Обработка уведомлений о выходе из процессов

В приведенном выше коде функция уведомления процессов называется `OnProcessNotify`, а ее прототип был представлен ранее в этой главе. Эта функция обратного вызова обрабатывает события создания и завершения процессов. Начнем с выхода из процессов, так как это событие намного проще создания процесса (как вы вскоре увидите). Общая схема функции обратного вызова выглядит так:

```
void OnProcessNotify(PEPROCESS Process, HANDLE ProcessId,
    PPS_CREATE_NOTIFY_INFO CreateInfo) {
    if (CreateInfo) {
        // Создание процесса
    }
    else {
        // Завершение процесса
    }
}
```

В случае выхода из процесса есть только идентификатор процесса, который необходимо сохранить (наряду с данными заголовка, общими для всех событий). Сначала необходимо выделить память для всей структуры, представляющей событие:

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

Если попытка выделения памяти завершается неудачей, драйвер ничего сделать не сможет, поэтому он просто возвращает управление из функции обратного вызова.

Затем нужно заполнить общую информацию: время, тип и размер элемента. Получить все эти данные несложно:

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

PushItem(&info->Entry);
```

Сначала мы обращаемся к самому элементу данных (в обход `LIST_ENTRY`) через переменную `info`. Затем заполняется информация заголовка: тип элемента хорошо известен, так как текущей является ветвь, обрабатывающая уведомления о завершении процессов; время можно получить при помощи

функции KeQuerySystemTimePrecise, возвращающей текущее системное время (UTC, не местное время) в формате 64-разрядного целого числа, с отчетом от 1 января 1601 года. Наконец, размер элемента — величина постоянная, равная размеру структуры данных, предоставляемой пользователю (а не размеру FullItem<ProcessExitInfo>).



Функция API KeQuerySystemTimePrecise появилась в Windows 8. В более ранних версиях следует использовать функцию API KeQuerySystemTime.

Дополнительные данные при завершении процесса состоят из идентификатора процесса. В коде используется функция HandleToULong для корректного преобразования объекта HANDLE в 32-разрядное целое без знака.

А теперь остается добавить новый элемент в конец связного списка. Для этого мы определим функцию с именем PushItem:

```
void PushItem(LIST_ENTRY* entry) {
    AutoLock<FastMutex> lock(g_Globals.Mutex);
    if (g_Globals.ItemCount > 1024) {
        // Слишком много элементов, удалить самый старый
        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++;
}
```

Сначала код захватывает быстрый мьютекс, так как функция может вызываться сразу несколькими потоками одновременно. Все дальнейшее делается под защитой быстрого мьютекса.

Кроме того, драйвер ограничивает количество элементов связного списка. Такая предосторожность необходима, потому что ничто не гарантирует, что клиент будет быстро потреблять эти события. Драйвер не должен допускать неограниченное потребление данных, так как это может повредить системе в целом. Значение 1024 выбрано совершенно произвольно. Правильнее было бы читать это число из раздела драйвера в реестре.



Реализуйте это ограничение с чтением из реестра в DriverEntry. Подсказка: используйте такие функции API, как ZwOpenKey или IoOpenDeviceRegistryKey, а также ZwQueryValueKey.

Если счетчик элементов превысил максимальное значение, самый старый элемент удаляется; фактически связанный список рассматривается как очередь (RemoveHeadList). При освобождении элемента его память должна быть

освобождена. Указателем на элемент не обязательно должен быть указатель, изначально использованный для выделения памяти (хотя в данном случае это так, потому что объект `LIST_ENTRY` стоит на первом месте в структуре `FullItem<>`), поэтому для получения начального адреса объекта `FullItem<>` используется макрос `CONTAINING_RECORD`. Теперь элемент можно освободить вызовом `ExFreePool`.

На рис. 8.2 изображена структура объектов `FullItem<T>`.

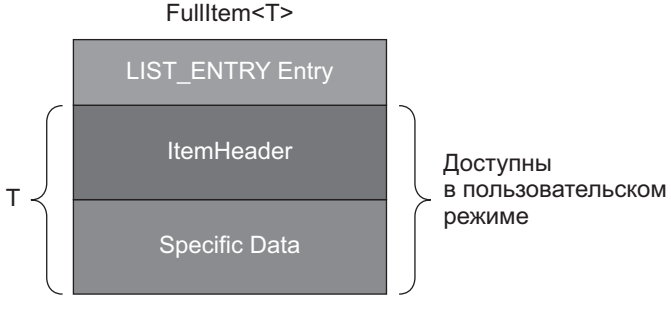

Рис. 8.2. Структура FullItem<T>

Наконец, драйвер вызывает `InsertTailList`, чтобы добавить элемент в конец списка, а счетчик элементов увеличивается на 1.

> Использовать атомарные операции инкремента/декремента в функции `PushItem` не обязательно, потому что операции со счетчиком элементов всегда выполняются под защитой быстрого мьютекса.

Обработка уведомлений о создании процессов

Обработка уведомлений о создании процессов создает больше проблем из-за непостоянного объема информации. Например, длина командной строки изменяется в зависимости от процесса. Сначала необходимо решить, какая информация должна сохраняться для создания процесса. Первая попытка:

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

В структуре сохраняется идентификатор процесса, идентификатор родительского процесса и командная строка. На первый взгляд такое решение работает и не создает проблем, потому что размер известен заранее.



Какие проблемы могут возникнуть при использовании приведенного определения?

Потенциальная проблема связана с командной строкой. Объявление командной строки с постоянным размером — решение простое, но проблематичное. Если командная строка окажется длиннее выделенного блока, драйвер будет вынужден произвести усечение (возможно, с потерей важной информации). Если командная строка короче выделенной, драйвер будет неэффективно расходовать память.



А можно ли использовать решение следующего вида:

```
struct ProcessCreateInfo : ItemHeader {
    ULONG ProcessId;
    ULONG ParentProcessId;
    UNICODE_STRING CommandLine; // Будет работать?
};
```

Нет, такое решение работать не будет. Во-первых, `UNICODE_STRING` обычно не определяется в заголовках пользовательского режима. Во-вторых (что намного хуже), внутренний указатель на символы обычно будет указывать в системное пространство, недоступное для пользовательского режима.

Ниже приведен другой вариант, который мы используем в драйвере:

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

В структуре будет храниться длина командной строки и ее смещение от начала структуры. Сами символы командной строки будут следовать за структурой в памяти. В этом случае мы не ограничиваем длину командной строки и не теряем память для коротких командных строк.

С таким объявлением можно приступить к построению реализации 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;
}
```

Суммарный размер выделяемого блока зависит от длины командной строки. Начнем с заполнения неизменяющейся информации, а именно заголовка, идентификаторов процесса и родительского 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 вычисляться с учетом базовой структуры и длины командной строки.

Затем необходимо скопировать командную строку по адресу за базовой структурой, а также обновить длину и смещение:

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



Добавьте в структуру `ProcessCreateInfo` имя файла образа по той же схеме, что и для командной строки. Будьте внимательны при вычислении смещения.

Передача данных в пользовательский режим

Затем следует понять, как передать собранную информацию клиенту пользовательского режима. Есть несколько возможных вариантов, но в нашем драйвере клиент будет запрашивать информацию у драйвера при помощи запроса чтения. Драйвер заполняет предоставленный буфер максимально возможным количеством событий (до исчерпания буфера или до последнего события в очереди).

Начнем обработку запроса чтения с получения адреса пользовательского буфера с применением прямого ввода/вывода (настраивается 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); // Используем прямой ввод/вывод

    auto buffer = (UCHAR*)MmGetSystemAddressForMdlSafe(Irp->MdlAddress,
        NormalPagePriority);
    if (!buffer) {
        status = STATUS_INSUFFICIENT_RESOURCES;
    }
    else {
```

Теперь необходимо обратиться к связанному списку и извлечь элементы из заголовка:

```
AutoLock lock(g_Globals.Mutex); // C++ 17
while (true) {
    if (IsListEmpty(&g_Globals.ItemsHead)) // также можно проверить
                                           // 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) {
        // Пользовательский буфер заполнен, вставить элемент обратно
        InsertHeadList(&g_Globals.ItemsHead, entry);
        break;
    }

    g_Globals.ItemCount--;
    ::memcpy(buffer, &info->Data, size);
    len -= size;
    buffer += size;
    count += size;
    // Освободить данные после копирования
    ExFreePool(info);
}
```

Сначала мы захватываем быстрый мьютекс, так как уведомления процессов продолжают поступать. Если список пуст, то делать нечего, и выполнение цикла прерывается. После этого извлекается заголовочный элемент, и если его размер не превышает размер оставшейся части пользовательского буфера, копируется его содержимое (без поля `LIST_ENTRY`). Далее цикл продолжает извлекать элементы от заголовка списка, пока список не опустеет или пользовательский буфер не заполнится.

Наконец, запрос завершается с текущим статусом, а в поле `Information` сохраняется значение переменной `count`:

```
Irp->IoStatus.Status = status;
```

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

К функции выгрузки также стоит присмотреться повнимательнее. Если в связном списке присутствуют элементы, они должны быть освобождены явно; в противном случае возникнет утечка ресурсов:

```
void SysMonUnload(PDRIVER_OBJECT DriverObject) {
    // Отмена регистрации уведомлений процессов
    PsSetCreateProcessNotifyRoutineEx(OnProcessNotify, TRUE);

    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\sysmon");
    IoDeleteSymbolicLink(&symLink);
    IoDeleteDevice(DriverObject->DeviceObject);

    // Освобождение оставшихся элементов
    while (!IsListEmpty(&g_Globals.ItemsHead)) {
        auto entry = RemoveHeadList(&g_Globals.ItemsHead);
        ExFreePool(CONTAINING_RECORD(entry, FullItem<ItemHeader>, Entry));
    }
}
```

Клиент пользовательского режима

После того как все будет готово, можно написать клиент пользовательского режима, который запрашивает данные вызовом `ReadFile` и выводит результаты.

Функция `main` вызывает `ReadFile` в цикле с небольшой приостановкой, чтобы поток не потреблял ресурсы процессора постоянно. Поступившие данные отправляются для вывода:

```
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]; // 64-килобайтный буфер

    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 разобраться в структуре полученного буфера. Так как все события начинаются с общего заголовка, функция различает события по значению `ItemType`. После того как событие будет обработано, поле `Size` 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;
    }
}
```

Для правильного извлечения командной строки в коде используется конструктор класса C++ `wstring`, который может построить строку по указателю и длине строки. Вспомогательная функция `DisplayTime` форматирует время в виде, удобном для чтения:

```
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 устанавливается и запускается так, как было описано в главе 4.

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

sc start sysmon
```

Пример вывода, полученного при запуске SysMonClient.exe:

```
C:\Book>SysMonClient.exe
12:06:24.747: Process 13000 Exited
12:06:31.032: Process 7484 Created. Command line: SysMonClient.exe
12:06:42.461: Process 3128 Exited
12:06:42.462: Process 7936 Exited
12:06:42.474: Process 12320 Created. Command line: "C:\$WINDOWS.~BT\Sources\
                                                     mighost.\
exe" {5152EFE5-97CA-4DE6-BBD2-4F6ECE2ABD7A} /InitDoneEvent:MigHost.
                                                {5152EFE5-97CA-4D\
E6-BBD2-4F6ECE2ABD7A}.Event /ParentPID:11908 /LogDir:"C:\$WINDOWS.~BT\Sources\
                                                 Panthe\
r"
12:06:42.485: Process 12796 Created. Command line: \??\C:\WINDOWS\system32\
                                                    conhost.e\
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 потоков

Ядро предоставляет обратные вызовы создания и уничтожения потоков, аналогичные обратным вызовам процессов. Для регистрации используется функция API `PsSetCreateThreadNotifyRoutine`, а для ее отмены — другая функция, `PsRemoveCreateThreadNotifyRoutine`. В аргументах функции обратного вызова передается идентификатор процесса, идентификатор потока, а также флаг создания/уничтожения потока.

Расширим существующий драйвер SysMon, чтобы он получал не только уведомления процессов, но и уведомления потоков. Начнем с добавления значений перечисления и структуры, представляющей информацию, — все это добавляется в заголовочный файл SysMonCommon.h:

```
enum class ItemType : short {
    None,
    ProcessCreate,
    ProcessExit,
    ThreadCreate,
    ThreadExit
};

struct ThreadCreateExitInfo : ItemHeader {
    ULONG ThreadId;
    ULONG ProcessId;
};
```

Затем можно добавить вызов регистрации в `DriverEntry`, непосредственно за вызовом регистрации уведомлений процессов: