

machinalis





GDB

Felipe Manzano
felipe.andres.manzano@gmail.com

Machinalis

06/06/2011



Temas - GDB

Introducción

Primera sesión de gdb

Ejecutando programas desde cero

Controlando el entorno

Depurando programas en ejecución

Depurando programas con hilos o forks

Depurando programas con señales

Interrumpiendo la ejecución

Continuando la ejecución

Analizando la pila

Comandos definidos por el usuario

Crashes y coredumps

Depurando remotamente

Referencias



Introducción - historia

El propósito de un depurador como GDB es permitir al usuario ver lo que está ocurriendo “dentro” de otro programa mientras que se está ejecutando — o lo que estaba haciendo en el momento que falló.

del 'man gdb'

- ▶ GDB fue inicialmente escrito por Richard Stallman en 1986
- ▶ Es software libre con licencia GNU
- ▶ Actualmente está en la versión 7.2.x



Introducción - características

GDB puede hacer cuatro tipos de magia para capturar errores en el acto:

- ▶ Ejecutar un programa desde cero, especificando cualquier cosa que pueda afectar a su comportamiento
- ▶ Detener la ejecución de un programa al cumplirse ciertas condiciones especificadas
- ▶ Examinar qué ha pasado en un crash post-mortem
- ▶ Cambiar cualquier detalle de la ejecución de un programa, memoria, registros e incluso ejecutar funciones del programa
- ▶ Puede depurar C, C++, y código nativo entre otros.
- ▶ Robusto y automatizable



Introducción - Entrando al gdb

- ▶ GDB se ejecuta simplemente con el comando `gdb`
- ▶ Las formas más usuales de llamar al `gdb` son:
 - ▶ `gdb -exec programa`
 - ▶ `gdb -exec programa -core core` donde `core` es un file con un *coredump*
 - ▶ `gdb -exec programa -core 1234` donde 1234 es el PID
 - ▶ `gdb -silent` no imprime la versión ni la licencia
 - ▶ `gdb -help` imprime la ayuda para la línea de comandos
- ▶ otra interesante es `-x file.txt` donde se le pasa una lista de comandos internos que `gdb` ejecutará al principio.



Introducción - Una vez adentro del gdb

- ▶ Una vez adentro GDB presenta un *shell* con todos los beneficios de GNU `readline`
- ▶ Usar y abusar del CTRL+R(○ ESC-?) y TAB
- ▶ Se sale con `quit` o CTRL-D
- ▶ Se puede llamar al shell directamente desde gdb así:
(gdb) `shell ls -al`
- ▶ Es fácil recompilar sin salir del gdb:
(gdb) `make makeargs`
es equivalente a
(gdb) `shell make makeargs`
- ▶ Tiene un `help` y un `apropos`



Introducción - Compilar para depuración

- ▶ La información de depuración se incluye con `-g` o `-ggdb`
`$ gcc -ggdb -o test.dbg test.c`
- ▶ Para borrar esta información antes de un release:
`$ cp test test.dbg`
`$ strip test`
- ▶ Nos quedamos con una copia de la versión que contiene la información de depuración
- ▶ Los *coredump* generados con la versión de release podrán usarse con la versión de depuración
- ▶ Para analizar bugs privados es mejor usar una versión sin optimizaciones:
`$ gcc -ggdb -O0 -o test.dbg test.c`
- ▶ Para incluir información sobre los macros usar algo así:
`$ gcc -gdwarf-2 -g3 sample.c -o sample.`



Primera sesión de gdb - Un ejemplito (dataentry)

```
1  #include<stdio.h>
2  #include<assert.h>
3  int
4  main (int argc, char *argv[])
5  {
6      int edad;
7      char nombre[10];
8      printf ("Ingrese su nombre:_");
9      scanf ("%100s", nombre);
10     printf ("Ingrese su edad:_");
11     scanf ("%d", &edad);
12     printf ("Hola _%s, _ud _tiene _%d _pirulos!!\n", \
13                                     nombre, edad);
14     return 0;
15 }
```



Primera sesión de gdb - comandos útiles (1/2)

- ▶ Se compila así:

```
$ gcc -O0 -g dataentry.c -o dataentry
```

- ▶ El depurador se ejecuta así:

```
$ gdb dataentry
```

- ▶ Comandos útiles:

- ▶ start
- ▶ print edad
- ▶ print nombre
- ▶ next
- ▶ shell man scanf
- ▶ quit

- ▶ Maximiliano(18) y Segismundo(65) han reportado problemas... debug!



