Operating Systems (234123)

Scheduling

BATCH (NON-PREEMPTIVE) SCHEDULERS
Consider a different context...

• In this course, we typically consider general purpose OS
  – Such as the one running in our own computers

• For some of the topics in this lecture, need to change the mind set
  – (Although much of what we’ll learn nevertheless applies to general purpose OSes)

• New context:
  – “Batch scheduling” on “supercomputers”
  – We will explain both of these terms next
Supercomputers

• Comprised of hundreds (199x) to millions (201x) of cores
  – Many multicores connected with a high-speed network
• Used by 100s to 1000s of scientific users
  – Physicists, chemists...
• Users submit “jobs” (=programs)
  – Jobs can be serial (one core) or parallel (many cores)
  – A parallel job simultaneously uses N cores to solve a single scientific problem quickly (ideally N times faster)
  – N is the “size” of the job (determined by the submitting user)
  – While the program is running, cores communicate with each other and exchange information
  – Typical workload: jobs running from a few seconds to 10s of hours
  – Different jobs can have different sizes (often a power of 2)
    • For a given running job, size is fixed throughout its lifetime
Think of jobs as rectangles in core X time plane

- **Terminology:**
  - Size: jobs are said to be “wide” = ”big” or “narrow” = ”small”
  - Runtime: jobs are said to be “short” or “long”
This is how a schedule might look like

(Numbers indicate arrival order)
Batch scheduling

• It’s important that all the processes of a job run together (as they repeatedly & frequently communicate)
  – Each process runs on a different core
  – What will happen if they don’t?

• Jobs are often tailored to use the entire physical memory on each individual multicore
  – Can’t really share cores with other jobs via multiprogramming

• Thus, supercomputers typically use “batch scheduling”
  – When a job is scheduled to run, it gets its own cores
  – Cores are dedicated (not shared)
  – Each job runs to completion (until it terminates)
  – Only after, the cores are allocated to other waiting jobs
  – “Non-preemptive” (see later on)
METRICS TO EVALUATE PERFORMANCE OF BATCH SCHEDULERS

Wait time, response time, slowdown, utilization, throughput
Average wait time & response time

• **Average wait time**
  - The “wait time” of a job is the interval between the time the job is submitted to the time the job starts to run
  - \( \text{waitTime} = \text{startTime} - \text{submitTime} \)
  - The shorter the average wait time, the better the performance

• **Average response time**
  - The “response time” of a job is the interval between the time the job is submitted to the time the job terminated
  - \( \text{responseTime} = \text{terminationTime} - \text{submitTime} \)
  - \( \text{responseTime} = \text{waitTime} + \text{runTime} \)
  - The shorter the average response time, the better the performance

• **Wait vs. response**
  - Users typically care more about response time (wait for their job to end)
  - But batch schedulers can typically only affect wait time
Connection between average wait time & average response time

• Claim
  – In our discussion, job runtimes (and thus their average) are a given; they stay the same regardless of the scheduler
  – Thus, for batch schedulers, the difference between average wait time and average response time of a given schedule is a constant
  – The constant is the average runtime of all jobs

• Proof
  – For each job $i$ ($i = 1, 2, 3, \ldots N$)
    • Let $W_i$ be the wait time of job $i$
    • Let $R_i$ be the runtime of job $i$
    • Let $T_i$ be the response time of job $i$ ($T_i = W_i + R_i$)
  – With this notation we have

$$
\frac{1}{N} \sum_{1}^{N} T_i = \frac{1}{N} \sum_{1}^{N} (W_i + R_i) = \frac{1}{N} \sum_{1}^{N} W_i + \frac{1}{N} \sum_{1}^{N} R_i
$$

The average response time is the sum of the average wait time and the average runtime.
Average slowdown (aka “expansion factor”)

• The “slowdown” of a job is
  – The ratio between its response time & its runtime
  – slowdown = responseTime / runTime = Ti/Ri
  – slowdown = (waitTime + runTime) / runTime = (Wi+Ri) / Ri = Wi/Ri + 1

