Los rincones del API Win32
La pila

Introducción

La pila es una de las estructuras más importantes dentro del modelo de memoria de un proceso en Win32. En el artículo anterior, dijimos que cada vez que se ejecuta la función CreateProcess (ya sea iniciando un programa o haciendo una llamada directa), se crea un espacio de direcciones virtuales de 4 Gb (para más detalles sobre este espacio de direcciones consultar el artículo sobre La Memoria Virtual en Win32 en www.lawebdejm.com/?id=22101), pero otras de las operaciones que se hace son:

- Reservar un bloque de memoria auxiliar dentro del espacio de direcciones del proceso. Este espacio se denomina Montón (o Heap).

- Crear el hilo principal de ejecución a través de la función CreateThread. Una de las operaciones que hace esta función es reservar un espacio de memoria denominado Pila (en inglés Stack).

En este artículo vamos a detallar el funcionamiento de la pila, y en el próximo (www.lawebdejm.com/?id=22130) haremos lo mismo para el montón, cubriendo así el funcionamiento de la dos estructuras de memoria más importantes dentro de un proceso.

Teoría de la pila

Una pila es una estructura de datos de tipo LIFO (Last In, First Out), es decir: el último elemento en entrar es el primero que sale. Por poner un ejemplo, podríamos decir que una pila es como una torre de libros, en el que el primero en colocarse (la base) será el último en poder ser extraído (se sacará cuando se retiren todos los que estén encima de él), y el último en colocarse (la cima) será el primero en salir. Las operaciones básicas con una pila son:

- Meter(X): (Push), consiste en meter un elemento X en la pila. Ese elemento pasará a estar en la cima de la pila y será el primero en poder ser sacado.

- X <- Sacar (Pop): consiste en sacar el elemento que en ese momento esté en la cima de la pila: el elemento X.

Representación interna

Una pila, como otras muchas estructuras de datos, se almacena internamente como un array o vector. El tamaño del array será el tamaño máximo de objetos que podrá albergar la pila. Así el siguiente array o vector:

```
var
   DatosPila: array[0..99] of integer;
begin

end;
```

Podrá albergar hasta 100 números enteros, y cuando se intente introducir el elemento 101, se producirá un error. Este error se denomina Desbordamiento de Pila, o *Stack Overflow*, y seguro que no es un nombre nuevo para muchos de vosotros.

Como dato curioso (aunque realmente no nos hes necesario), deciros que la pila se va llenando de la posición superior hacia la inferior, es decir, el primer elemento residirá en la posición de memoria X, el segundo en X-1, el tercero en X-2, y así sucesivamente hasta que llegemos hasta el final de la pila.

Para representar una pila, además de un array, hace falta un puntero que indique en qué posición se sitúa el elemento superior (el decir, el último introducido). Hay que decir, que esto no es realmente así, ya que en plataformas x86, se utilizan el registro EBP (registro base) y ESP (que apunta a la cima). Por simplicidad no utilizaremos los registros del procesador ya que sería necesario escribir en ensamblador.

Una representación completa de una pila podría ser:

```
var
  DatosPila:  array[0..99] of integer;
  PunteroPila: ^Integer; { indica la dirección del elemento superior }
```

Así, al iniciar la ejecución "PunteroPila" valdrá **nil**, es decir, la pila estará vacía. Cada vez que se inserte un elemento, "PunteroPila" se decrementará una posición, indicando así que la cima está una posición más arriba (en realidad es al revésesto es así por la razón que os comentaba antes: que la pila va de más a menos), y al sacar un elemento, "PunteroPila" se incrementará una posición (hasta llegar a **nil** si se vacía). Si "PunteroPila" se incrementa por encima de la posición 100, se produce un desbordamiento de pila.

En realidad, esto es precisamente al revés, ya que al crecer la pila se está avanzando hacia posiciones de memoria inferiores, y al disminuir la pila, se está avanzando hacia la posición más altas (por lo que os comentaba antes sobre que la pila va de más a menos). De todas formas, y aunque esto sea así, nosotros vamos a explicarlo de este otro modo, ya que puede clarificar más los conceptos.

