6.5810: Tail latency + Cores that don't count

Adam Belay <abelay@mit.edu>

Logistics

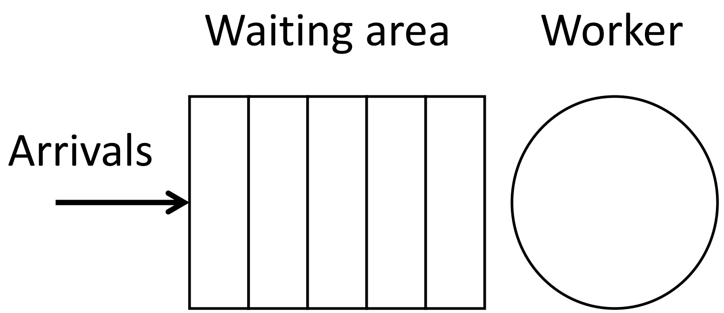
- Reminder: Send paper questions before each lecture!
  - This includes Wednesday’s lecture
- Signing up for paper presentations is past due
  - If you haven’t yet, please notify us ASAP

Agenda today

- High-level overview of queueing theory
- Tail latency
- Cores that don’t count

Kendall's notation

- A/S/c
  - A: Arrival process
  - S: Service time distribution
  - c: Number of workers



Some useful examples:

- M (Markovian):
  - Poisson process: Exponential interarrival; exponential service time
- D (Degenerate):
  - Deterministic: Fixed interarrival process; or fixed service time
- G (General)

Queueing disciplines

- The priority order that jobs in the queue are served
- Many disciplines are possible!
- Some examples: FIFO, PLIFO, PS, SRTF
- Kendall notation update -> A/S/c/D, where D is the discipline

Some terminology

- *Preemptive* -> can interrupt the worker to switch jobs
- *Work conserving* -> the worker is always busy if there are jobs

Q: What minimizes average completion time?

- i.e., minimize the sum of completions time -> $1||\sum C_j$

Q: What minimizes average completion time?

- i.e., minimize the sum of completions time -> $1||\sum C_\mathrm{j}$
- SRTF does! There is a proof!

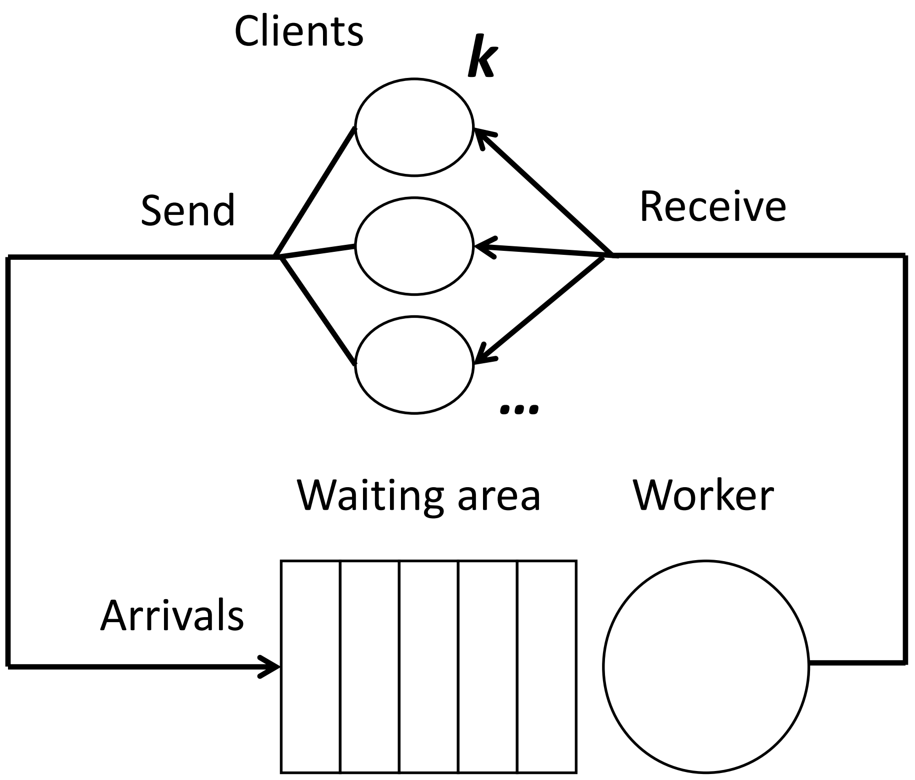
Q: What minimizes tail completion time?

Q: What minimizes tail completion time?

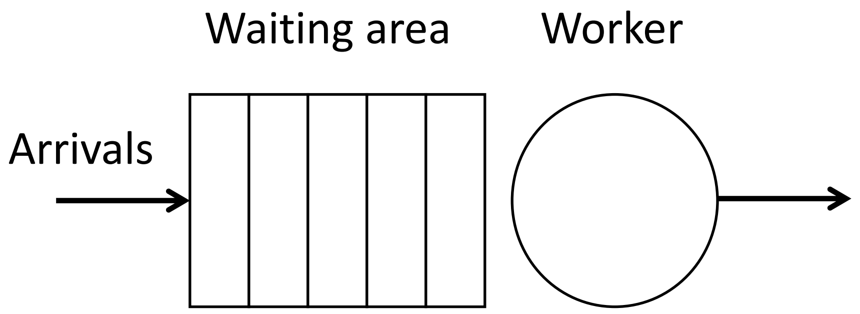
- In general, no single non-learning policy can
- However, if you know the service time behavior…
- **Light tailed**: <= exponential distribution
  - Intuition: Finish the heavy requests fast, they determine the tail
  - Optimal discipline: FIFO
- **Heavy tailed:** > exponential distribution
  - E.g., log-normal and pareto distribution
  - Intuition: A heavy request will take so long that it’s better for the tail to handle another request instead
  - Optimal discipline: SRPT, PS, etc.

Closed vs. open queueing systems

Closed

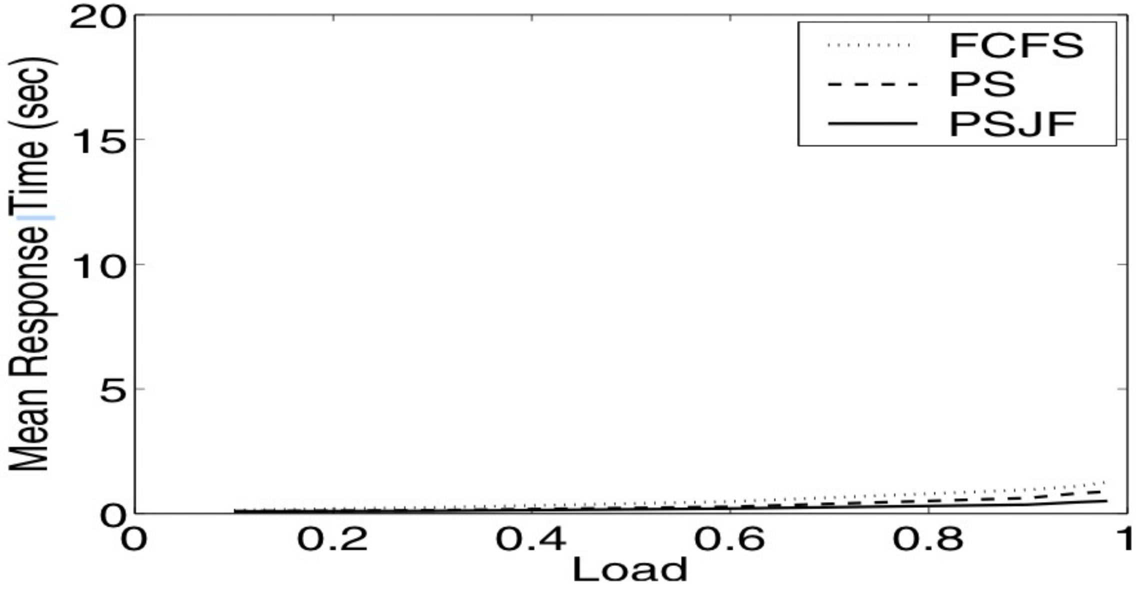


Open

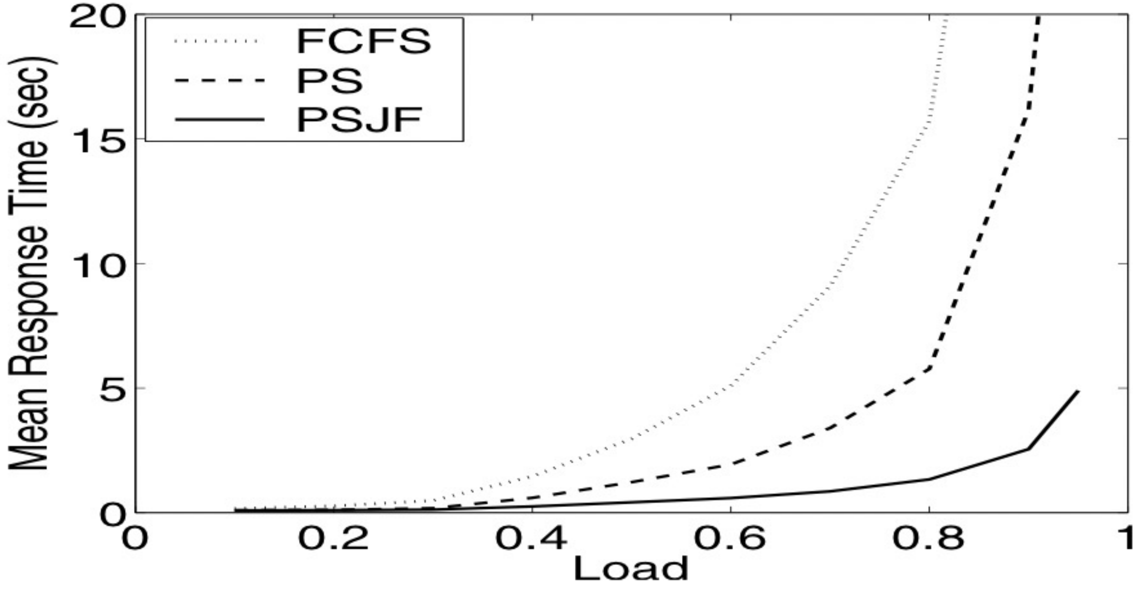


Closed vs. open queueing systems

Closed

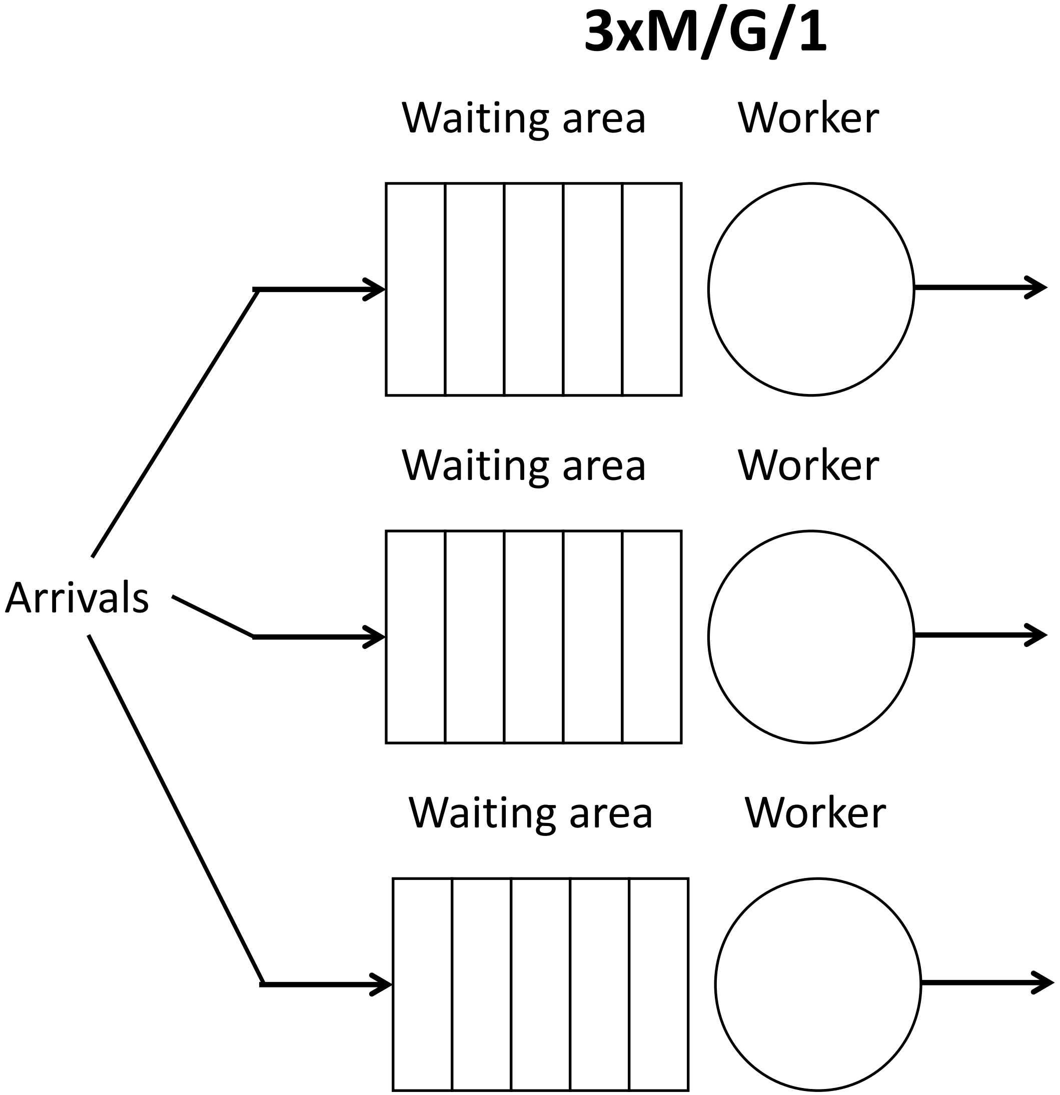


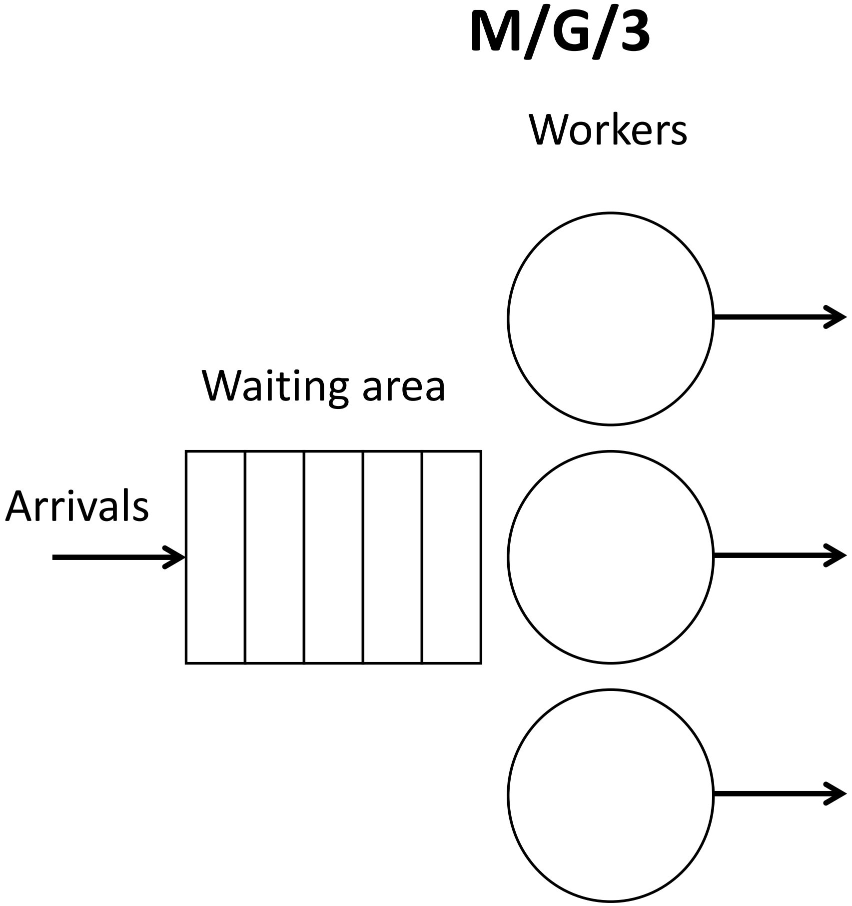
Open



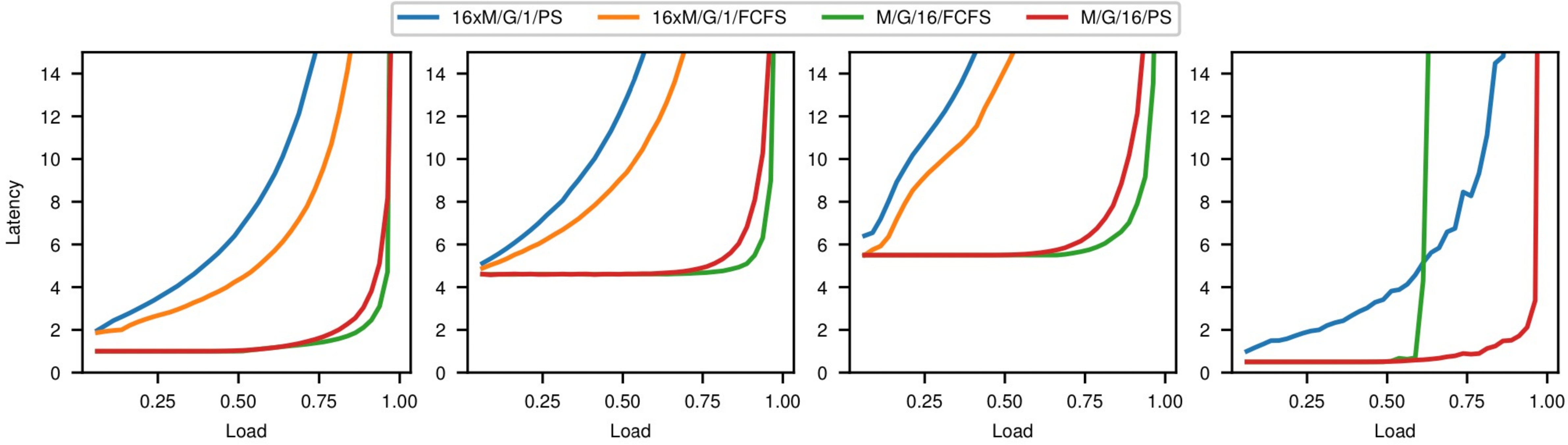
Open Versus Closed: A Cautionary Tale. Schroeder et. Al. NSDI’06

Q: Which is better?





Simulating queueing policies



(a) Deterministic (b) Exponential (c) Bimodal-1 (d) Bimodal-2

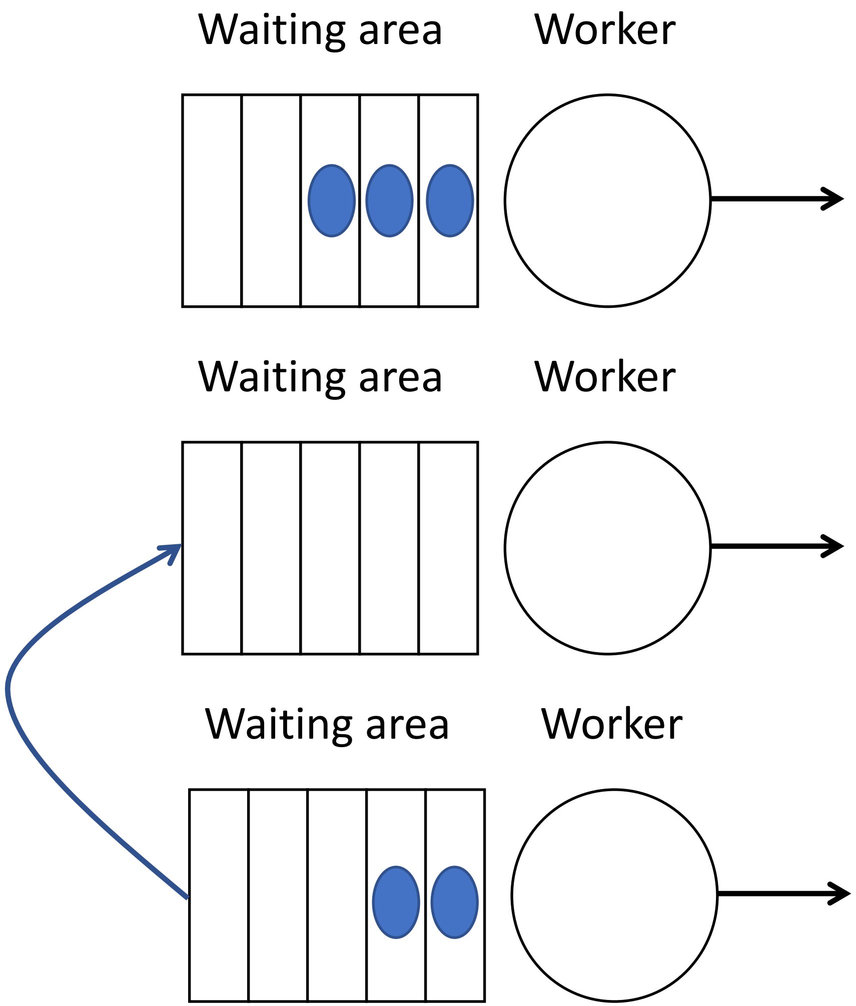
Figure 2: Simulation results for the 99^{th} percentile tail latency for four service time distributions with $\bar{S} = 1$.

ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. Prekas et. Al. SOSP'17

Problem: Modern CPUs have tons of cores

- Hashing and spreading work across cores has poor tail behavior
- But using a centralized queue has high synchronization overhead
- What should we do?

Solution: Work stealing



1. Workers with empty queues search for work (at random) in other worker's queues
2. Then it steals half the work

Approximates M/G/n, but low overhead!

Don't do work shedding!

Formal proof that work stealing has optimal messaging costs; intuition -> only idle cores send messages

See "Scheduling Multithreaded Computations by Work Stealing" Blumofe et. Al.

Debate: Is low tail latency achievable at scale?

How can we make systems tail tolerant?

- **Hedged requests:** Issue the same request to multiple replicas, use the result from whichever responds first
  - Can wait e.g., 95% of typical response time before issuing backup request to reduce resource use, but this also reduces benefit
- **Tied requests:** Issue both requests… When one replica finishes, cancel the other request (if it hasn't started yet)

Challenge overall: Extra resource consumption, through replication + request handling and cancelling

And overhead can't be reduced for very short requests.

Fail-stop processors

- “A *fail-stop processor* never performs an erroneous state transformation due to a failure. Instead, the processor halts and its state is irretrievably lost.” – Fred Schneider
- Today: What happens when processors are *not* fail-stop?

Silent data corruption (SDC)

- Well known problem; happens at scale
- Historically cosmic rays may have been the predominant cause
  - e.g., roughly one error per month in 256 MB RAM
  - But this paper reveals the issue is more complex now

Mercurial cores

- Cores with defects not detected during manufacturing
- A few cores per several thousand machines
- Cannot necessarily be mitigated by microcode updates
- May be associated with specific components; typically, specific cores
- Silent data corruption: Only symptom is erroneous computation

Why now?

- Authors speculate smaller feature sizes + more complexity in chip design are pushing up error rates
- Normally chip designers formally verify components are correct
- Some mercurial cores may only start to misbehave after they age

Some examples

- Violations of lock semantics
- Data corruption on load, store, vector, or coherence operations
- Deterministic AES mis-computation
- Corruption during garbage collection
- Database index corruption on some cores but not others
- Repeated bit-flips in strings at a particular position
- Corruption of kernel state

Why are compute errors different?

- Disks lose blocks all the time! Very unreliable overall
- But with disks or networks the “right answer” is obvious
  - It’s the identity of the data you’re storing or transporting
  - That means checksums and coding-based techniques work great

Why is this happening?

- Steady increase in complexity
- Nanometer CPU features leave smaller margin for error
- CPUs are transforming into sets of discrete accelerators around a shared register file
  - This increases the surface of behaviors to verify
- Sometimes strongly frequency sensitive, sometimes not
- Sometimes lower frequencies trigger errors (b.c. voltage changes too)

Debate… could some mercurial errors be SW?

- How can we avoid false positives?

Conclusion

- Bridging theory and practice can yield systems with great tail behavior
- But systems must still be *tail tolerant*, i.e., tolerant to high tail latency
- CPUs are becoming less reliable as they get more complex and smaller; new systems challenges emerging