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 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

Performance Indicators

- Quantitative measures of the performance of a system under test

- How long does it take to respond to a button press? (*response time*)
- How many users can the system handle at one time? (*throughput*)
- 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

Performance Indicators: Categories

- There are largely two categories of performance indicators

1. Service-Oriented
2. 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, …

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

Efficiency-Oriented Indicators Explain QoS Problems

- If **response time** violates QoS expectations
  - Check if workload exceeds server **throughput**
  - Check if CPU, Memory, I/O, network bandwidth **utilization** is the bottleneck

- If **availability** violates QoS expectations
  - More often a “brown out” rather than a total “black out”
  - Look for **resource saturation** that leads to:
    - Requests timing out because they are queued up in CPU task queue
    - Requests timing out because of memory thrashing and repeated page faults
    - Request packets getting dropped because of network congestion

Efficiency-Oriented Indicators Measure Operating Cost

- Service-Oriented indicators measure QoS for the end-user
  - They do not measure the **cost** of maintaining that QoS

- CPU / Memory / Storage / Network / Energy utilization incur cost
  - In terms of infrastructure cost and energy bills
  - Utilization indicators deserve to be KPIs in their own right

Example: KPIs for a Web Server

- 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: CPU Utilization
  - Performance target: 10 milliseconds per request
  - Performance threshold: 100 milliseconds per request
- KPI: Memory Utilization
  - Performance target: 10 MBs of memory per request
  - Performance threshold: 100 MBs of memory per request

☞Both service-oriented and efficiency-oriented indicators are represented in KPIs!

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 measure sub-second latencies
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 automated tools

