**AMPLIACIÓN DE SISTEMAS OPERATIVOS - HOJA DE EJERCICIOS**
**Hoja de Ejercicios 1.5**

**Nombre y Apellidos:Badr Guaitoune Akdi**
**Nombre y Apellidos:Danel Ledesma Ventura**

**Grupo:3D**

# Ejercicio 2.2 - Buffering

## Cuestiones

**Cuestión 1 - /dev/zero.** ¿Qué funcionalidad tiene el fichero de dispositivo `/dev/zero`?, ¿qué ocurre cuando leemos de él?, ¿y si escribimos?

**Es** un archivo especial que provee tantos caracteres null como se lean de el.
Todas las escrituras a `/dev/zero` ocurren sin ningún efecto.
Todas las lecturas a `/dev/zero` retornan tantos caracteres NULLs como sean requeridos.

**Cuestión 2 - Copia con dd.** Consulta el manual de la orden `dd`. A continuación, ejecuta la orden:

```
BS=128
dd if=/dev/zero of=salida.bin bs=${BS} count=$(( 16*1024*1024 / BS ))

¿Qué hace esa orden?
```

```
Hace una copia dese el archivo origen if= al destino of=
Copia count bloques de Tamaño BS.
```

**Cuestión 3 - Más con dd.** Prueba nuevamente la orden anterior pero dando valores BS = 1, 128, 512, 1024, 2048, 2200, 4096, 5000. Anota el tiempo de ejecución de cada prueba y trata de explicar las diferencias en el tiempo de ejecución.

```
BS=1
16777216+0 registros leídos
16777216+0 registros escritos
16777216 bytes (17 MB, 16 MiB) copied, 27,0921 s, 619 kB/s
BS=128
131072+0 registros leídos
131072+0 registros escritos
16777216 bytes (17 MB, 16 MiB) copied, 0,232671 s, 72,1 MB/s
```

```
BS=512
32768+0 registros leídos
32768+0 registros escritos
16777216 bytes (17 MB, 16 MiB) copied, 0,0701229 s, 239 MB/s
BS=1024
16384+0 registros leídos
16384+0 registros escritos
16777216 bytes (17 MB, 16 MiB) copied, 0,0446253 s, 376 MB/s
BS=2048
8192+0 registros leídos
8192+0 registros escritos
16777216 bytes (17 MB, 16 MiB) copied, 0,0506898 s, 331 MB/s
BS=2200
7626+0 registros leídos
7626+0 registros escritos
16777200 bytes (17 MB, 16 MiB) copied, 0,0332291 s, 505 MB/s
BS=4096
4096+0 registros leídos
4096+0 registros escritos
16777216 bytes (17 MB, 16 MiB) copied, 0,0240935 s, 696 MB/s
BS=5000
3355+0 registros leídos
3355+0 registros escritos
16775000 bytes (17 MB, 16 MiB) copied, 0,0255293 s, 657 MB/s
```
Cuanto mayor sea el bloque(BS), menos tarda en hacer la copia porque el numero de bloques que tiene que copia(count) se reduce.

## Ejercicios - E/S síncrona

**Ejercicio 1 - copia.c** Considera nuevamente el código de copia `rw.c` implementado en la hoja anterior. Realiza la apertura del fichero en el que se escribirá con el flag `O_SYNC` y compara los tiempos con los observados anteriormente usando distintos tamaños de lectura/escritura (usa como entrada el fichero `salida.bin` generado en el apartado anterior). Usa también el flag en el fichero de entrada.

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>


#ifndef TAM_BLOQUE
#define TAM_BLOQUE 1
#endif

// declarar array para almacenar los bytes leidos
char *buff[TAM_BLOQUE];

int main(int argc, char *argv[]) {

  int   fdo, fdd,n;
```

```
  char  c;

