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 (3 puntos)

Implementa un código denominado **"initvector.c"** que inicializa un vector de enteros de tantos elementos como indica el primer parámetro de entrada que acepta este programa (puedes asumir que siempre será un número). Ahora bien, se tiene que comprobar que se trata de un número par y, en caso contrario finalizar el programa mostrando un mensaje de error por el canal estándar de error. El programa tiene que pedir memoria del heap para el vector de enteros y assignar a cada posición el valor entero del índice correspondiente. Es decir, el entero "0" a la primera posición, el "1" a la siguiente, etc. A continuación, si el programa tiene un segundo parámetro de entrada, creará un fichero (con el nombre indicado por este segundo parámetro) con permisos de lectura y escritura para el usuari, lectura para el grup y ninguno para el resto de usuaris. Si el fichero ya existiera, entonces se borrará el contenido. Si no hay un segundo parámetro de entrada, la escritura se realizará por la salida estándar. En ambos casos, sólo se escribirán enteros en formato interno (es decir, integers), donde el primero es un entero con la medida del vector y, a continuación, el contenido del vector utilizando una única llamada al sistema. Por último, se tiene que liberar la memoria del heap y finalizar el programa.

Para validar el correcto funcionamiento del programa, puedes hacer las siguientes ejecuciones (***NOTA:*** *"xxd" es un programa que muestra, en hexadecimal, el contenido del fitxer pasado como parámetro de entrada. Mira el "man" por si necesitas más detalles*):

```
$ ./initvector
Error: uso: ./initvector <número_par> [nombre_fichero]
$ ./initvector 6 file.dat (o $ ./initvector 6 > file.dat)
$ xxd file.dat
00000000: 0600 0000 0000 0000 0100 0000 0200 0000
00000010: 0300 0000 0400 0000 0500 0000
```

Ejercicio 2 (6 puntos)

Crea otro programa que se llame **"sumador.c"**. Este programa acepta un único parámetro de entrada y creará un sistema para hacer sumas, formado por 3 etapas que se ejecutarán a la vez.

En la primera etapa, el proceso principal (el padre) creará un proceso hijo que se comunicará con el proceso padre mediante una pipe sin nombre. Este proceso hijo mutará para ejecutar el programa del ejercicio anterior, pasando como parámetro de entrada el mismo paràmetro de entrada que "sumador". No se pasa un segundo parámetro de entrada porque lo tenemos que configurar para escribir los valores por la pipe sin nombre. Si no has podido implementar correctamente el programa anterior, puedes utilitzar un ejecutable correcto (nombrado "initvector_ok") que puedes encontrar adjunto a este examen.
La segunda etapa está comunicada con la primera, mediante la pipe sin nombre anterior, y con la tercera etapa, mediante una pipe con nombre, denominada "MIPIPE", que se tiene que crear

en el código (***NOTA:*** *se ha de tener en cuenta que si ya existe la pipe no representa un error para finalizar la ejecución*). En esta segunda etapa, el proceso padre tiene que leer de la pipe los valores que le llegan. Primero, la medida del vector, para saber cuántos elementos deberá leer. A continuación, para cada par de valores leidos, creará un proceso hijo que mutará para ejecutar el comando "expr" y así poder hacer una suma (por ejemplo, "expr 1 + 2" daría como resultado "3", que se escribe por la salida estándar). Se tiene que configurar para que escriba los resultados por la pipe con nombre. Por otro lado, el proceso padre esperará la muerte de este proceso y comprobará el estado de finalización, mostrando un mensaje por el canal estándar de error con el formato "El proceso <PID> ha finalizado voluntariamente" si ha finalizado invocando 'exit' (donde "<PID>" es el pid del hijo que acaba de morir) o con el formato "El proceso <PID> ha finalizado involuntariamente" si ha finalizado por la recepción de un signal. Después esperará un segundo, haciendo una espera pasiva, antes de seguir con el siguiente par de números.

Por último, en la tercera etapa, el proceso padre creará un proceso hijo que debe mutar para ejecutar la línea de comandos "wc -l" para mostrar el número total de sumas que se han hecho (***NOTA:*** *el programa "wc" cuenta el número de líneas con el flag "-l". Dado que al final de cada operación el programa "expr" introduce un salto de línea, cada suma supone una línea*). Este proceso hijo, se tiene que configurar para que lea desde la pipe con nombre.

Durante toda la ejecución, todos los procesos deben estar protegidos respecto la llegada de cualquier signal, excepto el proceso padre (que durante la espera de un segundo debe poder reaccionar ante la llegada del signal que lo despertará). Por último, esperará la muerte de los procesos hijo de la primera y de la última etapa y acabará mostrando un último mensaje para indicar que ha acabado, también por el canal estándar de error. Para poder validar este programa, puedes ejecutarlo tal y como se indica a continuación:

```
$ ./sumador
Error: uso: ./sumador <numero>
$ ./sumador 6
El proceso <PID1> ha finalizado voluntariamente
El proceso <PID2> ha finalizado voluntariamente
El proceso <PID3> ha finalizado voluntariamente
3
El proceso padre acaba
```

Aparece el valor "3" por pantalla porque es el resultado que ha escrito la tercera etapa, dado que se han hecho tres sumas.

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 FinalLab.tar.gz Makefile initvector.c sumador.c
```