• Like wait & response, we aspire to minimize the average slowdown
  – slowdown = 1 ⇔ job was immediately scheduled
  – The greater the slowdown, the longer the job is delayed

• Examples
  – If Ri = 1 minute, and Wi = 1 minute
    • Job has a slowdown of (Wi+Ri)/Ri = (1+1)/1 = 2
    • Namely, job was slowed by a factor of 2 relative to its runtime
  – If Ri = 1 hour, and Wi = 1 minute, then the slowdown is much smaller
    • (Wi+Ri)/Ri = (60+1)/60 ≈ 1.02
    • The delay was insignificant relative to the job’s runtime

• The slowdown metric is also called
  – “Expansion factor”
Utilization & throughput

- **Utilization (aspire to maximize)**
  - Percentage of time the resource (CPU in our case) is busy
  - In this example, the utilization is:
    \[100 \times \frac{(3 \times 7 - 6)}{3 \times 7} = 71\%\]

- **Throughput (aspire to maximize)**
  - How much work is done in one time unit
  - Examples
    - Typical **hard disk** throughput: 50 MB/second (sequential access)
    - **Database** server throughput: transactions per second
    - **Web server** throughput: requests per second
    - **Supercomputer** throughput: job completions per second
      - In above example: \(4(\text{jobs})/7(\text{e.g., sec}) = 0.6 \text{ jobs per sec}\)
Which metric is more important?

• Depends on your point of view
  – Users typically care about wait time, response time, slowdown
  – System owners typically care more about utilization & throughput

• For users: wait time or slowdown?
  – Different notions of “fairness”
    • Wait time corresponds to FCFS
    • Slowdown depends on your runtime (aka “service time”)
    • What’s fairer? No definite answer – system owners decide
    • Performance analysts typically keep an eye on (=evaluate) both
  – Notice that
    • Average wait time tends to be dominated by long jobs
    • Average slowdown tends to be dominated by short jobs
    • Why? Slowdown: easy. Wait time: try to answer after next few slides
FCFS, EASY, backfilling, RR, SJF

BATCH SCHEDULING EXAMPLES
FCFS (First-Come First-Served) scheduling

• Jobs are scheduled by their arrival time
  – If there are enough free cores, a newly arriving job starts to run immediately
  – Otherwise it waits until enough cores are freed

• Pros:
  – Easy to implement (FIFO wait queue)
  – Typically perceived as most fair (we tend to dislike “אני רק שאלה”)

• Cons:
  – Creates fragmentation (unutilized cores)
  – Small/short jobs might wait for a long, long while.. (we disallow “אני רק שאלה”)

(Numbers indicate arrival order)
EASY (= FCFS + backfilling) scheduling

- The “backfilling” optimization
  - A short waiting job can jump over the head of the wait queue
  - Provided it doesn’t delay the job @ head of the wait queue

- EASY algorithm: whenever a job arrives (=submitted) or terminates:
  1. Try to start the job @ head of wait queue (FCFS)
  2. Then, iterate over the rest of the waiting jobs (in FCFS order) and try to backfill them
EASY (= FCFS + backfilling) scheduling

- **Pros**
  - Better utilization (less fragmentation)
  - Narrow/short jobs have better chance to run quicker

- **Con:**
  - Must know runtimes in advance
    - To know width of holes
    - To know if backfill candidates are short enough to fit holes

---

**FCFS**

1. Core 1
2. Core 2

**FCFS + Backfilling = “EASY”**

1. Core 1
2. Core 2
3. Core 3
4. Core 4

---

Time

OS (234123) - scheduling
EASY (= FCFS + backfilling) scheduling

• Backfilling mandates users to **estimate** how long their job will run
  • Upon job submission

• **If a job tries to overrun its estimate?**
  – It is killed by the system
  – Provides incentive to supply accurate estimates
    • Short estimate => better chance to backfill
    • Too short => jobs will be killed

• **EASY (and FCFS) are very popular**
  – Many supercomputer schedulers use them by default

• **BTW, EASY stands for**
  – Extensible Argonne Scheduling sYstem
  – Developed @ Argonne National Laboratory (USA) circa 1995
SJF (Shortest-Job First) scheduling

