ECE391 Computer System Engineering Lecture 18

Dr. Zbigniew Kalbarczyk
University of Illinois at Urbana- Champaign

Fall 2022

Lecture Topics

- Scheduling
 - Philosophy
 - Algorithm
 - Data structures

Aministrivia

- MP3 Checkpoint 2
 - Due by 6:00pm Monday, October 24

ECE391 EXAM 2

- EXAM 2 November 1 (Tuesday)
 - Time: 7:00pm to 9:00pm
- Conflict Exam
 - Deadline to request conflict exam:

Thursday, October 27, by 5:00pm

(by email to: <u>kalbarcz@Illinois.edu</u>)

ECE391 EXAM 2

- Exam 2 Review Session in Class:
 - During the Lecture Time on Thursday, October 27
- Exam 2 Review Session in collaboration with HKN
 - TBD

ECE391 EXAM 2: Topics

Topics covered by the EXAM 2

- Material covered in lectures
 (Lecture 12, Lecture 13, Lecture 14, Lecture 15, Lecture 16, Lecture 17, Lecture 19)
- MP2 (ModeX, Tux controller)
- MP3.1 and MP3.2 (file system)
- Material covered in discussions

NOTE: Scheduling NOT included in Exam 2

NO Lecture on Tuesday, November 1

Scheduling Philosophy

Several types of jobs exist

- Interactive
 - examples: editors, GUIs
 - driven by human interaction (e.g., keystrokes, mouse clicks)
 - little or no work to do after each event
 - but important to respond quickly
- Batch
 - examples: compilation, simulation
 - usually only time to completion matters
 - want a fair share of CPU computation

Scheduling Philosophy (cont.)

- Real-time
 - examples: music, video, teleconferencing
 - periodic deadlines (e.g., 30 frames per second of video)
 - work often only useful if finished on time
- alternative taxonomy: I/O-bound vs. compute-bound
- Goal of scheduling: efficient, fair, and responsive (all at once!)

Scheduling Philosophy (cont.)

General strategy

- break time into slices
- allow interactive jobs to preempt the current job based on interrupts
 - typically, a job becomes runnable when an interrupt occurs (e.g., a key is pressed)
 - Linux checks for rescheduling after each interrupt, system call, and exception
- Linux 2.4 does NOT preempt code running in kernel (code may yield)
- Linux 2.6 DOES preempt code running in kernel to support real-time

General Scheduling Algorithm Used by Linux

Time broken into epochs

- each task given a quantum of time (in ticks of 10 milliseconds)
- run until no runnable task has time left
- then start a new epoch

Real-time jobs

- always given priority over non-real-time jobs
- prioritized amongst themselves

Static and dynamic priorities used for non-real-time jobs

General Scheduling Algorithm Used by Linux (cont.)

- Interactive jobs handled with heuristics
 - heuristic estimates job interactiveness
 - an interactive job
 - can continue to run after running out of time
 - takes turns with other interactive jobs
 - philosophy is that they don't usually use up quantum
 - heuristic ensures that job can't use lots of CPU and still be "interactive"

Task State (fields of task_struct)

- State field can be one of
 - TASK_RUNNING
 - task is executing currently or waiting to execute
 - task is in a run queue on some processor
 - TASK_INTERRUPTIBLE
 - task is sleeping on a semaphore/condition/signal
 - task is in a wait queue
 - can be made runnable by delivery of signal
 - TASK_UNINTERRUPTIBLE
 - task is busy with something that can't be stopped
 - e.g., a device that will stay in unrecoverable state without further task interaction
 - cannot be made runnable by delivery of signal

Task State (fields of task_t) (cont.)

TASK_STOPPED

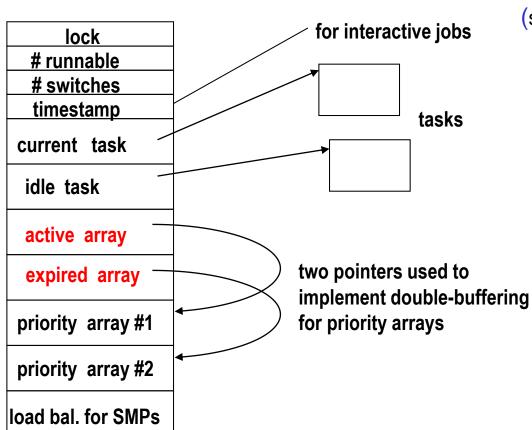
- task is stopped
- task is not in a queue; must be woken by signal

TASK_ZOMBIE

- task has terminated
- task state retained until parent collects exit status information
- task is not in a queue

The swapper process is always runnable (it's an idle loop)

