Trabajo Práctico Especial - Informe

Arquitecturas de Computadoras

12 de junio de 2012

Autores:

Pomar, Federico (Legajo Nº 50409)
Bond, Federico (Legajo Nº 52247)

Docentes:

Santiago Vallés
Román Cunci
Augusto Nizzo Mc Intosh

1. Introducción

El objetivo del Trabajo Práctico Especial consistió en la implementación de un sistema booteable que trabaje en modo protegido de memoria de procesadores Intel. El sistema implementado provee al usuario de una shell que permita al usuario ejecutar un conjunto reducido de comandos.

El informe está dividido en dos grandes áreas. "Espacio de usuario" "Espacio de kernel". Si bien las diferencias entre estos dos espacios dentro del proyecto no es absolutamente clara, la división ayuda a entender la interacción entre los distintos componentes.

2. Espacio de Kernel

El kernel busca estar lo más aislado posible de los programas de usuario. Para esto se realizó toda la interacción entre ellos a traves de system calls. Existen algunas excepciones a esta regla pero no son graves. Una de ellas es la librería estándar que actualmente es utilizada simultáneamente por ambos.

La idea detrás del diseño implementado fue la de observar al kernel como un gestor de hardware. Para esto se dividió su función en módulos básicos que proveen distintas funcionalidades. Entre ellos, los más fundamentales son:

- El gestor de memoria (`mm.c`)
- El driver de video en modo texto (`vgatext.c`)
- El driver de teclado (`kbd.c`)
- El driver de puerto serie con su gestor (`serial.c` y `serialman.c`)
- El servicio que provee terminales (`tty.c` y `ttyman.c`)
- El servicio que provee el tiempo (`ktime.c`)

Estos módulos intentan ser independientes uno de otro y finalmente se encuentran vinculados por el módulo principal del kernel (kernel.c) que gestiona el arranque, inicializa los dispositivos y luego delega la ejecución a lo que sería, dentro de un esquema verdaderamente dividido, la aplicación de usuario.

En las secciones subsiguientes se dan detalles sobre el funcionamiento de cada uno de los módulos.

2.1. Gestor de memoria

El gestor de memoria brinda el servicio de alocación dinámica al resto del kernel. Consiste básicamente de dos funciones (`mm_malloc` y `mm_free`) que como su nombre lo indica, en cada caso, alocan y liberan memoria según corresponda.

El método utilizado para gestionar el espacio de direcciones es extremadamente minimalista. Se mantiene la posición de memoria a devolver en un puntero que, a cada pedido, se incrementa en un número igual a la cantidad de bytes requerida (WaterMark allocation). No existen tablas que mantengan información sobre el estado de alocación. Este puntero se incrementa infinitamente, desperdiciando, si se quiere, las zonas de memoria que podrían ser liberadas y reutilizadas. Se prefirió este esquema por sobre uno más complejo dado que la cantidad de espacio requerido por el proyecto es tan pequeña que con el tamaño de las memorias de la actualidad sobra para brindar un funcionamiento más que bueno.

2.2. Driver de video (Modo Texto)

El driver de video en modo texto provee funciones básicas para escribir y dibujar en pantalla. Abstrae a los demás módulos del espacio de memoria de video brindando direccionamiento tanto lineal como bidimensional (xy). También provee funciones para manejar el cursor de video, lo que es una tarea engorrosa, si se intentara realizar directamente. En vgatext.h pueden verse todos los modos de escritura en video diferentes que ofrece.

2.3. Driver de teclado

El driver de teclado provee fundamentalmente dos funcionalidades importantes para la interfaz con el usuario. Éstas son las de informar a su usuario de la ocurrencia de un cambio de estado del teclado, mediante un callback que es llamado por una interrupción. Así como también un mapa de teclas configurable desde el exterior del módulo. Al llamar al callback, el teclado informa tanto de la tecla que ha cambiado (A través de su scancode), como de su conversión a el carácter ascii correspondiente, mediante un mapa seleccionable. Actualmente el driver está provisto de dos layouts de teclas diferentes, “Español” e “Inglés”.

Es importante notar lo siguiente:

Luego de cada interrupción, debe leerse el scancode generado, o de lo contrario el controlador de teclado no seguirá interrumpiendo al microprocesador. Luego de obtener un cambio de estado, se informa al suscriptor a

través de un callback que debe encargarse de encolar la información según sea necesario.

Driver de puerto serie

El driver de puerto serie está modularizado por puerto (serial.c). Luego, cada puerto está controlado por un gestor (serialman.c) que maneja los puertos disponibles y realiza la interfaz entre éstos y el resto del kernel. Con respecto a cada puerto, tuvieron que ser tomados ciertos recaudos, devinendo en el siguiente procedimiento para manejarlo.

Cuando se inicia el kernel se configura cada uno de los puertos que estén configurados en el gestor de esta manera:

Se configura la velocidad en 9600 baudios, paridad par, 1 stop bit y 8 bits de datos. Se configuran las interrupciones para ocurrir en el cambio de cualquiera de los registros de estado del puerto, así como la llegada de datos o disponibilidad de buffer de salida. Se activan por default las lineas de `DTR` (Data terminal ready) y `RTS` (Request to send). Se activa el uso del buffer FIFO y se configura el puerto para que la interrupción ocurra al haber un solo byte en él. Se leen los registros de estado de modem, linea e interrupción para responder a cualquier interrupción que hubiera ocurrido antes de estar configurado el puerto.

Al ocurrir una interrupción generada por el puerto, se leen todos los registros de estado y se verifica tanto la llegada de nueva información como la disponibilidad del buffer de salida, enviándose o recibiéndose los datos de las colas de salida y entrada (en memoria RAM) de cada puerto, según corresponda.

2.4. Servicio de tiempo

Para proveer de fecha y hora al sistema, el servicio de tiempo (`ktime.c`) provee toda la interacción con el RTC (Real Time Clock), devolviendo un timestamp en el formato estándar de POSIX. Para esto simplemente se leen los registros de estado del RTC, sin importar si estos están siendo actualizados o no. Dado que es muy improbable que ocurra un cambio de estado durante un pedido, no se realiza esta comprobación. En un sistema de producción que requiriera solidez, sería necesario preguntar al RTC si éste está siendo actualizado, y sólo realizar el pedido una vez que deje de estarlo.

2.5. Terminal (`ttyman.c` y `tty.c`)

La terminal (`tty`) es una abstracción que se encuentra entre los drivers de entrada/salida (`vgatext.c` y `kbd.c`) y la shell. Permite el ingreso y salida de manera independiente del programa que esté ejecutando como shell, asi como el manejo de buffers e impresión a pantalla.

3. Espacio de usuario

Se implementaron las librerías `stdio.h`, `stdlib.h`, `string.h`, `ctype.h` y `time.h` lo más parecido posible a los estándares POSIX. Varias funciones fueron omitidas por ser consideradas de poca utilidad para el trabajo actual, pero también se incluyeron funciones con útiles que no necesariamente se usaron en otros lugares.

3.1. Librerías de sistema

3.1.1. stdio.h

La implementación realizada de esta librería contiene las siguientes funciones. Las únicas funciones que no bloquean a la espera de datos son read y write, con lo cual se uso la instrucción hlt del procesador para bloquear todas las demás. Esto nos permitió tener la flexibilidad de hacer entrada de datos no bloqueando, llamando directamente a las primitivas.

```
int     putc(int ch, FILE *stream);
int     getc(FILE *stream);

int     putchar(int ch);
int     getchar(void);
int     ungetc
void    ungetc(int ch);

int     puts(const char *str);

char   *gets(char *s, int size);
char   *fgets(char *s, int size, FILE *stream);

int     printf(const char *fmt, ...);
int     sprintf(char *s, const char *fmt, ...);
int     fprintf(FILE *f, const char *fmt, ...);
```

```
int     vsprintf(char *str, const char *fmt, va_list ap);
int     vfprintf(FILE *f, const char *fmt, va_list ap);

int     scanf(const char *fmt, ...);
int     sscanf(char *str, const char *fmt, ...);

ssize_t write(int fd, const void *buf, size_t count);
ssize_t read(int fd, void *buf, size_t count);
```

3.1.2. stdlib.h

```
int     rand(void);
void    srand(unsigned int seed);

int     atof(const char *nptr);
int     atoi(const char *nptr);
long    atol(const char *nptr);

char     *itoa(int value, char *str, int base);
char     *utoa(unsigned int value, char *str, int base);

int     abs(int x);
long    labs(long x);
```

Las semilla de srand se define inicialmente de manera estática, pero se puede usar la función time para obtener una semilla aleatoria con la hora de sistema.

3.1.3. ctype.h

Las funciones de ctype suelen implementarse como macros en varias librerías estándar. En nuestro caso por simplicidad se las implementó como funciones, pero se las podría optimizar de mejor manera en caso de ser necesario. Otra posibilidad es tomar ventaja de las directivas para funciones inline del compilador, que no son estándar.

```
int isdigit(int c);
int isupper(int c);
int islower(int c);
int toupper(int c);
int tolower(int c);
```

```
int isalpha(int c);
int isalnum(int c);
int iscntrl(int c);
int isspace(int c);
int isprint(int c);
int isxdigit(int c);
int isvowel(int c);
```

3.1.4. string.h

Las implementaciones de string.h responden en general al estándar POSIX y funcionan tal como uno esperaría que funcionen en cualquier sistema Unix. La única funcion no estándar es trim, que permite eliminar espacios en blanco al principio y al final de una cadena de caracteres. Tener en cuenta que la función cambia la cadena, no devuelve una nueva.

```
char   *strcpy(char *dest, const char *src);
char   *strncpy(char *dest, const char *src, size_t n);

char   *strcat(char *dest, const char *src);

size_t  strlen(const char *s);

int     strcmp(const char *s1, const char *s2);
int     strncmp(const char *s1, const char *s2, size_t n);

char   *strrev(char *str);

char   *strtoupper(char *str);
char   *strtolower(char *str);

char   *strchr(const char *s, int c);
char   *strrchr(const char *s, int c);

void   *memcpy(void *dest, const void *src, size_t n);
int     memcmp(const void *dest, const void *src, size_t n);
void   *memset(void *s, int c, size_t n);

void    trim(char *str);
```

3.1.5. time.h

Las función time de esta librería lee los registros del Real Time Clock (RTC) para obtener la fecha y hora actual. Por cuestiones de tiempo no se pudo probar en detalle que la hora se devuelva como un timestamp válido, pero por el momento se puede usar sin problemas como semilla para las funciones que generan números aleatorios.

```
time_t time(time_t *time);
```

3.1.6. stddef.h

Además de las librerías mencionadas previamente, las siguientes definiciones en stddef.h resultaron útiles:

```
typedef unsigned int size_t;
typedef int ssize_t;

typedef enum { false = 0, true } bool;
```

3.2. Shell o Intérprete de comandos

Como consecuencia de la abstracción creada con la terminal (tty), implementar el shell o intérprete de comandos resultó muy fácil. Uno de los puntos a destacar de la implementación es que tratamos de que los programas que se ejecuten por encima de la shell (y más adelante los que lo hagan como binarios independientes) tengan acceso a un entorno POSIX que facillite la portabilidad. Por esta razón, todos los programas son ejecutados con los parámetros argc y argv característicos de este entorno. La shell se hace cargo de separar la cadena de argumentos y llamar al programa con los valores correctos de estas variables.

3.3. Programas de demostración

Los siguientes programas permiten demostrar la funcionalidad del sistema implementado. Pueden ejecutarse invocando el nombre como primer argumento en el shell.

3.3.1. laws

Este es el programa más sencillo desde un punto de vista técnico. Demuestra el uso más sencillo de la función 'printf'.

3.3.2. help

El programa help imprime los posibles programas a ejecutar, junto con sus respectivas opciones o argumentos de línea de comandos. Es similar al anterior desde el punto de vista técnico.

3.3.3. fortune

Este programa construye sobre una base similar al anterior. Cuenta con un vector de cadenas de caracteres y elige uno al azar (mediante la función rand) para mostrarlo en pantalla.

3.3.4. echo

Este programa demuestra un uso más avanzado del los argumentos dados. Además de la opción de imprimir la versión del programa, acepta como modificador la opción -n, que evita la impresión del caracter de nueva línea final.

3.3.5. cowsay

Este programa se escribió desde un punto de vista más recreacional, para mostrar la capacidad de impresión en pantalla de una manera más visual. No introduce ningún concepto que no haya sido utilizado en los programas mencionados previamente.

3.3.6. chat

Este es el programa más complejo desde un punto de vista técnico, dado que requirió de una fuerte interacción con el hardware para la comunicación en serie. Gracias al buen nivel de abstracción conseguido, gran parte de esta complejidad está oculta. Como se mencionó previamente, las primitivas read y write son las únicas de la librería de stdio que no bloquean a la espera de datos. Este fue un requisito necesario para mantener sencilla la implementación del programa. Lo que se hace en un ciclo principal es leer tanto del puerto serie como de la entrada estándar (stdin) y como no bloquea en ninguna de las llamadas, podemos hacer este bloqueo manualmente una vez concluidas. Cualquier interrupción (en este caso del teclado o del puerto serie) destraba el ciclo y hace que vuelva a leer de ambas entradas. Esto es equivalente a un select en Unix y permite leer los datos en el momento que llegan sin tener que hacer busy waiting.

4. Conclusiones

El Trabajo Práctico Especial permitió reforzar los conceptos aprendidos durante la materia y ganar una visión global de los componentes involucrados en la creación de un sistema booteable bajo la arquitectura Intel.