• Instead of
  – Ordering jobs (or processes) by their arrival time (FCFS)

• Order them by
  – Their (typically estimated) runtime

• Perceived as unfair
  – At the extreme: causes “starvation” (whereby a job waits forever)

• But optimal (in a sense) for performance
  – As we will see later on, using some theoretical computations

• NOTE: limit of job-scheduling theoretical reasoning (called: queuing theory)
  – Hard (sometimes impossible?) to do them for arbitrary parallel workloads
  – Therefore, for theoretical reasoning
    • We will assume jobs are serial (job=process)
  – Empirically, intuitions for serial jobs often also apply to parallel jobs
Average wait time example: FCFS vs. SJF

• Assume
  – Processes P1, P2, P3 arrive together in the very same second
  – Assume their runtimes are 24, 3, and 3, respectively
  – Assume FCFS orders them by their index: P1, P2, P3
  – Whereas SJF orders them by their runtime: P2, P3, P1

• Then
  – The average wait time under FCFS is
    • \((0+24+27)/3 = 17\)

  – The average wait time under SJF is
    • \((0+3+6)/3 = 3\)

• SJF seems better than FCFS
  – In terms of optimizing the average wait time metric
“Convoy effect”

- Slowing down all (possibly short) processes due to currently servicing a very long process
  - As we’ve seen in the previous slide

- Does EASY suffer from convoy effect?
  - Sure, it might, but less than FCFS
  - Empirically, when examining real workloads (recordings of the activity of real supercomputers), we find that there are often holes in which short/narrow jobs can fit, such that many of them manage to start immediately

- Does SJF suffer from convoy effect?
  - Sure, it might (when?), but less than FCFS

- Can we eliminate convoy effect altogether?
Optimality of SJF for average wait time

• **Claim**
  – Given: a 1-core system where all jobs are serial
  – If: (a) all processes arrive together and (b) their runtimes are known
  – Then: the average wait time of SJF is equal to or smaller than the average wait time of any other batch scheduling order S

• **Proof outline**
  – Assume the scheduling order S is: P(1), P(2), ..., P(n)
  – If S is different than SJF, then there exist two processes P(i), P(i+1) such that R(i) = P(i).runtime > P(i+1).runtime = R(i+1)
  – If we swap the scheduling order of P(i) and P(i+1) under S, then we’ve increased the wait time of P(i) by R(i+1), we’ve decreased the wait time of P(i+1) by R(i) and this sums up all the changes that we’ve introduced
  – And since R(i) > R(i+1), the overall average is reduced
  – We do the above repeatedly until we reach SJF
Fairer variants of SJF

• Motivation: we don’t want to starve jobs
• SJBF (Shortest-job *Backfilled* First)
  – Exactly like EASY in terms of servicing the head of the wait queue in FCFS order (and not allowing anyone to delay it)
  – But the *backfilling* traversal is done in SJF order

• LXF (Largest eXpansion Factor)
  – Recall that the “slowdown” or “expansion factor” metric for a job is defined to be:
    • slowdown = (waitTime + runtime) / runtime
  – LXF is similar to EASY, but instead of ordering the wait queue in FCFS, it orders jobs based on their current slowdown (greater slowdown means higher priority)
  – The backfilling activity is done “against” the job with the largest current slowdown (= the head of the LXF wait queue)
  – Note that every scheduling decision (when jobs arrive/finish) requires a re-computation of slowdowns (because wait time has changed)
PREEMPTIVE SCHEDULERS

RR, selfish RR, negative feedback, multi-level priority queue
Reminder

• **In a previous lecture**
  – We’ve talked about several processes states

• **In this lecture**
  – We only focus on two:
    • Ready & running

• **Namely, we assume that**
  – Processes only consume CPU
  – They never do I/O (of course, actually, they do, but we ignore that)
  – They are always either
    • Running, or
    • Ready-to-run
Preemption

• The act of suspending one job (process) in favor of another
  – Even though it is not finished yet

• Exercise
  – Assume a one-core system
  – Assume two processes, each requiring 10 hours of CPU time
  – Does it make sense to do preemption (say, every few milliseconds)?

