Estudio de un Sistema Operativo
Diseño de Sistemas Operativos

Funciones. Uso de la pila.

Juan Carlos Pérez & Sergio Sáez

Sergio Sáez ssaez@disca.upv.es
Juan Carlos Pérez jcperez@disca.upv.es

Introducción

- En esta sesión abordaremos diversos aspectos del mecanismo de invocación a funciones y del paso de parámetros.
- El concepto de función es básico en C. El código de inicialización generado por el compilador (enlazado por el cargador) se encarga de que la función principal del programa `int main (int argc, char * argv[])` se ponga en ejecución al inicio.
- La función `main()` recibirá los parámetros que el programa que ejecuta el binario (típicamente el intérprete de órdenes) le haya pasado a la llamada al sistema `exec(3)`.
- La pila del proceso es el lugar donde se guardan estos parámetros, las variables locales de cada función y la dirección de retorno.

Funciones

Empezaremos estudiando un programa simple que utiliza dos funciones: `atof()` y `sqrt()`. Cópialo en tu editor Sería recomendable que crearas en tu directorio *home* un subdirectorio `prog1`, guardaras ahí este programa como `prog1.c` y trabajaras dentro de ese directorio.

```
  /*  prog1.c */

#include <stdio.h>

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

    printf("Argumento 1 (texto): %s\n", argv[1]);
    valor= atof(argv[1]);
    printf("Argumento 1 (número): %.2f\n", valor);
    printf("Función: sqrt(%.2f) = %.2f\n", valor, sqrt(valor));
    return 0;
} /* end main */
```

- Compila el programa: `gcc -fno-builtin prog1.c -o prog1`
- Observa el error que aparece. Intenta entenderlo.

```
/tmp/ccwpB6Hg.o(.text+0x66): In function `main':
: referencia a `sqrt' sin definir
collect2: ld returned 1 exit status
```

- El enlazador espera encontrar definidas todas las funciones en alguno de los ficheros objeto o bibliotecas que se le especifican.
- Para que encuentre la función `sqrt()` hay que incluir en la línea de compilación, como vimos en la sesión anterior, la opción de enlazar con la biblioteca matemática. (Si no recuerdas cómo hacerlo, busca en el enunciado de la sesión anterior)

- Ahora no tenemos ningún error. Analiza los resultados ejecutando el programa primero sin argumentos y después con un argumento numérico. ¿Por qué se produce cada uno de estos errores de funcionamiento?

```
$ prog1
Argumento 1 (texto): (null)
Segmentation fault

$ prog1 16
Argumento 1 (texto): 16
Argumento 1 (número): 0.00
Función: sqrt(0.00) = 0.00
```

- El compilador de C considera por omisión (cuando la función no se ha definido en ese fichero y no se ha declarado, especificando los detalles, mediante un prototipo) que las funciones devuelven un entero y esperan recibir el tipo de parámetros que se le pasan. Una función se *declara* cuando se especifica su tipo y los tipos de sus parámetros y se *define* cuando además se especifica su código. Una función puede declararse muchas veces pero sólo puede definirse una vez. La declaración de una función en C se denomina también *prototipo*.
- Efectivamente, el compilador asume que `sqrt` devuelve un entero y por tanto interpreta mal el valor de retorno de dicha función.
- Miramos la página de manual de `sqrt` y vemos que su prototipo está en el fichero de cabecera `math.h`. Añade el correspondiente `#include` en el programa. El fichero `math.h` se encuentra en un directorio estándar de ficheros de cabecera. Se lo hemos de indicar al preprocesador de C poniéndolo entre < y >. Si estuviera en el directorio actual lo escribiríamos entre comillas.
- Compila y ejecuta el programa con un argumento numérico
- ¿Da algún aviso el compilador? ¿Funciona bien el programa?
- ¿Cuál crees que es la razón de este funcionamiento?

- Existe una opción muy útil del compilador que nunca está de más activar: `-Wall` . Con ella pedimos que se generen todos los avisos, aun los a priori menos importantes. Prueba a incluir esta opción en la compilación. Puedes añadir cualquier opción en cualquier punto de la línea de compilación excepto, por supuesto, entre `-o` y `prog1`

```
prog1b.c:18: warning: implicit declaration of
  function `atof'
```

- Miramos la página del manual de la función `atof()` y vemos que también devuelve un tipo `double` y no un entero como asume el compilador por omisión. Vemos además que su prototipo está en un fichero de cabecera determinado. Inclúyelo en tu código y compila de nuevo.

- El programa ya funciona cuando lo invocamos con un argumento numérico, pero fallará al invocarlo sin argumentos.
- Modifícalo para que compruebe el número de argumentos recibido y si no es el que espera imprima algo como:

```
$ prog1
Uso:
        prog1 <valor>
        Calcula la raiz cuadrada de <valor>
```

Uso de la pila

- Sabemos que la pila es la estructura que el compilador emplea para pasar los parámetros a las funciones y almacenar las variables locales. Vamos a ver en la práctica cómo funciona este mecanismo y a hacer algunas pruebas para entenderlo mejor.
- Crea un nuevo subdirectorio `prog2` y bájate los ficheros funciones.c, funciones.h y pila.h.
- Examina estos ficheros:
  - En los dos primeros se define la función `resta()` que simplemente resta dos números y muestra el contenido de la pila (3 enteros a partir del puntero de pila).
  - En el tercero se definen una serie de macros para acceder a la pila, modificarla, saltar a una función usando instrucciones de salto a subrutina y salto incondicional, etc.

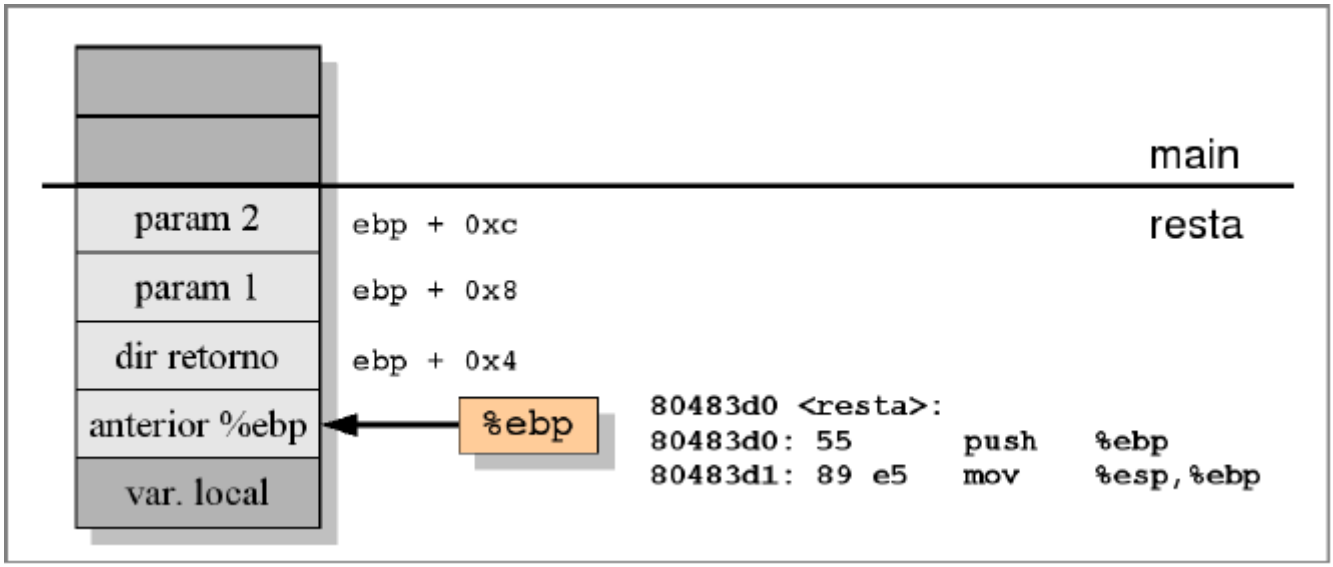
- Copia en tu editor el siguiente programa:

```
/* prog2.c */
#include <stdio.h>
#include <stdlib.h>
#include "funciones.h"
#include "pila.h"

int main (int argc, char * argv[]) {
    int e;
    int p1, p2;

    p1= atoi(argv[1]);
    p2= atoi(argv[2]);
    e= resta(p1,p2);
    printf("resta(%d,%d) = %d\n", p1, p2, e);
    return 0;
} /* end main */
```

- Compila y ejecuta `prog2`. Se recomienda utilizar las opciones del compilador `-Wall -O3`. ¿Funciona correctamente? ¿Eres capaz de interpretar el resultado?
- El compilador usa el registro `EBP` como *frame pointer*, es decir, como base para indexar los parámetros y v. locales en la pila. De este modo, a cada variable se le asocia un desplazamiento respecto a esta base y el compilador se halla libre de modificar el puntero de pila.



