Operating Systems

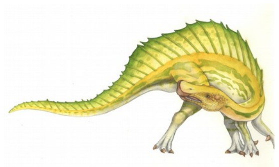
Isfahan University of Technology
Electrical and Computer Engineering Department
1400-1 semester

Zeinab Zali

Session 12: Thread and Linux Scheduling

Thread Scheduling

- Distinction between user-level and kernel-level threads
- **When threads supported, threads scheduled, not processes**
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope** (**PCS**) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope** (**SCS**) – competition among all threads in system



Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - **PTHREAD_SCOPE_PROCESS** schedules threads using PCS scheduling
  - **PTHREAD_SCOPE_SYSTEM** schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM
- Scheduling policies
  - FIFO
  - RR
  - OTHER

Linux scheduling

- pthread_attr_init(&attr)
- Config Scope
  - pthread_attr_getscope(&attr,&scope)

    pthread_attr_setscope(&attr,PTHREAD_SCOPE_PROCESS)

- Config scheduling algorithm
  - pthread_attr_getschedpolicy(&attr,&policy)

    pthread_attr_setschedpolicy(&attr,SCHED_FIFO)

- Shell commands
  - chrt

Linux Scheduling in Version 2.6.23 +

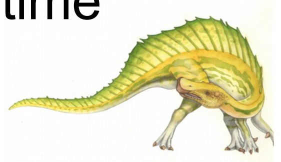
- Features:
  - constant order *O*(1) scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
    - Real-time range from 0 to 99 and time sharing range from 100 to 140
  - **Higher priority gets larger q**

Linux Scheduling in Version 2.6.23 +

- Scheduling in the Linux system is based on scheduling classes
  - Each class is assigned a specific priority
  - To decide which task to run next, the scheduler selects the highest-priority task belonging to the highest-priority scheduling class
- By using different scheduling classes, the kernel can accommodate different scheduling algorithms based on the needs of the system and its processes.
  - Ex. The scheduling criteria for a Linux server, may be different from those for a mobile device running Linux.

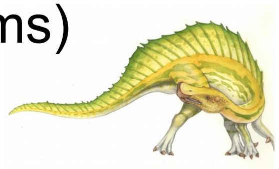
Completely Fair Scheduler (CFS)

- Standard Linux kernels implement two scheduling classes
  - a default scheduling class using the CFS scheduling algorithm
  - a real-time scheduling class
  - New scheduling classes can, of course, be added
- ***Completely Fair Scheduler*** (CFS)
  - assigns a fair proportion of CPU processing time to each task (the quantum value)
  - **Scheduling decision:** CFS will pick the process with the lowest vruntime to run next
  - **Scheduling quantum (Time Slice):** Tasks with lower nice values receive a higher proportion of CPU processing time



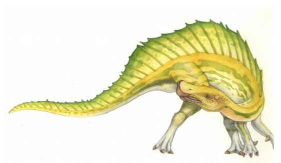
CFS: Nice Value

- Lower nice value indicates a higher relative priority
- Tasks with lower nice values receive a higher proportion of CPU processing time than tasks with higher nice values.
  - If a task increases its nice value from, say, 0 to +10, it is being nice to other tasks in the system by lowering its relative priority (allow other tasks to scheduled earlier and for longer quantum)
- sched_latency: (Targeted latency) : an interval of time during which every runnable task should run at least once.
  - Proportions of CPU time are allocated from the value of sched latency
  - But what if there are “too many” processes running? Wouldn’t that lead to too small of a time slice, and thus too many context switches?
    - Yes! Solution is considering min granularity (Ex: 6ms)



CFS: Virtual runtime

- CFS scheduler maintains per task **virtual run time** in variable **`vruntime`**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
  - For runtime= 200, what is **virtual runtime** for normal, high, low priority?
- To decide next task to run, scheduler picks task with lowest virtual run time
- Which one has higher priority?
  - IO bound or CPU bound process?



Calculating time slice and vruntime

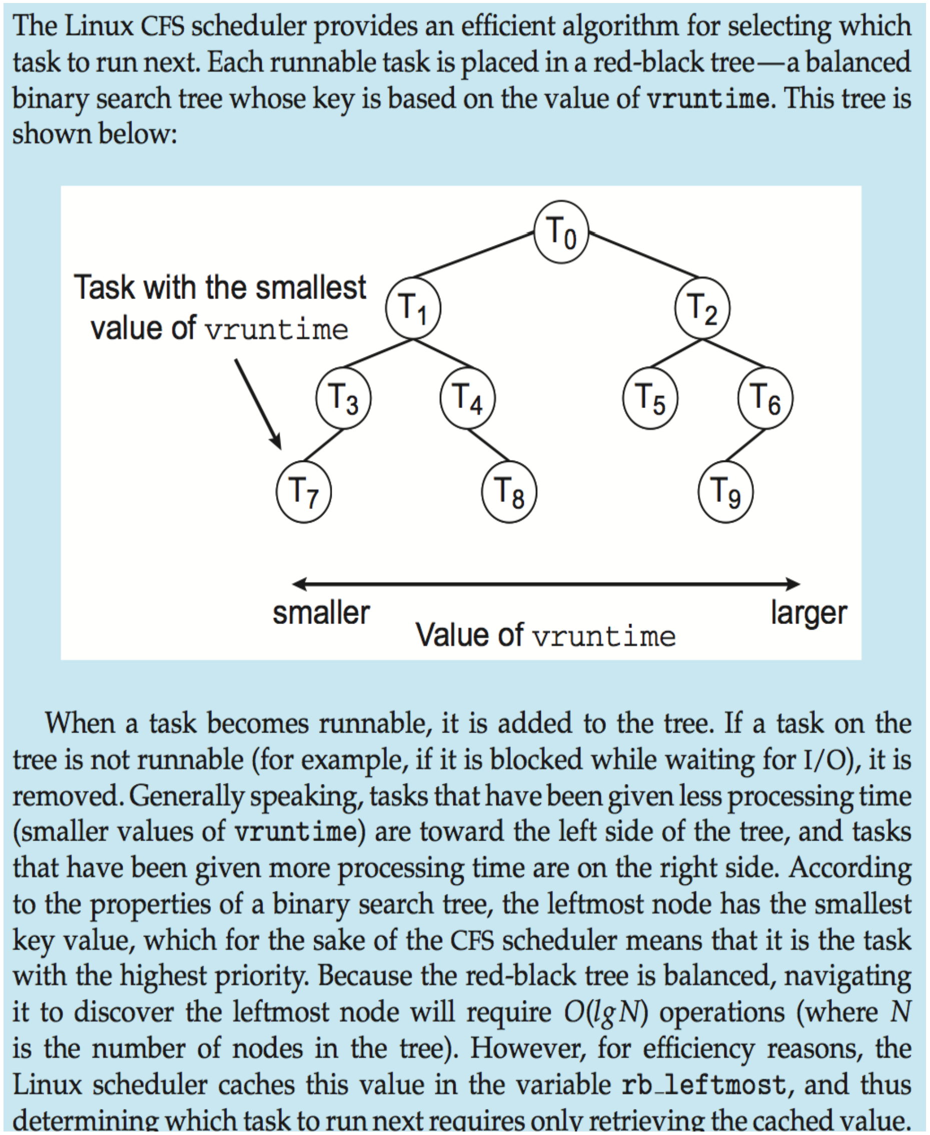
- CFS maps the nice value of each process to a weigh

```
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,
};
```

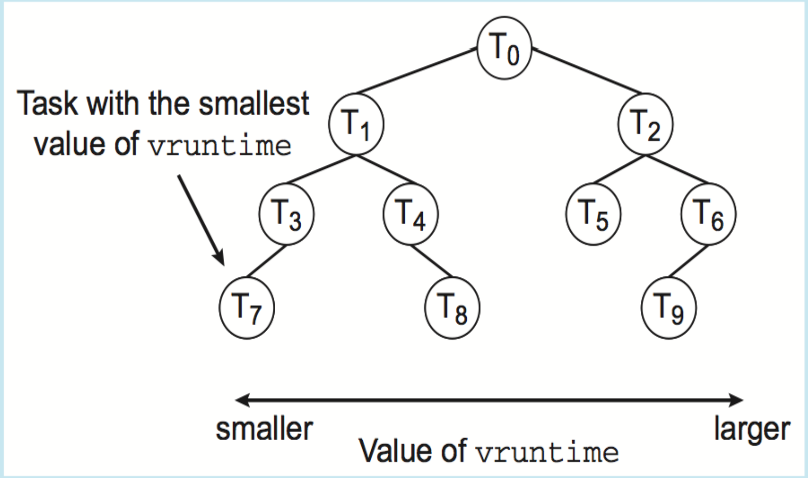
$$\mathtt{time\_slice}_k = \frac{\mathtt{weight}_k}{\sum_{n=0}^{n-1} \mathtt{weight}_i} \cdot \mathtt{sched\_latency}$$

$$\mathtt{vruntime}_i = \mathtt{vruntime}_i + \frac{\mathtt{weight}_0}{\mathtt{weight}_i} \cdot \mathtt{runtime}_i$$

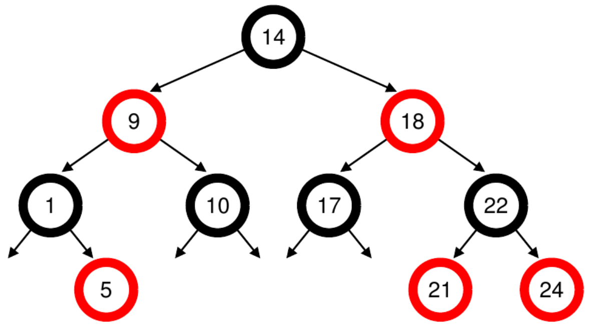
CFS: red-black tree



The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:

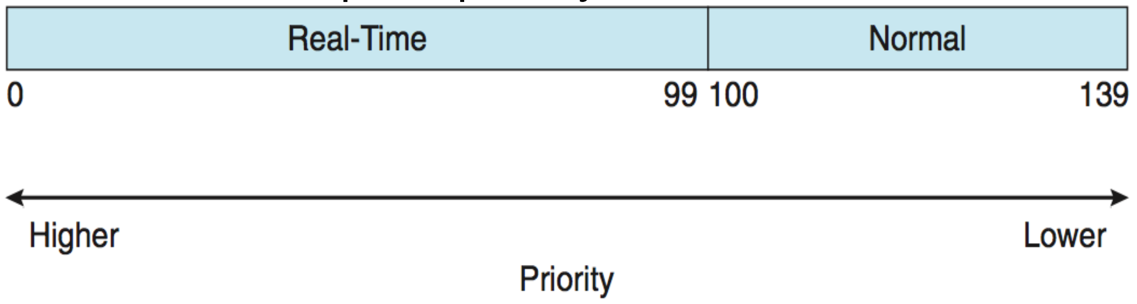


When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.



Linux Scheduling: realtime

- Linux also implements real-time scheduling using the POSIX standard
  - SCHED FIFO
  - SCHED RR
- real-time policy runs at a higher priority than normal (non-realtime)
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139



Top command represents PR and NI values for processes
for normal processes (sched_other): PR = 20 + NI
(NI is nice and ranges from -20 to 19)
for real time processes (sched_fifo or sched_rr): PR = - 1 – real_time_priority
(real_time_priority ranges from 1 to 99)
Try chrt -m

Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
- **Queueing Models**
- **Simulations**
- **Implementation**