INSTITUTO TECNOLÓGICO DE BUENOS AIRES

72.08 - Arquitectura de Computadoras

Primer Cuatrimestre 2022

TRABAJO PRÁCTICO ESPECIAL

CFL Kernel

Bautista Canevaro	**62179**
Alejo Flores Lucey	**62622**
Nehuén Gabriel Llanos	**62511**

Introducción:

Este trabajo práctico especial consiste en realizar un kernel booteable, utilizando como base Pure 64, que administre los recursos de hardware de la computadora y permita correr programas de usuario. El kernel debe proveer una API para que dichos programas puedan utilizar los recursos de hardware.

En este informe se desarrollará el proceso de implementación del trabajo práctico. Se ahondará en la división de tareas del grupo, en las decisiones de diseño y en las dificultades encontradas y sorteadas.

Estructura del Kernel:

El diseño del trabajo práctico especial consta de dos secciones, el Kernel Space y el User Space. La razón de esta separación se encuentra en el hecho de que un usuario no debería poder editar secciones sensibles en cuanto al funcionamiento del sistema operativo y la comunicación con el hardware de la computadora. Sin embargo, para que el usuario pueda hacer uso de los periféricos, como la pantalla, el teclado, entre otros, el sistema operativo le ofrece las system calls, que son funciones para realizar ciertas acciones con un periférico desde la sección de código del usuario.

Se tomó la decisión de dividir los servicios en User Space y Kernel Space de la siguiente manera:

- Kernel Space:
    - Rutina de inicialización de la *Interrupt Descriptor Table* (IDT).
    - Rutinas de atención a las excepciones e interrupciones.
    - Implementación de las system calls.
    - Implementación del driver de pantalla.
    - Implementación del driver de teclado.
    - Implementación del driver de Timer Tick.
    - Implementación del *scheduler*.
- User Space:
    - Implementación de la terminal, donde el usuario ingresa los comandos.
    - Implementación de los programas indicados en la consigna.
    - Implementación de la librería de funciones útiles.

Kernel Space:

Inicialización de la Interrupt Descriptor Table:

En esta sección se cargan las entradas de la tabla IDT mediante la función `load_idt()` y setup_IDT_entry(), proveídas por la cátedra. Ambas utilizan una estructura que simula una entrada de la tabla. Lo único que se modificó fue la máscara de los PIC (Programmable Interrupt Controller) que permite activar o desactivar las interrupciones de Hardware.

Excepciones e Interrupciones:

Para el manejo de las excepciones e interrupciones se debió modificar las instrucciones a ejecutar cuando se reciben alguna de estas. En particular, se guarda el valor de todos los registros en una zona segura de memoria con el fin de restaurar el contexto más adelante. Además se agregaron funciones que permiten acceder a dichas zonas de memoria desde otras rutinas del Kernel Space.

System Calls:

Como se dijo previamente, el sistema operativo ofrece servicios que pueden ser utilizados por el User Space y para poder hacer uso de los mismos, se provee una colección de funciones. Estas

últimas se comunican directamente con los drivers de los periféricos y realizan la acción pedida. Véase el manual de usuario para obtener una lista de las *system calls* implementadas.

Driver de pantalla:

Se proveen funciones para escribir a la pantalla del kernel, haciendo uso del modo texto. Se tienen un set diferente de funciones para manejar la pantalla cuando esta se utiliza completa y cuando esta se utiliza dividida. Para poder llevar a cabo esto, se implementó el uso de *File Descriptors*. De esta manera, se le indica a la función en qué "pantalla" se desea escribir. Asimismo, se diseñaron métodos para imprimir a pantalla con diferentes colores de relleno y fondo.

Driver de teclado:

La función del driver de teclado es guardar el caracter ASCII de todas las teclas que se presionan. Dicho esto, se implementó un driver que dependiendo el *Scan Code* de la tecla presionada, se guarda en un buffer el caracter ASCII asociado a esa tecla[1]. Se agregaron las funcionalidades especiales de reconocimiento de la tecla SHIFT y CAPS LOCK, para guardar el caracter acorde. Asimismo, se establecieron otras teclas especiales para el funcionamiento de los programas (véase el manual de usuario para más información).

Driver de Timer Tick:

A las funciones proveídas por la cátedra, como el contador de ticks y los segundos transcurridos, se agregaron las funcionalidades del *Scheduler* (véase más adelante) y el parpadeo del cursor, cuando se está ejecutando la terminal.

Scheduler:

Para cumplir con lo dicho en la consigna y poder correr dos procesos en "simultáneo", se decidió implementar un Scheduler siguiendo el algoritmo de scheduling Round-Robin. Cabe destacar que se tomó esa idea pero la implementación realizada es simplificada y acotada. El marco teórico utilizado es la explicación dada en la clase teórica de conmutación de tareas; se debió complementar dicho marco teórico con búsquedas en la web y el libro "Modern Operating Systems"[2].

El scheduler nos permite alternar entre los procesos que deberían correrse de manera ordenada. Se realiza un intercambio de contexto (*context switching*) donde, primero se guarda el contexto actual almacenando todos los registros en una estructura y el stack, y luego se restaura el contexto del siguiente proceso a correr. Aquí es donde se utilizan las funciones para acceder a las zonas de memoria donde se guardan los registros cuando ocurrió una interrupción.

Cuando se inicia un proceso, se debe generar un *stack frame* especial para el proceso. Dicho *stack frame* coincide con el que genera el procesador cuando ocurre una interrupción, pues se utilizará la instrucción `iretq` que desarmará dicho *stack frame*. Antes de armar lo mencionado previamente se *pushea* al stack la dirección a una función que manejará la finalización del proceso.

