Scheduling

Milana Maric

September 24th, 2024

OS Scheduler

The **OS scheduler** decides which process runs next, when, and for how long.

- Optimizes for system resources: gives higher priority to certain processes and prevents starvation of any processes.
- The OS scheduler has two parts:
  - **Mechanism** to stop one running process and start another ⇒ *Context Switch.*
  - **Policy** to pick which process to run ⇒ *FIFO, Round-Robin, CFS, etc.*

Context Switch Mechanism

A **Context Switch** is a mechanism that allows the OS to store the current process state and switch to another, previously stored context.

- **Process Control Block (PCB)** ⇒ The context of the process.
  - OS maintains PCB for each process.
  - PCB includes x86 registers, process state (running, blocked, ready), process number, list of opened files, etc.

Context Switch Procedure

1. Saves the running process's execution state.
2. Selects the next thread.
3. Restores the execution state of the next process.
4. Passes control to resume the next process.

Note: A context switch introduces some overhead, typically ranging from 100 ns to 1 µs in modern operating systems. To minimize its impact and ensure efficiency, the time a process runs should be significantly longer than the context switch duration.

Scheduling Policy

Scheduling Policy is a set of rules determining what runs when.

IO-bound vs CPU-bound Processes

- **IO-bound Processes**:
  - Spend most of their time waiting for IO, shorter time spent on the CPU
  - Response time is important
  - Examples: Typing, copying files, etc.
- **CPU-bound Processes**:
  - Heavy use of the CPU
  - Caches stay hot when processes run for a long time
  - Examples: Scientific computation

Multitasking

Multitasking is the simultaneous execution of more than one process.It was first introduced with mainframes.

Single-core vs Multi-core Multitasking

- **Single-core**: The OS gives the illusion of multiple processes running concurrently
- **Multi-core**: The processor enables true parallelism

Cooperative vs Preemptive Multitasking

- **Cooperative Multitasking**:
  - Old OSes and some language runtimes
  - A process runs until it voluntarily yields the CPU
  - The OS cannot enforce fair scheduling
- **Preemptive Multitasking**:
  - Modern OSes
  - OS can interrupt the execution of a process after its **timeslice** expires
  - Preemptive scheduler relies on a hardware timer to interrupt processes
    - OS sets a timer before scheduling a process
    - Hardware generates an interrupt after timeslice expires and interrupts execution
    - Interrupt leads to switching to kernel mode OS decides if process can continue
  - In most cases timeslices depend on **process priority**

Process Priority

- OS ranks processes based on their importance and need for CPU time
- Processes with higher priority run before those with lower priority

Linux Process Priority

- **Nice Value**:
  - Ranges from -20 (highest priority) to +19 (lowest priority).
  - Default is 0.
- **Real-Time Priority**:
  - Ranges from 0 (lowest real-time priority) to 99 (highest real-time priority).
  - Real-time processes always execute before normal processes.
- Command to view real-time and nice values:

```
ps ax -eo pid,ni,rtprio,cmd
```

Timeslice

- Defines how long a process should run before being preempted.
- Timeslice is too long -> Bad for interactive processes
- Timeslice is too short -> High context switch overhead

Linux CFS

- Linux scheduler does not use an absolute timeslice
- The timeslice a process receives is a function of the system load and process priority

Completely Fair Scheduler (CFS)

Goal: Fairly divide the CPU evenly among all processes

- If we run n tasks in parallel on the CPU, each would get $\frac{1}{n}$ of the CPU processing power
- A process runs for some time, then swaps for the runnable process that has run the least

Virtual Runtime (vruntime)

- Virtual runtime specifies when a task's next timeslice should start execution
- As a process runs, it accumulates virtual runtime
- For high priority process, the vruntime accumulates slowly; for low priority, quickly
- The process with the smallest vruntime is picked next for execution

Targeted Latency

- Period during which all runnable processes should be scheduled at least once
- The floor is set at 1ms by default

Algorithm

- Each process has a rate at which its vruntime increases
- Divide the targeted latency into the number of quantums required for each process to be scheduled equally
- After each quantum, increase the vruntime of the process that just ran according to its rate and choose the new process with the lowest vruntime

Note: CFS maintains a red-black tree for the virtual runtimes of processes

Load Balancing

- Each CPU has its own queue and is self-scheduling
- Two main types of load balancing:
  - **Push Migration**: A process periodically checks the load on each CPU. If it detects an imbalance, it redistributes the processes
  - **Pull Migration**: An idle CPU can actively pull tasks from a busy processor
- We must be able to preempt a process from a CPU

How/When to Preempt

- If the CPU needs rescheduling, it sets a flag to trigger this.
- This happens on system calls such as:
  - **scheduler_tic()**: A process uses its timeslice.
  - **try_to_wake_up()**: A higher-priority process has woken up
- The kernel checks `need_resched()` at certain points and, if safe, calls `schedule()`

