CS1632, LECTURE 13: 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 CPU does a standard query on the database take up? (*utilization*)
  - 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 end-user experience
  - Oblivious to the internals (like blackbox testing)
- Two subcategories:
  - Response Time
    - How quickly does the system respond to a user request?
  - Availability
    - What percentage of time can a user access the services of the system?

Efficiency-Oriented Performance Indicators

- Measures how well a system makes use of computational resources
  - Measures the “internals” of a system
  - May indicate *why* an end-user is having a good / bad experience
- Two subcategories:
  - Utilization
    - How much compute resources does the system use?
  - Throughput
    - How many requests can be processed in a given amount of time?

Service-Oriented vs. Efficiency-Oriented

- In the end, user experience is what’s important (service-oriented)
- But efficiency-oriented indicators directly impact user experience
  - App *utilizes* too much CPU time → *response time* suffers
  - App *utilizes* too much memory and you run out → *availability* suffers
  - App *throughput* cannot handle a surge of requests → *response time* suffers
  - App *throughput* is chronically low → *availability* suffers
- Service-oriented indicators show *whether* users may be dissatisfied
- Efficiency-oriented indicators show *why* users may be dissatisfied
  - And point to a solution to the problem (e.g. install extra memory)

Testing Service-Oriented Performance Indicators

Response Time / Availability

Testing Response Time

- Easy to do!
  - Do something
  - Click “start” on stopwatch
  - Wait for response
  - Click “stop” on stopwatch
  - Write down number on stopwatch!
- Any problems with this approach?

Problems with Response Time Manual Testing

1. Impossible to measure sub-second response times
2. Impossible to measure responses not visible to end-user
3. Human error
4. Time-consuming
5. Probably the most boring thing a person can do

☛ Performance testing relies heavily on automation and statistics.

Statistics? Why?

- You should never trust a single result in performance testing
  - Always try multiple times to get the average value
  - Also look at min/max values to check for large variances
- Why? So many things can go wrong in a single test run:
  - Other processes taking up CPU time
  - Having to swap in memory pages from hard disk
  - Network bandwidth occupied by some other machine
- A single test run is almost worthless.

