Глава 8: Процессы и потоки, уведомления

Один из мощных механизмов, доступных для драйверов ядра - это возможность получать уведомления, когда происходят важные события. В этой главе мы рассмотрим некоторые из этих событий, а именно создание процесса.
А также создание и разрушение потоков и загрузка образов.
В этой главе:
• Уведомления о процессах.
• Регистрация уведомлений о процессах.
• Предоставление данных в пользовательском режиме.
• Уведомления о потоках.
• Уведомления о загрузке образов.
• Упражнения.

Уведомления о процессах
Каждый раз, когда процесс создается или уничтожается, заинтересованные драйверы могут быть уведомлены ядром об этом факте. Это позволяет драйверам отслеживать процессы, возможно, связывая некоторые данные с этими процессами.

Как минимум, это позволяет драйверам отслеживать создание/уничтожение процесса в в реальном времени. Под «в реальном времени» я подразумеваю, что уведомления отправляются «в оперативном режиме».

Как часть создания процесса драйвер не может пропустить какие-либо процессы, которые могут быть созданы и быстро уничтожены.
В момент создания процессов драйверы также могут остановить создание процесса, возвращая ошибку вызывающего абонента, инициирующего создание процесса.

Этот можно сделать только в режиме ядра.

Основной API для регистрации уведомлений о процессах - PsSetCreateProcessNotifyRoutineEx, определяется так:

```
NTSTATUS
PsSetCreateProcessNotifyRoutineEx (
                _In_ PCREATE_PROCESS_NOTIFY_ROUTINE_EX NotifyRoutine,
                _In_ BOOLEAN Remove);
```

Первый аргумент - это процедура обратного вызова драйвера, имеющая следующий прототип:

```
typedef void
                (*PCREATE_PROCESS_NOTIFY_ROUTINE_EX) (
                                _Inout_ PEPROCESS Process,
                                _In_ HANDLE ProcessId,
                                _Inout_opt_ PPS_CREATE_NOTIFY_INFO CreateInfo);
```

Второй аргумент PsSetCreateProcessNotifyRoutineEx указывает, является ли драйвер регистрации или отмена регистрации обратного вызова (FALSE указывает на первое). Обычно драйвер вызывает этот API со значением FALSE в подпрограмме DriverEntry и вызывает тот же API с TRUE в его выгрузке.

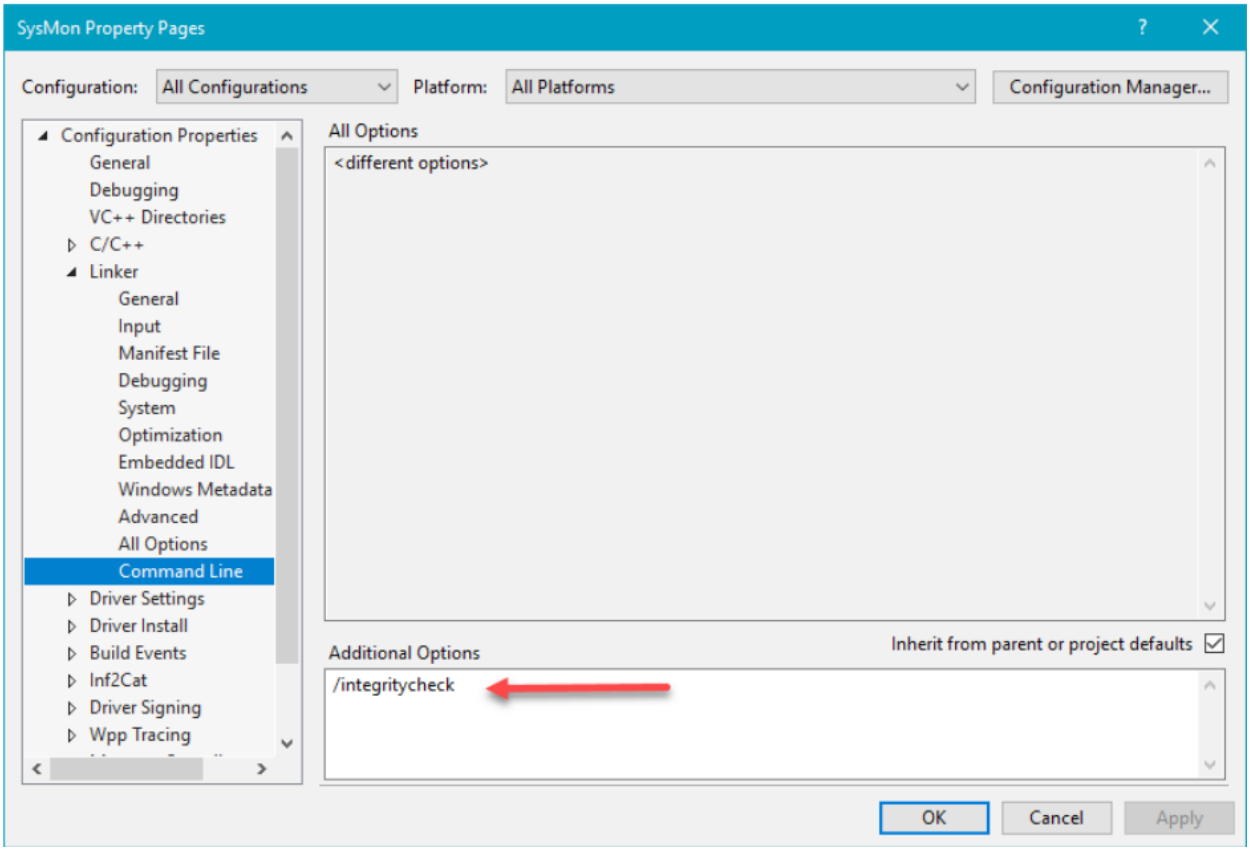
Аргументы процедуры уведомления следующие:
• Процесс - объект процесса вновь созданного процесса или уничтожаемого процесса.
• Идентификатор процесса - уникальный идентификатор процесса. Хотя он объявлен с типом HANDLE, он на самом деле ID.
• CreateInfo - структура, содержащая подробную информацию о создаваемом процессе. Если процесс уничтожается, этот аргумент равен NULL.
Для создания процесса процедура обратного вызова драйвера выполняется создающим потоком. Для выхода из процесса, обратный вызов выполняется последним потоком для выхода из процесса. В обоих случаях обратный вызов вызывается внутри критической области (обычные APC ядра отключены).

Начиная с Windows 10 версии 1607, есть еще одна функция для уведомлений о процессах: PsSetCreateProcessNotifyRoutineEx2. Эта «расширенная» функция устанавливает обратный вызов аналогичен предыдущему, но обратный вызов также вызывается в процессах Pico. Пико процессы используются для размещения процессов Linux для подсистемы Windows для Linux (WSL).

Если драйвер заинтересован в таких процессах, он должен зарегистрироваться в расширенной функции.
Драйвер, использующий эти обратные вызовы, должен иметь флаг IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY в заголовке образа Portable Executable (PE). Без него вызоы функции регистрации возвращает STATUS_ACCESS_DENIED.

В настоящее время Visual Studio не предоставляет пользовательский интерфейс для установки этого флага. Это должно быть установлено в параметрах командной строки компоновщика, параметр /integritycheck.

На рисунке ниже показано как это сделать.



Структура данных, предоставляемая для создания процесса, определяется следующим образом:

```
typedef struct _PS_CREATE_NOTIFY_INFO {
        _In_ SIZE_T Size;
        union {
                _In_ ULONG Flags;
                struct {
                        _In_ ULONG FileOpenNameAvailable : 1;
                        _In_ ULONG IsSubsystemProcess : 1;
                        _In_ ULONG Reserved : 30;
                };
        };
        _In_ HANDLE ParentProcessId;
        _In_ CLIENT_ID CreatingThreadId;
        _Inout_ struct _FILE_OBJECT *FileObject;
        _In_ PCUNICODE_STRING ImageFileName;
        _In_opt_ PCUNICODE_STRING CommandLine;
        _Inout_ NTSTATUS CreationStatus;
} PS_CREATE_NOTIFY_INFO, *PPS_CREATE_NOTIFY_INFO;
```

Вот описание важных полей в этой структуре:
• CreatingThreadId - комбинация потока и идентификатора процесса, вызывающего создание процесса.
• ParentProcessId - идентификатор родительского процесса (не дескриптор).
• ImageFileName - имя файла образа исполняемого файла, доступное, если установлен флаг FileOpenNameAvailable.
• CommandLine - полная командная строка, используемая для создания процесса. Обратите внимание, что это может быть NULL.
• IsSubsystemProcess - этот флаг устанавливается, если этот процесс является процессом Pico. Это может быть только в том случае, если драйвер зарегистрирован в PsSetCreateProcessNotifyRoutineEx2.
• CreationStatus - это статус, который вернется к вызывающему. Здесь драйвер может остановить создание процесса, указав статус отказа (например, STATUS_ACCESS_-DENIED).

Регистрация уведомлений о процессах
Чтобы продемонстрировать как создавать уведомления о процессах, мы создадим драйвер, который собирает информацию о процессе.
А также информацию о создании и уничтожении, а также возможность использования этой информации клиентом пользовательского режима.

Это аналогично таким инструментам, как Process Monitor от Sysinternals, который использует уведомления процесса (и потока) для отчета об активности процесса (и потока). В ходе реализации этого драйвера, мы воспользуемся некоторыми приемами, изученными в предыдущих главах.
Имя нашего драйвера будет SysMon (не связанное с инструментом SysMon от Sysinternals), и он будет хранить всю информацию о создании уничтожении процесса в связанном списке (используя структуры LIST_ENTRY).
Поскольку к этому связанному списку могут обращаться одновременно несколько потоков, нам необходимо защитить его с помощью мьютекса или быстрого мьютекса, мы будем использовать быстрый мьютекс, так как он более эффективен.
Собранные нами данные в конечном итоге попадут в пользовательский режим, поэтому мы должны объявить общие структуры, которые создает драйвер и получает клиент пользовательского режима.

Мы добавим общий заголовочный файл с именем SysMonCommon.h в проект драйвера и определим несколько структур. Мы начнем с
общих заголовков для всех информационных структур, определенных так:

```
enum class ItemType : short {
	None,
	ProcessCreate,
	ProcessExit
};
struct ItemHeader {
	ItemType Type;
	USHORT Size;
	LARGE_INTEGER Time;
};
```

Структура ItemHeader содержит информацию, общую для всех типов событий: тип события, время события (выраженное 64-битным целым числом) и размер полезной нагрузки. Размер важен, так как каждое событие имеет свою информацию. Если мы позже захотим упаковать массив этих событий и (скажем) предоставить их клиенту пользовательского режима, клиент должен знать, где заканчивается каждое событие и следующее начинается.
Когда у нас есть этот общий заголовок, мы можем выводить другие структуры данных для конкретных событий.

Давайте начнем с самого простого - выход из процесса:

```
struct ProcessExitInfo : ItemHeader {
	ULONG ProcessId;
};
```

Поскольку нам нужно хранить каждую такую структуру как часть связанного списка, каждая структура данных должна содержать экземпляр LIST_ENTRY, указывающий на следующий и предыдущий элементы. Поскольку эти объекты LIST_ENTRY не должны отображаться в пользовательском режиме, мы определим расширенные структуры, содержащие эти записи, в другой файл, который не используется в пользовательском режиме.
В новом файле с именем SysMon.h мы добавляем общую структуру, которая содержит LIST_ENTRY вместе с фактической структуры данных:

```
template<typename T>
struct FullItem {
	LIST_ENTRY Entry;
	T Data;
};
```

Шаблонный класс используется, чтобы избежать создания множества типов, по одному для каждого конкретного типа события.
Например, мы могли бы создать следующую структуру специально для события выхода из процесса:

```
struct FullProcessExitInfo {
	LIST_ENTRY Entry;
	ProcessExitInfo Data;
};
```

Заголовок нашего связанного списка должен где-то храниться. Мы создадим структуру данных, которая будет содержать все глобальное состояние драйвера вместо создания отдельных глобальных переменных. Вот определение нашей структуры:

```
struct Globals {
	LIST_ENTRY ItemsHead;
	int ItemCount;
	FastMutex Mutex;
};
```

Используемый тип FastMutex - тот же, что мы разработали в главе 6. Наряду с C ++ RAII Оболочка AutoLock, также из главы 6.

Процедура DriverEntry

DriverEntry для драйвера SysMon аналогичен драйверу Zero из главы 7. Нам необходимо добавить регистрацию уведомления о процессе и правильную инициализацию нашего объекта Globals:

```
Globals g_Globals;
extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING) {
	auto status = STATUS_SUCCESS;
	InitializeListHead(&g_Globals.ItemsHead);
	g_Globals.Mutex.Init();
	PDEVICE_OBJECT DeviceObject = nullptr;
	UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\sysmon");
	bool symLinkCreated = false;
	do {
		UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\\Device\\sysmon");
		status = IoCreateDevice(DriverObject, 0, &devName,
				FILE_DEVICE_UNKNOWN, 0, TRUE, &DeviceObject);
		if (!NT_SUCCESS(status)) {
			KdPrint((DRIVER_PREFIX "failed to create device (0x%08X)\n",
					status));
			break;
		}
		DeviceObject->Flags |= DO_DIRECT_IO;
		status = IoCreateSymbolicLink(&symLink, &devName);
		if (!NT_SUCCESS(status)) {
			KdPrint((DRIVER_PREFIX "failed to create sym link (0x%08X)\n",
					status));
			break;
		}
		symLinkCreated = true;
		// register for process notifications
		status = PsSetCreateProcessNotifyRoutineEx(OnProcessNotify, FALSE);
		if (!NT_SUCCESS(status)) {
			KdPrint((DRIVER_PREFIX "failed to register process callback (0x
%08X)\n",
					status));
			break;
		}
	} while (false);
	if (!NT_SUCCESS(status)) {
		if (symLinkCreated)
			IoDeleteSymbolicLink(&symLink);
		if (DeviceObject)
			IoDeleteDevice(DeviceObject);
	}
	DriverObject->DriverUnload = SysMonUnload;
	DriverObject->MajorFunction[IRP_MJ_CREATE] =
```

```
                DriverObject->MajorFunction[IRP_MJ_CLOSE] = SysMonCreateClose;
        DriverObject->MajorFunction[IRP_MJ_READ] = SysMonRead;
        return status;
}
```

Позже мы воспользуемся процедурой отправки чтения, чтобы вернуть информацию о событии в пользовательский режим.

Обработка уведомлений о выходе из процесса

Функция уведомления процесса в приведенном выше коде - OnProcessNotify имеет прототип описанный ранее в этой главе. Этот обратный вызов обрабатывает создание и завершение процесса. Давайте начнем с выхода из процесса, так как это намного проще, чем создание процесса (как мы скоро увидим).

Базовый код обратного вызова выглядит следующим образом:

```
void OnProcessNotify(PEPROCESS Process, HANDLE ProcessId,
                PPS_CREATE_NOTIFY_INFO CreateInfo) {
        if (CreateInfo) {
                // process create
        }
        else {
                // process exit
        }
}
```

Для выхода из процесса у нас есть только идентификатор процесса, который нам нужно сохранить, а также общие данные заголовка.
Во-первых, нам нужно выделить хранилище для всего элемента, представляющего это событие:

```
auto info = (FullItem<ProcessExitInfo>*)ExAllocatePoolWithTag(PagedPool,
                sizeof(FullItem<ProcessExitInfo>), DRIVER_TAG);
if (info == nullptr) {
        KdPrint((DRIVER_PREFIX "failed allocation\n"));
        return;
}
```

Если выделение памяти не удается, то драйвер ничего не может сделать, поэтому он просто возвращается из обратного вызова.
Пришло время заполнить общую информацию: время, тип и размер элемента, и все это легко получить:

```
auto& item = info->Data;
KeQuerySystemTimePrecise(&item.Time);
item.Type = ItemType::ProcessExit;
item.ProcessId = HandleToULong(ProcessId);
item.Size = sizeof(ProcessExitInfo);
PushItem(&info->Entry);
```

Сначала мы работаем с самим элементом данных (минуя LIST_ENTRY) с помощью переменной info.
Далее мы заполняем информацию заголовка: Тип элемента хорошо известен, так как мы находимся в ветви, обрабатывающей уведомление о выходе из процесса.
Время можно получить с помощью KeQuerySystemTimePrecise, который возвращает текущее системное время (UTC, а не местное время) в виде 64-битного целого числа с 1 января 1601 года.
Наконец, размер элемента постоянен и является размером структуры данных, ориентированной

на пользователя (а не размером FullItem <ProcessExitInfo>). Все, что осталось сделать, это добавить новый элемент в конец связанного списка. Для этого мы определите функцию с именем PushItem:

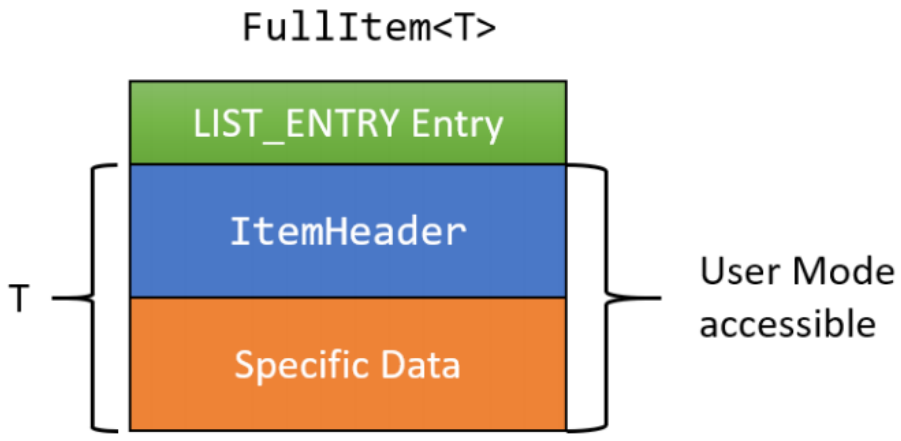
```
void PushItem(LIST_ENTRY* entry) {
	AutoLock<FastMutex> lock(g_Globals.Mutex);
	if (g_Globals.ItemCount > 1024) {
		// too many items, remove oldest one
		auto head = RemoveHeadList(&g_Globals.ItemsHead);
		g_Globals.ItemCount--;
		auto item = CONTAINING_RECORD(head, FullItem<ItemHeader>, Entry);
		ExFreePool(item);
	}
	InsertTailList(&g_Globals.ItemsHead, entry);
	g_Globals.ItemCount++;
}
```

Код сначала получает быстрый мьютекс, так как несколько потоков могут вызывать эту функцию одновременно.
Все после этого делается под защитой быстрого мьютекса.
Затем драйвер ограничивает количество элементов в связанном списке. Это необходимая мера предосторожности, так как нет гарантии того, что клиент сразу воспользуется этими событиями. Драйвер никогда не должен позволять данным потребляться без ограничений, так как это может поставить под угрозу систему в целом. Выбрано значение 1024 здесь совершенно произвольно. Лучше, чтобы этот номер считывался из реестра в драйвера, через сервисный ключ.

Если количество элементов превышает лимит, код удаляет самый старый элемент, по существу обрабатывая связанный список в виде очереди (RemoveHeadList).

Если элемент удален, его память должна быть освобождена.

Макрос CONTAINING_RECORD используется для перехода к началу объекта FullItem <>. Теперь ExFreePool может освободить объект.
На следующем рисунке показано расположение объектов FullItem <T>.



Наконец, драйвер вызывает InsertTailList, чтобы добавить элемент в конец списка, и количество элементов увеличивается.

Нам не нужно использовать атомарные операции увеличения/уменьшения в функции PushItem, потому что манипуляции с подсчетом элементов всегда выполняются под защитой быстрого мьютекса.

Обработка уведомлений о создании процесса

Уведомления о создании процесса более сложны, поскольку объем информации варьируется. Например, длина командной строки для разных процессов разная. Для начала нам нужно понять, что за информация для создания процесса.

Вот первая попытка:

```
struct ProcessCreateInfo : ItemHeader {
      ULONG ProcessId;
      ULONG ParentProcessId;
      WCHAR CommandLine[1024];
};
```

Мы решили сохранить идентификатор процесса, идентификатор родительского процесса и командную строку. С этой структурой можно работать, и с ней довольно легко иметь дело, потому что ее размер известен заранее.

Потенциальная проблема здесь связана с командной строкой. Объявление командной строки постоянного размера просто, но проблематично. Если командная строка длиннее, чем выделено, драйверу придется обрезать ее, возможно, скрывая важную информацию. Если командная строка короче заданного предела, структура тратит память.

Вот еще один вариант, который мы будем использовать в нашем драйвере:

```
struct ProcessCreateInfo : ItemHeader {
      ULONG ProcessId;
      ULONG ParentProcessId;
      USHORT CommandLineLength;
      USHORT CommandLineOffset;
};
```

Мы будем хранить длину командной строки и ее смещение от начала структуры. Сами символы командной строки будут следовать этой структуре в памяти. Таким образом, мы не ограничены длиной командной строки и не тратим память на короткие командные строки. Учитывая это объявление, мы можем начать реализацию для создания процесса:

```
USHORT allocSize = sizeof(FullItem<ProcessCreateInfo>);
USHORT commandLineSize = 0;
if (CreateInfo->CommandLine) {
      commandLineSize = CreateInfo->CommandLine->Length;
      allocSize += commandLineSize;
}
auto info = (FullItem<ProcessCreateInfo>*)ExAllocatePoolWithTag(PagedPool,
            allocSize, DRIVER_TAG);
if (info == nullptr) {
      KdPrint((DRIVER_PREFIX "failed allocation\n"));
      return;
}
```

Общий размер для распределения зависит от длины командной строки (если есть). Пришло время заполнить неизменяющаюся информацию, а именно заголовок, а также идентификаторы процесса:

```
auto& item = info->Data;
KeQuerySystemTimePrecise(&item.Time);
item.Type = ItemType::ProcessCreate;
item.Size = sizeof(ProcessCreateInfo) + commandLineSize;
item.ProcessId = HandleToULong(ProcessId);
item.ParentProcessId = HandleToULong(CreateInfo->ParentProcessId);
```

Размер элемента должен быть рассчитан с учетом базовой структуры и длины командной строки. Далее нам нужно скопировать командную строку на адрес конца базовой структуры и обновить длину и смещение:

```
if (commandLineSize > 0) {
    ::memcpy((UCHAR*)&item + sizeof(item), CreateInfo->CommandLine->Buffer,
        commandLineSize);
    item.CommandLineLength = commandLineSize / sizeof(WCHAR);   // length in WCHARs
    item.CommandLineOffset = sizeof(item);
}
else {
    item.CommandLineLength = 0;
}
PushItem(&info->Entry);
```

Предоставление данных в пользовательском режиме

Следующее, что нужно рассмотреть, - как предоставить собранную информацию клиенту пользовательского режима. Там можно использовать несколько вариантов, но для этого драйвера мы позволим клиенту опросить драйвер с помощью запроса на чтение.

Драйвер заполнит предоставленный пользователем буфер таким количеством событий насколько возможно, пока буфер не будет исчерпан или в очереди не останется событий. Мы начнем запрос на чтение с получения адреса пользовательского буфера с помощью Direct I/O (настроенного в DriverEntry):

```
NTSTATUS SysMonRead(PDEVICE_OBJECT, PIRP Irp) {
	auto stack = IoGetCurrentIrpStackLocation(Irp);
	auto len = stack->Parameters.Read.Length;
	auto status = STATUS_SUCCESS;
	auto count = 0;
	NT_ASSERT(Irp->MdlAddress); // we're using Direct I/O
	auto buffer = (UCHAR*)MmGetSystemAddressForMdlSafe(Irp->MdlAddress,
			NormalPagePriority);
	if (!buffer) {
		status = STATUS_INSUFFICIENT_RESOURCES;
	}
	else {
```

Теперь нам нужно получить доступ к нашему связанному списку и вытащить элементы из его головы:

```
AutoLock lock(g_Globals.Mutex);
// C++ 17
while (true) {
	if (IsListEmpty(&g_Globals.ItemsHead)) // can also check
g_Globals.ItemCount
		break;
	auto entry = RemoveHeadList(&g_Globals.ItemsHead);
	auto info = CONTAINING_RECORD(entry, FullItem<ItemHeader>, Entry);
	auto size = info->Data.Size;
	if (len < size) {
		// user's buffer is full, insert item back
```

```
            InsertHeadList(&g_Globals.ItemsHead, entry);
            break;
        }
        g_Globals.ItemCount--;
        ::memcpy(buffer, &info->Data, size);
        len -= size;
        buffer += size;
        count += size;
        // free data after copy
        ExFreePool(info);
}
```

Сначала мы получаем быстрый мьютекс, так как уведомления процесса могут продолжать поступать. Если список пуст, делать нечего и выходим из цикла. Затем мы вытягиваем головной элемент, и если это больше, чем оставшийся размер пользовательского буфера - скопируем его содержимое (без поля LIST_ENTRY).

Далее продолжаем извлекать элементы из головы до тех пор, пока либо список не станет пустым, либо буфер пользователя не заполнится.

Наконец, мы завершим запрос с любым статусом и установим для информации значение count переменной:

```
Irp->IoStatus.Status = status;
Irp->IoStatus.Information = count;
IoCompleteRequest(Irp, 0);
return status;
```

Нам также нужно взглянуть на процедуру выгрузки. Если в связанном списке есть элементы, они должны быть освобожденным явно, иначе у нас будет утечка:

```
void SysMonUnload(PDRIVER_OBJECT DriverObject) {
        // unregister process notifications
        PsSetCreateProcessNotifyRoutineEx(OnProcessNotify, TRUE);
        UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\sysmon");
        IoDeleteSymbolicLink(&symLink);
        IoDeleteDevice(DriverObject->DeviceObject);
        // free remaining items
        while (!IsListEmpty(&g_Globals.ItemsHead)) {
```

```
            auto entry = RemoveHeadList(&g_Globals.ItemsHead);
            ExFreePool(CONTAINING_RECORD(entry, FullItem<ItemHeader>,
Entry));
      }
}
```

Клиент пользовательского режима

Теперь мы можем написать клиент пользовательского режима, который опрашивает данные с помощью ReadFile и отображает результаты. Основная функция вызывает ReadFile в цикле, немного ожидает, чтобы поток не всегда потреблял ЦПУ.

```
int main() {
      auto hFile = ::CreateFile(L"\\\\.\\SysMon", GENERIC_READ, 0,
            nullptr, OPEN_EXISTING, 0, nullptr);
      if (hFile == INVALID_HANDLE_VALUE)
            return Error("Failed to open file");
      BYTE buffer[1 << 16];
      // 64KB buffer
      while (true) {
            DWORD bytes;
            if (!::ReadFile(hFile, buffer, sizeof(buffer), &bytes, nullptr))
                  return Error("Failed to read");
            if (bytes != 0)
                  DisplayInfo(buffer, bytes);
            ::Sleep(200);
      }
}
```

Функция DisplayInfo должна определять буфер, который ей был предоставлен. Поскольку все события начинаются с общего заголовка, функция различает различные события на основе ItemType. После обработки события, поле размер в заголовке указывает, где начинается следующее событие:

```
void DisplayInfo(BYTE* buffer, DWORD size) {
	auto count = size;
	while (count > 0) {
		auto header = (ItemHeader*)buffer;
		switch (header->Type) {
		case ItemType::ProcessExit:
		{
			DisplayTime(header->Time);
			auto info = (ProcessExitInfo*)buffer;
			printf("Process %d Exited\n", info->ProcessId);
			break;
		}
		case ItemType::ProcessCreate:
		{
			DisplayTime(header->Time);
			auto info = (ProcessCreateInfo*)buffer;
			std::wstring commandline((WCHAR*)(buffer + info-
>CommandLineOffset),info->CommandLineLength);
			printf("Process %d Created. Command line: %ws\n", info-
>ProcessId,
				commandline.c_str());
			break;
		}
		default:
			break;
		}
		buffer += header->Size;
		count -= header->Size;
	}
}
```

Чтобы правильно извлечь командную строку, в коде используется конструктор класса wstring C ++, который может построить строку на основе указателя и длины строки. Форматы вспомогательной функции DisplayTime время в удобочитаемом виде:

```
void DisplayTime(const LARGE_INTEGER& time) {
	SYSTEMTIME st;
	::FileTimeToSystemTime((FILETIME*)&time, &st);
	printf("%02d:%02d:%02d.%03d: ",
			st.wHour, st.wMinute, st.wSecond, st.wMilliseconds);
}
```

Драйвер можно установить и запустить, как описано в главе 4, примерно так:

```
sc create sysmon type= kernel binPath= C:\Book\SysMon.sys
sc start sysmon
```

Вот пример вывода при запуске SysMonClient.exe:

```
xe 0xffffffff -ForceV1
12:07:09.575: Process 6784 Created. Command line: "C:\WINDOWS\system32\cmd.exe"
12:07:09.590: Process 7248 Created. Command line: \??\C:\WINDOWS\system32\conhost.ex\
e 0xffffffff -ForceV1
12:07:11.387: Process 7832 Exited
12:07:12.034: Process 2112 Created. Command line: C:\WINDOWS\system32\ApplicationFra\
meHost.exe -Embedding
12:07:12.041: Process 5276 Created. Command line: "C:\Windows\SystemApps\Microsoft.M\
icrosoftEdge_8wekyb3d8bbwe\MicrosoftEdge.exe" -ServerName:MicrosoftEdge.AppXdnhjhccw\
3zf0j06tkg3jtqr00qdm0khc.mca
12:07:12.624: Process 2076 Created. Command line: C:\WINDOWS\system32\DllHost.exe /P\
rocessid:{7966B4D8-4FDC-4126-A10B-39A3209AD251}
12:07:12.747: Process 7080 Created. Command line: C:\WINDOWS\system32\browser_broker\
.exe -Embedding
12:07:13.016: Process 8972 Created. Command line: C:\WINDOWS\System32\svchost.exe -k\
 LocalServiceNetworkRestricted
12:07:13.435: Process 12964 Created. Command line: C:\WINDOWS\system32\DllHost.exe /\
Processid:{973D20D7-562D-44B9-B70B-5A0F49CCDF3F}
12:07:13.554: Process 11072 Created. Command line: C:\WINDOWS\system32\Windows.WARP.\
JITService.exe 7f992973-8a6d-421d-b042-6afd93a19631 S-1-15-2-3624051433-2125758914-1\
423191267-1740899205-1073925389-3782572162-737981194 S-1-5-21-4017881901-586210945-2\
666946644-1001 516
12:07:14.454: Process 12516 Created. Command line: C:\Windows\System32\RuntimeBroker\
.exe -Embedding
12:07:14.914: Process 10424 Created. Command line: C:\WINDOWS\system32\MicrosoftEdge\
SH.exe SCODEF:5276 CREDAT:9730 APH:1000000000000017 JITHOST /prefetch:2
12:07:14.980: Process 12536 Created. Command line: "C:\Windows\System32\MicrosoftEdg\
eCP.exe" -ServerName:Windows.Internal.WebRuntime.ContentProcessServer
12:07:17.741: Process 7828 Created. Command line: C:\WINDOWS\system32\SearchIndexer.\
exe /Embedding
12:07:19.171: Process 2076 Exited
12:07:30.286: Process 3036 Created. Command line: "C:\Windows\System32\MicrosoftEdge\
CP.exe" -ServerName:Windows.Internal.WebRuntime.ContentProcessServer
12:07:31.657: Process 9536 Exited
```

Уведомления о потоках

Ядро обеспечивает обратные вызовы создания и уничтожения потоков, аналогично обратным вызовам процессов.

Для этого используется PsSetCreateThreadNotifyRoutine, а для отмены регистрации - другой API, PsRemoveCreateThreadNotifyRoutine. Аргументы, предоставленные подпрограмме обратного вызова: идентификатор процесса, идентификатор потока и то, создается или уничтожается поток.

Мы расширим существующий драйвер SysMon для получения уведомлений потоков, а также уведомлений о процессах.

Сначала мы добавим значения перечисления для событий потока и структуру, представляющую информацию, все в заголовочном файле SysMonCommon.h:

```
enum class ItemType : short {
	None,
	ProcessCreate,
	ProcessExit,
	ThreadCreate,
	ThreadExit
};
struct ThreadCreateExitInfo : ItemHeader {
	ULONG ThreadId;
	ULONG ProcessId;
};
```

Теперь мы можем добавить правильную регистрацию в DriverEntry:

```
status = PsSetCreateThreadNotifyRoutine(OnThreadNotify);
if (!NT_SUCCESS(status)) {
	KdPrint((DRIVER_PREFIX "failed to set thread callbacks (status=
%08X)\n", status)\
	);
	break;
}
```

Сама процедура обратного вызова довольно проста, поскольку структура события имеет постоянный размер. Здесь процедура обратного вызова потока целиком:

```
void OnThreadNotify(HANDLE ProcessId, HANDLE ThreadId, BOOLEAN Create) {
	auto size = sizeof(FullItem<ThreadCreateExitInfo>);
	auto info =
(FullItem<ThreadCreateExitInfo>*)ExAllocatePoolWithTag(PagedPool,
			size, DRIVER_TAG);
	if (info == nullptr) {
		KdPrint((DRIVER_PREFIX "Failed to allocate memory\n"));
		return;
	}
```

```
        auto& item = info->Data;
        KeQuerySystemTimePrecise(&item.Time);
        item.Size = sizeof(item);
        item.Type = Create ? ItemType::ThreadCreate : ItemType::ThreadExit;
        item.ProcessId = HandleToULong(ProcessId);
        item.ThreadId = HandleToULong(ThreadId);
        PushItem(&info->Entry);
}
```

Большая часть этого кода должна выглядеть довольно знакомой. Чтобы завершить реализацию, мы добавим клиенту код, который знает, как отображать поток создания и удаления (в DisplayInfo):

```
case ItemType::ThreadCreate:
{
        DisplayTime(header->Time);
        auto info = (ThreadCreateExitInfo*)buffer;
        printf("Thread %d Created in process %d\n",
                info->ThreadId, info->ProcessId);
        break;
}
case ItemType::ThreadExit:
{
        DisplayTime(header->Time);
        auto info = (ThreadCreateExitInfo*)buffer;
        printf("Thread %d Exited from process %d\n",
                info->ThreadId, info->ProcessId);
        break;
}
```

Вот несколько примеров выходных данных для обновленного драйвера и клиента:

```
13:06:29.631: Thread 12180 Exited from process 11976
13:06:29.885: Thread 13016 Exited from process 8820
13:06:29.955: Thread 12532 Exited from process 8560
13:06:30.218: Process 12164 Created. Command line: SysMonClient.exe
13:06:30.219: Thread 12004 Created in process 12164
13:06:30.607: Thread 12876 Created in process 10728

...

13:06:33.260: Thread 4524 Exited from process 4484
13:06:33.260: Thread 13072 Exited from process 4484
13:06:33.264: Process 4484 Exited
13:06:33.264: Thread 4960 Exited from process 5776
13:06:33.264: Thread 12660 Exited from process 5776
13:06:33.265: Process 5776 Exited
13:06:33.272: Process 2584 Created. Command line: "C:\$WINDOWS.~BT\Sources\mighost.e\
xe" {CCD9805D-B15B-4550-94FB-B2AE544639BF} /InitDoneEvent:MigHost.{CCD9805D-B15B-455\
0-94FB-B2AE544639BF}.Event /ParentPID:11908 /LogDir:"C:\$WINDOWS.~BT\Sources\Panther\
"
13:06:33.272: Thread 13272 Created in process 2584
13:06:33.280: Process 12120 Created. Command line: \??\C:\WINDOWS\system32\conhost.e\
xe 0xffffffff -ForceV1
13:06:33.280: Thread 4200 Created in process 12120
13:06:33.283: Thread 4400 Created in process 12120
13:06:33.284: Thread 9632 Created in process 12120
13:06:33.284: Thread 6064 Created in process 12120
13:06:33.289: Thread 2472 Created in process 12120
```

Уведомления о загрузке изображений

Последний механизм обратного вызова, который мы рассмотрим в этой главе, - это уведомления о загрузке образов.

Когда файл образа (EXE, DLL, драйвер) загружается, драйвер может получить уведомление.

Функция PsSetLoadImageNotifyRoutine регистрирует эти уведомления, а PsRemoveImageNotifyRoutine используется для отмены регистрации.

```
typedef void (*PLOAD_IMAGE_NOTIFY_ROUTINE)(
            _In_opt_ PUNICODE_STRING FullImageName,
            _In_ HANDLE ProcessId,
            // pid into which image is being mapped
            _In_ PIMAGE_INFO ImageInfo);
```

Аргумент FullImageName несколько сложен. Как указано в аннотации SAL, это необязательно и может быть NULL. Даже если он не равен NULL, он не всегда дает правильное имя файла образа.

Причины этого уходят корнями глубоко в ядро и выходят за рамки этой книги. В большинстве случаев это работает нормально, а формат пути - внутренний формат NT, начиная с «\ Device \ HadrdiskVolumex \ ...», а не «c: \ ...».

Перевод можно сделать несколькими способами. Смотри более подробно об этом в главе 11.

Аргумент ProcessId - это идентификатор процесса, в который загружается образ. Для драйверов (образов ядра), это значение равно нулю. Аргумент ImageInfo содержит дополнительную информацию об образе, объявленную следующим образом:

```
typedef struct _IMAGE_INFO {
      union {
            ULONG Properties;
            struct {
                  ULONG ImageAddressingMode
                  ULONG SystemModeImage
                  ULONG ImageMappedToAllPids
                  ULONG ExtendedInfoPresent
                  ULONG MachineTypeMismatch
                  ULONG ImageSignatureLevel
                  ULONG ImageSignatureType
                  ULONG ImagePartialMap
                  ULONG Reserved
            };
      };
      PVOID
      ImageBase;
      ULONG
      ImageSelector;
      SIZE_T
      ImageSize;
      ULONG
```

```
        ImageSectionNumber;
} IMAGE_INFO, *PIMAGE_INFO;
```

Вот краткое изложение важных полей в этой структуре:

• SystemModeImage - этот флаг установлен для образа ядра и не установлен для образа пользовательского режима.

• ImageSignatureLevel - уровень подписи (Windows 8.1 и выше).

• ImageSignatureType - тип подписи (Windows 8.1 и выше).

• ImageBase - виртуальный адрес, по которому загружается образ.

• ImageSize - размер образа.

• ExtendedInfoPresent - если этот флаг установлен, то IMAGE_INFO является частью более крупной структуры IMAGE_INFO_EX.

```
typedef struct _IMAGE_INFO_EX {
        SIZE_T
        Size;
        IMAGE_INFO
        ImageInfo;
        struct _FILE_OBJECT *FileObject;
} IMAGE_INFO_EX, *PIMAGE_INFO_EX;
```

Чтобы получить доступ к этой более крупной структуре, драйвер использует CONTAINING_RECORD следующим образом:

```
if (ImageInfo->ExtendedInfoPresent) {
        auto exinfo = CONTAINING_RECORD(ImageInfo, IMAGE_INFO_EX, ImageInfo);
        // access FileObject
}
```

Расширенная структура добавляет только один значимый член - файловый объект, используемый для управления образом. Драйвер может добавить ссылку на объект (ObReferenceObject) и использовать ее в других функциях если это необходимо.

Упражнения

1. Создайте драйвер, который отслеживает создание процесса и позволяет клиентскому приложению настраивать исполняемые пути, выполнение которых не должно быть разрешено.

2. Напишите драйвер (или добавьте к драйверу SysMon) возможность обнаруживать создание удаленных потоков - потоки, созданные в процессах, отличных от их собственных.

Подсказка: первый поток в процессе всегда создан «удаленно». Сообщите клиенту пользовательского режима, когда это произойдет. Напишите тестовое приложение, которое использует CreateRemoteThread для проверки вашего обнаружения.

Резюме

В этой главе мы рассмотрели некоторые механизмы обратного вызова, предоставляемые ядром: процесс, поток и образы. В следующей главе мы продолжим работу с другими механизмами обратного вызова - объектами и реестром.