Chapter 6: Advanced Scheduling

CSCI 3753 Operating Systems

Instructor: Chris Womack

University of Colorado at Boulder

All material by Dr. Rick Han

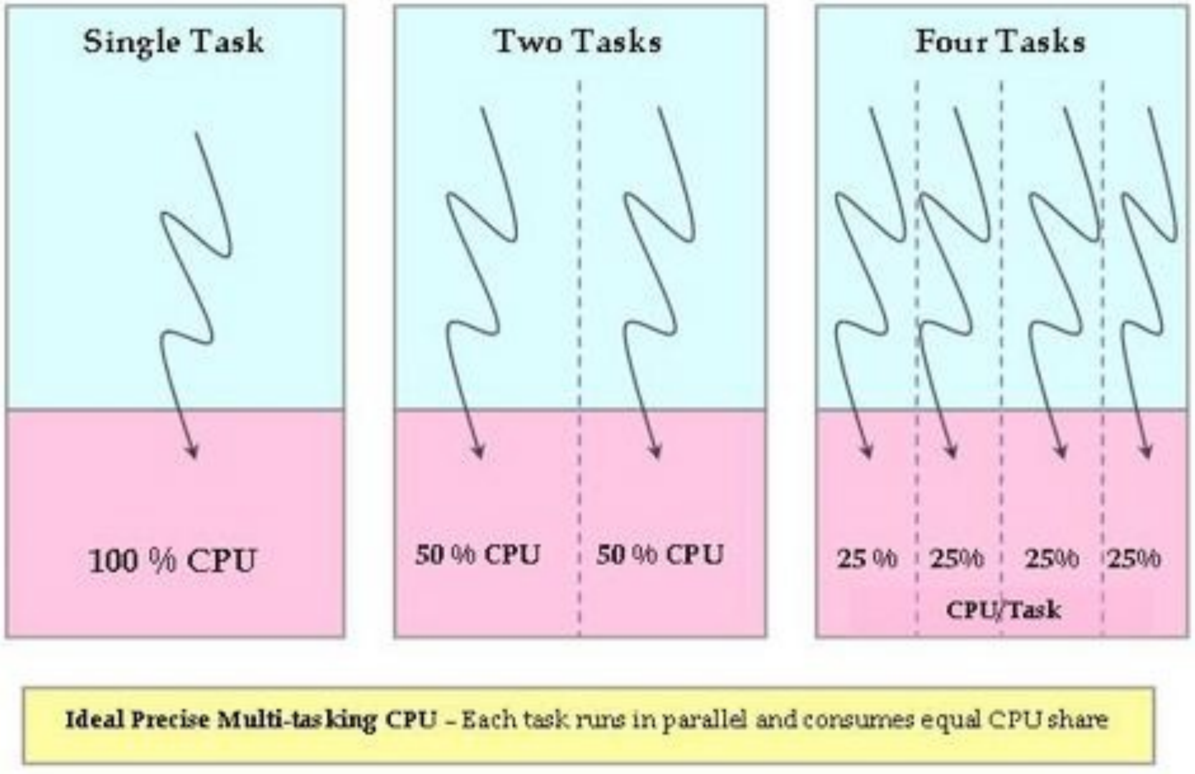
University of Colorado **Boulder**

Recap

- Linux O(1) Scheduler
  - An active and expired priority queues
  - After all tasks have moved from active to expired, swap the queues
  - Search time is constant hence O(1), bounded by # of priorities, not # of tasks
  - Priority value can change based on interactivity – heuristic is complex

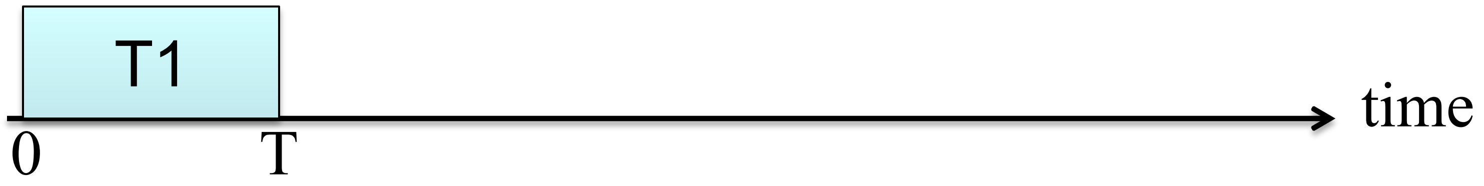
Completely Fair Scheduler (CFS) in Linux

- Linux 2.6.23+/3.* has a "completely fair" scheduler
- Based on concept of an "ideal" multitasking CPU
- If there are N tasks, an ideal CPU gives each task 1/N of CPU *at every instant of time*


Single Task
Two Tasks
Four Tasks
100 % CPU
50 % CPU
50 % CPU
25 %
25%
25%
25%
CPU/Task
Ideal Precise Multi-tasking CPU – Each task runs in parallel and consumes equal CPU share

University of Colorado **Boulder**

CFS Intuition

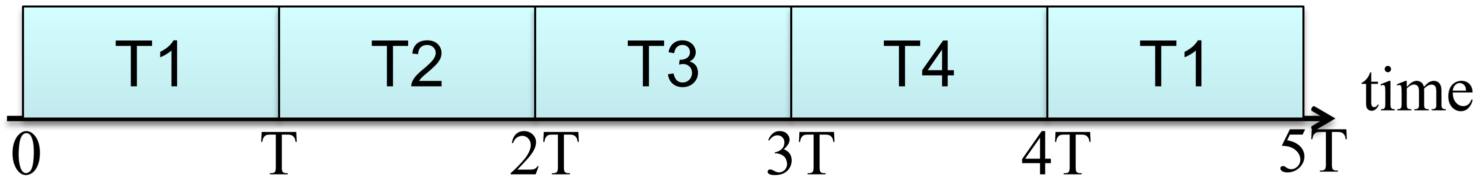


- Example:
  - The current accounting balance is summarized in the table below

	Time owed to task, i.e. wait time W_i:			
Giving time T to task:	T1	T2	T3	T4
T1	-3T/4	T/4	T/4	T/4

  - In general, at any given time t in the system, each task T_i has an amount of time owed to it on an ideal CPU, i.e. the amount of time it was forced to wait, its *wait time $W_i(t)$,*

CFS Intuition



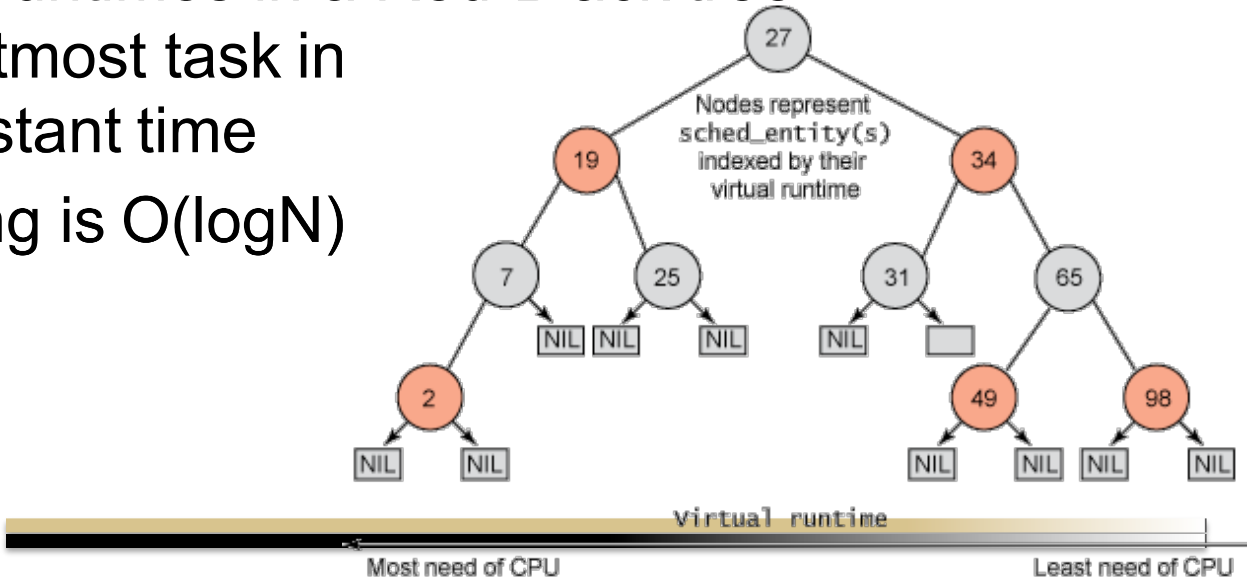
- Example: let's have round robin over 4 tasks

