mgr inż. Przemysław Kudłacik

Trwałe przechwytywanie okien w systemach MS Windows

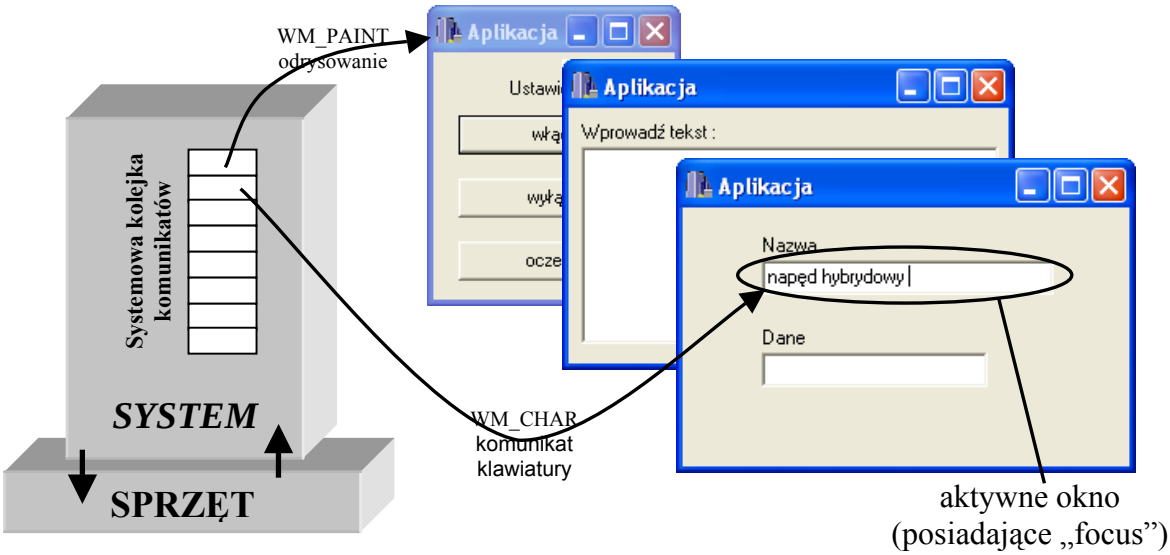
1. Komunikacja z oknami

Komunikacja z oknami w systemie operacyjnym Microsoft® Windows opiera się na przesyle komunikatów. Wysłanie komunikatu może być spowodowane zmianą stanu urządzeń zewnętrznych komputera, jak na przykład zmiana położenia kursora urządzenia wskazującego czy przerwanie zegara czasu rzeczywistego, lub generowane bezpośrednio przez procesy.
Komunikaty mogą być adresowane do grupy bądź do konkretnych okien.
Każde okno posiada swój unikalny identyfikator zwany uchwytem, który pozwala na jednoznaczne wskazanie okna i jest wykorzystywany chociażby przy rozsyłaniu komunikatów.
Istnieje wiele komunikatów systemowych sklasyfikowanych według źródła pochodzenia, wiele komunikatów o różnych priorytetach zależnych od ważności.
Jednym z podstawowych zadań systemu operacyjnego jest właściwe zarządzanie przesyłaniem wszystkich komunikatów.

Po wygenerowaniu komunikatu jest on wprowadzany do systemowej kolejki komunikatów, skąd później jest wysyłany do swego przeznaczenia. W zależności od obciążenia systemu oraz rodzaju komunikatu niektóre z nich mogą być łączone ze sobą i przesłane jako jeden komunikat. Dzieje się tak na przykład z komunikatem o niskim priorytecie WM_PAINT nakazującym odrysowanie pola danego okna.
Wątek okna pobiera komunikat dla niego przeznaczony za pomocą funkcji ***GetMessage()*** lub ***PeekMessage()***. Różnica między tymi funkcjami polega na tym, że druga z nich nie usuwa wiadomości z systemowej kolejki komunikatów umożliwiając tym samym sprawdzenie komunikatu przeznaczonego dla okna.

W przypadku komunikatów pochodzących z klawiatury i urządzenia wskazującego, w danej chwili otrzymuje je tylko okno aktywne (posiadające tzw. “focus”[1]). Ogólnie mówiąc jest to okno, z którym w danej chwili pracuje użytkownik.

[1] ang. “skupienie”



Rysunek 1, Komunikaty w systemie Windows

Problemem jest możliwość wskazania przez użytkownika okna innej aplikacji, do którego można wysyłać komunikaty.
Aby komunikować się z wybranym oknem należy pobrać jego uchwyt, za pomocą którego można to okno zaadresować. Jednym ze sposobów pobrania uchwytu dowolnego, widocznego okna jest jego zaznaczenie za pomocą urządzenia wskazującego (np. wskazanie okna przyciskiem myszy).
Takie rozwiązanie wymaga jednak możliwości przeglądania wszystkich komunikatów urządzenia wskazującego ponieważ, jak wcześniej wspomniano, komunikaty tego typu[2] otrzymuje tylko jedno okno w danej chwili aktywne (okno z którym pracuje użytkownik). Dlatego wskazując okno w innej aplikacji przesłany komunikat nie będzie odebrany przez inne procesy.
Z tego powodu należy zapewnić mechanizm pozwalający procesowi na przeglądanie wszystkich komunikatów pochodzących z urządzenia wskazującego przechodzących przez system. Taką funkcjonalność udostępniają haki Windows[3].

2. Mechanizm haków systemu Windows

W systemie operacyjnym Microsoft® Windows *hak* jest mechanizmem pozwalającym funkcji przechwytywać komunikaty zanim osiągną one przeznaczenie. Funkcja ta może operować komunikatem oraz w niektórych przypadkach modyfikować, lub nawet anulować komunikat. Funkcje, które otrzymują komunikaty w ten sposób, nazywane są *funkcjami filtrującymi* i są klasyfikowane według typu przechwytywanego komunikatu.

[2] komunikaty z klawiatury oraz urządzenia wskazującego
[3] patrz [1], [2]

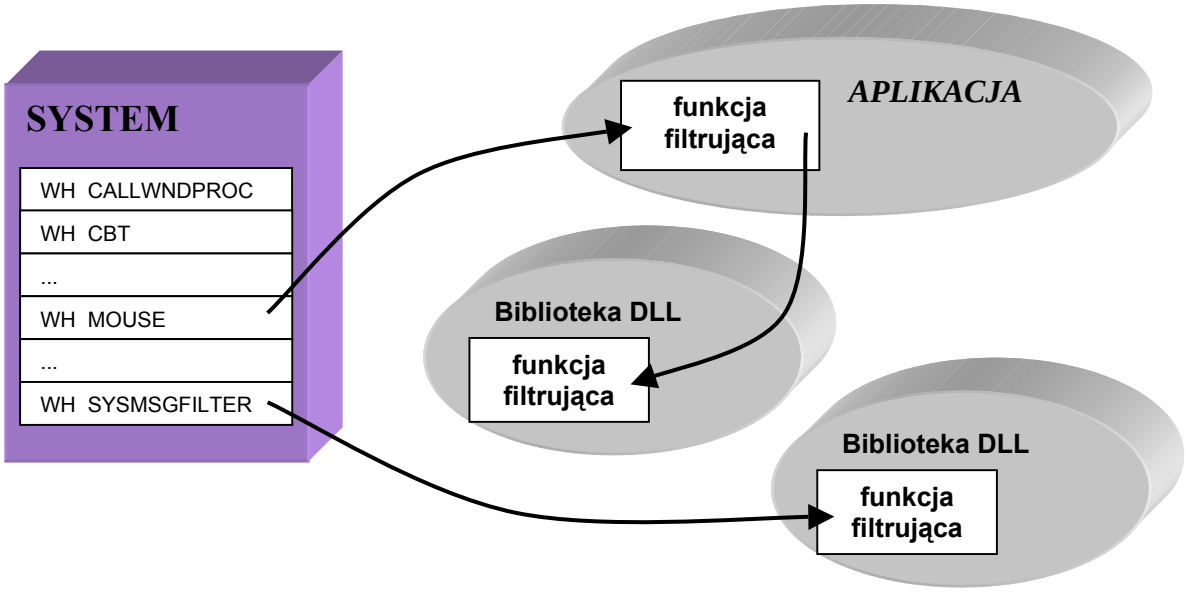
Aby funkcja mogła przechwytywać komunikaty musi być wywołana przez system w momencie jego pojawienia się, lub przed usunięciem. Dlatego funkcję taką należy zainstalować, to znaczy przyłączyć, do tzw. haka Windows. Przyłączanie jednej lub więcej funkcji do haka jest nazywane *ustawieniem haka*.
Jeśli hak ma przyłączoną więcej niż jedną funkcję, system tworzy łańcuch funkcji filtrujących, w którym ostatnio zainstalowana funkcja znajduje się na początku listy. W takim wypadku gdy nadejdzie komunikat uruchamiający dany hak, system wywołuje pierwszą funkcję filtrującą w łańcuchu. Ta akcja nazywana jest *wywołaniem haka*.

Istnieje dwanaście typów haków różniących się między sobą typem komunikatów przekazywanych funkcjom filtrującym, momentem, w którym komunikat jest funkcji filtrującej przekazywany czy też możliwością modyfikacji/anulowania komunikatu.
Większość haków można ustawić w zakresie systemowym (tzw. hak systemowy, przechwytujący komunikaty niezależnie od przeznaczenia) oraz na konkretny wątek (komunikaty przeznaczone jednemu wątkowi). Ustawienie haka na jeden wątek wiąże się z dużo mniejszym dodatkowym obciążeniem systemu i jest to rozwiązanie bezpieczniejsze. Istnieją jednak haki, które można ustawić wyłącznie na system.

Nazwa haka (identyfikator)	***Zakres***	***Opis***
WH_CALLWNDPROC	system lub wątek	uruchamiany gdy system wywołuje funkcję ***SendMessage()***. Funkcja filtrująca nie może modyfikować komunikatu.
WH_CBT	system lub wątek	wywoływany przed komunikatami dotyczącymi zmiany stanu okna komunikatami klawiatury/myszy. Stworzony specjalnie dla aplikacji CBT (ang. "computer-based training").
WH_DEBUG	system lub wątek	uruchamiany przed wywołaniem przez system funkcji filtrującej. Nie może zmienić informacji przesyłanej docelowej funkcji ale może anulować jej wywołanie, wykorzystywany przy wykrywaniu błędów
WH_GETMESSAGE	system lub wątek	uruchamiany przed zwróceniem danych z funkcji ***GetMessage()*** i ***PeekMessage()***, hak umożliwia zmianę zwracanego komunikatu
WH_JOURNALRECORD	tylko system	wywoływany przy usuwaniu z kolejki systemowej zdarzeń pochodzących z klawiatury i urządzenia wskazującego, nie może zmienić komunikatu, stworzony w celu nagrywania działań użytkownika
WH_JOURNALPLAYBACK	tylko system	używany w celu odtworzenia działań użytkownika nagranych wcześniej za pomocą **JournalRecord**, gdy jest włączony standardowe wejście klawiatury i urządzenia wskazującego pozostają nieaktywne
WH_FOREGROUNDIDLE	system lub wątek	informuje o stanie bezczynności wątku pierwszoplanowego (brak akcji użytkownika na wątku pierwszoplanowym)
WH_SHELL	system lub wątek	informuje o akcjach na oknie pierwszoplanowym, nie umożliwia modyfikacji czy anulowania komunikatu.

WH_KEYBOARD	system lub wątek	uruchamiany przed zwróceniem z funkcji ***GetMessage()*** i ***PeekMessage()*** komunikatów pochodzących z klawiatury, hak umożliwia anulowanie komunikatu
WH_MOUSE	system lub wątek	uruchamiany przed zwróceniem z funkcji ***GetMessage()*** i ***PeekMessage()*** komunikatów pochodzących z urzadzenia wskazującego, hak umożliwia anulowanie komunikatu
WH_MSGFILTER	system lub wątek	wywoływany kiedy **DialogBox**, **MessageBox**, **ScrollBar** czy **Menu** otrzymują komunikaty oraz gdy użytkownik naciśnie kombinację *ALT + TAB* lub *ALT + ESC*
WH_SYSMSGFILTER	tylko system	okoliczności wywołania identyczne z **MSGFILTER**, rożnicą jest zakres (wyłącznie systemowy) oraz wywołanie przed hakiem **MSGFILTER** (może on zablokować przesłanie komunikatu do funkcji filtrującej haka **MSGFILTER**)

Tabela 1, Rodzaje haków Windows



Rysunek 2, Haki i funkcje filtrujące w Windows.

W przypadku przeglądania komunikatów pochodzących bezpośrednio od użytkownika, jak komunikaty klawiatury i urządzenia wskazującego, pomocne wydają się haki oznaczone w dokumentacji stałymi WH_KEYBOARD, WH_MOUSE, WH_JOURNALRECORD oraz WH_CBT.
Hak WH_JURNALRECORD został stworzony z myślą o nagrywaniu działań użytkownika z możliwością późniejszego odtworzenia jego akcji z użyciem haka WH_JOURNALPLAYBACK.

Funkcja filtrująca podłączona do tego haka otrzymuje wszystkie komunikaty z klawiatury i urządzenia wskazującego, które opuszczają kolejkę systemową. Komunikaty są już przetworzone i nie można ich modyfikować lub anulować.
Haki WH_KEYBOARD i WH_MOUSE są wywoływane gdy system przetwarza komunikaty pochodzące kolejno z klawiatury i urządzenia wskazującego. Umożliwiają one anulowanie komunikatu.
Hak WH_CBT pozwala przeglądać największą grupę komunikatów. Wśród nich znajdują się także komunikaty klawiatury i urządzenia wskazującego, które system usuwa z kolejki systemowej. Z tego powodu nie można ich anulować w przeciwieństwie do innych komunikatów jakie otrzymuje funkcja filtrująca tego haka.

Najbardziej odpowiednim do tego zadania wydaje się być hak WM_MOUSE ponieważ pozwala anulować przechwycony komunikat. Daje to możliwość wskazania okna z jednoczesnym zablokowaniem jego reakcji na tą akcję.
Z drugiej jednak strony wykorzystując hak WH_JOURNALRECORD uzyskuje się w jednej funkcji filtrującej dostęp do komunikatów klawiatury i urządzenia wskazującego. Daje to możliwość współpracy obu urządzeń we wskazaniu okna.
Z punktu widzenia docelowego zadania wszystkie z czterech ostatnio wymienionych haków pozwalają na rozwiązanie problemu. Dlatego na tym etapie można rozważać wykorzystanie każdego z nich.

3. Użycie mechanizmu haków

Ustawienie haka wiąże się z dodatkowym obciążeniem systemu. Dotyczy to szczególnie haków działających w zakresie systemowym. Ponadto źle napisana funkcja filtrująca może powodować błędne przetwarzanie niektórych komunikatów (np. klawiatury) co może dać użytkownikowi wrażenie zawieszenia systemu. Przy wystąpieniu jakichkolwiek problemów z hakami możemy je rozwiązać używając skrótu klawiszy *CTRL+ALT+DEL*, który zawsze pozostanie aktywny. Jednak nie wszyscy użytkownicy mogą zdawać sobie z tego sprawę dlatego haki należy stosować bardzo ostrożnie i koniecznie wyłączać gdy nie są już potrzebne.

System Windows® umożliwia pracę z hakami za pomocą specjalnych *funkcji haków* ***SetWindowsHookEx(), UnhookWindowsHookEx()*** oraz ***CallNextHookEx()***.
Pierwsze dwie funkcje służą do podłączenia i odłączenia funkcji filtrującej do haka. Ostatnia pozwala na wywołanie następnej funkcji w łańcuchu, w przypadku istnienia większej ilości funkcji filtrujących podłączonych do jednego haka.
SetWindowsHookEx() pozwala określić typ haka[4] oraz zakres jego działania (sytem lub wątek). Należy także określić funkcję filtrującą.

Funkcje filtrujące mogą znajdować się w plikach wykonywalnych EXE lub dynamicznie dołączanych bibliotekach DLL. Funkcje przeznaczone do filtrowania komunikatów własnej aplikacji (tego samego procesu) mogą znajdować się bezpośrednio w jej pliku wykonywalnym.
Konstrukcja sytemu operacyjnego nakazuje aby funkcje filtrujące haków używane w zakresie systemowym lub dla innej aplikacji (innego procesu), znajdowały się w dynamicznie

[4] patrz tabela 1, *Rodzaje haków Windows*

dołączanej bibliotece DLL. Wyjątek stanowią funkcje filtrujące haków **JournalRecord** i **JournalPlayback**, które bez względu na położenie uzyskują systemowy zakres działania.

Cykl założenia haka rozpoczyna się napisaniem funkcji filtrującej umieszczając ją w zależności od potrzeb w dynamicznie dołączanej bibliotece, lub bezpośrednio w kodzie aplikacji.
W funkcji filtrującej, po przetworzeniu komunikatu, wywołujemy ***CallNextHookEx()*** aby umożliwić działanie innym funkcjom filtrującym podłączonym do tego samego haka.
Poniżej przykładowa funkcja filtrująca ***CBTProc()*** umieszczona w dynamicznie dołączanej bibliotece.

```
extern "C" __declspec(dllexport) LRESULT __stdcall
     CBTProc(int code, WPARAM wParam, LPARAM lParam)
{
     //przetwarzanie przechwyconej wiadomości
     ...
     //wywołanie kolejnego haka w łańcuchu
     return CallNextHookEx(hHook,code,wParam,lParam);
}
```

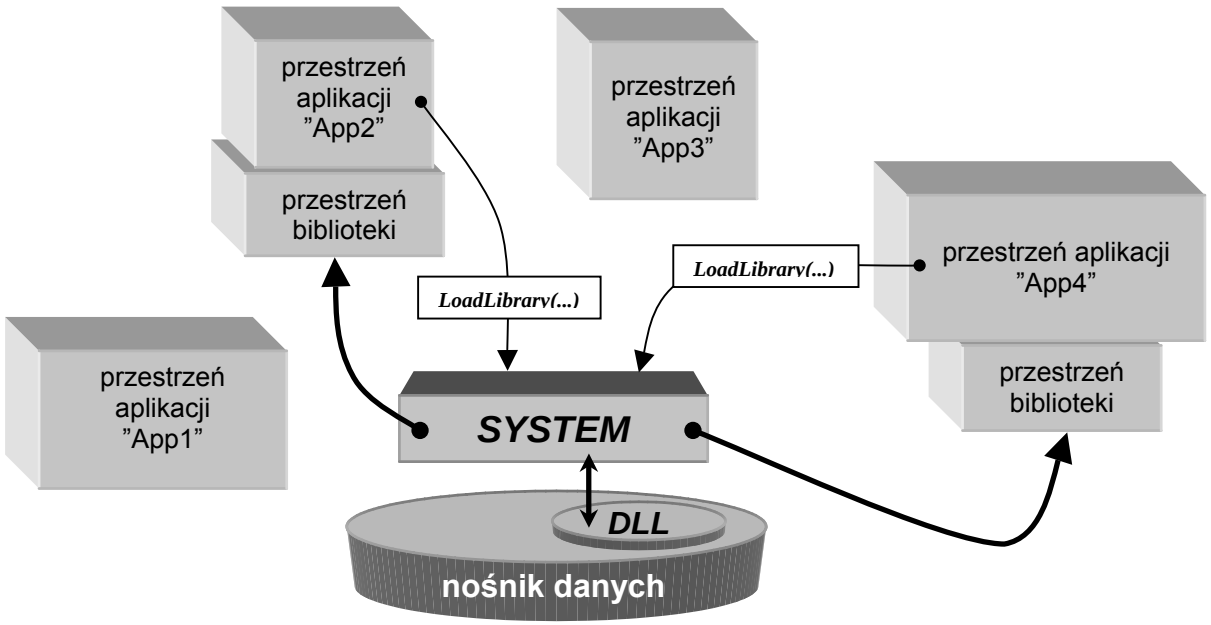
Przygotowaną funkcję można wykorzystać do filtrowania ustawiając hak za pomocą ***SetWindowsHookEx()***, wybierając między innymi zakres działania haka oraz określając zdefiniowaną funkcję. W poniższym przykładzie, wykorzystując funkcję biblioteczną, konieczne jest wcześniejsze załadowanie biblioteki oraz pobranie adresu funkcji filtrującej.

```
     //załadowanie biblioteki
LibraryHandle = LoadLibrary("hook.dll");
     //pobranie adresu funkcji filtrującej
Function = (TypeOfFunction) GetProcAddress(LibraryHandle, "CBTProc");
     //ustawienie haka o zakresie systemowym
HookHandle = SetWindowsHookEx(WH_CBT,(HOOKPROC)Function, LibraryHandle,0);

     ... //wykorzystanie haka

     //odłączenie haka
UnhookWindowsHookEx(HookHandle);
```

W przypadku ustawiania haka na konkretny wątek należy podać identyfikator tego wątku w ostatnim parametrze funkcji ***SetWindowsHookEx()***. Przykład odnosi się do haka zakresu systemowego dlatego parametr ten wynosi **0** wskazując na wszystkie wątki.
Gdy hak nie jest już potrzebny odłącza się funkcję filtrującą za pomocą ***UnhookWindowsHookEx()***.

4. Biblioteki przechowujące funkcje filtrujące

Biblioteki DLL (ang. ”dynamic linked library”) w systemie operacyjnym Microsoft® Windows są niezależnymi od aplikacji zbiorami skompilowanych funkcji, które mogą być wykorzystane przez więcej niż jedną aplikację w tym samym czasie.
Właśnie niezależność funkcji zamkniętych w oddzielnym pliku była powodem stworzenia systemu bibliotek DLL.
Po pierwsze tworząc biblioteki DLL możemy gromadzić w nich wspólne mechanizmy wykorzystywane w różnych aplikacjach. Tym sposobem kod wspólnie wykorzystywany znajduje się w jednym pliku na dysku, a nie w każdej aplikacji.
Po drugie pozwala to niezależnie rozwijać aplikację i funkcje narzędziowe zawarte w bibliotekach (ułatwienie procesu tworzenia nowych wersji części kodu przy zachowaniu jednego interfejsu).
Trzecią zaletą, wynikającą z poprzednich, jest automatyczna aktualizacja aplikacji korzystających z danej biblioteki w momencie utworzenia jej nowszej wersji.



Rysunek 3, Dynamiczne ładowanie bibliotek.

Na rysunku 3 w bardzo uproszczony sposób przedstawiono proces dynamicznego dołączania bibliotek. Można zauważyć, że każda aplikacja ładująca bibliotekę poszerza swoją przestrzeń o jej środowisko[5]. Przestrzenie bibliotek załadowanych przez różne aplikacje są zupełnie rozłączne. Stają się one częścią przestrzeni ładujących je procesów i mają do nich dostęp.

Wywołanie haka w zakresie systemowym wymaga dostępu funkcji filtrującej do wielu procesów otrzymujących komunikaty. Z tego właśnie powodu konieczne jest umieszczenie jej w dynamicznie dołączanej bibliotece.

[5] środowisko biblioteki zawiera jej funkcje oraz zdefiniowane zmienne/dane

Hak zakresu systemowego podczas pracy dołącza bibliotekę zawierającą funkcję filtrującą procesom otrzymującym filtrowane komunikaty. W ten sposób funkcja wywoływana jest w przestrzeniach różnych procesów.
Rodzi to problem przesyłania danych między funkcjami filtrującymi i aplikacją zakładającą hak, działającymi przecież w różnych przestrzeniach.

Rozwiązaniem jest utworzenie segmentu pamięci dzielonej, do której mogłyby uzyskać dostęp różne procesy. Funkcja filtrująca mogłaby wtedy wykorzystywać taki segment dzielony w celu pobrania uchwytu haka oraz zwrócenia wyników swego działania.
Uchwyt haka jest potrzebny przy wywołaniu następnej funkcji w łańcuchu metodą ***CallNextHookEx()***.

Środowisko programistyczne **VisualC++** firmy Microsoft® pozwala w prosty sposób utworzyć segment pamięci dzielonej za pomocą dyrektyw kompilatora jak w poniższym przykładzie.

```
#pragma data_seg(”SharedData”)//początek segmentu

... //tu definicje zmiennych mających się znaleźć w segmencie dzielonym

#pragma data_seg()            //koniec segmentu

      //wiadomość dla linkera o utworzeniu segmentu dzielonego
#pragma comment(linker,”/section:SharedData,RWS”)
```

Rozwiązaniem bardziej uniwersalnym, niezależnym od platformy programistycznej, jest wykorzystanie mechanizmów systemowych pozwalających stworzyć wspólną pamięć.
Jednym z takich mechanizmów są obiekty MMF[6] (ang. ”memory mapped files”).

W zasadzie obiekty MMF rozszerzają sposób dostępu do plików. Pozwalają one potraktować plik lub jego część jak pamięć operacyjną, sprowadzając pracę z plikiem do zwykłych odwołań do pamięci i wykorzystania wskaźników.
Mechanizm MMF zapewnia spójność danych podczas równoległego dostępu do pliku przez różne procesy, dlatego jest alternatywnym sposobem stworzenia segmentu pamięci dzielonej.
Fizyczny dostęp do dysku jest wykonywany przez systemowego zarządcę pamięci wirtualnej VMM (ang. ”virtual-memory manager”) co zapewnia zoptymalizowane działanie. Strony o rozmiarach 4 KB są ładowane lub zapisywane w razie potrzeby, a więc praca z segmentami mniejszymi, zawierającymi kilkanaście zmiennych, praktycznie eliminuje jakikolwiek fizyczny dostęp do dysku.

Aby wykorzystać mechanizm MMF jedynie jako pamięć dzieloną nie jest konieczne tworzenie lub otwieranie plików. Możliwe jest stworzenie obiektu odwzorowującego część systemowej pamięci stronicowania (ang. ”system pagefile”).

Obiekty MMF są jednoznacznie identyfikowane przez system za pomocą unikalnej nazwy podawanej w momencie tworzenia. Pozwala to na znalezienie stworzonego wcześniej obiektu i otwarcia go z innego procesu.

[6] patrz [1], [2]

Otwarcie lub stworzenie obiektu MMF jest pierwszym etapem. Następnie należy za jego pomocą stworzyć widok bezpośrednio zwracający uchwyt do pamięci, którą można już wykorzystywać.

Cykl pracy z pamięcią dzieloną MMF rozpoczyna się od utworzenia obiektu za pomocą funkcji systemowej ***CreateFileMapping()***. Pierwszy parametr funkcji określa uchwyt pliku. W tym przypadku konkretny plik nie ma znaczenia dlatego parametr nakazuje funkcji użycie pliku stronicowania. Kolejny parametr pozwala określić atrybuty ochrony obiektu, które w tym wypadku są standardowe.
Następne określają kolejno dostęp, rozmiar i unikalną nazwę. Rozmiar jest podawany w bajtach przez dwa 32 bitowe parametry w całości tworzące wartość 64 bitową. W tym wypadku określono niewielki rozmiar 256 bajtów, aż nadto wystarczający potrzebnej pamięci dzielonej.

```
            //stworzenie obiektu MMF
hMMF = CreateFileMapping ((HANDLE)0xFFFFFFFF,
                                  NULL,
                                  PAGE_READWRITE,
                                  0,
                                  256,
                                  "tmp_shared_data");
```

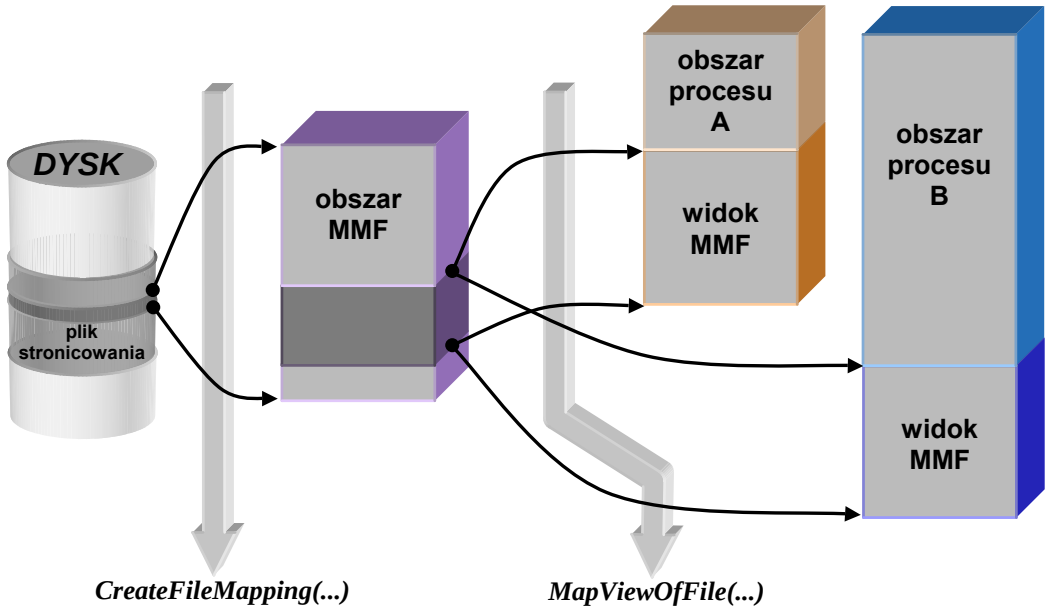
Gdy obiekt przestaje być potrzebny zwalnia się go przez zwykłe zamknięcie uchwytu.

```
CloseHandle (hMMF);
```

Korzystanie z utworzonej w ten sposób pamięci dzielonej w innym procesie rozpoczyna się od pobrania uchwytu obiektu MMF za pomocą funkcji systemowej ***OpenFileMapping()***. Obiekt jest wyszukiwany na podstawie unikalnej nazwy podanej w trzecim parametrze. Pozostałe parametry, podobnie jak przy tworzeniu obiektu, określają dostęp i atrybuty ochrony.

```
      //otwarcie istniejącego obiektu MMF
hMMF = OpenFileMapping (FILE_MAP_WRITE,FALSE,"tmp_shared_data");
      //uzyskanie wskaźnika pamięci (widoku)
hView = MapViewOfFile (hMMF,FILE_MAP_WRITE,0,0,256);

      ... //wykorzystanie pamięci adresowanej hView

      //zwolnienie wskaźnika pamięci (widoku)
UnmapViewOfFile (hView);
```

Otwarty obiekt jest jedynie bazą i przygotowaniem od strony sytemu. Uzyskanie końcowego wskaźnika obszaru pamięci realizuje się za pomocą funkcji systemowej ***MapViewOfFile()***.
Pierwsze parametry tej funkcji wskazują otwarty obiekt MMF oraz określają zasady dostępu. Ostatnie trzy parametry określają 64 bitowe przesunięcie (w dwóch 32 bitowych wartościach) oraz rozmiar pożądanego widoku. W ten sposób uzyskuje się dostęp do całości lub części obszaru zdefiniowanego przez obiekt MMF.
Zwolnienie widoku po wykorzystaniu pamięci dzielonej realizuje funkcja systemowa ***UnmapViewOfFile()***.



Rysunek 4, MMF jako pamięć dzielona.

Pobranie uchwytu obszaru dzielonego może być wykonane w ciele funkcji filtrującej, która musi znać jedynie unikalną nazwę obiektu MMF. Z tego powodu ten sposób tworzenia pamięci dzielonej nadaje się dla funkcji filtrujących działających w zakresie systemowym, umieszczonych w dynamicznie dołączanych bibliotekach.

5. Trwałe przechwycenie okna

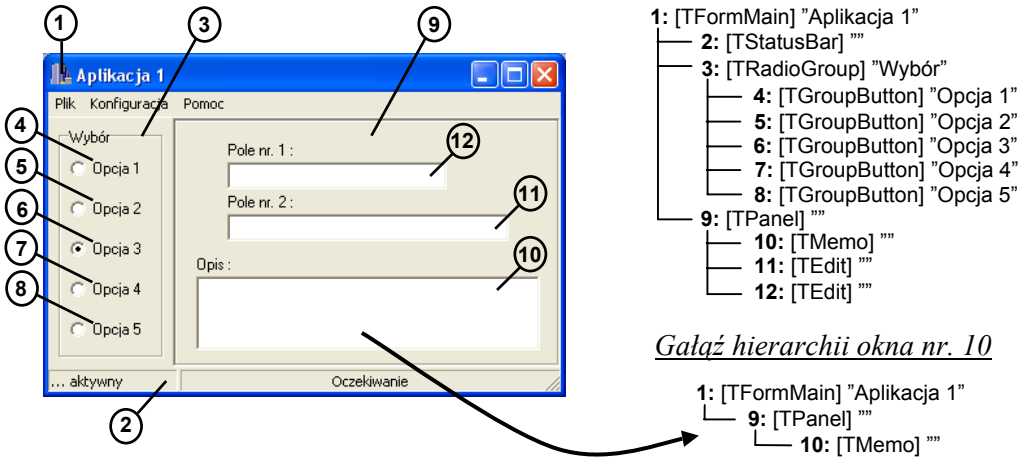
Jeśli przechwytywaniem okien nazwiemy pobieranie ich uchwytów w celu wysyłania komunikatów, to trwałym przechwyceniem okna będzie możliwość odwołania się do niego w różnych instancjach przechwyconej aplikacji. Jest to problem bardzo skomplikowany ze względu na zmienność uchwytów okien w różnych wywołaniach aplikacji.
Uchwyt zostaje określony w momencie tworzenia okna i jest zależny wyłącznie od aktualnego stanu zmiennych systemowych za to odpowiedzialnych.
Nie można zatem bazować w tym wypadku na uchwycie. Musi on posłużyć w momencie wskazania okna jako źródło innych informacji pozwalających zaadresować to okno w przyszłości.
Należy podkreślić, że okno od tego momentu nie jest obiektem istniejącym jedynie w danej instancji aplikacji, ale może być rozumiane jako pewna stała cecha każdej instancji.

Jeżeli uchwyt nie będzie już jednoznacznie identyfikował okna, należy za jego pomocą pobrać informacje pozostające niezmienne. Takimi informacjami mogą być : **nazwa klasy okna**, **nazwa okna**, **kolejność okna względem innych okien rodzica**, **położenie okna względem rodzica**, **rozmiar** i inne.
Nazwa okna w przypadku wielu okien (zwłaszcza kontrolek) często pozostaje pusta lub też zmienia się w czasie działania programu. Jedyną stałą cechą w ogólnym rozwiązaniu jest z ww. **nazwa klasy okna**. Spotyka się co prawda rozwiązania w których nazwa klasy może się

zmieniać, lecz w ogólnym przypadku można założyć tworzenie okien w różnych instancjach aplikacji na podstawie tych samych, lub bardzo podobnych klas. Reszta cech będzie zależała od konkretnego zastosowania.
Przykładowo, przechwytując okno tekstowe **EditBox** umieszczone na formularzu, wszystkie z wyżej wymienionych cech można traktować jako stałe. Natomiast w przypadku innego, przesuwalnego okna stała może być tylko nazwa klasy.

Zebrane dane okna mają posłużyć uzyskaniu jego aktualnego uchwytu lub stwierdzeniu, że takie okno jeszcze nie istnieje. Jedynym rozwiązaniem w tym wypadku staje się przeszukanie wszystkich okien istniejących aktualnie w systemie i sprawdzeniu na ile odpowiadają one szukanemu.
Proste przeszukanie wszystkich istniejących okien byłoby jednak kłopotliwe ze względu na ilość danych jakie należałoby o oknie zebrać, aby stosunkowo precyzyjnie określić zgodność.
Najbardziej cenną cechą okna w tym momencie staje się jego położenie w hierarchii okien danej aplikacji. Te dane bardzo upraszczają system wyszukiwania. Szczególnie, że istnieje ku temu wsparcie z poziomu systemu.
Zapamiętując całą gałąź hierarchii, począwszy od wskazanego okna, kończąc na jego najstarszym rodzicu można stworzyć bardzo precyzyjne źródło informacji. W tym wypadku ilość zapamiętywanych cech każdego z okien w takiej liście nie musi być duża.

Należy przypomnieć, iż wyszukuje się okien kontrolek, z reguły zagnieżdżonych głębiej w hierarchii aplikacji (okna edycyjne, do których są wprowadzane dane). Dlatego najkrótsza z możliwych gałęzi może mieć dwa poziomy w przypadku aplikacji zawierającej okna edycyjne w swoim głównym oknie. Nawet w takiej sytuacji liczba przeszukiwanych okien staje się dużo mniejsza.
Ponadto, praktycznie wszystkie główne okna aplikacji tworzone są na bazie unikalnych nazw klas. Wykorzystując tą cechę mamy bardzo duże prawdopodobieństwo znalezienia głównego okna aplikacji jedynie po nazwie klasy.



Rysunek 5, Hierarchiczna struktura okien aplikacji. Gałąź hierarchii.

Wspomnianym wcześniej wsparciem od strony systemu przy wyszukiwaniu okien są funkcje ***EnumWindows()*** oraz ***EnumChildWindows()***.

Funkcja ***EnumWindows()*** pozwala na przeglądnięcie wszystkich głównych okien aktualnie obecnych w systemie, bez względu na to czy są widoczne czy nie.
Okna główne są bezpośrednimi potomkami głównego okna ekranu, czyli pulpitu. Są one najstarszymi oknami-rodzicami każdej aplikacji (ang. ”top-level windows”).
Natomiast funkcja ***EnumChildWindows()***, pozwala przeglądnąć wszystkie okna potomne (okna-dzieci) jakiegokolwiek okna.

Za pomocą tych funkcji oraz zebranych danych w gałęzi hierarchii szukanego okna, można w dość łatwy sposób przeprowadzić wyszukiwanie.
Należy do tego celu stworzyć pewien system oceny, pozwalający wybierać okna najbardziej podobne, lub odrzucać je całkowicie w przypadku zbyt dużych różnic.
Do systemu oceniania zostaną zaliczone wcześniej wspomniane parametry takie jak **nazwa klasy** okna, **nazwa okna**, **rozmiary** oraz **położenie**. Parametry zostały wymienione w kolejności reprezentującej ich wartość w porównywaniu.
Na pierwszym miejscu postawiona została nazwa klasy ze względu na jej niezmienność w różnych instancjach aplikacji. Aby dostosować rozwiązanie do większej liczby spotykanych sytuacji będzie porównywana także pierwsza połowa nazwy klasy.
Takie rozwiązanie sprawdza się w przypadku generowania nazw klas okien na podstawie stałego przedrostka i zmiennego, najczęściej liczbowego zakończenia[7].
W przypadku gdy okno nie będzie spełniało warunku podobieństwa dla co najmniej połowy nazwy klasy, zostaje ono całkowicie wykluczone z dalszej oceny jako nie pasujące.
Kolejnym atrybutem oceny jest nazwa okna. Niewielki problem w tym przypadku powodują nazwy puste oraz nazwy zmienne w czasie działania programu.
Główne okna aplikacji często wskazują aktualny stan poprzez swą nazwę, wyświetlaną w tytule. Mogą na przykład wskazywać nazwę pliku podczas jego edycji lub imię i nazwisko osoby zalogowanej do systemu bazy danych. Okna bardziej zagnieżdżone w hierarchii danej aplikacji często nie mają żadnej nazwy.
Z tego właśnie powodu nie można traktować tego parametru jako głównego kryterium oceny. Jednakże zgodność nazwy okna szukanego i aktualnie rozpatrywanego przy jednoczesnej zgodności nazwy klasy, praktycznie rozstrzyga wynik wyszukiwania.

Dwa ostatnie atrybuty oceny dotyczą rozmiarów i położenia. Są to wartości mniej znaczące ze względu na ich zmienność.
W przypadku okienek edycyjnych na formularzach często zachowywany jest ten sam rozmiar, którego nie można zmienić. Także położenie okna względem obszaru rodzica w wielu przypadkach będzie tutaj wartością stałą. Z tego powodu te dwa atrybuty dołączono do parametrów porównywania.
Będą one miały większe znaczenie w momencie przeszukiwania okien w etapach głębszego zagnieżdżenia w hierarchiczną strukturę danej aplikacji. Okna główne zasadniczo pozwalają zmieniać swe położenie i rozmiary na pulpicie.

Funkcje ***EnumWindows()*** i ***EnumChildWindows()*** współdziałają z dodatkową funkcją definiowaną przez programistę, która zostaje wywoływana w pętli dla każdego przeglądanego okna. Właściwie ta funkcja jest motorem całego mechanizmu. W niej zawarte będą akcje porównywania okien oraz oceny.

[7] rozwiązanie spotykane np. przy generacji nazw klas dla kolejnych okien niektórych aplikacji MDI (dodawane są kolejne numery na koniec nazw klas)

Po zakończeniu działania ***EnumWindows()*** lub ***EnumChildWindows()*** zostaje wybrane okno o najwyższej ocenie (najbardziej podobne od szukanego).
W przypadku zaproponowanego rozwiązania ilość wywołań tych funkcji będzie zależała od ilości okien w gałęzi. Począwszy od wyszukania najstarszego rodzica (***EnumWindows()***) algorytm przejdzie kolejno do okien coraz niżej w hierarchii, przeszukując już tylko dzieci znajdowanych okien (***EnumChildWindows()***).

Rozpoczynając omawianie cyklu wyszukiwania w przyjętym rozwiązaniu należy opisać funkcję używaną przez ***EnumWindows()*** oraz ***EnumChildWindows()*** .
Otrzymuje ona poprzez parametry uchwyt przeglądanego okna oraz dodatkową wartość pozwalającą przesłać funkcji dowolne dane. Z reguły parametr ten jest wykorzystywany do przesłania adresu obiektu lub struktury pozwalającej na zwrócenie wyników oraz przekazanie większej ilości danych. Przykład takiej funkcji przedstawiono poniżej (aby zwiększyć czytelność zachowano jedynie elementy najistotniejsze pomijając definicje zmiennych i ich inicjowanie).

```
BOOL CALLBACK EnumFunction(HWND PassedhWnd,LPARAM lParam)
{
    ...
        //Pobranie adresu struktury TData
        // - zawiera ona dane szukanego okna potrzebne do porównania
        //   oraz pola do zwrócenia wyniku
    TData   *Data = (TData *) lParam;

        //zebranie danych okna przekazanego
    GetWindowText(PassedhWnd,bufor,200);   //pobranie nazwy okna
    ...
    GetClassName(PassedhWnd,bufor,200);    //pobranie nazwy klasy
    ...
    GetWindowPlacement(PassedhWnd,&pPos);  //pobranie położenia i rozmiarów

        // okno jest brane pod uwagę jeśli zgadza się co najmniej
        // połowa nazwy klasy
    if (PassedHalfOfClassName == Data->SearchedHalfOfClassName){

            //porównanie całych nazw klas
        if (PassedClassName == Data->SearchedClassName)   mark += 50;
            //porównanie nazw okien
        if (PassedWindowName == Data->SearchedWindowName) mark += 50;
            //porównanie pozycji okien (względem rodzica)
        if ((PassedPosX == Data->SearchedPosX) &&
            (PassedPosY == Data->SearchedPosY) )          mark += 20;
            //porównanie rozmiarów okien
        if ((PassedWidth == Data->SearchedWidth) &&
            (PassedHeight == Data->SearchedHeight)        mark += 20;

            //zapisanie okna jeżeli zdobyło wyższą ocenę niż poprzednie
        if (mark > Data->BestMark){
            Data->FoundHandle = PassedhWnd;
            Data->BestMark  = mark;
        }
    }
    return true;
}
```

Algorytm rozpoczyna wyszukaniem głównego okna przechwyconej aplikacji (pierwszego w zebranej gałęzi hierarchii). Osiąga to przeglądając wszystkie główne okna w systemie funkcją ***EnumWindows()***.
Przyjmuje ona tylko dwa parametry. Pierwszym jest adres procedury otrzymującej kolejne uchwyty okien, a drugi określa dodatkowe dane dla tej procedury.
W przypadku znalezienia okna głównego algorytm pobiera dane kolejnego okna w zachowanej gałęzi hierarchii i wyszukuje tylko wśród potomków głównego rodzica funkcją ***EnumChildWindows()***.
Proces znajdywania kolejnych obiektów całej gałęzi szukanego okna przebiega analogicznie. ***EnumChildWindows()*** jest wywoływana w pętli do momentu niepowodzenia lub do znalezienia ostatniego elementu gałęzi - czyli docelowego okna.

Poniższy przykład implementuje opisany algorytm.

```
//pobranie danych pierwszego okna w zapamiętanej gałęzi (index = 0)
IndexOfWindow = 0;
GetWindowData(indexOfWindow, &Data);

//zainicjowanie zmiennych i wyszukanie pierwszego okna
Data->FoundHandle = NULL; Data->BestMark = 0;
EnumWindows (&EnumFunction, &Data);

WindowHandle = Data->FoundHandle;

    //jeśli okno główne znalezione to kontynuuj wyszukiwanie
If (WindowHandle != NULL){
       //pętla dla reszty okien w zapamiętanej gałęzi
   for(indexOfWindow+=1;indexOfWindow<=indexOfLastWindow; indexOfWindow++){
          //pobranie danych kolejnego okna w gałęzi
      GetWindowData (indexOfWindow, &Data);
          //wyszukanie kolejnego okna wśród potomków wcześniej znalezionego
      Data->FoundHandle = NULL; Data->BestMark = 0;
      EnumChildWindows (WindowHandle, &EnumFunction, &Data);
          //jeśli okna nie znaleziono to koniec
      if (Data->FoundHandle == NULL) break;
          //zapamiętaj znaleziony uchwyt do kolejnego wyszukiwania
      WindowHandle = Data->FoundHandle;
   }
}
```

Po zakończeniu pracy algorytm pozostawia w zmiennej **WindowHandle** aktualny uchwyt okna lub też wskaźnik zerowy **NULL** oznaczający niepowodzenie całego wyszukiwania.

Podsumowując zaproponowany sposób trwałego przechwycenia okna należy zwrócić uwagę na jego słabe punkty. Mianowicie każda próba wyszukania jednego okna z gałęzi hierarchii wyłania obiekt o największym podobieństwie. Problem powstaje w przypadku gdy więcej niż jedno okno otrzymuje taką samą, najwyższą ocenę.
Podany algorytm zachowuje jedynie pierwsze okno, które w wyliczeniu taką ocenę otrzymało. Taka sytuacja mogłaby zajść przy bazowaniu na tych samych nazwach klas oraz przy zmiennych lub nawet braku nazw okien, co jest dość częstym zjawiskiem spotykanym przy kontrolkach.
Zasadniczą rolę pełni znalezienie głównego okna. Rozpoczyna to cały proces, okrajając duży zbiór wszystkich okien obecnych w systemie.

Na szczęście ogromna większość aplikacji bazuje swoje główne okna na klasach, które może z programowego punktu widzenia są często prawie identyczne, ale posiadają oryginalne nazwy. Dlatego znalezienie głównego okna, nawet na podstawie samej nazwy klasy, jest trafieniem o dużym prawdopodobieństwie.
Inaczej ma się rzecz z ostatnimi oknami w gałęzi, na dole hierarchii. W ich przypadku nie można już ufać rozstrzyganiu na podstawie nazwy klasy ponieważ może być ona cechą wspólną wielu okien-dzieci danego rodzica (np. przywoływane już wcześniej okna edycyjne **EditBox** na formularzu zazwyczaj bazują na tej samej nazwie klasy).
W tym momencie liczą się rozmiary i położenie, które dla okien najwyżej w hierarchii nie mają takiego znaczenia. Rozwiązuje to problem prawidłowego wyboru z wielu okien kontrolnych przy ostatnim, lokalnym wyszukiwaniu. Dlatego można założyć wysoką dokładność trafienia również w tym wypadku.
Znaczenie kolejnych elementów oceny w wyszukiwaniu okien pomiędzy głównym i ostatnim może rozkładać się różnie. Trudno w przypadku ogólnym zadecydować, które z nich mogą przeważyć o ostatecznym wyborze. Należy założyć, że wszystkie są ważne i rozstrzygające w ocenie.
Na tym etapie pracy algorytmu istnieje największe prawdopodobieństwo powstania wyżej wspomnianej sytuacji błędu. Jeśli spowoduje ona zły wybór okna to próba kontynuowania poszukiwań według zapamiętanej gałęzi hierarchii najpewniej zakończy się niepowodzeniem.

Można wprowadzić modyfikację przedstawionego rozwiązania, które magazynowałoby znalezione uchwyty okien otrzymujących lokalnie te same, najwyższe oceny. Pozwiliłoby to na powroty w przypadku niepowodzenia i prowadzenie przeszukiwań inną ścieżką.
Niestety zwiększa to prawdopodobieństwo znalezienia nieprawidłowego okna. Dlatego należałoby także zwiększyć liczbę atrybutów jakie wchodzą w skład oceny podobieństwa.
Z drugiej strony samo rozbudowanie systemu oceny może byłoby lepsze.

W każdym bądź razie rozwiązanie bez powrotów w praktyce okazuje się wystarczająco skuteczne, więc dodatkowe urozmaicanie sposobu wyszukiwań, czy oceny okien wydaje się zbędne.

BIBLIOGRAFIA:

1. “Microsoft® Win32® Programmer's Reference”, Microsoft® Corporation, 1996
2. Baza MSDN firmy Microsoft®