Sistemas Operativos

Departamento de Computación – FCEyN – UBA

Primer cuatrimestre de 2022

Segundo parcial – 28/6 - Primer cuatrimestre de 2022

Nombre y apellido: _____

Nº orden: _____ L.U.: _____ Cant. hojas: _____

1	2	3	Nota

- Numere las hojas entregadas. Complete en la primera hoja la cantidad total de hojas entregadas.
- Entregue esta hoja junto al examen, la misma **no** se incluye en la cantidad total de hojas entregadas.
- Cada ejercicio debe realizarse en hojas separadas y numeradas. Debe identificarse cada hoja con nombre, apellido y LU.
- **Cada código o pseudocódigo debe estar bien explicado y justificado en castellano. ¡Obligatorio!**
- La devolución de los exámenes corregidos es personal. Los pedidos de revisión se realizarán por escrito, antes de retirar el examen corregido del aula.
- Los parciales tienen tres notas: I (Insuficiente): 0 a 59 pts, A- (Aprobado condicional): 60 a 64 pts y A (Aprobado): 65 a 100 pts. No se puede aprobar con A- ambos parciales. Los recuperatorios tienen dos notas: I: 0 a 64 pts y A: 65 a 100 pts.

Ejercicio 1. Sistemas de Archivos (30 puntos)

El comando `ls` de la terminal de **Unix** permite listar el contenido de un directorio. La forma de utilizarlo es la siguiente: *ls [OPCIONES] ... [ARCHIVO]*. Por ejemplo, al ejecutar `ls -ali`[1] sobre el directorio `/dir1` en una terminal de **Unix**, el *output*[2] es el siguiente:

```
$ ls -ali /dir1
7958965 drwxr-xr-x   4 carlovich  staff     272  Nov 26 22:33 .
      2 drwxr-xr-x  31 root       wheel     1122 Aug 10 13:09 ..
7958967 -rwxr-xr-x   1 carlovich  staff     1196 Oct  8 17:24 a
7692324 drwxr-xr-x  19 carlovich  staff     646  Nov 26 22:33 b
7958971 -rw-r--r--   2 carlovich  staff     274  Oct  8 17:24 c
7688464 drwxr-xr-x  17 carlovich  staff     578  Nov 26 22:04 d
```

Se pide implementar una función *int lsParcial(char * dir)*, que tome el *path* de un directorio y tenga el mismo comportamiento que *ls -ali*. Se cuenta con las siguientes estructuras para representar inodos y entradas de directorio:

```
struct Ext2FSDirEntry {
    unsigned int inode;
    unsigned short record_length;
    unsigned char name_length;
    unsigned char file_type;
    char name[];
};
struct Ext2FSInode {
    unsigned short mode; // info sobre el tipo de archivo y los permisos
    unsigned short uid; // id de usuario
    unsigned int size; // tama\~no en bytes
    unsigned int atime;
    unsigned int ctime;
    unsigned int mtime; // fecha ultima modificacion
    unsigned int dtime;
    unsigned short gid; // id de grupo
    unsigned short links_count; // cantidad de enlaces al archivo
    unsigned int blocks;
    unsigned int flags;
    unsigned int os_dependant_1;
    unsigned int block[15];
    unsigned int generation;
    unsigned int file_acl;
    unsigned int directory_acl;
    unsigned int faddr;
    unsigned int os_dependant_2[3];
};
```

Se cuenta también con la constante `BLOCK_SIZE` y con las siguientes funciones:

[1] Parámetros: -a lista todos los archivos, incluyendo los ocultos; -l lista con formato largo, mostrando los permisos; -i lista el número de inodo.