User Preemption

- Available in all OSes
- Triggered by system calls or interrupt handlers

Kernel Preemption

- Available in Linux, most UNIX-like OSes are non-preemptive (especially servers)
- A process can be preempted in the kernel by more latency-sensitive processes, if it is in a safe state without holding any locks

Linux Scheduling Policies

- Linux does not support hard real-time scheduling, but it does support soft real-time scheduling.
- **SCHED_FIFO**:
  - Fixed priority (ranges from 1 to 99)
  - A task runs until it blocks or yields
  - Only a higher-priority real-time (RT) task can preempt it
  - Round-robin for tasks of the same priority
- **SCHED_RR**:
  - Same as **SCHED_FIFO**, but with a fixed timeslice
  - Once the timeslice expires, the task can be preempted by another task with the same priority
- **SCHED_DEADLINE**:
  - For predictive real-time scheduling.
  - Uses earliest deadline first (EDF), based on activation period and worst-case execution time (WCET)
- Linux also has scheduling policies for non-real-time tasks:
  - **CFS**: Completely Fair Scheduler
  - **SCHED_BATCH**: Non-real-time, low-priority background jobs
  - **SCHED_IDLE**: Non-real-time, very low-priority background jobs

Note: Real-time processes always run before non-real-time processes.

CFS Load Balancing

- One runqueue per core
- Load balancing algorith runs:
  - Periodically
  - Upon idle core
  - Upon the creation or awakening of a new thread
- Load balancing is an expensive procedure:
  - **Computation-wise**: Requires iterating over runqueues
  - **Communication-wise**: Requires modifying cached data structures, which can cause expensive cache misses and synchronization issues

Load Metric

- Each runqueue should have the same number of threads, but not all threads have the same priority
- The sum of weights on each runqueue should be balanced, but what if a high-priority task runs alone on a core and often goes idle?
- **Load** = weight × average CPU utilization.
- If one process has significantly more threads than another, it might get more CPU time, which is not fair. Threads of the same process belonging to the same **cgroup** and their load is further divided by the number of threads in the group.

Hierarchical Approach to Balancing Tasks

- The cores are organized in a hierarchy, at the bottom of which is a single core
- Cores are grouped based on how they share physical resources
- Each level of the hierarchy is called a **scheduling domain**
- A subset of nodes within a scheduling domain, within which the load is balanced, is called **scheduling group**
- Load balancing starts from each scheduling domain and runs bottom-up:
  - At each domain, one node is responsible for balancing the nodes
  - Average load is computed for each scheduling group
  - The busiest group is picked, and CPU load is balanced within the group

Optimization

- Avoid waking up idle nodes on every clock tick and running load-balancing. Idle cores can enter **tickless idle states** to reduce power. If there is an overloaded core, wake up a tickless core and assign it a balancing role.
- When a thread is awakened by another thread, the scheduler favors cores that share a cache with the waking thread to improve cache reuse.

Work-Conserving Scheduler

- A work-conserving scheduler: No idle cores if some cores have several threads in their runqueues
- The system aims to have no idle cores, but CFS is not work-conserving in some cases
- Another example of a non-work-conserving scheduler is one that allows binding processes to specific CPUs

Bugs in Linux Scheduler

- The paper *"The Linux Scheduler: A Decade of Wasted Cores"* describes four bugs discovered in the Linux scheduler and proposes solutions to them
- These bugs are hard to detect with conventional testing or performance tools because they occur in specific situations and do not cause system crashes, but silently degrade performance over time
- The paper also presents two tools: **Sanity Checker** and **Scheduler Visualization Tool** to enable easier identification of scheduler bugs

The Group Imbalance Bug

- Sometimes the load inside a scheduling group cannot be balanced (e.g., when there is only one task with a high weight, and the rest of the cores are idle) and other scheduling groups have idle cores
- Issue: The load between scheduling groups and inside them are balanced but there are idle cores in one scheduling group while other group might have sore with more than one task in a runqueue
- Fix: Instead of comparing average loads, compare the minimum loads across groups

The Scheduling Group Construction Bug

- Grouping of cores may result in non-disjoint scheduling groups
- Issue: Same nodes may appear in different groups, leading to inefficient load balancing
- Fix: Instead of computing scheduling groups at system boot, compute them dynamically for each core from its perspective

The Overloaded Wake-up Bug

- To improve cache locality, a waking thread is assigned to the local CPU
- Issue: This CPU may already be overloaded, increasing the system load further
- Fix: Wake up on local core if it is idle. If there are other idle cores in the system wake up on the one that was idle for the longest time. If no cores are idle use the original algorithm

The Missing Scheduling Domain Bug

- In complex machines with multiple NUMA nodes or sockets, when a scheduler creates a scheduling domain (e.g., after a node joins or leaves), it may exclude some cores
- Issue: Excluding cores from load balancing leads to improper scheduling and performance degradation
- Fix: Check that all cores are included in the scheduling hierarchy during load balancing