System Performance and Highly Concurrent Systems (2)

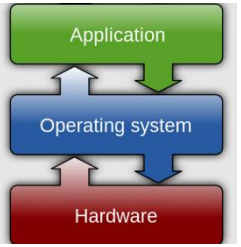
Lecture 13

Hartmut Kaiser

https://teaching.hkaiser.org/spring2026/csc4103/

Goals for Today

- Finish up discussion of highly concurrent systems
- Start exploring OS memory management (if time permits)



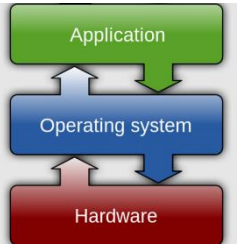
Recall: Little's Law

- The number of "things" in a system is equal to the bandwidth times the latency (on average)

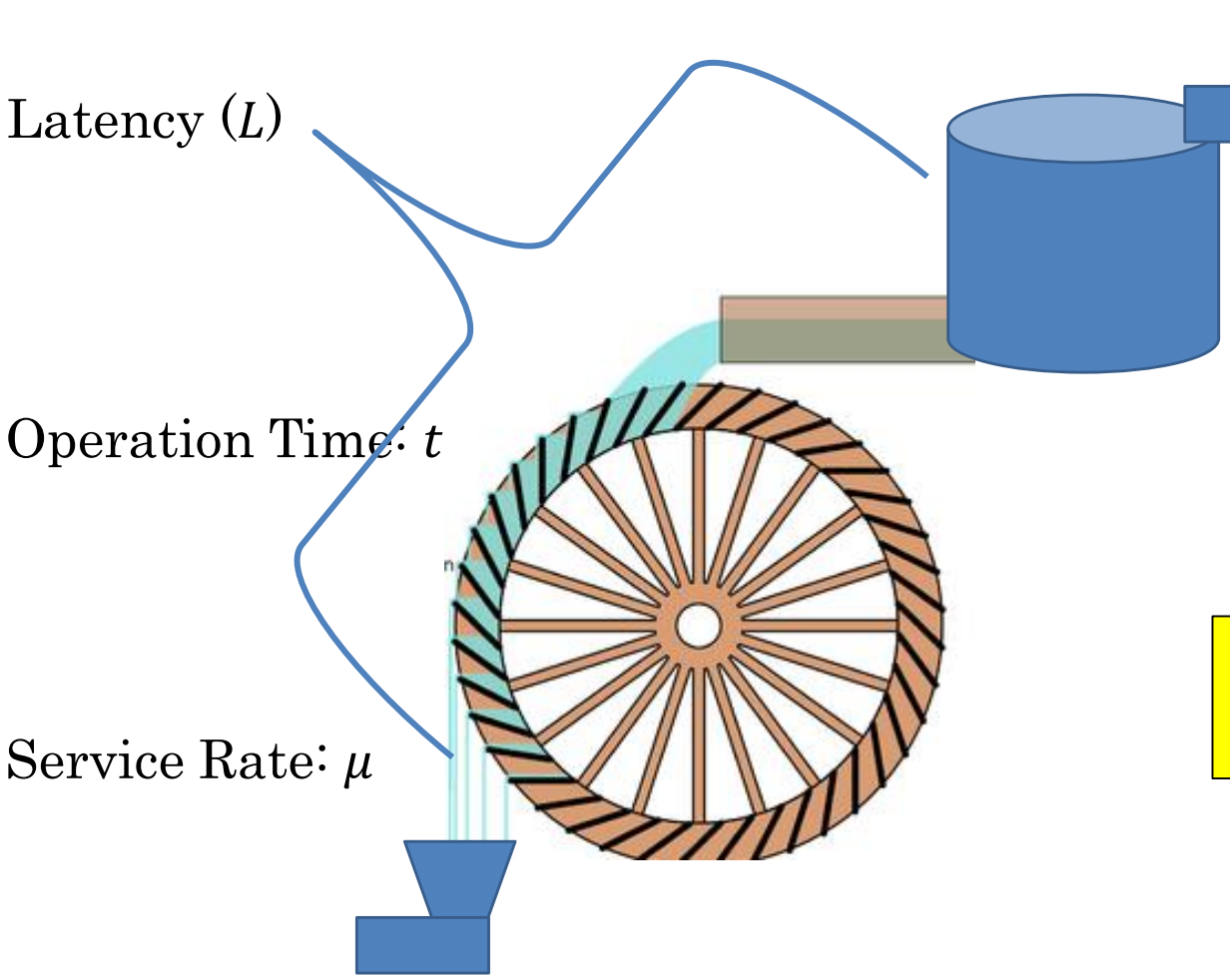
$$n = L\,B$$

- Applies to any stable system (arrival rate = departure rate)

- Can be applied to an entire system:
  - Including the queues, the processing stages, parallelism, whatever
- Or to just one processing stage:
  - i.e., disk I/O subsystem, queue leading into a CPU or I/O stage, …

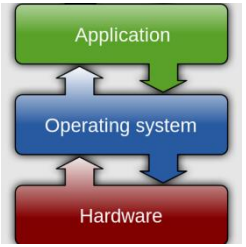


Recall: A Simple System Performance Model



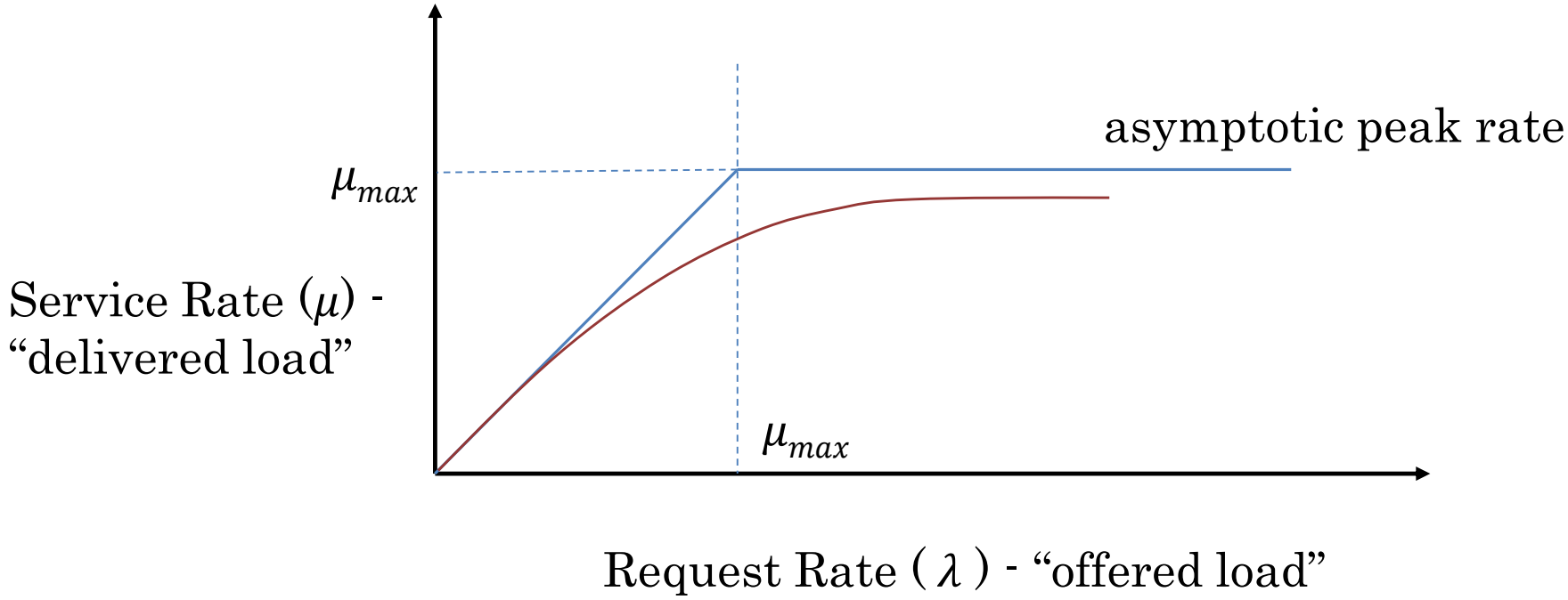
The maximum service rate μ_{max} is a property of the system – the "bottleneck"

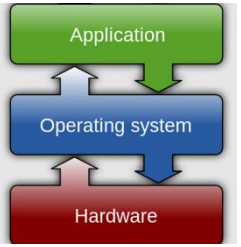
Utilization: $\rho = \frac{\lambda}{\mu_{max}}$



