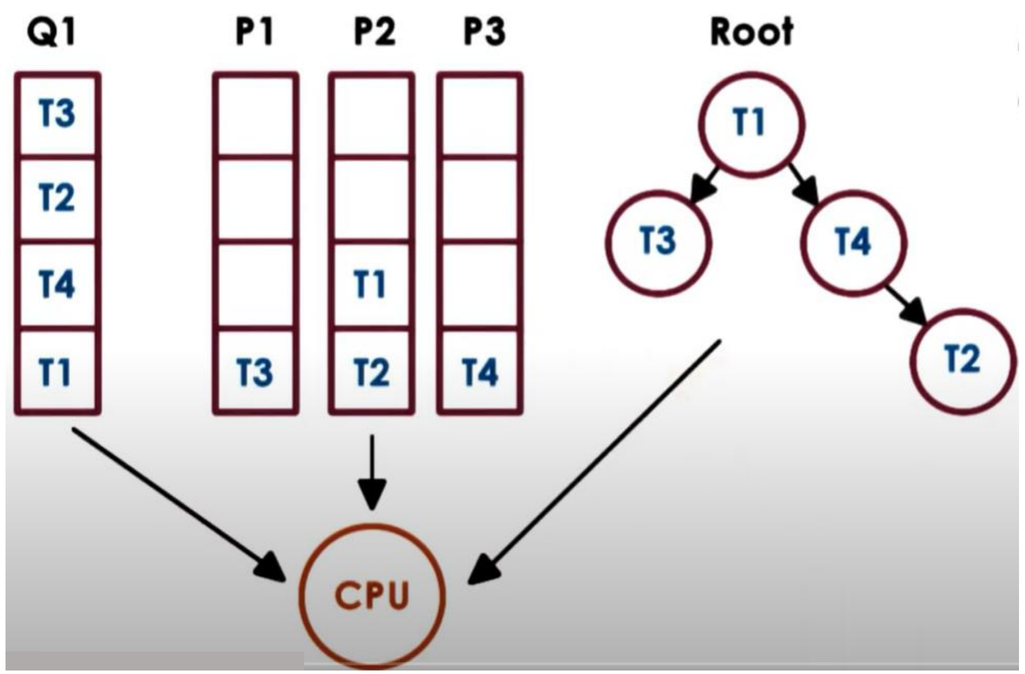
Operating systems

Week 5-2

Scheduling

Run-queue Data Structure

- It is a logical representation, also can be represented in multiple queues dealing with different priorities, or it could be a tree or some other type of data structure

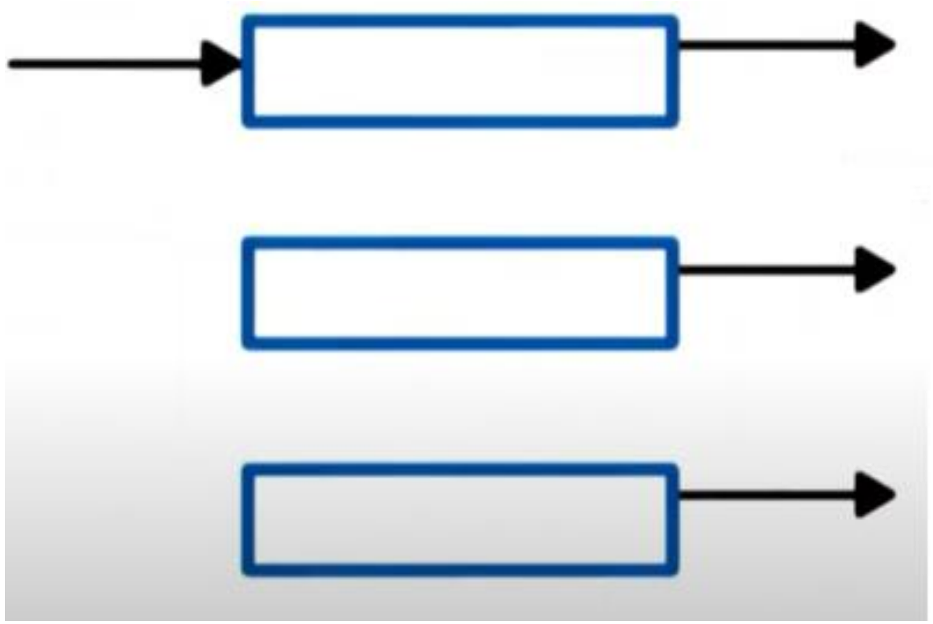


- Regardless of the data structure, what is important is that it should be easy for the scheduler to find the next thread to run, given the scheduling criteria

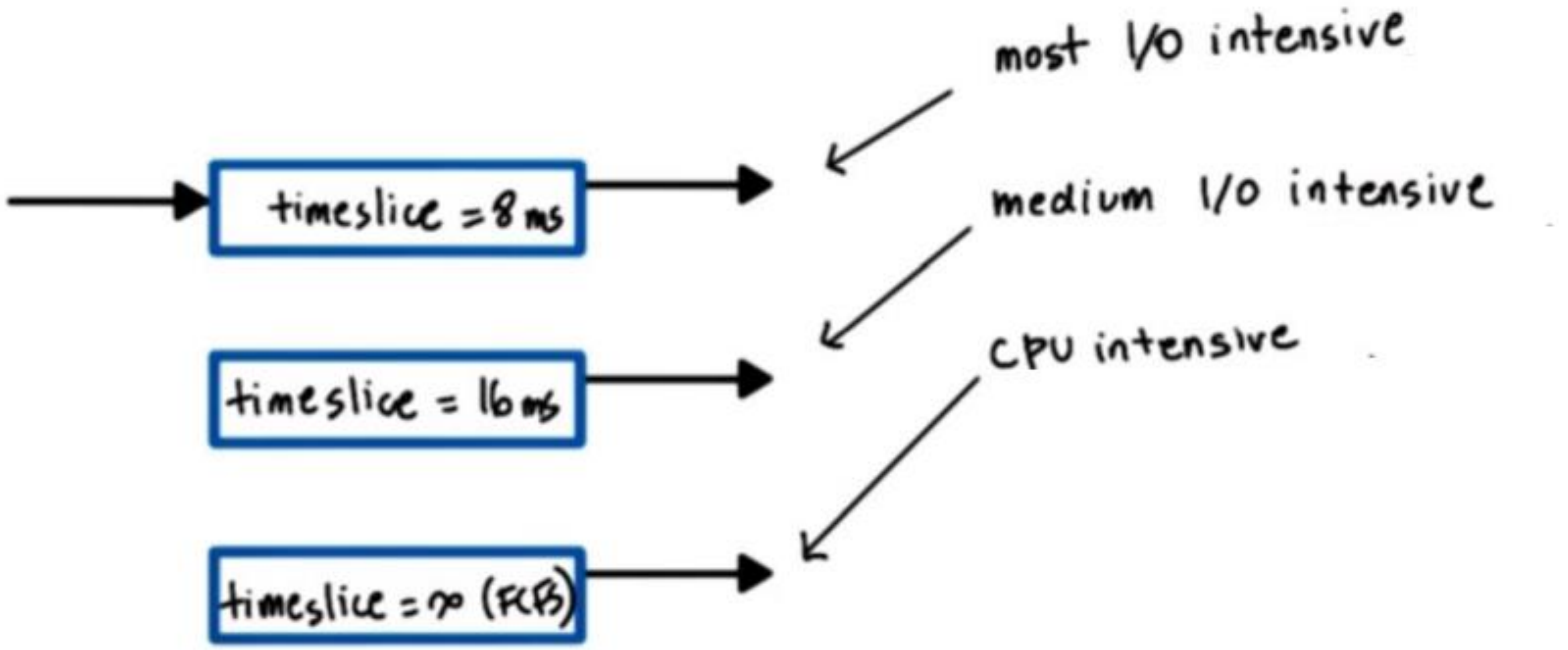
Run-queue Data Structure

- I/O or CPU bound tasks with different time-slices:
  - Same run-queue check type← different policies applied
  - Separate I/O and CPU bound tasks into two different data structures← different polices applied