• When would we want a scheduler to be preemptive?
  – When responsiveness to user matters (they actively wait for the output of the program), and
  – When runtimes vary (some jobs are shorter than others)

• Examples
  – Two processes, one needs 10 hours of CPU time, the other needs 10 seconds (and a user is actively waiting for it to complete)
  – Two processes, one needs 10 hours of CPU time, the other is a word processor (like MS Word or Emacs) that has just been awakened because the user clicked on a keyboard key
Quantum

• The maximal amount of time a process is allowed to run before it is preempted
  – 10s to 100s of milliseconds in general-purpose OSes (like Linux or Windows)

• Quantum is oftentimes set per-process
  – Processes that behave differently get different quanta, e.g., in Solaris:
    • A CPU-bound process gets long quanta but with low priority
    • Whereas an I/O-bound process gets short quanta with high priority
  – In Linux, the process “nice” value affects the quantum duration
    • “Nice” is a system call and a shell utility (see man)
Performance metrics for preemptive schedulers

- **Wait time (aspire to minimize)**
  - Same as in batch (non-preemptive) systems

- **Response time (or “turnaround time”; aspire to minimize)**
  - Like batch systems, stands for
    - Time from process submission to process completion
  - But unlike batch systems, instead of
    - \[ \text{responseTime} = \text{waitTime} + \text{runTime} \]
  - We have
    - \[ \text{responseTime} \geq \text{waitTime} + \text{runTime} \]
  - Because
    - Processes can be preempted, plus context switches have a price

- **Overhead (aspire to minimize); consists of**
  - How long a context switch takes, and how often context switches occurs

- **Utilization & throughput (aspire to maximize)**
  - Same as in batch (non-preemptive) systems
  - Although, may want to account for context switching overhead
RR (Round-Robin) scheduling

• Processes are arranged in a cyclic ready-queue
  – The head process runs, until its quantum is exhausted
  – The head process is then preempted (suspended)
  – The scheduler resumes the next process in the circular list
  – When we’ve cycled through all processes in the run-list (and we reach the head process again), we say that the current “epoch” is over, and the next epoch begins

• Requires a timer interrupt
  – Typically, it’s a periodic interrupt (fires every few millisecond)
  – Upon receiving the interrupt, the OS checks if its time to preempt

• Features
  – For small enough quantum, it’s like everyone of the N processes advances in 1/N of the speed of the core (sometime called “virtual time”)
  – With a huge quantum (infinity), RR becomes FCFS
RR in a parallel system?

• Yes! It’s called “gang scheduling”
  – Time is divided to slots (seconds or minutes)
  – Every job has a native time slot
  – Algorithm attempts to fill holes in time slots by assigning to them jobs from other native slots (called “alternative slots”)
    • Challenge: to keep contiguous chunks of free cores
    • Uses a “buddy system” algorithm
      http://www.cs.huji.ac.il/~feit/parsched/jsspp96/p-96-6.pdf
  – Algorithm attempts to minimize slot number using slot unification when possible
  – Why might gang scheduling be useful?
    • Don’t need to know runtime in advance (but what about memory?)
  – Supported by most commercial supercomputer schedulers
    • However, seems to be rarely used
    • If lots of memory is “swapped out” upon context switch, context switching will take a very long time... (often not worth it)
Gang scheduling - example

RR scheduling of (jobs in) time slots

CPU slots

CPUs
Gang scheduling – alternative slots
Gang scheduling – slot unification

Previous lecture ended here
Price of preemption: example

• Assume
  – One core, 1 second quantum
  – 10 processes, each requires 100 seconds of CPU time

• Assuming no context switch overhead (takes zero time)
  – Then the “makespan” metric (time to completion of all processes) is
    • $10 \times 100 = 1000$ seconds
  – Both for FCFS and for RR

• Assume context switch takes 1 second
  – The makespan of FCFS is
    • $10 \text{ (proc)} \times 100 \text{ (sec)} + 9 \text{ (ctx-sw)} \times 1 \text{ (sec)} = 1009$
  – Whereas the makespan of RR is
    • $10 \text{ (proc)} \times 100 \text{ (sec)} + 100 \text{ (proc)} \times 100 \text{ (ctx-se)} - 1 = 10,999$

