Scheduling

Reading: Stallings, Chapter 9

Note: many of the topics in the text are only peripherally relevant to general purpose operating systems. The topics covered in the course notes are the ones I consider most important.

Scheduling Definitions

- Scheduling refers to the **decision** as to which process to run next on a processor.
- Your text lists 3 types of scheduling
  - long term ("admitting" processes to the system)
  - medium term (which processes to swap in)
  - short term (deciding which process should run next)
- In terms of general purpose systems, the first two are obsolete.
- We focus only on short-term scheduling

Non-preemptive vs. Preemptive

- Non-preemptive:
  - once selected, a process continues running until either (1) it terminates or (2) it blocks itself
- Preemptive:
  - the currently running process can be interrupted, and the scheduler can select a different process to run
  - example? Timeslicing - via clock interrupt
- Again, while non-preemptive scheduling has its place in certain types of systems, general-purpose systems use preemptive scheduling

BTW, types of processes (jobs)

- Processor-bound (CPU-bound):
  - process performs relatively little I/O
  - tends to use all of time slice
- I/O bound:
  - performs heavy I/O
  - often interactive
  - uses small part of time slice before waiting on I/O

Scheduling criteria

- Turnaround time:
  - interval between submission of a process and its completion.
- Response time:
  - for interactive jobs, time between submission of input and corresponding output
- Throughput:
  - number of processes completed per unit time
  - meaningful for collections of CPU-bound jobs

Scheduling Criteria (cont.)

- Processor utilization
  - percentage of time processor is busy
- Priorities Enforced:
  - scheduling should favor higher-priority processes
- Fairness
  - no processes starved
  - processes of the same priority treated equally
  - can also be applied to users (Fair Share Scheduling)

Straws in the Wind?

- Policies that, alone, are not meaningful on general-purpose, time-shared computing systems
  - Shortest Job Next
  - Shortest Remaining Time
- Generally, the required information is not available, except when repeatedly running the same CPU-bound process…
- There do exist such applications…
- BUT, we'll still run them on a general-purpose system

First Come, First Served

- Simplest scheduling policy
- Just pick the process that has been in the ready queue (a.k.a. run queue) the longest
- Problem: tends to favor CPU-bound processes
- Why?

Round Robin

- Use clock interrupt to implement time slicing
- Issues: clock interval and time slice duration
  - not usually the same (multiple clock ticks per time slice)
  - too short, high overhead
  - too long, system not responsive enough
- Simple round robin still tends to favor CPU-bound processes. Why? Use all their timeslice

Multi-level Feedback Queue Scheduling

- Generally, we can't know how much longer a process will run.
- But we **do know** how long it has run already.
- Use this information as follows:
  - maintain multiple queues of processes
  - each queue has a different priority (high to low)
  - round-robin scheduling within each queue
  - a process that uses its full time slice drops in priority
- MLFQ is the basis for scheduling in:
  - Traditional Unix scheduling (discussed below)
  - Mac OS X (actually the Mach half of the OS)
  - Windows (NT) and some Linux schedulers

