CSE451 I/O Systems and the Full I/O Path Spring 2001

Gary Kimura
Lecture #26
May 23, 2001

Today's Topics

- The I/O subsystem
- Disk partitions, volume management, and RAID
- Goal is to cover the full I/O code path, but we still need a few more important items
  – Object management
  – Worker queues and other asynchronous worker threads
- Now the full I/O code path for an open, read/write and close. i.e., from the user API down to the disk and back again with a few stops in between

I/O Subsystem

- Some hardware features
  – Polling versus as a means of controlling devices
  – Direct Memory Access hardware to transfer data directly to and from main memory
- Features of the I/O programming paradigm
  – Character-stream versus block I/O
  – Sequential versus random access
  – Synchronous versus asynchronous I/O
- Implementation considerations in the kernel
  – Scheduling the I/O
  – Buffered versus non-buffered I/O (and who's buffer do we use)
  – Software Caching

Disk partitions and volume management

- The disk driver's job is to essentially present to the rest of the OS a virtual disk drive or volume.
  – In the simplest case the volume matches one for one with the actual hardware drives.
  – Or the driver can divide a single drive into multiple volumes (or partitions)
  – Or the driver can build a larger volume from multiple disks
- Look at WinDisk example
- Each volume is its own wholly contained file system structure
- Using multiple disks we can also improve reliability and performance and that is where RAID comes in

RAID
The Basic Problem

- Disks are improving, but a lot slower than CPUs
- We can use multiple disks for improving performance
  - By striping files across multiple disks (placing parts of each file on a different disk), we can use parallel I/O to improve access time
- Striping reduces reliability -- 100 disks have 1/100th the MTBF (mean time between failures) of one disk
- So, we need striping for performance, but we need something to help with reliability / availability
- To improve reliability, we can add redundant data to the disks, in addition to striping

RAID

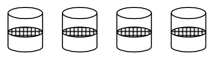
- A RAID is a Redundant Array of Inexpensive Disks
- Disks are small and cheap, so it's easy to put lots of disks (10s to 100s) in one box for increased storage, performance, and availability
- Data plus some redundant information is striped across the disks in some way
- How that striping is done is key to performance and reliability.

Some Raid Issues

- Granularity
  - fine-grained: stripe each file over all disks. This gives high thruput for the file, but limits to transfer of 1 file at a time
  - course-grained: stripe each file over only a few disks. This limits thruput for 1 file but allows more parallel file access
- Redundancy
  - uniformly distribute redundancy info on disks: avoids load-balancing problems
  - concentrate redundancy info on a small number of disks: partition the set into data disks and redundant disks

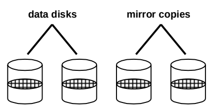
Raid Level 0

- Level 0 is nonredundant disk array
- Files are striped across disks, no redundant info
- High read thruput
- Best write thruput (no redundant info to write)
- Any disk failure results in data loss



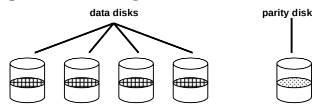
Raid Level 1

- Mirrored Disks
- Data is written to two places
- On failure, just use surviving disk
- On read, choose fastest to read



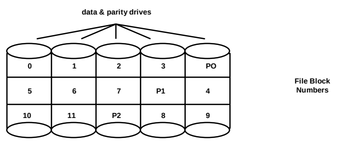
Raid Levels 2 and 3

- Use ECC (error correcting code) or Parity disks
- E.G., each byte on the parity disk is a parity function of the corresponding bytes on all the other disks
- A read accesses all the data disks
- A write accesses all data disks plus the parity disk
- On disk failure, read remaining disks plus parity disk to compute the missing data



Level 5

- Block Interleaved Distributed Parity
- Like parity scheme, but distribute the parity info over all disks (as well as data over all disks)
- Better read performance, large write performance



Now Back to Window NT

- Our goal is to dissect (as much as we can) the complete code path for doing I/O in the system. But we still need two more items
- The Object Manager
- File System worker threads

Object Management in Windows NT

- The Object Manager in Windows NT is used throughout the system as a tool for creating, referencing (using ref counts), accessing, and destroying kernel mode objects.
- These objects include, semaphores, events, file, memory sections, etc.
- In kernel mode some of the calls are ObCreateObject, ObOpenObject, ObReferenceObject, ObDereferenceObject, and NtClose to manipulate objects through handles, names or pointers.
- User mode code isn't allowed to point directly to the object instead user's get a handle that the Object Manager translates into an object that the kernel can reference and use
- The Object Manager defines a tree-like name space for all objects (look at objdir, winobj, and objmon)

Worker Threads

- For asynchronous I/O operations there is a pool of worker threads available to complete the work
- Using the worker thread paradigm it is easy to program up multiple asynchronous operations
- Everything isn't quite that simple for I/O.
  – There are some issues with capturing and returning data back to the user

The Full I/O Path for CreateFile

- CreateFile is a user mode routine that packages everything up for the kernel API call to NtCreateFile in the I/O system
- NtCreateFile calls Object Manager to create a new file object
- The Object Manager resolves the name to a device with an associated create function that it calls (this gets us back into I/O system but now we know the volume it is destined for)
- The I/O system allocates an I/O Request Packet (IRP) which contains all the information needed to process the request and calls the file system passing down the IRP.
- The file system can return to the I/O system at anytime. This behavior depends on if the user asked for a synchronous or asynchronous operation
- When it is done processing the request the system completes the IRP

Completing an I/O Request

- When the file system is done with the request is calls IoCompleteRequest passing in the IRP.
- IoCompleteRequest asynchronously completes the request back to the user
- The user can actually be in a few states
  – The user thread itself could be the one calling IoCompleteRequest
  – The user thread could be waiting on the IRP to finish
  – The user thread could be off doing something else.
- In the case where another thread finishes the IRP then thread needs to attach to the user address space to complete the request

The IRP

- IRPs are used for all I/O in the system to represent communication between the devices, file systems, and the I/O system.
- Each IRP contains a stack of operations and can be reissued multiple times before completing the original request
- For example, A CreateFile might need to read and write the disk to complete its work. Instead of allocating a new IRP to talk to the device driver the file system can use the original CreateFile IRP.

Fast I/O

- The fast read/write path uses memory mapped files

- ReadFile like CreateFile is a user mode routine that calls down to NtReadFile.
- NtReadFile uses the Object Manager to translate the user handle into a file object.
- It then calls a fast I/O function for the file object to see if the data is cached and can be read through a fast path. If so then the I/O is completed without allocating an IRP
- If the data is not cached then the I/O system allocates ad IRP and calls down to the file system
- As part of doing the read the file system will fill the cache managers read buffer.

Things to come

- Accounting, protection and security
- Distributed Systems and RPC
- Take a deep breath and final exam day