Advanced Operating Systems (CS 202)

Scheduling (1)

L4 microkernel family

- Successful OS with different offshoot distributions
  - Commercially successful
    - OKLabs OKL4 shipped over 1.5 billion installations by 2012
      - Mostly qualcomm wireless modems
      - But also player in automative and airborne entertainment systems
    - Used in the secure enclave processor on Apple's A7+ chips
      - All iOS devices have it! 100s of millions

Big picture overview

- Conventional wisdom at the time was:
  - Microkernels flexible with nice abstractions
  - …but are inherently low performance
    - border crossings and IPC
  - …because they are inefficient they are inflexible
- This paper refutes the performance argument
  - Main takeaway: its an implementation issue
- Several insights on how microkernels should (and shouldn't) be built
  - E.g., Microkernels should not be portable
- What are the implications if true?

Paper argues for the following

- Only put in anything that if moved out prohibits functionality
- Assumes:
  - We require security/protection
  - We require a page-based VM
  - Subsystems should be isolated from one another
  - Two subsystems should be able to communicate without involving a third

Abstractions provided by L3

- Address spaces (to support protection/separation)
  - Grant, Map, Flush
  - Handling I/O
- Threads and IPC
  - Threads: represent the address space
  - End point for IPC (messages)
  - Interrupts are IPC messages from kernel
    - Microkernel turns hardware interrupts to thread events
- Unique ids (to be able to identify address spaces, threads, IPC end points etc..)

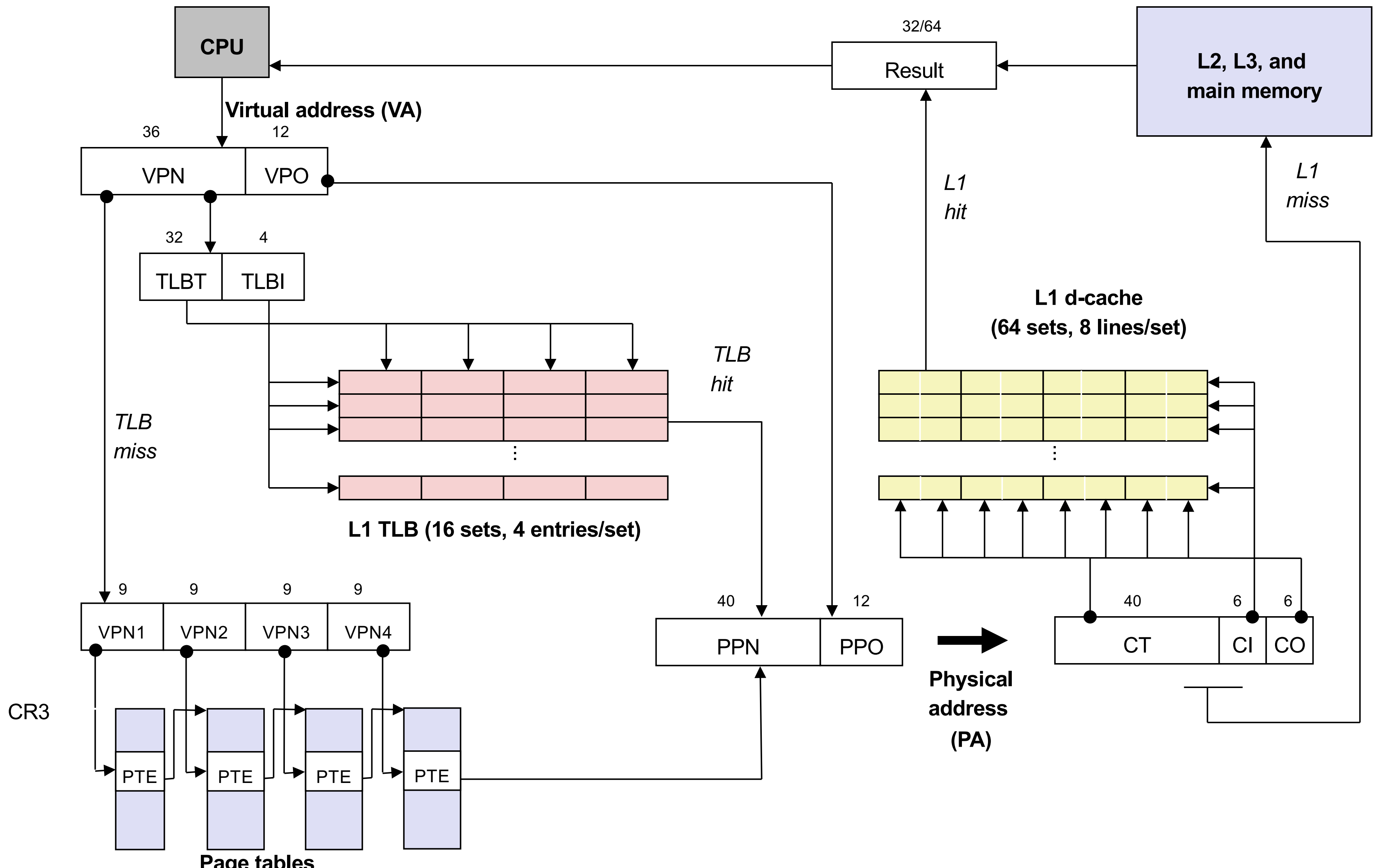
Debunking performance issues

- What are the performance issues?
  1. Switching overhead
     - Kernel user switches
     - Address space switches
     - Threads switches and IPC
  2. Memory locality loss
     - TLB
     - Caches

Mode switches

- System calls (mode switches) should not be expensive
  - Called context switches in the paper
- Show that 90% of system call time on Mach is "overhead"
  - What? Paper doesn't really say
    - Could be parameter checking, parameter passing, inefficiencies in saving state…
  - L3 does not have this overhead

Review: End-to-end Core i7 Address Translation



Thread/address space switches

- If TLBs are not tagged, they must be flushed
  - Today? x86 introduced tags but they are not utilized
- If caches are physically indexed, no loss of locality
  - No need to flush caches when address space changes
- Customize switch code to HW
- Empirically demonstrate that IPC is fast

Tricks to reduce the effect

- TLB flushes due to AS switch could be very expensive
  - Since microkernel increases AS switches, this is a problem
  - Tagged TLB? If you have them
  - Tricks with segments to provide isolation between small address spaces
    - Remap them as segments within one address space
    - Avoid TLB flushes

Memory effects

- Chen and Bershad showed memory behavior on microkernels worse than monolithic
- Paper shows this is all due to more cache misses
- Are they capacity or conflict misses?
  - Conflict: could be structure
  - Capacity: could be size of code
- Chen and Bershad also showed that self-interference more of a problem than user-kernel interference
- Ratio of conflict to capacity much lower in Mach
  - → too much code, most of it in Mach

Conclusion

- Its an implementation issue in Mach
- Its mostly due to Mach trying to be portable
- Microkernel should not be portable
  - It's the hardware compatibility layer
  - Example: implementation decisions even between 486 and Pentium are different if you want high performance
  - Think of microkernel as microcode

Today: CPU Scheduling

Today: CPU Scheduling

- Scheduler runs when we context switching among processes/threads on the ready queue
  - What should it do? Does it matter?
- Making the decision on what thread to run is called scheduling
  - What are the goals of scheduling?
  - What are common scheduling algorithms?
  - Lottery scheduling
- Scheduling activations
  - User level vs. Kernel level scheduling of threads

Scheduling

- Right from the start of multiprogramming, scheduling was identified as a big issue
  - CCTS and Multics developed much of the classical algorithms
- Scheduling is a form of resource allocation
  - CPU is the resource
  - Resource allocation needed for other resources too; sometimes similar algorithms apply
- Requires mechanisms and policy
  - Mechanisms: Context switching, Timers, process queues, process state information, …
  - Scheduling looks at the policies: i.e., when to switch and which process/thread to run next

Preemptive vs. Non-preemptive scheduling

- In *preemptive* systems where we can interrupt a running job (involuntary context switch)
  - We're interested in such schedulers…
- In *non-preemptive* systems, the scheduler waits for a running job to give up CPU (voluntary context switch)
  - Was interesting in the days of batch multiprogramming
  - Some systems continue to use cooperative scheduling
  - Example algorithms: RR, Shortest Job First (how to determine shortest), …

Scheduling Goals

- What are some reasonable goals for a scheduler?
- Scheduling algorithms can have many different goals:
  - CPU utilization
  - Job throughput (# jobs/unit time)
  - Response time ($Avg(T_{ready})$: avg time spent on ready queue)
  - Fairness (or weighted fairness)
  - Other?
- Non-interactive applications:
  - Strive for job throughput, turnaround time (supercomputers)
- Interactive systems
  - Strive to minimize response time for interactive jobs
- Mix?