Al meter elementos en la pila no estamos reservando ningún espacio, sino que simplemente ocupamos un espacio ya reservado. De igual modo, al sacar elementos de la pila tampoco liberamos, sino que lo dejamos en la misma situación para poder ser utilizado con posterioridad. Ni siquiera se limpia el contenido con ceros, sino que se deja con el valor de la última variable que ha ocupado esa posición (como luego veremos, esta es la razón por la que las variables locales pueden tomar valores correctos cuando hemos olvidado inicializarlas).

Debido a que las operaciones de Push y Pop no reservan ni liberan memoria, podemos garantizar que su ejecución es muy rápida, porque simplemente se desplaza el puntero por el espacio de la pila.

Usos y disfrutes

Todo hilo de ejecución cuenta con una pila de tamaño fijo (definido en tiempo de enlazado), y su uso es continuo. La tarea más importante para la que se utiliza es para almacenar todas aquellas variables locales y parámetros de un procedimiento o función, así el encadenamiento de llamadas de una función a otra (en realidad lo que se almacena es la dirección de retorno de cada una de las llamadas).

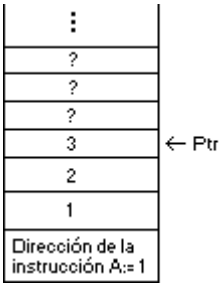
Vamos a poner un ejemplo para entenderlo mejor. Supongamos los siguientes bloques de código:

```
01    procedure ProcA;
02    var
03      A: integer;
04    begin
05      ProcB(1, 2, 3);
06      A := 1;
07    end;

08    procedure ProcB(x, y, z: integer);
09    var
10      var1, var2: integer;
11    begin
12      var1 := 55;
13      var2 := 22;
14    end;
```

- La ejecución de la **línea 05** supone los siguientes pasos:

    1. Meter en la pila la dirección donde reside la siguiente instrucción (A:=1);
    2. Meter en la pila los valores 1, 2, 3 (el orden puede variar)
    3. Hacer un salto en la ejecución hasta la dirección donde resida ProcB

    Al finalizar estos pasos la pila tendrá el siguiente aspecto:

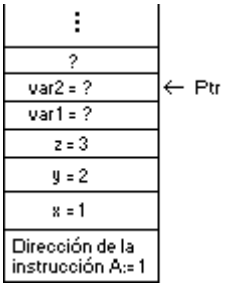


- Al entrar en ProcB (en la **instrucción 11**) se realizan las siguientes operaciones:

    1. Situar las variables “x”, “y” y “z” en sus posiciones de memoria en la pila.
    2. Meter en la pila las variables locales var1 y var2.
    3. Comenzar la ejecución

    Nota: En la Figura 2 se han marcado con el símbolo “?” los datos desconocidos de algunas posiciones de la pila. Esto significa que en esas posiciones habrá datos basura, siendo los valores que tuvieron las últimas variables en ocupar esas posiciones.

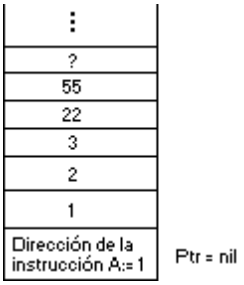
Al finalizar estos pasos la pila tendrá el siguiente aspecto:



- Al salir de ProcB (la ejecución de la **instrucción 14**) se realizan las siguientes operaciones:

    1. Sacar de la pila las dos variables locales.
    2. Sacar de la pila los 3 parámetros.
    3. Sacar de la pila la dirección de retorno y continuar la ejecución en ese punto (A:=1);

    Al finalizar estos pasos la pila tendrá el siguiente aspecto:



Como hemos visto, las variables locales son almacenadas en la pila por lo que su reserva y liberación será muy rápida. Además, no tenemos que preocuparnos de liberar la memoria, ya que al decrementar el puntero de pila, las variables quedan automáticamente descartadas, aunque el espacio no se libera (ni se vacía el contenido). En esta situación, la pila se considera vacía (aunque la memoria esté reservada y con datos), porque esas casillas pueden ser ocupadas por las próximas variables.

La principal ventaja de la pila es que su manipulación es muy rápida aunque también tiene inconvenientes: su tamaño es fijo (definido durante la etapa de enlazado) y limitado (para algunos casos demasiado pequeño), y además sólo es posible almacenar variables de tamaño conocido en tiempo de compilación (porque es necesario saber cuantos bytes debemos incrementar o decrementar el puntero de pila.