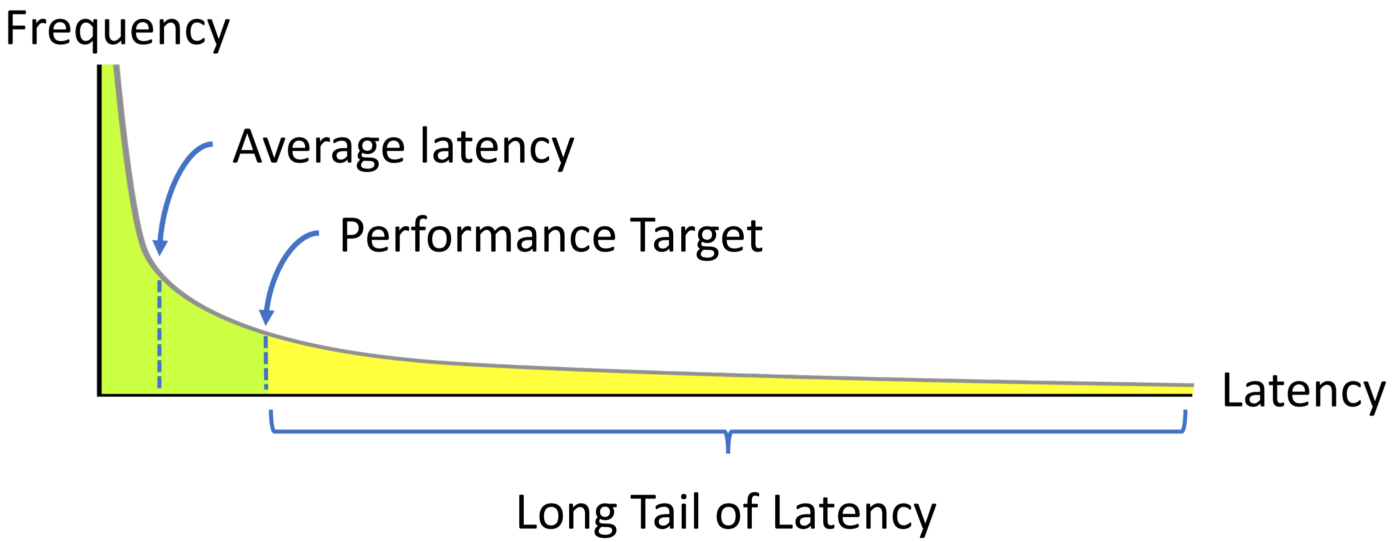
Performance testing is a science

- Eliminate all variables OTHER THAN THE CODE UNDER TEST
  - Make sure you are running on the same hardware configuration
  - Make sure you have identical Library / OS / device driver versions
  - 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 result

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”

Kinds of events to test for response time

- Time for calculation to take place
- Time for character to appear on screen
- Time for image to appear
- Time to download
- Time for server response
- Time for page to load

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!

Time Measurement 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

Testing availability

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

- Often guaranteed in a 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.”

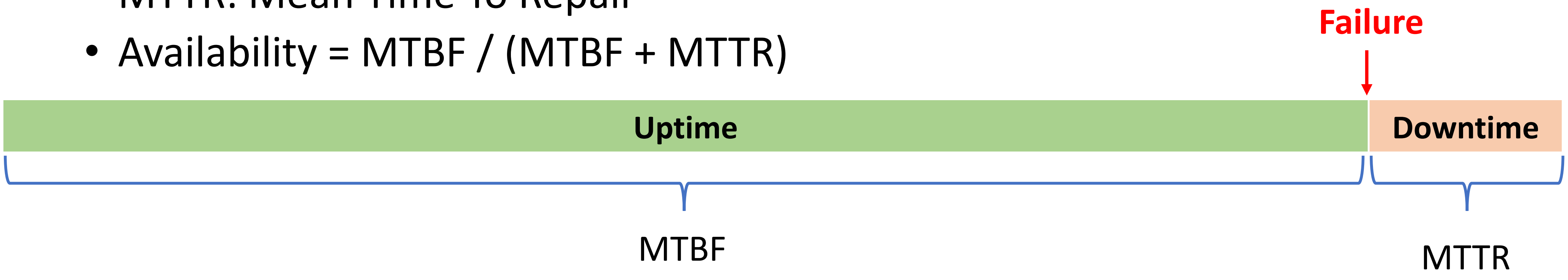
Nines

- 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 base
  - 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

- Many major service providers “breach” their SLAs in a given month
  - Including Microsoft Azure and Amazon Web Services
  - Usually, money is refunded automatically

Developing a Service-Oriented Test Plan

- Think from a user's perspective!
  - How fast do I expect this to be on average?
  - Are large variances in response time allowed?
  - How often do I expect this to be available?

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

Why use Efficiency-Oriented Indicators?

1. More granular than service-oriented indicators
    - Easier to pin down where exactly the performance problem lies

2. Possible to determine how problem can be solved:
    - Software modification (better algorithm, better data structures, etc.)
    - OS tuning (upgrade OS, tune OS scheduler, tune I/O device driver, etc.)
    - Hardware modification (scaling hardware, upgrading hardware, etc.)

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)

How’s that different from service-oriented testing?

1. Users don’t know and don’t care about throughput
   - Users only observe the Quality of Service (QoS) that they are getting (As in, do I notice slow response times or disruption in service?)
   - But engineers care because it helps diagnose and foresee a problem (e.g. if server is close to peak throughput, time to add more machines)

2. More granular
   - Describes a specific system behavior not visible to user (e.g. pages served / sec, SQL queries / sec, DNS queries /sec etc.)
   - Can measure performance of each subcomponent of a system (web server, database server, router, etc.)