• Shorter quantum
  – Potentially quicker response/wait time (why?), but higher overhead price
Batching vs. preemption

• **Claim**
  – Let \( \text{avgResp}(X) \) be the average response time under algorithm \( X \)
  – Assume a single core system, and that all processes arrive together
  – Assume \( X \) is a preemptive algorithm with context switch price = 0
  – Then there exists a non-preemptive algorithm \( Y \) such that \( \text{avgResp}(Y) \leq \text{avgResp}(X) \)

• **Proof outline**
  1. Let \( P_k \) be the last preempted process to finish computing
     \[
     P_k \quad P_r \quad \ldots \quad P_k \quad P_r \quad P_k
     \]
  2. Compress all of \( P_k \)'s quanta to the “right” (assume time progresses left to right), such that the last quantum remains where it is and all the rest of the quanta move to the right towards it
     \[
     P_r \quad P_r \quad \ldots \quad P_k \quad P_k \quad P_k
     \]
     (\( P_k \)'s response time didn’t change, and the response time of the other processes didn’t become longer)
  3. Go to 1 (until no preempted processes are found)
Aftermath

• Corollary: Based on (1) the previous slide and (2) the proof about SJF optimality from a few slides ago
  – SJF is also optimal relative to preemptive schedulers as well (that meet our assumptions)

\[
\text{avgResponse}(\text{SJF}) \leq \text{avgResponse}(\text{batch or preemptive scheduler})
\]

• So why do we use preemptive schedulers?
Connection between RR and SJF

• Claim
  – Assume:
    • 1-core system
    • All processes arrive together (and only use CPU, no I/O)
    • Quantum is identical to all processes + no context switch overhead
  – Then
    • $\text{avgResponse}(RR) \leq 2 \times \text{avgResponse}(SJF)$ // RR up to $2 \times$ “slower”

• Notation / definitions:
  – Process P1, P2, ..., P_N
  – $R_i =$ runtime of $P_i$
  – $W_i =$ wait time of $P_i$
  – $T_i = W_i + R_i =$ response time of $P_i$
  – $\text{delay}(i,k) =$ the delay $P_i$ caused $P_k$ under RR (how long $k$ waited due to $i$)
  – $\text{delay}(i,i) = R_i$
  – Note that $T_k = \sum_{i=1}^{N} delay(i,k)$
Connection between RR and SJF

**Proof outline**

- For any scheduling algorithm A, $N^* \text{avgResponse}(A)$ is always
  \[
  \sum_{k=1}^{N} T_k = \sum_{i=1}^{N} \sum_{j=1}^{N} \text{delay}_A(i, j)
  \]

- For $A=\text{SJF}$, we have
  \[
  \sum_{i=1}^{N} R_i + \sum_{1 \leq i < j \leq N} \left[ \text{delay}_A(i, j) + \text{delay}_A(j, i) \right]
  \]
  (because, given two processes, the shorter delays the longer)

- And for $A=\text{RR}$, we have
  \[
  \sum_{i=1}^{N} R_i + \sum_{1 \leq i < j \leq N} 2 \cdot \min(R_i, R_j)
  \]
  (assume $P_i$ is the shorter, then $P_i$ delays $P_j$ by $R_i$, and since it’s a perfect RR, $P_j$ also delays $P_i$ by $R_i$)
SRTF (Shortest-Remaining-Time First)

- Assume different jobs may arrive at different times
- SJF is not optimal
  - As it is not preemptive
  - And a short job might arrive while a very long job is running
- SRTF is just like SJF, but
  - Is allowed to use preemption
  - Hence, it’s “optimal” (assuming a zero context-switch cost)
- Whenever a new job arrives or an old job terminates
  - SRTF schedules the job with the shortest remaining time
  - Thereby making an optimal decision
Selfish RR

