Makefile (0,5 puntos)

Crea un fichero Makefile que permita generar todos los programas del enunciado a la vez, así como también cada uno por separado. Añade una regla (clean) para borrar todos los binarios y/o ficheros objeto y dejar sólo los ficheros fuente. Los programas se han de generar si, y solo si, ha habido cambios en los ficheros fuente.

Control de errores (0,5 puntos)

Para todos los programas que se piden a continuación hace falta comprobar los errores de TODAS las llamadas a sistema (excepto el write por stoud/stderr), controlar los argumentos de entrada y definir la función Usage().

Ejercicio 1 (1,5 puntos)

Desarrolla un programa denominado **parser.c** que buscará los saltos de línea dentro de un texto. Si el programa recibe el nombre de un fichero como parámetro de entrada, *nombre_fichero*, lo abrirá para leerlo. En caso de no recibir ningún parámetro, leerá el texto desde la entrada estándar. Cada vez que encuentre un salto de línea (es decir, el carácter '\n') se guardará su posición dentro del fichero (es decir, el offset) en un vector de enteros. Este vector de enteros se guardará en la región de memoria heap (se valorará que se utilice un único vector con tantos elementos como número de saltos de línea se encuentren, pero sin reservar más memoria de lo necesario en el heap). Una vez alcanzado el final del fichero, se escribirá con una única llamada al sistema (se valorará que solo se utilice una única llamada para hacerlo), todo el contenido del vector de enteros (en formato interno) por el canal 4. Al final, antes de finalizar, se escribirá un mensaje por el canal de error que indique el número total de saltos encontrados y la cantidad de memoria heap, en Bytes, utilizada para guardar este vector.

Para validar este ejercicio, hemos proporcionado unos ficheros de muestra ("text1", "text2", "text3" que son copias de los ficheros originales "text1_orig", "text2_orig", "text3_orig", respectivamente). Se trata de ficheros de texto que tienen los saltos de línea en los offsets indicados en los ficheros "text1_orig.out", "text2_orig.out", "text3_orig.out", respectivamente. El contenido de este segundo conjunto de ficheros se puede analizar con la línea de comandos "xxd file". Este comando muestra, en formato hexadecimal, el valor de los bytes del fichero indicado como parámetro de entrada.

Ejemplos de salida1:

```
$ ./parser  text1 4> text1.out
El texto tiene 5 saltos de línea y se han utilizado 20 Bytes del heap

$ ./parser < text1 4> text1.out
El texto tiene 5 saltos de línea y se han utilizado 20 Bytes del heap

$ cat text1 | ./parser 4> text1.out
El texto tiene 5 saltos de línea y se han utilizado 20 Bytes del heap
```

NOTA: La línea de comandos "xxd text1.out" debería mostrar la siguiente salida:

```
00000000: 0100 0000 0400 0000 0800 0000 0d00 0000  ................
00000010: 1800 0000                                ....
```

Estos valores en hexadecimal corresponden a los números en entero (formato interno de la máquina) de las posiciones "1", "4", "8", "13" y "24", respectivamente. El contenido del fichero "text1.out" debería ser el mismo que "text1_orig.out". La columna de la izquierda (antes del ':') indica, en hexadecimal, la posición dentro del fichero que muestra el comando "xxd" y la columna de la derecha (los puntos) la representación en ASCII, si es posible (en caso contrario pone un punto), de cada Byte.

Ejemplos de salida2 y 3:

```
$ ./parser text2 4> text2.out
El texto tiene 7 saltos de línea y se han utilizado 28 Bytes del heap

$ ./parser text3 4> text3.out
El texto tiene 18 saltos de línea y se han utilizado 72 Bytes del heap
```

ATENCIÓN! Este ejercicio se evaluará en función de la ejecución correcta del programa según se indica a continuación: 1 punto si funciona como se espera; 0,25 puntos adicionales si, funcionando como se espera, la cantidad de memoria heap que se reserva y asigna es la justa y necesaria para guardar los saltos de línea que se encuentren; y 0,25 puntos adicionales si, funcionando como se espera, se utiliza una única invocación a la llamada al sistema correspondiente para escribir todo el contenido del vector de enteros por el file descriptor "4".

