Diagram

Description automatically generatedRecall: Internal OS File Description

* Internal Data Structure describing everything about the file
  + Where is resides
  + Its status
  + How to access it

Recall: Scheduling

* How is the OS to decide which of several tasks to take off a queue?
* Scheduling: deciding which threads are given access to resources from moment to moment
  + Often we think in terms of CPU time, but could also think about access to resources such as Network BW, disk access
* Triggered by, for example:
  + Timer interrupts
  + System calls that go to sleep (eg, read to disk)

Recall: Scheduling Policy

* Minimize Response Time
  + Minimize elapsed time to do an operations (or job)
  + Response time is what user sees
    - Time to echo a keystroke
    - Time to compile a program
    - Real-time Tasks: Must meet deadlines imposed by World
* Maximize Throughput
  + Maximize operations (or jobs) per second
  + Throughput related to response time, but not identical
    - Minimize response time will lead to more context switching than if you only maximized throughput
  + Two parts to maximize throughput
    - Minimize overhead (for example, context switching)
    - Efficient use of resources
* Fairness
  + Share CPU among users in some equitable way
  + Fairness is not minimizing average response time
    - Better average response time makes the system less fair (prioritizes smaller tasks/shorter burst time)
* Examples:
  + Round Robin
    - Better for short jobs (+)
    - Context switching time adds up for long jobs(-)

Recall: What if we knew the future?

* Could we always mirror best FCFS?
* Shortest Job First (SJF)
  + Run whatever job has least amount of computation to do
  + Sometimes called “Shortest Time to Completion First” (STCF)
* Shortest Remaining Time First (SRTF)
  + Preemptive version of SJF: if job arrives and has shorter completion time than the remaining time on the current job, immediately preempt CPU
  + Sometimes called “Shortest Remaining Time to Completion First” (SRTCF)
* These can be to whole program or current CPU burst
  + Idea is to get short jobs out of system
  + Big effect on short jobs only, small effect on long ones
  + Result is better avg response time
* Pros and Cons:
  + Optimal (avg response time) +
  + Hard to predict future –
  + Unfair –

How to Handle Simultaneous Mix of Diff Types of Apps?

* Consider mix of interactive and high throughput apps:
  + How to best schedule them?
  + How to recognize one from the other?
    - Do you trust app to say that it is “interactive”?
  + Should you schedule the set of apps identically on servers, workstations, tablet, and cellphones (is every platform the same)?
* For instant is Burst Time(observed) useful to decide which application gets CPU time?
  + Short bursts 🡺 Interactivity 🡺 High Priority
* Assumptions encoded into many schedulers
  + Apps that sleep a lot and have short bursts must be interactive apps – they should get high priority
  + Apps that compute a lot should get lower priority, since they wont notice intermittent bursts from interactive apps
  + Hard to characterize apps:
    - What about apps that sleep a long time but then compute a long time
    - Or, what about apps that must run under all circumstances

Graphical user interface

Description automatically generated with medium confidenceMulti-Level Feedback Scheduling

* Another method of exploiting past behavior
* Multiple queues, each with a different priority
  + Higher priority often considered “foreground” tasks
* Each queue has its own scheduling algorithm
  + Foreground – RR, Background – FCFS
* Adjust each job’s priority as follows:
  + Job starts in highest priority
  + If timeout expires, drop one level
  + If timeout doesn’t expire, push up one level
  + Long-Running compute tasks demoted to low priority
* Result approximates SRTF
  + CPU bound jobs drop like a rock
  + Short-running I/O bound jobs stay near top
* Scheduling must be done between the queues
  + Fixed priority scheduling:
    - Serve all from highest priority, then next priority
      * Starvation easily happens
    - Time slice:
      * Each queue gets a certain amount of CPU time
      * Eg, 70% to highest, 20% next, 10% lowest
  + Counter measure: user action can foil intent of the OS designer
    - For multilevel feedback, put in a bunch of meaningless I/O just to keep the job’s priority high

