FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

# Herramientas útiles

Lucas H. Pandolfo

Sistemas distribuidos I - 2016

FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

# GDB

El **GNU Debugger** o **gdb** permite examinar un proceso mientras se ejecuta. Provee una interfaz simple por línea de comandos aunque existen interfaces mas avanzadas como la provista por **Eclipse**, **kdevelop**, **NetBeans**, **Qt Creator**, **emacs**, etc.

Las operaciones básicas incluyen la inspección del **stack trace**, las variables locales del frame activo, las variables globales y manejo de **breakpoints**. Otras funciones incluyen examinar el código fuente o el código assembly generado, ver los registros del procesador, examinar core dumps, examinar la memoria, etc.

FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

# GDB - Uso práctico

Consideremos el programa **division**:

```
 1: #include <stdio.h>
 2: #include <unistd.h>
 3:
 4: int main(int argc, char** argv){
 5:     int i, j=2, k=3, l, m=25;
 6:     for(i=4;i>=0;i--){
 7:         printf("10/%i = %i\n", i, 10/i);
 8:     }
 9:     return 0;
10: }
```

Es importante compilar el programa con **-g**, **-g3** o **-ggdb** para tener información extra a la hora de utilizar **gdb**.

# GDB - Uso práctico

Dado el comando **run** el programa programa cargado comienza su ejecución y continúa hasta que se produce un error, sea interrumpido (por ejemplo **C-c**) o termine normalmente.

```
$ gdb division
...
(gdb) run
Starting program: ejemplos/division
10/4 = 2
10/3 = 3
10/2 = 5
10/1 = 10

Program received signal SIGFPE, Arithmetic exception.
0x000000000040057b in main (argc=1, argv=0x7fffffffdf28) at division.c:7
7               printf("10/%i = %i\n", i, 10/i);
```

# GDB - Uso práctico

En este caso el proceso aborta al intentar realizar una division por cero y **gdb** muestra que el error ocurrió en la línea 7 de **division.c** junto con la expresión en cuestión. Para examinar el valor de i al momento del error se puede utilizar el comando **print**.

```
(gdb) print i
$1 = 0
```

# GDB - Breakpoints

Si se quiere ejecutar un proceso hasta cierto punto se puede utilizar lo que se conoce como **breakpoints**. Cada vez que un proceso pasa por un **breakpoint** su ejecución es interrumpida.

```
(gdb) break division.c:7
Breakpoint 1 at 0x40058a: file division.c, line 7.
```

Los **breakpoints** se pueden establecer especificando **archivo:linea**, el nombre de una función (por ejemplo **main**, **funcion_2** o **Clase::metodo(tipo1,tipo2, …)**) o una dirección de memoria.

# GDB - Breakpoints

El comando **info breakpoints** muestra una lista con los breakpoints activos. Cada vez que se agrega un **breakpoint** se le asigna un número de órden (se puede ver al consultar los **breakpoints**). Utilizando ese número se pueden habilitar y deshabilitar individualmente utilizando con el comando **disable** o ignorarlo una determinada cantidad de veces con **ignore <numero-de-orden> <veces-a-ignorar>**.

# GDB - Examinar el código

El comando **list** muestra el código fuente y **disassemble** muestra el assembly.

```
(gdb) list
2       #include <unistd.h>
3
4       int main(int argc, char** argv){
5           int i, j=2, k=3, l, m=25;
6           for(i=4;i>=0;i--){
7               printf("10/%i = %i\n", i, 10/i);
8           }
9           return 0;
10      }
```

# GDB - Control de flujo

- **next**: *Step over,* ejecuta una línea del código. Si es una función devuelve el control luego de ejecutar la llamada
- **step**: *Step into,* ejecuta la próxima línea de código. Si es una función, entra a la llamada y devuelve el control dentro de la misma
- **return**: fuerza el retorno de la función actual. Opcionalmente toma como parámetro el valor que debe retornar la función
- **continue**: continúa la ejecución hasta que el proceso termine o se encuentre con algún breakpoint

# GDB - Watchpoints

Los watchpoints se utilizan para interrumpir la ejecución del proceso cuando se detecta un cambio en una **expresión**, sin importar dónde se originó el cambio. Se pueden pensar como **breakpoints** pero sobre datos en vez de código. Se pueden utilizar en tres formatos: de lectura (**rwatch**), de escritura (**watch**) y de lectura/escritura (**awatch**).

