Arquitectura de Sistemas

Práctica 1: Entorno de desarrollo GNU

Gustavo Romero López

Actualizado: 22 de febrero de 2018

Arquitectura y Tecnología de Computadores

Material complementario

Manuales:

- Hardware:
  - AMD: `http://developer.amd.com/documentation.jsp`
  - Intel: `http://www.intel.com/design/pentium4/manuals/index_new.htm`
- Software:
  - as: `http://sourceware.org/binutils/docs/as/index.html`
  - nasm: `http://sourceforge.net/projects/nasm`
- Chuleta 8086: `http://jegerlehner.ch/intel/IntelCodeTable_es.pdf`
- Chuleta GDB: `www.digilife.be/quickreferences/QRC/GDBQuickReference.pdf`

Resursos en Internet:

- Programming from the Ground Up: `http://download.savannah.gnu.org/releases/pgubook/ProgrammingGroundUp-0-8.pdf`
- Linux Assembly: `http://www.linuxassembly.org`

Objetivos

- Ya sabéis programar en ensamblador... recordad **EC**.
- Linux es tu amigo: si no sabes algo pregúntale (**man**).
- Hoy haremos varias cosas:
  - Recordaremos ejemplos básicos.
  - Veremos como aprender de otros que saben más: **gcc**.
  - Recordaremos viejos amigos:
    - **makefile**: hará el trabajo sucio y rutinario por nosotros.
    - **as**: el ensambladores.
    - **ld**: el enlazador.
    - **gcc**: el compilador.
    - **nm**: lista los símbolos de un fichero.
    - **objdump**: el desensambladores.
    - **gdb** y **ddd** (gdb con cirugía estética): los depuradores.

Ejemplo básico en ensamblador: pienso.s

```
.data
        msg:    .string "cognito, ergo sum\n"
        tam:    .int . - msg

.text
        .global _start

_start:
        mov   $1, %rax     # write
        mov   $1, %rdi     # stdout
        mov   $msg, %rsi   # texto
        mov   tam,  %rdx   # tamaño
        syscall            # llamada a write

        mov   $60,  %rax   # exit
        xor   %rdi, %rdi   # 0
        syscall            # llamada a exit
```

Ejemplo básico: ¿cómo hacer ejecutable mi programa?

¿Cómo hacer ejecutable el código anterior?

- opción a: ensamblar + enlazar
  - `as pienso.s -o pienso.o`
  - `ld pienso.o -o pienso`
- opción b: compilar
  - `gcc pienso.s -o pienso`
- opción c: que lo haga alguien por mi $\Longrightarrow$ makefile.
  - fichero especial con dos tipos de líneas: definiciones y objetivos.

https://pccito.ugr.es/ gustavo/as/practicas/01/makefile

```
SRC = $(wildcard *.s *.c *.cc)
EXE = $(basename $(SRC))
ATT = $(EXE:=.att)
DATA = $(EXE:=.data)
PERF = $(DATA:=.perf)

ASFLAGS = -g -nostartfiles
CFLAGS = -g -march=native -Ofast -Wall
CXXFLAGS = $(CFLAGS)

all: $(EXE) $(ATT)

clean:
    $(RM) $(ATT) $(DATA) $(EXE) $(OBJ) core.* *.
        data *.data.old *~

%.att: %
    objdump -C -D $< > $@

fib-: CFLAGS+=-fno-optimize-sibling-calls
fib+: CFLAGS+=-foptimize-sibling-calls

for0: CFLAGS+=-O0
for3: CFLAGS+=-O3
```

Ejemplo básico en C++: pienso2.cc

```
#include <iostream>

using namespace std;

int main()
{
    cout << "cognito, ergo sum" << endl;
}
```


¿Qué hace gcc con mi programa?

¿Qué instrucciones ejecuta?

- Sintaxis AT&T:

  `objdump -C -D pienso2 | less`
- Sintaxis Intel:

  `objdump -C -D pienso2 -M intel | less`

¿De qué está compuesto?

- `nm pienso`
- `nm pienso2`

¿Qué bibliotecas utiliza?

- `ldd pienso`
- `ldd pienso2`

Optimizar o no optimizar?... esa es la cuestión: for.cc

```
int main()
{
    for (int i = 0; i < 10; ++i);
}
```

- Repetir compilar y desensamblar modificando CXXFLAGS en el fichero makefile:
  - Sin optimización: `CXXFLAGS = -g3 -Wall`
  - Optimizando en espacio: `CXXFLAGS = -g3 -Os -Wall`
  - Optimizando en tiempo: `CXXFLAGS = -g3 -O3 -Wall`
- ¿Cómo ha cambiado el código de la función `main`?

Optimizar o no optimizar?... esa es la cuestión: for.cc

sin optimización: gcc -O0

```
00000000004005b6 <main>:
  4005b6:       55                      push   %rbp
  4005b7:       48 89 e5                mov    %rsp,%rbp
  4005ba:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  4005c1:       83 7d fc 09             cmpl   $0x9,-0x4(%rbp)
  4005c5:       7f 06                   jg     4005cd <main+0x17>
  4005c7:       83 45 fc 01             addl   $0x1,-0x4(%rbp)
  4005cb:       eb f4                   jmp    4005c1 <main+0xb>
  4005cd:       b8 00 00 00 00          mov    $0x0,%eax
  4005d2:       5d                      pop    %rbp
  4005d3:       c3                      retq
```

máxima optimización: gcc -O3

```
00000000004004c0 <main>:
  4004c0:       31 c0                   xor    %eax,%eax
  4004c2:       c3                      retq
```

Vamos a ser unos chicos malos :) cli.cc

```
int main()
{
  asm("cli");

  return 0;
}
```

- Compilar e intentar ejecutar.
- Ejecutar: `ulimit -c unlimited`[1].
- Ejecutar de nuevo para obtener fichero `core`.
- Depurar con gdb/ddd para buscar el error.

[1] Es conveniente añadir esta orden a nuestro fichero `.bashrc`

Responda a las siguientes cuestiones sobre abcd.cc

```
int a = 0x1234, b;

int main()
{
  int c = 0x1234, d;

  return a + b + c + d;
}
```

- ¿En qué direcciones de memoria, y de qué zonas, coloca gcc las variables a, b, c y d?

Responda a las siguientes cuestiones sobre sum.cc

```
int main()
{
    int sum = 0;

    for (int i = 0; i < 1000000; ++i)
        sum += i;

    return sum;
}
```

- ¿Cómo implementa gcc los bucles `for`?
- Tenga en cuenta que el uso de optimizaciones puede variar el código generado por el compilador por lo que se recomienda probar con ellas activadas y desactivadas.

Responda a las siguientes cuestiones sobre fib.cc

```
int fib(int i)
{
    if (i < 2)
        return i;
    else
        return fib(i - 1) + fib(i - 2);
}

int main()
{
    return fib(5);
}
```

- Describa cómo la función `main` pasan y recibe parámetros de la función `fib` y cómo se utilizan estos en su interior.
- Añada la opción `-fno-optimize-sibling-calls` a la linea `CXXFLAGS` y busque las diferencias con el binario que no emplea esta opción de compilación.

Descubra el fallo en bug.cc

```
int main()
{
    const int N = 1000;

    int a = 0, b = 1, c = 2, d = 3, e = 4, f
        = 5, g = 6;

    for (int i = N; i > 0; --i)
    {
        a = b + c - i;
        b = c - d * i;
        c = d * e / i;
        d = e / (f % i);
        e = f % (g + i);
    }

    return a + b + c + d + e + f + g;
}
```

- Consejo: utilice el depurador **gdb**.

gdb

orden		**significado**	**ejemplo**
`bt`	`backtrace`	muestra lista de llamadas a función	`backtrace`
	`break` posición	crea punto de ruptura	`break ejemplo.c:25`
	`clear` [posición]	elimina punto de ruptura	`clear 3`
`c`	`continue`	continua la ejecución de un programa	`continue`
	`display` expresión	muestra el valor de una expresión tras cada punto de ruptura	`display x`
	`file` nombre	carga un fichero	`file ejemplo`
	`finish`	ejecuta mediante `next` hasta el final de un a función	`finish`
	`help` [asunto]	muestra ayuda	`help stack`
`i`	`info` elemento	muestra información	`info breakpoints`
`l`	`list` [fuentes]	muestra código fuente	`list ejemplo.c`
`n`	`next`	ejecuta una linea del programa (incluyendo llamadas a subrutinas)	`next`
`p`	`print` expresión	muestra el valor de una expresión	`print x`
`q`	`quit`	sale de gdb	`quit`
`r`	`run` [argumentos]	ejecuta el programa	`run 1 2 3`
`s`	`step`	ejecuta una linea del programa	`step`

Jugando con gdb: pila.cc

```
int fa(int a)
{
  return a + 1;
}

int fb(int b)
{
  return fa(b) * 2;
}

int fc(int c)
{
  return fb(c + 1) / fa(c - 1);
}

int main()
{
  return fc(3);
}
```