Resumen 1P- Arquitectura de Computadoras

Type	Resumen
Created time	@April 27, 2024 3:04 PM
Materials	x86 Assembly Guide.html
Revised	☐

Resumen Primer Parcial Arqui

Clase 1.

Compilación y linkedición en C con gcc

- Preprocesador cpp: realiza la macroexpansión de directivas como #include o #define
- gcc: convierte el .c en Assembler (.S)
- gas: compilador gnu de Assembler genera el código objeto
- ld: linkeditor. Puede generar el ejecutable en distintos formatos.

Generación de un ejecutable

1. La línea de comando (./file)
2. Se llama al sistema operativo
3. **Se hace la lectura de disco**
4. **Se asigna el espacio en memoria al programa**
5. **Se hace un CALL o JMP a la dirección de memoria del programa**
6. Se ejecuta el programa

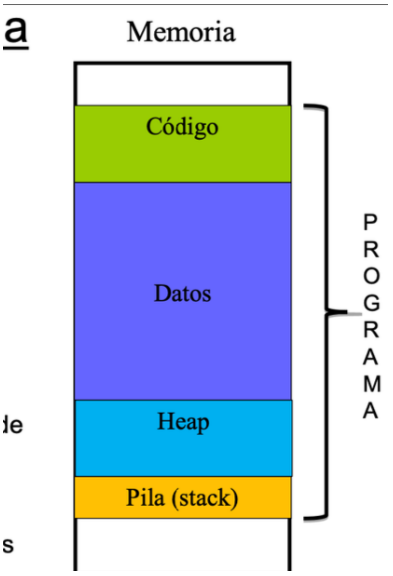
💡 ¿Para que sirve pasarlo a memoria luego de leerlo en el disco?

La copia a memoria sirve porque los loops precisan ser leídos más rápidamente. Es una cuestión de *rapidez*

Almacenamiento de un programa en memoria

¿Qué se almacena en cada segmento?

- **Código**: las instrucciones
- **Datos**: variables estáticas y globales
- **Heap**: memoria dinámica (se reserva y libera en ejecución)
- **Pila o Stack**: Argumentos y variables locales



💡 Los punteros a cada zona de memoria se almacenan en los registros. Ejemplo: esp stack pointer apunta al inicio del stack

Almacenamiento de un programa en disco

→ Depende enteramente del S.O.

Encabezados del archivo	Datos del archivo

Encabezados: Encabezado de segmentos del programa y de secciones del programa (.text, .data, etc.)

Datos: Está el código y los datos

Llamados a funciones

CALL: La instrucción de assembler que permite el llamado a una función

RET: La función debe terminar con la instrucción “return”

→ La dirección de retorno se guarda en la pila → RET hace pop del valor y realiza un JUMP TO a ese valor de memoria

> 💡 ¿Qué pasa si no hago ret? Se seguirá leyendo memoria hasta pisar una zona que *no* es parte de la memoria reservada para la función → `Segmentation Fault`

Llamadas a Sistema Operativo

(System Calls)

¿Qué hacen las sistem calls? Permiten gestionar los recursos de la máquina, son llamados a funciones propias del sistema operativo.

- Es una llamada a ejecución en el *kernel space* (es decir, la memoria asignada al sistema operativo, en contraposición al *user space* que se asigna al resto de programas)
- Existen distintas formas de ejecutar una system call. Nosotros estamos usando la interrupción nro. 80 (no es la forma más eficiente)

Obs! Es el sistema operativo el que maneja todas las acciones.

- Cada syscall tiene un id único (obviamente solo nos interesan los de linux)

Llamado a una sistem call en Assembler:

```
* Put the system call number in the EAX register.
* Store the arguments to the system call in the registers
    EBX, ECX, etc.
* Call the relevant interrupt (80h).
* The result is usually returned in the EAX register.
```

Ejemplo con llamado a WRITE:

```
mov ecx, cadena ; ptr de la cadena
mov edx, len    ; cargo el largo de la cadena

mov ebx, 1  ; FileDescriptor (STDOUT)
```

```
mov eax, 4  ; ID del Syscall WRITE
int 80h
```

Lenguajes Assembler de Intel

En la materia vemos solo la sintaxis de Intel.

