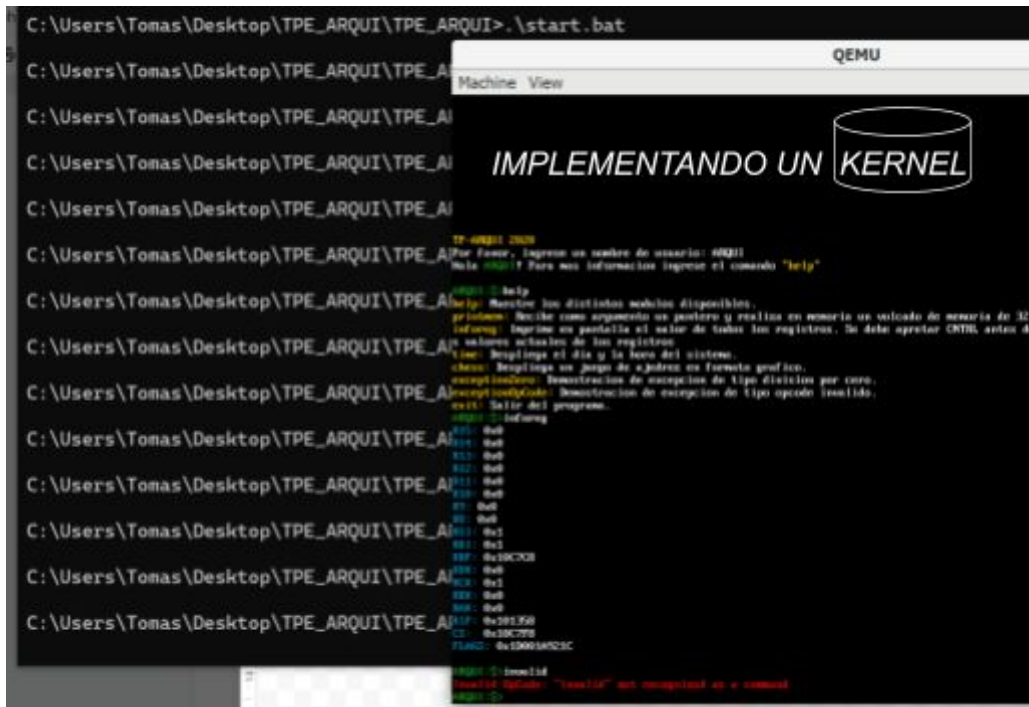


# ARQUITECTURA DE LAS COMPUTADORAS

Curso 2020 - Segundo Cuatrimestre

## TP ESPECIAL



GRUPO 10

Fecha de entrega:

8/11/2020

Integrantes del grupo:

SANTIAGO GARCIA MONTAGNER - 60352

TOMÁS CERDEIRA - 60051

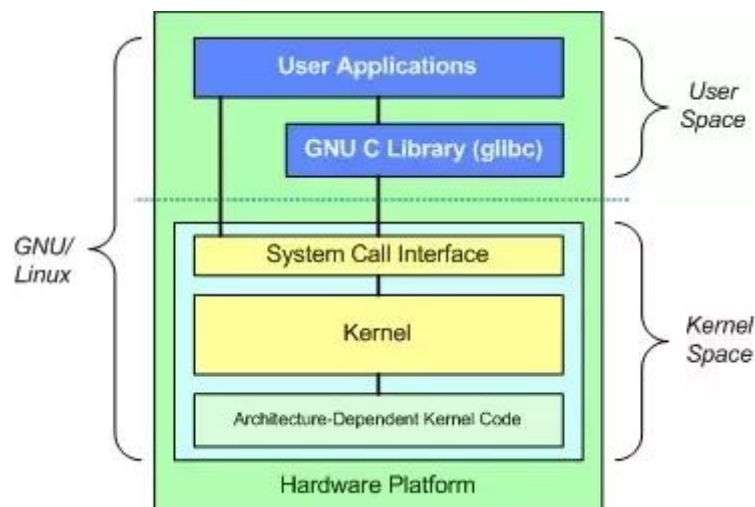
# Objetivos

Implementar un *kernel* que administre los recursos de hardware de una computadora y muestre características del modo protegido de Intel.

Se busca definir **dos espacios** claramente separados;

- *kernel-space*: lugar en donde se implementan aquellas funciones encargadas de interactuar directamente con el hardware, brindando una API a utilizar en el *user-space*.
- *user-space*: se refiere a los diversos programas y bibliotecas que el sistema operativo usa para interactuar con el *kernel*.

Este acceso será implementado a través de la interrupción de software *80h*, ya que estos dos módulos se encontrarán en distintos espacios de memoria. Se busca que dicha API se asemeje a la de *Linux*. Se definirán además, un set de funciones para interactuar con ella, equivalente a la “biblioteca estándar de C” en *Linux*.



**Figura 1<sup>1</sup>:** Representación visual de la separación entre el *kernel-space* y el *user-space*

El sistema se implementará para una Arquitectura Intel de 64 bits en *Long Mode*.

<sup>1</sup> <https://www.quora.com/What-is-a-Kernel-as-in-linux-kernel>

## ***kernel-space***

### IDT: Interrupciones y Excepciones

Se configuraron en total 5 posiciones de la IDT (*Interrupt descriptor table*):

- Para las interrupciones:
  - *0x20* : Timer Tick
  - *0x21* : KeyBoard
  - *0x80* : System Call
- Para las excepciones:
  - *0x00* : "Division by zero"
  - *0x06* : "Invalid Opcode"

Para ello, se definieron *Handlers* encargados de manejar dichas interrupciones y efectuar la/s acción/es correcta/s/necesaria/s en cada caso.