	Time owed to task, i.e. wait time W_i:			
Giving time T to task:	T1	T2	T3	T4
T1	-3T/4	T/4	T/4	T/4
then T2	-T/2	-T/2	T/2	T/2
then T3	-T/4	-T/4	-T/4	3T/4
then T4	0	0	0	0

- After 1 round robin, the balances owed all = 0, so every task receives its fair share of CPU over time 4T

CFS Scheduler

- Selects the task with the maximum wait time
- This is equivalent to selecting the task with the minimum virtual run time
  - See derivation from last lecture
  - To quickly find the task with the minimum vruntime, order the vruntimes in a Red-Black tree
  - Choose leftmost task in tree in constant time
  - Rebalancing is O(logN)



CFS Scheduler and Priorities

- All non-RT tasks of differing priorities are combined into one RB tree
  - Don't need 40 separate run queues, one for each priority - elegant!

1. Higher priority tasks get larger run time slices
   - Each task has a weight that is a function of the task's niceness priority
   - The *run time allocated to a task = (a default target latency of 20 ms for desktop systems) * (task's weight) / (sum of weights of all runnable tasks)*
   - Lower niceness => higher the priority => more run time is given on the CPU

CFS Scheduler and Priorities

- The weight is roughly equivalent to 1024 / (1.25 ^ nice_value).
  - So the relative ratio between different niceness priorities is geometric
- Recall niceness ranges from -20 to +19
  - -20 corresponds to Linux priority 100
  - +19 corresponds to Linux priority 139
- Example: tasks 1 & 2 have niceness 0 & 5
  - Ratio of weights of task 1/task 2 = 1024/335 = 3X
  - Every 20 ms, Task 1 gets run time = 20 ms*1024/(1024+335) = 15 ms
  - Every 20 ms, Task 2 gets run time = 5 ms

CFS Scheduler and Priorities

2. In addition, higher priority tasks are scheduled more often
   - *virtual runtime += (actual CPU run time) * NICE_0_LOAD / task's weight*
   - Higher priority

$\Rightarrow$higher weight

$\Rightarrow$ less increment of vruntime

$\Rightarrow$task is further left on the Red-Black tree and is scheduled sooner

CFS Scheduler and Groups

- While CFS is fair to tasks, it is not necessarily fair to applications
  – Suppose application A1 has 100 threads T1-T100
  – Suppose application A2 is interactive and has one thread T101
  – CFS would give A1 100/101 of CPU while giving the interactive app A2 only 1/101 of the CPU
- Instead, Linux CFS supports fairness across groups: A1 is in group 1 and A2 is in group 2
  – Groups 1 and 2 each get 50% of CPU – fair!
  – Within Group 1, 100 threads share 50% of CPU
  – Multi-threaded apps don't overwhelm single thread apps

Real Time Scheduling in Linux

- Linux also includes three real-time scheduling classes:
  - Real time FIFO – soft real time (SCHED_FIFO)
  - Real time Round Robin – soft real time (SCHED_RR)
  - Real time Earliest Deadline First – hard real time as of Linux 3.14 (SCHED_DEADLINE)
- Only processes with the priorities 0-99 have access to these RT schedulers

Real Time Scheduling in Linux

- "When a Real time FIFO task starts running, it continues to run until it voluntarily yields the processor, blocks or is preempted by a higher-priority real-time task.
  - It has no timeslices.
  - All other tasks of lower priority will not be scheduled until it relinquishes the CPU.
  - Two equal-priority Real time FIFO tasks do not preempt each other."