# GDB - Watchpoints

```
(gdb) watch i
Hardware watchpoint 2: i
(gdb) c
Continuing.
Hardware watchpoint 2: i

Old value = 0
New value = 4
0x0000000000400588 in main (argc=1, argv=0x7fffffffdec8) at division.c:6
6           for(i=4;i>=0;i--){
(gdb) c
Continuing.
10/4 = 2
Hardware watchpoint 2: i

Old value = 4
New value = 3
0x00000000004005ad in main (argc=1, argv=0x7fffffffdec8) at division.c:6
6           for(i=4;i>=0;i--){
```

# GDB - Watchpoints

Además de poder utilizarse nombres de variables para especificar el watchpoint, también se pueden especificar direcciones de memoria (util por ejemplo para poner **watchpoints** en espacio de **shared memory**).

```
(gdb) p &i
$1 = (int *) 0x7fffffffdddc
(gdb) watch *0x7fffffffdddc
Hardware watchpoint 2: *0x7fffffffdddc
(gdb) c
Continuing.
Hardware watchpoint 2: *0x7fffffffdddc

Old value = 0
New value = 4
0x0000000000400588 in main (argc=1, argv=0x7fffffffdec8) at division.c:6
6           for(i=4;i>=0;i--){
```

# GDB - Backtrace y frames

El comando **backtrace** muestra la lista de **frames** que estan en uso en el proceso actual. Con el comando **frame numero** se puede saltar al frame especificado para examinarlo (por ejemplo con **info locals**). El análisis de backtraces es muy útil cuando se requiere encontrar errores a partir de core dumps.

# GDB - Interfaz extendida

Utilizando la combinación de teclas **C-x a** (**control-x** y a continuación **a**) se divide la pantalla en dos y aw muestra en la parte superior el código fuente y debajo la consola de comandos de **gdb**. Para cambiar el foco entre la parte superior a la inferior se utiliza la cmbinación de teclas **C-x o**. **C-x 2** cambia la vista a otra que muestra el assembly junto con el codigo fuente.

También se puede lograr el mismo resultado con los comandos **layout**, **layout src**, **layout asm**, etc.

**NOTA:** Si por producto de alguna salida por pantalla del programa se corrompe la pantalla, generalmente puede arreglarse con **C-l** o **refresh**.

# GDB - Comandos mas utilizados

- **set args**: Sirve para especificar los argumentos del programa a debuggear.
- **start** (**r**): Inicia (o reinicia) el proceso y queda a la espera en el punto de entrada (**main**).
- **run**: Corre el programa desde el inicio.
- **step** (**s**), **next** (**n**), **continue** (**c**)
- **break** (**b**)
- **backtrace** (**bt**)

# GDB - Comandos mas utilizados

Hay que tener en cuenta que **gdb** no requiere el nombre completo de los comandos. Siempre y cuando el comando no se confunda con otro su nombre puede ser abreviado (por ejemplo **cont** en vez de **continue**, **backt** en vez de **backtrace**, etc). Además existen alias para los comandos mas comúnes (como **c** para **continue** o **bt** para **backtrace**).

# GDB - Otros comandos

En el caso de **step** y **next**, muchas veces es necesario repetir la operación varias veces. Si se presiona **enter** sin ingresar ningún comando **gdb** repite el comando anterior.

El comando **thread** muestra el hilo de ejecución actual y permite cambiar de hilo al especificar un identificador. Para listar los identificadores se utiliza **info threads**.

También existen comandos para especificar que comportamiento se debe adoptar en los **fork**, comandos para debuggear procesos que estan en ejecución (sin haber sido abiertos desde **gdb**) y muchos mas.

FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

# GDB - Print

El comando **print** (**p**) muestra por pantalla el valor de una expresión. La expresión puede ser una variable del programa (por ejemplo **print i**), una variable de **gdb** (**print $pc**) o incluso llamadas a funciones, casteos, indirecicones y modificacion de variables:

```
Temporary breakpoint 3, main (argc=1, argv=0x7fffffffdec8) at division.c:5
5           int i, j=2, k=3, l, m=25;
(gdb) n
6           for(i=4;i>=0;i--){
(gdb) p i+j+k
$7 = 5
(gdb) p $pc
$8 = (void (*)()) 0x400581 <main+36>
(gdb) p perror("Sin errores")
Sin errores: Success
$9 = 0
(gdb) p m = 64
$10 = 64
(gdb) p (char*)&m
$11 = 0x7fffffffddd0 "@"
```

# GDB - Procesos hijos

**gdb** permite especificar un comportamiento a la hora de ejecutar un fork con el comando **set follow-fork-mode** que toma como parametro **parent** (para que **gdb** siga al padre) o **child** (para que siga al proceso hijo). En cualquiera de estos casos el otro proceso queda libre de seguir su camino. Si se quiere evitar esto se puede utilizar el comando **set detach-on-fork off**, en cuyo caso el otro proceso queda en modo de espera bajo el control de **gdb**. En este último caso **info inferiors** muestra los procesos en espera e **inferior <identificador>** permite pasar a controlar otro proceso.