- Utilizando la herramienta **objdump** podemos echarle un vistazo al código en ensamblador generado para invocar a la función **resta()**.

```
; Con anterioridad se ha preparado un hueco en la pila con una
; instrucción del tipo   "sub    $0x8, %esp"
8048494: 89 44 24 04      mov    %eax,0x4(%esp)   ; Copia en la pila el parámetro 2
8048498: 89 c3            mov    %eax,%ebx        ; Almacena %eax en un registro
                                                  ; puesto que el registro %eax es
                                                  ; el de retorno
804849a: 89 34 24         mov    %esi,(%esp)      ; Copia en la pila el parámetro 1
804849d: e8 2e ff ff ff   call   80483d0 <resta>  ; Invocación de la función resta
80484a2:                                          ; Dirección de retorno
```

- Vemos que el compilador no utiliza la instrucción **push** para meter los parámetros en la pila, si no que hace alguna optimización.
- Observa que la dirección de retorno debería coincidir con la mostrada al ejecutar el programa **prog2**.

- A continuación vamos a invocar a la función **resta()** directamente en ensamblador. Para ello utilizaremos unas macros que nos van a permitir insertar directamente código en ensamblador en el programa en "C". La sintaxis de la directiva **__asm__** la analizaremos en la sesión PA5.
- Las macros de las que disponemos son:
  - **guarda_registros()**: Almacena en la pila los registros **%ebx**, **%ecx** y **%edx** que se modifican en la función **resta()**.Como la función resta tiene código directamente en ensamblador, el compilador no es capaz de determinar que registros se ven modificados por dicha función, y por lo tanto los tenemos que guardar nosotros antes de invocar a la función.
  - **parametro(p)**: Inserta el parámetro **p** en la pila.
  - **invoca(f,r)**: Invoca a la función **f** (utilizando la instrucción en ensamblador **call**), reestablece el estado de la pila (elimina el espacio ocupado por los parámetros sumándole 8 al puntero de pila) y almacena el valor de retorno en **r**. El valor de retorno en los programas en "C" se suele almacenar en el registro **%eax**.
  - **recupera_registros()**: Recupera los registros almacenados antes de la invocación de la función **resta()**.

- Así pues, el código de la invocación a la función:

```
e= resta(p1,p2);
```

quedaría:

```
guarda_registros();
parametro(p2);
parametro(p1);
invoca(resta, e);
recupera_registros();
```

- Es interesante destacar que los parámetros se insertan en la pila en orden inverso al que aparecen en la declaración de la función `resta()`.
- Compila y ejecuta el programa con los dos argumentos numéricos. ¿Da algún aviso el compilador? ¿Funciona el programa correctamente?

- Si utilizamos la opción `-E` del compilador podremos observar el código generado finalmente en "C".

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

    int e;

    int p1, p2;
    p1= atoi(argv[1]);
    p2= atoi(argv[2]);

    __asm__ __volatile__( "pushl %ebx\n" "pushl %ecx\n" "pushl %edx\n" );
    __asm__ __volatile__( "push %%eax\n" : : "a" (p2) );
    __asm__ __volatile__( "push %%eax\n" : : "a" (p1) );
    __asm__ __volatile__( "call ""resta""\n" "add $0x8, %%esp\n" : "=a" (e) );
    __asm__ __volatile__( "popl %edx\n" "popl %ecx\n" "popl %ebx\n" );

    printf("resta(%d,%d) = %d\n", p1, p2, e);

    return 0;
}
```

  En el código se puede observar que nosotros hemos utilizado la instrucción `push` para almacenar los parámetros en la pila.

- Si le echamos un vistazo al código en ensamblador veremos que varia con respecto al generado por el compilador.

```
8048496:    push   %ebx            ; Almacena algunos registros
8048497:    push   %ecx
8048498:    push   %edx
8048499:    push   %eax            ; Inserta en la pila el parámetro 2
804849a:    mov    %ebx,%eax
804849c:    push   %eax            ; Inserta en la pila el parámetro 1
804849d:    call   80483d0 <resta> ; Invocación de la función resta
80484a2:    add    $0x8,%esp       ; Elimina los parámetros de la pila
80484a5:    pop    %edx            ; Reestablece los registros
80484a6:    pop    %ecx
80484a7:    pop    %ebx
```

- La dirección de retorno debería coincidir con la mostrada al ejecutar el programa `prog2`.

La dirección de retorno

- La instrucción en ensamblador `call` introduce automáticamente la dirección de retorno en la pila, de forma que la instrucción `ret` al final de cada función sepa dónde debe continuar la ejecución del programa.
- Este comportamiento se puede "simular" introduciendo la dirección de retorno en la pila manualmente (con un `push`) y utilizando una instrucción `jmp` para saltar a la dirección de comienzo de la función.

  Cuando se ejecute la instrucción `ret` volverá a la dirección que encuentre en la pila. El comportamiento será idéntico al que tendría se se utilizará la instrucción `call`.

- Vamos a comprobar este comportamiento mediante el uso de la macro **invoca_salto(f,r)**. Esta macro se comprta igual que la macro **invoca()**, pero generá un código en ensamblador que utiliza la instrucción `jmp`.

- Modificad el código de `prog2` para que utilice la macro **invoca_salto()**.
- Si mostramos el código preprocesado se puede observar el uso de la etiqueta `$1f` para determinar la dirección de retorno.

```
...
    __asm__ __volatile__( "pushl %ebx\n" "pushl %ecx\n" "pushl %edx\n" );
    __asm__ __volatile__( "push %%eax\n" : : "a" (p2) );
    __asm__ __volatile__( "push %%eax\n" : : "a" (p1) );
    __asm__ __volatile__( "pushl $1f\n" "jmp ""resta""\n" "1:" "add $0x8, %%esp\n" : "=a" (e) );
    __asm__ __volatile__( "popl %edx\n" "popl %ecx\n" "popl %ebx\n" );

    printf("resta(%d,%d) = %d\n", p1, p2, e);
...
```

En ensamblador quedaría:

```
8048496:   push   %ebx            ; Almacena algunos registros
8048497:   push   %ecx
8048498:   push   %edx
8048499:   push   %eax            ; Inserta en la pila el parámetro 2
804849a:   mov    %ebx,%eax
804849c:   push   %eax            ; Inserta en la pila el parámetro 1
804849d:   push   $0x80484a7      ; Inserta la dirección de retorno
80484a2:   jmp    80483d0 <resta> ; Salta a la dirección de la función
80484a7:   add    $0x8,%esp       ; Elimina los parámetros de la pila
80484aa:   pop    %edx            ; Reestablece los registros
80484ab:   pop    %ecx
80484ac:   pop    %ebx
```

- Si compilamos y ejecutamos la nueva versión del programa, el comportamiento debería ser el mismo.

- Aunque este comportamiento no tiene una utilidad aparente, se puede utilizar para que una función retorne a un punto distinto del programa, en vez de a la instrucción siguiente. Esto se utiliza en el núcleo de Linux para llevar a cabo parte del cambio de contexto.
- Modificando nuestro pragrama `prog2` para comprobar este comportamiento, el progrma quedaía como sigue:

```
...
    guarda_registros();
    parametro(p2);
    parametro(p1);
    salto(resta, salida);

    printf("Este mensaje no se imprime\n");

    destino(salida, e);
    recupera_registros();
    printf("Salida alternativa= resta(%d,%d) = %d\n", p1, p2, e);
...
```

- Si compilamos y ejecutamos la nueva versión: ¿Cuál es el comportamiento? ¿Se ejecuta la función `printf(...)` que hay a continuación de la macro **salto()**?
- El contenido final del programa en "C" una vez preprocesado sería:

```
...
    __asm__ __volatile__( "pushl %ebx\n" "pushl %ecx\n" "pushl %edx\n" );
    __asm__ __volatile__( "push %%eax\n" : : "a" (p2) );
    __asm__ __volatile__( "push %%eax\n" : : "a" (p1) );
    __asm__ __volatile__( "pushl $""salida""\n" "jmp ""resta""\n" );

    printf("Este mensaje no se imprime\n");

    __asm__ __volatile__( "salida"":\n" "add $0x8, %%esp\n" : "=a" (e) );
    __asm__ __volatile__( "popl %edx\n" "popl %ecx\n" "popl %ebx\n" );
    printf("Salida alternativa= resta(%d,%d) = %d\n", p1, p2, e);
...
```

En ensamblador quedaría:

```
80484d2:   push   %ebx             ; Almacena algunos registros
80484d3:   push   %ecx
80484d4:   push   %edx
80484d5:   push   %eax             ; Inserta en la pila el parámetro 2
80484d6:   mov    %esi,%eax
80484d8:   push   %eax             ; Inserta en la pila el parámetro 1
80484d9:   push   $0x80484ef       ; Inserta la dirección de retorno
80484de:   jmp    8048410 <resta>  ; Salta a la dirección de la función
80484e3:   movl   $0x804865a,(%esp) ; Este código no se ejecuta
80484ea:   call   80482f4 <printf>

080484ef <salida>:
80484ef:   add    $0x8,%esp        ; Elimina los parámetros de la pila
80484f2:   pop    %edx             ; Reestablece los registros
80484f3:   pop    %ecx
80484f4:   pop    %ebx
```