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 }

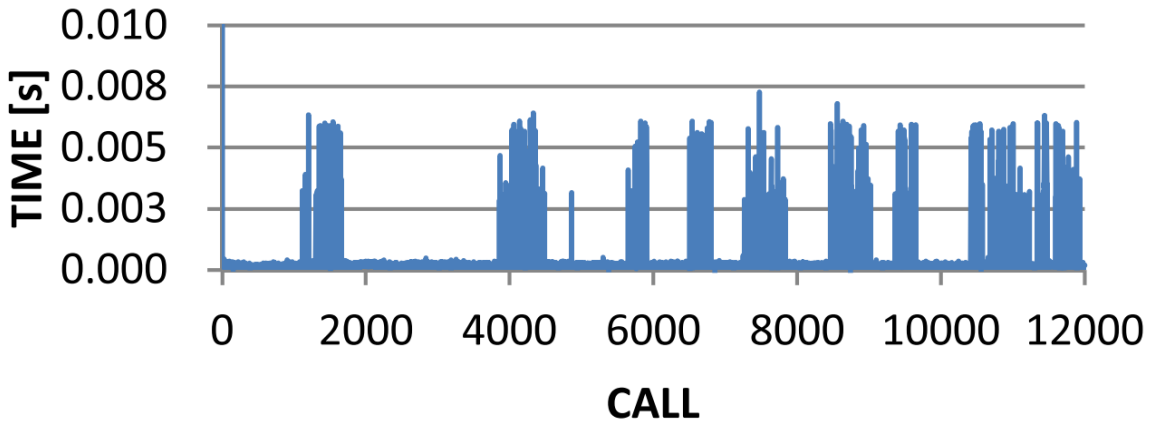
```
-bash$ time curl http://www.example.com
<!doctype html>
<html>
…
</html>

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 may run while testing, taking up CPU time
  - Network may experience traffic while testing from unrelated sources
  - Memory / CPU cache may contain different data at time of test

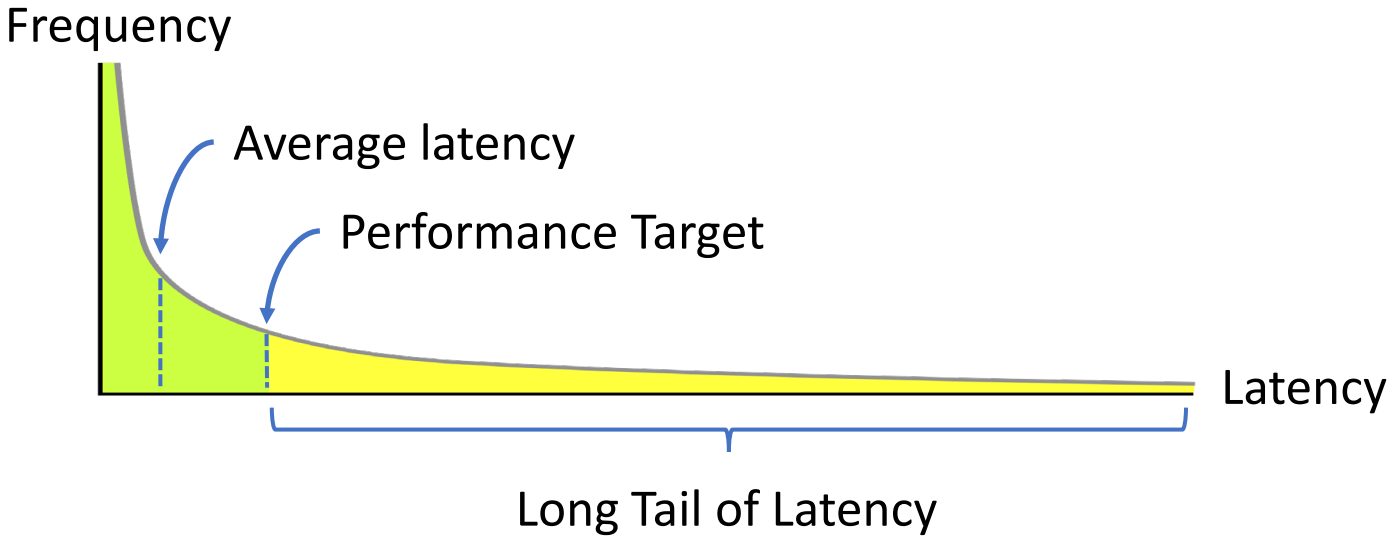
Minimizing and Dealing with Variability

- 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

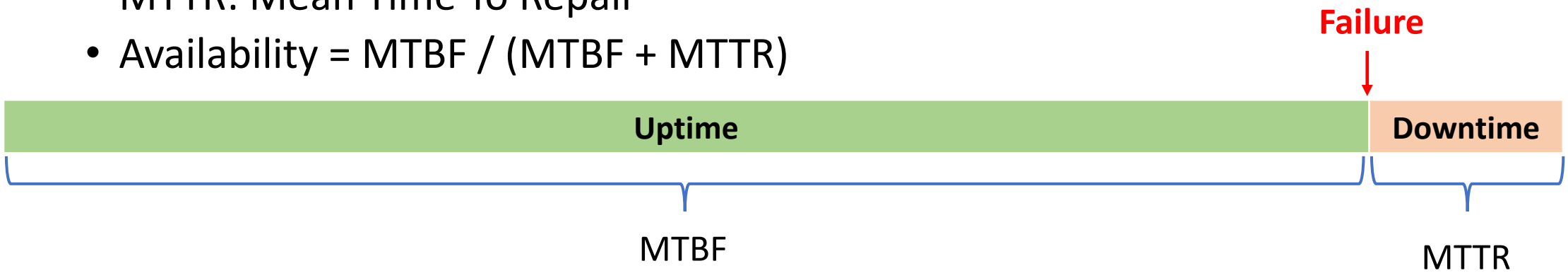


- Often the “long tail” is more important than average latency
  - These are the response times that fail the performance target
- Many runs are required not only to accurately measure the average, but also to detect the length and height of the “long tail”

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 requests (load) system can handle in a given timeframe

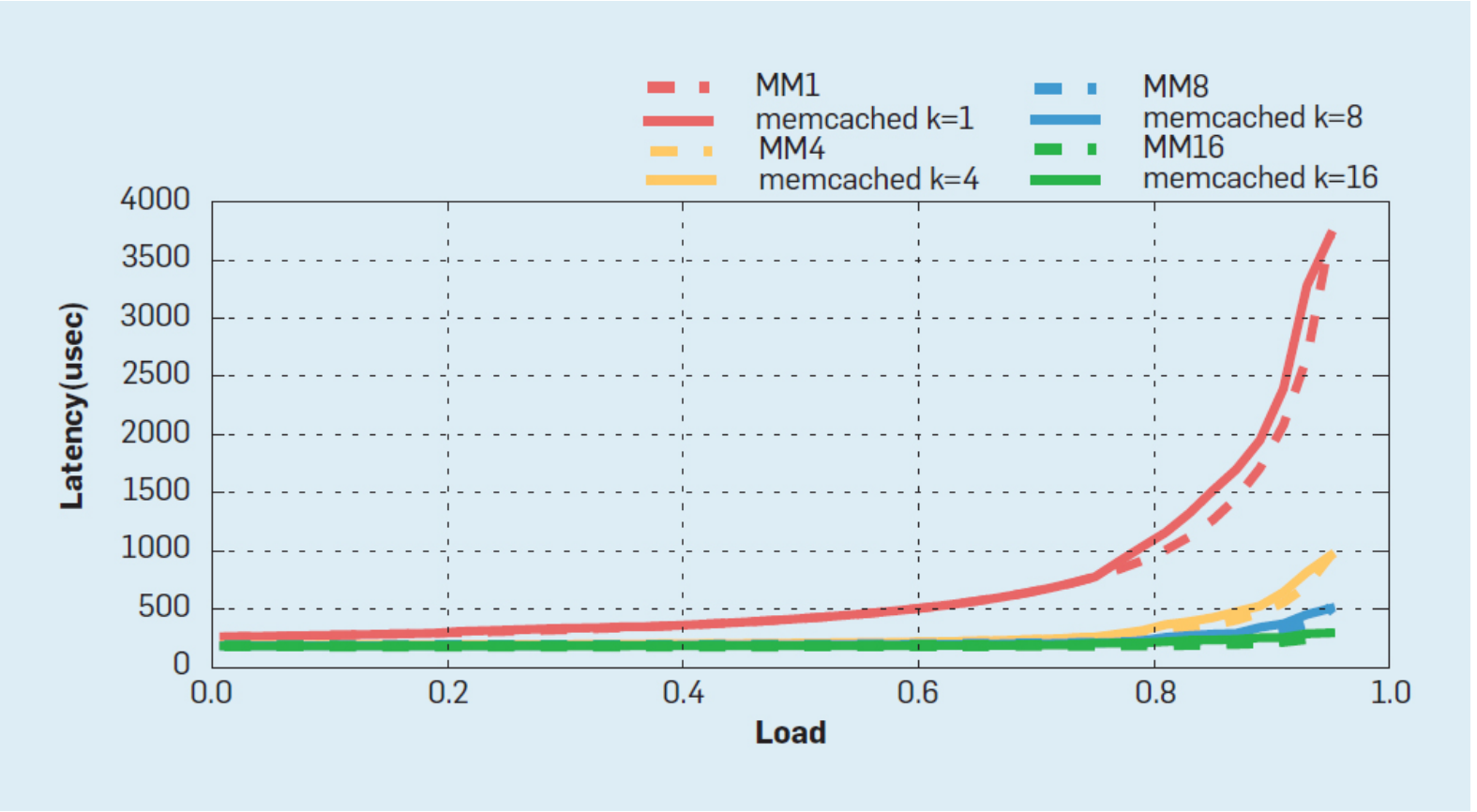
- Examples:
  - Packets per second (that can be handled by a router)
  - Pages per minute (that can be served by a web server)
  - Number of concurrent users (that a game server can handle)

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 load until response time falls below performance target
  - Resulting load 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 for a given load?

- Examples:
  - How many *CPU cycles* are 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?

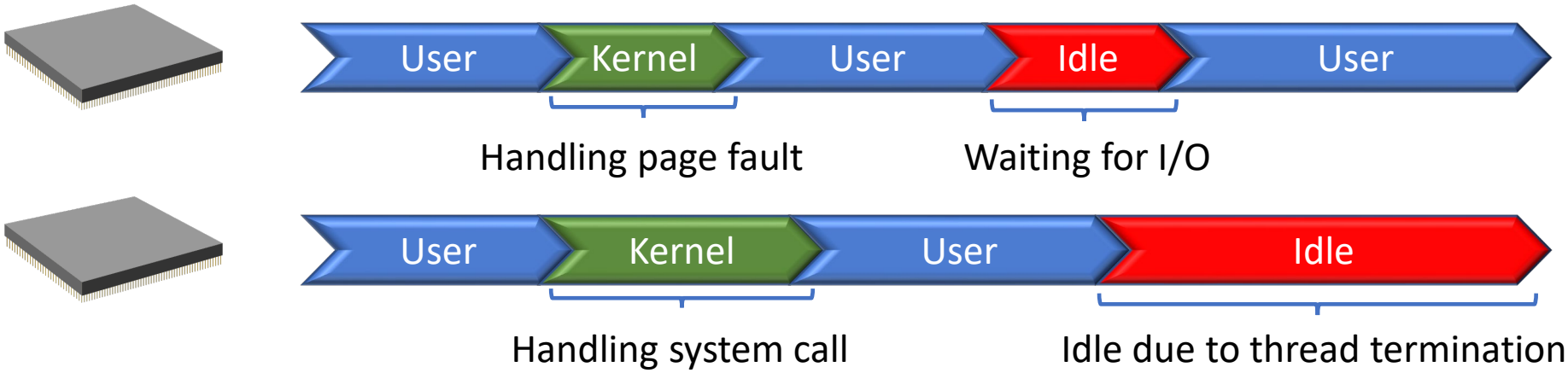
Measuring CPU Utilization

- 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 not executing on CPU waiting for some event (waiting for I/O, synchronization, CPU time slot, …)

Sometimes, Real Time < Total 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, synchronization, or a time slot
  - CPU utilization is not the problem. Look for efficiency issues somewhere else.
  - Need to reduce I/O bandwidth (by compressing data)?
  - Need more CPUs so that threads are not idling waiting for time slots?

Testing Utilization: General Purpose Tools

- General purpose monitoring tools:
  - Measures system-wide / per-processor resource utilization (CPU, memory space, network bandwidth, energy…)

- Examples:
  - Windows - Task Manager, **perfmon**
  - OS X - Activity Monitor, **Instruments**
  - Linux - top, iostat, sar, time, **perf**
  - **Perfmon, Instruments, perf**: accesses OS / CPU performance counters

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

- Performance analysis tools:
  - Analyzes and pinpoints the cause behind high resource utilization

- Examples:
  - Profilers (VisualVM, Jprofiler, gprof, Instruments, perf)
    - Uses instrumentation / sampling to analyze per-method, per-object data
    - Can tell which methods / objects are causes of CPU / memory utilization
  - Protocol Analyzers (Wireshark, tcpdump)
    - Software packet sniffers that monitor packets passing through network stack
    - Analyzes exactly what packets are causes of network utilization

Now Please Read Textbook Chapter 19