GPFS Troubleshooting

Yuri Volobuev
Ex-Spectrum Scale Development

April 2017

What can possibly go wrong? A lot…

- **From the end user point of view:**
  - IO fails (“Stale NFS file handle”)
  - IO hangs (“ls is stuck! df is stuck too!”)
  - File system won’t mount
  - IO performance is below sewer level
  - IO results are wrong
- **From the sysadmin point of view:**
  - Kernel crash
  - mmfsd crash
  - File system corruption
  - User data corruption
  - GPFS deadlock
  - Kernel deadlock
  - Quorum loss/node expel
  - Etc…

Debug data

Where to start looking?

- **There's no single place**
  - Many separate pieces of data need to be considered, depending on the nature of the problem:
  - GPFS log (/var/adm/ras/mmfs.log.*)
  - OS log (syslog on Linux, errpt on AIX, Event Viewer on Windows, someday)
  - GPFS state (various mm* command output)
  - GPFS traces
  - Kernel dumps (vmcore files)
  - Process core files
  - Strace/truss output
  - Specialized disk and network logs
  - Application logs
  - Dead Sea scrolls

How do I react to a GPFS problem?

- **“If in doubt, collect dumps and traces. Then repeat.”**
- **The first step upon a problem discovery is to run a debug data collection script**
  - gpfs.snap on customer installs
    - If a deadlock is suspected, run “gpfs.snap –deadlock”
  - Using a custom debug data collection script should be avoided as much as possible, as it very often leads to critical pieces of debug data being missed. Don’t underestimate how complex this procedure is.
- **Leaving a system “to be looked at live” is less work, but is only a good idea in a few specific cases**
  - It takes time to work GPFS problems. Keeping a live system in a bad state should be an exception, not a rule

What exactly is “debug data”?

- **The Holy Trinity**
  - Traces
  - Dumps
  - Logs
- **Almost all GPFS problems need the data above. Some problems may need additional data, for example**
  - GPFS command output
  - Application output (with timestamps, inasmuch as possible)
  - System config info (kernel version, NSD config, etc.)
  - Copies of GPFS binaries
  - User data (copies of bad files, “ls –li” of problematic directories, etc)

Traces

- **GPFS is *very* complex. A *very* detailed record of events is needed to tell what was going on prior to the interesting events**
- **On AIX and Windows, GPFS utilizes OS-provided trace facility, on Linux GPFS uses its own custom tracing facility**
- **Logically, tracing is done in a circular buffer**
  - The size of the buffer (in memory or on disk) determines how much trace data is captured in one shot
  - Once the buffer is full, trace “wraps”, i.e. further events will be put at the beginning of the buffer
  - If simply left enabled, GPFS traces won’t fill up the disk, the tracing memory and disk footprint is fixed
  - Once something interesting occurs, traces need to be “recycled” aka “cut”, to capture a trace snapshot covering the interesting events
  - Enabling tracing after the interesting events doesn’t help

Trace modes on Linux

- **Blocking mode (old)**
  - 2 smaller memory buffers, larger raw trace file on disk
  - When both memory buffers fill up, trace submitters wait
  - Data in the raw trace file is compressed
  - *Pros*: smaller memory footprint, more trace coverage (due to compression and storing raw trace data on disk)
  - *Cons*: more run-time performance overhead
- **Overwrite mode (new)**
  - One large memory buffer, same size raw trace file on disk
  - When the memory buffer fills up, it wraps around
  - Trace data is only written to disk on request
  - *Pros*: less run-time performance overhead
  - *Cons*: higher memory footprint, less trace coverage

GPFS dumps

- **A “daemon dump”, aka “internal dump” is a snapshot of the internal mmfsd state**
  - It’s a plain text, human-readable file
  - Can be very large but typically compress very well (10:1)
  - Can be generated on demand: “mmfsadm dump all > dump.all.out”
  - Will be generated by GPFS automatically when certain problems occur, provided the dump dir (“/tmp/mmfs” by default) exists
  - Not a safe operation to do on busy node, may crash mmfsd (pre 4.1)
- **A “kthreads dump” is a snapshot of stacks of *all* kernel threads**
  - Also a plain text file
  - Typically not large
  - Can be generated on demand: “mmdumpkthreads > kthreads.out”
  - Included in some daemon dumps
  - Safe to generate

Logs

- **GPFS logs are intentionally placed in a hard-to-find location**
  - GPFS 4.1 and later also sends log events to OS syslog
  - /var/adm/ras/mmfs.log.*
  - All log file names incorporate hostname and timestamp
  - mmfs.log.latest and mmfs.log.previous are symlinks
  - Logs are auto-rotated when GPFS is restarted by sysadmin
- **OS logs**
  - Linux: syslog
    - Typically /var/log/messages*, but may be forwarded to a central server
    - dmesg has kernel log messages since last boot if syslog is misconfigured
  - AIX: Error log (“errpt -a” output)
  - Windows: Event Log
- **Any application or management command logs that could be relevant**
  - Timestamps are very valuable in any log

Kernel dumps

- **Kernel dumps**
  - Linux
    - No kernel dump in a default config, kdump needs to be enabled (through a distro-specific procedure)
    - When kdump is enabled, vmcore* files are placed under /var/crash
    - In order to look at a vmcore, “kernel-debuginfo*” and “crash” rpms need to be installed on a node of the same architecture
  - AIX
    - Kernel dumps are generated by default
    - A dump device of ample size needs to be configured beforehand
    - A dump can be copied out using “savecore”, and examined using “kdb”
  - Windows
    - No kernel dump in a default config, but can be enabled through boot options
    - MEMORY.DMP will be generated in OS-specific dir
    - Debugging Tools for Windows and symbol packages need to be installed to view

Other commonly needed debug data

- **Copies of GPFS binaries**
  - Some problems, e.g. mmfsd SIGSEGV and kernel oops events, require reading of disassembled GPFS code
  - It's critical to have binaries that were in use when the problem occurred for generating disassembly listings
- **Environmental data**
  - Many environmental details have great significance to GPFS
  - gpfs.snap collects many pieces of config data
- **Command output**
  - Output of certain GPFS admin commands, e.g. mmfsck, is invaluable
- **Console log**
  - Having a prompt with embedded hostname and timestamps is very helpful

Probing GPFS state live

- **“mmgetstate -a”: outlook of cluster membership for all nodes**
  - First step in most live troubleshooting scenarios
- **“mmlsmount -L”: outlook of who has what mounted**
  - Both internal and external mounts are shown
- **“mmlsmgr”: shows fsmgr for each fs in use, and the clmgr**
- **“mmfsadm dump”: GPFS troubleshooter’s best friend**
  - Very powerful and possibly dangerous command
  - Can be used to display various types of internal mmfsd state
  - Don’t use it on production systems unless you really know what you’re doing
- **mmdiag: the official, safe, and supported counterpart to ‘mmfsadm dump’**
  - Doesn’t dump as much data, but won’t crash mmfsd

Probing GPFS state live, cont

- **mmfsadm: gateway to mmfsd**
  - A standalone command that mostly sends instructions to mmfsd
  - Has a large number of subcommands that allow one to examine and tweak GPFS state
  - Some mmfsadm subcommands are dangerous to use on a busy node
    - In GPFS 4.1, fault handling was added to dump code making it safer
- **Most common use is to dump GPFS internal state**
  - “mmfsadm dump all”: generates an “internal dump” -- large amount of text written to stdout containing human-readable internal GPFS state
  - “mmfsadm saferdump”: same as “dump”, but safer (but still not safe!)
  - An internal dump is the main source of info for deadlock troubleshooting, without it there is little hope of figuring out what is going on
  - A live node is needed to run mmfsadm (don’t kill nodes without collecting an internal dump first!)
  - A full internal dump varies in size from 1-5 MB for a typical HPC client to tens to hundreds of MB for manager nodes with large caches

Different Problem Types

Daemon asserts

- **An “assert” is a basic sanity check**
  - A programmer makes an assertion that a certain condition must be true
  - If the condition is false, something is badly wrong, and it’s not safe to proceed
    - “failed assert”, “assert went off”, “assert hit”
  - An assert raises SIGABRT signal, which leads to an mmfsd abnormal shutdown and restart
  - An internal dump will be auto-generated by the signal handler (if the dump dir exists), using a filename with “signal”, e.g.
    - internaldump.2013.06.25.07.19.53.signal.16278.c12c4apv11.gz
  - In most cases, a failed assert indicates that there’s a bug in GPFS, but there are exceptions
    - In some cases an assert may go off due to an environmental problem, e.g. lack of memory (which is not really right)
  - An assert is identified by its reason string and location (source file name and line number). Some asserts are “generic”, i.e. have varying causes

