CPU Scheduling

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Today’s Topics

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multi-processor Scheduling
- Operating Systems Examples
CPU Scheduling

- **CPU scheduling**
  - Deciding which process to run next, given a set of runnable processes.
  - Happens frequently, hence should be fast.

- **Scheduling points**

![Diagram of CPU scheduling states](image-url)
Schedulers (1)

- **Short-term scheduler (or CPU scheduler)**
  - Selects which process should be executed next
  - Sometimes the only scheduler in a system
  - Invoked frequently (milliseconds)

- **Long-term scheduler (or job scheduler)**
  - Selects which processes should be brought into the ready queue
  - Invoked infrequently (seconds, minutes)
  - Controls the degree of multiprogramming

- **Long-term scheduler strives for good process mix**
Schedulers (2)

- Medium-term scheduler
  - To decrease the degree of multiple programming due to resource shortage
  - Swapping
    - Remove process from memory
    - Store on disk
    - Bring back in from disk to continue execution
To maximize CPU utilization in multiprogramming

Process execution consists of

- CPU execution
- I/O wait
Execution Characteristics (1)
Execution Characteristics (2)

- **CPU burst vs. I/O burst**
  - A CPU-bound process
  - An I/O-bound process

![Diagram](image)
CPU Scheduling (1)

- Short-term scheduler selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is nonpreemptive
- All other scheduling is preemptive
CPU Scheduling (2)

- Non-preemptive scheduling
  - The scheduler waits for the running job to voluntarily yield the CPU.
  - Jobs should be cooperative.

- Preemptive scheduling
  - The scheduler can interrupt a job and force a context switch.
  - What happens
    - If a process is preempted in the midst of updating the shared data?
    - If a process in a system call is preempted?
CPU Scheduling (3)

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
CPU Scheduling (4)

- Scheduling Algorithm Optimization Criteria
  - Max CPU utilization
  - Max throughput
  - Min turnaround time
  - Min waiting time
  - Min response time
CPU Scheduling (4)

- Starvation
  - A situation where a process is prevented from making progress because another process has the resource it requires.
    - Resource could be the CPU or a lock.
  - A poor scheduling policy can cause starvation
    - If a high-priority process always prevents a low-priority process from running on the CPU.
  - Synchronization can also cause starvation
    - One thread always beats another when acquiring a lock.
    - Constant supply of readers always blocks out writers.
CPU Scheduling (5)

Scheduling Goals

- All systems
  - No starvation
  - Fairness: giving each process a fair share of the CPU
  - Balance: keeping all parts of the system busy

- Batch systems
  - Throughput: maximize jobs per hour
  - Turnaround time: minimize time between submission and termination
  - CPU utilization: keep the CPU busy all the time

- Interactive systems
  - Response time: respond to requests quickly
  - Proportionality: meet users’ expectations

- Real-time systems
  - Meeting deadlines: avoid losing data
  - Predictability: avoid quality degradation in multimedia system
First-Come, First-Served (FCFS)

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>24</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3</td>
</tr>
</tbody>
</table>

Suppose that the processes arrive in the order: $P_1$, $P_2$, $P_3$

- The Gantt Chart for the schedule is:

```
0                                                        24          27         30
```

- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$
First-Come, First-Served (FCFS)

Suppose that the processes arrive in the order: P2, P3, P1

- The Gantt chart for the schedule is:

```
   P2 | P3 | P1
   0  | 3  | 6
   +---+---+---
   30
```

- Waiting time for P1 = 6; P2 = 0; P3 = 3
- Average waiting time: \((6 + 0 + 3)/3 = 3\)
- Much better than previous case

- **Convoy effect**
  - Short process behind long process
Shortest-Job-First (SJF)

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time

- SJF is optimal
  - Gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
    - Could ask the user or estimate

- SJF may starve long processes
Example of SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>6</td>
</tr>
<tr>
<td>$P_2$</td>
<td>8</td>
</tr>
<tr>
<td>$P_3$</td>
<td>7</td>
</tr>
<tr>
<td>$P_4$</td>
<td>3</td>
</tr>
</tbody>
</table>

