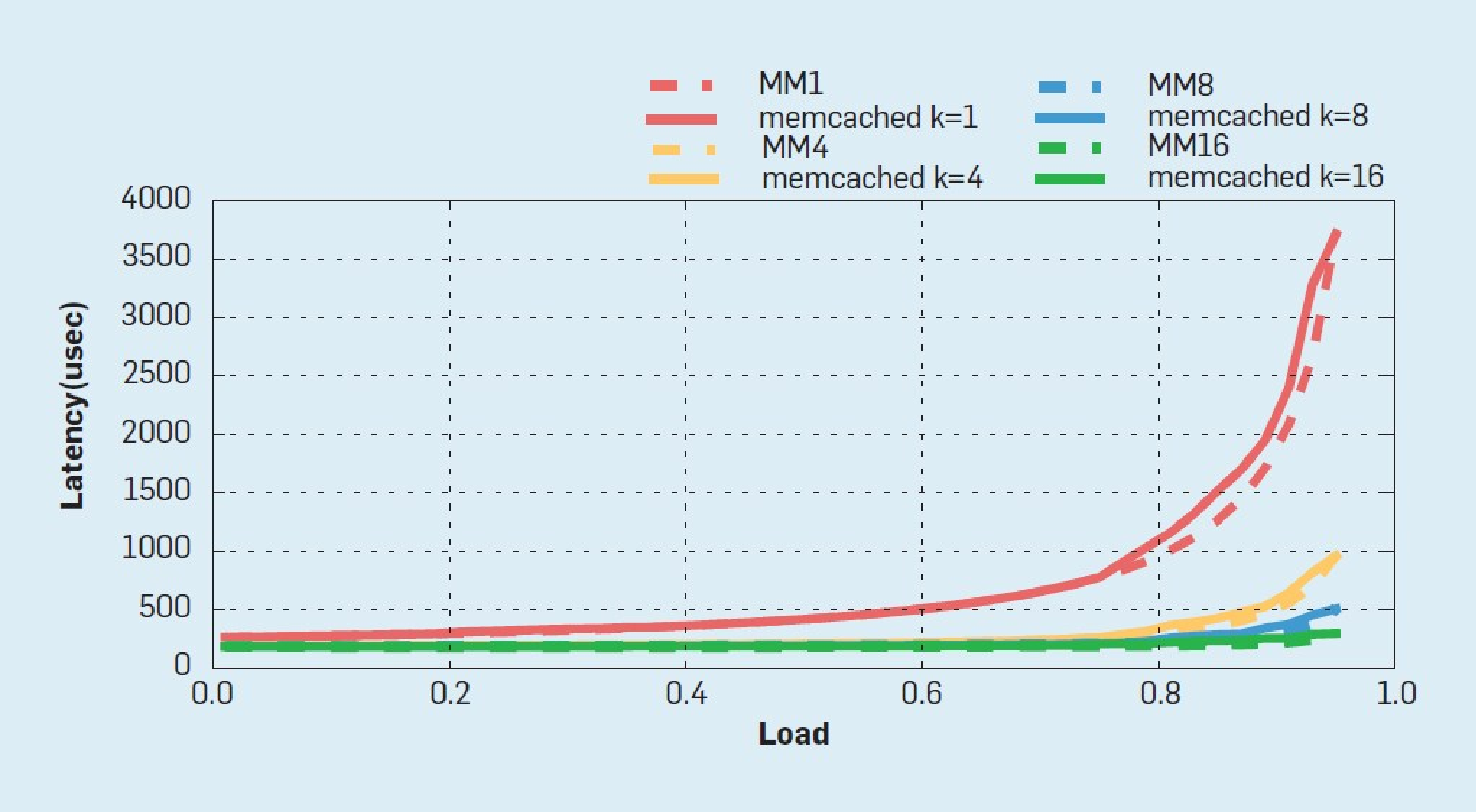
- *Throughput*
  - Number of events a system can handle in a given timeframe

- 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)
- Basically measure the maximal load that system can handle
  - Without degrading Quality of Service (QoS)
  - Increase 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/magazines/2018/8/229764-amdahls-law-for-tail-latency/fulltext

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 or Instruments, top
  - Unix systems - top, iostat, sar, time

- Program-Specific Tools

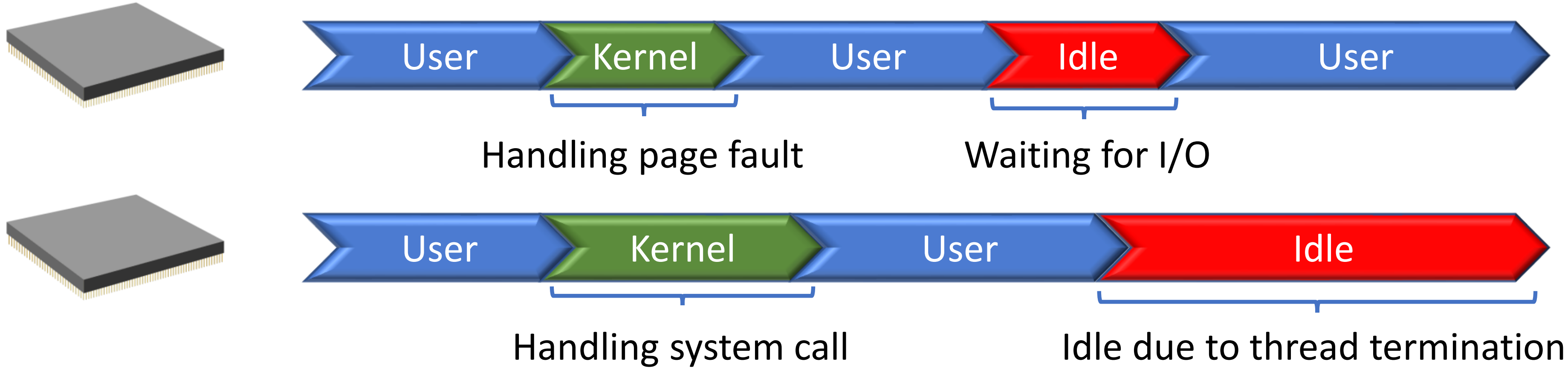
Measuring CPU Utilization

- 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, …)

Time Measurement Example

- 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

- Disk cache misses – indicates high hard disk utilization
- CPU cache misses – indicates high memory bandwidth utilization
- Page faults – indicates high hard disk bandwidth utilization
- File flushes – indicates possible high disk I/O
- Network packets discarded – indicates high network 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