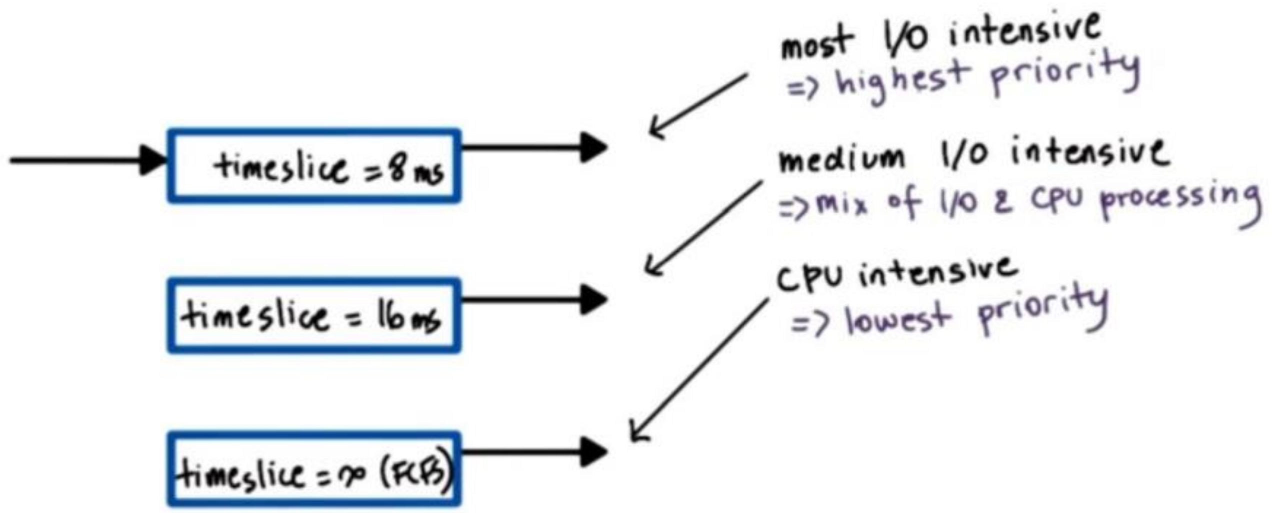
- One solution is applying multi-queue data structure



Dealing with Different Time-slice Values



- Tasks are placed in the appropriate queue depending on their type
- Scheduler selects the task to run based on its priority, highest to lowest
- This structure provides time-slice and I/O bound tasks benefits and avoids time-slice overheads for the CPU bound tasks

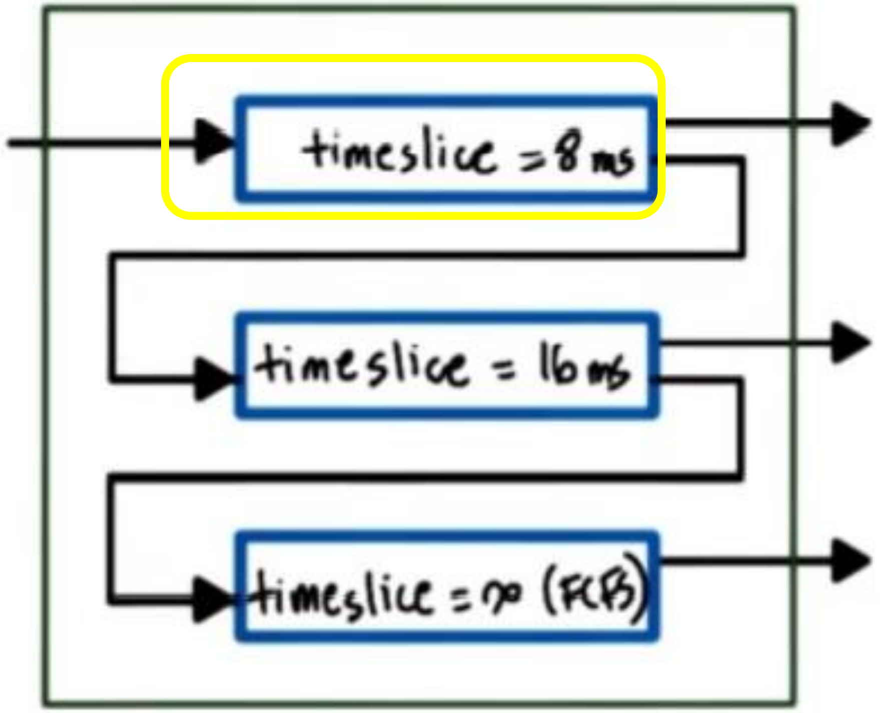


Dealing with Different Time-slice Values

- Is the task CPU or I/O intensive?
- How do we know how I/O intensive the task is?
- To answer such questions, run history-based heuristics→ like slide the task run and then decide what to do with it
- Now, what about new tasks, or what about tasks that have dynamic changes in their behavior?
- Solution, deal with the multi-queue data structure as one single multi-queue data structure

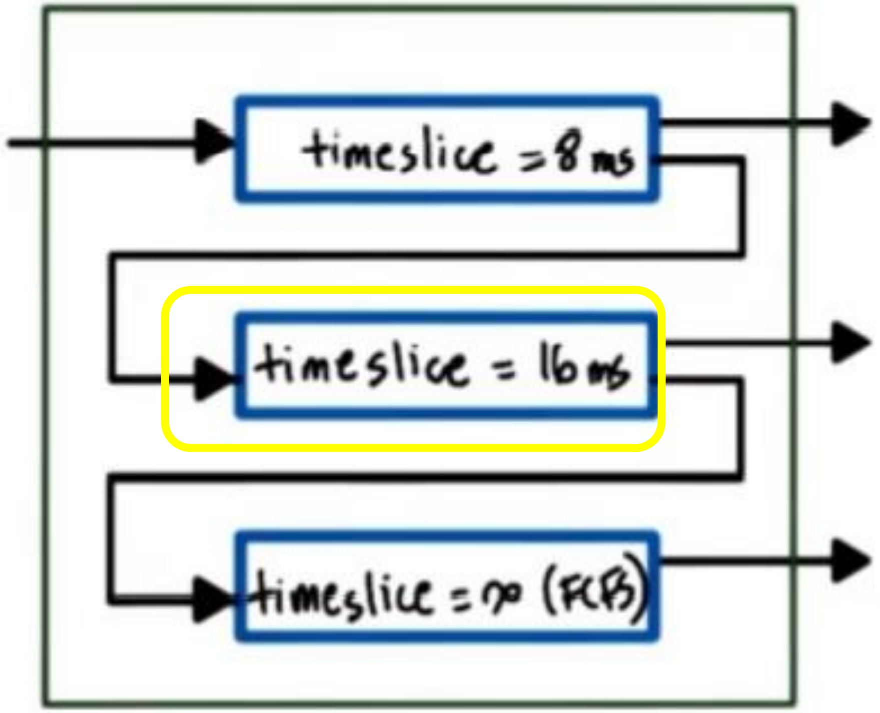
Single Multi-queue Data Structure

- A newly created task will enter in the topmost queue, lowest time-slice associated and expected to be most demanding task
- If the task yields for I/O or it is interrupted, then context switch
- Means our choice of queue was good