   // mantener este fichero para pruebas iniciales.
  // cambiar por uno mayor para medir tiempos
  fdo = open(argv[1], O_RDONLY);

  if( fdo == -1 ) {
    perror("open");
    exit(1);
  }

  // Abrir fichero destino para escritura
  fdd = open (argv[2],O_CREAT|O_WRONLY|O_TRUNC|O_SYNC, 0660);
  if (fdd == -1) {
  }

  printf("Comienza la copia con TAM_BLOQUE=%d bytes\n", TAM_BLOQUE);
  // Incluir las llamadas de temporizacion de vuestra libreria (start...)
  // Bucle de copia: leer TAM_BLOQUE btyes por llamada. Escribir lo leido
  int i = 0;
    while((n=read(fdo,buff, TAM_BLOQUE)) > 0) {
         if (write(fdd, buff, n) != n) {
            perror("Error en write");
             }
    }
  if (n == -1)
    perror("read");
  close(fdo);

  return 1;
}
```

## Ejercicios - Mapeo de ficheros en memoria

**Ejercicio 2 - mmap.c** Escribe un programa llamada `mmap.c` que mapee los 200 primeros bytes de un fichero de texto existente (puedes crear uno copiando cualquier código fuente en un nuevo fichero) a memoria. Si el fichero fuese menor de 200 bytes, se mapeará el fichero completo. Posteriormente, se pasará a mayúsculas todo el texto del fichero.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <ctype.h>



int main(int argc, char* argv[]) {
  int fd;
  struct stat buf;
```

```
  size_t mapSize;
  char *texto;
  int i;

  if(argc!=3) {
    printf("Error. Uso: %s nomFich\n", argv[0]);
    exit(-1);
  }

  if((fd = open(argv[1], O_RDWR)) == -1) {
    perror("Fallo en open fd");
    exit(-1);
  }

  //Usar fstat para obtener informacion del fichero.
  if(fstat(fd,&buf) == -1) {
      perror("Fallo en fstat");
      close(fd);
      exit(-1);
   }

   //2. Comprobar que es un fichero regular
   if(!S_ISREG(buf.st_mode)) {
    fprintf(stderr, "%s no es un directorio.\n", argv[1]);
    exit(1);
  }

  //Obtener el tamaño del fichero
  mapSize = buf.st_size;
  if(mapSize > 200){
    mapSize = 200;
  }

  if(!(texto= mmap(0,mapSize, PROT_READ, MAP_PRIVATE, fd, 0))) {
     printf ("mmap error for input%s aaa",texto);
     return 0;
  }


  for (i = 0; i < mapSize; i++){
    texto[i] = toupper(texto[i]);
  }

 for (i = 0; i < mapSize; i++)
    printf("%c", texto[i]);

 munmap(texto, mapSize);
 close(fd);
 return 1;
}
```

## Ejercicios - E/S biblioteca

**Ejercicio 3 - fcopia.c Una vez más, crea una nueva versión de la copia de ficheros, pero usando las llamadas de la librería estándar (`fopen`, `fread`, ...). Vuelve a realizar diversas pruebas con diferentes tamaños de lectura/escritura midiendo los tiempos. ¿Las diferencias de tiempo**

**son igual de notables cuando se usan tamaños de lectura/escritura de unos pocos bytes? Repite las pruebas incluyendo una llamada a `fflush(NULL)` tras cada escritura. ¿Se producen diferencias de tiempo?**

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>



#ifndef TAM_BLOQUE
#define TAM_BLOQUE 1
#endif
#ifndef FFLUSH
#define FFLUSH 1
#endif

// declarar array para almacenar los bytes leidos
char *buff[TAM_BLOQUE];

int main(int argc, char *argv[]) {

  int   fdo, fdd,n;
  char  c;

   // mantener este fichero para pruebas iniciales.
  // cambiar por uno mayor para medir tiempos
  fdo = fopen(argv[1], "r");

  if( fdo == -1 ) {
    perror("open");
    exit(1);
  }

  // Abrir fichero destino para escritura
  fdd = fopen (argv[2],"w");
  if (fdd == -1) {
  }

  printf("Comienza la copia con TAM_BLOQUE=%d bytes\n", TAM_BLOQUE);
  // Incluir las llamadas de temporizacion de vuestra libreria (start...)
  // Bucle de copia: leer TAM_BLOQUE btyes por llamada. Escribir lo leido
  int i = 0;
    while((n=fread(buff,1,TAM_BLOQUE,fdo)) > 0) {
         if (fwrite(buff, 1,n, fdd) != n) {
            perror("Error en write");
             }
          if(FFLUSH==1 && flush(NULL)!=0){
             perror("Error en flush");
         }
    }
  if (n == -1)
    perror("read");
  fclose(fdo);

  return 1;
}
```

```
Si se producen diferencias de tiempo ya que con fopen, fread, fwrite no se
tiene que vambiar el modo del procesador por lo que te ahorras loa operaciones
de cambiar modo, guardar el estado...
```

**Ejercicio 5 - vbuf.c** Estudia, compila y ejecuta el código del fichero `vbuf.c`. A continuación, usa `setvbuf` para probar el efecto de las 3 estrategias de *buffering* disponibles en la librería estándar: sin *buffer*, *buffer* de línea y *buffer* completo. ¿En qué se diferencia el comportamiento de cada una y por qué? ¿Cuál es la estrategia por defecto cuando escribimos en el terminal?