Real Time Scheduling in Linux

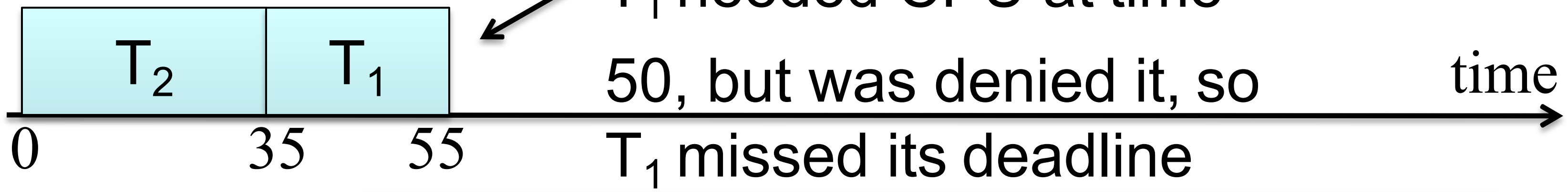
- "SCHED_RR is similar to SCHED_FIFO, except that such tasks are allotted timeslices based on their priority and run until they exhaust their timeslice"
- Non-real time tasks continue to use CFS algorithm

Rate-Monotonic Scheduling

- For periodic tasks
  - Each task T_i needs the CPU at a period of every Δ_i
- A task's priority = 1/(period of the task) = $1/\Delta_i$
  - Assigns a higher priority to tasks that require the CPU more often, i.e. have smaller task periods
- If a task's priority is higher than another task, it *preempts* that task

Rate-Monotonic Scheduling

- Example 1: tasks T_1 and T_2 have periods $\Delta_1 = 50$ and
  $\Delta_2 = 100$.
- Processing times for T_1 and T_2 are $X_1 = 20$ and $X_2 = 35$.
- Each task must complete its execution by the start of the next period.
- Suppose T_2 is assigned higher priority than T_1, but there is no preemption (this is not rate monotonic)

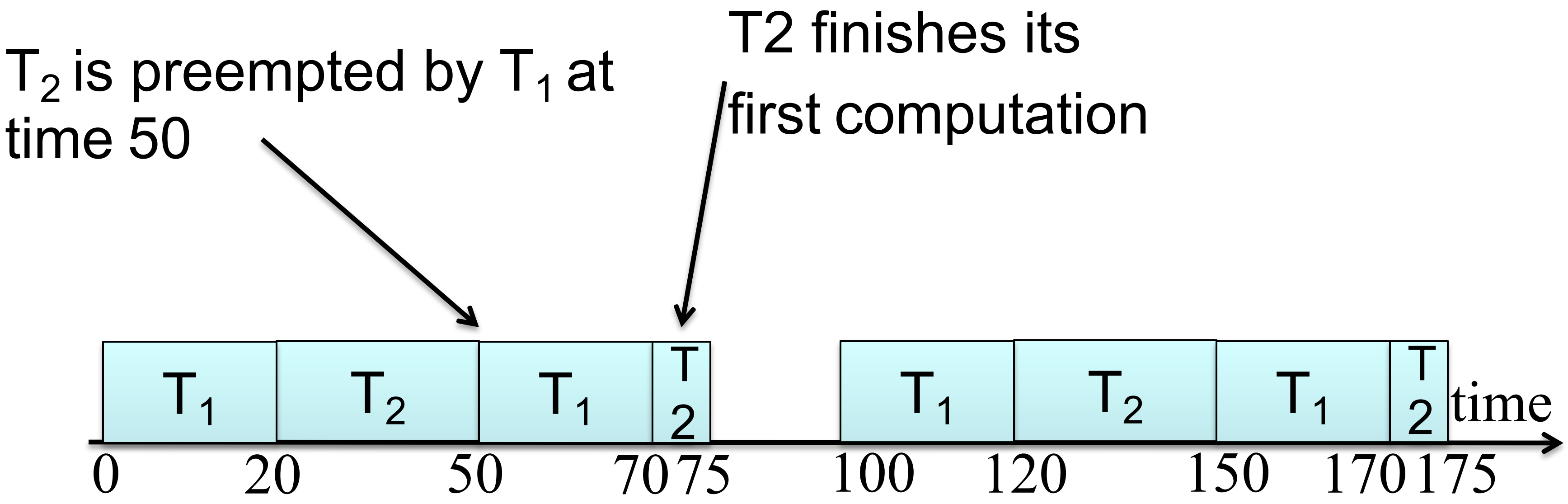


T_1 needed CPU at time 50, but was denied it, so T_1 missed its deadline

University of Colorado **Boulder**

Rate-Monotonic Scheduling

- Example 2: same parameters as Example 1, except now assign priorities according to rate-monotonicity
- T_1's priority is higher because T_1 has a shorter periodicity according to rate monotonicity



