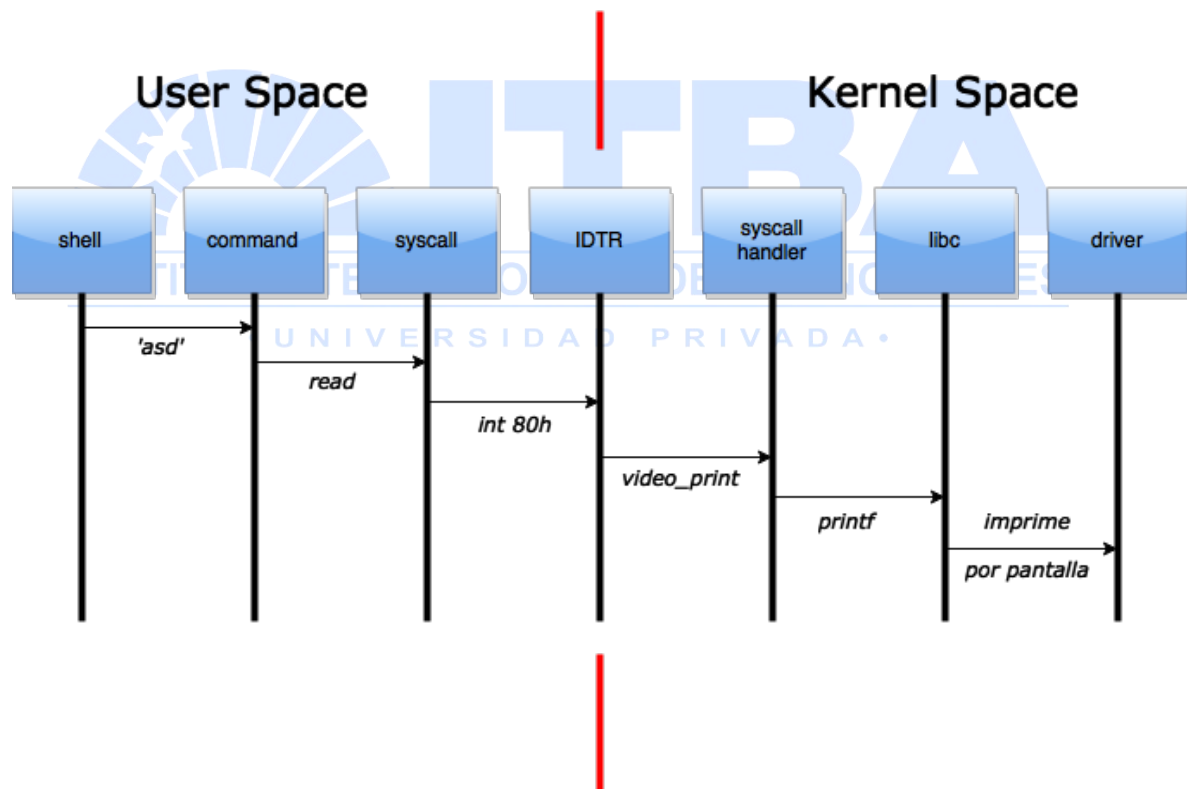


Desarrollo y decisiones de diseño

Para comenzar se obtuvieron y configuraron varios programas recomendados por la cátedra para el desarrollo del TPE, como **VirtualBox** y **Parallels**. En los mismos virtualizamos una versión de Linux con arquitectura Intel64. Una vez instalado y configurado el sistema operativo, descargamos e instalamos las herramientas **nasm** y **qemu**.

Una vez interiorizados con los programas de virtualización y las herramientas provistas para la compilación y ejecución kernel, clonamos el repositorio otorgado por la cátedra: *x64BareBones*, donde encontramos un setup básico para desarrollar sistemas operativos para la arquitectura de 64 bits de Intel.

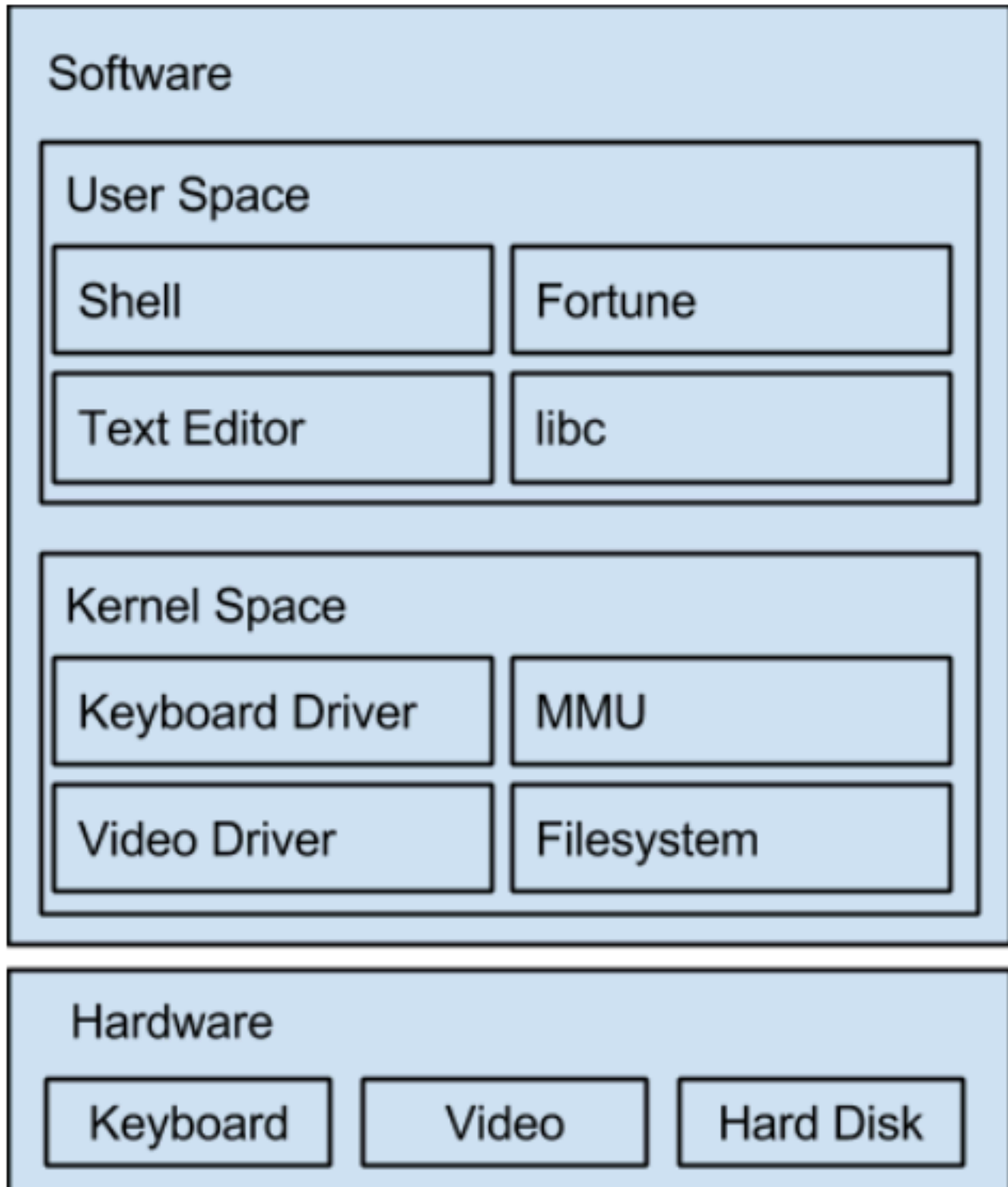
Basándonos en la teoría vista en clase, realizamos un diagrama de secuencia estableciendo las interacciones de cada uno de los actores (partes) del sistema operativo.





Con el diagrama de secuencia realizado decidimos comenzar el desarrollo aplicando un diseño bottom-up.

Antes de iniciar con la programación, planificamos cada una de las partes del sistema y cómo ensamblarlas a lo largo del proceso de trabajo.


Para ello nos fue útil el ejemplo de módulos provisto por la cátedra, cuya representación gráfica también se adjunta en el presente informe.




Aprovechando las funcionalidades de la plataforma GitHub establecimos issues como tareas pendientes, generamos labels para determinar el proceso actual de cada tarea y definimos milestones progresivos asignando cada una de las tareas. Esto nos permitió visualizar de manera intuitiva el progreso del trabajo, el tiempo empleado para cada tarea, y las cosas pendientes para el siguiente milestone.


