Análisis y Diseño de Programas

Examen: En esta prueba de examinación se pide analizar ciertos fragmentos de código en lenguaje C para poder responder los ejercicios señalados a través de este documento. El código fuente que se pide analizar corresponde a un kernel de sistema operativo conocido como JOS.
La estructura `struct Elf` está declarada en `inc/elf.h` como

```
#define ELF_MAGIC 0x464C457FU /* "\x7FELF" in little endian */

struct Elf {
  uint32_t e_magic; // must equal ELF_MAGIC
  uint8_t e_elf[12];
  uint16_t e_type;
  uint16_t e_machine;
  uint32_t e_version;
  uint32_t e_entry;
  uint32_t e_phoff;
  uint32_t e_shoff;
  uint32_t e_flags;
  uint16_t e_ehsize;
  uint16_t e_phentsize;
  uint16_t e_phnum;
  uint16_t e_shentsize;
  uint16_t e_shnum;
  uint16_t e_shstrndx;
};
```

En el archivo `boot/main.c` podemos ver la función `bootmain()`

```
#define SECTSIZE 512
#define ELFHDR ((struct Elf *) 0x10000) // scratch space

void readsect(void*, uint32_t);
void readseg(uint32_t, uint32_t, uint32_t);

struct multiboot_info *mbi;

void
bootmain(void)
{
  struct Proghdr *ph, *eph;

  // read 1st page off disk
  readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);

  // is this a valid ELF?
  if (ELFHDR->e_magic != ELF_MAGIC)
```

```
    goto bad;

  // load each program segment (ignores ph flags)
  ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
  eph = ph + ELFHDR->e_phnum;
  for (; ph < eph; ph++)
    // p_pa is the load address of this segment (as well
    // as the physical address)
    readseg(ph->p_pa, ph->p_memsz, ph->p_offset);

  // set up multiboot: fill struct multiboot_info
  // boot.S sets mbi to the end of mmap entries
  mbi->flags = MULTIBOOT_INFO_MEM_MAP;
  mbi->mmap_length = (uint32_t)mbi & (4096 - 1); // 4K aligned
  mbi->mmap_addr = (uint32_t)mbi - mbi->mmap_length;

  // 3.2 Machine state:
  // * eax: contain MULTIBOOT_BOOTLOADER_MAGIC
  // * ebx: contain the 32-bit physical address
  asm ("movl %0, %%eax;"
       "movl %1, %%ebx;"
       : : "r"(MULTIBOOT_BOOTLOADER_MAGIC), "r"(mbi)
       : "eax", "ebx");

  // call the entry point from the ELF header
  // note: does not return!
  ((void (*)(void)) (ELFHDR->e_entry))();

bad:
  outw(0x8A00, 0x8A00);
  outw(0x8A00, 0x8E00);
  while (1)
    /* do nothing */;
}
```

En esta vemos que el primer llamado a función es a

```
readseg(uint32_t pa, uint32_t count, uint32_t offset)
```

cuyo código se incluye aquí

```
// Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
// Might copy more than asked
void
readseg(uint32_t pa, uint32_t count, uint32_t offset)
{
  uint32_t end_pa;
```

```
  end_pa = pa + count;

  // round down to sector boundary
  pa &= ~(SECTSIZE - 1);

  // translate from bytes to sectors, and kernel starts at sector 1
  offset = (offset / SECTSIZE) + 1;

  // If this is too slow, we could read lots of sectors at a time.
  // We'd write more to memory than asked, but it doesn't matter --
  // we load in increasing order.
  while (pa < end_pa) {
    // Since we haven't enabled paging yet and we're using
    // an identity segment mapping (see boot.S), we can
    // use physical addresses directly.  This won't be the
    // case once JOS enables the MMU.
    readsect((uint8_t*) pa, offset);
    pa += SECTSIZE;
    offset++;
  }
}
```

Ejercicio 1
En el llamado a esta función, determine el valor numérico de la variable `end_pa` después de la ejecución de la sentencia

```
end_pa = pa + count;
```

Realice e incluya explícitamente los cálculos para llegar a su respuesta.
Ejercicio 2
En el llamado a esta función, determine el valor numérico de la variable `pa` después de la ejecución de la sentencia

```
pa &= ~(SECTSIZE - 1);
```

Realice e incluya explícitamente los cálculos para llegar a su respuesta.
Ejercicio 3
En el llamado a esta función, determine el valor numérico de la variable `offset` después de la ejecución de la sentencia

```
offset = (offset / SECTSIZE) + 1;
```

Realice e incluya explícitamente los cálculos para llegar a su respuesta.
Ejercicio 4
Determine cuántas veces se ejecutará el ciclo `while` en este llamado a la función

```
readseg(uint32_t pa, uint32_t count, uint32_t offset)
```

Realice e incluya explícitamente los cálculos para llegar a su respuesta.

En el archivo `inc/elf.h` está la declaración de `struct Proghdr`.

```
struct Proghdr {
  uint32_t p_type;
  uint32_t p_offset;
  uint32_t p_va;
  uint32_t p_pa;
  uint32_t p_filesz;
  uint32_t p_memsz;
  uint32_t p_flags;
  uint32_t p_align;
};
```

De regreso a boot/main.c el campo `ph->p_pa` de cada header del programa contiene la dirección física de destino del segmento (en este caso es realmente una dirección física, aunque la especificación ELF es vaga respecto al significado real de este campo).

El BIOS carga el sector boot en memoria comenzando en la dirección 0x7c00, así esta es la dirección de carga del sector boot (LMA). Esta también es la dirección desde donde se ejecuta el sector boot, así que es también la dirección de enlazado (VMA). Establecemos la dirección de enlazado pasando el parámetro -Ttext 0x7c00 al enlazador en boot/Makefrag, así que el linker producirá la dirección de memoria correcta en el código generado.

Por ahora, solo se mapearán los primeros 4MB de memoria física, lo cual será suficiente para ponernos arriba y corriendo. hacemos esto usando el directorio de página y tabla de página estáticamente inicializados, escritos a mano en `kern/entrypgdir.c`. Por ahora no es necesario entender los detalles de como trabaja, solo el efecto que consigue. (Se incluye aquí el archivo `kern/entrypgdir.c`)

```
#include <inc/mmu.h>
#include <inc/memlayout.h>

pte_t entry_pgtable[NPTENTRIES];
```

```
// The entry.S page directory maps the first 4MB of physical memory
// starting at virtual address KERNBASE (that is, it maps virtual
// addresses [KERNBASE, KERNBASE+4MB) to physical addresses [0, 4MB)).
// We choose 4MB because that's how much we can map with one page
// table and it's enough to get us through early boot.  We also map
// virtual addresses [0, 4MB) to physical addresses [0, 4MB); this
// region is critical for a few instructions in entry.S and then we
// never use it again.
//
// Page directories/tables must start on a page boundary, hence the
// "aligned" attribute.
__attribute__((aligned(PGSIZE)))
pde_t entry_pgdir[NPDENTRIES] = {
  // Map VA's [0, 4MB) to PA's [0, 4MB)
  [0]
      = 0x000000 | PTE_P | PTE_PS,
  // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
  [KERNBASE>>PDXSHIFT]
      = 0x000000 | PTE_P | PTE_W | PTE_PS
};
```

- Hasta ahora `kern/entry.S` establece la bandera `CR0_PG`, las referencias de memoria son tratadas como direcciones físicas (estrictamente hablando, son direcciones lineales, pero en `boot/boot.S` se establece un mapeo identidad de las direcciones lineales a las direcciones físicas y en este kernel nunca cambiaremos eso).

- Una vez que `CR0_PG` es establecido las referencias a memoria son direcciones virtuales que se traducen por medio del hardware de memoria virtual a direcciones físicas. `entry_pgdir` traduce direcciones virtuales en el rango de `0xf0000000` a `0xf0400000` a direcciones físicas en el rango de `0x00000000` a `0x00400000`, así como direcciones virtuales en el rango de `0x00000000` a `0x00400000` a direcciones físicas en el rango de `0x00000000` a `0x00400000`. Cualquier dirección virtual que no esté en uno de esos dos rangos causará una excepción de hardware la cual, dado que aun no se ha establecido el manejo de interrupciones, causará que la computadora se reinicie una y otra vez indefinidamente.

Impresión con formato a la consola

La mayoría de la gente toma funciones como `printf()` como algo cuya existencia está garantizada, algunas veces incluso piensan que funciones como esta son "primitivas" del lenguaje C. Pero en un kernel de sistema operativo, debemos implementar toda la I/O nosotros mismos.

Lea a través de los archivos `kern/printf.c`, `lib/printfmt.c`, y `kern/console.c`, y asegurese de entender cómo se relacionan.

Ejercicio 5

Se ha omitido un pequeño fragmento de código — el código necesario para

imprimir números octales usando patrones de la forma `"%o"`. Encuentre dónde falta ese fragmento de código y escriba el código necesario para que un número se imprima en octal.

Al ejecutar `make grade` el código debe pasar la prueba `printf`.