Scheduling in Different Systems

Rushikesh Dumbre

I. Linux

- Linux scheduling is based on the time-sharing technique i.e. several processes are allowed to run "concurrently“
- An **I/O-bound process** spends much of its time submitting and waiting on I/O requests. Such a process is runnable for only short durations, because it eventually blocks waiting on more I/O.
- **Processor-bound processes** spend much of their time executing code. Thet tend to run until they are preempted because they do not block on I/O requests very often.
- The runnable process with timeslice remaining and the highest priority always runs.

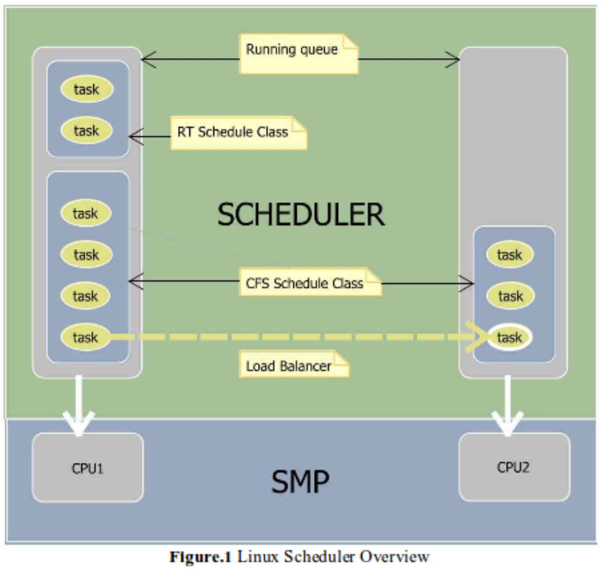


Figure 1. Linux Scheduling

- **Running queue:** a running queue (rq) is created for each processor (CPU)
- There are two schedule classes:
  - **Completely Fair Schedule class:** schedules tasks following Completely Fair Scheduler (CFS) algorithm. Tasks which have policy set to SCHED_NORMAL (SCHED_OTHER), SCHED_BATCH, SCHED_IDLE are scheduled by this schedule class.
  - **RealTime schedule class:** schedules tasks following real-time mechanism defined in POSIX standard. Tasks which have policy set to SCHED_FIFO, SCHED_RR are scheduled using this schedule class.
  - **Load balancer:** If running queues are unbalanced, load balancer will try to pull idle tasks from busiest processors to idle processor.
- **Static Priority:**
  - **Nice value** (a number from –20 to +19 with a default of 0) is the standard priority range:
    - Processes with a lower nice value (higher priority) receive a larger proportion of the system's processor, and vice versa.
    - If the two are the only processes with same nice values, each would be guaranteed half of the processor's time
    - In Linux, the nice value is a control over the proportion of timeslice.
- **Real-time priority** (configurable values that by default range from 0 to 99):
  - Higher real-time priority values correspond to a greater priority.
  - All real-time processes are at a higher priority than normal processes. A value of “-” means the process is not real-time.

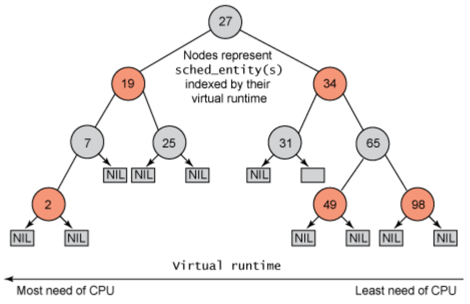


Figure 2. CFS

- **Virtual Runtimes :**
  - Each process has virtual runtime(vruntime) in PCB
  - If a process executes for 't' ms, then vruntime += t
- At timer interrupt (Round Robin time slice) select process with minimum vruntime
- CFS uses Red-Black tree data structure
- Nodes on left are processes with low vruntime compared with those on right
- min_vruntime variable points to the leftmost node in the tree
- Dynamic time slices is interrupted based on vruntime
- At timer interrupt, executing process reinserted into tree and the next leftmost process is picked
- Same process may be picked again
- Total Time Taken = O(log n)
- RBT is used as it is self balancing and all operations take O(log n)
- vruntime += t*(weight based on nice values); This is how priority is implemented

- I/O processes need high priority; This is achieved automatically as I/O processes' vruntime increases very slowly
- New process put at leftmost node.

II. Unix

- Unix uses multilevel feedback queues.