☐  **6 Open**  **21 Closed**


☐

 **Manual de usuario** **TODO**


#27 opened 6 hours ago by cbiancucci  Entrega Final


☐

 **Informe Final** **WIP**


#26 opened 6 hours ago by cbiancucci  Entrega Final


☐

 **Shell** **WIP**


#24 opened 11 days ago by cbiancucci  Shell


☐

 **Screensaver** **WIP**


#20 opened 11 days ago by cbiancucci  Entrega Final


☐

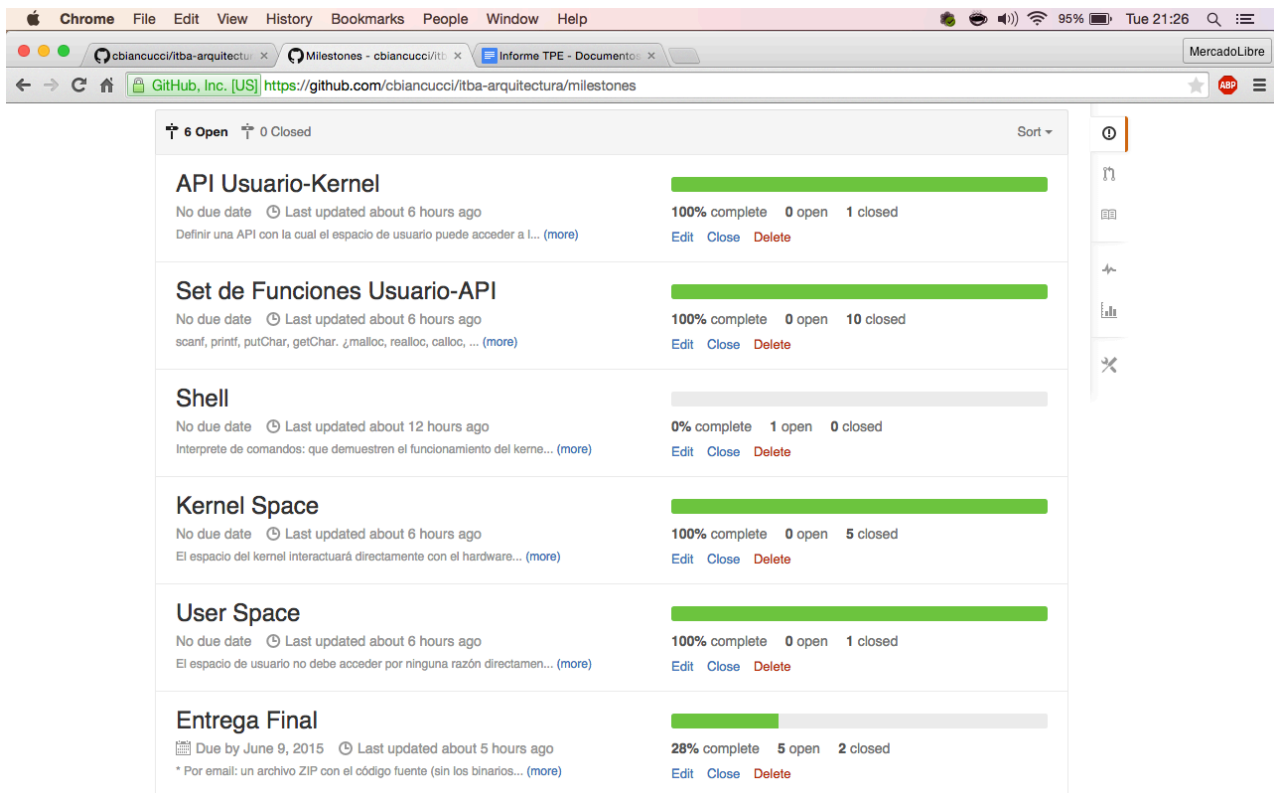
 **Cambiar la hora del sistema** **DONE**

#17 opened 11 days ago by cbiancucci  Entrega Final

☐

 **Visualizar la hora del sistema** **DONE**

#16 opened 11 days ago by cbiancucci  Entrega Final



La primera planificación de tareas se basó en la especificación de la librería `libc`, dónde indicamos qué funciones serían indispensables y cuáles opcionales. Utilizando el diagrama de secuencia y considerando los flujos más importantes del sistema determinamos dos tipos de categorías:

- Funciones relacionadas con memoria.
- Funciones relacionadas con input/output.

De este análisis también surgió la definición de estructuras comunes para ser utilizadas a lo largo de todo el proyecto.

