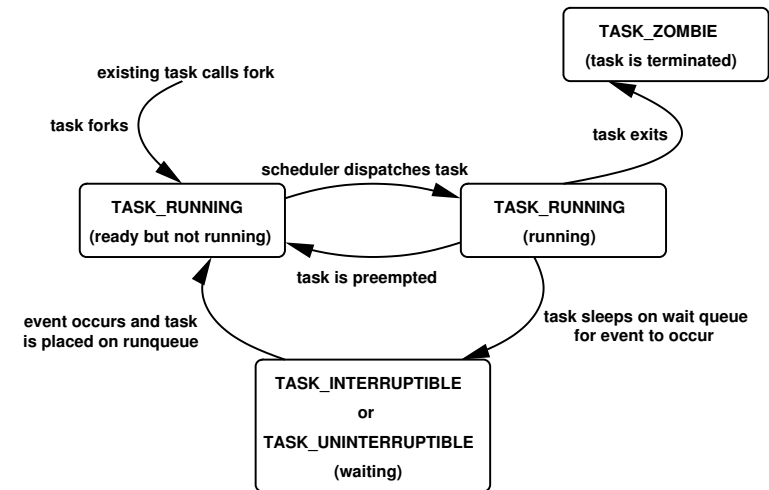


## Overview

- We take a fairly detailed look at how the 2.6 Linux scheduler is implemented
- We highlight the improvements the Linux 2.6 scheduler brings compared to earlier versions
- Material drawn from *Understanding the Linux 2.6.8.1 CPU Scheduler*, by Asas



## Process State Transitions



## Static Priority

- Stored in the PCB of each process is its static priority or `nice` value
- Static priorities range from  $[-20, 19]$  and the higher the value the lower the priority (the more `nice` it is to other processes)
- Static priorities are inherited across `fork` and have a default value of zero
- A user can modify the static priority of a process with the `nice` command but only the superuser can lower the value
- This priority is static because it will not change over the lifetime of the process (not without manual intervention by the user)



## Dynamic priority

- Also stored in the PCB of each process is its dynamic priority, a value occupying the range  $[0, 139]$
- The higher the value the lower the priority
- The Linux scheduler decides which process to dispatch next based on this dynamic priority value
- So what role does static priority play?
  1. Dynamic priority is initially calculated from static priority (adding 120 to the static priority maps it into  $[0, 139]$ )
  2. Timeslice is derived directly from static priority with a longer timeslice allocated to higher priority tasks
- Unlike static priority, dynamic priority changes over the lifetime of the process as interactivity is factored in



## Runqueues

- Associated with each processor (Linux supports SMP machines) is a list of processes maintained in a runqueue data structure
- To avoid deadlock, code that manipulates runqueues must always acquire the corresponding locks in ascending order
- A runqueue contains two priority arrays which allow the scheduler to efficiently:
  1. Find the highest priority runnable process (irrespective of the load on the system)
  2. Ensure all processes are scheduled and none are starved of CPU time



## Priority Arrays

- A runqueue contains two priority arrays, the active array and the expired array
- The active array is populated with processes which have timeslice remaining
- The expired array is populated with processes which have exhausted their timeslice
- Each array contains three members:
  1. The number of tasks in the array
  2. A bitmap depicting runnable priority levels
  3. An array of lists, one for each priority level (140)