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 (QoA)
  - Increase events / second until response time falls below threshold
  - Resulting events / second is the throughput of the system

Testing Utilization

- *You need tools for this*
  - Unless you can tell by the sound of your fan exactly how many operations your program is running on the CPU

Tools

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

- Program-Specific Tools

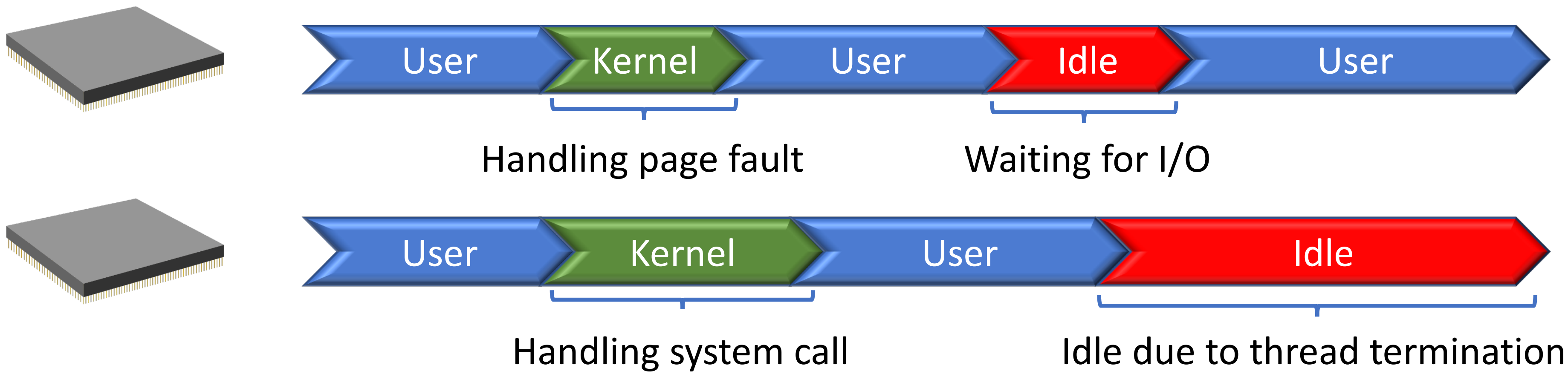
Measuring CPU Utilization

- 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

Performance Indicators for CPU Utilization

- For service-oriented testing
  - Users only care about real time (response time)

- For efficiency-oriented testing
  - Developers also care about other times that help analyze efficiency issues
  - High proportion of **user time**?
    - Need to optimize algorithm or use efficient data structure
  - High proportion of **kernel time**?
    - OS is using a lot of CPU time to handle system calls or handle interrupts
  - Neither? i.e. High proportion of **idle time**?
    - App is spending time waiting for I/O or synchronization
    - CPU utilization is not the problem. Look for efficiency issues somewhere else.

Resources watched by general purpose tools

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

Other Utilization Performance Indicators

- Disk cache misses – may be the reason high disk utilization
- CPU cache misses – may be the reason for high memory bandwidth
- File flushes – frequent forced flushes may contribute to high disk I/O
- Outbound Network Packets discarded – network bandwidth issue?

General purpose tools only give general info

- Lots of memory being taken up…
  - …but by what objects / classes / data?

- Lots of CPU being taken up…
  - …but by what methods / functions?

- Lots of packets sent…
  - …but why? And what’s in them?

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 objects are taking up memory
  - What methods are being called and how often with what duration

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 a Web API showing which cats are available to rent

1. Service-oriented testing
   - Response time: List-sorted-cats API misses performance target of 100 ms
2. Efficiency-oriented testing
   - Throughput testing: Max throughput = 100 requests / second
     Response time deterioration happens on a much lighter load (10 requests / second)
   - Utilization testing (per request):
     Network bandwidth usage is 1%
     Memory usage is 2%
     But the CPU is pegged at 99% for 1 second
   - Diagnosis: Not a throughput problem; problem due to delays in CPU processing

From Service-Oriented Test to Solution

3. Efficiency-oriented testing --- CPU Profiling
    - VisualVM tells you that the sortCats() method is taking most of the time

4. Solution
    - Cats sorted with insertion sort – Use better sorting algorithm
    - Consider using a parallel sorting algorithm if that’s not sufficient

Other Possible Issues and Solutions

- If a lot of time is used in garbage collection
  - Profile memory and try to reduce objects taking up the most memory
  - Tune garbage collector such that it runs less frequently and more efficiently
- If network utilization is the issue
  - Minify webpage JavaScript code to reduce network bandwidth
  - Cache sorted listings in proxy servers at multiple locations

Track Performance throughout Versions

- Performance testing should be part of your regression suite
- Just like for functional defects, you should be able to tell exactly when/where a performance issue 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