El siguiente paso fue la implementación del handler `syscalls`. Este es el encargado de realizar el vínculo entre las llamadas de la interrupción de la IDTR (int 80h) con las librerías que desarrollamos para el Kernel Space. Además de las `syscalls` de memoria se diseñaron dos primitivas: `read` & `write` para separar las funciones de entrada de las de salida. En esta instancia, el handler determina si debe utilizar el driver de video para mostrar algo por pantalla, o el driver de teclado para obtener caracteres del buffer.

Para la implementación de la IDTR utilizamos la información provista por la cátedra en Pure64. Allí inicializamos, a través de una llamada, al Kernel, dónde se cargan los módulos y se define la dirección del stack. Posterior a esto definimos los handlers de interrupciones y determinamos el PIT handler (Programmable Interval Timer).

- Interrupciones de teclado.
- Interrupciones de software.

Una vez iniciado el PIT se llama al main del Kernel.

El próximo paso de nuestro proceso de desarrollo fue la implementación de las syscalls. Para esto, definimos un conjunto de interrupciones posibles desde User Space y las declaramos en la API. La implementación en *Userland* tiene como objeto ejecutar las interrupciones a la IDTR para que luego sean manejadas por el handler syscall desde el Kernel Space.

El shell es una estructura usada para representar la consola, a partir de la cual el usuario interactúa con el sistema. La misma está representada por un vector determinado por el ancho y la altura de la pantalla. Esta cuenta con un cursor que apunta a la posición dónde se escribirá/borrará. Dicho cursor está vinculado con las funciones implementadas previamente desde el Kernel Space para manejar driver de video.

Su propósito es llevar a cabo los llamados a las rutinas correspondientes para cada comando; es decir, el shell es el intérprete de comandos. Para soportar la lectura y ejecución de comandos por parte del shell, se trabajó en un parser para determinar la validez y la coherencia de las instrucciones ingresadas.

El usuario interactúa con el sistema a través del teclado. El mismo está representado por un vector dónde se vincula cada una de las direcciones físicas de las teclas con un símbolo o carácter que debe ser representado en pantalla, o bien una acción en particular que debe realizarse a partir de haberse presionado cierta tecla. Este caso se produce por ejemplo cuando el usuario presiona ENTER, SHIFT, o BACKSPACE.

Las ventajas de optar por esta estrategia para desarrollar el trabajo práctico especial se vieron reflejadas a la hora de ensamblar los diferentes componentes del sistema. Al haber comenzado desde la librería estándar hacia el shell y la interacción con el usuario, pudimos ir incorporando cada una de las etapas del sistema, sin tener que realizar grandes modificaciones a las anteriores. Esto nos permitió tener un código más sencillo y fácil de acoplar con el resto. Además, pudimos detectar rápidamente si habíamos pasado algo por alto e implementarlo de inmediato sin la necesidad de modificar diferentes funciones y refactorizar el código.

Sin embargo, este tipo de metodología conlleva una contra importante: los cambios realizados en la implementación no podían ser visualizados por pantalla. Por lo tanto, los tests de integración, para probar el funcionamiento de todos los comandos y los componentes, no se pudieron llevar a cabo hasta último momento.

Si en esta etapa nos hubiésemos encontrado con un error de diseño importante, corregirlo hubiese sido sumamente tedioso debido a que tendríamos que modificar cada interacción entre las capas que comprenden este sistema operativo.

Fuentes de Investigación y Referencias

Para el desarrollo de este trabajo práctico se utilizó como fuente de información principalmente la página wiki.osdev.org.

Como fuentes de consulta recurrimos a forums.osdev.org y stackoverflow.com (fundamentalmente para corregir algunos warnings en la compilación de archivos C).

Para la implementación de la biblioteca estándar se utilizó como referencia el libro 'The C Programming language' de Kernighan & Ritchie.

Dentro del código se dejaron algunas referencias a las URL específicas que nos sirvieron para el desarrollo.

<http://www.cs.usask.ca/classes/332/t1/os161/src/common/libc/string/memcpy.c>

http://wiki.osdev.org/Text_Mode_Cursor

<https://github.com/blt/bltos/blob/master/kernel.cpp>

Para compilar y probar el TPE en Mac OS X recibimos la ayuda de otro equipo, el cuál nos instruyó en la configuración de Makefiles y el desarrollo de un script para configurar las variables de entorno y poder correr tanto **nasm**, **qemu** y un **cross-compiler** en la plataforma *macos*.