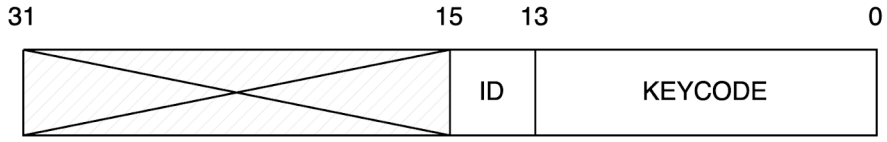
[2] Columnas: inodo | tipo de archivo + permisos | número de enlaces hacia este archivo | propietario | grupo | tamaño en bytes | Fecha de la última modificación | nombre

```
char * tipo_y_permisos(unsigned short mode)
// dado el campo mode de un inodo, devuelve una cadena denotando el tipo y
// permisos tal como lo hace ls

struct Ext2FSInode * Ext2FS::inode_for_path(const char * path)
// dado un path, devuelve su inodo

void Ext2FS::read_block(unsigned int block_address, unsigned char * buffer)
// dada una direccion de bloque y un buffer, carga el bloque indicado en el // buffer

unsigned int Ext2FS::get_block_address(struct Ext2FSInode * inode, unsigned int block_number)
// dados un inodo y un numero de bloque, recorre el inodo buscando la
// direccion del bloque de datos indicado

struct Ext2FSInode * Ext2FS::load_inode(unsigned int inode_number)
// dado un numero de inodo, busca el inodo en el grupo y lo devuelve
```

Aclaración: las dos primeras entradas de directorio en los datos de un inodo directorio refieren al propio inodo ('.') y al directorio en el que éste se encuentra ('..')

Ejercicio 2. Sistema de E/S - Drivers // Seguridad: (45 puntos)
Se pide programar el driver para un teclado gamer. Este teclado es muy novedoso, ya que es mucho más grande y tiene muchas más teclas que los teclados normales, y puede ser utilizado al mismo tiempo en hasta tres juegos distintos. Esto permite que hasta tres personas a la vez puedan jugar con el mismo teclado, pero cada quién con su propia aplicación, y sin interferirse entre ellos. Además, el teclado cuenta con tres visores inteligentes que permiten mostrar información sobre el juego que se esté jugando (por ejemplo, la vida del jugador, o mensajes importantes). También permite activar, desactivar y remapear teclas, y configurar macros, para cada usuario por separado. Además de todas estas impresionantes características, tiene luces de todos los colores posibles (también configurables).

El funcionamiento del controlador del dispositivo es el siguiente: cada vez que el usuario presiona una tecla el teclado levanta la interrupción `IRQ_KEYB` y coloca en el registro `KEYB_REG_DATA` un entero de 32 bits que contiene en sus 14 bits menos significativos el código `KEYCODE` correspondiente a la tecla presionada, y en los 2 bits siguientes un identificador `P`, que vale `0` cuando la tecla debe ser recibida por todas las aplicaciones conectadas, y en caso contrario un número entre `1` y `3` que indica a qué aplicación está destinada esa tecla. Considerar que, al ser una interrupción, esta debe ser atendida en un tiempo acotado, es decir, no debe bloquearse ni quedarse esperando. Se le deberá informar al dispositivo que la tecla pudo ser almacenada escribiendo correctamente escribiendo el valor `READ_OK` en el registro `KEYB_REG_CONTROL`. En caso contrario se le deberá escribir el valor `READ_FAILED`.



Por otro lado, el dispositivo cuenta con tres direcciones de memoria (una por cada aplicación posible) `INPUT_MEM_0`, `INPUT_MEM_1`, `INPUT_MEM_2`, siendo cada una un arreglo de 100 bytes desde los que el dispositivo leerá el input de las aplicaciones. Estas direcciones serán mapeadas a memoria por el driver durante su carga en el Kernel, y se mapearán en el arreglo `input_mem`.

Cuando una aplicación se conecte al teclado, se le deberá informar al mismo escribiéndole el valor `APP_UP` en el registro `KEYB_REG_STATUS`, y el id de la aplicación correspondiente en el registro `KEYB_REG_AUX`. Cuando una aplicación se desconecte se deberá hacer lo mismo pero escribiendo el valor `APP_DOWN`.

