SESSIÓN TP10

Sistemas Operativos 1
Mayo 2023

1 INTRODUCCIÓN

En esta sesión se repasarán varios aspectos con los que se debe tener cuidado a la hora de programar en C. En particular, se harán experimentos con dos aspectos de la programación en C que suelen traer dificultades:

1. La **pila**, una zona de memoria en la que se almacenan las variables locales declaradas a una función. La pila es una región de memoria relativamente pequeña y debe evitarse utilizar la pila para almacenar vectores estáticos que tengan un tamaño relativamente grande. ¿Qué recomendamos para evitar dificultades a la hora de programar?
2. El **acceso a los elementos de un vector**, sea estático o dinámico. El lenguaje C no realiza ninguna comprobación si el acceso es válido como otros lenguajes, como Python o Java. Esto puede provocar que el programa "pete" en una línea de código diferente de la que se ha producido el problema. ¿Cómo detectar estos problemas?

2 TAMAÑO DE LA PILA

Empecemos por intentar analizar el tamaño de la pila en un proceso. En concreto, se propone realizar una prueba con el código *recursive.c*.

```
void funcion(int a)
{
   int b[1000];
   printf("%d\n", a);
   funcion(a+1);
}

int main(void)
{
   funcion(0);

   return 0;
}
```

Observar que el código es una función recursiva en la que, en cada iteración, reserva memoria para vector estático de enteros (cada entero ocupa 4 bytes de memoria). Ejecuta el código y observa la iteración en la que el programa deja de funcionar.

Pregunta: En la pila se almacena, en cada llamada recursiva, al menos el parámetro a, el vector b y la dirección de retorno de la función (8 bytes). ¿Cuál es el tamaño, aproximadamente, máximo que un sistema operativo asigna a la pila?

Vamos a hacer un experimento para mostrar que reservar espacios de memoria grandes en la pila puede producir problemas inesperados y que pueden ser difíciles de resolver si no se es consciente del tamaño de la pila.

Analiza el código *proves_mida_pila.c*.

```
#define NCOLS 256

void funcio(int nrows)
{
   char matriz[nrows][NCOLS];

   printf("Hago asignacion y operacion\n");
   // Esto se hace para que el compilador no se queje
   // que no se utiliza la variable matriz
   matriz[0][0] = 0;
   matriz[0][0] = matriz[0][0] + 1;
   printf("Salgo asignacion y he hecho operacion\n");
}

int main(int argc, char **argv)
{
   int nrows;

   if (argc != 2) {
     printf("Uso: %s <nrow>\n", argv[0]);
     éxito(1);
   }

   nrows = atoi(argv[1]);

   printf("Valor de nrows: %d\n", nrows);
   printf("Llamado a funcion\n");

   funcion(nrows);

   printf("Salgo de funcion, todo ha ido bien!\n");

   return 0;
}
```

Observar que el código reserva una matriz estática. Al ejecutar el código puede pasarse como argumento el número de filas de la matriz.

Intente ejecutar el código con diferentes valores (positivos) de `nrow`: 10, 100, 1000, 10.000, 20.000, 30.000, 40.000 y así sucesivamente. Vaya aumentando el valor hasta que vea que la aplicación deje de funcionar.

Pregunta: ¿A partir de qué valor de `nrow`, aproximadamente, peta la aplicación? ¿Cuál es el tamaño en bytes correspondiente?

Este experimento demuestra que la pila es relativamente pequeña. Por eso se recomienda que, en general, utilizar memoria dinámica en vez de memoria estática si se quieren utilizar vectores (o matrices) de tamaño grande.

¿Cuál es la eficiencia asociada si (`read` o `fread`) le lee un fichero de byte en byte o en bloques de tamaño 4096 bytes?

3 LÍMITES DEL VECTOR

En el lenguaje C se pueden utilizar vectores, sean estáticos (los que se almacenan en la pila) o dinámicos (cuya memoria se reserva con `malloc`). Una característica particular del lenguaje C es que no se comprueba en tiempo de ejecución si los accesos que se realizan a vectores son válidos. Veamos un ejemplo, código *vector_estatic.c*.

```
int main(void)
{
  int a[10];

  printf("Faig assignacions\n");

  a[100] = 1234;
  printf("a[100] = %d\n", a[100]);

  a[1000] = 4321;
  printf("a[1000] = %d\n", a[1000]);

  printf("Surto del main\n");

  return  0;
}
```

Observar que por el vector estático sólo se ha reservado memoria para 10 enteros. En el código se puede ver que se acceden a elementos fuera de vector. ¿Qué ocurre al ejecutar el código?

Con el ejemplo anterior hemos visto que el lenguaje C no avisa si escribimos fuera del vector. La pregunta que podemos hacernos es: ¿dónde estamos escribiendo? ¿Qué estamos sobreescribiendo? Por eso se proponen dos experimentos. A continuación se muestra el primer código, *vector_estatic_funcio.c*

```
void funcio()
{
  int a[1000];
  int b;

  b = 5;

  printf("Direccio de b: %x\n", &b);
  printf("Valor de b: %d\n", b);

  printf("Direcció de a[0]: %x\n", &a[0]);
  printf("Direccio de a[-1]: %x\n", &a[-1]);

  a[-1] = 10;

  printf("Valor de b: %d\n", b);
}


int main(void)
{
  funcio();
}
```

Se pide compilar el código (sin opciones de optimización) y ejecutarlo. Observe lo que ocurre al modificar `a[-1]`, ¡estamos modificando el valor de `b`!.

Pregunta: ¿por qué al modificar `a[-1]` estamos modificando el valor de `b`? Realizar un dibujo de la pila para verlo claro.

Uno de los problemas asociados a utilizar vectores es que, al sobrescribir un valor fuera del vector, podemos modificar los valores de otras variables. Esto puede ocurrir tanto con vectores estáticos como dinámicos.

Vamos algo más lejos con el siguiente ejemplo. Sabemos que en la pila se almacena la dirección de retorno de la función (es decir, el punto donde debe continuar la ejecución al devolver de la función). ¡Vamos a intentar sobrescribirla para petar el programa! Aquí se muestra el ejemplo *vector_estatic_funcio2.c*

```
int funcio(void)
{
  int a[10];
  int i;

  printf("Entro al bucle\n");

  for(i = 12; i < 20; i++)
  {
    a[i] = 2 * i;
    printf("a[%d] = %d\n", i, a[i]);
  }

  printf("Surto del bucle\n");

  return  0;
}

int main(void)
{
  printf("Entro a la funcio\n");
  funcio();
  printf("Surto de la funcio\n");
}
```

Intenta ejecutar el código. En el bucle que hay estamos sobrescribiendo información crítica que hay en el montón (entre otras cosas la dirección de retorno de la función). El código peta al salir de la función, no al hacer el bucle.

En conclusión, estos ejemplos muestran que utilizar vectores estáticos pueden dar problemas de ejecución en el lenguaje C. Además, estos problemas son difíciles de encontrar y arreglar. Por eso se recomienda utilizar, en general, vectores dinámicos para almacenar datos en un vector. Aunque el lenguaje C tampoco da un error en caso de que accedamos fuera del vector, existen en la actualidad herramientas que permiten detectar y encontrar rápidamente los accesos inválidos que se realizan al utilizar vectores dinámicos.

4 VECTORES DINÁMICOS

Empecemos con un ejemplo similar a los anteriores en los que demostramos que C no avisa, en tiempo de ejecución, si accedemos (sea por lectura o escritura) fuera de un vector ubicado de forma dinámica. Utilizaremos la función `malloc`, que es una llamada a librería de usuario. Internamente, la función `malloc` utiliza la función `sbrk`, que es la llamada a sistema para pedir memoria dinámica. La función `malloc` intenta minimizar el número de llamadas a `sbrk` para evitar las llamadas al sistema. Veamos el código *vector_dinamic.c*.

```
int main(void)
{
   int y;
   int *a;

   a = malloc(10 * sizeof(int));

   e = 0;
   while (1)
   {
     a[i] = 2 * i;
     printf("a[%d] = %d\n", y, a[i]);
     i++;
   }

   free(a);

   return 0;
}
```

Ejecuta este código y observa que el código puede escribir fuera de la zona de memoria dinámica que hemos reservado. Nosotros sólo hemos pedido 40 bytes de memoria. Sin embargo, internamente la función `malloc` reserva una zona de memoria mayor que los 40 bytes que se piden. La función `malloc` pide una zona grande y gestiona los espacios libres.

En nuestro ejemplo, hemos pedido sólo 40 bytes. `Malloc` ha pedido mucho más que 40 bytes en el sistema operativo. Dado que esta zona ha sido asignada al proceso, puede escribirse en ella.

Una de las ventajas de utilizar memoria dinámica es que estos tipos de errores de acceso se pueden detectar fácilmente con herramientas específicas. Una de estas herramientas es el **valgrind** (que está instalado en la máquina virtual).

5 MÁS EJEMPLOS

Se presentan a continuación otros ejemplos. Son códigos no válidos, pero que dan resultados asombrosos.

Ejecuta los códigos y a ver si puede explicar el comportamiento del código. Éste es el primer código, *vector_dinamic_free.c*

```
int main(void)
{
   int y;
   int *a;

   a = malloc(100 * sizeof(int));

   for(i = 0; y < 100; i++)
     a[i] = 2 * i;

   free(a);

   printf("a[50] = %d\n", a[50]);

   return 0;
}
```

Pregunta: Observe que se libera la memoria del vector y luego se imprime el valor de `a[50]`. ¿Puede explicar por qué se imprime el valor correcto?

Aquí tiene un ejemplo similar, *vector_dinamic_free2.c*.

```
int main(void)
{
   int y;
   int *a, *b;

   a = malloc(100 * sizeof(int));

   for(i = 0; y < 100; i++)
     a[i] = 2 * i;

   free(a);

   b = malloc(1000 * sizeof(int));

   b[50] = 1000;
   printf("a[50] = %d\n", a[50]);

   free(b);

   return 0;
}
```

Pregunta: observe que en el código modificamos el valor de `b[50]`, y después imprimimos el valor de `a[50]`. ¿Puede explicar lo que sucede?

6 EJERCICIO A ENTREGAR

Cómo se ha podido ver el lenguaje C no realiza, en tiempo de ejecución, comprobaciones sobre si el acceso que se hace es válido. Tiene la ventaja de que las aplicaciones en C suelen ser más rápidas que las equivalentes en otros lenguajes. Existe el inconveniente de que es más difícil rastrear los posibles problemas de acceso a memoria que tenga nuestra aplicación.

En la actualidad existen herramientas que nos facilitan el trabajo para encontrar estos problemas. En Linux una herramienta conocida es la aplicación **valgrind**. Intuitivamente, esta aplicación implementa su propia versión de `malloc` lo que permite detectar los problemas de acceso a memoria y un montón de cosas más.

El objetivo en este ejercicio es ver si la aplicación **valgrind** es capaz de encontrar los problemas que existen en nuestras aplicaciones. ¿Cómo se utiliza?

Para obtener información sobre los problemas es necesario

1. Compilar la aplicación en modo "debugger". Por eso es necesario utilizar la opción "-g" a la hora de compilar y no "-O" que es la opción que indica que se quiere optimizar el código.
2. Ejecutar la aplicación **valgrind** pasándole como argumento la aplicación compilada.

Veamos un ejemplo (código incluido en el directorio "ejercicio")

```
$ gcc -g codi_vector1.c -o codi

$ valgrind ./codi
```

Observe qué información nos da el **valgrind** al ejecutar la aplicación. ¿Te ayuda a detectar dónde está el problema? Repita el experimento los códigos codi_vector2.c, codi_vector3.c, codi_vector4.c, codi_vector5.c.

7 PROCESO DE EVALUACIÓN

Se pide entregar un **único informe** en **formato PDF** por pareja (grupXX.pdf, donde XX es el número de grupo de prácticas) de **máximo tres páginas** (sin incluir la portada/índice en caso de que desee incluirlos). Dicho informe debe incluir las pruebas que ha realizado ejecutando el **valgrind** con los códigos del ejercicio. Incluye, para cada código, el mensaje de error que le da el valgrind así como la conclusión que saque del experimento. En particular, ¿dónde está el problema? ¿Qué información le da valgrind al respecto?

Ser breves y concis@s a la hora de comentar qué información le da el **valgrind** respecto al problema que valgrind le indica que tiene el código.

Entregar un **único informe en formato PDF por grupo** (grupoXX.pdf, donde XX es el número de su grupo de prácticas). La evaluación por parte del profesor será entre 0 y 10. Para realizar esta evaluación se utilizarán criterios equivalentes a los que se realizan en los informes que entrega a prácticas. Es importante pues que interprete y comente los resultados que obtiene al ejecutar el código con el **valgrind**, y presente la información de forma clara y limpia.