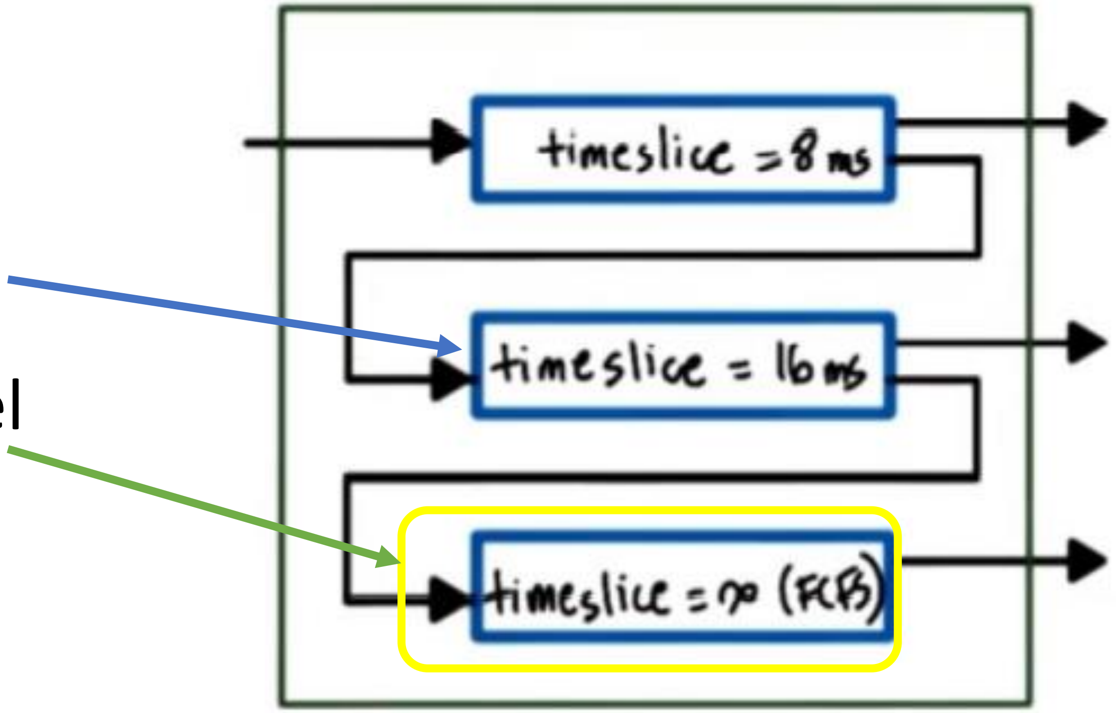
Single Multi-queue Data Structure

- The task is indeed fairly I/O interactive, and so we want to keep the task in this level
- When task becomes runnable, after that I/O operation completes, it will be placed in this exact same queue



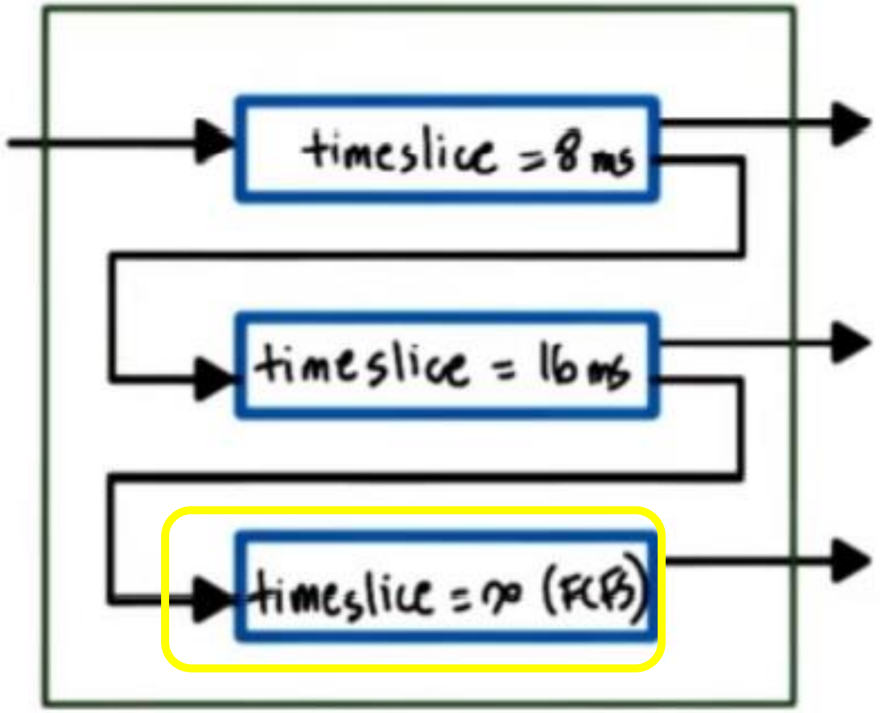
Single Multi-queue Data Structure

- However, if the task time-slice expires means it was more CPU intensive (than we thought)
- Task will be preempted from second level
- Task will be pushed down lower level
- Note if this task to run in a different time, it will be scheduled in the second level



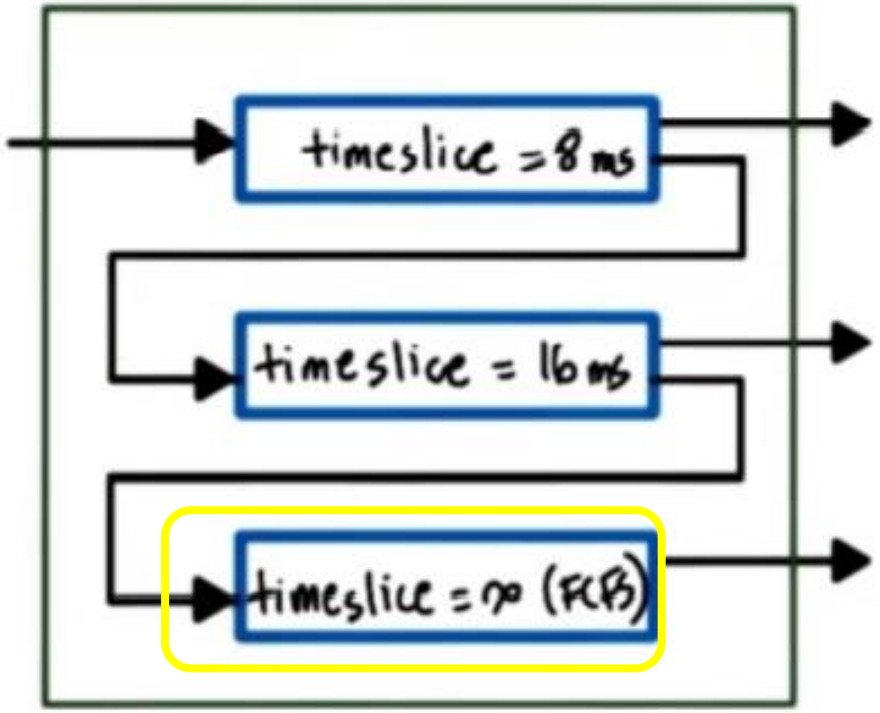
Single Multi-queue Data Structure

- If the task ends up getting preempted when it was scheduled from second queue as this example, so it used up its entire millisecond time slice. That means that it's even more CPU intensive
- Task will get pushed down to the bottom queue→ mechanism to push the task down these levels based on its historic information