El driver deberá ir almacenando los caracteres ASCII correspondientes a las distintas pulsaciones del teclado en un buffer, a la espera de ser leídas por las aplicaciones que se encuentren conectadas. Para ello, se cuenta con una función `keycode2ascii(int keycode)` que traduce los códigos de pulsación en su correspondiente caracter ascii.

Los procesos de usuario que deseen leer el input del teclado deberán hacerlo mediante una operación de `read()` bloqueante. Esta operación sólo puede retornar cuando haya leído la cantidad de bytes solicitados. En caso de haber más de un proceso conectado al dispositivo, cada proceso deberá poder consumir cada caracter leído de forma independiente. Ejemplo: suponer que hay tres procesos conectados, y desde el dispositivo se presionaron las teclas correspondientes a "sistemas", en todos los casos con el identificador `P=0`. En tal caso si p0 hace un read de 2 bytes, obtendrá "si", si p1 luego hace un read de 5 bytes, obtendrá "siste", si luego p0 hace un read de 1 byte obtendrá "s", y si luego p2 hace un read de 8 bytes obtendrá "sistemas".

Además, los procesos conectados al teclado podrán utilizar la función `write()` para informarle al teclado el estado del jugador, comandar los colores del teclado, reconfigurar teclas, y varias otras funciones que provee el dispositivo.

Se cuenta con el siguiente código:

```
char input_mem[3][100];
char buffer_lectura[3][1000];
atomic_int buffer_start[3];
atomic_int buffer_end[3];
boolean procesos_activos[3];

void driver_load() {
    // Se corre al cargar el driver al kernel.
}

void driver_unload() {
    // Se corre al eliminar el driver del kernel.
}

int driver_open() {
    // Debe conectar un proceso, asignandole un ID y retornandolo,
    // o retornando -1 en caso de falla.
}

void driver_close(int id) {
    // Debe desconectar un proceso dado por parametro.
}

int driver_read(int id, char* buffer, int length) {
    // Debe leer los bytes solicitados por el proceso ''id''
}

int driver_write(char* input, int size, int proceso) {
    copy_from_user(input_mem[proceso], input, size);
    return size;
}
```

a) Implementar las funciones `driver_load`, `driver_unload`, `driver_open`, `driver_close` y `driver_read`. Implementar cualquier función o estructura adicional que considere necesaria (tener en cuenta que en el kernel no existe la *libc*). Se podrán utilizar las siguientes funciones vistas en la práctica:

```
unsigned long copy_from_user(char* to, char* from, uint size)
unsigned long copy_to_user(char* to, char* from, uint size)
void* kmalloc(uint size)
void kfree(void* buf)
void request_irq(int irqnum, void* handler)
void free_irq(int irqnum)
void sema_init(semaphore* sem, int value)
void sema_wait(semaphore* sem)
void sema_signal(semaphore* sem)
int IN(int regnum)
void OUT(int regnum, int value)
void mem_map(void* source, void* dest, int size)
void mem_unmap(void* source)
```

Para resolver la función `driver_read` es posible implementar una cola circular del modo descripto a continuación. Las variables `buffer_start[i]` y `buffer_end[i]` indican el inicio y el final del *buffer*. Al hacer una lectura, la variable `start` crece. Al hacer una escritura, la variable `end` crece. Si la variable `end` llega al final del *buffer*, debe comenzar por el principio, siempre teniendo cuidado de no pisarse con `start`. Si vale que `end+1` es igual a `start`, entonces se considera que el *buffer* está lleno. Para facilitar esta implementación, se cuenta con las funciones, que son todas atómicas:

- `int get_buffer_length(int i)`: dado un número de *buffer* devuelve la cantidad de caracteres ocupados en el mismo.
- `boolean write_to_buffer(int i, char src)`: intenta escribir un caracter en el *buffer* correspondiente, y retorna un booleano indicando si la escritura se pudo realizar.
- `boolean write_to_all_buffers(char src)`: intenta escribir un caracter en todos los *buffers*, y retorna un booleano indicando si la escritura se pudo realizar.
- `void copy_from_buffer(int i, char* dst, int size)`: lee la cantidad de caracteres indicada desde el *buffer*

correspondiente, y los copia en un segundo *buffer* pasado como parámetro. Tener en cuenta que esta función no realiza ningún tipo de chequeo sobre los *buffers*, sino que simplemente copia.

Importante: El código entregado en este ejercicio debe ser correcto y no debe presentar fallas de seguridad ni de concurrencia.

b) Un jugador malicioso ha comprado este teclado recientemente y le ha contratado para realizar una auditoría del driver en busca de vulnerabilidades. Su objetivo es invitar a sus amigos para jugar un torneo de su juego *open source* favorito, utilizando para ello su nuevo teclado, de forma tal de poder lograr que al tocar la tecla que se encuentra a la izquierda del **1**, en el resto de los jugadores se generen pulsaciones aleatorias que descontrolen momentáneamente a sus personajes para así poder ganarles. Para ello, le solicita programar una versión modificada del juego que le permita tener esta ventaja. Considerar que el jugador malicioso no debe resultar afectado por estas pulsaciones aleatorias. Explicar si es posible realizar dicha modificación, y en caso afirmativo brindar una explicación precisa de la misma. Considerar que este jugador siempre jugará desde la posición **0**.

Ejercicio 3. Sistemas distribuidos: (25 puntos)
Se tiene un sistema distribuido de la siguiente forma:

- Un servidor principal que recibe *requests* de usuarios de una aplicación. Hay m grupos de nodos agrupados donde cada uno tiene un líder. Tienen una topologia donde todos estan conectados con todos.
- Cada grupo tiene asignado un nivel de prioridad, del 1 al 3 donde 1 es el más crítico y 3 es el de prioridad más baja.
- Cuando llega una petición al servidor central, este le envía a cada líder instrucciones de cómo tiene que actuar. Cada grupo procesa la solicitud y le responde al servidor central. El *server* debe devolver una *request* de éxito al usuario bajo las siguiente condiciones:
  - Se tiene que tener el resultado de todos los líderes.
  - Se tiene ese resultado antes de un *timeout* de k segundos.
- Si el tiempo de procesamiento es mayor al *timeout*, se prosigue de la siguiente forma:
  - El servidor tiene que tener el resultado de todos los lideres con la prioridad crítica.
  - Basta con tener m respuesta de aquellas con prioridad **2** y l de los de prioridad **3**.

a) Se pide describir un protocolo que permita al sistema operar manteniendo su consistencia de las dos formas:
   - Antes de cumplir el *timeout*, con todos los grupos de nodos.
   - Después de cumplir el *timeout*, con la especificación pedida.

b) Muestre una análisis del comportamiento de su protocolo ante la pérdida de mensajes y su tolerancia a fallas.
   - Puede suponer que el servidor principal puede caerse pero durante un tiempo muy corto ya conocido.

c) Cada grupo de nodos, cuando un líder se cae, tiene una forma particular de elegir un nuevo líder. Todos los nodos tienen internamente una función que dado un número entero i, devuelve un número real en un intervalo dado. Todos tienen inicializado el valor de i en 0 y a medida que van utilizando la función, incrementan esa variable en 1. Con el número devuelto por la función (que lo tienen todos los nodos), se elige aquel nodo cuya identificación está más cerca de aquel número en modulo.
   - Describa un algoritmo para poder resolver este problema teniendo en cuenta que los nodos pueden fallar. Considerando que todos los nodos tienen que mantener la consistencia con respecto al valor de i, puede ocurrir que un nodo falle y quede con un valor anticuado.