T_2 is preempted by T_1 at time 50
T2 finishes its first computation
T_1
T_2
T_1
T_2
T_1
T_2
T_1
T_2
time
0
20
50
70
75
100
120
150
170
175

University of Colorado Boulder

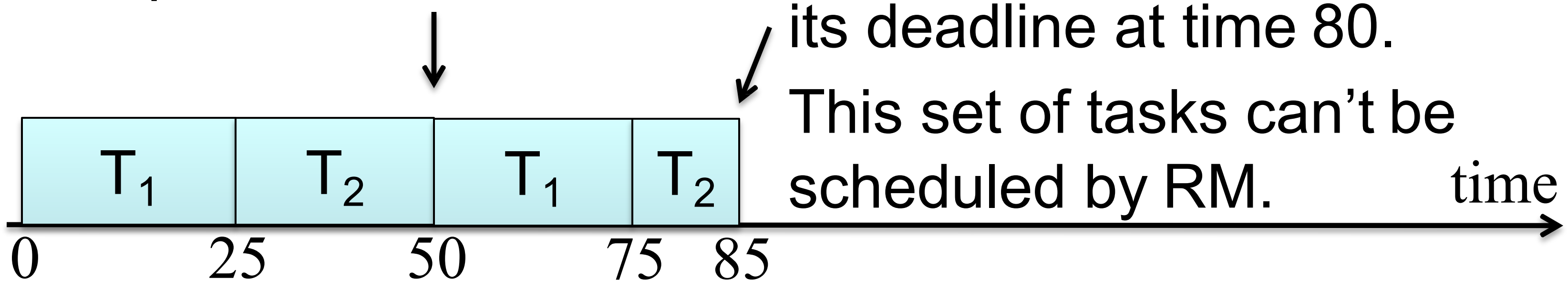
Rate-Monotonic Scheduling

- Example 3: tasks T_1 and T_2 have periods $\Delta_1 = 50$ and $\Delta_2 = 80$. Processing times for T_1 and T_2 are $X_1 = 25$ and $X_2 = 35$.
- T_1's CPU utilization is 25/50 = 50% while T_2's CPU utilization is 35/80 = 44%, so system would seem to be schedulable since total utilization = 94% < 100%
- T_1 has highest priority by rate monotonicity

T_2 preempted with 10 left

T_2 finishes at 85, but misses its deadline at time 80.

This set of tasks can't be scheduled by RM.



