Laboratorium wprowadzające

laboratorium

2022

K.M. Ocetkiewicz, T. Goluch

1. Procesy

Uruchomienie dowolnego pliku wykonywalnego (EXE) tworzy nowy proces. Każdy proces posiada własną przestrzeń adresową (przestrzeń adresowa to zbiór wszystkich komórek pamięci, które można wskazać wskaźnikiem), w której umieszczony jest jego kod i dane. Procesy są izolowane od siebie – każdy z nich posiada własną przestrzeń adresową. Powoduje to, że wskaźniki mają znaczenie jedynie wewnątrz przestrzeni adresowej, w której zostały utworzone – jeżeli proces A utworzy np. zmienną typu int, pobierze jej adres i przekaże ten adres procesowi B, zaś proces B zapisze wartość w miejscu wskazanym przez odebrany wskaźnik, to zapis ten nie będzie miał żadnego wpływu na zawartość pamięci w procesie A. B zmodyfikuje jedynie cztery bajty w swojej przestrzeni adresowej. Z jednej strony zabezpiecza to procesy przed ingerencją z zewnątrz (błąd w procesie B nie ma prawa wpłynąć na zachowanie procesu A, złośliwe oprogramowanie w procesie B nie ma prawa odczytać zawartości pamięci procesu A, np. haseł). Z drugiej powoduje, że wszelka komunikacja pomiędzy procesami musi zachodzić za pośrednictwem systemu operacyjnego.

2. Biblioteki statyczne i dynamiczne

Napisaliśmy pewien komponent – uniwersalny fragment kodu, z którego będziemy korzystać w różnych programach. Jak możemy z tego komponentu skorzystać?

Najprostszym rozwiązaniem jest trzymanie go w postaci źródłowej i dołączanie źródeł do każdego projektu, z którego korzystamy. Rozwiązanie to ma jednak swoje wady – komponent będzie na nowo kompilowany przy każdej przebudowie projektu, mimo że jego kod nie będzie się zmieniał. Po drugie, aktualizacja kodu komponentu wymaga skompilowania na nowo całego projektu. Po trzecie, ta metoda powoduje, że każdy proces korzystający z komponentu posiada własną kopię jego kodu, co, jeżeli komponent jest duży, powoduje, że marnowana jest pamięć. Wreszcie, taki sposób dystrybucji komponentu powoduje, że musimy udostępnić jego kod źródłowy, co nie zawsze jest pożądane.

Biblioteki statyczne

Drugim sposobem jest zamknięcie kodu komponentu w bibliotece statycznej (pliki .lib w systemach Windows, .a w systemach Linuxowych). Rozwiązuje to problem udostępniania kodu źródłowego – udostępniamy już skompilowany kod, oraz problem wielokrotnej kompilacji – biblioteka dołączana jest do projektu dopiero w fazie konsolidacji, która, w porównaniu z kompilacją, zajmuje zazwyczaj niewiele czasu. Takie rozwiązanie wprowadza jednak dodatkowy problem, związany z nazewnictwem funkcji.

Name mangling

W języku C funkcje identyfikowane są po nazwie. Nie ma mechanizmu przeciążania funkcji, więc nazwa funkcji jest jednoznaczna. W C++ sytuacja wygląda inaczej. Można przeciążać funkcje, więc możemy utworzyć kilka funkcji o tej samej nazwie różniących się jedynie typem parametrów. Co więcej, możemy utworzyć konkretyzacje szablonu funkcji (również o takiej samej nazwie) które będą różniły się jedynie parametrem generycznym. Do tego dochodzą przestrzenie nazw – w różnych przestrzeniach nazw możemy mieć funkcje o takiej samej nazwie. Powoduje to, że do identyfikacji funkcji potrzebujemy jej nazwy, listy typów parametrów, listy parametrów generycznych oraz przestrzeni nazw w której się znaduje. Jednak w plikach półskompilowanych („obiektach” – pliki .obj lub .o) funkcje identyfikowane są tylko po nazwie (z przyczyn historycznych). Aby sobie z tym poradzić kompilator koduje wszystkie dane potrzebne mu do identyfikacji funkcji w jej nazwie. Proces ten nazywa się dekorowaniem nazw (name mangling). W konsekwencji np. funkcja `void f(int, char)` w rzeczywistości (w pliku obj a także w pliku lib) będzie nosiła nazwę np. `?f@@YAXHD@Z`. Nie byłoby z tym żadnego problemu gdyby nie fakt, że każdy kompilator koduje te informacje na swój sposób, niekompatybilny z innymi kompilatorami. Powoduje to, że udostępnianie kodu tworzonego w języku C++ przez biblioteki statyczne jest praktycznie niemożliwe.

