Lab1: manejo del RTC

Objetivos Generales

- Manejar un dispositivo simple.
- Agregar una *system call*.
- Programar algo sencillo en *userspace* y *kernelspace*.

Objetivos Particulares

- Afianzarse en el ciclo edición-compilación-ejecución de `xv6`.
- Utilizar `git` para el ciclo de producción del programa.
- Proveer de parches para la implementación.

Primera parte: lectura del RTC

El *real-time clock* es un dispositivo de entrada/salida que está desde la IBM PC/AT en 1984.
Este dispositivo mantiene un reloj de tiempo real alimentado de manera permanente.
El chip que implementa esto es el Motorola MC146818AS, pero ya se encuentra integrado dentro del *south-bridge* y no necesita de batería alguna ya que se alimenta con supercapacitores.

El RTC es en realidad una memoria permanente de 64 bytes donde sus primeros 10 bytes se actualizan con un reloj interno muy estable para dar la siguiente información:

Address Location	Function	Decimal Range	Range		Example*	
			Binary Data Mode	BCD Data Mode	Binary Data Mode	BCD Data Mode
0	Seconds	0-59	$00-$3B	$00-$59	15	21
1	Seconds Alarm	0-59	$00-$3B	$00-$59	15	21
2	Minutes	0-59	$00-$3B	$00-$59	3A	58
3	Minutes Alarm	0-59	$00-$3B	$00-$59	3A	58
4	Hours (12 Hour Mode)	1-12	$01-$0C (AM) and $81-$8C (PM)	$01-$12 (AM) and $81-$92 (PM)	05	05
	Hours (24 Hour Mode)	0-23	$00-$17	$00-$23	05	05
5	Hours Alarm (12 Hour Mode)	1-12	$01-$0C (AM) and $81-$8C (PM)	$01-$12 (AM) and $81-$92 (PM)	05	05
	Hours Alarm (24 Hour Mode)	0-23	$00-$17	$00-23	05	05
6	Day of the Week Sunday = 1	1-7	$01-$07	$01-$07	05	05
7	Date of the Month	1-31	$01-$1F	$01-$31	0F	15
8	Month	1-12	$01-$0C	$01-$12	02	02
9	Year	0-99	$00-$63	$00-$99	4F	79

¿Cómo leer los bytes?

Toda la interfaz del MC146818AS se encuentra **mapeada en puertos** `0x70` y `0x71`.
El puerto `0x70` sirve para indicar la dirección de la memoria del RTC y el `0x71` para leer o escribir esa dirección de memoria.

Veamos un ejemplo para leer el año desde *userspace*: `year.c`.

```
#include "types.h"
#include "user.h"

#define RTC_ADDR 0x70
#define RTC_DATA 0x71

uchar rtc_read(const ushort addr){
  ushort port = 0;
  uchar val = 0;

  port = RTC_ADDR;
  val = addr;
  /* http://wiki.osdev.org/Inline_Assembly/Examples */
  asm volatile("outb %0,%1"::"a"(val), "d" (port));

  port = RTC_DATA;
  asm volatile("inb %1,%0" : "=a" (val) : "d" (port));
  return val;
}


int
main(void)
{
  uint year = 0;
  year = rtc_read(0x09);

  printf(1, "%d\n", year);
  exit();
}
```

NOTA 1: la línea `asm volatile` es para hacer *inline assembler*, es decir meter código ensamblador en medio del código "C". Ustedes no tendrán que hacer esto ya que `xv6` provee funciones para entrada y salida de puertos.

NOTA 2: ¡Esto no debería poder usarse! (todo lo que sigue puede ser críptico, no prestarle mucha atención)
Para solucionarlo hay que poner en el TSS todos los bits a 1 del *permission IO bitmap*.
En Linux esto se controla con `ioperm()`. Notar que `IOPL` está puesto a 0, pero el manual de Intel es claro, no solo basta con que `!(CPL<=IOPL)`, sino que además los **io permission bits** del `TSS` tienen que estar a 1.
Ejercicio: escribir esto en Linux y ver que el sistema operativo mata el proceso.

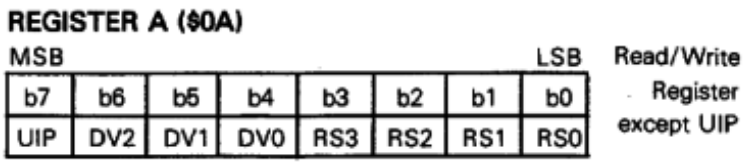
Se puede comprobar que la representación decimal del año no es la adecuada.
Hay que transformar un número BCD de dos dígitos en un número binario.

Una vez obtenidos todos los datos de la fecha hay que armar un número entero que indique la cantidad de segundos que transcurrieron desde **`the epoch`** (más información en inglés), que es el **1 de Enero de 1970 a las 0:00hs** respecto de la zona horaria UTC, o sea la de Islandia.

Existe un **problema de concurrencia** en la lectura del RTC.
Podría suceder que estemos leyendo el valor **concurrentemente** a que el hardware esté actualizando uno a uno los valores.
Por ejemplo si la actualización es de derecha a izquierda, el valor `14-07-22 17:46:59` pasa a `14-07-22 17:47:00`, pero podemos leer en una actualización intemedia: `14-07-22 17:46:00`, o sea leemos **con un minuto de atraso**. Este mismo fenómeno puede causar retrasos de una hora, un día, un mes, y lo peor: de

un año.

Para solucionar este problema el RTC deshabilita la lectura de la memoria durante la actualización, y por lo tanto **el resultado está indefinido** durante ese periodo.
El tiempo de actualización se indica con un bit denominado `UIP` (*update in progress*) y está en el **Registro A** que se encuentra justamente en la posición `0x0A` de la RAM del RTC.



La hoja de datos dice en su página 15:

> UIP - The update in progress (UIP) bit is a status flag that may be monitored by the program. When UIP is a "1", the update cycle is in progress or will soon begin. When UIP is "0", the update cycle is not in progress and will not be for at least 244 μs (for all time bases).

Para evitar lecturas erroneas hay que hacer una **espera ocupada** (*busy-wait*) mientras el bit 7 del Registro A sea 1.

Implementar

- Una función dentro de `main.c` con signatura `int rtcread(void)` que lea el RTC teniendo en cuenta la condición `UIP` y lo convierte en segundos desde `the epoch`.
- Agregar una función `main.c:int rtcinit(void)` que llame a `rtcread()` y devuelva la cantidad de segundos desde `the epoch` al momento del inicio.
- Agregar en `main.c:main()` la llamada a `rtcinit()` e imprimir por consola que se inicializó el RTC correctamente mostrando la lectura. Ejemplo: `RTC: boottime 1406557139`.

NOTA: Aunque `int rtcinit(void)` solo llama a `int rtcread(void)`, en la segunda parte habrá código propio en `rtcinit`.

Estilo de código

- **Mantener de manera exacta el estilo de código de `xv6`** en cuanto a tabulaciones, nombres de variables, formas de declarar las funciones, posición de las llaves y corchetes, espacios antes y después de los operadores, etc.
- Declarar todas las funciones auxiliares, es decir las que no se necesita usar fuera de un módulo como `static`.
- Declarar como `const` **todos** los parámetros y variables que **no se modifican**.

Ayudas

- `xv6` **ya tiene** funciones de lectura bytes en puertos `x86.h:inb(ushort port)` y de escritura de bytes en puertos `x86.h:outb(ushort port, uchar data)`.
- Para pasar de BCD a binario hay que armar una función sencilla que utilice operadores *bitwise* de máscaras `&` y desplazamientos `>>` para desarmar el número BCD y luego rearmarlo como binario.

- Para la lectura correcta teniendo en cuenta UIP, habrá que leer el byte de la dirección `0x0A` de la RTC

RAM y hacer una máscara *bitwise* adecuada.

    - No revisar el valor de `UIP` antes de leer los valores puede traer consecuencias no deseadas como la lectura de una fecha inválida.
- La función de transformación de `(s,m,h,d,m,a)->segundos` desde `the epoch` no es sencilla. Conviene adaptar `mktime()` del kernel de Linux.
- Tener especial cuidado con la representación de los años.
- Tener en cuenta que la cantidad de segundos desde `the epoch` es del orden de `1406557139` en estos días, la representación `int` llega hasta `2**31 -1 = 2147483647` asi que estamos medio jugados, el 19 de Enero de 2038 se acaba el mundo UNIX.

- Para comprobar que el resultado sea correcto, se puede utilizar desde el sistema operativo anfitrión (*host OS*) el comando `date "+%s"` que imprime el tiempo actual desde `the epoch`. A la inversa, para ver que fecha y hora es un *unix time* se puede usar `date --date='@1406557139'`.

Notar

- `xv6` ya tiene definido el puerto del RTC en `lapic.c:#define IO_RTC  0x70`, pero se usa para el reinicio (*reset*) de la computadora. Como no está puesto en una cabecera (*header* `*.h`) conviene redefinirlo.

Segunda parte: agregar `gettimeofday` usando el RTC

Hasta ahora logramos una función que imprime el tiempo desde `the epoch` al inicializar el kernel (*booting*). Pero esto no sirve de mucho, ya que no podemos consultar ni la hora de inicio ni la hora actual desde los programas de usuario.

Queremos brindar la hora actual a través de la *syscall* `int gettimeofday(void)` sin tener que leer el RTC cada vez que se llama a la función.
Pedimos esto porque requiere muchas entradas y salidas de puerto además de entrar en *busy wait* para esperar que no se dé `UIP` antes de leer los valores.
Todas estas operaciones, aunque no resultan complejas, hacen que la llamada al sistema sea **innecesariamente costosa**, y el sistema operativo tiene que ser lo más liviano posible en todas sus llamadas a sistema.

Hay dos estrategias de implementación.

- Absoluto.
- Relativo a `uptime()`.

En la primera, almacenamos el entero con la cantidad de segundos desde `the epoch` para la inicialización en una variable global `timeofday`, y en cada interrupción de reloj, lo incrementamos adecuadamente.
La llamada a sistema `sys_gettimeofday` solo retorna el valor de `timeofday`.

Para la segunda estrategia, almacenamos el `rtcread()` en una variable global `boottime` y cuando se llama a la `sys_gettimeofday()` se suma el tiempo del inicio al tiempo que está prendida la máquina.

Implementar

- Modificar la llamada a `rtcinit()` desde `main.c:main()` para que además de imprimir almacene el valor del RTC en la variable global `timeofday` o `boottime` según el esquema elegido.
- Escribir la *syscall* `gettimeofday()` que devuelva la cantidad de segundos desde `the epoch`.
- Implementar un programa de usuario `date` que muestra los segundos que pasaron desde `the epoch`.

Comprobar que está sincronizado con el reloj de la cocina.

Ayudas

- Para agregar una nueva *syscall* deberán modificar **varios** archivos. Tomar alguna llamada a sistema y ver en todos los puntos que aparece.
- Si es relativo al `uptime`.
    - Al inicializar almacenar en la variable `bootime` el tiempo desde `the epoch`.
    - En la *syscall* `gettimeofday()`, calcular el tiempo actual a partir del uptime almacenado en `ticks`, más el `bootime`.
    - ¡Una vez más cuidado con mezclar unidades! el valor de `ticks` se incrementa más frecuentemente que 1 vez por segundo.
- Si es absoluto:
    - Detectar la parte del código que realiza la *syscall* `uptime()`.
    - Identificar donde se declara y modifica la variable principal que cuenta el tiempo: `ticks`.
    - Copiar esa estructura para mantener un nuevo contador de *ticks*: `int ticks_epoch`.
    - Este contador se tiene que incrementar en cada interrupción.
    - ¡Cuidado, la frecuencia de la interrupción periódica `IRQ_TIMER` manejanda en `trap.c` es mayor a 1 vez por segundo!
- Además del comando `date` en el *hostOS*, hay sitios con conversores de unix time a human time.

Extras

- Imprimir por consola un `"UIP"` cada vez que se detecte la condición de *update in progress* para saber si este fenómeno es común o raro.
- Modificar la signatura de `gettimeofday` para que cargue una `struct timespec *ts`, ya que esta es la signatura estándar de esta función en POSIX.
- Re-implementar `date` para que muestre la fecha actual en formato `AAAA-MM-DD HH:MM:SS`.
- Hacer la implementación **modular**, definiendo `time.{h,c}` cuyas únicas funciones públicas son `rtcinit()` y `sys_gettimeofday()`.

Entrega

Este laboratorio se entregará con *commits* subidos al proyecto `Lab1` dentro del equipo `so2014gXX` que tienen asignado en el repositorio git de Bitbucket y deberá incluir un archivo `README.md` en formato markdown con el informe sobre el proceso de desarrollo del lab.

- La organización del git deberá ser:
    - Dado el número de grupo `XX` usar el **equipo** `so2014gXX` en Bitbucket.
    - Crear un **repositorio** denominado `Lab1`.
    - Dentro del repositorio poner la implementación dentro de un **directorio** `xv6`.
    - El informe `README.md` deberá ir en la **raíz** del repositorio.
- Implementación funcional en el repositorio respetando a rajatabla el estilo de código de `xv6`.
- Informe en formato markdown mostrando el proceso de desarrollo.

- ¡No hay que repetir el enunciado!

Manejo de `qemu`

Repetimos estos consejos:

- Salir de QEMU: `<CTRL-a> x`.
- Para que inicie qemu sin pantalla VGA: `make qemu-nox`.