# DISTRIBUTED SYSTEMS CS6421
# **RESOURCE MANAGEMENT**

Prof. Tim Wood  and Prof. Roozbeh Haghnazar

Includes material adapted from Van Steen and Tanenbaum's Distributed Systems book
&
Silberschatz, Galvin and Gagne @ 2013

# THIS WEEK…

Resource Management in Distributed Systems

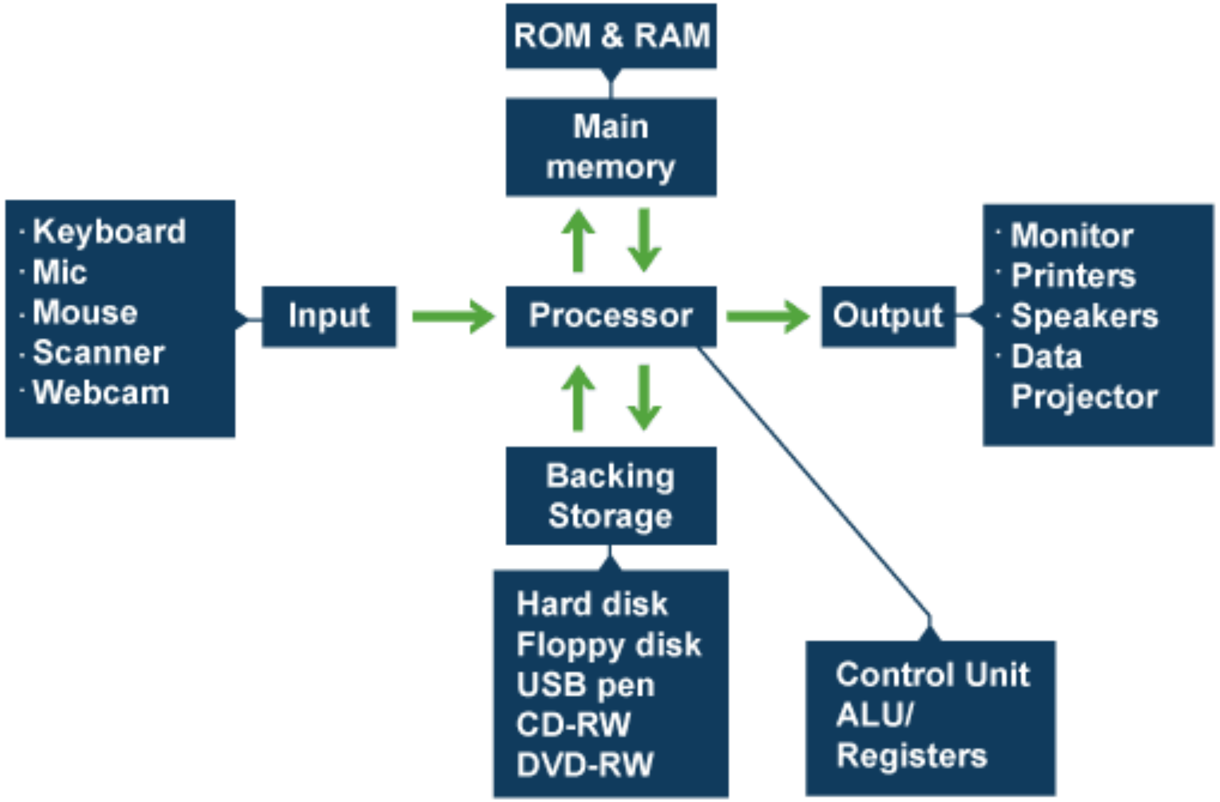
- HW Resources: CPU, Memory, Disk, Network
- Single node vs Cluster management

Common Resource Management Problems

- Placement – entire processes/VMs/containers
- Task Scheduling – long running tasks/jobs
- Load Balancing – fine grained requests

# OS AND RESOURCES

- An operating system has three main functions:
  - manage the computer's resources, such as the central processing unit, memory, disk drives, and network,
  - establish a user interface,
  - Execute and provide services for applications software.
- Operating System
  - CPU Management
  - Memory Management
  - Process Management
  - I/O Management (Disk, Network, etc.)
  - User Management

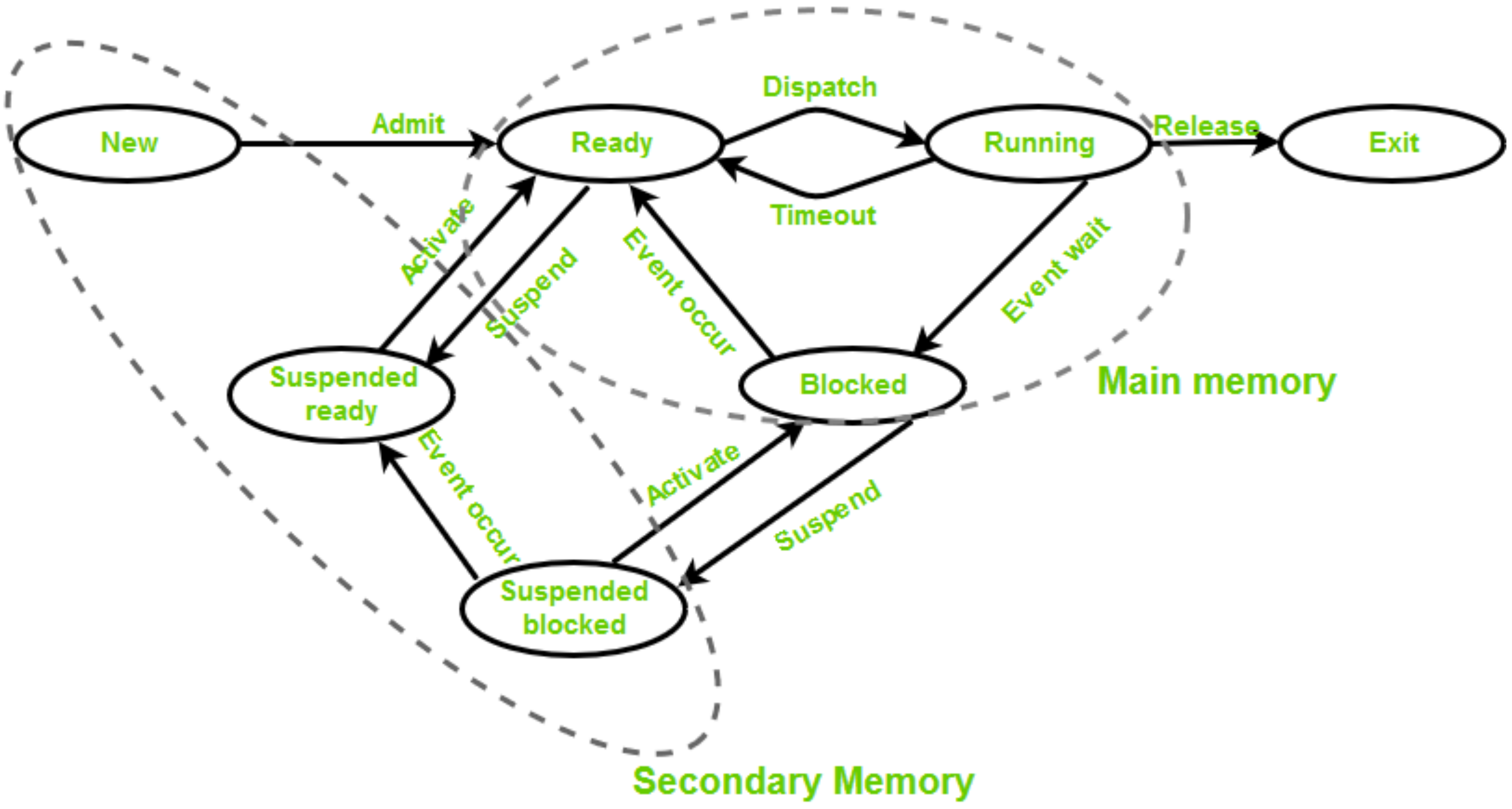


# OS SCHEDULING: A REVIEW

- OS manages resources on my laptop
  - CPU Scheduler – policies to "timeslice" the processor
  - Memory management – apps can be greedy
  - IO – apps can be greedy

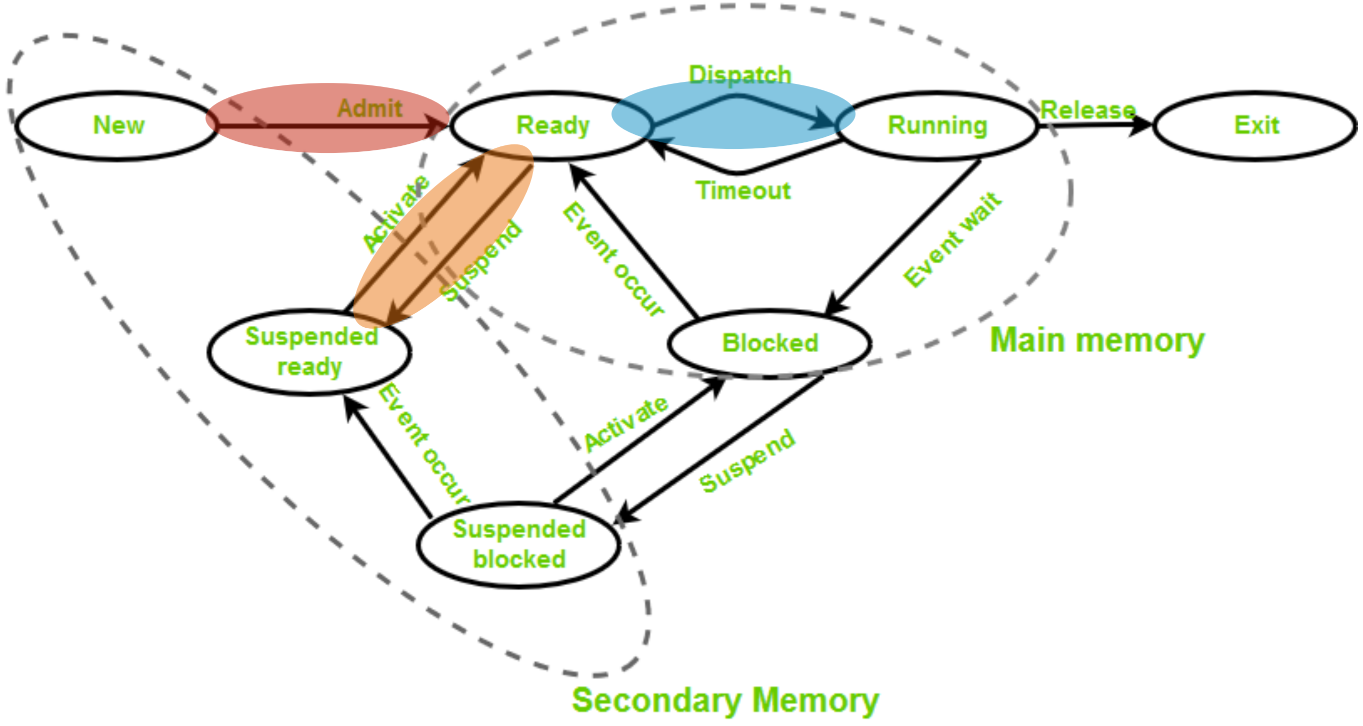
- Linux CPU scheduler decides what to run based on current state of all processes

# PROCESS STATE DIAGRAM



# CLASSIFICATION OF SCHEDULING ACTIVITY

- **Long term – performance** – Makes a decision about how many processes should be made to stay in the ready state
- **Short term – Context switching time** – Short term scheduler will decide which process to be executed next and then it will call dispatcher.
- **Medium term – Swapping time** – Suspension decision is taken by medium term scheduler. Medium term scheduler is used for swapping that is moving the process from main memory to secondary and vice versa.



# SCHEDULING CRITERIA

What are the goals of a CPU scheduler?

# SCHEDULING CRITERIA

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output  (for time-sharing environment)

Do we **maximize** or **minimize** these?

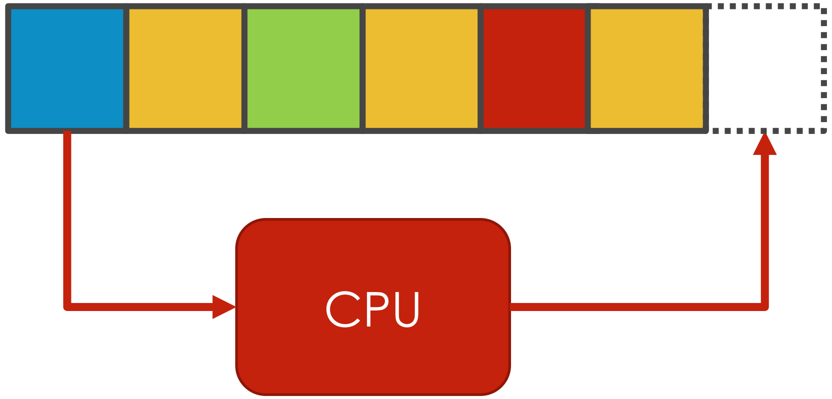
# OPERATING SYSTEM EXAMPLES

Pros/Cons?

