Operating Systems

January 31, 2025

Contents

1 Chapter 2

- We have abstractions that allow a single program to behave as the only process on the machine

- Mechanisms are how is the code implemented
- Policies are how we decide when to use the mechanisms
- Primary way we expose resources is through virtualization
  - CPU, RAM, disk
  - Make virtual forms for the program to use

1.1 Evolution

- Early OS was just a lib, only one process at a time
- Then we gain protections, raw data storage is dangerous
- Then we gained multiprogramming. Running more than one job at a time
- As we talk about various OS pieces, we will show the evolution into the modern versions .

2 Chapter 8

2.1 MLFQ: Multilevel Feedback Queue

- Multiple queues
- each is assigned a priority level
- When we make a decision we go to highest priority
  - For jobs within the same queue we use round-robin

2.1.1 Rules

1. If priority(A) > priority(B) => A
2. If priority(A) == priority(B) => Round-robin
3. When a job enters it is at highest priority
4. If it uses its entire timeslice its priority is reduced, no matter how many times it gives up the CPU
5. If it gives up the CPU before its timeslice is up it stays

6. After some time point S, move all jobs in the system to the topmost queue

2.2 How to select Priority?

- Job that often yields (getting keyboard input) should be prioritized for quicker response time
- If a job keeps making syscalls, we should priortize it as it is probably an interactive process
- If it uses the cpu a lot we should prioritize turn-around time

But job priority cannot be entirely static, it must be able to change over time.

2.3 How to change Priority?

- Give each job an allotment, this is the amount of time it can exist in a queue without reducing its priority
- When a job begins it is placed at the highest priority
- If it uses its entire timeslice its priority is reduced
- If a job gives up the CPU before running through its allotment it can stay at the same priority

Highly interactive jobs will always stay at the highest priority because they never use up an entire timeslice without making a syscall

A program can also game the scheduler by always issuing IO right before the time slice is up

A program can also change its behavior from CPU intensive to IO intensive

2.4 Priority Boost

Process could be getting starved, but if we boost everything, they will all then get a fair chance to run again

2.5 Better Accounting

- To prevent gaming, change rule 4
- If you use up your 10ms timeslice (even across schedules) then you move down

3 Chapter 9

Proportional Share Scheduling

3.1 A New Metric

- Tickets represent the share of a resource that an entity should recieve
- The percent of tickets it hold represent the % they get

3.1.1 Lottery

- Draw a random ticket, schedule the process that holds that ticket
- More tickets you have the more likely you are to be scheduled

3.1.2 Ticket Currency

- Allows a user to create their own currency
  - Global currency and then whatever currency each user creates
- Example both A and B are given 100 global tickets
  - A runs A1 and A2
    - A1/A2 both get 500 A bucks
  - B runs B1
    - B1 gets 10 B bucks
  - A1 and A2 are getting 50 global tickets each
  - B1 gets 100 global tickets

3.1.3 Ticket Transfer

- allows a process to lend tickets to another process
- Client/server running on same machine
  - Client sends request, wait on server
  - So it also send some of its tickets over to the server as well
  - The server will send them back when done

3.1.4 Ticket Inflation

- Can just grow its own ticket amount, only really makes sense in a co-op environment

3.1.5 Impl

- Good RNG
- List to hold processes
- Total # of tickets
- Generate number N
- Traverse list, add up ticket values
- Winner once the total is greater than N
- Linear time

3.1.6 Fairness

- 2 jobs of the same length how fair is the scheduler?
- Randomness affects short jobs
- Fairness = $\text{job}_{\text{finish1}}$ / $\text{job}_{\text{finish2}}$
- Want fairness to be 1

3.2 Stride Scheduling

- Randomness isn't fair
- Deterministic fair scheduler
- Stride = (some large number) / tickets
  - Inverse in proportion to the number of tickets it has
- Each process has a running pass value starting at 0
- When a process is scheduled, increment its pass by stride
- Always schedule the process with the lowest pass breaking ties arbitrarily
- Why do we even do this?
  - We now have global state, lottery doesnt
  - What happens if we add a new process? What should its pass be? Do we start at 0, or start at average

3.3 Linux Completely Fair Scheduler

- Highly efficient and scalable fair-share scheduler
- Aims to spend very little time making choices
- Important to not waste resources
  - Google used 5% of CPU time scheduling
- Reducing overhead is a key goal in modern schedulers
- Goal is to divide the CPU evenly among all competing processes
- It does so with a virtual runtime (vruntime)

3.3.1 CFS: Basic Operation

- Each process runs an accumulates vruntime
- Always picks the process with lowest vruntime
- CFS varies with $\text{sched}_{\text{latency}}$
  - Represents the largest time slice size possible
  - Determined by $\text{sched}_{\text{latency}}$/number of processes

3.3.2 Niceness

- Adds weighting to the time slice calculation
- time slice = Poriton of all processes running * max time slice
- vruntime = previous vruntime + time just ran * weighting based on niceness

$$\frac{1024}{2048} * \text{sched}_l atency\text{sched}_l atency_{\text{sched}_l atency_{\text{sched}_l atency}}$$