Dekorowanie nazw jest dodatkowo skomplikowane przez sposób wołania funkcji (calling convention). Opisuje on w jaki sposób przekazywane są parametry do funkcji (przez rejestry czy przez stos, w jakiej kolejności umieszczane są one na stosie), w jaki sposób przekazywana jest wartość zwracana, kto zajmuje się usuwaniem parametrów ze stosu

itp. W języku C++ na systemy Windows najpopularniejsze są trzy sposoby: cdecl (typowy dla języka C i domyślny), stdcall (korzysta z niego całe API systemu Windows) oraz fastcall (jeżeli chcemy skorzystać z rejestrów do przekazywania parametrów). Sposób wołania funkcji ma wpływ na dekorowanie jej nazwy – zmiana sposobu wołania funkcji zmieni udekorowaną nazwę. Np.:

`int f();` – funkcja z domyślnym sposobem wołania; zazwyczaj cdecl ale można to zmodyfikować ustawiając odpowiednie opcje kompilatora,
`int _stdcall f();` – funkcja wołana sposobem stdcall,
`int _fastcall f();` – funkcja wołana sposobem fastcall,
`int _cdecl f();` – funkcja wołana sposobem cdecl.

Możemy także określić funkcję jako pochodzącą z języka C. Powoduje to, że funkcji takiej nie można przeciążać, ale redukuje ilość informacji, które muszą być umieszczone w jej nazwie do sposobu jej wołania. Aby określić funkcję jako pochodzącą z języka C należy przed jej deklaracją umieścić extern ”C”, np.:

```
extern ”C” int f1(); extern
”C” int _stdcall f2(); extern
”C” int _fastcall f3(); extern
”C” int _cdecl f3();
```

Biblioteki dynamiczne

Biblioteki dynamiczne (.dll lub .so) różnią się od statycznych tym, że ich kod nie jest dołączany do pliku wykonywalnego. Aby skorzystać z takiej biblioteki należy ją samodzielnie załadować (umieścić w przestrzeni adresowej procesu) np. funkcją `LoadLibrary` (w przypadku systemów Windows) a następnie pobrać z niej adres interesujących nas funkcji (np. funkcją `GetProcAddress`). Zalety takiego rozwiązania płyną z oddzielenia jej od plików wykonywalnych – aktualizacja kodu biblioteki nie wymaga ponownej kompilacji plików wykonywalnych. Dodatkowo, system operacyjny może, odpowiednio manipulując pamięcią, spowodować, że kilka procesów będzie korzystało z jednej kopii biblioteki w pamięci (nie umożliwi to jednak komunikacji między procesami – jeżeli proces nadpisze kod czy dane takiej biblioteki, to system utworzy indywidualną kopię zmodyfikowanego obszaru pamięci dla tego procesu). Rozwiązanie to nie jest jednak pozbawione wad – takie korzystanie z funkcji jest dużo bardziej uciążliwe. Pobierając adres funkcji musimy podać nazwę, pod jaką figuruje ona w bibliotece, a więc po dekoracji. Ponadto podajemy tylko nazwę a w rezultacie otrzymujemy wskaźnik typu void *. Musimy samodzielnie rzutować go na wskaźnik na funkcję odpowiedniego typu i wszelkie popełnione błędy obciążą nas – kompilator nie będzie miał możliwości zweryfikowania, czy dobrze to zrobiliśmy.