• New processes wait in a FIFO queue
  – Not yet scheduled

• Older processes scheduled using RR

• New processes are scheduled when
  1. No ready-to-run “old” processes exist
  2. “Aging” is being applied to new processes (some per-process counter is increased over time); when the counter passes a certain threshold, the “new” process becomes “old” and is transferred to the RR queue

• Fast aging
  – Algorithm resembles RR

• Slow aging
  – Algorithm resembles FCFS
Back to the context of general-purpose OSes

**PRIORITY-BASED, PREEMPTIVE SCHEDULERS**
Scheduling using priorities

• Every process is assigned a priority
  – That reflects how “important” it is
  – Can change over time

• Processes with higher priority are favored
  – Scheduled before processes with lower priorities

• The “priority” concept is also used for batch scheduler
  – SJF: priority = runtime (smaller => higher)
  – FCFS: priority = arrival time (earlier => higher)
Negative feedback principle

• General-purpose OSes (Linux, Windows, ...) embody a negative feedback policy in their scheduler
  – Running reduces priority to run more
  – Not running increases priority to run

• I/O-bound vs. CPU-bound
  – I/O-bound processes (that seldom use the CPU) get higher priority
  – Which is why editors are responsive even if they run in the presence of CPU-bound processes like
    • while(1) {
      sqrt( time() );
    }

• How about a video with high-frame rate? Or a 3D game?
  – Negative feedback doesn’t help them (they consume lots of CPU)
  – Need other ways to identify/prioritize them (nontrivial)
Multi-level priority queue

• Several RR queues
  – Each is associated with a priority
  – Higher-priority queues are at the “top”
  – Lower-priority queues are at the “bottom”

• Processes migrate between queues = have a dynamic priority
  – “Important” processes (e.g., I/O-bound or “interactive”) move up
  – “Unimportant” processes (e.g., CPU-bound or “non-interactive”) move down

• General-purpose OSes typically use some variant of multi-level priority queue
  – Priority is greatly affected by how much CPU is consumed by processes
    • I/O bound ↔ move up; CPU-bound ↔ move down
  – Some OSes allocate short quanta to the higher priority queues
  – Some don’t (i.e. do the opposite)
A comprehensive example

THE LINUX <= 2.4 SCHEDULER
The $\leq 2.4$ scheduler maintenance period

from here until here

$O(1)$

CFS

OS (234123) - scheduling
Definitions: task

• **For the reminder of this presentations**
  - All definitions henceforth relate to the <=2.4 scheduler
  - Not necessarily to other (Linux) schedulers

• **Within the Linux kernel**
  - Every process is called a “task”, likewise
  - Every thread (to be defined later on) is also called a “task”
Standard POSIX scheduling policies

• By POSIX
  – Each task is associated with one of three scheduling policies
    • “Realtime” policies
      – SCHED_RR (round robin),
      – SCHED_FIFO (first-in, first-out), or
    • The default policy
      – SCHED_OTHER
        » As opposed to the realtime policies, POSIX defines that the meaning of SCHED_OTHER is determined by OS
        » Typically employs some multilevel priority queue with the negative feedback loop
  – Realtime tasks are always favored by the scheduler, if exist

• Focusing on SCHED_OTHER
  – Ignoring realtime
Definitions: epoch

• (As mentioned earlier for RR...)
• Every runnable task gets allocated a quantum
  – CPU time the task is allowed to consume before it’s stopped by the OS
• When all quanta of all runnable tasks becomes zero
  – Start a new epoch, namely
  – Allocate an additional running time to all tasks
    • (Runnable or not)
Definitions: static/dynamic priorities

• **Task’s priority**
  – Every task is associated with an integer
  – Higher value indicates higher priority to run
  – Every task has two different kinds of priorities:

• **Task’s static priority component**
  – Doesn’t change with time
  – Unless user invokes the nice() system call
  – Indirectly determines the max quantum for this task

• **Task’s dynamic priority component**
  – The (i) remaining time for this task to run and (ii) its current priority
  – Decreases over time (while the task is assigned a CPU and is running)
  – When reaches zero, OS forces task to yield the CPU until next epoch
  – Reinitialized at the start of every epoch according to the static component
Definitions: HZ, resolution, ticks

