Recall: Real-Time Scheduling

* Goal: Predictability of Performance!
  + We need to predict with confidence worst case response time systems!
  + In RTS, performance guarantees are:
    - Task- and/or class centric and often ensured a priori
  + In conventional systems, performance is:
    - System/throughput with prost-processing
  + Real-time is about enforcing predictability and does not equal fast computing
* Hard real-time: for time-critical safety-oriented systems
  + Meet all deadlines (if at all possible)
  + Ideally: determine in advance if this is possible
  + EDF, LFF, etc
* Soft real-time: for multimedia
  + Attempt to meet deadlines with high probability
  + Constant Bandwidth Server (CBS)

Stride Scheduling

* Achieve proportional share of scheduling with resorting to randomness (hence overcoming law of small numbers problem)
* “Stride” of each job is big#W/Ni
  + The larger your share of tickets, the smaller your stride
  + Ex: W=10000, A=100, B=50,C=250
  + A stride: 100, B:200, C:40
* Each job has a “pass” counter
* Scheduler: pick job with lowest pass, runs it, add its stride to its pass
  + Job with twice the tickets gets to run twice as often

Linux CFS Scheduler

* Goal: Each process gets an equal share of CPU
  + N threads “simultaneously” execute on 1/N of CPU
  + The model is somewhat like simultaneous multithreading – each thread gets 1/N of the cycles
  + In general can’t do this with real hardware
    - OS needs to give out the full CPU in time slices
    - Thus, we must use something to keep the threads roughly in sync with one another
* Basic Idea: track CPU time per thread and schedule threads to match up average rate of execution
  + Scheduling decision:
    - “Repair” illusion of complete fairness
    - Choose thread with minimum CPU time
    - Closely related to Fair Queueing
  + Use a heap-like scheduling queue for this…
    - O(logN) to add/remove threads, where N is the number of threads
  + Sleeping threads don’t advance their CPU time, so they get a boost when they wake up again
    - Get interactivity automatically
  + In addition to fairness, we want low response time and starvation freedom
    - Make sure that everyone gets to run at least a little bit
  + Constraint 1: Target Latency
    - Period of time over which process gets service
    - Quanta = Target\_Latency / n
    - Ex: Target Latency – 20ms, 4 Processes
      * Each process gets 5ms time slice
    - Ex: Target Latency- 20ms, 200 Processes
      * Each process gets 0.1ms time slice 🡪 Large context switching overhead for each time slice
  + Constraint 2: Minimum Granularity
    - Minimum length of any time slice
    - Ex: Target Latency – 20ms, Minimum Granularity 1 ms, 200 processes
      * Each process gets 1 ms time slice

Aside: Priority in Unix – Being Nice

* The industrial operating systems of 60s and 70s provided priority to enforce desired usage policies
* “nice” values range from -20 to 19
  + Negative values are “not nice”
  + If you wanted to let your friends to get more time, you would nice up your job
* Scheduler puts higher nice-value tasks (lower priority) to sleep more…
  + In O(1) scheduler, this translated fairly directly to priority
* How does this idea translate to CFS?
  + Change the rate of CPU cycles given to threads to change relative priority (higher priority has a higher rate of execution)

Linux CFS: Proportional Shares

* What if we want to give more CPU to some and less to others in CFS (proportional share)?
  + Allow different threads to have different rates of execution
* Use weights! Key Idea: Assign a weight wi to each process i to compute the switching quanta Qi
  + Basic Equal Share: Qi = Target Latency \* 1/N
  + Weighted Share: Qi =
  + Reuse nice value to reflect share, rather than priority
    - Remember that nice value 🡪 higher priority
    - CFS uses nice values to scale weights exponentially: Weight = 1024/(1.25)nice
      * Two CPU tasks separated by a nice value of. 5 🡺 Task with lower nice value has 3 times the weight since 1.255 ~ 3
  + Ex: Target Latency: 20ms, Minimum Granularity: 1ms
    - Two CPU-Bound Threads 🡺 Thread A, Weight 1 Thread B, Weight 4
    - Time Slice for A? 4ms
    - Time Slice for B? 16ms
  + Track a thread’s virtual runtime rather than its true physical runtime
    - Higher weight: Virtual runtime increases more slowly
    - Lower weight: Virtual runtime increases more quickly
  + Scheduler’s decision based on virtual CPU time
  + Use of Red-Black tree to hold all runnable processes as sorted on vruntime variable
    - O(1) time to find next thread to run (top of heap!)
    - O(logN) time to perform insertion/deletions
      * Cache the item at far left (item with earliest vruntime)
    - When ready to schedule, grab version with smallest vruntime (which will be item at far left)
  + Diagram, text

    Description automatically generatedSlower than O(1) scheduler, but much easier to understand (simpler heuristics)

When do the details of the scheduling policy and fairness really matter?