Primera sesión de gdb - más comandos útiles

- ▶ GDB usa expresiones C-like para hacer referencia a los datos
- ▶ El comando `print` `EXPR` espera cosas como:
 - ▶ `print (int) argc`
 - ▶ `print (char*) argv[0]`
 - ▶ `print *(struct mi_estructura*) puntero`
- ▶ Para inspeccionar memoria se usa el comando `x/` así:
 - ▶ `x/x ADDR` imprime un entero apuntado por `ADDR`
 - ▶ `x/s ADDR` imprime un string que empieza en `ADDR`
 - ▶ `x/c ADDR` imprime un byte apuntado por `ADDR`



Primera sesión de gdb - gdb shell

```
1  #include <stdio.h>
2  struct mi_estructura{
3      int entero;
4      float flotante;
5      char cadena[20];
6  };
7  int main(){
8      struct mi_estructura datos;
9      memcpy(&datos, "\x39\x30\x00\x00\xcd\xcc\x8c\x3f"
10             "\x48\x6f\x6c\x61\x2c\x20\x6d\x75"
11             "\x6e\x64\x6f\x00", 28);
12
13      if (datos.entero + datos.flotante > 0.2)
14          printf("datos.cadena:_%s\n",
15                datos.cadena);
16  }
```

Primera sesión de gdb - gdb shell

- ▶ Le línea de comando de gdb esta paginada.
- ▶ Si un comando imprime información de mas de una pagina se de tiene la ejecución
- ▶ El tamaño de la página se define así:
`(gdb) set height 100`
- ▶ Toda la salida puede guardarse en un archivo de *log* con:
`(gdb) set log on /tmp/archivo.log`
- ▶ para dejar de *loggear*
`(gdb) set log off`
- ▶ El código fuente se puede listar con el comando `list`
- ▶ ENTER a secas repite el comando anterior



Primera sesión de gdb - Expresiones automaticas

- ▶ Se puede *seguir* el valor de una variable o expresión con el comando `display`
- ▶ `display /fmt EXPR` agrega una expresión ala lista de expresiones automáticas
- ▶ `undisplay NUM` remueve cierta expresión de la lista
- ▶ `info display` muestra la lista de expresiones automáticas
- ▶ Ejemplos:
 - ▶ `display /s miString`
 - ▶ `display /d miEntero`
 - ▶ `display *(struct MiStruct*) miStruct`



Ejecutando programas en GDB

- ▶ Para iniciar la depuración de un programa se usan:
 - ▶ `run` ejecuta desde el comienzo
 - ▶ `start` idem pero se detiene en `main`
- ▶ La traza de ejecución de un programa depende de la información proporcionada por el entorno
- ▶ Todo esto puede controlarse desde `gdb`:
 - ▶ Los *argumentos* de la línea de comando
 - ▶ Las *variables de entorno* ver `env`
 - ▶ El *directorio de trabajo*
 - ▶ La *salida y entrada estándar*



Los argumentos

- ▶ Los argumentos se pueden asignar previamente así:
`(gdb) set args arg1 arg2 arg3`
`(gdb) run`
- ▶ o se pueden pasar normalmente a `run` o `start`
`(gdb)run arg1 arg2 arg3`
(si no se especifica nada se usa lo seteado anteriormente)
- ▶ Para verlos: `show args`
- ▶ Si se puede son procesados por el shell. Es decir * se reemplazará por los nombres de archivo del directorio actual.



Los argumentos - Ejemplito

```
1  int
2  main (int argc, char *argv[])
3  {
4      int i;
5      /* Esperamos al menos 1 argumento */
6      assert (argc==2);
7      /* En el primer argumento esperamos un numero */
8      sscanf(argv[1],"%u",(unsigned int*)&i);
9      /* un numero menor que 100*/
10     assert(i<100 && i>=0);
11     printf("El parametro %d esta entre 0 y 100\n",i);
12     return 0;
13 }
```