Biblioteki dynamiczne mają swoją „funkcję main”, nazywaną punktem wejścia (entry point). Ma ona następującą deklarację:

```
BOOL APIENTRY DllMain(HMODULE hModule, DWORD reason, LPVOID reserved)
```

zaś jej parametry to:

`hModule` – uchwyt naszej biblioteki,
`reason` – powód wywołania funkcji, jedna ze stałych:
- `DLL_PROCESS_ATTACH` – załadowanie biblioteki do przestrzeni adresowej procesu,
- `DLL_PROCESS_DETACH` – wyładowanie biblioteki z przestrzeni adresowej procesu,
- `DLL_THREAD_ATTACH` – uruchomienie nowego wątku w procesie (wywoływane w kontekście tego wątku),
- `DLL_THREAD_DETACH` – zakończenie wątku („czyste”; wywołane w kontekście kończonego wątku).

`reserved` – parametr zarezerwowany.

Funkcja powinna zwrócić TRUE jeżeli nie ma przeszkód do korzystania z biblioteki. Zwrócenie FALSE spowoduje że załadowanie biblioteki (funkcją LoadLibrary) się nie powiedzie. Jeżeli bibliotekę ładuje system operacyjny (model biblioteka dynamiczna + statyczna o którym za chwilę) zwrócenie FLASE z DllMain spowoduje, że plik wykonywalny z niej korzystający nie zostanie uruchomiony. Typowo w funkcji DllMain umieszcza się kod tworzący dane specyficzne dla procesu czy wątku. Jednak najczęściej pozostaje ona pusta:

```
BOOL APIENTRY DllMain( HMODULE hModule, DWORD  reason, LPVOID lpReserved) {
      switch (reason) { case DLL_PROCESS_ATTACH: case DLL_THREAD_ATTACH:
      case DLL_THREAD_DETACH:  case DLL_PROCESS_DETACH:
            break;
            }
      return TRUE;
      }
```

Biblioteki dynamiczne + biblioteka pomocnicza

Rozwiązanie to eliminuje większość problemów bibliotek dynamicznych. Tworzymy tu bibliotekę dynamiczną, identyczną jak w poprzednim punkcie. Dodatkowo jednak powstaje niewielka biblioteka statyczna, która udostępnia wszystkie funkcje udostępnione przez bibliotekę dynamiczną. Zadaniem tej biblioteki statycznej jest udawanie pełnoprawnej biblioteki, jednak wszystko czym się ona zajmuje to z jednej strony udawanie pełnoprawnej biblioteki statycznej (dzięki czemu korzystamy z biblioteki dynamicznej tak łatwo jak z biblioteki statycznej z wszelką weryfikacją typów parametrów itp.), z drugiej informowanie systemu operacyjnego, że plik wykonywalny, do którego jest dołączona, wymaga pewnego zbioru funkcji z określonej biblioteki dynamicznej.

Podczas uruchamiania pliku wykonywalnego korzystającego z takiej biblioteki, system operacyjny ładuje odpowiednią bibliotekę i pobiera adresy potrzebnych funkcji za nas.

Lista eksportowanych i importowanych symboli

Aby funkcja była „widoczna" w bibliotece dynamicznej i można było z niej skorzystać, należy ją wyeksportować. Można zrobić to na dwa sposoby. Pierwszym z nich to umieszczenie przed sygnaturą funkcji (ale po ewentualnym extern "C") napisu __declspec(dllexport), np.:

```
__declspec(dllexport) int f1() ... extern
"C" __declspec(dllexport) int f2() …
```

Drugim sposobem jest utworzenie pliku definicji modułu (module definition file, rozszerzenie .def) o treści:

```
LIBRARY nazwa.dll

EXPORTS
      funkcja1
      funkcja2
      ...
```

gdzie nazwa.dll to nazwa naszej biblioteki, zaś po EXPORTS umieszczamy listę nazw funkcji (niedekorowanych) które chcemy wyeksportować. Skorzystanie z takiego pliku powoduje, że wymienione w nim funkcje otrzymują niejawnie modyfikator extern "C". Skorzystanie z takiego pliku wymaga ustawienia opcji projektu project → properties → linker → input → module definition file na nazwę (najlepiej z pełną ścieżką) do utworzonego przez nas pliku.

Aby skorzystać z takiej funkcji w modelu biblioteka dynamiczna+pomocnicza należy ją zaimportować, czyli umieścić w kodzie jej deklarację poprzedzoną napisem __declspec(dllimport):

```
__declspec(dllimport) int f1() ... extern "C"
__declspec(dllimport) int f2() ...
```

Jak widać, pliki wykonywalne zawierają listy importowanych funkcji, zaś biblioteki dynamiczne zawierają listy eksportowanych funkcji (mogą także zawierać listy importowanych funkcji, jeżeli biblioteka korzysta z innej biblioteki dynamicznej). Listy te możemy obejrzeć narzędziem dumpbin:

```
dumpbin /exports nazwapliku
```

wyświetla listę eksportowanych funkcji zaś

```
dumpbin /exports nazwapliku
```

wyświetla listę importowanych funkcji i biblioteki, z których mają być one zaimportowane. Narzędzie to pozwala nam zweryfikować, czy funkcja jest w ogóle eksportowana z biblioteki i sprawdzić, pod jaką nazwą w niej widnieje. Narzędzie dumpbin ma dużo większe możliwości – pozwala np. wyświetlić listę symboli z biblioteki statycznej czy obiektu, wyświetlić listę sekcji pliku wykonywalnego a nawet przedstawić jego kod w asemblerze.

Pełen opis formatu plików DEF można znaleźć pod adresem:
https://docs.microsoft.com/en-us/cpp/build/reference/module-definition-dot-def-files?view=vs-2017

3. Statyczna/dynamiczna biblioteka standardowa

Biblioteka standardowa dołączana do każdego naszego programu także może mieć postać statyczną lub dynamiczną (w modelu dynamiczna+pomocnicza). Możemy wybrać, z którego wariantu chcemy skorzystać. Służy do tego opcja project → properties → C/C++ → code generation → runtime library, w której możemy wybrać:

Multi-Threaded (/MT) – korzystamy ze statycznej biblioteki standardowej,
Multi-Threaded (/MTd) – korzystamy ze statycznej biblioteki standardowej w wersji debug,
Multi-Threaded DLL (/MD) – korzystamy ze dynamicznej biblioteki standardowej,
Multi-Threaded DLL (/MDd) – korzystamy ze dynamicznej biblioteki standardowej w wersji debug.

Skorzystanie ze statycznej biblioteki standardowej spowoduje, że plik wykonywalny będzie większy (nawet o kilka megabajtów), ale nie będzie wymagał do działania żadnych bibliotek dynamicznych poza systemowymi. Skorzystanie z dynamicznej biblioteki standardowej powoduje, że nasz program wymaga do działania zainstalowanej w systemie biblioteki msvcr<wersja>.dll (dla /MD, Visual Studio do 2013), msvcr<wersja>d.dll (dla /MDd, Visual Studio do 2013) lub msvcp<wersja>.dll, vcruntime<wersja>.dll i ucrtbase.dll (/MT, Visual Studio 2015) czy msvcp<wersja>d.dll, vcruntime<wersja>d.dll i ucrtbased.dll (/MDd, Visual Studio 2015). Wersja to numer wersji Visual Studio (dla VS2010 jest to 100, dla VS2012 jest to 110, dla VS2013 jest to 120, dla VS2015 i VS2017 jest to 140).

4. Assembly .NET

Dodać opis Assembly .NET

Wersje .NET

Opisać czym różni się .Net Framework, Core i Standard

Biblioteki .NET

Utworzenie biblioteki .NET jest dużo łatwiejsze. W pierwszym