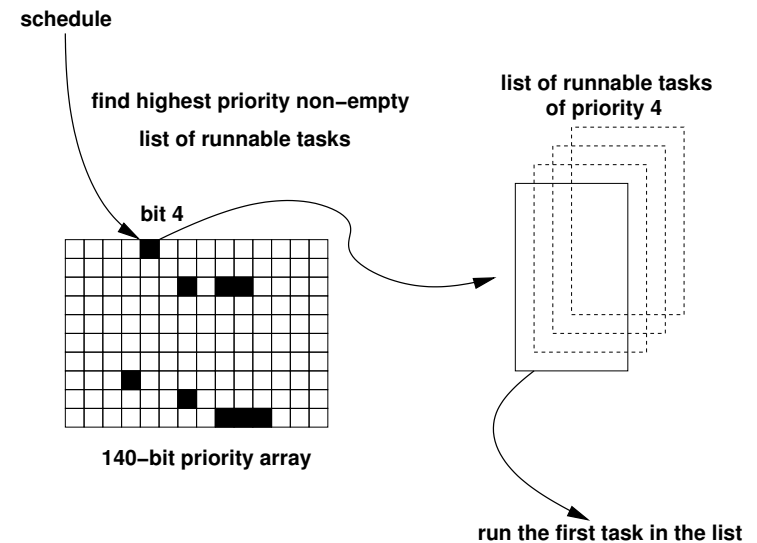


## Priority Arrays

- If a bit is turned on in the bitmap then the corresponding process list is non-empty
- To find the highest priority runnable process the scheduler searches the active array bitmap for the first set bit
- The scheduler then dispatches the process at the head of the corresponding process list
- Crucially, there is a single  $x86$  instruction that returns the first set bit in a bitmap (and it completes in constant time)
- When a process has exhausted its timeslice it is moved to the expired array
- If a process is preempted by a higher priority one but still has timeslice remaining it is appended to the corresponding priority list



## Priority Arrays



## Scheduler Characteristics

- If an “execution round” is the time taken to run all processes until their timeslice is exhausted (when they are moved to the expired array) then. . .
- Per execution round, high priority tasks will run longer than low priority ones (due to their longer timeslices)
- Per execution round, high priority tasks will run more often than low priority ones (due to their higher dynamic priority and preemption)
- Per execution round, every process gets to run, even low priority ones so no one can starve as eventually all processes use up their allotted timeslice
- Selecting the next process to run is  $O(1)$  (was  $O(N)$  in 2.4)



## Swapping Arrays

- When a process has used up its timeslice it must be moved to the expired array so that other lower priority processes get a chance to run
- Before being moved its timeslice is recalculated for the next execution round
- When all processes have exhausted their timeslice the active array is empty
- Active and expired arrays are then swapped, active becomes expired, expired becomes active and a new execution round can begin
- Swapping merely entails switching two pointers



## Calculating Priorities and Timeslices

- Timeslices are simply derived from static priority: the higher the priority the longer the timeslice
- Linux tries to give a higher dynamic priority to interactive tasks
- This ensures that per execution round an I/O bound process will preempt CPU bound processes
- So the scheduler somehow needs to measure the interactivity of a process if it is to reward it
- Interactive processes sleep more than others



## Rewarding Interactive Processes

- The scheduler keeps track of how long a process has been waiting on I/O in an entry in its PCB, `sleep_avg`
- When a process wakes the time spent sleeping is added to its `sleep_avg`
- Static priority and `sleep_avg` are combined to calculate dynamic priority
- Over time `sleep_avg` is decremented so that only regular sleepers get a priority boost
- Thus dynamic priority is based on a user supplied value (the `nice` value) giving the user some but not total control since task interactivity must be factored in



## Timeslices

- Lowest priority tasks get 5ms, highest priority get 800ms and middle priority get 100ms
- Problem: if a highly interactive task exhausts its timeslice early in a round it will be moved to the expired array. . .
- If it were to remain there until all currently active processes had exhausted their timeslice responsiveness might suffer (and the user become displeased)
- Therefore, highly interactive tasks are not moved to the expired array unless there are already some “close to starving” tasks in the expired array
- To satisfy the starving processes a new execution round must begin and therefore even the highly interactive processes must be moved into the expired array



## Scheduler Performance

- The 2.4 scheduler suffered from poor scalability
- It implemented algorithms that were  $O(N)$  i.e. its running time (time taken to dispatch the highest priority runnable process) was proportional to the number of processes on a runqueue (albeit linearly proportional)
- The 2.6 scheduler is  $O(1)$  i.e. its running time is constant and is independent of the number of processes in a runqueue
- It takes a fixed time interval to locate the highest priority process, recalculate a timeslice, switch arrays etc.
- This means the 2.6 scheduler scales well to a server context

