### kern2-stack

**Explicar: ¿qué significa “estar alineado”?**

“Estar alineado” consiste en una restricción que impone el sistema operativo hacia las direcciones permitidas para los tipos de datos primitivos, requiriéndoles a los mismos que sean un múltiplo de K (típicamente 2, 4 u 8). Estas restricciones simplifican el diseño del hardware, formando una interfaz entre el procesador y la memoria.

**Mostrar la sintaxis de C/GCC para alinear a 32 bits el arreglo kstack anterior.**

La sintaxis para alinear a 32 bits el arreglo sería de la siguiente manera:

unsigned char kstack[8192] \_\_attribute\_\_ ((aligned (4));

**¿A qué valor se está inicializando kstack? ¿Varía entre la versión C y la versión ASM? (Leer la documentación de as sobre la directiva .space.)**

.space únicamente reserva un espacio en memoria pero no indica nada acerca del alineamiento del espacio reservado.

En el código de C el arreglo siempre debería tener una dirección múltiplo de 4.

**Explicar la diferencia entre las directivas .align y .p2align de as, y mostrar cómo alinear el stack del kernel a 4 KiB usando cada una de ellas.**

La diferencia consiste en que .p2align k avanza el location counter a la siguiente dirección más cercana que sea múltiplo de 2^k.

Por el otro lado .align k avanza el location counter a una dirección múltiplo de k.

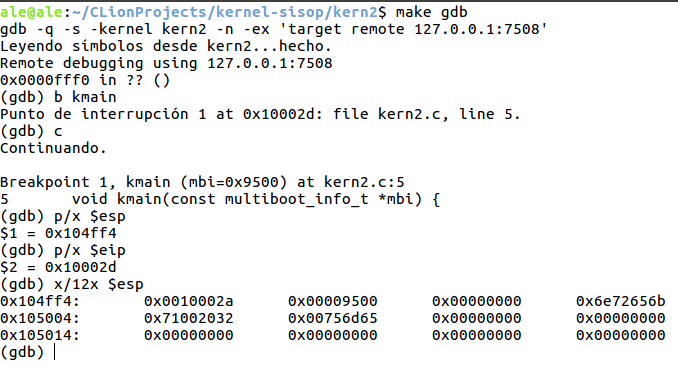
Para alinear el stack a 4KB se puede hacer:

.p2align 12 // 2^12 = 4096 bytes = 4KB

.align 4096 // 4096 bytes = 4KB

La nueva versión del archivo boot.S se detalla en la sección final que muestra el código fuente.

**Finalmente: mostrar en una sesión de GDB los valores de %esp y %eip al entrar en kmain, así como los valores almacenados en el stack en ese momento.**



### kern2-cmdline

**Mostrar cómo implementar la misma concatenación, de manera correcta, usando strncat(3).**

if (mbi->flags) {

char buf[256] = "cmdline: ";

char \*cmdline = (void \*) mbi->cmdline;

strncat(buf, cmdline, sizeof(buf)-strlen(buf)-1);

vga\_write(buf, 9, 0x07);

print\_mbinfo(mbi);

}

De esta manera se evita escribir mas alla del tamaño real de buf (256).

**Explicar cómo se comporta strlcat(3) si, erróneamente, se declarase buf con tamaño 12. ¿Introduce algún error el código?**

En ese caso, lo que se quiera concatenar luego de los 9 caracteres (cmdline: ) serán solo 2 caracteres mas, ya que uno quedará reservado para el caracter '\0' y sin introducir ningun error en el código.

**Compilar el siguiente programa, y explicar por qué se imprimen dos líneas distintas, en lugar de la misma dos veces:**

#include <stdio.h>

static void printf\_sizeof\_buf(char buf[256]) {

printf("sizeof buf = %zu\n", sizeof buf);

}

int main(void) {

char buf[256];

printf("sizeof buf = %zu\n", sizeof buf);

printf\_sizeof\_buf(buf);

}

Porque el sizeof buf de la función *print\_sizeof\_buf* lo que hace es devolver el tamaño de un char\*, que es 8.

### kern2-meminfo

El código realizado para esta sección se encuentra en la próxima sección.

### Código Fuente

**boot.S**

#include "multiboot.h"

#define KSTACK\_SIZE 8192

.align 4

multiboot:

.long MULTIBOOT\_HEADER\_MAGIC

.long 0

.long -(MULTIBOOT\_HEADER\_MAGIC)

.globl \_start

\_start:

movl $0, %ebp

movl $kstack\_bottom, %esp

push %ebp

movl $0, %ecx

cmp $MULTIBOOT\_BOOTLOADER\_MAGIC, %eax

cmove %ebx, %ecx

push %ecx

call kmain

halt:

hlt

jmp halt

.data

.p2align 12

kstack:

.space KSTACK\_SIZE

kstack\_bottom:

**decls.h**

#ifndef KERN2\_DECL\_H

#define KERN2\_DECL\_H

#include <stdint.h>

#include <stdbool.h>

#include <stddef.h>

#include <stdint.h>

struct multiboot\_info;

// mbinfo.c (ejercicio opcional kern2-meminfo)

void print\_mbinfo(const struct multiboot\_info \*mbi);

bool fmt\_int(uint64\_t val, char \*s, size\_t bufsize);

// stacks.S

void two\_stacks(void);

// kern2.c

void two\_stacks\_c(void);

// tasks.S

void task\_exec(uintptr\_t entry, uintptr\_t stack);

void task\_swap(uintptr\_t \*esp);

// contador.c

void contador\_run(void);

// interrupts.c

void idt\_init(void);

void idt\_install(uint8\_t code, void (\*handler)(void));

void irq\_init(void);

// idt\_entry.S

void divzero(void);

void breakpoint(void);

void ack\_irq(void);

void timer\_asm(void);

void keyboard\_asm(void);

// handlers.c

void timer(void);

void keyboard(void);

// funcs.S

\_\_attribute\_\_((regparm(3))) void vga\_write2(const char \*s,

int8\_t linea,

uint8\_t color);

// write.c

void vga\_write(const char \*s, int8\_t linea, uint8\_t color);

\_\_attribute\_\_((regparm(2))) void vga\_write\_cyan(const char \*s, int8\_t linea);

#endif

**kern2.c**

#include "decls.h"

#include "multiboot.h"

#include "lib/string.h"

void kmain(const multiboot\_info\_t \*mbi) {

vga\_write("kern2 loading.............", 8, 0x70);

if (mbi->flags) {

char buf[256] = "cmdline: ";

char \*cmdline = (void \*) mbi->cmdline;

strlcat(buf, cmdline, sizeof(buf));

vga\_write(buf, 9, 0x07);

print\_mbinfo(mbi);

}

asm("hlt");

}

**mbinfo.c**

#include "decls.h"

#include "lib/string.h"

#include "multiboot.h"

#define KB\_TO\_MB\_SHIFT 10 // 1KB\*2^10->1MB

void print\_mbinfo(const struct multiboot\_info \*mbi){

char mem[256] = "Physical memory: ";

char tmp[64] = {0};

uint32\_t total\_size = mbi->mem\_upper - mbi->mem\_lower;

if (fmt\_int(total\_size>>KB\_TO\_MB\_SHIFT, tmp, sizeof tmp)) {

strlcat(mem, tmp, sizeof mem);

strlcat(mem, "MiB total", sizeof mem);

}

memset(tmp,0, sizeof(tmp));

if (fmt\_int(mbi->mem\_lower, tmp, sizeof tmp)) {

strlcat(mem, " (", sizeof mem);

strlcat(mem, tmp, sizeof mem);

strlcat(mem, " KiB base", sizeof mem);

}

memset(tmp,0, sizeof(tmp));

if (fmt\_int(mbi->mem\_upper, tmp, sizeof tmp)) {

strlcat(mem, ", ", sizeof mem);

strlcat(mem, tmp, sizeof mem);

strlcat(mem, " KiB extended) ", sizeof mem);

}

vga\_write(mem, 10, 0x07);

}

**write.c**

#include "multiboot.h"

#include "decls.h"

#define VGABUF ((volatile char \*) 0xB8000)

#define ROWS 25 // numero de filas de la pantalla

#define COLUMNS 80 // numero de columnas de la pantalla

static size\_t int\_width(uint64\_t val) {

size\_t width = 0;

while (val>0){

val/=10;

width++;

}

return width;

}

// Escribe en ‘s’ el valor de ‘val’ en base 10 si su anchura

// es menor que ‘bufsize’. En ese caso devuelve true, caso de

// no haber espacio suficiente no hace nada y devuelve false.

bool fmt\_int(uint64\_t val, char \*s, size\_t bufsize) {

size\_t l = int\_width(val);

if (l >= bufsize) // Pregunta: ¿por qué no "l > bufsize"?

// Respuesta: para agregar el \0

return false;

for (size\_t i = l; i > 0; i--) {

char ascii\_digit = '0'+val %10;

s[i-1]= ascii\_digit;

val/=10;

}

s[l]='\0';

return true;

}

void vga\_write(const char \*s, int8\_t linea, uint8\_t color) {

if (linea < 0) {

linea = ROWS + linea;

}

volatile char\* buff = VGABUF + linea \* COLUMNS \* 2;

while (\*s != '\0') {

\*buff++ = \*s++;

\*buff++ = color;

}

}

**Makefile**

CFLAGS := -g -std=c99 -Wall -Wextra -Wpedantic

CFLAGS += -m32 -O1 -ffreestanding -fasm # para el kernel

SRCS := $(wildcard \*.c)

SRCS += $(wildcard lib/\*.c)

OBJS := $(patsubst %.c,%.o,$(SRCS))

IP := 127.0.0.1:7508

QEMU := qemu-system-i386 -serial mon:stdio

KERN := kern2

BOOT := -kernel $(KERN) $(QEMU\_EXTRA)

LIBGCC := $(shell $(CC) $(CFLAGS) -print-libgcc-file-name)

qemu: $(KERN)

$(QEMU) $(BOOT)

qemu-gdb: $(KERN)

$(QEMU) -kernel kern2 -S -gdb tcp:127.0.0.1:7508 $(BOOT)

gdb:

gdb -q -s -kernel kern2 -n -ex 'target remote $(IP)'

.PHONY: qemu qemu-gdb gdb

kern2: boot.o $(OBJS) # en el enunciado no esta el boot.o , sacar?

ld -m elf\_i386 -Ttext 0x100000 $^ $(LIBGCC) -o $@

# Verificar imagen Multiboot v1.

grub-file --is-x86-multiboot $@

%.o: %.S

$(CC) $(CFLAGS) -c $<

clean:

rm -f kern2 \*.o core

.PHONY: clean