Recall: Ideal System Performance

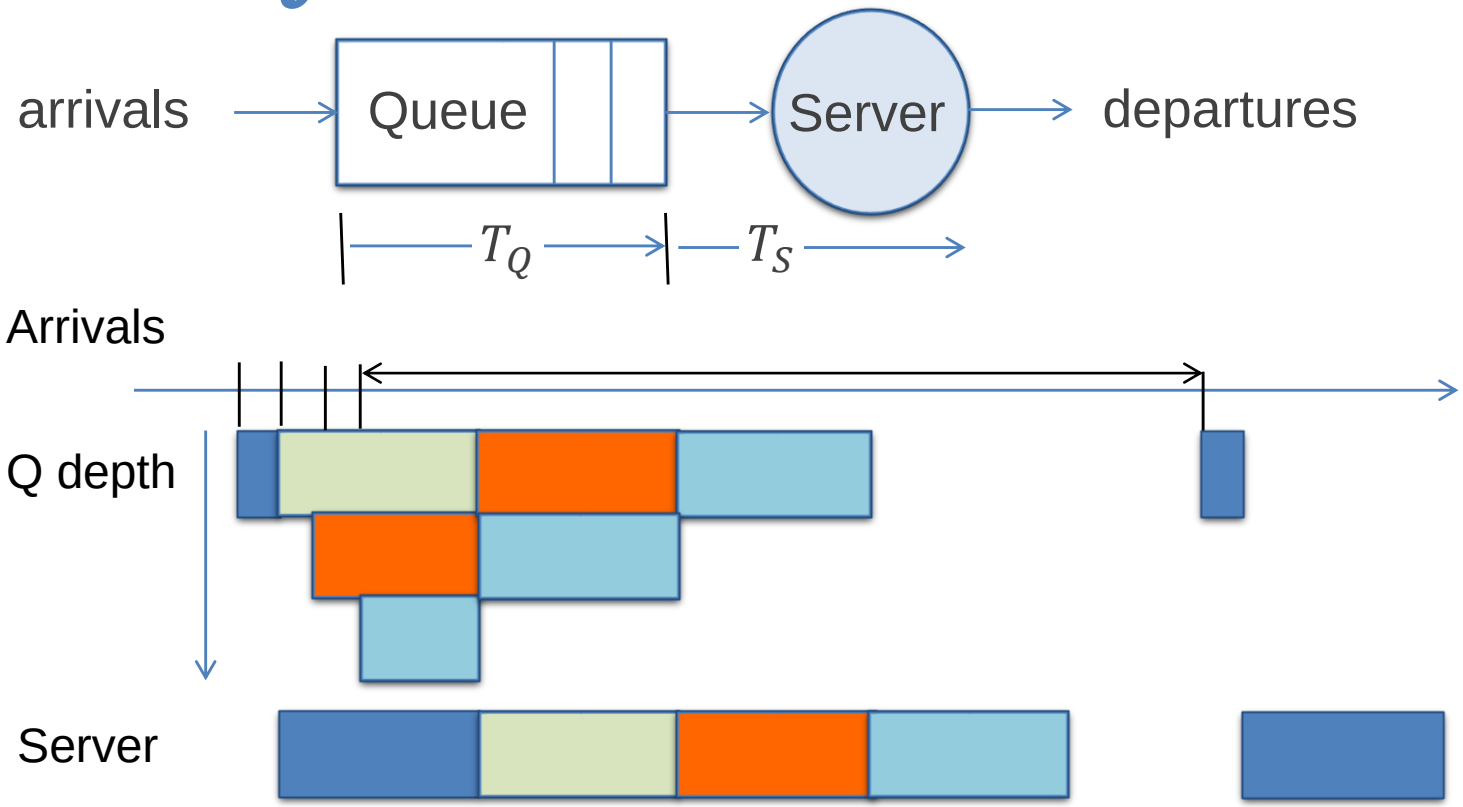
- How does μ (service rate) vary with λ (request rate)?



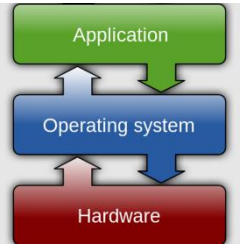


Recall: A Bursty World

- T_A: time between arrivals
  - Now, a random variable
- T_S: service time
  - $\mu = {}^{k}/_{T_S}$
- T_Q: queuing time
  - $L = T_Q + T_S$



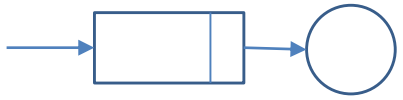
- Requests arrive in a burst, must queue up until served
- Same average arrival time, but almost all of the requests experience large queue delays (even though average utilization is low)



Recall: Little's Law Applied to a Queue

- Before, we had $n = LB$ (for a stable system):
  - B: bandwidth
  - L: latency
  - n: number of operations in the system

- When applied to a queue, we get:

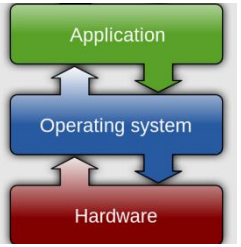


$$L_Q = \lambda T_Q$$

Average Arrival Rate

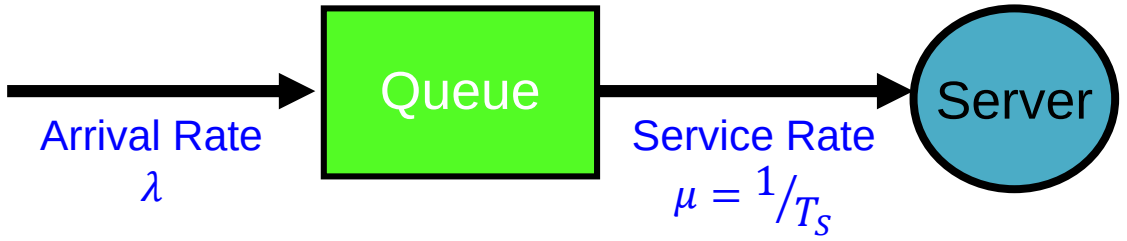
Average length of the queue

Average time "waiting"

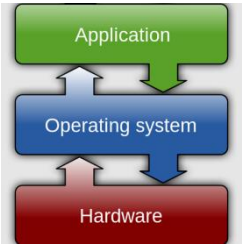


Recall: Some Results from Queuing Theory

- Assumptions: system in equilibrium, no limit to the queue, time between successive arrivals is random and memoryless



- λ: arrival rate
- T_S: mean time to service a customer
- C: squared coefficient of variance $({}^{\sigma^2}/_{T_S^2})$
- μ: service rate $({}^{1}/_{T_S})$
- ρ: utilization $({}^{\lambda}/_{\mu})$



Recall: Some Results from Queuing Theory

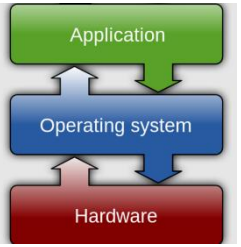
- Memoryless service distribution ($C = 1$) - an "M/M/1 queue":