El ejemplo que os he presentado solamente involucra la llamada a una función pero, como os podréis imaginar, este proceso se repetirá por cada llamada a una función, ya sea de forma independiente, anidadas o incluso recursivamente. A cada grupo de elementos de la pila referidos a una misma función (la dirección de retorno, los parámetros y las variables locales), se le denomina **marco de pila** (*stack frame*).

Paso de parámetros a través de la pila

Ya hemos visto como se utiliza la pila como medio para pasar parámetros de una función a otra, pero ha habido un punto que hemos dejado en el aire: ¿en qué orden se introducen esos parámetros en la pila?

La respuesta a esta pregunta no es única, ya que se utilizan distintos métodos para realizar esta tarea. Desde los inicios de la programación se han utilizado varios métodos, cada uno aplicado a un lenguaje de programación o a una situación concreta, hoy en día se sigue sin estandarizar el método.

Para saber en qué orden se introducen en la pila los parámetros de una función, se utiliza lo denominado convenios de llamada (*calling convention*) que no es más que una forma establecida y conocida de pasar parámetros a funciones. Inicialmente había un convenido de llamada para cada lenguaje de programación, aunque con el tiempo se han ido reduciendo a cinco. Estos convenios de llamada, además de definir el orden en que se introducen los parámetros en la pila, definen quién es el responsable de sacar estos parámetros para que la pila quede vacía.

Los cinco convenios de llamada utilizados hoy son:

- **Pascal**
  Los parámetros se evalúan e introducen en la pila de izquierda a derecha y será la función llamada la que saque estos parámetros de la pila antes de retornar. Este convenio es el que se utilizó en un principio en los primeros compiladores de Pascal.

- **cdecl**
  Los parámetros se evalúan e introducen en la pila de derecha a izquierda (empezando por el último) y será la función que realiza la llamada la responsable de limpiar la pila una vez que se ha ejecutado de la función.
  Este es el convenido por defecto utilizado en los compiladores ANSI C/C++. Esta es la razón por la que las funciones de C/C++ permiten tener un número desconocido de parámetros (printf, scanf, etc.) ya que utilizan este convenido de llamadas.

- **stdcall**
  Los parámetros se evalúan e introducen en la pila de derecha a izquierda (empezando por el último) y será la función llamada la que saque los parámetros de la pila antes de retornar.
  Este convenido es el utilizado en las llamadas a las funciones del API Win32.

- **safecall**
  Los parámetros se evalúan e introducen en la pila de derecha a izquierda (empezando por el último) y será la función llamada la que saque los parámetros de la pila antes de retornar.
  Este convenio fue creado para utilizarse en la programación COM.

- **register**
  Los parámetros se evalúan e introducen en la pila de izquierda a derecha y será la función llamada la que saque estos parámetros de la pila antes de retornar. La diferencia con el convenio pascal es que los tres primeros parámetros no se introducen en la pila, sino que se utilizan los 3 registros de la CPU siempre que sea posible. Si no es posible guardar los datos en registros (por ejemplo si no se pueden representar como valores de 32 bits) y para los parámetros por encima del tercero, se utilizará la pila como en el resto. Esto hace que las llamadas a las funciones se realicen mucho más rápido, por lo que el convenio register es el más eficiente (de ahí el nombre que se da a este convenio en C/C++: *fastcall*). Este es el convenio por defecto utilizado por Delphi

(si está activa la optimización), aunque puede ser cambiado con los modificadores correspondientes en los prototipos de las funciones:

```
procedure UnaFuncionRapida(parametro: integer); register;
function UnaFuncionEstandar(parametro: integer): integer; stdcall;
```

Peligros de la pila: el "buffer overflow"

Como hemos visto, los datos en la pila se van situando consecutivamente, y el orden de estos dependerá del convenio de llamadas que estemos utilizando. Independientemente de este orden, lo que es indudable es que corremos el riesgo de sobrescribir un dato si no respetamos bien las fronteras.