```
#include <stdio.h>
#include <unistd.h>

#ifndef BUF_MODE
#define BUF_MODE 0
#endif

int main() {
  int i = 0;
  // setvbuf(stdout,(char*)NULL,_IONBF,BUFSIZ);
  switch(BUF_MODE) {
    case 1:
/
*******************************************************************************
/
        setvbuf(stdout,(char*)NULL,_IONBF,BUFSIZ);
/
*******************************************************************************
/
        break;
    case 2:
/
*******************************************************************************
/
         setvbuf(stdout,(char*)NULL,_IOLBF,BUFSIZ);
/
*******************************************************************************
/
        break;
    case 3:
/
*******************************************************************************
/
         setvbuf(stdout,(char*)NULL,_IOFBF,BUFSIZ);
/
*******************************************************************************
/
        break;
    default:
        break;
  }
  for (i = 0; i< 5; i++) {
    printf("Esta es la ");
    sleep(1);
    printf("Linea %d\n", i);
    getc(stdin);
```

```
  }
  printf("Final\n");
  return 0;
}
```

**_IOFBF:** Cuando un acumulador esté vacío, la siguiente operación intentará llenar el acumulador completamente. En salida, el acumulador será completamente llenado antes de cualquier dato sea escrito en el fichero.
**_IOLBF:** Cuando un acumulador esté vacío, la siguiente operación intentará llenar el acumulador completamente. En salida, sin embargo, el acumulador será transmitido y despejado en cuanto un carácter de línea nueva sea escrito en el fichero.
**_IONBF:** Los parámetros **acumulador** y **tamaño** son ignorados. Cada operación de entrada leerá directamente desde el fichero, y cada operación de salida inmediatamente escribirá los datos al fichero.

**Ejercicio 6 - copyBuf.c** Estudia, compila y ejecuta el código del fichero `copyBuf.c` que copia un fichero byte a byte usando la librería estándar. Compara el tiempo de ejecución (usando `time`) con el código de `fcopia.c` con tamaño de copia de 1 byte. ¿Hay diferencias de tiempo? ¿Por qué?

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

int main(int argc, char **argv) {
  if (argc != 3) {
    printf("Uso: %s fichero-a-copiar fichero-destino\n", argv[0]);
    exit(1);
  }
  FILE* fp = fopen(argv[1], "r");
  FILE* fpo = fopen(argv[2], "w");
  int ch;
  struct stat stats;

  if (fp == NULL) {
    perror("fopen"); return 1;
  }

  if (fstat(fileno(fp), &stats) == -1) { // POSIX only
    perror("fstat"); return 1;
  }

     printf("BUFSIZ is %d, but optimal block size is %ld\n", BUFSIZ,
stats.st_blksize);
  if (setvbuf(fp, NULL, _IOFBF, stats.st_blksize) != 0) {
    perror("setvbuf failed"); // POSIX version sets errno
    return 1;
  }

  while ((ch=fgetc(fp)) != EOF) fwrite(&ch,1,1,fpo);

  fclose(fp);
  fclose(fpo);
```

```
}



Si si hay diferencias el fcopia.c es mas lento que el copyBuf.c

ya que el copyBuf calcula el tamaño de bloque optimo y al realizar copias de
dichos bloques, tarda menos en cioar el archivo.
```

**Ejercicio 3.** Crea un **makefile** que compile todos los ejemplos del directorio de modo que se generen ejecutables con el mismo nombre que el fuente, pero con extensión **.elf**. El **makefile** deberá incluir una regla **clean** que borre todos los ficheros **.o** y **.elf** generados.

```
CC = gcc
CFLAGS = -g -Wall -Werror

all: copia.elf copyBuf.elf fcopia.elf mmap.c vbuf.c

%.elf: %.o
	$(CC) $(CFLAGS) $^ -o $@

&.o: %.c
	$(CC) -c $< -o $@

.PHONY: clean


clean:
	rm -f *.o *.elf
```

## Compilación y Entrega

Entrega este documento exportado a PDF con las respuestas a las cuestiones y por otro lado el código fuente de todos los ejercicios, comprimido como .zip.