Operating Systems (CSE531)
Lecture # 11

Manish Shrivastava

LTRC, IIIT Hyderabad

Past Schedulers: 1.2 & 2.2

- 1.2: circular queue with round-robin policy.
  - Simple and minimal.
  - Not focused on massive architectures.
- 2.2: introducing scheduling classes (real-time, non-preemptive, non-real-time).
  - Support SMP.

Past Schedulers : 2.4

- 2.4: O(N) scheduler.
  - Epochs → slices: when blocked before the slice ends, half of the remaining slice is added in the next epoch (as an increase in priority).
  - Iterate over tasks with a goodness function.
  - Simple, inefficient.
  - Lacked scalability.
  - Weak for real-time systems.

Past Schedulers: 2.4

- Static priority:
  - The maximum size of the time slice a process should be allowed before being forced to allow other processes to compete for the CPU.
- Dynamic priority:
  - The amount of time remaining in this time slice; declines with time as long as the process has the CPU.

Past Schedulers: O(1)

- An independent runqueue for *each* CPU
  - Active array
  - Expired array
- Tasks are indexed according to their priority [0,140]
  - Real-time [0, 99]
  - Nice value (others) [100, 140]
- When the active array is empty, the arrays are exchanged.

Past Schedulers: O(1)

- Real-time tasks are assigned static priorities.
- All others have dynamic priorities:
  - *nice* value ± 5
  - Depends on the tasks interactivity: more interactivity means longer blockage.
- Dynamic priorities are recalculated when tasks are moved to the expired array.

Scheduling Goals

- O(1) scheduling; 2.4 scheduler iterated through
  - Run queue on each invocation
  - Task queue at each epoch
- Scale well on multiple processors
  - per-CPU run queues
- SMP affinity
- Interactivity boost
- Fairness
- Optimize for one or two runnable processes

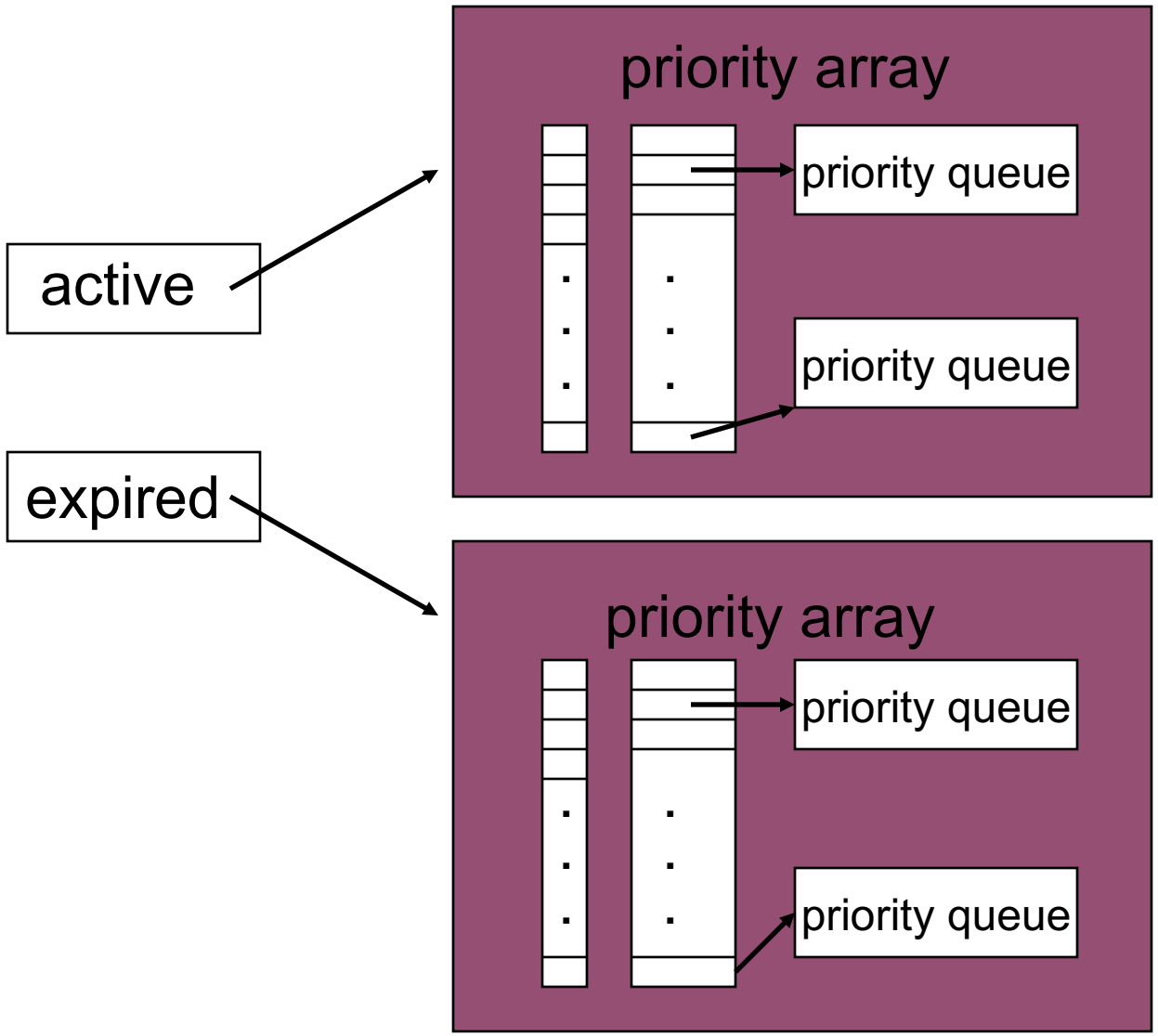
Basic Philosophies

- 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
  - Quantum != clock tick
- Associate processes to CPUs
- Do everything in O(1) time