Variables de entorno y directorio

- ▶ Las variables de entorno se setean así
`(gdb) set environment VARIABLE="Este es el valor"`
- ▶ Se pueden revisar así:
`(gdb) show environment VARIABLE`
- ▶ Y borrar así:
`(gdb) unset environment VARIABLE`
- ▶ El directorio actual o de trabajo se setea con `cd` y se revisa con `pwd` directamente desde el `gdb`



Variables de entorno y directorio - Ejemplito

```
1  int
2  main (int argc, char *argv[])
3  {
4      char cwd[1024];
5      char * variable;
6      variable = getenv("VARIABLE");
7      /* La VARIABLE de entorno debe existir */
8      assert (variable != NULL);
9      printf("VARIABLE=%s\n",variable);
10     /* El directorio actual debe ser /etc */
11     assert(getcwd(cwd, sizeof(cwd)) != NULL);
12     assert(strcmp("/etc",cwd)==0);
13     printf("El_directorio_actual_es_%s\n",cwd);
14     return 0;
15 }
```



Entrada y salida estándar

- ▶ La entrada y salida estándar se puede direccionar al igual que en bash
- ▶ Funciona con el con run y start así:
`(gdb) run < archivo.in >archivo.out`



Entrada y salida estándar - Ejemplito

```
1  int main (int argc, char *argv[])
2  {
3      FILE *f;
4      char bufferA[1024];
5      char bufferB[1024];
6      char cwd[1024];
7      /* El directorio actual debe ser /etc */
8      assert (getcwd (cwd, sizeof (cwd)) != NULL);
9      assert (strcmp ("/etc", cwd) == 0);
10     printf ("El_directorio_actual_es_%s\n", cwd);
11     f = fopen("passwd","r");
12     fread(bufferA,1,1024,f);
13     fread(bufferB,1,1024,stdin);
14     assert(memcmp(bufferA,bufferB,1024)==0);
15     printf("stdin_es_identico_a_/etc/passwd\n");
16     return 0;
17 }
```

Depurando programas en ejecución

- ▶ Se usa `attach`, `dettach` y `kill` para depurar un programa en ejecución
- ▶ Inmediatamente después de un `attach` el programa queda suspendido
- ▶ El programa siendo depurado puede interrumpirse con `CTRL+C` y reanudarse con `continue`
- ▶ La memoria o variables del programa se inspeccionan con `print` o `x` así:
 - ▶ `(gdb) print flag`
 - ▶ `(gdb) x/s tema`
- ▶ Para modificar variables se puede usar `set` o incluso `print` así:
 - ▶ `(gdb) set flag=0`
 - ▶ `(gdb) p strcpy(tema,"Me puedo debugear")`



Depurando programas en ejecución - Ejemplito

```
1  int
2  main (int argc, char *argv[])
3  {
4      char tema[]="Me_puedo_programar";
5      char banda[]="Virus";
6      int flag=1;
7      printf("Hola_mi_pid_es_%d\n", getpid());
8      while(flag);
9
10     printf("Mi_tema_preferido_es_'%s'_de_la_banda"\
11            "'%s'.\n", tema, banda);
12     printf("Link_http://www.youtube.com/"\
13            "watch?v=r23-Y6ElWV4.\n");
14     return 0;
15 }
```



Depurando programas con hilos

GDB provee algunas facilidades para depurar hilos

- ▶ Notifica automáticamente la creación de hilos
- ▶ `info threads` lista los hilos existentes
- ▶ En cada momento se está examinando el contexto de un hilo determinado
- ▶ `thread NUM` cambia el hilo actual
- ▶ `thread apply all` ejecuta un comando GDB en todos los hilos. Por ej.
`(gdb) thread apply all p flag`
`(gdb) thread apply all p flag=0`
- ▶ Para ejecutar un solo hilo por vez usar algo así:
`(gdb) set scheduler-locking on`
- ▶ Se puede consultar con `show scheduler-locking`



Depurando programas con hilos - Ejemplito

```
1 void * inc (void *ptr){
2     int flag = 1;
3     /* Loop loco */
4     while (flag==1)
5         ++(*(int *) ptr);
6     printf ("Salio_despues_de_%d_iteraciones\n",*(int *) p
7     return NULL;
8 }
9 int main (int argc, char * argv[]){
10     pthread_t thread[2];
11     int x = 0, y = 0;
12     pthread_create (&thread[0], NULL, inc, &x);
13     pthread_create (&thread[1], NULL, inc, &y);
14     pthread_join(thread[1], NULL);
15     pthread_join(thread[0], NULL);
16     return 0;
17 }
```