Cada proceso tiene un estado asociado, entre ellos encontramos a ACTIVE (activo), PAUSED (pausado) y FINISHED (finalizado). Cabe recalcar que cuando un proceso es intercambiado por otro, este sigue activo. La finalidad de estos estados es saber a qué procesos darle tiempo de procesamiento. Es decir, cuando un proceso se encuentre en los estados PAUSED o FINISHED el scheduler no restaurará su contexto y solamente lo hará para aquellos que se encuentren activos (ACTIVE). Asimismo, si un proceso está en estado FINISHED y se carga un nuevo proceso, este último ocupará el espacio en el stack del proceso finalizado.

---

[1] Se utilizó la siguiente documentación.
[2] Tanenbaum, Andrew S. - Modern Operating Systems. Upper Saddle River, N.J: Pearson Prentice Hall, 2008.

User Space:

Implementación de la terminal:

La funcionalidad de la terminal consta de una lectura constante de las teclas presionadas y la impresión de los caracteres correspondientes en la pantalla, mediante las system calls correspondientes. Todo aquello que se va leyendo, se guarda en un *buffer* y cuando se detecta un *ENTER* se decodifica lo escrito por el usuario. Si lo ingresado es un comando válido (véase el manual de usuario para ver los comandos) se procede a correr el programa correspondiente.

Implementación de los programas indicados en la consigna:

Véase el manual de usuario para ver una lista de los programas disponibles. Se tuvieron algunas consideraciones para algunos programas en específico:

- Primos y Fibonacci: los algoritmos fueron implementados utilizando técnicas de programación dinámica para evitar tener que recalcular datos previamente calculados.
- Invalid Opcode: se utilizó una instrucción de Assembler que permite generar este tipo de excepción, permitiendo igualmente que compile el programa. La instrucción en cuestión es `ud2.`
- Inforeg y Printmem: se implementaron system calls especiales para cada uno de ellos, pues es una falla de diseño que el usuario pueda acceder libremente a todas las direcciones de memoria de la computadora.

Implementación de funciones útiles:

Se implementaron funciones que permiten analizar y decodificar strings, la función `flush_buffer()` que se asemeja a la función que se encuentra en la librería estándar de C denominada `fflush()`, la función `sleep()` que espera una cantidad de segundos para continuar la ejecución y la función `uint_to_base()` recuperada de lo proveído por la cátedra.

Dificultades encontradas:

A la hora de afrontar la implementación del scheduler se tuvieron grandes dificultades debido a la falta de conocimiento sobre el mismo. No se tenía claro cómo manejar la actualización del contexto y cómo manejar los procesos pausados y finalizados. Luego de horas de trabajo, y con la ayuda de la búsqueda en la web se pudo encontrar una línea de pensamiento común y se pudo construir sobre eso. Para solucionar el problema de la actualización de los contextos, se descubrió que con armando el *stack frame* de interrupciones y haciendo uso de la instrucción de Assembler `iretq` se podía delegar al procesador el intercambio de punteros a código y pila. Por otro lado, para manejar los procesos pausados y finalizados se agregó un dato extra en la estructura y una validación a la hora de actualizar los contextos.

Se descubrió que cuando no había ningún proceso activo, la rutina de *scheduler* entraba en un ciclo infinito y por ende la interrupción del *Timer Tick* nunca terminaba. Esto generaba que ninguna interrupción fuera escuchada y debido a esto el Kernel se congelaba. Para sortear este problema se implementó un proceso basura o inútil que solamente corre cuando no existe proceso con estado activo.

Por último, se pueden detallar los problemas que aparecieron a la hora de imprimir en doble pantalla y realizar el *scrolling*. El principal inconveniente fue que los punteros a las pantallas derecha o izquierda sean actualizados acordemente, así como que su contenido sea desplazado una fila para arriba. Se solucionó este problema dedicándole el tiempo necesario y realizando esquemas de la pantalla.

Distribución de tareas:

Para la realización de este trabajo práctico no existió una distribución de tareas marcada. La resolución del mismo se realizó casi en su totalidad con todos los integrantes presentes, principalmente cuando se realizó la toma de decisiones respecto al diseño a implementar. De igual manera, hubo implementaciones de algunas funcionalidades más simples que fueron distribuídas entre los tres integrantes del grupo de manera equitativa.

Herramientas utilizadas:

Las herramientas que se utilizaron son las siguientes:

- Discord: comunicación sincrónica virtual entre los integrantes del grupo.
- Visual Studio Code: IDE.
- GitHub: control de versiones y sincronización entre los entornos de trabajo de los integrantes del grupo.
- Notion: documentación de avances diarios.
- Terminal + Docker + Make + QEMU + GCC: compilación y ejecución del kernel.
- GDB: debuggeo del kernel.
- Multipass: virtualización de un entorno Linux para macOS.

Limitaciones del Kernel:

La principal limitación del Kernel es que el scheduler puede manejar como máximo un total de cuatro procesos. Esto se debe a la falta de conocimiento sobre el tema, la falta de alocación de memoria dinámica y que sinceramente no se necesitaba nada más que eso para este trabajo práctico especial.

En relación a la limitación anterior, se puede comentar que la falta de memoria dinámica obligó a correr los procesos del Kernel en zonas de memoria arbitrarias y fijas.

Por último, se cree que otro aspecto a mejorar es que la impresión a pantalla desde User Space se pueda realizar con diferentes colores de relleno y fondo. Esto no fue implementado porque no se creyó necesario para la resolución del trabajo práctico especial.