Case Study: Linux O(1) Scheduler

* Priority based scheduler: 140 priorities
  + 40 for “user tasks”, 100 for “Realtime/Kernel”
  + Lower priority value 🡺 higher priority
  + Highest priority value 🡺 lower priority
  + All algorithms O(1)
    - Time slices/priorities/interactivity credits all computed when job finishes time slice
    - 140-bit bit mask indicates presence or absence of job at given priority level
  + Two separate priority-queues: “active” and “expired”
    - All tasks in the active queue use up their timeslices and get placed on the expired queue, after which queues swapped
  + Timeslice depends on priority – linearly mapped onto timeslice range
    - Like a multi-level queue (one queue per priority) with different timeslice at each level
    - Execution split into “Time slice Granularity” chunks – round robin through priority
  + Lots of ad-hoc heuristics
    - Try to boost priority of I/O-bound tasks
    - Try to boost priority of starved tasks
  + Heuristics
    - User-task priority adjusted +-5 based on heuristics
      * p->sleep\_avg = sleep\_time – run\_time
      * Higher sleep\_avg 🡺 more I/O bound the task, more reward
    - Interactive credit
      * Earned when a task sleeps for a “long” time
      * Spent when a task runs for a “long” time
      * IC is used to provide hysteresis to avoid changing interactivity for temporary changes in behavior
    - However, “interactive tasks” get special dispensation
      * To try to maintain interactivity
      * Placed back into active queue, unless some other task has been starved for too long…
    - Real-Time Tasks
      * Always preempt non-RT tasks
      * No dynamic adjustment of priorities
      * Scheduling schemes:
        + SCHED\_FIFO: preempt other tasks, no timeslice limit
        + SCHED\_RR: preempt normal tasks, RR scheduling amongst tasks of same priority

So Does the OS Schedule Processes for Threads?

* Many textbooks use the “old model” – one thread per process
* Want single protection domain with lots of concurrency 🡺 many threads per proccess
* Usually, its really: threads (e.g. in Linux)
* One point to notice: switching threads vs switching process incurs different cost:
  + Switch threads: Save/restore registers
  + Switch processes: Chance active address space too!
    - Expensive
    - Disrupts caching

Multi-Core Scheduling

* Algorithmically, note a huge different from a single-core scheduling
* Implementation-wise, helpful to have per-core scheduling data structures
  + Cache coherence
* Affinity scheduling: once a thread is scheduled on a CPU, OS tries to reschedule it on the same CPU
  + Cache reuse

Recall: Spinlocks for multiprocessing:

* Spinlock implementation:
  + int value = 0; // Free

Acquire() {  
 while (test&set(&value)) {}; // spin while busy  
}

Release() {  
 value = 0; // atomic store  
}

* Spinlock doesn’t put the calling thread to sleep – it just busy waits
  + When might this be preferable?
    - Waiting for limited number of threads at a barrier in multiprocessing (multicore) program (quick wakeup)
    - Wait time at barrier would be greatly increased if threads must be woken inside kernel (would require loading in state and going into kernel if put to sleep)
  + Every test&set() is a write, which makes value ping-pong around between core-local caches 🡺 want to use test&test&set()

Gang Scheduling and Parallel Applications

* When multiple threads work together on a multi-core system, try to schedule them together
  + Makes spin-waiting more efficient (inefficient to spin-wait for a thread that’s suspended)
* Alternative: OS informs a parallel program how many processors its threads are scheduled on
  + Application adapts to number of cores that is has scheduled
  + “Space sharing” with other parallel programs can be more efficient, because parallel speedup is often sublinear with the number of cores

Real-Time Scheduling

* Goal: Predictability of Performance!
  + We need to predict worse case response time for systems!
  + In RTS, performance guarantees are:
    - Task- and/or class centric and often ensured a priori
  + In conventional system, performance is:
    - System/throughput oriented with post-processing(… wait and see …)
  + Real-time is about enforcing predictability, and does not equal fast computing!!
  + Hard real-time: for time-critical safety-oriented systems
    - Meat all deadlines (if at all possible)
    - Ideally: determine in advance if this is possible
    - Earliest Deadline First (EDF), Least Laxity First (LLF), etc
  + Soft real-time: for multimedia
    - Attempt to meet deadlines with high probability
    - Constant Bandwith Server (CBS)

Example: Workload Characteristics

* Tasks are preemptable, independent with arbitrary arrival (=release) times
* Tasks have deadlines (D) and known computation times (C)

