Process Scheduling

Agenda

- Recap
- General scheduling mechanism
- Metrics and goals of scheduling
- Scheduling algorithms: SJF, STCF, RR, MLFQ, EDF
- Case studies: UNIX, Solaris, Windows XP
- Proportional share scheduler: case of Linux
- Scheduling in multiprocessors
- Scheduling in thread libraries
- Evaluating scheduling algorithms

Recap

- Concept of process
  - Loosely speaking, anything that runs on the CPU
- System calls to control what a process does (other than computing)
  - Creation of new process/thread
  - Communicating with another process/thread: message passing, shared memory, event signal
  - Saving and restoring the context of a process
- Multithreading libraries

General scheduling mechanism

- Process scheduling is the activity of selecting the process that will run next on the CPU
- If the scheduler needs to run in the kernel mode, there has to be a mode switch in the currently running process before scheduling can take place
  - The mode switch is usually a result of a system call or an interrupt
  - A scheduling decision may have to be taken only on a long-latency system call and some interrupts such as the timer interrupt
- A scheduler saves the context of the currently running process, selects a process from the ready queue, restores the context of selected process

General scheduling mechanism

- The scheduler can be invoked in four possible circumstances
  - The currently running process goes on a long-latency system call i.e. transitions from running to sleep state
  - A new process is created or a process completes a long-latency system call i.e., a process transitions from created to ready or from sleep to ready
  - The currently running process terminates
  - The currently running process receives a timer interrupt
  - The first and the third cases lead to non-preemptive or co-operative scheduling
  - The remaining two cases lead to pre-emptive scheduling

Goals of process scheduling

- A scheduling algorithm can target a subset of the following
  - Maximize throughput: rate at which processes complete (a performance metric)
  - Minimize turnaround time: time of completion minus time of arrival (a performance metric)
    - Average, maximum, standard deviation?
  - Minimize waiting time in the RUNNABLE state
    - Direct measure of scheduler efficiency
    - Average, maximum, standard deviation?
  - Minimize response time (a fairness metric)
    - Time to the next I/O call (read/write to a file)
    - Important for interactive systems
    - Average, maximum, standard deviation?

Scheduling algorithms: FCFS

- Non-preemptive first-come-first-serve
- May lead to a convoy effect if one process has large CPU bursts and others have small CPU bursts
- In the worst case, FCFS has an unbounded average waiting time

Scheduling algorithm: SJF

- Non-preemptive shortest next CPU burst scheduling
  - Popularly known as shortest job first scheduling
  - Provably optimal that achieves the minimum average waiting time and average turnaround time
  - Drawback: need to know future CPU bursts
  - One popular way of estimating CPU bursts is exponential averaging: $s(n+1) = at(n) + (1-a)s(n)$
    - Need to start with a guess for s(0), but little effect in long run; “a” is a parameter of the estimation algorithm
  - Pre-emptive version is called shortest remaining time first (SRTF) or shortest time to completion first (STCF)
    - Needed for handling arrival of short jobs at any time

Why SJF is optimal

- Intuitively, SJF minimizes the overall waiting time
- Consider a set of n processes
  - Consider a schedule S = <P_1, P_2, P_3, …, P_n>
  - Process P_k has a CPU burst of length t_k
  - Average TA time of this schedule = TA_S = $(t_1+(t_1+t_2)+(t_1+t_2+t_3)+…+(t_1+t_2+…+t_n))/n$
  - We will gradually convert S to SJF and in each step we will improve the average TA time
  - Let P_k be the first process in S such that $t_1<t_2<t_3<…<t_{k-1} > t_k$ i.e., this is the first point where S appears to deviate from SJF
  - We derive schedule S' from S by moving P_k forward to a position, say, after P_j such that $t_1<t_2<…<t_j<t_k<t_{j+1}<…<t_{k-1}$

Why SJF is optimal

- Consider a set of n processes
  - Consider a schedule S = <P_1, P_2, P_3, …, P_n>
  - Average TA time of this schedule = TA_S = $(t_1+(t_1+t_2)+(t_1+t_2+t_3)+…+(t_1+t_2+…+t_n))/n$
  - We derive schedule S' from S by moving P_k forward to a position, say, after P_j such that $t_1<t_2<…<t_j<t_k<t_{j+1}<…<t_{k-1}$
  - Completion times of P_1, P_2, …, P_j remain unchanged
  - Completion times of P_{j+1}, …, P_{k-1} increase by t_k each
    - Total increase in TA time = $(k-1-j-1+1)*t_k$
  - Completion time of P_k decreases by $t_{j+1}+t_{j+2}+…+t_{k-1}$
    - Since t_k is less than each of these terms, the total decrease in TA time is bigger than the total increase
  - Completion times of P_{k+1}, …, P_n remain unchanged

