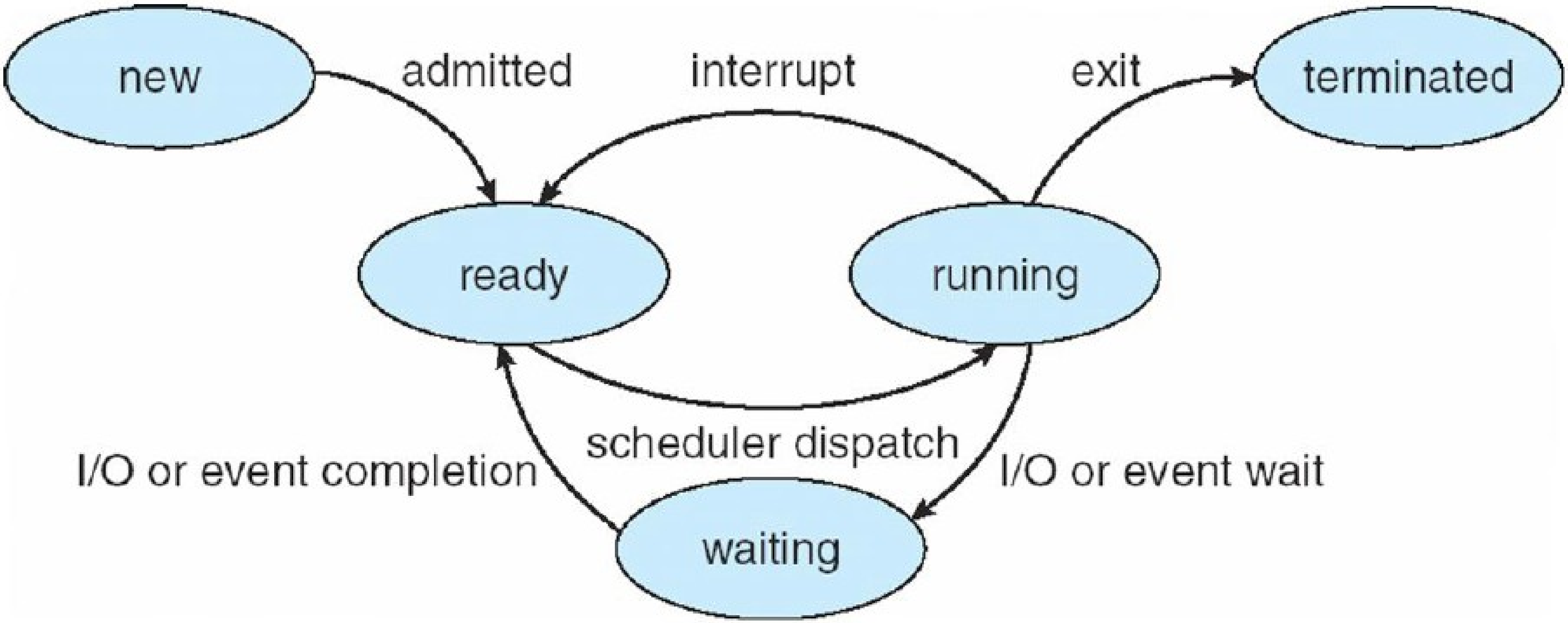
Linux Scheduler

OS-323 (Spring 2013)

Process states



CPU scheduler

- Makes the computer more productive by switching the CPU among processes
- CPU scheduling may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from new or waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- The other scheduling is **preemptive**

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority
- Problem: Starvation – low priority processes may never execute
- Solution: Ageing – as time progresses increase the priority of the ready processes

Round Robin (RR)

- Each process gets allocated a small unit of CPU time (time quantum)
- After this time has elapsed, the process is preempted
- With N processes in the ready queue and the time quantum of q, no process waits more than (N-1)q time units.
- q must be large with respect to context switch, otherwise overhead is too high

Scheduling in Linux

- Initially a circular queue with a round-robin scheduling policy
  - Efficient adding and removing of processes
  - Simple and fast for a small number of tasks
- Linux 2.4: O(N) scheduler
- Linux 2.6: O(1) scheduler
- Linux 2.6: Completely Fair Scheduler (CFS)

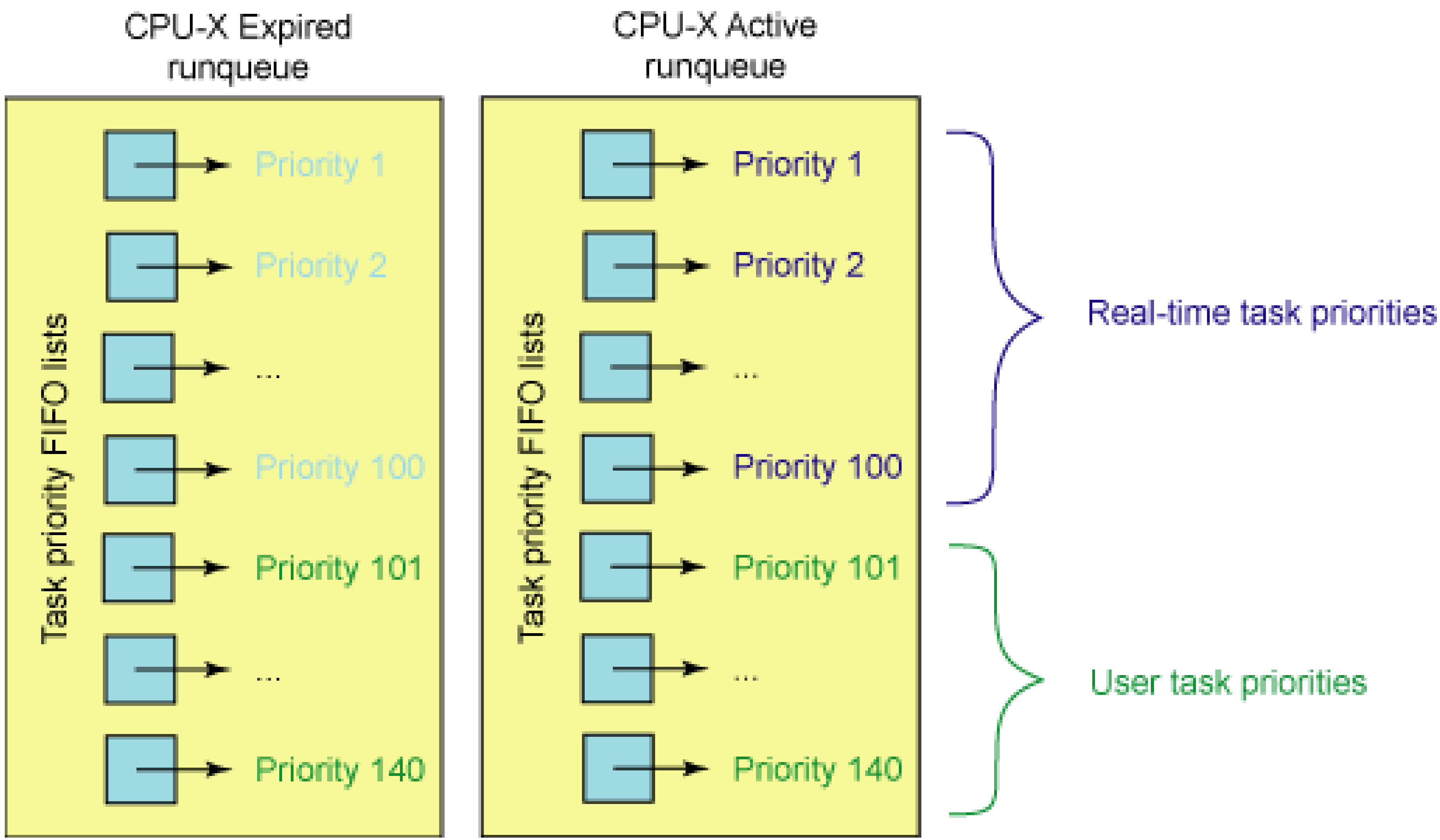
O(N) Scheduler

- Multilevel feedback queue with 140 priority levels
- CPU time divided into epochs
- Within each epoch, every task was allowed to execute up to its time slice
- During each scheduling event, the scheduler would iterate over all tasks, applying a *goodness* function (metric) to determine which task to execute next

O(1) Scheduler

- Scheduling done completely in O(1)
- Runqueue consists of two arrays – active and expired
- Each array consists of 140 linked lists, that are serviced in FIFO order, each representing a priority level
- The first 100 levels are reserved for real-time tasks, and the remaining 40 for user tasks
- Each task is assigned a timeslice based on its priority

O(1) Scheduler



O(1) Scheduler

- Enqueue: add to the end of one of the fixed number of lists
- Dequeue: deletion from a linked list is const. time
- Pick next task: choose the first task in the highest priority list that is not empty. A bitmap with one bit per list is used to quickly determine the first nonempty list.
- Preventing starvation: When a task on the active array uses up its timeslice, it is moved to the expired array. Once all tasks move to the expired array, arrays are swapped.
- All scheduling operations depend only on the number of priority levels, and not the number of tasks

O(1) Scheduler

- SMP support:
  - Per-CPU runqueues: better cache utilization
  - Big-lock architecture replaced with a lock per runqueue
  - Task migration for load balancing

O(1) Scheduler

- Task priorities are dynamically altered during execution (and accordingly timeslices)
- Goal: prioritize interactive tasks over batch tasks
- Interactive tasks typically spend most time sleeping, they have short bursts of CPU activity, before blocking waiting for user input
- A heuristic is used to determine whether a task is interactive based on the ratio of sleep time to execution time

CFS

- Motivated by Rotating Staircase Deadline Scheduler by Con Kolivas which showed that a simple fair scheduler performed better than O(1) scheduler’s interactivity heuristics
- Main idea: Allocate the fair share of CPU time to each of the ready tasks. If there are N ready tasks, each should get 1/N of CPU power.