- SJF scheduling chart

- Average waiting time = $(3 + 16 + 9 + 0) / 4 = 7$
Determining Length of Next CPU Burst

- Exponential averaging
  - Predicting the length of next CPU burst
  - Using the length of previous CPU bursts

1. \( t_n \) = actual length of \( n^{th} \) CPU burst
2. \( \tau_{n+1} \) = predicted value for the next CPU burst
3. \( \alpha, 0 \leq \alpha \leq 1 \)
4. Define: \( \tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n \)
   \[
   \tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \ldots + (1 - \alpha)^j \alpha t_{n-j} + \ldots + (1 - \alpha)^{n+1} \tau_0
   \]

- Commonly, \( \alpha \) set to \( \frac{1}{2} \)
Prediction of the Length of Next CPU Burst

CPU burst ($t_i$)  6  4  6  4  13  13  13  ...  
"guess" ($\tau_i$)  10  8  6  6  5  9  11  12  ...  

![Graph showing prediction of CPU burst length](graph.png)
Shortest Remaining Time First (SRTF)

- Preemptive version of SJF.
- If a new process arrives with CPU burst length less than remaining time of current executing process, preempt.

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>$P_3$</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>
Round Robin (RR)

- Ready Q is treated as a circular FIFO Q.
- Each job is given a time slice (or time quantum).
  - Usually 10-100 ms.
- Great for timesharing
  - No starvation
  - Typically, higher average turnaround time than SJF, but better response time.
- Preemptive
- What do you set the quantum to be?
  - A rule of thumb: 80% of the CPU bursts should be shorter than the time quantum.
- Treats all jobs equally
Example of RR (1)

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>24</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3</td>
</tr>
</tbody>
</table>

The Gantt chart is:

<table>
<thead>
<tr>
<th></th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_1$</th>
<th>$P_1$</th>
<th>$P_1$</th>
<th>$P_1$</th>
<th>$P_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>0</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>14</td>
<td>18</td>
<td>22</td>
<td>26</td>
</tr>
</tbody>
</table>

Typically, higher average turnaround than SJF, but better *response*
Example of RR (2)

- Time quantum and context switch time

- Process time = 10

- Quantum
  - 12
  - 6
  - 1

- Context switches
  - 0
  - 1
  - 9
Example of RR (3)

- Turnaround time varies with the time quantum
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer = highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem
  - Starvation – low priority processes may never execute
- Solution
  - Aging – as time progresses increase the priority of the process
Priority Inversion Problem

- A situation where a higher-priority job is unable to run because a lower-priority job is holding a resource it needs, such as a lock.

- What really happened on Mars?
Solutions to Priority Inversion

- **Priority inheritance protocol (PIP)**
  - The higher-priority job can donate its priority to the lower-priority job holding the resource it requires.

- **Priority ceiling protocol (PCP)**
  - The priority of the low-priority thread is raised immediately when it gets the resource.
  - The priority ceiling value must be predetermined.
Multilevel Queue

- Ready queue is partitioned into separate queues
  - **Foreground** (interactive)
  - **Background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - Foreground – RR
  - Background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling
    - i.e.) serve all from foreground then from background
    - Possibility of starvation.
  - Time slice
    - Each queue gets a certain amount of CPU time which it can schedule amongst its processes
    - i.e.) 80% to foreground in RR and 20% to background in FCFS
Multilevel Queue Scheduling

highest priority

- system processes

interactive processes

interactive editing processes

batch processes

student processes

lowest priority
Multilevel Feedback Queue

- Multilevel queue scheduling, which allows a job to move between the various queues.
- Queues have priorities.
  - Batch, interactive, system, CPU-bound, I/O-bound, ...
- When a process uses too much CPU time, move to a lower-priority queue.
  - Leaves I/O-bound and interactive processes in the higher-priority queues.
- When a process waits too long in a lower priority queue, move to a higher-priority queue.
  - Prevents starvation.
Multi-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- Homogeneous processors within a multiprocessor
  - Asymmetric multiprocessing
    - Only one processor accesses the system data structures, alleviating the need for data sharing
  - Symmetric multiprocessing (SMP)
    - Each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
- Processor affinity
  - Process has affinity for processor on which it is currently running
  - Soft affinity
  - Hard affinity
NUMA and CPU Scheduling

