CS194-24
Advanced Operating Systems Structures and Implementation
Lecture 16

Specialized File Systems (con't)
Scheduling

March 31st, 2014

Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs194-24

Goals for Today

- Application-Specific filesystems (con't)
- Scheduling

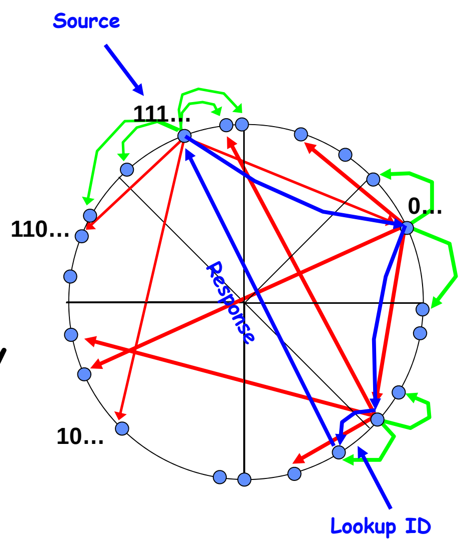
Interactive is important!

Ask Questions!

Note: Some slides and/or pictures in the following are adapted from slides ©2013

Recall: Lookup with Leaf Set

- Assign IDs to nodes
  - Map hash values to node with closest ID
- Leaf set is successors and predecessors
  - All that's needed for correctness
- Routing table matches successively longer prefixes
  - Allows efficient lookups
- Data Replication:
  - On leaf set



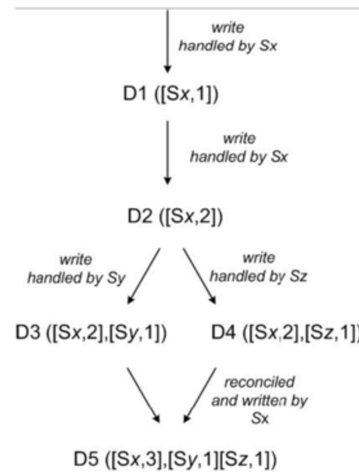
Recall: Dynamo Assumptions

- Query Model – Simple interface exposed to application level
  - Get(), Put()
  - No Delete()
  - No transactions, no complex queries
- Atomicity, Consistency, Isolation, Durability
  - Operations either succeed or fail, no middle ground
  - System will be eventually consistent, no sacrifice of availability to assure consistency
  - Conflicts can occur while updates propagate through system
  - System can still function while entire sections of network are down
- Efficiency – Measure system by the 99.9th percentile
  - Important with millions of users, 0.1% can be in the 10,000s
- Non Hostile Environment
  - No need to authenticate query, no malicious queries
  - Behind web services, not in front of them

Data Versioning

- A put() call may return to its caller before the update has been applied at all the replicas
- A get() call may return many versions of the same object.
- **Challenge:** an object having distinct version sub-histories, which the system will need to reconcile in the future.
- **Solution:** uses vector clocks in order to capture causality between different versions of the same object
  - A vector clock is a list of (node, counter) pairs
  - Every version of every object is associated with one vector clock
  - *If the counters on the first object's clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten.*

Vector clock example

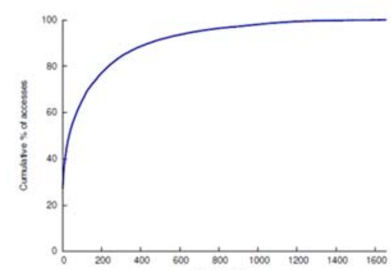


Conflicts (multiversion data)

- Client must resolve conflicts
  - Only resolve conflicts on reads
  - Different resolution options:
    - » Use vector clocks to decide based on history
    - » Use timestamps to pick latest version
  - Examples given in paper:
    - » For shopping cart, simply merge different versions
    - » For customer's session information, use latest version
  - Stale versions returned on reads are updated ("read repair")
- Vary N, R, W to match requirements of applications
  - High performance reads: R=1, W=N
  - Fast writes with possible inconsistency: W=1
  - Common configuration: N=3, R=2, W=2
- When do branches occur?
  - Branches uncommon: 0.06% of requests saw > 1 version over 24 hours
  - Divergence occurs because of high write rate (more coordinators), not necessarily because of failure

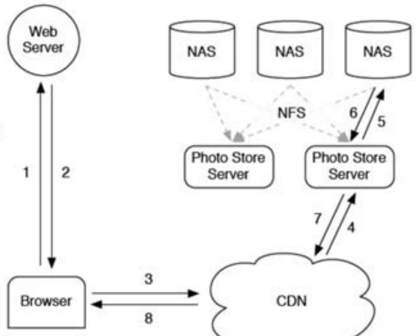
Haystack File System

- Does it ever make sense to adapt a file system to a particular usage pattern?
  - Perhaps
- Good example: Facebook's "Haystack" filesystem
  - Specific application (Photo Sharing)
    - » Large files!, Many files!
    - » 260 Billion images, 20 PetaBytes (10^{15} bytes!)
    - » One billion new photos a week (60 TeraBytes)
  - Presence of Content Delivery Network (CDN)
    - » Distributed caching and distribution network
    - » Facebook web servers return special URLs that encode requests to CDN
    - » Pay for service by bandwidth
  - Specific usage patterns:
    - » New photos accessed a lot (caching well)
    - » Old photos accessed little, but likely to be requested at any time ⇒ NEEDLES



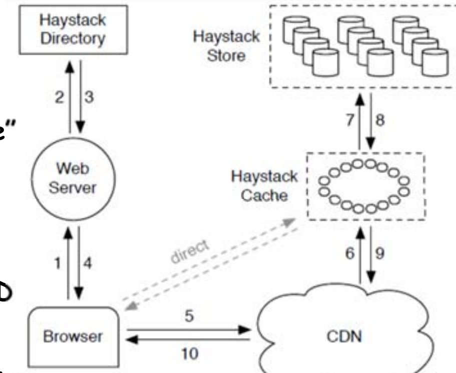
Number of photos requested in day

Old Solution: NFS



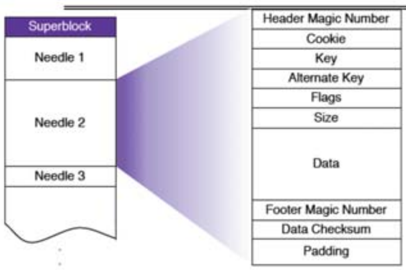
- Issues with this design?
- Long Tail ⇒ Caching does not work for most photos
  - Every access to back end storage must be *fast* without benefit of caching!
- Linear Directory scheme works badly for many photos/directory
  - Many disk operations to find even a single photo
  - Directory's block map too big to cache in memory
  - "Fixed" by reducing directory size, however still not great
- Meta-Data (FFS) requires ≥ 3 disk accesses per lookup
  - Caching all iNodes in memory might help, but iNodes are big
- Fundamentally, Photo Storage different from other storage:
  - Normal file systems fine for developers, databases, etc

New Solution: Haystack



- Finding a needle (old photo) in Haystack
- Differentiate between old and new photos
  - How? By looking at "Writeable" vs "Read-only" volumes
  - New Photos go to Writeable volumes
- Directory: Help locate photos
  - Name (URL) of photo has embedded volume and photo ID
- Let CDN or Haystack Cache Serve new photos
  - rather than forwarding them to Writeable volumes
- Haystack Store: Multiple "Physical Volumes"
  - Physical volume is large file (100 GB) which stores millions of photos
  - Data Accessed by Volume ID with offset into file
  - Since Physical Volumes are large files, use XFS which is optimized for large files

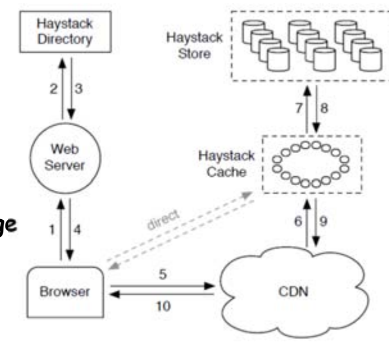
Haystack Details



Field	Explanation
Header	Magic number used for recovery
Cookie	Random number to mitigate brute force lookups
Key	64-bit photo id
Alternate key	32-bit supplemental id
Flags	Signifies deleted status
Size	Data size
Data	The actual photo data
Footer	Magic number for recovery
Data Checksum	Used to check integrity
Padding	Total needle size is aligned to 8 bytes

- Each physical volume is stored as single file in XFS
  - Superblock: General information about the volume
  - Each photo (a "needle") stored by appending to file
- Needles stored sequentially in file
  - Naming: [Volume ID, Key, Alternate Key, Cookie]
  - Cookie: random value to avoid guessing attacks
  - Key: Unique 64-bit photo ID
  - Alternate Key: four different sizes, 'n', 'a', 's', 't'
- Deleted Needle Simply marked as "deleted"
  - Overwritten Needle – new version appended at end

Haystack Details (Con't)

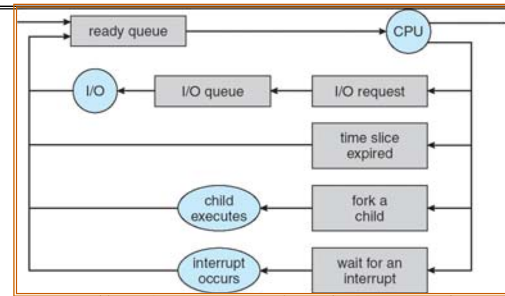


- Replication for reliability and performance:
  - Multiple physical volumes combined into logical volume
    - » Factor of 3
  - Four different sizes
    - » Thumbnails, Small, Medium, Large
- Lookup
  - User requests Webpage
  - Webserver returns URL of form:
    - » http://<CDN>/<Cache>/<Machine id>/<Logical volume,photo>
    - » Possibly reference cache only if old image
  - CDN will strip off CDN reference if missing, forward to cache
  - Cache will strip off cache reference and forward to Store
- In-memory index on Store for each volume map:
  - [Key, Alternate Key] ⇒ Offset

Administrivia

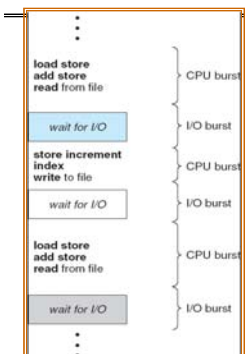
- How to specify behaviors for Scheduling Algorithm?
- One option: Statistically
  - Set up a bunch of jobs to run
  - Let them run – keep statistics about their behavior
    - » How much CPU time they get
    - » Do they ever miss deadlines
  - Sample state of scheduler regularly to make sure that statistical behavior is good
- A more detailed option: Snapshot state machine
  - We ask you to build a snapshot facility to grab state of scheduler (or any other part of kernel!) with a set of snapshot commands
    - » Think of the setup for this as a specification of a set of snapshot commands that must run
  - Then, grab results after snapshots have happened
- Export of test information?
  - /proc file system

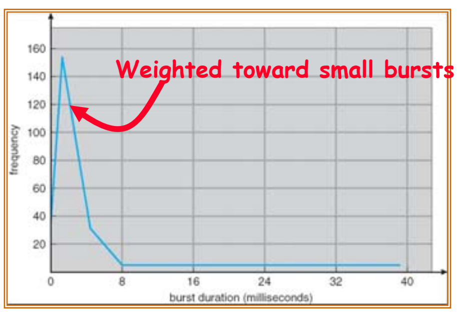
Review: CPU Scheduling



- Earlier, we talked about the life-cycle of a thread
  - Active threads work their way from Ready queue to Running to various waiting queues.
- Question: How is the OS to decide which of several tasks to take off a queue?
  - Obvious queue to worry about is ready queue
  - Others can be scheduled as well, however
- Scheduling: deciding which threads are given access to resources from moment to moment

Recall: Assumption: CPU Bursts





- Execution model: programs alternate between bursts of CPU and I/O
  - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
  - With timeslicing, thread may be forced to give up CPU before finishing current CPU burst

Scheduling Policy Goals/Criteria

- Minimize Response Time
  - Minimize elapsed time to do an operation (or job)
  - Response time is what the user sees:
    - » Time to echo a keystroke in editor
    - » 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:
    - » Minimizing response time will lead to more context switching than if you only maximized throughput
  - Two parts to maximizing throughput
    - » Minimize overhead (for example, context-switching)
    - » Efficient use of resources (CPU, disk, memory, etc)
- Fairness
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    - » Better *average* response time by making system *less* fair

Real-Time Scheduling Priorities

- Efficiency is important but **predictability** is essential
  - In RTS, performance guarantees are:
    - » Task- and/or class centric
    - » Often ensured a priori
  - In conventional systems, performance is:
    - » System oriented and often throughput oriented
    - » Post-processing (… wait and see …)
  - Real-time is about enforcing predictability, and does not equal to fast computing!!!
- Typical metrics:
  - Guarantee miss ratio = 0 (hard real-time)
  - Guarantee Probability(missed deadline) < X% (firm real-time)
  - Minimize miss ratio / maximize completion ratio (firm real-time)
  - Minimize overall tardiness; maximize overall usefulness (soft real-time)
- EDF (Earliest Deadline First), LLF (Least Laxity First), RMS (Rate-Monotonic Scheduling), DM (Deadline Monotonic Scheduling)

Recall: Round Robin (RR)

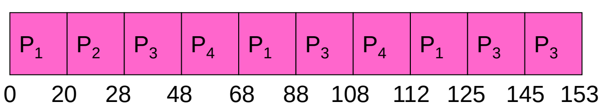
- Round Robin Scheme
  - Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds
  - After quantum expires, the process is preempted and added to the end of the ready queue.
  - n processes in ready queue and time quantum is $q \Rightarrow$
    - » Each process gets $1/n$ of the CPU time
    - » In chunks of at most q time units
    - » No process waits more than $(n-1)q$ time units
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ Interleaved (really small $\Rightarrow$ hyperthreading?)
  - q must be large with respect to context switch, otherwise overhead is too high (all overhead)
- How do you choose time slice?
  - What if too big?
    - » Response time suffers
  - What if infinite (∞)?
    - » Get back FIFO
  - What if time slice too small?
    - » Throughput suffers!

Example of RR with Time Quantum = 20

- Example:

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

  - The Gantt chart is:



  - Waiting time for $P_1=(68-20)+(112-88)=72$
    $P_2=(20-0)=20$
    $P_3=(28-0)+(88-48)+(125-108)=85$
    $P_4=(48-0)+(108-68)=88$
  - Average waiting time = $(72+20+85+88)/4=66\frac{1}{4}$
  - Average completion time = $(125+28+153+112)/4 = 104\frac{1}{2}$
- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

Comparisons between FCFS and Round Robin

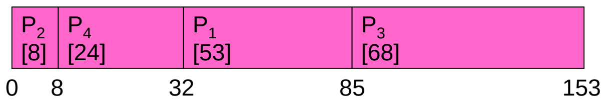
- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example: 10 jobs, each take 100s of CPU time
  RR scheduler quantum of 1s
  All jobs start at the same time
- Completion Times:

Job #	FIFO	RR
1	100	991
2	200	992
…	…	…
9	900	999
10	1000	1000

  - Both RR and FCFS finish at the same time
  - Average response time is much worse under RR!
    - » Bad when all jobs same length
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
  - Total time for RR longer even for zero-cost switch!

Earlier Example with Different Time Quantum

Best FCFS:



	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	$31\frac{1}{4}$
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	$61\frac{1}{4}$
	Q = 8	80	8	85	56	$57\frac{1}{4}$
	Q = 10	82	10	85	68	$61\frac{1}{4}$
	Q = 20	72	20	85	88	$66\frac{1}{4}$
	Worst FCFS	68	145	0	121	$83\frac{1}{2}$
Completion Time	Best FCFS	85	8	153	32	$69\frac{1}{2}$
	Q = 1	137	30	153	81	$100\frac{1}{2}$
	Q = 5	135	28	153	82	$99\frac{1}{2}$
	Q = 8	133	16	153	80	$95\frac{1}{2}$
	Q = 10	135	18	153	92	$99\frac{1}{2}$
	Q = 20	125	28	153	112	$104\frac{1}{2}$
	Worst FCFS	121	153	68	145	$121\frac{3}{4}$

Lottery Scheduling

- Yet another alternative: Lottery Scheduling
  - Give each job some number of lottery tickets
  - On each time slice, randomly pick a winning ticket
  - On average, CPU time is proportional to number of tickets given to each job
- How to assign tickets?
  - To approximate SRTF, short running jobs get more, long running jobs get fewer
  - To avoid starvation, every job gets at least one ticket (everyone makes progress)
- Advantage over strict priority scheduling: behaves gracefully as load changes
  - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses

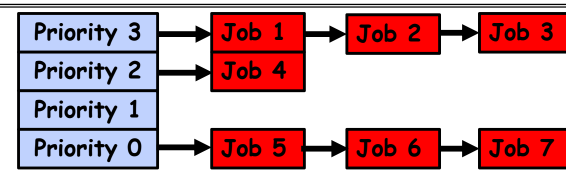
Lottery Scheduling Example

- Lottery Scheduling Example
  - Assume short jobs get 10 tickets, long jobs get 1 ticket

# short jobs/ # long jobs	% of CPU each short jobs gets	% of CPU each long jobs gets
1/1	91%	9%
0/2	N/A	50%
2/0	50%	N/A
10/1	9.9%	0.99%
1/10	50%	5%

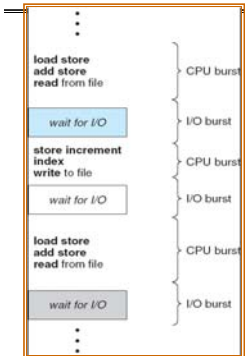
  - What if too many short jobs to give reasonable response time?
    - » If load average is 100, hard to make progress
    - » One approach: log some user out

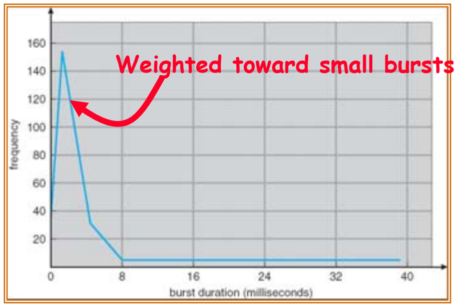
Strict Priority Scheduling



- Execution Plan
  - Always execute highest-priority runable jobs to completion
- Problems:
  - Starvation:
    - » Lower priority jobs don't get to run because higher priority tasks always running
  - Deadlock: Priority Inversion
    - » Not strictly a problem with priority scheduling, but happens when low priority task has lock needed by high-priority task
    - » Usually involves third, intermediate priority task that keeps running even though high-priority task should be running
- How to fix problems?
  - Dynamic priorities – adjust base-level priority up or down based on heuristics about interactivity, locking, burst behavior, etc…

Recall: Assumption: CPU Bursts





- Execution model: programs alternate between bursts of CPU and I/O
  - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
  - With timeslicing, thread may be forced to give up CPU before finishing current CPU burst

How to handle simultaneous mix of different types of applications?

- Can we use Burst Time (observed) to decide which application gets CPU time?
- 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, pads, and cellphones?
- 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 low(er?) priority, since they won't notice intermittent bursts from interactive apps
- Hard to characterize apps:
  - What about apps that sleep for a long time, but then compute for a long time?
  - Or, what about apps that must run under all circumstances (say periodically)

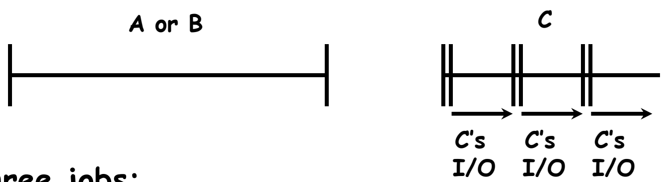
What if we Knew the Future?

- Could we always mirror best FCFS?
- Shortest Job First (SJF):
  - Run whatever job has the 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 a shorter time to completion than the remaining time on the current job, immediately preempt CPU
  - Sometimes called "Shortest Remaining Time to Completion First" (SRTCF)
- These can be applied either to a whole program or the current CPU burst of each program
  - Idea is to get short jobs out of the system
  - Big effect on short jobs, only small effect on long ones
  - Result is better average response time

Discussion

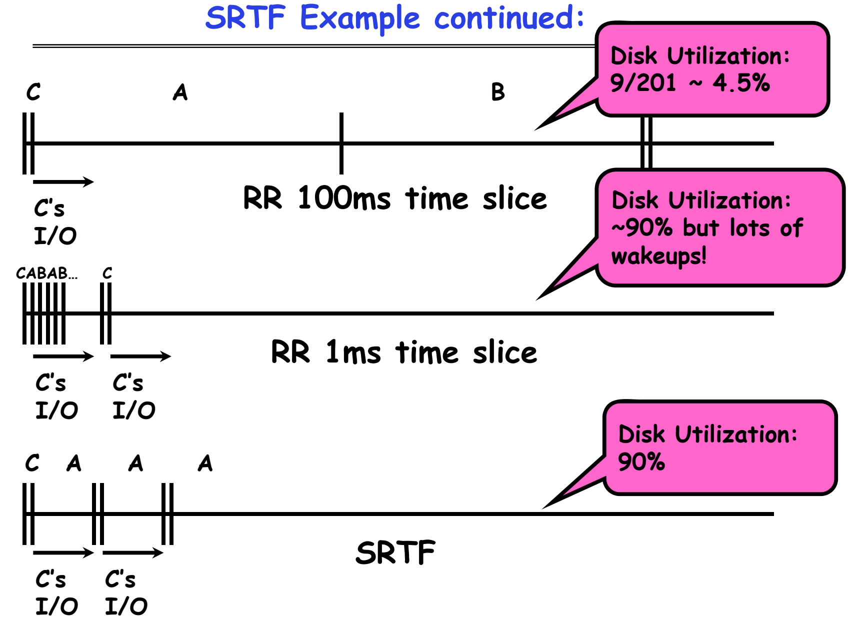
- SJF/SRTF are the best you can do at minimizing average response time
  - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
  - Since SRTF is always at least as good as SJF, focus on SRTF
- Comparison of SRTF with FCFS and RR
  - What if all jobs the same length?
    - » SRTF becomes the same as FCFS (i.e. FCFS is best can do if all jobs the same length)
  - What if jobs have varying length?
    - » SRTF (and RR): short jobs not stuck behind long ones

Example to illustrate benefits of SRTF



- Three jobs:
  - A,B: both CPU bound, run for week
    C: I/O bound, loop 1ms CPU, 9ms disk I/O
  - If only one at a time, C uses 90% of the disk, A or B could use 100% of the CPU
- With FIFO:
  - Once A or B get in, keep CPU for two weeks
- What about RR or SRTF?
  - Easier to see with a timeline

SRTF Example continued:

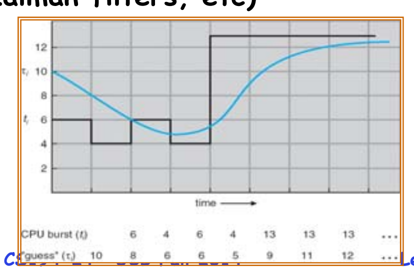


SRTF Further discussion

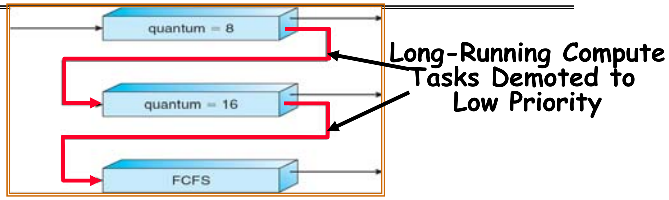
- Starvation
  - SRTF can lead to starvation if many small jobs!
  - Large jobs never get to run
- Somehow need to predict future
  - How can we do this?
  - Some systems ask the user
    - » When you submit a job, have to say how long it will take
    - » To stop cheating, system kills job if takes too long
  - But: Even non-malicious users have trouble predicting runtime of their jobs
- Bottom line, can't really know how long job will take
  - However, can use SRTF as a yardstick for measuring other policies
  - Optimal, so can't do any better
- SRTF Pros & Cons
  - Optimal (average response time) (+)
  - Hard to predict future (-)
  - Unfair (-)

Predicting the Length of the Next CPU Burst

- **Adaptive**: Changing policy based on past behavior
  - CPU scheduling, in virtual memory, in file systems, etc
  - Works because programs have predictable behavior
    - » If program was I/O bound in past, likely in future
    - » If computer behavior were random, wouldn't help
- Example: SRTF with estimated burst length
  - Use an estimator function on previous bursts:
    Let t_{n-1}, t_{n-2}, t_{n-3}, etc. be previous CPU burst lengths.
    Estimate next burst $\tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \ldots)$
  - Function f could be one of many different time series estimation schemes (Kalman filters, etc)
  - For instance,
    exponential averaging
    $\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}$
    with $(0<\alpha\leq 1)$



Multi-Level Feedback Scheduling



- Another method for exploiting past behavior
  - First used in CTSS
  - Multiple queues, each with different priority
    - » Higher priority queues often considered "foreground" tasks
  - Each queue has its own scheduling algorithm
    - » e.g. foreground – RR, background – FCFS
    - » Sometimes multiple RR priorities with quantum increasing exponentially (highest:1ms, next:2ms, next: 4ms, etc)
- Adjust each job's priority as follows (details vary)
  - Job starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn't expire, push up one level (or to top)

Scheduling Details

- 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, etc.
  - Time slice:
    - » each queue gets a certain amount of CPU time
    - » e.g., 70% to highest, 20% next, 10% lowest
- Countermeasure: user action that can foil intent of the OS designer
  - For multilevel feedback, put in a bunch of meaningless I/O to keep job's priority high
  - Of course, if everyone did this, wouldn't work!
- Example of Othello program:
  - Playing against competitor, so key was to do computing at higher priority the competitors.
    - » Put in printf's, ran much faster!

Scheduling Fairness

- What about fairness?
  - Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
    - » long running jobs may never get CPU
    - » In Multics, shut down machine, found 10-year-old job
  - Must give long-running jobs a fraction of the CPU even when there are shorter jobs to run
  - Tradeoff: fairness gained by hurting avg response time!
- How to implement fairness?
  - Could give each queue some fraction of the CPU
    - » What if one long-running job and 100 short-running ones?
    - » Like express lanes in a supermarket—sometimes express lanes get so long, get better service by going into one of the other lines
  - Could increase priority of jobs that don't get service
    - » What is done in some variants of UNIX
    - » This is ad hoc—what rate should you increase priorities?
    - » And, as system gets overloaded, no job gets CPU time, so everyone increases in priority⇒Interactive jobs suffer

Case Study: Linux O(1) Scheduler

- Priority-based scheduler: 140 priorities
  - 40 for "user tasks" (set by "nice"), 100 for "Realtime/Kernel"
  - Lower priority value ⇒ higher priority (for nice values)
  - Lower priority value ⇒ Lower priority (for realtime values)
  - All algorithms O(1)
    - » Timeslices/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:
  - The "active queue" and the "expired queue"
  - All tasks in the active queue use up their timeslices and get placed on the expired queue, after which queues swapped
  - 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

O(1) Scheduler Continued

- 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 (and vice versa)
  - Interactive Credit
    - » Earned when a task sleeps for a "long" time
    - » Spend when a task runs for a "long" time
    - » IC is used to provide hysteresis to avoid changing interactivity for temporary changes in behavior
- Real-Time Tasks
  - Always preempt non-RT tasks
  - No dynamic adjustment of priorities
  - Scheduling schemes:
    - » SCHED_FIFO: preempts other tasks, no timeslice limit
    - » SCHED_RR: preempts normal tasks, RR scheduling amongst tasks of same priority

What about Linux "Real-Time Priorities" (0-99)?

- Real-Time Tasks: Strict Priority Scheme
  - No dynamic adjustment of priorities (i.e. no heuristics)
  - Scheduling schemes: (Actually – POSIX 1.1b)
    - » SCHED_FIFO: preempts other tasks, no timeslice limit
    - » SCHED_RR: preempts normal tasks, RR scheduling amongst tasks of same priority
- With N processors:
  - Always run N highest priority tasks that are runnable
  - Rebalancing task on every transition:
    - » Where to place a task optimally on wakeup?
    - » What to do with a lower-priority task when it wakes up but is on a runqueue running a task of higher priority?
    - » What to do with a low-priority task when a higher-priority task on the same runqueue wakes up and preempts it?
    - » What to do when a task lowers its priority and causes a previously lower-priority task to have the higher priority?
  - Optimized implementation with global bit vectors to quickly identify where to place tasks
- More on this later…

Linux Completely Fair Scheduler (CFS)

- First appeared in 2.6.23, modified in 2.6.24
- "CFS doesn't track sleeping time and doesn't use heuristics to identify interactive tasks—it just makes sure every process gets a fair share of CPU within a set amount of time given the number of runnable processes on the CPU."
- Inspired by Networking "Fair Queueing"
  - Each process given their fair share of resources
  - Models an "ideal multitasking processor" in which N processes execute simultaneously as if they truly got 1/N of the processor
    - » Tries to give each process an equal fraction of the processor
  - Priorities reflected by weights such that increasing a task's priority by 1 always gives the same fractional increase in CPU time – regardless of current priority

CFS (Continued)

- Idea: track amount of "virtual time" received by each process when it is executing
  - Take real execution time, scale by weighting factor
    - » Lower priority ⇒ real time divided by greater weight
    - » Actually – multiply by sum of all weights/current weight
  - Keep virtual time advancing at same rate
- Targeted latency (T_L): period of time after which all processes get to run at least a little
  - Each process runs with quantum $\left(W_p / \sum W_i\right) \times T_L$
  - Never smaller than "minimum granularity"
- Use of Red-Black tree to hold all runnable processes as sorted on vruntime variable
  - O(log n) time to perform insertions/deletions
    - » Cash the item at far left (item with earliest vruntime)
  - When ready to schedule, grab version with smallest vruntime (which will be item at the far left).

CFS Examples

- Suppose Targeted latency = 20ms, Minimum Granularity = 1ms
- Two CPU bound tasks with same priorities
  - Both switch with 10ms
- Two CPU bound tasks separated by nice value of 5
  - One task gets 5ms, another gets 15
- 40 tasks: each gets 1ms (no longer totally fair)
- One CPU bound task, one interactive task same priority
  - While interactive task sleeps, CPU bound task runs and increments vruntime
  - When interactive task wakes up, runs immediately, since it is behind on vruntime
- Group scheduling facilities (2.6.24)
  - Can give fair fractions to groups (like a user or other mechanism for grouping processes)
  - So, two users, one starts 1 process, other starts 40, each will get 50% of CPU

Characteristics of a RTS

- Extreme reliability and safety
  - Embedded systems typically control the environment in which they operate
  - Failure to control can result in loss of life, damage to environment or economic loss
- Guaranteed response times
  - We need to be able to predict with confidence the worst case response times for systems
  - Efficiency is important but predictability is essential
    - » In RTS, performance guarantees are:
      - Task- and/or class centric
      - Often ensured a priori
    - » In conventional systems, performance is:
      - System oriented and often throughput oriented
      - Post-processing (… wait and see …)
- Soft Real-Time
  - Attempt to meet deadlines with high probability
  - Important for multimedia applications

Summary

- Scheduling: selecting a waiting process from the ready queue and allocating the CPU to it
- Round-Robin Scheduling:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs
  - Cons: Poor when jobs are same length
- Lottery Scheduling:
  - Give each thread a priority-dependent number of tokens (short tasks⇒more tokens)
  - Reserve a minimum number of tokens for every thread to ensure forward progress/fairness

Summary (Con't)

- 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 to do
  - Pros: Optimal (average response time)
  - Cons: Hard to predict future, Unfair
- Multi-Level Feedback Scheduling:
  - Multiple queues of different priorities
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF
- Linux O(1) Scheduler: Priority Scheduling with dynamic Priority boost/retraction
  - All operations O(1)
  - Fairly complex heuristics to perform dynamic priority alterations
  - Every task gets at least a little chance to run
- Realtime Schedulers: RMS, EDF, CBS
  - All attempting to provide guaranteed behavior