Ejercicio 2 (2 puntos)

Implementa un programa llamado **modifier.c** que aceptará al menos dos parámetros de entrada: un nombre y, al menos, un número (puedes asumir que son números positivos). El objetivo de este programa es cambiar los saltos de línea del fichero de texto por un guion (es decir, '-'). El programa abrirá el fichero pasado como parámetro de entrada, *nombre_fichero*, y su correspondiente ".out" creado en el ejercicio anterior. Los números indicarán los saltos de línea que se quieren cambiar. Se valorará mostrar mensajes de error por pantalla para aquellos números pasados como parámetro de entrada que no ocupen posiciones válidas dentro del fichero ".out", así como que el programa SOLO acceda a las posiciones indicades, tanto en el fichero ".otu" como en el fichero de texto. El cambio se actualizará en el mismo fichero de texto. El fichero ".out" NO se tiene que modificar. **Por lo que antes de hacer una prueba se aconseja utilitzar una copia del fichero original.**

Ejemplo de salida 1:

```
$ ./modifier  text1 2 20
El salto de línea 20 no esta en el rango de este fichero
```

El fichero "text1" pasaría a tener:

```
1
23-456
7890
1234567890
```

Ejemplo de salida 2:

```
$ ./modifier  text2 1 2 3 8 4 5 6 7 9
El salto de línea 8 no esta en el rango de este fichero
El salto de línea 9 no esta en el rango de este fichero
```

El fichero "text2" pasaría a tener:

```
ABCD--EFGHIJKLMNO-PQ-RSTUVW-XYZ-1234567890-
```

Ejercicio 3 (5,5 puntos)

Haz un programa llamado **multiparser.c** que aceptará tantos parámetros de entrada como nombres de ficheros de texto que se quieran procesar. ***NOTA:*** *se recomienda usar ficheros de entrada con nombres diferentes.* Antes de nada, el proceso principal tiene que asegurarse de bloquear todas las signals.

Para cada parámetro de entrada, se creará un proceso child, siguiendo un esquema de creación y ejecución secuencial. Para cada proceso creado, el proceso principal activará una alarma de un segundo y esperará, sin consumir CPU, hasta recibir el signal correspondiente. Al recibirlo el proceso principal tiene que "desbloquear" al proceso child usando una pipe con nombre, pero sin enviar datos. A continuación, el proceso principal escribirá el contenido del fichero i-ésimo a una pipe sin nombre conectada al proceso child correspondiente. Una vez procesado todo el

fichero, el proceso principal monitorizará la finalización del proceso hijo y mostrará por pantalla si ha finalizado correctamente o no. Por último, el proceso principal ha de finalizar mostrando un mensaje por el canal de error que indique "Final del proceso principal".

Por su parte, el proceso child i-ésimo tiene que crear un fichero de salida que tendrá el mismo nombre que el fichero que procesará, pero añadiendo el sufijo ".out". Por ejemplo, el fichero "file" implicaría crear "file.out". Si no existe, se crea con permisos de lectura y escritura para el usuario creador, lectura para el grupo y ninguno para los otros. Si el fichero ya existí, se borra el contenido anterior. Por último, se quedará esperando, sin consumir CPU, a que el proceso principal lo desbloquee, mediante la pipe con nombre, para poder continuar y mutar para ejecutar el programa "parser" del ejercicio 1. Si no has podido completarlo con éxito, te proporcionamos uno correctamente implementado.

Para comprobar que el programa se ha ejecutado correctamente, deberán salir tantos mensajes del programa "parser" (uno por segundo) y creado tantos ficheros ".out" como ficheros distintos se han introducido como parámetros de entrada.

Ejemplo de salida:

```
$ ./multiparser  text1 text2 text3
Proceso principal despertando proceso child 1
Proceso child 1 despertado
El texto tiene 5 saltos de línea y se han utilizado 20 Bytes del heap
El proceso child 1 ha finalizado voluntariamente
Proceso principal despertando proceso child 2
Proceso child 2 despertado
El texto tiene 7 saltos de línea y se han utilizado 28 Bytes del heap
El proceso child 2 ha finalizado voluntariamente
Proceso principal despertando proceso child 3
Proceso child 3 despertado
El texto tiene 18 saltos de línea y se han utilizado 72 Bytes del heap
El proceso child 3 ha finalizado voluntariamente
Final del proceso principal
```

NOTA: En este ejemplo se generarían los ficheros de salida "text1.out", "text2.out" y "text3.out". Además, hemos añadido mensajes para aclarar el comportamiento que deben tenir los procesos.

Qué hay que hacer

- El Makefile
- Los códigos de los programas en C
- La función Usage() para cada programa que sea necesario

Qué se valora

- Que sigas las especificaciones del enunciado
- Que el uso de las llamadas al sistema sea el correcto
- Que se comprueben los errores de todas las llamadas al sistema
- Que el código sea claro y correctamente indentado
- Que el Makefile tenga bien definidas las dependencias y objetivos
- Que la función Usage() muestre por pantalla como debe invocarse correctamente el programa en el caso que los argumentos recibidos no sean los adecuados

Qué hay que entregar

Un único fichero tar.gz con el código de todos los programas, el Makefile:

```
tar zcvf examenlab.tar.gz Makefile *.c
```