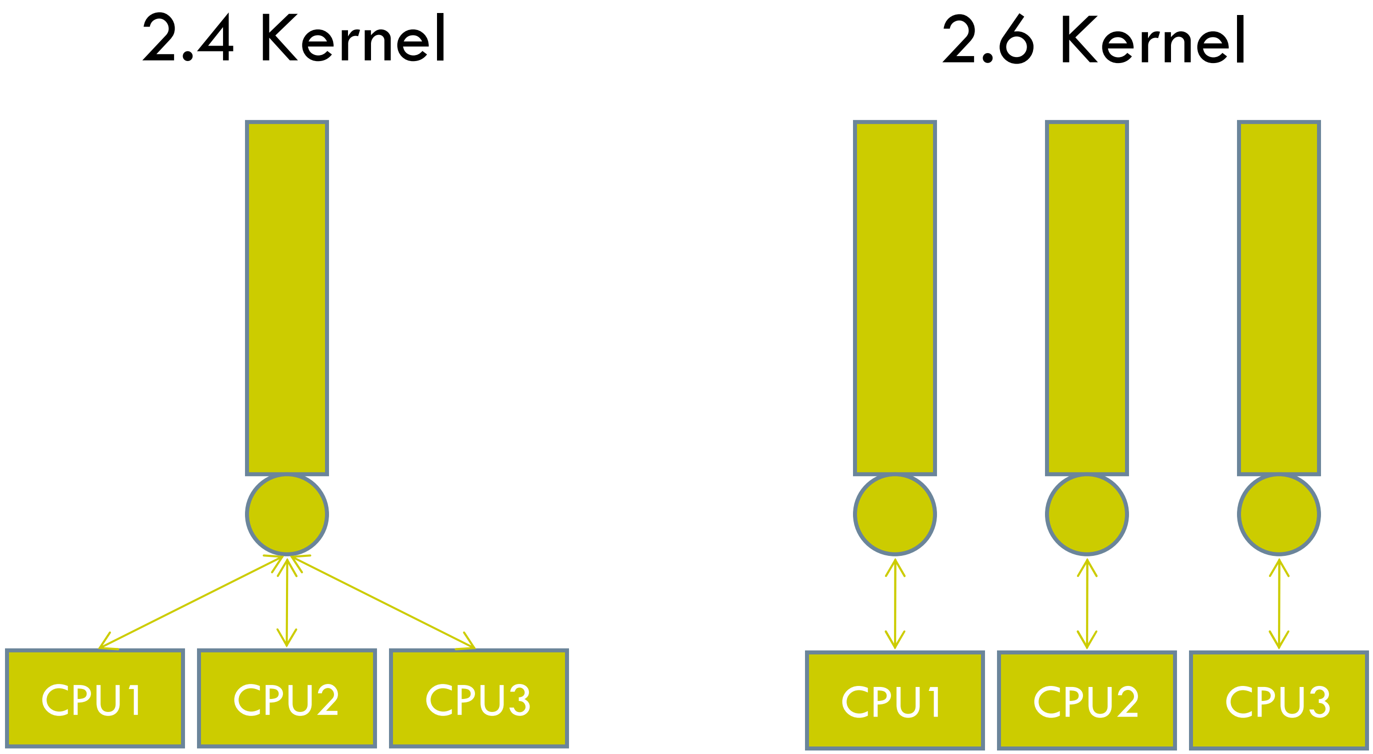
CPU Scheduling

Linux scheduler history

Linux version	**Scheduler**
Linux pre 2.5	Multilevel Feedback Queue
Linux 2.5-2.6.23	O(1) scheduler
Linux post 2.6.23	Completely Fair Scheduler

We will be talking about the O(1) scheduler

SMP Support in 2.4 and 2.6 versions



Linux Scheduling

- 3 scheduling classes
  - SCHED_FIFO and SCHED_RR are real-time classes
  - SCHED_OTHER is for the rest
- 140 Priority levels
  - 1-100 : RT priority
  - 101-140 : User task priorities
- Three different scheduling policies
  - One for normal tasks
  - Two for Real time tasks

- Pre-emptive, priority based scheduling.
- When a process with higher real-time priority (rt_priority) wishes to run, all other processes with lower real-time priority are thrust aside.
- In SCHED_FIFO, a process runs until it relinquishes control or another with higher real-time priority wishes to run.
- SCHED_RR process, in addition to this, is also interrupted when its time slice expires or there are processes of same real-time priority (RR between processes of this class)
- SCHED_OTHER is also round-robin, with lower time slice

- SCHED_OTHER: Normal tasks
  - Each task assigned a “Nice” value
  - Static priority = 120 + Nice
    - Nice value between -20 and +19
  - Assigned a time slice
  - Tasks at the same priority are round-robined
    - Ensures Priority + Fairness

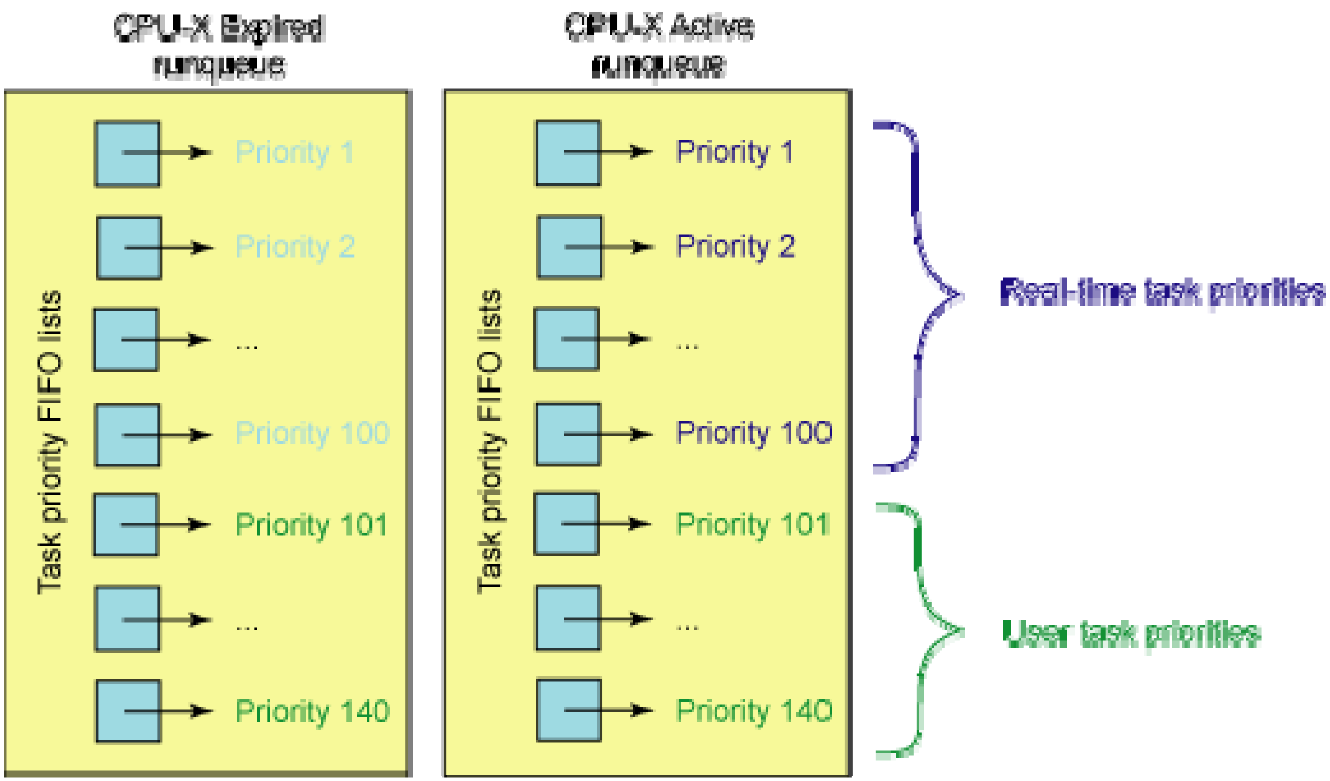
Basic Philosophies

- Priority is the primary scheduling mechanism
- Priority is *dynamically adjusted* at run time
  - Processes denied access to CPU get increased
  - Processes running a long time get decreased
- Try to distinguish interactive processes from non-interactive
  - Bonus or penalty reflecting whether I/O or compute bound
- Use large quanta for important processes
  - Modify quanta based on CPU use
- Associate processes to CPUs
- Do everything in O(1) time

The Runqueue

- 140 separate queues, one for each priority level
- Actually, two sets, active and expired
- Priorities 0-99 for real-time processes
- Priorities 100-139 for normal processes; value set via nice()/setpriority() system calls

Linux 2.6 scheduler runqueue structure



Scheduler Runqueue

- A scheduler runqueue is a list of tasks that are runnable on a particular CPU.
- A *rq* structure maintains a linked list of those tasks.
- The runqueues are maintained as an array *runqueues*, indexed by the CPU number.
- The *rq* keeps a reference to its idle task
  - The idle task for a CPU is never on the scheduler runqueue for that CPU (it's always the last choice)
- Access to a runqueue is serialized by acquiring and releasing *rq->lock*

Basic Scheduling Algorithm

- Find the highest-priority queue with a runnable process
- Find the first process on that queue
- Calculate its quantum size
- Let it run
- When its time is up, put it on the expired list
  - Recalculate priority first
- Repeat

Process Descriptor Fields Related to the Scheduler

- thread_info->flags
- thread_info->cpu
- state
- prio
- static_prio
- run_list
- array
- sleep_avg
- timestamp
- last_ran
- activated
- policy
- cpus_allowed
- time_slice
- first_time_slice
- rt_priority

The Highest Priority Process

- There is a bit map indicating which queues have processes that are ready to run
- Find the first bit that's set:
  - 140 queues → 5 integers
  - Only a few compares to find the first that is non-zero
  - Hardware instruction to find the first 1-bit
    - bsfl on Intel
  - Time depends on the number of priority levels, not the number of processes

Scheduling Components

- Static Priority
- Sleep Average
- Bonus
- Dynamic Priority
- Interactivity Status

Static Priority

- Each task has a static priority that is set based upon the nice value specified by the task.
  - *static_prio* in *task_struct*
  - Value between 0 and 139 (between 100 and 139 for normal processes)
- Each task has a dynamic priority that is set based upon a number of factors
  - tries to increase priority of interactive jobs

Sleep Average

- Interactivity heuristic: sleep ratio
  - Mostly sleeping: I/O bound
  - Mostly running: CPU bound
- Sleep ratio approximation
  - *sleep_avg* in the *task_struct*
  - Range: 0 .. *MAX_SLEEP_AVG*
- When process wakes up (is made runnable), *recalc_task_prio* adds in how many ticks it was sleeping (blocked), up to some maximum value (*MAX_SLEEP_AVG*)
- When process is switched out, *schedule* subtracts the number of ticks that a task actually ran (without blocking)
- *sleep_avg* scaled to a bonus vale

Average Sleep Time and Bonus Values

Average sleep time	Bonus
>= 0 but < 100 ms	0
>= 100 ms but < 200 ms	1
>= 200 ms but < 300 ms	2
>= 300 ms but < 400 ms	3
>= 400 ms but < 500 ms	4
>= 500 ms but < 600 ms	5
>= 600 ms but < 700 ms	6
>= 700 ms but < 800 ms	7
>= 800 ms but < 900 ms	8
>= 900 ms but < 1000 ms	9
1 second	10

Bonus and Dynamic Priority

- Dynamic priority (*prio* in *task_struct*) is calculated in from static priority and bonus
  - = max (100, min(static_priority – bonus + 5, 139))

Calculating Time Slices

- *time_slice* in the *task_struct*
- Calculate Quantum where
  - If (SP < 120): Quantum = (140 − SP) × 20
  - if (SP >= 120): Quantum = (140 − SP) × 5

  where SP is the *static priority*
- Higher priority process get longer quanta
- Basic idea: important processes should run longer
- Other mechanisms used for quick interactive response

Nice Value vs. static priority and Quantum

	Static Priority	NICE	Quantum
High Priority	100	-20	800 ms
	110	-10	600 ms
	120	0	100 ms
	120	+10	50 ms
Low Priority	139	+19	5 ms

$$\text{Quantum} = \begin{cases} (140 - \text{SP}) \times 20 & \text{if SP} < 120 \\ (140 - \text{SP}) \times 5 & \text{if SP} \geq 120 \end{cases}$$

Interactive Processes

- A process is considered interactive if

  bonus − 5 >= (Static Priority / 4) − 28

  - (Static Priority / 4) − 28 = interactive delta
- Low-priority processes have a hard time becoming interactive:
  - A high static priority (100) becomes interactive when its average sleep time is greater than 200 ms
  - A default static priority process becomes interactive when its sleep time is greater than 700 ms
  - Lowest priority (139) can never become interactive
- The higher the bonus the task is getting and the higher its static priority, the more likely it is to be considered interactive.

Using Quanta

- At every time tick (in *scheduler_tick*) , decrement the quantum of the current running process (*time_slice*)
- If the time goes to zero, the process is done
- Check interactive status:
  - If non-interactive, put it aside on the *expired* list
  - If interactive, put it at the end of the *active* list
- Exceptions: don’t put on *active* list if:
  - If higher-priority process is on *expired* list
  - If expired task has been waiting more than *STARVATION_LIMIT*
- If there’s nothing else at that priority, it will run again immediately
- Of course, by running so much, its bonus will go down, and so will its priority and its interactive status

Avoiding Starvation

- The system only runs processes from active queues, and puts them on expired queues when they use up their quanta
- When a priority level of the active queue is empty, the scheduler looks for the next-highest priority queue
- After running all of the active queues, the active and expired queues are swapped
- There are pointers to the current arrays; at the end of a cycle, the pointers are switched

The Priority Arrays

```
struct prio_array {
      unsigned int nr_active;
      unsigned long bitmap[5];
      struct list_head queue[140];
};
struct rq {
      spinlock_t lock;
      unsigned_long nr_running;
      struct prio_array *active, *expired;
      struct prio_array arrays[2];
      task_struct *curr, *idle;
      …
};
```

Swapping Arrays

```
struct prioarray *array =
    rq->active;
if (array->nr_active == 0) {
    rq->active = rq->expired;
    rq->expired = array;
}
```

Why Two Arrays?

- Why is it done this way?
- It avoids the need for traditional aging
- Why is aging bad?
    - It's O(n) at each clock tick

Linux is More Efficient

- Processes are touched only when they start or stop running
- That's when we recalculate priorities, bonuses, quanta, and interactive status
- There are no loops over all processes or even over all runnable processes

Real-Time Scheduling

- Linux has soft real-time scheduling
  - No hard real-time guarantees
- All real-time processes are higher priority than any conventional processes
- Processes with priorities [0, 99] are real-time
  - saved in *rt_priority* in the *task_struct*
  - scheduling priority of a real time task is: 99 - *rt_priority*
- Process can be converted to real-time via *sched_setscheduler* system call

Real-Time Policies

- First-in, first-out: SCHED_FIFO
  - Static priority
  - Process is only preempted for a higher-priority process
  - No time quanta; it runs until it blocks or yields voluntarily
  - RR within same priority level
- Round-robin: SCHED_RR
  - As above but with a time quanta (800 ms)
- Normal processes have SCHED_OTHER scheduling policy

Multiprocessor Scheduling

- Each processor has a separate run queue
- Each processor only selects processes from its own queue to run
- Yes, it's possible for one processor to be idle while others have jobs waiting in their run queues
- Periodically, the queues are rebalanced: if one processor's run queue is too long, some processes are moved from it to another processor's queue

Locking Runqueues

- To rebalance, the kernel sometimes needs to move processes from one runqueue to another
- This is actually done by special kernel threads
- Naturally, the runqueue must be locked before this happens
- The kernel always locks runqueues in order of increasing indexes
- Why? Deadlock prevention!

Processor Affinity

- Each process has a bitmask saying what CPUs it can run on
- Normally, of course, all CPUs are listed
- Processes can change the mask
- The mask is inherited by child processes (and threads), thus tending to keep them on the same CPU
- Rebalancing does not override affinity

Load Balancing

- To keep all CPUs busy, load balancing pulls tasks from busy runqueues to idle runqueues.
- If *schedule* finds that a runqueue has no runnable tasks (other than the idle task), it calls *load_balance*
- *load_balance* also called via timer
  - *schedule_tick* calls *rebalance_tick*
  - Every tick when system is idle
  - Every 100 ms otherwise

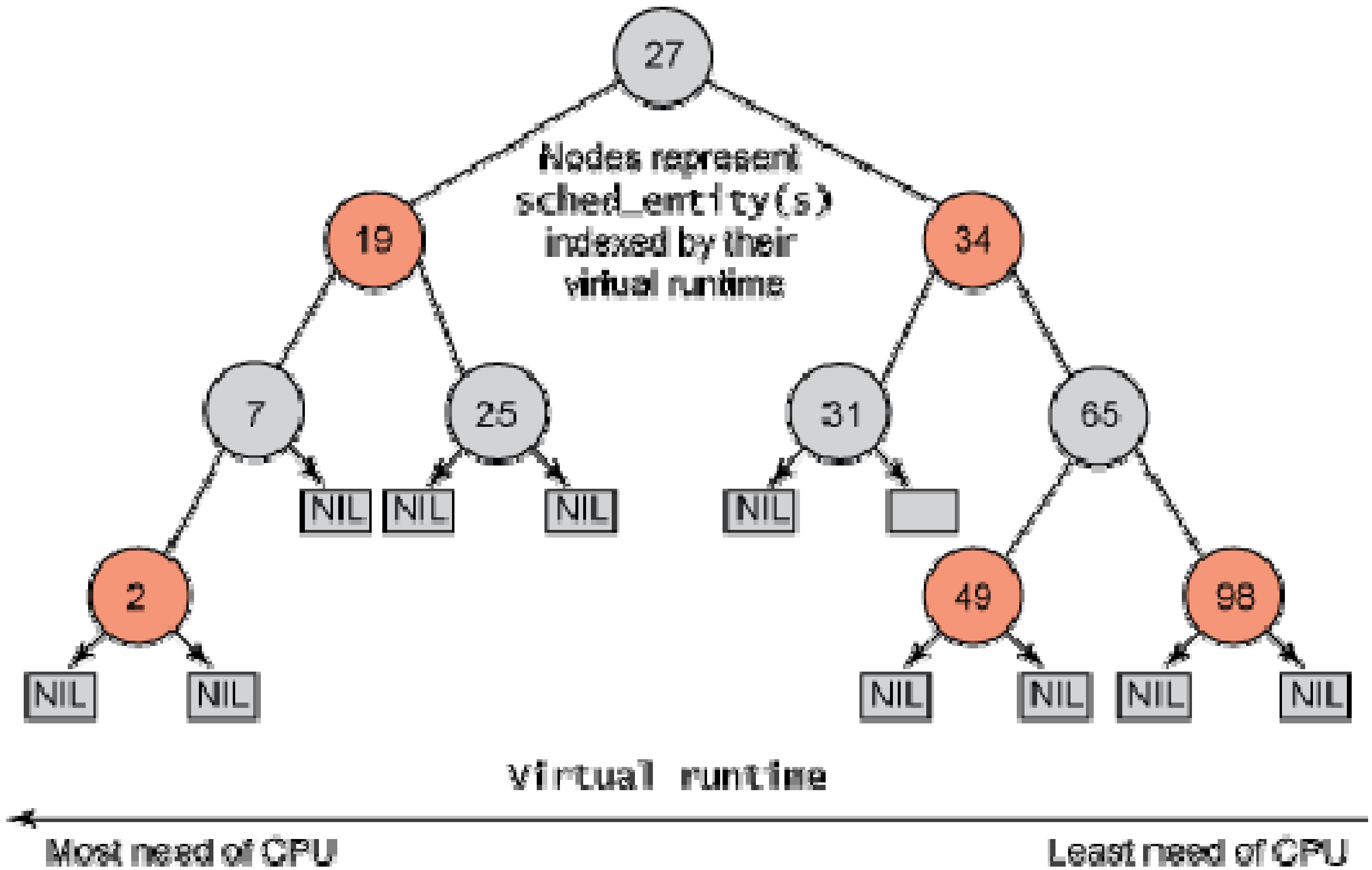
Load Balancing

- *load_balance* looks for the busiest runqueue (most runnable tasks) and takes a task that is (in order of preference):
  - inactive (likely to be cache cold)
  - high priority
- *load_balance* skips tasks that are:
  - likely to be cache warm (hasn't run for *cache_decay_ticks* time)
  - currently running on a CPU
  - not allowed to run on the current CPU (as indicated by the *cpus_allowed* bitmask in the *task_struct*)

Linux 2.6 CFS Scheduler

- Was merged into the 2.6.23 release.
- Uses red-black tree structure instead of multilevel queues.
- Tries to run the task with the "gravest need" for CPU time

Red-Black tree in CFS



Red-Black tree properties

- Self Balance
- Insertion and deletion operation in O(log(n))
  - With proper implementation its performance is almost the same as O(1) algorithms!

The `switch_to` Macro

- **`switch_to()`** performs a process switch from the **`prev`** process (descriptor) to the **`next`** process (descriptor).
- **`switch_to`** is invoked by **`schedule()`** & is one of the most hardware-dependent kernel routines.
  - See **`kernel/sched.c`** and **`include/asm-*/system.h`** for more details.