Índice

1. Memoria en Unix

1.1. Introducción

En esta clase, se tratará de establecer la relación de los objetos de un programa con la virtualización de memoria provista por el sistema operativo; ello se hará prácticamente con cinco programas y sus resultados.

Los programas se encuentran en el directorio **03-memory**.

La idea fundamental es que, a través de la producción de programas en C (el menor nivel de programación antes del *assembler*) y utilizando algunas herramientas de gerenciación del sistema operativo, tratar de visualizar qué ocurre con variables y código desde el punto de vista del sistema operativo que influyan sobre el flujo y los resultados del programa.

Como ayuda general en relación a los programas de manejo y visualización relacionados con memoria, se aconseja ver las páginas de Web que más abajo se muestran.

1.1.1. Linux

- **5 Commands for Checking Memory Usage in Linux**
- **How to Check Memory Usage Per Process on Linux**

1.1.2. MacOS

- **View memory usage in Activity Monitor**.

1.2. 31-basic_sizes.c:

1.2.1. Código:

El objetivo de este programa es ver, en forma muy sencilla, los tamaños en bytes que el compilador adjudica a cada tipo existente en C.

Obviamente, este programa es muy sencillo y el único problema que puede tener para es el macro *show_size*.

Sin embargo, la forma de ver como se expande el macro, es usa el *preprocesador de C* para pedirle que haga las expansiones correspondientes y analizarlas.

Esta expansión puede producirse mediante el siguiente comando:

```
$ cpp 31-basic_sizes.c
```

1.2.2. Ejecución:

La ejecución solamente muestra la salida que aquí se reproduce

```
---C basic sizes:
1:  char
2:  short
4:  int
8:  long
8:  long long
8:  void *
4:  float
8:  double
16: long double
```

El resultado muestra que los tipos definidos por los lenguajes arrastran el problema de la *finitud* de la representación, a lo cual el programador debe estar atento para no producir *overflows* o *underflows* que cambien fundamentalmente los resultados e, incluso, la secuencia del programa.

Más aún, también estos resultados pueden cambiar dependiendo de la arquitectura subyacente e incluso, como se verá en el siguiente ejercicio, de la forma de direccionamiento que realiza el procesador (puede ser un procesador con una ALU de 32 bits pero direccionar mediante palabras de 16 bits, ya que la memoria está organizada de esa manera).

Como ejemplo de lo dicho, C nace en el ambiente de una PDP-11 de 16 bit (justamente para escribir Unix) y el primer compilador de C se desarrolla en esa arquitectura; el tipo *long long* obviamente no existía y solo existían los primeros 4 tipos.

El único tipo que tiene un tamaño constante (hasta ahora) es que el *char*, el cual es de 1 byte y no ha cambiado en los 50 años que median desde 1970.

Lo que se puede encontrar como *de facto standard* de aquella época es el libro de Kernighan-Ritchie *The C Programming Language* que lo único que establece como norma es la siguiente: *char* es de un byte, *int* tiene el tamaño *natural* de la máquina, *short* debe ser menor o igual al tamaño de *int* y *long* debe ser mayor o igual al tamaño del *int*.

Esto significó que en una máquina donde el tamaño de palabra es de 16 bit, resulta *short* e *int* de 16 bit y *long* de 32 bit; cuando se pasa a una máquina de 32 bit de palabra *short* pasa a ser de 16 bit e *int* y *long* de 32 bit.

Como C debe ser considerado dentro del ambiente de los programadores que trabajan *close to the metal* como el lenguaje de preferencia (sea para la construcción de sistemas operativos o para trabajar en lo que se denomina *embedded computers*) y además es importante que los programas sean transportables entre distintas plataformas, se debe tener un muy extremo cuidado en la elección de tipos.

1.3. 32-all_sizes.c

1.3.1. Código:

Este programa es una extensión del anterior para considerar lo que son los tipos denominados **agregados** (*aggregate types*) de los cuales las *estructuras y uniones* son parte de ellos.

El ejemplo de estructuras está tomado de la siguiente página de Web a la cual aconsejamos referirse: **About sizeof of structs**.

Las tres primeras definiciones de tipos de estructuras se corresponden exactamente con los ejemplos mostrados en la página de Web referida.

También se ha agregado un cuarto ejemplo de estructura, con miembros distintos de los anteriores, donde se ha agregado una instrucción para el compilador cual

es ___*attribute*___*((packed))*

Por último, y para completar con el otro tipo de agregado cual es la *union*, se han colocado tres *uniones* cuyos miembros coinciden con aquellos de los tres primeros ejemplos de estructuras.

1.3.2. Ejecución;

Es fácil analizar de la ejecución que el resultado de los tres primeros ejemplos coincide con los resultados mostrados en la página de Web aludida y ver que estos resultados son un poco sorprendentes, ya que *el tamaño total en bytes de cada estructura no es igual a la suma de los tamaños de cada uno de los miembros que la conforman.*

De acuerdo a la discusión establecida en clase, es obvio que el compilador trata de acomodar la forma de organización de los miembros en memoria a los requerimientos de direccionamiento de la arquitectura subyacente, haciendo que dicho direcionamiento sea más eficiente en código y en tiempo de ejecución pero no así en el espacio ocupado por la estructura que, obviamente, puede ser un poco mayor.

Sin embargo, el problema de ocupación de la estructura en memoria es de mucho menos importancia que la ocupación del código adicional y del tiempo de ejecución.

De todas maneras, hay casos en que el programador desea que los miembros de la estructura estén colocados en memoria en forma absolutamente adyacente, independiente de los requerimientos que imponga la arquitectura, de alguna manera *torciéndole la mano al compilador.*

Ello se logra con la instrucción para el compilador ___*attribute*___*((packet))* que, al ejecutar el programa, mostrará que en este caso *coincide la cantidad de bytes total de la estructura con la suma de la cantidad de bytes que ocupan cada uno de sus miembros.*

Sencillamente, una *union* se parece mucho en su definición a una *struct*; sin embargo, distinto del caso de la estructura donde cada miembro ocupa una zona de memoria propia, en el caso de la unión, *todos los miembros parecen ocupar la misma zona de memoria*, observándose que el tamaño de cada unión en el programa es el tamaño del mayor de los miembros.

¿Para qué se necesita un agregado como éste?. No es tan usual encontrar uniones en programas como lo es en el caso de las estructuras pero sin embargo, permiten varias aplicaciones que no serían imposibles de realizar sin su existencia, pero que hacen que sea mucho más sencillo el código y la comprensión si se usan las uniones.

Se remite a un hilo de **StackOverflow - Advantages of using unions** donde podrán encontrar algunos ejemplos interesantes, como el de *jmanning2k* y el de

Dav3xor.

1.4. 33-ambient.c

1.4.1. Código:

Este ejemplo es para completar la interfase del *shell* mediante los parámetros pasados a un proceso.

En efecto, hasta ahora se ha considerado que la función *main*, punto de comienzo de ejecución del programa cargado por una función de bibliotca de tipo *exec*, recibía solo dos argumentos, a saber *argc* y *argv*, para que el programa en ejecución pueda leer los argumentos con que había sido invocado.

Sin embargo, existe un tercer argumento (que si no se va a usar obviamente puede ignorarse) cual es *envp* o como se lo conoce con el nombre de *puntero al ambiente*.

Puede ver una rápida presentación del ambiente de un programa en Unix en **Unix/Linux Environement**.

El código del programa lista el ambiente, barriendo el arreglo *envp* mediante un índice y mostrándolo en la salida *standard* mediante un *for* deteniéndose cuando el índice direcciona un puntero nulo.

1.4.2. Ejecución:

La ejecución arroja el listado del ambiente; generalmente, este listado es bastante largo así que se aconseja leerlo con el comando *less* como en

```
$ ./33-ambient | less
```

Existe un comando *env* del *shell* (ejecutado simplemente como **$ env**) que justamente debería dar el mismo listado; por favor, verifique que tanto la salida del programa así como la salida de *env* coinciden (y si no coinciden, explique porqué).

1.5. 34-locality.c

1.5.1. Código:

El objeto de este programa es determinar en qué zonas de memoria virtual informa el sistema operativo están cargadas cada una de las variables del programa.

Para poder ver una explicación resumida del tema, se sugiere consultar **Memory Layout of C programs**.

Para poder entender el tema de las variables automáticas, los argumentos de pasaje a llamadas a funciones y las direcciones de retorno de funciones, se aconseja ver un simulador en el Web que está en fase experimental pero que he encontrado bastante interesante: **C tutor - Visualize C**; se pueden simular también otro tipo de programas, por ejemplo los que manejan el heap. Yo he probado con el siguiente programa y es bastante claro de ver el seguimiento de la evolución del *stack*:

```
long process( int c1, long l1 )
{
  return c1 * l1;
}

int main(int argc)
{
  int c = 2;
  long l = 30L;
  long result;

  result = process( c, l );
  return 0;
}
```

1.5.2. Ejecución:

Básicamente, el programa es muy sencillo: declara distintos tipos de variables: globales fuera de la función y *static* dentro de la función *main*, no inicializadas e inicializadas, así como constantes y variables automáticas.

Se guarda en la variable *pid* la identificación de proceso y se listan las direcciones de dónde se encuentran todas las variables así como los argumentos *argc* y *argv*.

Posteriormente, se bloquea el proceso mediante una llamada a *sleep* donde la cantidad de segundos es dependiente de si no hay argumentos adicionales, con lo cual el tiempo de bloqueo en *sleep* es de 2 segundos o, sino, es del tiempo que se declare en el argumento opcional de la invocación del programa; terminado el tiempo de bloqueo, se completa el proceso.

La idea es ejecutar el programa en condiciones que no exista ASLR, para lo cual se deben tomar las siguientes medidas:

- En MacOS nada, pues en el makefile ya está conisderado que *todos* los programas generados con ese makefile no consideren *ASLR*.
- En el caso de Linux, previamente debe invocarse el *script* **linux_no_aslr**,

lo cual cargará un nuevo *shell* por encima del existente de manera de no permitir el cambio aleatorio de las direcciones virtuales de carga del proceso.

La forma de hacer la evaluación de este programa es correrlo en dos instancias:

- La primera es sin argumentos y redireccionando la salida *standard* a un archivo de texto que se guardará en el directorio existente *results-34* que está debajo del directorio de trabajo: úsese para ello el siguiente comando, de acuerdo a la máquina donde trabaja:
  - En MacOS: **$ ./34-locality > results-34/mac-output**
  - En Linux: **$ ./34-locality > results-34/lx-output**
- La segunda es correr el programa en *background* con un argumento que indique un tiempo de bloqueo grande mediante el comando **$ ./34-locality 60&**: como el *shell* queda liberado para otro comando, coloque inmediatamente el siguiente (reemplazando *pid* por el que ve en la salida de la ejecución del programa) y antes que expire el tiempo de bloqueo del proceso, ejecutar.
  - En MacOS **$ vmmap *pid* > results-34/mac-dmem**
  - En Linux **$ sudo pmap *pid* > results-34/lx-dmem**

Nota: usando el manual en línea, investigue el comando *vmmap* o *pmap* (según su caso) para comprender este último comando y qué resultados arroja.

Terminado el proceso original, ubíquese en el directorio *results-34*, compare las direcciones que aparecen en el archivo **-output* con las direcciones de las áreas de memoria virtual del proceso cargado que se encuentran en **-dmem*, de manera de entender dónde se encuentra cada variable y porqué.

Tal cual Ud. obtiene del repositorio los archivos, el directorio *results-34* y la generación de los archivos **-output* y **-dmem* ya se encuentran realizados en una máquina Linux 18.04 o MacOS Catalina (según el caso) siguiendo las instrucciones anteriores, de manera que puede hacer el análisis inmediatamente, si no desea repetir la experiencia (aunque sería muy intersante que lo hiciese).

1.6. 35-dynamic.c

1.6.1. Código:

El objeto de este programa es mostrar el manejo de memoria dinámica usando las siguientes funciones de biblioteca:

- *malloc*
- *realloc*

- *free*

Como siempre, se aconseja ver el manual en línea **man** para ver una descripción de estas llamadas así como la siguiente página de Web **Malloc: Dynamic Memory Management in C**.

Se podrá mostrar como se realiza la asignación de memoria dinámica y se verá con una herramienta de perfilado y depurado de gestión de memoria, denominada *valgrind*.

El ejemplo que se mostrará aquí es muy sencillo, por lo cual si se quiere investigar y/o consultar más profundamente sobre *valgrind*, se sugiere ver la siguiente página de Web: **El perfilador de memoria Valgrind**.

En dicha página se muestran ejemplos sobre:

- *Lecturas y escrituras ilegales*
- *Variables sin inicializar*
- *Liberaciones ilegales de memoria*
- *Fuga de memoria*

La idea del programa es realizar una asignación de memoria original de cierto tamaño, escribir en dicha memoria un texto y luego realizar una reasignación de memoria, aumentando el tamaño de la memoria asignada y verificando que el texto permanece incólume dentro de la nueva zona de memoria.

La primera asignación de memoria se realiza mediante *malloc* en un tamaño de *INIT_SIZE* bytes y la dirección de memoria asignada se aloja en el puntero *pinit*.

Luego, se copia un texto a la memoria asignada a través de este puntero.

A continuación, y a través de la llamada a *realloc*, se solicita expandir la zona de memoria a un valor que será *NEW_SIZE1* si no hay argumentos y *NEW_SIZE2* si hay un argumento en el comando. (Esta última es *exageradamente* mayor que la primera).

El resultado de la memoria procedente de esta reasignación se resguarda en el puntero *pnew*.

Se imprimen el puntero original y el nuevo para saber si la asignación cambió la zona de memoria o mantuvo la misma.

Al hacer la reasignación deberían haberse mantenido los datos que se encontraban en la zona de memoria, para lo cual se lo imprime para determinar visualmente si han cambiado.

Se libera la zona de memoria mediante *free* utilizando el segundo puntero que es el válido.

También, y erróneamente, se libera mediante el puntero original.

Por último, se vuelve a imprimir el supuesto *buffer* que, después de la liberación, ya no pertenece más a la zona de la memoria direccionable por este proceso.

1.6.2. Ejecución:

1.6.2.1. En Linux

Para ejecutar este programa, nuevamente se solicita que estemos en condiciones de anular ASLR; si se continúa desde la ejecución del programa anterior, ya se estaba en esa situación y, sino, ejecútese *linux_no_aslr*.

Primero vamos a ejecutar el programa sin argumento como: **$ ./35-dynamic**; es muy probable que obtenga algo parecido a lo siguiente:

```
Size of buffers -> 1st: 20, 2nd: 30
Content of buffer = Hello World
pinit = 0x555555757270, pnew = 0x555555757270
Pointers pinit and pnew are same
Content of buffer = Hello World
pnew ok freed done
pinit bad freed done
Content of buffer = pruUUU
```

Se observa claramente que la zona de memoria original y la expandida se encuentran en el mismo lugar, ya que los dos punteros apuntan a la misma dirección.

Esto es posible pues probablemente, como la diferencia en este caso de tamaños es entre 20 y 30 bytes, el asignador de memoria dinámica disponía suficiente memoria para expandirla *in situ*.

También es probable, como en este caso, que algo incorrecto está ocurriendo hacia el fin del programa luego de la segunda liberación falsa con *free* ya que la impresión del *buffer* puede dar un contenido sin sentido, como en este caso.

Sin embargo, podría dar un contenido correcto pues a pesar de no estar reservada esa zona de memoria para el proceso, podría haber sido o no reescrita; de aqui que sea necesario una herramienta para hacer el análisis del proceso cuando se usa memoria dinámica: *es común que un programa que tiene un problema como éste funcione aparentemente bien la mayor parte de las veces.*

Ahora vamos a realizar la segunda experiencia: invoque la ejecución mediante **$ ./35-dynamic X**; muy probablemente se obtenga algo así:

```
Size of buffers -> 1st: 20, 2nd: 1048576
Content of buffer = Hello World
pinit = 0x555555756670, pnew = 0x7ffff7ec9010
Pointers pinit and pnew are different
Content of buffer = Hello World
```

```
pnew ok freed done
pinit bad freed done
Segmentation fault (core dumped)
```

Varias cosas se observan: ahora el tamaño de los requerimientos de memoria inicial y siguiente son *muy disímiles*. En efecto, se pasa de 20 bytes a 1MByte, con lo cual es probable que el gerenciador de memoria dinámica deba cambiar de área de memoria, lo cual se evidencia claramente por las distintas direcciones que se imprimen.

No obstante, *realloc* trabajó correctamente pues, a pesar de cambiar de área de memoria, los datos que se habían escrito en la primer área fueron trasladados correctamente a la segunda.

El cambio más importante lo da el último texto que aparece *Segmentation fault (core dumped)*: este error está lanzado por el sistema operativo que ha encontrado que en el último *printf* del programa intentó direccionar una memoria virtual que no pertenecía al proceso ofensivo. **Nota**: sería muy interesante razonar porque dió este error en este segundo caso y no en el primero.

Al ver que se intentó un procedimiento prohibido, el sistema operativo lo detuvo *intempestivamente*, pues lo consideró un proceso ofensivo para la integridad de los demás procesos y lo retiró de la tabla de procesos y de memoria.

Una **nota importante**: el listado de salida anterior *no se obtuvo por redireccionamiento de la salida standard del proceso* sino copiando la salida que quedó en la terminal; si tiene dudas, inténtelo Ud. corriendo el programa nuevamente y redireccionando la salida *standard* a un archivo como en:

```
$ ./35-dynamic X > out
```

Verifique, posteriormente, cual es el tamaño del archivo mediante el comando **ls -l out**: verá que el archivo posee un tamaño nulo; **trate de explicar que pasó**.

Por lo tanto, y para asegurarnos que la salida es correcta, corra ambas instancias del programa y copie la salida que ve en consola (*mouse* mediante) en el directorio **results-35** sobre archivos de nombres, respectivamente, *out-1* y *out-2*.

En estas condiciones, ya puede comparar el contenido de los dos archivos *out-1* y *out-2* mediante herramientas como los comandos **diff** o **vimdiff** o cualquiera que Ud. considere adecuado y obtener conclusiones.

Para ver ahora exactamente qué errores nos arroja el perfilador *valgrind* vamos a repetir la experiencia de la siguiente manera: se van a correr ambas instancias pero desde *valgrind* como se indica en cada uno de los siguientes comandos:

```
$ valgrind -v ./35-dynamic >/dev/null 2>results-35/valg-1
$ valgrind -v ./35-dynamic X >/dev/null 2>results-35/valg-2
```

Así se obtendrán los siguientes archivos sobre el directorio **results-35** mediante el comando **$ ls -l results-35**:

```
total 48
-rw-r--r-- 1 tedmar tedmar   247 abr 30 10:19 out-1
-rw-r--r-- 1 tedmar tedmar   263 abr 30 10:22 out-2
-rw-r--r-- 1 tedmar tedmar 16749 abr 30 10:25 valg-1
-rw-r--r-- 1 tedmar tedmar 16811 abr 30 10:26 valg-2
```

Es el momento, entonces, que basado en las sencillas explicaciones de la página de Web sobre *valgrind* que anteriormente se proveyó, obtenga conclusiones de los resultados obtenidos estudiando simultáneamente el programa fuente *35-dynamic.c*, la salida del programa (*out-?*) y lo listado por *valgrind* (*valg-?*) en cada uno de ambos casos.

- **Nota**: obsérvese que se ha agregado en la definición de **CFLAGS** del archivo *makefile* **-g** para solicitar al compilador que arrastre información del programa fuente, como números de línea, nombres de variables y de funciones a los fines que la salida de *valgrind* pueda referirse al programa fuente.
- **Otra**: Tal cual Ud. obtiene del repositorio los archivos, el directorio *results-35* y la generación de los archivos *out-1*, *out-2*, *valg-1* y *valg-2* ya se encuentran realizados en una máquina Linux 18.04 siguiendo las instrucciones anteriores, de manera que puede hacer el análisis inmediatamente, si no desea repetir la experiencia (aunque sería muy intersante que lo hiciese).

1.6.2.2. En MacOS

Las instrucciones anteriores son las mismas, aunque los resultados pueden ser no fundamentalmente diferentes; por lo tanto, se recomienda realizar básicamente la misma ejercitación, cuidando las pocas diferencias.

Como se indicó en el caso de *34-locality.c*, nada hay que hacer para desconectar ASLR.