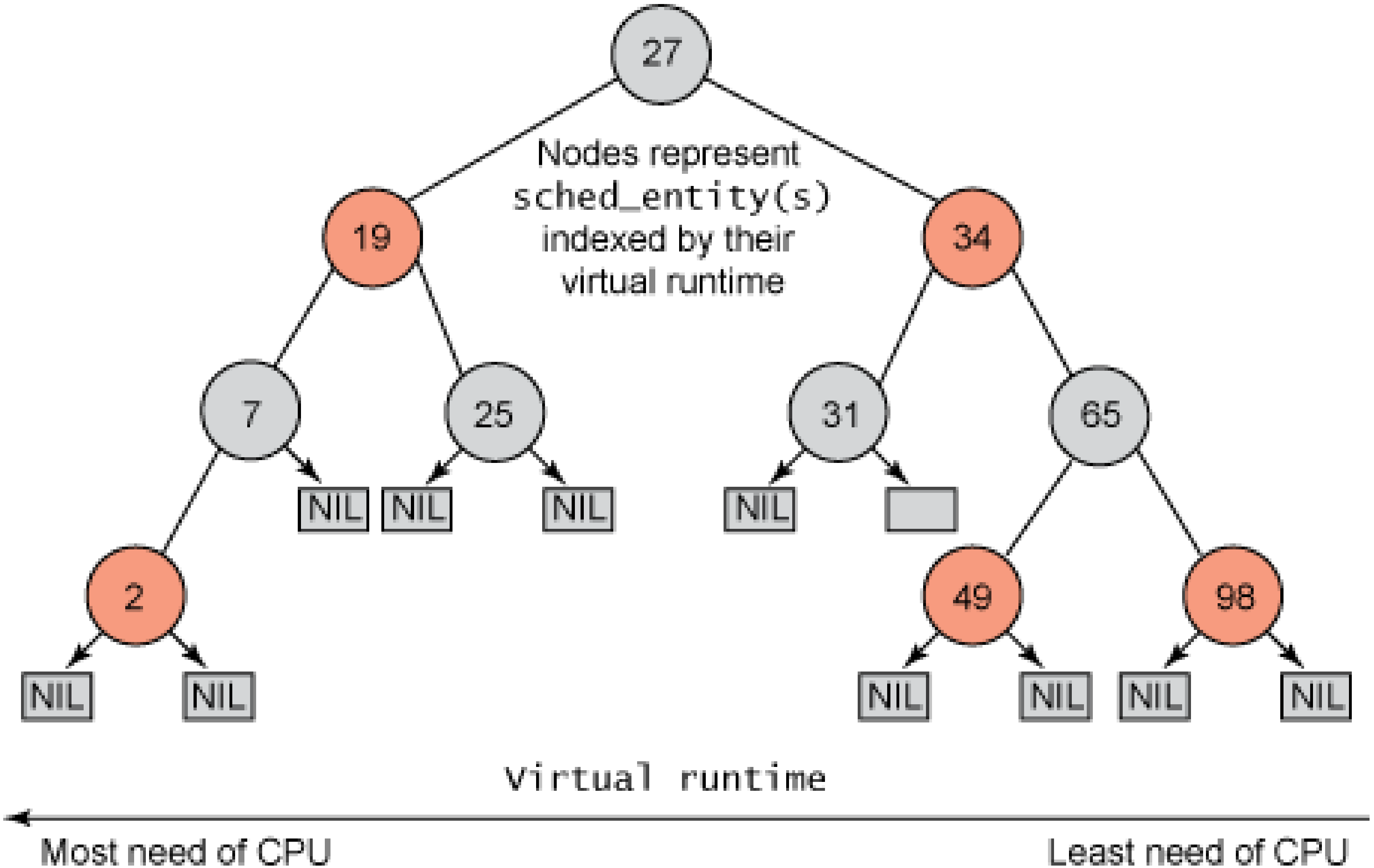
CFS

- Each task has a virtual runtime value, which is its actual runtime normalized to the number of ready tasks
- Task priority is incorporated as a weight factor into this formula
- The CPU is allocated to the task with the smallest virtual runtime value
- The concept of timeslices is removed from CFS
- Instead, there is a scheduling granularity parameter (in ns), that defines how often a new task is chosen

CFS

- The runqueue in CFS is replaced by a red-black tree
- A red-black tree is a self-balancing binary search tree. It has guaranteed O(logN) complexity of insertion, deletion and rebalancing.
- The red-black tree is ordered by task virtual runtime
- The leftmost leaf of the tree always has the smallest virtual runtime and is chosen for scheduling in O(1)
- During execution, its virtual runtime increases and it moves to the right in the tree, so that the next task with the smallest virtual runtime comes to the leftmost position
- Whenever the running task's CPU usage is accounted for, it is reinserted in the tree in O(logN) time

CFS



27
Nodes represent
sched_entity(s)
indexed by their
virtual runtime
19
34
7
25
31
65
NIL
NIL
NIL
NIL
2
49
98
NIL
NIL
NIL
NIL
NIL
NIL
Virtual runtime
Most need of CPU
Least need of CPU

CFS

```
struct cfs_rq{
  struct load_weight load;
  unsigned int nr_running;

  u64 min_vruntime;

  struct rb_root tasks_timeline;
  struct rb_node *rb_leftmost;

  struct sched_entity *curr;
}
```