Scheduling Data Structures



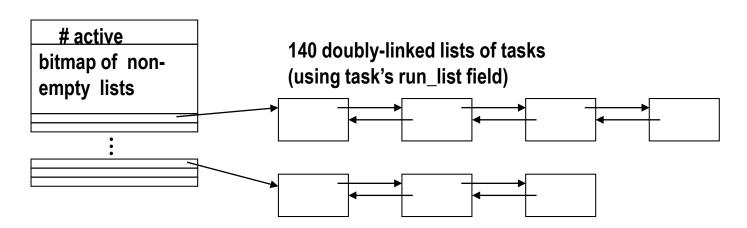
- Each processor has a run queue (struct runqueue in sched.c)
 - each run queue has two priority arrays, i.e., lists of tasks of each priority
 - they are double-buffered to implement epochs

Active processes: runnable processes that have not yet exhausted their time quantum/slice and hence, are allowed to run

Expired processes: runnable processes that have exhausted their time quantum/slice and hence, are not allowed to run until all active processes expire

Priority Array Structure (struct prio_array in sched.c)

- 100 real-time priorities
- 40 regular priorities
- One list per priority and a bitmap to make finding non-empty list fast



Scheduler Policies: SCHED_FIFO

First-In, First-Out real-time process. When the scheduler assigns the CPU to the process, it leaves the process descriptor in its current position in the runqueue list. If no other higher-priority real time process is runnable, the process will continue to use the CPU as long as it wishes, even if other real-time processes having the same priority are runnable.

- Run until they relinquish the CPU voluntarily
- Priority levels maintained
- Not pre-empted !!

Scheduler Policies: SCHED_RR

Round Robin real-time process. When the scheduler assigns the CPU to the process, it puts the process descriptor at the end of the runqueue list. This policy ensures a fair assignment of CPU time to all SCHED_RR real-time processes that have the same priority.

- Assigned a time slice and run till the time slice is exhausted.
- Once all RR tasks of a given priority level exhaust their time slices, their time slices are refilled and they continue running
- Priority levels are maintained

Scheduler Policies: SCHED_NORMAL

A conventional, time-shared process (used to be called SCHED_OTHER) for normal tasks. Task priority recalculated using a predefined formula. Tasks at the same priority are scheduled using round robin policy.

Rescheduling and Yielding

- Task can change (called a context switch) when
 - current task yields by calling schedule (sched_yield at user level)
 - current task runs out of time

- Other places where a task may yield implicitly
 - semaphores
 - wake_up_process
- At every timer tick (interrupt, generated every 10 milliseconds)
 - run a function (scheduler_tick) to reduce current task's time
 - executes with IF=0
 - interrupt is IRQ0

```
Linux Scheduler
void scheduler tick (void)
   unsigned long long now = sched clock ();
   struct task struct* p = current;
                                                    idle cpu returns 1 if
                                                                                    heduler_tick and task_running_tick
                         = smp_processor_id ();
   int cpu
   int idle at tick
                         = idle cpu (cpu);
                                                    CPU is running the idle task
   struct rq* rq
                         = cpu rq (cpu);
                                                                                                                           (kernel/sched.c)
                                               update cpu clock tracks nanoseconds
   update cpu clock (p, rq, now);
                                               since last tick/context switch
   if (!idle at tick)
                                               for all but idle task, call task_running_tick
       task running_tick (rq, p);
static void task running tick (struct rq* rq, struct task struct* p
   if (p->array != rq->active)
                                                  Take care of task which already expired
       set tsk need resched (p);
       return;
                                          Critical section begins
   spin lock (&rq->lock);
                                                                       Handle real-time tasks
   if (rt task (p)) {
       if ((p->policy == SCHED RR) && !--p->time slice) {
           p->time slice = task timeslice (p);
           p->first time slice = 0;
           set tsk need resched (p);
           requeue task (p, rq->active);
       goto out unlock;
                                              if a task's time slice expires
   if (!--p->time slice) {
       dequeue task (p, rq->active);
                                                  dequeue and reschedule the task
       set tsk need resched (p);
       p->prio = effective prio (p);
       p->time slice = task timeslice (p);
       p->first time slice = 0;
       if (!rq->expired timestamp)
           rq->expired timestamp = jiffies;
       if (!TASK INTERACTIVE (p) | expired starving (rq))
           enqueue task (p, rq->expired);
           if (p->static prio < rq->best expired prio)
              rq->best_expired_prio = p->static_prio;
       } else
           enqueue task (p, rq->active);
   } else
       /* Prevent long timeslices that allow a task to monopolize the
        * CPU by splitting up the timeslice into smaller pieces. We
        * requeue this task to the end of the list on this priority
        * level, which is a round-robin of tasks with equal priority. */
                                                                                      breaks long time slices into pieces
       if (TASK INTERACTIVE (p) &&
           !((task timeslice (p) - p->time slice) % TIMESLICE GRANULARITY (p)) &&
           (p->time slice >= TIMESLICE GRANULARITY (p)) && (p->array == rq->active)) {
           requeue_task (p, rq->active);
           set tsk need resched (p);
out unlock:
   spin unlock (&rq->lock);
                                             Critical section ends
```

Comments on scheduler_tick function

idle_cpu returns 1 if CPU is running the idle task

 update_cpu_clock tracks nanoseconds since last tick/context switch

For all but idle task, call task_running_tick

Comments on task_running_tick function

- If the task has already expired
 - make sure that it gets rescheduled on return from interrupt
 - by setting TIF NEED RESCHED

The remainder is a critical section for the run queue

- Next handle real-time tasks
 - real-time round-robin tasks take turns by placing themselves at the end of the list of their priority
 - otherwise real-time tasks just keep rescheduling (until yield or preemption by higher priority)

Comments on task_running_tick function

If a task's time slice expires

- take it out of the run queue
- mark it as needing to be removed from processor
- give it a new time slice
- if it's an interactive job, and expired tasks are not being starved by interactive ones, put it back into run queue (at end)
- normal tasks go into expired queue

Last block

- breaks long time slices into pieces
- round-robin between tasks at same priority

```
asmlinkage void schedule (void)
                                                              Linux Scheduler (Part1)
    task t* prev
                   = current;
    task t* next;
    runqueue t* rq = this rq ();
                                                                    schedule function (kernel/sched.c)
    prio array t* array;
    list t* queue;
    int idx;
    if (in interrupt ())
        BUG();
    release kernel lock (prev, smp processor id ());
                                            Sleep timestamp records time at which task leaves CPU
    prev->sleep timestamp = jiffies;
    spin lock_irq (&rq->lock);
                                         Critical section begins
    switch (prev->state) {
        case TASK INTERRUPTIBLE:
                                              if the task is trying to go to sleep check for receipt of signal
            if (signal pending (prev)) {
                prev->state = TASK RUNNING;
                break;
        default:
            deactivate_task (prev, rq); default case removes task from run queue
        case TASK RUNNING:
                                       no tasks are runnable
    if (!rq->nr running) {
        next = rq->idle;
        rq->expired timestamp = 0;
        goto switch tasks;
    array = rq->active;
                                       nothing is left in the active run queue
    if (!array->nr active) {
          Switch the active and expired arrays.
        rq->active = rq->expired;
        rq->expired = array;
        array = rq->active;
        rq->expired timestamp = 0;
                                                           find the next task to run
    idx = sched find first bit (array->bitmap);

    find first no-zero bit in the bitmask of the active set

    queue = array->queue + idx;
    next = list_entry (queue->next, task t, run_list);
                                                            - take the first task at the list indicataed by that bit
```

Comments on schedule function

- Schedule() function called when processor needs to change to new task
 - time has expired for current task, or
 - task has voluntarily yielded (by calling this function)
- Sleep timestamp records time at which task leaves CPU
- Remainder is run queue critical section
- If the task is trying to go to sleep
 - check for receipt of signal
 - handles race condition with sleeping in wait queue
- Default case removes task from run queue

Comments on schedule function (cont.)

- If no tasks are runnable
 - run the idle task
 - reset forced expiration of interactive tasks
- If nothing is left in the active run queue
 - the epoch is over
 - swap the priority arrays
 - and reset forced expiration of interactive tasks
- Find the next task to run
 - find first bit selects the priority
 - then take the first task at that priority (linked list)

Linux Scheduler (Part2)

schedule function (kernel/sched.c)

```
switch tasks:
    prefetch (next);
                                         clear need_resched flag for next time current task is scheduled
    prev->need resched = 0;
    if (prev != next) {
                                          Switch if newly selected task is not the current one
        rq->nr switches++;
        rg->curr = next;
        context switch (prev, next); call architecture-dependent switch function
          * The runqueue pointer might be from another CPU
          * if the new task was last running on a different
          * CPU - thus re-load it.
        barrier ();
        rq = this rq ();
                                              Critical section ends
    spin unlock irq (&rq->lock);
    reacquire kernel lock (current);
    return;
```

Comments on schedule function (cont.)

- This portion of the code implements the actual switch
- First clears need_resched flag for next time current task is scheduled

Switch only occurs if newly selected task is not the current one

- Switch does two things
 - does some accounting
 - calls architecture-dependent switch function (context_switch)