Welcome to …

Supporting Support

Microsoft Global Escalation Summit, 2009

Presented by :

T.Roy

CodeMachine Inc.

www.codemachine.com

Speaker Introduction

- T.Roy
  - Masters Degree in Computer Engineering
  - 20 years experience in system software development
  - 10 years international teaching experience
  - Specialization in Windows Driver Development and Debugging
  - Founder of CodeMachine
- CodeMachine Inc.
  - Consulting and Training Company
  - Based in Palo Alto, CA, USA
  - Custom Driver Development and Debugging Services
  - Corporate on-site training in Windows Internals, Networking, Device Drivers and Debugging
  - http://www.codemachine.com

CodeMachine Courses

- Internals Track
    - Windows User Mode Internals
    - Windows Kernel Mode Internals
- Debugging Track
    - Windows Basic Debugging
    - Windows User Mode Debugging
    - Windows Kernel Mode Debugging
- Development Track
    - Windows Network Drivers
    - Windows Kernel Software Drivers
    - Windows Kernel Filter Drivers
    - Windows Driver Model (WDM)
    - Windows Driver Framework (KMDF)

Why This Talk…

- The problem
  - Developer and Technical support folks have to deal with crashes and hangs day in & day out
  - In many cases ONE crash dump is all they have to root cause a problem
  - Often critical pieces of information that are required to nail down a problem is missing from that one crash dump

So what can the developers do to help the support folks do their job better and faster ?

- This talk covers some simple programming techniques
  - To improve diagnosability of your code
  - To help support folks get more out of the crash dumps
  - To enable them determine root cause of an issue from a single crash dump
    - So they don't have to ask the customer to reproduce the problem again to get them yet another crash dump

Key Takeaways…

- In-memory data logging
- Preventing overwrite of important information
- Making data easily locatable and identifiable
- Logging relevant data and presenting it properly
- Complementing the OS’s data tracking
- Understanding OS support for run time data capture
- Capturing performance related data

Techniques discussed here clearly apply to kernel mode drivers but …

They can be easily adapted to user mode code as well

Agenda

- Memory Trace Buffers
- Freed Pool Memory
- Structure Tracking
- Information Presentation
- State Logging
- Lock Owners
- Run Time Stack Traces
- Timing Information

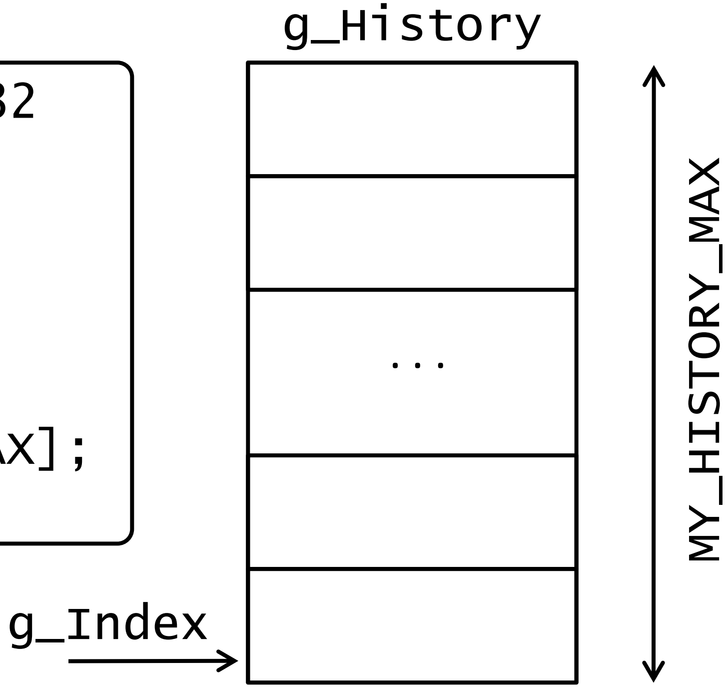
Memory Trace Buffers

- Crash Dumps offer a temporal snapshot of a system
  - Provides no historical information
  - Often historical events are critical to root causing issues
- Log run time information into memory trace buffers
  - Non-Paged buffers available in kernel and complete dumps
  - Use circular buffer with wrap around feature
    - Retains most recent events by replacing old ones
    - Good compromise between memory usage & history length
  - Avoid locking when logging events in memory
    - Costly due to IRQL changes
    - Use Interlocked operations instead
- Trace buffer information can be retrieved using ‘dt –a’
- Enable/Disable logging code using registry keys
- Kernel internally uses this type of logging
  - Example : In-Flight Recorder (IFR) Logs
  - Example : PnP State History inside Device Node (DEVNODE)

Implementation

- Data Structures

```
#define MY_HISTORY_MAX                    32

typedef struct _MY_HISTORY {
       PVOID Information;
} MY_HISTORY, *PMY_HISTORY;

MY_HISTORY g_History[MY_HISTORY_MAX];
ULONG g_Index = 0;
```



- Function

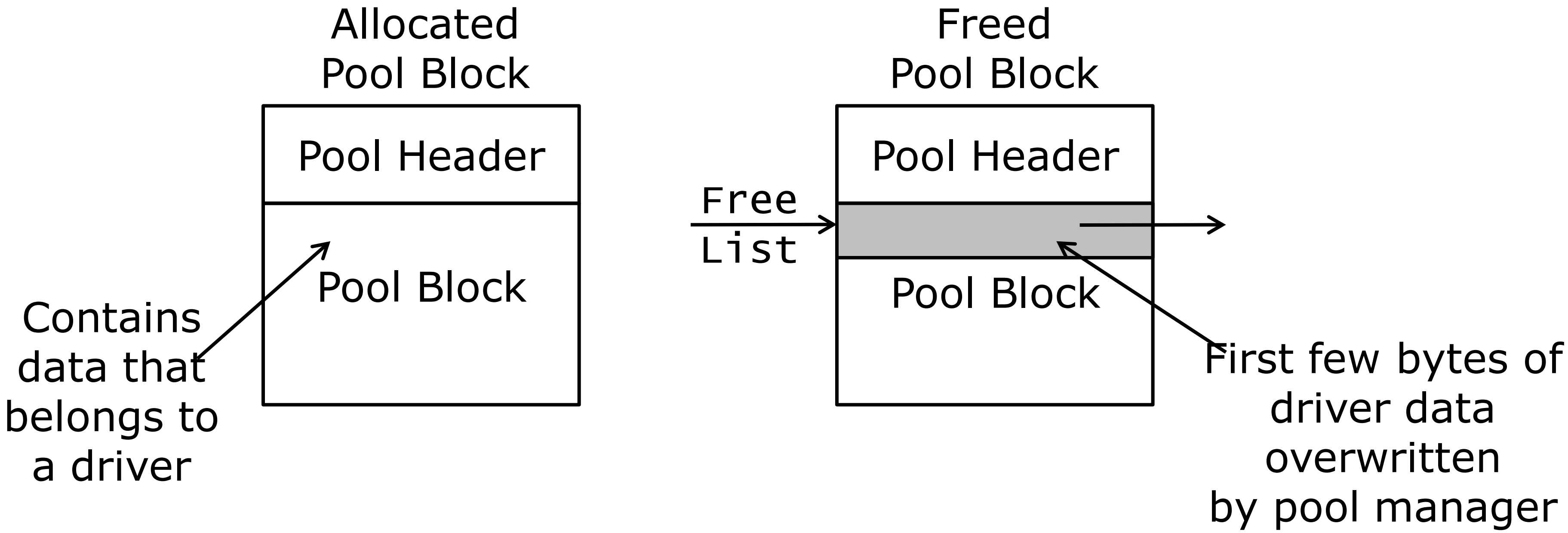
```
LoggingFunction( PVOID Information )
{
       ULONG Index = InterlockedIncrement (&g_Index);
       PMY_HISTORY History =
              &g_History[Index % MY_HISTORY_MAX];
       History->Information = Information;
}
```

Case Study

- WDM Filter Driver for Modem Stack
  - Sitting between the modem driver and serial driver
  - Filtering Read, Write and Device I/O Control IRPs
- System bug-checked at random points whenever the serial device being filtered was accessed
  - Crash dumps pointed to the kernel’s timer related code
  - Unfortunately timers were used all over the place in the driver
- Added Trace Buffer to log all IRPs filtered by driver
  - Each IRP entry contained
    - IRP Pointer
    - Major Function Code
    - Major Function specific information
- New crash dumps helped establish relationship between a particular IOCTL IRP and subsequent crash
  - Problem was traced to un-cancelled timer on thread’s stack
  - Bug was in error handling code path

Freed Pool Memory

- Memory that is freed back to pool is owned by the memory manager
  - Pool Manager uses data area of freed pool block to track the freed block in internal pending and free lists
    - 1st pointer sized value used for Pending Block List
    - 1st 2 pointer sized values used for Freed Block List
  - Driver stored data is overwritten by these pointers
  - Cannot retrieve this data when examining freed pool blocks



Preserving Data

- Avoid maintaining critical data in first 2 pointer sized locations within pool allocated structures
  - 'Critical' refers to any data that may be important during crash dump analysis
- To achieve this
  - Declare the first field of such structures as 'Reserved'

```
typedef struct _MY_STRUCT {
       LIST_ENTRY Reserved; // don't use me
       . . .
} MY_STRUCT , *PMY_STRUCT;
```

- Does not address the issue of a pool block being
  - Freed back to pool
  - Immediately reallocated
  - Completely overwritten by the next owner

Caching freed structures

- Problems in drivers typically related to the most recent data structures that were operated upon
  - Structures are freed back to pool after processing is complete
  - Attempt to retrieve state information from the freed data structure generally futile
  - Information overwritten as freed pool memory is reallocated
- Make a copy of the contents of the structure just before they get freed
  - May not need to copy complete structure contents
    - i.e. If structure is too big then only cache fields relevant to debugging
- Maintaining a cache of last 2 freed structures of each structure type used in the driver is typically adequate

Implementation

- Caching the structure contents in a global array

```
#define MY_CACHE_SIZE     2
MY_STRUCT g_MyStructCache[MY_CACHE_SIZE];

VOID FreeMyStruct ( PMY_STRUCT pMyStruct)
{
      // backup the previously freed structure
      g_MyStructCache [1] = g_MyStructCache [0];

      // cache the structure we are about to free
      g_MyStructCache [0] = *pMyStruct;

      // now that the contents are cached, free it
      ExFreePool ( pMyStruct );
}
```

- Does incur the cost of two copies at every free
  - This cost can be mitigated with optimization

Case Study

- Transport Driver Interface (TDI) Filter Driver
  - Intercepted TCP traffic on ports like HTTP, SMTP, POP3 etc
  - Each open socket represented by a socket context structure
    - Allocated from non-paged pool during a bind() operation
    - Freed when socket was closed by the application
- Freed socket context structures were cached
  - Contents of socket context structure just before being freed were retained in memory
- Intermittent hangs in IE, Firefox, Outlook
  - Analysis of crash dumps generated after hang, established temporal relationship of hangs to socket close operations
- Investigation of cached socket context structures in crash dumps revealed a synchronization issue
    - New socket I/O was being queued just before close
    - Request was never processed, blocking application indefinitely

Structure Tracking

- Divers are asynchronous in nature
  - Process multiple requests at the same time
  - Each request can be in a different processing stage
- Hard to track down request or memory leaks in a production environment
  - Without enabling special tools like Driver Verifier
- Maintain dynamically allocated structures in a list
  - Maintain all instances of a structure of a particular type in a separate linked list
  - Add to list after allocation and remove from list before freeing
  - Counter keeps track of the number of requests in progress
- When unloading driver verify the list is empty
  - Else log an error, at least you will know there is a problem
- List can be walked in a debugger using ‘dt –l’ command

Implementation

```
KSPIN_LOCK MyListLock;
LIST_ENTRY MyListHead;
ULONG      MyListCount;

typedef struct _MY_STRUCT {
      LIST_ENTRY Link;
      . . .
} MY_STRUCT, *PMY_STRUCT;
```

```
AllocateMyStruct()
{
      // allocate and initialize pMyStruct
      KeAcquireSpinLock ( &MyListLock, &Irql );
      InsertTailList ( &MyListHead, &pMyStruct->Link );
      MyListCount++;
      KeReleaseSpinLock ( &MyListLock, Irql );
}
```

```
kd> dt poi(MyListHead) _MY_STRUCT -l Link.Flink
```

Debugger
Command

State Logging

- Is this structure currently queued ?

```
kd> dt mydriver!_MY_STRUCT 87a8a7b0  Links
   +0x080 Links : _LIST_ENTRY [ 0x81d4d990 - 0x87ed7560 ]
```

  - Hard to tell which queue (if any) a structure is sitting in looking at the LIST_ENTRY contents
- Add a ‘State’ field that contains this information
  - When inserting and removing the structure from a list update this ‘State’
  - Must be updated with the queue lock held
  - Use an ‘enum’ instead of a ‘#define’
    - Enables the debugger to show you meaningful state as opposed to a meaningless numeric value
- When processing system defined structures (e.g. IRP)
  - Associate a driver defined context with the system structure
  - Driver stores state in context & links it to lists for debugging

Implementation

- Include a ‘State’ field in the structure to track which queue it is currently in

```
typedef enum _MY_STATE {
        NotQueued = 0,
        InDeviceQueue = 1,
        InCompletionQueue = 2
} MY_STATE;

typedef struct _MY_STRUCT {
        . . .
        LIST_ENTRY Links;
        MY_STATE State;
        . . .
} MY_STRUCT, PMY_STRUCT;
```

- This time it is easy to tell which queue it is in

```
kd> dt mydriver!_MY_STRUCT 87a8a7b0 Links State
   +0x080 Links : _LIST_ENTRY [ 0x81d4d990 - 0x87ed7560 ]
   +0x088 State : 2 ( InCompletionQueue )
```

Case Study

- NDIS USB Driver
  - NDIS sends NBL to driver in transmit path
  - NBL goes through multiple stages in driver & then completed
    - Priority Queuing
    - Point to Point Protocol (PPP) State Machine
    - Hayes Modem AT Command State Machine
    - USB Device Stack
- DRIVER_POWER_STATE_FAILURE (9f) on Vista
  - Cause of this failure is typically NBLs pending in driver
  - Preventing NDIS from putting system in lower power state
- Challenge was to locate the NBL that was stuck
- Structure Tracking & State Logging to the rescue
  - Driver associated context structure with NBL
  - Context structure linked to a per adapter list
  - Context maintained processing stage the NBL was currently in

Information Presentation

- Windows software trace Pre-Processor (WPP) offers a low overhead mechanism for run time logging
  - Developers are strongly encouraged to use this facility
- But then what would you rather see in a WPP trace ?

```
MyRead() called
```

OR

```
MyRead(#253, Buffer=0xff801000 Offset=1200 Length=4096)
```

- Log state information that may be useful in debugging
  - Instead of just meaningless text messages
  - Log related structures together, to get to one from the other
  - Log state of data at request entry and exits points in driver
- Debugging should be data centric not code centric
  - Especially TRUE for a crash dump
  - No execution and no execution control
  - All you have is snapshot of data structures to examine

Sequence Numbers

- Which one is easier to comprehend and track ?

```
Request @ 0xffff8569004001870
```

OR

```
Request # 27 @ 0xffff8569004001870
```

- Associate a sequence number with structures
  - Store sequence number in the structure itself
    - Generated from a globally incrementing sequence counter
  - Include this sequence number along with the structure pointer in the traces
- Applications
  - Can be used to match request ingress and egress
    - IRPs arriving in a driver in at a DispatchRoutine
    - IRPs exiting from a driver through IoCompleteRequest()
  - Can be used to match frequent allocations with frees
  - Can be used to track a structure as it flows through various processing stages within a driver

Implementation

```
// global epoch counter
ULONG g_SequenceCounter = 0;

// structure to be tagged with epoch
typedef struct _MY_STRUCT {
       . . .
       ULONG Sequence;
       . . .
} MY_STRUCT, *PMY_STRUCT;
```

```
PMY_STRUCT pMyStruct;

pMyStruct->Sequence =
       InterlockedIncrement ( &g_SequenceCounter );
```

Case Study

- Custom application talking to a USB Input Device
  - Application sends read request (IRP) to custom USB driver
  - Driver builds URB and associates it with the request (IRP)
  - IRP sent down to USB Bus Driver
  - IRP completion routine queues a work item to process IRP
  - Worker routine performs post-processing and completes IRP
- Application hangs under heavy load conditions
- WPP tracing in the USB driver comes to the rescue
  - Allocate and associate context with each request (IRP)
  - Store request sequence number in this context
  - Log sequence number, IRP + URB pointers etc. in WPP trace
  - Log in dispatch routine, completion routine and work item
- Examined WPP traces from stress test run
  - Application hang attributed to out-of-order completion of IRPs from worker thread context

Lock Owner

- ERESOURCEs, Fast Mutexes & Mutexes store the owning thread identifier
  - SpinLocks don't
  - Hard to track down spin lock owner during a livelock
    - LiveLock is when all CPUs are spinning on locks
- Store owning Thread ID along with lock
  - When declaring a spin lock declare another variable to store the current lock owner
  - Call these APIs through a wrapper, instead of calling directly
    - KeAcquireSpinLock()/KeAcquireInStackQueuedSpinLock()
    - KeReleaseSpinLock()/KeReleaseInStackQueuedSpinLock()
  - Wrapper should store the current thread ID as soon as the KeAcquireXXX() returns.
    - Use PsGetCurrentThreadId()
- Helps identify lock owners in a crash dump

Implementation

```
KSPIN_LOCK MyLock;
HANDLE MyLockOwner;
```

```
AcquireLock( )
{
        KeAcquireSpinLock ( &MyListLock, &Irql );
        MyLockOwner = PsGetCurrentThreadId();
}
```

```
ReleaseLock( )
{
        MyLockOwner = NULL;
        KeReleaseSpinLock ( &MyListLock, Irql );
}
```

Run Time Stack Traces

- Breakpoints used to obtain stacks during live debug
- Situations under which breakpoints are not feasible
  - Timing sensitive issues
  - Breakpoint triggers too often
  - Live Debug not possible
  - Debugging production systems
- Capture run-time stack traces
  - Kernel Mode API RtlCaptureStackBackTrace()
  - Caller specifies number of frames to skip and capture
  - Returns stack fingerprint
    - Used to identify duplicate stacks and store only unique ones
  - Does not work if stack contains FPO functions
- Stack displayed by 'dps' or 'dt' command in a dump
- Used internally by multiple gflags options
  - Kernel Mode Stack Trace Database (kst) etc.

Prototype

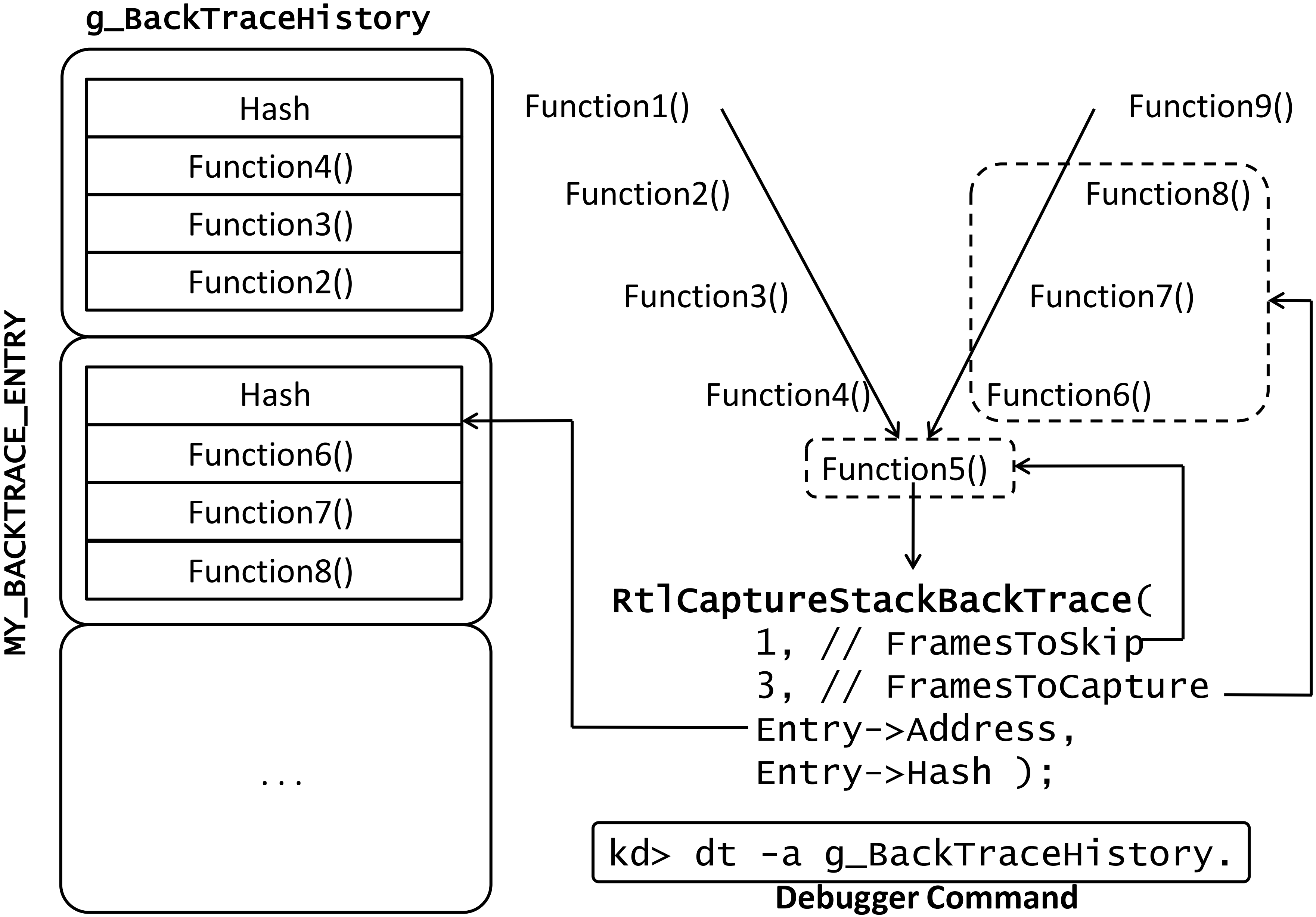
```
USHORT WINAPI
RtlCaptureStackBackTrace(
   ULONG FramesToSkip,
   ULONG FramesToCapture,
   PVOID* BackTrace,
   PULONG BackTraceHash );
```

Implementation

```
 #define RET_ADDR_COUNT    3

typedef VOID (*PRET_ADDR)(VOID);
typedef struct _MY_BACKTRACE_ENTRY {
       ULONG Hash;
       PRET_ADDR Address[RET_ADDR_COUNT];
} MY_BACKTRACE_ENTRY, *PMY_BACKTRACE_ENTRY;

#define MY_BACKTRACE_COUNT 1024

ULONG g_BackTraceIndex = 0;
MY_BACKTRACE_ENTRY g_BackTraceHistory[BACKTRACE_COUNT];
```

```
VOID CaptureStack ( VOID )
{
       ULONG Index = InterlockedIncrement (&g_BackTraceIndex);
       PMY_BACKTRACE_ENTRY Entry =
              &g_BackTraceHistory[Index % MY_BACKTRACE_COUNT];
       RtlCaptureStackBackTrace (
              1, 3, Entry->Address, &Entry->Hash );
}
```

Implementation



g_BackTraceHistory

MY_BACKTRACE_ENTRY

Hash
Function4()
Function3()
Function2()

Hash
Function6()
Function7()
Function8()

...

Function1()
Function2()
Function3()
Function4()
Function5()
Function6()
Function7()
Function8()
Function9()

```
RtlCaptureStackBackTrace(
        1, // FramesToSkip
        3, // FramesToCapture
        Entry->Address,
        Entry->Hash );
```

```
kd> dt -a g_BackTraceHistory.
```

Debugger Command

Case Study

- File System Mini-Filter Driver
  - Allocates stream context (one such instance per open file)
  - Context was referenced and dereferenced all over the driver
- Reference count leak in stream context
  - Preventing filter from getting unloaded
- Added call to RtlCaptureStackBackTrace() in both reference and dereference functions
  - Both stack trace & the current reference count were stored
  - Separate stack trace buffer was allocated for every stream context from NPP and associated with the context
- New crash dumps contained necessary stacks, but
  - Number of entries we have used initially were not large enough to capture the leak
- Doubled the number of entries to 64
  - Caught the leak in error handling code path of a function

Timing Information

- Performance Issues
  - Different in nature from crashes and hangs
  - Difficult to track down from a crash dump
  - Profiling tools KernRate yield much better results
- Logging timing information can help immensely
  - How long did it take your driver to process a request
  - How long is you driver holding waitable locks
  - How long was a thread waiting inside a driver
  - How long is it taking your driver to search a list
- Measuring time
  - KeQueryPerformance[Counter|Frequency]()
  - KeQueryInterruptTime()/KeQueryTimeIncrement()
  - KeQueryTickCount()/KeQueryTimeIncrement()
- Log this information so that it is available in a dump
  - Compute and store peak timing information

Conclusion

- Small little changes improve driver debugability
- Retain critical historical data in circular buffers
- Keep the cost of logging as low as possible
- Preserve information before it gets overwritten
- Carefully choose what information to log
- Alternatives to live debugging & breakpoints do exist
- Log timing information for performance issues

Questions ?

Please email your questions or comments to

msges2009@codemachine.com