Process Scheduling II

COMS W4118
Prof. Kaustubh R. Joshi
krj@cs.columbia.edu

http://www.cs.columbia.edu/~krj/os

References: Operating Systems Concepts (9e), Linux Kernel Development, previous W4118s
Copyright notice: care has been taken to use only those web images deemed by the instructor to be in the public domain. If you see a copyrighted image on any slide and are the copyright owner, please contact the instructor. It will be removed.

Outline

- Advanced scheduling issues
  - – Multilevel queue scheduling
  - – Multiprocessor scheduling issues

- Linux/Android Scheduling
  - – Scheduler Architecture
  - – Scheduling algorithm
    - O(1) RR scheduler
    - CFS scheduler
  - – Other implementation issues

Motivation

- No one-size-fits-all scheduler
  - Different workloads
  - Different environment

- Building a general scheduler that works well for all is difficult!

- Real scheduling algorithms are often more complex than the simple scheduling algorithms we've seen

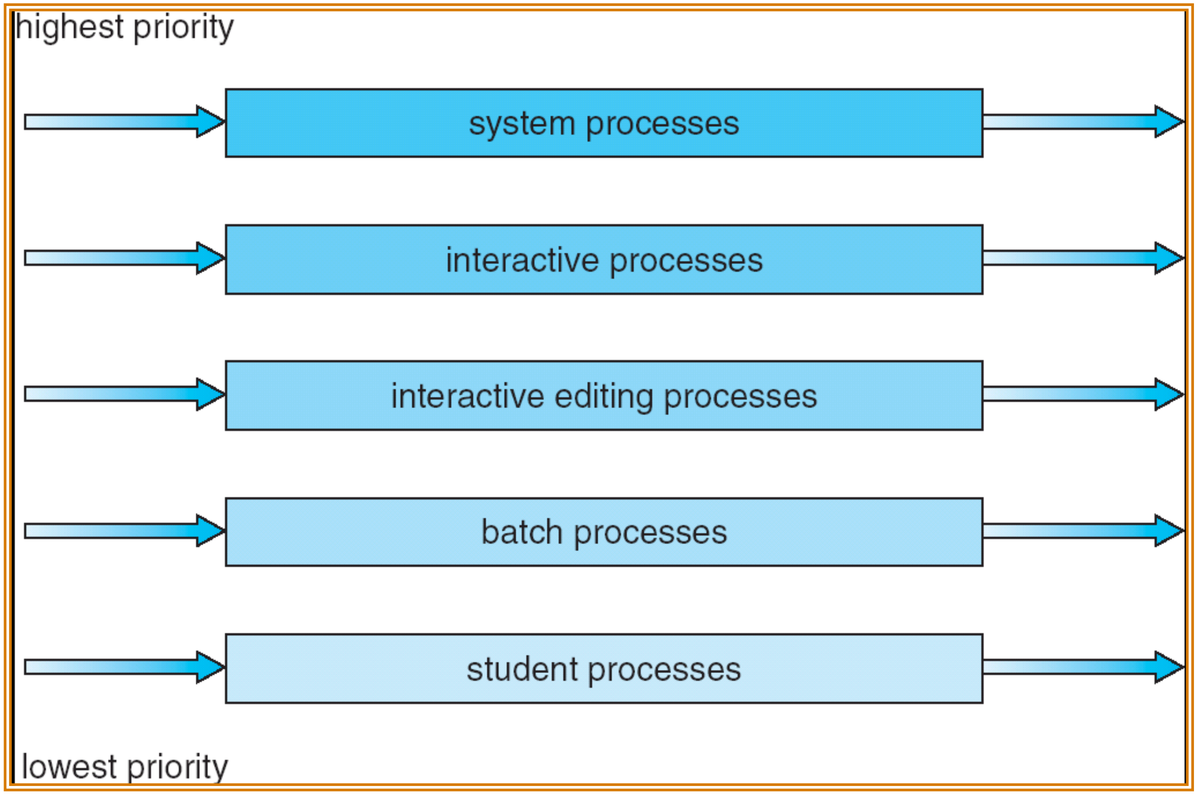
Combining scheduling algorithms

- Multilevel queue scheduling: ready queue is partitioned into multiple queues

- Each queue has its own scheduling algorithm
  - Foreground processes: RR (e.g., shell, editor, GUI)
  - Background processes: FCFS (e.g., backup, indexing)

- Must choose scheduling algorithm to schedule between queues. Possible algorithms
  - RR between queues
  - Fixed priority for each queue
  - Timeslice for each queue (e.g., RR gets 80%, FCFS 20%)

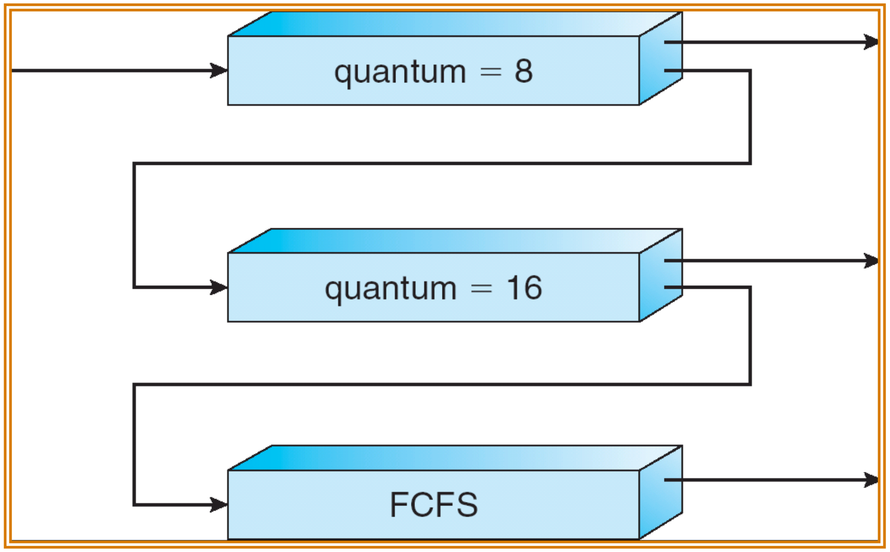
Movement between queues



- No automatic movement between queues
- User can change process queue at any time

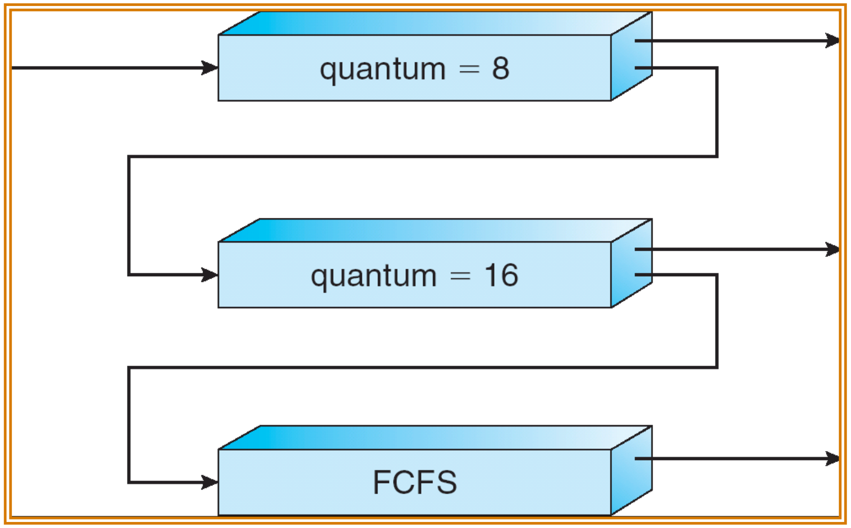
Multilevel Feedback Queue

- Process automatically moved between queues
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process



- Used to implement
  - Aging: move to higher priority queue
  - Monopolizing resources: move to lower priority queue

Aging using Multilevel Queues



- A new job enters queue Q_0 which is served RR. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
- At Q_1 job is again served RR and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to low priority FCFC queue Q_2.

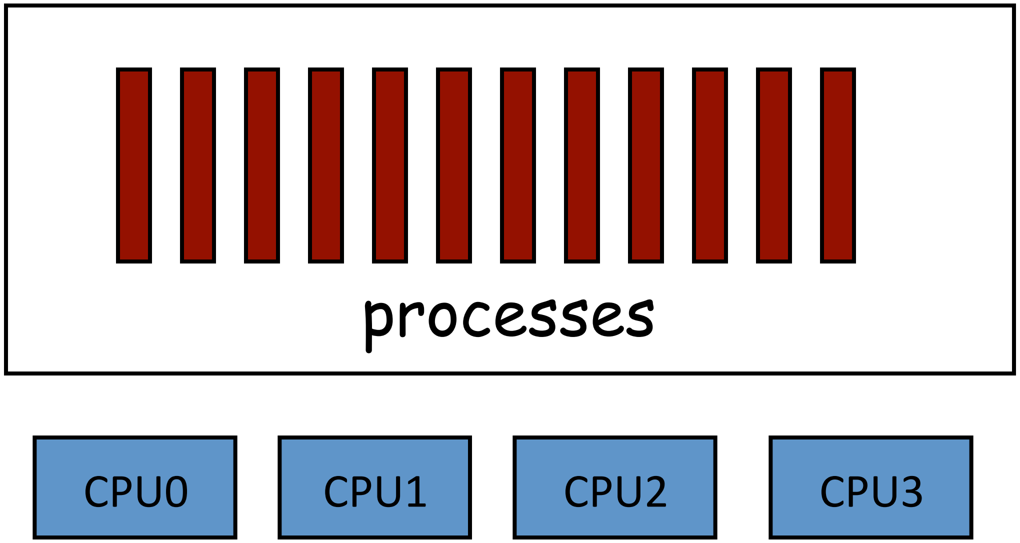
Outline

- Advanced scheduling issues
  - Multilevel queue scheduling
  - Multiprocessor scheduling issues

- Linux/Android Scheduling
  - Scheduling algorithm
    - O(1) RR scheduler
    - CFS scheduler
  - Setting priorities and time slices
  - Other implementation issues

Multiprocessor scheduling issues

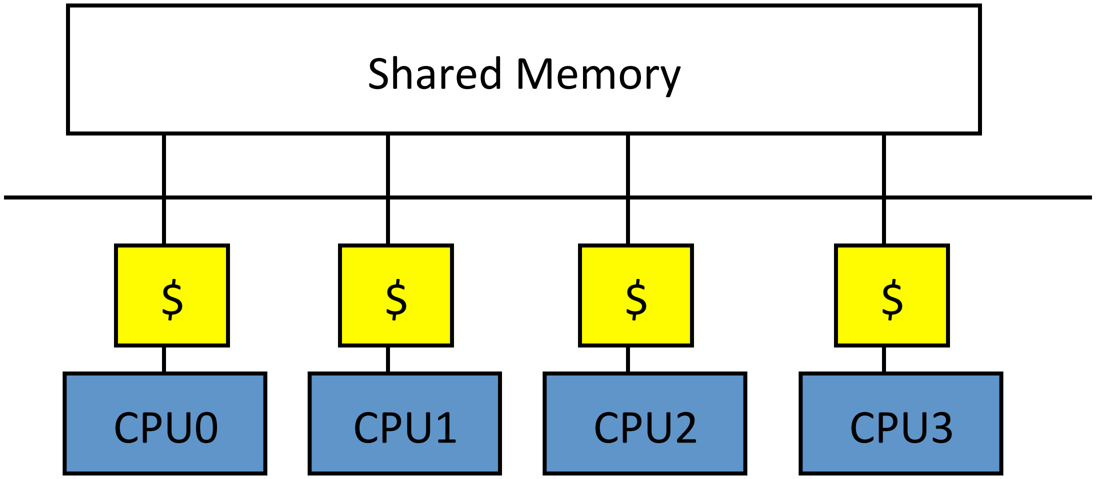
- Shared-memory Multiprocessor



- How to allocate processes to CPU?

Symmetric multiprocessor

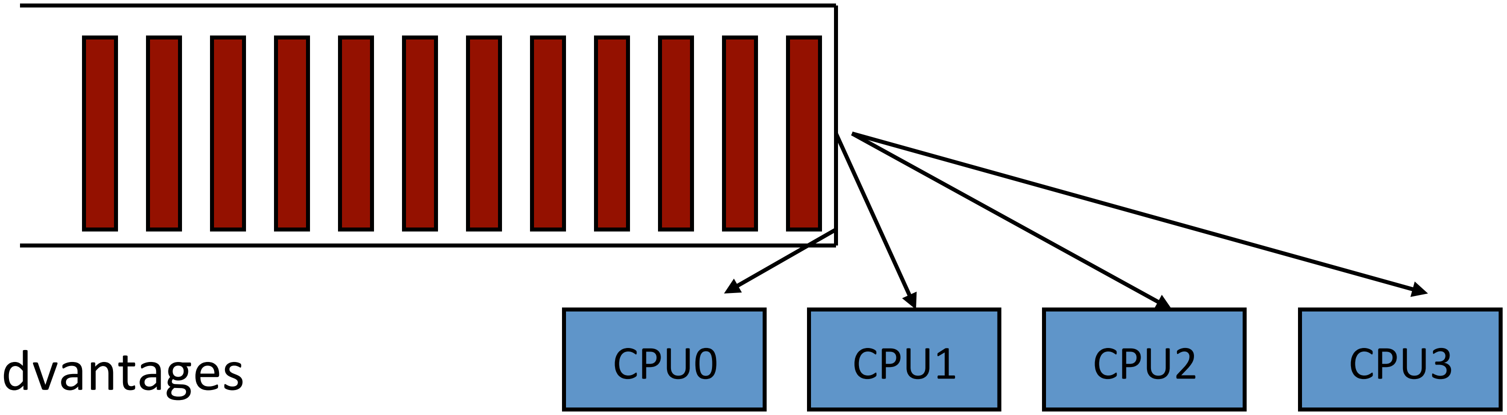
- Architecture



- Small number of CPUs
- Same access time to main memory
- Private cache
  - Memory
  - Memory mappings (TLB)

Global queue of processes

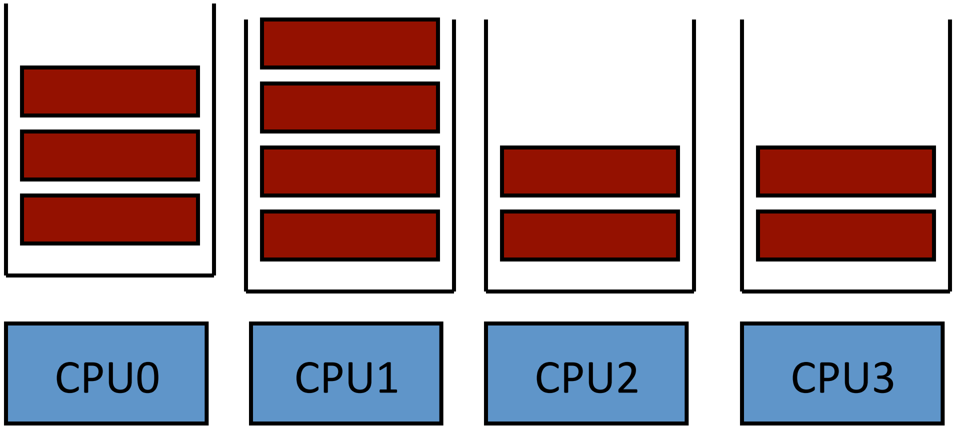
- One ready queue shared across all CPUs



- Advantages
  - Good CPU utilization
  - Fair to all processes
- Disadvantages
  - Not scalable (contention for global queue lock)
  - Poor cache locality
- Linux 2.4 uses global queue

Per-CPU queue of processes

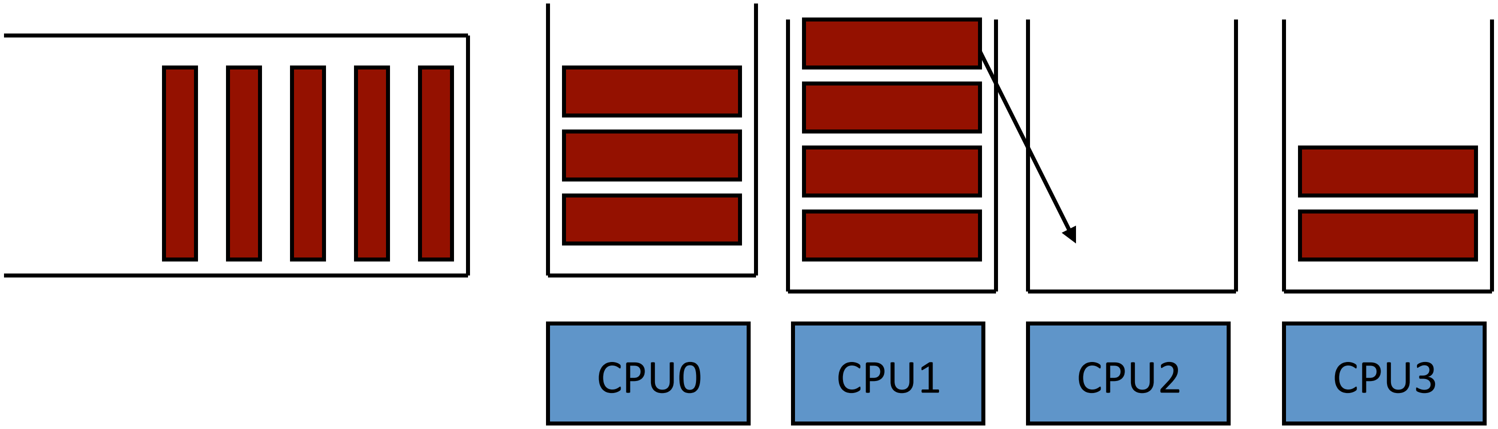
- Static partition of processes to CPUs



- Advantages
  - Easy to implement
  - Scalable (no contention on ready queue)
  - Better cache locality
- Disadvantages
  - Load-imbalance (some CPUs have more processes)
    - Unfair to processes and lower CPU utilization

Hybrid approach

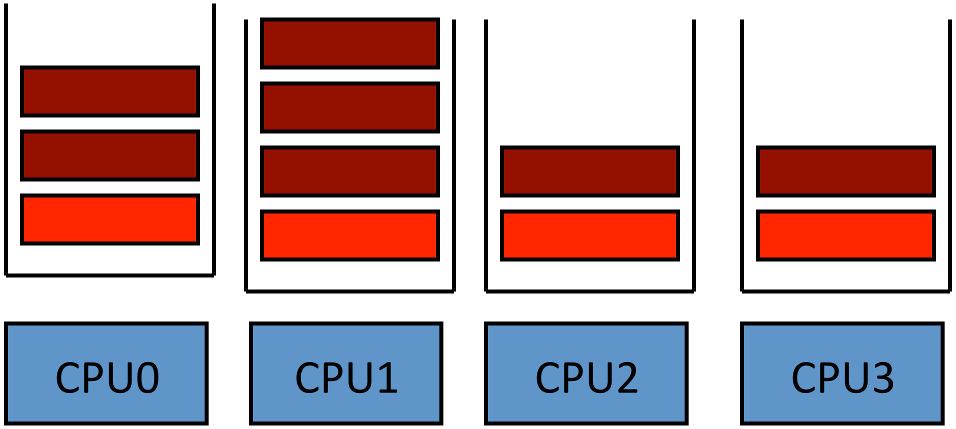
- Use both global and per-CPU queues
- Balance jobs across queues



- Processor Affinity
  - Add process to a CPU's queue if recently run on the CPU
    - Cache state may still present
- Linux 2.6 uses a very similar approach

SMP: "gang" scheduling

- Multiple processes need coordination
- Should be scheduled simultaneously



- Scheduler on each CPU does not act independently
- Coscheduling (gang scheduling): run a set of processes simultaneously
- Global context-switch across all CPUs

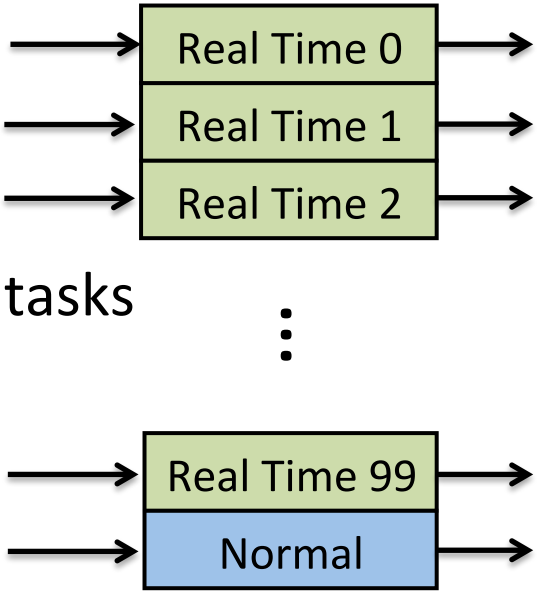
Outline

- Advanced scheduling issues
  - Multilevel queue scheduling
  - Multiprocessor scheduling issues

- Linux/Android Scheduling
  - Scheduler Architecture
  - Scheduling algorithms
    - O(1) RR scheduler
    - CFS scheduler
  - Other implementation issues

Linux Scheduler Class Overview

- Linux has a hierarchical scheduler
  - Soft Real-time scheduling policies
    - SCHED_FIFO (FCFS)
    - SCHED_RR (real time round robin)
    - Always get priority over non real time tasks
    - One of 100 priority levels (0..99)
  - Normal scheduling policies
    - SCHED_OTHER: standard processes
    - SCHED_BATCH: batch style processes
    - SCHED_IDLE: low priority tasks
    - One of 40 priority levels (-20..0..19)



Real Time 0

Real Time 1

Real Time 2

⋮

Real Time 99

Normal

Linux Hierarchical Scheduler

Code from kernel/sched.c:

```
class = sched_class_highest;
        for ( ; ; ) {
                p = class->pick_next_task(rq);
                if (p)
                        return p;
                /*
                 * Will never be NULL as the idle class always
                 * returns a non-NULL p:
                 */
                class = class->next;
        }
```

The runqueue

- All run queues available in array runqueues, one per CPU
- struct rq (kernel/sched.c)
  - Contains per-class run queues (RT, CFS) and other per-class params
    - E.g., CFS: a list of task_struct in struct list_head tasks
    - E.g., RT: array of active priorities
    - Data structure rt_rq, cfs_rq,
- struct sched_entity (kernel/sched.c)
  - Member of task_struct, one per scheduler class
  - Maintains list head for class runqueue, other per-task params
- Current scheduler for task is specified by task_struct.sched_class
  - Pointer to struct sched_class
  - Contains functions pertaining to class (object-oriented code)

sched_class Structure

```
static const struct sched_class fair_sched_class = {
        .next                   = &idle_sched_class,
        .enqueue_task           = enqueue_task_fair,
        .dequeue_task           = dequeue_task_fair,
        .yield_task             = yield_task_fair,
        .check_preempt_curr     = check_preempt_wakeup,
        .pick_next_task         = pick_next_task_fair,
        .put_prev_task          = put_prev_task_fair,
        .select_task_rq         = select_task_rq_fair,
        .load_balance           = load_balance_fair,
        .move_one_task          = move_one_task_fair,
        .set_curr_task          = set_curr_task_fair,
        .task_tick              = task_tick_fair,
        .task_new               = task_new_fair,
        .prio_changed           = prio_changed_fair,
        .switched_to            = switched_to_fair,
}
```

Multiprocessor scheduling

- Per-CPU runqueue
- Possible for one processor to be idle while others have jobs waiting in their run queues
- Periodically, rebalance runqueues
  - Migration threads move processes from one runque to another
- The kernel always locks runqueues in the same order for deadlock prevention

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

Processor affinity

- Each process has a bitmask saying what CPUs it can run on
  - By default, all CPUs
  - Processes can change the mask
  - Inherited by child processes (and threads), thus tending to keep them on the same CPU
- Rebalancing does not override affinity

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
  - currently running on a CPU
  - not allowed to run on the current CPU (as indicated by the *cpus_allowed* bitmask in the *task_struct*)

Priority related fields in *struct task_struct*

- static_prio: static priority set by administrator/ users
  - Default: 120 (even for realtime processes)
  - Set use sys_nice() or sys_setpriority()
    - Both call set_user_nice()
- prio: dynamic priority
  - Index to prio_array
- rt_priority: real time priority
  - prio = 99 – rt_priority
- include/linux/sched.h

Adding a new Scheduler Class

- The Scheduler is modular and extensible
  - New scheduler classes can be installed
  - Each scheduler class has priority within hierarchical scheduling hierarchy
  - Priorities defined in sched.h, e.g. #define SCHED_RR 2
  - Linked list of sched_class sched_class.next reflects priority
  - Core functions: kernel/sched.c, include/linux/sched.h
  - Additional classes: kernel/sched_fair.c,sched_rt.c
- Process changes class via sched_setscheduler syscall
- Each class needs
  - New runqueue structure in main struct runqueue
  - New sched_class structure implementing scheduling functions
  - New sched_entity in the task_struct

Outline

- Advanced scheduling issues
  - Multilevel queue scheduling
  - Multiprocessor scheduling issues

- Linux/Android Scheduling
  - Scheduler Architecture
  - Scheduling algorithms
    - O(1) RR scheduler
    - CFS scheduler
  - Other implementation issues

Real-time policies

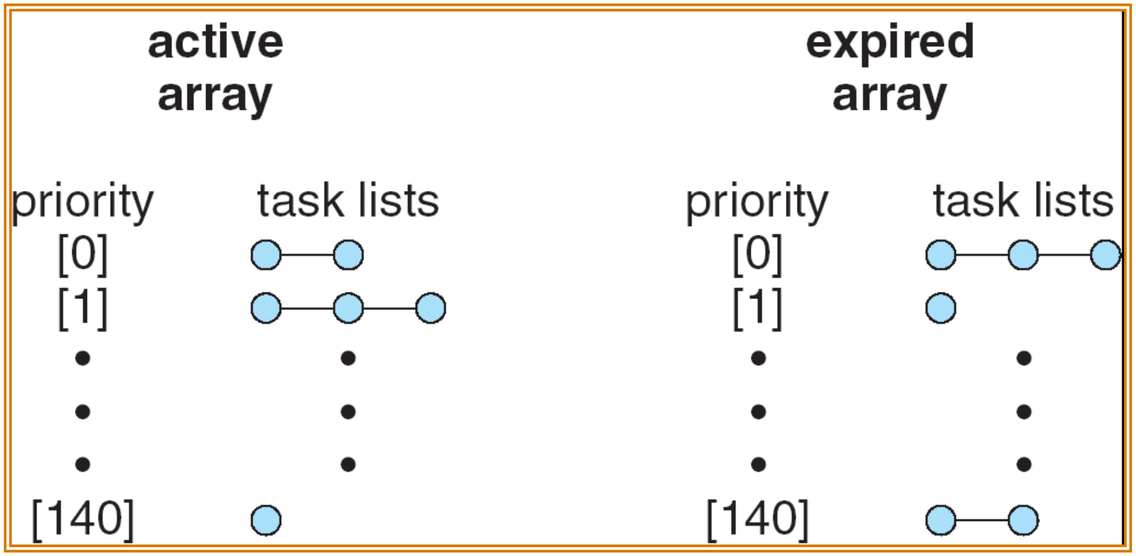
- First-in, first-out: SCHED_FIFO
  - Static priority
  - Process is only preempted for a higher-priority process
  - No time quanta; it runs until it blocks or yields voluntarily
  - RR within same priority level
- Round-robin: SCHED_RR
  - As above but with a time quanta
- Normal processes have SCHED_NORMAL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

Old Linux O(1) scheduler

- Old Linux scheduler (until 2.6.22) for SCHED_NORMAL
  - Round robin fixed time slice
- Boost interactivity
  - Fast response to user despite high load
  - Inferring interactive processes and dynamically increase their priorities
  - Avoid starvation
- Scale well with number of processes
  - O(1) scheduling overhead
- Scale well with number of processors
  - Load balance: no CPU should be idle if there is work
  - CPU affinity: no random bouncing of processes

runqueue data structure

- Two arrays of priority queues
  - active and expired
  - Total 140 priorities [0, 140)
  - Smaller integer = higher priority



Aging: the traditional algorithm

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

Problem: O(N). Every process is examined on each schedule() call!

This code is taken almost verbatim from 6th Edition Unix, circa 1976.

Scheduling algorithm for normal processes

1. Find highest priority non-empty queue in rq->active; if none, simulate aging by swapping active and expired
2. next = first process on that queue
3. Adjust next’s priority
4. Context switch to next
5. When next used up its time slice, insert next to the right queue in the expired array and call schedule() again

Simulate aging

- Swapping active and expired gives low priority processes a chance to run
- Advantage: O(1)
  - Processes are touched only when they start or stop running

Find highest priority non-empty queue

- Time complexity: O(1)
  - Depends on the number of priority levels, not the number of processes

- Implementation: a bitmap for fast look up
  - 140 queues → 5 integers
  - A few compares to find the first non-zero bit
  - Hardware instruction to find the first 1-bit
    - bsfl on Intel

Adjusting priority

- Goal: dynamically increase priority of interactive process
- How to determine interactive?
  - Sleep ratio
  - Mostly sleeping: I/O bound
  - Mostly running: CPU bound
- Implementation: per process sleep_avg
  - Before switching out a process, subtract from sleep_avg how many ticks a task ran
  - Before switching in a process, add to sleep_avg how many ticks it was blocked up to MAX_SLEEP_AVG (10 ms)

Calculating time slices

- Stored in field time_slice in struct task_struct
- Higher priority processes also get bigger time-slice
- task_timeslice() in sched.c
  - If (static_priority < 120) time_slice = (140-static_priority) * 20
  - If (static_priority >= 120) time_slice = (140-static_priority) * 5

Example time slices

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	19	5 ms

Problems with O(1) RR Scheduler

- Not easy to distinguish between CPU and I/O bound
  - I/O bound typically need better interactivity
  - CPU bound need sustained period of CPU at lower priority
- Finding right time slice isn't easy
  - Too small: good for I/O, but high context switch overhead
  - Too large: good for CPU bound jobs, but poor interactivity
- Prioritization by increasing timeslice isn't perfect
  - I/O bound processes want high priority, but small timeslice!
  - CPU bound processes want low priority but large timeslice!
  - Need complex aging to avoid starvation
- Priority is relative, but time slice is absolute
  - Nice 0, 1: time slice 100 and 95 msec: 5% difference!
  - Nice 19, 20: time slice 10 and 5: 100% difference!
- Time slice has to be multiple of tick, how to give priority to freshly woken up tasks even if their time slice has expired?
- Lots of heuristics to fix these problems
  - Problem: heuristics can be attacked, several attacks existed

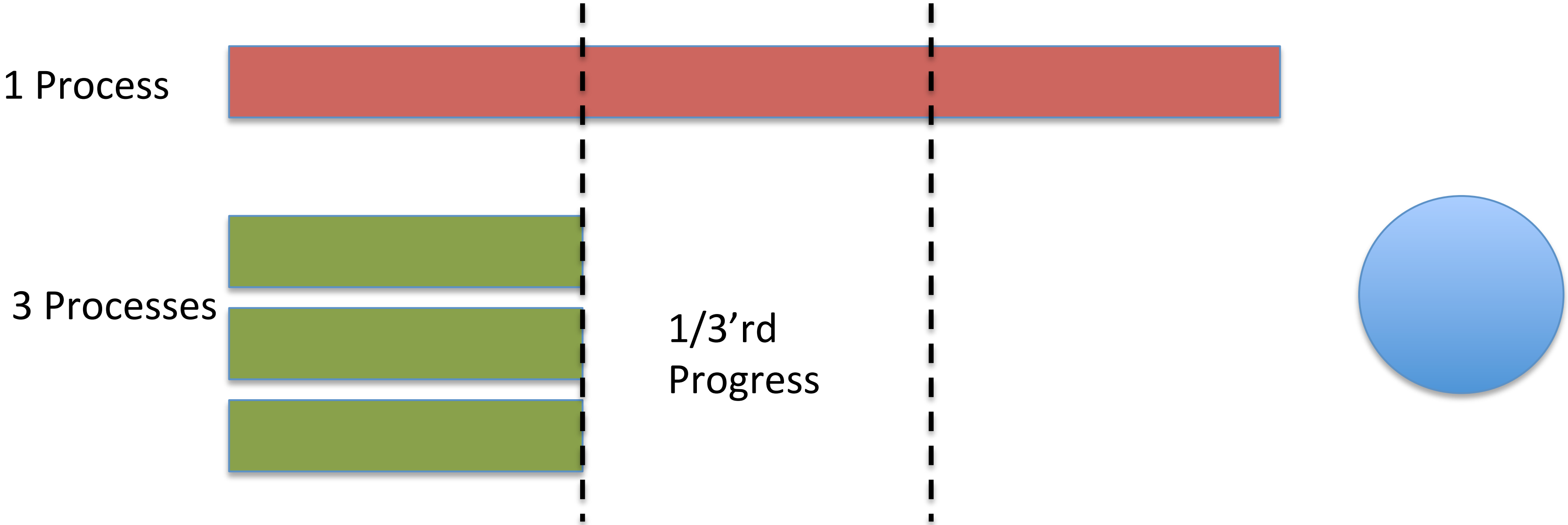
Outline

- Advanced scheduling issues
  - Multilevel queue scheduling
  - Multiprocessor scheduling issues

- Linux/Android Scheduling
  - Scheduler Architecture
  - Scheduling algorithms
    - O(1) RR scheduler
    - CFS scheduler
  - Other implementation issues

Completely Fair Scheduler (CFS)

- Introduced in kernel 2.6.23
- Models an ideal multitasking CPU
  - Infinitesimally small timeslice
  - n processes: each progresses uniformly at 1/n'th rate



- Problem: real CPU can't be split into infinitesimally small timeslice without excessive overhead

Completely Fair Scheduler

- Core ideas: dynamic time slice and order
- Don't use fixed time slice per task
  - Instead, fixed time slice across all tasks
  - Scheduling Latency
- Don't use round robin to pick next task
  - Pick task which has received least CPU so far
  - Equivalent to dynamic priority

Scheduling Latency

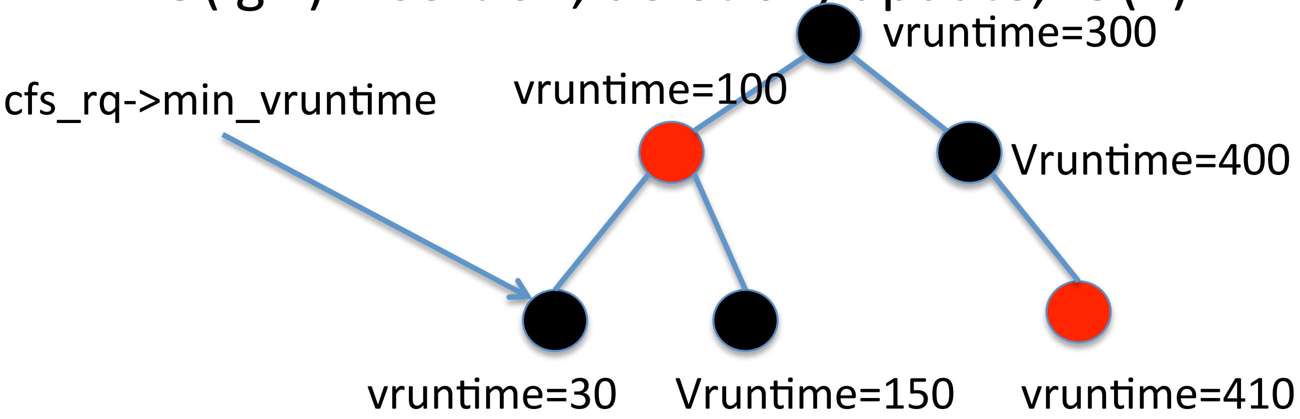
- Equivalent to time slice across all processes
  - Approximation of infinitesimally small
  - Default value is 20 msec
  - To set/get type: $ sysctl kernel.sched_latency_ns
- Each process gets equal proportion of slice
  - Timeslice(task) = latency/nr_tasks
  - Lower bound on smallest slice: default 4 msec
  - To set/get: $ sysctl kernel.sched_min_granularity_ns
  - Too many tasks? sched_latency = nr_tasks*min_granularity
- Priority through proportional sharing
  - Task gets share of CPU proportional to relative priority
  - Timeslice(task) = Timeslice(t) * prio(t) / Sum_all_t'(prio(t'))
- Maximum wait time bounded by scheduling latency

Picking the Next Process

- Pick task with minimum runtime so far
  - Tracked by vruntime member variable
  - Every time process runs for t ns, vruntime +=t (weighed by process priority)
- How does this impact I/O vs CPU bound tasks
  - Task A: needs 1 msec every 100 sec (I/O bound)
  - Task B, C: 80 msec every 100 msec (CPU bound)
  - After 10 times that A, B, and C have been scheduled
    - vruntime(A) = 10, vruntime(B, C) = 800
    - A gets priority, B and C get large time slices (10msec each)
- Problem: how to efficiently track min runtime?
  - Scheduler needs to be efficient
  - Finding min every time is an O(N) operation

Finding Lowest Runtime Efficiently

- Need to update vruntime and min_vruntime
  - When new task is added or removed
  - On every timer tick, context switch
- Balanced binary search tree
  - Red-Black Trees
  - Ordered by vruntime as key
  - O(lgN) insertion, deletion, update, O(1): find min



- Tasks move from left of tree to the right
- min_vruntime caches smallest value

Outline

- Advanced scheduling issues
  - Multilevel queue scheduling
  - Multiprocessor scheduling issues

- Linux/Android Scheduling
  - Scheduler Architecture
  - Scheduling algorithms
    - O(1) RR scheduler
    - CFS scheduler
  - Other implementation issues

Bookkeeping on each timer interrupt

- scheduler_tick()
  - Called on each tick
    - timer_interrupt ➔ do_timer_interrupt ➔ do_timer_interrupt_hook ➔ update_process_times

- If realtime and SCHED_FIFO, do nothing
  - SCHED_FIFO is non-preemptive
- If realtime and SCHED_RR and used up time slice, move to end of rq->active[prio]
- If SCHED_NORMAL and used up time slice
  - If not interactive or starving expired queue, move to end of rq->expired[prio]
  - Otherwise, move to end of rq->active[prio]
    - Boost interactive
- Else // SCHED_NORMAL, and not used up time slice
  - Break large time slice into pieces TIMESLICE_GRANULARITY

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