CMPSCI 754 Multimedia Systems **Spring 2001**

Lecture 13: 27th March '01

Lecturer: Prashant Shenoy *Scribe: Vijay Sundaram*

Today's Lecture - Case Study: Integrated File System

13.1 Need for Integrated File Systems

Different data types have significantly different characteristics:

- **Textual data:** requests are aperiodic in nature, have small reads/writes, and are non real-time i.e., they do not have deadlines associated with them.
- **Continuous Media (CM):** requests are periodic in nature, have sequential access patterns and large data rates and also have deadlines associated with them.

Most existing file systems are optimized for a single data type.

- **Conventional file systems** (e.g UNIX) are optimized for text i.e., they treat all requests as belonging to the same class (which can be thought of as a best-effort class of service).
- **CM servers** are however optimized for serving audio and video files.

In the current scenario, diverse applications (text and real-time) co-exist on a single operating system. So, there is a need to develop an integrated file system which can efficiently support multiple data types.

13.2 Design Philosophy

13.2.1 Logical Integration

- Use a separate file system for each data type, and design an integration later for providing unified access.
- **Advantages:**
  - Does not require any modification to the individual file servers (i.e., the text and CM file servers etc.) and hence simplifies implementation.
  - Static partitioning of resources, hence no interference (one class does not influence/constrain the performance of the other data-type class).
- **Disadvantages:** Under-utilization of resources due to the
  - Difficulty in altering the resource partitioning dynamically.
  - Mismatch in the bandwidth and space requirements of objects.

13.2.2 Physical Integration

- Share storage resources among all the data types.
- **Advantages:**
  - Co-locating objects with different bandwidth-space requirements leads to higher utilization.
  - On-demand resource allocation facilitates in easily accommodating dynamic changes in access patterns.
- **Disadvantages:**
  - Interference among service classes.

13.3 Integrated File System Design: Fundamentals

- File system manages three resources
  - Disk bandwidth
  - Storage space
  - Buffer cache
- Resource management techniques
  - Disk bandwidth: disk scheduling algorithms
  - Storage space: placement and failure recovery
  - Buffer cache: caching techniques

13.3.1 Managing Disk Bandwidth

- Applications have different service requirements
  - *Interactive applications* such as editors and compilers require low average response times.
  - *Throughput intensive applications* such as ftp and http servers require the file system to sustain high throughput across multiple requests but are less concerned about the response time of individual requests.
  - *Real-time applications* such as virtual reality and streaming media require the operating system to provide performance guarantees in the form of deadlines.
- Existing disk scheduling algorithms are optimized for a single class of service
  - *Best-effort:* SCAN, SATF, .. (minimize seek and access time respectively)
  - *Real-time:* SSEDO, SCAN-EDF, FD-SCAN, ..(oriented to meet the deadlines associated with requests).
- **Requirements:**
  - Export multiple classes of service (real-time, interactive, ..).
  - Disk schedulers that align the service provide with the application needs.

13.3.2 Managing Storage Space

- Placement policies are governed by *data characteristics* and *application requirements.*
- **Examples:**
  - CM applications: large data rates
    - large block size ($64 - 128\ KB$) desirable.
  - Conventional applications: small object sizes
    - small block size ($4 - 8\ KB$) desirable.
- A single placement policy would be unsuitable for all data types.
- **Requirements:**
  - Should support multiple data type specific placement policies.
  - Should employ mechanisms to enable their co-existence.

13.3.3 Managing Buffers

- Caching policies are governed by access characteristics.
- **Examples:**
  - *CM applications*: sequential access
    - LRU ineffective, Interval Caching suitable.
  - *Conventional applications:* locality of reference
    - LRU caching policy suitable.
- Again, no single policy is suitable for all the application classes.
- **Requirements:**
  - Should support multiple data type specific caching policies
  - Should employ mechanisms to enable sharing of the buffer cache

13.4 Architectural Requirements: Summary

- **Key Principle:** "One size does not fit all"
- Support multiple data type specific policies
  - Should support multiple classes of service.
  - Align the service provided with the application needs.
  - Support multiple policies for placement and caching.
  - Employ mechanisms to enable their co-existence.
- Efficiently support present and future applications.
- File system must be extensible and customizable.

13.5 Symphony Architecture

- Case study: *Symphony* an integrated file system.
- Symphony has a *layered architecture*, consisting of two layers:
  - Data type independent layer: which consists of mechanisms that implement core file system functionality such as disk scheduling, buffer management etc.
  - Data type specific layer: which consists of modules that address the data type specific placement, retrieval and caching policies etc.

Note : remember that the file system manages three resources. the storage space, the disk bandwidth and the buffer cache.

13.6 Symphony: Data Type Independent Layer

The data type independent layer comprises of *three managers* to manage the three resources managed by the file system, namely the storage space, the disk bandwidth and the buffer cache.

- **Storage Manager:** Manages the placement of data and meta-data (meta-data refers to the *data about the data*) blocks on disk, and also addresses fault tolerance issues.
- **Bandwidth Manager:** Manages the retrieval architecture (client pull or server push), the disk bandwidth and also handles resource reservation (resource reservation is an optional feature).
- **Buffer Manager:** Responsible for managing the buffer cache.

We will now discuss in detail the *disk scheduling framework* of the *bandwidth manager* and then will go to give a high level picture of the storage manager and the buffer manager.

13.6.1 Bandwidth Manager: Disk Scheduling Issues

- **Requirement:** (discussed before) Integrated file systems must support multiple applications with diverse performance requirements (e.g., interactive best effort applications such as editors and compilers, soft real time applications such as virtual reality and video players etc.)
- **Existing disk scheduling algorithms** are optimized for a single class of applications. For e.g., best-effort disk scheduling algorithms such as SCAN. SATF schedule requests based on relative positions on disk and aim at minimizing the seek and rotational latency overheads, so these do not meet the requirements of requests with deadlines. Real-time disk scheduling algorithms (e.g., SSEDO, SCAN-EDF) start from an EDF schedule and then reorder requests (with the same deadline) to reduce seek and rotational latency overheads.
- **Existing Disk Scheduling Algorithms: Limitations**
  - Poor performance for mixed workloads. For e.g., SCAN does not take the requirements (e.g., deadlines) of requests into account while scheduling the requests and hence is unsuitable for real-time applications. (slide 124 shows the graph for increased deadline violations for a real-time application with increasing number of text clients i.e., # text clients $\uparrow \Rightarrow$ %tage deadline violation real-time clients $\uparrow$).

  - Using a priority based approach can however lead to starvation (for. e.g if you priortize the real time requests and there is a constant stream of real time requests that keep arriving at the disk, you might starve some best-effort text request etc.)

- **Supporting Multiple Application Classes**
  - *Coarse-grain time slicing:* this involves partitioning the bandwidth among classes by allocating consecutive slots (which may differ in size) to different application classes, and using different scheduling policies in these consecutive slots according to the requirements of the application class being serviced in that slot. The waiting time of a request increases as it has to wait for the time slot corresponding to it's application class before it gets scheduled. So, this results in worse response time. (graph in slide 125 depicts this behavior, i.e. time slice $\uparrow \Rightarrow$ response time $\uparrow$)
  - *Fine-grain time slicing: hierarchical schedulers:* Hierarchical schedulers exist for fine grain scheduling of CPU and network bandwidth. However, uisng a fine time slice does not work well for partitioning disk bandwidth as the overhead of switching between classes increases (potentiall larger seeks, and in the worst case only one request served from a class and you keep switching between classes arbitrarily). Fine grain scheduling works for CPU/network bandwidth scheduling as the switch time is fixed, but this differs for disk bandwidth (i.e., the disk resource is a different kind) and the switch overhead (between two classes) depends on the disk head position.

- **Why is Servicing Multiple Classes Hard?**
  - As mentioned above the disk bandwidth is fundamentally different from other system resources. It has a variable context switch time (seek and rotational latency) and to maximize the throughput one should reduce the latency overheads.
  - Fine grain allocation leads to large latency overheads (i.e., as time slice $\downarrow$ latency overhead $\uparrow$, see graph on slide 126).
  - So, neither fine grain allocation, nor coarse grain allocation works fine in all situations.

- **Disk Scheduling Requirements**
  - Export multiple classes of service (interactive best-effort, real time, ..).
  - Align the service provided with application needs.
  - Reduce seek time and rotational latency overheads.
  - Protect application classes from each other.
  - Computationally efficient.

Next we describe ***Cello*** which is disk scheduling framework employed by Symphony and meets the disk scheduling requirements outlined above.

13.6.2 Bandwidth Manager: Cello

13.6.2.1 Cello: Key Principles

The unique feature of the *Cello* disk scheduling framework is that it allocates bandwidth at two time scales

- A *coarse time-scale bandwidth allocation*, which allocates a certain fraction of disk bandwidth to each class (decides how much of the share a class should receive).

- A *fine time-scale bandwidth allocation*, that determines a schedule which achieves the goals of reducing the latency overheads (seek time and rotational latency) and helps align the service to the application needs.

Note that the length of alternate time chunks can vary dynamically (depending on the number of requests in an application class, the weight allocated to application class etc.).

13.6.2.2 Cello Disk Scheduling Framework

- Cello has a fixed two level scheduling hierarchy. The top level in the scheduling hierarchy consists of class specific pending queues, and requests coming in are first queued up in the pending queue corresponding to the application class of the request. In every interval, these requests are then moved to the pending queue which then sends in the requests to the disk. Next we describe the functions of the schedulers at the two levels of the hierarchy.
- The ***class-independent scheduler*** performs two functions:
  - First, it determines when and how many requests from each class should be inserted in the scheduled queue.
  - Second, it exports an interface that enables the class-specific schedulers to determine where to insert requests in the scheduled queue.
- The ***class-specific schedulers*** perform the following functions:
  - They order the requests in their pending queue in accordance with the application requirements.
  - When invoked by the class-independent scheduler, they determine the position for inserting the selected request into the scheduled queue.
  - Thus, they create a schedule that meets the request needs and provides service differentiation between application classes.

Thus, this two level scheduling hierarchy aims at meeting the *disk-scheduling requirements* outlined in Section 13.6.1. Next, we look at the class-independent and the class-specific schedulers in some detail.

13.6.2.3 Class-Independent Scheduler

- Assign weights to each class: the bandwidth is then allocated to the application class in proportion to it's weight. The weighted allocation could be *by time* or *by byte*. In *proportional-time allocation* the fraction of an interval for which the disks services requests form an application class is proportional to its weight, whereas in *proportional-byte allocation* the amount of data accessed in an interval for an application class is proprotional to its weight.
- Algorithm: (from class slides)
  - Invoke a class specific scheduler for a request.
  - Insert a request at a specified location if
    - class has sufficient unused allocation.
    - total used allocation $\leq$ interval length.
  - Update used allocation.
  - Reallocate unused allocation to classes with pending requests (work conserving fair scheduler).

13.6.2.4 Class-Specific Schedulers

- Determine the insertion position based on the requirements of requests (e.g., deadlines, or low average response time etc.) and the state of the scheduled queue.
- Cello supports three application classes, an *interactive best-effort* class for requests desiring a low average response time, a *throughput-intensive best-effort* class for requests desiring high throughput, and a *real-time* class for requests with deadlines.
- The class-specific scheduler corresponding to the *interactive best-effort* class inserts requests using slack-stealing (examine the scheduled queue and look for a slack, if you see a slack insert the best-effort request)
- The class-specific scheduler corresponding to the *throughput-intensive best-effort* class inserts the requests at the tail of the scheduled queue in SCAN order.
- The class-specific scheduler corresponding to the *real-time* class inserts the requests in the SCAN-EDF order.

13.6.2.5 Cello: Performance Evaluation

- The graphs in *slides 133 - 137* show the performance of the *Cello disk scheduling framework*.
- *Slide 133* shows how Cello
  - protects application classes from each other or in other words, how it achieves application isolation . The first graph shows how increasing the number of video clients has little impact on the response time of the *interactive best-effort* text clients, and the second graph shows the isolation of a *real-time* video client from text clients. So, cello isolates one application class from another and thus an increasing number of clients in one class does not drastically constrain or influence the performance of another application class. The graph also shows how SCAN performs worse, with increased response time for the *interactive best-effort* text clients and increase in deadline violations for the *real-time* video clients.
  - Aligns the service to the application class. Thus, the first graph shows how Cello attains low average response time for *interactive best-effort* text clients and meets deadline requirements for the *real-time* video clients. In each case the performance of the SCAN scheduling algorithm is much worse then Cello.
- *Slide 134* shows that Cello attains better response times for the *interactive best-effort* requests as compared to the *throughput-intensive best-effort* requests.
- *Slide 135* depicts the *fair work-conserving* nature of Cello. The interactive and the real time class were given a weight of 1 each. In the time range $200 - 400s$ there were no real-time requests and the graph shows how the entire disk bandwidth is taken up by the interactive class.
- *Slide 136* shows the predicatble allocation achieved by Cello. All three classes (the interactive-best effirt class, the throughput-intensive class and the real-time class) were given a weight of 1 each. The graph on the left shows that the time allocated to each class as a function of time. The fraction allocated to each class is close to 0.33 which is in accordance to the weight allocated to the application classes ($\frac{1}{1+1+1} = \frac{1}{3} = 0.33$). The graph on the right shows how on the other hand that when we use SCAN the time allocated to the application class depicts no steady behavior and varies randomly in each interval.
- *Slide 137* shows that the seek and rotational latency incurred are comparable to that of SCAN even though we use a more sophisticated scheduler (note that SCAN simply optimizes the seek overheads).
- The table in *slide 137* shows how Cello is also computationally efficient when compared to SCAN in terms of the average and worst case insertion time of requests.

13.6.3 Storage Manager

- Objective: Enable the co-existence of multiple stripe unit sizes, degrees of striping and striping policies.
- Multiple stripe unit sizes
  - Define a block size at file system creation time.
  - Construct a stripe unit using a sequence of continuous base blocks (note that this allocation of continuous base blocks is intelligent, also zones could be used to reduce fragmentation)
- Multiple degrees of striping and striping policies
  - Provide control over placement of blocks using location hints (but no guarantees).
  - Export the presence of multiple disks.

In the next class we'll look at the *fault tolerance* achieved by the *storage manager*, the *resource reservation* option of the *bandwidth manager* and will then give an overview of the *buffer manager*. Then we'll go on to describe the *data type specific layer* in the *symphony two-layer architecture*.