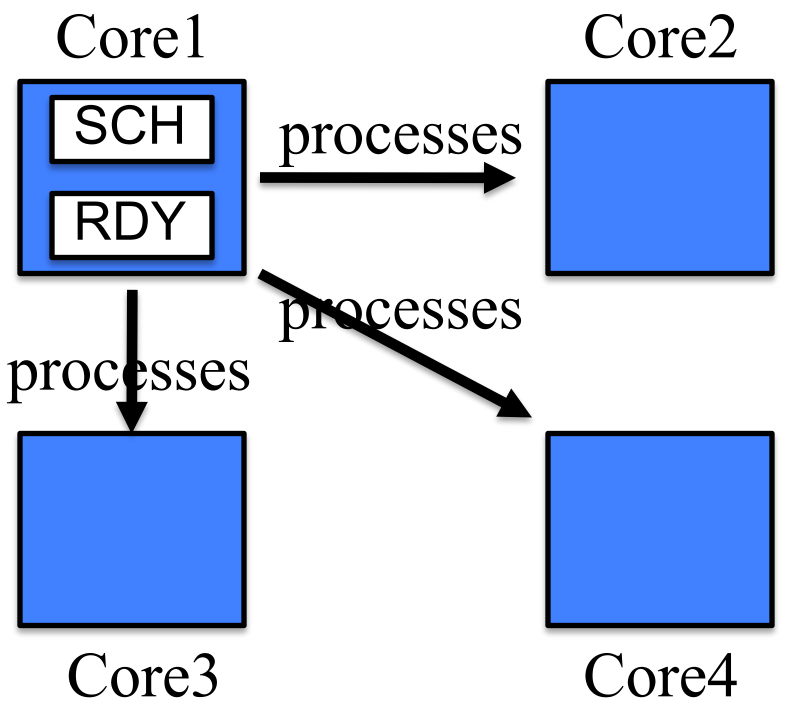
University of Colorado Boulder

Rate-Monotonic Scheduling

- Even if total CPU utilization is less than 100%, the set of tasks may still not schedulable under RM
- RM scheduling is considered optimal
  - if a set of tasks cannot be scheduled by this algorithm, it cannot be scheduled by any other algorithm assigning static priorities

Multi-core Scheduling

- Scheduling over multiple processors or cores is a new challenge.
  - A single CPU/processor may support multiple cores

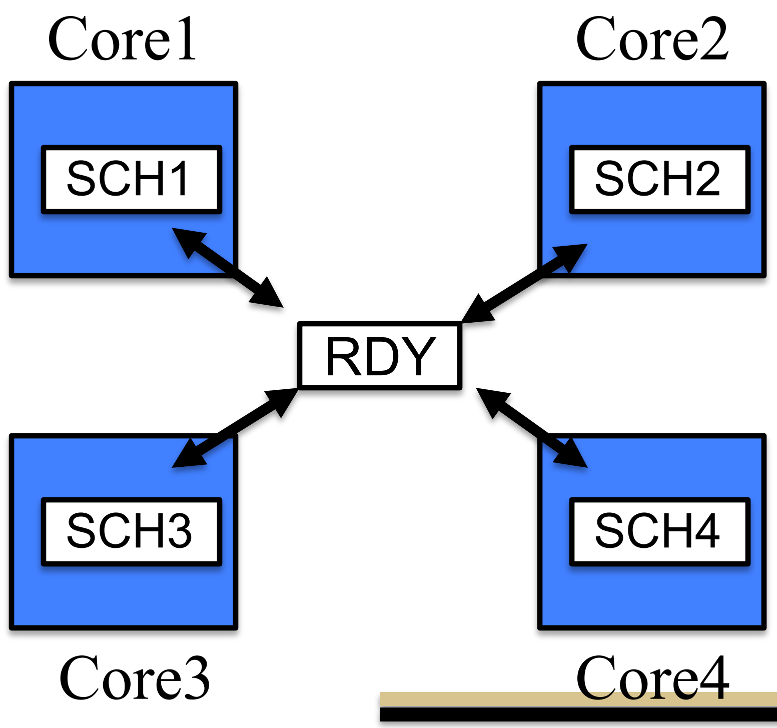


SCH = Scheduler, RDY = Ready Queue

- Variety of multi-core schedulers being tried. We'll just mention some design themes.
- In *asymmetric multiprocessing* (left) – 1 CPU handles all scheduling, decides which processes run on which cores