Depurando programas con forks

Para controlar el comportamiento de gdb en tiempo de fork se usa `follow-fork-mode`

- ▶ `set follow-fork-mode child` instruye al gdb a *seguir* siempre al proceso hijo
- ▶ `set follow-fork-mode parent` instruye al gdb a *quedarse* en el proceso hijo
- ▶ `show follow-fork-mode` muestra el estado actual de la variable



Depurando programas con forks - Ejemplito

```
1  #include <sys/types.h>
2  #include <unistd.h>
3  #include <stdio.h>
4  int
5  main (int argc, char *argv[])
6  {
7      pid_t pid = fork ();
8      if (!pid)
9          printf ("Proceso_hijo:_PID_%d\n", getpid());
10     else
11         printf ("Proceso_padre:_PID_%d\n", getpid());
12     return 0;
13 }
```



Depurando programas con señales

- ▶ Las *signals* son eventos asíncronos provistos por el sistema operativo. (ver `man 7 signal`)
- ▶ `info signals` imprime la lista de todas las señales y cómo gdb esta configurado para manejarlas
- ▶ Cada señal puede ser interpretada por gdb de diferentes maneras, este comportamiento se configura con:

`(gdb)handle SIGNAL KEYWORD`

... donde KEYWORD es uno de:

- ▶ `stop` y `nostop` detiene o no la ejecución
- ▶ `print` y `noprint` imprime o no un mensaje informativo
- ▶ `pass` y `nopass` pasa o nopasa la señal al programa (es decir si ejecuta o no el manejador asociado en el programa)



Depurando programas con Señales - Ejemplito

```
1  int cont = 0; int flag = 1;
2  void handler (int sig) {
3      signal (SIGALRM, SIG_IGN);
4      if(flag){
5          printf ("SIGALRM_exiting!_:(");
6          exit(1);
7      }
8      alarm (2);
9      signal (SIGALRM, handler);
10 }
11 void main (int argc, char *argv[]){
12     signal (SIGALRM, handler);
13     alarm (2);
14     while (cont<500){
15         sleep(1);
16         cont++;
17     }
18     printf("Ganaste!\n");
19 }
```



Interrumpiendo la ejecución - Motivación

```
1  int readSUM(istream& in){
2      char buffer[256];
3      int i, co;
4      assert(in.read (buffer,256).good());
5      for(i=0,co=0;i<in.gcount();i++)
6          co+=buffer[i];
7      return co;
8  }
9  int main (int argc, char* argv[]) {
10     int checksum;
11     checksum = readSUM(cin);
12     assert(checksum > 1000 && checksum < 16640);
13     cout << "El_checksum_es:_" << checksum <<endl;
14     return 0;
15 }
```



Interrumpiendo la ejecución

- ▶ El propósito principal de usar un depurador es poder detener el procesamiento en puntos de interés (o analizar cuando se está en algún crash/deadlock)
- ▶ GDB provee tres formas de detener la ejecución en lugares interesantes:
 - ▶ *breakpoints*, detienen el programa en un lugar o punto del código
 - ▶ *watchpoints*, detienen el programa cuando cierto dato se accede o modifica
 - ▶ *catchpoints*, detiene el programa cuando se da una excepción de C++ o se carga un módulo o biblioteca
- ▶ gdb asigna un número a cada *breakpoint*, *watchpoint* y *catchpoint* que se crea
- ▶ ese número se usa como referencia desde otros comandos, sirve para la habilitación, deshabilitación y borrado de los mismos



Interrumpiendo la ejecución - Breakpoints

- ▶ Los *breakpoints* se pueden setear de muchas maneras.
 - ▶ `break nombre_de_funcion` pone un breakpoint al comienzo de una función
 - ▶ `break archivo.c:línea` pone un breakpoint en el número de línea especificado
 - ▶ `break a secas` pone un breakpoint en la posición actual
 - ▶ `rbreak regExp` pone un breakpoint en todas la funciones que cumplan con la expresión regular `regExp`. Ej
`rbreak .*alloc.*`