The Run Queue

- 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() system call

Runqueue for O(1) Scheduler



Higher priority
more I/O
800ms quanta

lower priority
more CPU
10ms quanta

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
- Repeat

The Highest Priority Process

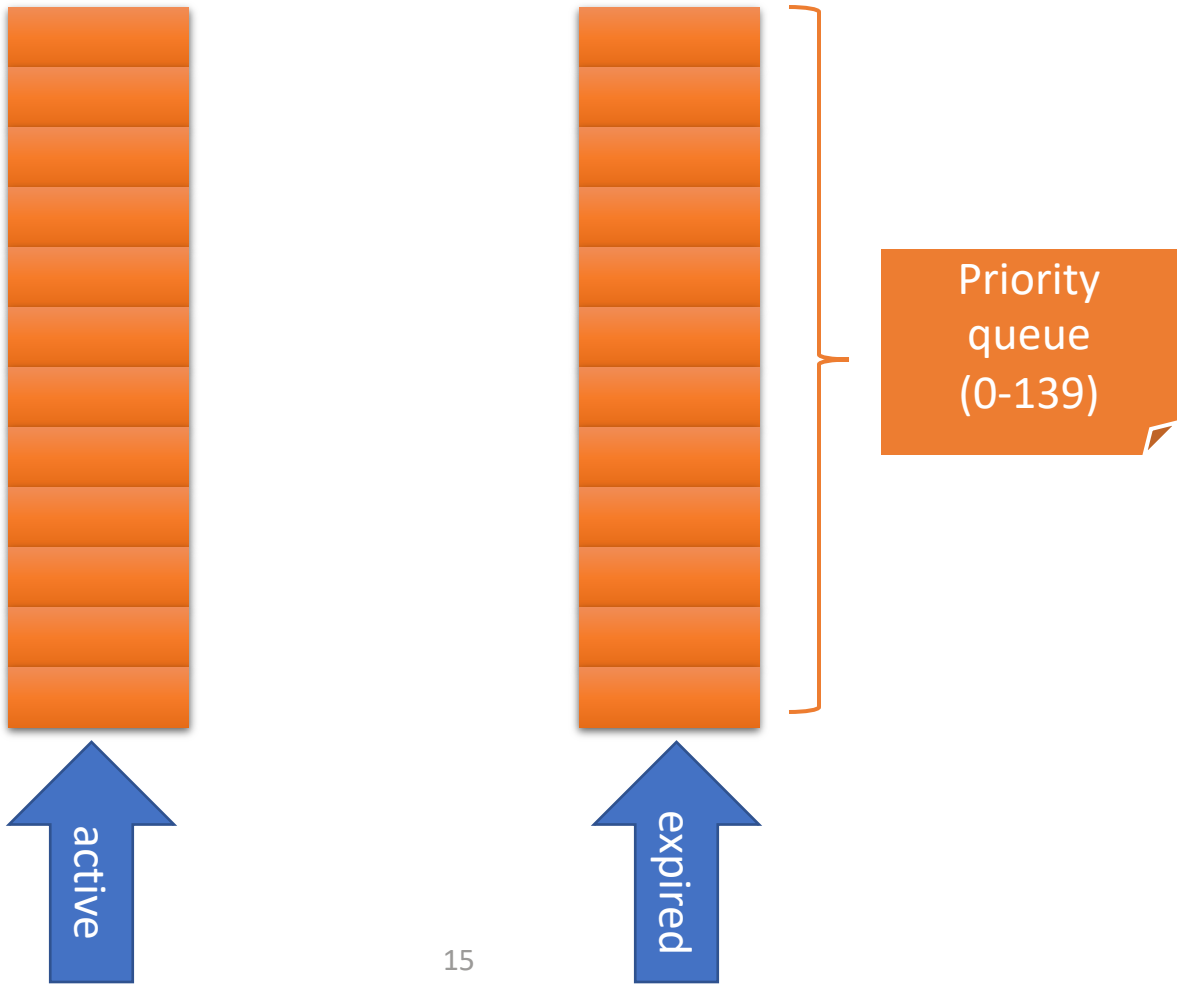
- There is a bit map indicating which queues have processes that are ready to run
- Find the first bit that's set:
  - 140 queues → 5 integers
  - Only a few compares to find the first that is non-zero
  - Hardware instruction to find the first 1-bit
    - bsfl on Intel
  - Time depends on the number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

Scheduling Components

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

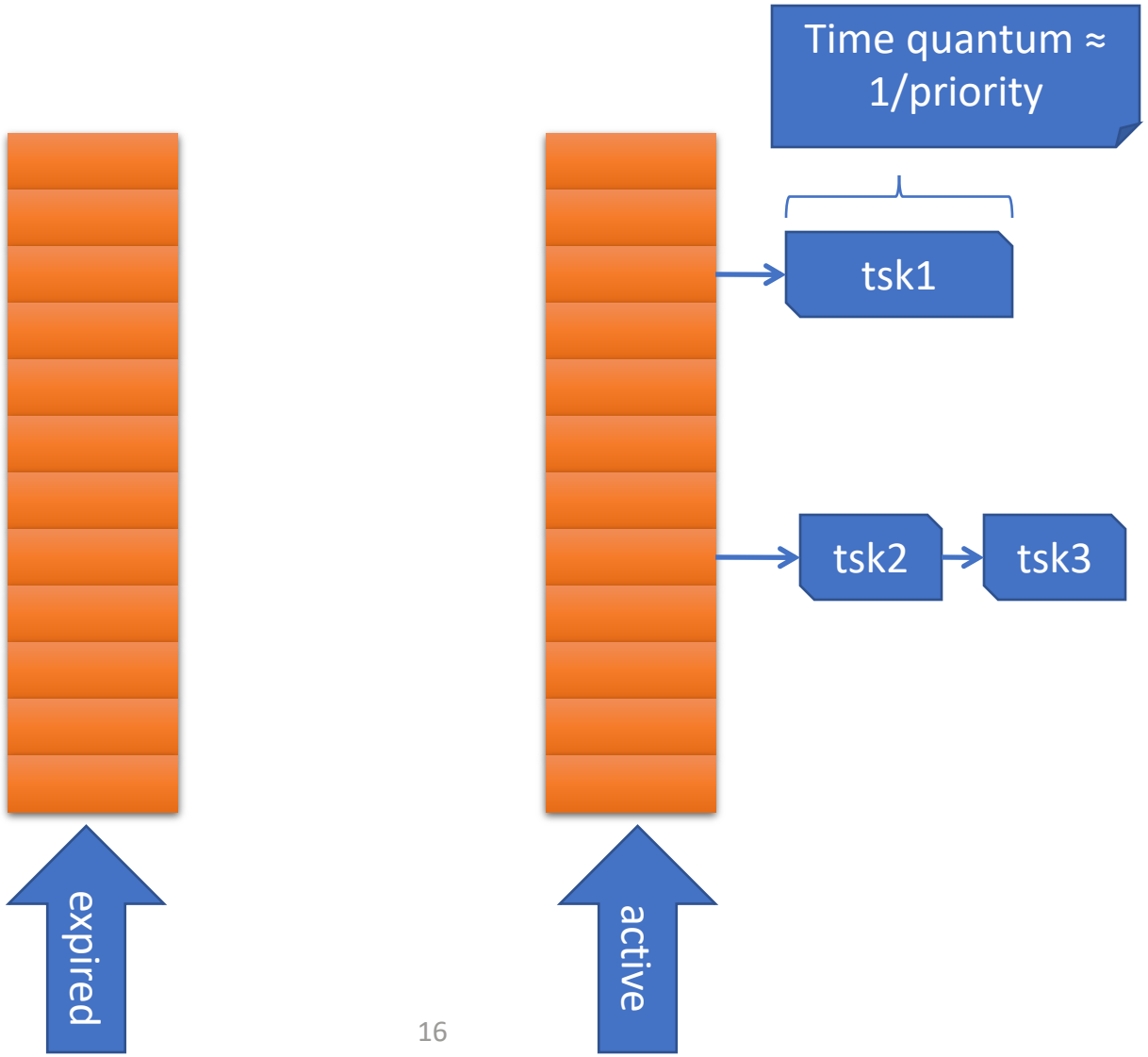
runqueue

- Each runqueue contains two priority arrays – active and expired.
- Each of these priority arrays contains a list of tasks indexed according to priority



