Manual Rápido de Valgrind
CC31A

Profesor: José Miguel Piquer
Auxiliares: C. Hurtado, S. Kreft, P. Valenzuela

21 de agosto de 2006

1. Introducción

Valgrind es una suite de aplicaciones que permiten depurar(debug) y perfilar(profile) programas de Linux por medio de simulación. Las principales herramientas son:

- Memcheck
  Detecta problemas de manejo de memoria
- Cachegrind
  Realiza un profile del caché utilizado, realizando una simulación detallada de los caché L1,D1 y L2
- Helgrind
  Encuentra data-races en programas multithreaded(ver CC41B)

Valgrind se puede descargar desde el sitio http://valgrind.org/. Si se usa Debian, Ubuntu o alguna distribución similar se puede obtener por medio de apt-get con el comando

```
apt-get install valgrind
```

2. Memcheck

Memcheck es una herramienta que verifica el manejo de memoria. Verificando todas las lecturas y escrituras, así como también las llamadas a malloc(new), free(delete). Los errores que puede detectar son:

- Uso de memoria no inicializada
- Lectura o escritura de memoria liberada con free
- Lectura o escritura fuera del área pedida con malloc
- Lectura o escritura inapropiada de la pila
- Memory leaks(fugas de memoria), cuando se pierden los punteros a un área de memoria pedida con malloc
- Correspondencia entre malloc y free
- Traslape de memoria de origen y destino en *memcpy* y funciones relacionadas

2.1. Advertencia

Una observación ha realizar antes de utilizar el programa es que *Memcheck* realiza sus pruebas por medio de la simulación, por lo que es posible que no encuentre algunos errores de manejo de memoria. Por lo que se recomienda que cuando hagan algún módulo, creen aparte un programa tester que haga un uso intensivo de dicho módulo para minimizar las posibilidades de errores en el manejo de memoria.

2.2. Opciones de Uso

Memcheck es la herramienta por defecto de Valgrind, pero si se desea se puede especificar la opción **- -tool=memcheck**.
Para ejecutar *memcheck* sobre el programa que comúnmente se ejecuta como

```
./test arg1 arg2 ... argn
```

se hace de la siguiente manera

```
valgrind [opciones] ./test arg1 arg2 ... argn
```

Las principales opciones de uso de memcheck son las siguientes:

- **- -leak-check=<no|summary|yes|full> [default: summary]**
  Cuando está activada busca memory leaks. Si es summary se indica el número de memory leaks, y si es yes o full se entrega información detallada de cada fuga.
- **- -leak-resolution=<low|med|high> [default: low]**
  Muestra información detallada de las fugas(La opción por defecto es la recomendada)
- **- -num-callers=<number> [default: 4]**
  Muestra información detallada de las fugas(La opción por defecto es la recomendada)
- **- -freelist-vol=<number> [default: 5000000]**
  Cuando un programa libera memoria usando free, no se libera automáticamente, sino que se marca como inaccesible y se agrega a una cola de bloques libres. El propósito de esta cola es retardar lo más posible el momento en que los bloques vuelven a recircular. Esta opción indica el tamaño máximo de la cola en bytes. El incrementar el valor incrementa la memoria usada por *memcheck*, pero puede detectar manejos de memoria inválidos que de otra manera no se podría.
- **- -undef-value-errors=<yes|no> [default: yes]**
  Indica si se verifican las variables no inicializadas.No se recomienda deshabilitar esta opción.

2.3. Errores Reportados

2.3.1. Lectura/Escritura no permitida

Con el siguiente código, ocurre un error de lectura permitida, ya que no tenemos permiso para escribir en la porción de memoria 0xFF.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
  int *a=(int*)0xFF;
  *a=1;
  return EXIT_SUCCESS;
}
```

El error que se reporta es el siguiente

```
Invalid write of size 4
   at 0x8048352: main (in /home/skreft/cc31a/iwrite)
 Address 0xFF is not stack'd, malloc'd or (recently) free'd
```

2.3.2. Uso de variables no inicializadas

Cuando no se inicializa una porción de memoria, y luego se intenta utilizar para una comparación o como aragumento de otra función, entonces se produce un error. El siguiente código posee una variable no inicializada que se trata de escribir

```
#include <stdio.h>
#include <stdlib.h>
int main(){
  int x;
  printf("%d\n",x);
  return EXIT_SUCCESS;
}
```

y el error desplegado es

```
Use of uninitialised value of size 4
   at 0x406CFE7: (within /lib/tls/i686/cmov/libc-2.3.6.so)
   by 0x4070459: vfprintf (in /lib/tls/i686/cmov/libc-2.3.6.so)
   by 0x40766EF: printf (in /lib/tls/i686/cmov/libc-2.3.6.so)
   by 0x804838E: main (in /home/skreft/cc31a/ninit)

Conditional jump or move depends on uninitialised value(s)
   at 0x406CFEF: (within /lib/tls/i686/cmov/libc-2.3.6.so)
   by 0x4070459: vfprintf (in /lib/tls/i686/cmov/libc-2.3.6.so)
   by 0x40766EF: printf (in /lib/tls/i686/cmov/libc-2.3.6.so)
   by 0x804838E: main (in /home/skreft/cc31a/ninit)
```

2.3.3. Free inválido

Cuando se trata de liberar un área de memoria que no fue pedida con malloc, o se trata de liberar dos veces la misma zona se producen estos errores:
Caso 1:

```
#include <stdlib.h>
int main(){
  char *a=(char *)0xFF;
  free(a);
  return EXIT_SUCCESS;
}
```

y el error es

```
Invalid free() / delete / delete[]
   at 0x401BFCF: free (vg_replace_malloc.c:235)
   by 0x804838D: main (in /home/skreft/cc31a/finvalid)
 Address 0xFF is not stack'd, malloc'd or (recently) free'd
```

Caso 2:

```
#include <stdlib.h>
int main(){
  char *a=(char *)malloc(sizeof(char));
  free(a);
  free(a);
  return EXIT_SUCCESS;
}
```

en este caso el error es

```
Invalid free() / delete / delete[]
   at 0x401BFCF: free (vg_replace_malloc.c:235)
   by 0x80483D4: main (in /home/skreft/cc31a/finvalid2)
 Address 0x4163028 is 0 bytes inside a block of size 1 free'd
   at 0x401BFCF: free (vg_replace_malloc.c:235)
   by 0x80483C9: main (in /home/skreft/cc31a/finvalid2)
```

2.3.4. Parámetros con permisos inadecuados

Cuando una función recibe un buffer para leer/escribir y este no ha sido inicializado o no es escribible se genera un error. El siguiente código lo refleja

```
#include <stdlib.h>
#include <unistd.h>
int  main(){
  char *arr=(char *)malloc(10*sizeof(char));
  int *arr2=(int *)malloc(sizeof(int));
  write(1,arr,10);/*Trata de escribir en la salida estandar 10 caracteres, pero hay basura*/
  exit(arr2[0]);/*se trata de salir con un valo no inicializado*/
}
```

y el error generado es

```
Syscall param write(buf) points to uninitialised byte(s)
   at 0x4000772: (within /lib/ld-2.3.6.so)
   by 0x4047EA1: __libc_start_main (in /lib/tls/i686/cmov/libc-2.3.6.so)
 Address 0x4163028 is 0 bytes inside a block of size 10 alloc'd
   at 0x401B422: malloc (vg_replace_malloc.c:149)
   by 0x80483EF: main (in /home/skreft/cc31a/iparameter)

Syscall param exit_group(exit_code) contains uninitialised byte(s)
   at 0x4000772: (within /lib/ld-2.3.6.so)
   by 0x8048429: main (in /home/skreft/cc31a/iparameter)
```

2.3.5. Sobreposición de memoria de origen y destino

Cuando se superponen las zonas de origen y destino en funciones como *memcpy* ocurre este error(No pude reproducirlo, así que se los debo).

2.3.6. Detección de memory leaks

Cuando no se libera una porción de memoria generalemente se notifica como *Possibly Lost*, y cuando se pierde el puntero se notifica como *Definetely Lost*. El siguiente programa muestra los dos casos

```
#include <stdlib.h>
int main(){
  int *a=(int *)malloc(1024*sizeof(int));
  int *b=(int *)malloc(10*sizeof(int));
  *a=1;
  b=(int *)NULL;
  return EXIT_SUCCESS;
}
```

Y acá la salida

```
LEAK SUMMARY:
   definitely lost: 40 bytes in 1 blocks.
   possibly lost: 4,096 bytes in 1 blocks.
   still reachable: 0 bytes in 0 blocks.
   suppressed: 0 bytes in 0 blocks.
```

2.4. Observación sobre variables no inicializadas

Como se vio anteriormente, el programa entrega errores cuando detecta el uso de variables no inicializadas. Pero esto sólo se hace cuando las variables se usan para alguna comparación o parámetro de alguna función. Sin embargo, si solo se copian valores basura no hay problema. Con lo que el siguiente código no genera errores, ya que solo se copian valores no inicializados, no se usan por ejemplo en comparaciones.

```
#include <stdlib.h>
int main(){
  int i, j;
  int a[10], b[10];
  for ( i = 0; i < 10; i++ ) {
      j = a[i];
        b[i] = j;
  }
  return EXIT_SUCCESS;
}
```

La razón por la cual esto ocurre, es porque internamente las estructuras con tamaño no múltiplo del tamaño de un entero se redondean en tamaño. Así el siguiente código perfectamente válido tendría errores, por culpa de la implementación. Ya que los últimos 3 bytes contendrían basura. Sin embargo, como al copiar áreas no inicializadas de memoria no hay problema, valgrind se comporta según lo esperado.

```
#include <stdio.h>
#include <stdlib.h>
struct S { int x; char c; };
int main(){
  struct S s1, s2;
  s1.x = 42;
  s1.c = 'z';
  s2 = s1;
  printf("size:%d\n",sizeof(struct S));/*escribira 8 en pantalla*/
  return EXIT_SUCCESS;
}
```

Referencias

[1] Valgrind Manual, http://valgrind.org/docs/manual/

[2] *man valgrind*