La sintaxis AT&T, por ejemplo, es bastante parecida, pero solo nos vamos a concentrar en la de intel pues es el procesador más vendido y no tiene sentido ver todas.

Registros

¿Qué son?

Son parte física del procesador donde se almacena memoria. Se almacena información que al procesador le interesa tener disponible fácilmente.

Registros notables:

- Instruction pointer (IP): EIP en 32b. Puntero a la proxima instrucción a ejecutarse
- Stack Pointer (SP): ESP en 32b. Punteros a la pila para guardar y extraer datos.
- Registros a segmentos: Code segment (CS), Data segment (ES, FS, GS), Stack Segment (SS).

💡 Por compatibilidad hacia atrás los registros de segmentos mantienen su tamaño en arquitecturas de 32 y 64 bits

🔑 **Importante:** Los registros actúan de a pares. Por ejemplo SS (stack segment pointer) indica el puntero a la primera posición del stack, y ESP indica el *offset* o posición relativa respecto a SS. Al menos en la primera parte de la materia ignoramos esto, pues llamar a [ESP] es equivalente a llamar a SS:[ESP] (es equivalente en el assembler)

💡 Por compatibilidad hacia atrás pueden accederse desde la arquitectura de 32 bits a los registros de 16, y 8 sin problema. **Atención**! Sigue siendo el mismo lugar físico.

Arquitectura de 80386

Este es el procesador que veremos.

Características

- **Multitarea**. (Aclaración: el procesador tiene la capacidad de correr "un poquito" de cada tarea; es una velocidad tan alta que a la percepción humana *parece* que ejecuta más de una cosa a la vez)
- **Multiusuario**. Más de un usuario tiene acceso a la CPU
- **Tiempo compartido o time slot**. El S.O. asigna un tiempo para cada tarea
- **Tiempo real**. La conmutación de tareas viene dada por acontecimientos externos
- **Sistema de protección.** Existen mínimo dos niveles de protección: usuario y supervisor.

Clase 2.

Sintaxis de Assembler Intel

¿Qué es una instrucción? Un flujo de bytes que el procesador interpreta y en base al cual realiza una acción.

→ En Intel el tamaño de las instrucciones es variable

Instrucción MOV

```
mov eax, [ebx]        ; Move the 4 bytes in memory at the address contained in EBX into EAX
mov [var], ebx        ; Move the contents of EBX into the 4 bytes at memory address var. (Note, var is a 32-bit constant).
mov eax, [esi-4]      ; Move 4 bytes at memory address ESI + (-4) into EAX
mov [esi+eax], cl     ; Move the contents of CL into the byte at address ESI+EAX
mov edx, [esi+4*ebx]  ; Move the 4 bytes of data at address ESI+4*EBX into EDX
```

Some examples of invalid address calculations include:

```
mov eax, [ebx-ecx]        ; Can only add register values
mov [eax+esi+edi], ebx    ; At most 2 registers in address computation
```

Atención! [bx] significa ir a bx, traer su contenido y usarlo como puntero a memoria.

💡 ASM funciona en base al tamaño del **destino**. Por tanto si pido 2 bytes y los voy a buscar a una dirección que solo contiene 1 byte, entonces me traerá el contenido del espacio de memoria siguiente. Por obvias razones esto es peligroso y puede darme `segmentation fault`

💡 ¿Por qué se mantuvo que cada espacio de memoria sea 1 byte? Por compatibilidad hacia atrás

⚠️ El move no es un move literal sino una ***copia.***

Instrucciones PUSH y POP

push: Me deja el valor indicado en el stack y **decrementa** el stack pointer.

pop*: R*ecupera el último valor de la pila, dejándolo en el lugar que le indico (en un registro, por ejemplo), e **incrementa** el stack pointer

⚠️ Recordar que la pila crece "hacia abajo"

Datos en Assembler

En Assembler no existen los *tipos* de datos sino que existe solo el *tamaño* de los mismos.

`db` → 1 byte

`dw` → 2 bytes

`dd` → 4 bytes

Secciones

¿Qué me indican?

Me indican en qué segmento estoy

→ No me importa el orden en que estén declaradas

.text → sección de código

.bss → "best save space"

.data → cadenas, len, etc.

