Process Scheduling

Operating Systems

Oleg Goldshmidt

ogoldshmidt@computer.org

Lecture 4, Part I

Scheduling Criteria

- CPU utilization — percentage of time CPU is not idle
- throughput — # of processes completed per time unit
- turnaround time $=$ completion $-$ submission
  - includes loading, waiting in the ready queue, waiting for I/O, execution
- waiting time — only time in the ready queue (scheduling does not affect execution or I/O)
- response time $=$ first response $-$ submission
  - important for interactive processes
  - CPU keeps working while results are output
  - turnaround time is often I/O-limited
- variance in response time (emphasizes predictability)

Scheduling Algorithms

- First Come First Served (FCFS)
- Shortest Job First
- Priority Scheduling
- Round Robin

FCFS Scheduling

- by far the simplest (and non-preemptive)
- easy to implement: a FIFO ready queue with `push_back()` and `pop_front()` operations
- big problem: average waiting time may be long
  - e.g., 3 processes arrive at $t = 0$ with burst times of 24, 3, and $3\,\mathrm{ms}$
  - longest last: $\langle t_{wait} \rangle = (0 + 3 + 6)/3 = 3\,\mathrm{ms}$
  - longest first: $\langle t_{wait} \rangle = (0 + 24 + 27)/3 = 17\,\mathrm{ms}$
- another big problem: “convoy effect”
  - 1 CPU-bound process P, N I/O-bound processes
  - P holds the CPU, others finish I/O and wait
  - P waits for I/O, others finish bursts and wait

Shortest Job First Scheduling

- shortest next CPU burst first, use FCFS to break ties
- e.g., the processes in the ready queue have burst times $\{6, 8, 7, 3\}$
  - FCFS: $\langle t_{wait} \rangle = (0 + 6 + 14 + 21)/4 = 10.25$
  - SJF: $\langle t_{wait} \rangle = (0 + 3 + 9 + 16)/4 = 7$
- SJF is optimal in terms of average waiting time
- how do we know the length of the next burst?
  - long-term scheduling of batch jobs — specified by the users (who are unreliable)
  - prediction based on history (see "exponential historical average" in Silberschatz & Galvin)
- can be preemptive — shortest remaining time first

Shortest Remaining Time First

- need to take arrival times into account
- example: processes P_1, P_2, P_3, and P_4 arrive at times $\{0, 1, 2, 3\}$ with burst times $\{8, 4, 9, 5\}$
  - at $t = 1$ P_1 is preempted, P_2 runs
  - at $t = 2$ P_2 runs, the queue is $\{P_1(7), P_3(9)\}$
  - at $t = 3$ P_2 runs, the queue is $\{P_4(5), P_1(7), P_3(9)\}$
  - P_4 starts at $t = 5$
  - P_1 restarts at $t = 10$
  - P_3 starts at $t = 17$
- $\langle t_{wait} \rangle = ((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 6.5$
- without preemption:
  $\langle t_{wait} \rangle = (0 + (8 - 1) + (12 - 2) + (21 - 3))/4 = 8.75$

Priority Scheduling

- processes have priorities, scheduler chooses a process with the highest priority to run, uses FCFS to break ties
  - SJF is a special case: priority is the inverse of the (predicted) burst length
- priority is usually a number (from 0 to N)
  - need to know the convention: is 0 the highest or the lowest (we assume the highest)
- internal and external priorities
  - internal — derived from the process' characteristics
  - external — whose process is it? how much has the owner paid? will he grade my exam?
- can preempt lower priority processes
- avoid starvation — increase priority with age

Round-Robin Scheduling

- designed for time-sharing systems
- FCFS with preemption based on “time quanta”
- scheduler sets a timer to interrupt after 1 time quantum ($\Delta(t)$) and dispatches
  - if the burst is less than $\Delta(t)$ the process yields the CPU
  - otherwise the process is preempted
- long time quantum — FCFS (not very good)
- short time quantum — context switch overhead
- time quantum must be longer than context switch time
- turnaround time depends on the time quantum

Linux Scheduler: Policies

- preemptive multitasking scheduler; sources: `kernel/sched.c`, `include/linux/sched.h`
- each process has a scheduling policy
  - `SCHED_OTHER/SCHED_NORMAL` (normal) — do not run if there are real-time processes ready (i.e., in state `TASK_RUNNING`)
  - `SCHED_FIFO` (real-time) — can only be preempted by a real-time process with a higher priority
  - `SCHED_RR` (real-time) — round-robin scheduling between processes of the same priority
- see `sched_setscheduler(2)`, `sched_getscheduler(2)`
- we shall only cover normal (`SCHED_OTHER`) processes

Linux Scheduler: Basics

- CPU time is divided into "epochs"
- every process has a "time slice"
- at the end of a time quantum the scheduler chooses a process from the "runqueue"
  - must be ready, have the highest priority, have time left in the allocated "slice"
- during an epoch all ready processes are scheduled until each either exhausts its time slice or goes to sleep
- an epoch ends when there are no ready processes that have not finished their time slices
- at the end of an epoch a new epoch starts and every ready process is allocated a new time slice

Linux Scheduler: Priorities I

- static priority — inherited from parent
  - can be changed using `nice(2)`
    - a process is “nicer” if its priority is lower
  - also see `getpriority(2)`, `setpriority(2)`
- dynamic priority — modified according to what the process is doing
  - “affirmative action” for processes that are likely to wait in the medium to long term: increase their priority short term — they won’t be in the way in the future
  - I/O-bound processes are preferred over CPU-bound processes of the same static priority in the short term

Linux Scheduler: Priorities II

- priorities are integers from `0` to `139` (`MAX_PRIO-1`)
- real-time priorities are from `0` to `99` (`MAX_RT_PRIO-1`)
- normal priorities are from `100` to `139`
- higher numbers mean lower priorities
- default static priority for a normal process is `120`
- "niceness" = priority − 120
  - between `-20` and `19` for normal processes

```
#define NICE_TO_PRIO(nice) \
        ((nice)+MAX_RT_PRIO+20)
#define PRIO_TO_NICE(prio) \
        ((prio)-MAX_RT_PRIO-20)
#define TASK_NICE(p) \
        PRIO_TO_NICE((p)->static_prio)
```

Linux Scheduler: Timeslices

- timeslices scale with process priority
  - minimal timeslice — 5 ms
  - default timeslice — 100 ms
  - maximal timeslice — 800 ms
- even processes with the lowest priority get a timeslice of 5 ms

Linux Scheduler: Runqueues

- a (per CPU) runqueue contains "process descriptors" of all running and ready processes
  - `nr_running`: # of processes in the runqueue (not counting the swapper)
  - `curr`: pointer to descriptor of the running process
  - `idle`: pointer to the swapper's descriptor
  - "active" queue array: an array of queues of processes in state `TASK_RUNNING` and time left in allocated slices
  - "expired" queue array: an array of queues of processes in state `TASK_RUNNING` that have exhausted their slices
  - `expired_timestamp`: when did the 1st process move from "active" to "expired" during this epoch?

Linux Scheduler: Queue Arrays

- the "active" and "expired" queue arrays contain:
  - nr_active: # of processes in the queue
  - bitmap[ ]: bit vector of length MAX_PRIO
    - bit m is on if there are processes of priority m in the queue
    - the first process to run is the one with the highest priority in the active queue
    - the bitmap allows very efficient computation ("O(1) scheduler")
  - the array of queues itself (of length MAX_PRIO)
- at the end of an epoch processes are allocated new time slices and "active" and "expired" arrays are swapped

Linux Scheduler: Process Descriptors

- scheduling policy (SCHED_OTHER)
- prio: the priority of the process
- static_prio: the static priority of the process
- sleep_timestamp: when the context was last switched from the process (i.e., when was the last time it yielded the CPU)
- sleep_avg: average waiting time for the process
- time_slice: the remainder of the time slice in the current epoch

Linux Scheduler: Dynamic Priorities I

- every time a process goes to sleep [schedule()] we note the time (now = sched_clock() is the current time):

```
p->sleep_timestamp = now;
```

- every time a process wakes up [activate_task()] we update sleep_avg [in recalc_task_prio()]:

```
sleep_time = now - p->sleep_timestamp;
p->sleep_avg += sleep_time;
if (p->sleep_avg > MAX_SLEEP_AVG)
        p->sleep_avg = MAX_SLEEP_AVG;
```

- every clock tick [scheduler_tick()]:

```
if (p->sleep_avg)
        p->sleep_avg--;
```

Linux Scheduler: Dynamic Priorities II

- I/O-bound processes will have a high `sleep_avg`
- CPU-bound processes will have a low `sleep_avg`
- dynamic priority calculation [when time slice expires of when we return from wait — `effective_prio()`]

```
bonus = 10*(sleep_avg/MAX_SLEEP_AVG)-5
/* bonus is limited: (-5 < bonus < 5) */
prio = static_prio-bonus;
if (prio < MAX_RT_PRIO)
        prio = MAX_RT_PRIO;
if (prio > MAX_PRIO-1)
        prio = MAX_PRIO-1;
```

- what is "long wait"? `DEF_TIME_SLICE*10`

Linux Scheduler: Interactive Processes

- especially long waits (for input from user)
- special rights — additional time slices in the same epoch (for fast response)
- can lead to starvation of non-interactive processes: they will finish their time slices and will be stuck
  - starvation of “expired” processes is limited — when the limit is reached interactive processes do not get additional time slices in this epoch
  - the limit is proportional to the number of processes in the runqueue
    - if the load is high interactive processes get higher priority compared to non-interactive ones