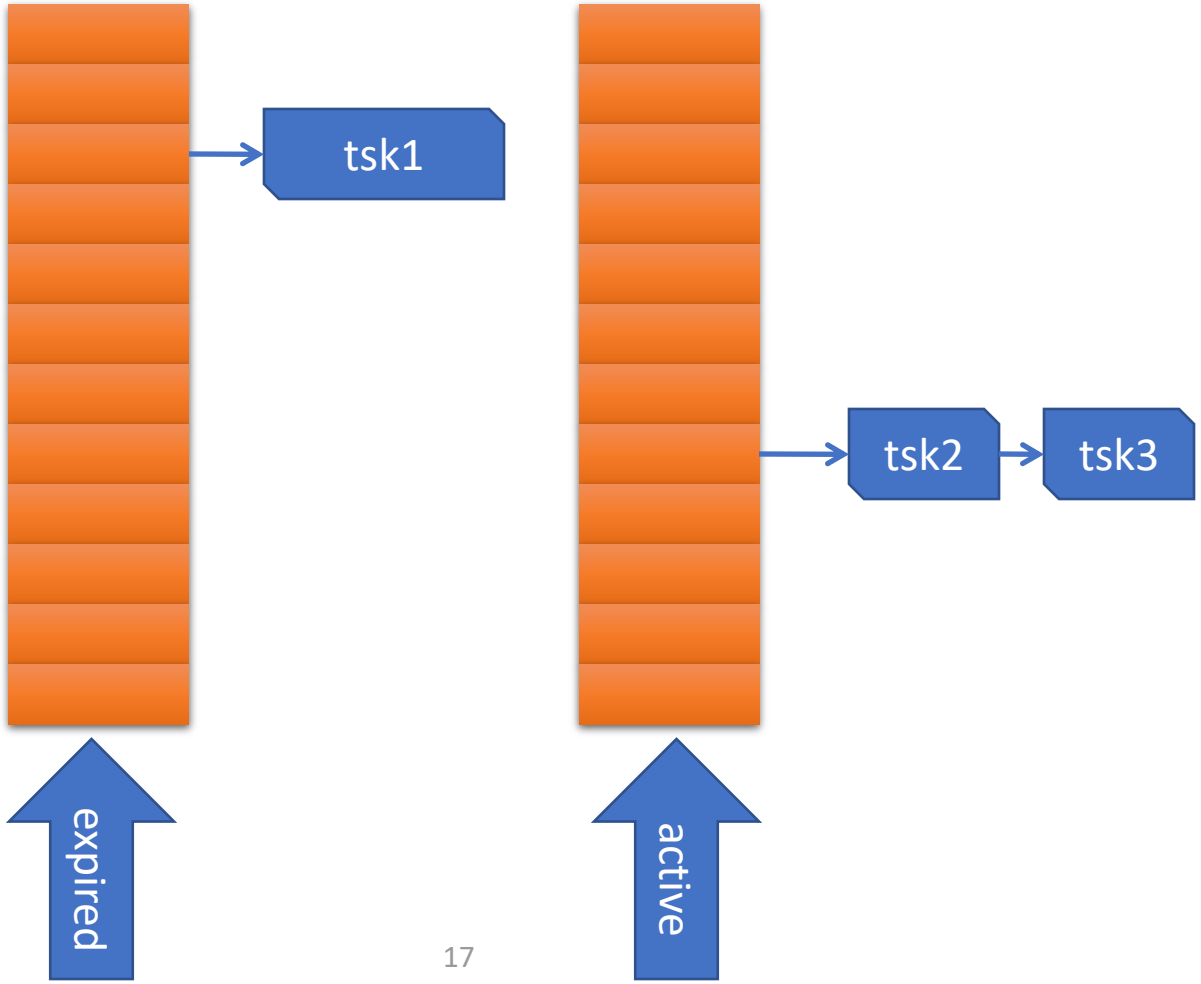
runqueue

- Linux assigns higher-priority tasks longer time-slice

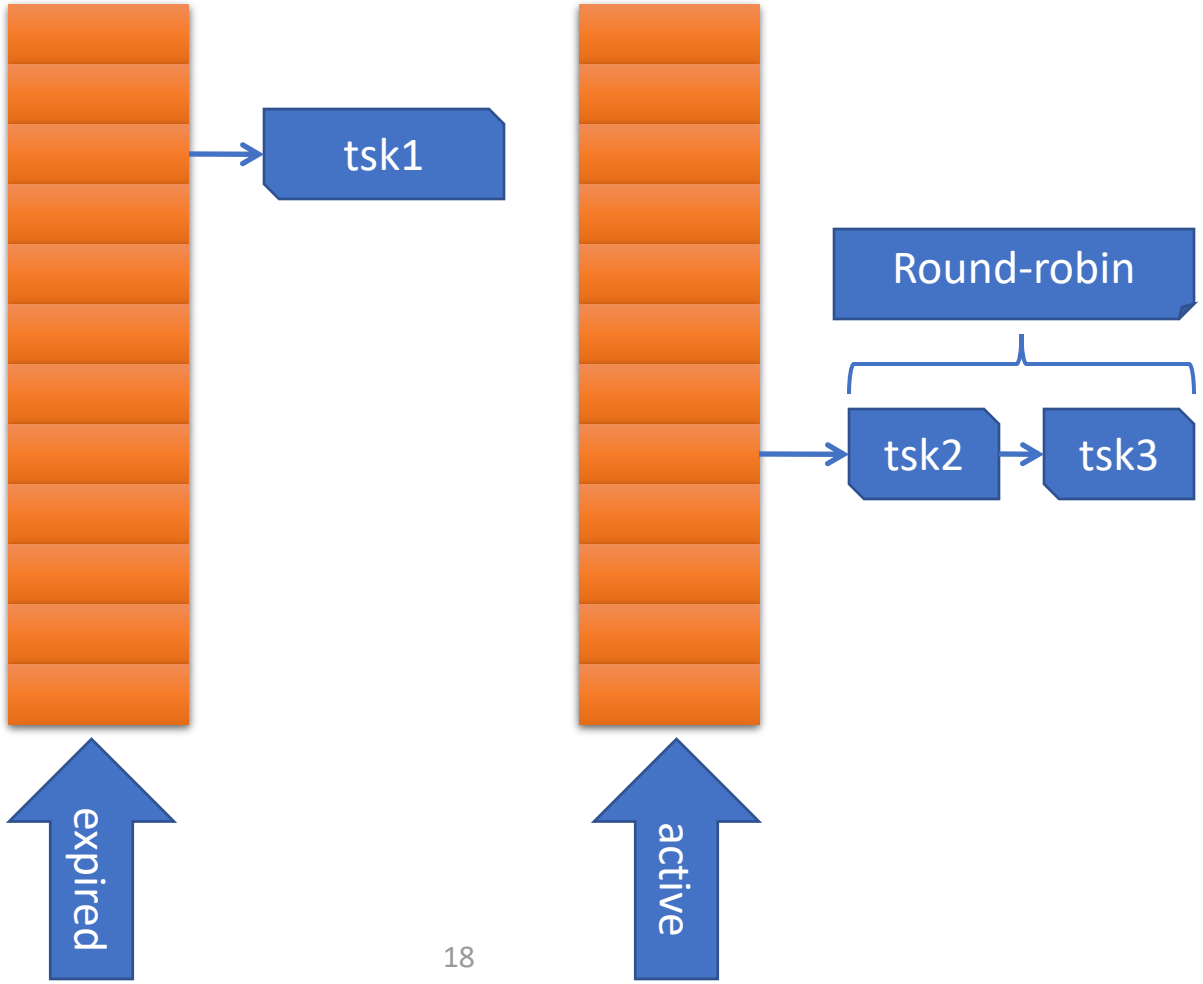