.rodata → “read only data”, datos que *no* pueden modificarse

Pasaje de Argumentos en Assembler

En Assembler puedo pasarle los argumentos a una función pusheandolos al stack o dejándolos en los registros, es indistinto.

Como no existen convenciones, esto nos proporciona mayor libertad pero es, a su vez, muy desorganizado.

Obs! En C la convención es pasarlo por stack en 32bits

Registro de Flags

También es un elemento físico dentro del procesador, cada flag del registro puede tomar 2 valores (0 o 1), y avisan de un cambio particular luego de que se haya ejecutado una instrucción que cumpla con ese cambio.

Clase 3.

Diferencias entre compiladores

NASM: Precisa que la primera instrucción que recibe se llame *_start*. Si compilo con nasm la primera instrucción que veré en el debugger es la que escribí.

GCC: Precisa que la primera función se llame *main*. La primera instrucción que veré al debuggear será distinta a la primera que escribí pues el GCC agrega validaciones y chequeos propios.

Mezcla de Lenguajes en un Ejecutable

¿Cómo solucionar los retornos entre C y Assembler entre funciones?

Utilizo la pila y los registros.

→ A la hora de mezclar ASM y C adoptaremos las convenciones de C pues ASM no tiene convenciones propias.

> 💡 Las convenciones de C forman parte de su **ABI** (*Application Binary Interface*)

Pasaje de Argumentos en C

→ El pasaje puede hacer por **valor** o por **referencia**

- Arquitectura de 32 bits

Se pasan por PILA.

> 💡 Recordar que siempre deben pasarse de *derecha a izquierda*

- Arquitectura de 64 bits

Se cargan los argumentos en los registros, se llama a la función, y luego los args se copian en el stack y son referenciados desde el mismo.

> ⚠️ ¿Por qué se pasan por registros? Es una cuestión de seguridad: la pila se puede modificar fácilmente y se puede ejecutar código en ella

Los argumentos se pasan en distintos registros dependiendo de su tipo:

Tipo	Dato	Registros usados
INTEGER	char, short, int, long, long long, punteros	rdi, rsi, rdx, r8, r9 (en orden)
SSE	floats y doubles	xmm0 a xmm7 (en orden)
MEMORY	datos mayores a 8 bytes y datos desalineados	Se pasan y devuelven por igual que en 32bits

Luego los retornos se manejan de forma tal que:

Tipo	Registros de Retorno
INTEGER	rax y rdx
SSE	xmm0 y xmm1

Hay registros que pertenecen siempre a la función llamadora, por ende deben mantener su valor al terminar la función:

- rbp
- rsp

- rbx
- r12
- r13
- r15

Manejo de la pila en C

El manejo de la pila es *fundamental* pues para que el ret retorne correctamente la función llamada debe dejar la pila sin modificar antes de terminar.

- Se utiliza el registro EBP para acceder a los parámetros que puede haber en la pila o a las variables locales, para no modificar el registro ESP

> ⚠ De aquí sale el *armado de stackframe* → al empezar una función. me guardo en la pila el valor de *esp* y trabajo exclusivamente con ebp.

Armado de Stack frame

```
push ebp

mov ebp, esp
```

Desarmado de Stack frame

```
mov esp, ebp
pop ebp
```

¿Qué es *frame*? Es el espacio de memoria asignado a una función

Retorno de funciones

Si el valor es menor a 32 bits: se retorna en `EAX`

Si el valor es mayor: parte alta en `EDX` y parte baja en `EAX`

Si es un dato más complejo (p.e. CDTs): se retorna un puntero formado por `EDX:EAX`

Llamada de ASM a C

En el .asm declaro la función como extern. Recordar para compilar con gcc que mi .asm debe tener como primera función a *main*.

Por ejemplo:

```
extern printf
;luego en el código:
call printf
```

Luego en mi .c contendrá la función que declaré.

Llamada de C a ASM

En mi .c declaro la función como externa

Por ejemplo:

```
extern unsigned int siete(void);
```

Luego en otro archivo .asm compilo la función siete.

Inline ASM

Existe la posibilidad de escribir el assembler directamente en el .c, pero es de muy mal estilo por ende no vamos a ahondar en ello.

Clase 4.

Obs! En esta clase la mayoría son ejemplos.

Salidas en ASM

Para obtener el equivalente al código en Assembler debo compilar con la flag `-S` en el gcc.

Optimización

La flag `-o nbr` indica la cantidad nbr de pasadas del compilador.

¿Qué cosas optimiza el compilador?

Reservas de espacio de más y movimientos del stack pointer innecesarios.

Macros de ASM

Las macros `enter` y `leave` sirven para el armado y desarmado de stack

SPP (Stack Smashing Protector)

Es un chequeo que implementa gcc.

¿Qué hace?

Ubica (pushea) un número entre EBP y las variables locales llamado ***CANARY*.**

Utiliza la función `__stack_chk_fail` justo antes de retornar que revisa que el valor pusheado no haya sido modificado.

Si fue modificado termina prematuramente la ejecución del programa.

¿De qué está protegiendo?

Si no valido los datos de entrada de un programa, por ejemplo, puedo estar pisando la dirección de retorno (intencionalmente) con código que me interesa que se ejecute.

Este tipo de vulnerabilidad se denomina *code injection*

Cuestiones Adicionales

📌 **El PID (Process ID) es un número que utiliza Linux para identificar los procesos que están corriendo.**
Si se corre el comando `ps x`, se puede ver una lista de los procesos que están corriendo actualmente que son del usuario.
Obviamente, como ps es un proceso, también aparece en la lista.

📌 **Variables:**

globales → Pueden accederse desde otro archivo, cuando termine la función mantiene su valor

estáticas → preservan su valor al salir del scope, permanece en memoria mientras el programa "llamador" se esté ejecutando

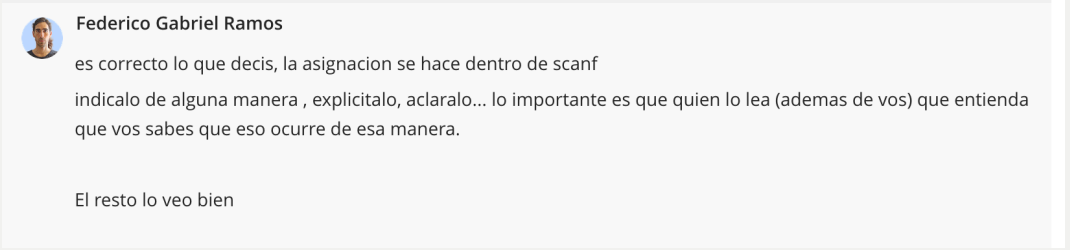
externas → quiere decir que fue definida en otro archivo. No reserva espacio

locales → solo se acceden desde esa función, se guardan en el stack y al terminar la función dejan de existir.

📌 `CPUID` guarda en eax lo que le pidas según lo que había en el registro. Si lo que quiero es el vendor id (fabricante del procesador)

```
MOV EAX, 00H
CPUID
```

📌 **Sobre scanf y funciones similares en seguimiento de stack:**
Cuando una variable resuelve algo (i.e. asigna un valor a una variable por fuera de sí misma) y no tengo la implementación de la función, está bien poner la asignación como paso posterior al retorno de la función siempre y cuando lo aclares que no es realmente así.



📌 `ALIGN 4` Alinea la sección de código que le indico.

`add esp, -16` Alinea a 4 el esp (es una cuestión de eficiencia)

📌 Para el pasaje de nbr_to_string le sumo un ‘0’ al registro:
En cada iteración del ciclo tengo el resto en mi registro (por ejemplo un 3), al añadirle un ‘0’ me queda
`ecx = ‘0’ + 3 = 48 + 3 = 51 = ‘3’` → pues es la representación ascii del 3! Así queda como string finalmente.

ESP	Cantidad de Argumentos
ESP + 4	Path al programa
ESP + 8	dirección del 1er argumento
ESP + 12	dirección del 2do argumento
ESP + 16	dirección del 3er argumento
ESP + (n+2)*4	dirección del n-argumento
	NULL (4 bytes)

📌 Obs! `getpid()` returns the process ID (PID) of the calling process