Chart

Description automatically generated with medium confidence

Diagram

Description automatically generatedExample: Round-Robin scheduling doesn’t work (same schedule)

Round robin has no deadlines, it’s about multiplexing and doesn’t account for deadlines

Earliest Deadline First (EDF)

* Tasks periodic with period P and computation C in each period, (Pi, Ci) for each task i
* Preemptive priority-based dynamic scheduling:
  + Each task is assigned a (current) priority based on how close the absolute deadline is (Dt+1 = Dit + Pi for each task)
  + The scheduler always schedules the active task with the closest absolute deadline
* Chart, box and whisker chart

  Description automatically generatedAs long as system is not overloaded, EDF will meet deadlines
* For n tasks with computation time C and deadline D and period P,

Ensuring Progress

* Starvation: thread fails to make progress for an indefinite period of time
* Starvation != Deadlock because starvation could resolve under right circumstances
* Causes of starvation:
  + Scheduling policy never runs a particular thread on CPU
  + Thread waits for each other or are spinning in way that will never be resolved

Strawman: Non-Work-Conserving Scheduler

* A work conserving scheduler is on that does not leave the CPU idle when there is work to do
* A non-work-conserving scheduler could trivially lead to starvation
* In this class, we’ll assume that the scheduler is work-conserving (unless stated otherwise)

Strawman: Last-Come, First-Served Scheduler (LCFS)

* Stack (LIFO) as a scheduling data structure
  + Late arrivals get fast service
  + Early ones wait – extremely unfair
  + In the worst case – starvation
* When would this occur?
  + When arrival rate (offered load) exceeds service rate (delivered load)
  + Queue builds up faster than it drains
  + Queue can build FIFO too, but “serviced in the other received”

Is FCFS Prone to Starvation?

* If a task never yields, then other tasks don’t get to run
* Problem with all non-preemptive schedulers
  + And early personal OSes such as original MacOS, Windows 3.1, etc

Is Round Robin prone to starvation?

* Each of N processes gets 1/N of CPU
  + With quantum length Q, processes waits at most (N-1)\*Q ms to run again
  + So a process can’t be kept waiting indefinitely
* So RR is fair in terms of waiting time
  + Not necessarily in terms of throughput

Is Priority Scheduling Prone to Starvation?

* Recall: Priority Scheduler always runs the thread with highest priority.
  + Low priority thread might never run! 🡺 Starvation
* But there are more serious problems as well…
  + Priority inversion: even high priority threads might become starved by low priority threads by incorrect circumstances

Chart, diagram

Description automatically generatedChart

Description automatically generatedChart, waterfall chart

Description automatically generatedPriority Inversion:

* Job 1 stalls Job 3
* If Job 2 begins to run, it stalls Job 1 due to priority, which stalls Job 3 due to acquiring lock (medium priority task can starve a high priority one)
* Need to get Job 1 to run and ignore Job 2 so it can release lock for Job 3
* Solution: Priority Donation

Priority Inversion

* Where high priority task is blocked waiting on low priority task
* Low priority one must run for high priority to make progress
* Medium priority task can starve a high priority one

When else might priority lead to starvation or “live lock”?

Graphical user interface, application

Description automatically generated

Solution: Priority Donation

* Job 3 temporarily grants Job 1 its “high priority” to run on its behalf

Diagram

Description automatically generatedChart, waterfall chart

Description automatically generated

Mars Rover:

* Priority inversion caused low priority task grabs mutex trying to communicare with high priority task
* Solution: Turn priority donation back on and upload fix!
* (Original developers turned off priority donation 🡺 worried about performance costs of donating priority)

Are SRTF and MLFQ Prone to Starvation:

* In SRTF, long jobs are starved in favor of shorter ones 🡺 MLFQ is approximation of SRTF so it suffers from same problem

Cause for Starvation: Priorities?