• HZ
  – Linux gets a timer interrupt HZ times a second
  – Namely, it gets an interrupt every $1/HZ$ second
  – (HZ=100 for x86 / Linux 2.4)

• Tick
  – Two meanings (overloaded term):
    1. The time the elapses between to consecutive timer interrupts
       • Namely, tick = $1/HZ$ = 10 milliseconds, by default, for x86 / Linux 2.4
    2. The timer interrupt that fires every $1/HZ$ = 10milliseconds

• Scheduler timing resolution
  – The OS measures the passage of time by counting ticks
  – The units of the dynamic priority component are ‘ticks’
Definitions: per-task scheduling info

• Every task is associated with a task_struct
• Every task_struct has 5 fields used by the scheduler (to be discussed in the next few slides)
  1. nice
  2. counter
  3. processor
  4. need_resched
  5. mm
Definitions: task’s nice (kernel vs. user)

• The static component
  – In kernel, initialized to 20 by default
  – Can be changed by nice() and sched_setscheduler()
    • To be any value between 1 ... 40

• User’s nice (POSIX parameter to the nice system call)
  – Between -20 ... 19;
  – Smaller value indicates higher priority (<0 requires superuser)

• Kernel’s nice (field in task_struct used by the scheduler)
  – As noted, between 1 ... 40
  – Higher value indicates higher priority (in contrast to user’s nice)

• Conversion
  – Kernel’s nice = 20 - user’s nice
Definitions: task’s counter

• The dynamic component (time to run in epoch, & priority)
  – Upon task creation (integer arithmetic)
    • child.counter = parent.counter/2; parent.counter -= child.counter;
  – Upon a new epoch
    • task.counter = task.counter/2 + \( \frac{\text{NICE\_TO\_TICKS}(\text{task.nice})}{4} + 1 \)
      = half of dynamic + convert_to_ticks(static)
  – Decremented upon each tick (task.counter -= 1)

• NICE\_TO\_TICKS
  – Scales 20 (=DEF\_PRIORITY) to number of ticks comprising 50+ ms
  – Namely, scales 20 to 5+ ticks (recall that each tick is 10 ms by default):
    • \#define NICE\_TO\_TICKS(kern\_nice) ( (kern\_nice)/4 + 1 )

• Quantum range (without epoch accumulation) is therefore
  – ( 1/4 + 1=) 1 tick = 10 ms (min)
  – (20/4 + 1=) 6 ticks = 60 ms (default)
  – (40/4 + 1=) 11 ticks = 110 ms (max)
Definitions: processor, need_resched, mm

• Task’s processor
  – Logical ID of last core upon which task has executed most recently
  – If task is currently running
    • ‘processor’ = logical ID of core upon which the task executes now

• Task’s need_resched
  – Boolean checked by kernel just before switching back to user-mode
  – If set, check if there’s a “better” task than the one currently running
  – If so, context switch to it
  – Since this flag is checked only for the currently running task
    • Usually easier to think of it a per-core rather than per-task variable

• Task’s mm
  – A pointer to the task’s “memory address space” (more details later on)
Scheduler is comprised of 4 functions

1. **goodness(task,cpu)**
   - Given a task and a CPU, return how desirable it is for that CPU
   - Compare tasks by this value to decide which will run next on CPU

2. **schedule()**
   - Actual implementation of the scheduling algorithm
   - Uses goodness to decide which task will run next on a given core

3. **__wake_up_common()**
   - Wake up task(s) when waited-for event has happens
   - Event may be, e.g., completion of I/O