Multi-core Scheduling

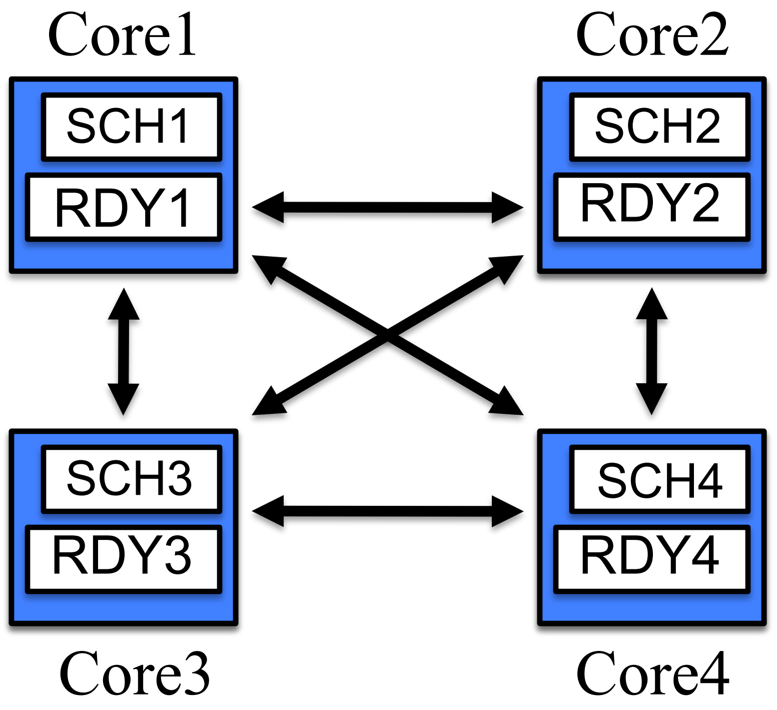
- In symmetric multi-processing (SMP), each core is self-scheduling. All modern OSs support some form of SMP. Two types:
  1. All cores share a single global ready queue.



- Here, each core has its own scheduler. When idle, each scheduler requests another process from the shared ready queue. Puts it back when time slice done.
  - Synchronization needed to write/read shared ready queue

Multi-core Scheduling

2. Another self-scheduling SMP approach is when each core has its own ready queue
   - Most modern OSs support this paradigm



- a typical OS scheduler plus a ready queue designed for a single CPU can run on each core…
- Except that processes now can migrate to other cores/processors
  - There has to be some additional coordination of migration

Multi-core Scheduling

- Caching is important to consider
  - Each CPU has its own cache to improve performance
  - If a process migrates too much between CPUs, then have to rebuild L1 and L2 caches each time a process starts on a new core/processor
  - L3 caches that span multiple cores can help alleviate this, but there is a performance hit, because L3 is slower than L1 and L2.
  - In any case, L1 and L2 caches still have to be rebuilt.

Multi-core Scheduling

- To maximally exploit caching, processes tend to stick to a given core/processor = processor affinity
  - In hard affinity, a process specifies via a system call that it insists on staying on a given CPU core
  - In soft affinity, there is still a bias to stick to a CPU core, but processes can on occasion migrate.
  - Linux supports both

Multi-core Scheduling

- Load balancing
  - Goal: Keep workload evenly distributed across cores
  - Otherwise, some cores will be under-utilized.
  - When there is a single shared ready queue, there is automatic load balancing
    - cores just pull in processes from the ready queue whenever they're idle.

Multi-core Scheduling

- Load balancing when there are separate ready Q's,
  - push migration – a dedicated task periodically checks the load on each core, and if imbalance, pushes processes from more-loaded to less-loaded cores
  - Pull migration – whenever a core is idle, it tries to pull a process from a neighboring core
  - Linux and FreeBSD use a combination of pull and push

Multi-core Scheduling

- Load balancing can conflict with caching
  - Push/pull migration causes caches to be rebuilt
- Load balancing can conflict with power management
  - Mobile devices typically want to save power
  - One approach is to power down unused cores
  - Load balancing would keep as many cores active as possible, thereby consuming power
- In systems, often conflicting design goals

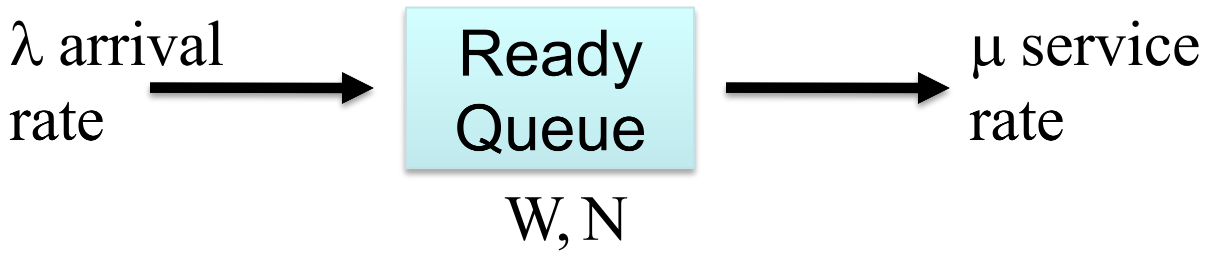
Hardware Multithreading

- Multiple hardware threads per core
- Threads can become blocked waiting for data – called a memory stall – so that core waits for a long time
  - e.g. a cache miss that may take thousands of CPU cycles to retrieve the data from main memory.
  - Instead of waiting, core switches immediately to another thread to run
  - Also called hyperthreading (Intel-specific)

