CPU, Memory & Disk: Limitations

* Status quo
  + CPUs execute an endless stream of instructions in memory
  + All system memory is in a contiguous physical address space
  + The disk is a finite set of blocks
  + All instructions execute in privileged mode
* To handle concurrent programs, the OS must separate the execution of different programs, providing the illusion to programs that each program is the only running program

Processes

* Definition: a process is an instance of the code we are running (the program)
  + One of the most profound ideas in CS
  + Not the same as “program” or “processor”
* Processes provide each program with two key abstractions
  + Logical control flow
    - Each program appears to have exclusive use of the CPU
    - Context switching
  + Private address space
    - Each program seems to have exclusive use of main memory
    - Virtual memory

Concurrent Processes

* Each process is a logical control flow
* Two processes run concurrently if their flows overlap in time
* Otherwise, they are sequential

Context switching

* Processes are managed by a shared chunk of memory-resident OS code called the kernel
  + The kernel is not a separate process but rather runs as part of some existing process
* Control flow passes from one process to another via context switching
* A context switch is a mechanism that allows the OD to store the current process state and switch to some other, previously stored context
* Reasons for a context switch:
  + The process completes/exits
  + The process is not using any CPU, so instead of waiting for it to finish, give the CPU to another process
  + The hardware requires OS help
  + The OS decides to stop the currently running process and start another to ensure all processes are getting started In a fair way

Process scheduling

* Scheduling has two aspects:
  + How to switch from one process to another
  + What process should run next
* How des the kernel switch from one process to another?
  + System has several ready processes
* How does the kernel stay in control?
  + Processes may yield() or execute I/O
  + Configurable timer interrupts let OS take control

Preemption

* If a task never gives up control, exits, or performs I/O, it could run forever ad the OS could not gain control
* Getting the control back from CPU is not easy
* The OS therefore sets a timer before scheduling a process
  + If the timer expires, the hardware interrupts the execution of the process and switches to the kernel
  + The kernel then decides if the process should continue

Process states

* Running, ready, blocked
* When a process becomes ready, it is scheduled to run, it does not run instantly because the CPU may be used
* Enters blocked state when it is waiting for something, and not actually using the CPU

2-13-24

Process

* A process is an instance of a running program
* Process provides each program with two key abstractions
  + Logical control flow – each program seems to have exclusive use of the CPU
  + Private address space – each program seems to have exclusive use of main memory
* Process states: running, ready, blocked

Scheduling policy

* The scheduling policy determines which process should run next. The policy decides which order to execute processes

Vocabulary

* Job
  + View as the current CPU burst of a process
  + Process alternates between cpu and I/O
  + Process moves between ready and blocked queues
* Workload
  + The description of your job (arrival time, run time)
* Scheduler
  + Responsible for implementing the scheduler logic
* Metric
  + Measurement of how good or bad the scheduling quality is

Scheduling metrics

* Turnaround time
  + The difference between when your job arrives and when it completes
  + T(turnaround) = t(completion) – t(arrival)
* Response time
  + Difference between when you first get to use the cpu and when you arrive. So the time it takes for the first execution.
  + T(response) = t(firstrun) – t(arrival)
* Fairness
  + All processes get the same amount of CPU over time
  + Performance and fairness are often at odds in scheduling
  + If you want to improve turnaround time and response time, you have to sacrifice fairness

Workload assumptions

* We start with some simplifying assumptions
  + Each job runs for the same amount of time
  + All jobs arrive at the same time
  + All jobs only use the CPU (there is no I/O)
  + The run-time of each job is known

First in first out (FIFO)

* First come, first served (FCFS)
  + Very simple and easy to implement
  + Run jobs in arrival time order
* Example:
  + A arrived just before B, which arrived just before C
  + Each job runs for 10 seconds
  + What is the average turnaround time? (the avg time for each job to complete after arriving)
    - (30 + 20 + 10) / 3 = 20 seconds
  + What is average response time
    - (0 + 10 + 20) / 3 = 10 seconds
* This strategy is easy, simple, straightforward. What are the drawbacks?
  + There will be issues if the drop the workload assumptions (lets say each job is not necessarily equal length)
* Example:
  + Take the previous example, but say A now takes 100 seconds.
  + Now job B and C are waiting a very long time to execute
  + Avg turnaround time = 110 seconds
  + Avg response time = 70 seconds
* Long jobs delay short jobs, and turnaround / response times suffer (like a slow truck holding up traffic and not pulling over)

Shortest job first (SJF)

* Long running tasks delay other tasks, so we schedule the job with the shortest runtime to go next
* It knows which job is shortest by the forth assumption that we know the run-time of each job
* Example: same as prev where A is 100 secs, and C are 10 seconds.
  + Avg turnaround time = (10 + 20 + 120) / 3 = 50 seconds
  + Avg response time = (0 + 10 + 20) / 3 = 10 seconds
* Now lets drop the assumption that all jobs arrive at the same time
* Same example, but A arrives before B and C. now a is running already when B and C arrive and the avg turnaround / response times go back up.

Preemptive scheduling

* Previous schedulers (fifo, sjf) are non-preemptive
* Preemptive schedulers may take CPU control at any time to switch to another process according to the scheduling policy
* New schedule: add preemtion to SJF
  + Shortest job runs first, but when new jobs enter, if they are shorter we will pause the current long job to knock out the shorter jobs, then resume the long job.

Net metric: response time

* So far, we have optimized for turnaround time (completing the tasks as fast as possible)
* STCF and related disciplines are not particularly good for response time
* How can we build a scheduler that is sensitive to response time?
* Optimize response time:
  + Alternate ready processes every fixed-length time slice

Round Robin (RR) Scheduling

* Time slicing scheduling
  + Run a job for a time slice and then switch to the next job in the run queue until the jobs are finished
  + Time slice is sometimes called a scheduling quantum
* It repeatedly des so until the jobs are finished
* The length of a time slice must be a multiple of the timer-interrupt period
* Example:
  + A, B, C arrive at the same time and each run for 5 seconds
  + SJF response time would be bad (5 seconds average)
  + Give each job a 1 second time slice.
    - Now the avg initial response time is 1 second instead of 5.
    - But now they all finish at the same ish time, which is later for some of them, so average turnaround time is worse (14 seconds instead of 10)
* RR is fair to the jobs because each job gets equal opportunity to run but performs bad on turnaround time (this is by design because you have to sacrifice turnaround time for fairness)
* The length of the time slice is critical
  + The shorter time slice gives better response time, but the cost of context switching so much will kill overall performance
  + The longer time slice cuts back on the total cost of switching (amortize), but worse response time
* Now lets drop the assumption that all jobs only use CPU (no I/O)

Incorporating I/O

* Example:
  + A and B need 50 ms of CPU time each
  + A runs for 10 ms and then issues an I/O request
  + I/Os each take 10 ms
  + B simply uses the CPU for 50 ms and performs no I/O
  + The scheduler runs A first, then B after
    - The CPU will be unused for a significant amount of time while A is waiting on I/O
    - So we schedule B in the free time slots
    - **Look at slide 26 for** the chart of this scenario vs time
* When a job initiates an I/O request: the job is blocked (waiting for I//O completion), so schedule another job on the CPU
* When the I/O completes: an interrupt is raised, the OS moves the process from blocked back to ready state

We are still assuming all job run-times are known

Advanced Scheduling: Multi-Level Feedback Queue (MLFQ)

* General-purpose scheduling
* Challenge: the scheduler must support both long-running background tasks (batch processes) and low-latency foreground tasks (interactive processes)
  + Batch process: response time not important, reduce cost of context switches, cares about lots of CPU
  + Interactive process: response time is critical
* A scheduler that learns from the past to predict the future
* Basic rules
  + Approach: multiple levels of round-robin
  + MLFQ has a number of distinct queues, each queue has a different priority level
  + A job that is ready to run is on a single queue
  + Rule 1: the highest priority queue runs first
  + Rule 2: if 2 jobs have the same priority, they both run in round robin
  + Rule 3: when a job enters the system, it gets highest priority
  + Rule 4a: if the job uses up its entire initial time slice, its priority is reduced
  + Rule 4b: if a job finishes before its time slice runs out, it stays at the same priority
  + Rule 4: rewrite 4a and 4b: once a job uses up its time allotment at a given level, its priority is reduced
  + Rule 5: after some time period T, move all the jobs in the system to the topmost queue (reset system behavior)
* Mlfq varies the priority of a job based on its observed behavior
* Ex: a job that waits on a lot of IO gets high priority, but a job that uses a lot of CPU for a long time gets lower priority
* Problems:
  + Starvation: if there are too many short jobs, long running jobs will never receive any CPU time
  + Game the scheduler: after running 99% of a time slice, issue an IO operation. The job stays high priority and gains a higher percentage of CPU time
* How to prevent gaming
  + Rewrite rules 4a and 4b: once a job uses up its time allotment at a given level, its priority is reduced
* We can tune the queue by giving lower priority levels longer time slices

**2-27 Proportional share schedulers (lottery, stride, CFS)**

Proportional Share Scheduler

* Fair share scheduler
  + Guarantee that each job obtains a certain percentage of CPU time
  + Tickets:
    - Represent the share of a resource that a process should receive
  + Ex:
    - Process A has 75 tickets and gets 75% of CPU, B has 25 tickets and gets 25%

Lottery Scheduling

* The scheduler picks a winning ticket, runs the process that has the winning ticket
* Check slide 6 for example

Ticket mechanisms

* Ticket currency
  + A user allocates tickets among their own jobs in whatever currency they would like
  + The system converts the currency into the correct global value
* Ex:
  + There are 200 tickets (global currency)
  + Check slide 7
* Ticket transfer
  + A process can transfer excess tickets to another process
* Ticket inflation
  + A process can temporarily raise or lower the number of tickets it owns
  + If any one process needs more CPU time, it can boost its tickets
* Ex: (slide 9)
  + There are processes A, B, and C
  + Generate a random number say 300
  + For each job in the linked list, add the job’s ticket count to the sum
  + When the sum reaches 300, that job is the winner
  + Note jobs with the most tickets will get the most wins because they have a higher chance of pushing the current sum over the random number
* Fairness gets better the longer the jobs are

Deterministic Approach: Stride Scheduling

* Stride of each process = a large number / the number of tickets of the process
* Ex:
  + Define a large number 10000
  + Process A has 100 tickets, stride of A is 10000/100 = 100
  + B has 50 tickets: stride is 200
* A process runs, increment a counter (pass value) for it by its stride
* Pick the process to run that has the lowest pass value
* Ex: (slide 13)
  + All processes start with a pass value = 0, so arbitrarily pick one
  + We picked A, so update its pass value by its stride
  + Repeat, picking the process with the lowest pass value
* Stride scheduling outputs the exact right proportions always, whereas lottery has to utilize probability to get better fairness over time.

Lottery vs Stride Scheduling

* Stride scheduling need sto maintain the per-process pass value
* If a new job enters with a pass value 0, it will monopolize the CPU
* Advantages of lottery scheduling:
  + Check the slide 14

Linux Completely fair Scheduling (CFS)

* The current CPU scheduler In Linux
* Non-fixed timeslice: CFS assigns process’s timeslice a proportion of the processor
* Priority: enables control over priority by using nice value
* Efficient data structure: use red-black tree for efficient search, insertion and deletion of a process

CFS basics

* Virtual runtime (vruntime)
  + Denotes how long the process has been executing
* Per-process variable: increase in proportion with physical time when it runs
* CFS will pick the process with the lowest vruntime to run next
* How does the scheduler know how long to run the current process?
  + CFS switches too often: fairness is increased, but performance cost increases because too much context switching
  + CFS switches less often: performance is increased, but fairness is decreased
* CFS control parameters
  + to determine how long a process should run before switching, use sched\_latency
    - sched\_latency is a constant and has nothing to do with the processes
    - A typical value is 48ms
    - Process’s timeslide = sched\_latency / the number of processes
  + Ex on slide 18
* What if there are too many processes running?
  + Too small of timeslices, too much context switching
* Use min\_granularity
  + The minimum timeslice: usually 6ms
  + So if there are a fuck ton of processes, the timeslice calculated by sched\_latency / n is ignored and we use min\_granularity instead
* Nice value
  + CFS enables controls over process priority
  + Nice parameter is an integer value and can be set from -20 to +19
    - Positive nice values imply lower priority
    - Negative nice values imply higher priority
  + The nice value is mapped to a weight (value is not important)
  + Slide 21 for timeslice formula
    - Process A has -5 nice value so it has high priority, B has lower priority
  + Weight0 is the weight of the process with nice value 0 (this is arbitrary)
* I/O and sleeping process
  + Some processes can monopolize the CPU by sleeping strategically
  + CFS solution: set the vruntime of the process to the minimum value found in the tree when it wakes up