Process Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Operating Systems Examples
Basic Concept

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

- Scheduling points
Histogram of CPU-burst Times
Alternating Sequence of CPU And I/O Bursts
Alternating Sequence of CPU And I/O Bursts

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

(a) Long CPU burst

(b) Short CPU burst

Waiting for I/O
CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

- 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.
Preemptive VS Non-Preemptive

- **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?
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)
Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time
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
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
First-Come, First-Served (FCFS) Scheduling

<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:

<table>
<thead>
<tr>
<th></th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>24</td>
<td>27</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td>30</td>
</tr>
</tbody>
</table>

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

- Suppose that the processes arrive in the order P2, P3, P1.
- The Gantt chart for the schedule is:

```
<table>
<thead>
<tr>
<th>P_2</th>
<th>P_3</th>
<th>P_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>
```

- 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) Scheduling

- 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
- SJF may starve long processes
Example of SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0.0</td>
<td>6</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2.0</td>
<td>8</td>
</tr>
<tr>
<td>$P_3$</td>
<td>4.0</td>
<td>7</td>
</tr>
<tr>
<td>$P_4$</td>
<td>5.0</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

- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging

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: \( \tau_{n+1} = \alpha \ t_n + (1 - \alpha) \tau_n \).
Prediction of the Length of the Next CPU Burst

<table>
<thead>
<tr>
<th>CPU burst ($t_i$)</th>
<th>6</th>
<th>4</th>
<th>6</th>
<th>4</th>
<th>13</th>
<th>13</th>
<th>13</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;guess&quot; ($\tau_i$)</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>
Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count

- $\alpha = 1$
  - $\tau_{n+1} = \alpha \cdot t_n$
  - Only the actual last CPU burst counts

- If we expand the formula, we get:
  
  $\tau_{n+1} = \alpha \cdot t_n + (1 - \alpha) \cdot \alpha \cdot t_{n-1} + \ldots$
  
  $+ (1 - \alpha)^j \cdot \alpha \cdot t_{n-j} + \ldots$
  
  $+ (1 - \alpha)^{n+1} \cdot \tau_0$

- Since both $\alpha$ and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor
**SRTF**

- **Shortest Remaining Time First**
  - 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>

![Diagram showing SJF and SRTF scheduling](chart.png)
Round Robin

- 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 with Time Quantum = 4

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>24</td>
</tr>
<tr>
<td>P₂</td>
<td>3</td>
</tr>
<tr>
<td>P₃</td>
<td>3</td>
</tr>
</tbody>
</table>

- The Gantt chart is:

<table>
<thead>
<tr>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₁</th>
<th>P₁</th>
<th>P₁</th>
<th>P₁</th>
<th>P₁</th>
</tr>
</thead>
<tbody>
<tr>
<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
Time Quantum and Context Switch Time

Process time = 10

<table>
<thead>
<tr>
<th>Quantum</th>
<th>Context switches</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>
Turnaround Time Varies With The Time Quantum

<table>
<thead>
<tr>
<th>process</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>6</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>7</td>
</tr>
</tbody>
</table>

![Graph showing turnaround time varying with time quantum](image)
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 a priority scheduling where priority is the 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?*

![Diagram of priority inversion](image)
Priority Inversion Solutions

- 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)
- 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:
A process can move between the various queues
- Aging can be implemented this way

Multilevel-feedback-queue scheduler defined by the following parameters
- Number of queues
- Scheduling algorithms for each queue
- Method used to determine when to upgrade a process
- Method used to determine when to demote a process
- Method used to determine which queue a process will enter when that process needs service
Example of Multilevel Feedback Queue

- Three queues
  - Q0 – RR with time quantum 8 milliseconds
  - Q1 – RR time quantum 16 milliseconds
  - Q2 – FCFS

- Scheduling
  1. A new job enters queue Q0 which is served FCFS
  2. When it gains CPU, job receives 8 milliseconds
  3. If it does not finish in 8 milliseconds, job is moved to queue Q1
  4. At Q1 job is again served FCFS and receives 16 additional milliseconds
  5. If it still does not complete, it is preempted and moved to queue Q2
Multilevel Feedback Queues

quantum = 8

quantum = 16

FCFS
Multiple-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
UNIX Scheduler

- 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
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?
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
Priorities in Linux 2.4 Scheduling

- **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 Task structure in Linux 2.4

- **long counter;**
  - time remaining in the task’s current quantum (represents dynamic priority)

- **long nice;**
  - task’s nice value, -20 to +19. (represents static priority)

- **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

- **unsigned long cpus_runnable;**
  - ~0 if the task is not running on any CPU
    
  - (1<<cpu) if it’s running on a CPU

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

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

- **unsigned long rt_priority;**
  - real-time priority
Scheduling Policies in Linux 2.4

- **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 in Linux 2.4

- 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)
```
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.
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 in Linux 2.4

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.
Scalability of Linux 2.4 Scheduler

- Scalability problems in Linux 2.4 scheduler
  - A single run queue
  - Recalculating goodness() for every task on every invocation of the scheduler

- Linux 2.6: O(1) scheduler (prior to 2.6.23)
  - The idea of “fair scheduling” by Con Kolivas

- CFS (Completely Fair Scheduler) by Ingo Molnar
  - Many ideas borrowed from the Kolivas’ scheduler
  - Officially included in Linux 2.6.23
  - Made Kolivas leave Linux kernel development
  - Kolivas returns with BFS (Brain Fuck Scheduler) in 2009