Multi-level feedback scheduling

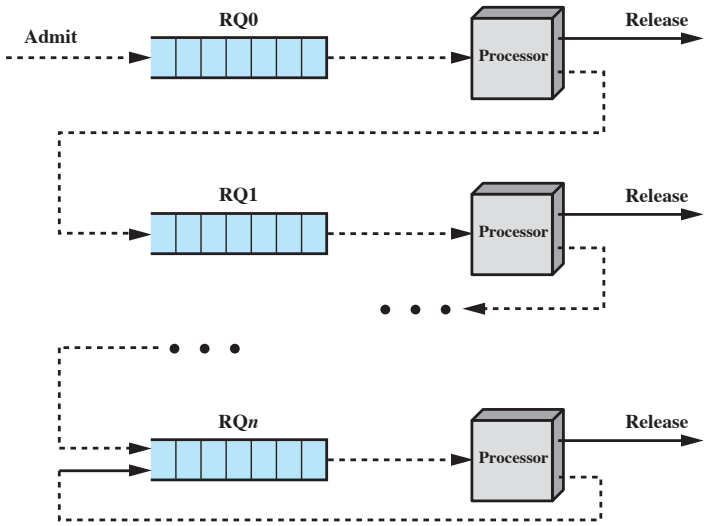


Figure 9.10 Feedback Scheduling

- Potential problem? Solutions? Starvation of CPU-bound jobs

Fair Share Scheduling

- Take into account users (and groups of users) in scheduling decisions
  - a user with many processes cannot "hog" the system
  - basically, give each user a virtual system that runs proportionally slower than the full system
- Has been implemented on various systems
- See example in Stallings…

Unix-like Scheduling

- Common features
  - all versions of Unix support a time-slice scheduler (time slice value has changed over the years)
  - process may also give up processor when it waits on an event
  - the scheduler always executes in the context of a user process
    - E.g., put self on wait queue, invoke schedule()
    - What about when a time slice expires?
- Let's look at some examples

Traditional Unix Scheduling

- Multi-level feedback scheduling (low val = high priority)
  - Priorities recomputed once per second
  - Base priority defines "bands" of processes
  - Nice value enables user to lower priority of a process

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + nice_j$$

where

$CPU_j(i)$ = measure of processor utilization by process j through interval i

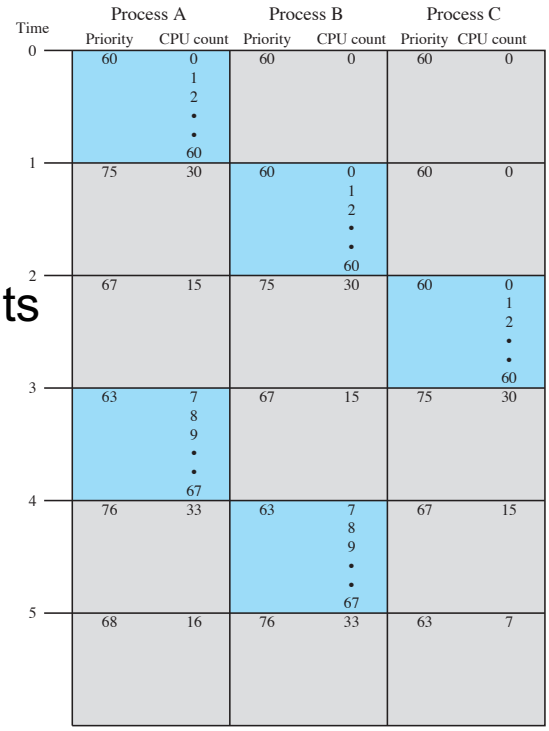
$P_j(i)$ = priority of process j at beginning of interval i; lower values equal higher priorities

$Base_j$ = base priority of process j

$nice_j$ = user-controllable adjustment factor

Traditional Unix Scheduling (example)

- Three CPU-bound processes
- Notice how the CPU count differs among processes at different scheduling points
- How would this affect I/O-bound processes?



Time	Process A		Process B		Process C	
	Priority	CPU count	Priority	CPU count	Priority	CPU count
0	60	0 1 2 • • 60	60	0	60	0
1	75	30	60	0 1 2 • • 60	60	0
2	67	15	75	30	60	0 1 2 • • 60
3	63	7 8 9 • • 67	67	15	75	30
4	76	33	63	7 8 9 • • 67	67	15
5	68	16	76	33	63	7

Example Scheduler (old Linux version)

- Operation
  - run whenever process is voluntarily relinquishes control or time-slice expires
  - selects “most deserving” process on run queue
- Priority
  - in priority field of task_struct
  - equal to the number of clock ticks (jiffies) for which it will run if it does not relinquish processor
  - counter field in task struct initially set to priority of process
  - decremented with each clock tick
- This basic approach used through 2.4 kernel (2003)

Process Selection

- Step through the run queue, and note the process with the highest priority
- Uses goodness() function to compute priority (linux/sch

```
/*
 * This is the function that decides how desirable a process is..
 * You can weigh different processes against each other depending
 * on what CPU they've run on lately etc to try to handle cache
 * and TLB miss penalties.
 *
 * Return values:
 *  -1000: never select this
 *      0: out of time, recalculate counters (but it might still
 *  be selected)
 *    +ve: "goodness" value (the larger, the better)
 *  +1000: realtime process, select this.
 */
static inline int goodness(struct task_struct * p,
       struct task_struct * prev, int this_cpu)
{
int weight;
```


Process Selection (cont.)

```
/*
 * Realtime process, select the first one on the
 * runqueue (taking priorities within processes
 * into account).
 */
if (p->policy != SCHED_OTHER)
return 1000 + p->rt_priority;

/*
 * Give the process a first-approximation goodness value
 * according to the number of clock-ticks it has left.
 *
 * Don't do any other calculations if the time slice is
 * over..
 */
weight = p->counter;
if (weight) {

#ifdef __SMP__
/* Give a largish advantage to the same processor...   */
/* (this is equivalent to penalizing other processors) */
if (p->processor == this_cpu)
weight += PROC_CHANGE_PENALTY;
#endif

/* .. and a slight advantage to the current process */
if (p == prev)
weight += 1;
}

return weight;
}
```

Example scheduler code

```
/*
 *  'schedule()' is the scheduler function. It's a very simple
 * and nice scheduler: it's not perfect, but certainly works
 * for most things.
 *
 * The goto is "interesting".
 *
 *   NOTE!!  Task 0 is the 'idle' task, which gets called when
 * no other tasks can run. It can not be killed, and it cannot
 * sleep. The 'state' information in task[0] is never used.
 */
asmlinkage void schedule(void)
{
  int lock_depth;
  struct task_struct * prev, * next;
  unsigned long timeout;
  int this_cpu;

  need_resched = 0;
  prev = current;
  this_cpu = smp_processor_id();
  if (local_irq_count[this_cpu])
    goto scheduling_in_interrupt;
  release_kernel_lock(prev, this_cpu, lock_depth);
  if (bh_active & bh_mask)
    do_bottom_half();
```


Example scheduler code

```
  spin_lock(&scheduler_lock);
  spin_lock_irq(&runqueue_lock);

  /* move an exhausted RR process to be last.. */
  if (!prev->counter && prev->policy == SCHED_RR) {
    prev->counter = prev->priority;
    move_last_runqueue(prev);
  }
  timeout = 0;
  switch (prev->state) {
    case TASK_INTERRUPTIBLE:
      if (prev->signal & ~prev->blocked)
        goto makerunnable;
      timeout = prev->timeout;
      if (timeout && (timeout <= jiffies)) {
        prev->timeout = 0;
        timeout = 0;
    makerunnable:
        prev->state = TASK_RUNNING;
        break;
      }
    default:
      del_from_runqueue(prev);
    case TASK_RUNNING:
  }
```

Scheduler

```
{
    struct task_struct * p = init_task.next_run;
    /*
     * This is subtle.
     * Note how we can enable interrupts here, even
     * though interrupts can add processes to the run-
     * queue. This is because any new processes will
     * be added to the front of the queue, so "p" above
     * is a safe starting point.
     * run-queue deletion and re-ordering is protected by
     * the scheduler lock
     */
    spin_unlock_irq(&runqueue_lock);
#ifdef __SMP__
    prev->has_cpu = 0;
#endif

/*
 * Note! there may appear new tasks on the run-queue during
 * this, as interrupts are enabled. However, they will be
 * put on front of the list, so our list starting at "p"
 * is essentially fixed.
 */
```


Example scheduler code

```
/* this is the scheduler proper: */
    {
      int c = -1000;
      next = idle_task;
      while (p != &init_task) {
        if (can_schedule(p)) {
          int weight = goodness(p, prev, this_cpu);
          if (weight > c)
            c = weight, next = p;
        }
        p = p->next_run;
      }

      /* Do we need to re-calculate counters? */
      if (!c) {
        struct task_struct *p;
        read_lock(&tasklist_lock);
        for_each_task(p)
          p->counter = (p->counter >> 1) + p->priority;
        read_unlock(&tasklist_lock);
      }
    }
  }
```

End of an "epoch"
Note: **all** processes!

Example scheduler code

```
if (prev != next) {
  struct timer_list timer;

  kstat.context_swtch++;
  if (timeout) {
    init_timer(&timer);
    timer.expires = timeout;
    timer.data = (unsigned long) prev;
    timer.function = process_timeout;
    add_timer(&timer);
  }
  get_mmu_context(next);
  switch_to(prev,next);

  if (timeout)
    del_timer(&timer);
}
spin_unlock(&scheduler_lock);

reacquire_kernel_lock(prev, smp_processor_id(), lock_depth);
return;

scheduling_in_interrupt:
   printk("Scheduling in interrupt\n");
*(int *)0 = 0;
}
```

Dispatcher

- Whereas the job of the scheduler is to select the next program to run, the dispatcher actually switches the selected process into the processor
- Tasks
  - Save current context, restore saved context
    - Processor registers, address space
  - Switch processor to user mode
  - Jump to the proper location (PC) in the user program to restart that program.
- Context switching happens a lot, needs to be fast!
- Setting up page tables would take a long time?

Improved Schedulers

- Problem(s) with the previous scheduler on a uniprocessor?
  Linear time
- What about a multiprocessor?
  Shared data structure
- Possible solutions?
  Multiple run queues

Linux O(1) Scheduler

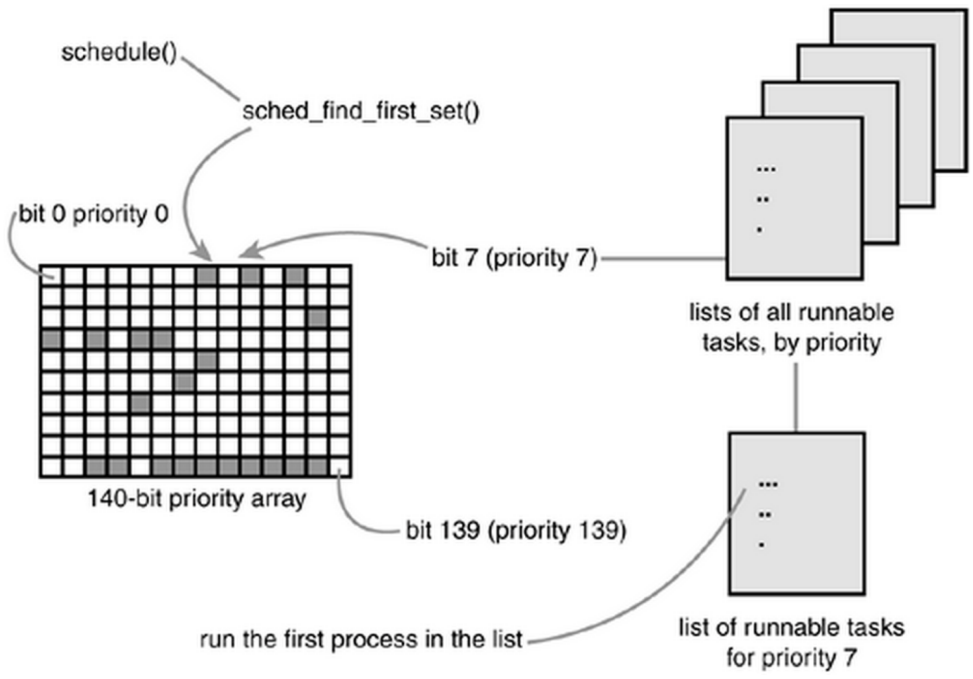
- Introduced with 2.6 kernel (circa 2004)
- Designed by Ingo Molnar
- Goals:
  - constant-time scheduling, regardless of num processes
  - SMP scalability – each processor gets its own runqueue
  - enhance SMP affinity; try to avoid migrating processes
  - fairness, no starvation
  - optimize for common case of only one or two runnable processes
  - scale to many processors with many processes (threads) each

 Source: http://www.makelinux.net/books/lkd2/ch04lev1sec2

Bitmaps and Priority Arrays

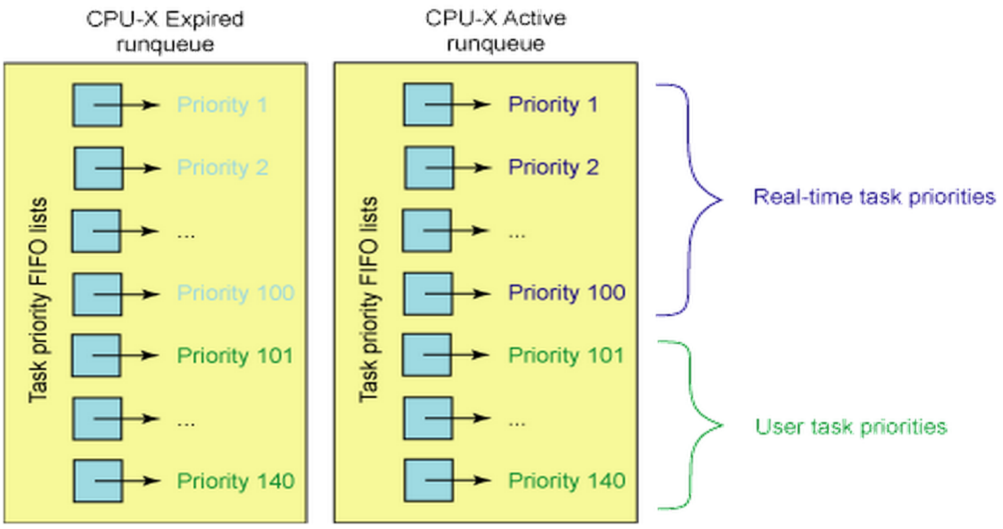
- Each runqueue has:
  - Bitmap identifying which priorities (of 140) have runnable processes
  - Two arrays of queue (active and expired)
  - Processes on active array have time slice remaining
  - Those on expired array have no time slice remaining
- When all time slices are expired, simply swap arrays
- Time slice of a processes is recalculated as when current slice expires
  - eliminates loop (and locks) to recalculate for all processes

Bitmaps and Priority Arrays



Priorities

- First 100 priorities reserved for real-time tasks
- Last 40 for “regular” user processes



schedule() function

- Resulting “guts” of schedule()

```
struct task_struct *prev, *next;
struct list_head *queue;
struct prio_array *array;
int idx;


prev = current;
array = rq->active;
idx = sched_find_first_bit(array->bitmap);
queue = array->queue + idx;
next = list_entry(queue->next, struct task_struct, run_list);
```

- Many processors support finding first bit set in a word (e.g., bsf on x86)
- Processes on same priority queue are executed round robin

Scheduler Wars?

- Completely Fair Scheduler (CFS)
  - Also designed by Molnar (2008)
  - maintain balance in providing CPU to processes
  - run queue replaced with red-black tree of sched entities
  - leftmost node has used processor least, so selected to run
- BFS
  - Designed by Con Kolivas (2009)
  - Not intended to scale to large SMPs, not intended for mainline kernel
  - Simple, single system-wide run queue, concept of “virtual deadlines”
  - Integrated into some Linux distributions
- BFS? Uh, google it. More proof that we made up, and continue to make up, computer “science.”

What we have learned…

- Main topics
  - What a PCB is, what it contains, how it is used
  - How system calls and signals work
  - A few details of bottom half handling
  - Mechanisms for suspension/resumption of processes
  - How an address space is implemented
  - How one process spawns another
  - How a (simple) scheduler/dispatcher
- In short, we have seen how the operating system realizes the concept of a process.
- Next up: Multiple threads within one process!!