Hardware Multithreading

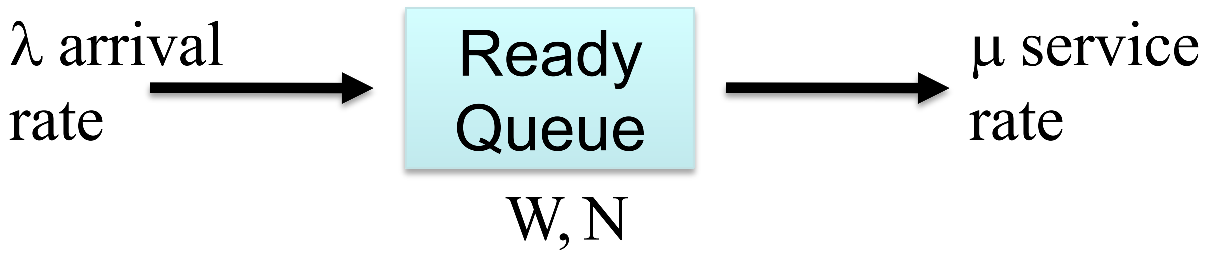
- If N hyperthreads are supported per core, it appears as if there are N logical cores per physical core
  - Each logical core runs at about 1/N the speed of the physical core
  - e.g. A dual core dual threaded CPU has four logical cores
- Requires 2 levels of scheduling:
  - map threads to hardware threads;
  - which hardware thread to run, e.g. RR

Little's Law



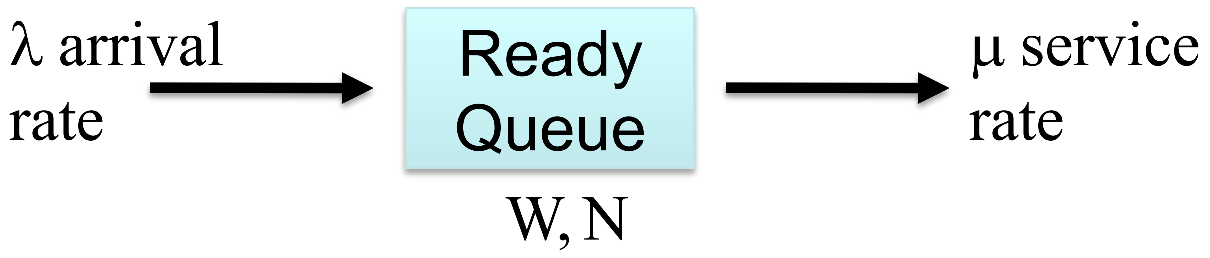
- How big of a ready queue do we need?
  - Employ stochastic queueing theory to answer this question
  - Consider an abstract ready queue where λ = arrival rate of tasks in the queue, μ = service rate of tasks exiting the queue
  - In steady state, λ must be less than or equal to μ, i.e. $\lambda \leq \mu$

Little's Law



- How big of a ready queue do we need?
  - Let W = average wait time in the queue per process
    - Note W is not the inverse of the service time m. Analogy: If we have a series of cars that go into a tunnel, eventually they will emerge on the other side, and the steady state service (exit) rate of the cars will equal the arrival (entry) rate. However, the exit rate doesn't tell us what is the average time spent by each car in the tunnel. The length of the tunnel (and car speed) tells how long each car spent in the tunnel.

Little's Law



- How big of a ready queue do we need?
  - Let N = average queue length
- Then $N = \lambda * W$ (Little's Law)
  - In W seconds, on average λ*W tasks arrive in the ready queue.
  - This conclusion is valid for any scheduling algorithm and arrival distribution.
  - Useful to know how big on average the queue size N should be for allocation by the OS.