runqueue

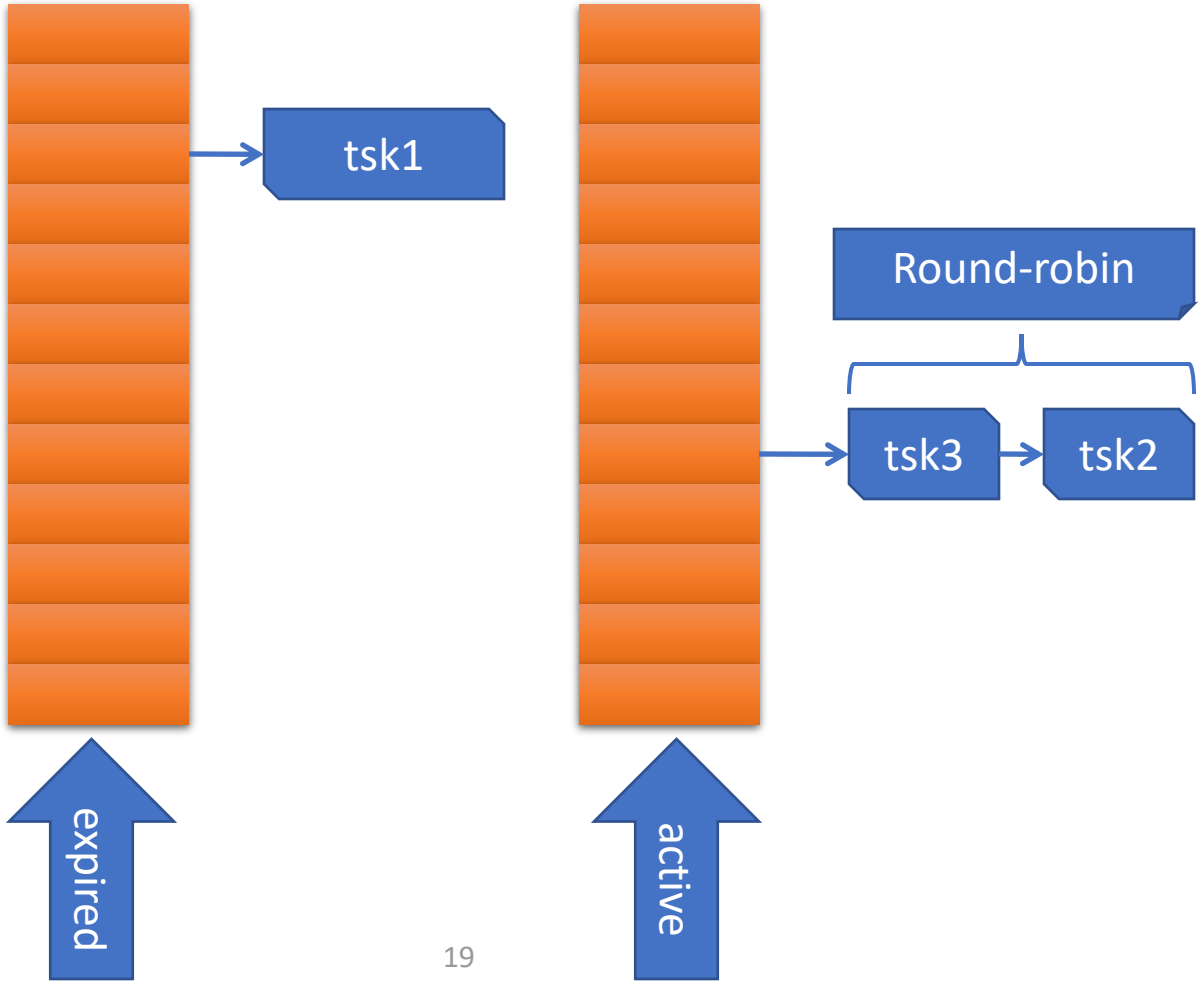
- Linux chooses the task with the highest priority from the active array for execution.



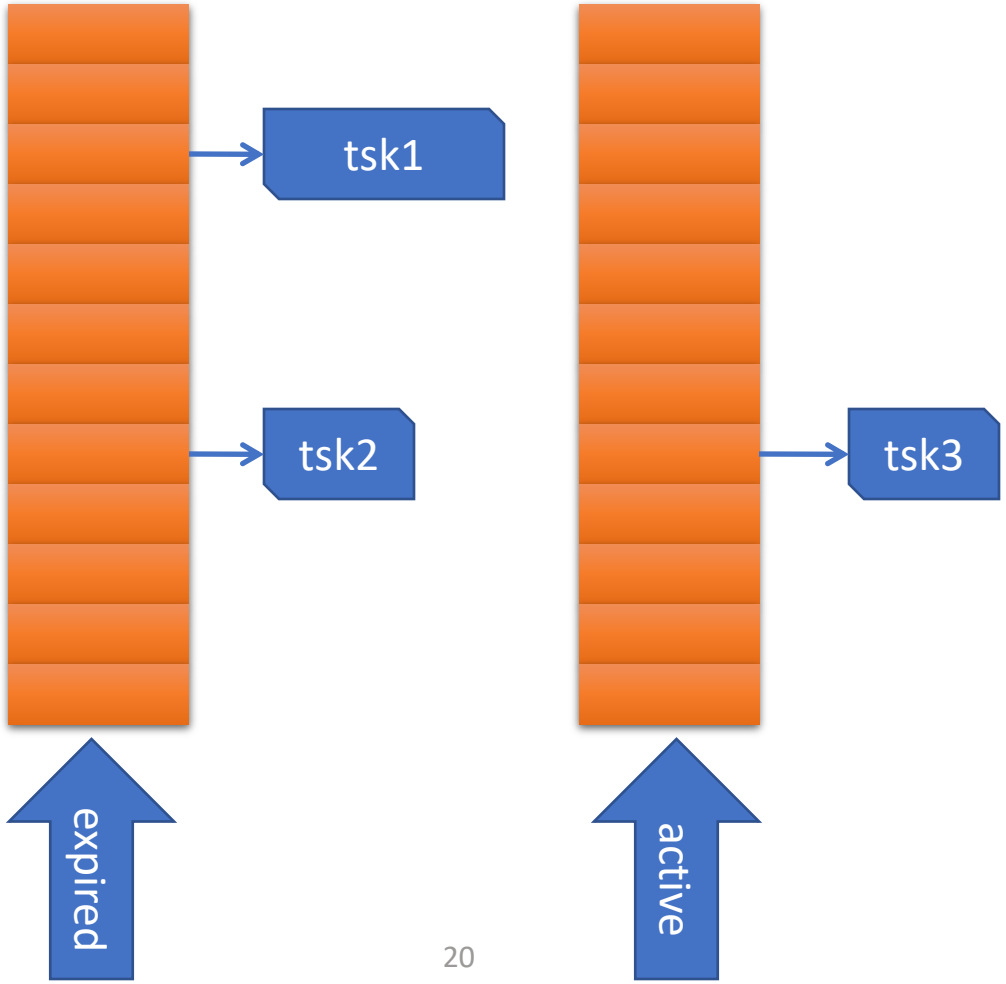
runqueue



runqueue

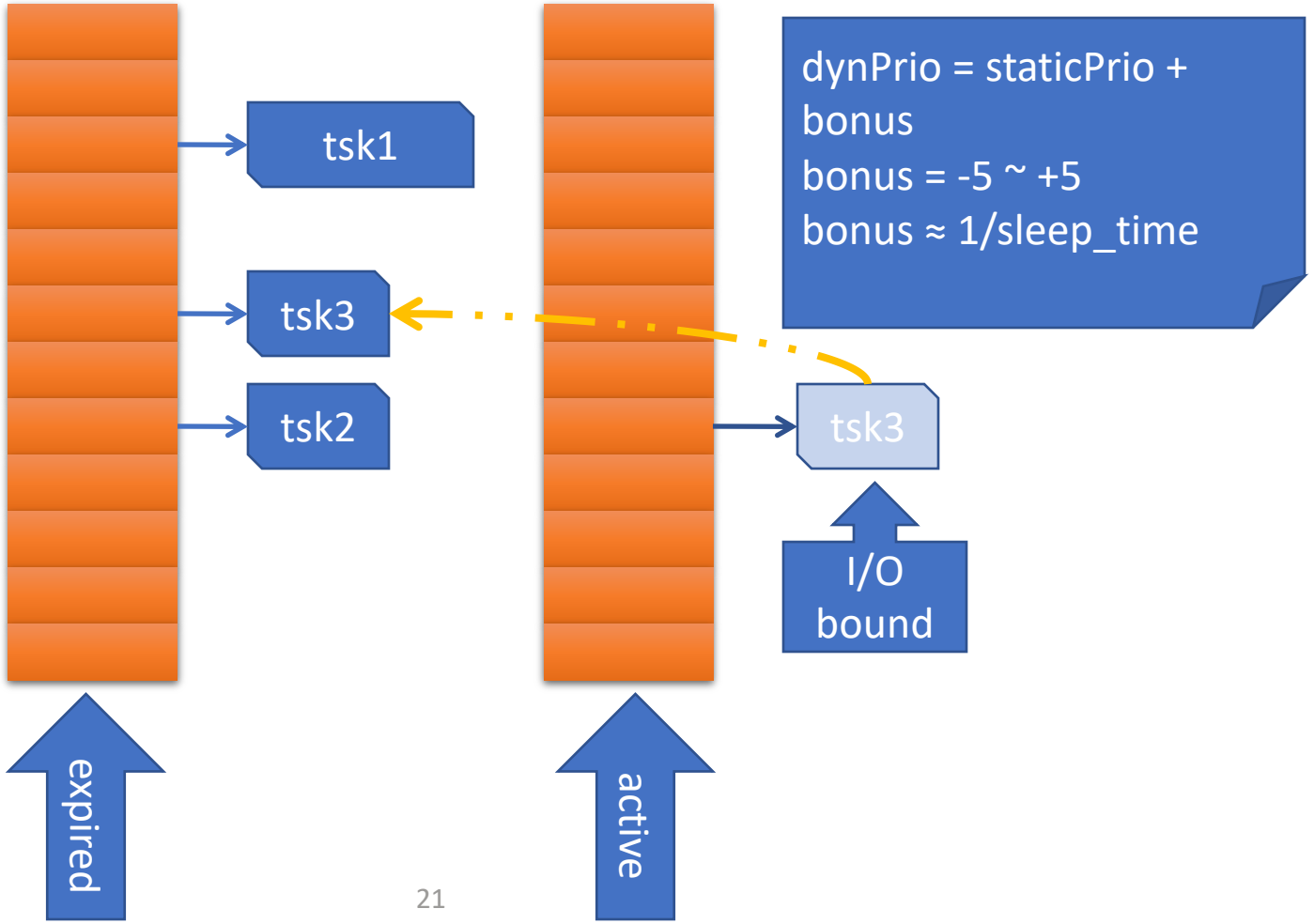


runqueue



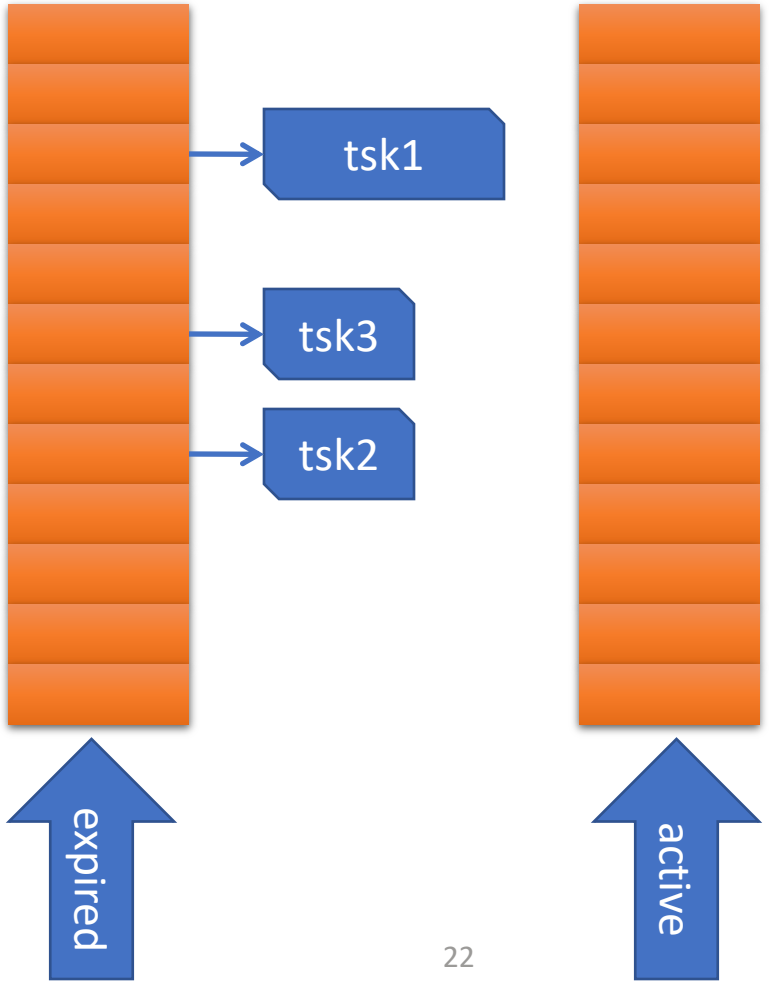
runqueue

- Most tasks have dynamic priorities that are based on their “nice” value (static priority) plus or minus 5
- Interactivity of a task ≈ 1/sleep_time

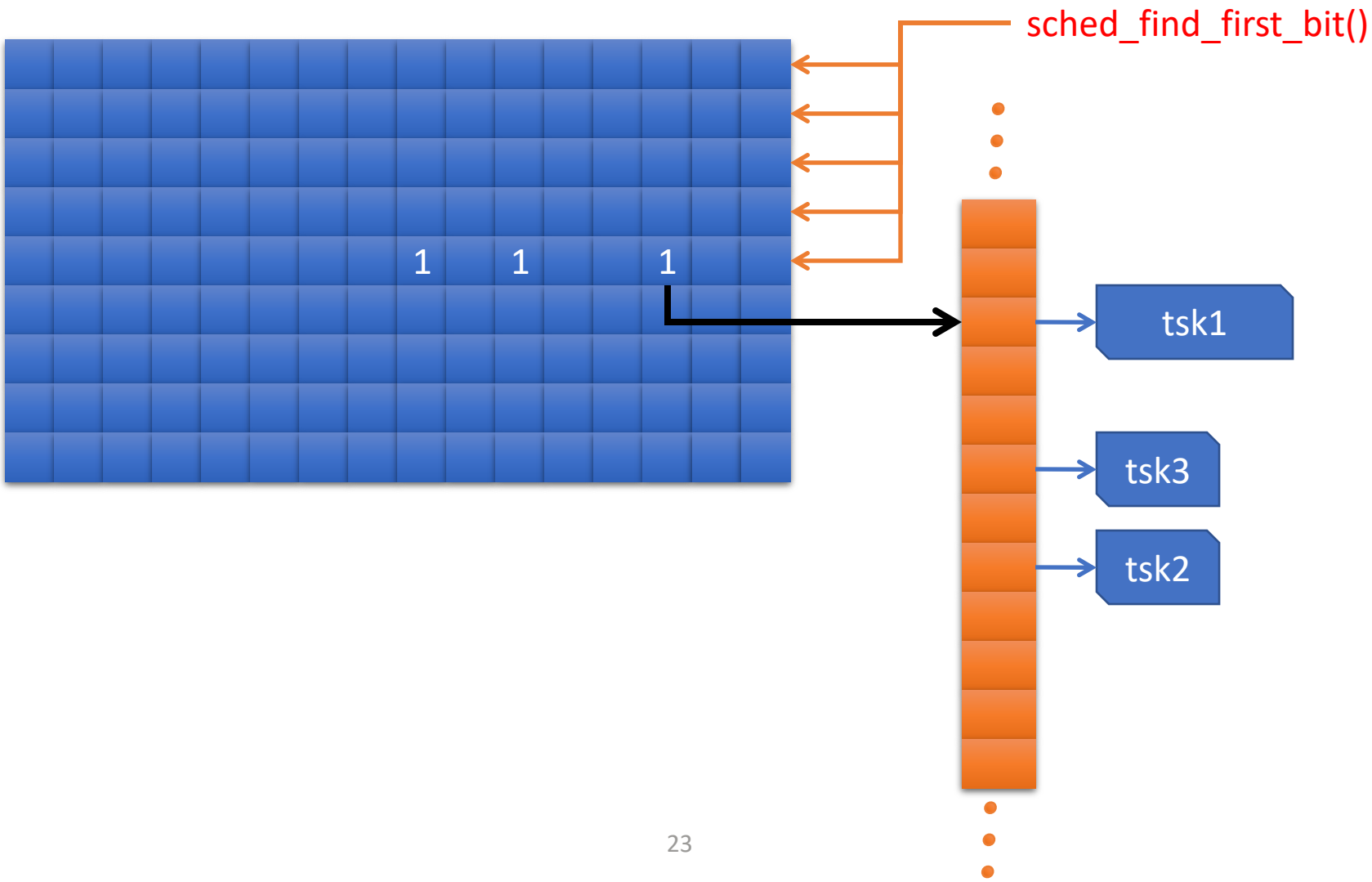


runqueue

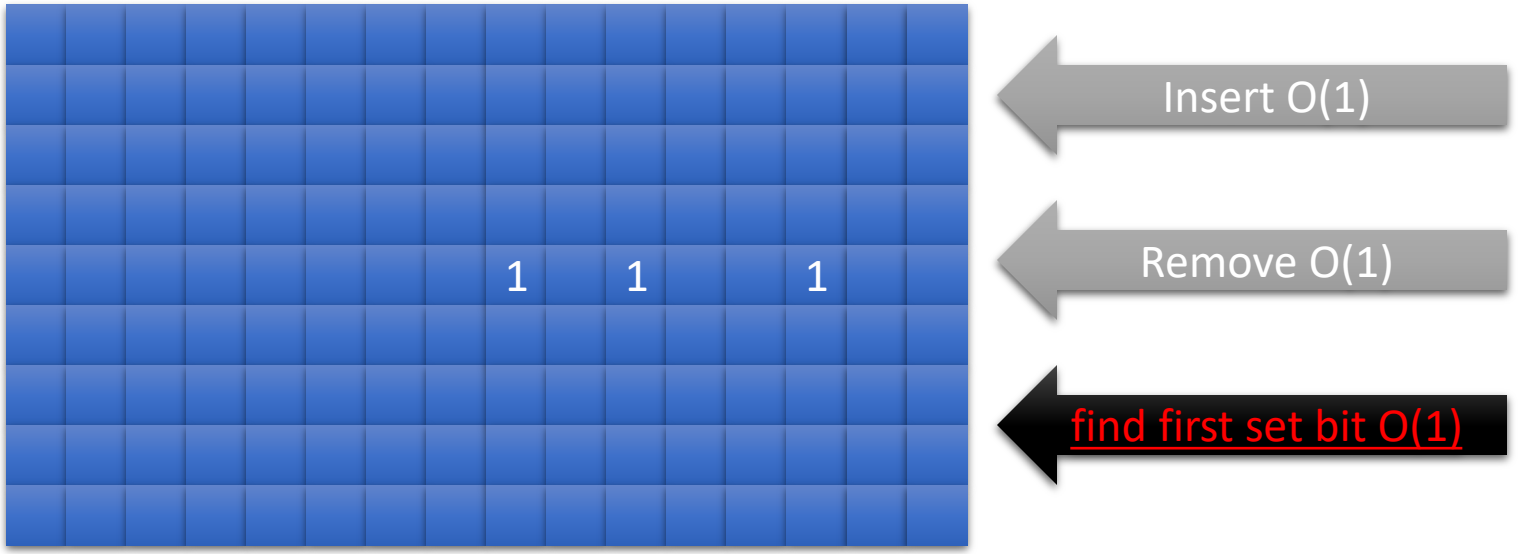
- When all tasks have exhausted their time slices, the two priority arrays are exchanged!



The O(1) scheduling algorithm



The O(1) scheduling algorithm



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*
- The nice value is in a range of 0 to 39, with the default value being 20. Only privileged tasks can set the nice value below 20.
- For normal tasks, the static priority is 100 + the nice value.
- Each task has a dynamic priority that is set based upon a number of factors

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* (10 ms)
- 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)

Bonus and Dynamic Priority

```
/* We scale the actual sleep average
 * [0 .... MAX_SLEEP_AVG] into the
 * -5 ... 0 ... +5 bonus/penalty range.
```

- Dynamic priority (*prio* in *task_struct*) is calculated in *effective_prio* from static priority and bonus (which in turn is derived from *sleep_avg*)
- Roughly speaking, the bonus is a number in [-5, 5] that measures what percentage of the time the process was sleeping recently; 0 is neutral, 5 helps, -5 hurts:

$$DP = SP - bonus + 5$$

$$DP = \min(139, \max(100, DP))$$

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
- As we will see, other mechanisms are used for quick interactive response

Typical Quanta

Priority:	Static Pri	Niceness	Quantum
Highest	100	-20	800 ms
High	110	-10	600 ms
Normal	120	0	100 ms
Low	130	10	50 ms
Lowest	139	20	5 ms

Interactive Processes

- A process is considered interactive if

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

- 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

The Traditional Algorithm

```
for(pp = proc; pp < proc+NPROC; pp++) {
        if (pp->prio != MAX)
                pp->prio++;
        if (pp->prio > curproc->prio)
                reschedule();
}
```

Every process is examined, quite frequently (This code is taken almost verbatim from 6th Edition Unix, circa 1976.)

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*)

Optimizations

- If next is a kernel thread, borrow the MM mappings from prev
  - User-level MMs are unused.
  - Kernel-level MMs are the same for all kernel threads
- If prev == next
  - Don't context switch

Sleep Time and Bonus

Average Sleep Time (ms)	Bonus	Time Slice Granularity
000 to 100	0	5120
100 to 200	1	2560
200 to 300	2	1280
300 to 400	3	640
400 to 500	4	320
500 to 600	5	160
600 to 700	6	80
700 to 800	7	40
800 to 900	8	20
900 to 999	9	10
1 second	10	10

CFS Overview

- Since 2.6.23
- Maintain balance (fairness) in providing CPU time to tasks.
- When the time for tasks is out of balance, then those out-of-balance tasks should be given time to execute.
- To determine the balance, the amount of time provided to a given task is maintained in the virtual runtime (amount of time a task has been permitted access to the CPU).

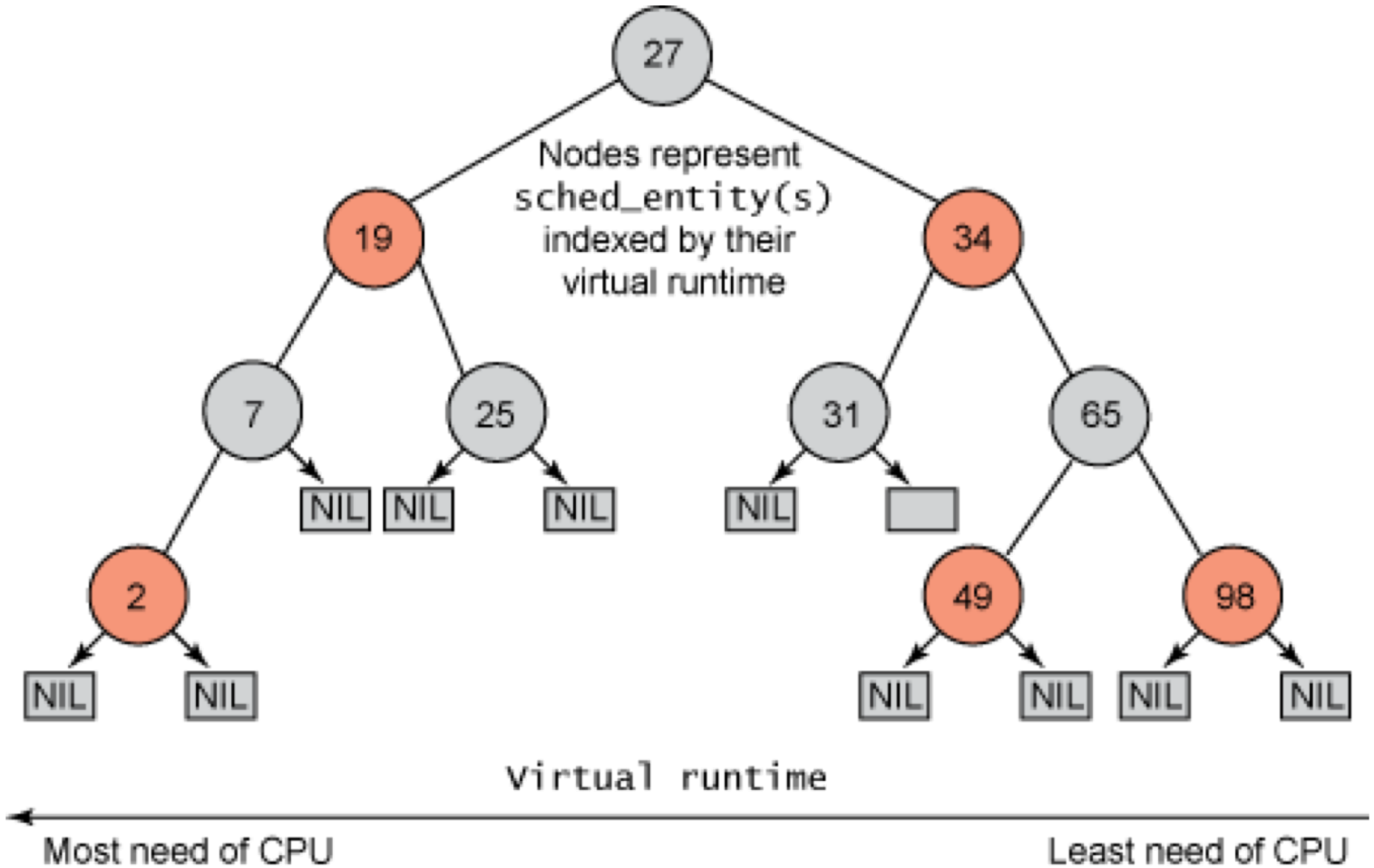
CFS Overview

- The smaller a task's virtual runtime, the higher its need for the processor.
- The CFS also includes the concept of sleeper fairness to ensure that tasks that are not currently runnable receive a comparable share of the processor when they eventually need it.

CFS Structure

- Tasks are maintained in a time-ordered red-black tree for *each* CPU, instead of a run queue.
  - self-balancing
  - operations on the tree occur in O(log n) time.
- Tasks with the gravest need for the processor (lowest virtual runtime) are stored toward the left side of the tree, and tasks with the least need of the processor (highest virtual runtimes) are stored toward the right side of the tree.
- The scheduler picks the left-most node of the red-black tree to schedule next to maintain fairness.