$$T_Q = \frac{\rho}{1-\rho} \cdot T_S$$

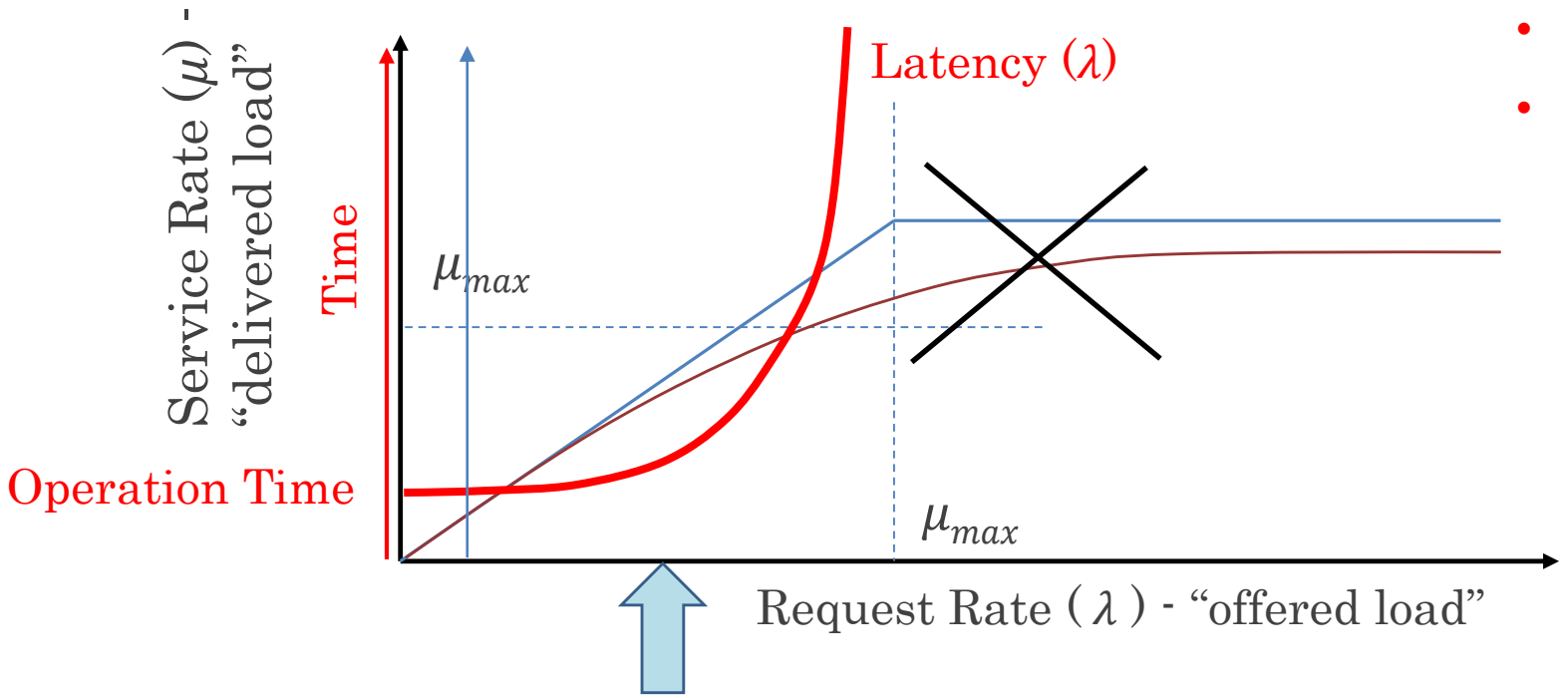
- General service distribution (no restrictions) - an "M/G/1 queue":

$$T_Q = \frac{1+C}{2} \cdot \frac{\rho}{1-\rho} \cdot T_S$$

- λ: arrival rate
- T_S: mean time to service a customer
- C: squared coefficient of variance (σ^2/T_S^2)
- μ: service rate ($1/T_S$)
- ρ: utilization (λ/μ)



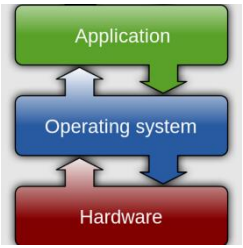
Recall: Ideal System Performance



- $T_Q \sim \frac{\rho}{1-\rho}$, $\rho = {}^{\lambda}/_{\mu_{max}}$
- Why does latency blow up as we approach 100% utilization?
  - Queue builds up on each burst
  - But very rarely (or never) gets a chance to drain

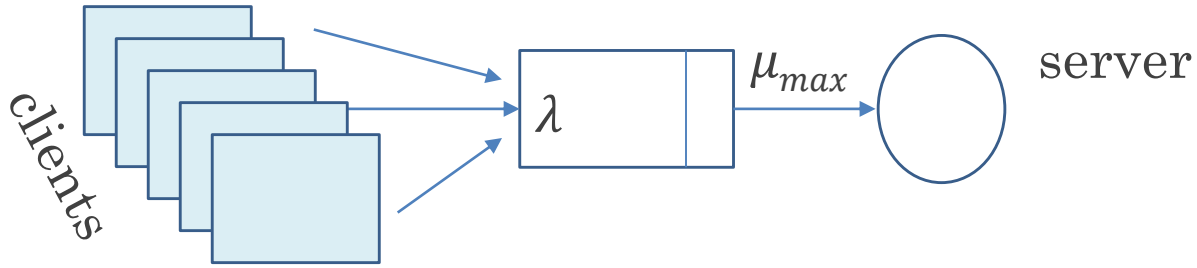
"Half-Power Point": load at which system delivers half of peak performance

- Design and provision systems to operate roughly in this regime
- Latency low and predictable, utilization good: ~50%

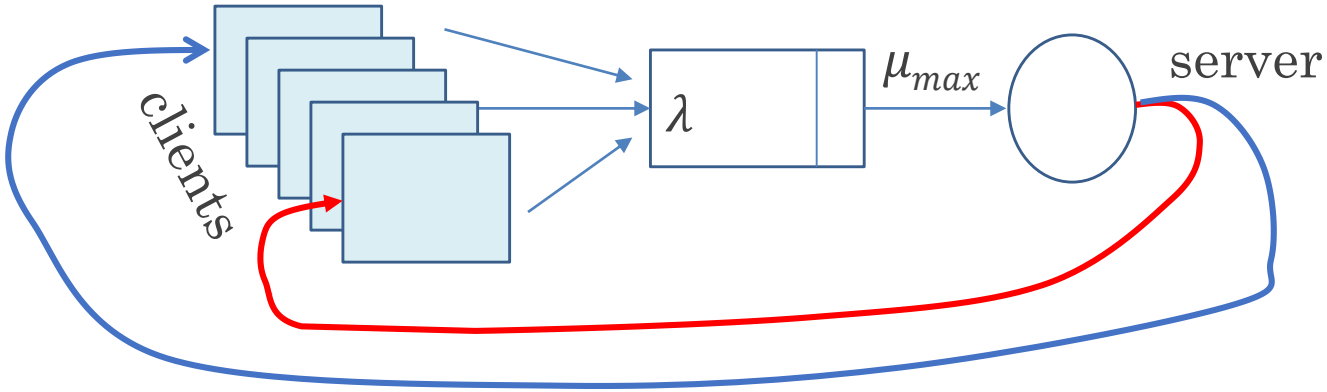


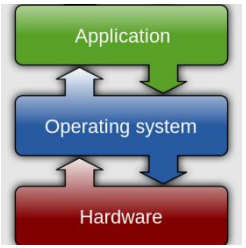
Do real systems really hit a wall as utilization approaches 100%?

Open System



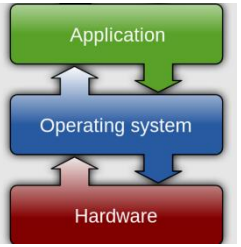
Closed System





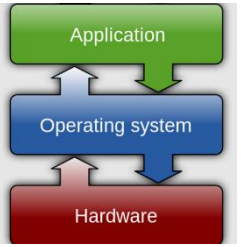
Closed System

- Clients generating the load depend on completion of previous requests
  - Request-response protocols
  - Humans in-the-loop waiting for results
- Model of client: {request, wait}+ repeat
  - Request rate determined by length of wait
- In closed system, wait time depends on response time (latency = operation time + queuing delay)
- As system saturates (utilization → 100%) delay increases, request rate is limited by service rate
  - Queueing smooths bursts, but does not grow unbounded due to rate mismatch

