

# Arquitecturas de Computadoras

## TP Especial

Mini Sistema operativo

### **Integrantes**

MAGNORSKY, Alejandro  
MATA SUÁREZ, Andrés  
MERCHANTE, Mariano

### **Cátedra**

VALLÉS, Santiago Raúl  
EL NOMBRE DEL NUEVO NO ME LO SE NI LO DICE IOL

### **Fecha de entrega**

Viernes 18 de junio de 2010

# Índice

1. Inconvenientes surgidos - FALTA BOCHA*****	3
2. Bibliografía consultada	5
3. Código fuente	6
3.1. Directorio src . . . . .	6

## 1. Inconvenientes surgidos - FALTA BOCHA\*\*\*\*\*

Dada la imposibilidad de uso de las librerías de C oficiales, en determinadas situaciones nos vimos obligados a ajustar el diseño de nuestro sistema de acuerdo con la implementación de nuestra propia librería de C. Tales percances y sus resoluciones son expuestos a continuación:

### Alocación en memoria

Quizás este fue el problema con el que más frecuentemente tuvimos que lidiar. En otras circunstancias, el uso de la función de reserva de espacio en memoria `malloc` de C nos hubiera permitido no muy difícilmente solucionar la cuestión. Sin embargo, al no tener la bases suficientes como para implementar una función similar a la mencionada en nuestra librería, en los casos que así lo dispusieron optamos por reservar cierta cantidad fija en memoria y sacrificar la manipulación dinámica de ese espacio, mediante la declaración de vectores.

### Máxima cantidad de caracteres ingresados por el usuario

La función `scanf` de `lib.c`, en un principio, solicita al usuario ingrese una string, invocando a la función `getString` de la misma librería. Esta última función recibe un parámetro de tipo `char *` donde se guardará cada caracter leído con `getchar`, dejando un último lugar para el caracter nulo `\0`. No devuelve nada; el parámetro recibido es también de retorno.

Ahora bien, es tarea de la función invocadora (en este caso, `scanf`) pasarle a `getString` un `char *` con suficiente espacio reservado para que la función se encargue de rellenarlo. Si el parámetro no tiene suficiente espacio para guardar todo lo introducido por el usuario, se producirá un error de segmentación en memoria y el sistema se reiniciará.

Y es aquí donde se toma la decisión. Con las constantes globales, `MAX_ARGUMENTS` y `MAX_ARGUMENT_LENGTH`, se limita la cantidad de argumentos y caracteres por argumento que puede introducir el usuario, respectivamente:

en `shell.h`:

```
. . .
#define MAX_ARGUMENT_LENGTH 32
#define MAX_ARGUMENTS 8
. . .
```

Dicho esto, dentro de `scanf` y antes de la llamada a `getString`, se declara la variable que posteriormente será utilizada como el parámetro de retorno que esta última función requiere de la siguiente manera:

en `libc.c`:

```

. . .
#define MAX_USER_LENGTH MAX_ARGUMENTS*MAX_ARGUMENT_LENGTH
. . .
int scanf(const char * str, ...){
    . . .
    char strIn[MAX_USER_LENGTH+1];
    . . .
}

```

Así, se reserva el suficiente espacio para que el usuario pueda meter `MAX_USER_LENGTH` caracteres, más uno para hacer la string `null-terminated`.

## 2. Bibliografía consultada

- **OSDev.org**  
[http://wiki.osdev.org/Main\\_Page](http://wiki.osdev.org/Main_Page)
- **Programación del PIC**  
<http://www.beyondlogic.org/interrupts/interupt.htm#6>  
<http://www.osdever.net/tutorials/pdf/pic.pdf>
- **ASCII y ScanCodes**  
<http://www.win.tue.nl/~aeb/linux/kbd/scancodes-1.html>  
<http://www.cdrummond.qc.ca/cegep/informat/Professeurs/Alain/files/ascii.htm>  
<http://www.ee.bgu.ac.il/~microlab/MicroLab/Labs/ScanCodes.htm>
- **Manejo de excepciones**  
[http://pdos.csail.mit.edu/6.828/2008/readings/i386/s09\\_08.htm](http://pdos.csail.mit.edu/6.828/2008/readings/i386/s09_08.htm)
- **Command bytes del teclado**  
**Command bytes**  
[http://www.arl.wustl.edu/~lockwood/class/cs306/books/artofasm/Chapter\\_20/CH20-2.html](http://www.arl.wustl.edu/~lockwood/class/cs306/books/artofasm/Chapter_20/CH20-2.html)  
**LEDs** [http://wiki.osdev.org/PS2\\_Keyboard#Keyboard\\_LEDs](http://wiki.osdev.org/PS2_Keyboard#Keyboard_LEDs)  
**Reboot** <http://wiki.osdev.org/Reboot>