Single Multi-queue Data Structure

- If a task that's in one of the lower queues all of a sudden starts repeatedly releasing the CPU earlier
- Scheduler will think of this task as I/O intensive more than (we originally thought)
- Scheduler can push such task up at one of the queues on the higher levels → multi-level feed back queue



Notes

- Different scheduling polices that are associated with each of the different levels that are part of this data structure
- This data structure incorporates a feedback mechanism, that allows over time to adjust which one of these levels will place a task, and when we're trying to figure out what is the best time-sharing schedule for the sub-task in the system
- Linux is a multi-level feedback queue with 60 levels. So, 60 sub-queues. Also, some fancy feedback rules that determine how and when a thread gets pushed up and down these different levels

Linux O (1) Scheduler…. Priority, time-slice

- It is able to perform task management operations, such as selecting a task from the run queue, or adding a task to it in constant time
- Regardless of the total number of active tasks in the system. It’s a preemptive and priority-based scheduler
- Has total of 140 priority levels, with zero being the highest and then 139 the lowest

Linux O (1) Scheduler… Priority, time-slice

- Priority levels are organized into two different classes (tasks)
  - Real time (0-99)
  - Time-sharing (100-139)

- All user processes have one of the time-sharing priority levels. Their default priority is 120 but it can be adjusted with a so-called nice value

- There’s a system call that can be used to do this. And the nice values can be between (negative) -20 and 19, so as to span the entire set of time-sharing priorities

Linux O (1) Scheduler….. Feed-back

- The O (1) scheduler borrows from the multilevel feedback queue scheduler, in that it associates different time-slice values with different priority levels
- And it also uses feedback from how the tasks behaved in the past to determine how to adjust their priority levels in the future. It differs however, in how it assigns the time-slice values to priorities and how it uses the feedback
- It assigns time-slice values based on the priority level of the task, similar to the multilevel feedback queues
- However, it assigns smallest time-slice values to the low priority, CPU bound tasks and it assigns high time-slice values to the more interactive high priority tasks.
- The feedback it uses for the time-sharing tasks is based on the time that the task spends sleeping, the time that it was waiting for something or idling

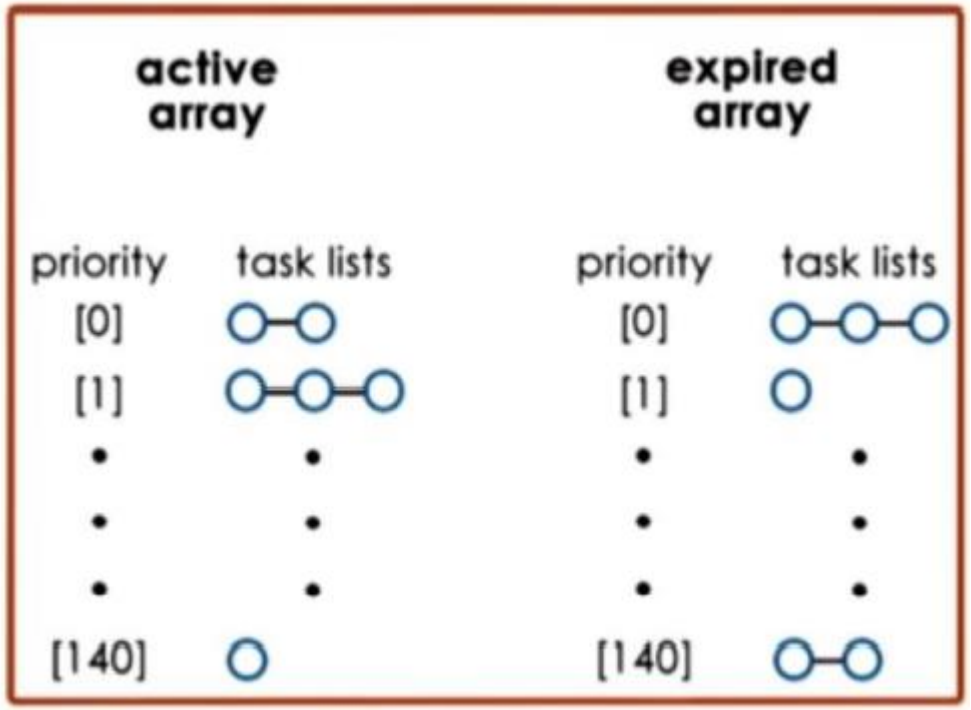
Linux O (1) Scheduler….. Feed-back adjustments

- Longer sleep times indicate that the task is interactive, it's spent more time waiting→ increase the priority of the task by (-5) from the priority level of the task

- Smaller sleep times are indicative of the fact that the task is compute intensive→ lower its priority by (+5) to it up to a maximum

Linux O (1) Scheduler….. Run-queue

- Organized as two arrays of task queues
    - Active array
    - Expired array
- Each array element points to the first runnable task at the corresponding priority level



Run-queue….Active Array

- The primary array← the scheduler will use to pick the next task to run

- It takes constant time to select/add a task
    - index into this array based on the priority level of the task and then follow the pointer to the end of the task list to queue the task there

- It takes constant time to select a task because the scheduler relies on certain instructions that return the position of the first set bit in a sequence of bits

Active Array in Details

- If the sequence of bits corresponds to the priority levels and a bit value of 1 indicates that there are tasks at that priority level
- Then, it will take a constant amount of time to run those instructions to detect what is the first priority level that has certain tasks on it
- And then, once that position is known, it also takes a constant time to index into this array and select the first task from the run-queue that’s associated with that level
- If tasks yield the CPU to wait on an event or are preempted due to higher priority task becoming runnable<u>. The time they spend on the CPU is subtracted from the total amount of time</u>, and if it is less than the time-slice they’re still placed on the corresponding queue in the active list

Expired Arrays

- Only after a task consumes its entire time-slice will it be removed from the active list and placed on the appropriate queue in the expired array
- The expired array contains the tasks that are not currently active. In the sense that the scheduler will not select tasks from the expired array as long as there are still tasks on any of the queues in the active array
- When there are no more tasks left in the active array, at that point, the pointers of these two list will be swapped, and the expired array will become the new active one and vice versa. The new expired array will start holding all the tasks that are to be removed from the active array because they were inactive

Expired Arrays…..continued

- Once tasks placed on the expired array, they will not be scheduled. And therefore, we want the low priority to have a low time-slice value so that yes, they will get a chance to run
- However, they won’t disrupt the higher priority tasks, they won’t delay them by too much
- Also note that the fact that we have these two arrays also serves like an aging mechanism so these high priority tasks will ultimately consume their time-slice and placed on the expired array and ultimately, the low priority tasks will get a chance to run for their small-time amount