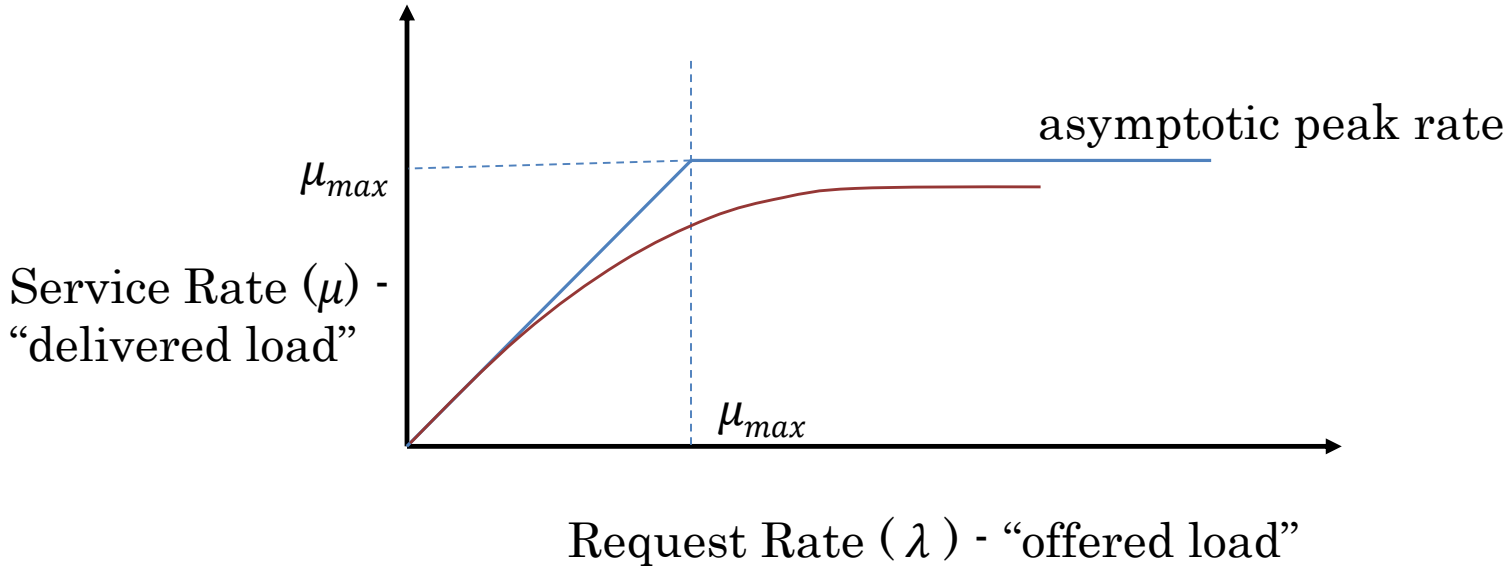


What Causes Systems to Close?

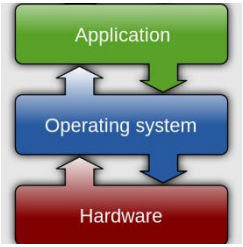
- Protocols are designed to have self-limited behavior
  - Request-response, bounded number of outstanding requests per client
- Underlying system induces "back pressure" even if higher level services and applications don't
  - Bounded size queues (not just because of memory size)
  - What happens when it fills up?



Recall: Well-Conditioned Systems



- A system that behaves this way is well-conditioned
  - Delivered load increases with offered load until pipeline saturates
  - As offered load increases further, throughput remains high



Recall: Non-Well-Conditioned Systems

- A server that spawns a new pthread per request is not well-conditioned!

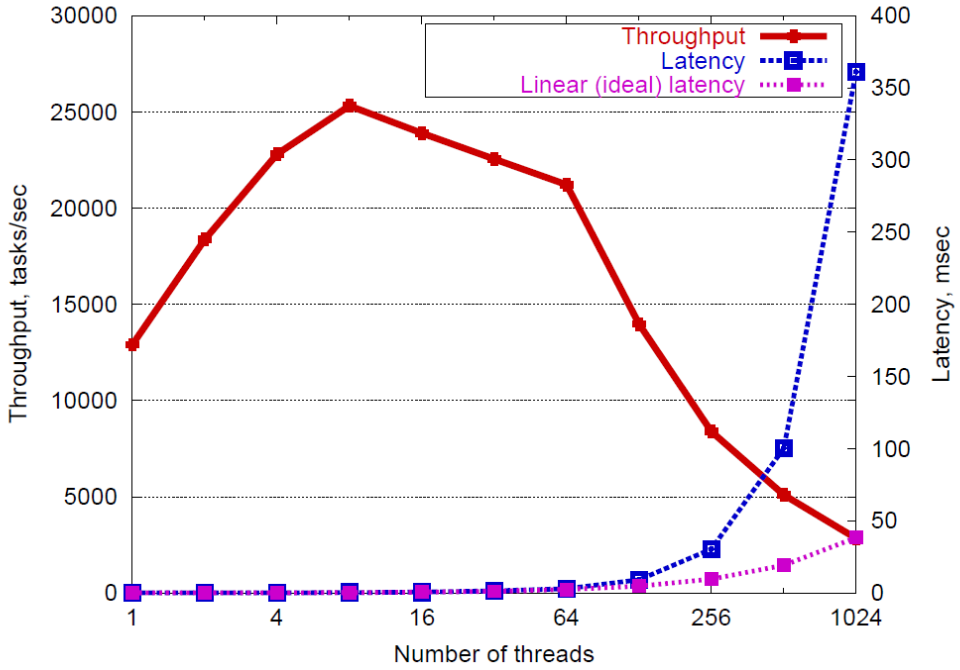
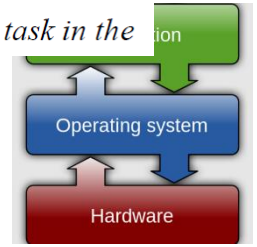


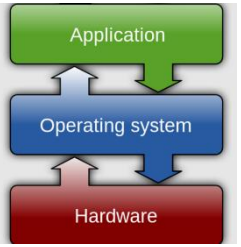
Figure 2: **Threaded server throughput degradation:** *This benchmark measures a simple threaded server which creates a single thread for each task in the*

- Figure from SEDA Section 2 reading (Welsh 2001)



Building Well-Conditioned Systems

- Spawning a new thread or process for each request is not well-conditioned
- Too many threads is bad
  - Scheduling overhead becomes large
  - Context switch overhead becomes large
    - E.g., Poor cache performance
  - Synchronization overhead becomes large
    - E.g., Lock contention
- Was our original (v1) server well-conditioned?
  - The one that handles requests one at a time, with no concurrency?

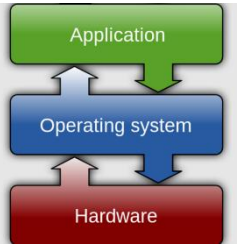


System Performance and Highly Concurrent Systems

Part 2

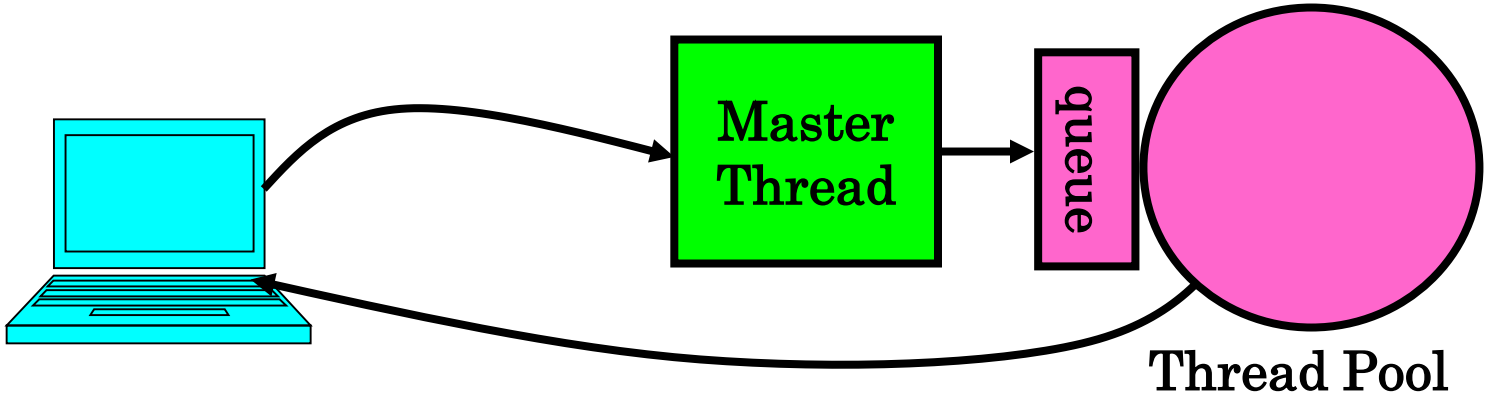
Concurrent, Well-Conditioned Systems

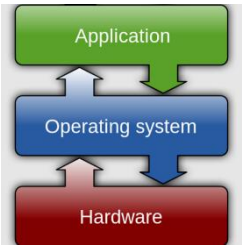
- Thread Pools
- User-Mode Threads
- Event-Driven Execution



Thread Pools

- Key idea: limit the number of threads
  - Before throughput starts to degrade
- Instead, allocate a bounded "pool" of worker threads, representing the maximum level of multiprogramming

