CS 3204
Operating Systems

Lecture 18

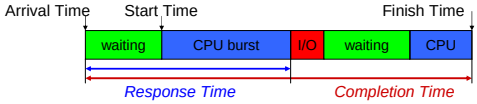
Godmar Back

Announcements

- Project 1 is due Tonight, 11:59pm
  - Honor code pledge: please include in design document
  - Please do a "make clean" before tarring up
- Office hours today 3-4
- Project 2 Help Sessions
  - Wed Mar 1 & Th Mar 2 from 7 to 9pm in MCB 126
- Reading assignments:
  - Stallings Chapter 7.1-7.4 (but read Pintos Project 2 documentation first!)

CPU Scheduling Terminology

- A job (sometimes called a task, or a job instance)
  - Activity that's scheduled: process or part of a process
- Arrival time: time when job arrives
- Start time: time when job actually starts
- Finish time: time when job is done
- Completion time (aka Turn-around time)
  - Finish time – Arrival time
- Response time
  - Time when user sees response – Arrival time
- Execution time (aka cost): time a job needs to execute



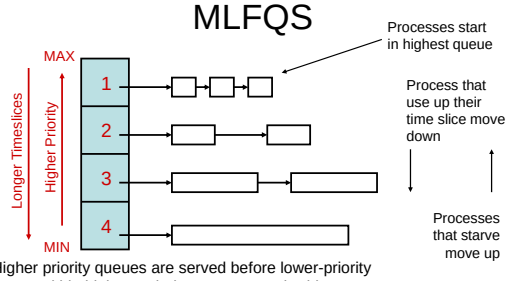
Scheduling: Recap

- FCFS: simple
  - unfair to short jobs & poor I/O performance (convoy effect)
- RR: helps short jobs
  - loses when jobs are equal length
- SPN: optimal average completion time
  - unfair to long jobs
  - requires knowing (or guessing) the future

Multi-Level Feedback Queue Scheduling

- Kleinrock 1969
- Want:
  - preference for short jobs (tends to lead to good I/O utilization)
  - longer timeslices for CPU bound jobs (reduces context-switching overhead)
- Problem:
  - Don't know type of each process – algorithm needs to figure out
- Use multiple queues
  - queue determines priority
  - usually combined with static priorities (nice values)
  - many variations of this

MLFQS

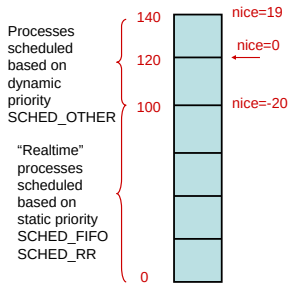


Higher priority queues are served before lower-priority ones - within highest-priority queue, round-robin

Only ready processes are in this queue - blocked processes leave queue and reenter same queue on unblock

Case Study: Linux Scheduler

- Variant of MLFQS
- 140 priorities
  - 0-99 “realtime”
  - 100-140 nonrealtime
- Dynamic priority computed from static priority (nice) plus “interactivity bonus”



Linux Scheduler (2)

- Instead of recomputation loop, recompute priority at end of each timeslice
  - dyn_prio = nice + interactivity bonus (-5…5)
- Interactivity bonus depends on sleep_avg
  - measures time a process was blocked
- 2 priority arrays (“active” & “expired”) in each runqueue (Linux calls ready queues “runqueue”)

Linux Scheduler (3)

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

/* find the highest-priority ready thread */
 idx = sched_find_first_bit(array->bitmap);
 queue = array->queue + idx;
 next = list_entry(queue->next, task_t, run_list);
```

```
/* Per CPU runqueue */
struct runqueue {
 prio_array_t *active;
 prio_array_t *expired;
 prio_array_t arrays[2];
 …
}
```

- Finds highest-priority ready thread quickly
- Switching active & expired arrays at end of epoch is simple pointer swap (“O(1)” claim)

Linux Timeslice Computation

- Linux scales *static* priority to timeslice
  - Nice [-20 … 0 … 19] maps to [800ms … 100 ms … 5ms]
- Various tweaks:
  - “interactive processes” are reinserted into active array even after timeslice expires
    - Unless processes in expired array are starving
  - processes with long timeslices are round-robin’d with other of equal priority at sub-timeslice granularity

Linux SMP Load Balancing

- Runqueue is per CPU
- Periodically, lengths of runqueues on different CPU is compared
  - Processes are migrated to balance load
- Migrating requires locks on both runqueues

```
static void double_rq_lock(
    runqueue_t *rq1,
    runqueue_t *rq2)
{
    if (rq1 == rq2) {
        spin_lock(&rq1->lock);
    } else {
        if (rq1 < rq2) {
            spin_lock(&rq1->lock);
            spin_lock(&rq2->lock);
        } else {
            spin_lock(&rq2->lock);
            spin_lock(&rq1->lock);
        }
    }
}
```


Basic Scheduling: Summary

- FCFS: simple
  - unfair to short jobs & poor I/O performance (convoy effect)
- RR: helps short jobs
  - loses when jobs are equal length
- SPN: optimal average completion time
  - unfair to long jobs
  - requires knowing (or guessing) the future
- MLFQS: approximates SPN without knowing execution time
  - Can be unfair to long jobs

Proportional Share Scheduling

- Aka “Fair-Share” Scheduling
- None of algorithms discussed so far give direct way of assigning CPU shares
  - E.g., give 30% of CPU to process A, 70% to process B
- Proportional Share algorithms assign “tickets” or “shares” to processes
  - Process get to use resource in proportion of their shares to total number of shares
- Lottery Scheduling, Stride Scheduling [Waldspurger 1995]

Lottery Scheduling

- Idea: number tickets between 1…N
  - every process gets p_i tickets according to importance
  - process 1 gets tickets [1… p_1-1]
  - process 2 gets tickets [p_1… p_2-1] and so on.
- Scheduling decision:
  - Hold a lottery and draw ticket, holder gets to run for next timeslice
- Nondeterministic algorithm
- Q.: what’s the complexity of this algorithm?
- Q.: what if a process is blocked?
- Q.: how to implement priority donation?

Scheduling Summary

- OS must schedule all resources in a system
  - CPU, Disk, Network, etc.
- CPU Scheduling affects indirectly scheduling of other devices
- Goals:
  - Minimizing latency
  - Maximing throughput
  - Provide fairness
- In Practice: some theory, lots of tweaking