RT Scheduler

- Schedules tasks for 100 highest priority levels
- The first task in the highest nonempty priority list is chosen
- FIFO policy: The running task executes without preemption
- Round robin policy: The tasks have timeslices, and are preempted and put at the end of the list upon timeslice expiry

Priorities

- Priorities
  - [0-99] are kernel RT priorities
  - [100-139] are kernel user priorities
  - User priorities [100-139] correspond to nice priorities [-20-19]
  - Default priority is kernel 120 (or nice 0)
- static_prio: static priority, can be changed only through system calls (setpriority, nice)
- normal_prio: inverse of static priority used in RT scheduler
- prio: dynamic priority

Scheduling classes

- The CFS scheduler also introduced scheduling classes, an extensible hierarchy of scheduling modules
- Each class is implemented with a sched_class structure that contains hooks to functions to be called when some interesting events occur

Scheduling classes

```
struct sched_class {
  const struct sched_class *next;

  void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
  void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);

  void (*yield_task) (struct rq *rq);
  void (*check_preempt_curr) (struct rq *rq, struct task_struct *p, int flags);

  struct task_struct * (*pick_next_task) (struct rq *rq);
  void (*put_prev_task) (struct rq *rq, struct task_struct *p);
  void (*set_curr_task) (struct rq *rq);

  void (*task_tick) (struct rq *rq, struct task_struct *p, int queued);

  void (*switched_from) (struct rq *this_rq, struct task_struct *task);
  void (*switched_to) (struct rq *this_rq, struct task_struct *task);
  void (*prio_changed) (struct rq *this_rq, struct task_struct *task, int oldprio);
};
```

Scheduling classes

- enqueue_task/dequeue_task
  - Called when a task enters/leaves a ready state
  - Activate/deactivate (new task/terminataion, wakeup/sleep)
  - Changed priority (rt_mutex_setprio, set_user_nice)
  - Changed scheduling policy (sched_setscheduler)
  - Migrating tasks between CPUs
  - Moving tasks between groups
- Flags
  - #define ENQUEUE_WAKEUP 1
  - #define ENQUEUE_HEAD 2
  - #define DEQUEUE_SLEEP 1

Scheduling classes

- yield_task
  - Called as a result of sched_yield system call by the running task
  - Yields CPU to other tasks
- check_preempt_curr
  - Should check if the current task should be preempted by a new ready task and call resched_task if so
  - Called on a (new) task wakeup

Scheduling classes

- pick_next_task
  - Should pick the next task to execute
  - Called from schedule() to get the task to switch to
- put_prev_task
  - Called when the running task is rescheduled
  - Called from schedule() right before pick_next_task
- set_curr_task
  - Called when task priority, group or scheduling policy changes, if the task is running at the time (rt_mutex_setprio, sched_setscheduler)
  - Always called in combination with put_prev_task

Scheduling classes

- task_tick
  - Called on timer with HZ frequency
  - This is the place to do time accounting and to run timeslice-based preemption
- switched_from/switched_to
  - Called when the scheduling class changes (due to the change in priority)
- prio_changed
  - Called when the priority changes, but not the scheduling class (rt_mutex_setprio, sched_setscheduler)

Assignment: Dummy scheduler

- FCFS non-preemptive scheduler with one priority level
- Your task:
  - Priority scheduling with support for 5 priority levels
  - Preemption due to a task of a higher priority becoming available
  - Preemption due to running task's timeslice expiry
  - A mechanism to prevent the starvation of processes with lower priority (ageing)
- To have a task scheduled by the dummy scheduler, you need to assign it a priority in the range [15-19]

UML

- User Mode Linux is a Linux virtual machine that runs on a Linux host
- It is included in the mainline kernel
- Download sources from kernel.org and build using

```
make ARCH=um defconfig

make ARCH=um
```

References

- Understanding the Linux Kernel, Daniel Bovet, Marco Cesati; O’Reilly, 3rd eddition
- Professional Linux Kernel Architecture, Wolfgang Mauerer