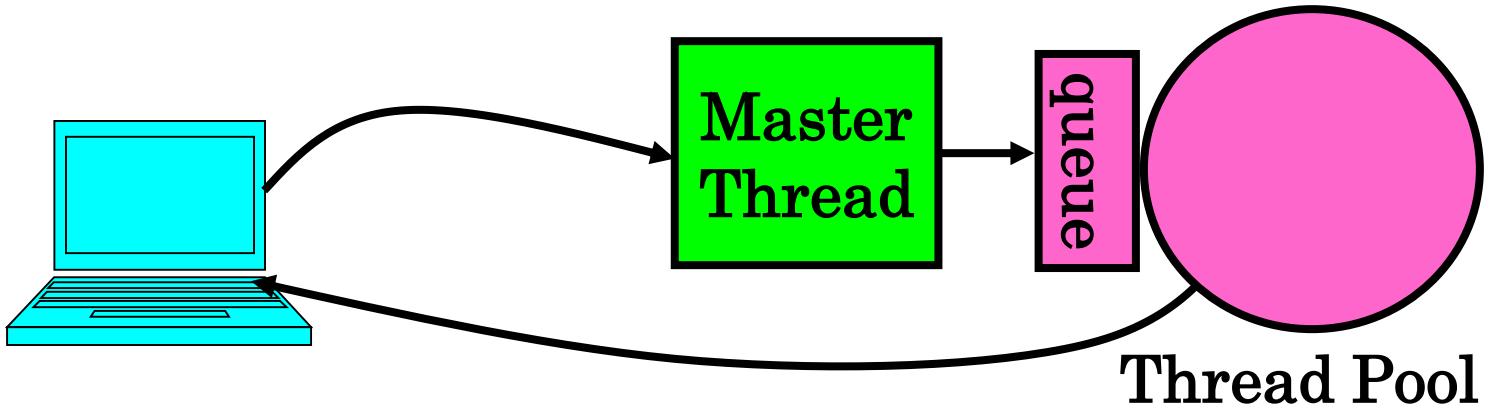


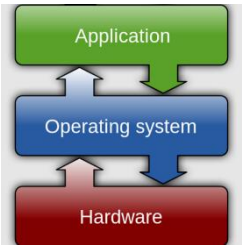


Thread Pools

```
master() {
  allocThreads(worker, queue);
  while(TRUE) {
    con = AcceptCon();
    // Blocks if full
    Enqueue(queue, con);
  }
}
```

```
worker(queue) {
  while(TRUE) {
    // Blocks if empty
    con = Dequeue(queue);
    ServiceWebPage(con);
  }
}
```



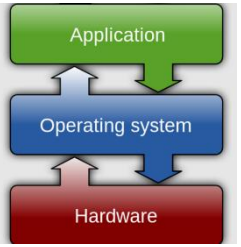


We've Looked At: Kernel-Supported Threads

- Thread pools work well, but they somewhat limit concurrency
  - Why?

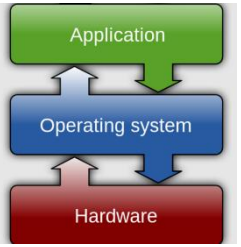
  - Threads run and block (e.g., on I/O) independently
  - One process may have multiple threads waiting on different things
  - Two mode switches for every context switch (expensive)
  - Create threads with syscalls

- Is there a good alternative?
  - Multiplex several streams of execution (at user level) on top of a single OS thread
  - E.g., Java, Go, … (and many many user-level threads libraries before it)



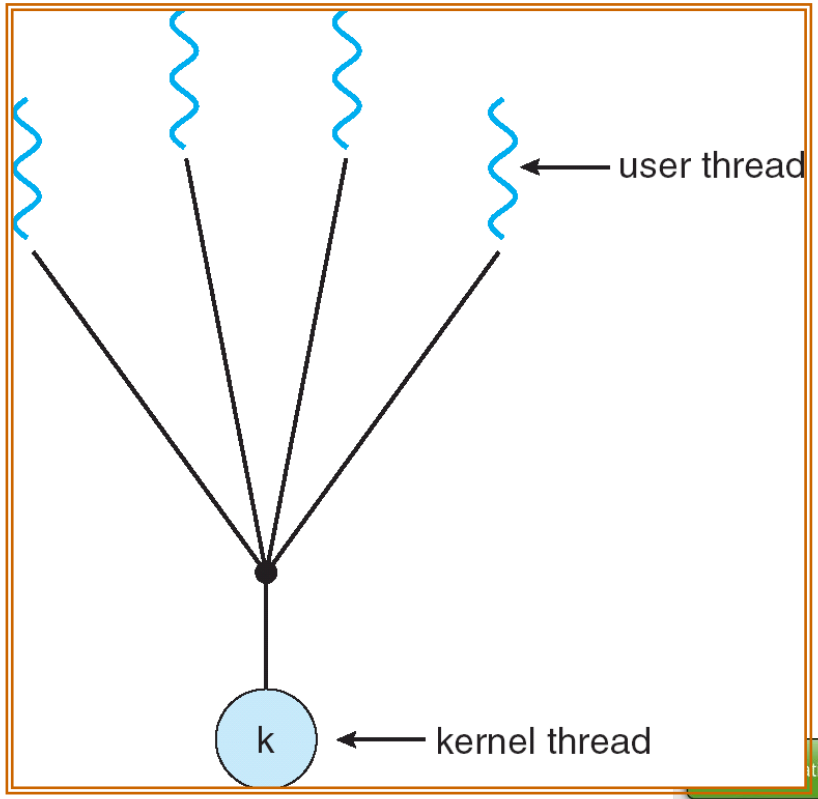
Concurrent, Well-Conditioned Systems

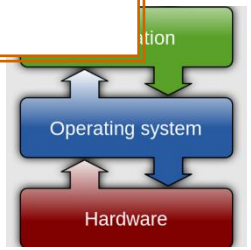
- Thread Pools
- User-Mode Threads
- Event-Driven Execution



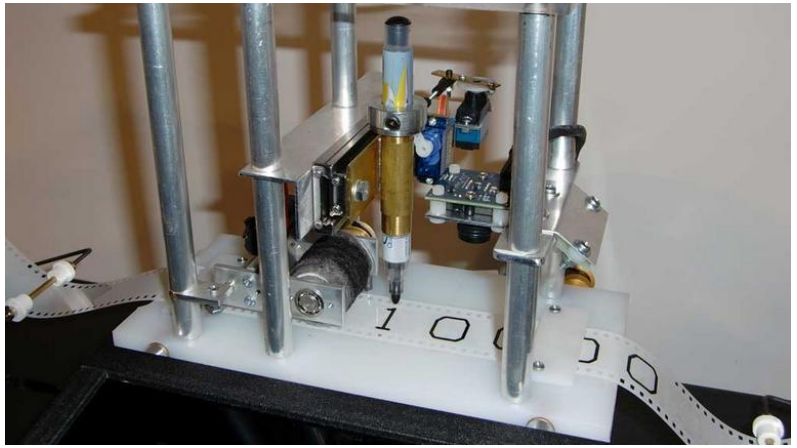
User-Mode Threads

- User program contains its own scheduler
- Several user threads per kernel thread
- User threads may be scheduled non-preemptively
  - Only switch on yield
- Context switches cheaper
  - Copy registers and jump (switch in userspace)

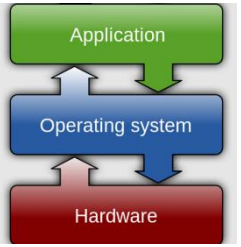




Recall: What's a 'Thread'?

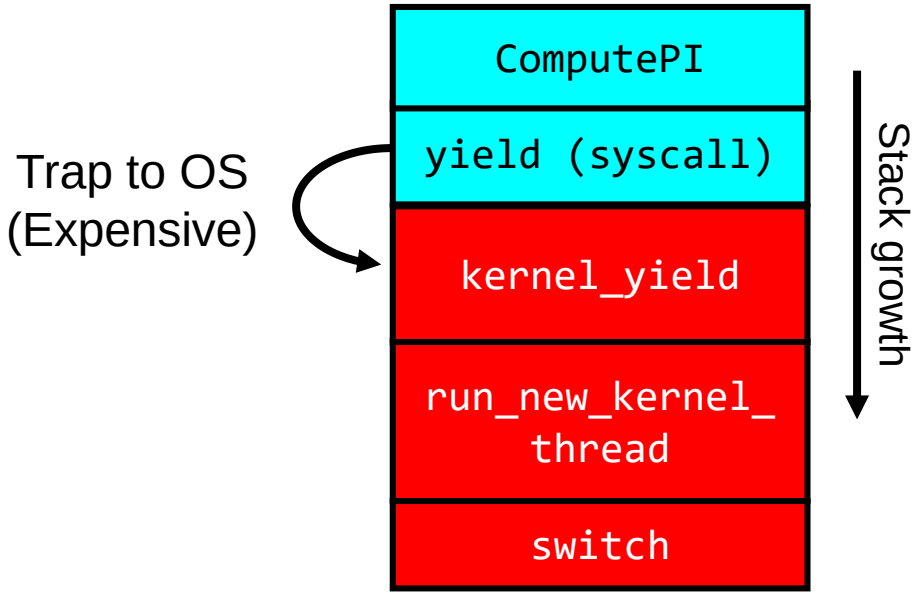


- An entity which exposes 4 properties:
  - A single flow of control (sequence of op-codes)
  - A program counter marking what's currently being executed
  - An associated execution context (stack, register set, static and dynamic memory, thread local variables, etc.)
  - A state (initialized, pending, suspended, terminated, etc.)