Linux O (1) Performance

- Despite its nice property of being able to operate in constant time, the O (1) scheduler really affected the performance of interactive tasks significantly
- Increased workload introduced jitter

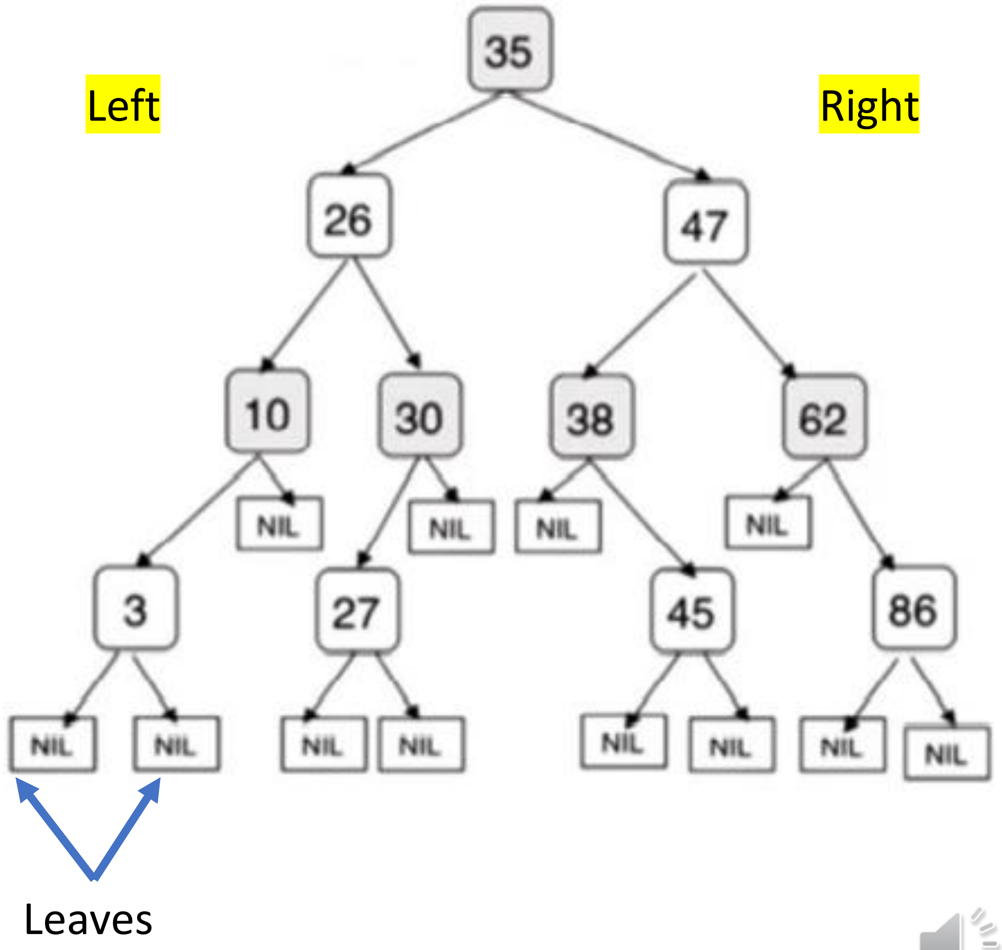
- The fix : Linux O(1) was replaced by CFS scheduler that became the default scheduler starting in the Linux 2.6.23 kernel

Linux CFS Scheduler

- Problems with O(1)
  - Performance of interactive tasks
  - Fairness
- Fairness→ in a given time interval; all of the tasks should be able to run for an amount of time that is proportional to their priority
- The main idea behind the Completely Fair Scheduler is that it uses a so-called a Red-Black Tree as a Run-queue Structure

Linux CFS Scheduler

- Dynamic tree structure that have a property that as nodes are added or removed from the tree, the tree will self balance so that all the paths from the root to the leaves of the tree are approximately of the same size
- Virtual run-time→ Tasks are ordered in the tree based on the amount of time that they spend running on the CPU



CFS Scheduling in Details

- CFS tracks this virtual runtime in a nanosecond granularity. Each of the internal nodes in the tree corresponds to a task, and the nodes to the left of the task correspond to those tasks which had less time on the CPU, they had spent less virtual time
- Nodes with less virtual time they need to be scheduled sooner
- The children to the right of a node are those that have consumed more virtual time, more CPU time. They don’t have to be scheduled as quickly as the left side nodes
- The leaves in the tree, really don’t play any role in the scheduler

CFS Scheduling in Details .... Continued

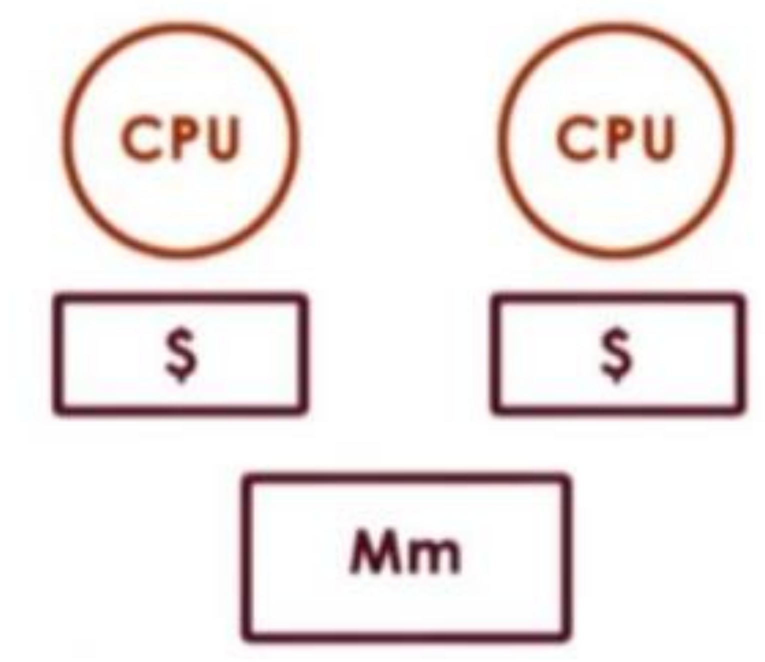
- CFS always schedules the task which had the least amount of time on the CPU, so that typically would be the left most node in the tree
- Periodically CFS will increment the vrun-time of the task that’s currently executing on the CPU. And at that point, it will compare the virtual runtime of the currently running task to the vrun-time of the leftmost tasks in the tree
- If the currently running task has a smaller vrun-time compared to the one that’s in the leftmost node in the tree, the currently running task will continue executing. Otherwise, it will be preempted, and it will be placed in the appropriate location in the tree

- To account for differences in the task priorities or in their niceness value. CFS changes the effective rate at which the task’s virtual time progresses
- Lower priority tasks, time passes more quickly. Their virtual run time value progresses faster → they will likely lose their CPU more quickly, because their virtual run time will increase, compared to other tasks in the system
- High priority tasks, time passes more slowly. Their virtual runtime value will progress at a much slower rate, and therefore, they will get to execute on the CPU longer

Summery

- Rate faster for low priority
- Rate slower for high priority
- Same tree for all priorities
- Selecting a task from this run queue to execute takes O of 1 time. Takes constant amount of time since it's typically just a matter of selecting the leftmost node in the tree→ O(1)
- Adding a task to the run queue takes logarithmic time relative to the total number of tasks in the system. Given the typically levels of load in current system, this log of n time is acceptable→ O(log N)