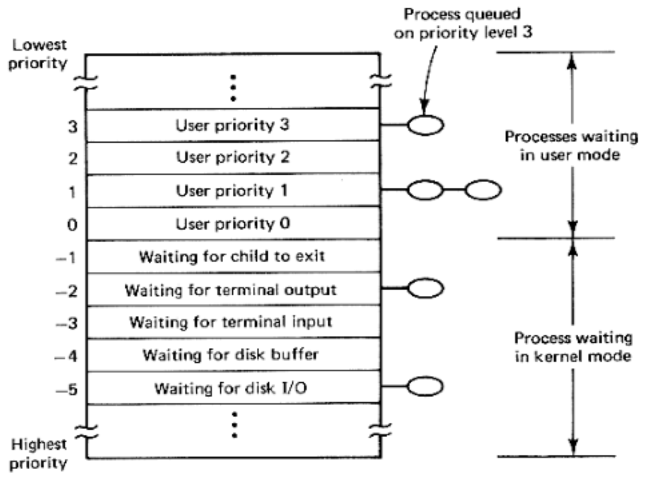

Process queued on priority level 3
Lowest priority
3 User priority 3
2 User priority 2
1 User priority 1
0 User priority 0
−1 Waiting for child to exit
−2 Waiting for terminal output
−3 Waiting for terminal input
−4 Waiting for disk buffer
−5 Waiting for disk I/O
Highest priority
Processes waiting in user mode
Process waiting in kernel mode

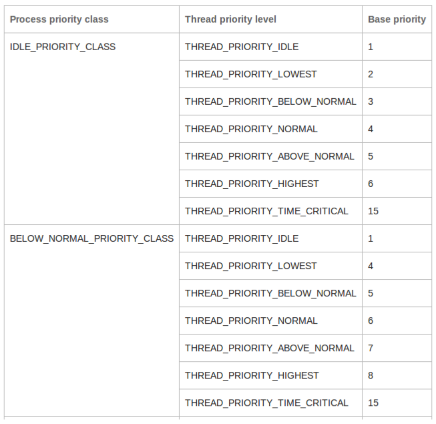
Figure 3. Unix Priority Queues

- All runnable processes are assigned a scheduling priority that determines which queue they are placed in.
- Each queue uses **Round Robin**. Priorities are dynamically adjusted.
- When a nonrunning process with a higher priority becomes available, it immediately preempts the current running process if the current process is in user mode, otherwise (kernel mode) it switches when the current process exits the kernel.
- Unix processes have a priority value called **niceness**, which ranges from -20 (highest priority) to +20 (lowest priority).
- The default value for a user process is zero. A user can run a process with a lower priority with the nice command at the shell.
- It is called nice because users who use this are being nice to other users by running their jobs at a lower priority. Only the superuser is allowed to increase the priority of a job.
- Short term priority of user processes is based on two values, recent cpu utilization and niceness. Every four clock ticks (40 msec) priority is recalculated based on this equation.
  - **priority = PUSER + (recentcputime/4) + 2 * Niceness**
  - recentcputime is incremented each time that the system clock ticks, and the process is found to be executing.
  - PUSER is a constant.
- **recentcputime = ((2 * load) / (2 * load + 1)) * recentcputime + nice**
  - load is the average length of the run queue.
  - Thus, this function uses a decay function as described above with the rate of decay determined by system load. When the system is lightly loaded, decay is rapid, when the system is heavily loaded, decay is slower.
- Each type of job has its own priority. This is determined by the constant in the equation.
- User jobs have the value PUSER, Processes running in kernel mode have a higher priority than user jobs, so the constant is higher.

III. Windows

- Threads are scheduled to run based on their scheduling priority.
- Each thread is assigned a scheduling priority. The priority levels range from zero (lowest priority) to 31 (highest priority).
- Only the zero-page thread can have a priority of zero. (The zero-page thread is a system thread responsible for zeroing any free pages when there are no other threads that need to run.)
- The system treats all threads with the same priority as equal.
- The system assigns time slices in a round-robin fashion to all threads with the highest priority.
- If none of these threads are ready to run, the system assigns time slices in a round-robin fashion to all threads with the next highest priority.
- If a higher-priority thread becomes available to run, the system ceases to execute the lower-priority thread (without allowing it to finish using its time slice), and assigns a full time slice to the higher-priority thread.
- The priority of each thread is determined by the following criteria:
  - The priority class of its process
  - The priority level of the thread within the priority class of its process
- The priority class and priority level are combined to form the base priority of a thread.
- Each process belongs to one of the following **Priority Classes:**
  - IDLE_PRIORITY_CLASS
  - BELOW_NORMAL_PRIORITY_CLASS
  - NORMAL_PRIORITY_CLASS
  - ABOVE_NORMAL_PRIORITY_CLASS
  - HIGH_PRIORITY_CLASS
  - REALTIME_PRIORITY_CLASS
- By default, the priority class of a process is NORMAL_PRIORITY_CLASS.
- Processes that monitor the system, such as screen savers or applications that periodically update a display, should use IDLE_PRIORITY_CLASS. This prevents the threads of this process, which do not have high priority, from interfering with higher priority threads.

- Use HIGH_PRIORITY_CLASS with care. If a thread runs at the highest priority level for extended periods, other threads in the system will not get processor time.
- If several threads are set at high priority at the same time, the threads lose their effectiveness.
- The high-priority class should be reserved for threads that must respond to time-critical events.
- You should almost never use REAL-TIME_PRIORITY_CLASS, because this interrupts system threads that manage mouse input, keyboard input, and background disk flushing.
- The following are **Priority Levels** within each priority class:
  - THREAD_PRIORITY_IDLE
  - THREAD_PRIORITY_LOWEST
  - THREAD_PRIORITY_BELOW_NORMAL
  - THREAD_PRIORITY_NORMAL
  - THREAD_PRIORITY_ABOVE_NORMAL
  - THREAD_PRIORITY_HIGHEST
  - THREAD_PRIORITY_TIME_CRITICAL
- All threads are created using THREAD_PRIORITY_NORMAL. This means that the thread priority is the same as the process priority class.
- A typical strategy is to use THREAD_PRIORITY_ABOVE_NORMAL or THREAD_PRIORITY_HIGHEST for the process's input thread, to ensure that the application is responsive to the user.
- Background threads, particularly those that are processor intensive, can be set to THREAD_PRIORITY_BELOW_NORMAL or THREAD_PRIORITY_LOWEST, to ensure that they can be preempted when necessary.



Process priority class	Thread priority level	Base priority
IDLE_PRIORITY_CLASS	THREAD_PRIORITY_IDLE	1
	THREAD_PRIORITY_LOWEST	2
	THREAD_PRIORITY_BELOW_NORMAL	3
	THREAD_PRIORITY_NORMAL	4
	THREAD_PRIORITY_ABOVE_NORMAL	5
	THREAD_PRIORITY_HIGHEST	6
	THREAD_PRIORITY_TIME_CRITICAL	15
BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE	1
	THREAD_PRIORITY_LOWEST	4
	THREAD_PRIORITY_BELOW_NORMAL	5
	THREAD_PRIORITY_NORMAL	6
	THREAD_PRIORITY_ABOVE_NORMAL	7
	THREAD_PRIORITY_HIGHEST	8
	THREAD_PRIORITY_TIME_CRITICAL	15

Figure 4. Windows Process Scheduling

IV. VMWare

- There are different levels of cache available to a core.
- **Last-level cache (LLC)** refers to the slowest layer of on-chip cache beyond which a request is served by the memory

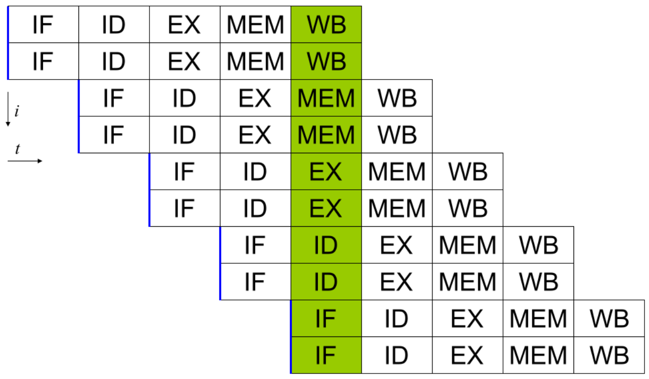


Figure 5. Superscalar Architecture

- **Logical Cores** are the abilities of a single core to do 2 or more things simultaneously
  - For each processor core that is physically present, the operating system addresses two virtual (logical) cores and shares the workload between them when possible.
  - The main function of hyper-threading is to increase the number of independent instructions in the pipeline
  - It takes advantage of superscalar architecture, in which multiple instructions operate on separate data in parallel.
  - HyperThreading needs Simultaneous Multithreading (SMT) support in OS
  - SMT is implemented using multiple **Concurrent Threads** per core
- Unlike a traditional dual-processor configuration that uses two separate physical processors, the logical processors in a hyper-threaded core share the execution resources.
- These resources include the execution engine, caches, and system bus interface; the sharing of resources allows two logical processors to work with each other more efficiently, and allows a logical processor to borrow resources from a stalled logical core (assuming both logical cores are associated with the same physical core).
- Hyper-threading works by duplicating certain sections of the processor—those that store the architectural state—but not duplicating the main execution resources.
- This allows the operating system to schedule two threads or processes simultaneously and appropriately.
- This technology is transparent to operating systems and programs. The minimum that is required to take advantage of hyper-threading is symmetric multiprocessing (SMP) support in the operating system, as the logical processors appear as standard separate processors.
  - Symmetric multiprocessing (SMP) involves a multiprocessor computer hardware and software architecture where two or more identical processors are connected to a single, shared main memory, have full access to all input and output devices, and are controlled by a single operating system instance that treats all processors equally.

- A virtual machine is assigned a number of virtual CPUs, or vCPUs.

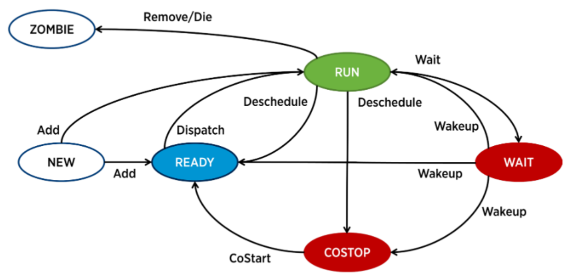


Figure 6. vCPU State Transition Diagram

- Time quantum expiration or the state change from RUN to WAIT invokes the CPU scheduler , which then searches for the next ready vCPU to be scheduled.
- The ready vCPU can be found from a local ready queue or from a remote queue.
- If none is found, an idle vCPU is scheduled.
- When a vCPU wakes up and changes its state from WAIT to READY , likely from an interrupt or a signal, the CPU scheduler is invoked to determine where the newly ready vCPU can be scheduled.
- It may be put into the ready queue waiting to be scheduled in the next iteration , migrated to a different pCPU with less contention, or the currently running world may be preempted.

A. Proportional Share-Based Algorithm

- If the target processor is already occupied, the scheduler needs to determine whether or not to preempt the currently running vCPU on behalf of the chosen one.
- VMWare implements a proportional share–based algorithm, which allocates CPU resources to vCPUs based on their resource specifications.
- Users can specify the CPU allocation using shares, reservations, and limits.
- A vCPU may not fully consume the entitled amount of CPU due to CPU contention (competition for limited resource).
- When making scheduling decisions, the ratio of the consumed CPU resources to the entitlement is used as the priority of the vCPU.
- If there is a world that has consumed less than its entitlement, the world is considered high priority and will likely be chosen to run next.
- It is crucial to accurately account for how much CPU time each world has used. Accounting for CPU time is also called charging.
- One way to understand prioritizing by the CPU scheduler is to compare it to the CPU scheduling that occurs in UNIX. The key difference between CPU scheduling in UNIX and ESXi involves how a priority is determined.
- In UNIX , a priority is arbitrarily chosen by the user. If one process is considered more important than others, it is given higher priority. Between two priorities, it is the relative order that matters, not the degree of the difference.
- In ESXi, a priority is dynamically re-evaluated based on the consumption and the entitlement. The user controls the entitlement, but the consumption depends on many factors including scheduling, workload behavior, and system load.
- Also, the degree of the difference between two entitlements dictates how much CPU time should be allocated.
- The proportional share-based scheduling algorithm has a few benefits over the priority-based scheme:
  - First, users can accurately control the CPU allocation of virtual machines by giving a different number of shares.
    - For example, if a virtual machine, vm0, has twice as many shares as vm1, vm0 would get twice as much CPU time compared to vm1, assuming both virtual machines highly demand CPU resources. It is difficult to achieve this with the UNIX scheduler because the priority does not reflect the actual CPU consumption.
  - With the proportional share-based scheduler, CPU resource control is encapsulated and hierarchical. Resource pools are designed for such use. The capability of allocating compute resources proportionally and hierarchically in an encapsulated way is quite useful.
    - For example, consider a case where an administrator in a company datacenter wants to divide compute resources among various departments and to let each department distribute the resources according to its own preferences. This is not easily achievable with a fixed priority-based scheme.

B. Relaxed Co-Scheduling

- Co-scheduling executes a set of threads or processes at the same time to achieve high performance.
- Because multiple cooperating threads or processes frequently synchronize with each other, not executing them concurrently would only increase the latency of synchronization.
- Synchronous execution of co-process is achieved using relaxed co-scheduling of the multiple vCPUs of a multiprocessor virtual machine.
- This implementation allows for some flexibility while maintaining the illusion of synchronous progress. It meets the needs for high performance and correct execution of guests.
- For this purpose, the progress of a vCPU is measured where a vCPU is considered making progress when it executes guest instructions or it is in the IDLE state.
- Then, the goal of co-scheduling is to keep the difference in progress between sibling vCPUs, or the " skew ," bounded.

1) Strict Co-Scheduling in ESX 2.x:

- In the strict co-scheduling algorithm, the CPU scheduler maintains a cumulative skew per each vCPU of a multiprocessor virtual machine.
- The skew grows when the associated vCPU does not make progress while any of its siblings makes progress.
- If the skew becomes greater than a threshold, typically a few milliseconds, the entire virtual machine would be stopped (co-stop) and will only be scheduled again (co-start) when there are enough pCPUs available to schedule all vCPUs simultaneously.
- This ensures that the skew does not grow any further and only shrinks. The strict co-scheduling might cause CPU fragmentation.
  - For example, a 4 -vCPU multiprocessor virtual machine might not be scheduled even if there are three idle pCPUs.
  - This results in scheduling delays and lower CPU utilization. It is worth noting that an idle vCPU does not incur co-scheduling overhead even with strict co-scheduling.
- Since there is nothing to be executed in the vCPU, it is always considered making equal progress as the fastest sibling vCPU and does not co-stop the virtual machine. For example, when a single-threaded application runs in a 4 - vCPU virtual machine, resulting in three idle vCPUs, there is no co-scheduling overhead and the virtual machine does not require four pCPUs to be available.

2) Relaxed Co-Scheduling in ESX 3.x a nd Later Versions:

- While in the strict co-scheduling algorithm , the existence of a lagging vCPU causes the entire virtual machine to be co-stopped.
- In the relaxed co-scheduling algorithm , a leading vCPU decides whether it should co-stop itself based on the skew against the slowest sibling vCPU.
- If the skew is greater than a threshold, the leading vCPU co-stops itself.
- Note that a lagging vCPU is one that makes significantly less progress than the fastest sibling vCPU, while a leading vCPU is one that makes significantly more progress than the slowest sibling vCPU.
- By tracking the slowest sibling vCPU, it is now possible for each vCPU to make its own co-scheduling decision independently.
- Like co-stop, the co-start decision is also made individually. Once the slowest sibling vCPU starts progressing, the co-stopped vCPUs are eligible to co-start and can be scheduled depending on pCPU availability.
- This solves the CPU fragmentation problem in the strict co-scheduling algorithm by not requiring a group of vCPUs to be scheduled together.
- In the previous example of the 4 -vCPU virtual machine, the virtual machine can make forward progress even if there is only one idle pCPU available. This significantly improves CPU utilization.
- By not requiring multiple vCPUs to be scheduled together, co-scheduling wide multiprocessor virtual machines becomes efficient.
- Finding multiple available pCPUs would consume many CPU cycles in the strict co-scheduling algorithm.

