Developing Kernel Drivers With Modern C++

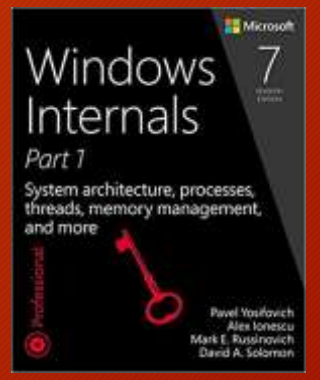
Pavel Yosifovich

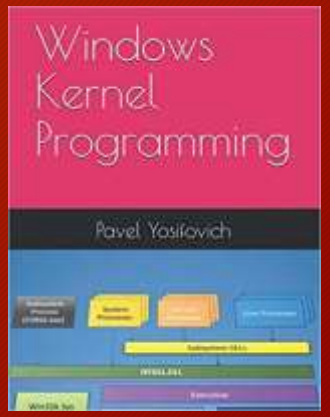
@zodiacon

zodiacon@live.com

About Me

- Developer, Trainer, Author and Speaker
- Book author
  - “Windows Kernel Programming” (2019)
  - “Windows Internals 7th edition, Part 1” (co-author, 2017)
  - “WPF 4.5 Cookbook” (2012)
- Pluralsight author
- Author of several open-source tools (http://github.com/zodiacon)
- Blogs: http://blogs.microsoft.co.il/pavely, http://scorpiosoftware.net





Agenda

- Kernel Programming
- User Mode vs. Kernel Mode
- C++ in the Kernel
- Summary
- Q & A

Kernel Programming

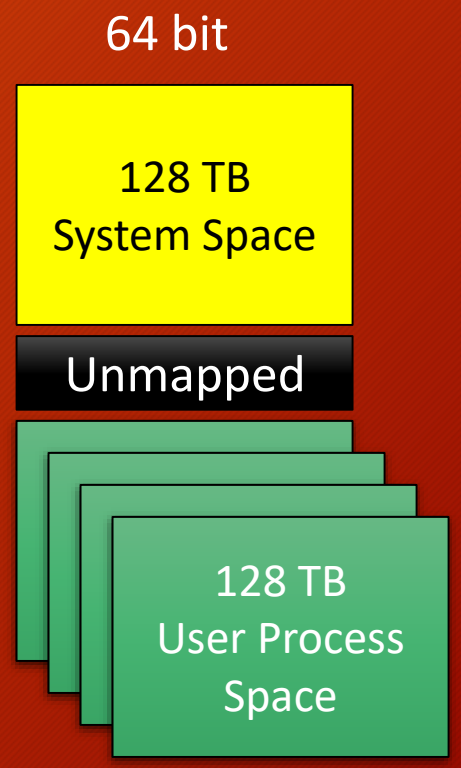
- Kernel programming refers to Kernel device drivers running in kernel mode (and kernel space)
- Classically written in C
- All Microsoft driver samples are still written in C
- Since 2012, Microsoft officially supports using C++ in drivers
  - Shame MS not taking advantage of this themselves (at least not publicly)
- However, using C++ in kernel mode is not the same as user mode

User Mode vs. Kernel Mode Development

	User Mode	Kernel Mode
Unhandled Exceptions	Unhandled exceptions crash the process	Unhandled exceptions crash the system
Termination	When process terminates, all private memory and resources are freed automatically	If driver unloads without freeing everything it was using, there is a leak, only resolved in the next boot
Return values	API errors sometimes ignored	Should (almost) never ignore errors
IRQL	Always `PASSIVE_LEVEL` (0)	May be `DISPATCH_LEVEL` (2) or higher
Bad coding	Typically localized to the process	Can have system-wide effect
Testing and Debugging	Typically testing and debugging done on the developer machine	Must use another machine for testing and debugging
Libraries	Can use almost any C/C++ library (e.g. STL)	Most standard libraries cannot be used
Exception Handling	Can use C++ exceptions or Structured Exception Handling (SEH)	Only SEH can be used
C++ Usage	Full C++ runtime available	No C++ runtime

C++ in the Kernel

- No runtime
  - No exception handling
  - No new/delete operators
  - No constructors called for global objects
  - The standard library is (almost) useless
- All other features work normally
  - `auto`, range-based `for`, lambdas, …
  - Constructors / destructors, virtual functions
  - Templates of all sorts
  - Move semantics
  - Operator overloading



RAII

- Resource Acquisition Is Initialization
- C++ cleanup pattern
- Just as important (if not more so) in the kernel

RAII Example


```
template<typename TLock>
struct AutoLock {
    AutoLock(TLock& lock) : _lock(lock) {
        _lock.Lock();
    }

    ~AutoLock() {
        _lock.Unlock();
    }

private:
    TLock& _lock;
};
```

```
class FastMutex {
public:
    void Init();

    void Lock();
    void Unlock();

private:
    FAST_MUTEX _mutex;
};
```

```
AutoLock<FastMutex> locker(MyMutex); // C++ 14
AutoLock locker(MyMutex);            // C++ 17
```

```
void FastMutex::Init() {
    ExInitializeFastMutex(&_mutex);
}

void FastMutex::Lock() {
    ExAcquireFastMutex(&_mutex);
}

void FastMutex::Unlock() {
    ExReleaseFastMutex(&_mutex);
}
```

More RAII Examples

- Handles
- (in Mini-filter) File Name Information
- (in Mini-filter) Contexts
- Memory allocations

Dynamic Memory Allocations

- Drivers allocate memory from the kernel memory pools
  - Non paged pool or paged pool
- Use **`ExAllocatePoolWithTag`** to allocate
  - Tag is a 4-byte value useful for identifying the allocating driver (or a component that is part of a driver)
  - Typically specified as a 4-character ASCII string
  - Can view with the *Poolmon* WDK tool or the kernel debugger
- Use **`ExFreePool`** to free the buffer
- How to invoke ctor/dtor?

```
PVOID ExAllocatePoolWithTag (
    _In_ POOL_TYPE PoolType,
    _In_ SIZE_T NumberOfBytes,
    _In_ ULONG Tag);
```

Memory Allocation Example (1)


```
void* operator new(size_t size, POOL_TYPE type, ULONG tag = 0);
```

```
void* operator new(size_t size, POOL_TYPE type, ULONG tag) {
    auto p = tag == 0 ? ExAllocatePool(type, size) : ExAllocatePoolWithTag(type, size, tag);
    if (p == nullptr) {
        KdPrint(("Failed to allocate %d bytes\n", size));
    }
    return p;
}
```

```
template<typename T>
using vector = kvector<T, DRIVER_TAG>;

struct MyData {
    int Count;
    FastMutex Lock;
    vector<PDEVICE_OBJECT> Devices;
};
```

```
auto data = new (NonPagedPool, DRIVER_TAG) MyData;
```

Memory Allocation Example (2)


```
// placement new
void* operator new(size_t size, void* p);
```

```
void* operator new(size_t, void* p) {
    return p;
}
```

```
auto data = static_cast<MyData*>(ExAllocatePoolWithTag(
    NonPagedPool, sizeof(MyData), DRIVER_TAG));
new (data) MyData;
```

```
data->~MyData();
ExFreePool(data);
```

Strings

- Most kernel APIs work with UNICODE_STRING structures
  - Length and MaximumLength are in bytes
- UNICODE_STRING is not necessary the owner of its characters
  - An owner-focused wrapper will not be useful
- Can create a generic string class
- Provide conversion to a UNICODE_STRING
  - Maintain ownership

```
typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWCH   Buffer;
} UNICODE_STRING;
typedef UNICODE_STRING *PUNICODE_STRING;
typedef const UNICODE_STRING *PCUNICODE_STRING;
```

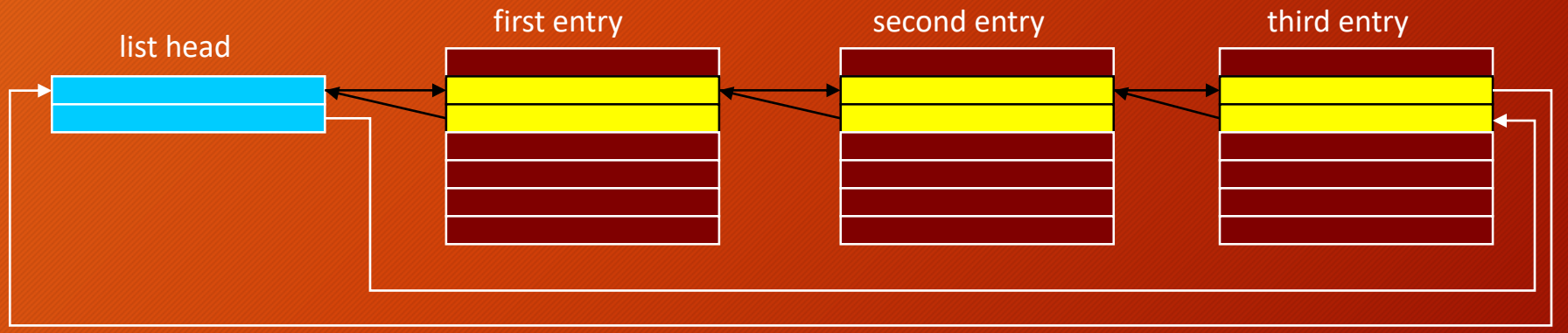
Strings

14

Demo

Lists and Queues

- The kernel uses circular doubly–linked lists in many data structures
  - `LIST_ENTRY` structure
  - API is fully documented
  - Drivers can use the same structures
- Normally part of a larger structure
- WDK headers provide the CONTAING_RECORD macro



Linked Lists


```
#include "AutoLock.h"
#include "FastMutex.h"

template<typename T, typename TLock = FastMutex>
class LinkedList {
public:
    void Init() {
    }

pri

};
```

```
struct MyData {
    int Data;
    LIST_ENTRY Entry; // must be named "Entry"
    int MoreData;
};

LinkedList<MyData> MyList;
```

```
MyList.Init();
auto item = new (PagedPool, DRIVER_TAG) MyData;
item->Data = 8;
item->MoreData = 42;

MyList.PushBack(item);
```

```
// expects a LIST_ENTRY named "Entry"

void PushBack(T* item) {
    AutoLock locker(_lock);
    InsertTailList(&_head, &item->Entry);
}

void PushFront(T* value) {
    AutoLock locker(_lock);
    InsertHeadList(&_head, &item->Entry);
}

T* RemoveHead() {
    AutoLock locker(_lock);
    auto entry = RemoveHeadList(&_head);
    return CONTAINING_RECORD(entry, T, Entry);
}

T* GetHeadItem() {
    AutoLock locker(_lock);
    auto entry = _head->Flink;
    return CONTAINING_RECORD(entry, T, Entry);
}
```

Components

- Some scenarios require objects to be
  - Reference counted
  - Support functionality querying
- The Component Object Model (COM) provides these ideas
  - Implementation in user mode (ATL, WRL)
- What about something like `std::shared_ptr<>`?
  - Not good enough
  - No way to query for functionality
- Solution: implement it ourselves!

Kernel COM


```
struct IFilter abstract : IComponent {
    enum { IID = 10 };

};
```

```
struct MyFilter : ComponentBase<IFilter, IProcessNotify> {
    template<POOL_TYPE type = PagedPool, ULONG tag = 0>
    static IComponent* Create() {
        return static_cast<IFilter*>(new (type, tag) MyFilter);
    }

    void FilterRequest(PIRP Irp) override {
        // do something
    }

    void ProcessCreated(H
        // do something
    }

private:
    MyFilter() {}
};
```

```
IFilter* filter;

auto c = MyFilter::Create<NonPagedPool>();
if (c == nullptr)
    return STATUS_INSUFFICIENT_RESOURCES;

c->QueryInterface(IFilter::IID, reinterpret_cast<void**>(&filter));
if (filter) {
    filter->FilterRequest(...);
    filter->Release();
}
```

Summary

- Kernel code no longer stuck with C
- C++ fully supported
- Using C++ properly in the kernel can lead to more robust drivers
- There is no going back!

Thank you!