Why SJF is optimal

- Consider a set of n processes
  - Consider a schedule S = <P_1, P_2, P_3, …, P_n>
  - Average TA time of this schedule = TA_S = $(t_1+(t_1+t_2)+(t_1+t_2+t_3)+…+(t_1+t_2+…+t_n))/n$
  - We derive schedule S' from S by moving P_k forward to a position, say, after P_j such that $t_1<t_2<…<t_j<t_k<t_{j+1}<…<t_{k-1}$
  - We have proved that $TA_{S'} < TA_S$
  - In the next step, we derive schedule S'' by correcting the position of the next process that appear to deviate from SJF
  - In each step, the turnaround time goes down leading finally to SJF with the minimum turnaround time

Scheduling algorithms: Priority

- Each process is assigned a priority
  - Either by kernel based on the resource usage profile of the process or externally by the user
- The process with the highest priority is scheduled
  - Can be pre-emptive or non-preemptive
- A steady flow of CPU bursts from the high-priority processes can starve the low-priority ones
  - Age-based priority modulation solves this problem
- SJF is a special-case priority scheduling policy
  - Priority is inversely proportional to the next CPU burst length

Scheduling algorithms: Round-robin

- Pre-emptive quantum scheduling
  - The ready queue is treated as a circular FIFO and each process in the FIFO order is assigned a fixed time slice or quantum for execution
  - If the running process goes to sleep or terminates before the expiry of the quantum, the next process in the FIFO order is scheduled
  - If the running process's quantum expires, it is put back at the tail of the ready queue
  - The time quantum should be chosen to be larger than the context switch overhead
    - Too large a time quantum leads to FCFS sheduling
    - 80% of the CPU bursts should be shorter than the time quantum: is this thumb rule useful?

Scheduling algorithms so far

- SJF, STCF: optimizes turn-around time, bad for response time
  - Penalizes response time of long jobs
  - Unfair
- Round-robin: optimizes response time, bad for turn-around time
  - Gives turn to all jobs
  - Fair
- Notice the trade-off between fairness and performance
  - Performance is inversely related to fairness

I/O bursts

- When a process initiates an I/O, it goes to blocked state and de-scheduled
  - The scheduler picks another job for running
  - Overlaps I/O burst of a process with the CPU burst of another process
  - Ideally, the scheduler's goal is to overlap all I/O bursts with available CPU bursts
- The scheduler views a process as a sequence of interleaved CPU and I/O bursts
  - A scheduling algorithm simply schedules a CPU burst from among the CPU bursts of all ready processes
  - STCF or SJF needs to estimate the next CPU burst length of each ready (runnable in xv6 lingo) process

I/O bursts

- Consider two processes A and B
  - A has five CPU bursts each of 10 ms length interspersed by 10 ms long I/O bursts
  - B has one CPU burst of length 50 ms and no I/O
  - SJF or SCTF would schedule A first for 10 ms, then B for 10 ms, then A for 10 ms, …
  - Round-robin with time quantum of 5 ms would schedule A (could schedule B also), B, A, B, B, A, B, A, B, B, A, …
- In general, a scheduling algorithm schedules CPU bursts only
  - I/O bursts get taken care of on their own

Multi-level feedback queue (MLFQ)

- Recap: two primary goals of a scheduling policy
  - Minimize turnaround time (performance goal)
    - Schedule short jobs first
  - Minimize response time (fairness goal)
    - Schedule I/O-intensive jobs without much delay
- Need to achieve both goals without any knowledge of future CPU burst lengths of jobs
  - Recall that SJF minimizes turnaround time with future knowledge, but hurts fairness; Round-robin offers fairness, but hurts turnaround time
- MLFQ is a family of scheduling algorithms that tries to achieve these goals

Multi-level feedback queue (MLFQ)

- Why both goals are important?
  - For handling different kinds of processes
  - Foreground processes would require a scheduling policy that minimizes response time e.g., round-robin
  - Background processes are usually CPU-bound, less interactive, and need to minimize turnaround time
- How about multiple ready queues each having its own policy?
  - One for each type of processes (how to know type?)
  - Order queues by priority for scheduling across queues
  - Within a queue, the scheduling algorithm depends on the type of the processes in the queue

Multi-level feedback queue (MLFQ)

- Multiple ready queues each having its own policy
  - Drawback: need to know the type/priority of a process beforehand to decide its ready queue
  - Drawback: a process cannot migrate from one queue to another even if its behavior has changed
    - The priority of a process may change over its life time
  - Such a static arrangement is not attractive
- Multi-level feedback queue (MLFQ)
  - Maintain multiple ready queues ordered by priority
  - Processes within each queue are scheduled using RR
  - A new process always enters the highest priority queue, but can move between queues during its life

Multi-level feedback queue (MLFQ)

- Multi-level feedback queue (MLFQ)
  - A process is enqueued in exactly one queue at any given point in time
  - Scheduling rules
    - Processes in a given queue Q are scheduled using RR provided all queues at higher priority levels than Q are empty
    - If a running process uses up the complete scheduling quantum of RR, it is inferred to be a less interactive process and demoted one level down in priority i.e., enqueued into the next lower priority queue
    - If a running process invokes a blocking I/O before completing the scheduling quantum of RR, its priority is not changed and kept in the same queue

Multi-level feedback queue (MLFQ)

- Multi-level feedback queue (MLFQ) case studies
  - Assume that all queues have the same scheduling quantum of T
  - Suppose the ready queues are Q1, Q2, …, Qn ordered by increasing priority levels
  - Scenario#1: one job with long CPU burst
    - Runs for T time in Qn, another T time in Qn-1, …, T time in Q2, settles in Q1 for the rest of its life

Multi-level feedback queue (MLFQ)

- Multi-level feedback queue (MLFQ) case studies
  - Scenario#2: two jobs A and B; A has a long CPU burst and B has a short CPU burst; both bursts are bigger than T; B arrives later than A but before A finishes
    - When B arrives, A will be in a lower priority queue; so B gets scheduled immediately after arriving
    - B will keep getting higher priority until it finishes or joins A in the same queue whichever happens earlier
    - This scenario shows how MLFQ approximates SJF by assuming that all jobs have short CPU bursts to begin with
      - If they don't, they will gradually settle with lower priority
      - Note that this is only an approximation and deviation from SJF can be large in certain pathological scenarios

Multi-level feedback queue (MLFQ)

- Multi-level feedback queue (MLFQ) case studies
  - Scenario#3: two jobs A and B; A has a long CPU burst and B is I/O-intensive having short CPU bursts (less than T) interspersed with I/O bursts
    - B will stay in Qn throughout its life while A will gradually get demoted to Q1
    - A will be scheduled only when B is doing I/O because only at that time, Qn will be empty
    - This scenario shows how MLFQ minimizes response time by giving higher priority to more interactive jobs
    - This scenario also opens up two drawbacks of MLFQ presented so far
      - What if there is a large number of I/O-intensive processes? They can completely monopolize the CPU and starve other processes
      - What if someone purposefully writes a program that has CPU bursts of length 0.99T and a very short I/O? Can monopolize CPU

Multi-level feedback queue (MLFQ)

- Multi-level feedback queue (MLFQ) drawbacks
  - The second drawback shows a possible DoS attack
  - Additional drawback: what if a process exhibits different behavior during different phases of execution?
    - Switches between being CPU-bound and I/O-bound
    - Easy to imagine a program that initially takes a lot of inputs from stdin or other files and then computes on the data and finally, writes a lot of outputs to stdout or files
      - Starts of as I/O-intensive, then becomes CPU-bound, and finally, becomes I/O-intensive
    - One-way switch from I/O-intensive to CPU-bound is already handled, but not the other way
    - Need to have a mechanism of priority boost

Multi-level feedback queue (MLFQ)

- Two general ways to do priority boost in MLFQ
  - Gradual boost: move a process one priority level up if it fails to use up the scheduling quantum for N consecutive schedule instances
    - Possible to devise other gradual boosting algorithms
  - One-shot boost: move all processes to Qn at a regular interval of S timer ticks
  - For both, we need to fix the parameter N or S
    - Fixing any such constants is a difficult task because a designer would not know what would work best for a large collection of jobs of different kinds
  - Priority boost addresses the first and the third drawbacks adequately

Multi-level feedback queue (MLFQ)

- To avoid the DoS attack, MLFQ needs to revise its priority demotion algorithm
  - Revised algorithm: keep an accumulated sum of CPU time consumed by a process in a queue; once the sum crosses a threshold t, it is moved down by one priority level
    - Fixing the threshold t for each queue is another difficult job
    - An easy solution could be to set t=T for all queues
    - A process's accumulated CPU time is reset to zero when it enters a new queue
    - Highly interactive jobs will lose priority slowly over time while a malicious CPU hog process will lose priority faster

Multi-level feedback queue (MLFQ)

- Multi-level feedback queue (MLFQ) parameters
  - Another important parameter in MLFQ is the scheduling quantum T
    - We have assumed it to be same for all queues, but need not be
  - It makes sense to have a smaller T for higher priority queues so that the scheduler can cycle through all the jobs in those queues quickly
    - These are highly interactive jobs
  - In general, T should grow as one moves down the priority levels
    - Also, it makes sense to have a large T for low priority queues because these processes have large CPU bursts

Multi-level feedback queue (MLFQ)

- MLFQ parameters
  - Number of priority levels (number of queues)
  - Scheduling quantum of each queue
  - Algorithm for priority demotion
    - How frequently invoked?
  - Algorithm for priority boost
    - How frequently invoked?
- Some implementations hardcode the constants based on empirical study with a large pool of jobs
- Some implementations compute the parameters at run-time by observing system behavior

Scheduling algorithms: Deadlines

- In real-time systems, usually a deadline is attached with each process
  - Scheduling algorithm needs to minimize the average, maximum, or standard deviation of overshoot
  - Two types of processes
    - With hard deadlines (must be met)
    - With soft deadlines (minimize some function of overshoot)
  - Tempting to sort the processes by deadline and scheduling them in earliest-deadline-first order
    - No guarantee on overshoot if all the deadlines cannot be met
  - How to handle a continuous flow of processes?
    - All deadlines not known a priori (pre-emptive EDF)

Scheduling algorithms: Deadlines

- How to make sure that the owner of a process submits the true deadline?
  - Possible to submit an earlier deadline than the actual to gain priority in scheduling
    - Particularly problematic if the user knows that the scheduling algorithm is EDF
    - Any priority scheme that depends on deadline alone will suffer
- No solution would be perfect in this case
  - If there are too many processes, the scheduler could switch to round-robin
  - Important for the user to specify correct deadlines when the hard deadlines are mission-critical

Case study: UNIX

- Priority-based fixed quantum scheduling
  - Priority of a process may change during its life time
  - A higher priority value indicates lower priority
  - The timer interrupt handler keeps track of the CPU usage of the currently running process
  - Every one second, the priorities of all the user mode processes are updated by the timer interrupt handler
    - Set CPU usage of each process to CPU usage/2
    - Set priority value of each process to (base priority + CPU usage/2). The base priority is the minimum user mode priority value. Priority is inversely related to CPU usage.

Case study: UNIX

- Priority-based fixed quantum scheduling
  - A process on entering the kernel mode receives a priority value lower than the user mode base priority indicating a higher priority than all user mode processes
  - A process about to enter the sleep state (in kernel mode) receives a predetermined priority value
    - Value depends on the reason to sleep
    - I/O calls from lower levels of the kernel receive very high priority because these calls hold a lot of resources
    - Disk I/O always receives higher priority than a process waiting for some memory resources
    - This priority is used to schedule the process right after it returns to the ready queue at the end of the sleep (still in kernel mode)

Case study: UNIX

- Priority-based fixed quantum scheduling
  - A switch from kernel to user mode sets the process priority value to at least the user mode base priority
  - The priority of a process can be controlled by the owner of the process through the nice() call
    - Takes an integer argument, which is usually non-negative
    - The passed argument is added to the priority value of the process
    - The value of the argument is usually called the "nice value" indicating how nice the process is to the other processes
    - The nice value, the priority value, and the CPU usage value of a process are stored in the process table entry
    - Forked children inherit the nice value of the parent
  - Priority ties are broken by scheduling the process with a larger waiting time in the ready queue

Case study: Solaris

- Four classes of processes
  - Time sharing, interactive, system, and real-time
- A user process is classified as time sharing unless it is interactive (e.g., GUI process) or real-time
  - Multi-level feedback queue scheduling with queues ordered by priority; a lower priority queue can be scheduled only if all higher priority queues are empty
  - The processes in the highest priority queue get the smallest time quantum (can be switched quickly so that all high priority processes make some progress)
  - When a process returns from sleep state to ready state, its priority is boosted to somewhere between 50 and 59

Case study: Solaris

- Interactive class executes a similar scheduling algorithm as the time sharing class
  - The GUI processes are assigned the highest priority
- System class includes all kernel processes
  - User processes executing in kernel mode are not in this class
  - Scheduler, system daemons, etc. are in this class
  - Each system process has a pre-determined fixed priority, based on which they get scheduled
- Real-time processes are time-critical
  - Assigned highest global priority

Case study: Solaris

- At the time of selecting a new process for scheduling
  - The scheduler translates all local priorities within each class into global priorities
    - Done through a pre-determined priority order among the classes
  - Selects the process with the highest global priority
    - If there are multiple such processes, all of them are chained up in a single special queue
    - This special queue is scheduled in a round-robin fashion with a fixed pre-determined time quantum assigned to each process
  - A process is pre-empted if it goes to sleep state, or its time quantum expires, or a new higher-priority process arrives

Case study: Windows XP

- Pre-emptive priority-based scheduling
  - 42 priority levels
  - Six priority classes: REALTIME, HIGH, ABOVE_NORMAL, NORMAL (this is default), BELOW_NORMAL, and IDLE
  - Seven relative priority levels within each priority class: TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL (this is default), BELOW_NORMAL, LOWEST, and IDLE
  - The global priority level of a process is determined based on its priority class and its priority level within the class
    - The Win32 kernel thread library allows the user to control these two parameters from the user program

Case study: Windows XP

- Pre-emptive priority-based scheduling
  - The priority has an inverse relationship with CPU usage like in UNIX
  - A process returning to the ready queue from sleep state undergoes a priority boost
    - The amount of boost depends on the reason for sleep
    - A process sleeping on keyboard I/O always gets a bigger priority boost compared to one sleeping on disk I/O
  - Interactive processes use a different variable-quantum scheduling algorithm (next slide)

Case study: Windows XP

- Pre-emptive priority-based scheduling
  - The process currently selected on the screen is called a foreground process and everything else is a background process
    - The foreground processes always get a bigger quantum than the background processes
    - When a background process moves to foreground, its quantum is multiplied by a constant positive integer larger than one (usually three)

Proportional share scheduling

- Consider a set of processes with static priorities
  - p_1, p_2, …, p_n
  - These are sometimes referred to as tickets
    - Tickets to access CPU; bigger tickets mean more important
  - If the priorities do not change, it would be unfair to schedule the process with the highest priority always
  - Would like to offer a process a share of the CPU that is proportional to its priority
  - Let T = $\sum_{i=1}^{n} p_i$
  - Weight of process i is $w_i = p_i/T$
  - Let $S_k = \sum_{i=1}^{k} w_i$

Proportional share scheduling

- Randomized scheduling algorithm
  - Popularly known as lottery scheduling
  - Generate a random number X uniformly distributed in [0, 1)
  - Schedule process k such that $S_{k-1} < X \leq S_k$
  - Probability that process k is scheduled = $Pr_k = Pr(S_{k-1} < X \leq S_k) = Pr(X \leq S_k) - Pr(X \leq S_{k-1}) = S_k - S_{k-1} = w_k$
  - Note the elegance of the algorithm: needs absolutely no per-process state to be maintained
  - Only downside is that in a short span of time, it may not be as fair as a deterministic algorithm would be
    - In the long run, the scheduler will achieve the goal of proportional share as Pr_k tends to w_k

Proportional share scheduling

- Deterministic scheduling algorithm
  - Popularly known as stride scheduling algorithm
  - Let stride of process k be $S_k = LCM(p_1, p_2, \ldots, p_n)/p_k$
  - Let progress of process k be P_k initialized to 0
  - The scheduling algorithm schedules the process k with minimum P_k and sets P_k to $P_k + S_k$
  - Example: processes A, B, C have priorities 10, 20, 30
    - Schedule: A, B, C, C, B, C, A, B, C, C, B, C, …
  - Even within short span it offers proportional share of CPU to the processes

Proportional share scheduling

- Deterministic scheduling algorithm
  - Downsides
    - Needs to maintain per-process progress
    - Needs an efficient algorithm to find the process with minimum progress from among all processes
    - Dealing with a new process is very problematic because if we initialize its progress to zero, it will monopolize the CPU for quite some time
- Completely fair scheduler (CFS) of Linux implements the basic idea of stride scheduling
  - Short-term fairness is important
  - Optimizes data structures to speed up min finding
  - Updates priorities

Linux CFS

- CFS answers two basic questions related to process scheduling
  - Which process to schedule?
  - When a process is scheduled, how long should it run before undergoing a context switch?
  - CFS keeps track of the total number of CPU cycles consumed by each process
    - Referred to as the virtual runtime (vruntime) of the process
  - CFS schedules the process with minimum vruntime

Linux CFS

- CFS answers two basic questions related to process scheduling
  - When a process is scheduled, how long should it run before undergoing a context switch?
  - If a set of n processes has equal priority, max(min_granularity, sched_latency/n) is the amount of time each process runs before undergoing a context switch
    - Over each window of sched_latency time, CFS maintains proportional share
  - Usually processes have different priorities
    - Need to take this into account in CFS

Linux CFS

- Priorities in CFS are derived from nice values
  - Nice values range from -20 to +19 with zero being the default nice value of a process
  - Nice value of a process can be changed through nice() function call
  - Higher the nice value, lower is the priority
    - +19 represents the lowest priority
    - priority value = 120 + nice value; ranges from 100 to 139
    - A lower priority value means higher priority
  - CFS maps these forty nice values (or equivalently forty priority values) to integer weights maintaining monotonicity
    - Lower nice value means higher weight

Linux CFS

- Nice value to weight mapping
  - CFS sets a goal that if the nice value of a process goes down by one the process will get 10% more CPU time
  - Understanding how this goal is translated to weights
    - Consider two processes A and B having the same nice value n (and hence the same weight w(n))
    - sched_latency=2t is divided into the two processes equally and let's suppose each process gets a time quantum of t
    - Suppose A decreases its nice value by one; the CFS goal says that now sched_latency of 2t will be divided among A and B as 1.1t and 0.9t respectively (proportional share)
    - Notice how changing A's nice value affects the quanta of both processes due to a fixed sched_latency window
    - So, we need w(n-1)/w(n) = 1.1/0.9 = 1.22
    - CFS approximates this ratio as 1.25

Linux CFS

- Nice value to weight mapping
  - Given nice value n in the range -20 to +19, CFS uses $w(n) = [1024/(1.25)^n]$ where [x] is the integer part of x
  - w(0) is 1024 which is the default weight of a process
- Now we can calculate the two key values for CFS
  - Time quantum for process k = max(min_granularity, sched_latency*($w_k/\sum_{i=1}^{n} w_i$))
    - When process k is scheduled, it will run this long
  - Virtual run time of a process k is accumulated as $vruntime_k$ += (($w(0)/w_k$)*$runtime_k$)
  - Virtual and actual run times of a process with nice value zero are same

Linux CFS

- CFS scheduling policy is simple: schedule the process with minimum vruntime
  - Operations
    - find and delete the entry of the process with minimum vruntime from ready queue (needed at the time of scheduling the process)
    - insert a process into the ready queue (needed when a process enters or returns to ready state)
  - At first blush, a binary heap appears to be the ideal data structure for maintaining the ready processes
    - Finding the process with minimum vruntime has O(1) time
    - But needs to maintain the heap property on deletion and insertion (both require O(log n) time, n being the number of processes)

Linux CFS

- CFS scheduling policy is simple: schedule the process with minimum vruntime
  - Operations
    - find and delete the entry of the process with minimum vruntime from ready queue (needed at the time of scheduling the process)
    - insert a process into the ready queue (needed when a process enters or returns to ready state)
  - At first blush, a binary heap appears to be the ideal data structure for maintaining the ready processes
    - Finding the process with minimum vruntime has O(1) time
    - But needs to maintain the heap property on deletion and insertion (both require O(log n) time, n being the number of processes)

Linux CFS

- CFS scheduling policy is simple: schedule the process with minimum vruntime
  - One drawback of binary heap is that the implementation of a heap inside the Linux kernel is array-based and the way the space for the array is allocated requires contiguous physical memory
  - Getting large contiguous physical memory may be difficult
    - May need to move several processes to create a large enough free region in physical memory
  - Of course, it is possible to rewrite the binary heap implementation to not require contiguous memory
    - However, there are other equally good solutions

Linux CFS

- CFS scheduling policy is simple: schedule the process with minimum vruntime
  - CFS maintains the ready processes in a red-black tree which is a balanced binary search tree
  - Finding the process with minimum vruntime and deleting it has O(log n) time complexity
  - Insertion also has O(log n) time complexity
  - Note that a linked list of ready processes would lead to very poor performance
    - Number of processes is usually several hundreds to thousand in busy servers

Linux CFS

- Observations
  - Weight of a process does not change throughout its life unless explicit nice() calls are made by the user program
    - Default nice value of a kernel process is negative to make sure that it enjoys a bigger weight that a default user process
  - Interactive processes are expected to have the lowest virtual runtime because they do not use CPU much
    - A process can monopolize the CPU for a long time after returning from sleep/blocked state until its virtual runtime catches up with others'
    - To avoid this problem, when a process returns from the sleep/blocked state, its virtual runtime is set to the current minimum virtual runtime among all ready processes

Multiprocessor scheduling

- Two important aspects: load balancing and data affinity
- Where does the OS code run? Options:
  - A group of processors dedicated to carry out kernel activities (known as asymmetric scheduling)
  - A single OS node simplifies OS design and improves performance
  - The OS code runs on the processor that asks for a kernel service (known as symmetric scheduling)
- Design of the ready queue
  - Centralized (one queue: SQMS), distributed (one per processor: MQMS), hierarchical (combination of both)

Data affinity in multiprocessors

- A process can be scheduled on different processors during different quanta given to the process
  - Known as process migration
  - Each migration comes with the overhead of cache warm-up and possible remote memory accesses
  - A process can specify its affinity toward a processor through system calls
    - Prevents the scheduler from migrating the process to a different processor
  - A multiprocessor scheduler must be aware of data affinity and the overheads involved in migration

Multiprocessor scheduling: SQMS

- SQMS is easy to implement
  - Whenever a CPU needs to pick a new process to schedule, it can grab one from SQMS following any scheduling algorithm discussed so far
  - Maintains load balance: each CPU is equally busy
- Two major drawback with SQMS
  - The deletion from and insertion into the ready queue must be serialized and cannot be executed concurrently by multiple CPUs: hurts performance
    - Can run into correctness problems (e.g., if two CPUs try to grab the process at the head of the queue)
  - Can severely hurt data affinity

Multiprocessor scheduling: SQMS

- SQMS and data affinity
  - Consider five processes A, B, C, D, E and four CPUs
  - Each CPU grabs the process at the head of queue
  - When a process's time quantum finishes, it is inserted at the tail of the queue
  - CPU0: A, E, D, C, B, A, …
  - CPU1: B, A, E, D, C, B, …
  - CPU2: C, B, A, E, D, C, …
  - CPU3: D, C, B, A, E, D, …
  - Notice that a process keeps getting migrated hurting data affinity

Multiprocessor scheduling: SQMS

- SQMS and data affinity
  - One possibility is to keep the set of processes fixed for a CPU
  - CPU0: A, E, A, E, …
  - CPU1: B, B, B, B, …
  - CPU2: C, C, C, C, …
  - CPU3: D, D, D, D, …
  - This leads to load imbalance: CPU0 is more loaded than other CPUs

Multiprocessor scheduling: SQMS

- SQMS and data affinity
  - Another possibility is to maintain load balance while minimizing migration
  - CPU0: A, E, A, A, A, A, …
  - CPU1: B, B, E, B, B, B, …
  - CPU2: C, C, C, E, C, C, …
  - CPU3: D, D, D, D, E, D, …
  - This leads to migration of E only
  - Note that every process gets a fair share of CPU time otherwise
  - In general, both load balance and data affinity cannot be satisfied

Multiprocessor scheduling: MQMS

- MQMS uses one ready queue per CPU and as a result, solves the data affinity problem
  - Processes normally do not migrate from one CPU's ready queue to another's
  - Load imbalance is a problem with MQMS because the number of processes in the ready queues may not be equal all the time
    - Rebalancing through cross-CPU migration needs to happen periodically
    - This is sometimes known as work stealing (one CPU steals some work from another CPU)
    - An important parameter is the frequency of rebalancing
      - Too frequent: high migration overhead, but good load balance
      - Less frequent: low migration overhead, but poor load balance

Load balancing in multiprocessors

- Distributed and hierarchical ready queue designs may lead to load imbalance
  - Scheduler's responsibility is to keep all CPUs equally busy
  - Receiver-initiated and sender-initiated diffusion (RID and SID)
    - Also known as pull migration and push migration
    - RID is invoked on a CPU whose ready queue size has dropped below a threshold; migrates a number of processes from a CPU whose ready queue size is above the threshold
    - SID is invoked on a CPU whose ready queue size has gone above a threshold

Load balancing in multiprocessors

- RID and SID algorithms can work together
  - In multiprocessor Linux, every 200 ms the SID algorithm is invoked on every CPU and the RID algorithm is invoked whenever the ready queue of a CPU is empty
- Load balancing and data affinity have conflicting goals
  - Achieving both is usually challenging

Linux multiprocessor scheduler

- Three popular schedulers in Linux community
  - O(1) scheduler
    - Priority-based, derived from MLFQ, uses multiple queues (one per CPU)
  - CFS
    - Proportional share-based, derived from stride scheduling algorithm, uses multiple queues (one per CPU)
  - BFS
    - Proportional share-based, uses single queue (SQMS)

Scheduling in thread libraries

- Kernel-level threads are scheduled by the OS and the user-level threads are scheduled by the thread libraries
  - A group of user-level threads may be mapped to one or more kernel-level threads
- Process contention scope is used to carry out user-level thread scheduling within a process
  - OS scheduler maps the user-level threads to the available kernel-level threads
  - When a kernel-level thread is scheduled, the user-level threads mapped to it share the quantum through user-level thread switching or one of the mapped threads runs
  - User-level thread priorities are interpreted relative to the global priority

Scheduling in thread libraries

- System contention scope
  - All threads are visible to the kernel
  - Thread priorities are interpreted relative to the process threads only
  - Linux supports only system contention scope
- Contention scopes in the POSIX thread library
  - pthread_attr_getscope() and pthread_attr_setscope() for getting and setting scheduler scope
    - Two defined scopes: PTHREAD_SCOPE_PROCESS and PTHREAD_SCOPE_SYSTEM

Scheduling in thread libraries

- POSIX thread library allows one to specify the thread scheduling policy and thread priorities
  - Priorities are non-negative integers less than 100
  - The scheduling algorithm can be read and written to using the pthread_attr_getschedpolicy() and pthread_attr_setschedpolicy() functions
  - The thread priorities can be read and written to using the pthread_attr_getschedparam() and pthread_attr_setschedparam() functions

Evaluating scheduling algorithms

- Must evaluate a set of algorithms before picking one
  - Need to decide the optimization criterion first
  - Possible example: maximize CPU utilization under the constraint that the maximum response time is T
  - Possible example: maximize system throughput under the constraint that the turnaround time of each process is linearly proportional to its execution time
    - Short-burst processes wait less
  - Once the criterion is decided, a set of candidate algorithms must be evaluated to select the best one

Evaluating scheduling algorithms

- Analytical modeling
  - Design a mathematical model for the entire system that takes as input the description of the scheduling algorithm and the description of the processes possibly in terms of the distribution of arrival times, values of CPU and I/O burst lengths or simply a sequence of CPU and I/O bursts
  - The model computes the target criterion
  - Such a model can be as simple as a single formula or as complicated as a queuing model
  - Usually difficult to capture the real-world behavior of the system, but useful for eliminating some obviously poor scheduling algorithms

Evaluating scheduling algorithms

- Simulation
  - Rigorous simulation must be carried out before selecting a few good candidates
  - A simulator is a software model of the system, but simpler than the full OS
  - The simulator can accept real-world user programs and simulate them to evaluate the target criterion
    - NachOS is a simplified OS simulator
- Final phase of the design involves incorporating the shortlisted candidate algorithms in the real OS and evaluating the target criterion