- *irqHandlerMaster* : manejo de *0x20* y *0x21*
- *sysHandlerMaster* : manejo de *0x80*
- *exceptionHandlerMaster* : manejo de *0x00* y *0x06*

Estos, mediante *Dispatchers*; *irqDispatcher.c*, *exceptionDispatcher.c* y *sysHandler.c*, delegan el manejo de las rutinas de atención.

## API: System Calls

Para invocar alguna de las siguientes funciones, se debe cargar al registro **RCX** con el número (**x**) antes de correr la interrupción *80h*. Recordar que, teniendo en cuenta las convenciones de una arquitectura de 64 bits, **los parámetros se reciben por registros**. En el caso de que no alcancen, se los recibe por stack.

Orden de los parámetros por registros: RDI, RSI, RDX, RCX, R8, R9.  
(par1, par2, par3, par4, par5, par6)

- (0) writeScreen(par1, par2, par3)
  - par1 → buffer donde se encuentra lo que quiero escribir
  - par2 → color de la letra
  - par3 → color del fondo

**OBS:** Escribe datos de un búfer declarado por el usuario a un dispositivo determinado. En nuestra implementación el único dispositivo posible es la salida estándar. El *par2* y *par3* definen el estilo con el que se escribe.

- (1) read(par1, par2)
  - par1 → buffer donde se va a escribir lo leído
  - par2 → longitud del buffer

**OBS:** lee hasta *par2* bytes de entrada estándar, copiandolos comenzando a partir de *par1*.

- (2) getDecimalTime((uint8\_t \*) par1, par2)
  - par1 → int \* donde se copia lo pedido
  - par2 → parámetro que indica si quiere la hora, minutos o segundos

**OBS:** devuelve en *par1* lo pedido por *par2*. (0: hora, 1: minutos, 2: segundos).

- (3) getRegisterState(par1, stackFrame)
  - par1 → int \* de entrada salida
  - stackFrame → puntero al stack el cual contiene a todos los registros

**OBS:** el *stackFrame* viene en el registro *R8*.

- (4) getMemoryState((unsigned char \*)par1, par2)
  - par1 → unsigned char \* de entrada salida
  - par2 → int que indica el principio del lugar de memoria pedido
- (5) borrado

- (6) `printFigure((unsigned char *)par1, par6[0], par6[1], par2, par3, par5[0], par5[1])`
  - `par1` → puntero al bitmap de la figura a dibujar
  - `par2` → ancho de la figura
  - `par3` → alto de la figura
  - `par5[0]` → color del trazo con el que se dibuja
  - `par5[1]` → color del fondo con el que se dibuja
  - `par6[0]` → coordenada x en donde se desea dibujar en la pantalla
  - `par6[1]` → coordenada y en donde se desea dibujar en la pantalla

**OBS:** esta función trabaja en conjunto con;

*`drawFigure(unsigned char *code, int x, int y, int width, int height, int color, int background_color)`*

la cual se encuentra en *`videoDriver.c`* y es la encargada de “manejar” los pixeles.

- (7) `deleteN(par2)`
  - `par2` → cantidad de caracteres que se desean borrar

**OBS:** sirve para borrar *par2* caracteres anteriores al cursor.

- (8) `clearConsoleIn(par2)`
  - `par2` → int que representa los segundos de delay antes de hacer un borrado de la pantalla
- (9) `setCronometro(par2)`
  - `par2` → int para activar o desactivar el cronómetro
    - `par2 == 0` lo desactiva
    - `par2 != 0` lo activa
- (10) `getSecondsCronometro()`

**OBS:** devuelve por *par1* (*RDI*) el int de los segundos hasta ese instante cronometrados.

Tanto (9) como (10) son syscalls que interactúan con *timer.c*, el encargado de manejar lo relacionado con el tiempo y los ticks del CPU.

- (11) `setCursor(par2, par3)`
  - `par2` → coordenada x a donde se desea mover el cursor
  - `par3` → coordenada y a donde se desea mover el cursor

- (12) disableCursor()

**OBS:** (11) modifica la posición del cursor cambiandola a (*par1*, *par2*). Para retornarlo al lugar anterior, se usa (12).

- (13) drawRectangle(*par1*[0], *par1*[1], *par2*, *par5*[0], *par5*[1])
  - *par1*[0] → coordenada x en donde se desea dibujar en la pantalla
  - *par1*[1] → coordenada y en donde se desea dibujar en la pantalla
  - *par2* → color del trazo con el que se dibuja
  - *par5*[0] → ancho
  - *par5*[1] → alto

**OBS:** dibuja un rectángulo de *par5*[0] x *par5*[1] tomando como centro las coordenadas (*par1*[0], *par1*[1]) de la pantalla.

### Justificación y comentarios de las decisiones tomadas:

- Respetando el modo protegido de Intel, todos los accesos a recursos protegidos del computador que se deseen acceder desde *user-space* son a través de *syscalls* implementadas por el *kernel*. (las mencionadas anteriormente)
  - para lograrlo, se crearon archivos como; *videoDrivers.c* y *keyboardDriver.c*
- Si el usuario desea saber el ancho y alto de la pantalla, se decidió NO hacer una *syscall* que devuelva estos valores, si no que es responsabilidad de el leer la documentación en donde se los especifica.
- Se creó un archivo, *timer.c*, en el *kernel-space*, para todo lo relacionado al manejo del tiempo en el CPU. Se decidió crear *syscalls* para obtener esta información, simulando así el concepto de “multiprocesos”.
- Priorizamos escribir código e implementar librerías en C y no en ASM para hacerlo lo más portable posible.
- Para asegurarnos el correcto funcionamiento entre módulos y procesos, se preservan todos los registros al entrar y salir de funciones.