* When there aren’t enough resources to go around

When should you simply buy a faster computer?

* (Or network link, or expanded highway, or…)
* One approach Buy it when it will pay for itself in improved response time
* Diagram, shape

  Description automatically generatedNever want to run anything at 100% 🡺 Queue can get filled up

An interesting implication of this curve:

* Most scheduling algorithms work fine in the “linear” portion of the load curve, fail otherwise
* Argues for buying a faster X when hit “knee” of curve

Deadlock: A Deadly type of Starvation

* Starvation: thread waits indefinitely
  + Example: low priority thread waiting for resources constantly in use by high-priority threads
* Deadlock: circular waiting for resources
  + Thread A owns Resource 1 and is Waiting for Resource 2
  + Thread B owns Resource 2 and is Waiting for Resource 1
* Deadlock 🡪 Starvation but not vice versa
  + Starvation can end (but doesn’t have to)
  + Deadlock can’t end without external intervention

Graphical user interface, diagram

Description automatically generated

Text, letter

Description automatically generated with medium confidenceDeadlock with locks (non deterministic)

Text

Description automatically generated

Diagram

Description automatically generated

Unlucky (Left) and Unlucky (Right) case

Sometimes, scheduler won’t trigger deadlock!

Train Example (Wormhole-Routed Network)

* Circular dependency (Deadlock!)
  + Each train wants to turn right, but is blocked by other trains
* Similar problem to multiprocessor networks
  + Wormhole-Routed Network: Message trail through network like a “worm”
* Diagram

  Description automatically generatedFix? Imagine grid extends in all four directions
  + Force ordering of channels (tracks)
    - Protocol: Always go east-west first, then north-south
  + Called “dimension ordering” (X then Y)

Threads often block waiting for resources

* Locks
* Terminals
* Printers
* CD drives
* Memory

Threads often block waiting for

* Pipes
* Sockets

Text

Description automatically generated with medium confidence  
You can deadlock on any of these!

Five chopsticks/Five lawyers (really cheap restaurant)

* Free-for all: Lawyers will grab any one they can
* Need two chopsticks to eat

What if all grab at same time?

* Deadlock!

How to fix deadlock

* Make one of them give up a chopstick (Hah!)
* Eventually everyone will get chance to eat

How to prevent deadlock?

* Never let lawyer take last chopstick if no hungry lawyer has two chopsticks afterwards
* Can we formalize this requirement somehow?

Four requirements for deadlock

* Mutual Exclusion
  + Only one thread at a time can use a resource
* Holding and waiting for resource
  + Thread holding at least one resource is waiting to acquire additional resources held by other threads
* No preemption
  + Resources are released only voluntarily by the thread holding the resource, after thread is finished with it (can’t take away resource from somebody)
* Circular wait
  + There exists a set of{T1, T2, T3… Tn}
  + T1 is waiting for T2
  + T2 is waiting for T3
  + …
  + Tn is waiting for T1

Shape, square

Description automatically generated

Detecting Deadlock: Resource-Allocation Graph

* System Model
  + A set of threads T1, T2… Tn
  + Resource types R1, R2, … Rm
  + Each resource Ri has Wi instances
  + Each thread utilizes a resource as follows:
    - Request()
    - Use()
    - Release()
* Resource-Allocation Graph:
  + V is partitioned into two types
    - T = {T1, T2… Tn}, the set of threads in the system
    - R = {R1, R2… Rm}, the set of resource types in system
  + Request edge – directed edge T1 🡪 Rj
  + Assignment edge – directed edge Rj 🡪 Ti

Diagram, shape

Description automatically generated

A cycle is necessary for a deadlock, but not sufficient

Deadlock Detection Algorithm

* Let [X] represent an m-ary vector of non-negative integets
* [FreeResources]: Current free resources of each type
* [RequestX]: Current requests from thread X
* [AllocX]: Current resources held by thread X