Goals II: Avoid Resource allocation pathologies

- Starvation no progress due to no access to resources
  - E.g., a high priority process always prevents a low priority process from running on the CPU
  - One thread always beats another when acquiring a lock
- Priority inversion
  - A low priority process running before a high priority one
  - Could be a real problem, especially in real time systems
    - Mars pathfinder: http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html
- Other
  - Deadlock, livelock, convoying …

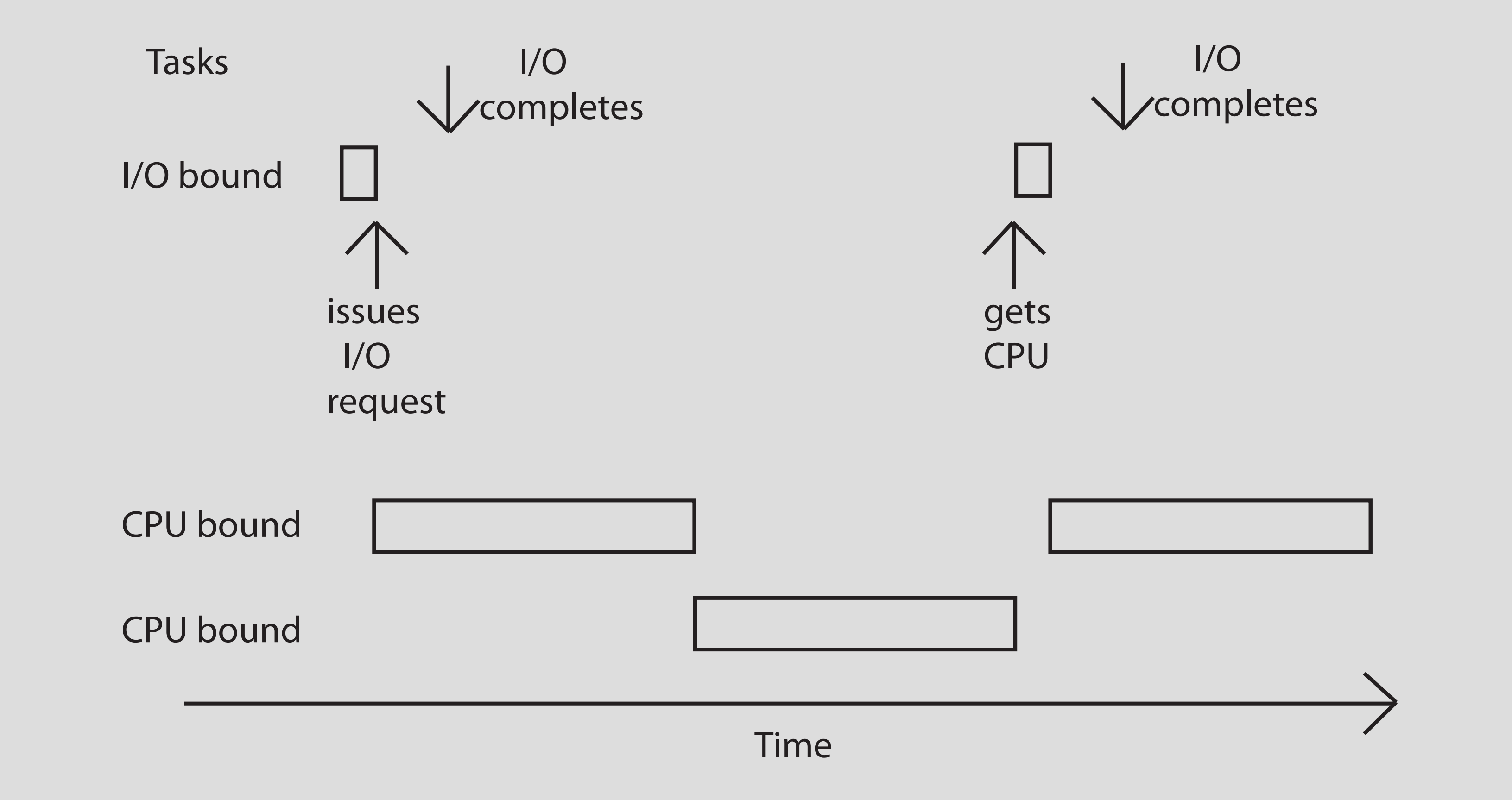
Non-preemptive approaches

- Introduced just to have a baseline
- FIFO: schedule the processes in order of arrival
  - Comments?
- Shortest Job first
  - Comments?

Preemptive scheduling: Round Robin

- Each task gets resource for a fixed period of time (time quantum)
  - If task doesn’t complete, it goes back in line
- Need to pick a time quantum
  - What if time quantum is too long?
    - Infinite?
  - What if time quantum is too short?
    - One instruction?

Mixed Workload

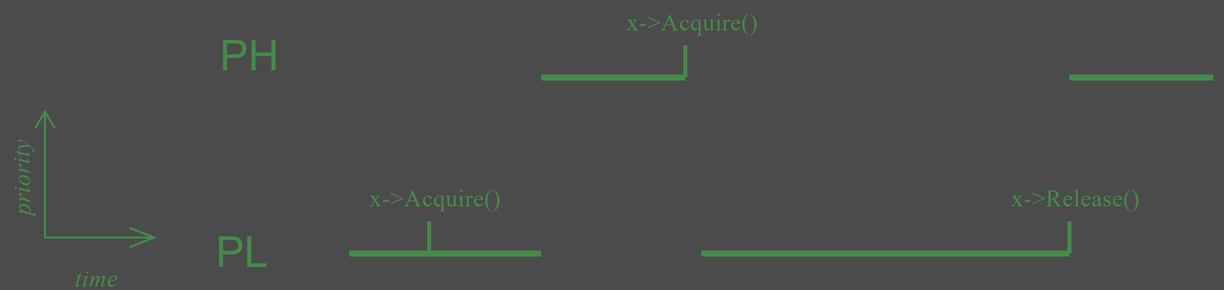


Priority Scheduling

- Priority Scheduling
  - Choose next job based on priority
    - Airline check-in for first class passengers
  - Can implement SJF, priority = 1/(expected CPU burst)
  - Also can be either preemptive or non-preemptive
- Problem?
  - Starvation – low priority jobs can wait indefinitely
- Solution
  - "Age" processes
    - Increase priority as a function of waiting time
    - Decrease priority as a function of CPU consumption

More on Priority Scheduling

- For real-time (predictable) systems, priority is often used to isolate a process from those with lower priority. *Priority inversion* is a risk unless all resources are jointly scheduled.



[redacted]

Combining Algorithms

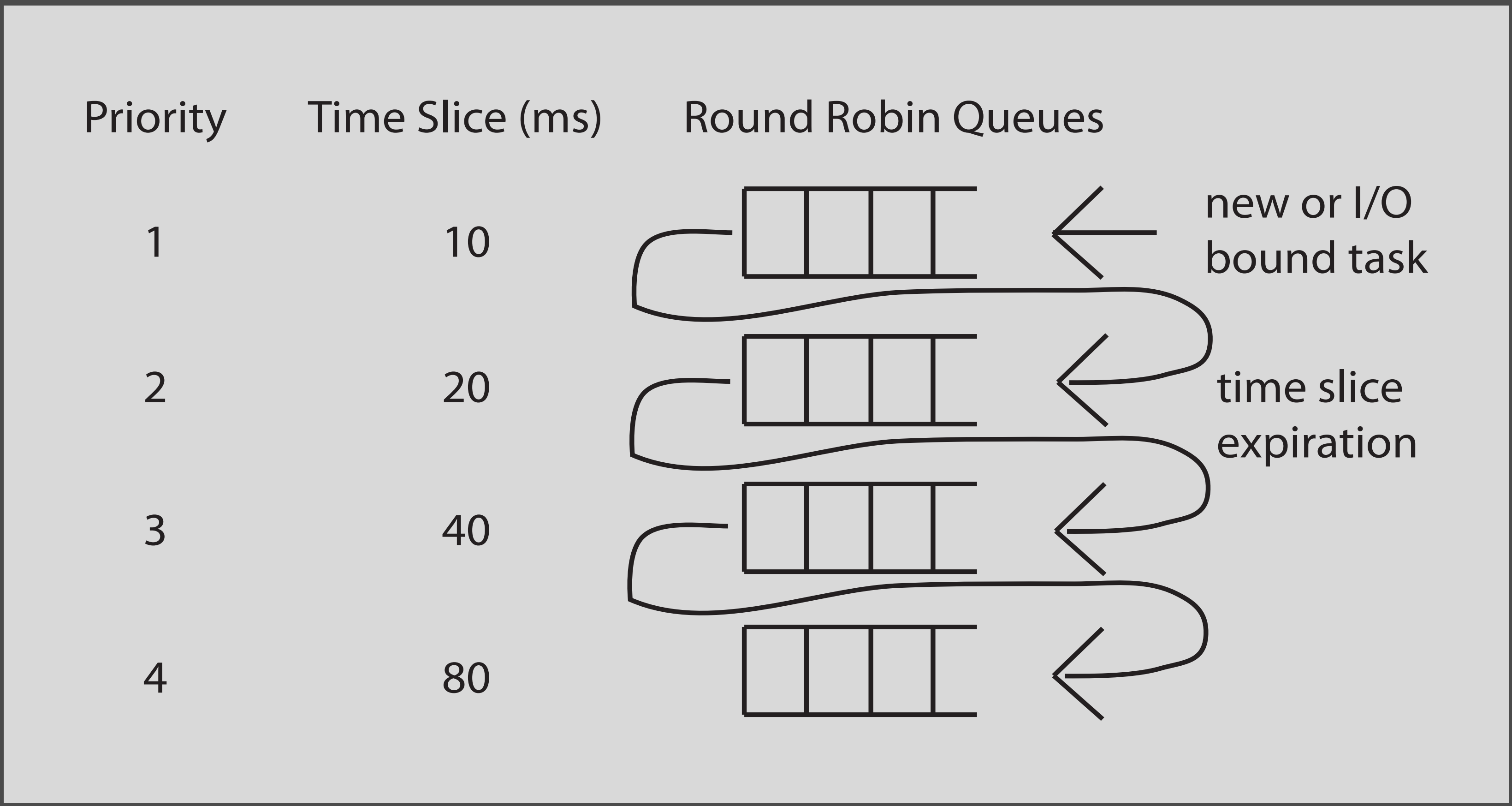
- Scheduling algorithms can be combined
  - Have multiple queues
  - Use a different algorithm for each queue
  - Move processes among queues

- Example: Multiple-level feedback queues (MLFQ)
  - Multiple queues representing different job types
    - Interactive, CPU-bound, batch, system, etc.
  - Queues have priorities, jobs on same queue scheduled RR
  - Jobs can move among queues based upon execution history
    - Feedback: Switch from interactive to CPU-bound behavior

Multi-level Feedback Queue (MFQ)

- Goals:
  - Responsiveness
  - Low overhead
  - Starvation freedom
  - Some tasks are high/low priority
  - Fairness (among equal priority tasks)
- Not perfect at any of them!
  - Used in Unix (and Windows and MacOS)

MFQ


Priority
Time Slice (ms)
Round Robin Queues
1
10
2
20
3
40
4
80
new or I/O
bound task
time slice
expiration

Unix Scheduler

- The canonical Unix scheduler uses a MLFQ
  - 3-4 classes spanning ~170 priority levels
    - Timesharing: first 60 priorities
    - System: next 40 priorities
    - Real-time: next 60 priorities
    - Interrupt: next 10 (Solaris)
- Priority scheduling across queues, RR within a queue
  - The process with the highest priority always runs
  - Processes with the same priority are scheduled RR
- Processes dynamically change priority
  - Increases over time if process blocks before end of quantum
  - Decreases over time if process uses entire quantum

Linux scheduler

- Went through several iterations
- Currently CFS
  - Fair scheduler, like stride scheduling
  - Supersedes O(1) scheduler: emphasis on constant time scheduling regardless of overhead
  - CFS is O(log(N)) because of red-black tree
  - Is it really fair?
- What to do with multi-core scheduling?

Our scheduler reading

- Ticket/Stride
  - Problem: How to control allocation of CPU in a principled way
- Scheduler activations
  - How to let the application control scheduling
    - Reminds you of SPIN/extensibility?
- How to do scheduling on emerging systems
  - Multicore, cloud, multiple resources..