Recall: 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 smaller weight
  - Keep virtual time advancing at same rate among processes.
    Thus, scaling factor adjusts amount of CPU time/process
- More details
  - Weights relative to Nice-value 0
    » Relative weights~: (1.25)^nice
      » vruntime = runtime/relative_weight
  - Processes with nice-value 0 ⇒
    » vruntime advances at same rate as real time
  - Processes with higher nice value (lower priority) ⇒
    » vruntime advances at faster rate than real time
  - Processes with lower nice value (higher priority) ⇒
    » vruntime advances at slower rate than real time

Recall: EDF: Schedulability Test

Theorem (Utilization-based Schedulability Test): A task set $T_1, T_2, ..., T_n$ with $D_i = P_i$ is schedulable by the earliest deadline first (EDF) scheduling algorithm if

$$\sum_{i=1}^{n} \left( \frac{C_i}{D_i} \right) \leq 1$$

Exact schedulability test (necessary + sufficient)
Proof: [Liu and Layland, 1973]
Recall: Constant Bandwidth Server
• Intuition: give fixed share of CPU to certain of jobs
  - Good for tasks with probabilistic resource requirements
• Basic approach: Slots (called “servers”) scheduled with EDF, rather than jobs
  - CBS Server defined by two parameters: $Q_s$ and $T_s$
  - Mechanism for tracking processor usage so that no more than $Q_s$ CPU seconds used every $T_s$ seconds when there is demand. Otherwise get to use processor as you like
• Since using EDF, can mix hard-realtime and soft realtime:

What is Fair Sharing?
• $n$ users want to share a resource (e.g., CPU)
  - Solution:
    - Allocate each 1/n of the shared resource
• Generalized by max-min fairness
  - Handles if a user wants less than its fair share
    - E.g. user 1 wants no more than 20%
• Generalized by weighted max-min fairness
  - Give weights to users according to importance
    - User 1 gets weight 1, user 2 weight 2

Why is Fair Sharing Useful?
• Weighted Fair Sharing / Proportional Shares
  - User 1 gets weight 2, user 2 weight 1
• Priorities
  - Give user 1 weight 1000, user 2 weight 1
• Reservations
  - Ensure user 1 gets 10% of a resource
    - Give user 1 weight 10, sum weights ≤ 100
• Isolation Policy
  - Users cannot affect others beyond their fair share

Properties of Max-Min Fairness
• Share guarantee
  - Each user can get at least 1/n of the resource
    - But will get less if her demand is less
• Strategy-proof
  - Users are not better off by asking for more than they need
    - Users have no reason to lie
• Max-min fairness is the only “reasonable” mechanism with these two properties
Why Care about Fairness?

• Desirable properties of max-min fairness
  – Isolation policy:
    A user gets her fair share irrespective of the demands of other users
  – Flexibility separates mechanism from policy:
    Proportional sharing, priority, reservation,...

• Many schedulers use max-min fairness
  – Datacenters: Hadoop’s fair sched, capacity, Quincy
  – OS:       rr, prop sharing, lottery, linux cfs,...
  – Networking: wfq, wf2q, sfq, drr, csfq,...

When is Max-Min Fairness not Enough?

• Need to schedule multiple, heterogeneous resources
  – Example: Task scheduling in datacenters
    » Tasks consume more than just CPU - CPU, memory, disk, and I/O

• What are today’s datacenter task demands?

Heterogeneous Resource Demands

Most task need ~ <2 CPU, 2 GB RAM>
Some tasks are CPU-intensive
Some tasks are memory-intensive

2000-node Hadoop Cluster at Facebook (Oct 2010)

Problem

Single resource example
  – 1 resource: CPU
  – User 1 wants <1 CPU> per task
  – User 2 wants <3 CPU> per task

Multi-resource example
  – 2 resources: CPUs & memory
  – User 1 wants <1 CPU, 4 GB> per task
  – User 2 wants <3 CPU, 1 GB> per task
  – What is a fair allocation?
**Problem definition**

How to fairly share multiple resources when users have heterogeneous demands on them?

**Model**

- Users have *tasks* according to a *demand vector*
  - e.g. <2, 3, 1> user's tasks need 2 R1, 3 R2, 1 R3
  - Not needed in practice, can simply measure actual consumption
- Resources given in multiples of demand vectors
- Assume divisible resources

**A Natural Policy: Asset Fairness**

- **Asset Fairness**
  - Equalize each user's *sum of resource shares*

  Problem
  User 1 has < 50% of both CPUs and RAM
  Better off in a separate cluster with 50% of the resources

  - Asset fairness yields
    - U1: 15 tasks: 30 CPUs, 30 GB (Σ=60)
    - U2: 20 tasks: 20 CPUs, 40 GB (Σ=60)

  A Natural Policy: Asset Fairness

<table>
<thead>
<tr>
<th>CPU</th>
<th>User 1</th>
<th>User 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>100%</td>
<td>43%</td>
<td>43%</td>
</tr>
<tr>
<td>50%</td>
<td>28%</td>
<td>57%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RAM</th>
<th>User 1</th>
<th>User 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>43%</td>
<td>30%</td>
<td>40%</td>
</tr>
</tbody>
</table>

- Midterm I: Wednesday 3/13 (Next Wednesday!)
  - All topics up to Today's class
    - Research papers are fair game, as is material from the Love and Silbershats text books
    - 1 sheet of handwritten notes, both sides
  - Midterm details:
    - Wednesday, 3/13.
    - Here in 3106 Etcheverry
    - 4:00pm - 7:00pm
    - Extra office hours during day:
      - I'll try to be available during the afternoon for questions
Share Guarantee

- Every user should get $1/n$ of at least one resource
  - Intuition:
    - "You shouldn't be worse off than if you ran your own cluster with $1/n$ of the resources"

Desirable Fair Sharing Properties

- Many desirable properties
  - Share Guarantee
  - Strategy proofness
  - Envy-freeness
  - Pareto efficiency
  - Single-resource fairness
  - Bottleneck fairness
  - Population monotonicity
  - Resource monotonicity
  - DRF focuses on these properties

Cheating the Scheduler

- Some users will *game* the system to get more resources

- Real-life examples
  - A cloud provider had quotas on map and reduce slots
    - Some users found out that the map-quotas was low
      » Users implemented maps in the reduce slots!
  - A search company provided dedicated machines to users that could ensure certain level of utilization (e.g. 80%)
    » Users used busy-loops to inflate utilization

Two Important Properties

- Strategy-proofness
  - A user should not be able to increase her allocation by lying about her demand vector
  - Intuition:
    » Users are incentivized to make truthful resource requirements

- Envy-freeness
  - No user would ever strictly prefer another user's lot in an allocation
  - Intuition:
    » Don't want to trade places with any other user
Challenge

- A fair sharing policy that provides
  - Strategy-proofness
  - Share guarantee

- Max-min fairness for a single resource had these properties
  - Generalize max-min fairness to multiple resources

Dominant Resource Fairness

- A user's *dominant resource* is the resource she has the biggest share of
  - Example:
    - Total resources: <10 CPU, 4 GB>
    - User 1's allocation: <2 CPU, 1 GB>
    - Dominant resource is memory as 1/4 > 2/10 (1/5)

- A user's *dominant share* is the fraction of the dominant resource she is allocated
  - User 1's dominant share is 25% (1/4)

Dominant Resource Fairness (2)

- Apply max-min fairness to dominant shares
- Equalize the dominant share of the users
  - Example:
    - Total resources: <9 CPU, 18 GB>
    - User 1 demand: <1 CPU, 4 GB> dominant res: mem
    - User 2 demand: <3 CPU, 1 GB> dominant res: CPU

<table>
<thead>
<tr>
<th>CPU</th>
<th>mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>3 CPUs</td>
</tr>
<tr>
<td>User 2</td>
<td>6 CPUs</td>
</tr>
</tbody>
</table>

  66% | 66%

  50% | 50%

  0% | 0%

DRF is Fair

- DRF is strategy-proof
- DRF satisfies the share guarantee
- DRF allocations are envy-free

See DRF paper for proofs
**Online DRF Scheduler**

Whenever there are available resources and tasks to run: 
*Schedule a task to the user with smallest dominant share*

- \(O(\log n)\) time per decision using binary heaps
- Need to determine demand vectors

**Determining Demand Vectors**

- They can be *measured*
  - Look at actual resource consumption of a user
- They can be *provided* by the user
  - What is done today
- In both cases, strategy-proofness incentivizes user to consume resources wisely

**Alternative: Use an Economic Model**

- Approach
  - Set *prices* for each good
  - Let users buy what they want
- How do we determine the right prices for different goods?
- Let the market determine the prices
- *Competitive Equilibrium from Equal Incomes (CEEI)*
  - Give each user \(1/n\) of every resource
  - Let users trade in a perfectly competitive market
- Not strategy-proof!

**DRF vs CEEI**

- User 1: \(<1\) CPU, \(4\) GB>  User 2: \(<3\) CPU, \(1\) GB>
  - DRF more fair, CEEI better utilization
- User 1: \(<1\) CPU, \(4\) GB>  User 2: \(<3\) CPU, \(2\) GB>
  - User 2 increased her share of both CPU and memory
Example of DRF vs Asset vs CEEI

- Resources <1000 CPUs, 1000 GB>
- 2 users A: <2 CPU, 3 GB> and B: <5 CPU, 1 GB>

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>User A</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>User B</td>
<td>50%</td>
<td>50%</td>
</tr>
<tr>
<td>a) DRF</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>b) Asset Fairness</td>
<td>50%</td>
<td>50%</td>
</tr>
<tr>
<td>c) CEEI</td>
<td>100%</td>
<td>0%</td>
</tr>
</tbody>
</table>

The Swarm of Resources

- What system structure required to support Swarm?
  - Discover and Manage resource
  - Integrate sensors, portable devices, cloud components
  - Guarantee responsiveness, real-time behavior, throughput
  - Self-adapting to adjust for failure and performance predictability
  - Uniformly secure, durable, available data

Support for Applications

- Clearly, new Swarm applications will contain:
  - Direct interaction with Swarm and Cloud services
    - Potentially extensive use of remote services
    - Serious security/data vulnerability concerns
  - Real Time requirements
    - Sophisticated multimedia interactions
    - Control of/interaction with health-related devices
  - Responsiveness Requirements
    - Provide a good interactive experience to users
  - Explicitly parallel components
    - However, parallelism may be “hard won” (not embarrassingly parallel)
    - Must not interfere with this parallelism
- What support do we need for new Swarm applications?
  - No existing OS handles all of the above patterns well!
    - A lot of functionality, hard to experiment with, possibly fragile, ...
    - Monolithic resource allocation, scheduling, memory management...
  - Need focus on resources, asynchrony, composability

Resource Allocation must be Adaptive

- Three applications:
  - 2 Real Time apps (RayTrace, Swaptions)
  - 1 High-throughput App (Fluidanimate)
Guaranteeing Resources

- **Guaranteed Resources** ⇒ **Stable software components**
  - Good overall behavior
- **What might we want to guarantee?**
  - Physical memory pages
  - BW (say data committed to Cloud Storage)
  - Requests/Unit time (DB service)
  - Latency to Response (Deadline scheduling)
  - Total energy/battery power available to Cell
- **What does it mean to have guaranteed resources?**
  - Firm Guarantee (with high confidence, maximum deviation, etc)
  - A Service Level Agreement (SLA)?
  - Something else?
- **“Impedance-mismatch” problem**
  - The SLA guarantees properties that programmer/user wants
  - The resources required to satisfy SLA are not things that programmer/user really understands

New Abstraction: the Cell

- **Properties of a Cell**
  - A user-level software component with guaranteed resources
  - Has full control over resources it owns (“Bare Metal”)
  - Contains at least one memory protection domain (possibly more)
  - Contains a set of secured channel endpoints to other Cells
  - Hardware-enforced security context to protect the privacy of information and decrypt information (a Hardware TCB)
- **Each Cell schedules its resources exclusively with application-specific user-level schedulers**
  - Gang-scheduled hardware thread resources (“Harts”)
  - Virtual Memory mapping and paging
  - Storage and Communication resources
    - Cache partitions, memory bandwidth, power
    - Use of Guaranteed fractions of system services
- **Predictability of Behavior** ⇒
  - Ability to model performance vs resources
  - Ability for user-level schedulers to better provide QoS

Applications are Interconnected Graphs of Services

- **Component-based model of computation**
  - Applications consist of interacting components
  - Explicitly asynchronous/non-blocking
  - Components may be local or remote
- **Communication defines Security Model**
  - Channels are points at which data may be compromised
  - Channels define points for QoS constraints
- **Naming (Brokering) process for initiating endpoints**
  - Need to find compatible remote services
  - Continuous adaptation: links changing over time!

Impact on the Programmer

- **Connected graph of Cells** ⇒ **Object-Oriented Programming**
  - Lowest-Impact: Wrap a functional interface around channel
    - Cells hold “Objects”, Secure channels carry RPCs for “method calls”
    - Example: POSIX shim library calling shared service Cells
  - Greater Parallelism: Event triggered programming
- **Shared services complicate resource isolation:**
  - How to guarantee that each client gets guaranteed fraction of service?
  - Distributed resource attribution (application as distributed graph)
  - Must somehow request the right number of resources
    - Analytically? AdHoc Profiling? Over commitment of resources?
    - Clearly doesn’t make it easy to adapt to changes in environment
Allocation of Resources
Discovery, Distribution, and Adaptation

Two Level Scheduling: Control vs Data Plane

- Split monolithic scheduling into two pieces:
  - Course-Grained Resource Allocation and Distribution to Cells
    - Chunks of resources (CPUs, Memory Bandwidth, QoS to Services)
    - Ultimately a hierarchical process negotiated with service providers
  - Fine-Grained (User-Level) Application-Specific Scheduling
    - Applications allowed to utilize their resources in any way they see fit
    - Performance Isolation: Other components of the system cannot interfere with Cells use of resources

Adaptive Resource-Centric Computing (ARCC)

- Goal: Meet the QoS requirements of a software component (Cell)
  - Behavior tracked through application-specific "heartbeats" and system-level monitoring
  - Dynamic exploration of performance space to find operation points
- Complications:
  - Many cells with conflicting requirements
  - Finite Resources
  - Hierarchy of resource ownership
  - Context-dependent resource availability
  - Stability, Efficiency, Rate of Convergence, ...

Resource Allocation
Tackling Multiple Requirements: Express as Convex Optimization Problem

Continuously minimize using the penalty of the system

(subject to restrictions on the total amount of resources)

Resource-Value Function

Penalty Function

Penalty \_1

Penalty \_2

Runtime \_1

Runtime \_2

Graph

Stencil

Space-Time Partitioning \xrightarrow{\text{Cell}}

Spatial Partition: Performance isolation
- Each partition receives a vector of basic resources
  » A number HW threads
  » Chunk of physical memory
  » A portion of shared cache
  » A fraction of memory BW
  » Shared fractions of services

Partitioning varies over time
- Fine-grained multiplexing and guarantee of resources
  » Resources are gang-scheduled
- Controlled multiplexing, not uncontrolled virtualization
- Partitioning adapted to the system's needs

Efficient Space-Time Partitioning

Communication-Avoiding Gang Scheduling

- Supports a variety of Cell types with low overhead
  - Cross between EDF (Earliest Deadline First) and CBS (Constant Bandwidth Server)
  - Multiplexers do not communicate because they use synchronized clocks with sufficiently high precision

Brokering Service: The Hierarchy of Ownership

- Discover Resources in “Domain”
  - Devices, Services, Other Brokers
  - Resources self-describing?
- Allocate and Distribute Resources to Cells that need them
  - Solve Impedance-mismatch problem
  - Dynamically optimize execution
  - Hand out Service-Level Agreements (SLAs) to Cells
  - Deny admission to Cells when violates existing agreements
- Complete hierarchy
  - Throughout world graph of applications
Adaptive, Second-Level Preemptive Scheduling Framework

- **PULSE**: Preemptive User-Level Scheduling Framework for adaptive, preemptive schedulers:
  - Dedicated access to processor resources
  - Timer Callback and Event Delivery
  - User-level virtual memory mapping
  - User-level device control
- **Auxiliary Scheduler**:
  - Interface with policy service
  - Runs outstanding scheduler contexts past synchronization points when resizing happens
  - 2nd-level Schedulers not aware of existence of the Auxiliary Scheduler, but receive resize events
- A number of adaptive schedulers have already been built:
  - Round-Robin, EDF, CBS, Speed Balancing

Example: Tessellation GUI Service

- Operate on user-meaningful “actions”
  - E.g. “draw frame”, “move window”
- Service time guarantees (soft real-time)
  - Differentiated service per application
  - E.g. text editor vs video
- Performance isolation from other applications

Feedback Driven Policies

- Simple Policies do well but online exploration can cause oscillations in performance
- Example: Video Player interaction with Network
  - Server or GUI changes between high and low bit rate
  - Goal: set guaranteed network rate:
- Alternative: Application Driven Policy
  - Static models
  - Let network choose when to decrease allocation
  - Application-informed metrics such as needed BW
Summary

- **DRF provides multiple-resource fairness in the presence of heterogeneous demand**
  - First generalization of max-min fairness to multiple-resources
  - DRF’s properties
    » Share guarantee, at least 1/n of one resource
    » Strategy-proofness, lying can only hurt you
    » Performs better than current approaches
- **Adaptive Resource-Centric Computing**
  - Use of Resources negotiated hierarchically
  - Underlying Execution environment guarantees QoS
  - New Resources constructed from Old ones:
    » Aggregate resources in combination with QoS-Aware Scheduler
    » Result is a new resource that can be negotiated for
  - Continual adaptation and optimization
- **Components of future OS environment**
  - Cells as Basic Unit of Resource and Security
    » User-Level Software Component with Guaranteed Resources
    » Secure Channels to other Cells
  - Observation, Monitoring, and Adaptation layers
    » Machine learning, Convex Optimization

3/11/13