4. **reschedule_idle()**
   - Given a task, check whether it can be scheduled on some core
   - Preferably on an idle one, but if there aren't any, by preempting a less desirable task (according to goodness)
   - Used by both __wake_up_common() and by schedule()
int goodness(task t, cpu this_cpu) {  // bigger = more desirable
    g = t.counter
    if( g == 0 )
        // exhausted quantum, wait until next epoch
        return 0
    if( t.processor == this_cpu )
        // try to avoid migration between cores (why?)
        g += PROC_CHANGE_BONUS
    if( t.mm == this_cpu.current_task.mm )
        // prioritize threads sharing same address space
        // (why?)
        g += SAME_ADDRESS_SPACE_BONUS
    return g
}
Wakeup blocked task(s)

void __wake_up_common(wait_queue q) {
    // blocked tasks residing in q wait for an event that has just happened
    // so try to reschedule all of them...
    foreach task t in [q]
        remove t from q
        add t to ready-to-run list
        reschedule_idle(t)
}

void reschedule_idle(task t) {
    next_cpu = NIL
    if( t.processor is idle ) // t’s most recent core is idle
        next_cpu = t.processor
    else if( there exists an idle cpu ) // some other core is idle
        next_cpu = least recently active idle cpu
    else // no core is idle; is t more desirable
        // than a currently running task?
        threshold = PREEMPTION THRESHOLD
        foreach cpu c in [all cpus] // find c where t is most desirable
            gdiff = goodness(t,c) - goodness( c.current_task,c)
            if( gdiff > threshold )
                threshold = gdiff
                next_cpu = c
    if( next_cpu != NIL ) // found a core for t
        prev_need = next_cpu.current_task.need_resched
        next_cpu.current_task.need_resched = true
        if( (prev_need == false) && (next_cpu != this_cpu) )
            interrupt next_cpu
}
The heart of the scheduler

void schedule(cpu this_cpu) {
    // called when need_resched of this_cpu is on, when switching from
    // kernel mode back to user mode. need_resched can be set by, e.g.,
    // tick interrupt handler, or by I/O device drivers that initiate a slow op
    // (and hence move the associated tasks to a wait_queue)
    prev = this_cpu.current_task

    START:
    if( prev is runnable )
        next = prev
        next_g = goodness(prev, this_cpu)
    else
        next_g = -1

    foreach task t in [runnable && not executing]
        // search for ‘next’ = the next task to run on this_cpu
        cur_g = goodness(t, this_cpu)
        if( cur_g > next_g )
            next = t
            next_g = cur_g
The heart of the scheduler – continued

// ...continue function from previous slide...

if ( next_g == -1 ) // no ready tasks
    end function // schedule “idle task” (halts this_cpu)
else if( next_g == 0 ) // all quanta exhausted => start new epoch
    foreach task t
        t.counter = t.counter/2 + NICE_TO_TICKS(t.nice)
        goto START;
else if( next != prev )
    next.processor = this_cpu
    next.need_resched = false // 'next' will run next
    context_switch(prev, next);
    if( prev is still runnable )
        reschedule_idle(prev) // perhaps on another core

// ‘next’ (which may be equal to ‘prev’) will run next....
Definitions: task’s counter

• **Recall that**
  - task.counter = task.counter/2 + \( \frac{\text{NICE_TO.Ticks}(\text{task.nice})}{\alpha} \)
    = half of dynamic + convert_to_ticks(static)

• **Claim**
  - The counter value of an I/O-bound task will quickly converge to \( 2\alpha \)
  - (Prove it yourselves)

• **Corollary**
  - By default, an I/O bound task will have a counter of 12 ticks (=120 ms)
  - (So long as it remains “I/O bound” that consumes negligible CPU time)

• **Notice**
  - As mentioned earlier, this is why text editors remain responsive
  - Even in the face of heavy CPU-bound background tasks
  - When awakened upon event (e.g., key press), usually immediately get the CPU
Aftermath

• <=2.4 is in fact a multi-level priority queue
  – Every priority level integer is (logically) a RR queue
  – When counter is decreased/increased => task moves between queues

• Drawback
  – It is linear scheduler...
  – Indeed, the “O(1)” scheduler (successor of <=2.4) is O(1)
  – The CFS scheduler (successor of O(1)) is O(logn)
Cost of linearity – motivating O(1) scheduler

The charts show the duration of the `schedule()` function in cycles vs. the number of runnable tasks in the system for CPU 1 and CPU 2 of a system with 4 CPUs. The data suggests a linear relationship between the number of tasks and the duration of scheduling, which is a key characteristic of an O(1) scheduler.