V. XEN

- Schedulers can be distinguished as **Proportional Share based or Fair Share based**
- Proportional share schedulers aim to provide an instantaneous form of sharing among the active clients according to their weights.
- In contrast, fair-share schedulers attempt to provide a time-averaged form of proportional sharing based on the actual use measured over long time periods.
  - For example, consider a simple situation where two clients C1 and C2 share a system with equal CPU shares.
  - Suppose C1 is actively computing for some time, while C2 is temporarily inactive (e.g., blocked).
  - When C2 becomes active, a fair-share scheduler will allocate a large CPU share to C2 to “catch up” with C1.
  - In contrary, a proportional CPU scheduler will treat C1 and C2 equally because it is “unfair” to penalize C1 for consuming otherwise idle resources.
- CPU schedulers can be further distinguished as supporting **work-conserving (WC-mode)** and/or **non work-conserving (NWC -mode)** modes.
- In the WC-mode, the shares are merely guarantees, and the CPU is idle if and only if there is no runnable client.
- It means that in a case of two clients with equal weights and a situation when one of these clients is blocked, the other client can consume the entire CPU.
- With the NWC-mode, the shares are caps, i.e., each client owns its fraction of the CPU. It means that in a case of two clients with equal weights, each client will get up to 50% of CPU, but the client will not be able to get more than 50% even if the rest of the CPU is idle.
- We also distinguish **preemptive** and **non-preemptive** CPU schedulers.
- Preemptive schedulers rerun the scheduling decision whenever a new client becomes ready.
- If the new client has “priority” over the running client, the CPU preempts the running client and executes the new client.
- Non-preemptive schedulers only make decisions when the running client gives up the CPU.
- Non-preemptive schedulers allow every running client to finish its CPU slice. Having a preemptive scheduler is important for achieving good performance of I/O intensive workloads in shared environment.

A. Borrowed Virtual Time (BVT)

- It is a fair-share scheduler based on the concept of virtual time, dispatching the runnable VM with the smallest virtual time first.

- Additionally, BVT provides low-latency support for real-time and interactive applications by allowing latency-sensitive clients to "warp" back in virtual time to gain scheduling priority.
- The client effectively "borrows" virtual time from its future CPU allocation.
- In summary, BVT has the following features:
  - **preemptive (if warp is used), WC-mode only;**
  - optimally-fair: the error between fair share and actual allocation is never greater than context switch allowance C plus one mcu (minimum charging unit);
  - low-overhead implementation on multiprocessors as well as uniprocessors.

B. Simple Earliest Deadline First (SEDF)

- It uses real-time algorithms to deliver guarantees.
- Each domain specifies its CPU requirements with a tuple (s ,p ,x) , where the **slice s** and the **period p** together represent the CPU share that Dom requests:
  - Dom i will receive at least s units of time in each period of length p.
- The **boolean flag x** indicates whether domain is eligible to receive extra CPU time (WC-mode).
- SEDF distributes this slack time fairly manner after all runnable domains receive their CPU share.
- One can allocate 30% CPU to a domain by assigning either (3 ms, 10 ms, 0) or (30 ms, 100 ms, 0).
- In summary, SEDF has the following features:
  - **preemptive, WC and NWC modes;**
  - fairness depends on a value of the period.
  - implements per CPU queue: this implementation lacks global load balancing on multiprocessors.

C. Credit Scheduler

- It is Xen's latest PS scheduler featuring automatic load balancing of virtual CPUs across physical CPUs on an SMP host.
  - When a CPU doesn't find a vCPU of priority UNDER on its local run queue, it will look on other CPUs for one.
  - This ensures load balancing automatically.
- Before a CPU goes idle, it will consider other CPUs in order to find any runnable VCPU.
- This approach guarantees that no CPU idles when there is runnable work in the system.
- Each VM is assigned a **weight** and a **cap**.
- Weight is used to assign ratios of CPU time (between 1 to 65535).
- If the cap is 0, then the VM can receive any extra CPU (WC-mode).
- A non-zero cap (expressed as a percentage) limits the amount of CPU a VM receives (NWC-mode).
  - 100 is 1 physical CPU, 50 is half a CPU, 400 is 4 CPUs, etc. The default, 0, means there is no cap.
- The Credit scheduler uses 30 ms time slices for CPU allocation. A VM (VCPU) receives 30 ms before being preempted to run another VM.
- Once every 30 ms, the priorities (credits) of all runnable VMs are recalculated. The scheduler monitors resource usage every 10 ms.
  - As a VCPU runs, it consumes credits. A system-wide accounting thread recomputes how many credits each active VM has earned and bumps the credits. Negative credits imply a priority of **OVER.** Until a vCPU consumes its alloted credits, it priority is **UNDER.**
- In summary, Credit has the following features:
  - **non-preemptive (using context-switch rate limiting), WC and NWC modes;**
  - global load balancing on multiprocessors.