* The policies we’ve studied so far
  + Always prefer to give the CPU to a prioritized job
  + Non-prioritized jobs may never get to run
* But priorities were a means, not an end
* Our end goal was to serve a mix of CPU-bound, I/O bound, and Interactive jobs effectively on common hardware
  + Give the I/O bound ones enough CPU to issue their next file operation and wait (on those slow discs)
  + Give the interactive ones. Enough CPU to respond to an input and wait (on those slow humans)
  + Let the CPU bound ones grind away without too much disturbance

Changing Landscape of Scheduling

* Priority-based scheduling rooted in “time-sharing”
  + Allocating precarious, limited resources across a diverse workload
    - CPU bound vs interactive vs I/O bound
* 80’s brought personal computers, workstations, and servers on networks
  + Different machines of different types for different purposes
  + Shift to fairness and avoiding extremes (starvation)
* 90’s emergence of the web, rise of internet-based services, the data-center-is-the-computer
  + Server consolidation, massive clustered services, huge flash crowds
  + It’s about predictability, 95th percentile performance guarantees

Proportional Share Scheduling

* Policies we’ve studied so far
  + Always prefer to give the CPU to a prioritized job
  + Non-prioritized jobs may never get to run
* Instead we can share the CPU proportional
  + Give each job a share of the CPU according to its priority
  + Low-priority jobs get to run less often
  + But all jobs can at least make progress (no starvation)
* Recall: Lottery Scheduling
  + Give a set of jobs, proved each with a share of a resource
  + Idea: Give out tickets according to proportion each should receive
  + Every quantum tick, draw one at random, schedule that job to run
  + Unfairness: Given two jobs A and B of same run time that are each supposed to receive 50%
    - Takes long time for two equal jobs to finally receive equal CPU time
* Stride Scheduling (Proportional scheduling w/o randomness)
  + Stride of each job 🡺 big#W/Ni
    - W = 10000, A = 100 tickets, B = 50, C = 250
    - A stride: 100, B: 200, C: 40
  + The larger your share of tickets, the smaller your stride
  + Each job has a “pass” counter
  + Scheduler: pick job with lowest pass, runs it, add its stride to its pass
  + Low-stride jobs (lots of tickets) run more often
    - Job with twice the tickets get to run more often

Linux Completely Fair Scheduler (CFS)

* Goal: Each process gets an equal share of CPU
* N threads “simultaneously” execute on 1/Nth of CPU
* Can’t do this with real hardware
  + OS needs to give out full CPU in time slices
* Instead: track CPU time given to a thread so far
* Scheduling Decision:
  + “Repair” illusion of complete fairness
  + Choose thread with minimum CPI time
  + Use a heap-like scheduling queue for this…
* In addition to fairness, we want low response time
* Constraint 1: Target Latency
  + Period of time over which every process gets service
  + Quanta = Target\_Latency / n
* Target Latency: 20ms, 4 Processes
  + Each process gets 5ms time slice
* Target Latency: 20ms, 200 processes
  + Each process gets 0.1ms time slice
  + Recall Round-Robin: large context-switching overhead of slice gets too small
* Goal: Throughput
  + Avoid excessive overhead
* Constraint 2: Minimum Granularity
  + Minimum length of any time slice
* Target Latency 20ms, Minimum Granularity 1ms
  + Each process gets 1ms time slice

Table

Description automatically generated

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

* When there aren’t enough resources to go around

Summary:

* Scheduling Goals:
  + Minimize response time (for human interaction)
  + Maximize throughput (for large computations)
  + Fairness (Proper sharing of resources)
  + Predictability (Hard/Soft Realtime)
* Round-Robin scheduling
  + Give each thread a small amount of CPU time when it executes; cycle between threads
  + Pros: Better for short jobs
* Shortest Job First (SJF)/Shortest Remaining Time First (SRTF)
  + Run whatever job has the least amount of computation to do/least remaining amount of computation left
* Multi-Level Feedback Scheduling
  + Multiple queues of different priorities and scheduling algorithms
  + Automatic promotion/demotion of process priority in order to approximate SJF/SRTF
* Realtime Schedulers such as EDF
  + Guaranteed behavior by meeting deadlines
  + Realtime tasks defined by tuple of compute time and period
  + Schedulability test: is it possible to meet deadlines with proposed set of processe?
* Lottery Scheduling
  + Give each a priority-dependent number of tokens (short task 🡺 more tokens)
* Linux CFS Scheduler: Fair fraction of CPU
  + Approximates an “ideal” multitasking processor
  + Practical example of “fair queueing”