Supongamos un código en el que tengamos un vector de números y además, una variable que contiene un número importante (por ejemplo una clave de usuario, un código hash o algo de vital importancia). Es sencillo codificar esto en un Pascal muy básico:

```
var
    i:            integer;
    importante:   integer;  { el valor de esta variable es my importante }
    numeros:      array[0..4] of integer;
    una_variable: integer;
begin
    i            := 0;
    importante   := 222;
    una_variable := 0;

    for i := 0 to 5 do
       numeros[i] := 999;

    WriteLn(Format('importante vale %d', [importante]));
end;
```

Para que este código compile correctamente desde Delphi, debemos desactivar todas las optimizaciones de código y las opciones *Range checking* y *Overflow Checking* en la ventana *Project - Options - Compiler - Runtime errors*.

Una vez compilado, ¿qué se muestra por pantalla? No es muy compilado. Sólo asignamos valor una vez a la variable "importante", antes del bucle, por lo que su valor permanecerá constante, y en la pantalla aparecerá "importante vale 222".

Demasiado fácil. El código tiene pequeño bug, aunque no lo hemos apreciado hasta ahora. Para destapar este bug vamos a cambiar simplemente el orden en que declaramos las variables. Nos bastará con situar la variable "importante" por debajo de la variable "una_variable". El código queda así:

```
var
    i:             integer;
    numeros:       array[0..4] of integer;
    una_variable: integer;
    importante:    integer;  { el valor de esta variable es muy importante }
begin
    i             := 0;
    importante    := 222;
    una_variable := 0;

    for i := 0 to 5 do
       numeros[i] := 999;

    WriteLn(Format('importante vale %d', [importante]));
end;
```

Y ahora... ¿qué se muestra por pantalla? Cualquiera diría que lo mismo. ¿en qué va a influir el orden de las variables al resultado final? Eso sería cierto si no hubiéramos cometido el bug que os comentaba. Si nos molestamos en compilar y ejecutar este código, veremos que por pantalla aparece un "importante vale 999". ¡Pero qué está pasando aquí!

Veamos el aspecto de la pila en este último código, después de ejecutar la instrucción "importante := 222":

Dirección	Variable	Valor
⋮	⋮	⋮
10	i	0
14	numeros[0]	???
18	numeros[1]	???
22	numeros[2]	???
26	numeros[3]	???
30	numeros[4]	???
34	importante	222
38	una_variable	???
42	dir. retorno	358
⋮	⋮	⋮

El esquema contiene tres columnas:

- **Dirección**: la dirección que ocupa en la memoria el elemento de la pila que estamos representando. Como podemos ver, las direcciones van dando saltos de 4 en 4, ya que cada uno de los elementos de la pila ocupa 4 bytes (una palabra de 32 bits). Así, en la dirección 10 se sitúa el primer elemento, 4 bytes más adelante el segundo, 4 bytes más el tercero, y así sucesivamente.

- **Variable**: Nombre del símbolo (es decir, la variable) que ocupa esta dirección de memoria.

- **Valor**: Valor que toma esa dirección de memoria en cada momento.

Como vemos, la variable "i" contiene el valor "0" (establecido en la primera línea de código), la variable "importante" contiene el valor "222", y el resto de variables (incluidos cada uno de los elementos del vector) contienen valores desconocidos o *basura.*

Conforme vamos ejecutando cada una de las vueltas del bucle, vamos asignando valores a los elementos del vector, desde i=0 hasta i=5:

- **i=0**: se asigna el número 999 al elemento numeros[0]
- **i=1**: se asigna el número 999 al elemento numeros[1]
- **i=2**: se asigna el número 999 al elemento numeros[2]
- **i=3**: se asigna el número 999 al elemento numeros[3]
- **i=4**: se asigna el número 999 al elemento numeros[4]
- **i=5**: se asigna el número 999 al elemento numeros[5]. Este elemento en realidad no existe, ya que el vector tiene 5 elementos, desde 0 hasta 4. El compilador de Delphi es capaz de detectar esta situación, pero al haber desactivado la opción de compilación *Range Checking* estamos evitando este tipo de comprobaciones. Al producirse este error, lo que ocurre es que se almacena el dato **en el supuesto siguiente elemento**, es decir, 4 bytes más adelante del último elemento del vector. Como podemos ver, en esa posición está situada la variable "importante", por lo que esta última vuelta del bucle asignará el valor 999 a la variable "importante".