Scheduling on Multiprocessors

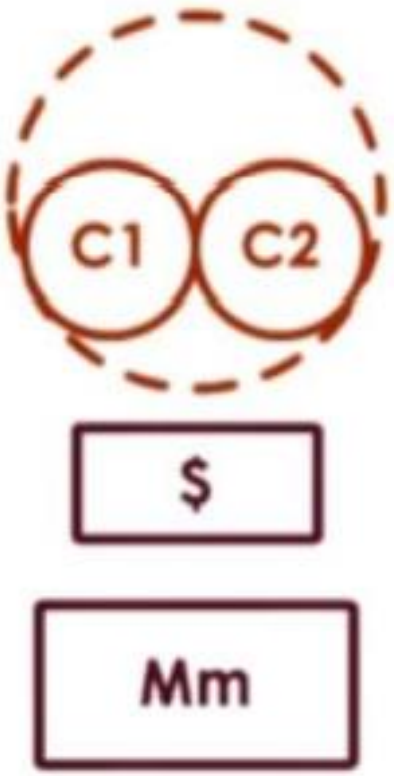


←CPUs and their private / on-chip caches (L1, L2…)

←Last level cache (LLC)

←Memory (DRAM)

Shared memory Multi-processor (SMP)



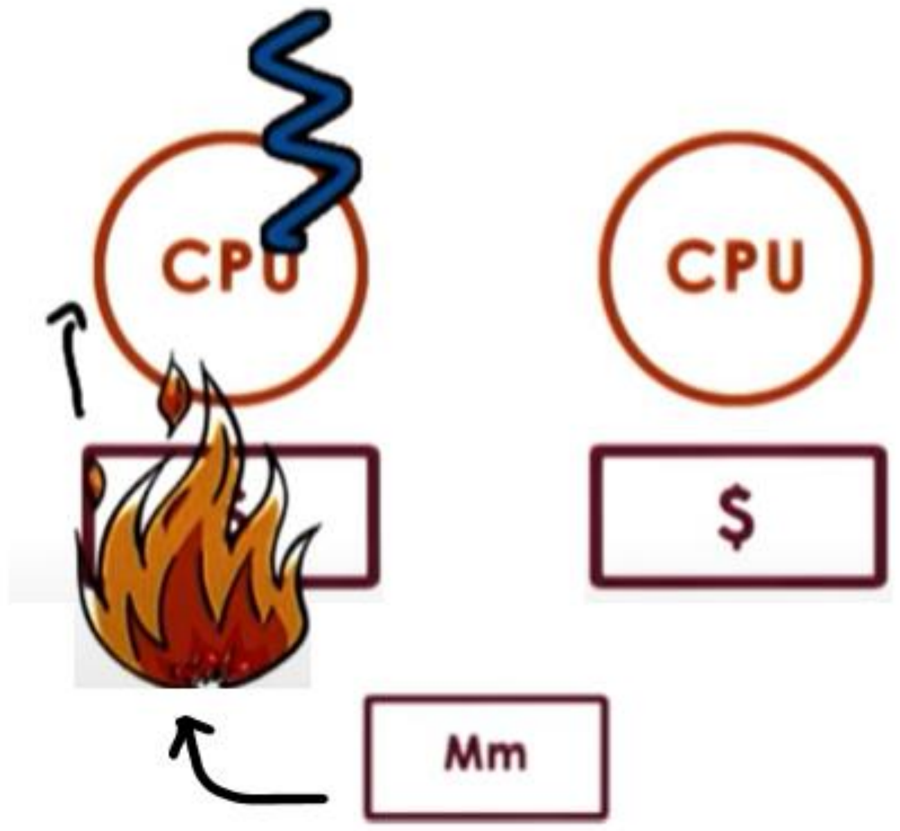
CPU with multiple cores

←Cores have private L1....

←Shared LLC

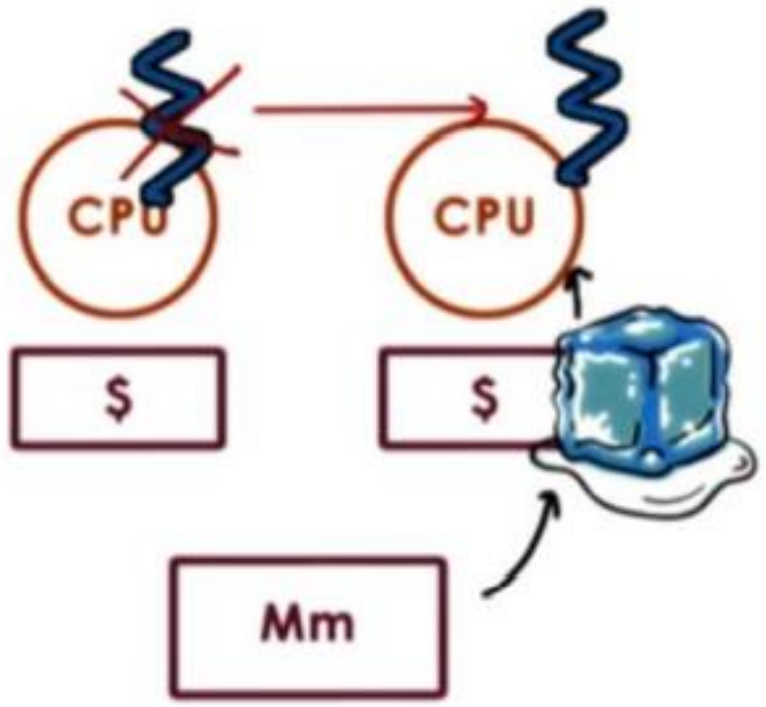
←Memory

Scheduling on Multi-CPU Systems



- Thread was executing on CPU 1 first
- Over time this thread was slightly able to bring a lot of the state that it needs in LLC and the private cache available on the CPU
- When caches are hot, this helps with the performance

Scheduling on Multi-CPU Systems



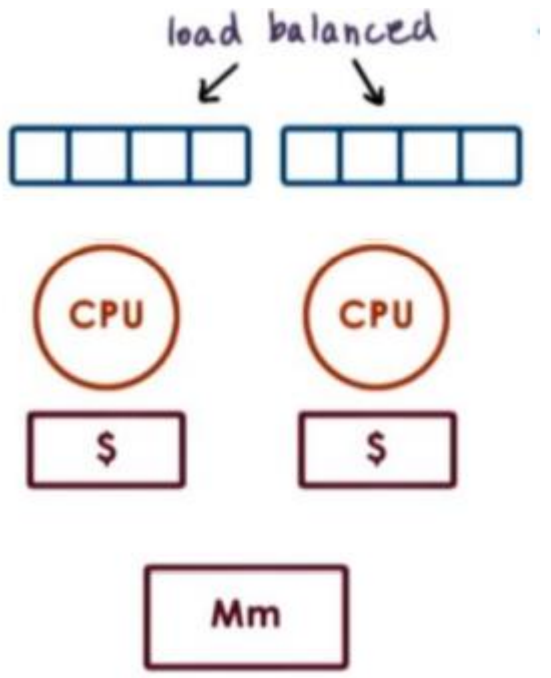
- The next time around, if the thread is scheduled to execute on the other CPU, none of its state will be there so the thread will operate with a cold cache
- We'll have to bring all the state back in and that will affect performance
- The goal →try to schedule the thread back on the same CPU where it executed before for HOT cache