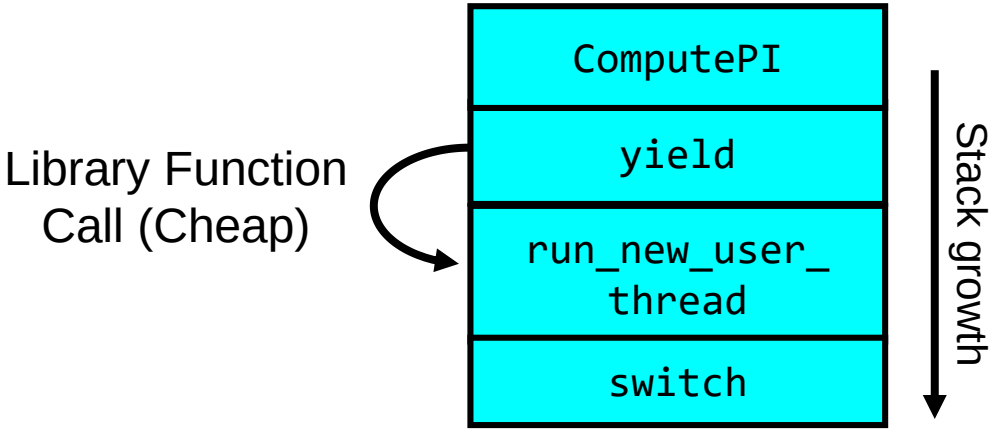


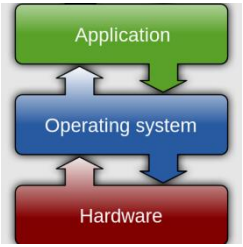
Thread Yield

Kernel-Supported Threads



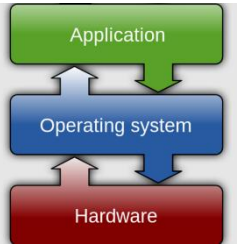
User-Mode Threads





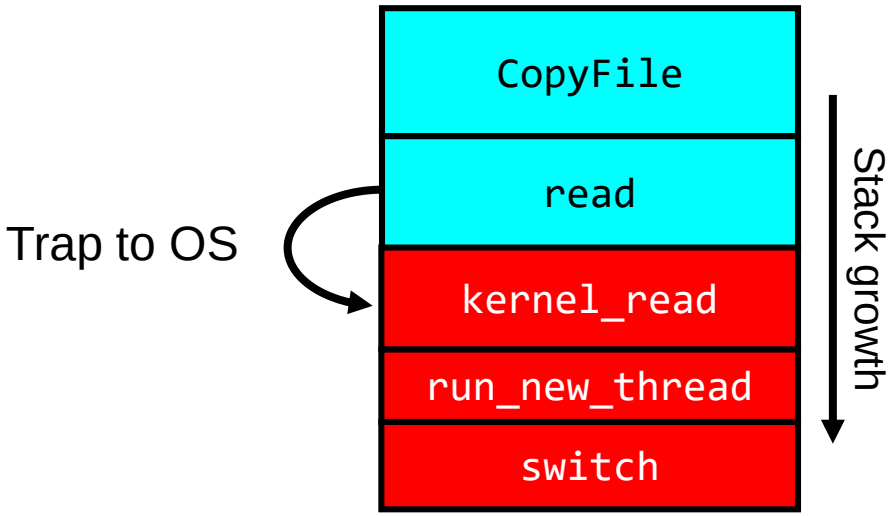
User-Mode Threads: Problems

- One user-level thread blocks on I/O: they all do
  - Kernel cannot adjust scheduling among threads it doesn't know about
- Multiple Cores?
  - Need mechanisms like work-stealing
- Can't completely avoid blocking (syscalls, page fault)
- One Solution: Scheduler Activations
  - Have kernel inform user-level scheduler when a thread blocks
  - Evolving the contract between OS and application
  - Not all OS support this (actually close to none)
- Alternative Solution: Language Support?
  - Make the scheduler aware of the blocking operation

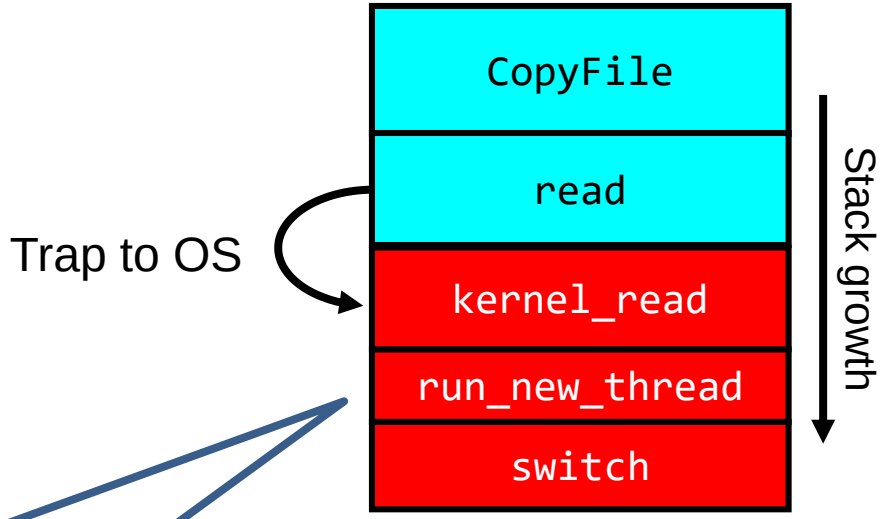


Thread I/O

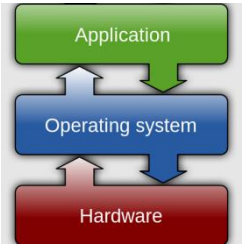
Kernel-Supported Threads



User-Mode Threads



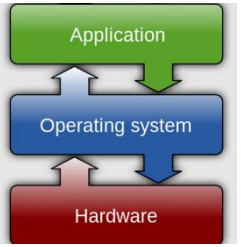
- Selects a new *kernel thread* to run
- Bypassing user-level scheduler



Library Solutions: Example

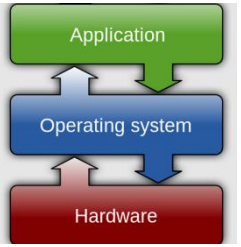
- HPX – A C++ Asynchronous Many-task Runtime System
  - A library linking to your application

- At it's heart, HPX is a very efficient (user-level) threading implementation
- Several functional layers are implemented on top:
  - C++ standards-conforming API exposing everything related to parallelism and concurrency
  - Full set of C++17/C++20/C++23 (parallel) algorithms
  - Full set of senders/receivers (currently being discussed for standardization)
    - Implemented using C++17
  - Distributed operation
    - Extending the standard interfaces for use on tightly coupled clusters (super-computers)
    - Global address space, load balancing, uniform API for local and remote operations



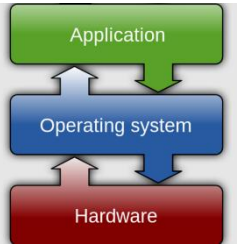
Go Goroutines

- Goroutines are lightweight, user-level threads
  - Scheduling not preemptive (relies on goroutines to yield)
  - Yield statements inserted by compiler
- Advantages relative to regular threads (e.g., pthreads)
  - More lightweight
  - Faster context-switch time
- Disadvantages
  - Less sophisticated scheduling at the user-level
  - OS is not aware of user-level threads