Si se quiere que un proceso **inferior** siga su camino normalmente se debe utilizar el comando **detach inferiors**.

# GDB - Attach / Detach

Otra de las funcionalidades de **gdb** es adherirse a un proceso que está corriendo para examinarlo. Para ello se requiere conocer el **pid** de dicho proceso. El comando **attach pid** hace que **gdb** abandone el proceso que estaba debuggeando (si habia alguno) y que tome el control del proceso cuyo **pid** fue especificado (si existe).

Otra forma es especificar por línea de comandos un **pid** despues del binario (por ejemplo **gdb proceso1 33256**).

Para abandonar el proceso y dejarlo seguir se utiliza el comando **detach**. Si se sale de **gdb** despues de un **attach**, automáticamente se realiza un **detach**. Si se intenta hacer un **run** el proceso recibe un **kill**.

# GDB - Ayuda

**gdb** incluye una gran cantidad de informacion de sí mismo. El comando **help** es un punto de partida para conocer más acerca de alguna función. Por ejemplo **help break** muestra la descripción del comando **break** y explica las diferentes posibilidades.

También existe el comando **apropos**, que busca por palabra clave. Por ejemplo **apropos child** muestra todos los comandos **gdb** relacionados con los procesos hijos.

Adicionalmente **gdb** provee una función de **autocomplete** utilizando **tab**

# strace

**Strace** es una utilidad que permite visualizar las llamadas a **system calls** y manejadores de señales que son invocados en un proceso mientras se ejecuta.

Resulta de grán utilidad para diagnosticar muchos de los errores comúnes (como lecturas/escrituras a file descriptors inválidos, flags erróneos, **exec** de binarios inválidos o con permisos insuficientes, etc).

# strace - Ejemplo

Supongamos un programa sencillo **hola** con comportamiento indefinido:

```
 1: #include <stdio.h>
 2: #include <unistd.h>
 3:
 4: int main(int argc, char** argv){
 5:     int fd = open("/tmp/dummy", "r");
 6:     char buffer[100];
 7:
 8:     read(fd, buffer, 100);
 9:     printf("%s", buffer);
10:     return 0;
11: }
```

```
$ ./hola
````

# strace - Ejemplo

Se puede ver como falla la invocación a **open** devolviendo **-1** y como falla luego el **read** con **EBADF** (se intenta utilizar el descriptor **-1**). Al final también se puede ver como la llamada a **printf** se traduce en un **write** al file descriptor **1**.

```
$ strace hola
execve("./hola", ["./hola"], [/* 90 vars */]) = 0
...
open("/tmp/dummy", O_RDONLY|O_CREAT|0x400014, 03777777777760170) = -1 EINVAL (Invalid argument
)
read(-1, 0x7fffffffdf10, 100)           = -1 EBADF (Bad file descriptor)
fstat(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 2), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ffff7ff6000
write(1, "\377\377\377\377", 4����)         = 4
exit_group(0)                           = ?
+++ exited with 0 +++
```

# strace - Ejemplo

**Strace** permite establecer el tipo de **system calls** trazadas. Por ejemplo **-e trace=open,read** hace que solo se muestren invocaciones a **open** y **read**; **-e trace=file** indica que se procesan solo **system calls** que toman como parámetro un nombre de archivo (**open**, **stat**, **chmod**, etc).

Básicamente la opción **-e trace=set** permite especificar **system calls** individuales o los grupos predefinidos **file**, **process**, **network**, **ipc**, **signal**, **desc** y **memory**.

Mas información en **man 1 strace**.

# strace - Procesos hijos

**Strace** permite realizar un seguimiento de los procesos hijos. Para ello se utiliza el flag **-f** (**fork**, **vfork** y **clone**). Sin embargo, al realizar el trazado de los procesos hijos en la misma consola la salida se puede tornar un poco cofusa.

# strace - Procesos hijos

```
[pid 12121] <... clone resumed> child_stack=0,
flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7ffff7fc39d0) = 12129
Process 12129 attached
[pid 12121] clone( <unfinished ...>
[pid 12124] open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC <unfinished ...>
[pid 12129] execve("./hola", ["./hola"], [/* 92 vars */]Process 12130 attached
 <unfinished ...>
[pid 12121] <... clone resumed> child_stack=0,
flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7ffff7fc39d0) = 12130
[pid 12126] open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC <unfinished ...>
[pid 12121] clone( <unfinished ...>
[pid 12130] execve("./hola", ["./hola"], [/* 92 vars */]Process 12131 attached
 <unfinished ...>
[pid 12121] <... clone resumed> child_stack=0,
flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7ffff7fc39d0) = 12131
[pid 12126] <... open resumed> )        = 3
[pid 12124] <... open resumed> )        = 3
[pid 12121] clone( <unfinished ...>
[pid 12130] <... execve resumed> )      = 0
[pid 12131] execve("./hola", ["./hola"], [/* 92 vars */]Process 12132 attached
```

# strace - Procesos hijos

Supongase el programa **forker**:

```c
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char** argv){
    int i=0;
    for(i=0;i<50;i++){
        if(fork()==0){
            execl("./hola", "./hola", 0);
            exit(-1);
        }
    }

    return 0;
}
```

# strace - Procesos hijos

Para facilitar el seguimiento de procesos hijos se utilizan los flags **-ff** y **-o**. Por ejemplo:

```
$ strace -ff -o forker.strace -e trace=open,read,write,process ./forker
```

Hace que se tracen todas las llamadas a **open**, **read**, **write** y las relacionadas con procesos (**fork**, **wait**, **exec**), que se siga a los procesos hijos y que se guarde el resultado de la traza de cada proceso en el archivo **forker.strace.PID**, donde **PID** es diferente para cada proceso. De esta forma se tiene un archivo por cada proceso lanzado y no todo mezclado.

# netstat

Netstat permite mostrar las tablas de ruteo, la configuración de las interfaces de red y entre otras cosas examinar las conexiones de red del equipo. En los ejemplos que siguen se puede agregar el flag **-n** para inhibir la resolución de nombres (se muestran las direcciones IP en vez del host y los numeros de puerto en vez del nombre del protocolo conocido).

# netstat

Para ver la tabla de ruteo:

```
$ netstat -r
```

Para ver las estadísticas de las diferentes interfaces:

```
$ netstat -i
```

Para ver la información detallada de las interfaces:

```
$ netstat -ie
```

# netstat

En la materia el uso mas común de netstat es el de monitorear las conexiones de red. A continuación un ejemplo:

```
$ netstat -utacp
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program na
me
tcp        0      0 *:netbios-ssn           *:*                     LISTEN      -
tcp        0      0 *:sunrpc                *:*                     LISTEN      -
tcp        0      0 192.168.1.228:47901     client-12b.v.drop:https ESTABLISHED 1387/dropbox
tcp       38      0 192.168.1.228:58802     client-15a.v.drop:https CLOSE_WAIT  1387/dropbox
udp        0      0 *:bootpc                *:*                                 -
udp        0      0 *:sunrpc                *:*                                 -
udp        0      0 *:db-lsp-disc           *:*                                 1387/dropbox
```

# netstat

El campo **Proto** indica el tipo de conexion: *UDP* (flag **-u**) o *TCP* (flag **-t**). El campo **Send-Q** indica la cantidad de bytes enviados pendientes de **ACK** y **Recv-Q** indica la cantidad de bytes recibidos que todavía no fueron copiados al proceso que utiliza el socket.

Las conexiones que carecen de **Foreign Address** indican sockets pasivos (operación **listen** de la api de sockets, en espera de conexiones externas) y se muestran con el flag **-a**.

El flag **-c** hace que la información se actualice constantemente y se vuelva a mostrar cada segundo. Por último el flag **-p** hace que se muestre, cuando sea posible, el **PID** y proceso dueño del socket. Pueden ser necesarios privilegios de administrador para mostrar procesos que no hayan sido iniciados por el usuario.

# netstat - Timers

Existe la posibilidad también de mostrar la información asociada a los timers de una conexión *TCP*. Para ello se utiliza el flag **-o** o **–timers**.

La columna del timer tiene el siguiente formato: **estado (t1/t2/t3)**.

Estado puede ser **off** (no hay timer para la conexion), **on** (timer de retransmision), **keepalive** (timer de keepalive), **timewait** (socket en estado TIME_WAIT), **unkn** (timer para verificacion de TCP **Zero Window Probe**).

El campo **t1** indica el tiempo restante del timer. **t2** cuenta la cantidad de retransmisiones que se produjeron y **t3** es siempre **0** para sockets no establecidos o en estado TIME_WAIT y para sockets conectados es la cantidad de TCP **Zero Window Probe** pendientes de respuesta.

# ulimit y core dumps

**Ulimit** es un comando del *shell* que permite consultar y modificar los límites del usuario dentro del sistema mediante los *syscalls* **getrlimit** y **setrlimit**. Los límites modificados tienen validez sólo dentro del shell que invocó **ulimit** y los procesos lanzados dentro del mismo. Para ver un listado de los límite actuales se utiliza el comando **ulimit -a**. El listado completo de los límites que se pueden establecer se encuentran en el manual de bash. Lo que nos interesa en este caso es aumentar los límites de los archivos **core**, que por defecto suele ser 0 bytes (no se crea el core dump).

# Core dumps

Un **core dump** es un archivo que contiene una imagen de un proceso en memoria. Al producirse ciertos **signals** el sistema operativo crea un **core dump** del proceso antes de abortar. Como por defecto el límite de tamaño de dicho archivo es **0 bytes** el archivo no es creado. Para habilitar los **core dump** se debe modificar dicho límite:

```
$ ulimit -c unlimited
```

# Core dumps - nombres

Por defecto el nombre del **core dump** es **core**. Si se tienen varios procesos corriendo y dos o mas de ellos abortan un **core** sobreescribe al anterior, perdiendose los dumps anteriores. Para evitar esto se puede establecer un patron de nombre para los dumps escribiendo al archivo **/proc/sys/kernel/core_pattern** (kernels mayores a 2.4.21 o a 2.6). Por ejemplo para hacer que los dumps contengan el nombre del ejecutable y su pid se puede utilizar el comando:

```
$ echo "%e-%p.dump" > /proc/sys/kernel/core_pattern
```

Mas detalles en **man 5 core**.

# Core dumps - gdb

Utilizando el **core dump** se puede saber exactamente qué estaba haciendo el proceso antes de abortar.

```
$ gdb proceso1
Reading symbols from proceso1...done.
(gdb) core proceso1-12721.dump
```

A partir de la carga del dump se puede examinar por ejemplo el backtrace y saber qué causó que abortara el proceso.

# C^z, jobs, fg, bg, etc

Las consolas **UNIX** proveen varios mecanismos para interactuar con los procesos lanzados.

Con un proceso lanzado en primer plano, **C^Z** (control-z) hace que el proceso entre en estado de suspensión. El comando **jobs** muestra un listado de los trabajos suspendidos. Los comandos **fg** y **bg** hacen que el numero de trabajo especificado (o el mas reciente si no se especifica) pase a primero plano o a segundo plano respectivamente.

```
$ sleep 10m
^Z
[1]  + 13354 suspended  sleep 10m
```

# C^z, jobs, fg, bg, etc

C^C interrumpe el proceso en primer plano y C^D envía **EOF** al proceso en primer plano. El comando **kill** puede aceptar un número de trabajo en vez de un **PID** (se precede con **%** para distinguirlo).

```
$ jobs
[1]    suspended  sleep 10m
[2]  - suspended  sleep 20m
[3]  + suspended  sleep 50m
$ kill %2
[2]  - 13408 terminated  sleep 20m
% fg %1
[1]  - 13354 continued  sleep 10m
^C
$ bg %3
[3]  - 13460 continued  sleep 50m
$ jobs
[3]  + running    sleep 50m
```

# htop

**htop** es una aplicación de consola, interfaz mejorada de **top**. Muestra los procesos con toda la información importante como **PID**, Usuario, memoria utilizada, **cpu**, comando, etc. Permite ordenar los procesos por cualquiera de los campos, organizarlos en una vista de árbol, buscar, filtrar y elegir un proceso en particular para mandarle una señal, cambiar su prioridad, examinar el proceso con **strace** o ver los archivos que está utilizando con **lsof**.

# killall

Se utiliza para matar o enviar señales a procesos por nombre en vez de **PID**. Se pueden utilizar expresiones regulares para especificar procesos (flag **-r**) o por usuario. Por ejemplo **killall -SIGHUP -u pedro** envía la señal **HUP** a todos los procesos del usuario **pedro** , **killall miproceso** envía la señal **TERM** a todas las instancias del proceso **miproceso** y **killall -r 'proces.' -SIGKILL** envía **KILL** a los procesos que concidan con la epxresion como **proceso**, **procesa**, **process**, **procese**, **proces1**, etc.

En vez de especificar las señales por nombre se puede utilizar el número: **killall -9 proceso**.

# Links de interés

- https://sourceware.org/gdb/current/onlinedocs/gdb/
- http://linux.die.net/man/5/core
- http://linux.die.net/man/1/strace
- http://linux.die.net/man/8/netstat