CPU

fast access

memory

slow access

computer

CPU

fast access

memory
UNIX Scheduler (1)

- Characteristics
  - Preemptive
  - Priority-based
    - The process with the highest priority always runs.
    - 3 – 4 classes spanning ~170 priority levels (Solaris 2)
  - Time-shared
    - Based on timeslice (or quantum)
  - MLFQ (Multi-Level Feedback Queue)
    - Priority scheduling across queues, RR within a queue.
    - Processes dynamically change priority.
UNIX Scheduler (2)

- General principles
  - Favor I/O-bound processes over CPU-bound processes
    - I/O-bound processes typically run using short CPU bursts.
    - Provide good interactive response; don’t want editor to wait until CPU hog finishes quantum.
    - CPU-bound processes should not be severely affected.
  - No starvation
    - Use aging
  - Priority inversion?
Linux 2.4 Scheduling (1)

- General characteristics
  - Linux offers three scheduling algorithms.
    - A traditional UNIX scheduler: SCHED_OTHER
    - Two “real-time” schedulers (mandated by POSIX.1b): SCHED_FIFO and SCHED_RR
  - Linux scheduling algorithms for real-time processes are “soft real-time”.
    - They give the CPU to a real-time process if any real-time process wants it.
    - Otherwise they let CPU time trickle down to non real-time processes.
  - Here, we study the scheduling algorithm implemented in the Linux 2.4.18 kernel.
Linux 2.4 Scheduling (2)

- **Priorities**
  - **Static priority**
    - The maximum size of the time slice a process should be allowed before being forced to allow other processes to complete for the CPU.
  - **Dynamic priority**
    - The amount of time remaining in this time slice; declines with time as long as the process has the CPU.
    - When its dynamic priority falls to 0, the process is marked for rescheduling.
  - **Real-time priority**
    - Only real-time processes have the real-time priority.
    - Higher real-time priority values always beat lower values.
Related fields in the task structure

- **long counter;**
  - time remaining in the task’s current quantum (represents dynamic priority)
  - task’s nice value, -20 to +19.
  - (represents static priority)

- **long nice;**
  - task’s nice value, -20 to +19.

- **unsigned long policy;**
  - SCHED_OTHER, SCHED_FIFO, SCHED_RR

- **struct mm_struct *mm;**
  - points to the memory descriptor

- **int processor;**
  - processor ID on which the task will execute
  - ~0 if the task is not running on any CPU
  - (1<<cpu) if it’s running on a CPU

- **unsigned long cpus_runnable;**
  - CPUs allowed to run

- **unsigned long cpus_allowed;**

- **struct list_head run_list;**
  - head of the run queue

- **unsigned long rt_priority;**
  - real-time priority
Linux 2.4 Scheduling (4)

- **Scheduling policies**
  - **SCHED_OTHER**
  - **SCHED_FIFO**
    - A real-time process runs until it either blocks on I/O, explicitly yields the CPU, or is preempted by another real-time process with a higher rt_priority.
    - Acts as if it has no time slice.
  - **SCHED_RR**
    - It’s the same as SCHED_FIFO, except that time slices do matter.
    - When a SCHED_RR process’s time slice expires, it goes to the back of the list of SCHED_FIFO and SCHED_RR processes with the same rt_priority.
### Scheduling quanta

- Linux gets a timer interrupt or a *tick* once every 10ms on IA-32. (HZ=100)
  - Alpha port of the Linux kernel issues 1024 timer interrupts per second.
- Linux wants the time slice to be around 50ms.
  - Decreased from 200ms (in v2.2)

```c
/* v2.4 */
#if HZ < 200
#define TICK_SCALE(x)    ((x) >> 2)
#endif
#define NICE_TO_TICKS(nice)    (TICK_SCALE(20-(nice))+1)

/* v2.2 */
#define DEF_PRIORITY     (20*HZ/100)
```
Linux 2.4 Scheduling (6)