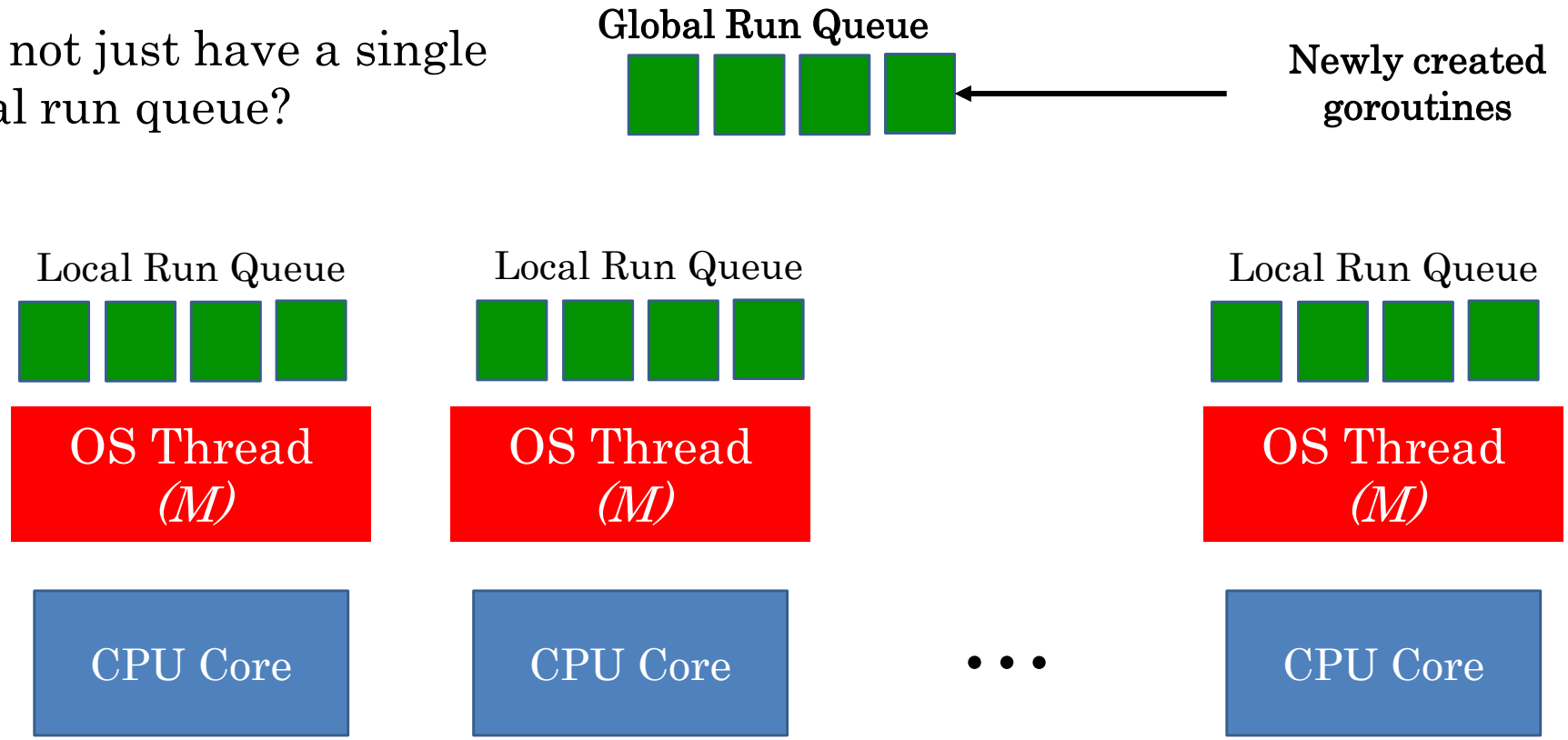
Go User-Level Scheduler

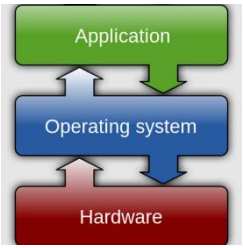
- Why this approach?
- 1 OS (kernel-supported) thread per CPU core: allows Go program to achieve parallelism not just concurrency
  - Fewer OS threads? Not utilizing all CPUs
  - More OS threads? No additional benefit
    - We'll see one exception to this involving syscalls
- Keep each (kernel-supported) bound to a core
- Keep goroutine on same OS thread: affinity, nice for caching and performance



Go User-Level Thread Scheduler

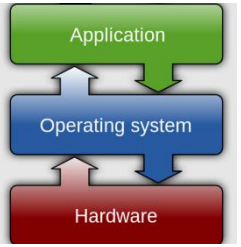
- Why not just have a single global run queue?



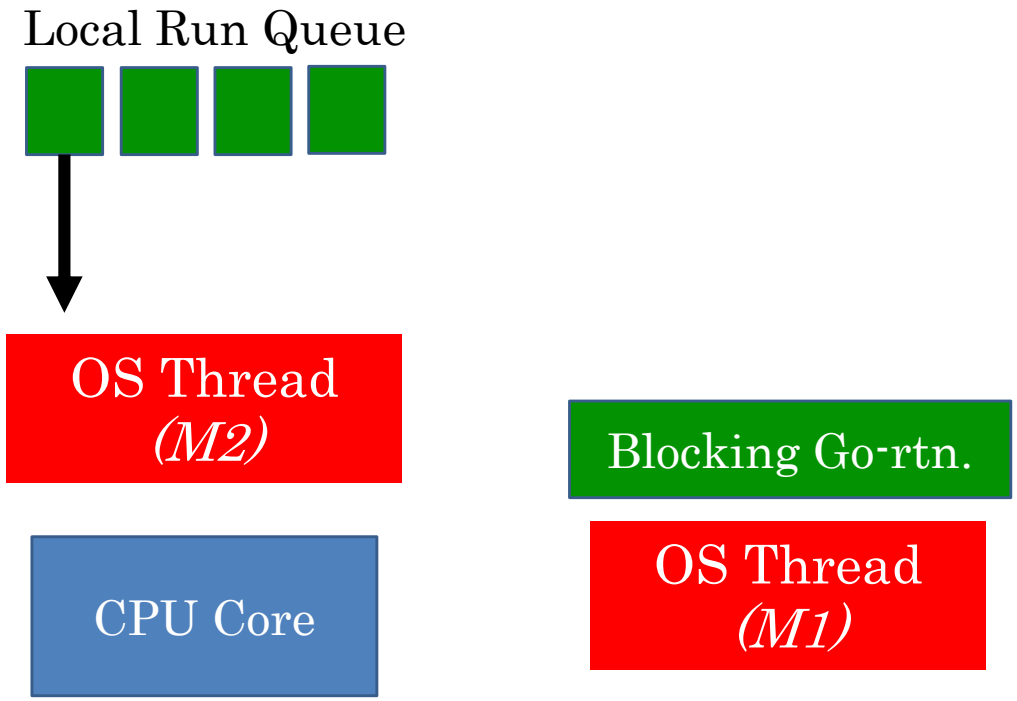


Dealing with Blocking Syscalls

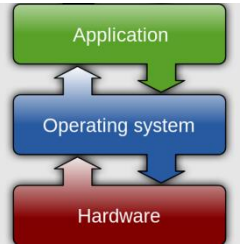
- What if a goroutine wants to make a blocking syscall?
  - Example: File I/O