Interrumpiendo la ejecución - Watchpoints

- ▶ Los *watchpoints* monitorean el estado de ciertas partes de la memoria
 - ▶ `watch` `EXPR` detiene la ejecución cuando `EXPR` es modificada
 - ▶ `rwatch` `EXPR` detiene la ejecución simplemente cuando `EXPR` es leída
 - ▶ `awatch` `EXPR` detiene la ejecución cuando `EXPR` es modificada y/o leída
- ▶ `delete`, `disable` y `enable` hacen lo esperado
- ▶ `info watchpoints` muestra la lista de watchpoints



Interrumpiendo la ejecución - Catchpoints

- ▶ Se pueden utilizar los *catchpoints* para detener la ejecución en ciertos eventos del programa. Se usan así:

(gdb) catch EVENT

donde EVENT es alguno de los siguientes eventos:

- ▶ throw cuando se lanza una excepción de C++
- ▶ catch cuando se maneja una excepción de C++
- ▶ load LIBNAME cuando se carga LIBNAME a la memoria del proceso
- ▶ unload LIBNAME cuando se descarga LIBNAME de la memoria del proceso



Interrumpiendo la ejecución - Administración


- ▶ `delete NBREAK` borra el breakpoint
- ▶ `disable NBREAK` lo deshabilita temporalmente
- ▶ `enable NBREAK` lo vuelve a habilitar
- ▶ `ignore NBREAK NUM` ignorara NUM veces el breakpoint NBREAK
- ▶ `info breakpoints` (o `watchpoints`, o `catchpoints`) muestra el estado de los breakpoints (o `watchpoints`, o `catchpoints`)



Interrumpiendo la ejecución - Ejemplito

```
1  class NewDelete{
2      char local[256];
3      char *c;
4      size_t len;
5      int error;
6  public:
7      NewDelete (size_t size) {
8          try{
9              c = new char[size];
10             len = size;
11         }catch (...) {
12             cout << "No_hay_suficiente_memoria" <<endl;
13             globalerror = error = 1;
14         }
15         error = 0;
16     }
17
18     ~NewDelete(){ delete c; }
19 };

```



Interrumpiendo la ejecución - Ejemplito (..cont)

```
1 void
2 newdelete(size_t size){
3     NewDelete nd(size);
4 }
5 int main () {
6     int checksum;
7     checksum = readSUM(cin);
8     newdelete(checksum);
9     cout << "El_checksum_es:_" << checksum <<endl;
10    return 0;
11 }
```



Interrumpiendo la ejecución - Efectos secundarios

- ▶ Se puede asignar condiciones a casi cualquier cosa de esta manera:

```
(gdb) condition NBREAK flag==1
```

- ▶ y se remueven así:

```
(gdb) condition NBREAK
```

- ▶ También se puede ejecutar una lista de comandos en cada breakpoint!

```
(gdb) command NBREAK  
print "BREAKPOINT!"  
continue  
end
```

- ▶ y se quita asignándole la lista vacía de comandos:

```
(gdb) command NBREAK  
end
```



Continuando la ejecución

- ▶ La ejecución puede reanudarse con `continue`
- ▶ `next` ejecuta la próxima línea
- ▶ `stepi` ejecuta la próxima línea metiéndose dentro de las funciones
- ▶ `finnish` ejecuta hasta salir de la función actual
- ▶ `until` ejecuta hasta pasar número de línea especificado. Optimo para salir de bucles!
- ▶ `jump NUMLINEA` continua ejecutando desde la linea de codigo especificada
- ▶ `return VALOR` ignora el comportamiento restante de la funcion actual y retorna VALOR a la llamante



Analizando la pila

- ▶ OK, el programa se detuvo. Pero como llegué ahí?
- ▶ Cada vez que se realiza una llamada a procedimiento se *apila* información para poder retornar del mismo
- ▶ Esto se denomina marco de activación o *stack frame*
- ▶ GDB asigna un número a cada marco de activación anidado y se listan así:

```
(gdb) backtrace
```

- ▶ Sólo pueden accederse a las variables locales del marco actual
- ▶ Se puede navegar o cambiar de marco con:

```
(gdb) frame N
```
- ▶ O iterativamente usando up y down
- ▶ `info args` e `info locals` muestran los argumentos y las variables locales del marco actual
- ▶ `info catch` lista los manejadores de excepciones activos