# LINUX SCHEDULING HISTORY

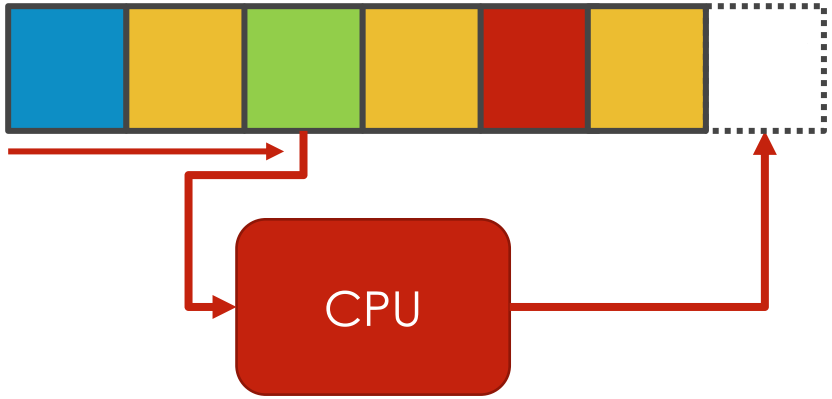
1995
- Version 1.2: Round Robin
  - **Queue** holds all processes
  - Run for a **quantum**, then **preempt**
  - Add to end of queue

RR Run Queue




2001
- Version 2.4: O(n) Scheduler
  - Scan list and pick process with highest "**goodness**"
  - Based on amount of time quantum used and last scheduling time

O(n) Run Queue

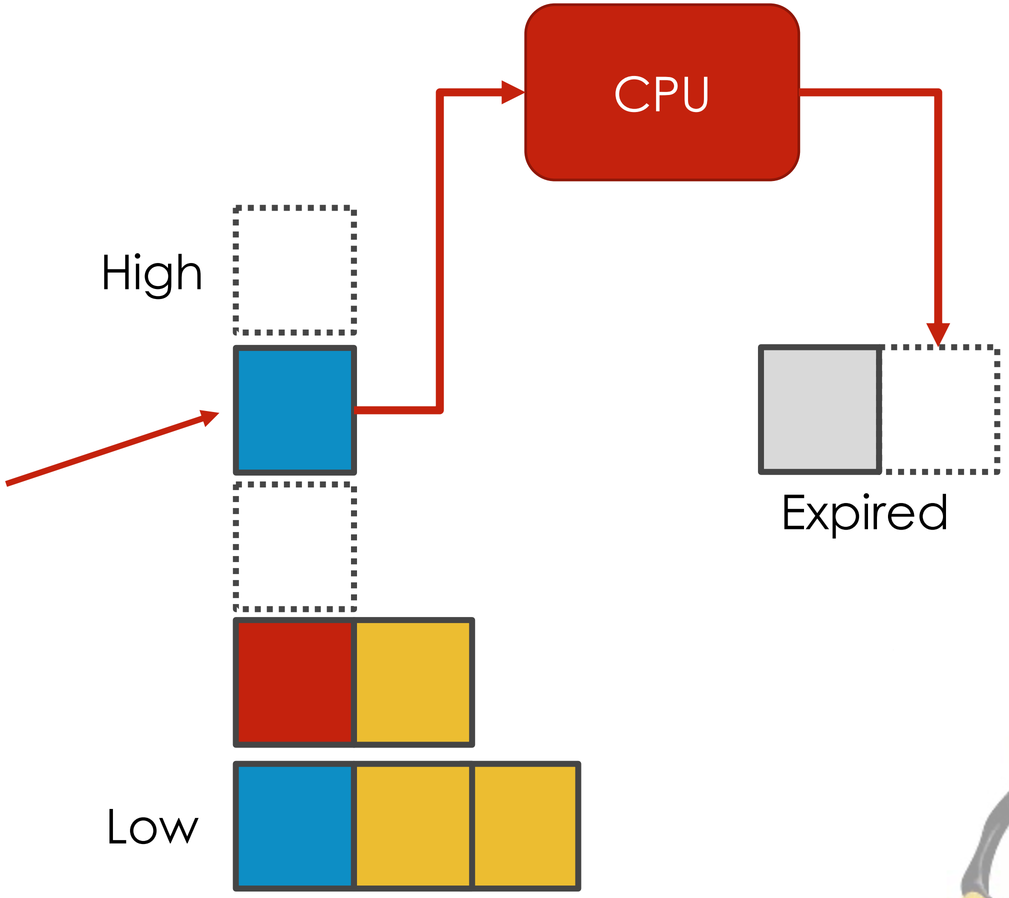


# LINUX SCHEDULING HISTORY

2002

- Version 2.5: O(1) Scheduler
  - Multiple **Priority** Queues (sorted)
  - Pick a priority, take head entry
  - At end of quantum, recalculate time slice and adjust priority
  - Better multi-CPU/multi-core support
  - More complex but efficient

Pros/Cons?



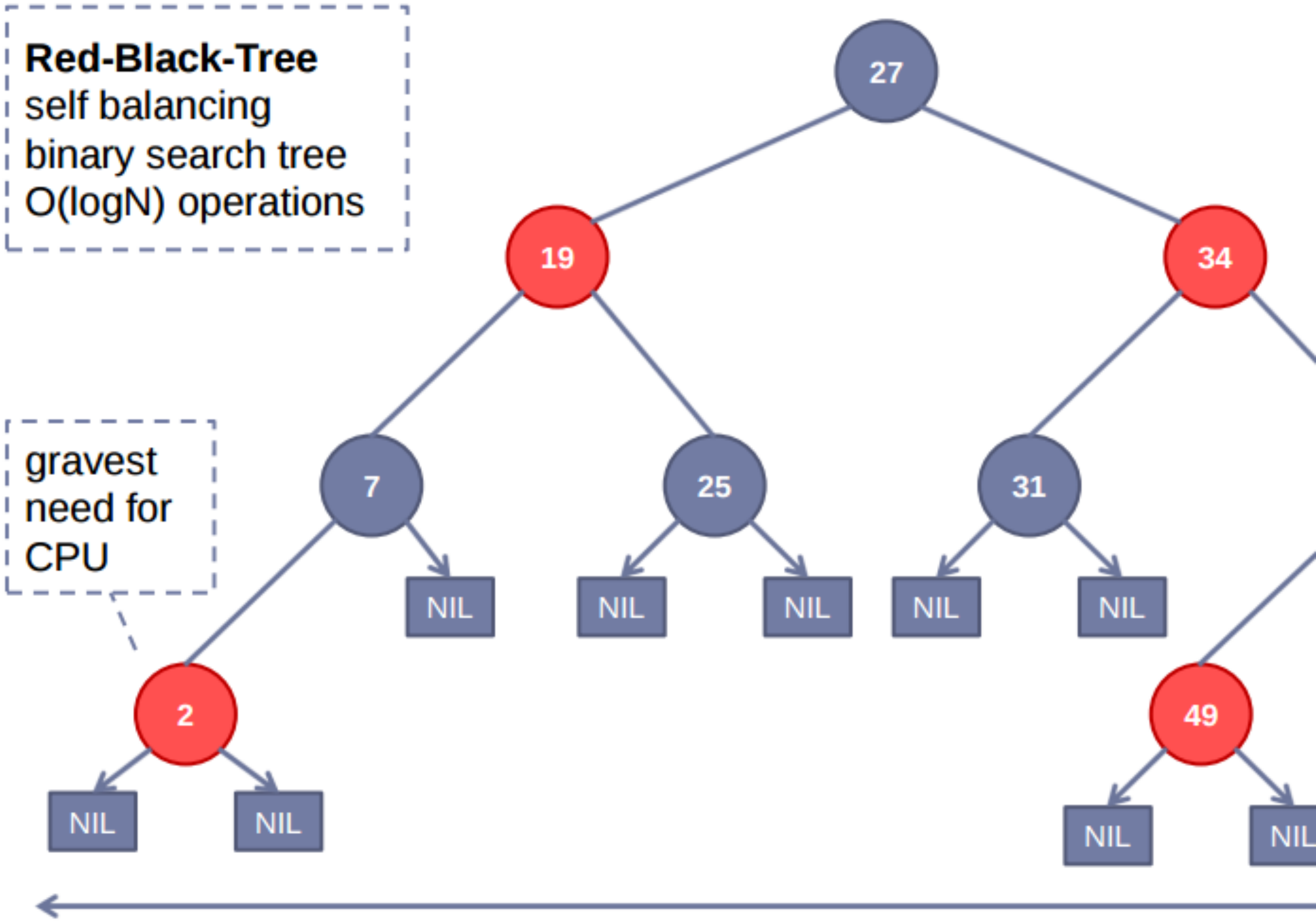

**Array of Priority Queues**

# LINUX SCHEDULING HISTORY

2004 to present

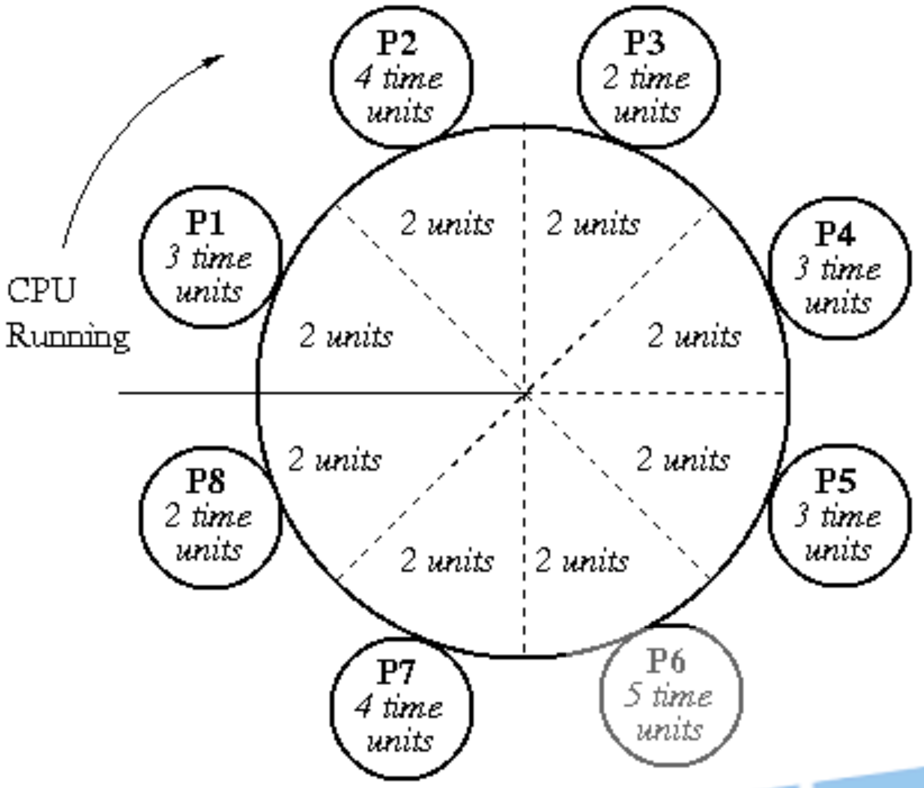
- Version 2.6: CFS
  - Red-Black-Tree instead of queues
  - Processes sorted based on "need"
  - Tries to **fairly** allocate time
  - Schedules interactive tasks more frequently, for shorter times

Pros/Cons?



# WINDOWS SCHEDULER

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- **Dispatcher** is scheduler
- The system assigns time slices in a round-robin fashion to all threads with the highest priority.
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs **idle thread**
- **Multilevel feedback queue** algorithm is used on **windows 10**

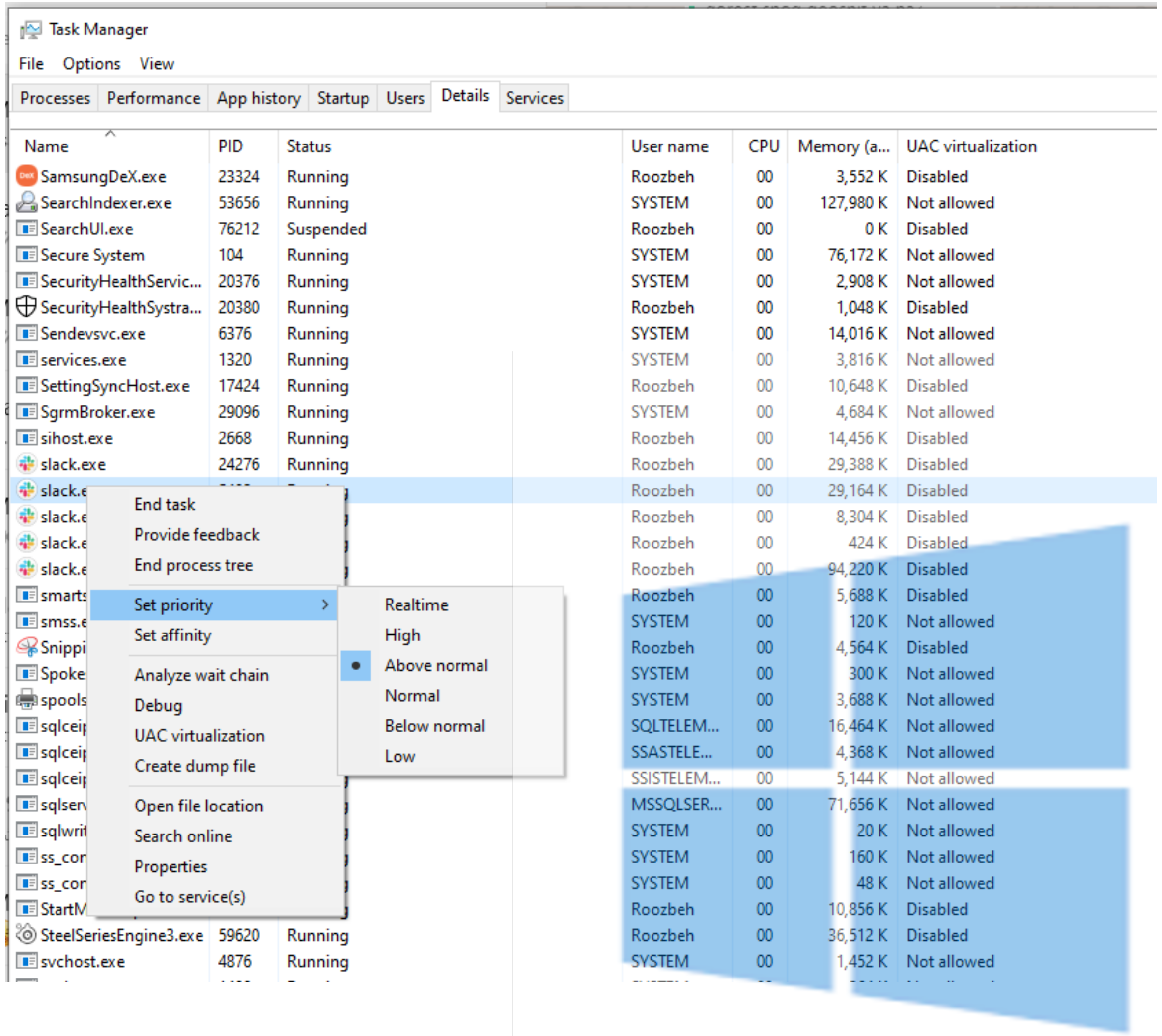


# WINDOWS SCHEDULER

Process Priority Class

Thread Priority Class

| | real-time | high | above normal | normal | below normal | idle priority |
|---|---|---|---|---|---|---|
| time-critical | 31 | 15 | 15 | 15 | 15 | 15 |
| highest | 26 | 15 | 12 | 10 | 8 | 6 |
| above normal | 25 | 14 | 11 | 9 | 7 | 5 |
| normal | 24 | 13 | 10 | 8 | 6 | 4 |
| below normal | 23 | 12 | 9 | 7 | 5 | 3 |
| lowest | 22 | 11 | 8 | 6 | 4 | 2 |
| idle | 16 | 1 | 1 | 1 | 1 | 1 |

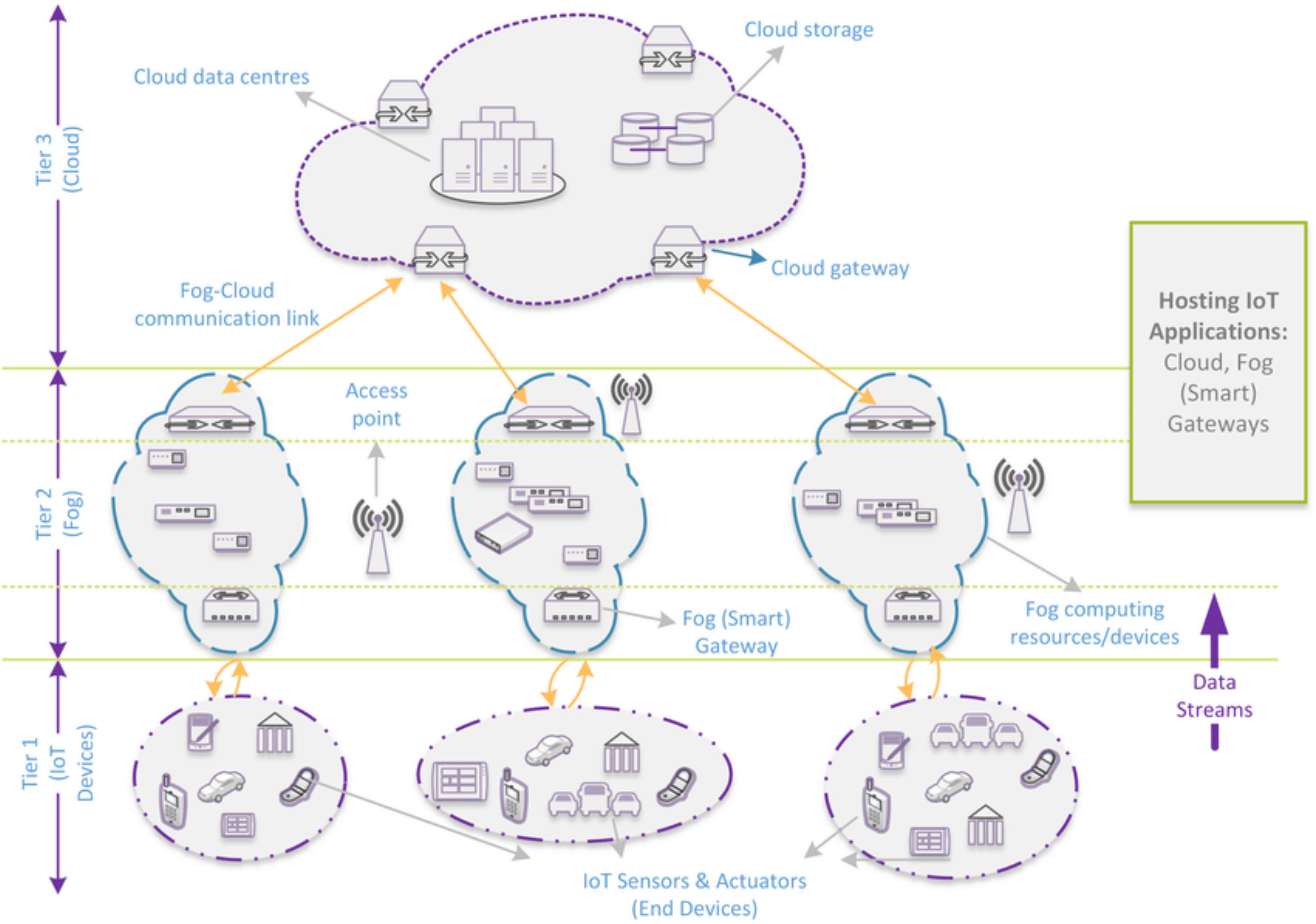



Prof. Tim Wood & Prof. Roozbeh Haghnazar

# OS SCHEDULING SUMMARY

- Scheduler decides which task should run next to meet a **policy**
  - High vs Low priority
  - Real time vs interactive vs batch
  - Fairness between processes
- Scheduler should minimize overhead
  - O(n) vs O(log n) vs O(1)
  - Time quantum

# RESOURCE MANAGEMENT IN DISTRIBUTED SYSTEMS




Prof. Tim Wood & Prof. Roozbeh Haghnazar

# RESOURCE MANAGEMENT VS SCHEDULING

- **Scheduling**
  - Method by which work is assigned to resources that complete the work
  - Focus is on the policy goals (response time, fairness, etc)
  - Typically at fine time scales (milliseconds/seconds)

- **Resource management**
  - Dynamic allocation and de-allocation of processor cores, memory pages, and various types of bandwidth to computations that compete for those resources.
  - Focus is on HW resources (utilization, power consumption, etc)
  - Typically at long time scales (minutes/hours/days)

# DISTRIBUTED RESOURCE MANAGEMENT

Multiple types of HW infrastructure

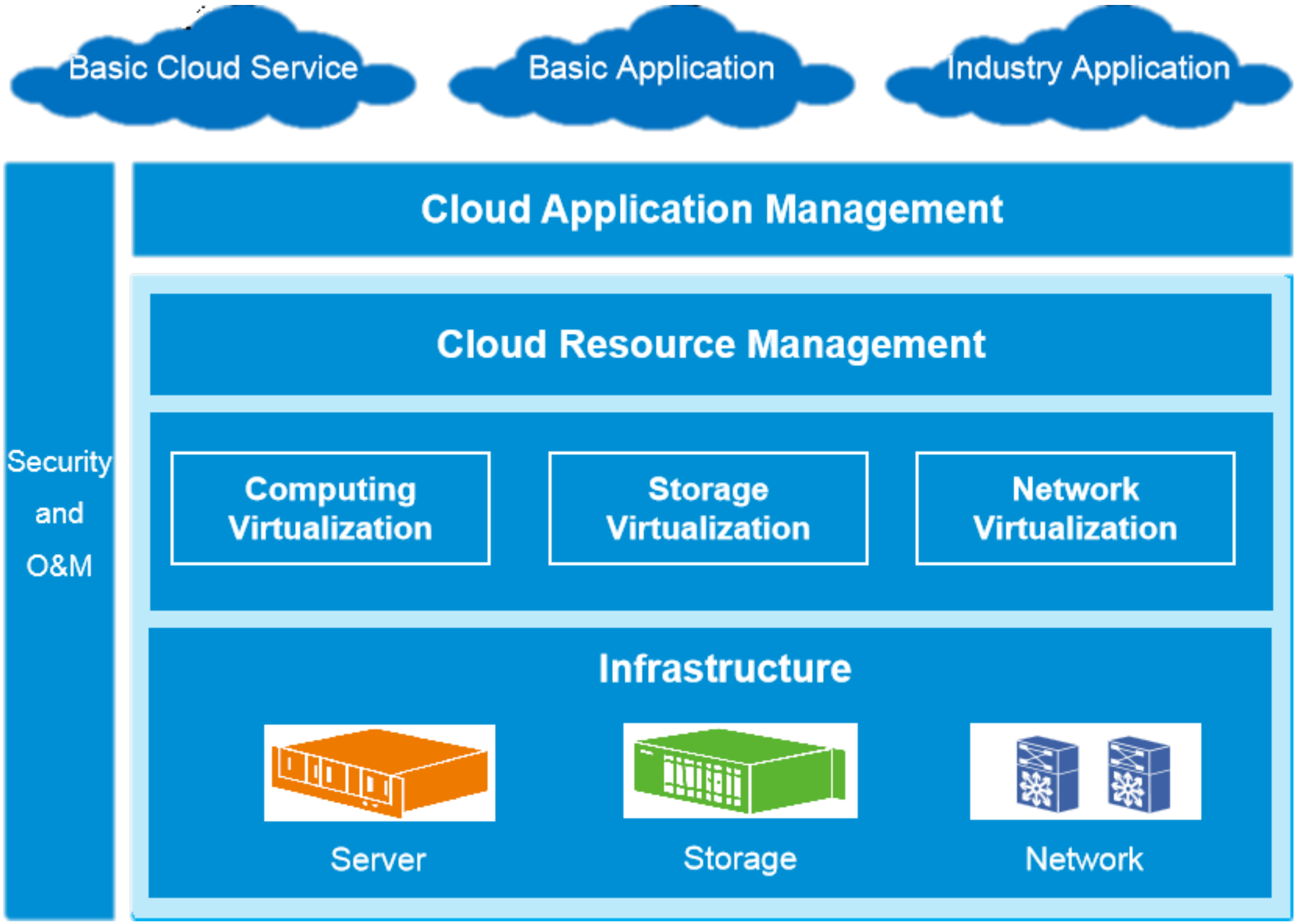
- Compute, Storage, Network

Virtualization lets us “slice” resources

- VMs, storage pools, virtual networks

Resource Management is layered

- Cloud Applications
- Cloud infrastructure
- Individual Servers

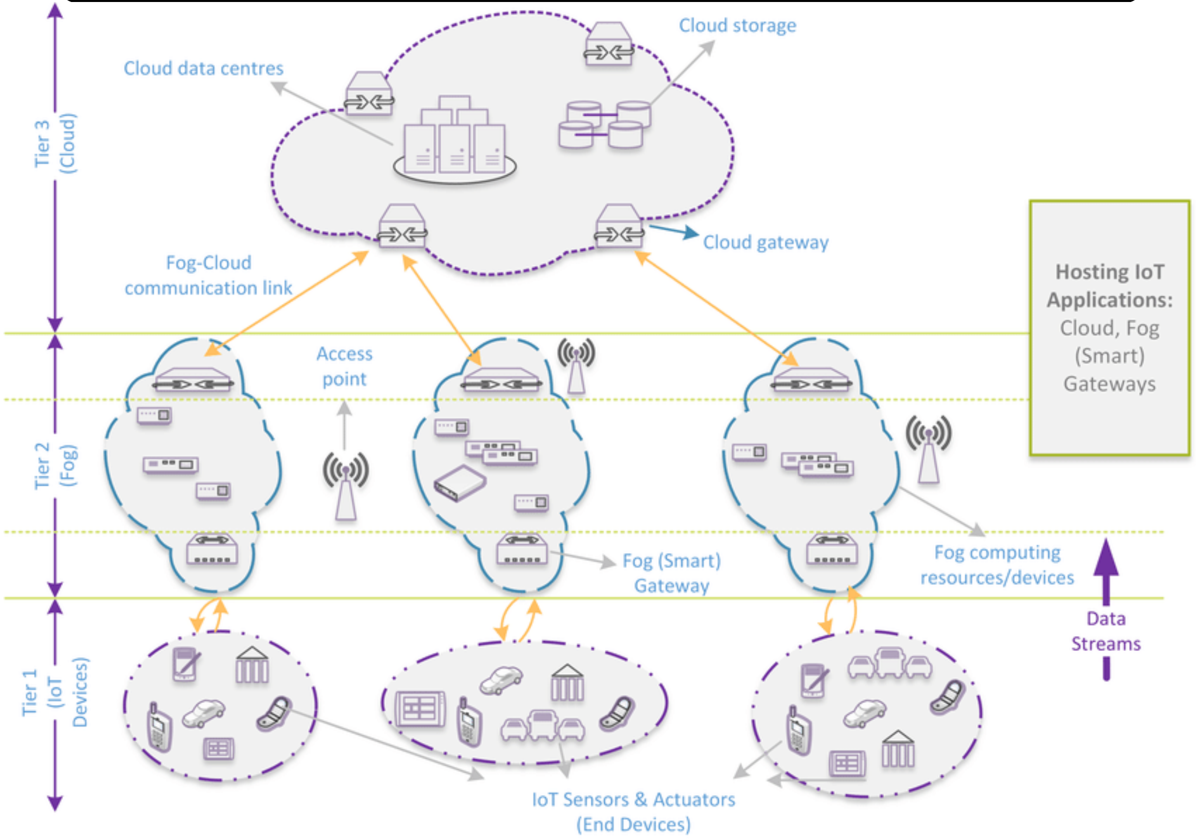



ZTE uSmartCloud Data Center

# REQUIREMENTS

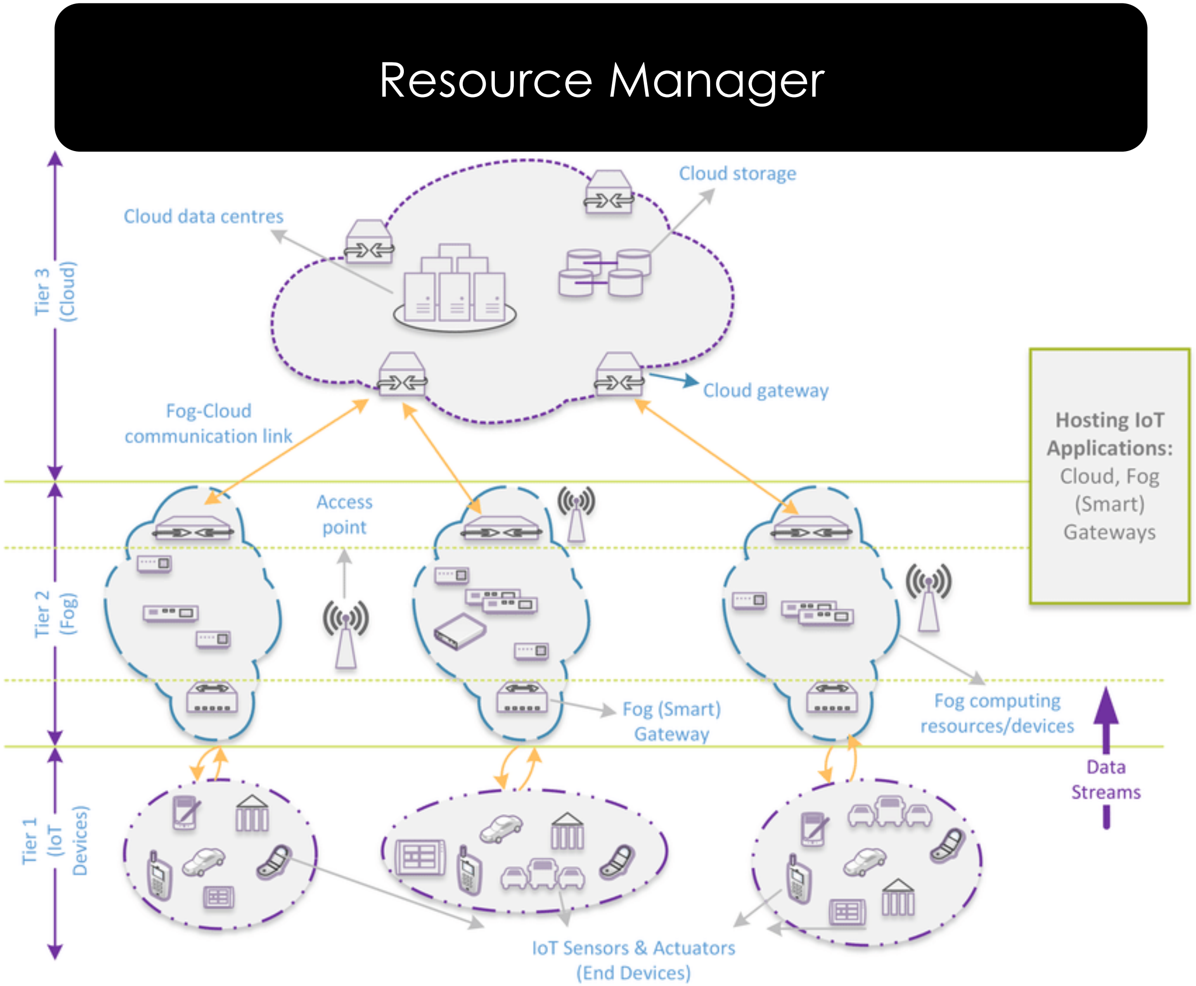
- What **properties** do we want for this Manager?
- What **goals** might it have?

Resource Manager



# REQUIREMENTS

- **Properties:**
  - Scalable
  - Comprehensive
  - Customizable
  - Topology Aware
  - Fault Tolerant
- **Goals:**
  - Throughput, Latency
  - Resource efficiency
  - Fairness
  - Isolation

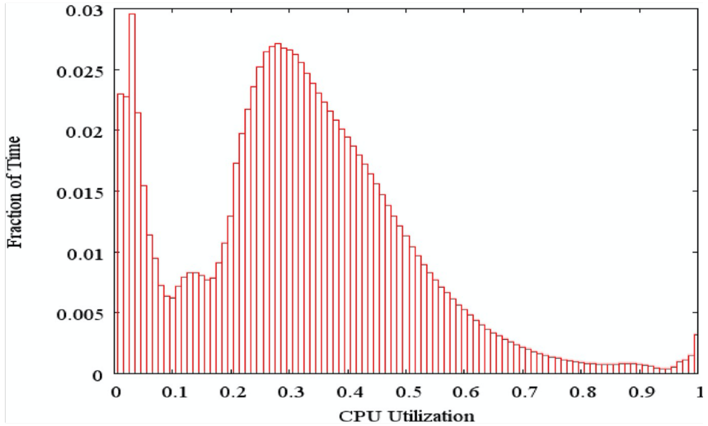


# IMPORTANCE

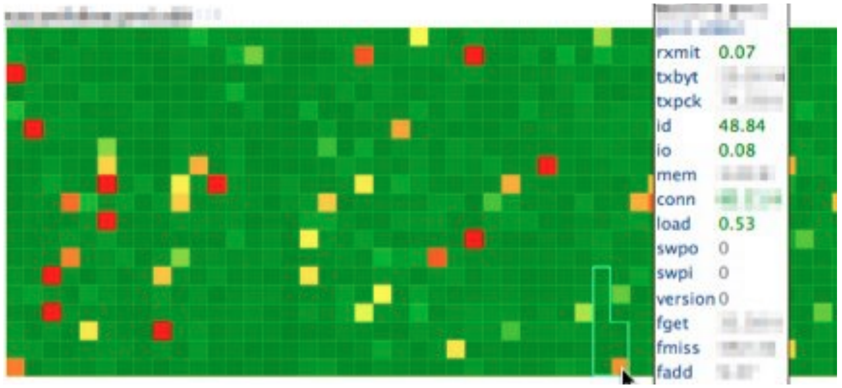
- Efficiently managing a cloud data center important:
  - Consume lots of power!
  - Servers cost lots of money!
- But keeping load evenly balanced is very difficult

"U.S. data centers use more than 90 billion kilowatt-hours of electricity a year... Global data centers used roughly 3% of the total electricity…"
https://www.forbes.com/sites/forbestechcouncil/2017/12/15/why-energy-is-a-big-and-rapidly-growing-problem-for-data-centers/#6d07e9555a30

More than 50% of the cost of running a cloud data center comes from buying servers. Idle servers are a waste of money!
https://perspectives.mvdirona.com/2010/09/overall-data-center-costs/

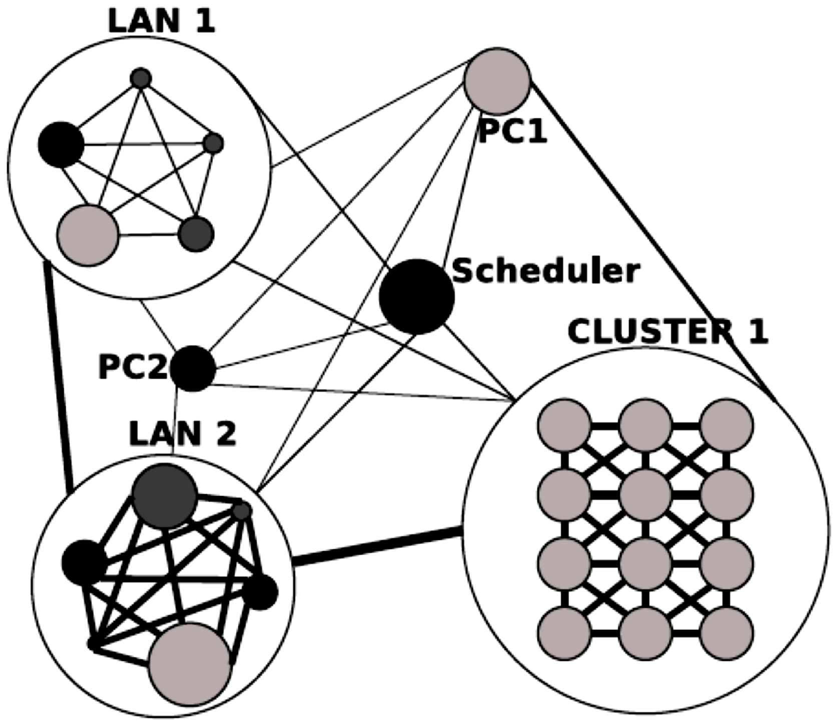


From Google's The Data Center as a Computer report

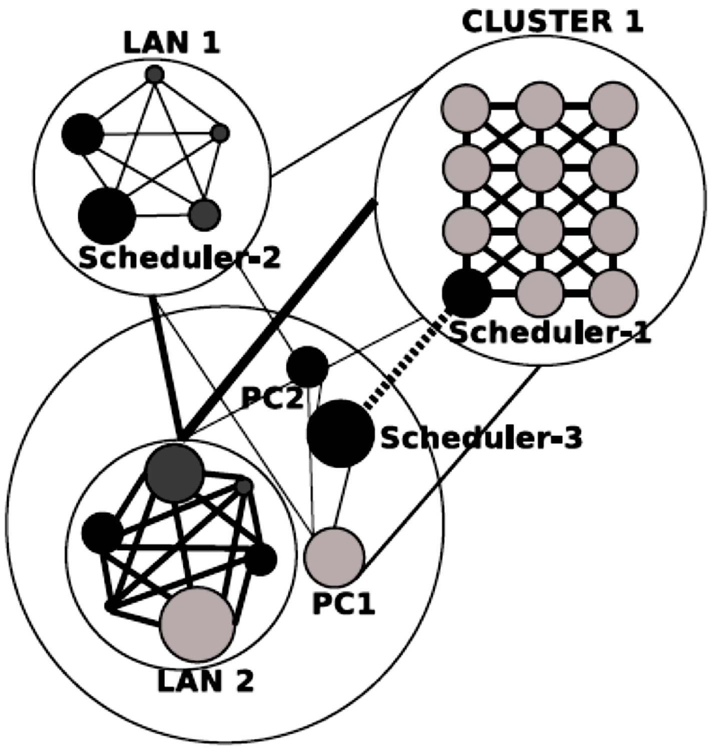


From Facebook Engineering blog on Memcached hotspots

# DISTRIBUTED RESOURCE MANAGEMENT APPROACHES

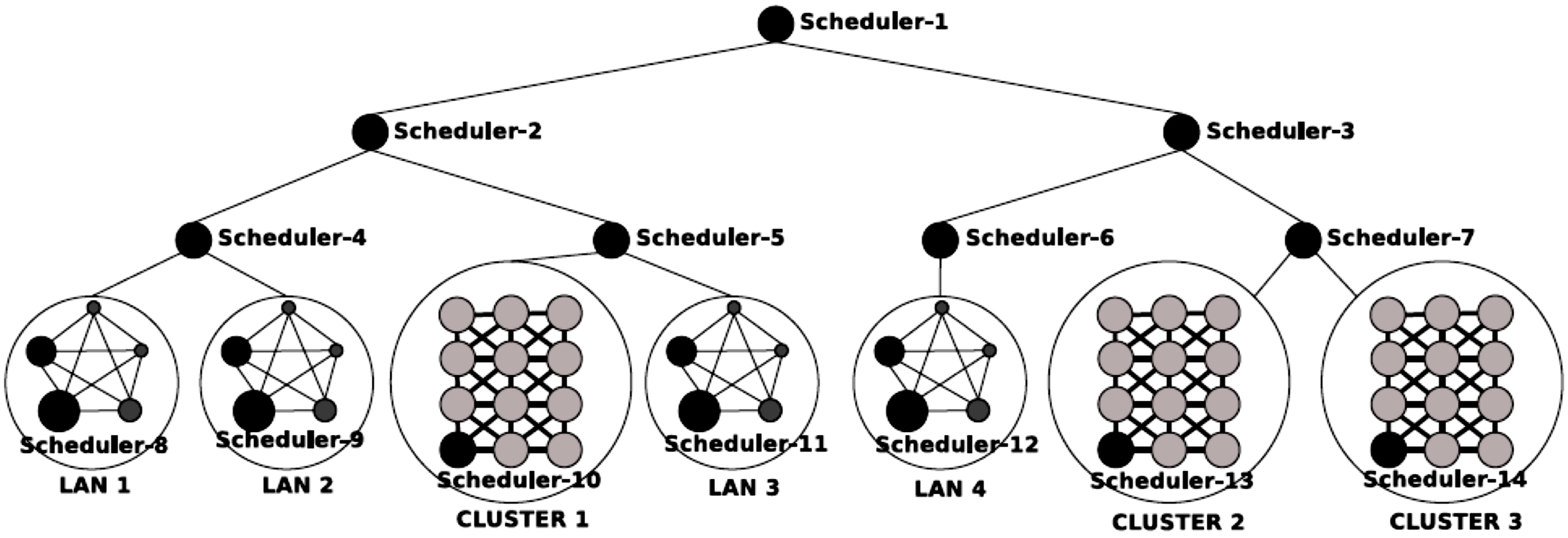



(a) Centralized infrastructure.




(b) Decentralized infrastructure.

# DISTRIBUTED RESOURCE MANAGEMENT APPROACHES



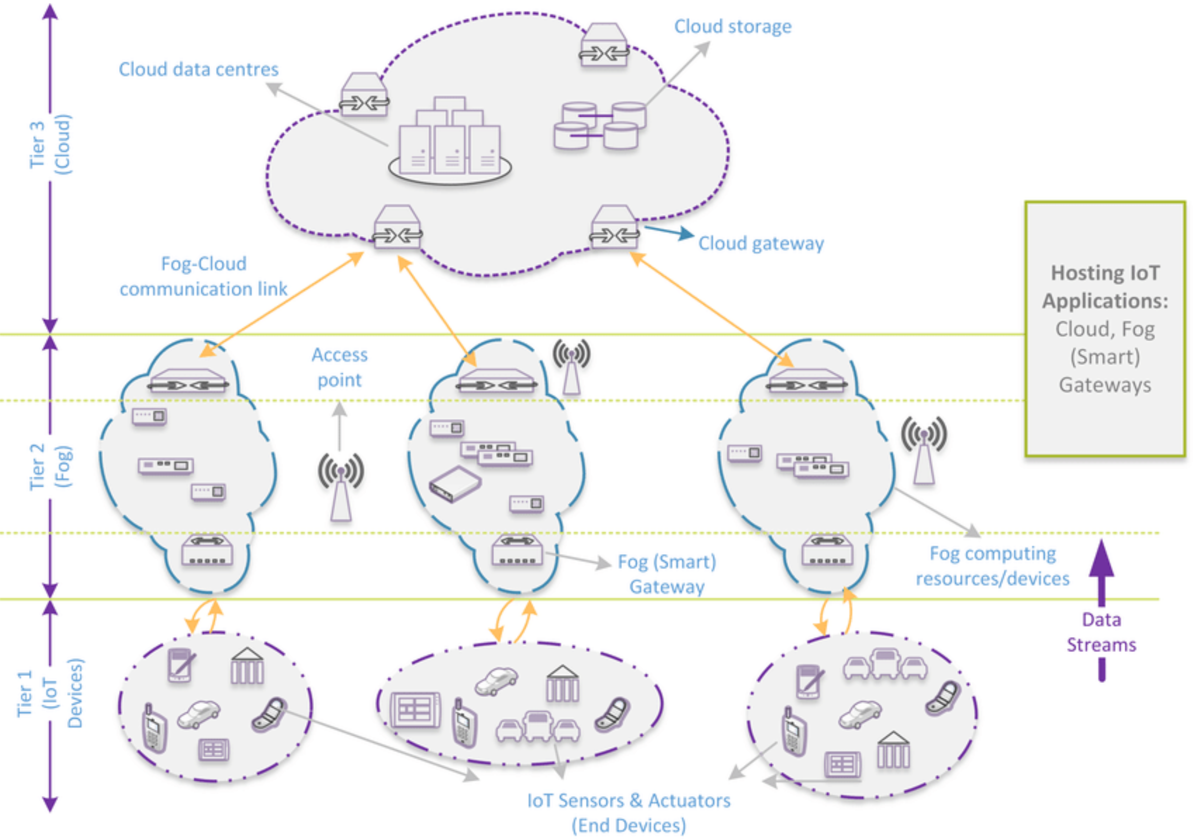

(c) Hierarchical infrastructure.

# DECENTRALIZED MANAGEMENT

- The main characteristics of a decentralized approach are the following:
  - Increased Availability
  - Fault tolerance
  - Enhanced performance
  - Better Scalability
  - Greater Autonomy
- Despite its advantages, there are a lot of challenges in the decentralized management model which are discussed below:
  - Balancing the level of autonomy
  - Complexity of decentralized management
  - How often to share information
  - Decisions based on partial information
  - Scalability
  - Robustness
  - Long delays
  - Fast optimization techniques

# 3 RESOURCE MANAGEMENT CHALLENGES

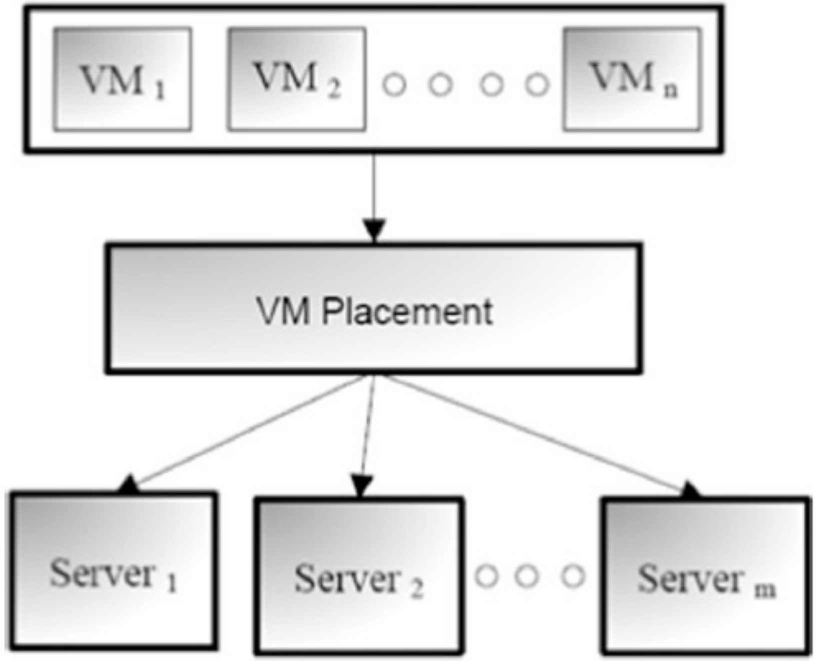
1. Placement – entire processes/VMs/containers
2. Task Scheduling – long running tasks/jobs
3. Load Balancing – fine grained requests



# 1. PLACEMENT

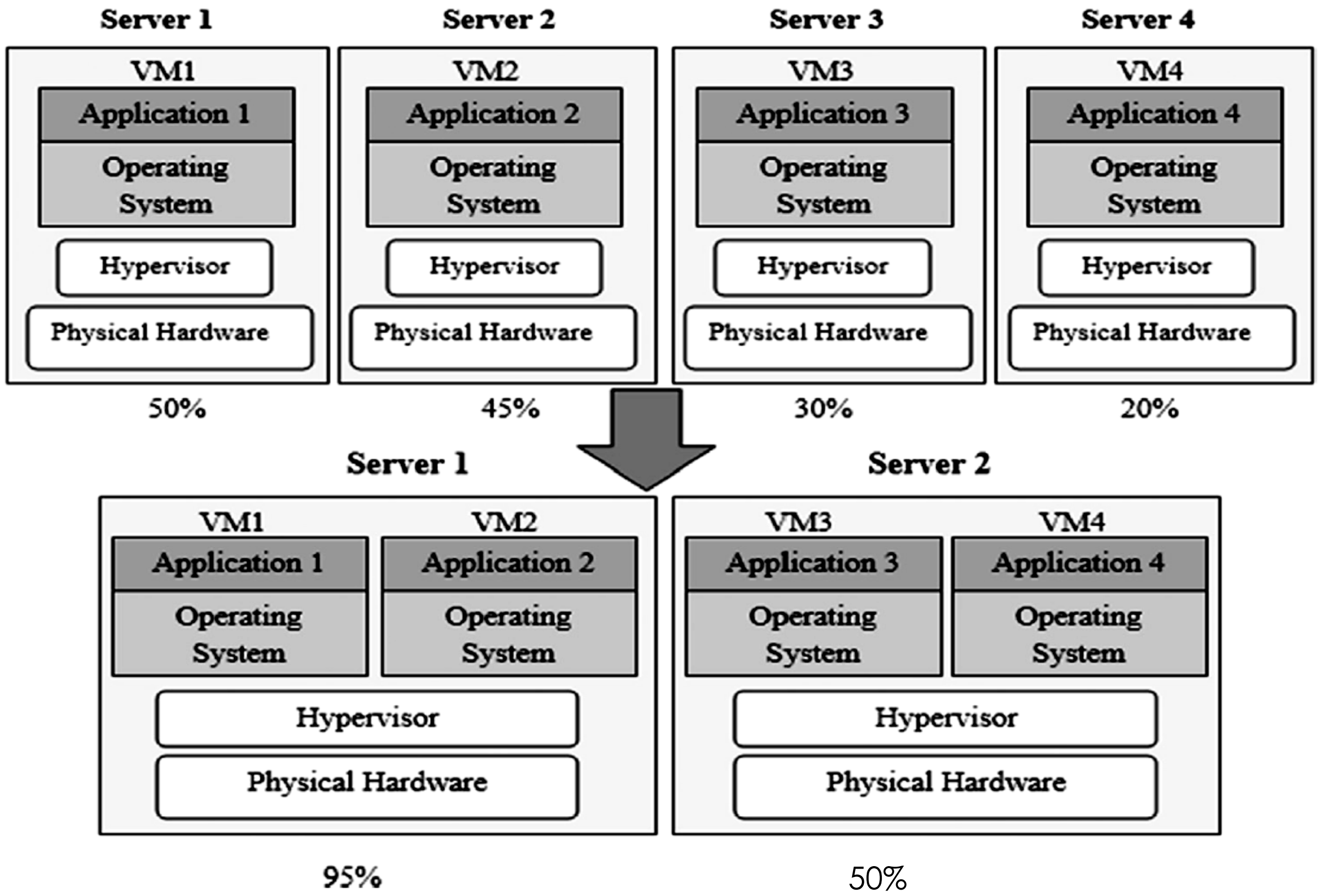
- What should we run on each host in our cluster / data center?
- Depends on the type of distributed system!
  - Super Computer: Run one giant application across all servers
  - Cloud Computing: Divide up each server into many parts and run

- Placement: Where to run each process/VM/container?
  - What factors will affect how difficult this problem is?



# VM CONSOLIDATION

- Increase the energy efficiency by resource management



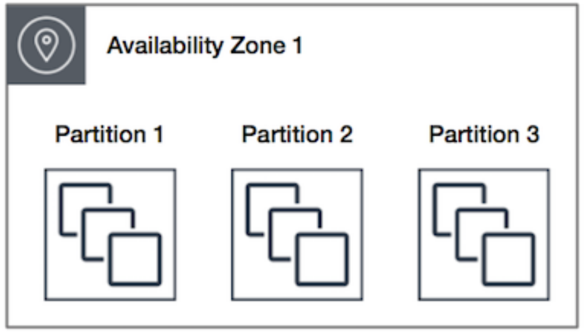
# VM PLACEMENT IN EC2

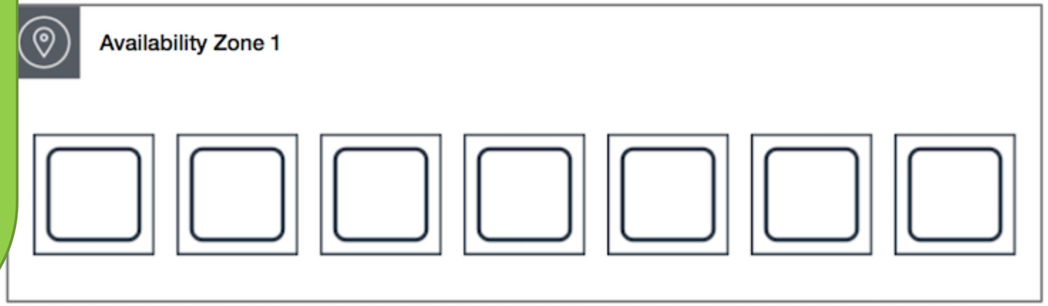
- Depending on the type of workload, you can create a placement group using one of the following placement strategies:

Cluster → Low latency/high throughput network performance/typical of HPC applications

Partition → reduce the likelihood of correlated hardware failures. typically used by large distributed and replicated workloads, such as Hadoop, Cassandra, and Kafka

Spread → to reduce correlated failures. Useful for applications that have a small number of critical instances that should be kept separate from each other. Reduces the risk of simultaneous failures

# PLACEMENT PROBLEM

- Inputs
  - List of VMs
    - CPU and Memory needs
  - List of hosts
    - CPU/memory capacity
- How to assign VMs to hosts?

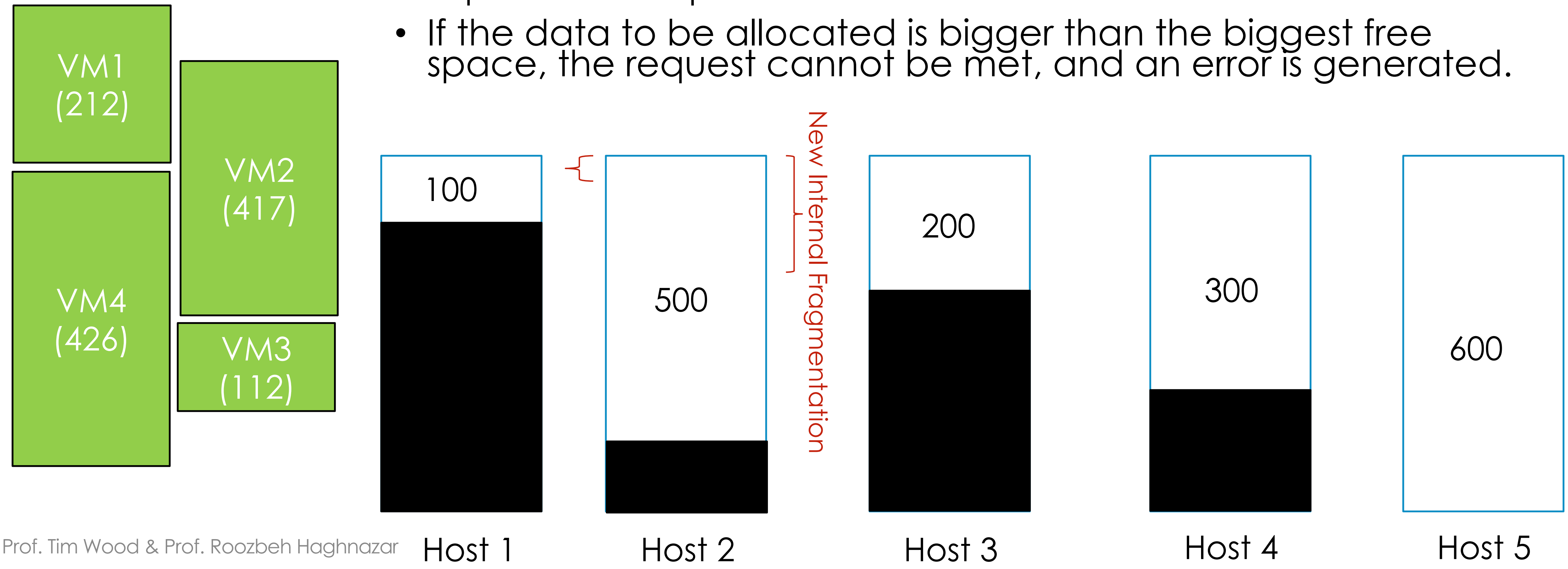
Bin-packing / knapsack

# PLACEMENT HEURISTICS

- In fact, placement problem is a many objective problem space since you have to consider CPU capacity, Memory, Power consumption, Network, and etc. as several dimensions or objective.
- **First Fit**
- **Best Fit**
- **Worst Fit**
- In the following slides we just consider Memory blocks as a single objective to explain the algorithms

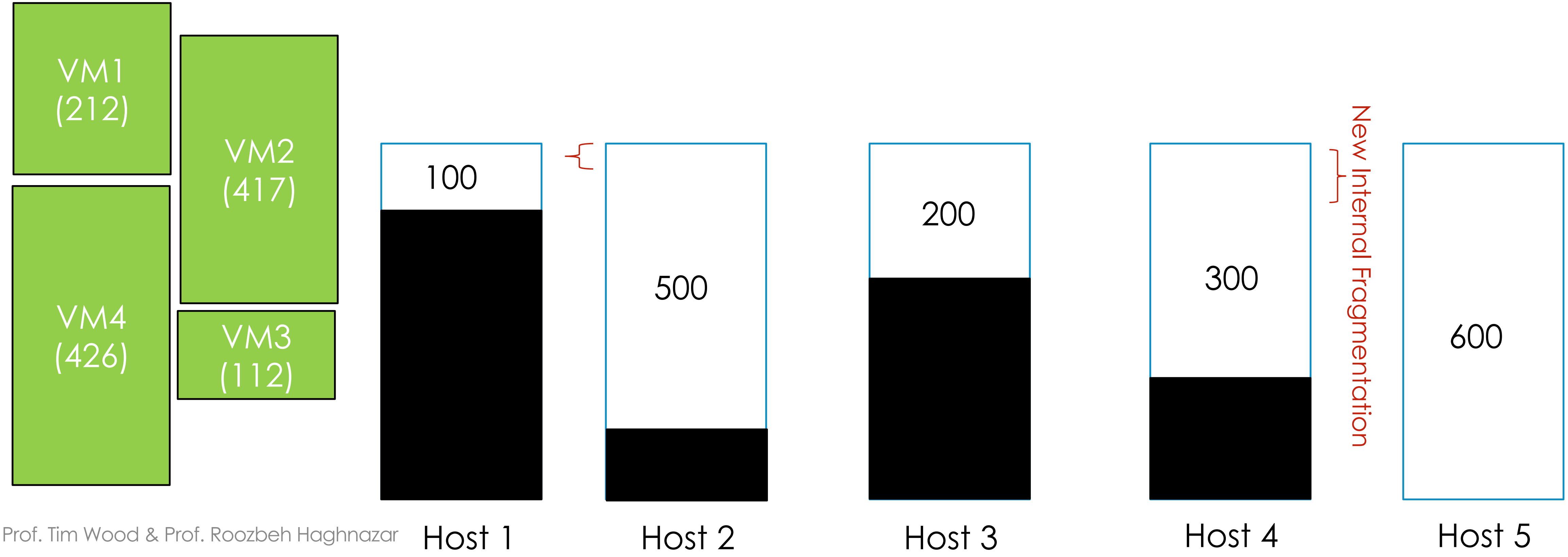
# FIRST FIT (FF)

- A resource allocation scheme (usually for memory). First Fit fits VM into the host by scanning from the beginning of available hosts to the end, until the first free space which is at least big enough to accept the VM is found. This space is then allocated to the data. Any left over becomes a smaller, separate free space.
- If the data to be allocated is bigger than the biggest free space, the request cannot be met, and an error is generated.



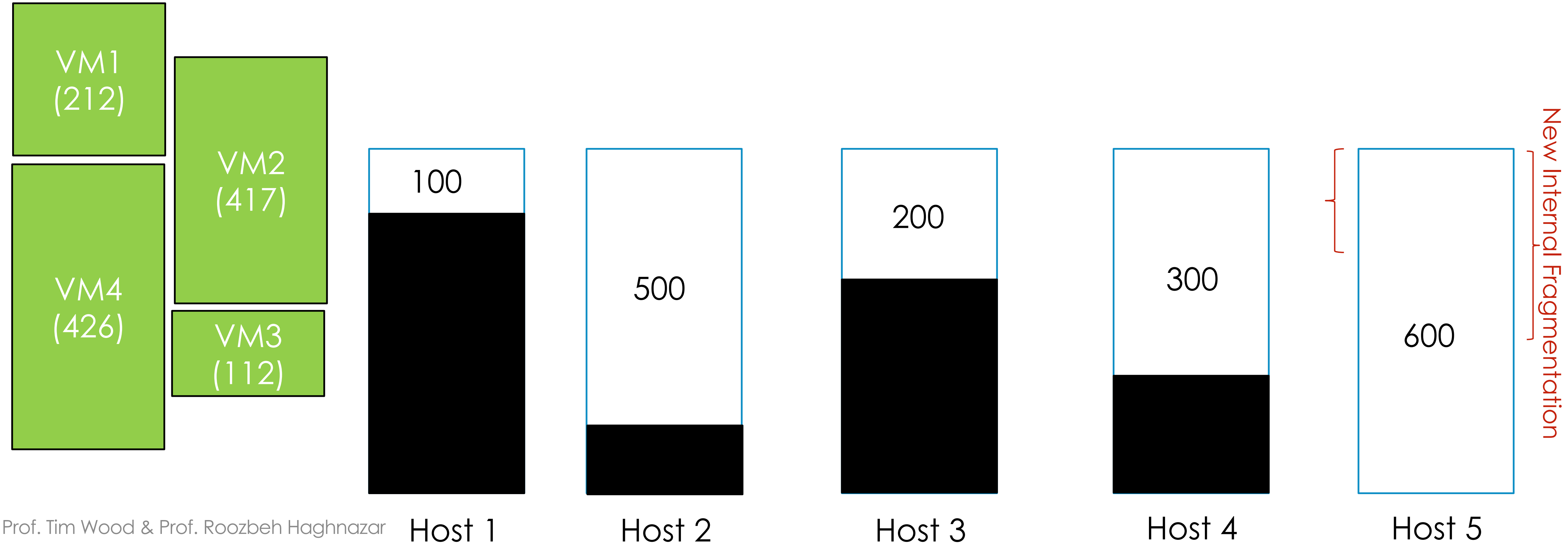
# BEST FIT (BF)

- The best fit deals with allocating the smallest free block which meets the required capacity of the VMs. This algorithm first search the entire list of available hosts then selects the best option – which is the smallest partition – to place the VM. In this method, the space wastage is minimal



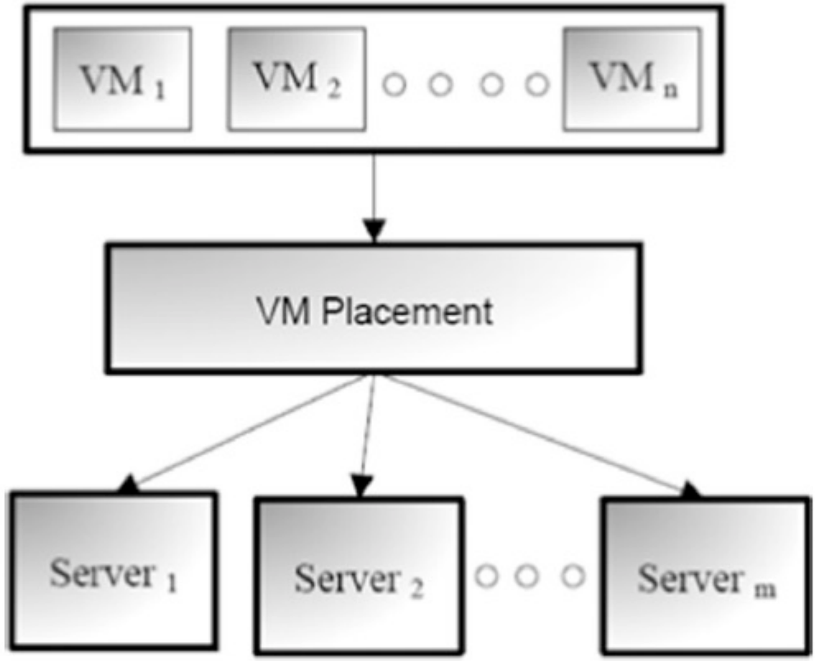
# WORST FIT (WF)

- Worst Fit allocates a VM to the partition which is largest sufficient among the freely available partitions available in the host. If a large process comes at a later stage, then memory will not have space to accommodate it.



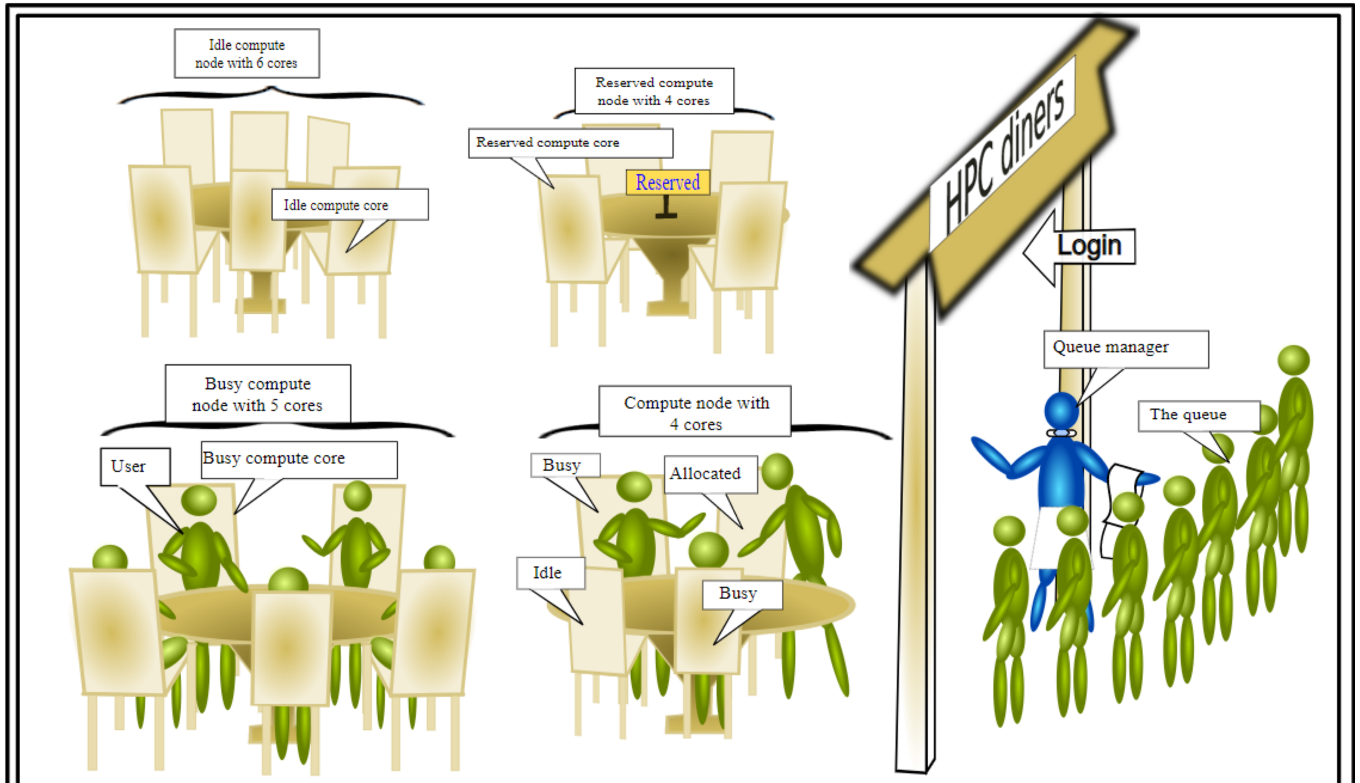
# STATIC VS DYNAMIC PLACEMENT

- VM placement schemes can be classified as dynamic and static:
  - Static VM placement: in which the mapping of the VMs is fixed throughout the lifetime of the VM and it will not be recomputed for a long period of time.
  - Dynamic VM placement: in which the initial placement is allowed to change due to some changes in the system load
    - Reactive VM placement
    - Proactive VM placement



# 2. TASK SCHEDULING

- Given a set of nodes running a service, how should we assign incoming jobs?
  - Finer grained than placement – jobs/tasks typically last seconds-minutes



# SCHEDULING ALGORITHMS

- **Job Scheduling** is invoked *after* services have been deployed by a placement engine
  - Placement engine might deploy a Map Reduce worker node, then a Scheduler determines the order that it processes incoming jobs
- Similar algorithms/policies as **OS CPU scheduling**, but typically focuses on longer time scale
- For our purposes: task = job
  - But this varies by system, e.g., in MapReduce a job is split into tasks but in Real Time Systems, a Task is broken down into jobs…

# FIRST- COME, FIRST-SERVED (FCFS) SCHEDULING

| Process | Exec Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

- Suppose that the processes arrive in the order: $P_1$ , $P_2$ , $P_3$


- Waiting time for $P_1$ = 0; $P_2$ = 24; $P_3$ = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

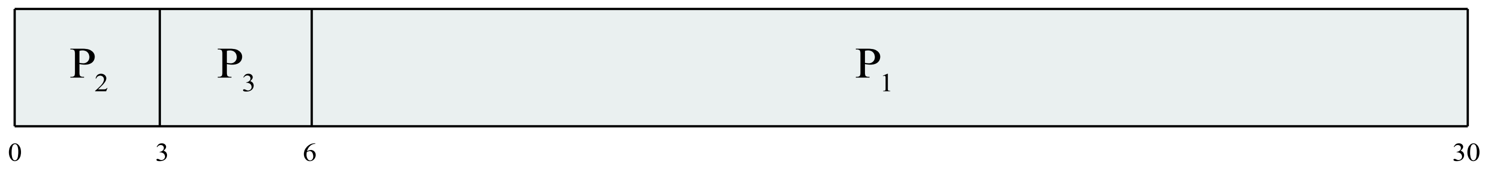
Pros/Cons?

# FCFS SCHEDULING

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:

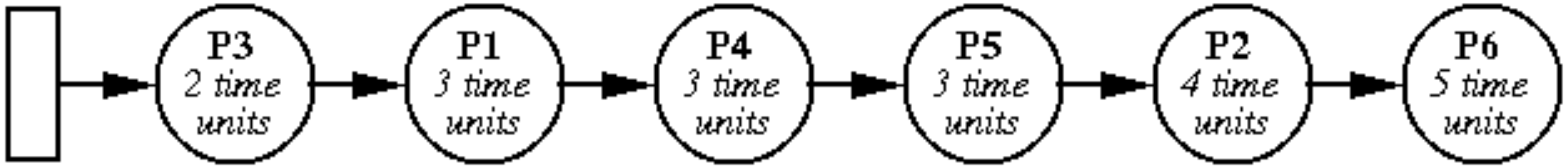



- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case… but need to be lucky!
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

# SHORTEST-JOB-FIRST (SJF) SCHEDULING

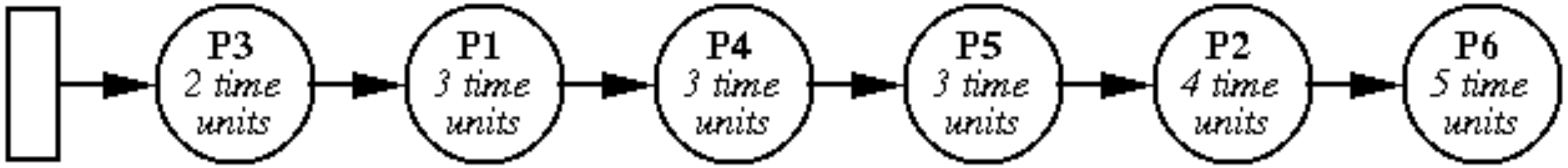
- Sort tasks by the length of their execution time
  - Process shortest tasks first

Pros/Cons?



# SHORTEST-JOB-FIRST (SJF) SCHEDULING

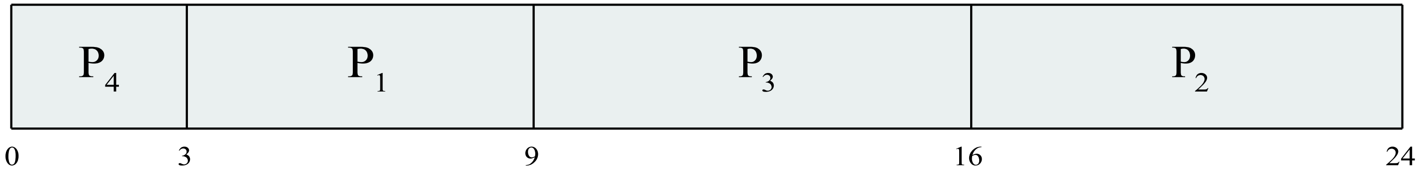
- Sort tasks by the length of their execution time
  - Process shortest tasks first
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user
- **Unfortunately, SJF requires knowledge of the future.**
  - Sometimes we can use past performance to predict future performance!



# SHORTEST-JOB-FIRST (SJF) SCHEDULING

| Process | Exec Time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 8 |
| $P_3$ | 7 |
| $P_4$ | 3 |

Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

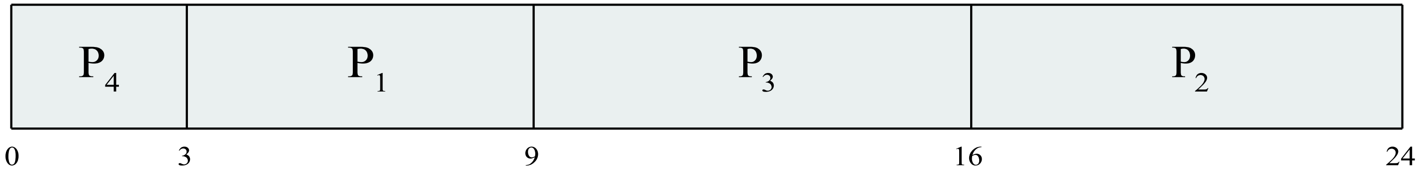


# SHORTEST-JOB-FIRST (SJF) SCHEDULING

| Process | Exec Time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 8 |
| $P_3$ | 7 |
| $P_4$ | 3 |

Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

But what if the user wants P2 done first??!



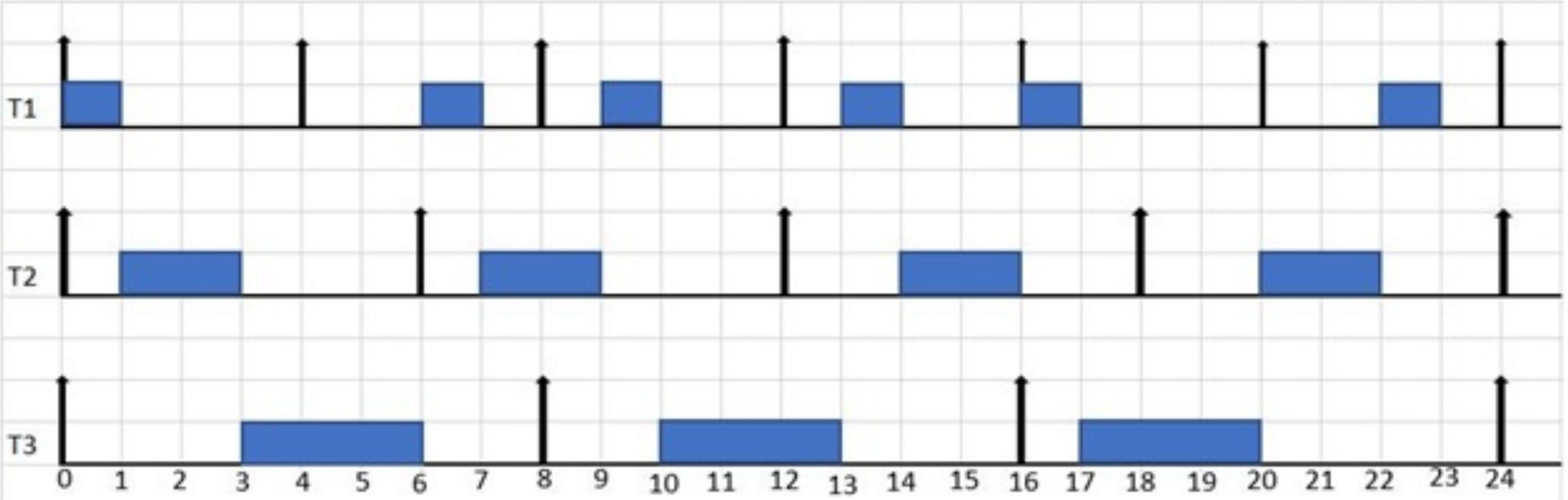
# EARLIEST DEADLINE FIRST (EDF)

- **Earliest Deadline First (EDF)** is an optimal dynamic priority scheduling algorithm used in real-time systems.
- All new tasks announce their **deadline, execution time**, and **period** (interval between arrivals)
- EDF will always schedule the task with the earliest deadline
  - Simple scheduling policy
  - Has provable guarantees about meeting deadlines if possible
- To be optimal, an executing task must be preempted if any other task with an earlier deadline arrives (increases system complexity)
- EDF has been utilized and implemented in the many systems (either as CPU scheduler or Job Scheduler):
  - Linux (SCHED_DEADLINE) and the Xen Virtualization Platform
  - Real Time OSes: S.Ha.R.K, ERIKA Enterprise, Everyman, MaRTE OS, others

# EARLIEST DEADLINE FIRST (EDF)

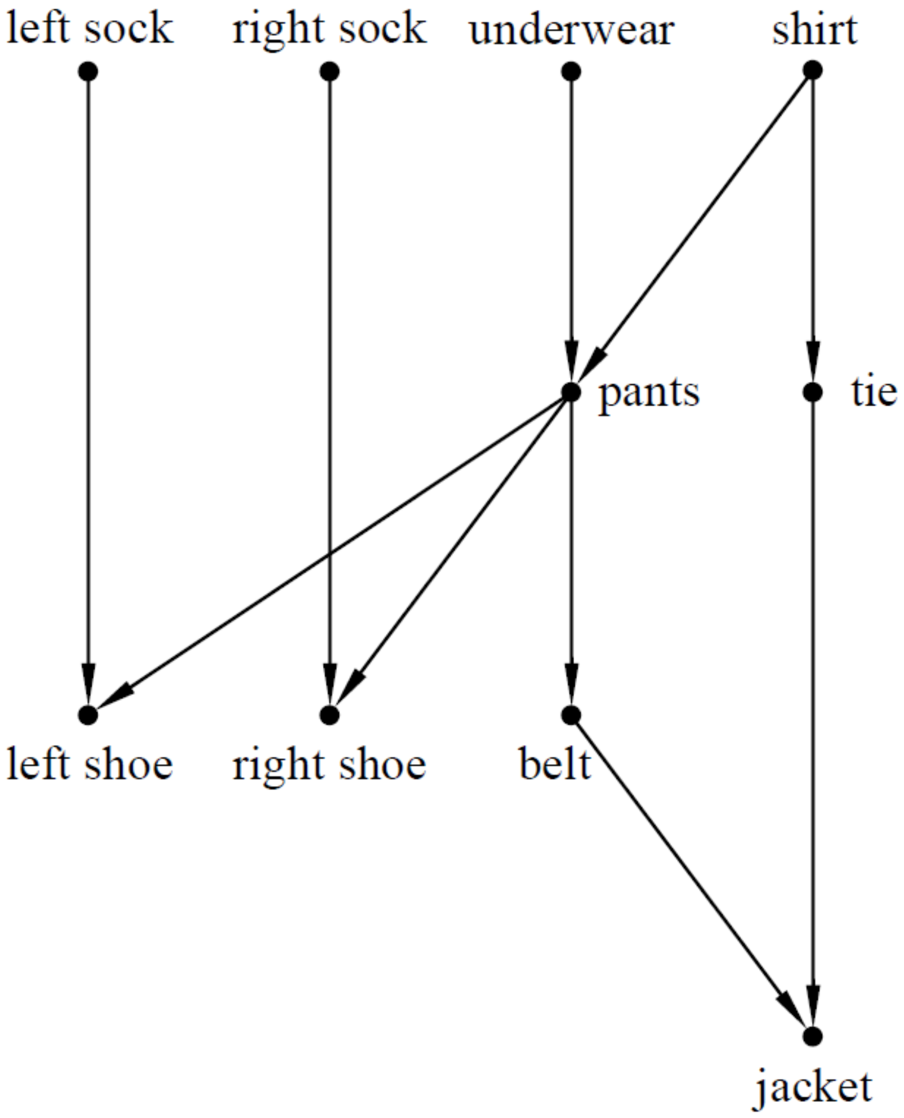
- Job stats let us predict overall system utilization
- CPU Utilization: $\frac{1}{4} + \frac{2}{6} + \frac{3}{8} = 95\% > 100\%$ ✓
  - Can meet all deadlines!

| Process | Arrival | Deadline | Time Period |
|---|---|---|---|
| $P_1$ | 1 | 4 | 4 |
| $P_2$ | 2 | 6 | 6 |
| $P_3$ | 3 | 8 | 8 |



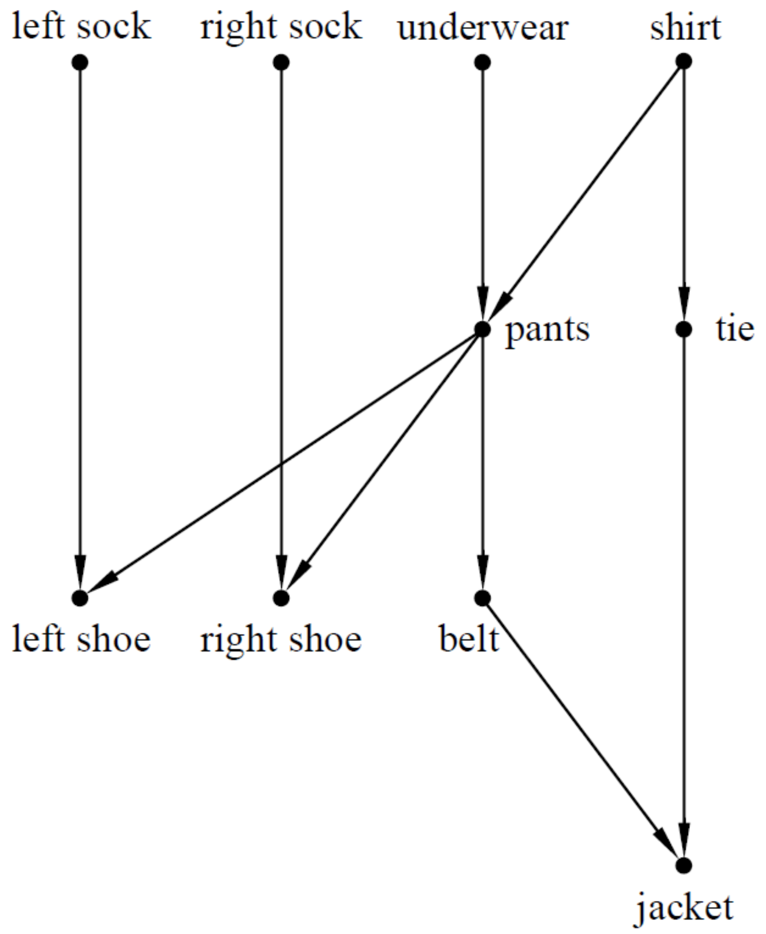
# TASK DEPENDENCIES

- What if tasks have dependencies between them?
- A **directed acyclic graph (DAG)** is a directed graph with no cycles.
- DAG is a useful concept in analyzing task scheduling and concurrency control.
- When distributing a program across multiple processors, we're in trouble if one part of the program needs an output that another part hasn't generated yet!
- A **topological sort** of a finite DAG is a list of all the vertices such that each vertex $\boldsymbol{v}$ appears earlier in the list than every other vertex reachable from $\boldsymbol{v}$.



# DIRECTED ACYCLIC GRAPHS & SCHEDULING

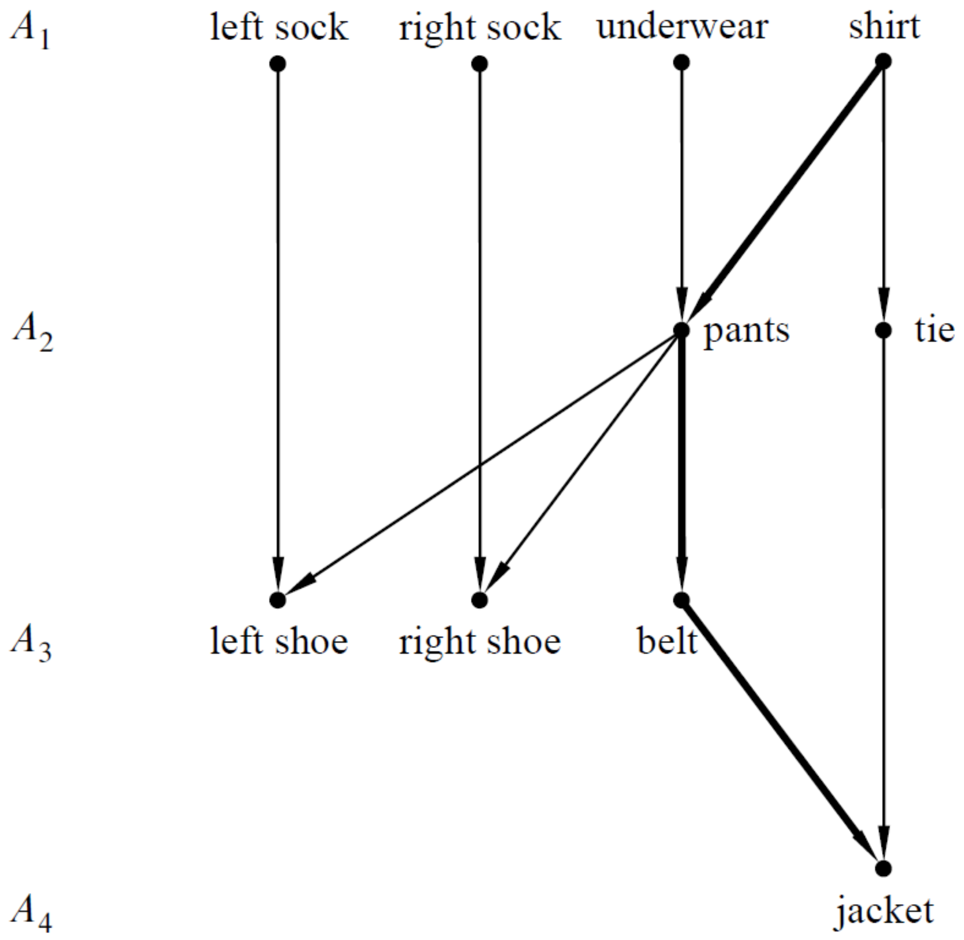
| (a) | (b) |
|---|---|
| underwear | left sock |
| shirt | shirt |
| pants | tie |
| belt | underwear |
| tie | right sock |
| jacket | pants |
| left sock | right shoe |
| right sock | belt |
| left shoe | jacket |
| right shoe | left shoe |



# DAG & PARALLELISM

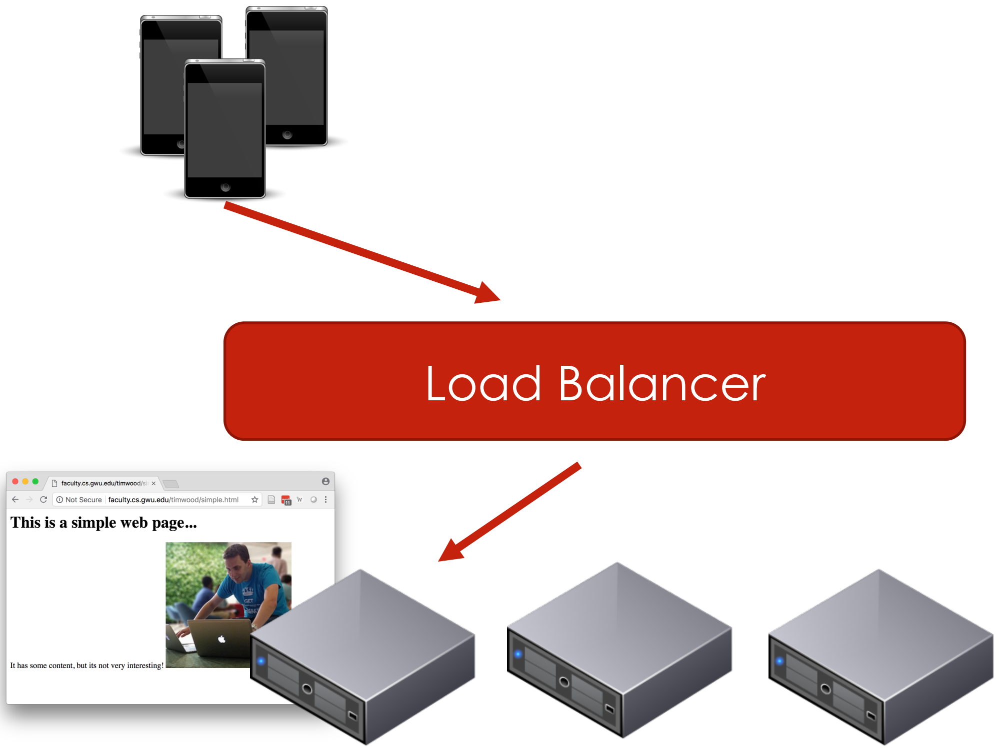
- The tasks in $A_i$ can be performed in step $i$ for $1 \le i \le 4$.
- A chain of 4 tasks (the critical path in this example) is shown with bold edges.
- The time it takes to schedule tasks, even with an unlimited number of processors, is at least as large as the number of vertices in any chain.
- A ***partition*** of a set A is a set of nonempty subsets of A called the ***blocks*** of the partition, such that every element of A is in exactly one block.
  - Ex: one possible partition of the set $\{a, b, c, d, e\}$ is:

    $\{a, c\}$ $\{b, e\}$ $\{d\}$
- A parallel schedule for a DAG, D, is a partition of $V(D)$ into blocks $A_0, A_1, \ldots$ such that when $j < k$ , no vertex in $A_j$ is reachable from any vertex in $A_k$



# 3. LOAD BALANCING

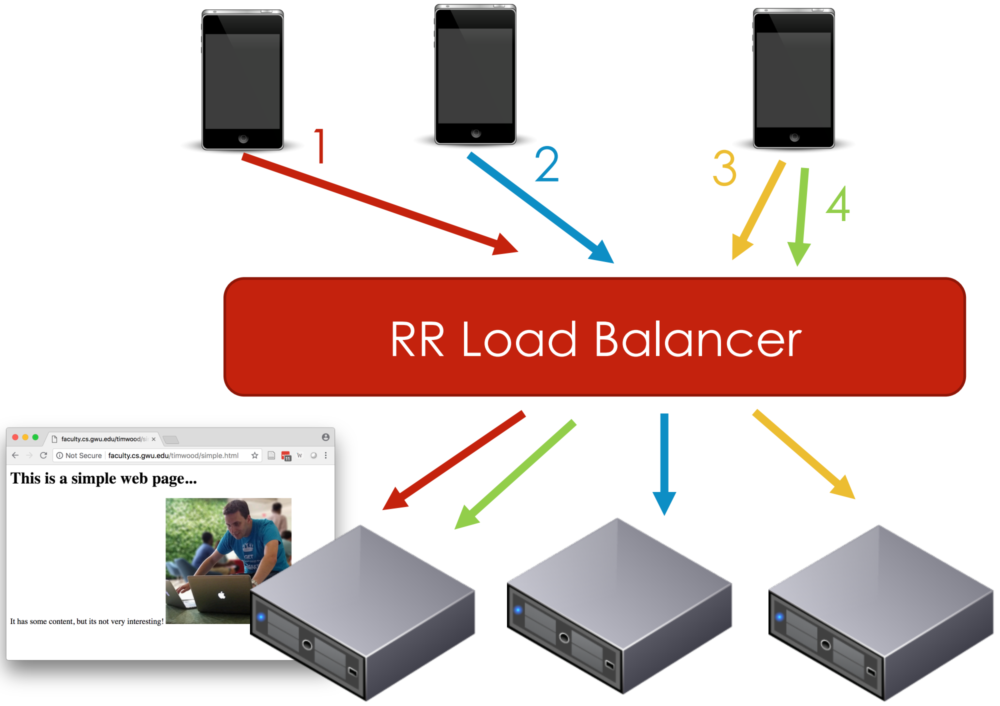
- What if tasks are arriving really really quickly?
    - Web requests arriving to Facebook – 100 Million requests per second!
- We need to **quickly** assign requests to a backend server
- We want to **evenly balance** the load across the servers



# ROUND ROBIN

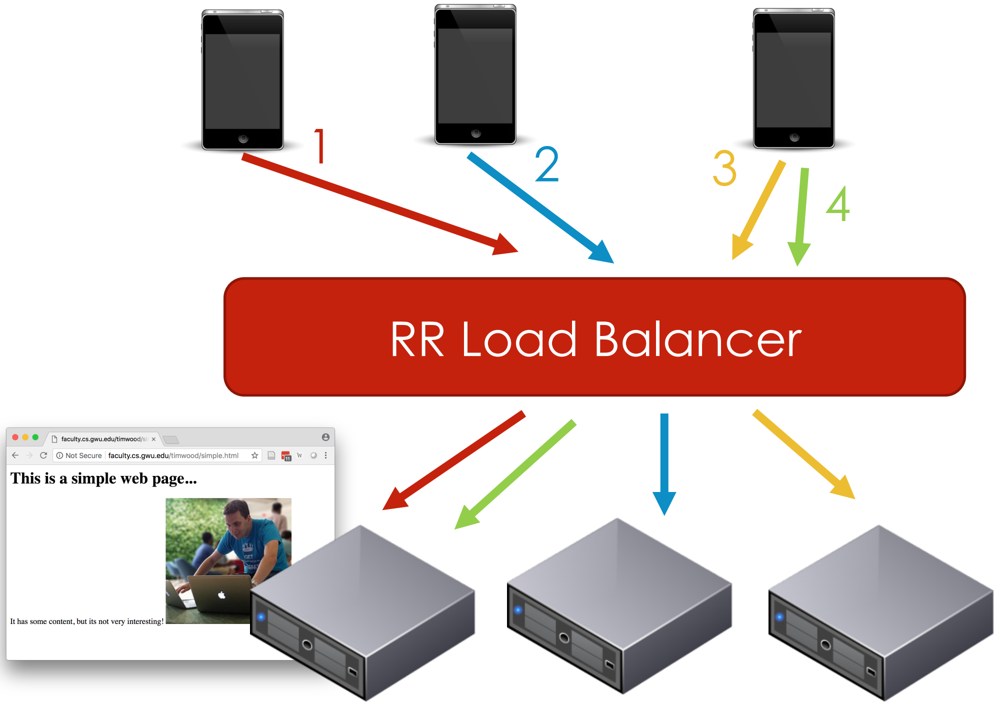
- Simplest load balancing policy
  - LB tracks where last request was sent
  - Send next request to next server in list
  - Loop back to first server
- Evenly distributes requests to servers

Pros/Cons?



# ROUND ROBIN

- Simplest load balancing policy
  - LB tracks where last request was sent
  - Send next request to next server in list
  - Loop back to first server
- Evenly distributes requests to servers
- Benefits:
  - Efficient to implement, low overhead
  - Number of requests is evenly balanced
- Issues:
  - What if servers are heterogeneous?
  - What if requests are heterogeneous?
  - No server affinity



# RANDOM

- **Even simpler** load balancing policy!
  - Round Robin requires **state** at the LB
  - Instead, just **randomly** assign each request to a server
  - If number of requests is high, load should be **approximately equal**
- Has similar pros/cons as RR
  - Can provide affinity if randomness is based on Src IP
- Weighted RR/ Weighted Random
  - Can purposefully skew requests based on server capacity

# JOIN THE SHORTEST QUEUE

- Send the request to the server with the **shortest queue** of requests
  - Load aware policy
- Sounds perfect!
- … what's the problem?

Pros/Cons?

# JOIN THE SHORTEST QUEUE

- Send the request to the server with the **shortest queue** of requests
  - Load aware policy
- Need to query servers to find out queue length
  - Solution: Randomly probe N servers
  - See “Power of Two Choices”
- Adds overhead in the critical path to check the queues
- What if there are multiple LBs?