Daemon asserts, cont

- **There are different flavors of asserts**
  - LOGASSERT
    - Always enabled, in all builds, used to catch bad problems where
  - DBGASSERT
    - Only enabled in builds with TEMPSHIP_DEFINES enabled
      - Pre-GA builds, first few PTFs of a major release
    - Used to catch conditions that shouldn't happen but do, and don't represent an obvious threat to GPFS integrity (it's OK to skip those asserts)
  - DYNASSERT
    - Controlled dynamically by a config parameter ("dynassert")
    - User to dynamically control asserts that may be bothersome to customers but are interesting to testers
- **FSSTRUCT assert**
  - If "assertOnStructureError" is set, an assert will go off if an FSSTRUCT error is encountered

Fatal signals

- **If mmfsd code performs an illegal instruction, e.g. dereferences a bad pointer, it may raise a fatal signal**
  - Typically SIGSEGV (Signal 11), although SIGFPE, SIGILL and SIGBUS are also not uncommon
- **Since a failed assert results in SIGABRT, the outcome of an assert and a different fatal signal is essentially the same**
  - mmfsd restarts, auto-generated dump is left behind
- **A given problem instance is identified by the address of the offending instruction (RIP)**
  - The absolute RIP only makes sense for a specific binary, so disassembly must be done on a binary from the same build
  - Relative offsets within a function usually don't change between builds if the code isn't changed, so some analysis could be made with another binary
- **If another signal is hit while processing a signal, core is dumped**

Abnormal daemon shutdown

- **Some errors, e.g. hitting ENOMEM in a very inopportune spot, or disk lease thread getting stuck for an hour, result in mmfsd shutting down abnormally**
- **The symptoms are largely similar to what happens after an assert/fatal signal, but not exactly**
  - No traceback in mmfs.log, the reason for the shutdown needs to be discerned from the corresponding internal dump
  - An internal dump is auto-generated, but it has ‘shutdown’ instead of ‘signal’ in the filename, e.g.
    - internaldump.2013.06.17.15.56.35.shutdown.18010.hs22n21.gz

Kernel panic

- **In kernel code, an exception that would lead to a fatal signal in userspace leads to a kernel crash**
  - "kernel oops" on Linux
  - "kernel exception" on AIX/Windows
- **Visible symptoms depend on OS config**
  - On AIX and Windows, a kernel dump would be produced by default, followed by a reboot
  - On Linux, if panic_on_oops is disabled, a node may stay up and appear functional after an oops
    - The thread that caused an oops would be killed, and if it was holding any locks at the time, those won't be released, which may lead to deadlock symptoms
    - If the oops is caused by an mmfsd thread, the mmfsd Healthchecker thread will notice this and trigger an abormal shutdown, otherwise GPFS may be unaware
    - Proactive syslog monitoring for oops'es is needed in such a configuration

Kernel assert

- **An assert going off in kernel code will lead to a kernel panic**
  - mmfsd will get a mailbox message telling it to commit suicide
  - there will be !"kernel requested to die“ assert symptoms in mmfs.log, and the kernel assert reason string will be shown as well:

```
Wed Dec 14 16:24:17.592 2011: logAssertFailed: (!"kernel requested to die")
Wed Dec 14 16:24:17.593 2011: return code 0, reason code 0, log record tag 0
mmfsd: svfs.C:379: void logAssertFailed(UInt32, const char*, UInt32, Int32, Int32,
   UInt32, const char*, const char*): Assertion `(!kernel requested to die")
threadId 24480 Failure at line 2827 in file mmfsnode.C rc 0 reason 0 data
(!"unexpected gnode found in readOSNode")
```

  - The bulk of the debug data (traceback, registers dump) will need to be located in the kernel dump and/or syslog

Dealing with kernel crashes on Linux

- **There are two basic approaches to debug data collection for kernel crashes on Linux:**
  - Enable kernel dump
    - This is good for working problems where kernel state needs to be examined
    - ‘crash’ can be used to examine vmcore, and will provide reliable traceback and disassembly commands
    - No internal dumps from mmfsd
    - The content of the in-memory trace buffers can be extracted from a vmcore, but if tracing was running in the blocking mode, some trace events may be lost
      - Ask for assistance from service
    - vmcore files can be very large and awkward to deal with
  - Disable panic_on_oops and reboot on panic (echo 0 > /proc/sys/kernel/panic)
    - In most cases, the node will remain up and useable after an oops, and traces and dumps can be collected in a usual fashion
    - Oops traceback in the syslog may be very noisy on x86_64

Deadlocks

- **This topic will be covered in greater detail later**

File system metadata corruption

- **GPFS can detect most cases where metadata is obviously corrupt, or is in a self-inconsistent state**
  - However, GPFS can't tell whether user file content is correct
- **When real or potential metadata corruption is detected, FSSTRUCT error is generated**
  - The error goes into syslog (Linux) or errpt (AIX), but not in mmfs.log
  - If assertOnStructureError is enabled, mmfsd will restart, otherwise the show will go on, and the problem may go unnoticed
    - In some cases, e.g. when one of the metadata replicas is corrupt, but the other is OK, there will be no visible problems from the user standpoint
  - An FSSTRUCT error does not necessarily mean something is really wrong on disk, the problem may be transient and in memory only
  - Use /usr/lpp/mmfs/samples/debugtools/fsstruct[lx].awk to format FSSTRUCT log entries into more palatable form
  - Each FSSTRUCT instance has an error code, e.g. FSErrValidate = 108, which can be used to match it to a corresponding LogFSError call

File system corruption and mmfsck

- **When metadata on disk is damaged, the only (official) recourse is to run mmfsck to conduct repairs**
  - Metadata corruption is not a well-defined problem, there's a practically infinite number of possible corruption scenarios, and thus repairing arbitrary damage is never a sure thing
  - If at all possible, avoid doing blind repairs, i.e. running 'mmfsck -y'. Always try first to run mmfsck in the read-only mode ('mmfsck -n')
  - There is always an outside chance that trying to repair damage could end up making things worse
    - mmfsck is not good at recognizing relative value of different objects. If, say, inode file and a user data file share a cross-linked block, mmfsck will punch a hole in both to repair the problem, which may could do grave damage
    - An analysis of 'mmfsck -n' output by a knowledgeable GPFS developer is always a good idea before repairing anything non-trivial
- **File system corruption is like a snowflake**
  - Each one is unique

Application errors

- **This is a particularly difficult type of problem, because GPFS proper may or may not see anything amiss**
  - Generally, ‘grep “gpfs_[ifsd]” trcrpt*’ will give provide some idea of what errors were returned by GPFS code to the callers, but there isn’t always an explicit error returned by GPFS
  - Some problems, e.g. a spurious ENOENT, may not even involve GPFS at the point where the error is returned (e.g. when a stale negative dentry is present), but may stem from earlier misbehavior
  - Finding the relevant events in the traces can be very challenging, since there are no obvious reference points (like logAssertFailed)
  - It is critical to cut traces very promptly upon seeing an unexpected error, and provide high-resolution timestamps, names and inode numbers of objects involved
  - strace/truss of the problematic app can be very helpful in narrowing down the problem location

Working basic problems

Narrowing down the problem

- **Unless a problem is found by an experienced user, many initial problem reports are exceedingly vague**
  - "I'm seeing Stale [NFS] File Handles" is a popular one. It means an IO request has failed with ESTALE. Possible causes are many:
    - Quorum loss (and the resulting file system panic)
    - File system panic due to environmental problems (e.g. down disks)
    - mmfsd assert/signal/abnormal shutdown
    - Force-closed files (e.g. after mmunlinkfileset -f)
  - "GPFS file systems are not mounted"
  - "GPFS is not up"
- **The first steps is always to examine mmfs.log.latest, on the node where the problem was reported**
- **Next, locate fsmgr/clmgr nodes, via mmlsmgr, and examine mmfs.log there**

Quorum loss

- **In most cases, quorum loss is caused by network problems, but there are many exceptions**
  - mmfsd failures often contribute
  - Find (former or current) clmgr (by looking for ‘is now Cluster Manager’ in logs), study the sequence of events, then look at logs on individual nodes
  - Be sure to check whether tiebreaker disk is in use (meaning minority quorum, a very different model from the default majority quorum)
  - If a node is expelled due to ECONNRESET or disk lease non-renewal, it may be an actual network problem, or it could be GPFS code misbehaving
    - Don’t blindly blame network without examining traces from both sides
    - Look at what’s happening with ‘ccMsgDiskLease’ RPCs, on server and receiver side
    - Look at what DiskLeaseThread thread is up to (is it stuck?)
    - Be especially wary on diskless clients, where /var/mmfs in NFS-mounted

Disk problems

- **FS panics and mount failures are often caused by disk errors**
  - E_IO (5), E_NODEV (19), E_DISK_UNAVAIL (217), E_ALL_UNAVAIL (218) are typical errors
  - Check the disk availability picture: ‘mmlsnsd -M’
  - If some disks are not visible where they must be, e.g. NSD server, look closer:
    - ‘tspreparedisk -s’ shows a list of block devices with NSD IDs
    - ‘mmdevdiscover’ shows all block devices visible to GPFS
    - check whether /var/mmfs/etc/nsddevices exists
    - if a device is visible and has correct NSD ID, but GPFS won’t recognize it, compare ‘device type’ shown by ‘tspreparedisk -s’ and ‘mmlsnsd -X’
    - if a specific device looks like it should be a GPFS NSD, but isn’t being used, some of the on-disk descriptors may be damaged/overwritten
    - use ‘mmfsadm test readdescraw devname’ to see what descriptors are there
  - Look for SCSI/MP errors in syslog/errpt
  - If EIO errors are returned for a healthy disk, check for SCSI reservations
  - If SCSI-3 PR is in use, special techniques must be employed

Deadlocks

What *is* a deadlock?

- **Various threads stop making progress, usually waiting for some resource or event, and end up waiting forever**
- **Typical symptoms**
  - Application threads hung in IO syscalls; usually can't be killed, even with SIGKILL (kill -9)
    - "ls -l hangs and can't be killed! df hangs too!"
  - mmfsd threads with long wait times in mmfsadm dump waiters
  - Nodes either completely or partially unresponsive (console sessions hung, remote commands cannot be executed); usually a sign of a deadlock with some kernel resources held
- **Pseudo-deadlocks**
  - When things slow down by a large enough margin, even though some progress is being made, the system will look effectively deadlocked to users

Why do deadlocks happen?

- **Bugs, in GPFS or OS kernel**
  - The heavily multithreaded nature of GPFS and an infinite number of possible failure scenarios are conducive to deadlocks due to locking order problems, resource starvation, etc.
- **Stuck disk IO requests**
  - GPFS requires non-blocking disk IO semantics: an IO should complete or fail, but not hang. There is no way to cancel a stuck IO (short of reboot)
- **Packets stuck or dropped in the network layer**
  - GPFS relies on TCP to provide a reliable transport. Most of the time TCP does a good job, but in some cases (usually due to buggy device drivers/firmware) packets may be lost or stuck in some kernel queue
- **POSIX semantics require that GPFS waits for current operations to finish before allowing a conflicting operation to proceed**
  - This magnifies the impact of certain problems: a problem local to a given node may deadlock other nodes, because they have to wait politely for the node to finish what it's doing

Causes of pseudo-deadlocks

- **Very slow IO**
  - With millions of IOs, slow IO times add up quickly
  - From the user standpoint, it looks like a real deadlock
  - IO completion times >500 ms are a cause for concern
  - IO completion times > 2-3 sec are a definite sign of trouble
- **Excessive swapping**
  - May slow things down to a near-halt
- **Extreme overload**
  - Same symptoms as very slow IO/swapping

Automated deadlock detection (4.1+)

- **Starting with V4.1, GPFS itself actively monitors certain long waiters**
  - Waiters that can be legitimately long as excluded
  - If waiters are found exceeding **deadlockDetectionThreshold** (300 sec by default), a warning is produced in the logs, and debug data is automatically collected
    - Short trace snapshot, internal dumps, kernel threads dump
  - No more than **deadlockDataCollectionDailyLimit** (10 by default) sets of debug data per 24 hours can be collected
  - Heuristics are used to detect a situation when a cluster is overloaded, in which case deadlock detection thresholds are implicitly raised
  - Despite heuristics, it is still possible to hit a false positive on a heavily loaded node when some operations are slow rather than stuck
  - Adjusting detection thresholds based on actual cluster observation is recommended

What about Linux “hung task” warnings?

- **I saw this in syslog. This is clearly a bug, right?**
  - INFO: task cmsRun:22628 blocked for more than 120 seconds.
    "echo 0 > /proc/sys/kernel/hung_task_timeout_secs" disables this message.
    Call Trace:
    <ffffffff884a1b14> :mmfslinux:cxiWaitEventWait+0x114/0x1d0
    <ffffffff80288e63> default_wake_function+0x0/0xe
    <ffffffff8022f0c3> __wake_up+0x38/0x4f
    <ffffffff88556039> :mmfs:_ZN6ThCond12internalWaitEP16KernelSynchStatejPv+0xf9/0x1c0
    <ffffffff88555b8c> :mmfs:_ZN6ThCond5kWaitEiPKc+0x8c/0x190
    <ffffffff884e9db4> :mmfs:_ZN13KernelMailbox21sendToDaemonWithReplyEv+0x144/0x2a0
    <ffffffff88505cbf> :mmfs:_Z12kShHashFetchP15KernelOperationP8CacheObjtsiPvij+0x1cf/0x220
    <ffffffff884f3a5a> :mmfs:_Z11lockInodeMGP15KernelOperationP8OpenFileN5LkObj12LockModeEnumEiR7FileUIDR13LookupDetails+0x15a/0x180
    <ffffffff884ec0ae> :mmfs:_Z11lockGetattrP15KernelOperationP13gpfsVfsData_tR7FileUIDiR13LookupDetailsR10WhatLockedRP8OpenFileP10cxiVattr_t+
    0x4ee/0xd10
    <ffffffff884ee811> :mmfs:_ZN10gpfsNode_t6lookupEP15KernelOperationP13gpfsVfsData_tPS_PvPcjP10ext_cred_tPS5_PS4_PjS9_Pi+0x291/0xa60
    <ffffffff88515513> :mmfs:_Z10gpfsLookupP13gpfsVfsData_tPvP9cxiNode_tS1_PcPS1_PS3_PjP10cxiVattr_tS7_P10ext_cred_tjS5_Pi+0x4b3/0x670
    <ffffffff8852877d> :mmfs:_Z10gpfsAccessP13gpfsVfsData_tP9cxiNode_tiiPvPiP10ext_cred_t+0x10d/0x5d0
    <ffffffff88515060> :mmfs:_Z10gpfsLookupP13gpfsVfsData_tPvP9cxiNode_tS1_PcPS1_PS3_PjP10cxiVattr_tS7_P10ext_cred_tjS5_Pi+0x0/0x670
    <ffffffff884b01f7> :mmfslinux:gpfs_i_lookup+0x267/0x3e0
    <ffffffff884b1b66> :mmfslinux:gpfs_i_permission+0x106/0x260
    <ffffffff802108e8> generic_file_buffered_write+0x20b/0x60c
    <ffffffff8020d60e> do_lookup+0xe5/0x1e6
    <ffffffff8020a334> __link_path_walk+0x3a6/0xf5b
    <ffffffff8020ee4c> link_path_walk+0x42/0xb2
    <ffffffff8020d3a9> do_path_lookup+0x275/0x2f1
    <ffffffff8020b33a> kmem_cache_alloc+0x62/0x6d
    <ffffffff80223f35> __path_lookup_intent_open+0x56/0x97
    <ffffffff8021b4b8> open_namei+0x73/0x6d5
    <ffffffff802676cd> do_page_fault+0xfae/0x12e0
    <ffffffff80227cac> do_filp_open+0x1c/0x38
    <ffffffff8021a366> do_sys_open+0x44/0xbe
- **Not necessarily. This merely means the kernel is waiting for mmfsd.**
  - Next step: look at the waiters. There may or may not be something truly stuck.

I think there's a deadlock. Where to start?

- **Pre-4.1:**
  - Look for "long waiters", i.e. mmfsd threads that wait for something
    - 'mmfadm dump waiters' or 'mmdiag --waiters'
- **4.1+: Use automated deadlock detection**
  - 'mmdiag –deadlock' shows a filtered list of long waiters
- **Are all nodes alive/responsive?**
  - Yes:
    - Get internal dumps, kthread dumps, and traces (if not collected automatically)
      - 'gpfs.snap --deadlock'
  - No:
    - Force a kernel dump on non-responsive nodes
      - By sending an NMI through IPMI/IMM/BladeCenter Console (x86_64)
      - By forcing a dump directly through HMC (Power)

I captured debug data. What next?

- **GPFS deadlocks tends to involve multiple nodes**
- **In a large cluster, typically a subset of nodes is directly involved**
  - A first-pass analysis of debug data can dramatically reduce the amount of debug data to upload to IBM servers by identifying “interesting nodes”
  - Debug data should still be collected on all nodes and kept while analysis is going on
- **One needs to start with a node where hang symptoms were reported, or pick a random node with long waiters, and follow the chain of dependencies**
  - Deadlock analysis starts with an examination of internal dumps
  - Node currently serving certain roles are often involved in cluster-wide deadlocks
- **Sounds good, so how do I interpret an internal dump?**
  - You have to know some basics of GPFS anatomy

Basic GPFS concepts

Basic concepts: clusters and cluster membership

- **GPFS cluster is a statically defined collection of nodes constituting a single *administrative* domain**
  - mm commands (mmchconfig, mmaddnode, etc) have “cluster” scope
  - Each GPFS resource (NSD object, file system object) is owned by a cluster
  - Each node in the cluster has its own configuration
    - Core GPFS config is stored in “mmsdrfs” file
    - Starting with V4.1, Cluster Configuration Repository (CCR) is used to manage config of GPFS and other components
- **Multi-cluster functionality allows nodes from different clusters to mount the same file system**
  - Mount operation makes remote nodes dynamically join the “home cluster”
  - Once a node joins a cluster, it effectively becomes a member: it participates in the group protocol (and may deadlock it)
  - All nodes mounting a given file system participate in file system activities mostly the same way (a remote node may cause a deadlock too)
  - Deadlocks don’t respect cluster boundaries

Basic concepts: stripe group

- **“Stripe Group” is the original term for an object now publicly known as “file system”**
  - Commonly abbreviated as “SG”
- **Most resources are local in scope to a given stripe group**
  - Many deadlocks only affect a specific file system
- **Each SG has a dynamically appointed manager node (sgmgr) which performs most tasks requiring centralized processing**
  - Most deadlocks involve sgmgr
  - mmlsmgr shows currently appointed sgmgr’s
  - One node may be sgmgr for multiple SGs
- **If GPFS detects a very bad problem, an SG may be “panicked”**
- **sgmgr role may be shifted to another node due to sgmgr node failure, SG panic, or mmchmgr command**
  - Due to complexities of SG recovery, the above events historically have been a fertile ground for deadlocks

Basic concepts: node roles

- **Cluster manager (clmgr, formerly "config manager" or cfgmgr)**
  - One node dynamically elected out of a pool of GPFS quorum nodes
  - Drives GPFS group protocol (node joins/leaves), appoints sgmgr nodes, grants disk leases, expels bad nodes. Group protocol is another usual deadlock suspect
  - mmlsmgr reports which node is clmgr
- **Stripe Group manager (sgmgr), aka File System manager (fsmgr)**
  - One per file system; selected from a pool of manager nodes
  - Implicitly has other roles: token manager, allocation manager, quota manager, Parallel Inode Traversal (PIT) master.
- **Token manager (tmmgr, aka TM server)**
  - If a sufficiently large number of clients mount given SG, extra TM server nodes will be appointed, in addition to sgmgr.
  - Basic function is to serve tokens and store token state in local memory
- **Config server (primary + backup) static config management (when CCR is not enabled)**
- **NSD server**

Basic concepts: node failure recovery

- **When properly configured, GPFS can tolerate a failure of any node without returning IO errors to applications**
  - Great concept, comes at a great cost in terms of code complexity
  - The vast majority of deadlocks happen during/due to failure recovery
- **What happens after a node fails**
  - clmgr runs group leave protocol
  - One of the protocol phases (phase3) involves SG recovery, run by sgmgr
  - sgmgr runs log recovery, token recovery, alloc recovery, etc
- **If the failed node was serving in a special capacity (clmgr, sgmgr, tmmgr) node recovery becomes far more complex**
  - Election held for new clmgr in case of clmgr failure
  - Token recovery in case of tmmgr / sgmgr failure
  - SG takeover in case of sgmgr failure
  - NSD failover in case of NSD server failure

Basic concepts: kernel and userspace GPFS code

- **Traditionally, a file system is implemented entirely in kernel**
- **GPFS has a kernel code piece, and a userspace piece**
  - mmfsd is the process where GPFS userspace code lives
  - mmfsd performs tasks that are cumbersome to do in the kernel: network communications, complex file operations (e.g. mkdir), prefetch / writebehind, user exit execution, etc.
  - mmfsd is heavily multithreaded: at any given time it has dozens to hundreds of threads (depending on IO workload and config settings)
- **Kernel code generates work requests and sends them to mmfsd using *mailboxes***
  - A mailbox call is a reverse of syscall
  - Mailbox requests are handled by mailbox worker threads (worker1threads pool)
  - Typical requests: “acquire a token”, “fill buffer with data”, “create directory”

Basic concepts: Locks, tokens, et al

- **GPFS uses a wide assortment of synchronization primitives**
  - **Lock**: generic term used to refer to many things; usually it means a lock on a GPFS “Lockable object”, aka LkObj. Locks can be local to a given node, or be backed by **tokens** (i.e. a token needs to be acquired in order to lock an object)
  - **Token**: a right to perform certain operations on certain objects. Tokens are granted by **token manager** nodes and have file system scope. Tokens exist for regular objects (e.g. inodes) or byte ranges.
  - **Mutex**: node-local synchronization primitive. GPFS has its own implementation of mutexes that spans kernel and userspace code. GPFS uses mutexes to serialize thread execution.
  - **Condvar** (conditional variable): another local synchronization primitive. GPFS code uses condvars to efficiently wait for particular events/conditions.
  - **Other**: various GPFS objects have specialized synchronization primitives: FlushFlag, exclusive holder, etc.

Basic concepts: token operations

- **A token must be acquired prior to accessing certain objects**
  - tmMsgTellAcquire1 RPC sent to tmmgr
  - tmmgr checks existing token state for conflicts with tokens held by other nodes; if there are no conflicts, token is granted, otherwise tmmgr replies with a *copyset*: list of nodes holding conflicting tokens
- **If copyset is returned, TM client performs token revoke**
  - tmMsgRevoke RPC sent to other nodes
- **Once the revoke op is finished, TM client performs "second tell request"**
  - tmMsgTellAcquire2 sent to tmmgr
- **BR tokens have corresponding code paths / RPCs**
- **Token ops are the cornerstone of GPFS operations, and TM RPCs constitute a large portion of overall RPC traffic**
  - When analyzing a deadlock, most tsMsg* RPCs can be ignored on the initial pass; most of them are just background noise
  - If the deadlock indeed involves TM code, it is usually exceedingly difficult to figure out – punt to GPFS developers

Internal dump

GPFS internal dump: waiters

- **Shows a list of mmfsd threads waiting for something**
  - Two threads waiting for RPC replies; target IP address not shown because there are multiple destinations (check dump tscomm)

```
0x110E645E8 waiting 82786.356687395 seconds, GroupProtocolDriverThread: on ThCond
   0x110E8E4F8 (0x110E8E4F8) (MsgRecord), reason 'RPC wait' for ccMsgGroupLeave
0x110BF5F48 waiting 144731.007430298 seconds, SGExceptionAdjustServeTMThread: on
   ThCond 0x110BE77F8 (0x110BE77F8) (MsgRecord), reason 'RPC wait' for
   sgmMsgTMServe
```

  - Thread waiting for IO completion (SAN / disk problem)

```
0x111941B48 waiting 152.372374286 seconds, parallel tsrestripefs worker: for I/O
   completion on disk t1f8sq2h16v
```

  - Thread waiting for a mutex

```
0x110B33488 waiting 15295.359606679 seconds, SG async recovery: on ThMutex
   0xF1000012A0932A50 (0xF1000012A0932A50) (SG management mgr data)
```

  - Thread waiting on a condvar

```
0x110B38A28 waiting 15301.176843049 seconds, tsdefragfs command: on ThCond
   0xF1000012A0892A78 (0xF1000012A0892A78) (SG management mgr data), reason
   'waiting for stripe group to recover'
```

GPFS internal dump: threads

- **Stack tracebacks and extra info on mmfsd threads**
- **SG async recovery: desc 0x110B33488 handle 0x0078C03E parm 0x3000 highStackP 0x110B84790**

```
    pthread 0x3E3F kernel thread id 299919 (slot 36) pool 1
    per-thread gbls:
      0:0x0 1:0x0 2:0x0 3:0x3 4:0xFFFFFFFFFFFFFFFF 5:0x0 6:0x0 7:0x0
      8:0x0 9:0x0 10:0x0
    waiting on ThMutex 0xF1000012A0932A50 (0xF1000012A0932A50) (SG management mgr
data)
    waiting for 15295.362996862 seconds
    thread traceback:
      0xFFFFFFFFFFFFFFFC
      0x1002B90A0 SGMgrData::acquireMgrMutex() + 0x34
      0x1002C2D00 SGMgrData::doAsyncRecovery() + 0x17C
      0x1002C2B0C StripeGroup::handleAsyncRecovery() + 0x18
      0x1002E0804 StripeGroupCfg::exceptionHandlerBody(void*) + 0x15C
      0x100033640 Thread::callBody(Thread*) + 0xD8
      0x100003608 Thread::callBodyWrapper(Thread*) + 0xB0
      0x90000000017143C _pthread_body() + 0xDC
      0xFFFFFFFFFFFFFFFC
```

- **Note: sometimes threads may be stuck without having ‘aux wait reason’ set, and won’t show up in the waiters list. Try to look through all non-idle threads**

GPFS internal dump: mutexes and condvars

- **Who's holding this mutex? "dump mutex" section**

```
'SG management mgr data' at 0xF1000012A0932A50 (0xF1000012A0932A50) :
   held by thread 451787, 'Mount handler' at 0x1100CA968 (tp 110147800)
   lockWord 0x110147801 mutexEventWordP 0xF10001401D329028
```

- **What object does a given condvar correspond to? "dump condvar" section**

```
'LkObj' at 0x18025D1C698 (0xFFFFC20025D1C698)
   (mutex 'InodeCacheObj' at 0x18025D1C4D0 (0xFFFFC20025D1C4D0))
   waitCount 1 condvarEventWordP 0xFFFF810288B7E148
```

  - LkObj condvar and mutex addresses can be used to locate the corresponding OpenFile and follow the chain further

```
OpenFile:  key 3684B64D4D6EA8C0:0000000000011C34:0000000000000000, addr 0x18025D1C490
  cached 1, refBit 1, holdCount 4, tokenCount 5, mutex 0x18025D1C4D0 (0xFFFFC20025D1C4D0)
    list 0 lruNextP 0x18025D1B728 (0xFFFFC20025D1B728) lruPrevP 0x18025D1D1F8 (0xFFFFC20025D1D1F8)
  Inode: status valid, eff token xw, addr 0x18025D1C660, cond 0x18025D1C698 (0xFFFFC20025D1C698)
    lock state [ xw ] x [] flags [ dmn rvk ], writer 0x7CCDCD0, hasWaiters 1
    waiting revoke:
    key 3684B64D4D6EA8C0:0000000000011C34:0000000000000000, state LockWait, reqMode ro
      reqFlags 0x20000004, sequence_number 179541, revokeReq 0x2AAAB421FB30
        tmClient: Mode xw State STABLE Type Inode Flags 0x0 () seqNum 179541 ThCond 0x18025D1C668
```

GPFS internal dump: tscomm

- **Which nodes have not responded to RPCs?**

```
Pending messages:
  msg_id 6420, service 2.1, msg_type 34 'sgmMsgTMServe', n_dest 5, n_pending 1
  this 0x110BE7688, n_xhold 1, ccP 0xF1000004304D7A48 cbFn 0x0
    sent by 'SGExceptionAdjustServeTMThread' (0x110BF5F48)
    dest 9.114.94.65     status node_failed, err 233, reply len 0
    dest 9.114.94.66     status node_failed, err 233, reply len 0
    dest 9.114.94.81     status node_failed, err 233, reply len 0
    dest 9.114.9.115     status pending   , err 0, reply len 0
    dest 9.114.9.116     status node_failed, err 233, reply len 0
  msg_id 44892, service 1.1, msg_type 26 'ccMsgGroupLeave', n_dest 7, n_pending 1
  this 0x110E8E388, n_xhold 1, ccP 0xF1000004304D7A48 cbFn 0x0
    sent by 'GroupProtocolDriverThread' (0x110E645E8)
    dest 9.114.94.73     status success   , err 0, reply len 0
    dest 9.114.9.115     status pending   , err 0, reply len 0
    dest 9.114.9.116     status success   , err 0, reply len 0
    dest 9.114.94.65     status success   , err 0, reply len 0
    dest 9.114.94.66     status success   , err 0, reply len 0
    dest 9.114.94.81     status success   , err 0, reply len 0
    dest 9.114.94.82     status success   , err 0, reply len 0
```

GPFS internal dump: cfgmgr

- **Miscellaneous information about cluster state**

```
node                       primary      admin --status---   join fail  SGs other ip addrs,
  no   idx host name       ip address    func tr p    rpc  seqNo  cnt mngd last failure
---- ----- -------------- ------------ ----- ---------- ------ ---- ---- ------------------
   1     7 c100c3rp1      192.168.100.41 qml -- J      up       5    0    0
   2     0 c100c3rp2      192.168.100.42 qml -- J      up       1    0    0
   3     1 c100c3rp3      192.168.100.43 qml -- J      up       1    0    0
   4     2 c100c3rp4      192.168.100.44 qml -- J      up       1    0    0
   5     4 c100c4rp1      192.168.100.45 -ml -- J      up       2    0    0
   6     5 c100c4rp2      192.168.100.46 -ml -- J      up       3    0    0
   7     3 c100c4rp3      192.168.100.47 qml -- J      up       1    0    0
   8     6 c100c4rp4      192.168.100.48 q-l -- l failed       4    1    0  2007-10-16 14:34:11
(cluster C0A8642D468962FF clusterName jazz.cluster)
  21     9 c100c1rp1      192.168.100.33 --- -- J      up      33    1    0  2007-10-15 14:59:43
   2    12 c100c1rp2      192.168.100.34 --- -- J      up      32    1    0  2007-10-15 14:59:43
   3    11 c100c1rp3      192.168.100.35 --- -- J      up      34    3    0  2007-10-16 09:35:37
  22    14 c100c1rp4      192.168.100.36 --- -- J      up      37    3    0  2007-10-16 09:35:46
  24    15 c100c2rp1      192.168.100.37 --- -- J      up      37    3    0  2007-10-16 09:35:46
   6    16 c100c2rp2      192.168.100.38 --- -- J      up      36    3    0  2007-10-16 09:35:46
   7    10 c100c2rp3      192.168.100.39 --- -- J      up      35    3    0  2007-10-16 09:35:46
   8    13 c100c2rp4      192.168.100.40 --- -- J      up      37    3    0  2007-10-16 09:35:39
  23     8 c35f2n11       192.168.110.30 --- -- J      up      31    1    0  2007-10-15 14:59:06
Groupleader 192.168.100.42 0x00000000 (other node)
Cluster configuration manager is 192.168.100.42 (other node); pendingOps 0
group quorum formation time 2007-10-15 12:40:07
gid 471397e8:c0a8642a elect <2:23146> seq 37 pendingSeq 38, gpdLeave phase 3, joined
GroupLeader: joined 1 gid <2:23146>
```

GPFS internal dump: stripe

- **Miscellaneous information about SGs**

```
State of StripeGroup "fs2" at 0xF1000003403CF1F8, uid 6429C0A8:46A0FBA1, local id 1:
  homeCluster myAddr 192.168.100.43 <0:1> ipAddr 192.168.100.43
  Device id 8000000200000097, mount id 77; use count 2, vfs users 22
  Stripe group manager is 192.168.100.41, mgrChangeCount 2 sgmgr seq 1675804828
  Stripe group is online, formatted, and mountable read/write
  File times: atime enabled, mtime accurate
...
  Root squashing disabled
  Policy file version number 1
  Excluded disks:
  Mount point from /etc/filesystems: "/fs2"
   Dmapi is disabled for the stripe group
   Is mounted at this node in RW mode
   State is not panicked, reason 0, msg printed 0,
   TmServing invited
...
 quota file inodes: user quota 174080, group quota 174081 fileset quota 174082
...
 1: hd2c4: uid C0A8642B:46E1825A, status InUse, availability OK,
             created on node 192.168.100.43, Fri Sep  7 12:54:50 2007
             type 'nsd', sector size 512, failureConfigVersion 4759
             quorum weight {0,0}, failure group: id 3, fg index 0
             nSectors 19531264 (0:12A0600) (9536 MB), inode0Sector 4098
             alloc region: no of bits 1824, seg num 0, offset 48, len 528
             suballocator 0xF1000003403CEF74 type 0 subSize 1824 dataOffset 52
               nRows 6 len/off: 1/0 2/1 4/3 8/7 15/15 29/30
             storage pool: 0
             holds data and metadata
```

GPFS internal dump: files

- **Lots and lots of state info about individual inodes**

```
OpenFile:  key 09727C25:469EB437:00055011:00000000, addr 0xF100000A91064F60
  cached 1, holdCount 4, tokenCount 5, mutex 0xF100000A91064FA0 (0xF100000A91064FA0)
    list 0 lruNextP 0xF100000A90DFEB50 (0xF100000A90DFEB50) lruPrevP 0xF100000A912FAA80 (0xF100000A912FAA80)
   Inode: status valid, eff token wf, addr 0xF1000004305E9480, cond 0xF1000004305E94C8 (0xF1000004305E94C8)
    lock state [ acq ro: 8 wf: 5 ] x [] flags [ rvk ] new_token wf/ro, writer 0xFFFFFFFF, hasWaiters 0
    waiting revoke:
    key 09720973:46A20A96:00009801:00000000, state LockWait, reqMode rs
      reqFlags 0x37008004, sequence_number 7690, reqAddr 9.114.94.81
        tmClient: Mode wf State ACQUIRING Type Inode flags 0x0 seqNum 7691 ThCond 0xF1000004305E9490
...
 BR: addr 0xF100000A91065220
        tmClient: Mode nl State STABLE Type BR flags 0x0 seqNum 7 ThCond 0xF100000A91065230
        treeP 0xF100000A9131B848 btIsClient 1 btFastTrackP 0x0 (0x0) 1 ranges mode RO/XW:
          F100000A9136CC68: BLK [B780000,INF] mode XW node <13>
 inode 348177 snap 0 nlink 1 genNum 0x10095 mode 0200100644: -rw-r--r--
  metanode 192.168.124.4 (other node) takeovers 1 surrenders 1 fail+panic count 0
  locks held in mode xw:
    0xF100000A91065748: 0xB780000-0xD07FFFF tid 1849027 gbl 0 mode xw rel 0
    0xF100000A91065720: 0xD080000-0xD080FFF tid 1697163 gbl 0 mode xw rel 0
  nRevokes 33 nSlotWaiters 0
  destroyOnLastClose 0, vfsReference 0, dioCount 0, dioFlushNeeded 1
  invalidateCachedPages 0, mappedRW 0, mmapIndflush 0, mmapDataP 0x0
  lastAllocLsn 0xF2B6F072, inodeStatusFieldChangeLSN 0x0
  inodeFlushFlag 1, inodeFlushHolder 1849027, inodeFlushWaiters 1, openInstCount 1
```

GPFS internal dump: tokenmgr

Information about TM servers and server state

```
Domain fs3 id 09725E51:469E3716 tcRecoveryState 1 this 0xF1000093A03E64A8
    tcLockNest 0 tcRecoveryState 1 tcFlags 0x0 tcPctMaxFilesToCache 100
    Servers: 7 [ 8:3 3:3 7:3 6:66 4:66 5:66 2:66 ] changeCount 66
    Client stats: acquires 2275518 revokes 1126804 nTellServers 3316368
```

Queued revokes:

```
revokeReq list:
        RevokeReq 0x112094048 Mode 0x02 Flags 0x4 err 0
           msgHdr: magic F3689038 from 192.168.124.8 msg_id 406730 type 16 len 136
        RevokeReq 0x112093E88 Mode 0x02 Flags 0x4 err 0
           msgHdr: magic F3689038 from 192.168.124.72 msg_id 455622 type 16 len 136
        RevokeReq 0x112094348 Mode 0x02 Flags 0x4 err 0
           msgHdr: magic F3689038 from 192.168.124.4 msg_id 460483 type 16 len 136
```

Transition wait queue:

```
Pending requests transition wait queues:
           msgHdr: magic F3689038 from 192.168.209.99 msg_id 13191575 type 6 len 56
             request: mode ww node <11> flags 0x1020 seqNum 51825
               majType 1 minType 7 tokType Mnode key 09727766:4695844B:00009B35:00000000
               tokP 0x301C2A58 stP 0x301C2AA8
           msgHdr: magic F3689038 from 192.168.209.108 msg_id 5527607 type 6 len 56
             request: mode ww node <7> flags 0x1000 seqNum 5244
               majType 1 minType 7 tokType Mnode key 09727766:4695844B:00009B35:00000000
               tokP 0x301C2A58 stP 0x301C2AA8
```

GPFS internal dump: tokens

- **Server-side token state info**

```
TokenID: majType 1 minType 7 key 00042E83:00000000:01070000 at 0x70000000029E398
   tRefCount 17 tMutex 0x70000000029E3B8 tCondvar 0x70000000029E3E0
      SubToken: type Inode stFlags 0x82 state COPYSET stTransNode <3>
      SubToken: type Mnode stFlags 0x0 state STABLE stTransNode none
      SubToken: type BR stFlags 0x81 state COPYSET stTransNode <0>
        treeP 0x7000000001785C0 btIsClient 0 btFastTrackP 0x7000000001DBA10 (0x7000000001DBA10) 2037 ranges mode
    RO/XW:
          700000000021DA8: BLK P 7000000001C7280 (7000000001C7280)[0,1FFF] mode RO list 0x7000000001134B8 2
    nodes: <1> <7>
...
 ClientNode 0x7000000004904F0 flags 0 node <3>
          Inode mode wf state COPYSET seqNum 6420425
          Mnode mode ro state STABLE seqNum 6420517
          BR mode ro state STABLE seqNum 9651369
          DMAPI mode nl state STABLE seqNum 0
          SMBOpen mode nl state STABLE seqNum 0
          SMBOpLk mode wf state STABLE seqNum 6417802
        ClientNode 0x70000000049EB20 flags 0 node <7>
          Inode mode wf state STABLE seqNum 5571312
          Mnode mode ro state STABLE seqNum 5571502
          BR mode ro state STABLE seqNum 13136078
          DMAPI mode nl state STABLE seqNum 0
          SMBOpen mode nl state STABLE seqNum 0
          SMBOpLk mode wf state STABLE seqNum 5565871
...
Cross-reference with “dump cfgmgr”
 2      3 c154n02        9.114.94.66    -m--l -- J      up       4     0    0      0 192.168.0.2
 3      7 c154n09        9.114.94.73    qm--l -- J      up       6     0    0      0 192.168.0.9
```

Beyond mmfsd: mmdumpkthreads

- **“dump threads” and “dump waiters” only cover mmfsd threads**
- **Application IO threads may be waiting in GPFS kernel code**
- **“mmfsadm dump kthreads” or “mmdumpkthreads” generates stack tracebacks of all kernel threads (using kdb command on AIX)**

```
                SLOT NAME     STATE     TID PRI   RQ CPUID  CL  WCHAN
pvthread+010F00  271 perl     SLEEP 10F0C5 03C    0         0  F1000100223F60E8
(0)> where 271
[000533D8]e_block_thread+0004E0 ()
[00053AC8]e_sleep_thread+00005C (??, ??, ??)
[05C10C80]internalWait__6ThCondFP16KernelSynchStateUiPv+000084 (??, ??, ??, ??)
[05C123F0]kWait__6ThCondFiPCc+000150 (??, ??, ??)
[05C4AA38]sendToDaemonWithReply__13KernelMailboxFv+000190 (??)
[05C338E8]kShHashFetch__FP15KernelOperationP8CacheObjUssiPvT5Ui+0002B8 (??, ??, ??, ??, ??, ??, ??,
   ??)
[05BFBF64]lockBufferForWrite__FP15KernelOperationP8OpenFileRC6ObjKeyiP9BufferReqN24P10ext_cred_tPP10
   BufferDescP23cxiIOB
ufferAttachment_t+0004C4 (??, ??, ??, ??, ??, ??, ??, ??)
[05BF2EB4]kSFSWriteFast__FP15KernelOperationP13gpfsVfsData_tP9MMFSVInfoP8cxiUio_tiP10gpfsNode_tP8Ope
   nFileP10ext_cred_tP
Ui+000524 (??, ??, ??, ??, ??, ??, ??, ??)
[05BEC1EC]gpfsWrite__FP13gpfsVfsData_tP15KernelOperationP9cxiNode_tiP8cxiUio_tP9MMFSVInfoP10cxiVattr
   _tT7P10ext_cred_tN2
4+0007FC (??, ??, ??, ??, ??, ??, ??, ??)
[05BD15FC]GpfsRdwr+0002B4 (??, ??, ??, ??, ??, ??, ??, ??)
[05BCF898]gpfs_v_rdwr_attr+000104 (??, ??, ??, ??, ??, ??, ??, ??)
[003F9E90]vnop_rdwr+00019C (??, ??, ??, ??, ??, ??, ??, ??)
[0043C3BC]vno_rw+000074 (??, ??, ??, ??)
[003CA208]rwuio+0000C8 (??, ??, ??, ??, ??)
[003CA414]rdwr+0000F4 (??, ??, ??, ??, ??)
[003C9C04]kwrite+000044 (??, ??, ??)
[00003810].svc_instr+000110 ()
[D0350DBC]write+000130 (??, ??, ??)
```

Livelocks

- **Not every deadlock produces long waiters**
- **In some cases, a thread may not be waiting for anything, but rather may be stuck in an infinite loop**
- **mmfsd will have appreciable CPU utilization**
- **Various clues (mutex, LkObj, FlushFlag, etc. ownership) need to be followed to find out which threads are blocking progress**
- **dump threads needs to be examined to find active threads. If the same active thread stays in the same call stack in repeated dump thread instances, it may be stuck in an infinite loop**
- **Typical example**
  - thread stuck trying to allocate a block on a near-full SG: alloc(), allocOneBlock(), allocReplica(), allocLocalReplica() may be on the call stack

When waiters are short

- **In some cases, instead of waiting on a condvar or mutex, a thread may sleep for short periods of time and wake up periodically to check some condition**
- **While no progress is being made, wait times in dump waiters will not grow for such threads**
- **Typical examples**

```
0x1095E060 waiting 0.995990672 seconds, Open handler: delaying for 0.004009328 more
   seconds, reason: tcAwaitRecovery pause 1 sec
0x109595F0 waiting 0.968994571 seconds, Open handler: delaying for 0.031005429 more
   seconds, reason: ClientToken::Acquire pause 1 sec
0x10822010 waiting 0.967994716 seconds, BRT handler: delaying for 0.032005284 more
   seconds, reason: ClientRangeToken::Acquire pause 1 sec
```

- **Usually such short waiters are not the primary deadlock cause themselves; look for stuck SG recovery**

Finding interesting waiters: RPC families

- **Certain RPCs are more interesting than others**
  - sgmMsg* family: sgmgr RPCs (sgmMsgTMServe, sgmMsgManage, ...)
    - sgmMsgExeTMPhase: token recovery RPCs
  - ccMsg* family: clmgr RPCs (ccMsgTakeoverQuery, ccMsgDiskLease, ...)
    - ccMsgGroup*, groupLeader*: group protocol (ccMsgGroupLeave, ccMsgGroupJoinPhaseN, GroupLeaderJoinMsg, ...)
  - nsdMsg* family: NSD RPCs (nsdMsgRead, nsdMsgGetInfo, ...)
    - stuck NSD IO request is as bad as a stuck local IO request
- **Other RPCs are usually less relevant**
  - tmMsg* family: TM RPCs (tmMsgTellServer, tmMsgRevoke,...)
    - some TM RPCs are exceptions: tmMsgMultiTellServer, tmMsgCleanup
  - allocMsg* family: Block allocation RPCs (allocMsgTypeRequestRegion,...)
  - quotaMsg* family: quota RPCs (quotaMsgRequestShare, quotaMsgRevoke,...)
- **As always, there are exceptions to rules**
  - It's OK to ignore some common RPCs waiters on the first pass – may need another pass later though

Finding interesting waiters: common local waiters

- **Certain waiters are very common – likely just noise**
  - LkObj lock

    ```
    Sync handler: on ThCond 0x41A8F554 (LkObj), reason 'waiting for RO lock'
    ```

  - acquirePending flag

    ```
    Remove handler: on ThCond 0xF1000093A05881A0 (LkObj), reason 'change_lock_shark
      waiting to set acquirePending flag'
    ```

  - Quota share

    ```
    ThCond 0x10ACF4F0 (Quota entry get share), reason 'wait for getShareIsActive = false'
    ```

  - If failure recovery is going on, there will be various “waiting for ... to recover” waiters
- **Some local waiters should not normally be very long**
  - IO completion waiters
  - Most mutex waiters
  - Log object waiters

Typical scenario: group protocol stuck

- After a node failure, clmgr has this waiter:

```
c26f3rp01: 0x20AE59B8 waiting 610.794815690 seconds, GroupProtocolDriverThread: on ThCond 0x216B2BF0
    (0x216B2BF0) (MsgRecord), reason 'waiting for RPC replies' for ccMsgGroupLeave
```

- Look at “dump tscomm” section to find who hasn’t replied yet...

```
 dest 9.114.54.71     status pending   , err 0, reply len 0 [9.114.54.71 is c26f2rp03]
```

- No obvious long waiters on c26f2rp03... Look at GPFS log:

```
Tue Jan 24 22:24:42 2006: GPFS: 6027-1716 Close connection to 192.168.3.32 c26f3rp02
Tue Jan 24 22:24:47 2006: GPFS: 6027-615 Recovering nodes: c26f3rp02
Tue Jan 24 22:24:47 2006: GPFS: 6027-611 Recovery: f3nsdfs, delay 104 sec. for safe recovery.
Tue Jan 24 22:24:49 2006: GPFS: 6027-1710 Connecting to 192.168.3.32 c26f3rp02
Tue Jan 24 22:24:52 2006: GPFS: 6027-1709 Accepted and connected to 9.114.54.74 c26f3rp02
Tue Jan 24 22:50:18 2006: GPFS: 6027-655 Recovered 1 nodes.
```

- Recovery took 6 minutes – too long! Slow IO is the likely suspect
- Look at “dump waiter” snapshots, observe repeating 2-3 sec IOs
- Something’s wrong with network on this node.

Typical scenario: sgmgr broadcast stuck

- **Symptom: mmchdisk command is stuck**
  - On sgmgr, there's a long waiter for an 'interesting' RPC

```
0x110DDDA88 waiting 70979.685493783 seconds, tschdisk command: on ThCond x110F4F010 (0x110F4F010) (MsgRecord),
   reason 'waiting for RPC replies' for sgmMsgSGDescUpdate
```

  - Looks at "dump tscomm" – only one node hasn't replied. Looks for the sgmMsgSGDescUpdate RPC handler there:

```
0x20C4D168 waiting 70979.356764703 seconds, Msg handler sgmMsgSGDescUpdate: for open disk device on disk sdgnsd
```

  - Look at the traceback for this thread:

```
0x104E49F4 getPvidOnDev(char*,NsdId*,long*,int*) + 0x134
0x104E557C findDiskForPvid(char*,NsdId*,long*,char*,int*) + 0x200
0x10169850 NsdDiskConfig::setLocalDevName() + 0x1CC
0x101DAF7C nsdOpen(ClusterConfiguration*,char*,NsdClientDisk**) + 0xF0
0x101E11D0 Disk::devOpen(StripeGroup*,int,Buffer*) + 0x1E4
0x101F1734 StripeGroupDesc::update(const StripeGroupDesc&,unsigned int) + 0xD1C
0x101A5400 StripeGroup::sgdesc_update(char*,int) + 0x164
0x101A1988 StripeGroupCfg::SGHandleUpdate(RpcContext*,char*) + 0x3B8
0x1007DFFC tscHandleMsg(ClusterConfiguration*,TscMsgHeader*,MsgDataBuf*,
NodeIncarnation*,NodeChar*) + 0x384
```

  - Look at "dump nsd" section to find out local device name for "sdgnsd"
  - Attempt dd read from this device – dd hangs. Disk device problem.

More complex scenario: recovery stuck

- **Symptoms: after a quorum loss event, mmlsdisk command hangs. No disk activity on server nodes, long waiters on many nodes**
  - Find clmgr (192.168.100.76). Waiters there:

```
0x108EB580 waiting 1449.009045600 seconds, GroupProtocolDriverThread: on ThCond 0x8000201600 (0x8000201600)
    (SGInfoCondvar), reason 'waiting for SGInfo access'
0x107A5040 waiting 1267.807035024 seconds, Msg handler ccMsgQuerySGMgr: on ThCond 0x8000201600 (0x8000201600)
    (SGInfoCondvar), reason 'waiting for SGInfo access'
0x1078EC90 waiting 1464.378971488 seconds, Msg handler ccMsgResignSGMgr: on ThCond 0x10787BF0 (0x10787BF0)
    (MsgRecord), reason 'RPC wait' for ccMsgNoMgr
```

  - Who's holding SGInfo access right? Look in "dump cfgmgr", under "Assigned stripe group managers"

```
"dmfs2"  id C0A8644D:471B81EF  cond 0x8000201600  sgiHold 4  SG mgr not appointed seq 149688828
 None appointed; done recovery=true
   DMAPI disposition is not set for any event.
 Exclusive access held by 0x1078EC90
```

  - What is thread 0x1078EC90 waiting for? Look in "dump tscomm"

```
msg_id 6420, service 1.1, msg_type  4 'ccMsgNoMgr', n_dest 13, n_pending 2
 this 0x10787A80, n_xhold 1, ccP 0x180005C9698 cbFn 0x0
   sent by 'Msg handler ccMsgResignSGMgr' (0x1078EC90)
   dest 192.168.100.92  status success   , err 0, reply len 0
   dest 192.168.100.73  status pending   , err 0, reply len 0
   dest 192.168.100.77  status success   , err 0, reply len 0
   dest 192.168.100.78  status pending   , err 0, reply len 0
   dest 192.168.100.79  status success   , err 0, reply len 0
```

More complex scenario: recovery stuck (cont.)

- Look at waiters on one of the non-responding nodes:

```
0x10782FE0 waiting 1405.759578272 seconds, Msg handler ccMsgNoMgr: on ThCond 0x106DB498 (0x106DB498)
    (StripeGroupTable), reason 'waiting for SG cleanup'
0x1076BB90 waiting 1468.739559378 seconds, Msg handler sgmMsgExeTMPhase: on ThCond 0x4000081C650
    (0x4000081C650) (MsgRecord), reason 'RPC wait' for tmMsgCleanup on node 192.168.100.80
```

- Most likely SG cleanup is stuck due to the pending tmMsgCleanup RPC. Look at waiters on 192.168.100.80:

```
0x107F3120 waiting 1464.446000000 seconds, SG Exception LocalPanic: on ThCond 0x108061E0 (0x108061E0)
    (MsgRecord), reason 'RPC wait' for ccMsgResignSGMgr on node 192.168.100.76
0x400040398C0 waiting 1464.386000000 seconds, Msg handler tmMsgCleanup: on ThCond 0x1800199D320
    (0xD00000000199D320) (aTokenClassThCond), reason 'Token domain recovery'
```

- Going back to 192.168.100.76 (clmgr) we see ccMsgResignSGMgr handler stuck, itself waiting for an RPC completion – Deadlock.
- No environmental factors in play here: the deadlock happens due to a GPFS bug during a complex failure pattern following loss of quorum. Collect debug data from all nodes involved, restart GPFS on 192.168.100.73 and 192.168.100.78, open a defect/PMR.

More complex scenario: infinite loop

- **Symptoms: stuck IO requests, stuck mm command**
  - Look at the waiters list to find interesting ones:

```
0x218BF918 waiting 3475.660947279 seconds, SharedHashTab fetch handler: on ThCond 0x21DD3A04
    (0x21DD3A04) (MsgRecord), reason 'RPC wait' for tmMsgTellAcquire1 on node 192.168.209.112
0x2063F588 waiting 3475.661020103 seconds, SharedHashTab fetch handler: on ThCond 0x40E7A04C
    (0x40E7A04C) (LkObj), reason 'change_lock_shark waiting to set acquirePending flag'
0x2008B5B8 waiting 3417.514357257 seconds, Sync handler: on ThCond 0x41A8F554 (0x41A8F554) (LkObj),
    reason 'waiting for RO lock'
```

  - Those are likely just noise. Look further:

```
0x20A1E658 waiting 3477.788359859 seconds, Create handler: on ThCond 0x21DBB674 (0x21DBB674)
    (MsgRecord), reason 'RPC wait' for llMsgCopyInodeBlock on node 192.168.209.101
```

  - This is an interesting one. Look at the waiters list on 192.168.209.101:

```
0x20FE8E08 waiting 3150.920868423 seconds, Msg handler llMsgCopyInodeBlock: on ThMutex 0x404C4144
    (0x404C4144) (LL append/trunc)
```

  - Who's holding that mutex? Look at the "dump mutex" section:

```
 'LL append/trunc' at 0x404C4144 (0x404C4144) :
    held by thread 63807, 'Msg handler llMsgCopyInodeBlock' at 0x2063C898, stack offset -1776
    lockWord 0x20997421 mutexEventWordP 0x37CD6614
```

  - What is thread 0x2063C898 doing?

```
0x2063C898 waiting 0.087133317 seconds, Msg handler llMsgCopyInodeBlock: on ThCond 0x221DC8B4
    (0x221DC8B4) (MsgRecord), reason 'RPC wait' for NSD I/O completion
```

  - No long wait here. Hmmm...

More complex scenario: infinite loop (cont.)

- What is this thread doing? Find its traceback in “dump threads”

```
Msg handler llMsgCopyInodeBlock: desc 0x2063C898 handle 0x133A5A36 parm 0x2095F708 highStackP 0x20997B10
     pthread 0x3637 kernel thread id 63807 (slot 249) pool 4
waiting on ThCond 0x206F398C (0x206F398C) (AllocCursorSearchCondVar), reason 'Waiting for prior cursor
  search'
     waiting for 0.012217137 seconds
     thread traceback:
       0x100080C4 ThCond::wait(int,const char*) + 0x1B0
       0x1011F110 SGAllocMap::waitForCursorSearch(AllocCursor*) + 0x178
       0x101087BC SGAllocMap::allocLocalReplica(SGDescCache*,unsigned int,int,...) + 0xC04
       0x101075AC SGAllocMap::allocReplica(SGDescCache*,const fsDiskAddr&,int,int,...) + 0x3C0
       0x101E4B94 SGAllocSet::allocReplica(SGDescCache*,int,const fsDiskAddr&,int,int,...) + 0x94
       0x10416D00 FileAllocator::allocOneBlock(int,int,fsDiskAddr*,long long*) + 0x2F4
       0x10416F54 FileAllocator::alloc(int,int,int,fsDiskAddr*,long long*) + 0x1CC
       0x101CC500 LLOpenFile::copyInodeBlock(int,unsigned int,unsigned int) + 0x450
       0x101CB3B8 LLOpenFile::mhSnapshotUpdate(RpcContext*,char*) + 0x61C
       0x10085880 ServiceRegister::handleMsg(RpcContext*,char*,Errno&) + 0x12C
       0x100897B8 tscHandleMsg(ClusterConfiguration*,TscMsgHeader*,MsgDataBuf*,...) + 0x1DC
```

- Wait reason changed already – this thread is actively doing something.
- Run “mmfsadm dump threads” a few more times – similar picture
- Evidently this thread is stuck in an infinite loop
- A bug in GPFS, but with an environmental assist: in this case fs was replicated, and metadata disks in one of the failure groups filled up
- Restart GPFS on sgmgr, run mmdf, try to free up some space

Following revokes across nodes

- What's happening with this revoke?

```
0x112712D88 waiting 20494.712973750 seconds, RenameHandlerThread: on ThCond 0x1142A7230
   (0x1142A7230) (MsgRecordCondvar), reason 'RPC wait' for tmMsgRevoke on node 192.168.200.55 <c0n0>

msg_id 77329560, service 13.1, msg_type 18 'tmMsgRevoke', n_dest 1, n_pending 1
  this 0x1142A7088, n_xhold 1, cl 0, cbFn 0x0, age 20494 sec
    sent by 'RenameHandlerThread' (0x112712D88)
    dest <c0n0>          status pending   , err 0, reply len 0
```

- On the target node:

```
Messages being serviced:
      msg_id 77329560  thread 21364915  age 20494.214  tmMsgRevoke

ReceiveWorkerThread: desc 0x1140D1DA8 handle 0x1234C08E parm 0x1142DE2E8 highStackP 0x1140F6780
 pthread 0x8E8F kernel thread id 21364915 (slot 2608) pool 4
 waiting on ThCond 0x1142DE310 (0x1142DE310) (RcvWorkerCondvar), reason 'idle waiting for work'
```

- **Was the RPC lost? Nope**

```
revokeReq list:
RevokeReq 0x1172B08A8 Mode ww Flags 0x1024 err 0
           msgHdr: magic F3689038 from <c0n1> msg_id 77329560 type 18 len 72

Revoke wait table:
    type Mnode key C0A8C8375189ABE9:0000000000067F93:0000000000000000, state LockWait, reqMode ww
      reqFlags 0x00001024, sequence_number 2369457, revokeReq 0x1172B08A8
```

Recovering from a deadlock

- **In most cases, the only practical way to break up a deadlock is to shoot mmfsd on some of the nodes**
- **If the deadlock is caused by an external factor (e.g. stuck IO) restarting GPFS alone may not help**
- **In some cases (e.g. restarting GPFS on sgmgr) this may let things continue without any applications getting EIO**
- **Most deadlocks are fairly localized: shooting mmfsd on one of two nodes will help**
- **4.1+: try 'mmcommon breakDeadlock'**
- **How to shoot mmfsd?**
  - reboot
  - mmshutdown/mmstartup
  - mmfsadm test assert
  - kill <mmfsd pid> (try to avoid kill -9)

Recovering from deadlocks (cont.)

- **In many cases, a deadlock is local to a particular file system, and restarting GPFS on sgmgr node blindly often helps**
- **However, if some of the client nodes are suffering from environmental problems (e.g. disk issues), those need to be identified and isolated or fixed – need to follow the trail of waiters – remember to look in *all* clusters in a multi-cluster scenario**
- **If group protocol or SG-wide broadcast is stuck, shooting GPFS on non-replying nodes may help**
- **DO NOT use mmchmgr during a deadlock – shoot mmfsd or reboot instead**
- **If an mm command is stuck, killing the command and restarting it won't help – need to find and fix the cause of the original hang**

Questions?

Q: Is GPFS troubleshooting an art or a science?

A: Yes