Introducción a Sistemas Operativos: Procesos

Clips xxx
Francisco J Ballesteros

1. Cuando las cosas se tuercen...

De vez en cuando un proceso ejecutará un programa que tendrá un error y no hará lo que esperamos o hará algo demasiado grave que casuse que UNIX lo mate (por ejemplo, accediendo a una dirección de memoria a la que no tiene permiso para acceder por estar fuera de un segmento o por ser una escritura de un segmento de sólo lectura).

En la mayoría de los errores, incluir algún `printf` para depurar tras pensar en qué puede estar fallando es la mejor herramienta. En otros casos, necesitaremos ayuda para ver qué sucede. Para eso podemos utilizar un depurador. Cuando un programa hace algo que cause que UNIX lo mate, UNIX permite que se haga un volcado del estado de la memoria del proceso a un fichero (y del valor de los registros en el momento de la muerte). Dicho fichero suele llamarse `core` dado que es un volcado de la memoria y, cuando se hizo UNIX, la memoria podía ser de núcleos (o "cores") de ferrita!). Según una de las personas que hizo UNIX, ¡las cosas empezaron a ir mal cuando los bits dejaron de verse a simple vista!

Vamos a hacer un programa que se comporte mal.

```
#include <stdlib.h>

static void
clearstr(char *p)
{
    p[0] = 0;
}

int
main(int argc, char* argv[])
{
    char *p;

    p = NULL;

    clearstr(p);
    // ... do other things here...
    exit(0);
}
```

Y vamos a ejecutarlo:

```
unix$ bad
Segmentation fault: 11
unix$
```

El programa ha intentado utilizar una dirección de memoria que no tiene y la causado una violación de segmento. Cuando pasa esto, el hardware eleva una excepción al intentar traducir la dirección de memoria y el

manejador de la excepción (el kernel) ve que el proceso no debería hacer eso. Como resultado, el kernel termina la ejecución del proceso. En nuestro caso, el shell se ha dado cuenta de lo que ha sucedido y ha impreso un mensaje para informar de ello.

Para conseguir un `core`, hemos de cambiar uno de los límites que tiene el proceso. Concretamente, el límite que indica el tamaño máximo de `core` que queremos obtener. Una vez más, los límites son también atributos del proceso. Desde C puedes usar *setrlimit(2)* para cambiar un límite y *getrlimit(2)* para consultar el valor actual. Desde el shell podemos ver los límites que tenemos con el comando *ulimit(1)*:

```
unix$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
file size               (blocks, -f) unlimited
max locked memory       (kbytes, -l) unlimited
max memory size         (kbytes, -m) unlimited
open files                      (-n) 256
pipe size            (512 bytes, -p) 1
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 709
virtual memory          (kbytes, -v) unlimited
```

Y cambiar el tamaño máximo de core como sigue:

```
unix$ ulimit -c unlimited
unix$
```

Ahora podemos ejecutar nuestro programa roto una vez más:

```
unix$ bad
Segmentation fault: 11 (core dumped)
unix$
```

El lugar en que UNIX guarda los ficheros `core` varía de un UNIX a otro. Hace tiempo era el directorio actual del proceso que muere. Hoy día depende mucho del tipo de UNIX. Por ejemplo, en MacOS (OS X), el manual dice...

```
CORE(5)                   BSD File Formats Manual                    CORE(5)

NAME
     core -- memory image file format

SYNOPSIS
     #include <sys/param.h>

DESCRIPTION
     A small number of signals which cause abnormal termination of a process
     also cause a record of the process's in-core state to be written to disk
     for later examination by one of the available debuggers.  (See
     sigaction(2).)  This memory image is written to a file named by default
     core.pid, where pid is the process ID of the process, in the /cores
     directory, provided the terminated process had write permission in the
     directory, and the directory existed.
```

Así pues, en este sistema:

```
unix$ ls -l /cores
total 1234232
-r--------  1 nemo  admin  631926784 Aug 19 15:25 core.92367
```

Donde `92367` era el *pid* del proceso que murió.

Podemos utilizar un depurador para ver un volcado de la pila en el momento de la muerte. Esto suele ser mas que suficiente para avergüar la causa del problema:

```
unix$ lldb -c /cores/core.92367
(lldb) target create --core "/cores/core.92433"
Core file ’/cores/core.92433’ (x86_64) was loaded.
(lldb) bt
* thread #1: tid = 0x0000, 0x0000000105a53f6c
    bad‘clearstr(p=0x0000000000000000) + 12
    at bad.c:10, stop reason = signal SIGSTOP
  * frame #0: 0x0000000105a53f6c
    bad‘clearstr(p=0x0000000000000000) + 12
    at bad.c:10
    frame #1: 0x0000000105a53f57
    bad‘main(argc=1, argv=0x00007fff5a1acb20) + 39
    at bad.c:20
    frame #2: 0x00007fff94ad65ad libdyld.dylib‘start + 1
(lldb) q
unix$
```

Hemos utilizado el depurador *lldb(1)*, que tiene la opción `-c` para indicarle que queremos inspeccionar un *core dump* (un fichero `core`). *LLdb* es un shell en el que podemos escribir comandos para depurar. El comando *bt* imprime un volcado o *backtrace* de la pila. El comando *q* termina la ejecución del depurador.

Como podrás ver, el program ha muerto en la función `clearstr` del ejecutable `bad`. Concretamente en la línea 10. A dicha función la ha llamado `main` en el ejecutable `bad`, desde la línea 20 de `bad.c`. Y a `main` lo ha llamado una función llamada `start` dentro del cargador de librerías dinámicas (mira el capítulo de introducción si no recuerdas lo que es una librería dinámica).

Pues bien, la línea 10 de `bad.c` es:

```
p[0] = 0;
```

Y podemos ver que el argumento de `clearstr`, `p`, tiene como valor `0`. Luego sucede que `p` es `NULL` y no podemos atravesar dicho puntero. Ese es el problema. Ahora habría que pensar en por qué ha sucedido y en qué deberíamos hacer para arreglarlo.

Un detalle importante es que para que el depurador pueda saber qué ficheros fuente y líneas corresponden a cada contador de programa, hay que pedir al compilador que incluya información de depuración en el ejecutable (que incluya una tabla con la correspondencia de nombre de fichero fuente y línea a contador de programa, y tal vez otra información como nombres de función y variables). Habitualmente, el flag `-g` del compilador de C consigue dicho efecto. Esto es, hemos compilado como sigue:

```
unix$ cc -g -o bad bad.c
```

Si no incluimos la información de depuración, el depurador no puede hacer magia. Podemos utilizar el comando *strip(1)* que elimina la información de depuración de un ejecutable y ver el resultado:

```
unix$ strip bad
unix$ bad
Segmentation fault: 11 (core dumped)
unix$
```

Y ahora...

```
unix$ ls /cores
core.92473
unix$ lldb -c /cores/core.92473
(lldb) target create --core "/cores/core.92473"
Core file ’/cores/core.92473’ (x86_64) was loaded.
(lldb) bt
* thread #1: tid = 0x0000, 0x00000001014c4f6c
    bad‘clearstr + 12, stop reason = signal SIGSTOP
  * frame #0: 0x00000001014c4f6c
    bad‘clearstr + 12
    frame #1: 0x00000001014c4f57
    bad‘main + 39
    frame #2: 0x00007fff94ad65ad libdyld.dylib‘start + 1
```

Como verás, sólo podemos ver el contador de programa en cada llamada registrada en la pila (en cada *registro de activación* de la pila). Concretamente, el programa murió en `bad‘clearstr + 12`. Esto es, 12 posiciones más allá del comienzo de `clearstr` en el fichero ejecutable `bad`. Pero claro, no sabemos ni el fichero fuente ni la línea.

Lo más aconsejable es compilar siempre con `-g` y dejar la información de depuración en los ejecutables.

Otro depurador muy popular es `gdb`. Lo mejor es que utilizes el manual o *google* para averigüar cómo obtener un volcado de pila de un fichero `core` con tu depurador, y que localices el directorio en que tu UNIX deja los *core dumps*.