CFS Overview

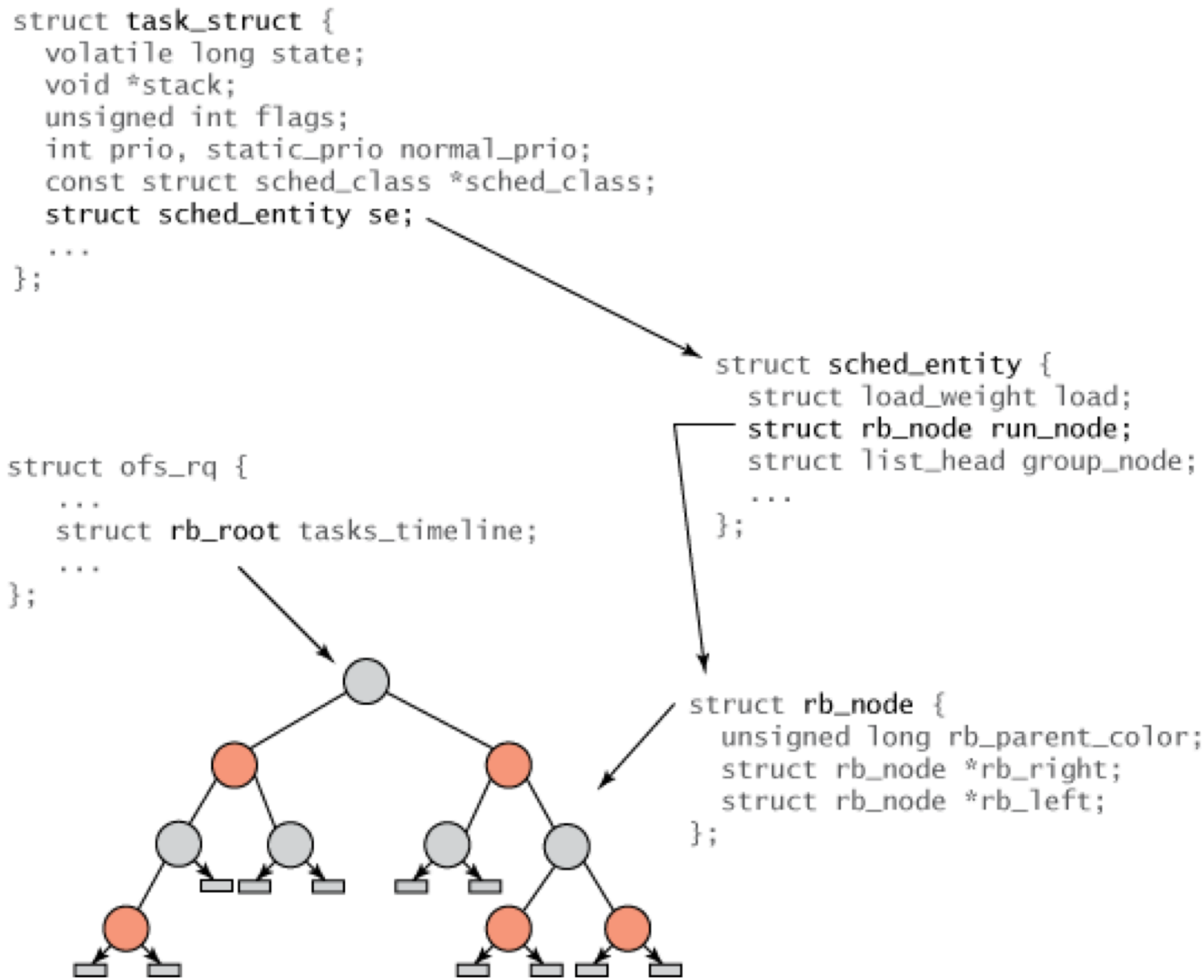


CFS Structure

- The task accounts for its time with the CPU by adding its execution time to the virtual runtime and is then inserted back into the tree if runnable.
- The contents of the tree migrate from the right to the left to maintain fairness.

CFS Internals

- All tasks are represented by a structure called `task_struct`.
- This structure fully describes the task: current state, stack, process flags, priority (static and dynamic)...
- ./linux/include/linux/sched.h.
- Not all tasks are runnable → No CFS-related fields in `task_struct`.
  - `sched_entity`
- Each node in the tree is represented by an rb_node, which contains the child references and the color of the parent.


struct task_struct {
volatile long state;
void *stack;
unsigned int flags;
int prio, static_prio normal_prio;
const struct sched_class *sched_class;
struct sched_entity se;
...
};
struct sched_entity {
struct load_weight load;
struct rb_node run_node;
struct list_head group_node;
...
};
struct ofs_rq {
...
struct rb_root tasks_timeline;
...
};
struct rb_node {
unsigned long rb_parent_color;
struct rb_node *rb_right;
struct rb_node *rb_left;
};

Scheduling

1. **`schedule()`** (`./kernel/sched.c`) preempts the currently running task – unless it preempts itself with **`yield()`**.
   - CFS has no real notion of time slices for preemption, because the preemption time is variable.
2. The currently running task (now preempted) is returned to the red-black tree through a call to **`put_prev_task`** (via the scheduling class).

Scheduling

3. When the schedule function comes to identifying the next task to schedule, it calls **`pick_next_task()`**, which calls the CFS scheduler through **`pick_next_task_fair()`** (`./kernel/sched_fair.c`).
4. It picks the left-most task from the red-black tree and returns the associated **`sched_entity`**.
   - With this reference, a simple call to **`task_of()`** identifies the **`task_struct`** reference returned.
5. The generic scheduler finally provides the processor to this task.

Priorities in CFS

- CFS doesn't use priorities directly.
- Decay factor for the time a task is permitted to execute.
  - Lower-priority tasks have higher factors of decay.
  - The time a task is permitted to execute dissipates more quickly for a lower-priority task than for a higher-priority task
  - Avoid maintaining run queues per priority.

CFS Group Scheduling

- Since 2.6.24
- Bring fairness to scheduling in the face of tasks that spawn many other tasks (e.g. HTTP server).
- Instead of all tasks being treated fairly, the spawned tasks with their parent share their virtual runtimes across the group (in a hierarchy).
  - Other single tasks maintain their own independent virtual runtimes.
  - Single tasks receive roughly the same scheduling time as the group.
- There's a `/proc` interface to manage the process hierarchies, giving full control over how groups are formed.
  - Fairness can be assigned across users, processes, or a variation of each.

CFS Scheduling Classes

- Each task belongs to a scheduling class, which determines how a task will be scheduled.
- A scheduling class defines a common set of functions (via **`sched_class`**) that define the behavior of the scheduler.
- For example, each scheduler provides a way to add a task to be scheduled, pull the next task to be run, yield to the scheduler, and so on.

CFS Scheduling Domains

- Scheduling domains allow you to group one or more processors hierarchically for purposes load balancing and segregation.
- One or more processors can share scheduling policies (and load balance between them) or implement independent scheduling policies to intentionally segregate tasks.

References

- Silberschatz, .A et. al. (2009), *Operating System Concepts*, 8th Edition.
  - Section 5.6.3 Example: Linux Scheduling.
- Jones M. T. (15 Dec 2009), Inside the Linux 2.6 Completely Fair Scheduler, *IBM developerWorks*, http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/
- Kumar, A. (08 Jan 2008), Multiprocessing with the Completely Fair Scheduler, *IBM developerWorks*, http://www.ibm.com/developerworks/linux/library/l-cfs/
- CFS Scheduler (./linux/Documentation/scheduler/sched-design-CFS.txt). Can also be retrieved from http://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt