START CRYING

STOP

Scheduling: Case Studies

CS 161: Lecture 5

2/14/17
Scheduling Basics

- Goal of scheduling: Pick the “best” task to run on a CPU
  - Often a good idea to prioritize IO-bound tasks
    - If IO comes from user (e.g., keyboard, mouse), we want interactive programs to feel responsive
    - IO is typically slow, so start it early!
  - No starvation: All tasks should eventually get to run!
- Scheduling speed: The scheduler is PURE OVERHEAD
  - Your A2 scheduler must be better than round-robin!
- Case studies:
  - Linux 2.4: O(n) scheduler
  - Linux 2.6.early: O(1) scheduler
  - Linux 2.6.23+: O(log n) CFS scheduler
Linux O(n) Scheduler

Global runnable queue

Scheduler logic

Single spinlock
Each Task Has Three Priorities

• Two static priorities (do not change over lifetime of task)
  • “Real-time” priority
    • Between 1 and 99 for “real-time” tasks, 0 for normal tasks
    • RT task runs to completion unless it issues a blocking IO, voluntarily yields, or is preempted by higher priority RT task
  • Niceness priority
    • Normally 0; set by “nice” command to [-20, 19]
• One dynamic priority
  • Scheduler divides time into epochs
  • At start of epoch, each task is assigned a positive counter value (“time slice”)
    • Unit is “scheduler ticks” or “jiffies”
    • #define HZ 1000 //Rate that the timer interrupt fires
  • Task’s time slice: remaining CPU time that task can use during the current epoch (measured in 1/HZ long quanta)
  • Timer interrupt decrements counter for currently executing task

```c
void do_timer(){
    jiffies++;
    update_process_times();
}

void update_process_times(){
    struct task_struct *p = current;
    p->counter--;
    //Other bookkeeping involving //time statistics for this task //and the cpu the task is //running on.
}
```
Linux O(n) Scheduler

struct task_struct{
    unsigned long rt_priority; // For “real-time” tasks
    int static_prio;           // The task’s nice value
    int counter;               // The task’s remaining
                               // time slice, i.e., the
                               // task’s dynamic priority

...
void schedule()
{
    struct task_struct *next, *p;
    struct list_head *tmp;
    int this_cpu = ..., c;

    spin_lock_irq(&runqueue_lock); // Disable interrupts,
    // grab global lock.

    next = idle_task(this_cpu);
    c = -1000; // Best goodness seen so far.
    list_for_each(tmp, &runqueue_head)
    {
        p = list_entry(tmp, struct task_struct, run_list);
        if (can_schedule(p, this_cpu))
        {
            int weight = goodness(p);
            if (weight > c)
            {
                c = weight;
                next = p;
            }
        }
    }

    spin_unlock_irq(&runqueue_lock);
    switch_to(next, ...);
}
Calculating Goodness

```c
int goodness(struct task_struct *p){
    if(p->policy == SCHED_NORMAL){
        //Normal (i.e., non-"real-time") task
        if(p->counter == 0){
            //Task has used all of its
            //time for this epoch!
            return 0;
        }
        return p->counter + 20 - p->nice;
    }
    else{  
        //"Real-time" task
        return 1000 + p->rt_priority;
        //Will always be
        //greater than
        //priority of a
        //normal task
    }
}
```
void schedule(){
    struct task_struct *next, *p;
    struct list_head *tmp;
    int this_cpu = ..., c;

    spin_lock_irq(&runqueue_lock);
    next = idle_task(this_cpu);
    c = -1000; //Best goodness seen so far.
    list_for_each(tmp, &runqueue_head){
        p = list_entry(tmp, struct task_struct, run_list);
        if (can_schedule(p, this_cpu)) {
            int weight = goodness(p);
            if(weight > c) {
                c = weight;
                next = p;
            }
        }
    }
    spin_unlock_irq(&runqueue_lock);
    switch_to(next);
}
void schedule()
{
    struct task_struct *next, *p;
    struct list_head *tmp;
    int this_cpu = ..., c;

    spin_lock_irq(&runqueue_lock);
    next = idle_task(this_cpu);
    c = -1000; //Best goodness seen so far.
    list_for_each(tmp, &runqueue_head){
        p = list_entry(tmp, struct task_struct, run_list);
        if (can_schedule(p, this_cpu)) {
            int weight = goodness(p);
            if(weight > c){
                c = weight;
                next = p;
            }
        }
    }
    spin_unlock_irq(&runqueue_lock);
}

if(!c){//c==0, no good tasks!
    struct task_struct *p;
    spin_unlock_irq(&runqueue_lock);
    read_lock(&tasklist_lock);
    for_each_task(p){
        p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
    } //Counters for next epoch now set
    read_unlock(&tasklist_lock);
    spin_lock_irq(&runqueue_lock);
    goto repeat_schedule;
}

repeat_schedule:
Boost priority of interactive tasks which sleep often!
Summary: Linux O(n) Scheduler

- “Real-time” tasks have high, unchanging static priority
- Regular tasks have low static priority, and low, dynamically changing priority
  - Dynamic priority (time slice) set at epoch start
  - Time slice decremented as task uses CPU
- When scheduler must pick a task:
  - Search global run queue for task with best goodness
  - If all runnable tasks have goodness == 0, start a new epoch: recalculate all time slices, then search runnable queue again
  - Once a task has a counter of 0, it cannot run again until the new epoch arrives!
Another problem . . .

Global runnable queue

Scheduler logic

Single spinlock

CONTENTION
Why Was The O(n) Scheduler Tolerated?

Beyonce is angry.
The O(n) Scheduler Wasn’t Too Bad For Single-core Machines!

BEYONCE IS HAPPY
PREMATURE OPTIMIZATION IS THE ROOT OF ALL EVIL.

Simple is better unless proven otherwise.

Thy shall profile before thy shall optimize.
Linux O(1) Scheduler

- Goal 1: Get sublinear scheduling overhead
- Goal 2: Remove contention on a single, global lock

```
struct task_struct{
  unsigned long rt_priority;          //For "real-time" tasks
  int static_prio;                    //The task's nice value
  unsigned int time_slice;            //CPU time left in epoch
  int prio;                           //The task's "goodness"
  unsigned long sleep_avg;            //Estimate of how long
                                        //task spends blocked on
                                        //IO versus executing on
                                        //CPU; goes up when task
                                        //sleeps, goes down when
                                        //task runs on CPU

  ...
}
```
Linux O(1) Scheduler

- Goal 1: Get sublinear scheduling overhead
- Goal 2: Remove contention on a single, global lock

```c
struct runqueue{
    spinlock_t lock;
    struct task_struct *curr;
    prio_array_t *active;
    prio_array_t *expired;
    ...
};

struct prio_array{
    unsigned int nr_active;
    struct list_head queue[MAX_PRIO];
    unsigned long bitmap[BITMAP_SIZE];
};
```

Think of queue as being indexed by “goodness”
schedule()
- Find the first non-empty queue
- Run the first task in the list
void scheduler_tick(){ //Called by the timer interrupt handler.
    runqueue_t *rq = this_rq();
    task_t *p = current;
    
    spin_lock(&rq->lock);
    if(!--p->time_slice){
        dequeue_task(p, rq->active);
        p->prio = effective_prio(p);
        p->time_slice = task_timeslice(p);
        if(!TASK_INTERACTIVE(p) ||
            EXPIRED_STARVING(rq)){
            enqueue_task(p, rq->expired);
        }else{ //Add to end of queue.
            enqueue_task(p, rq->active);
        }
    }else{ //p->time_slice > 0
        if(TASK_INTERACTIVE(p)){
            //Probably won’t need the CPU
            //for a while.
            dequeue_task(p, rq->active);
            enqueue_task(p, rq->active); //Adds to end.
        }
    }
    spin_unlock(&rq->lock); //Later, timer handler calls schedule().
}

//Calculate “goodness”.
int effective_prio(task_t *p){
    if(rt_task(p))
        return p->prio;
    bonus = CURRENT_BONUS(p);
    //Bonus higher if
    //p->sleep_avg is big
    return p->static_prio - bonus;
    //static_prio is p’s
    //nice value
}

//Time slices calculated
//incrementally, unlike
//O(n) scheduler! High
//priority tasks get
//longer time slices.
Timer interrupt fires, scheduler runs $t_0$

nr_active: 3
bitmap: 01001
queue[5]:

nr_active: 0
bitmap: 00000
queue[5]:
Timer interrupt fires, scheduler moves $t_0$ to expired list, runs $t_1$. 

nr_active: 2
bitmap: 01001
queue[5]:

nr_active: 1
bitmap: 00100
queue[5]:

Goodness

Active

Expired
Timer interrupt fires, scheduler moves $t_1$ to expired list, runs $t_2$
Later, scheduler moves $t_2$ to the expired list

 nr_active: 0
 bitmap: 00000
 queue[5]:

 nr_active: 3
 bitmap: 00110
 queue[5]:

 Later, scheduler moves $t_2$ to the expired list.
Scheduler notices that `nr_active` is 0, and swaps the “active” and “expired” pointers: O(1) running time!
Summary: Linux O(1) Scheduler

- Per-processor scheduling data structures (eliminate global lock!)
  - Active array of queues (1 queue per priority level)
  - Expired array of queues (1 queue per priority level)
  - Task priority: ("real-time" priority) or (nice value + bonus)
- Scheduler picks first task from highest priority non-empty active queue
  - Finding that queue is O(1): find first 1 bit via hardware instruction
  - Dequeueing the first item in the queue is O(1)
- Timer interrupt decrements time slice for current task
  - If time slice is 0, move task to queue in expired array . . .
  - . . . unless task is interactive: maybe keep it active!
  - Eventually force even high priority interactive tasks into expired array (avoids starvation)
- When active array queues are empty, flip array pointers: O(1)
Multi-level Feedback Queuing

- Goal: Use static priorities and history to find the right scheduling strategy for a task
  - Scheduler uses task history to guess whether task is interactive (IO-bound, should get CPU when runnable) or CPU-bound
  - Static priorities let developers influence the default scheduling decisions
  - Linux O(1) scheduler is an example of MLFQ
- Rule 1: If Priority(A) > Priority(B), schedule A
- Rule 2: A task that sleeps a lot is likely to be interactive (and should receive a high priority)
- Rule 3: A task that uses its full time slice is probably demoted in priority (but see Rule 2)
- Rule 4: No starvation (every task eventually runs!)
Linux’s “Completely Fair Scheduler” (CFS)

• The O(1) scheduler is fast, but hackish
  • Heuristics (e.g., TASK_INTERACTIVE(p) and EXPIRED_STARVING(rq)) are complex, seem gross, have corner cases that are unfair
  • CFS invented to provide a more “elegant” solution
• As we’ll see, Linux politics and personality conflicts also played a role!
Linux’s “Completely Fair Scheduler” (CFS)

- For now, make these simplifying assumptions:
  - There is only one CPU
  - All tasks have the same priority
  - There are always T tasks ready to run at any moment

- Basic idea in CFS: each task gets 1/T of the CPU’s resources
  - CFS tries to model an “ideal CPU” that runs each task simultaneously, but at 1/T the CPU’s clock speed
  - Real CPU: Can only run a single task at once!
  - CFS tracks how long each task has actually run; during a scheduling decision (e.g., timer interrupt), picks the task with lowest runtime so far
Red-black binary tree
- Self-balancing: Insertions and deletions ensure that longest tree path is at most twice the length of any other path
- Guaranteed logarithmic time: Insertions, deletions, and searches all run in $O(\log N)$ time

CFS scheduler
- Associate each task with its elapsed runtime (nanosecond granularity)
- For each core, keep all runnable tasks in a red-black tree (insertion key is elapsed runtime)
- Next task to run is just the left-most task in tree!
CFS scheduler

- Associate each task with its elapsed runtime (nanosecond granularity)
- For each core, keep all runnable tasks in a red-black tree (insertion key is elapsed runtime)
- Next task to run is just the left-most task in tree!

Scheduler picks this task to run, removes it from tree
Timer interrupt fires, scheduler runs
- Now, \( t_2 \) no longer has the smallest elapsed runtime
- So, scheduler reinserts \( t_2 \) into the tree and runs \( t_0 \)!

Runs for 20 time units
Classic CFS Example

• Suppose there are two tasks:
  • Video rendering application (CPU-intensive, long-running, non-interactive)
  • Word processor (interactive, only uses CPU for bursts)
• Both tasks start with an elapsed runtime of 0
  • Video rendering task quickly accumulates runtime . . .
  • . . . but word processor’s runtime stays low (task is mainly blocked on IO)
• So, whenever word processor receives keyboard/mouse input and wakes up, it will be the left-most task, and immediately get scheduled
Task Priorities in CFS

/*
 * Nice levels are multiplicative, with a gentle 10% change for every
 * nice level changed. I.e. when a CPU-bound task goes from nice 0 to
 * nice 1, it will get ~10% less CPU time than another CPU-bound task
 * that remained on nice 0.
 *
 * The "10% effect" is relative and cumulative: from _any_ nice level,
 * if you go up 1 level, it's -10% CPU usage, if you go down 1 level
 * it's +10% CPU usage. (to achieve that we use a multiplier of 1.25.
 * If a task goes up by ~10% and another task goes down by ~10% then
 * the relative distance between them is ~25%.)
 */

static const int prio_to_weight[40] = {
    /* -20 */ 88761,    71755,    56483,    46273,    36291,
    /* -15 */ 29154,    23254,    18705,    14949,    11916,
    /* -10 */ 9548,     7620,     6100,     4904,     3906,
    /* -5  */ 3121,     2501,     1991,     1586,     1277,
    /*  0  */ 1024,     820,      655,      526,      423,
    /*  5  */ 335,      272,      215,      172,      137,
    /* 10  */ 110,      87,       70,       56,       45,
    /* 15  */ 36,       29,       23,       18,       15,
};
Task Priorities in CFS

- CFS incorporates static priorities by scaling task’s elapsed runtime

  \[
  \text{delta}\_\text{exec} = \text{now} - \text{curr}\rightarrow\text{exec}\_\text{start};
  \]

  \[
  \text{delta}\_\text{exec}\_\text{weighed} = \text{delta}\_\text{exec} \times \left(\frac{\text{NICE}_0\_\text{LOAD}}{\text{t}\rightarrow\text{load}\_\text{weight}}\right);
  \]

  \[
  \text{curr}\rightarrow\text{vruntime} += \text{delta}\_\text{exec}\_\text{weighed};
  \]

- The end result is that:
  - [nice=0] Virtual execution time equals physical execution time
  - [nice<0] Virtual execution time less than physical execution time
  - [nice>0] Virtual execution time greater than physical execution time

- curr->vruntime is used as a task’s key in the RB tree
Summary: Linux CFS Scheduler

- Scheduler associates each task with elapsed runtime (not timeslice!)
  - Nanosecond-granularity tracking instead of jiffy granularity
  - Growth rate is modulated by task priority
- Scheduler maintains a per-core red-black tree
  - Tasks inserted using elapsed runtimes as keys
  - Left-most task is the task to run next!
  - Scheduling operations take O(log n) time
- Is CFS actually better than the O(1) scheduler? Hmmm . . .
  - Nanosecond-granularity elapsed runtimes seems better than jiffy-granularity timeslices . . .
  - . . . but O(1) seems faster than O(log n)?
  - vruntime values do seem fairer than timeslices/goodness/etc . . .
  - . . . but CFS has janky heuristics, just like the O(1) scheduler (Ex: “Usually run left-most task, unless we want to run the most recently preempted task to preserve cache locality”)
