Speicherallokation im Fremdprozess

Zur grundsätzlichen Architektur eines Multitasking-Betriebsystem gehört es, jedem Prozess einen eigenen Adressraum zuzuweisen, in dem er weitgehend unabhängig von allen anderen Prozessen agieren kann. Fremdzugriffe – ob lesend, schreibend oder allozierend – sind nur über Umwege möglich.

Diese strikte Trennung der Prozessadressräume ist auch sehr sinnvoll, hilft sie doch dem Betriebssystem, diverse Speicherzugriffsfehler laufender Anwendungen so zu lokalisieren, dass dadurch andere Prozesse nicht eingeschränkt werden sollten.

Versucht unser Prozess, beispielsweise aufgrund eines Programmierfehlers, an einer nicht dafür vorgesehenen Stelle im Speicher zu schreiben, so mag zwar unser Programm abstürzen, die anderen gerade laufenden Prozesse werden davon jedoch glücklicherweise nicht betroffen.

In manchen Fällen ist es jedoch unbedingt erforderlich, auf den Adressraum eines anderen Prozesses zugreifen zu können. Handelt es sich beim Fremdprozess um eine selbstgeschriebene Anwendung, so lassen sich über *Memory Mapped Files* (Vgl. Windows-API, *CreateFileMapping*) bestimmte Regionen im Prozessadressraum *sharen*, so dass beide bzw. mehrere Prozesse darauf lesend und schreibend zugreifen können.

Wollen wir jedoch eine Speicherseite in einem fremden Prozess allozieren, auf den wir keinen direkten Einfluss haben – wie beispielsweise auf den Explorer –, so müssen wir nach einer Möglichkeit suchen, direkt auf den Adressraum des fremden Prozesses zugreifen zu können.

Der Lese- und Schreibzugriff stellt hierbei kein all zu großes Problem dar. Mittels *OpenProcess* gewinnen wir eine *Zugriffshandle* auf den Fremdprozess, die wir wiederum für die Windows-API-Funktionen *ReadProcessMemory* bzw. *WriteProcessMemory* benötigen; letztere Routinen ermöglichen uns Lese- bzw. Schreibzugriff im Fremdprozess, sofern wir ihn mit den entsprechenden Rechten öffnen durften.

Doch wie sieht es mit der Allokation von Speicher im Fremdprozess aus? Unter Windows NT stehen uns hierfür die beiden Funktionen *VirtualAllocEx* und *VirtualFreeEx* zur Verfügung, die – genau wie *Read/WriteProcessMemory* – eine *Zugriffshandle* auf einen geöffneten Prozess erwarten. In der Windows 9x-Linie (Windows 95, 98, ME) fehlen die *Virtual-Ex*-Funktionen allerdings gänzlich.

Die Microsoft-Programmierer haben sich jedoch genau für diesen Zweck ein undokumentiertes Flag (Vgl. *Windows 95 System Programming Secrets* von *Matt Pietrek*) vorbehalten, das es uns ermöglicht, mit Hilfe der eigentlich lokal begrenzten Funktionen *VirtualAlloc* und *VirtualFree* eine Speicherseite zu allozieren, auf die der Fremdprozess vollen Zugriff hat.

Da jenes undokumentierte Flag *VirtualAlloc* anweist, eine Speicherseite in der *Shared-Memory-Region* – die in dieser ungeschützten Weise nur in der Windows 9x-Linie existiert – zu allozieren, erhält unser Prozess somit eine zugriffsfähige Seite, auf die neben dem Fremdprozess auch alle gerade laufenden Prozesse Zugriff haben. Der folgende Auszug aus PEWinMemory (auf Heft-CD) zeigt eine somit auf allen Windowsversionen funktionierende Kapselung:

```
Const
  { Shared Memory Flag. Undocumented Flag
    for Allocation Type of VirtualAlloc
    in Windows 9x }
  VA_SHARED = $8000000;

Function AllocPageEx(Const Process:
  THandle; Const Address: Pointer; Const
  Size, AllocationType, Protect:
  Cardinal): Pointer;
Begin
  Result := NIL;
  If IsWindowsNT Then
    Result := _VirtualAllocEx(Process,
      Address, Size, AllocationType,
      Protect)
  Else
    Result := VirtualAlloc(Address, Size,
    AllocationType Or VA_SHARED, Protect);
End;

Function FreePageEx(Const Process: THandle;
  Const Address: Pointer; Const Size,
  FreeType: Cardinal): Boolean;
Begin
  Result := False;
  If IsWindowsNT Then
    Result :=  VirtualFreeEx(Process,
      Address, Size, FreeType)
  Else
    Result := VirtualFree(Address, Size,
      FreeType);
End;
```

Möchten wir beispielsweise auf die Position der Desktop-Icons Zugriff erlangen, so wird dazu eine funktionsfähige, allozierte Speicherseite im Windows-Explorer-Prozess benötigt, da wir auf das Desktop-Fenster, welches die Icons verwaltet, nur über vordefinierte *Window Messages* zugreifen können; und diese speichern die angefragten Daten an einem vom Programmierer angegebenen Zeiger im Adressraum des Fensterprozesses.

Die auf Heft-CD verfügbare Beispielapplikation macht sich obige Funktionen zu Nutze, um eine Speicherseite im Explorer-Prozess zu allozieren, und ist somit in der Lage, auf die Position der Icons lesend und schreibend zugreifen zu können. Die Applikation sichert nämlich die Position der Icons und stellt diese bei Bedarf wieder her.

(Oliver Friedmann)