Analizando la pila - Ejemplito

```
1  #include <stdio.h>
2  int fib(int n)
3  {
4  if(n==1 || n==0)
5      return n;
6  else
7      return fib(n-1)+fib(n-2);
8  }
9
10 int main()
11 {
12     printf("Fibonacci(%d)=_%d\n\n",10,fib(10));
13     return 0;
14 }
```



Comandos definidos por el usuario

- ▶ GDB permite definir comandos personalizados
- ▶ Son una lista de comandos comunes de gdb empaquetados y con nombre
- ▶ Pueden tomar argumentos
- ▶ Tienen esta pinta

```
define suma  
  print $arg0 $arg1 $arg2  
end
```

- ▶ y se usan asi:

```
(gdb) suma 1 2 3  
$1 = 6
```



Comandos definidos por el usuario

- ▶ Tiene if, while y variables auxiliares como \$i y \$aux

```
(gdb) set $i=0
```

```
(gdb) while $i<100
```

```
set $i=$i+1
```

```
if $i%3==0
```

```
    printf "%d es multiple de 3\n", $i
```

```
end
```

```
end
```

- ▶ Los *scopes* se cierran con end
- ▶ Ideales para crear una biblioteca de funciones de conveniencia en .gdbinit
- ▶ Probar esto y luego depurar un programa (usando start):

```
define hook-start
```

```
printf "STAAAAART!"
```

```
end
```



Crashes y coredumps

In computing, a core dump, more properly a memory dump or storage dump, consists of the recorded state of the working memory of a computer program at a specific time, generally when the program has terminated abnormally

- ▶ la generación o no de coredumps se controla con el comando `ulimit`
`$ ulimit -c unlimited`
- ▶ tambien ver `man core` para información detallada
- ▶ algunas senales generan por defecto coredumps
 - ▶ Program terminated with signal 4, Illegal instruction.
 - ▶ Program terminated with signal 6, Aborted.
 - ▶ Program terminated with signal 8, Arithmetic exception.
 - ▶ Program terminated with signal 11, Segmentation fault.



Crashes y coredumps - División por cero

```
1 void
2 sigpfe ()
3 {
4     int a = 1;
5     int b = 0;
6     float c;
7     c = a/b;
8     c++;
9 }
```



Crashes y coredumps - Fallo de pagina de memoria

```
1 void
2 sigsegv ()
3 {
4     *(int*)2340=1000;
5 }
```



Crashes y coredumps - Abort!

```
1 void
2 sigabort ()
3 {
4     char * p = (char*)malloc(1);
5     free(p);
6     free(p);
7     //abort();
8 }
```



Crashes y coredumps - Instruccion ilegal

```
1 void
2 sigill ()
3 {
4     asm ( ".byte_-2\n" );
5 }
```

En otros OSs tambien aparece SIGBUS



Depurando remotamente - gdbserver

- ▶ gdbserver es un depurador reducido que puede controlarse via gdb
- ▶ gdb y gdbserver se comunican por TCP o una linea serial
- ▶ Una vez establecida la comunicación GDB se comporta normalmente
- ▶ El server se ejecuta así:

```
$ gdbserver host:2345 programa param1 param2
```
- ▶ Y desde el gdb remoto te conectas así:

```
(gdb) target remote host:2345
```

gdbserver is not a complete replacement for the debugging stubs, because it requires essentially the same operating-system facilities that GDB itself does. In fact, a system that can run gdbserver to connect to a remote GDB could also run GDB locally!



Depurando remotamente - otra opción?

- ▶ Poner un GDB completo en el target
- ▶ Conectarse por ssh/telnet
- ▶ Con `show directory` se listan los caminos donde gdb busca los archivos de código fuente
- ▶ Con `directory CAMINO` se agregan CAMINOS donde GDB buscare el código
- ▶ Montar un nfs en el target con el repositorio de fuentes en uso
- ▶ Probarlo con `list`



Referencias

- ▶ <http://gdb.gnu.org/>
- ▶ <http://www.acsu.buffalo.edu/~charngda/gdb.html>
- ▶ <http://www.ibm.com/developerworks/aix/library/au-gdb.html>
- ▶ <http://sourceware.org/gdb/onlinedocs/gdb/>
- ▶ <http://reverse.put.as/2011/03/07/small-update-to-gdbinit-and-to-the-website/>
- ▶ <mailto:felipe.andres.manzano@gmail.com>





the missing piece.