¿Y esto por qué no ocurría con el primer código que escribimos? Pues muy sencillo: porque la situación de las variables en la pila era distinta, ya que las habíamos declarado en un orden distinto. Concretamente, la situación era la siguiente:

Dirección	Variable	Valor
⋮	⋮	⋮
10	i	0
34	importante	222
14	numeros[0]	???
18	numeros[1]	???
22	numeros[2]	???
26	numeros[3]	???
30	numeros[4]	???
38	una_variable	???
42	dir. retorno	358
⋮	⋮	⋮

En rojo podemos ver que la variable "importante" está situada **antes** del vector, por lo que el bug, que sólo afecta a la variable que está almacenada detrás del vector, no sobrescribe el valor de la variable "importante", si no el valor que tenga la variable "una_variable" (en nuestro caso irrelevante).

A estas alturas ya tendréis una solución al bug: basta con hacer que el bucle termine una vuelta antes, concretamente en la instrucción **for**

```
begin
    { todas las instrucciones anteriores }

    for i := 0 to 4 do
       numeros[i] := 999;

    { todas las instrucciones posteriores }
end;
```

Se puede decir que este error es muy común para cualquier programador, aunque sea un experto. El ejemplo que he plateado es muy sencillo de detectar y corregir (de hecho, en situaciones normales el compilador lo detectará), pero cuando utilizamos aritmética de punteros, lo más

normal es que alguno *se nos vaya de madre*, y acabe sobrescribiendo otras variables. Este viejo error es tan común que incluso tiene nombre: el bug del **buffer overflow** (**desbordamiento de buffer**).

Uno de los mayores problemas de este bug es que lo suelen utilizar los **hackers** para forzar a un programa a que haga tareas que no debería. Básicamente se trata del mismo ejemplo que he puesto arriba, pero si nos fijamos con cuidado, podremos ver como más abajo de las variables "importante" y "una_variable" se sitúa la dirección de retorno. Esta dirección la utiliza el procesador para saber a qué punto del programa debe saltar una vez que ha terminado de ejecutar la función. En nuestro caso, saltará a la instrucción situada en la dirección de memoria 358.

La técnica consiste en detectar un bug de desbordamiento de buffer que contenga el programa a *hackear*, y aprovecharlo para sobrescribir la dirección de retorno, estableciendo un valor que corresponde con la dirección donde el hacker a situado un código propio.

En nuestro caso tan sólo sobrescribíamos 4 bytes por detrás del vector, pero podríamos haber sobrescrito, por ejemplo, 8 bytes: los 4 primeros corresponderían a la variable que se sitúa por detrás del vector, y los 4 siguientes corresponderían a la dirección de retorno. Si en la posición de la pila correspondiente a la dirección de retorno, situamos una dirección válida (nos vale con una dirección que pertenezca al segmento de código del proceso), habremos conseguido nuestro propósito.

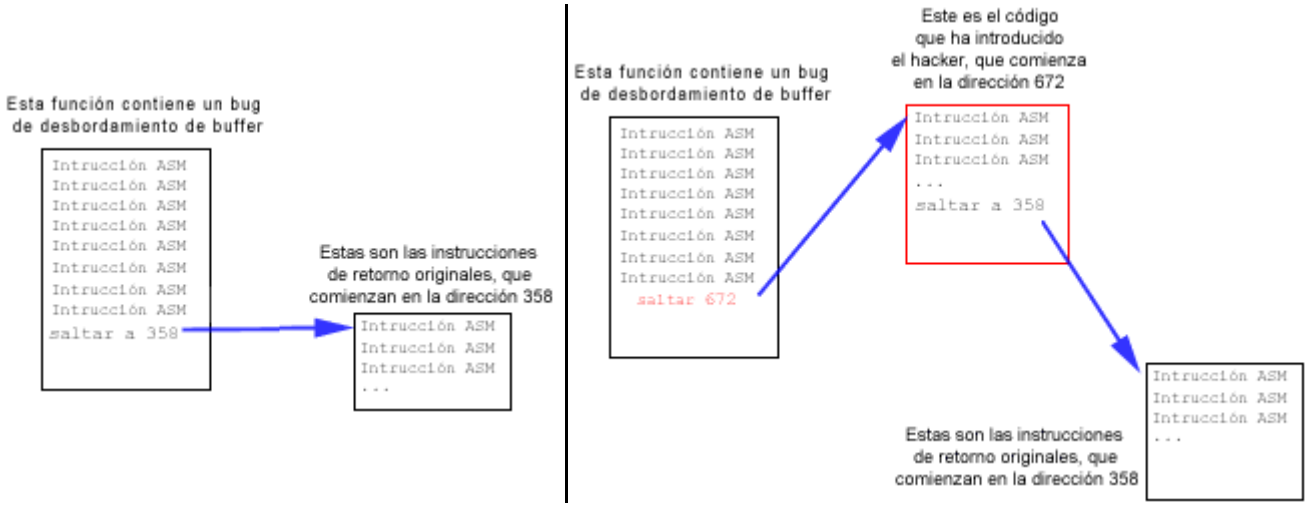
Un ejemplo: el pirata "Patapalo" ha escrito un código "maligno" y sabe (gracias a un *disassambler*) que este código comienza en la dirección de memoria 672.

La juguada está en conseguir que el programa sobrescriba la posición de memoria donde reside la dirección de retorno, sobrescribiendo el antiguo 358 por el nuevo 672.

Para conseguir esto, los hackers aprovechan puntos de entrada de datos al programa (lectura de ficheros, campos donde el usuario teclea un dato, etc.).
Si en alguno de estos puntos, el programa contiene un bug de desbordamiento de buffer, podemos introducir más datos de los que el programa espera, por lo que estaremos desbordando alguno de sus buffers internos. Si esta situación la controlamos, podemos sobrescribir hasta donde queramos, exactamente hasta la dirección de retorno, estableciendo el valor que queramos.

Una vez hecho esto, el procesador (ajeno a todo lo que ha pasado), terminará la ejecución y hará que se salte a la dirección 672, donde comenzará a ejecutarse el código maligno (por ejemplo enviar por mail el valor de una variable, almacenarlo en un archivo, cambiar valores a variables globales, etc.).
Cuando termine de ejecutarse este código maligno, el hacker habrá puesto cuidado en hacer que la ejecución continúe donde tendría que haberlo hecho en un principio: en la dirección de memoria 358.

En la siguiente imágen puede verse cómo quedaría la ejecución antes (a la izquierda) y después de la modificación del hacker (a la derecha). La caja roja representa el código que ha escrito el hacker, y la línea en rojo representa la línea que ha modificado el hacker.



En nuestros programas debemos intentar evitar situaciones como esta, siempre que sea posible. Un buen modo de hacerlo es limitando todos los puntos en que el usuario introduce datos, de un modo en que nuestros buffers (estáticos o dinámicos) nunca puedan desbordarse y sobrescribir otras variables o incluso la dirección de retorno.

Bueno, hasta aquí la teoría. Ahora vamos a ver cómo se implementa la pila de un hilo en Win32.

La Pila en Win32

Al crear un proceso, automáticamente se crea el hilo principal de ejecución a través de la función CreateThread. Esta función una de las tareas que hace es crear la pila. Evidentemente, si nuestra aplicación es multi-hilo, tendremos una pila por cada uno de los hilos de ejecución que hayamos creado.

La pila, al representarse en memoria como un vector, no será más que un bloque de memoria contigua. Y como ya dijimos en el anterior artículo del API Win32, todo bloque de memoria en Win32, es un bloque de memoria virtual. Así que la pila de un hilo no va a ser menos, y como tal, la reserva de la pila se hará internamente a través de la función VirtualAlloc.

Como también dijimos, la memoria virtual requiere de dos operaciones: reserva y compromiso, así que el sistema debe reservar un espacio de memoria prefijado (para disponer de un bloque de memoria contigua), y comprometerlo. Pero... ¿se compromete todo? ¿parte? ¿nada? Estos datos (tanto el tamaño total de la pila como la cantidad que debe ser comprometida) se definen en los parámetros del enlazador.
Esto lo podemos encontrar en la opción Project - Options - Linker: Veremos como en la sección "Memory Sizes" aparecen 2 campos referido a la pila (stack):

- Min Stack size: tamaño a comprometer en la pila del hilo principal del proceso.
- Max Stack size: Tamaño máximo de pila.

Los datos por defecto para la pila del hilo principal será de 1 MB ($00100000 en hexadecimal) y que inicialmente se comprometerán los primeros 16 KB ($00004000 en hexadecimal). Como ya dijimos, la operación de reserva de memoria virtual es muy rápida, pero la de compromiso es relativamente lenta, así que al comprometer sólo los primeros 16 KB, conseguimos que nuestro proceso se inicie más rápido. Como norma general, podemos decir que cuanto más pequeño sea el compromiso inicial, más rápido se iniciará la aplicación.

Hasta aquí todo claro, pero la primera duda que puede surgir es la siguiente: ¿qué pasa cuando la pila crece por encima de los primeros 16 KB? Ese espacio, al estar reservado pero no comprometido no puede ser utilizado, ya que se produciría una violación de acceso.

Bien, para entender esto hay que dar un vistazo a la implementación interna de una pila en Win32.

Implementación interna

Antes de comenzar este punto, , hay que decir que este aspecto varía mucho dependiendo del tipo de sistema Win32, ya sea un sistema NT (Windows NT, 2000 y XP) ó 95 (Windows 95, 98 y Me). La explicación que vamos a dar aquí se refiere a la implementación en los sistemas NT/2000/XP, ya que son los más evolucionados y su uso tiene a generalizarse.

La pila, como cualquier estructura que reside en memoria, ocupará páginas (de 4KB cada una en procesadores x86). Así que una pila de 1MB ocupará 256 páginas de memoria.

Al crearse la pila, se compromete el espacio indicado durante el enlazado, que como ya sabemos es por defecto de 16 KB, es decir 4 páginas. La última página que se comprometa tendrá una característica especial: se marcará con la bandera de protección PAGE_GUARD.

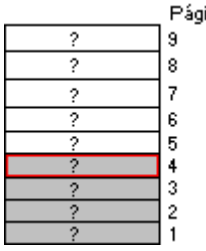
Si recordamos la descripción de la función VirtualAlloc, veíamos que su cuarto parámetro hacía referencia a la protección que se le daba a las páginas que estábamos reservando y/o comprometiendo. Ya explicamos que podíamos asignar banderas de PAGE_READWRITE, PAGE_NOACCESS y otras, pero no explicamos la existencia de la bandera PAGE_GUARD.

Esta bandera puede combinarse con otras durante la llamada a VirtualAlloc, y hace que cuando dicha página sea accedida, se lance la excepción de sistema STATUS_GUARD_PAGE y no se permita el acceso a dicha página. Hasta aquí su comportamiento es similar a una página marcada con PAGE_NOACCESS, excepto que una vez que se ha lanzado esta excepción, el sistema elimina automáticamente la marca de PAGE_GUARD, y los siguientes accesos se harán correctamente.

Por ejemplo, el siguiente código reserva y compromete una página y la marca como PAGE_GUARD. De los accesos siguientes tan sólo tendría éxito el segundo:

```
// ATENCIÓN: este código sólo funciona en sistemas Win NT/2000/XP, ya que
// los sistemas 95/98/Me no soportan la bandera PAGE_GUARD en la llamada
// a VirtualAlloc

var
  P: PChar;
begin
  P := VirtualAlloc(nil, 4096, MEM_RESERVE or MEM_COMMIT,
                    PAGE_READWRITE or PAGE_GUARD);

  try
    StrPCopy(p, 'intento de grabar dato, pero no lo voy a conseguir');
  except
    on e: Exception do
      MessageBox(GetActiveWindow, PChar(e.message),
                 'Excepción capturada', MB_ICONSTOP);
  end;

  StrPCopy(p, 'este dato sí que voy a poder guardarlo');
  MessageBox(GetActiveWindow, p, 'Dato guardado', MB_ICONINFORMATION);

end;
```

Bien, ahora ya sabemos cómo funcionan las páginas de guarda, y también sabemos que la última página que ha sido comprometida en la pila, estará marcada con la bandera PAGE_GUARD.



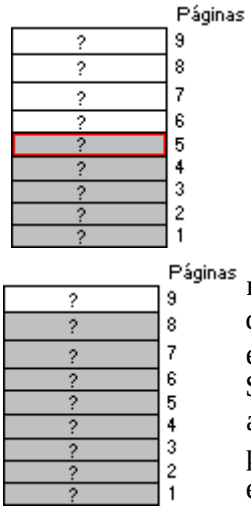
El aspecto de la pila de un hilo recién iniciado puede ser como el que aparece en la siguiente figura:

La pila completa está en estado reservado (páginas de color blanco), las 4 primeras páginas han sido comprometidas (en color gris) y la última página comprometida ha sido marcada como página de guarda (borde en rojo). Como ya hemos visto, el contenido de las páginas se marca con un interrogante porque contendrán datos basura, posiblemente restos de las últimas variables que han residido en esas posiciones de memoria.

Bien, conforme va evolucionando la ejecución de nuestro proceso, la pila se va llenando (cuanto más profundo sea el árbol de llamadas a funciones) o vaciando (según estas funciones van retornando). Puede ser que no sea necesario sobrepasar el almacenamiento de las 3 primeras páginas (12 KB), así que en este caso tan sólo habremos consumido 16 KB de memoria para almacenar la pila. Pero en otros muchos casos, llegará un momento en que algún dato sea necesario guardarlo en la cuarta página. En ese momento, al ser una página de guarda, se producirá una excepción de tipo STATUS_GUARD_PAGE. El sistema es capaz de gestionar sus propias excepciones, así que la capturará y realizará las siguientes operaciones:

1. Comprometer una página más de memoria (en nuestro caso, la 5ª)
2. Marcar con PAGE_GUARD las nueva página comprometida.
3. Volver a guardar el dato que no ha podido ser guardado por la excepción (recordad que la bandera PAGE_GUARD desaparece después del primer acceso).

El nuevo aspecto de la pila puede observarse en el siguiente esquema:



Como hemos visto, este mecanismo permite que la pila vaya creciendo conforme necesita más espacio. Si el árbol de llamadas diminuye, las páginas restantes de la pila no se liberan, sino que permanecen comprometidas para acelerar su acceso en un futuro.

Ahora supongamos que el árbol de llamadas crece hasta que es necesario acceder a la página 8 (la penúltima). Al ser una página de guarda, se producirá una excepción y el sistema la capturará del mismo modo y realizará las mismas operaciones, excepto que si la nueva página comprometida (la 9ª) es la última, no se marcará como página de guarda. El estado en que quedaría la pila sería el siguiente:
Si siguen produciéndose llamadas anidadas (y/o recursivas) y la pila intenta acceder a la 9ª página (que está reservada, pero no comprometida), se producirá una violación de acceso, y para evitar males mayores el sistema elimina automáticamente el proceso padre del hilo.

Como hemos visto, el sistema es capaz, tanto de hacer que la pila crezca automáticamente, como de asegurarse de que no crece indefinidamente. Pero para realizar esta última tarea, el tamaño útil de la pila se ve reducido en una página, ya que la página superior nunca será comprometida.

Esto se hace así para evitar sobrescribir páginas de memoria que estén por encima de la pila. Si suponemos que la última página se compromete normalmente, el sistema podría continuar almacenando datos en la pila hasta su límite superior, y si al intentar acceder a la siguiente página (la que esté por encima de la 9ª, que ya no pertenece a la pila). Si además se da la casualidad de que esa página ha sido comprometida (porque en ese punto reside otra estructura de datos), no se produciría ninguna excepción y todo parecería funcionar correctamente, por lo que el error sería difícilmente detectable.

Conclusión

En este artículo hemos dado una visión teórica de lo que es una pila, sus uso y su implementación en la arquitectura Win32.

No se puede decir que haya sido un artículo meramente práctico, sino de conceptos básicos, en el que espero que hayáis adquirido el conocimiento de fondo suficiente para solucionar problemas complejos de desbordamiento de pila.

En el próximo artículo profundizaremos en otra de las estructuras de memoria de Win32: Los montones (www.lawebdejm.com/?id=22130).

Autor: JM - http://www.lawebdejm.com