- **Epochs**
  - The Linux scheduling algorithm works by dividing the CPU time into epochs.
    - In a single epoch, every process has a specified time quantum whose duration is computed when the epoch begins.
    - The epoch ends when all runnable processes have exhausted their quantum.
    - The scheduler recomputes the time-quantum durations of all processes and a new epoch begins.
  - The base time quantum of a process is computed based on the nice value.
Selecting the next process to run

repeat_schedule:
next = idle_task(this_cpu);
c = -1000;
list_for_each(tmp, &runqueue_head) {
        p = list_entry(tmp, struct task_struct, run_list);
        if (can_schedule(p, this_cpu)) {
                int weight = goodness(p, this_cpu, prev->active_mm);

                if (weight > c)
                        c = weight, next = p;
        }
}
Recalculating counters

```c
if (unlikely(!c)) {
    /* New epoch begins ... */
    struct task_struct *p;

    spin_unlock_irq(&runqueue_lock);
    read_lock(&tasklist_lock);
    for_each_task(p)
        p->counter = (p->counter >> 1) +
            NICE_TO_TICKS(p->nice);
    read_unlock(&tasklist_lock);
    spin_lock_irq(&runqueue_lock);
    goto repeat_schedule;
}
```
Calculating goodness()

```c
static inline int goodness (p, this_cpu, this_mm) {
    int weight = -1;
    if (p->policy == SCHED_OTHER) {
        weight = p->counter;
        if (!weight) goto out;
        if (p->mm == this_mm || !p->mm)
            weight += 1;
        weight += 20 - p->nice;
        goto out;
    }
    weight = 1000 + p->rt_priority;
    out: return weight;
}
```

- `weight = 0`  
  p has exhausted its quantum.
- `0 < weight < 1000`  
  p is a conventional process.
- `weight >= 1000`  
  p is a real-time process.
Linux scheduler is not so scalable!

- A single run queue is protected by a run queue lock.
  - As the number of processors increases, the lock contention increases.
- It is expensive to recalculate goodness() for every task on every invocation of the scheduler.
  - A profile of the kernel taken during the VolanoMark runs shows that 37-55% of total time spent in the kernel is spent in the scheduler.
  - The VolanoMark benchmark establishes a socket connection to a chat server for each simulated chat room user. For a 5 to 25-room simulation, the kernel must potentially deal with 400 to 2000 threads.
Linux O(1) Scheduling

- Linux 2.5 moved to constant order O(1) scheduling
  - Preemptive, priority based
  - Two priority ranges: SCHED_OTHER, SCHED_FIFO, SCHED_RR
    - Real-time range from 0 to 99 and nice value from 100 to 140
    - Map into global priority with numerically lower values indicating higher priority
  - Higher priority gets larger quantum
  - Active tasks
    - Task who did not exhaust its time-slice
  - Expired tasks
    - Tasks who has no time-slice left
  - All run-able tasks tracked in per-CPU runqueue data structure
    - Two priority arrays (active, expired)
    - Tasks indexed by priority
    - When no more active, arrays are exchanged

- Worked well, but poor response times for interactive processes
Linux Scheduling in 2.6.23+

- **Completely Fair Scheduler (CFS)**
- **Scheduling classes**
  - Real-time classes: SCHED_FIFO, SCHED_RR
  - Default (fair-share) class
    - Tasks share CPU time proportionally
- **Quantum calculated based on nice value from -20 to +19**
  - Lower value is higher priority
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if number of active tasks increases
- **CFS scheduler maintains per task virtual runtime**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time $= \text{actual run time}$
- **Scheduler picks next task with lowest virtual runtime**
  - Task who has had the lowest CPU time
Linux Scheduling in 2.6.23+

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:

![Red-black tree diagram]

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 $O(lgN)$ 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.