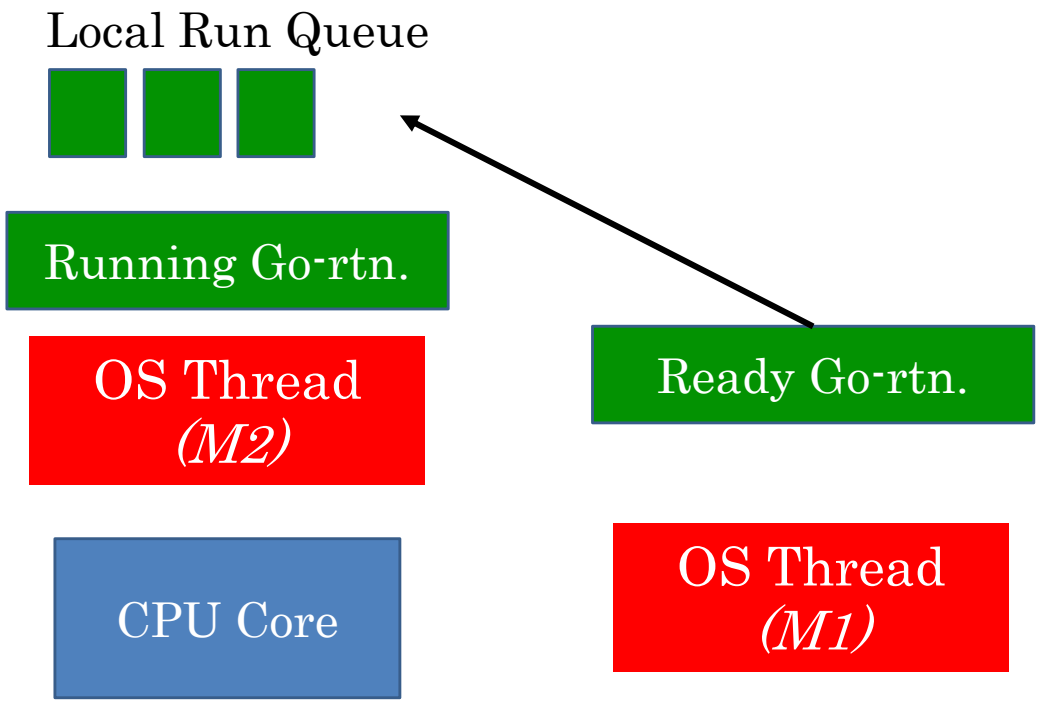
Dealing with Blocking Syscalls



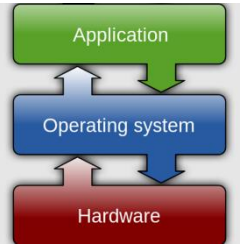
- What if a goroutine wants to make a blocking syscall?
  - Example: File I/O
- While syscall is blocking, allocate new OS thread (M2)
  - M1 is blocked by kernel, M2 lets us continue using CPU



Dealing with Blocking Syscalls

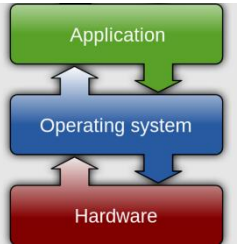


- Syscall completes: Put invoking goroutine back on queue
- Keep M1 around in a spare pool
- Swap it with M2 upon next syscall, no need to pay thread creation cost



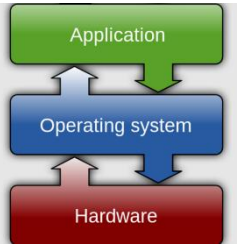
Announcements

- Assignment 3 due April 27
  - No deadline extensions will be possible
- No time to schedule another project
  - People are still asking for extensions
- We will do a final examination after all
  - May 6, 12.30pm-2.30pm
  - I will adjust the grading scale to include this



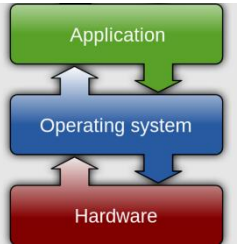
Concurrent, Well-Conditioned Systems

- Thread Pools
- User-Mode Threads
- Event-Driven Execution



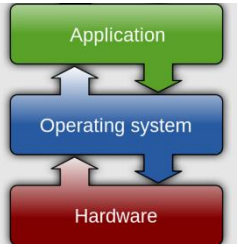
Event-Driven Execution

- Allows a system to handle MTAO with a single thread
  - Very lightweight
- Key idea: juggle different tasks within a single thread
  - The thread is being scheduled by the OS
  - All tasks' CPU bursts execute within a single thread
  - I/O bursts for each task happen in the background without an explicit backing thread – asynchronous I/O



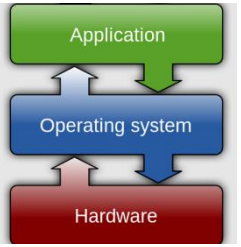
Event-Driven Server Concept

```
while (true) {
    int task_id = <wait for a task to become ready>
    <look up state for task_id>
    <execute next CPU burst for this task>
    if (task is done) {
        <forget state for task_id>
        continue;
    }
    <issue task's next (I/O) operation>
    <update state for task_id>
}
```



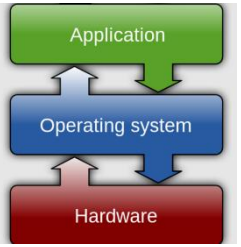
How to "Issue Task's Next I/O Operation"?

- So far, we've seen read and write, which block the calling thread
- We can put file descriptors into non-blocking mode
  - `read`: Just return whatever data is available
  - `write`: Just write whatever the kernel can buffer in its memory for now
  - So read/write calls may not read or write anything
- How to wait for the next task to become ready to perform its I/O?



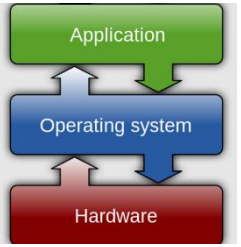
How to "Wait for Task to Become Ready"?

- POSIX provides a way to wait for one of several file descriptors to have data available
  - `select`/`poll` system calls
  - Provide a list of file descriptors
  - Blocks until at least one has "ready" data, then returns which ones do
  - Mixes well with non-blocking I/O, especially `sockets`



Alternative Asynchronous I/O APIs

- Unfortunately, non-blocking mode and `select`/`poll` don't work well with regular files
- Instead: there's the asynchronous I/O interface
  - `io_submit` issues a disk I/O
  - `io_getevents` syscall reaps completion of disk I/Os issued with `io_submit`
- Newer, better APIs still emerging (e.g., `io_uring`)

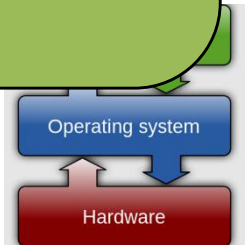


Event-Driven Server Concept

```
while (true) {
    int task_id = <wait for a task to become ready>
    <look up state for task_id>
    <execute next CPU burst for this task>
    if (task is done) {
        <forget state for task_id>
        continue;
    }
    <issue task's next (I/O) operation>
    <update state for task_id>
}
```

This looks kind of like the OS thread scheduler…

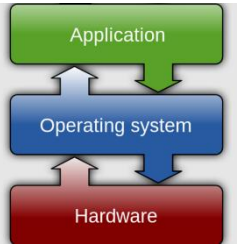
But it runs in the user program!



User-Mode Scheduler Based on Event Loop

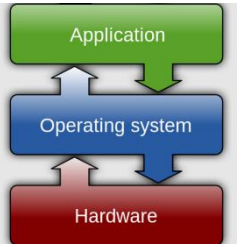
- User-mode scheduler can be an event-loop
- User threads use I/O library that issues async I/O operations
- Now user-mode scheduler can properly suspend the thread…

- But only works for I/O operations for which the kernel supports an asynchronous interface



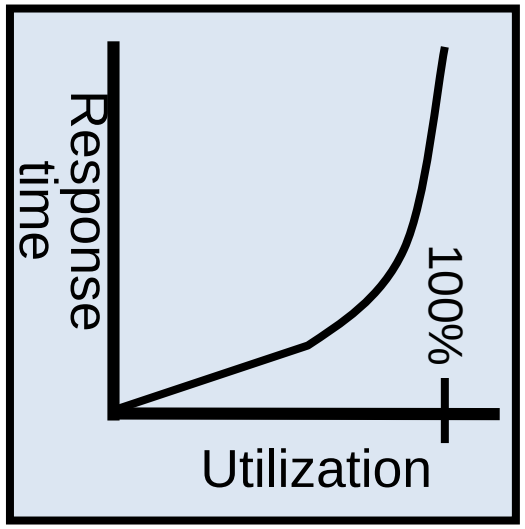
User-Mode Scheduling vs. Event Loops

- In user-mode scheduling:
  - You're still maintaining a separate stack for each thread
  - Must save PC, stack, registers when switching
  - Even if you use async I/O operations to properly suspend the user thread
- In pure event-driven scheduling:
  - All events execute in the same stack
  - All state to resume each task (e.g., which stage we're at) must be stored explicitly

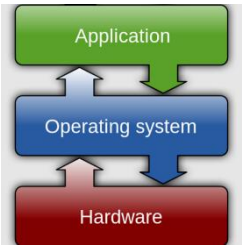


Final Word on Scheduling

- When do the details of the scheduling policy and fairness really matter?
  - Performance! Efficiency! Utilization!
- When should you simply buy a faster computer?
  - One approach: Buy it when it will pay for itself in improved response time
    - Perhaps you're paying for worse response time in reduced productivity, customer angst, etc…
    - Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization⇒100%



- An interesting implication of this curve:
  - Most scheduling algorithms work fine in the "linear" portion of the load curve, fail otherwise
  - Argues for buying a faster X when hit "knee" of curve



LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP