Curso LINUX

AREA 2 : Depuración y optimización

Depuración

- Los errores de programación son inevitables
- La depuración es el proceso de localizar y eliminar los errores de los programas
- Cuando algo sale mal y no se consigue averiguar porqué, es mejor apuntar con un depurador y ver cómo falla
- Se necesita añadir los símbolos de compilación al ejecutable (opción -g)

Depurador - gdb

- Un depurador permite:
  - controlar la ejecución
  - cambiar el estado del programa subordinado
- Depuradores en UNIX
  - V7 : adb
  - System III : sdb
  - gdb (1988 ->)

gdb

- The GNU project debugger
- depurador a nivel de código fuente
- portable
- fácil de utilizar
- Depurador de línea de comandos
- Existen múltiples front-ends gráficos
  - ddd
  - insight

gdb – C / C++

- gdb comprende C y C++
- Permite emplear los mismos nombre que en el código fuente
- comprende expresiones en C
  - `*ptr->x.a[1]->q`

Ejecución de gdb

- gdb [opciones] [ejecutable [archivo-core]]

- ejecutable : archivo a ejecutar
- archivo-core : coredump generado por un programa abortado

Ejemplo core

```
/* Archivo dia6-abort.c */
#include <stdio.h>
#include <stdlib.h>

void recurse (void)
{
        static int i;
        if (++i==3)
                abort();
        else
                recurse();
}

int main ()
{
        recurse();
}
```

gdb con core

- `ulimit -c 1024`
- `gcc -g dia6-abort.c -o dia6-abort`
- `./dia6-abort`
- `Aborted (core dumped)`
- `file core`
- core: ELF 32-bit LSB core file Intel 80386, version 1 (SYSV), SVR4-style
- `gdb dia6-abort core`

gdb – core (2)

```
GNU gdb 6.4-debian
...

Core was generated by `./ch15-abort'.
Program terminated with signal 6, Aborted.

warning: Can't read pathname for load map:
Input/output error.
Reading symbols from
/lib/tls/i686/cmov/libc.so.6...done.
Loaded symbols for
/lib/tls/i686/cmov/libc.so.6
Reading symbols from /lib/ld-
linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0xffffe410 in __kernel_vsyscall ()
(gdb)
```

gdb – core (3)

```
(gdb) where
#0  0xffffe410 in __kernel_vsyscall ()
#1  0xb7dda9a1 in raise () from /lib/tls/i686/cmov/libc.so.6
#2  0xb7ddc2b9 in abort () from /lib/tls/i686/cmov/libc.so.6
#3  0x08048382 in recurse () at ch15-abort.c:8
#4  0x08048387 in recurse () at ch15-abort.c:10
#5  0x08048387 in recurse () at ch15-abort.c:10
#6  0x080483aa in main () at ch15-abort.c:15
(gdb)
```

gdb - where

- El comando `where` imprime un seguimiento de la pila
- Lista de todas las funciones llamadas, empezando por la más reciente
- El comando `bt` (backtrace) es un alias para where.
- cada invocación de función en la pila se llama marco (frame), lo que aparece con #

gdb – frame / list

```
(gdb) frame 3
#3  0x08048382 in recurse () at ch15-abort.c:8
8                       abort();
(gdb) list
3
4       void recurse (void)
5       {
6               static int i;
7               if (++i==3)
8                       abort();
9               else
10                      recurse();
11      }
12
(gdb)
```

gdb - list

- `list` muestra las líneas de código de la función invocada
- Si se pulsa intro se repite la última acción
- El editor de comandos utiliza readline (como vi o bash), pueden usarse los mismos comandos.

gdb – puntos de interrupción

- Se puede definir un punto de interrupción por:
  - nombre de función
  - número de línea de código
  - archivo y numero de línea
  - etc.
- El comando `run` inicia el programa
- `break` fija el punto de ruptura

gdb - breakpoints

- `break` *`expresion`* : crea un punto de ruptura
- `b file:n` : crea un punto de ruptura coincidiendo con la línea n del alchivo file
- `clear` *`expresion`* : borra los breakpoints de esa expresion
- `delete` : borra todos los breakpoints
- `delete` *`rango`* : borra un rango de breakpoints

Ejemplo

```
gdb dia6-abort
GNU gdb 6.4-debian
...

(gdb) b recurse
Breakpoint 1 at 0x8048366: file dia6-abort.c, line 7.
(gdb) run
Starting program: /home/espinosa/curso/debug/dia6-abort

Breakpoint 1, recurse () at dia6-abort.c:7
7                if (++i==3)
(gdb)
```

Breakpoint condicionales

- Se pueden poner condiciones a los breakpoints:
  - `b hello if i==7`
- Se pueden cambiar las condiciones de los breakpoints con condition
  - `condition 1 i==8`
- `condition n` : elimina la condicion del breakpoint n

gdb – next / step

- **next** ejecuta la siguiente instrucción
- **step** ejecuta la siguiente invocación (entra en la función)

```
(gdb) next
10                      recurse();
(gdb) next

Breakpoint 1, recurse () at dia6-abort.c:7
7               if (++i==3)
(gdb)
```

gdb – cont / quit

- `cont` – continua con la ejecución del programa hasta el siguiente punto de ruptura o el final del mismo.
- quit – abandona la depuración del programa

```
(gdb) continue
Continuing.

Breakpoint 1, recurse () at dia6-abort.c:7
7                 if (++i==3)
(gdb) quit
The program is running.  Exit anyway? (y or n)
```

continue / return

- `continue` puede recibir como parámetro un número que indica el numero de breakpoints a ignorar antes de volver a parar
- `return` : continua la ejecución fuera de la función en la que nos encontremos
- `jump linea` : continua la ejecución en otra línea de código

gdb - watch

- Un punto de inspección es como uno de interrupción pero para datos
- `watch` nombre_de_variable
- La variable se revisa y cada vez que cambia el valor se para el programa
- En cualquier momento se puede interrumpir la ejecución con ctrl-c

gdb - watch

```
(gdb) watch recurse::i
Hardware watchpoint 1: recurse::i
(gdb) run
Starting program: dia6-abort
Hardware watchpoint 1: recurse::i
Hardware watchpoint 1: recurse::i
Hardware watchpoint 1: recurse::i
Hardware watchpoint 1: recurse::i

Old value = 0
New value = 1
0x08048373 in recurse () at dia6-abort.c:7
7             if (++i==3)
```

```
(gdb) c
Continuing.
Hardware watchpoint 1: recurse::i

Old value = 1
New value = 2
0x08048373 in recurse () at dia6-abort.c:7
7             if (++i==3)
(gdb) c
Continuing.
Hardware watchpoint 1: recurse::i

Old value = 2
New value = 3
0x08048373 in recurse () at dia6-abort.c:7
7             if (++i==3)
```

gdb - attach

- gdb permite engancharse a un proceso ya corriendo para ver qué está haciendo
- Si ese programa no dispone de los simbolos de depuración, simplemente veremos las llamadas a funciones
- `attach pid` intenta enlazarse con el proceso en ejecución con ese pid

ejemplo - attach

- Lanzar el servidor corba del dia 4 en background
- `ps axuw | grep servidor`
- ejecutar `gdb servidor`
- `attach pid`
- gdb nos permite informarnos sobre los threads
- `info threads`

Threads

- Notificación automática de threads
- info threads : información sobre los threads del programa
- thread n : cambia el contexto de depuración a ese thread
- thread apply : aplica una serie de comandos a un thread
- Se pueden fijar breakpoint por thread `break spec thread n`

Ejemplo

- Recompilar el cliente y servidor de corba con la opción -g
- ```
  g++ -g eg2_impl.cc echoSK.cc -o servidor_debug
  -lomnithread -lomniORB4 -I.
  ```
- Lanzar el servidor en background
- Lanzar el gdb contra ese proceso y poner un breakpoint en Echo_i::echoString
- Lanzar el cliente en otra ventana
  ```

Comandos gdb

- print : imprime una expresión
- shell comando : ejecuta un comando en la shell
- make argumentos : ejecuta un make
- info break : información sobre breakpoints
- help : ayuda
- show : muestra estado gdb

Depuración remota

- Desde la versión 5.3 existe la depuración remota en gdb
- Hace uso de gdbserver
- Es posible ejecutar el código sin los símbolos de depuración mientras se disponga de ellos en local

Depuración remota (2)

- Compilar `gdb` para la plataforma elegida
- Compilar `gdbserver` (bajo `gdb/gdbserver`) con el compilador cruzado
- Transferir `gdbserver` al sistema elegido
- Ejecutar `gdbserver foo:1234 programa`
- En el sistema de desarrollo ejecutar `gdb programa`

Depuración remota (3)

- Dentro del gdb
  - `target remote foo:1234`
- A partir de ese momento ya se ejecutan los comandos de depuración en remoto.

Ejemplo

- Depurar, dos a dos, en remoto el servidor corba

Técnicas de depuración

- Código de depuración en tiempo de compilación

```
#ifdef DEBUG
   fprintf (stderr,"myvar = %d\n",myvar);
   fflush(stderr);
#endif
```

- Añadir -DDEBUG a la línea de compilación provoca que se active la depuración

Consejos

- Enviar los mensajes a stderr
- Hacer flush después de cada mensaje
- Utilizar un simbolo propio, mejor que DEBUG, incluso múltiples símbolos por cada zona de depuración
- Crear macros para evitar los ifdef

```
#ifdef DEBUG

   #define DPRINT(msg) fprintf(stderr,msg)

#else

   define DPRINT(msg)
```

Opciones macros

- `#define DPRINT(stuff) fprintf stuff`
- requiere cambiar la llamada
- `#define DPRINT(msg,...) fprintf(stderr,mesg,__VA_ARGS__)`
- Solo compatible compiladores C99

Consejos

- Evitar las macros de expresiones
- Reordenar el código para mejorar la visibilidad en depuración
- Emplear enumeraciones en lugar de macros
- Crear funciones auxiliares de depuración
- Evitar las uniones cuando sea posible

Herramientas de depuración

- librería dbug
- Depuradores de asignación de memoria
  - mtrace
  - Electric Fence
  - dmalloc
  - Valgrind
  - Otros depuradores

El problema

- Pérdida de memoria: memoria asignada e imposible de alcanzar
- Memoria sin liberar
- Liberaciones incorrectas
- Uso de memoria ya liberada
- Saturación de memoria
- Uso de memoria sin inicializar

mtrace

- Funciona en sistemas Linux con glibc
- incorpora dos funciones para habilitar y deshabilitar el seguimiento de memoria
- #include <mcheck.h>
- void mtrace(void);
- void muntrace(void)
- La variable de entorno MALLOC_TRACE indica un archivo donde guargar la información

Ejemplo

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char **argv)
{
        char *p;
        int i;

        p=malloc(30);
        strcpy(p,"not 30 bytes");
        printf ("p = <%s>\n",p);

        if (argc==2) {
                if (strcmp(argv[1],"-b")==0)
                        p[42]='a';
                else if (strcmp(argv[1],"-f")==0) {
                        free(p);
                        p[0]='b';
                }
        }
        /* free(p);*/
        return 0;
}
```

Ejemplo - mtrace

- Modificar el ejemplo para que funcione con mtrace
  - export MALLOC_TRACE=trace.out
  - ejecutar el programa con/sin parametros
  - mtrace badmem1 trace.out

- ¿RESULTADO?

Electric Fence

- paquete electric-fence
- Sustituye al malloc con otra versión que provoca errores en el uso inadecuado
- `gcc -g badmem1.c -lefence -o badmem1`

```
gdb ./badmem1
(gdb) run -b
[Thread debugging using libthread_db enabled]
[New Thread -1210132800 (LWP 20346)]
  Electric Fence 2.1 Copyright (C) 1987-1998 Bruce Perens.
p = <not 30 bytes>

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread -1210132800 (LWP 20346)]
0x0804850f in main (argc=2, argv=0xbf962114) at badmem1.c:14
14                              p[42]='a';
```

Electric Fence (2)

```
(gdb) run -f
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/espinosa/curso/dia6/badmem1 -f
[Thread debugging using libthread_db enabled]
[New Thread -1209940288 (LWP 20729)]

  Electric Fence 2.1 Copyright (C) 1987-1998 Bruce Perens.
p = <not 30 bytes>

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread -1209940288 (LWP 20729)]
0x0804855a in main (argc=2, argv=0xbf992d84) at badmem1.c:17
17                              p[0]='b';
```

Electric Fence ef

- Podemos utilizar el comando ef para ejecutar un programa no compilado con la librería efence
- Ejemplo:
- `gcc -g badmem1.c -o badmem1`
- `ef badmem1 -b` (con ef)
- `bdamem1 -b` (sin ef)
- Es posible que ef no venga incluido con el paquete electric-fence

dmalloc

- www.dmalloc.com
- Se puede encontrar en el paquete libdmalloc
- requiere la variable del sistema DMALLOC_OPTIONS
- Escribir a mano las opciones para DMALLOC_OPTIONS es complicado

```
$ dmalloc () {
> eval `command dmalloc -b $*`
> }
```

dmalloc (2)

- Con esta función podemos trasmitir opciones como:
  - `-l` archivo de depuracion
  - `-i` numero de iteraciones
  - nivel de depuración o etiqueta
- `dmalloc -l dm-log -i 100 low`
- Al igual que electric-fence puede enlazarse estática o dinámicamente

dmalloc - ejemplo

```
export DMALLOC_OPTIONS=debug=0x4e40503,inter=100,log=dm-log
LD_PRELOAD=libdmalloc.so.4 ./badmem1 -b
more dm-log
```

```
1203415747: 1: max unused memory space: 34 bytes (53%)
1203415747: 1: top 10 allocations:
1203415747: 1:  total-size  count in-use-size  count  source
1203415747: 1:          30      1          30      1  ra=0x80483f2
1203415747: 1:          30      1          30      1  Total of 1
1203415747: 1: Dumping Not-Freed Pointers Changed Since Start:
1203415747: 1:  not freed: '0xb7f18fc8|s1' (30 bytes) from 'ra=0x80483f2'
1203415747: 1:  total-size  count  source
1203415747: 1:          30      1  ra=0x80483f2
1203415747: 1:          30      1  Total of 1
```

dmalloc + gdb

- Podemos utilizar gdb para examinar la dirección que nos proporciona:

```
gdb ./badmem1

(gdb) x 0x80483f2
0x80483f2 <main+42>:    0x8bf04589
(gdb) info line *(0x80483f2)
Line 8 of "badmem1.c" starts at address 0x80483e6 <main+30>
   and ends at 0x80483f5 <main+45>.
(gdb)
```

- Si se incluye `dmalloc.h` en el programa se puede ver en el informe las líneas de código diréctamente

Valgrind

- Es una suite de herramientas que cubre más aspectos además de la memoria dinámica.
- Instalar paquete valgrind
- Hay varios front-ends para la aplicación (alleyou, kdeveloper, etc.)
- Se compone de una parte principal que emula x86 y una serie de máscaras que son herramientas de depuración o creación de perfiles

Valgrid - mencheck

- Se comprueban todas las lecturas y escrituras de memoria, puede detectar:
  - Uso de memoria sin inicializar
  - R/W de memoria liberada
  - R/W fuera de los bloques malloc
  - R/W areas inadecuadas de pila
  - Perdidas de memoria
  - Uso de malloc/new frente a free/delete
  - Usos incorrectos de pthreads

Valgrid – otras máscaras

- `cachegrind` : simulación de caches de la CPU
- `addrcheck` : como memcheck pero sin chequeos de valores inciales. Ejecución más rápida
- `heldgrind` : accesos concurrentes a memoria

Valgrind - ejemplo

- valgrind badmem1 -b

```
LEAK SUMMARY:
   definitely lost: 30 bytes in 1 blocks.
     possibly lost: 0 bytes in 0 blocks.
   still reachable: 0 bytes in 0 blocks.
        suppressed: 0 bytes in 0 blocks.
```

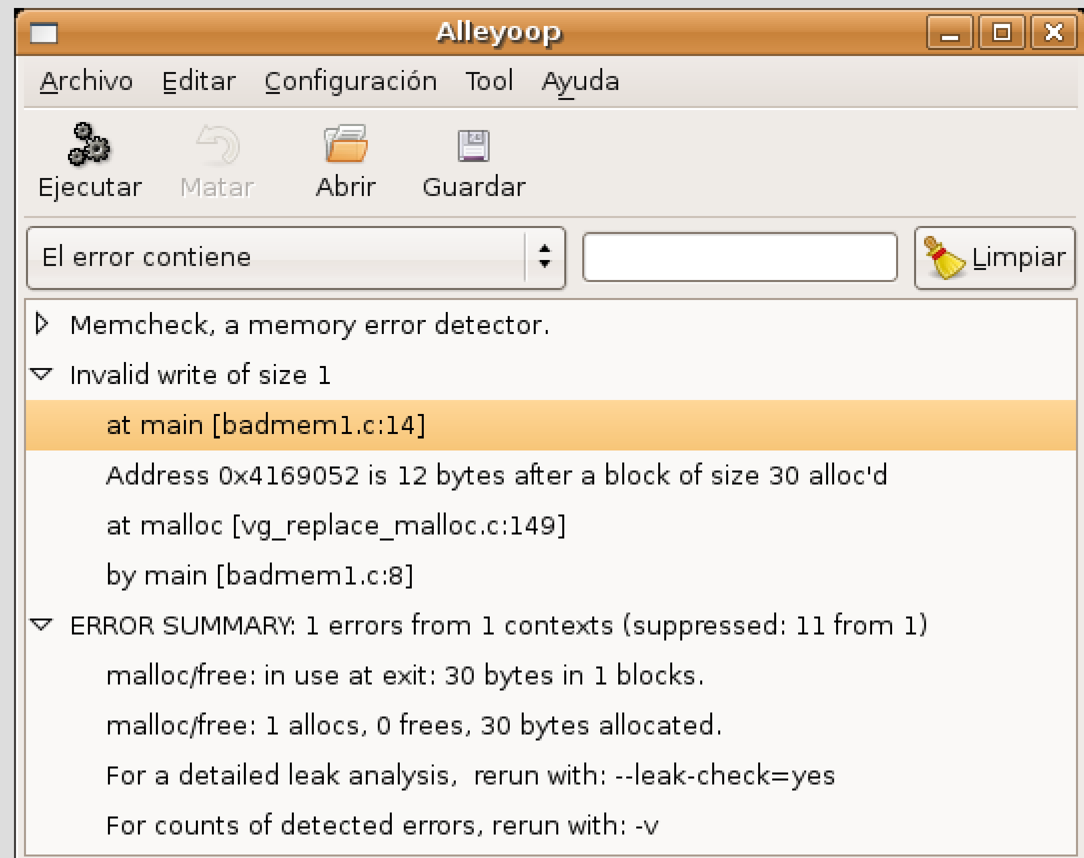
```
Memcheck, a memory error detector.

p = <not 30 bytes>
 Invalid write of size 1
    at 0x804846B: main (badmem1.c:14)
  Address 0x4169052 is 12 bytes after a block of size 30 alloc'd
    at 0x401C422: malloc (vg_replace_malloc.c:149)
    by 0x80483F1: main (badmem1.c:8)

 ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 11 from 1)
 malloc/free: in use at exit: 30 bytes in 1 blocks.
 malloc/free: 1 allocs, 0 frees, 30 bytes allocated.
 For counts of detected errors, rerun with: -v
 searching for pointers to 1 not-freed blocks.
 checked 80,228 bytes.
```

Alleyoo

- Front-end para valegrind



Valgrind

- Uso con badmem -f (escritura incorrecta)

p = <not 30 bytes>
==28291== Invalid write of size 1
==28291==   at 0x80484B6: main (badmem1.c:17)
==28291== Address 0x4169028 is 0 bytes inside a block of size 30 free'd
==28291==   at 0x401CFCF: free (vg_replace_malloc.c:235)
==28291==   by 0x80484B2: main (badmem1.c:16)
==28291==
==28291== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 11 from 1)
==28291== malloc/free: in use at exit: 0 bytes in 0 blocks.
==28291== malloc/free: 1 allocs, 1 frees, 30 bytes allocated.
==28291== For counts of detected errors, rerun with: -v
==28291== No malloc'd blocks -- no leaks are possible.

leak-check

- valgrind --leak-check=yes ./badmem1

```
==28471== 30 bytes in 1 blocks are definitely lost in loss record 1 of 1
==28471==    at 0x401C422: malloc (vg_replace_malloc.c:149)
==28471==    by 0x80483F1: main (badmem1.c:8)
==28471==
==28471== LEAK SUMMARY:
==28471==    definitely lost: 30 bytes in 1 blocks.
==28471==      possibly lost: 0 bytes in 0 blocks.
==28471==    still reachable: 0 bytes in 0 blocks.
==28471==         suppressed: 0 bytes in 0 blocks.
```

Ejemplo

- Crear un programa que utilice variables sin incializar
- Ejecutar valgrind

Valgrind - conclusión

- Herramienta potente
- Se ha utilizado a gran escala
  - KDE
  - OpenOffice
  - Konqueror
- Rivaliza con otros productos de pago
- Se puede utilizar junto con wine para depurar programas en VC++

Otros depuradores

- ccmalloc
- malloc de Mark Moraes
- mpatrol
- memwatch
- njamd : Not Just Another Malloc Debugger
- yamd : como EF pero más completo

Conclusiones

- Hay muchas herramientas valiosas para detectar problemas de uso de memoria
- Depende de las dependencias que tengamos elegiremos unas u otras
- Es posible utilizar varias herramientas en conjunto
- En conjunción con gdb dan un control completo sobre el rendimiento del programa.

Referencias

- Programación en Linux, casos prácticos (Arnold Robbins) Anaya Multimedia
- www.kegel.com/linux/gdbserver.txt
- www.delorie.com/gnu/docs/gdb/
- valgrind.kde.org
- www.linuxjournal.com/article.php?sid=6556
- www.linuxjournal.com/article.php?sid=6059
-