Cache Affinity

- The goal → try to schedule the thread back on the same CPU where it executed before for HOT cache

- Maintain a hierarchical scheduler architecture, where at the top level is a load balancing component divides the tasks among CPUs

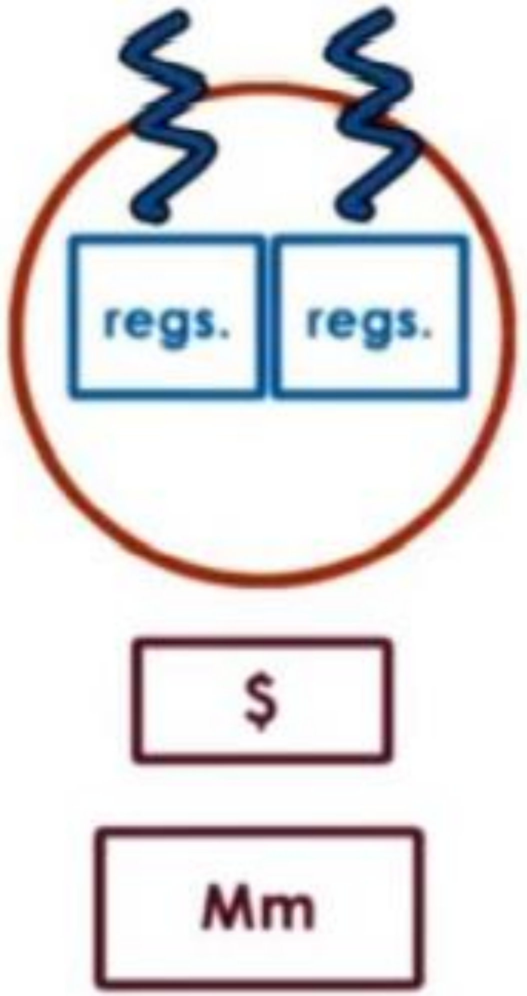
- Per-CPU scheduler with a per CPU run-queue repeatedly schedules those tasks on a given CPU as much as possible



- To balance the load across different CPUs and their per-CPU run-queue, the top-level entity in the scheduler can look at information such as the length of each of these queues to decide how to balance tasks across them
- Or potentially when a CPU is idle, it can at that point start looking at the other CPUs and try to get some more work from them
- In addition to having multiple processors, it is possible to also have multiple memory nodes, and the CPUs and the memory nodes will be interconnected via some type of interconnect
- We call these types of platforms non-uniform memory access platforms, or NUMA platforms and the schedulers are NUMA-aware scheduling

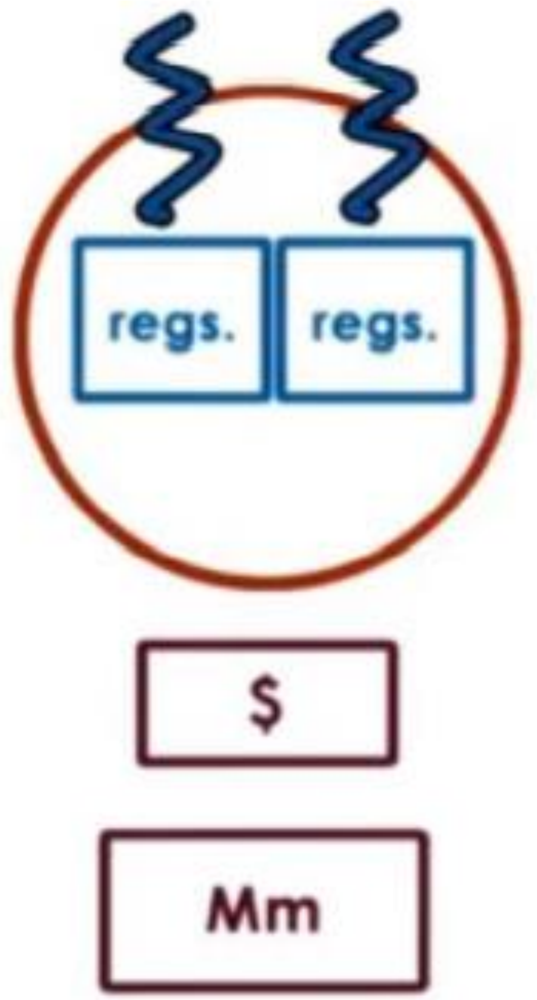
Hyperthreading

- We context switch among threads because the CPU has one set of registers to describe the active execution context for the thread that's currently running on the CPU e.g., SP, PC …
- One way to hide overheads associated with context switching is to have CPUs that have multiple set of registers, that each set of register can describe the context of a separate thread (separate execution entity)→ Hyperthreading



Hyperthreading

- Hyperthreading is multiple hardware that supports execution context among threads
- There's still just one CPU, so on this CPU only one of these threads can execute at a particular moment of time. However, the context switching between these threads is very fast.
- And just basically the CPU needs to switch from using this set of registers to another set of registers. Nothing has to be saved or restored



Hyperthreading OR….

- This mechanism is referred to by multiple names:
    - Hardware multithreading
    - Hyperthreading
    - Chip multithreading (CMT)
    - Simultaneous multithreading (SMT)

More on Hardware Multithreading

- You can enable or disable this hardware multithreading at boot time. Note trade-offs associated with this as always

- Enabled: each of these hardware contexts appears to the operating system’s scheduler as a separate context, a separate virtual CPU onto Which it can schedule threads given that it can load the registers with the thread context concurrently

SMT Decision Making

- Remember **If** $(\boldsymbol{t - idle}) > \mathbf{2} * (\boldsymbol{t - ctx - switch})$ **then context switch to hide idling time** (week 2-1)
- The time to perform a context switch among the two hardware threads is in the order of cycles. And the time to perform a memory access operation is in the order of hundreds of cycles, so it’s much greater→ then context switch
- This technology hyperthreading will help us even hide the memory access latency that threads are experiencing

More on Hyperthreading

- Hyperthreading rises other requirements when we’re trying to decide what kinds of threads should we co-schedule on the hardware threads in the CPU
- Chip Multithreaded Processors Need a New Operating System Scheduler

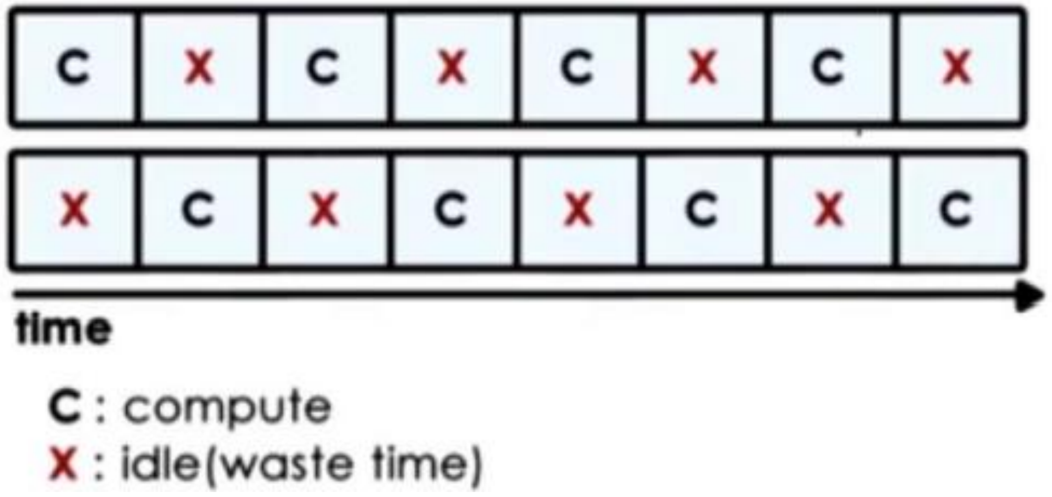
Scheduling for Hyperthreading Platforms

- Federova`s paper Assumptions
  1. Threads issue instruction on each cycle→ CPU bound thread, a thread issues instructions that need to run on the CPU will be able to achieve a maximum metric in terms of instructions per cycle

     We cannot have an IPC greater than one 'instruction per cycle'
  2. Memory access takes four cycles→ memory-bound thread will experience some idle cycles while it's waiting for the memory access to return
  3. Time takes to context switch among the different hardware threads is instantaneous→ no overheads into consideration
  4. Let's also assume that we have a SMT with two hardware threads

Case 1 .. Co-schedule on Two Hardware Contexts

- Two threads that are both compute-bound, so compute intensive, or CPU bound. But one can execute at a given time WHY?
- As a result, these two threads will interfere with each other. They will be competing for the CPU pipeline resources. And best case, every one of them will basically spend one cycle idling while the other thread issues its instruction

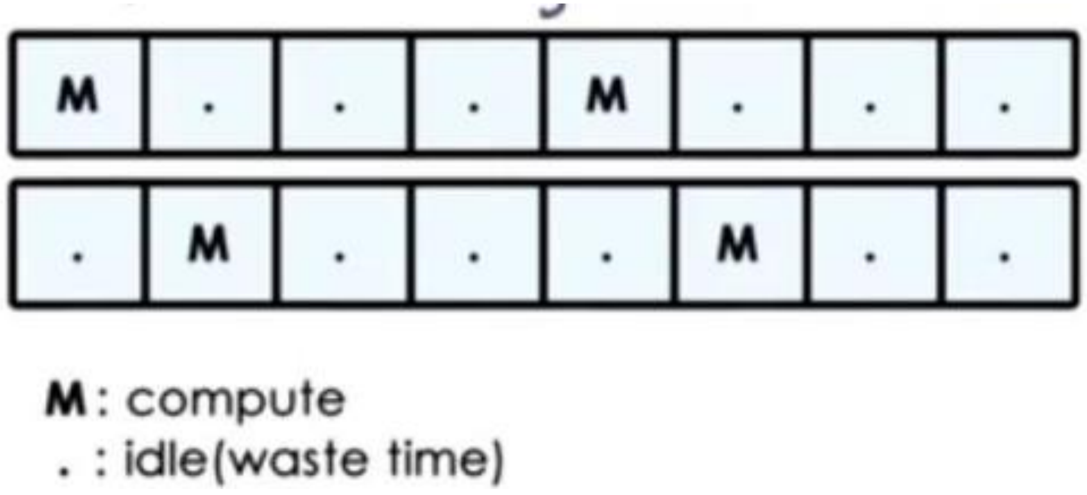


Case 1 … Results

- For each of the threads, its performance will degrade by a factor of two
- Furthermore, looking at the entire platform, we will notice that in this particular case our memory component, the memory controller, they’re idle. There’s nothing that’s scheduled that performs any kind of memory accesses. Well, that’s not good either

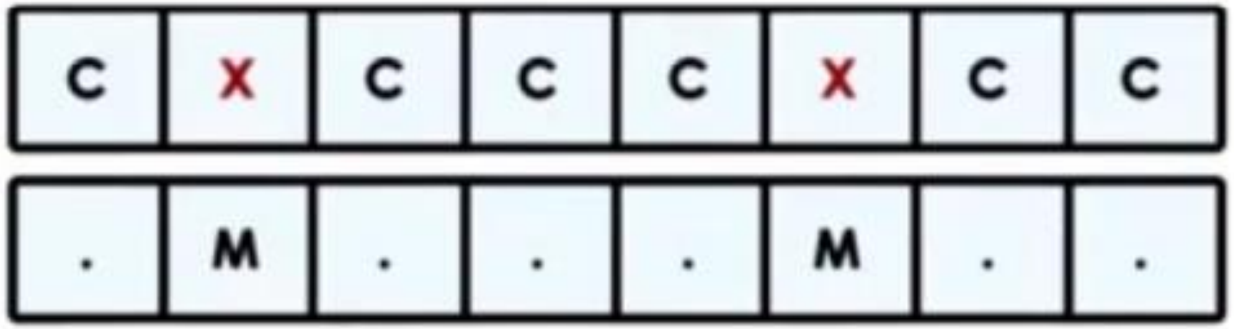
Case 2…. Co-schedule to Memory-bound Threads

- This will result some idle cycles because both threads end up issuing co-memory operations
- Threads need to wait four cycles until it returns. Therefore, we have two of the cycles that are unused. So, the strategy then to co-schedule memory bound threads leads to wasted CPU cycles



Case 3…. mix CPU and Memory Intensive Threads

- Fully utilizing each processor cycle → whenever there is a thread that needs to perform a memory reference, we context switch to that thread in hardware
- The thread issues the memory reference, and then we context switch back to the CPU intensive thread until the memory reference completes



Case 3 … Results

- Scheduling a mix of CPU and memory-intensive threads, allows us to avoid or at least limit the contention on the processor pipeline
- And then also, all the components, both the CPU and the memory will be well utilized
- Note that this still will lead to some level of degradation due to the interference between these two threads. However, this level of the degradation will be minimal given the properties of the underlying system architecture

CPU Bound or Memory Bound:

- How do we know if a thread is CPU bound versus a memory bound?
- To answer → use historic information

No answer yet …

- Sleep time differentiation won`t work:
- Thread is not really sleeping when it’s waiting on a memory reference, the thread is active and it’s just waiting in some stage in the processor pipeline and not on some type of software queue
- Software methods used to keep track of the sleep time, and that’s not acceptable. We cannot execute in software some computations to decide whether a thread is CPU bound or memory-bound
- Given the fact that the context switch takes order of cycles, so the decision what to execute should be very, very fast. Therefore, we somehow need hardware support to answer this question

The Fix

- Hardware counters: updated as the processor is executing and keep information about various aspects of the execution
  - Cache usage L1, L2…
  - LLC misses
  - # of instructions were retired
  - Power or energy usage→ used to compute IPC or other parameters

- Hardware is not uniform among different platforms, so how can hardware counters help a scheduler to make any kinds of scheduling decision?

(G)Estimate what kind of resources a thread needs

- A scheduler can look at a counter like the last level cash misses and using this counter, a scheduler can decide that a thread is memory bound, so its footprint doesn't fit in the cash
- Or the same counter can also tell the scheduler that something changed in the execution of the thread so that now it's executing with some different data in a different phase of its execution, and it's running with the cold cash

one counter can tell us different things about a thread

Continued…

- Scheduler can make informed decisions:
  - Typically, multiple counters
  - Different models with different architecture thresholds
  - Based on well understood workloads

The End

- Pop_Q slide 42