### 3. Código fuente

#### 3.1. Directorio src

##### libc.c:

```
#include "../include/kernel.h"
#include "../include/shell.h"
#include "../include/kc.h"

#define MAX_USER_LENGTH MAX_ARGUMENTS*MAX_ARGUMENT_LENGTH

int getchar(){
    return getc(stdin);
}

int getc(int fd){
    int character = -1;
    while (character == -1)
        read(fd, &character, 1);
    return character;
}

int atoi(const char *str){
    int i, acum, flagNegative = 0;
    for (i = 0, acum = 0; (!i && str[i]!='-') || isDigit(str[i]) && str[i]!='\0'; i++){
        if (str[i]!='-')
            flagNegative = 1;
        else
            acum = acum * 10 + str[i] - '0';
    }
    if (i==strlen(str))
        return flagNegative * -1 * acum + !flagNegative * acum;
    return 0;
}

/*
** NOTE:
```

```

**      This implementation of 'getString' is aimed for being used only by 's
** 'ans' is always a vector of MAX_STRLEN chars), as i counter is only compar
** constant (in the 'else if' clause), regardless the length of 'ans'.
*/
void getString(char * ans){
    int i, j;
    char c;
    for (i=0, j=0; (c = getchar()) != '\n' ; j++){
        printf("%c", c);
        if(c == '\b'){
            if (i)
                ans[--i] = '\0';
        } else if (i<MAX_USER_LENGTH){
            ans[i] = c;
            i++;
        }
    }
    ans[i] = '\0';
}

/*
** NOTE:
**      As we can't make use of 'malloc' function, we decided to set MAX_USER
** max quantity of characters the user can enter. Then, 'scanf' reserves MAX_
** vector strIn (strIn[MAX_USER_LENGTH] = '\0').
*/

int scanf(const char * str, ...){
    int i, j, k, acum, strLen, strTrueLen, strInLen, flagNegative;
    /*      strLen:          'str' length. Scanf's string.
    **      strTrueLen:      final 'str' length, once every %x in 'str' ha
    **      strInLen:        'strIn' length, user-entered string.
    */
    char strIn[MAX_USER_LENGTH+1]; /* Last position is reserved for '\0'
    void ** argv = (void **>(&str);
    getString(strIn); /* User enters a string */
    strLen = strlen(str);
    strTrueLen = strLen; /* At start, boths lengths are equals, becaus
    strInLen = strlen(strIn);

    for (i=0, j=0 ; i<strLen && j<strInLen; i++, j++){
        if ( i+1 != strLen && str[i] == '%' )

```

```

switch(str[++i]){
    case 'd':
        for(k=0, acum = 0, flagNegative = 0;
            if (strIn[j] != '-'){
                acum = acum * 10 + strIn[j] - '0';
            } else
                flagNegative = 1;
        }
        j--;
        if (!k){
            return -1;
        } else if (!isDigit(strIn[j]) && flagNegative)
            return -1;
        }
        *((int **)++argv) = flagNegative * acum;
        strTrueLen = strTrueLen - 2 + k; /*
        break;
    case 'c':
        *((char **)++argv) = strIn[j];
        strTrueLen--;
        break;
    case 's':
        argv++;
        /* NOTE:
        **      i+1 can be equal to strLen as
        ** So, str[strLen] = '\0';
        */
        if ( i+1 <= strLen){
            for (k=0; k<MAX_USER_LENGTH && strTrueLen > 0; k++){
                ((char *)((*((char **) argv))) [k] = strIn[i+1+k]);
            }
            j--;
            strTrueLen = strTrueLen -2 + k;
            break;

        /* NOTE:
        **      'default' case works this way to keep
        ** is neither 'd' nor 'c' nor 's', then it skip
        */
        default:
            j--;
            strTrueLen-=2;

```



```

                                break;
                                }
                                else
                                    if (str[i] != strIn[j])
                                        return -1;
                                }
                                if (strInLen != strTrueLen)
                                    return -1;
                                printf("\n");    /* The user entered '\n', so it must be used */
                                return 0;
}

int read(int fd, int * c, size_t count){
    return _int_80_hand(0, fd, c, count);
}

int strlen(const char * str){
    int i;
    for (i = 0; str[i] != '\0'; i++);
    return i;
}

int isDigit(int a){
    if( (a >= '0') && (a <= '9') )
        return 1;
    return 0;
}

int putchar(int c){
    return putc(stdout, c);
}

int putc(int fd, int c){
    return write(fd, &c, 1);
}

int printf(const char * str, ...){
    int c;
    int wait = 0;
    void ** argv = (void **>(&str);

```

```

        for(c=0;str[c] != NULL;c++){
            if(wait){
                switch(str[c]){
                    case 'd':
                        putInt( *( (int*) argv));
                        break;
                    case 'c':
                        putchar(*((char*)argv));
                        break;
                    case 's':
                        printf(*((char**)argv));
                        break;
                    default:
                        break;
                }
                wait = 0;
            } else {
                if(str[c] == '%'){
                    wait = 1;
                    argv++;
                }
                else
                    putchar(str[c]);
            }
        }
    }

int write(int fd, int * c, size_t count){
    return _int_80_hand(1, fd, c, count);
}

int pow(int n, int exp){
    int i;
    int out = 1;
    for(i=0;i<exp;i++)
        out*=n;
    return out;
}

void putInt(int n){

    int i,j;

```

```

    if(n<0){
        putchar('-');
        n = -n;
    }

    for(i=0;n/pow(10,i) > 0 ;i++); /* char quantity */
    for(j=i-1;j>0;j--){
        putchar( (n/pow(10,j))+ '0');
        n = n/pow(10,j);
    }
    putchar((n%10)+'0');
}

int strcmp(const char * str1, const char * str2){

    int i = 0;
    int out = 0;
    while(str1[i] != NULL || str2[i] != NULL){

        if( str1[i] != str2[i])
            return str1[i] - str2[i];

        i++;
    }

    if(str1[i] == NULL && str2[i] != NULL)
        return -str2[i];
    else if( str1[i] != NULL && str2[i] == NULL)
        return str1[i];

    return out;
}

int rand(){
    _outport(0x70, 0);
    int seconds = _inport(0x71);
    int acum1, acum2;

    /* NOTE 1:
    **      - Max value an integer can handle is 2^32. So 'seconds%32' ma
    **      - '(int)(seconds/32)' is added to avoid cases like:
    **      pow(2,59) = pow(2,27)          --> acum1 wou

```

```

**                Thanks to the addition:
**                pow(2,59) + 1 != pow(2,27) + 0 --> acum1 can
*/
acum1 = pow(2, seconds%32) + (int)(seconds/32);

/* NOTE 2:
**      - Max value an integer can handle is 2^32. But 3^20 < 2^32 <
**      - '(int)(seconds/20)' is added to avoid cases similar to the
*/
acum2 = pow(3, seconds%20) + (int)(seconds/20);

return acum2%acum1;
}

void wait(int seconds){
    int i, lastSec = -1;
    _outport(0x70, 0);
    int actSec = _inport(0x71);

    for (i=0; i <= seconds; i++){
        lastSec = actSec;
        _outport(0x70, 0);
        actSec = _inport(0x71);
        if (lastSec == actSec)
            i--;
    }
}

/*****
*setup_IDT_entry
* Inicializa un descriptor de la IDT
*
*Recibe: Puntero a elemento de la IDT
*        Selector a cargar en el descriptor de interrupcion
*        Puntero a rutina de atencion de interrupcion
*        Derechos de acceso del segmento
*        Cero
*****/

void setup_IDT_entry (DESCR_INT *item, byte selector, dword offset, byte acceso,
                    byte cero) {
    item->selector = selector;
    item->offset_l = offset & 0xFFFF;

```

```
    item->offset_h = offset >> 16;  
    item->access = access;  
    item->cero = cero;  
}
```